Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognize the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
The Design and Evaluation of an Application Programming Interface for Programming Social Robots

James Peter Diprose

Acknowledgments

I would like to thank the people who have helped make this thesis possible:

John Hosking and Peter Smith for encouraging me to pursue a doctorate.

My supervisors: Beryl Plimmer, Bruce MacDonald and John Hosking, whom I can’t thank enough for their guidance and support throughout my doctoral work! My main supervisor Beryl, who cares incredibly for her students and always has time for us. Bruce, for giving me the opportunity to explore the field of robotics, your excellent food at the robotics group BBQs cannot go unmentioned either! And John, for taking me on as an honours student and for your continuously thoughtful feedback.

Steph, Mum, Dad, William and Joanna for supporting me throughout my thesis, especially Mum for proofreading it!

Beryl’s HCI research group, which has been very rewarding. My academic writing and research skills have improved because of it. Thanks to everyone from the group: Andrew Luxton-Reilly, Paul Schmieder, Adam Roughton, Safurah Abdul Jalil, Rachel Blagojevic, Brent Whitley, Jackie Zhen, Phil Stevens, Sam Kavanagh and Craig Sutherland. Alan Blackwell, for the in-depth discussions on notation design while on sabbatical in New Zealand.

The fellow students who have journeyed with me in Bruce’s robotics research group over the past 7 years: Chandan Datta, Saqib Bajwa, Rishi Pandey, Stephen Thomas, Tony Kuo and Ian Chen. Special thanks to Chandan Datta, who shared his great knowledge of robotics with me.

My friends at university. Special thanks to Richard Hosking whom I began the doctoral programme with and have always been able to bounce ideas off. Same for the lunch/coffee/OGH/StarCraft/HOTS/office friends! Glen Robertson, Stefan Wender, Paul

Those who helped make videos for conferences and workshops: Raphael Stone, Andrew Meads, Amy Hawes, Joanna Diprose, Daniel Sullivan and Ryan Attwood.

The office staff, Robyn and Sithra, who helped with all sorts of things over the last four years like printing and binding my thesis!

The kind people who hosted me while visiting other cities and countries. Your hospitality is unmatched and the academics amongst you provided very valuable feedback for my doctoral work. Ruth & Jamie Agnew, John Grundy, Daniel Moody, Richard Roberts, John Lewis, Rhazes Spell, Minsu Jang, Chanku Park, Yeow Kee Tan, Ben Goertzel, Gino Yu and Mark Tilden.

Special thanks to Ben Goertzel, for providing the opportunity to work in Hong Kong and Jong Lee and The Hong Kong Polytechnic University for funding my flights. My time in Hong Kong helped crystalise the direction of my thesis. It was a great experience exploring the sights and sounds of Hong Kong with the friends I met there: Steph Chan, Alex van der Peet (thanks for reading that chapter yo!), Scott Jones, David Hart, Paulson Yiu and David Hanson.

And last, but not least, the University of Auckland Doctoral Scholarship and other university grants that supported my research.
Abstract

Despite there being much work bringing the vision of socially interactive robots to life, it is still a challenge to program them. There are two key reasons for this. First, many robot programming tools express primitives for programming social-interaction at low abstraction levels. Second, the usability of methods used to combine primitives, regardless of their abstraction levels, into social-interactions are not well understood. To address these problems, this thesis presents the iterative design and evaluation of application programming interfaces (APIs) for programming socially interactive robots.

An iterative design-based research method was used throughout this thesis. During each research iteration, an API was designed with user-centred design principles to address particular aspects of the social robot programming research problem. Each API was then evaluated with a user study and the data analysed with the Cognitive Dimensions of Notations.

The first API iteration began exploring what abstraction level is appropriate for programming social robot applications, through an evaluation of an API with high-level, domain-specific primitives. The results of the evaluation showed that the chosen abstraction level had positive effects on usability, however, a stakeholder interview suggested that finer control of social-interaction was needed. This evaluation also explored how primitives, regardless of their abstraction level, should be orchestrated to create higher level social-interaction. This was done by evaluating an imperative finite state machine API for authoring robot dialogue; the results showed that they have poor usability when used in this context. This is important because imperative finite state machines are a common means of orchestrating robot behaviour in the robotics community.

The second API iteration further refined what abstraction level is appropriate for programming social robot applications. Based on the results of a stakeholder interview of the previous API, its primitives were refactored into a slightly lower abstraction level, providing finer control of social-interaction. A user evaluation demonstrated the benefits of the refactoring which retained the key advantages from the previous API and also the
The positive effects include hiding of lower level implementation details, primitives having a close mapping to the social-interaction domain and a structure that enables programmers to use their domain knowledge to understand the API. The main trade-off occurs when implementation details are hidden, this can make progressive evaluation difficult because programmers are suddenly exposed to lower level implementation details when debugging errors.

Parallel to the iterative API design, I derived an expanded implementation independent taxonomy of the primitives required for programming robot social-interaction. The taxonomy defines robot actions, perceptions and the objects with which a social robot interacts. This is important for stakeholders who help create socially interactive robots.

The last stage of the research was the design and implementation of an extensible, vendor agnostic architecture to support the API. The architecture enables API users to define new entities (objects, people and robots) and create relationships between them. It also enables the API to interface with multiple social robot platforms through its vendor agnostic action and perception interfaces.

In summary, this thesis makes five main contributions. First, the implementation of an API with high-level, domain-specific primitives for programming socially interactive robots. Second, an in depth understanding of what abstraction level is appropriate for programming social robot applications and the effects that different abstraction levels have on the APIs usability. Third, an exploration of how primitives should be orchestrated into higher level social-interaction, in particular, the effects that imperative finite state machines have on usability when used to author robot dialogue. Fourth, a taxonomy of primitives that are set at an appropriate abstraction level for programming robot social-interaction. Fifth, the design and implementation of an extensible, vendor agnostic architecture to support social robot programming APIs with primitives at an appropriate abstraction level.
4.2 Iteration 2: Social-Interaction Refactored .......................................................... 98
4.3 A Taxonomy of Social Primitives ........................................................................... 99
4.4 Software Architecture ......................................................................................... 100
4.5 Summary ............................................................................................................... 101

5 Iteration 1: Social-Interaction for Nao ................................................................. 103

5.1 Example Programs ............................................................................................... 105
  5.1.1 Object spatial relations .................................................................................. 105
  5.1.2 Speaking ....................................................................................................... 106
  5.1.3 Speaking and gesturing ................................................................................ 108
  5.1.4 Listening for utterances .............................................................................. 108
  5.1.5 Dialogue with a state machine .................................................................... 110
5.2 API Design ........................................................................................................ 113
  5.2.1 Environment ............................................................................................... 114
  5.2.2 Object ......................................................................................................... 114
  5.2.3 Robot .......................................................................................................... 115
  5.2.4 Person ......................................................................................................... 118
  5.2.5 HriQuery .................................................................................................... 119
  5.2.6 Dialogue .................................................................................................... 121
5.3 User Evaluation .................................................................................................... 124
  5.3.1 Method ....................................................................................................... 124
  5.3.2 Results ........................................................................................................ 126
5.4 Stakeholder Interview .......................................................................................... 129
5.5 Discussion .......................................................................................................... 130
5.6 Summary ............................................................................................................ 132

6 Iteration 2: Social-Interaction Refactored ........................................................... 135

6.1 Example Programs ............................................................................................... 138
  6.1.1 Greet a person ............................................................................................ 139
  6.1.2 Grab a person’s attention .......................................................................... 139
  6.1.3 Listen to someone ....................................................................................... 141
6.2 API Design .......................................................................................................... 142
  6.2.1 Entity .......................................................................................................... 143
  6.2.2 Robot .......................................................................................................... 144
  6.2.3 Robot: actions ............................................................................................. 145
  6.2.4 Robot: action controls ................................................................................ 150
  6.2.5 Robot: action callbacks .............................................................................. 153
B.4.2 Section 2 – Definitions ........................................................................................................281
B.4.3 Section 3 – Questions about the main notation .................................................................282

APPENDIX C ITERATION 2: SOCIAL-INTERACTION REFACTORED .................................285
C.1 TUTORIAL ................................................................................................................................286
  C.1.1 Starting hri api ..................................................................................................................286
  C.1.2 Making Nao speak (Python) ..............................................................................................287
  C.1.3 Making Nao gesture (Python) ..........................................................................................289
  C.1.4 Making Nao gaze at people (Python) ...............................................................................291
  C.1.5 Making Nao speak and gaze at a group of people simultaneously (Python) .............292
  C.1.6 Querying the World for objects Nao can sense (Python) .............................................294
  C.1.7 Listening to what people say (Python) ............................................................................297
C.2 INTERVIEW QUESTIONS .......................................................................................................299

APPENDIX D A TAXONOMY OF SOCIAL PRIMITIVES .........................................................303
D.1 SUPPORTING DATA ...............................................................................................................304

APPENDIX E FORMS ................................................................................................................311
Figures

Figure 1.1. Social robots: (a) Keepon, (b) Kismet and (c) Geminoid-DK (c). Image sources (cropped):
Keepon - (Beatbots, n.d.); Kismet - (Daderot, 2013); Geminoid-DK (background altered).
(Jurvetson, 2012)

Figure 2.1. Derived primitives.

Figure 2.2. Inputs from a Pioneer P3-DX and a gamepad. From left to right; laser scanner, sonar sensors, bumpers and joystick axes.

Figure 2.3. Driving the robot with a joystick. The robot is currently stationary.

Figure 2.4. Decision primitive with true, false and unsure connectors.

Figure 2.5. Means from Likert scale questions (1=strongly disagree, 5=strongly agree): questions listed in Table 2.1.

Figure 2.6. Qualitative analysis themes and analysis.

Figure 2.7. Healthbots robots: left, iRobiQ; right, Cafero. Used with permission of Healthbots.

Figure 3.1. General structure of database search terms.

Figure 3.2. General structure of venue terms.

Figure 3.3. The process of searching for a desired object; adapted from (Stilos & Clarke, 2007).

Satisfying the required constructor is a bottleneck.

Figure 3.4. Method placement, adapted from (Stilos & Myers, 2008, p. 105)

Figure 3.5. Process involved in using helper classes, adapted from prose in (Stilos & Myers, 2008).

Figure 3.6. Five important factors of API documentation, from (Robillard & DeLine, 2011).

Figure 3.7. Finding types via (A) source code and (B) documentation, derived from (Endrikat et al., 2014; Kleinschmager et al., 2012).

Figure 3.8. Methods for evaluating APIs.

Figure 3.9. Part of a concept map illustrating a student’s understanding of scientific concepts. From (Joseph D Novak, 1990).

Figure 3.10. Social robots: (a) Keepon, (b) Kismet and (c) Geminoid-DK (c). Image sources (cropped):
Keepon - (Beatbots, n.d.); Kismet - (Daderot, 2013); Geminoid-DK (background altered).
(Jurvetson, 2012)

Figure 3.11. Abstraction levels of social-interaction & examples.

Figure 3.12. Choregraphe.

Figure 3.13. Interaction blocks.

Figure 3.14. Interaction composer. Used with permission of Dylan F. Glas (Glas et al., 2012).

Figure 3.15. (A) textual robotics command language, (B) TiWPE visual editor. Used with permission of Emilia Barakova (Barakova et al., 2013).

Figure 3.16. AIML example.
Figure 3.17. BML example...........................................................................................................84
Figure 3.18. Robot behaviour toolkit example. Used with permission of Chien-Ming Huang and Bilge
Mutlu (Huang & Mutlu, 2012).......................................................................................................85
Figure 3.19. Gostai studio ..............................................................................................................86
Figure 3.20. (a) SkillGUI Lua state machine definition, (b) state machine visualisation. I have tried
to get permission to use these images but do not have a reply yet.................................................87
Figure 3.21. (a) SMACH imperative state machine definition and (b) state machine visualisation.
Image sources: (Open Source Robotics Foundation, n.d.-b)..........................................................88
Figure 3.22. XABSL editor example, screenshot of play-soccer-xabsl, from: (XABSL developer-team,
n.d.)...................................................................................................................................................89
Figure 4.1. Iterative research methodology, based on process from (Peppers et al., 2007, p. 58).............95
Figure 5.1. Object spatial relations................................................................................................105
Figure 5.2. Object spatial relations output.....................................................................................106
Figure 5.3. Speaking to people.......................................................................................................107
Figure 5.4. Speaking and gesturing.................................................................................................108
Figure 5.5. Listening.......................................................................................................................109
Figure 5.6. Dialogue management..................................................................................................111
Figure 5.7. Overview of domain-specific aspects of the API............................................................113
Figure 5.8. Overview of state machine...........................................................................................114
Figure 5.9. Object spatial relations................................................................................................115
Figure 5.10. Say_to_action.............................................................................................................117
Figure 5.11. Associating utterances with meanings.........................................................................117
Figure 5.12. Using contexts...........................................................................................................118
Figure 5.13. Simple method chaining example.............................................................................119
Figure 5.14. Selecting people less than 1 metre.............................................................................119
Figure 5.15. Defining a new State..................................................................................................122
Figure 5.16. StateMachine example. .............................................................................................123
Figure 5.17. Game show setup.......................................................................................................125
Figure 5.18. Hanson Robotics human-like robots: (a) Zeno, (b) Diego-San & (c) Philip K. Dick. Image
sources: Zeno - (Jurfetson, 2008); Diego-San image used with permission of David Hanson;
Philip K. Dick - (Lerdorf, 2012). ...................................................................................................(130
Figure 6.1. Overview of the social-interaction API & framework.....................................................136
Figure 6.2. Humanoid robots: (a) Nao, (b) Zeno and (c) Zoidstein. Source for images: Nao - Used with
permission from Samar Pant; Zeno - (Jurfetson, 2008). Note that Zeno had a different body
from the one pictured above.............................................................................................................137
Figure 6.3. Example greeting program...........................................................................................139
Figure 6.4. Example grabbing attention program............................................................................140
Figure 6.5. Example listening program...........................................................................................141
Figure 6.6. API overview................................................................................................................142
FIGURE 6.7. ENTITIES........................................................................................................144
FIGURE 6.8. ROBOT CLASS..............................................................................................145
FIGURE 6.9. ACTION PRIMITIVES ..................................................................................146
FIGURE 6.10. SPEAKING ACTION ..................................................................................147
FIGURE 6.11. GAZE ACTION .........................................................................................147
FIGURE 6.12. NAO’S GESTURES ....................................................................................148
FIGURE 6.13. GESTURE ACTION ...................................................................................149
FIGURE 6.14. ZENO’S EXPRESSIONS ............................................................................149
FIGURE 6.15. EXPRESSION ACTION ..............................................................................150
FIGURE 6.16. ACTION OBJECTS ....................................................................................150
FIGURE 6.17. ACTION CONTROLS ..................................................................................151
FIGURE 6.18. WAITING FOR ACTIONS TO COMPLETE ..................................................151
FIGURE 6.19. CANCELLING ACTIONS ............................................................................152
FIGURE 6.20. SIMULTANEOUSLY COMMAND ...............................................................153
FIGURE 6.21. SIMULTANEOUSLY COMMAND ...............................................................153
FIGURE 6.22. ROBOT ACTION CALLBACKS ....................................................................154
FIGURE 6.23. SPEAKING ACTION CALLBACKS USING ROBOT.SAY FUNCTION .............155
FIGURE 6.24. SPEAKING ACTION CALLBACKS USING SAYGOAL ................................155
FIGURE 6.25. COMMUNICATION CALLBACKS ................................................................156
FIGURE 6.26. EXAMPLE LISTENING PROGRAM .............................................................157
FIGURE 6.27. PERSON INSTANCES ................................................................................158
FIGURE 6.28. WORLD CLASS .......................................................................................159
FIGURE 6.29. ENVIRONMENT - OLD API ......................................................................160
FIGURE 6.30. WORLD - NEW API ..................................................................................161
FIGURE 6.31. QUERIES ..................................................................................................161
FIGURE 6.32. QUERYING THE WORLD FOR OBJECTS ..................................................162

Figure 7.1: Example robots: (a) AIBO, (b) Charlie, (c) ASIMO, (d) Keepon, (e) Nabantag, (f) Nao, (g) Paro and (h) Zeno. Images cropped, sources: AIBO - (Jordan, 2012), Charlie - used with permission of Healthbots, ASIMO - (S. Sowden, 2005), Keepon - (Keysmaker, 2007), Nabantag - (Eussen, 2007), Nao - used with permission of Saman Pant, Paro - used with permission of Healthbots and Zeno - (Juvetson, 2008). ..................................................................................177

Figure 7.2. A snapshot of the (a) open and (b) axial coding processes ................................179

Figure 7.3. Examples of objects and their relationships: Arrows are inheritance relationships, dotted lines are composition relationships. Objects with composition relationships also inherit from Object; this is not shown visually for brevity. ..................................................180

Figure 8.1. CUSTOMISABLE ASPECTS OF THE API......................................................194
Figure 8.2. DEFINING ENTITIES ....................................................................................195
Figure 8.3. ROS TRANSFORMATION TREE EXAMPLE ..................................................196
Figure 8.4. DEFINING A HIERarchical ENTITY ...............................................................196
FIGURE 8.5. ARCHITECTURE OF DEFINING NEW ROBOTS.......................................................... 198
FIGURE 8.6. THE ROS ROBOT STATE PUBLISHER ARCHITECTURE. SEE THESE REFERENCES FOR MORE DETAILS:
(SUCAN, N.D.-A), (SUCAN, N.D.-B) AND (FOOTE, 2013).......................................................... 199
FIGURE 8.7. CODE TO DEFINE A NEW ROBOT.......................................................................... 199
FIGURE 8.8. GESTURES AND EXPRESSIONS............................................................................. 200
FIGURE 8.9. DEFINING NEW GESTURES..................................................................................... 201
FIGURE 8.10. DEFINING NEW EXPRESSIONS............................................................................ 201
FIGURE 8.11. SOCIAL-INTERACTION FRAMEWORK................................................................... 202
FIGURE 8.12. ACTION INTERFACE............................................................................................ 203
FIGURE 8.13. TEXT TO SPEECH ROS ACTION DEFINITION...................................................... 204
FIGURE 8.14. TARGET ROS ACTION DEFINITION...................................................................... 204
FIGURE 8.15. GESTURE ROS ACTION DEFINITION................................................................... 205
FIGURE 8.16. EXPRESSION ROS ACTION DEFINITION.............................................................. 205
FIGURE 8.17. VENDOR ACTION IMPLEMENTATIONS................................................................. 206
FIGURE 8.18. PERCEPTION INTERFACE..................................................................................... 209
FIGURE 8.19. PERCEPTION SYSTEM.......................................................................................... 210
FIGURE 8.20. PERCEPTION SYNTHESISER YAML CONFIGURATION FILE.................................. 211
FIGURE 9.1 CD QUESTIONNAIRE QUESTIONS FOR PROGRESSIVE EVALUATION (BLACKWELL & GREEN, 2000).... 219
FIGURE 9.2. A MOCK-UP OF A VISUAL TOOL FOR PROFESSIONAL ROBOTICS PROGRAMMERS: A VISUAL BEHAVIOUR TREE EDITOR PAIRED WITH THE LATEST SOCIAL-INTERACTION API, INSPIRED BY THE WINDOWS PRESENTATION FOUNDATION DESIGNER FOR VISUAL STUDIO (MICROSOFT, N.D.)........................................... 226
Tables

TABLE 2.1. LIKERT SCALE QUESTIONS. ................................................................. 45
TABLE 3.1. ROBOT SOCIAL-INTERACTION & BEHAVIOUR CONTROL PROGRAMMING TOOLS. ................................. 78
TABLE 6.1. SOCIAL-INTERACTION ACTION FEEDBACK MESSAGES. .................................................. 155
TABLE 6.2. PERSON ATTRIBUTES. ................................................................. 158
TABLE 6.3. ROBOT CHAT GESTURES. SECOND IMAGE USED WITH PERMISSION OF SAMAR PANT (MODIFIED) .... 166
TABLE 7.1. SOCIAL ROBOTS. ........................................................................... 177
TABLE 7.2. ROBOT SOCIAL ACTIONS, SEE APPENDIX D FOR MORE SUPPORTING DATA ............................... 181
TABLE 7.3. ROBOT SOCIAL PERCEPTIONS, SEE APPENDIX D FOR MORE SUPPORTING DATA .......... 184
Introduction

For almost a century humans have dreamed of creating robots so advanced that they can interact socially just as humans interact with each other. Much work has been done towards achieving this goal: from creating algorithms that allow robots to perceive and interact with the world as humans do; to creating walking, talking animatronic bodies that look like real humans. However, it is still a challenge for programmers to create applications for these socially interactive robots. This is because creating a social interaction application requires the composition of a wide range of interfaces and algorithms and the programming tools used to create them do not have well thought out abstraction levels.

Tales of social robots from science fiction serve as motivation for many in the field of robotics. In these fictional universes, robots, more commonly referred to as ‘androids’ act as servants, friends and sometimes even enemies of humankind. These stories are the prototypical vision of what robots in the future should be capable of: machines that act, look and feel like human beings.

_The Nexus-6. He had now come up against it. Rachael, he realized; she must be a Nexus-6. I'm seeing one of them for the first time. And they damn near did it; they came awfully damn close to undermining the Voigt-Kampff scale, the only method we have for detecting them._

(Dick, 1968).

One of the most famous fictional works describing human-like robots is Philip K. Dick’s novel, ‘Do Androids Dream of Electric Sheep’ (Dick, 1968), also known by its popular cinematic adaptation Blade Runner (1982). In this fictional universe, androids have been banned from use on Earth because of fears they will destroy society due to their human-like behaviour and appearance. Yet, they are still used on Mars as servants. Every so often androids on Mars break free from slavery by killing their masters and
illegally making their way to Earth. It is the job of bounty hunters on Earth, like Rick Deckard, the protagonist of the story, to find and eliminate these escaped androids. The bounty hunter’s job is difficult and dangerous because the newer Nexus-6 models are so similar to real humans in both behaviour and appearance, that it is almost impossible to differentiate human from android.

Human-like robots also feature in the works of Isaac Asimov, albeit his robots are more mild-mannered and pleasant than Dick’s murderous replicants. One of Asimov’s most well-known robots is Andrew Martin from the novel The Positronic Man (Asimov & Silverberg, 1992); who is also depicted in the cinematic adaptation Bicentennial Man (Touchstone Pictures, 1999).

The surgeon reacted with a visible twitch of his shoulders and a blinking of his photoelectric eyes. It was as if what Andrew had just said had no meaning for him whatever.

"Yes," said Andrew, "I know that I seem to be quite human, and that what you're experiencing now is the robot equivalent of surprise. Nevertheless I'm telling you the absolute truth. However human I may appear to you, I am simply a robot. A robot, Doctor. A robot is what I am, and nothing more than that. Believe me. And therefore you are free to operate on me. There is nothing in the First Law which prohibits a robot from performing actions on another robot. Even if the action that is performed should cause harm to that robot, Doctor." Asimov & Silverberg (1992).

Andrew Martin begins life as robot NDR-133, a butler for the Martin household. In contrast to robots of his time, NDR-113 quickly becomes a part of the Martin family, anglicising his model name “NDR” into his first name “Andrew” and adopting the family name Martin. Also unlike robots of his time, Andrew quickly discovers he has creative talent and with the support of his family begins to lead a self-sufficient lifestyle. Possessing the psyche of a human, but the body of a robot, Andrew seeks to become human, both for himself and in the eyes of humanity.

Stories like ‘Do Androids Dream of Electric Sheep’ and ‘The Positronic Man’ are motivation for many whose work involves the research, design, construction,
programming and sale of robots. Many strive toward creating robots that can socially interact with the skill and dexterity of Dick’s Rachel and Asimov’s Andrew Martin.

In the context of this thesis, a socially interactive robot is considered to be an embodied machine that is able to interact with people using human social abilities (Fong, Nourbakhsh, & Dautenhahn, 2003, p. 145). Human social abilities include the ability to: express and perceive emotion; communicate with high-level behaviour, such as turn taking (Breazeal, 2003), dialogue or joint attention (Moore & Dunham, 1995); understand and model human individuality; learn social abilities; display personality; and build relationships with people.

In the last couple of decades, the robotics community has made progress toward creating such advanced social robots. Notable examples include Paro (Kozima, Nakagawa, & Yasuda, 2007), Kismet (Breazeal & Scassellati, 1999) and the Geminoid series of robots (Dougherty & Scharfe, 2011; Nishio, Ishiguro, & Hagita, 2007). These robots are illustrated in Figure 1.1.

Kozima et al. (2009) developed Keepon (Figure 1.1a) as a companion for interacting with children by displaying joint attention through gaze and expressing emotion by moving its body. Kismet (Figure 1.1b) was created by Breazeal & Scassellati (1999) to explore the application of anthropomorphic infant behaviour to robots, including: showing facial expressions, searching for objects it desires, looking away from threatening stimulation and reinforcing people’s desirable behaviour. The Geminoid series of robots (Figure 1.1c) are teleoperated replicas of real people that can interact through speech, facial expressions and gestures (Nishio et al., 2007). They were developed to study how life-like appearance and behaviour affects human-robot interaction.
As discussed at the beginning of the chapter, despite much work being done to bring the vision of socially interactive robots to life, it is still a challenge to program them. This is because creating a social interaction application requires the composition of a wide range of interfaces and algorithms. These include low-level sensor and actuator interfaces, algorithms for perceiving human features, algorithms that control robot actions, and interfaces for orchestrating these algorithms into an interactive application. A number of programming tools have been designed to program robot social interaction, however, the abstraction levels of these programming tools are not well thought out, as is evident from the inconsistent mix of abstraction levels they provide. User evaluations of these programming tools do not provide in-depth data explaining the effects that different abstraction levels have on usability. Designers and users of social interaction programming tools would benefit tremendously from an in-depth understanding of how different abstraction levels affect the usability of robot social interaction programming tools.

Despite the lack of empirical data, it is possible to hypothesise what abstraction levels are appropriate for this task.

Primitives with lower levels of abstraction than necessary are likely problematic because the programmer has to perform more work to attain their goal. For example, to make a robot speak and gaze at someone with a programming tool that provides only low-level hardware and algorithm primitives, a programmer must give detailed commands for speech synthesis to make it speak, analyse results from a face detection algorithm to find where the person’s head is and use joint control to make the robot gaze at that head.

Raising the abstraction level by giving primitives a closer mapping to the social interaction domain is likely to make it easier for programmers to create social interaction applications. Green & Petre (1996) argue that the closer a notation is to its problem domain, the easier it should be to create solutions in that domain. However, this raises the equally important question of how highly the primitives should be abstracted?

Additionally, if a programming tool has high-level primitives, but doesn’t cover enough of the social interaction space, then programmers will frequently have to go back to the method of using low-level primitives to create social interaction. Having a comprehensive model of the primitives needed to program robot social interaction at an appropriate abstraction level is important for ensuring that tools provide enough of the right primitives.
Chapter 1 - Introduction

There is also the question of how programmers orchestrate primitives into higher level behaviour, regardless of abstraction level. This depends on what programmers wish their robots to do. The following list details three common examples:

1. Integrate the robot with a teleoperation interface, for example, an interface used in Wizard of Oz psychology study.
2. Author an autonomous social interaction scenario with programmer defined logic, for example with a state machine.
3. Control a robot’s social interaction with artificial intelligence algorithms, for example with a cognitive architecture.

The simplest way to orchestrate the primitives to produce robot social interaction is to integrate them with a Wizard of Oz interface. These types of interfaces are commonly used in psychology studies where researchers are studying how people perceive robot technologies, e.g. (Heerink, Krose, Evers, & Wielinga, 2006; Hüttenrauch, Severinson Eklundh, Green, & Topp, 2006; Villano et al., 2011).

The second common scenario is for a programmer to manually author an autonomous social interaction scenario, e.g. with a finite state machine. For example, the robots Minerva (Schulte, Rosenberg, & Thrun, 1999), Sage (Nourbakhsh et al., 1999), Fritz (Bennewitz, Faber, Joho, Schreiber, & Behnke, 2005) and BIRON (Spexard et al., 2006) all use human authored state machines to drive their social interaction. Human authoring agent behaviour is common outside of robotics as well; the ‘artificial intelligence’ behind many game characters are really human authored scripts, state machines, rule base systems and decision trees (Robertson & Watson, 2014). Techniques such as this are just as likely to work well for social robots.

The last scenario involves a programmer using the primitives in conjunction with more advanced artificial intelligence algorithms, such as a cognitive architecture. For example, Breazeal et al. (2009) created a cognitive architecture that gives the robot Leonardo cognitive abilities based on another individual's actions, including beliefs, intents and desires. Other examples in this category include the work of Burghart et al. (2005), Dautenhahn & Billard (1999), Sandini et al. (2007) and Shanahan (2006).
1.1 Research objective and questions

The high-level objective of this thesis is to “investigate methods for improving the usability of programming tools used to program socially interactive robots”. This overall objective is realised with a number of research questions:

RQ1 What abstraction level is appropriate for programming social robot applications?
   RQ1.1 What are the trade-offs associated with using different abstraction levels to program robot social interaction?
RQ2 How should primitives, regardless of their abstraction level, be orchestrated to create higher level social interaction such as dialogue and non-verbal behaviour?
RQ3 What primitives are needed to program robot social interaction at the appropriate abstraction level?
RQ4 What is a suitable software architecture to support the API with primitives at the abstraction level appropriate for programming robot social-interaction?

The first research question (RQ1) aims to discover what type of abstraction level is appropriate for programming socially interactive robot applications. Appropriate is defined as an abstraction level that is easy for programmers to use but provides them with enough expressivity to create useful applications. This research question is explored through the iterative design, implementation and evaluation of APIs for programming socially interactive robots. Each iteration examines the effect that different abstraction levels (RQ1.1) have on API usability and expressivity. This should result in an understanding of an abstraction level that is appropriate for programming socially interactive robot applications.

The second research question (RQ2) examines how primitives should be orchestrated into higher level social-interaction, regardless of their abstraction level. This is important because little work has been done on evaluating the usability of techniques for orchestrating complex robot behaviour. For example textual, imperative finite state machines are commonly used by robot programmers to author social behaviour, however, studies that evaluate their usability are rarely published.

The third research question (RQ3) aims to develop a taxonomy of primitives at an abstraction level appropriate for programming robot social-interaction. The taxonomy should flesh out the primitives derived through the iterative design, implementation and
evaluation (RQ1.1) of APIs for programming socially interactive robots. However, it should be independent of implementation details so that it is not artificially constrained by current technology.

The fourth research question (RQ4) focusses on designing a software architecture to support the API with primitives at the appropriate abstraction level for programming robot social-interaction applications. The architecture needs to enable the API to be extensible so that programmers can support their own use cases and scenarios. The architecture also needs to be vendor agnostic, so that a variety of robot platforms can be supported.

1.2 Contributions

The key contributions of this thesis are as follows:

1. An API with high-level, domain-specific primitives for programming socially interactive robots.
2. An in depth understanding of what abstraction level is appropriate for programming social robot applications. This includes knowledge of the effects that primitives of different abstraction levels have on usability when used to program robot social-interaction.
3. An investigation into how primitives, regardless of their abstraction level should be orchestrated to create higher level social-interaction. This revealed factors that negatively affect usability when orchestrating robot social-interaction with textual, imperative finite state machines.
4. A taxonomy of primitives at the appropriate abstraction level for programming socially interactive robots.
5. The design and implementation of a software architecture to support the social robot programming API with primitives at the appropriate abstraction level. The architecture is extensible, vendor agnostic and implemented with the Robot Operating System (ROS), a popular robot middleware solution.

1.3 Publications

The publications listed below were produced during my doctoral work. Parts of these publications are included in this thesis.
Chapter 1 - Introduction


1.4 Organisation of thesis

This thesis investigates the usability of tools used by programmers to create robot social-interaction applications. The chapters are described in more detail below:

**Chapter 2 – Motivation:** This chapter presents the evaluation of Ruru (J. P. Diprose, MacDonald, & Hosking, 2011), a novice visual programming language that motivated this thesis. The chapter starts by describing Ruru, which was designed and implemented outside of my doctoral work. This is presented to put the following evaluation and discussion, undertaken as a part of this thesis, in context. The next section describes the evaluation of Ruru and the results of the evaluation. The chapter finishes with a discussion of how Ruru motivated this thesis to explore how to improve the usability of APIs for programming social-interaction.

**Chapter 3 – Related work:** This chapter presents a review of API usability literature and robot social-interaction software. The first section presents a review of API
usability literature. It starts with an overview of the API usability space which is followed by a systematic review of API usability literature focusing on API usability guidelines, API design techniques and API evaluation techniques. This is followed by the implications of these findings for this thesis. The second section starts by providing an overview of social-interaction, paying particular attention to different abstraction levels used to create software for socially interactive robots. This is followed by an analysis of tools used to program social-interaction and finishes with a discussion of the implications of these tools for this research.

Chapter 4 – Methodology: This chapter sets out my approach to exploring how to improve the usability of tools used to program socially interactive robots. The overarching research methodology is an iterative research methodology inspired by design science. At a lower level, the methodology is a combination of user-centred design, user studies, case studies and the Cognitive Dimensions of Notations as a design and analysis tool.

Chapter 5 – Iteration 1: Social-interaction for Nao: This chapter presents the design, implementation and evaluation of an API for programming a Nao humanoid robot; this is the beginning of this thesis exploration into what primitives are appropriate for programming socially interactive robots (RQ1). This chapter also begins an investigation into how primitives, regardless of their abstraction level, should be orchestrated into higher level social-interaction (RQ2). In this case, the usability of textual, imperative state machines are evaluated because they are common means of authoring robot behaviour in the robotics community. The chapter starts with example programs created with the API. It then describes the API in detail, providing explanations for the design decisions made in its creation. A user evaluation of the API is presented, where a Cognitive Dimensions Questionnaire Optimised for Users (Blackwell & Green, 2000), is used as a method to solicit feedback from participants about the usability of the API. This is followed by a description of a stakeholder interview about how the API could be applied to use cases that involve programming life-like humanoid robots. The chapter finishes with a discussion of the evaluation and stakeholder interview results and how this leads to the next iteration of the API.

Chapter 6 – Iteration 2: Social-interaction refactored: This chapter describes a revised version of the API presented in the previous chapter. A number of changes were made to the API to further explore appropriate abstraction levels for
programming social-interaction (RQ1), including: a refactoring of its abstraction level to provide finer control of social-interaction as identified in a previous stakeholder interview, the support of multiple robots and fixes to usability issues identified in the evaluation of the previous API. The chapter starts with example programs created with the revised API and then describes the API design in more detail. The chapter finishes with an evaluation of the second version of the social-interaction API; where a group of four programmers used the API for two weeks to create a social-interaction application for the Nao humanoid robot.

**Chapter 7 – A social primitive taxonomy:** This chapter builds on the previous two chapters, by exploring in more detail what primitives are required for programming social-interaction at an appropriate abstraction level (RQ3). The chapter starts by describing the methodology that was used to derive the taxonomy. This is followed by a description of the taxonomy itself along with supporting data.

**Chapter 8 – Software architecture:** This chapter presents the software architecture (RQ4) that evolved as the first (Chapter 5) and second (Chapter 6) social-interaction APIs were developed. The chapter starts by describing the design goals of the software architecture, which fit into two categories: making the API extensible and vendor agnostic. The following two sections then describe how the API extensibility and vendor agnostic design goals were implemented.

**Chapter 9 – Discussion:** This chapter discusses the findings presented in this thesis. It starts with a chronology of findings presented in this thesis. The chapter then discusses the results of the four research questions and also questions that cut across research questions, such as methodological approaches.

**Chapter 10 – Conclusions and future work:** This chapter presents a summary of my research by outlining the key contributions made to the area of social robot programming as well as suggesting directions for future research.
2 Motivation

This chapter presents the evaluation of the visual language Ruru and a discussion of how Ruru and its evaluation led to this thesis’ focus on improving the usability of APIs for programming socially interactive robots.

The design and implementation of Ruru is not a part this thesis because it was developed during my honours dissertation and a summer scholarship project. However, a high-level overview of Ruru and its design are presented in this chapter to put the evaluation and subsequent discussion in context. The design and implementation are discussed in more detail in (J. P. Diprose et al., 2011). The evaluation of Ruru was conducted as a first step of my doctoral research.

Ruru is an end user visual language designed to program tasks from the Learning Computing with Robots course book (Kumar, 2009); a curriculum including a robot and Myro, a Python-based robot programming framework, to teach introductory computer science and make it “personal, hands on, real, practical and immediate” (Blank, 2006).

The immediacy of robotics introduces problems not encountered in general software development. Collett & MacDonald (2010) argue this relates to the robot platform, the task it performs, and its environmental interaction. A robot platform comprises many input and output devices which produce and consume data that is difficult to understand when solely represented as numbers. Robot tasks often involve movement, which makes the platform inaccessible to the programmer; prevents effective debugging, e.g. if a debug point is reached it will take the robot time to stop; and getting close to the robot may interfere with its execution. Lastly, a robot’s environment is continuous and real time; making scenarios unrepeatable and debugging harder.
Whilst Myro (IPRE, n.d.) considerably lowers the barrier to robot programming as Python is relatively easy to use and Myro’s domain-specific API is intuitive, the programmer still must understand many symbols and abstraction levels.

I developed the live visual programming language Ruru to further separate novices from irrelevant detail, mitigating these problems while retaining the immediacy of the robotics domain. A live visual programming language presents semantic information to the user in real time, Ruru has a level 4 liveness according to Tanimoto’s (1990) scale of liveness. This means that its visualisations are continuously updated based on external events as well as when users directly edit the program.

The chapter starts with an overview of Ruru’s design and an example program (Section 2.1); this is not a part of this thesis but serves to put later sections in context. This is followed by the user evaluation of Ruru, which was conducted as a first step of my doctoral research (Section 2.2). The chapter finishes with a discussion of how visual language design techniques were to be applied to healthcare robot programming and how this led to improving the usability of professional robot social-interaction programming tools (Section 2.3).

2.1 Ruru

This section presents Ruru’s high-level design along with an example program to put the evaluation and subsequent discussion in context. As stated in the introduction, Ruru’s design and implementation are not a part of this project.

2.1.1 Approach

Ruru’s design was based on five main design decisions:

1. The choice of primitives, informed by analysing existing work in the field of robotics.
2. The choice of visualizations for the primitives, influenced by Moody's Physics of Notations theory (Moody, 2009) and Collett & MacDonald’s (2010) augmented reality visualisation.
3. The choice of mechanisms to link the visualizations together.
4. The choice of an approach for complexity management, influenced by Brooks' subsumption robot architecture (Brooks, 1986).
5. The design of the user interaction.
Primitives were chosen so that typical novice robot programs could be expressed. Examples include, make a robot: avoid obstacles, follow a wall, follow/avoid light, follow/avoid a coloured ball, beep at burglars, follow a line, draw a five point star, draw a triangle, draw a square, mow an imaginary lawn and be manually control by a gamepad (Kumar, 2009).

Ruru has two abstraction levels. They are based on Collett’s (2007) robot program model which describes how the components of a robot program interact with each other. Inputs, such as laser sensor data, are sensed via the robots sensors and passed to the Kernel, which represents the logic of the program, such as an obstacle avoidance algorithm. The Kernel uses the Inputs to update its understanding of the world (Internal Data), such as recording the position of obstacles in the world. Lastly, with this updated information the Kernel decides what the robot should do by sending an Output that alters the robots behaviour e.g. a move command to drive the robot around an obstacle.

The higher abstraction level is composed of behaviours. A behaviour is defined as “the way in which an [entity] behaves in response to a particular situation or stimulus” (Oxford University Press, n.d.-a). A robot has multiple stimuli so can have multiple behaviours, each tuned for a particular stimulus. Examples are light following, avoiding obstacles, or being driven by a gamepad. Behaviours define high-level tasks robots can perform and are analogous to Collett’s (2007) model of a robot program as a whole.

The lower abstraction level defines how behaviours function as a combination of primitives. To aid graphic economy, I partition primitives into inputs, which obtain information from the environment; devices, i.e. unique combinations of inputs (such as a Pioneer P3-DX robot or a Scribbler + Fluke robot); thoughts, which process inputs; and
actions, which make a robot do something. Inputs, thoughts and actions respectively correspond to inputs, kernel and outputs of Collett’s model. Devices aid graphic economy by hiding irrelevant inputs, therefore reducing the semantic complexity of a robot program (in addition to partitioning).

Conceptually, inputs, thoughts, and actions form a behaviour. Thoughts make decisions based on data inputs. Decision outcomes alter the state of the robot and, therefore, change what the robot does via actions. Ruru’s primitives were designed by analysing existing robot programs from the Player robot middleware project (“Player Project,” 2006) and the Learning Computing with Robots course book (Kumar, 2009), the Player (“Player Project,” 2006) and Myro (IPRE, n.d.) robot application programming interfaces, existing robotics visual languages such as Microsoft Robotics Developer Studio (Jackson, 2007) and Choregraphe (Pot, Monceaux, Gelin, & Maisonnier, 2009) and robotics related literature such as Brageul (2008). A set of generic primitives was derived that express typical programs from the novice robot programming domain. Figure 2.1 shows a subset of the derived primitives sufficient to give an understanding of their range.

Visual forms were then designed for a subset of the derived primitives, sufficient for proof of concept. Their design was heavily influenced by the Physics of Notations principles (Moody, 2009) and Collett & MacDonald’s (2010) augmented reality debugging system. Their system augments the real world view of a robot with live visualisations of input data; enabling the robot developer to form a better understanding of the robot's perception of its environment. They determined appropriate visualisations for robot data types by classifying them based on whether they were inherently scalar, vector, geometric or abstract and if the data was spatial and or ordered. The design of Ruru’s visual forms was iterated a number of times using paper design techniques. With inputs, for example, we moved from a traditional box and line approach to a live, more concrete approach that explicitly shows input state in a semantically transparent way (Figure 2.2) akin to Collett & MacDonald’s (2010) augmented reality visualizations. Dataflow links were adopted for primitive composition.

At a higher level, a subsumption style approach was adopted so that the user can define semantics between multiple behaviours. This makes the robot act differently based on varying stimuli. Behaviours are ordered using a priority system.
The user interaction was designed to use direct manipulation, in line with the attention investment model (Blackwell, 2002). We avoided property sheets and the like to aid cognitive integration and reduce attention investment issues. Where possible, manipulation by mouse was favoured over the keyboard.

### 2.1.2 Design & examples

Figure 2.2 shows the input visualizations implemented for the Pioneer P3-DX robot device. The laser scanner (left) detects the distance of objects in an 180° arc starting from the left of the robot and finishing at the right (the origin is the black dot). The laser scan has been divided into three slices. The length of a slice equals the mean of all its range readings in that directional slice. For example, in Figure 2.2, the left, centre and right slices have distances of 4m, 3m and 2m respectively, indicating objects are situated at these distances in the vicinity of each slice. Sonar sensors detect the distance of objects from the robot at discrete intervals around the robot e.g. in Figure 2.2 (centre) there are objects near the right of the robot while the left is relatively clear. The bumpers (right) show if the robot has physically driven into an object or not e.g. in Figure 2.2 the second back bumper on the right is pressed in, indicating the robot has hit an object in this area.

![Figure 2.2. Inputs from a Pioneer P3-DX and a gamepad. From left to right: laser scanner, sonar sensors, bumpers and joystick axes.](image)

Figure 2.3 illustrates the X and Y axes of a joystick on a gamepad device. The X axis extends from -1.0 to 1.0 (0.1 in Figure 2.3) and the Y axis extends from 1.0 to -1.0 (-0.2 in Figure 2.3). Figure 3 shows a simple Ruru behaviour that manually controls a robot. A gamepad device provides the inputs for the program (1); this is represented as an icon, overlaid with the axes values for the left joystick. Dataflow connectors (e.g. (2)) connect inputs to thoughts. Here, the left X value is connected to a pair of comparators in a
decision thought (3), which together check if it’s value is less than -0.3 or greater than 0.3. A similar pair of comparators is connected to the Y axis (5).

![Diagram](image)

**Figure 2.3. Driving the robot with a joystick. The robot is currently stationary.**

The comparators within each decision are connected to their own state boxes (4, 5). Parameters from actions (6) can be dropped into state boxes. Actions are a unique combination of parameters, for example, the Move action in Figure 2.3 is composed of heading, direction, distance, speed and movement order parameters. These control how the robot moves. If a comparator evaluates to true, then its associated state and link turn green and a tick icon is displayed. This means that whatever is in the state is executed. Specifically, the parameters contained in the state are applied to the actions they link to and if any decisions are connected to the state they are evaluated. For example, in Figure 2.3, the Y axis has been pushed past 0.3, therefore, the decision “Y axis > 0.3” is true, executing its corresponding state, which sets the speed of the move action to full speed, making the robot drive forward. As can be seen in Figure 2.3, the other comparators have evaluated to false, therefore, their states and associated links are red and cross icons are displayed. Parameters or decisions contained in red states are not executed.
The entire visual notation is live; including visualizations of inputs, decisions (comparators, boxes) and actions. Additionally, actions execute in real time on the robot, even as the programmer edits the program and changes values. For example, Figure 2.3 would actually make the robot drive forward. This allows rapid debugging of robot programs and allows the programmer to develop a greater understanding of how a robot program functions. The current state of the robot is illustrated in the action e.g. in Figure 2.3 the speed of the robot was increased and this is now shown in the move action.

The visualizations of inputs are natural and intuitive as they use a strong spatial metaphor. This allows the position and orientation of an input relative to a device to be expressed visually and processed in parallel by the human perceptual system (Moody, 2009). It also allows one to understand how the environment relates spatially to a device, something that is very important to robot programming. All inputs have unique shapes to support the primacy of shape and different colours to support visual expressiveness.

The decision primitive uses a “thought bubble” metaphor (Figure 2.4). This is a semantically immediate representation reinforcing semantic transparency. The comparators inside decisions use a “fill in the blanks” metaphor so users intuitively know to put values into the comparators operands. The primacy of shape is utilized for the links connecting comparators to states because tick, cross and question mark icons are employed. The different colours, i.e. green for true, red for false and blue for unsure ensure that the most effective visual variable, colour, is put to use. The primacy of shape ensures colour is not relied on, however. Horizontal position is used to indicate which link corresponds to which comparator and state. A semantically immediate parent-child relationship is used when putting parameters in states.

![Decision primitive with true, false and unsure connectors.](image-url)
Actions utilize a clapperboard and film type metaphor i.e. “lights, camera, action!” This is semantically immediate, reinforcing the idea that an action makes a robot do something. The parameters contained in actions are also semantically immediate, utilize as many visual variables as possible and reinforce the primacy of shape. For instance, the direction parameter of the move action uses a gear stick metaphor (using vertical position, dual coding, primacy of shape and colour).

The dataflow links are curved to reinforce primacy of shape and black to utilize colour. Conversely, control transfer links (in-between comparators and states) are straight to reinforce primacy of shape. As explained previously, they also have unique icons (cross, tick, question mark) and are coloured.

User interaction is as direct as possible to decrease the attention investment needed to make Ruru programs. We have avoided property windows, favouring explicit visual parameters and direct mouse manipulation to make a more interactive user experience. For instance, in the move action, the heading parameter is mouse rotateable, the direction gear stick is pushed/pulled forward/backwards to put the robot’s gear into forwards/reverse, the distance on the ruler icon can be increased/decreased to increase/decrease the distance the robot will travel and the speedometer needle can be mouse rotated to specify different speeds. Also, data flow arrows are drawn automatically when users drag a parameter from an action to a state as it implicitly creates a relationship.

If the joystick input in Figure 2.3 changed so that the X axis had a value of -0.5 and the Y axis a value of 0.0 then all of the comparators would be false except for the left most one. That is, the robot’s forward motion will stop as the Y axis value has reduced below the threshold set by the rightmost comparator while the heading parameter that makes the robot turn left has been activated because of the increased negative left X axis value. Ruru programs are designed to be live to aid progressive evaluation: the program can be run partially completed.

2.2 Evaluation

At the beginning of my doctorate, I returned to Ruru and evaluated it with a user study. The purpose was to explore how the design approach (e.g. the application of the Physics of Notations principles) translates into practical efficacy for novice robot programming. Nine participants were recruited, none of them were programmers and they were all from disciplines other than Computer Science or Engineering. Participants were given a
tutorial, examples of how to use Ruru and time to familiarise themselves with Ruru and its environment. They were then given four tasks to perform: two required them to author Ruru programs to make a robot draw a square and to avoid obstacles, one asked them to discover what the robot sensors did, and one asked them to debug and correct a program. Participants were observed while undertaking these tasks and comments recorded. After completing the tasks, participants completed a questionnaire comprising a set of Likert scale questions exploring usability and understandability issues (Table 2.1) and a series of open-ended questions exploring strengths, weaknesses and enhancements. Note that some of the Likert questions were negatively phrased (i.e. strongly disagree is “good”) to check for consistency.

Table 2.1. Likert scale questions.

<table>
<thead>
<tr>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
</tbody>
</table>

The Likert question means are shown in Figure 9. Although there are only nine participants, the results were very encouraging with strong positive feedback shown for usability, understandability, learnability, rememberability, and the ability to get the robot to perform required tasks. Of considerable satisfaction was the degree of fun noted by participants and the degree to which they would recommend it to others, given the original motivation for our work in building novice interest in programming. The low scores on the need for pencil and paper to debug or understand a program and the high satisfaction level with specific language features suggest the abstraction gradient is sufficiently low.
to avoid a barrier to entry for novices. Participants did find it relatively easy to make errors (Q 15) but they also found it relatively easy to find them (Q 16).

![Figure 2.5. Means from Likert scale questions (1=strongly disagree, 5=strongly agree): questions listed in Table 2.1.](image)

The open-ended questions, observations and comments were analysed using a qualitative approach (Miles & Huberman, 1984). In the first step of analysis the questionnaire, observation and comment content was clustered and organised into themes for further analysis. Evidence from the Likert questions was then matched to the themes to broaden the perspective of the analysis.

The high-level analysis results are shown in Figure 2.6. This shows themes, sub-themes, factors contributing to sub-themes, numbers of observations/comments/answer fragments/Likert results supporting the factors and examples of these.

The clarity and intuitiveness of the notation come through strongly with many positive comments. Negative comments referred mainly to easily correctable features, such as an inability to distinguish the front from the back of the robot. The value of liveness was apparent, with participants valuing the progressive evaluation capabilities and immediacy of feedback as aids to understanding the programming tasks. This is an area where strong attention to detail is needed: negative observations related to a delay in the robot responding leading to an impression of task failure and a misunderstanding of the axis system of the gamepad. This carried through to user interaction, where the primacy of the mouse led to some confusion over left and right mouse clicks and their effect, and other
similar misunderstandings. While being observed, participants made many impromptu comments, highlighting the fun nature of the task and indicating a positive change in perception of programming and computer science. This seemed to carry through into observations around perseverance/repetition to gain understanding rather than giving up at the first hurdle.

In summary, the qualitative data and the Likert results are both supportive of the efficacy of Ruru as a novice robotic programming tool. Novice users found it motivating and readily understandable, suggesting it is a good bridge for them into a deeper understanding of computational concepts.

<table>
<thead>
<tr>
<th>Theme</th>
<th>Sub Theme</th>
<th>Factor</th>
<th># Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>User interaction</td>
<td>Positive</td>
<td>Keyboard</td>
<td>Everything could be done without use of keyboard</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Left/right click confusion</td>
<td>Confusion between left/mouse click</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Selecting form</td>
<td>Unable to select certain things until the larger icon is moved away</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Preservence/repetition</td>
<td>Experience, the more use uses a program the more easy/harfar it becomes.</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td></td>
<td>Ability to see the defaults/each instruction when 3 or 4 are linked to the same action set would be helpful</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User interface</td>
<td>Positive</td>
<td>Mouse</td>
<td>The symbols were really good – easy to understand exactly what they did</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>New diagram metaphor</td>
<td>The representation by flow diagram with action images seems very intuitive</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Situative actions/moves</td>
<td>Some such as those with visual as they coalesced with the axis</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>True/False boxes</td>
<td>The colour change of the true/false boxes</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Hard to tell front/back of</td>
<td>Which was the front?</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Dual coding</td>
<td>Adding text labels would help remembering what they do.</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Intuitive</td>
<td>The use of mathematical signs was also clear.</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Mindshare</td>
<td>Data interpretation around mathematics is possible</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Response times</td>
<td>The robot took a little while to respond sometimes so I didn’t realise I got things right.</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Perception</td>
<td>The basic use of the green tick and red cross showed this.</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Bumper symbols</td>
<td>The robot took a little while to respond sometimes so I didn’t realise I got things right.</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Clarity</td>
<td>The symbols were really good – easy to understand exactly what they did</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Math v No Math</td>
<td>The basic use of the green tick and red cross showed this.</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Intuitive</td>
<td>The symbols were really good – easy to understand exactly what they did</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Flow diagram metaphor</td>
<td>The representation by flow diagram with action images seems very intuitive</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>True/False boxes</td>
<td>The colour change of the true/false boxes</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Obvious</td>
<td>The big clear use of symbols ... easy to see the problem</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Symbols</td>
<td>The big clear use of symbols ... easy to see the problem</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Effort of True/False boxes</td>
<td>The basic use of the green tick and red cross showed this.</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Effect of X Y axes of gamepad</td>
<td>The basic use of the green tick and red cross showed this.</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Name gamepad stops</td>
<td>The big clear use of symbols ... easy to see the problem</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Name gamepad stops</td>
<td>The big clear use of symbols ... easy to see the problem</td>
</tr>
<tr>
<td></td>
<td>Positive</td>
<td>Selecting form</td>
<td>Unable to select certain things until the larger icon is moved away</td>
</tr>
<tr>
<td></td>
<td>Negative</td>
<td>Selecting form</td>
<td>Unable to select certain things until the larger icon is moved away</td>
</tr>
</tbody>
</table>

Figure 2.6. Qualitative analysis themes and analysis.

2.3 Exploration of healthbot programming

The evaluation of Ruru showed that the application of the Physics of Notation principles holds promise for overcoming some of the common disadvantages of existing robotics visual notations, such as lack of graphic economy, visual expressiveness, semiotic clarity and perceptual discriminability. The end user evaluation also demonstrates that Ruru is motivational for novice programmers, with users appreciating
the clarity of its notation and the liveness and immediacy of its environment as they come
to grips with computational concepts.

2.3.1 From end users to programmers

This motivated me to find out if the same techniques that were used to design Ruru
could be used to create a generalizable end user robot programming language capable of
programming more complex robot tasks, such as programming an unmanned aerial
vehicle to spray weeds autonomously (J. P. Diprose, 2011). I planned to create end-user
programming languages capable of programming robots used in three different
application domains, including healthcare, agriculture and meat processing.

I started by investigating how end users could program robots used in the University
of Auckland Healthbots project (Kuo, 2012). The scenarios being used with these robots
included: medication reminding, vital signs monitoring and a social companion. Two of
the robots, iRoboiQ and Cafero can be seen in Figure 2.7.

![Figure 2.7. Healthbots robots: left, iRoboiQ; right, Cafero. Used with permission of Healthbots.](image)

The scenarios used on the Healthbots project involved three different types of
programming:

1. User interface programming: programming the screens the users see on the robot.
2. Integration of medical devices: for example, retrieving data from a blood pressure
   monitor.
3. Programming human-robot interaction: for example, speaking, gazing and
gesturing at the patient.
Of the three types of programming used in the Healthbots scenarios, human-robot interaction was the “robotic” aspect that could contribute toward a generalised end user robot programming language. The other two areas could be addressed with existing technologies and methods: the user interface could be created with a user interface builder (Myers, Hudson, & Pausch, 2000); and the medical devices could be integrated by a programmer knowledgeable enough to use the device manufacturers APIs.

I discovered two key things as I investigated human-robot interaction and how it was used in the Healthbots scenarios. First, the main users of human-robot interaction capable robots are professional programmers. This is because human-robot interaction is still largely at the research stage and making a robot interact with humans is a challenging task that involves the development and integration of many low-level perception and actuation algorithms. Some robots used for human-robot interaction even include disclaimers that they require advanced programming and that they are only available to educational institutions and researchers, not end users. For example DARwIn-OP (RobotShop Distribution Inc, 2015), Nao (RobotShop Distribution Inc, 2012a) and Robokind R50 (RobotShop Distribution Inc, 2012b). The second key point is that existing end-user and developer oriented human-robot interaction programming tools appeared to have usability issues, regardless of whether they were: targeted at end users or professional programmers; APIs; visual languages; or domain-specific languages. The main problem seemed to be that they did not have well thought out abstraction levels; users were basically cobbling together low-level algorithms to create human-robot interaction.

This led me to focus on studying how to improve the usability of APIs for programming social-interaction, rather than on creating general end user robot programming tools (social-interaction is a subset of human-robot interaction). In this case, there is a greater immediate benefit in improving the usability of tools for programmers because they are the main users of robots that interact with people. The knowledge gained from this can be used to improve end user tools when these robots are more commonly used by novice programmers.
2.4 Summary

This chapter started by providing a high-level overview of Ruru, a visual language and environment for novice robot programming developed during my honours dissertation and a summer scholarship project.

Ruru was subsequently evaluated with a user study during my doctoral work to explore how its design approach, such as the application of the Physics of Notation principles (Moody, 2009) affected its usability. The evaluation demonstrates that Ruru is motivational for novice programmers, with users appreciating the clarity of its notation and the liveness and immediacy of its environment as they come to grips with computational concepts.

The chapter then discussed the implications of the results of the evaluation and how this led to the final direction of this thesis; which focuses on improving the usability of tools used by professional programmers to author robot social-interaction.

The next chapter reviews socially interactive robots and API usability literature. The review of socially interactive robots focusses on social-interaction programming tools and their limitations. The review of API usability literature focuses on API design guidelines, API design methods and API evaluation methods.
This chapter reviews two research areas relevant to this thesis: API usability (Section 3.1) and robot social-interaction programming tools (Section 3.2). The review of the API usability examines general guidelines and techniques used to design and evaluate the usability of APIs and discusses the implications of these findings for this thesis. The review of robot social-interaction programming tools examines relevant tools through the lens of social-interaction abstraction levels and discusses the implications of these findings for this thesis.

3.1 API usability

This section presents a review of API usability literature. It starts by providing an overview of the API usability space (Section 3.1) which is followed by a systematic review of API usability literature that focuses on API usability guidelines, API design techniques and API evaluation techniques. The systematic literature review starts by detailing the approach used for the literature review (Section 3.1.2). Next, the results of the literature review are presented (Sections 3.1.3, 3.1.4 & 3.1.5). The review finishes with a discussion about the implications of the findings (Section 3.1.6).

3.1.1 Introduction

Stylos & Myers (2007), pioneers in the field of API usability, have created a comprehensive overview of the API design field, including: what an API is, why developers use APIs, what makes an API good and who the stakeholders of APIs are. These are discussed in the following paragraphs.
An API is a collection of code, packaged with an interface to enable other developers to use it (Stylos & Myers, 2007). There are several synonyms for APIs, which include: development kits, frameworks, libraries, toolkits. Programming languages, tools and documentation are not APIs, however, they do influence the usability of APIs.

According to Stylos & Myers (2007), there are four main reasons why developers use APIs. First, APIs save time by allowing code to be reused. Second, APIs conceal implementation details, which: makes the application being programmed robust to lower level implementation changes; and makes it easier for the programmer to perform a task because the concepts are presented at an abstraction level closer to their problem domain (Green & Petre, 1996, p. 19). Third, APIs can provide functionality which is hard to implement, for example, the popular OpenCV computer vision library (Itseez, n.d.). Lastly, products built with an API provide a standard interaction experience for consumers.

The efficacy of an API is determined by its usability and its power (Stylos & Myers, 2007). A good API should be simple enough to learn and use (Roberts & Johnson, 1996). An API should be usable because usability has a significant impact on developer productivity (Clarke, 2004). Usability has a number of properties, including learnability, memorability, efficiency, low error rate and satisfaction (Nielsen, 2012). As well as being usable, a good API should be powerful enough to produce solutions for a particular problem domain (Roberts & Johnson, 1996, p. 1), otherwise, it won’t be useful to anyone! The power of an API has five properties: expressiveness, extensibility, evolvability, performance and robustness (Stylos & Myers, 2007).

The stakeholders involved in API design include API designers; the programmers who use the API; and the end users who consume the product created with the API (Stylos & Myers, 2007).

The remainder of this chapter details a systematic review of API usability literature uncovering API usability guidelines and API design and evaluation techniques.

3.1.2 Approach

The methodology for systematically searching literature used in this study is based on guidelines provided by Kitchenham et al. (2009). The goal of the systematic literature search was to gain a thorough overview of API usability literature.
Research questions

The research question that guided the literature search was:

RQ1  What guidelines exist for designing usable APIs?
RQ2  What techniques have been used to design usable APIs?
RQ3  What techniques have been used to evaluate the usability of APIs?

The first research question is concerned with finding general guidelines for designing usable APIs. The second research question is concerned with examining the techniques people use to design usable APIs and the last is concerned with finding techniques for evaluating API usability.

Inclusion & exclusion criteria

Papers were excluded if they were research abstracts.

Search process

Databases, workshops and conferences that are known to contain information related to API usability were searched. The databases are ACM Digital Library (ACM, n.d.), IEEEXplore (IEEE, n.d.), SpringerLink (AG, n.d.) and ScienceDirect (B.V., n.d.). The search terms for the databases were very specific to APIs and usability because otherwise thousands of results were returned in each database, an example is shown in Figure 3.1.

API* AND usability

Figure 3.1. General structure of database search terms.

Workshops and conferences that were known to contain literature related to API usability included: PPIG, VL/HCC and CHI. The search terms for these venues were able to be more general, an example is shown in Figure 3.2. The first part of the query contains synonyms for APIs and the second part contains synonyms related to API usability.

(API* OR “class library” OR library OR “software framework” OR “development kit” OR toolkit) AND (usability OR “human-computer interaction OR evaluat* OR design)

Figure 3.2. General structure of venue terms.
Data selection

During the data selection stage: the list of papers were compiled. From an initial corpus of 261 articles, the following process was carried out to refine and extract data:

1. Duplicate articles were filtered.
2. The sample was refined based on titles and abstracts.
3. The sample was refined based on the full text. Here I looked to see if the paper answered the research questions. If it did it was retained and if it didn’t it was discarded. Papers that had little or no experimental data were discarded.
4. Backward citations. While refining the sample based on full text, I looked for any relevant papers referenced in the text.
5. A free search was conducted using Google and Google Scholar to fill in any gaps in the initial search.

A total of 56 papers remained, 35 from the initial search and 21 from backwards citations and free searches (Appendix A.2). The following three sections present the results of the literature search, including API usability guidelines, API design methods and API usability evaluation methods.

3.1.3 API usability guideline results

Stylos & Myers (2007) suggest that API usability guidelines fall into two general categories: guidelines derived from experimental work and expert advice.

Guidelines derived from experimental work are discussed in the next three sections, including Design of API structure, Design of API documentation and Design of API environment. Guidelines based on expert advice are discussed in the fourth section: Expert API design advice.

Design of API structure

The Carnegie Mellon University Natural Programming Project (“Natural Programming,” n.d.), has conducted a number of usability studies that explore how to provide API design advice that is applicable to a range of APIs, not just the specific API being evaluated. Their design advice includes parameterising object’s constructors (Stylos & Clarke, 2007), avoiding the factory pattern (Ellis, Stylos, & Myers, 2007), and
appropriate method placement (Stylos & Myers, 2008). These are explained in the following paragraphs.

Through a think-aloud usability study, Stylos and Clark (2007) discovered that programmers perform better when they can set an object’s parameters via setter methods rather than the object’s constructor (which in this case has no parameters). This is because setting parameters on the object’s constructor interferes with a programmer’s strategy for finding the correct object. Specifically, when parameters are required in an object’s constructor, programmers are forced to stop their search for the correct object and instead focus on fixing compilation errors caused by incorrect constructor parameters. This is a form of premature commitment when explained with the Cognitive Dimensions of Notations. An overview of this process is illustrated in Figure 3.3.

![Figure 3.3. The process of searching for a desired object; adapted from (Stylos & Clarke, 2007). Satisfying the required constructor is a bottleneck.](image)

In a similar study, Ellis, Stylos and Myers (2007) evaluated the effect that the factory design pattern has on usability. The factory pattern is a method of creating objects that conform to a particular interface without needing to know what the specific object is. The results of the study indicate that factories are detrimental to usability because programmers have significant trouble understanding how to operate them. The authors propose that an alternative design pattern called a ‘class cluster’ should be used instead of factories because programmers find them easier to use and they offer many of the same benefits as factories. A class cluster instantiates concrete subclasses from a super class constructor, rather than the factory pattern, which instantiates concrete classes from a method on a factory class.
The last study by Stylos and Myers (2008) that provides generic API design guidelines evaluated the effect of method placement on API usability. This study found that programmers take more time to solve a task when the methods they have to use are spread across more than one object. An example is illustrated in Figure 3.4.

\begin{verbatim}
printer.print(document)
\end{verbatim}

or

\begin{verbatim}
document.print(printer)
\end{verbatim}

\textbf{Figure 3.4. Method placement, adapted from} (Stylos & Myers, 2008, p. 105)

The reason why programmers are less efficient when they need to use methods spread across multiple objects is because it interferes with the process programmers use to solve problems (Stylos & Myers, 2008). Programmers start by choosing an initial class that they think will solve their problem. They then explore the methods of that class to see which ones will meet the objectives of the task. If programmers cannot find a method that they expect to find (e.g. because it is on a helper class) they may doubt the initial class they chose. When they do realise the method they need is on another class they need to find it and continue the process of solving their problem. This is illustrated in Figure 3.5.

\textbf{Figure 3.5. Process involved in using helper classes, adapted from prose in} (Stylos & Myers, 2008).

Design of API documentation

As well as providing design guidelines for the structure of APIs, scientific literature has also experimentally derived design advice for API documentation. Robillard and DeLine (2011) have performed the most thorough work in this area, through a field study
of the problems 440 professional programmers encountered when learning new APIs. The study used surveys and interviews that specifically explored issues related to API documentation. The authors derived five factors to take into account when designing API documentation, these are illustrated in Figure 3.6.

![Figure 3.6. Five important factors of API documentation, from (Robillard & DeLine, 2011).](image)

*Documenting intent* involves explaining to API users why particular design decisions were made (Robillard & DeLine, 2011). It is beneficial to document intent because it helps API users understand how to use the API efficiently and correctly. This is particularly important when API users are implementing complex code or are debugging performance related issues.

*Code examples* are often provided by API documentation and show developers how to use an API (Robillard & DeLine, 2011, p. 718). However, it is beneficial if the code examples also show the best practices for using the API. Nasehi & Maurer (2010) have found that API unit tests can act as good examples of API usage.

*Matching scenarios to API elements* is important to show developers how to implement particular scenarios (Robillard & DeLine, 2011, p. 719). This is because a large part of learning an API involves finding out how to implement particular scenarios, for example finding out how to make a robot point at someone's head would be linked to an example or explanation of how this is implemented with an API.

*Documenting factors affecting penetrability* is important to minimise the degree that developers have to explore the internal workings of the API (Clarke, 2004; Robillard & DeLine, 2011). This includes: documenting the performance characteristics of API primitives; providing detailed information about error handling behaviour; and making it clear to developers which methods perform more than one important operation (Robillard & DeLine, 2011, p. 721).
The last factor to consider when designing API documentation is its format and presentation (Robillard & DeLine, 2011, p. 722). Robillard and DeLine found that most developers benefit from documentation presented with a linear narrative and that documentation presented as a bunch of hyperlinks can be hard to comprehend.

Design of API environment

The last set of experimentally derived guidelines relates to the environment from which an API is used. In particular, much work examines the effect of static and dynamically typed languages on API usability (Endrikat, Hanenberg, Robbes, & Stefik, 2014; Kleinschmager, Hanenberg, Robbes, Tanter, & Stefik, 2012; Petersen, Hanenberg, & Robbes, 2014; Spiza & Hanenberg, 2014). This research has found that static type systems are indeed beneficial to API usability (Kleinschmager et al., 2012), even: with and without type documentation (Endrikat et al., 2014); if the type checking functionality is turned off, that is, only type names remain (Spiza & Hanenberg, 2014); and even with an IDE (Petersen et al., 2014). These studies are explained below.

Kleinschmager et al. (2012) evaluated the effect of a static type system on API usability when developers performed maintenance tasks. Static type systems were found to improve development time when developers used classes they hadn’t encountered before and to fix type errors. This is because developers could discover an element’s type faster through the static type system, than by searching unaided through source code (Figure 3.7a). However, static type systems did not help developers fix semantic errors. This research spurred other researchers to examine the effects of static systems on usability in more detail.

Endrikat et al. (2014) compared the usability of APIs used with type documentation against those used with static type systems. They found that type documentation and static type systems both improve developer performance, however, developers were faster when using the static type system because they didn’t have the overhead associated with finding and reading documentation (Figure 3.7b).
In a similar study, Spiza and Hanenberg (2014) compared the usability of APIs used with type names that were not type checked vs a normal static type system. They found that just having type names alone improved usability, however, the static type system had a greater improvement on performance. Additionally, not having a type checker (as in the first condition) enables erroneous type names to be specified, which has a significant cost to performance.

The last important experiment shows that static type checking is beneficial to programmer performance even when an IDE is used (Petersen et al., 2014).

Expert API design advice

A number of authors have written books or articles about API design based on their experience building widespread APIs over their years as developers (Stylos & Myers, 2007). These include Joshua Block who worked on many of the core Java libraries during his time at Sun Microsystems (Bloch, 2001, 2006); Cwalina and Abrams, core .Net developers from Microsoft (Cwalina & Abrams, 2008); Jaroslav Tulach who developed the popular Netbeans platform (Tulach, 2008); and Michi Henning a former member of the architecture board for CORBA, the chief scientist at ZeroC and now a software engineer at Canonical (Henning, 2007).
3.1.4 API design method results

A number of methods are used to design APIs, including user-centred design, the Cognitive Dimensions of Notations, object-oriented methodologies, requirements analysis and the analysis of existing APIs. The following paragraphs describe each design technique and how they have been applied to API design.

User-Centred Design

User-centred design is a methodology that puts the needs of users and the tasks they perform at the centre of the design, an iterative process, where the solution evolves through continued development and evaluation (Norman, 2002). A number of methods are often used in conjunction with user-centred design, including: user participation, focus groups, questionnaires, ethnographic observations, cognitive walkthroughs, expert evaluations, usability studies, interviews, scenario modelling, activity modelling and personas (Bevan, 2003; Bødker, 2000; Gulliksen et al., 2003; Heim, 2008, pp. 85–86; Mijailović & Miliev, 2014; Norman, 2002).

McLellan et al. (1998), Clarke (2004) and Stylos et al. (2008a) pioneered the application of user-centred design as a key part of designing usable APIs. The techniques used along with the user-centred design of APIs include shareholder interviews (Stylos et al., 2008a), scenarios (Clarke, 2004), personas (Clarke, 2004), usability studies e.g. (J. K. Beaton, Myers, Stylos, Jeong, & Xie, 2008; Clarke, 2004) and the Cognitive Dimensions of Notations framework (Clarke, 2004).

Many of the methods mentioned in the previous two paragraphs are applied as API evaluation techniques as a part of the user-centred design process and are explained in more detail in Section 3.1.5.

Cognitive Dimensions of Notations

Green & Petre (1996) created the Cognitive Dimensions as a vocabulary to help designers weigh up design decisions that affect the usability of notations. Visual languages, graphical user interfaces and programming languages are all examples of notations that can be evaluated with the Cognitive Dimensions. The framework is comprised of 13 ‘dimensions’; each dimension describes a unique way that the structure of a notation can be altered to affect usability. The 13 dimensions and their definitions
are defined below; the definitions for dimensions 1 and 3-12 are quoted from (Green & Petre, 1996, p. 11) and dimensions 2 and 13 are paraphrased from (Green & Petre, 1996).

1. **Abstraction gradient (or level):** What are the minimum and maximum levels of abstraction?
2. **Closeness of mapping:** How close is the mapping between the notation and the domain it has been designed for?
3. **Consistency:** When some of the language has been learnt, how much of the rest can be inferred?
4. **Diffuseness & terseness:** How many symbols or graphic entities are required to express a meaning?
5. **Error-proneness:** Does the design of the notation induce ‘careless mistakes’?
6. **Hard mental operations:** Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what’s happening?
7. **Hidden dependencies:** Is every dependency overtly indicated in both directions?
8. **Premature commitment:** Do programmers have to make decisions before they have the information they need?
9. **Progressive evaluation:** Can a partially-complete program be executed to obtain feedback on ‘How am I doing’?
10. **Role-expressiveness:** Can the reader see how each component of a program relates to the whole?
11. **Secondary notation:** Can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the ‘official’ semantics of the language?
12. **Viscosity:** How much effort is required to perform a single change?
13. **Visibility & juxtaposability:** Are components visible? Can they be viewed next to each other?

It important to note that changing one dimension will very likely affect another, which is why designers need to be aware of the trade-offs that can occur between dimensions when they are changed (Green & Petre, 1996).

Clarke (2004) presents a seminal application of the Cognitive Dimensions to API usability. Clarke reports that he together with the others in the Microsoft Visual Studio usability group frequently use the Cognitive Dimensions to evaluate their APIs. They redefined the names of some of the dimensions to make them more applicable for describing aspects of API usability. However, in this thesis, I refer to the original 13 Cognitive Dimensions to be consistent with the wider research community as argued for by (Dagit et al., 2006).
Lastly, many others in the API usability research community have used the Cognitive Dimensions to design, evaluate or discuss the usability of their APIs, including: (J. K. Beaton et al., 2008; Clarke & Becker, 2003; Clarke, 2004, 2005; Farooq & Zirkler, 2010; Mijailovi & Miliev, 2014; Piccioni, Furia, & Meyer, 2013; Robillard & DeLine, 2011; Stylos & Clarke, 2007; Stylos, Clarke, & Myers, 2006; Stylos et al., 2008a).

**Object oriented research techniques**

The object-oriented research community has created its own methods for designing object-oriented frameworks, a type of API (Stylos & Myers, 2007). A prime example is the Evolving Frameworks Pattern Language (Roberts & Johnson, 1996), a set of design patterns that guide a designer through the process of creating an object oriented framework. The *three examples pattern* is one of the approaches advocated by this design methodology, which is used to create a generic framework. In this approach, the designer takes three exemplar applications and generalises from their common features to create the generic framework.

**Using existing work as a guide**

The last common method of API design is to use existing work as a guide for the design of the primitives in the API. This includes analysing existing APIs (Scheller & Kühn, 2013) or basing the design on existing applicable models (Schreckling, Posegga, & Hausknecht, 2012).

**3.1.5 API usability evaluation results**

The research community uses many API evaluation methods; these are illustrated in Figure 3.8. The following paragraphs describe how each evaluation technique has been applied to API evaluation.
Figure 3.8. Methods for evaluating APIs.

User studies

Lab controlled user studies are by far the most common method of evaluating APIs, evident from the large number of papers using this technique: (J. Beaton, Jeong, Xie, Stylos, & Myers, 2008; J. K. Beaton et al., 2008; Bell, Rieman, & Lewis, 1991; Brunet, Serey, & Figueiredo, 2011; Cardoso & José, 2013; Clarke, 2004; Duala-Ekoko & Robillard, 2012; Ellis et al., 2007; McLellan et al., 1998; Oney, Myers, & Brandt, 2014; Piccioni et al., 2013; Ramakrishnan et al., 2014; Scheller & Kuhn, 2012; Scheller & Kühn, 2013; Stylos & Clarke, 2007; Stylos & Myers, 2008; Stylos et al., 2008b).

User studies evaluate a product by testing it with target users (Dumas & Redish, 1999; Health & Human Services, n.d.). The goal of the evaluation is to discover aspects of the product that hinder its usability so that they can be fixed. The participants involved in a user study perform tasks whilst the researcher observes and records what they say and how they interact with the product. This data, which can be either qualitative or quantitative, is then analysed and used to improve the product.

In the field of API usability, there are a variety of tasks that are usually used during user studies, including programming tasks, implementing pseudo code, performing readability evaluations and performing tasks to evaluate API documentation. These are explained below.
Programming tasks are the most common task type performed during API usability studies; they require the participant to perform tasks with a real API that can be compiled and run (Brunet et al., 2011; Cardoso & José, 2013; Clarke, 2004; Duala-Ekoko & Robillard, 2012; Ellis et al., 2007; Piccioni et al., 2013; Scheller & Kuhn, 2012; Scheller & Kühn, 2013; Stylos & Clarke, 2007; Stylos & Myers, 2008; Stylos et al., 2008a). Programming tasks usually match common scenarios that would normally be programmed with the API, however, they can also focus on particular areas that need to be reviewed (Clarke, 2004).

Pseudo code based tasks are another common task type (Brunet et al., 2011; Stylos & Myers, 2008; Stylos et al., 2008a), often used as a part of a user-centred design process (Stylos & Myers, 2008; Stylos et al., 2008a). In a pseudo code task, a participant is given a task to implement and solves it by writing pseudo code instead of using a real API (Stylos et al., 2008a, p. 191). The participant has complete freedom over how they write the pseudo code, the purpose of this is to understand the API elements a participant expects to have for a particular scenario (Stylos et al., 2008a, p. 191).

Readability & comprehension evaluation (McLellan et al., 1998; O’Callaghan, 2010; Stylos & Clarke, 2007). In this type of evaluation, the participant reads pre-written code and either describes what it does (Brown, 2003; Stylos & Clarke, 2007) and or asks questions about what it does until they understand it (Brown, 2003; McLellan et al., 1998). The readability of the code is qualitatively assessed by either: evaluating how closely the programmer’s description matches what the code actually does, as Stylos & Clarke (2007) did; or using the questions programmers asked as indications of the APIs limitations (McLellan et al., 1998).

The last category of tasks are used to evaluate an APIs documentation (Friendly, 1996; Jeong et al., 2009). For example, Jeong et al. (2009) gave participants a task where they had to find the services required to create a sales order for an Enterprise Service Oriented Architecture (eSOA). The participants were given the documentation for the eSOA and had to find the required services by reading and searching it. The participants didn’t implement an application or use programming tools. Issues with the documentation were identified by observing participants thinking aloud (discussed next).

While the programmer is performing tasks, the researcher is collecting data. Data collection methods used during an API user study include the think aloud protocol (Lewis
& Rieman, 1993), quantitative data, questionnaires and interviews. These are explained in the following paragraphs.

The think aloud protocol is the most common method of gathering data during API usability evaluations (J. Beaton et al., 2008; J. K. Beaton et al., 2008; Brown, 2003; Brunet et al., 2011; Duala-Ekoko & Robillard, 2012; Ellis et al., 2007; McLellan et al., 1998; O’Callaghan, 2010; Piccioni et al., 2013; Ramakrishnan et al., 2014; Stylos & Clarke, 2007; Stylos & Myers, 2008; Stylos et al., 2008a). In this protocol the researcher asks the participant to verbalise what they are thinking as they perform the task, including: what they are doing, questions they have and what they can see (Lewis & Rieman, 1993, p. 83). The advantage of this method is the researcher can gain a rich understanding of how the participant solves a task with a particular API (Stylos et al., 2006, p. 132).

As with any user study, researchers can collect data in other ways, for example by making observations, recording what participants say and do, giving the participants questionnaires or interviewing them. Researchers have applied many of these techniques during API usability studies, for instance: (Brown, 2003; Duala-Ekoko & Robillard, 2012; Piccioni et al., 2013; Scheller & Kühn, 2013). Of particular importance is the Cognitive Dimensions Questionnaire Optimised for Users (CD Questionnaire) (Blackwell & Green, 2000), which has been used to gather data related to API usability. For instance, Clarke (2001) used this questionnaire to gather data about the usability of the programming language C#, parts of which are technically an API.

Case studies

Yin (2013, p. 2) defines a case study as a research methodology that explores a “contemporary phenomenon (the ‘case’) in a real-world context”. Case studies are frequently performed in the social sciences (Runeson & Höst, 2009) and are suited to answering exploratory how and why research questions where variables cannot easily be controlled, for example, the environment where the phenomenon occurs (Yin, 2013). Data triangulation is a key part of case study research, which is used to compensate for the lack of controlled variables (Yin, 2013).

Some researchers have used case studies to evaluate APIs (Burns, 2013; Gerken, Jetter, & Reiterer, 2010; Grill, Polacek, & Tscheligi, 2012; Lobato, Garcia, Lucena, & Romanovsky, 2006). The data collection techniques used in these case studies include
developer workshops, interviews, observations and concept mapping (a technique discussed later on).

**Expert evaluations**

Expert evaluations are where HCI researchers evaluate an API using a set of predetermined criteria (J. K. Beaton et al., 2008; Stylos & Myers, 2007). The goal of an expert evaluation is to understand the trade-offs associated with a particular design (Heim, 2008, pp. 172–173). The two sets of criteria used for expert evaluations in the programming field are Nielsen’s heuristics (Nielsen, 2005) and the Cognitive Dimensions of Notations (Green & Petre, 1996).

Nielsen’s heuristics are a set of 10 criteria that focus on making the designer aware of common ways that user interfaces can have a breakdown in usability (Nielsen, 2005). Beaton et al. (2008), members of the Carnegie Mellon University Natural Programming Project (“Natural Programming,” n.d.), argue that a subset of these heuristics are applicable to API design, including consistency and standards, error prevention and help & documentation. They later used these heuristics as a part of the design process for evaluating the SAP Gateway VS tool (Faulring, Myers, Oren, & Rotenberg, 2012). Nielsen’s heuristics have also been used to evaluate API documentation (Purho, 2000).

The Cognitive Dimensions of Notations was initially created as a tool to help designers weigh up design decisions that affect the usability of notations such as visual languages, graphical user interfaces, APIs and programming languages (Green & Petre, 1996). It has been applied as an expert evaluation technique by many in the visual language and domain-specific language community e.g. (Ford, 1996; Grundy, Mugridge, Hosking, & Kendall, 2001; Mapelsden, Hosking, & Grundy, 2002), however, few have used it as an API expert evaluation technique. Watson (2009) performed a detailed analysis of the consistency of a new Microsoft API and Mijailovi & Miliev’s (2014) performed a heuristic evaluation of their own API. Researchers applying the Cognitive Dimensions as an expert API evaluation technique should take care not to rely on the results as justification that their API is usable; but instead, the results should be used early in the design process to help design a better API (Dagit et al., 2006).
Cognitive walkthrough

A cognitive walkthrough is a form of role-play where the evaluator steps through the process of solving a scenario using a persona to inform them how to think like the target user (Wharton, Rieman, Lewis, & Polson, 1994). The goal is to find process related usability problems during the design stage before a large amount is invested in implementation. Heim (2008, pp. 163–164) provides five questions evaluators ask at each stage of the thought process:

1. *Can the user accomplish this goal?*
2. *Can the system recover from deviations from the scenario or user errors?*
3. *What choices are available?*
4. *Do I know how to proceed?*
5. *How difficult is it to accomplish this task?*

Cognitive walkthroughs have a long history of being applied to programming related tasks, for instance, Bell et al. (1991) used cognitive walkthroughs to evaluate the usability of the visual programming language ChemTrain. They termed their derived technique *programming walkthroughs,* which like traditional cognitive walkthroughs are also conducted during the design stage of a system. More recently researchers have used and adapted cognitive walkthroughs for the task of API evaluation (J. K. Beaton et al., 2008; Farooq & Zirkler, 2010; Faulring et al., 2012).

The Carnegie Mellon University Natural Programming Project (“Natural Programming,” n.d.) have used cognitive walkthroughs as a technique for API evaluation, for instance, Faulring et al. (2012) used cognitive walkthroughs to evaluate the SAP Gateway VS tool. The researchers built an application with the SAP Gateway VS tool, recording the problems they found along the way. Additionally, the CMU researchers have adapted the questions asked during a cognitive walkthrough to be more applicable to API evaluation (J. K. Beaton et al., 2008, p. 33), these include:

1. *What are the user’s goals?*
2. *What options does the user perceive?*
3. *Which perceived options seem most actionable?*
4. *What result will the user conclude has occurred?*
Farooq & Zirkler (2010) adapted cognitive walkthroughs into a collaborative usability evaluation method called *API Peer Reviews*. The API Peer Review starts with a planning session to choose the goals, code samples and reviewers. Next, the review takes place. A facilitator guides several reviewers through the steps needed to implement the code sample; at each step the facilitator asks the reviewers questions (e.g. what is unintuitive about this method name?), motivating them to provide feedback about the API.

**Concept mapping**

Concept mapping is a technique used to model a participant's mental model of a topic (Gerken, Jetter, Zöllner, Mader, & Reiterer, 2011; Joseph D Novak, 1990). This is accomplished by having the participant write the concepts associated with the topic on post-it notes, arranging them into a sensible structure and drawing relationships between them. The participant’s model can be compared with a ground truth model to gauge how well they understand the topic. An example is shown in Figure 3.9.

![Concept Map](image)

**Figure 3.9.** Part of a concept map illustrating a student's understanding of scientific concepts. From (Joseph D Novak, 1990),

Concept mapping was first used in educational research to understand children’s comprehension of scientific concepts (Joseph Donald Novak, 1977, pp. 88–93) and later to understand how this comprehension changed over time (Joseph D Novak, 1990). Gerken et al. (2011) applied the concept mapping technique to API evaluation, with the purpose of understanding how programmers mental models of an API changed with time.
API meta-data analysis

The last API evaluation technique is the analysis of API meta-data, including both quantitative and qualitative data. Several systems compute quantitative metrics based on the structure and implementation of an API (Hou, Rupakheti, & Hoover, 2008; Rama & Kak, 2013; Scheller & Kuhn, 2012; Scheller & Kühn, 2011, 2013, 2015; Souza & Bentolila, 2009). Scheller et al. (2012; 2011, 2013, 2015) have produced a framework that computes factors they argue are associated with API usability. Examples of computed factors include the number of methods in a class, the number of overloaded methods and the number of parameters in each method. Rama & Kak (2013) and Souza & Bentolia (2009) have also made tools that compute similar factors, however, Souza & Bentolia’s tool also has facilities for visualising the results. The products of an API can also be evaluated, for example, Alcock, Lorier, & Nelson (2012) evaluated the usability of their library Libtrace by comparing the lines of code required to create the same example programs with a variety of similar libraries.

Other systems compute metrics based on qualitative data. Ratiu & Deissenboeck (2006a, 2006b) and Ratiu & Jurjens (2007) have developed systems capable of comparing APIs to ontologies to understand how well they map to a particular problem domain. Their system finds mismatches between the concepts and relationships in an API to the domain the API encodes. They accomplish this by comparing the entities and relationships in an API to the concepts and relationships in WordNet (Leacock & Chodorow, 1998), an ontology of the English language.

Other meta-data analysis techniques manually analyse qualitative data, for example, the analysis of bug posts (Zibran, Eishita, & Roy, 2011), commit history (Bartsch, 2011) and forum discussions (Hou & Li, 2011).

3.1.6 Implications for research

Existing API design research has a number of implications for designing usable APIs and the research conducted in this thesis.

A number of studies have defined general API design guidelines, which inform designers of ways they can improve the usability of an API through its structure, documentation and environment. Whilst this research is incredibly useful to the API designer, a design method is still needed to design an API for a particular domain.
The systematic literature review shows that together: user-centred design (Norman, 2002), the Cognitive Dimensions of Notations (Green & Petre, 1996) and controlled user studies (Dumas & Redish, 1999; Health & Human Services, n.d.) are the most accepted method for designing usable APIs. In this process, user-centred design is used as the overarching API design methodology, which typically involves many iterations of design and evaluation to develop a solution suitable for the target user group. The Cognitive Dimensions of Notations (Green & Petre, 1996) is used during the design stage of each iteration to weigh up the efficacy of possible design choices (Clarke, 2004). Then in the evaluation stage of each iteration, controlled user studies are employed to evaluate the effect the design decisions have on API usability (Clarke, 2004).

The Cognitive Dimensions of Notations is often used as a framework for analysing the data gathered during the user study (J. Beaton et al., 2008; J. K. Beaton et al., 2008; Clarke, 2004; Scheller & Kuhn, 2012; Stylos et al., 2008b). The CD Questionnaire (Blackwell & Green, 2000) is a promising post-task questionnaire to use during a user study. It has been used as a questionnaire to evaluate a programming language (Clarke, 2001). It is promising because it allows the researcher to get API usability data through the lens of the Cognitive Dimensions straight from the user of a product.

Less common evaluation techniques could also be useful at various stages of the user-centred design process, including case studies, expert evaluations, cognitive walkthroughs and concept mapping.

Case studies can be used in a similar manner to controlled user studies during the user-centred design process. Case studies provide less control of variables than a controlled user study, however, they can have higher ecological validity because participants can spend more time with the API.

The Cognitive Dimensions of Notations expert evaluation technique is most appropriate when used early as a part of the design stage of the user-centred design process (Dagit et al., 2006). Expert evaluation techniques should not be the only evaluation method used in the user-centred design process because they have been shown to be unreliable. For instance, Lauesen (2004, pp. 19–20) suggests that in the user interface design domain expert evaluations generally miss 50% of the major problems users face. Additionally, 50% of the problems found don’t affect real users; fixing them, therefore, wastes resources.
Chapter 3 – Related work

3.2 Robot social-interaction programming tools

This section reviews tools used to program socially interactive robots. It starts by providing an overview of socially interactive robots (Section 3.2.1) and is followed by an overview of the abstraction levels that can be used to program them (Section 3.2.2). The approach used for the literature review is then discussed (Section 3.2.3) and the results presented (Section 3.2.4). It finishes with a discussion about the implications of these tools for this research (Section 3.2.5).

3.2.1 Socially interactive robots

Fong et al. (2003, p. 145) defined what it means for a machine to be a socially interactive robot, specifically, it is an embodied machine that is able to interact with people using human social abilities. Human social abilities are wide ranging and include the ability to: express and perceive emotion; communicate with high-level behaviour, such as turn taking (Breazeal, 2003), dialogue or joint attention (Moore & Dunham, 1995); understand and model human individuality; learn social abilities; display personality; and build relationships with people.

Notable examples of social robots include Keepon (Breazeal & Scassellati, 1999), Kismet (1999) and the Geminoid series of robots (Dougherty & Scharfe, 2011; Nishio et al., 2007). Examples are illustrated in Figure 3.10. These three robots exhibit a range of the social abilities described at the beginning of this section and are described in more detail below.

Keepon was developed by Kozima et al. (2009) as a companion for interacting socially with children (Figure 3.10a). It displays two high-level social behaviours, joint attention
and expression of emotion. These behaviours are enabled through lower level social abilities, including gazing by orienting its head at objects and gesturing by rocking or bobbing its body up and down. Keepon has been used for experiments that explore social development of children and has been sold as a mass manufactured children’s toy (beatbots, n.d.).

Kismet was created by Breazeal & Scassellati (1999) to explore the application of infant behaviour to robots (Figure 3.10b). Kismet exhibits several high-level social behaviours, including: showing emotion via facial expressions; searching for things it wants, for example a toy or a person’s face (which it can perceive); socially protecting itself by looking away from over stimulating perceptions; and regulating its responses, e.g. reinforcing a person’s behaviour with appropriate facial expressions.

Geminoids are a series of life-like robot replicas of real people (Dougherty & Scharfe, 2011; Nishio et al., 2007). An example Geminoid is illustrated in Figure 3.10c. Whilst teleoperated, Geminoids are able to convey emotions via their life-like faces and communicate with speech-based dialogue with another person. The Geminoids were developed to study how life-like appearance and behaviour affects human-robot interaction.

3.2.2 Social-interaction abstraction levels

In this chapter, I analyse robot social-interaction programming tools through the lens of social-interaction abstraction levels. To create this lens I decomposed robot social abilities into five abstraction levels: hardware primitives, algorithm primitives, social primitives, emergent primitives and primitive control methods. These five abstraction levels are illustrated in Figure 3.11, from the bottom up and explained and supported in the following paragraphs.
The first and lowest abstraction level consists of *hardware primitives*. These primitives allow programmers to control and retrieve data from hardware devices, for example, interfaces to output sound, control actuators and control LEDs; as well as interfaces to retrieve data from cameras, laser scanners and microphones. These abstractions are commonly found in robotics middleware’s hardware abstraction layers, for example, Player (Gerkey, Vaughan, & Howard, 2003) and ROS (Open Source Robotics Foundation, n.d.-a).

The second abstraction level consists of *algorithm primitives*. These primitives are abstractions for algorithms that give robots the capabilities to interact socially. *Hardware primitives* are used as inputs and outputs for *algorithm primitives*. Examples of these primitives include face tracking, speech recognition, sound source localisation, speech synthesis, inverse kinematics and keyframe animation. The platform for the Nao humanoid robot, NAOqi (Aldebaran Robotics, n.d.) provides many primitives at this abstraction level, including interfaces to configure and retrieve data from each algorithm.
Chapter 3 – Related work

The third level of abstraction consists of social primitives. These primitives are reusable atomic units of social-interaction, generally implemented with algorithm primitives and in some cases hardware primitives. For example, the ability to gaze at a person’s face (a social primitive) is a combination of face tracking, inverse kinematics (algorithm primitives) and servo control (hardware primitive). These differ from the algorithm and hardware primitives because they are designed to be domain-specific for the social-interaction domain.

The fourth level of abstraction consists of emergent primitives. This level consists of higher level functions that emerge when social primitives are combined together, for example, when gaze, gestures and facial expressions are encoded into a higher level function that makes the robot display emotion. Kopp et al. (2006) argue for the separation between reusable social skills and the creation of domain-specific content, e.g. content customised for healthcare. The social primitives are at the reusable social skills end of the spectrum, whilst emergent primitives are further toward the domain-specific content end of the spectrum. The emergent primitives are less reusable than social primitives because they begin to encode domain-specific content.

The last abstraction level consists of primitive control methods, which are used to control primitives and create higher level behaviour, regardless of the primitive abstraction level. For example, a finite state machine could be used as the mechanism to create an emotion displaying emergent primitive by combining social primitives for gaze, gestures and expressions. The same emergent primitive could also be created by using algorithm primitives instead of social primitives. This is similar to the distinction that Siepmann (2013) makes between his control layer (e.g. finite state machines) and reusable behaviour modules (e.g. follow a person) for programming behaviour of mobile robots.

3.2.3 Approach

This review focusses on analysing robot programming tools through the lens of the social-interaction abstraction levels defined earlier and is guided by four research questions:

RQ1 What notable programming tools have been designed to program socially interactive robots?
RQ2  What notable programming tools have been designed to control general robot behaviour?

RQ3  What abstraction levels do these programming tools provide?

RQ4  Have any evaluations of the tools been performed? If so, what have these evaluations revealed about the usability of the provided abstraction levels?

The first research question (RQ1) is concerned with finding programming tools specifically designed to program socially interactive robots. Robot social-interaction programming tools typically provide the infrastructure that enables a robot to be programmed to interact socially with people.

The second research question (RQ2) is concerned with finding programming tools designed to control robot behaviour, which provide a means for orchestrating primitives into higher level robot behaviour (including, but not limited to social-interaction). Both types of programming tool come in the forms of APIs, visual languages and domain-specific languages.

Notable programming tools are defined as significant or distinctive tools that are able to inform the research question. They were selected for analysis based on theoretical sampling, where cases are selected based on their ability to inform the problem (Glaser & Strauss, 1967). The theoretical sampling approach proposes the researcher’s informed intuition as a method of sampling.

Research question one and two (RQ1 & RQ2) exclude most robot middleware solutions from the review, because they are often not specifically designed for programming socially interactive robots or controlling robot behaviour. Robot-middleware solutions are designed to be the backbone of a robotic system, providing sensor and actuator drivers, hardware abstraction, networking and algorithms such as mapping, inverse kinematics and path planning. They often do not explicitly include social-interaction specific algorithms which are needed by socially interactive robots. Examples include: ROS (Quigley et al., 2009), OPRoS (Jang et al., 2010), OROCOS (Bruyninckx, 2001), OpenRTM (Ando, Suehiro, & Kotoku, 2008) and Player (T. H. Collett, MacDonald, & Gerkey, 2005; Gerkey et al., 2003).

The third research question (RQ3) is concerned with analysing the abstraction levels each programming tool provides and the trade-offs resulting from these abstraction levels.
The abstraction levels include hardware primitives, algorithm primitives, social primitives, emergent primitives and primitive control methods.

The fourth research question for this literature review (RQ4) aims to discover if the designers of these tools have evaluated them, and in particular if their evaluations have revealed anything about the usability of the abstraction levels found in their tools.

Table 3.1. Robot social-interaction & behaviour control programming tools.

<table>
<thead>
<tr>
<th>Category</th>
<th>Tool</th>
<th>Target audience</th>
<th>Notation</th>
<th>Abstraction levels</th>
<th>Primitive control methods</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Social-interaction programming tools</td>
<td>Choregraphe</td>
<td>Novice</td>
<td>Visual</td>
<td>✓</td>
<td>✓</td>
<td>(Pot et al., 2009)</td>
</tr>
<tr>
<td></td>
<td>Interaction Blocks</td>
<td>Novice</td>
<td>Visual</td>
<td></td>
<td></td>
<td>(Sauppé &amp; Mutlu, 2014)</td>
</tr>
<tr>
<td></td>
<td>Interaction Composer</td>
<td>Novice &amp; professional</td>
<td>Visual</td>
<td>✓</td>
<td>✓</td>
<td>(Glas, Satake, Kanda, &amp; Hagita, 2012)</td>
</tr>
<tr>
<td></td>
<td>TiViPE &amp; its textual robotics command language</td>
<td>Novice &amp; professional</td>
<td>Textual &amp; visual</td>
<td>✓</td>
<td>✓</td>
<td>(Lourens &amp; Barakova, 2011)</td>
</tr>
<tr>
<td></td>
<td>AIML</td>
<td>Professional</td>
<td>Textual</td>
<td></td>
<td></td>
<td>(Wallace, 2003)</td>
</tr>
<tr>
<td></td>
<td>BML</td>
<td>Professional</td>
<td>Textual</td>
<td>✓</td>
<td>✓</td>
<td>(Kopp et al., 2006)</td>
</tr>
<tr>
<td></td>
<td>BONSAI</td>
<td>Professional</td>
<td>Textual</td>
<td>✓</td>
<td>✓</td>
<td>(Siepmann, 2013)</td>
</tr>
<tr>
<td></td>
<td>Robot Behavior Toolkit</td>
<td>Professional</td>
<td>Textual</td>
<td>✓</td>
<td>✓</td>
<td>(Huang &amp; Mutlu, 2012)</td>
</tr>
<tr>
<td>General robot programming tools</td>
<td>Gostai studio</td>
<td>Professional</td>
<td>Visual</td>
<td></td>
<td></td>
<td>(Gostai, n.d.)</td>
</tr>
<tr>
<td></td>
<td>SkillGUI</td>
<td>Professional</td>
<td>Visual</td>
<td></td>
<td></td>
<td>(Niemüller, Ferrein, &amp; Lakemeyer, 2010)</td>
</tr>
<tr>
<td></td>
<td>SMACH</td>
<td>Professional</td>
<td>Visual</td>
<td></td>
<td></td>
<td>(Bohren &amp; Cousins, 2010)</td>
</tr>
<tr>
<td></td>
<td>XABSL editor</td>
<td>Professional</td>
<td>Visual</td>
<td></td>
<td></td>
<td>(Loetzsch, Risler, &amp; Jungel, 2006; Lötzsch, Bach, Burkhard, &amp; Jungel, 2004; Risler, 2010)</td>
</tr>
</tbody>
</table>

3.2.4 Results

Twelve notable robot programming tools were found which can be divided into two main categories: social-interaction programming tools and general robot programming tools. The tools and their abstraction levels are summarised in Table 3.1, an unabridged copy of the table with supporting source data is available in Appendix A.1. The rest of this section discusses the abstraction levels found in these tools.
Social-interaction programming tools

The tools discussed in this section have all been designed or used for programming robot social-interaction.

Choregraphe is an end-user visual programming environment for programming Nao humanoid robots (Pot et al., 2009), illustrated in Figure 3.12. It allows end users to program social-interaction at all three abstraction levels: algorithms & actuation, social primitives and primitive control (Table 3.1). Users create social-interaction by combining visual blocks together.

![Figure 3.12. Choregraphe.](image)

Most of Choregraphe’s (Pot et al., 2009) visual blocks are represented at both the hardware primitive and algorithm primitive abstraction levels. For example, it has visual representations for LED control, sonar sensors, a speech recognition algorithm and a face tracking algorithm. This analysis is even supported by Pot et al. (2009, p. 50) who state that “technically Choregraphe is just a graphical representation of NAOqi’s functions...” NAOqi is the software framework that supports Choregraphe, most if its functions are hardware primitives and algorithm primitives. Some of Choregraphe’s visual blocks are represented at the social primitive level, for example, users can make Nao speak, stand up and sit down. Users control visual blocks, forming them into social-interaction using two visual programming views: a timeline and a data flow diagram.
Chapter 3 – Related work

(\textit{primitive control methods}). The timeline is used to organise visual blocks in a linear fashion and to create animation content with a keyframe editor. The data flow diagram is used to create non-linear social-interaction, e.g., one can make Nao respond verbally to a word detected by its speech recogniser.

Interaction Blocks is a visual end user social-interaction programming tool designed to enable designers to prototype social-interaction scenarios (Sauppé & Mutlu, 2014), illustrated in Figure 3.13. Interaction Blocks provides visual blocks with which the user authors social-interaction, similar to Choregraphe (Pot et al., 2009) and Interaction Composer (Glas, Satake, Kanda, & Hagita, 2012). The main difference is that all of the visual blocks are presented at an \textit{emergent primitive} abstraction level, including the ability to define: an introductory monologue, a question-answer exchange, a comment-exchange, a monologue-comment pattern, an instruction-action pattern, a closing comment pattern and a wait pattern. Interaction Blocks provides a linear timeline to control the visual blocks (\textit{primitive control methods}).

The results of a System Usability Scale (Sauro, 2011) suggest that the Interaction Blocks environment and notation as a whole are usable (Sauppé & Mutlu, 2014). The interview data revealed that: some participants were confused about the differences between some of Interaction Blocks’ emergent primitive’s, e.g. the difference between the \textit{monologue-comment} primitive and the individual \textit{monologue} and \textit{comment} primitives (p. 1445); and that participants desired more primitives, e.g. to represent utterances that have the same underlying meaning (p. 1446).
Figure 3.13. Interaction blocks.

Interaction Composer, illustrated in Figure 3.14, is a programming environment supporting collaboration between programmers and end users to create social-interaction scenarios (Glas, Satake, Kanda, & Hagita, 2012). Programmers perform low-level tasks, such as face recognition; while interaction designers (akin to end-users) use Interaction Composer, a visual programming environment, to create the higher level dialogue and interaction sequences, such as a greeting scenario. The programmer developed modules are represented as visual blocks which the scenario designers use in the visual programming environment. Interaction Composer’s visual blocks cross three of the primitive abstraction levels: *algorithms primitives, social primitives* and *emergent primitives* (Table 3.1). For example, there are visual blocks for: directly querying the results of face detection and speech recognition algorithms (*algorithm primitives*); programming with social primitives such as speaking and gesturing (*social primitives*); and performing emergent behaviours such as asking a question and listening for a response (*emergent primitives*). End users control the visual blocks with a visual representation of imperative programming (*primitive control methods*).

An evaluation of Interaction Composer (Glas et al., 2012) focused on comparing the usability of its textual and visual variants. The abstractions provided by Interaction Composer were not explicitly evaluated, but the authors did reveal that some participants
misunderstood the meaning of primitives (e.g. LookForFace behaviour and faceDetected variable).

Lourens & Barakova (2011) and Barakova, Gillesen, Huskens, & Lourens (2013) present a ‘textual robotics command language’ for programming a Nao robot paired with the visual programming environment TiViPE (Lourens, 2004), illustrated in Figure 3.15. Lourens & Barakova use the textual robotics language and TiViPE to create a social scenario of a robot shaking a child’s hand. Most of the robotics command languages functions are at hardware primitive and algorithm primitive abstraction levels, for example control of the robot’s LEDs, audio and joint control. Some of the commands are represented at the social primitive level, for example, the say command allows control of what the robot says as well as the volume and language it says it at. As with Interaction Composer (Glas et al., 2012), Lourens et al. argue that programmers and scenario designers should collaboratively create social robot interaction: the scenario designer decides what behavioural “blocks” are needed and the programmer creates them using the API. The scenario designer then composes the behavioural blocks in TiViPE.

**Figure 3.14. Interaction composer.** Used with permission of Dylan F. Glas (Glas et al., 2012).
Programmers can orchestrate commands so that they run sequentially or in parallel with the textual robotics command language and end users can combine behaviours produced by the programmers together with the TiViPE visual editor (*primitive control methods*).

Barakova et al. (2013) evaluated the textual robotics command language and TiViPE with two user studies. The focus of the studies was on evaluating whether therapists and developers could together create training scenarios with the tool. The authors do not report data relating to the abstraction levels of its primitives.

```plaintext
flush()
wait (duration) ledset (name, value)
ledset (name, value)
ledto (name, value, duration)
say (text[, volume[, language[, personality]]])
play (filename)
move (name, angle, duration[, stiffness])
movem (name, [angle, duration,]+ ...)
stiff ([stiffness, duration, idle,]+ ...)
walk (distance, duration)
walks (distance, duration)
walkto (x, y, theta)
walkd (x, y, theta, frequency, duration)
```

**Figure 3.15.** (a) textual robotics command language, (b) TiViPE visual editor. Used with permission of Emilia Barakova (Barakova et al., 2013).

AIML is an XML-based markup language for authoring the content of natural language dialogue systems (Wallace, 2003). It is the technology behind the popular ALICE chatbot (“A. L. I. C. E. The Artificial Linguistic Internet Computer Entity,” n.d.). An example is illustrated in Figure 3.16. AIML has been applied as the dialogue engine behind physical robots including Valerie (Gockley et al., 2005) and Pearl (Torrey, Powers, Marge, Fussell, & Kiesler, 2006). The most important of AIMLs *primitive control method* abstractions are illustrated in Figure 3.16. They include category, pattern and template. The category abstraction is a unit of knowledge, it contains pattern and template elements. Text from a user input device is compared with the pattern elements inside each category. If the user input matches a pattern the chatbot says what is in the template element. For example, in Figure 3.16, if the user types “Who are you?” the chatbot will say, “I am a chatbot”. AIML has many more features and its programs are typically orders of magnitude more complex than the example in Figure 3.16 (Wallace, 2003).
Behaviour Markup Language (BML) is a textual domain-specific language for specifying the actions of Embodied Conversational Agents (Kopp et al., 2006), an example of BML code is shown in Figure 3.17. BML allows the definition of social-interaction by providing XML interfaces at the social primitive abstraction level for controlling speech, gesture, gaze and posture. As well as providing social primitives, BML has an XML-based control-primitives for synchronisation and event transitions. Whilst BML was designed for virtual characters, two BML realisers have been created for robotic systems (Holroyd & Rich, 2012; Le & Pelachaud, 2012).

BONSAI is a software framework designed to help robot software developer’s program household interactive robots (Siepmann, 2013), where much of their interaction is social. BONSAI has primitives at the hardware primitive and algorithm primitive abstraction levels, e.g. the laser sensor and navigation actuator interfaces. The social-interaction functionalities of BONSAI are represented at the social primitive abstraction level, for example, BONSAI has high-level person, object and speech interfaces. Programmers implement instances of these interfaces for particular robots in Java. A primitive control method abstraction is provided by using State Chart XML (SCXML), a generic XML-based finite state machine language and engine (W3C Consortium, n.d.). BONSAI uses SCXML as the primitive control abstraction to implement scenarios such as guiding a person around a house.
Siepmann (2013) performed a number of different evaluations of BONSAI, one of which focuses on evaluating its usability (p. 82). In this study, participants performed programming tasks and completed post-task questionnaires. One of the questions aimed to assess how well users perceived BONSAIs abstraction level: “Has Bonsai provided functionality in a meaningful way?” From the answers to this question, the authors concluded that BONSAIs abstractions were well chosen.

Like BONSAI (Siepmann, 2013), the Robot Behavior Toolkit (Huang & Mutlu, 2012) is a software framework designed to help robot programmer’s program robot interaction, the key difference being that it focuses solely on social-interaction. An example is illustrated in Figure 3.18. Social primitives are provided that enable the robot to speak and gaze at objects. Additionally, two primitive control methods are provided. The first is an XML-based markup language that enables the programmer to order speech and gaze behaviours sequentially and in parallel to each other (p. 28). The second is an XML-based markup language that is used to specify details for a cognitive system, which makes decisions about how to execute behaviour (pp. 27-28).

```xml
<behaviors>
  <channel type='gaze'>
    <action endTime='214.5' startTime='0' target='unspecified'/>
    <action endTime='1160' startTime='214.5' target='the green object with one peg'/>
    <action endTime='2735.4' startTime='1160' target='unspecified'/>
    <action endTime='3597' startTime='2735.4' target='the red box'/>
    <action endTime='4308' startTime='3597' target='unspecified'/>
    <action endTime='4963' startTime='4308' target='listener'/>
  </channel>
  <channel type='speech'>
    Could you help me put the green object with one peg into the red box, please?
  </channel>
</behaviors>
```

**Figure 3.18.** Robot behaviour toolkit example. Used with permission of Chien-Ming Huang and Bilge Mutlu (Huang & Mutlu, 2012).

**General robot programming tools**

The tools discussed in this section have been designed for controlling robot behaviour, and could be used to create robot social-interaction with primitives of any abstraction level. They provide a variety of notations for programming finite state machines, which are a common means of creating robot social-interaction (primitive control methods). For
example, the robots Minerva (Schulte et al., 1999), Sage (Nourbakhsh et al., 1999) and Fritz (Bennewitz et al., 2005) all use finite state machines to control their social-interactions.

Gostai Studio (Gostai robotics, 2010; Gostai, n.d.) allows users to visually author hierarchical finite state machines and visually debug the state machine while it is executing. Using a visual editor, users create states (circles) and draw transitions between them (arrows). Each state and transition have an associated script window that pops up when selected. Users write Urbiscript (Baillie, 2008) within this window, which is executed when its corresponding state or transition is fired.

SkillGUI (Niemüller et al., 2010) is a Lua based behaviour scripting engine. It was designed for programming robot soccer with the Nao humanoid robot (Aldebaran - SoftBank Group, n.d.). It contains a Lua state machine implementation and a visualiser (Figure 3.20). Users define states (lines 8-10) and create transitions from one state to another (lines 12-15). Users program state logic by overriding the init method of a state, (lines 17-20), which makes Nao stand up.
SMACH (Bohren & Cousins, 2010) is the de facto method of authoring robot behaviour in ROS (Open Source Robotics Foundation, n.d.-a). SMACH consists of a Python API for programming hierarchical finite state machines and a GUI for visualising the state machine (Figure 3.21). SMACH states are defined by creating Python classes and inheriting the class `smach.State`. The state logic is written within the classes overridden `execute` method. To transition to a new state, an “outcome” is returned from the `execute` method. Transitions are defined by mapping outcomes to a state id (a string), which indicates the state the state machine will transition to when that outcome is returned.

```python
1  fsm = SkillHSM.new(name=name, start="STANDUP")
2  depends_skills = ["servo", "getup"]
3  depends_interfaces = {
4  \ {v : "naomotion", type : "HumanoidMotionInterface"},
5  \ {v : "naohw", type : "NaoHardwareInterface"}
6  }
7  
8  fsm:new_jump_state("STANDUP")
9  fsm:new_jump_state("FROm_BACK")
10  fsm:new_jump_state("GETUP", getup, FINAL, FAILED)
11  
12  STANDUP: add_transition(GETUP,
13  \ "naohw:acc_x() >= -35 and naohw:acc_x() <= 35")
14  
15  STANDUP: add_transition(FROM_BACK, "naohw:acc_x() < -35")
16  
17  function FROM_BACK:init()
18  naomotion:msg_queue_copy{
19  \ naomotion.StandupMessage:new(naomotion.STANDUP_BACK)
20  end
```

Figure 3.20. (a) SkillGUI Lua state machine definition, (b) state machine visualisation. I have tried to get permission to use these images but do not have a reply yet.
```python
# define state Foo
class Foo(smach.State):
    def __init__(self):
        smach.State.__init__(self,
            outcomes=['outcome1','outcome2'])
        self.counter = 0
    def execute(self, userdata):
        rospy.loginfo('Executing state FOO')
        if self.counter < 3:
            self.counter += 1
            return 'outcome1'
        else:
            return 'outcome2'

# define state Bar
class Bar(smach.State):
    def __init__(self):
        smach.State.__init__(self,
            outcomes=['outcome2'])
    def execute(self, userdata):
        rospy.loginfo('Executing state BAR')
        return 'outcome2'

# main
def main():
    rospy.init_node('smach_example_state_machine')
    # Create a SMACH state machine
    sm = smach.StateMachine( outcomes=['outcome4', 'outcome5'] )
    # Open the container
    with sm:
        # Add states to the container
        smach.StateMachine.add('FOO', Foo(),
            transitions={'outcome1':'BAR',
                        'outcome2':'outcome4'})
        smach.StateMachine.add('BAR', Bar(),
            transitions={'outcome2':'FOO'})
    # Execute SMACH plan
    outcome = sm.execute()
    if __name__ == '__main__':
        main()
```

Figure 3.21. (a) SMACH imperative state machine definition and (b) state machine visualisation.

**Image sources:** (Open Source Robotics Foundation, n.d.-b).

XABSL (Risler, 2010) consists of a number of notations for editing and visualising hierarchical finite state machines (Figure 3.22). Like SkillGUI, it was primarily designed to create robot soccer applications, but it could also be used to create many other applications. The primary means of authoring state machines in XABSL is via the custom XABSL programming language, which has language primitives for programming states, transitions (goto), decision trees, inputs, outputs, behaviours and actions. The state machine can be visualised, but not visually edited (Figure 3.22).
3.2.5 Implications for research

The robot programming tools discussed in this chapter have a number of implications for the first two research questions of this thesis:

RQ1  What abstraction level is appropriate for programming social robot applications?

RQ1.1  What are the trade-offs associated with using different abstraction levels to program robot social-interaction?

RQ2  How should primitives, regardless of their abstraction level, be orchestrated to create higher level social-interaction such as dialogue and non-verbal behaviour?

The evaluations of existing social-interaction programming tools do not provide enough data to answer RQ1. Four of the eight social-interaction programming tools were evaluated with user studies, including TiViPE and its robotics command language (Barakova et al., 2013), Interaction Blocks (Sauppé & Mutlu, 2014), Interaction
Composer (Glas et al., 2012) and BONSAI (Siepmann, 2013). The evaluation of TiViPE and its robotics command language didn’t report data related to abstraction level usability. The evaluation of Interaction Blocks focused on evaluating its overall usability, whilst the evaluation of Interaction Composer focused on comparing the usability of its textual and visual variants. Both studies briefly mention factors related to the usability of abstraction levels, however, the authors make no conclusions based on this data. Siepmann’s (2013) evaluation of BONSAI was the only study to explicitly examine the appropriateness of its abstraction level. From the answers to one question¹, Siepmann concludes that BONSAI has an appropriate abstraction level. Even if we assume this is true, BONSAI has primitives across a number of abstraction levels (hardware, algorithm and social primitives), leaving RQ1 unanswered. More importantly, none of the evaluations explore the trade-offs caused by using different abstraction levels to program social-interaction (RQ1.1).

Whilst the evaluations of existing social-interaction programming tools provide little empirical data about the usability of their abstraction levels, the Cognitive Dimensions of Notations framework (Green & Petre, 1996) can be used to discuss possible usability trade-offs associated with using different abstraction levels to program robot social-interaction.

BONSAI (Siepmann, 2013), Choregraphe (Pot et al., 2009), Interaction Composer (Glas et al., 2012) and TiViPE (Lourens & Barakova, 2011) have a large proportion of lower level hardware primitives and or algorithm primitives. The Cognitive Dimensions (Green & Petre, 1996) indicate that these tools may have a number of problems when they are used to program social-interaction scenarios, including a high viscosity, hard mental operations and a low closeness-of-mapping. First, high viscosity is symptomatic of a system with many low-level primitives because programmers have to manipulate many lines of code to accomplish a task (pp. 38-39). Second, programmers effectively have to create their own higher level social primitives out of the lower level hardware primitives and algorithm primitives, this causes hard mental operations (pp. 38-39).

¹ “Has Bonsai provided functionality in a meaningful way?” (Siepmann, 2013)
Lastly, the *hardware primitives* and *algorithm primitives* have a distant mapping to the problem domain of social-interaction.

BML (Kopp et al., 2006) and the Robot Behavior Toolkit (Huang & Mutlu, 2012) provide primitives with a closer mapping to the social-interaction domain (*social primitives*). BML provides a comprehensive model of the actions a social agent can perform, the Robot Behaviour Toolkit provides fewer primitives than BML. They both lack primitives for social perception, such as symbolically representing people. Additionally, BML is an XML message definition, it is not an API that can be used to program social-interaction out of the box. Green & Petre (1996) argue that giving a notation a close mapping to a problem domain should make it easier for users to solve problems in that domain.

Interaction Blocks (Sauppé & Mutlu, 2014) and Interaction Composer (Glas et al., 2012) both contain *emergent primitives*, which retain a somewhat close mapping to the social-interaction domain, but have a higher abstraction level than *social primitives*. Emergent primitives may work well for interaction designers and people wanting to author the content of chat systems, however, professional programmers may need more flexibility. Raising the abstraction level too high can cause hidden dependencies (Green & Petre, 1996, p. 38).

Gostai Studio (Gostai, n.d.), SkillGUI (Niemüller et al., 2010), SMACH (Bohren & Cousins, 2010) and XABSL (Risler, 2010) demonstrate popular mechanisms for orchestrating primitives (*primitive control methods*) into higher level social-interaction (RQ2). All of these tools use finite state machines to create higher level robot behaviour. These tools have a variety of notations for editing finite state machines. Gostai Studio is the only one that allows visual editing of finite state machines. The remaining three enable programmers to edit state machines via textual notations. SkillGUI and SMACH both provide state machine APIs, SkillGUI provides a Lua based API and SMACH a Python-based API. I refer to these as imperative state machines because they are defined by the programmer with an imperative programming language, rather than a declarative language such as XML. XABSL is the only tool that provides a custom textual programming language for defining state machines. All three provide un-editable state machine visualisations. The implications of this are that the usability of different state machine notations should be evaluated to help answer RQ2. This would provide robot
social-interaction programming tool designers with information that helps them design tools for orchestrating primitives into higher level social-interaction.

### 3.3 Summary

This chapter reviewed two research areas: API usability and robot social-interaction programming tools.

The review of API usability started with an overview of the API design space and then systematically analysed the literature for general guidelines and techniques used to design and evaluate the usability of APIs. This section finished with the implications of these findings for this thesis and other related research.

The review of robot social-interaction programming tools started with an overview of social-interaction, paying particular attention to different abstraction levels used to create software for socially interactive robots. This is followed by an analysis of tools used to program social-interaction and finished with the implications of these tools for this thesis.

The next chapter presents the iterative user-centred design methodology used to design the APIs this thesis used to explore the remainder of the research questions.
This chapter describes my approach to exploring how to improve the usability of tools used to program socially interactive robots. It is an iterative research methodology inspired by design science, a method concerned with discovering how the design and creation of artefacts influences people and their environment (Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007). A high-level overview of the methodology is illustrated in Figure 4.1.

With this methodology the researcher (1) first formulates questions about a problem, e.g. research questions and then (2) designs an artefact as a solution to the problem (Peffers et al., 2007). This is not just engineering, the researcher uses his or her creativity to create a novel solution to a problem (Schön, 1987). The researcher then (3) evaluates the artefact they designed to understand how it addresses the problem, e.g. with a usability study. The researcher (4) synthesises the new knowledge from the evaluation with what
is already known and then uses this knowledge to reformulate the questions and conduct further investigation.

This thesis addresses the following four research questions using this overarching methodology. Each research question is addressed employing specific methodologies, which are discussed in this chapter.

RQ1 What abstraction level is appropriate for programming social robot applications?
   RQ1.1 What are the trade-offs associated with using different abstraction levels to program robot social-interaction?

RQ2 How should primitives, regardless of their abstraction level, be orchestrated to create higher level social-interaction such as dialogue and non-verbal behaviour?

RQ3 What primitives are needed to program robot social-interaction at the appropriate abstraction level?

RQ4 What is a suitable software architecture to support the API with primitives at the abstraction level appropriate for programming robot social-interaction?

The first and second research questions were addressed through two iterations of API design, evaluation and synthesis (Chapters 5 & 6). User-centred design (Norman, 2002) was used throughout this process, which involved using: the Cognitive Dimensions (Green & Petre, 1996) as a design tool, user studies (Dumas & Redish, 1999; Health & Human Services, n.d.) & case studies (Yin, 2013) as evaluation methods and the Cognitive Dimensions as an analysis framework. These are described in detail in Sections 4.1 and 4.2.

The third research question was addressed by deriving a taxonomy of primitives at an abstraction level appropriate for programming social-interaction. The taxonomy was iteratively derived throughout the iterations of API design, implementation and evaluation using a methodology inspired by grounded theory (Glaser & Strauss, 1967). This is described in detail in Section 4.3.

The fourth research question was answered by defining a set of requirements for a social-interaction API software architecture and evolving the architecture as the two
iterations of API development progressed and more knowledge about the effect of the designs were gained (Chapter 8). This is described in detail in Section 4.4.

4.1 Iteration 1: Social-interaction for Nao

The first API iteration explored research questions one and two. To explore research question one, an API with high-level, domain-specific primitives was designed for programming the social-interaction of a Nao humanoid robot. I started by deriving a taxonomy of primitives based on researchers’ descriptions of robot behaviour, this is presented in (Diprose, Plimmer, MacDonald, & Hosking, 2012). The design of the API was based on parts of this taxonomy. The API provides domain-specific primitives for making speech, performing gestures, modelling human features and understanding human speech. The primitives are a range of abstraction levels, but mainly social primitives and emergent primitives.

To explore research question two, an imperative state machine API was designed and implemented to orchestrate the primitives into social-interaction. Evaluating the usability of state machine representations is important because they are a very common method of orchestrating social robot behaviour, including Minerva (Schulte et al., 1999), Sage (Nourbakhsh et al., 1999), Fritz (Bennewitz et al., 2005) and BIRON (Spexard et al., 2006).

The API was designed with a user-centred design methodology (Norman, 2002), using the Cognitive Dimensions (Green & Petre, 1996) as a design framework to weigh up the merit of various design choices. Additionally, in the spirit of user-centred design, an exemplar scenario was used to guide the design and development of the API: a multiplayer game show scenario. Game shows are commonly used scenarios to explore social robot interaction (Kruijff-Korbayová, Athanasopoulos, & al., 2011; Moubayed, 2013). In this scenario, a Nao robot hosts a quiz and two human players compete against each other by answering Nao’s questions. This game uses all of the primitives realised in the API.

A lab-based user evaluation (Dumas & Redish, 1999; Health & Human Services, n.d.) was then conducted to evaluate the APIs domain-specific primitives and the imperative state machine API. The user evaluation used programming tasks to familiarise participants with the API and researcher observation and a post-task CD Questionnaire
(Blackwell & Green, 2000) to gather data. The Cognitive Dimensions (Green & Petre, 1996) was used as a framework to analyse the data from the evaluation.

After the user evaluation, a stakeholder interview was conducted to see how well the API addressed potential user’s requirements.

The user evaluation and stakeholder interview led to a greater understanding of what abstraction levels work well for programming robot social-interaction and some of the trade-offs associated with their use. This led to the next study.

4.2 Iteration 2: Social-interaction refactored

The knowledge gained from the evaluation of the previous API was used in the next study (Chapters 6), which delved deeper into exploring the trade-offs associated with using different abstraction levels to program robot social-interaction (RQ1.1). The goal of the second iteration was to refactor the API to provide finer control of social-interaction, and improve and broaden the API so that it could be used to program other scenarios and robots.

The second API iteration was also designed with a user-centred design methodology (Norman, 2002). To provide finer control of social-interaction, based on feedback from the stakeholder interview, the APIs primitives were refactored to a slightly lower abstraction level. To improve and broaden the API support for two additional humanoid robots were added, the main difference being that they had animatronic faces. Additionally, a new scenario was used as an exemplar use case during the design and development of the API. The exemplar scenario for this API was to be able to program the social-interactions of a life-like humanoid robot that could communicate with people emotionally. Like the previous iteration, the Cognitive Dimensions (Green & Petre, 1996) was used as a design tool to weigh up the merit of different design choices.

The second API was evaluated with a case study (Yin, 2013) to gain in-depth results with ecological validity. Four fourth-year software engineering students, all taking an advanced human-computer interaction source, used the API for a period of two weeks. In this study, they developed a social-interaction application for a Nao humanoid robot. The application is a robot chat application, where a robot is used as an interface to send instant messages to another person. The person situated with the robot can speak to it; the robot hears what was said and relays it to a person using a chat program. The robot also converts
emoticons and special characters that it encounters into physical gestures, for example if the robot encounters a question mark “?”, it puts it hand in the air.

The students each wrote a report analysing the usability of the API based on the experience they had using it. They were also interviewed with a semi-structured interview based on the CD Questionnaire (Blackwell & Green, 2000). The data was analysed against the Cognitive Dimensions framework (Green & Petre, 1996).

### 4.3 A taxonomy of social primitives

To address research question three, throughout this thesis I derived a taxonomy of primitives for programming robot social-interaction. The purpose of the taxonomy is to provide a comprehensive, implementation independent model of the primitives required for programming socially interactive robots, at an appropriate abstraction level.

The taxonomy was derived using a methodology based on grounded theory (Glaser & Strauss, 1967). Grounded theory is widely used in social science research to discover theory from systematically collected data. The researcher builds up a theory by iteratively identifying concepts and relationships that fit the underlying data. Eventually, a conceptualisation of the data emerges.

The first iteration of the taxonomy was derived based on researchers’ descriptions of healthcare robot behaviour, presented in (Diprose et al., 2012). Despite the descriptions of robot behaviour coming from the healthcare domain, many of the robots were, in fact, social robots.

The taxonomy was then iteratively refined as the first and second social-interaction APIs were developed and evaluated (Chapters 5, & 6), bringing it closer to an appropriate abstraction level. Other data that contributed to this iterative refinement included existing robot programming interfaces (Glas et al., 2012; Kopp et al., 2006; Lourens & Barakova, 2011; Pot et al., 2009), robot social-interaction literature (Fong et al., 2003; Li et al., 2009) and general social-interaction literature (Doupe & Kuhl, 1999; Liebal & Pika, 2005; McNeill, 1992; Sowden et al., 2013).

The final taxonomy is presented in Chapter 7. It defines the high-level objects, robot actions & perceptions and the features of each action and perception required for programming social-interaction.
4.4 Software architecture

The fourth research question, which aimed to create a suitable software architecture to support an API with primitives at an appropriate abstraction level, was investigated through iterative requirements engineering (Cao & Ramesh, 2008, pp. 63–64) and development. Cao & Ramesh argue that this methodology is suitable when system requirements aren’t clear when development begins. The software architecture presented in this thesis evolved as the social-interaction APIs were developed and more knowledge about the effect of different abstraction levels was gathered. Objectives and design goals were defined as they became clear, the objectives include:

O1. Provide mechanisms that enable the API to be extended to support a variety of use cases.
O2. Provide mechanisms that allow the API to be vendor agnostic at the social primitive abstraction level, enabling integration with a variety of social robot platforms.

The first objective specifies that API users should be able to customise the API to their needs, for example, by defining new entities, robots and the relationships between them. The second objective specifies that the API should be able to interface with a variety of vendor platforms at an abstraction level that is reusable, but has a close mapping to the social-interaction domain; this was identified from the iterations of API design, implementation and evaluation. Design goals were defined based on these objectives as they became clear, these are discussed below.

The API extensibility design goals focus on providing an architecture that enables users to extend the API. First, users need to be able to define new entities for the robot to interact with. For example, if a user makes a “cat” perception algorithm, they should be able to represent the notion of a “cat” symbolically in the API and interface it with the perception algorithm. Second, users who add support for a new robot to the API need to be able to define a new robot along with body parts to represent it. Lastly, users need to be able to symbolically represent the gestures and expressions that animators and API users author with animation tools or other software. For example, an animator who creates a waving gesture animation needs to represent this symbolically in the API.
Chapter 4 – Methodology

The second set of design goals focus on enabling the API to interface with a variety of vendor platforms at a reusable abstraction level with a close mapping to social-interaction. First, the architecture needs a high-level, social action interface that enables the API to control robot actions from a variety of robot platforms. Second, the architecture needs a high-level perception interface that receives perception events from a variety of different robot platforms.

4.5 Summary

This chapter outlined the iterative research methodology (Peffers et al., 2007) that was used in conjunction with user-centred design (Norman, 2002) to answer the research questions of this thesis.

The first research question focuses on discovering what abstraction level is appropriate for programming social robot applications. The second research question examines how primitives, regardless of their abstraction level should be orchestrated to create higher level social-interaction. These were both explored by iteratively designing and evaluating two APIs for programming socially interactive robots. User-centred design was used to design and evaluate the APIs.

The third research question was addressed by iteratively deriving a taxonomy of the primitives required for programming social-interaction using a methodology inspired by grounded theory (Glaser & Strauss, 1967).

The fourth research question focused on creating a software architecture to support an API with primitives at an appropriate abstraction level. It was explored through iterative requirements engineering (Cao & Ramesh, 2008, pp. 63–64) and development.

The next chapter presents the first API and its evaluation, which is used to investigate how different primitive abstraction levels and imperative state machines affect usability (RQ1 & RQ2).
Iteration 1: Social-interaction for Nao

The previous chapter outlined the iterative design based methodology used to answer the research questions of this thesis. This chapter presents the first iteration of this approach; the exploration of research questions one and two through the design, implementation and evaluation of an API for programming the social-interaction of a Nao humanoid robot. An early version of this chapter was published here: (JP Diprose, Plimmer, MacDonald, & Hosking, 2014).

RQ1 What abstraction level is appropriate for programming social robot applications?

RQ1.1 What are the trade-offs associated with using different abstraction levels to program robot social-interaction?

RQ2 How should primitives, regardless of their abstraction level, be orchestrated to create higher level social-interaction such as dialogue and non-verbal behaviour?

To investigate RQ1 and RQ1.1, an API was designed and implemented with high-level, domain-specific primitives. High-level, domain-specific primitives were chosen based on the expert Cognitive Dimensions (Green & Petre, 1996) analysis from Chapter 3.2.5, which hypothesised trade-offs caused by different primitive abstraction levels in existing robot programming tools (social and emergent primitives). The API provides primitives for making speech, performing gestures, modelling human features and understanding human speech.

To explore RQ3, an imperative finite state machine API inspired by ROS’s SMACH (Bohren & Cousins, 2010) was designed, implemented and evaluated.

Software developers that need to program social-interaction are the target audience of the API. For example, a developer programming dialogue between a robot and a person. The developer's code defines the logic and content of the social-interaction: the social primitives are used to make the robot perform social actions and also understand communication directed at the robot by people, and the imperative finite state machine is
used by the developer to orchestrate the primitives into dialogue or some other form of interaction.

The API was designed with a user-centred design methodology (Norman, 2002). The Cognitive Dimensions (Green & Petre, 1996) was used as a tool to evaluate design choices and an exemplar scenario was used to guide the design and development of the API. The exemplar scenario was a multiplayer game show scenario, which are commonly used to explore social robot interaction (Kruijff-Korbayová et al., 2011; Moubayed, 2013). In our scenario, a Nao robot hosts a quiz and two human players compete against each other by answering Nao’s questions.

A user-centred approach (Norman, 2002) was also used to evaluate the API. A user study was conducted where participants performed programming tasks with the APIs and filled out a post-task CD Questionnaire (Blackwell & Green, 2000). The Cognitive Dimensions was used as a framework to analyse researchers’ observations and the answers to the CD Questionnaire.

The results of the usability study indicate that high-level, domain-specific primitives have positive effects on usability. Users had an overall positive impression of the API and also indicated specific areas where the APIs design benefited usability. These include good visibility & role expressiveness, a close mapping to the social-interaction domain, and an appropriate level of terseness.

Additionally, the study found a number of usability issues with imperative finite state machines for authoring robot dialogue. They cause hard mental operations, are very diffuse, error prone and involve premature commitment. This indicates that other options for orchestrating social primitives should be explored in the future.

Subsequent to the user evaluation, a stakeholder interview was conducted with potential users of the API. This is similar to the user-centred methodology used by Stylos et al. (2008b), who performed stakeholder interviews to gather data for developing a wrapper to SAP’s BRFplus. The stakeholder interview provided valuable insight in addition to the user evaluation. The interview suggested that primitives with a slightly lower abstraction level may be more appropriate due to increased reusability.

Section 5.1 presents the API through a series of example programs. Section 5.2 gives more details about the API and discusses the design decisions that were made in the context of the Cognitive Dimensions of Notations framework (Green & Petre, 1996). Section 5.3 describes the results of the evaluation. Section 5.4 describes the results of the
stakeholder interview. Finally, Section 5.5 discusses the implications of the results on the next iterations of research.

5.1 Example programs

Before describing each API module and their design decisions in depth, five example programs are provided. The first four illustrate how the social primitives are used and the fifth illustrates how to use an imperative finite state machine to create robot dialogue.

5.1.1 Object spatial relations

The first API example illustrates how to find the spatial relationships between the robot and all of the objects it currently perceives (Figure 5.1). For each object, the following are printed to the terminal: its name; its distance to the robot; and whether it is left, right, in front of or behind the robot.

```python
# Initialise the Environment
env = Environment()

# Loop through objects and print their spatial relations to the robot
for obj in env.objects:
    print(str(obj))
    print('distance: ' + str(obj.distance_to(env.robot)))
    print('left: ' + str(obj.left_of(env.robot)))
    print('right: ' + str(obj.right_of(env.robot)))
    print('infront: ' + str(obj.infront_of(env.robot)))
    print('behind: ' + str(obj.behind(env.robot)))
```

**Figure 5.1. Object spatial relations.**

Example output is illustrated in Figure 5.2 below for a scenario where two people are in front of the robot. One is to the left, 1.2 metres from the robot and the other is to the right, 2.0 metres from the robot.

```
person_1_505509720
distance: 1.2m
left: True
right: False
infront: True
behind: False

person_2_505509708
distance: 2.0m
left: False
right: True
```
infront: True
behind: False

Figure 5.2. Object spatial relations output.

The following paragraphs explain how the code works.

```python
2 env = Environment()

This line instantiates an Environment object, which contains a list of objects currently perceived by the robot (env.objects), as well as the robot instance itself (env.robot).

6 for obj in env.objects:

This line loops through all of the objects from the Environment’s objects list. For each object (obj) the following is printed:

7 print str(obj)

The string representation of the object.

8 print 'distance: ' + str(obj.distance_to(env.robot))

The distance between the object and the robot.

9 print 'left: ' + str(obj.left_of(env.robot))
10 print 'right: ' + str(obj.right_of(env.robot))
11 print 'infront: ' + str(obj.infront_of(env.robot))
12 print 'behind: ' + str(obj.behind(env.robot))

And whether the various spatial relations are true or false.

5.1.2 Speaking

The second example illustrates how to make the robot speak to a group of people (Figure 5.3). In this example, the robot looks at a random person from the group of people that he can see and says “Hello humans” to them.
# Initialise the Environment
env = Environment()
robot = env.robot

# Select all of the people from the Environment
people = query(env.objects).of_type(Person)
env.add_query(people)

# Speak to them
robot.say_to("Hello humans", people)

---

The following paragraphs explain how the code works.

This line creates a query that selects all of the Person objects from env.objects, the objects in the robot’s environment. The query is assigned to the variable people.

We add the query to the Environment so that the robot can interact with the people found from the query.

The last line makes the robot speak to the group of people by calling the say_to method from robot. The first parameter specifies what the robot will say (in this case “Hello humans”) and the second parameter specifies who the robot should direct its nonverbal speaking behaviour toward.

In this example, the robot starts by gazing at a randomly selected person from the people query. When it has made eye contact it will begin synthesizing the text "Hello humans". If multiple sentences are supplied to the first parameter and multiple people are returned by the query people, then whenever a new sentence is reached, the robot will change its gaze to a random person.
5.1.3 Speaking and gesturing

The third example demonstrates how to make a robot speak and gesture simultaneously (Figure 5.4). In this example, the robot starts by waving to a group of people and saying “Hello how are you?” Then, the robot’s eyes glow red while it says “I’m a bad robot”

```python
1 # Initialise the Environment
2 env = Environment()
3 robot = env.robot
4
5 # Select all of the people from the Environment
6 people = query(env.objects).of_type(Person)
7 env.add_query(people)
8
9 # Speak and gesture to them
10 robot.say_to("<wave> Hello how are </wave> you?", people)
11 robot.say_to("<red-eyes> I'm a bad robot </red-eyes>", people)
```

Figure 5.4. Speaking and gesturing.

The following paragraphs explain how the code works. Lines 1 – 7 are exactly the same as the previous example.

```python
10 robot.say_to("<wave> Hello how are </wave> you?", people)
```

This line makes the robot make a waving gesture whilst saying, “Hello how are you?” to the people found by the query people. Gestures are specified by marking up the text parameter with gesture tags. The placement of the opening and closing tags indicate where the gesture will begin and end relative to the robot’s synthesised speech. For example, in line 10 the robot will start waving at “Hello” and finish waving after “are.”

5.1.4 Listening for utterances

The fourth example shows how to listen for utterances (Figure 5.5). In this example imagine that the robot had asked you a yes/no question and now it is listening for a response. When the robot hears one of the words “yes”, “sure”, “ok” or “yeah” it responds “Ok I heard you” It could easily be expanded to ask a question and handle synonyms for “no”, but it is kept simple for demonstration purposes.
# Initialise the Environment
env = Environment()
robot = env.robot

# Select all of the people from the Environment
people_query = query(env.objects).of_type(Person)
env.add_query(people_query)

meanings = Enum('yes')
robot.associate_utterances_with_meaning(meanings.yes, ['yes', 'sure', 'ok', 'yeah'])
robot.generate_language_models()

while True:
    people = people_query.to_list()
    for person in people:
        if person.said_to(meanings.yes, robot):
            robot.say_to("Ok I heard you", person)
        break

Figure 5.5. Listening.

The following paragraphs explain how the code works. Lines 1 – 7 are largely the same as the previous two examples.

meanings = Enum('yes')
robot.associate_utterances_with_meaning(meanings.yes, ['yes', 'sure', 'ok', 'yeah'])

This section of code defines high-level meanings via a set of enumerations; in this case, there is only one high-level meaning “yes” for when the person gives an affirmative answer (line 9). We then associate a list of verbal utterances with that meaning; these are all words that could be taken to be mean the same thing - an affirmative answer in this case (line 10).

robot.generate_language_models()

This line generates language models that enable the robot to listen for the utterances we defined in the previous section of code.
This section of code is an infinite loop. It starts by finding all of the people that the robot can currently perceive by converting the `people_query` query into a list of `Person` instances (line 15). It then iterates through this list (line 17); for each person instance, we check if they said something that meant yes to the robot (line 18). If they did then the robot says “Ok I heard you” to that particular person (line 19) and breaks out of the inner loop (line 20).

5.1.5 Dialogue with a state machine

The final example shows how to create dialogue with a state machine (Figure 5.6). The interaction starts with the robot waiting to be greeted; when a person does finally greet the robot it replies by saying “Hello, nice to meet you!” The robot then goes back into the listening state, waiting for another greeting.
def create_transitions(self, next_state):
    self.next_state = next_state

listen = Listen()  # Instantiate states
respond = Respond()

listen.create_transitions(respond)  # Define transitions
respond.create_transitions(listen)

sm = StateMachine(env)
sm.add_state(listen, first=True)  # Add to StateMachine
sm.add_state(respond)
sm.start()  # Start state machine

Figure 5.6. Dialogue management.

The following paragraphs explain how the code works. Lines 1 – 7 are largely the same as the previous examples.

meanings = Enum('greet')
robot.associate_utterances_with_meaning(meanings.greet, ['hello', 'hi'])

This section of code defines the meaning ‘greet’ and utterances that will be interpreted as a greeting.

class Listen(State):
    def create_transitions(self, next_state):
        q = people.where(lambda p: p.said_to(
            meanings.greet, robot))
        event = QueryEvent(q)
        self.add_transition(event, next_state, id="greeted")

This section of code defines the state that listens for greetings; this is accomplished by defining a new class Listen and inheriting the State class (line 15). An event driven state transition is created in the method create_transitions (line 16-20), it consists of: a query which finds all of the people who greeted the robot (line 17-18); a QueryEvent which fires when the query finds 1 or more person that greeted the robot (line 19); and a call to add_transition which associates the QueryEvent with a transition to another state.
This section of code defines the state that responds when the robot hears a greeting; this is accomplished by defining a new class `Respond` and inheriting the `State` class (line 22). The `execute` method is overridden, which is where the robot action logic is specified. In this example, if the variable `e.id` passed to the state equals “greeted” the robot will say “Hello, nice to meet you!” The `execute` method returns a transition object which transitions to the state specified by `self.next_state`. This variable is set later on in the script by a function call to `create_transitions` (line 28).

```
listen = Listen()  # Instantiate states
respond = Respond()
```

This section instantiates the states that we just defined.

```
listen.create_transitions(respond)  # Define transitions
respond.create_transitions(listen)
```

This section is where the `create_transitions` methods are called, which specify the actual states that each state should transition to.

```
sm =StateMachine(env)
sm.add_state(listen, first=True)  # Add to StateMachine
sm.add_state(respond)
```

This snippet instantiates the `StateMachine` object. The states we instantiated on lines 33-34 are added to the state machine with the `add_state` method.
This line starts the state machine.

5.2 API design

The API is designed to provide high-level domain-specific primitives, with abstraction levels on the social and emergent primitive spectrum. The API is split into several important classes that perform different tasks: Environment, Object (with subclasses Robot and Person), Query and StateMachine. A high-level overview of the relationships between important API classes is illustrated in Figure 5.7 and Figure 5.8.

Figure 5.7 illustrates classes important for the social-interaction specific aspects of the API. The Environment class contains the objects perceived by the robot’s perception system. The Object class encapsulates functions and attributes common to all objects; the API provides two subclasses of Object: Robot and Person. Robot contains methods to perform social actions and understand human speech. People that interact with the robot are represented by the Person class which contains methods for understanding human speech. The HriQuery class is used to filter objects perceived by the robot's perception system.
Figure 5.8 illustrates the imperative state machine and its associated classes, which is used to orchestrate dialogue, a cooperative process of communication where information is shared between two or more individuals (Fong et al., 2003).

The rest of this section describes each part of the API in more detail and gives explanations for their design decisions; in particular, how this benefits usability and social-interaction programming.

### 5.2.1 Environment

The Environment class encapsulates the objects the robot perceives in its environment, including the robot itself. These are represented by two attributes: objects and robot. The objects attribute is a list that contains the objects currently perceived by the robot. Objects are automatically added and removed from the objects list when they are perceived or disappear from the robot's perception system. The robot attribute references a Robot instance, which encapsulates the actions of a robot. The class and attribute names were chosen to support role expressiveness.

### 5.2.2 Object

The Object class contains functionality common to all objects and is inherited by all classes that represent a particular type of object. The API already defines two Object subclasses for programmers: Robot, which represents the robot being programmed; and
Person, which represents a human being and their body parts. The Object classes’ functions are explained in the rest of this section.

The Object classes’ most important functions are used to calculate spatial relations between objects, for example, calculating the distance between the robot and a particular person. These are explained below.

\textit{distance\_to(\text{other})}

Finds the distance from one object (caller) to another (other), e.g. the distance from the robot to a particular person.

\textit{left\_of(\text{other})}

Returns whether the object (caller) is left of another object (other) or not.

\textit{right\_of(\text{other})}

Returns whether the object (caller) is right of another object (other) or not.

\textit{infront\_of(\text{other})}

Returns whether the object (caller) is in front of another object (other) or not.

\textit{behind(\text{other})}

Returns whether the object (caller) is in behind another object (other) or not.

\begin{verbatim}
person.distance_to(robot) >> 2.0
person.left_of(robot) >> True
person.right_of(robot) >> False
person.infront_of(robot) >> True
person.behind(robot) >> False
\end{verbatim}

\textbf{Figure 5.9.} Object spatial relations.

5.2.3 Robot

The Robot class represents the robot being programmed. Its main design goal was to provide high-level implementations of the social primitives for making speech, performing gestures and understanding human speech described in Chapter 7. The Robot
classes’ action and understanding human speech functions are explained in the rest of this section.

**Actions**

The most important function for the **Robot** class is **say_to** (Figure 5.10), which makes the robot speak and gesture to a person or a group of people. The **say_to** function was designed as a high-level implementation of the requirements of robot speech and gesture presented in Chapter 7. These include: synthesise voice, specify who is being spoken to, synchronise gestures with speech and the ability to gesture. The **say_to** function is explained in more detail below.

**say_to**(text, audience)

When the **say_to** function is executed, the robot begins by making eye contact with the person specified by the **audience** parameter (specifies who is being spoken to). Once eye contact is made it starts synthesizing the text in the **text** parameter. If more than one person is supplied to the **audience** parameter, the initial person the robot gazes at will be selected randomly. Additionally, whenever a new sentence is reached the robot will change its gaze to a random person.

The text supplied to the **text** parameter can optionally be marked up with gesture tags to make the robot gesture in time with its speech. This fulfils most gesture making requirements, including body language, facial expressions, synchronise with speech and target gesture at an object. The following list has example gestures, including both body language (wave, hands on hips, point) and facial expressions (red-eyes, blue-eyes):

- Wave: "<wave> Hello human </wave>"
- Hands on hips: "<hips> I am angry with you </hips>"
- Point arm right: "<point-right> look at that over there </point-right>"
- Point arm left: "no that <point-left> thing looks more interesting </point-left>"
- Red eye colour: "<red-eyes> I am the start of the robopocalypse </red-eyes>"
- Blue eye colour: "<blue-eyes> maybe not </blue-eyes>"

The **say_to** function name was chosen to reinforce role expressiveness; **say_to**(text, audience) suggests the robot is able to say something (text) to one or more people (audience). The gesture markup language syntax was chosen to closely map to the act of synchronising gestures with speech, one of the social-interaction
requirements. To achieve this, tags surround the text: opening tags specify a gesture start "<wave>" and closing tags "</wave>" when it stops. Gesture tags are specified by the name of the gesture to keep the notation terse. Other systems such as Interaction Composer (Glas et al., 2012) and BML (Kopp et al., 2006) use a more diffuse syntax, e.g. "<gesture type='wave'> <gesture>".

1 robot.say_to("Hello", people)
2 robot.say_to("<wave> Hello </wave>", people)
3 robot.say_to("<point target={0}> Who is that? </point>", people, person1)

Figure 5.10. say_to action.

Understanding human speech

The Robot class also contains functions that are together used to understand human speech, in particular, verbal commands (Chapter 7). These functions can be used to add utterances that the robot should listen for and associate them with high-level meanings. These functions are explained in more detail below.

associate_utterances_with_meaning(meaning, utterances, context=None)

This function associate’s utterances people say with higher level meanings to enable verbal commands. An example is shown in Figure 5.11, two meanings are created: greet and insult. Different synonyms for these are created by associating a set of utterances people could say with those meanings. For example, ‘hello’ and ‘hi’ are both greetings while ‘bye’ and ‘see ya’ are farewells.

1 meanings = Enum('greet', 'farewell')
2 robot.associate_utterances_with_meaning(meanings.greet,
3    ['hello', 'hi'])
4 robot.associate_utterances_with_meaning(meanings.farewell,
5    ['bye', 'see ya'])

Figure 5.11. Associating utterances with meanings.

An optional feature is that utterances can be assigned to a meaning based on a particular context (a unique key). This is useful when you don’t want to treat two or

2 This function was called add_meaning_to_utterances_map in the version used in the evaluation, it has been renamed here for clarity.
more utterances as synonyms because they can mean different things in different contexts used in your application.

```python
1 # Define utterance meanings and generate language model
2 meanings = Enum('greet')
3 robot.associate_utterances_with_meaning(meanings.greet, ['oi'], context='UK')
4 robot.associate_utterances_with_meaning(meanings.greet, ['mate'], context='NZ')
```

**Figure 5.12. Using contexts.**

`generate_language_models()`

After all utterances and meanings have been associated, the language models that detect the utterances can be generated. Call this method to generate them.

`set_context(context)`

This method changes the context of the auditory perception system. This loads the appropriate language model to detect utterances for this context.

`restart_listening()`

Resets the state of the auditory perception system.

### 5.2.4 Person

The `Person` class represents a person that has been perceived by the robot's perception system. As stated earlier, `Person` objects are automatically instantiated by the `Environment` instance when the robot perceives them. The most significant function for the `Person` class is `said_to` which is used to find out if a specific person said an utterance with a certain meaning.

`said_to(meaning, other)`

It is used to find out if a specific person (who the speaker is) said an utterance with a particular meaning (what the speaker said) to another object, such as the robot (who they are speaking to). It returns a Boolean indicating if this is true or not. In an ideal world, realising this on a mobile robot would use sound source localisation, tracking and separation to isolate an audio stream for each person; each audio track is then processed individually by a separate speech recogniser e.g. (Grondin, Létourneau, Ferland, Rousseau, & Michaud, 2013). We do not implement such a system because it is beyond our scope, however, this API structure would allow programmers to take advantage of
three requirements of understanding human speech should it be implemented, including: what the speaker said, who is speaking and who they are speaking to.

5.2.5 HriQuery

The HriQuery class is used to filter objects from the robot’s environment. Objects can be filtered by type (e.g. Person objects) or by distance (e.g. objects closer than 2m), sorted by an attribute (e.g. closest object), or selected (e.g. people who said “yes” to the robot).

HriQuery is a subclass of Python-ASQ (Smallshire, n.d.), a Python-based fluent data query interface. Fluent interfaces focus on enabling the programmer to link method calls together as if they were forming linguistic sentences (Fowler, 2010). Python-ASQ realises its fluent interface using method chaining, which is where methods are called in sequence, each call returns an object which can be used to further modify the objects state. A simple example of method chaining is illustrated in Figure 5.13, where a five-year-old male cat called ‘Bella’ is created and assigned to a variable.

```python
bella = cat().name('Bella').age(5).gender(Male)
```

Figure 5.13. Simple method chaining example.

The HriQuery class has a number of methods that can be chained together, which allow objects to be matched by type (of_type), matched where a predicate returns true (where) and sorted by a key (order_by, order_by_descending). An example is illustrated in Figure 5.14, where we select all people that are closer than 1 metre to the robot.

```python
env = Environment()
close_people = query(env.objects)
        .of_type(Person)
        .where(lambda p: p.distance_to(env.robot) < 1)
```

Figure 5.14. Selecting people less than 1 metre.

The fluent Python-ASQ (Smallshire, n.d.) interface was chosen for use with the social-interaction API because it allows object filtering and sorting to be expressed in a much terser way than writing for loops and if statements. This is especially important because objects are commonly filtered and sorted in the exemplar game show application.
Additionally, the fluent Python-ASQ (Smallshire, n.d.) interface is also arguably easier to understand than list comprehensions (Wadler, 1987), an alternate way to make object filtering and sorting terser. The Query classes’ methods are explained in more detail below.

**query(iterable)**

Creates a query from an iterable object (e.g. a list or another HriQuery). Returns an HriQuery instance.

**of_type(classinfo)**

Selects all objects of a particular type, specified by the classinfo parameter, e.g. all Person objects. Returns an HriQuery instance.

**where(predicate)**

Selects all objects whose predicates return true. Returns an HriQuery instance. The predicate parameter is an inline function that returns a Boolean, in Python these are called lambda statements. For example, the following lambda statement returns whether a + b equal 4:

```python
func = lambda a, b: a + b == 4
func(2, 2)
>>> True
func(2, 1)
>>> False
```

A complete example of a query using the where method is shown below; a query is defined to find all of the objects to the left of the robot:

```python
env = Environment()
query(env.objects).where(lambda o: o.left_of(env.robot))
```

When this query is executed, the where function iterates through all of the objects in the environment instance. For each object, it evaluates whether the object is left of the robot. If the lambda statement returns true, the object is added to the list of items to return.

**order_by(key=identity)**
Sort the objects in an increasing fashion using a key. They key is a lambda, but in this case, the lambda must return a numerical value by which the objects can be sorted. A complete example is shown below, where the objects are sorted based on their distance to the robot. Returns an HriOrderedQuery, which has a few extra functions for sorting: then_increasing and then_decreasing.

```python
env = Environment()
objects = env.objects
robot = env.robot

query(objects).order_by(lambda o: o.distance_to(robot))

```

Sort the objects in a decreasing fashion using a key. Returns an OrderedQuery, which has a few extra functions for sorting: then_increasing and then_decreasing.

take(n=1)

Takes the first n items from the HriQuery.

to_list()

Executes an HriQuery, returning the list of items.

5.2.6 Dialogue

To explore how primitives, regardless of their abstraction level, can be orchestrated to create higher level social-interaction (RQ2), this iteration of the API uses an imperative finite state machine to combine primitives into dialogue. Dialogue is a cooperative process of communication where information is shared between two or more individuals (Fong et al., 2003). To create dialogue, three key classes are used: StateMachine, State and Transition, these are explained below.

In the StateMachine, dialogue is represented across a number of states, represented by the class State. The StateMachine can transition to a new state via an explicit Transition or a QueryEvent based transition. Much of the design of the state machine was inspired by ROS’s SMACH (Bohren & Cousins, 2010).

State

To define a new State, the programmer defines a new class to represent it and subclasses the State class (Figure 5.15, line 23 & 30). Programmer defined State’s
are typically instantiated after the states are defined (Figure 5.15, line 39-40). State has three key methods: execute, add_transition and create_transitions. These are explained below.

**execute(e)**

To specify robot behaviour, the programmer overrides the execute method of the State class and populates it with primitives to control robot social-interaction, e.g. robot.say_to commands (Figure 5.15, lines 23-27). Transitioning to another state is signalled by returning a Transition instance.

**add_transition(event, destination, id=None, scope=Scope.AFTER)**

Adds an event based Transition to a State. The parameter event accepts an Event and the parameter destination accepts a State (Figure 5.15, line 28). When event fires, the StateMachine transitions to the State specified by the destination parameter.

**create_transitions(s1, s2, s3, ...)**

This method is used to create event-based transitions (Figure 5.15, lines 25-28) or to store particular states for referencing in the overridden execute method (Figure 5.15, lines 37). This method is usually called after the programmer defined State’s have been instantiated (Figure 5.15, line 42-43).

```python
# Define states
class Listen(State):
    def create_transitions(self, next_state):
        q = people.where(lambda p: p.said_to(
            meanings.greet, robot))
        event = QueryEvent(q)
        self.add_transition(event, next_state, id="greeted")

class Respond(State):
    def execute(self, e):
        if e.id == "greeted":
            robot.say_to("Hello, nice to meet you!", people)
            return Transition(self.next_state)
        def create_transitions(self, next_state):
            self.next_state = next_state

listen = Listen() # Instantiate states
respond = Respond()
listen.create_transitions(respond) # Define transitions
respond.create_transitions(listen)
```

**Figure 5.15.** Defining a new State.
Transition

The Transition class is used to signal a transition from one state to another. It has one important function, its constructor, which is explained below.

\[
\text{\texttt{init}}(\text{destination})
\]

The parameter destination accepts an object of class State which the StateMachine will transition too.

QueryEvent

By default the QueryEvent class fires an event when a Query returns more one or more items. This can be customised through its constructor, which is explained below.

\[
\text{\texttt{init}}(\text{query, guard=lambda data: len(data) > 0, scope=Scope.AFTER})
\]

The parameter query (type Query) is the data source for the event. When the query is executed, the guard function is also evaluated, if it returns true then the event is fired. By default, the guard function returns true if the query returns one or more items.

StateMachine

The StateMachine is typically instantiated once everything else has been setup (Figure 5.16, line 38). It has three key methods, including its constructor, add_state and start.

\[
\text{\texttt{init}}(\text{environment})
\]

The StateMachine constructor takes an Environment instance as a parameter.

\[
\text{\texttt{add\_state(state)}}
\]

All State instances must be added to the StateMachine through this method (Figure 5.16, lines 39-40). The parameter state takes a State instance.

\[
\text{\texttt{start()}}
\]

This method starts the state machine (Figure 5.16, line 41).

```python
38  sm = StateMachine(env)
39  sm.add_state(listen, first=True)  # Add to StateMachine
40  sm.add_state/respond)
41  sm.start()  # Start state machine
```

Figure 5.16. StateMachine example.
In summary, my API provides high-level, domain-specific primitives through the Environment, Robot and Person classes. Objects are filtered and sorted with the HriQuery class. Dialogue is programmed with theStateMachine and its associated classes. The next section describes the evaluation of the API.

5.3 User evaluation

The API was evaluated with a usability study where programmers used the API to create a social application and then reflected on their experience. There were 9 participants in the study (P3 - P11). All were expert programmers with 3 to 10 years programming experience, except one, who withdrew due to a lack of object oriented programming experience. Five of the participants were male, three were female and the majority had no experience programming robots (one had six months experience working on a robotics related research project).

The specific tool used to evaluate the usability of the API was the CD Questionnaire (Blackwell & Green, 2000), which is designed to present Cognitive Dimensions in a way that end users of notations can readily understand. The goal is to enable end users, rather than designers, to evaluate a system with the Cognitive Dimensions.

5.3.1 Method

Before participants started the study, they completed a background questionnaire concerning their programming experience (summarised above). The study itself consisted of four phases:

1. Play game show with researcher and robot (5 minutes).
2. Read API documentation (20-30 minutes).
3. Complete a set of tasks (30-40 minutes).
4. Reflect on experience by completing the CD Questionnaire (Blackwell & Green, 2000) (30 minutes).

Participants first interacted with the robot to understand the types of interactions Nao was capable of. This interaction was a multiplayer game show (Figure 2) where the Nao

\[\text{\footnotesize \textsuperscript{3} P1 & P2 were pilot testers; their results were not included in the analysis.}\]
robot acts as the game show’s host. Nao interacts with two teams of people autonomously, asking aloud a multiple choice question for each round of the game (making speech). As Nao speaks to people, it gazes at them and makes gestures synchronised with its speech (making gestures). Each team has a button to press to answer a question. A team’s verbal response is recognized (understanding human speech): If the answer is correct then Nao increases the team’s score, otherwise, Nao does one of two things: give the other team a chance to answer or subtract points from the team that got the question wrong (dialogue). After a set number of questions, Nao announces the winner and loser of the show.

After interacting with the robot, participants spent 20-30 minutes reading the API documentation and an example program (Appendix B). This overviews the API’s most important classes, what their salient functions and attributes do and how they are used. In the example program, the robot greets a person and responds positively or negatively based on the user’s response. The example is provided with step-by-step explanations of how each part works.

![Figure 5.17. Game show setup.](image)

The participants then conduct a series of tasks to convert the example program into a new scenario, a number guessing game. Here, Nao asks a person to guess what number Nao is thinking of. The person responds with a number from one to three. If the response matches Nao’s number, Nao tells them they were correct, otherwise, Nao tells them they were wrong. Participants were observed while they completed the tasks; during this time the researcher took notes and asked questions if there was a need to clarify why they programmed in a particular way.
At the conclusion of the tasks, participants were given the CD questionnaire designed by Blackwell & Green (2000) to reflect on their experience using our API to program social-interaction.

5.3.2 Results

Results analysis consisted of classifying questionnaire responses by whether they were positive, equivocal or negative; based on how Blackwell & Green (2000) analysed responses to the questionnaire. This was further broken down into general positive and negative responses and specific positive and negative responses. General responses just indicate whether the notation was acceptable with respect to a particular dimension; “yes”, “no”, “easy” and “hard” are examples of this. Specific responses show how specific usability features perform against a particular dimension. In addition to this, the researchers’ observations were integrated with this data.

The general responses indicate an overall positive impression of the notation. Participants responded with 49 general positive comments and only 3 general negative comments. Specific reasons why participants had a positive impression include: its object-oriented nature (P6); it is clear, concise and the class definitions are well thought out (P6, P8, P10); the notation is easy to understand (P10); and it allows programmers to express emotional emphasis and empathy on the robot (P8). Individual dimensions with the most general positive comments include visibility & juxtaposability (8 comments), role expressiveness (9 comments), closeness of mapping (6 comments) and progressive evaluation (6 comments).

The specific responses provide formative feedback about how specific usability features perform with respect to a particular dimension. Participants’ specific positive and negative responses were fairly even, with 51 specific positive and 53 specific negative comments. Two themes emerge from these comments. First, participants generally appreciated the domain-specific interfaces of the API. Second, participants had difficulty using the imperative state machine. The following paragraphs discuss the APIs domain-specific interfaces and the imperative state machine in the context of the Cognitive Dimensions of Notations.

Domain-specific interfaces

The first theme that emerged from the specific responses was that participants appreciated the domain-specific interfaces of the API, including their visibility & role
expressiveness, close mapping to social-interaction and terseness. These are discussed in the following paragraphs.

**Visibility, Juxtaposability & Role Expressiveness.** Participant’s responses indicate that the organisation and object oriented nature of the API gave it good local visibility. This refers to the visibility of the APIs exposed classes and methods as opposed to the visibility of lower level code. For example, when asked how easy it is to find various parts of the notation, P10 responded that it was “simple because the organisation of the notation is clear and concise” and P6 said that it was “intuitive as it is object oriented.”

Participants commented that the API had good role expressiveness, for reasons related to its good local visibility. For example, P3 stated that it was easy to tell how each part of the API fits into the overall scheme of things because “the structures in this API are similar to those in any OO language.”

**Closeness of Mapping.** Participants indicated that much of the notation had a close mapping to the programs they created, for example, P4 stated that the notation was “pretty close in some parts (e.g. robot.say_to).” These parts of the API had a positive effect on the diffuseness & terseness of the notation which are discussed next.

**Diffuseness & Terseness.** Participants’ responses here indicate that the domain-specific aspects of the API had a positive effect on the terseness of the notation. For example, P4 stated that the API lets you say what you want reasonably briefly because the notation “is domain specific”. This is likely because domain-specific languages have a close mapping to the problem domain they describe, allowing programmers to express what they want with fewer primitives than a non-domain specific language. Similarly, P6 said that the API was “Brief & concise as the API is well-written” and P8 said the notation lets you say what you want reasonably briefly because “Each element (class definition) was well thought out.”

Recognising people’s speech was the exception to these responses because it took a lot of space to program speech recognition. For example, P3 commented that “it should have been a lot easier to determine exactly what a user said” and that certain speech related things would “have been tedious to program.” Observations indicate that speech recognition aspects of the API could have a closer mapping to the social-interaction domain, for example, the researcher observed P9 suggesting that a higher level language, e.g. listen, would have been better.
The second theme that emerged from the specific responses was that the imperative state machine was difficult to use. In fact, over 60% of the specific negative responses (31 of 53) related to the imperative state machine. The remaining 22 specific negative comments related to issues discussed in the previous paragraphs and minor usability issues such as inappropriate function names and easily fixable inconsistencies. The purpose of the imperative state machine is to author dialogue; a higher level aspect of social-interaction than primitives. The dimensions with the most specific negative responses for the imperative state machine include: diffuseness & terseness (6), premature commitment (4), hard mental operations (5) and error proneness (5); these are discussed below.

**Diffuseness & Terseness**. Responses about diffuseness & terseness indicated that the code required to transition the state machine was diffuse (6). For example, P4 responded that “many similar/grouped actions/events” took a lot of space to describe. Users have to instantiate several classes (Query, QueryEvent) and call several methods to create an event-based state transition, which is probably the reason why this part of the notation is diffuse. Participants were observed having trouble with these parts of the notation (P9, P10, P8, P4).

**Premature Commitment**. The order that the state machine is set up requires some premature commitment. States have to be defined first, then transitions created by calling create_transitions, supplying this method with specific state instances. States must then be added to the state machine. Last, the state machine should be started. P9 noticed this, stating that you have to “think about the sequence of instantiating the different parts such as states, objects etc…” This could create errors at runtime if the programmer defines things in the wrong order.

**Hard Mental Operations**. Specific negative responses indicated programming state changes to perform dialogue management required much mental effort (5). E.g. P4 said “Probably moving between States in the state machine & passing arguments to the state” required the most mental effort. Other participants had similar views, but that the notation was easily grasped if this was understood. P3 commented he had “Some difficulty with queries/events/ State changes at first, but once that was figured out it was all fairly simple.”
Participants made a number of comments that they needed to sketch or plan the design of the state machine before they programmed it. This is an indicator that they experience hard mental operations when constructing their state machines. For example, P3 stated that “you would need to have an idea of what States you need in the app, and how you move between them. This would be easier to sketch out first rather than doing it within the API” and P10 commented that they needed to “plan the design part in advance.” In addition, two participants were observed making comments about how they wanted a visual representation to help them create the state machine (P6 & P9).

**Error Proneness.** Most responses here related to creating dialogue with the state machine (5). Users reported they misnamed state id’s, made mistakes due to copy and pasting queries, events and state transitions and left query & event declarations unused.

**Other usability issues**

Participants were observed questioning why they needed to add Query instances to the Environment (P4, P6, P7, P9, P10, P11).

The names for queries were found to be inconsistent (P7). The documentation refers to them as Query, the class is called HriQuery and they are instantiated with the factory method query.

Over a third of participants had trouble using queries and lambda expressions which they attributed to a strange closeness of mapping (P11) and less than ideal role expressiveness (P6, P7).

### 5.4 Stakeholder interview

Subsequent to the user evaluation, I performed an email based stakeholder interview with members of Hanson Robotics (Hanson Robotics, n.d.) and the OpenCog Foundation (OpenCog Foundation, n.d.) to discuss how well the API addressed their requirements for programming socially interactive human-like robots. The participants in the interview were positive about how the API could address their use cases. However, the interview uncovered three areas where the API could be improved to support the stakeholders use cases. These are discussed in the following paragraphs.

First, Hanson Robotics makes robots with human-like faces, examples are illustrated in Figure 5.18. To support these robots, the API would need abstractions for programming robot facial expressions.
Second, the interview showed that the social-interaction domain had been abstracted too far for the stakeholders use cases. The API would need to provide a finer granularity of control to address this. For example, in the stakeholders use cases, programmers need to make the robot perform gestures or expressions while it isn’t speaking. This was not currently possible with the Robot classes’ say_to method which can only make the robot gesture whilst it is speaking. To address this, the say_to method would need to be split into finer-grained methods that enable independent control of speech, gaze, gestures and expressions. Additionally, in the stakeholders use cases, programmers need to be able to command the robot to perform multiple gestures or expressions simultaneously; this would also be addressed by splitting the say_to method into multiple methods.

Third, in some of the stakeholders use cases it was important to be able to store robot actions in data structures rather than method calls. This enables the programmer to store a particular action and use it multiple times in the future.

5.5 Discussion

The social-interaction API, presented earlier in this chapter, was created as a first step in exploring how to create usable social robot programming tools. The user evaluation and stakeholder interview address aspects of research questions one and two of this thesis:

**RQ1** What abstraction level is appropriate for programming social robot applications?

**RQ1.1** What are the trade-offs associated with using different abstraction levels to program robot social-interaction?
RQ2 How should primitives, regardless of their abstraction level, be orchestrated to create higher level social-interaction such as dialogue and non-verbal behaviour?

This section discusses how the user evaluation and stakeholder interview addressed these two research questions and the implications for the rest of this thesis.

The user evaluation results support the idea that high-level, domain-specific primitives improve the usability of social robot programming (RQ1). There are a number of reasons for this, including they provide good visibility & role expressiveness of various parts of the notation; they have a close mapping to the problem domain of social-interaction, which in theory makes it easier to express a solution in the problem domain in question (Green & Petre, 1996); and they are terse, allowing the programmer to express their program succinctly. Some of these findings fit with existing research, which has also found that domain-specific languages improve usability in similar ways, but when applied to different areas (Van Deursen, Klint, & Visser, 2000).

The stakeholder interview revealed that the social-interaction domain may have been abstracted too far, reducing the reusability of some primitives. For example, the stakeholders use cases required finer grained control of social-interaction, e.g. the ability to control gestures independently of speech. This is a trade-off caused by the APIs high abstraction level. To address this, a slightly lower level, finer grained set of primitives could be created and their usability evaluated.

The limitation of this study is that it was performed on primitives designed to be used in one exemplar use case and with one robot. To show that the results have more general application, a similar study should be performed with primitives designed to be used in a wider range of social-interaction scenarios and with more robots. Users didn’t get to spend much time with the API (45 – 60 minutes), which gives little time for developing an in depth understanding of the APIs benefits and limitations. This could be addressed by performing a study where users spend a more realistic period of time with the API, such as 1 – 2 weeks. This would produce more detailed, ecologically valid results. Lastly, in this study users were helping to answer two independent research questions (RQ1 & RQ2), this could be addressed by only focussing on one issue. The next chapter addresses a number of these limitations.

The user evaluation results also suggest that imperative state machines have a number of usability issues when used to orchestrate primitives into social-interaction (RQ2). This
is because: they require premature commitment in the order that things need to be defined; users often experience hard mental operations when programming them, for example, they often need to sketch or plan what the state machine will look like in advance; and lastly, they are error prone e.g. it is easy to misname state identifiers.

The main limitation of the findings for RQ2 is that the evaluation only looked at one imperative state machine representation. However, the representation was influenced by SMACH (Bohren & Cousins, 2010), a well-known imperative state machine representation used by the ROS community. It also doesn’t examine the effect of having a state machine visualizer, which tools such as SMACH often have. Nevertheless, the user evaluation still provides evidence that more usable notations are needed for combining primitives into social-interaction. Visual editors or visual editors in combination with a textual representation are possible solutions to this problem, especially since participants commented that they needed to sketch this type of logic rather than type it. Human-authored game character behaviour is very common in commercial game engines (Robertson & Watson, 2014); valuable insights are likely to be gleaned from this industry. The implications of these findings for future research are discussed in more detail in Chapter 9.

5.6 Summary

This chapter started by presenting the first iteration of an API designed to enable programmers to create social-interaction applications for a Nao humanoid robot. The API provides high-level, domain-specific primitives which can be used to: represent objects and compute their spatial relations; control the actions of a Nao humanoid robot including speech and gestures; listen for verbal utterances; and author dialogue and behaviour with a textual state machine.

The APIs usability was evaluated with a standard usability test along with the CD Questionnaire (Blackwell & Green, 2000). The evaluation revealed two key themes centred on the research questions of this thesis. First, high-level, domain-specific primitives show promise as an appropriate abstraction level for programming robot social-interaction (RQ1). This is because they give the notation good visibility & role expressiveness, a close mapping to the social-interaction domain and they are generally terse. The second theme showed that imperative state machines can be difficult to use to orchestrate primitives into social-interaction (RQ2). This is because they are diffuse, their
use involves premature commitment and hard mental operations and aspects of them are error prone.

A stakeholder interview was conducted subsequent to the user evaluation. The key finding of the stakeholder interview was that the social-interaction domain may have been abstracted too far.

The next chapter presents the second iteration of the social-interaction API which is used to further investigate what abstraction level is appropriate for programming social robot applications (RQ1). The next iteration refactors the primitives into a slightly lower abstraction level to provide a finer granularity of control, it also supports more robots and addresses other usability issues found from the user evaluation in this chapter.
Iteration 2: Social-interaction refactored

This chapter presents a revised version of the social-interaction API presented in the previous chapter. The previous API was designed to make it easy to program the social-interaction of a Nao humanoid robot. The API was evaluated with a user evaluation and the CD Questionnaire (Blackwell & Green, 2000). After the user evaluation, a stakeholder interview was performed to discuss how the API could be used to program their socially interactive human-like robots. The results of the evaluation and stakeholder interview have a number of implications for the first research question of this thesis, which are discussed below:

RQ1 What abstraction level is appropriate for programming social robot applications?

RQ1.1 What are the trade-offs associated with using different abstraction levels to program robot social-interaction?

The analysis of the CD Questionnaire showed that high-level, domain-specific primitives benefited the usability of the API by giving it good visibility & role expressiveness, a close mapping to the social-interaction domain and an appropriate level of terseness. Whilst the participants in the stakeholder interview were generally positive about how the API could be used to program their use cases, the interview also revealed that the social-interaction domain had likely been abstracted too far, taking too much control away from the programmer. This is an important finding for RQ1 because it suggests that a lower abstraction level than that used in the previous API may be more appropriate for programming socially interactive robots.

The evaluation and stakeholder interview revealed two limitations with the previous API. First, the API had only been designed for one use case (the multiplayer game show)
and one robot (Nao). Second, the CD Questionnaire unearthed a number of minor usability issues.

To further explore research question one of this thesis (RQ1) and the findings of the previous chapter, a second version of the social-interaction API was designed, implemented and evaluated. This iteration of development had four objectives:

O1. Refactor the APIs abstraction level to provide finer control of social-interaction
O2. Support multiple robots
   O2.1. Broaden the aspects of social-interaction the API can program by supporting new robots and scenarios
   O2.2. Redesign the APIs software architecture to enable it to support multiple robot platforms
O3. Address the usability issues of the previous API

The API underwent a number of changes to meet these objectives; the resulting API is split into several important classes that perform different tasks: Entity (with subclasses Robot and Person), World and Query. An overview is illustrated in Figure 6.1; this is discussed in more detail in Section 6.2.

![Figure 6.1. Overview of the social-interaction API & framework.](image-url)
The **Robot** class represents the robot being programmed, it was refactored to meet objectives 1 & 2, providing finer control of social-interaction as well as allowing more aspects of social-interaction to be programmed (facial-expressions). The abstractions are closer to the *social primitive* abstraction level than the previous API, which also had *emergent primitives*. In this iteration, the **Robot** class was given more action primitives, including the ability to speak, gaze at objects, make gestures and show facial expressions; finer means of controlling actions; the ability to receive callbacks about action events; and the ability to subscribe to social communication, in this case people’s speech. These changes to the **Robot** class are discussed from Sections 6.2.2 to 6.2.6.

To broaden the aspects of social-interaction the API can program (objective 2.1); two new humanoid robots and a new scenario were used as exemplar use cases during the design and development of the API. The two new humanoid robots are the Zeno and Zoidstein robots (Hanson Robotics, n.d.; OpenCog Foundation, n.d.; WowWee Group Limited, n.d.) illustrated in Figure 6.2. These two new robots can express a wider range of social-interaction than the Nao robot (Aldebaran - SoftBank Group, n.d.) due to their human-like faces. Support was maintained for the Nao robot.

The new exemplar scenario is an application designed to enable robots to communicate emotionally and exhibit unique personalities (OpenCog Foundation, n.d.). The algorithms that drive the interaction include a behaviour tree and a chat bot system. This was an ideal use case to explore because behaviour trees are commonly used by game industry to create artificial intelligent characters for games (Robertson & Watson, 2014).

![Figure 6.2. Humanoid robots: (a) Nao, (b) Zeno and (c) Zoidstein. Source for images: Nao - used with permission from Samar Pant; Zeno - (Jurvetson, 2008). Note that Zeno had a different body from the one pictured above.](image)
To meet objective 2.2, the APIs software architecture was redesigned to enable it to support multiple robot platforms. This enabled all three robots, Nao, Zeno and Zoidstein to be programmed with the API. The API is able to support multiple robot platforms because of its vendor agnostic action and perception interfaces, which abstract away robot specific implementation details (Figure 6.1). To make a robot communicate via the action interface, the robot vendor needs to implement abstract action server classes (one for each action type), filling in the abstract methods with calls to the libraries that make their robot perform actions. To make a robot communicate via the perception interface, the robot vendor needs to implement versions of their perception algorithms that publish data to the Perception Synthesiser and the ROS Transform Library (Foote, 2013). This is discussed in more depth in Chapter 7.

The last objective was to fix the usability issues discovered from the CD Questionnaire used in the previous usability study. This resulted in changes to the Person and Query classes and the introduction of World, discussed in Sections 6.2.7, 6.2.8 and 6.2.9 respectively.

The API was designed and with a user-centred design methodology (Norman, 2002), using the Cognitive Dimensions (Green & Petre, 1996) as a design framework to weigh up the merit of various design choices. Once the design and development were completed, the API was evaluated with a group of programmers over a period of two weeks. They spent their time building an application that made a Nao humanoid robot act as an interface to an online chat program.

Section 6.1 presents the API through a series of example programs. Section 6.2 gives more details about the API and discusses the design decisions that were made in the context of the Cognitive Dimensions framework (Green & Petre, 1996). The evaluation is presented in Section 6.3.

### 6.1 Example programs

Before describing each API module and their design decisions in depth, three short example programs are illustrated and explained. The first example program makes a Zeno robot greet a simulated person, the second makes a Nao robot grab a real person’s attention and the last makes a Nao robot listen to a person.
6.1.1 Greet a person

The first example makes Zeno greet a simulated person (Figure 6.3), taking advantage of the Zeno’s ability to make facial expressions. The interaction starts with Zeno initiating eye contact with a person. Once Zeno has made eye contact, he greets the person verbally with a big smile on his face.

```python
1 robot = Zeno()
2 person = Person(1)
3 robot.gaze_and_wait(person.head)
4 ah1 = robot.say("Good morning!")
5 ah2 = robot.expression(Expression.Smile)
6 robot.wait(ah1, ah2)
```

Figure 6.3. Example greeting program.

The following paragraphs explain how the code works.

Line 1 instantiates a Zeno robot class, which is used to control Zeno. Each robot is represented by its own class. Line 2 instantiates a Person instance. Usually, Person instances are created by the perception system, but for testing purposes they can be instantiated directly, as is the case here.

```python
4 robot.gaze_and_wait(person.head)
```

This line makes the robot gaze at the person's head. It is a blocking function, so it returns when the robot has made eye contact with the person.

```python
6 ah1 = robot.say("Good morning!")
7 ah2 = robot.expression(Expression.Smile)
8 robot.wait(ah1, ah2)
```

6.1.2 Grab a person’s attention

This section of code makes Zeno say "Good morning!" (line 6) smile at the same time (line 7) and wait for both actions to finish (line 8). The functions that make Zeno speak
and smile (say and expression) each return an ActionHandle which are used in the wait function (line 8). The expressions that a robot can perform, as well as their default parameters, are stored in an enum. For Zeno, this is the Expression enum. Grab a person’s attention

The second example program makes a Nao humanoid robot grab a real person’s attention (Figure 6.4). In this interaction, Nao tries to grab the person’s attention by yelling “Oi! What are ya doing?” whilst pointing at the person.

```python
1 robot = Nao()
2 world = World()
3
4 people = Query(world).select_type(Person)
5 person = random.choice(people)
6
7 ah1 = robot.say("Oi! What are ya doing?")
8 ah2 = robot.gesture(Gesture.PointLArm,
9     target=person.torso,
10    duration=2.0)
11
12 robot.wait(ah1, ah2)
```

Figure 6.4. Example grabbing attention program.

The following paragraphs explain how the code works.

This time we want the robot to interact with real, rather than simulated people. To do this, we need to access entities created by the perception system, by creating an instance of the World class (line 2). The World class provides access to all entities perceived by the robot.

```python
4 people = Query(world).select_type(Person)
5 person = random.choice(people)
```

The social-interaction API provides a Query API for selecting entities from the world with particular properties. In this example, we select all entities of the type Person (line 4). We then choose a random person to interact with (line 5).
This snippet of code is where we make Nao interact with the randomly selected person. Line 7 makes Nao say "Oi! What are ya doing?", whilst line 8 makes Nao point its left arm at the person's torso for a period of two seconds. Both of these functions are non-blocking. The last line waits for the speaking and gesture actions to finish (line 11).

### 6.1.3 Listen to someone

The last example program makes a Nao humanoid robot listen to a person’s speech (Figure 6.5). In this interaction, if Nao understands what it heard, it parrots what was said back to the speaker. If Nao didn’t understand what was said, it says, "I didn't hear you, speak up!"

```python
robot = Nao()

def i_heard_that(speech):
    if speech is not None:
        robot.say_and_wait(speech)
    else:
        robot.say_and_wait("I didn't hear you, speak up!")

robot.register_listen(i_heard_that)
rospy.spin()
```

The following paragraphs explain how the code works.

This function makes Nao respond to speech. It takes a string as input (speech). When Nao detects speech but doesn't recognise what was said, the result is None. Hence, we check to see if the speech string is None (line 4). If it isn't None, we make Nao say the text from the speech variable. Else we make Nao say, "I didn't hear you, speak up!"
This line is very important, it registers the function we just created (`i_heard_that`) with the robot class as a listen callback. This means that whenever Nao hears speech, the function `i_heard_that` is called with the parameter `speech` containing the text representation of what Nao heard. Line 10 is from ROS (Open Source Robotics Foundation, n.d.-a), not from my API, it is used to stop the Python program from exiting until a shutdown callback is received, e.g. Ctrl-c.

### 6.2 API Design

The API is split into several important classes that perform different tasks: **Entity** (with subclasses **Robot** and **Person**), **World**, **Query** and user-defined gesture and expression enumerations. A high-level overview of the relationships between important API classes is illustrated in Figure 6.6.

![Figure 6.6. API overview.](image-url)
The Entity class encapsulates functions and attributes common to all entities, or objects, including the robot itself. Robots are represented by a subclass of Entity called Robot. Robot contains a number of robot specific primitives, including the ability to call and control actions, e.g. to make gestures and facial expressions. User-defined gestures and expressions are represented as enumeration members. People are represented by the Person class, also a subclass of Entity. The World class contains all of the entities perceived by the robot’s perception system (except the Robot class instance) and can be filtered via the Query class.

The rest of this section describes each part of the API in more detail and explanations for their design decisions are given; in particular, how this benefits usability and social-interaction programming.

6.2.1 Entity

The Entity class encapsulates functions and attributes common to all entities (objects) that the robot interacts with. The functions provided by the Entity class are the same as in the previous API.

In the previous API, Entity was named Object. Programmers found this confusing because the highest level class in Python is also called object, the only difference being the lower case first letter. To fix this ambiguity, the Object base class was renamed Entity. This means that all real world objects, including robots, are now derived from the Entity class, as illustrated in Figure 6.7.

Entity has two principle subclasses already defined for programmers: Robot and Person (Figure 6.7).

The Robot class encapsulates all of the functions used to control a robot’s social-interaction with people. In the previous API programmers directly used the Robot class to program their robot. In this version the Robot class is abstract and cannot be instantiated directly, instead, each robot has its own class derived from Robot, e.g. the

---

4 Entity: “a thing with distinct and independent existence.” (Oxford Dictionaries, 2014)
Nao robot is represented by the Nao class. This is because each robot has its own unique body structure, which is represented within its own class.

The Person class models the human body and is used when programming interactions with people, e.g. look at a person’s left arm. Instances of people can either be manually instantiated or automatically populated into the World instance based on robot perception data (Sections 6.2.7 and 6.2.8).

![Diagram of entities]

Figure 6.7. Entities.

6.2.2 Robot

The Robot class represents the robot being programmed (Figure 6.8). In this iteration, the Robot class was given more action primitives, finer means of controlling actions and the ability to receive callbacks about action events and social communication.

The rest of the Robot class decisions revolved around usability. The function names were designed to favour terseness (Green & Petre, 1996) by employing short function names and the use of default parameters. Short function names help to make the API terser because they take up less space, for instance, the function names gesture and expression were chosen over make_gesture and show_expression. Default parameters also make the API terser because programmers don’t need to specify (or see) all function parameters when reading code; the downside is that it introduces hidden dependencies.

The next four sections describe the primitives provided by the Robot class.
6.2.3 Robot: actions

The first change to the Robot class was the addition of more action primitives which make robots: speak, gaze at objects, show expressions and perform gestures (Figure 6.8). This is a significant departure from the previous API, which abstracted speech, gaze and gesture into one action primitive (say_to). The say_to action primitive was divided into smaller primitives because there are many cases where a programmer needs individual control over the actions a robot performs (Goertzel, Hanson, & Yu, 2014).

Say

The say action primitive is used to make a robot speak. The Robot class has two functions that make a robot speak and another function that returns the length of time it takes to synthesise words in a sentence (Figure 6.10). These are explained below.
say(sentence)

Synthesise the text in the string `sentence`. This function is asynchronous, so it returns an `ActionHandle`; a unique identifier for the action being performed. An action handle can be used to wait for an action to complete or to cancel an action (Figure 6.10).

say_and_wait(sentence)

A synchronous version of the `say` function. The `and_wait` syntax signals that the function will not return until it completes; this was chosen to be consistent with other ROS libraries, such as actionlib (Marder-Eppstein & Pradeep, n.d.). No `ActionHandle` is returned because the action completes before it returns.

say_duration(sentence, start_word_index=None, end_word_index=None)

Returns the length of time it takes for a robot to speak a particular `sentence` or a part of that sentence. The default parameters `start_word_index` and `end_word_index` can be used to indicate a subset of the sentence to find the duration of.
robot = Nao()

ah = robot.say("Hello")

robot.wait(ah)

robot.say_and_wait("I'm a crazy robot")

print(robot.say_duration("I'm a crazy robot"))

> 2.0

print(robot.say_duration("I'm a crazy robot", 0, 1))

> 0.3

Figure 6.10. Speaking action.

Gaze

The gaze action primitive is used to make a robot gaze at an entity. The Robot class has two functions that make a robot gaze at entities (Figure 6.11). These are explained below.

gaze(target, speed=0.5)

Makes the robot gaze at a target (any subclass of Entity). The speed that the robot gazes at is controlled by the speed parameter normalised between \(0.0 < \text{speed} \leq 1.0\). This function is asynchronous and returns an ActionHandle.

gaze_and_wait(target, speed=0.5)

A synchronous version of the gaze function. No ActionHandle is returned (Figure 6.11).

robot = Nao()

person = Person(1)

ah = robot.gaze(person.head)

robot.wait(ah)

robot.gaze_and_wait(person.torso, speed=0.8)

Figure 6.11. Gaze action.

Gesture

The gestures that a particular robot can perform are represented using Python enumerations, a collection of symbolic names (members) tied to distinct values (Python Software Foundation, n.d.). Each gesture enumeration member contains the default
duration the gesture should be played for (more in Chapter 7). The Gesture enumeration members for the Nao robot are illustrated in Figure 6.12 below.

|-------------------|-------------------|------------------|------------------|------------------|------------------|------------------|------------------|-------------------|

**Figure 6.12. Nao’s gestures.**

These gestures can be executed using two functions from the Robot class (Figure 6.13). These are explained below.

* gesture(gesture, target=None, duration=Default())

  Makes the robot perform a gesture, e.g. wave. The gesture to perform is specified by the gesture parameter, a Python enumeration e.g. for Nao, Gesture.WaveLArm. The rest of the parameters have their values set to None, which means that the gesture will be performed using its default values. The target parameter is used to make the robot orient the gesture toward an object in the world (any subclass of Entity). The duration parameter is used to control the length of time the gesture plays for (seconds). This function is asynchronous and returns an ActionHandle.

* gesture_and_wait(gesture, target=None, duration=Default())

  A synchronous version of the gesture function. No ActionHandle is returned (Figure 6.13).

Alternative design decisions for gesture and expression representation included using a configuration file or hardcoding variables in a Python class. Enumerations were chosen over the two alternative design choices because, unlike the other two choices, enumeration members are autocompleted by IDEs, allowing for easier discoverability. A mixed camel case naming scheme for enumeration members was chosen to enhance readability; this differs from the default snake case Python naming convention for enumeration members.
Expression

The facial expressions that a particular robot can perform are represented using Python enumerations just as gestures are represented. Examples of the facial expressions defined for the Zeno robot are illustrated in Figure 6.14 below, they make Zeno smile, frown his mouth, open his mouth and frown his forehead respectively.

These expressions can be executed using two functions from the Robot class (Figure 6.15). These are explained below.

expression(expression, intensity=Default(), speed=Default(), duration=Default())

Makes the robot perform a facial expression, e.g. smile. The type of expression is specified by the expression parameter, a Python enumeration, e.g. Expression.Smile to make Zeno smile. The rest of the parameters have their values set to None, which means that expression will be performed using its default values (specified in the expression enumeration definition). The intensity parameter controls the strength of the expression, e.g. a smile with an intensity of 1.0 is the largest smile possible (normalized within 0.0 < speed <= 1.0). The speed parameter controls how fast the expression is performed, e.g. a smile expression with a speed of 1.0 is performed at the fastest possible speed (normalized within 0.0 < speed <= 1.0). Lastly, the duration parameter controls how long the expression lasts (seconds). This function is asynchronous and returns an ActionHandle.
expression_and_wait(expression, intensity=Default(), speed=Default(),
duration=Default())

A synchronous version of the expression function. No ActionHandle is returned (Figure 6.15).

```python
1 robot = Zeno()
2 robot.expression_and_wait(Expression.Smile, duration=1.0)
3 ah1 = robot.expression(Expression.FrownMouth)
4 ah2 = robot.expression(Expression.FrownForehead)
5 robot.wait(ah1, ah2)
6 robot.expression(Expression.Frown,
7     intensity=0.8, speed=1.0)
```

**Figure 6.15. Expression action.**

**Action objects**

It is also possible to create objects to represent the say, gaze, expression and gesture actions (Figure 6.16). Action objects are useful in situations where actions need to be pre-computed and stored for later use. Their constructor parameters are exactly the same as the Robot class actions functions discussed earlier in this section. Action objects can be executed by passing them to the Robot classes action control methods, discussed in the next section.

```python
1 say = SayAction("Hello I'm a robot")
2 gaze = GazeAction(person.head)
3 gesture = GestureAction(Gesture.WaveLArm, duration=6.0)
4 expression = ExpressionAction(Expression.Smile)
```

**Figure 6.16. Action objects.**

**6.2.4 Robot: action controls**

The second change to the Robot class was the addition of more action control primitives, which provide finer control of how actions are executed (Figure 6.17). The previous section explained the two basic methods of controlling actions: the ability to choose whether an action command returns straight away (no and_wait) or blocks until it has finished (and_wait). The rest of this section explains the other four action control primitives, which include the ability to: wait for actions to complete, stop actions and run actions simultaneously or consecutively.
Figure 6.17. Action controls.

```java
1  robot = Nao()
2  person = Person(1)
3
4  ah1 = robot.say("Hello, I will eat you alive")
5  robot.wait(ah1)
6
7  ah2 = robot.gaze(person.head)
8  robot.wait(ah2)
9
10 ah3 = robot.expression(Expression.FrownMouth)
11 robot.wait(ah3)
12
13 ah4 = robot.gesture(Gesture.WaveLArm)
14 Figure 6.18 robot.wait(ah4)
```

Figure 6.18. Waiting for actions to complete.

Wait

The wait control primitive is used to wait until an action has completed (Figure 6.18). This is explained below.
wait(*action_handles)

This function waits until the actions specified by *action_handles have completed. The *action_handles parameter is a variable length argument list which accepts one or more ActionHandle.

Stop

The stop control primitive is used to stop an action from running (Figure 6.19). This can be used to stop a robot from speaking, gazing, gesturing or showing an expression.

stop(*action_handles)

This function stops actions that are currently being performed. The actions to be stopped are specified by passing their action handles to the *action_handles parameter, a variable length list that accepts one or more ActionHandle.

```python
1  robot = Nao()
2  person = Person(1)
3
4  ah1 = robot.say("Hello, I will eat you alive")
5  time.sleep(1.0)
6  robot.stop(ah1)
7
8  ah2 = robot.gaze(person.head)
9  time.sleep(1.0)
10  robot.stop(ah2)
11
12  ah3 = robot.expression(Expression.FrownMouth)
13  time.sleep(1.0)
14  robot.stop(ah3)
15
16  ah4 = robot.gesture(Gesture.WaveLArm)
17  time.sleep(1.0)
18  robot.stop(ah4)
```

Figure 6.19. Cancelling actions.

Simultaneously

The simultaneously control primitive is used to make a robot execute actions in parallel (Figure 6.20).

simultaneously(*actions)

Executes actions simultaneously by directly passing action objects as parameters to the simultaneously function. Goal types include: SayAction, GazeAction, ExpressionAction and GestureAction. Action goals are passed to the *actions
parameter, a variable length list that accepts one or more items. This function is asynchronous returning an array of action handles; one for each goal. It is useful in situations where actions are pre-computed and will be executed later on (Figure 6.20).

```java
1 robot = Nao()
2 person = Person(1)
3 say = SayAction("Hello I'm a robot")
4 gaze = GazeAction(person.head)
5 gesture = GestureAction(Gesture.WaveLArm, duration=6.0)
6 expression = ExpressionAction(Expression.Smile)
7 ah = robot.simultaneously(say, gaze, gesture, expression)
8 robot.wait(ah)
```

Figure 6.20. Simultaneously command.

Consecutively

The consecutively control primitive is used to make a robot execute actions one after another (Figure 6.21).

```java
1 robot = Nao()
2 person = Person(1)
3 say = SayAction("Hello I'm a robot")
4 gaze = GazeAction(person.head)
5 gesture = GestureAction(Gesture.WaveLArm, duration=6.0)
6 expression = ExpressionAction(Expression.Smile)
7 ah = robot.simultaneously(say, gaze, gesture, expression)
8 robot.wait(ah)
```

Figure 6.21. Simultaneously command.

6.2.5 Robot: action callbacks

The third change to the Robot class is the ability to subscribe to callbacks that fire during the execution of actions (Figure 6.22). There are two types of callbacks for each action type: a feedback callback which indicates the current progress of the action; and a
done callback that indicates when the action has finished. There are two ways to subscribe to action callbacks. These are explained below.

![Diagram of robot action callbacks]

The first way to subscribe to an action callback is by assigning function references to the parameters `feedback_cb` and `done_cb` in one of the `Robot` classes action functions. An example is shown in Figure 6.23; the robot says, “Hello”, then every time a new word is spoken the `say_feedback` function is called printing out the current word number. When the robot has finished speaking the `say_done` function is called which prints out “I’ve finished speaking!” This method of subscribing to action callbacks was chosen because it allows fine control of behaviour based on individual robot actions as well as being consistent with the ROS actionlib library which it extends (Marder-Eppstein & Pradeep, n.d.).
robot = Nao()

def say_feedback(feedback_msg):
    print(feedback_msg.current_word_num)

def say_done():
    print("I've finished speaking!")

ah = robot.say("Hello there", \
    feedback_cb=say_feedback, \
    done_cb=say_done)
robot.wait(ah)

Figure 6.23. Speaking action callbacks using robot.say function.

Each action type has a customised feedback message. The feedback messages are illustrated in Table 6.1, they are by no means exhaustive at this point in time, but they can easily be extended if need be.

<table>
<thead>
<tr>
<th>Action type</th>
<th>Feedback parameters</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Say</td>
<td>int32 current_word_index</td>
<td>The index of the current word being spoken</td>
</tr>
<tr>
<td>Gaze</td>
<td>float32 distance_to_target</td>
<td>The distance from the gaze frame of reference to the target. This is calculated in the 2-dimensional y, z gaze plane.</td>
</tr>
<tr>
<td>Gesture</td>
<td>float32 distance_to_target</td>
<td>The distance from the gesture frame of reference to the target, in a 2-dimensional plane. For example, when pointing at a target, the gesture frame of reference is the arm end effector.</td>
</tr>
<tr>
<td>Expression</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The second way to subscribe to an action callback is by creating an action goal and assigning function references to the feedback_cb and done_cb parameters. An example of this is shown in Figure 6.30.

robot = Nao()

def say_feedback(feedback_msg):
    print(feedback_msg.current_word_num)

def say_done():
    print("I've finished speaking!")
say = SayAction("Hello there", \
    feedback_cb=say_feedback, \
    done_cb=say_done)
ah = robot.simultaneously(say)
robot.wait(ah)

Figure 6.24. Speaking action callbacks using SayGoal.
6.2.6 Robot: communication callbacks

The last change made to the Robot class is the ability to subscribe to social communication callbacks; this version provides the ability to subscribe to speech the robot has recognised (Figure 6.25).

Providing callbacks to subscribe to communication is a change from the previous API, which used a combination of queries, events and a state machine to process speech. The previous system was removed in favour of a simple callback system because of the usability issues the previous system had. In future API versions, there will also be callbacks that subscribe to the facial expressions and gestures people make.

/register_listen(callback)

Register a function as a listen callback. Whenever the robot recognises speech, the register function will be called; it must have a parameter (speech) to take the recognised
speech. If the recognition was so bad that the robot couldn’t understand anything, the function will be called and `speech` will contain `None`.

An example is illustrated in Figure 6.26 below.

```python
robot = Nao()

def i_heard_that(speech):
    if speech is not None:
        robot.say_and_wait(speech)
    else:
        robot.say_and_wait("I didn't hear you, speak up!")

robot.register_listen(i_heard_that)
rospy.spin()
```

**Figure 6.26. Example listening program.**

### 6.2.7 Person

The `Person` class models the human body and is used to program interactions with people, e.g. look at a person’s left arm (Figure 6.27).

The `Person` class represents most major external human body parts, including the person’s head, neck, torso, shoulders, elbows, hands, hips, knees and feet. Each body part type is defined as a Python class and assigned to an attribute in the `Person` class. These are illustrated in Table 6.2.

The only major change to the `Person` class is how it can be instantiated. In the previous API, `Person` classes could not be manually instantiated; instead, they were instantiated automatically based on perception data. This made progressive evaluation difficult because programmers had to stand in front of the robot every time they wanted to test their code. To address this, `Person` classes can now be manually instantiated and a ROS configuration file written to broadcast simulated coordinates. This makes it much easier for a programmer to progressively evaluate code because the programmer can test social-interaction with simulated people. An even better future step would be to create people instances based on data from a simulator.
Figure 6.27. Person instances.

Table 6.2. Person attributes.

<table>
<thead>
<tr>
<th>Person instance attribute</th>
<th>Class type</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td>Head</td>
</tr>
<tr>
<td>neck</td>
<td>Neck</td>
</tr>
<tr>
<td>torso</td>
<td>Torso</td>
</tr>
<tr>
<td>left_shoulder</td>
<td>Shoulder</td>
</tr>
<tr>
<td>left_elbow</td>
<td>Elbow</td>
</tr>
<tr>
<td>left_hand</td>
<td>Hand</td>
</tr>
<tr>
<td>left_hip</td>
<td>Hip</td>
</tr>
<tr>
<td>left_knee</td>
<td>Knee</td>
</tr>
<tr>
<td>right_shoulder</td>
<td>Shoulder</td>
</tr>
<tr>
<td>right_elbow</td>
<td>Elbow</td>
</tr>
<tr>
<td>right_hand</td>
<td>Hand</td>
</tr>
<tr>
<td>right_hip</td>
<td>Hip</td>
</tr>
<tr>
<td>right_knee</td>
<td>Knee</td>
</tr>
</tbody>
</table>
6.2.8 World

The APIs World class provides access to the entities perceived by the robot’s perception system (Figure 6.28). The World class is iterable, meaning that it can be iterated over just as a list or array can. Vendor specific perception algorithms communicate with the World instance via a perception synthesiser node and the ROS Transform Library (Foote, 2013). The perception synthesiser tells the World when to create objects to represent perceived entities and the ROS Transform Library manages object coordinates.

![Diagram of World class relationships]

In the previous API, the World class was named Environment. A number of changes were made to the Environment class in this iteration of development to improve usability.
First, the Environment class was renamed into the World class because “World” is shorter and more common (Lin et al., 2012). The aim is to make the notation as concise as possible (terse), without sacrificing comprehension.

Second, in the previous API, the objects in the Environment class were accessed via an attribute. In this iteration the World class was made iterable, meaning that it can be iterated over to search or filter the entities the perception system creates inside it. This was another small change aimed at making the API concise (terse).

Third, in the previous API, the Robot instance was retrieved via an attribute in the Environment class (now a subclass of Robot is used to program a robot, e.g. Nao). This creates a hidden dependency, because instead of simply instantiating the Robot class themselves, programmers had to instantiate the Environment class and then find the robot instance via the robot attribute. This design choice also hurt role expressiveness, by making it hard to understand what the Environment class does in the context of the wider notation. To remedy this, programmers now have to import their robot class and instantiate it themselves.

Lastly, one of the most common issues people raised in the previous usability study was why they had to add queries and other objects to the Environment to make the underlying framework aware of them (Chapter 5.3 - Other usability issues). This caused error proneness because participants often forgot to do it. In the current iteration, there is no longer any need to add entities and queries to the world as this is now done automatically.

The collective effect of these changes can be seen below; it takes much less space to describe the same concept in the current iteration than in the previous (Figure 6.29 and Figure 6.30).

```python
1 env = Environment()
2 objects = env.objects
3 robot = env.robot
4
5 people = query(objects).of_type(Person)
6 env.add_query(people)
7 robot.say_to("Hello", people)
```

Figure 6.29. Environment - old API.
6.2.9 Queries

The Query class provides a high-level interface for filtering, sorting and selecting entities perceived by a robot. It is similar to Microsoft’s LINQ (Meijer, Beckman, & Bierman, 2006) and based on Python-ASQ (Smallshire, n.d.) (Figure 6.31).

![Diagram of the Query class and its relationships with other classes such as Person, Robot, Gesture, and Expression.]

The functionality of the Query class is the same as the previous API, an example containing queries is illustrated in Figure 6.32; a Nao robot waits until one or more people are detected and then greets the closest person. Queries are used to select Person objects.
from the World (line 3) and then sort them based on their distance to the robot (lines 4-5).

```python
world = World()
robot = Nao()
people = Query(world).select_type(Person)
closest_person = people.sort_decreasing(
    lambda p: p.distance_to(robot)).take(1)

rate = rospy.Rate(2)  # A ROS function, not part of the API
# Makes loop iterate 2 times a second
# on line 15

while not rospy.is_shutdown():
    result = closest_person.execute()
    if len(result) > 0:
        break
    rate.sleep()  # Makes loop iterate times a second

robot.gaze_and_wait(result[0])
ah1 = robot.say("Hello human")
ah2 = robot.gesture(Gesture.WaveLArm)
robot.wait(ah1, ah2)
```

Figure 6.32. Querying the world for objects.

A number of changes were made to the Query class due to usability issues identified in the previous usability study. In the previous API, queries were documented as being the class Query, represented by the class HriQuery and instantiated by calling the factory method query. Participants found this confusing because it is inconsistent to call a query a HriQuery, and create it via a function rather than a class constructor (Chapter 5.3 - Other usability issues). To remedy this, HriQuery was renamed to Query and the query function was removed, leaving the Query constructor as the proper method to instantiate it. This change is consistent with Ellis et al. (2007) who found that constructors are easier to use than the factory pattern.

In the previous study, over a third of participants found queries difficult to understand, because of issues with closeness of mapping and role expressiveness (5.3.2 - Other usability issues). In this iteration, many of the query functions were renamed to make them more explicit, with a closer resemblance to SQL, a widely used database query language. For example, of_type was renamed select_type and where renamed select_where. The sorting methods in the previous Query class were ambiguous, for instance, order_by sorts in an ascending order, but this cannot be understood from the function name. To make this more explicit, the sorting functions were renamed:
order_by to sort_increasing and order_by_descending to sort_decreasing. The words increasing and decreasing were favoured over ascending and descending because they are more common (Lin et al., 2012).

An overview of the updated Query class is given in the following paragraphs. All of the functions below are part of the class Query:

__init__(iterable)

Query constructor, use this to create a Query. It accepts any type of iterable as input e.g. a list, set or another Query.

select_type(classinfo)

Select the entities with types that match classinfo.

select_where(predicate)

Select the entities where their predicates return true. A predicate is an inline function that returns true or false, in Python, these are called lambda statements. For example, the following lambda statement returns whether a + b equal 4:

```python
func = lambda a, b: a + b == 4
func(2, 2)
>>> True
func(2, 1)
>>> False
```

A complete example of a query using a select_where is shown below; a query is defined to find all of the entities to the left of the robot:

```python
21 world = World()
22 robot = Nao()
23
24 Query(world).select_where(lambda e: e.left_of(robot))
```

When this query is executed, the select_where function iterates through all of the entities in the world instance. For each entity, it evaluates whether the entity is left of the robot. If the lambda statement returns true, the entity is added to the list of items to return.
sort_increasing(key=identity)

Sort the entities in an increasing fashion using a key. They key is a lambda, but in this case, the lambda must return a numerical value by which the entities can be sorted. A complete example is shown below, where the entities are sorted based on their distance to the robot. Returns an OrderedQuery, which has a few extra functions for sorting: then_increasing and then_decreasing.

```python
25  world = World()
26  robot = Nao()
27  Query(world).sort_increasing(lambda e: e.distance_to(robot))
```

sort_decreasing(key=identity)

Sort the entities in a decreasing fashion using a key. Returns an OrderedQuery, which has a few extra functions for sorting: then_increasing and then_decreasing.

take(n)

Takes the first n items from the query. In the previous API n was a default parameter, it was changed to remove the hidden dependency, which caused some confusion previously.

execute()

Executes a Query, returning the list of items.

### 6.3 User evaluation

The second version of the API was evaluated with a case study (Yin, 2013). Four fourth-year software engineering students, all taking an advanced human-computer interaction course, used the API for a period of two weeks.

In this study, the participants developed a social-interaction application for a Nao humanoid robot. The application is a robot chat application, where a robot is used as an interface to allow two people separated from each other to communicate. The person co-located with the robot speaks and listens to it, whilst the distant person sends and receives messages via an instant messaging interface.
Chapter 6 – Iteration 2: Social-interaction refactored

The rest of this section provides a description of the evaluation method (Section 6.3.1) and the results of the case study (Section 6.3.2).

6.3.1 Method

A case study design consists of a set of research questions, criteria to select the case and subjects, a data collection procedure, an analysis procedure and a validity procedure (Runeson & Höst, 2009). These are explained in the following paragraphs.

Research questions

The objective of the case study was to further explore the second research question of this thesis (below), which was initially explored in the previous chapter.

RQ1 What abstraction level is appropriate for programming social robot applications?
   RQ1.1 What are the trade-offs associated with using different abstraction levels to program robot social-interaction?

More specifically, the case study aims to discover how well the design decisions made to create the second social-interaction API address research question one of this thesis (above).

Case & participant selection

The case was selected to reflect a typical real world application and time spent with the API.

The participants were fourth-year software engineering students studying an advanced human-computer interaction course. It was a course requirement for the students to perform usability research during the course.

The participants had previous experience with ROS (Open Source Robotics Foundation, n.d.-a) during a previous engineering course. Whilst the participant’s background differs from the previous iterations participants, their previous experience with ROS enabled them to compare their experience of the second API with an existing robotics framework.

Data collection

Data was collected from reports each participant wrote about their experience with the API as well as a direct focus group interview of the participants at the conclusion of the
study. The participant’s reports were assessed as a part of their course work, but not by me. The interview questions were based on questions from both the CD Questionnaire (Blackwell & Green, 2000) and the questions Clarke (2004) asks participants when measuring API usability (Appendix C.2).

The data was transcribed and then analysed using a common qualitative data analysis technique (Runeson & Höst, 2009) based on grounded theory (Glaser & Strauss, 1967). The process starts by grouping the transcribed data into categories (open-coding) and finishes when higher level themes emerge from the data.

Validity procedures

The validity procedure involved a form of data triangulation where each participant wrote a report about their experience with the API; the researcher conducted a direct interview with participants; and the participant’s code and reports were checked to find out what parts of the API they actually used.

The case: creating a robot chat interface

The participants started by reading the API documentation (Appendix C.1) and brainstorming possible applications they could create.

The participants created a robot chat application where the Nao robot acts as a conversational interface between two distributed people, one co-located with the robot and another using an instant messaging interface. The person co-located with the robot can speak to it; the robot hears what was said and relays it to the instant messaging interface. When the person using the instant messaging interface sends a message, the message is received by the robot, who verbalises it. The robot also converts emoticons and special characters that it encounters into physical gestures, for example if the robot encounters a question mark “?” it puts its hand in the air. These are illustrated in Table 6.3.

Table 6.3. Robot chat gestures. Second image used with permission of Samar Pant (modified).
<table>
<thead>
<tr>
<th>Condition</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:)</td>
<td>Wave</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>Sigh</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>Raise hand up</td>
<td></td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>Hands on hips</td>
<td></td>
</tr>
<tr>
<td>mirin</td>
<td>Mirin is short for admiring. Nao points when it parses this text.</td>
<td></td>
</tr>
</tbody>
</table>

The rest of this section discusses the results of the case study.

### 6.3.2 Results

The results are presented in three different sections which correspond to the three major components of an API: the interface used to create an application (API), the environment it is used with (Environment) and the documentation that supports it (Documentation). *API* describes the usability of the API in the context of the Cognitive Dimensions of
Notations (Green & Petre, 1996) as well as the participants perceived expressiveness of the API. Environment discusses how the environment affected usability. And lastly, Documentation discusses how the documentation affected usability.

In general, the participants thought that the API was easy to understand and use, which helped them develop their robot chat interface application. This is illustrated by the general positive comments that each participant made about the APIs usability, which are detailed below:

P1. “the [API] was found to be highly effective and usable to program Naobots”
P2. “The API itself was a pleasure to use, and the organisation of the functionality was well thought out”
P3. “As developers, we found the API very easy to understand and simple to develop with. This was impressive to us as we were familiar with the ROS programming language, which is very verbose”
P4. “Overall the [API] is a very usable API and fulfils its goals of creating an API for programmers who want to program robots without having much prior experience and also without having to delve into low-level programming”

API

This section uses the Cognitive Dimensions of Notations (Green & Petre, 1996) to interpret the data from the reports and the focus group interview. The data indicates that a number of factors benefited the APIs usability, including its: high abstraction-level, good visibility & role expressiveness, consistent naming conventions and close mapping to social-interaction. The data also revealed factors which impaired usability, including: in some cases premature commitment is required when using the API and progressive evaluation can be hard when debugging because the programmer is suddenly exposed to lower level implementation details (low remote visibility).

Abstraction level & gradient. The participants observed that the API has a consistently high abstraction level, which is “definitely suitable for programmers who may not have previous experience in programming robots as it is very simple and highly abstracted” (P4). The participants suggested that the advantages of the high abstraction
level are that it abstracts away lower level implementation details (P2, P4) and provides a terse notation that reduces the amount of code they had to write (P1).

The API is abstraction-tolerant in terms of its abstraction gradient; it does not force programmers to define new classes and functions before they can start programming (Interview). However, as discussed in the next Chapter it is possible for programmers to create their own abstractions when they need them (Chapter 8).

In the interview, the participants were asked if there were particular parts of the API where being able to extend it would make programming their scenario easier. The participants thought that it would be easier to programmatically create new gestures by combining a standard set of atomic gesture primitives together, rather than using an animation editor such as Choregraphe (Pot et al., 2009). For example, participant 2 said that “it seems like [Choregraphe is] slowing you down” (Interview).

**Closeness of mapping.** Much of the API has a close mapping to the social-interaction domain; this makes it easy for programmers to learn to use the API because they can use their existing domain knowledge to understand it. For example, participants 1 and 2 commented that the mapping between the API and the actions of the robot is very clear (P1, P2). This makes it easy to guess from the method names what their meaning is (Interview).

The exception is the **Query** class, which some participants thought had a more distant mapping to social-interaction than other parts of the API. This was revealed from the interview where participants were asked which parts of the API seem like a strange way of expressing something; participant 2 responded that “**Query** is less strong in terms of [closeness of mapping], you don’t query the world” This participant hadn’t used the **Query** class, but had read about it in the tutorial documentation and they thought that it would be fine for professional programmers but not for beginners. The other participants used it sparingly.

**Consistency.** The API has a consistent naming scheme, which has a positive effect on usability. For example, participant 2 commented that the API has “very intuitive naming conventions” which “has a very positive impact on the usability of the API.” The consistent naming scheme made it easy for the participants to “determine the effects of the methods before [using them]” (P1). For example, after learning that the **say** method makes the robot speak, it is easy to determine that the **say_and_wait** method makes the robot speak and wait until it has finished (P1, P2). It was also easy for the participants to
extrapolate how they were going to program other robots actions because all robot actions are methods within the robot class (Interview).

Whilst the API has a consistent naming scheme, the Query class is inconsistent with the other parts of the API because it has a much more distant mapping to the social-interaction domain. For example, participant 2 said that they “wouldn’t have guessed Query from learning the robot [part of the API]” (Interview).

**Diffuseness & terseness.** Participants commented that the high abstraction level makes the API terse, reducing the amount of code they had to write (P1). The API is terse in comparison with other robot programming APIs such as ROS, which would require much more code to accomplish the same goal. However, in the context of a high-level API, the participants believed the APIs terseness was just right. For instance, participants commented in the interview that the amount of code required to program a scenario was “about right, not much code at all – which is nice.”

**Premature commitment.** The participants didn’t notice any premature commitment at a granular level (P1, P4, Interview) which meant that they could start using the API with little effort. For example, P1 said that “the user is able to complete entire actions with minimal effort, and, therefore, minimal decisions made with much greater consequences”.

However, the API requires premature commitment when used at a finely grained level, in particular: robot action parameters cannot be modified once the action has begun (P1). Instead, the old action has to be stopped and a new action with new parameter values has to be started. This is a form of premature commitment and could be problematic in some cases, e.g. when dynamically changing the speed of an action.

**Progressive evaluation.** The participants found it hard to progressively evaluate their code because the APIs high abstraction level hides lower level functions that need to be examined when debugging (P1). Green & Petre (1996) term this low remote visibility which is a consequence of abstracting lower-level functions into classes and methods. This is a trade-off that occurs when raising the abstraction level of an API. When the participants were debugging their code, they ran into problems where the lower level functions “have either not begun or have executed completely” making it hard to understand the current state of the robot (P2). The participants were also not helped by less than informative error messages when executing actions (P1, Interview). The
participants had problems understanding the underlying implementation, which was made worse because it was not well documented (P1, P2).

Role expressiveness. The analysis showed that it was easy for participants to see the relationships between various parts of the API. For instance, participant 4 commented that “the relationship between the methods within the API are very obvious”. Green & Petre (1996) term this high local visibility, which arises when the abstraction level of a notation is raised. Additionally, in the interview participants said that it was “super simple” to tell what each section of code does.

Visibility & juxtaposability. The high abstraction level of the API leads to a trade-off between high local visibility (that helps role expressiveness) and low remote visibility (that hinders the ability to progressively evaluate).

Expressiveness. Participants also thought that they could have more control over social-interaction in some areas, for example, the ability to make the robot pause, or change its volume while it is speaking would have been helpful (P1). These are unimplemented features of the taxonomy of social primitives presented in Chapter 7 and could be added to the API relatively easily.

Environment

All four of the participants thought that starting the environment was the most difficult task required to use the API. This is illustrated by the comments that each participant made about how difficult it was to start the environment, some of these are detailed below:

P1. “the process of setting up the development environment to actually use the API was a very long and difficult process”
P2. “Setting up the associated technologies was relatively difficult, and we required technical support from [the researcher] at several stages”
P3. “setting up with environment was a difficult task as many different ROS commands needed to be launched in different terminal windows in order for the Nao Robot to work”
P4. “there [are] a lot of extra programs which need to be open and running before being able to run programs on the NAO Robot”

The participants found the environment difficult to start because a number of different commands had to be run from terminal windows to start supporting services (P1, P2, P3,
and P4). These commands start API services, low-level actuation drivers, sensor drivers and perception algorithms. This was mirrored in the interview, where participants said that a simpler way of setting up the environment would make things easier.

In addition to this, some of the services are unreliable and crash on start-up. The error messages given are “often non-informative and … it is highly infeasible to determine the source of these errors and a suitable fix…” (P1).

Documentation

All of the participants commented that the tutorial documentation (Appendix C.1) was good at teaching them how to use the API. The tutorial documentation explained the API in the context of examples, much like Chapters 5.1 and 6.1. Participant 2 explained that the tutorial documentation was “particularly useful for understanding the interactions with the interface, and getting started from scratch.”

However, the participants commented that the lack of traditional class and method documentation (P1, P2 and P3) hurt the usability of the API; especially when they needed to explore the APIs features in more depth. For instance, participant 2 commented that “there was not a categorised overview of the inputs and outputs of all relevant methods and interactions available” which made it difficult to “implement gestures, for which there was a single example detailing a single kind of gesture.”

6.4 Summary

This chapter presented a revised version of the social-interaction API originally presented in Chapter 5. The development of this API had three objectives: refactor the APIs abstraction level to provide more control of social-interaction, support multiple robots and address the usability issues of the previous API.

The refactoring of the APIs abstraction level gave it a number of new primitives. First, the API now has individual domain-specific action primitives, which give programmers the ability to control robot speech, gaze, gestures and expressions. Second, the API was given more action control primitives, which enable programmers to: wait for actions to complete, stop actions and start actions simultaneously or consecutively. Third, programmers can now subscribe to action callbacks, enabling them to receive domain-specific feedback messages for each action type. Lastly, programmers can now subscribe
to communication callbacks, which inform them of the state of communication made toward the robot (e.g. what it heard).

To enable the API to be abstract enough to program multiple robot platforms, support for two more robots was added (Zeno and Zoidstein) as well as maintaining support for Nao. Additionally, a new scenario, a behaviour tree based social-interaction application was used as an exemplar use case during the design and development of the API.

The chapter finished with a case study (Yin, 2013) based evaluation of the second API iteration. The case study results suggest that the APIs social primitives have an appropriate abstraction level for programming socially interactive robots (RQ1). The primitives have both positive and negative effects on usability. The positive effects include a close mapping to the social-interaction domain; a high abstraction level, which hides lower level implementation details; a terse notation, which means that problems can be expressed succinctly; and a good role-expressiveness. The main negative effect is that progressive evaluation is difficult due to low remote visibility. These findings are discussed in more detail in the discussion (Chapter 9).

The next chapter presents the taxonomy of social primitives; which emerged through the design, development and evaluation of robot programming APIs throughout this thesis. The taxonomy explores in more detail, from an implementation independent perspective, what primitives are required for programming social-interaction at an appropriate abstraction level (RQ3).
The previous two chapters presented APIs for programming socially interactive robots. Chapter 5 presented an API with high-level, domain-specific primitives designed to make it easy to program the social-interaction of a Nao humanoid robot. This API contains primitives which: make Nao speak, gaze and gesture in synchrony; discover if people said particular phrases; represent perception data as high-level objects; and query this perception data. Further development of this API was presented in Chapter 6, which discussed how it was refactored to provide finer control of social-interaction and the ability to program multiple robot platforms, including Nao (Aldebaran - SoftBank Group, n.d.), Zeno and Zoidstein (Chapter 6).

This chapter builds on the previous two chapters by exploring in detail what primitives are needed for programming social-interaction at an appropriate abstraction level (RQ3). This is accomplished by deriving an implementation independent taxonomy of primitives at the social primitive abstraction level. The taxonomy should inform robot social-interaction stakeholders of important primitives involved in creating social-interaction. These stakeholders include social-robot programming tool designers, algorithm developers, middleware designers and robot manufacturers.

The chapter starts by discussing the methodology used to derive the taxonomy of robot social-interaction (Section 7.1). This follows with the description of the taxonomy and appropriate evidence for each primitive (Section 7.2).

7.1 Method

The taxonomy was derived over the course of this thesis. It was initially derived as a taxonomy of the primitives used to program healthcare robot behaviour, presented in
(Diprose et al., 2012). It was then refined throughout the iterations of API design and evaluation described in Chapters 5 & 6 of this thesis. The taxonomy is a synthesis of people’s descriptions of social-interaction from the literature; the data gained through the design and evaluation of the APIs in Chapters 5 & 6; existing robot social-interaction programming interfaces; robot social-interaction literature; and more general social-interaction literature.

This section first describes how the data for the taxonomy was gathered (Section 7.1.1) and an overview of the procedure used to analyse the data (Section 7.1.2).

7.1.1 Data

The data supporting the final taxonomy originates from several areas, including:

- The data gained through designing and evaluating the APIs presented in Chapters 5 & 6 of this thesis.
- Existing robot social-interaction programming interfaces (Glas et al., 2012; Kopp et al., 2006; Lourens & Barakova, 2011; Pot et al., 2009)
- Robot social-interaction literature (Fong et al., 2003; Li et al., 2009)
- Human and animal social-interaction literature which define the types of gestures humans use and features of human speech (Doupe & Kuhl, 1999; Liebal & Pika, 2005; McNeill, 1992; Sowden et al., 2013)

The most resource intensive stage of data gathering was finding descriptions of robot social-interaction, which was gathered by systematically searching articles where researchers’ describe healthcare robot behaviour. Whilst the descriptions of robot behaviour originated from the healthcare domain, most of the robots from the study were social robots so the data and analysis were still useful for creating a taxonomy of social-interaction.

The systematic literature search identified 163 healthcare robots, 17 of these robots were selected based on theoretical sampling, i.e. where cases are selected based on their ability to inform the problem (Glaser & Strauss, 1967). This was later reduced to 16 robots, represented by seventeen articles when the original data was narrowed for the
social-interaction domain. The selected robots are listed in Table 7.1, with a subset illustrated in Figure 7.1.

Sentences that discussed robot social-interaction were extracted from the articles and placed in a spreadsheet along with the page number and paper from which the description originated. This is so that they could be used later during the analysis stage. The raw data is available here: (James Diprose, 2012).

### Table 7.1. Social robots.

<table>
<thead>
<tr>
<th>Robot</th>
<th>Reports Analysed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIBO</td>
<td>(Robins, Dautenhahn, Nehaniv, et al., 2005)</td>
</tr>
<tr>
<td>Actroid-F</td>
<td>(Yoshikawa, Matsumoto, Sumitani, &amp; Ishiguro, 2011)</td>
</tr>
<tr>
<td>ASIMO</td>
<td>(Sakagami et al., 2002)</td>
</tr>
<tr>
<td>Charlie</td>
<td>(Jayawardena et al., 2010)</td>
</tr>
<tr>
<td>COCO</td>
<td>(Racky et al., 2011)</td>
</tr>
<tr>
<td>Cody</td>
<td>(Chen, King, Thomaz, &amp; Kemp, 2011)</td>
</tr>
<tr>
<td>Huggable</td>
<td>(Stiehl et al., 2006)</td>
</tr>
<tr>
<td>Kaspar</td>
<td>(Amirabdollahian, Robins, Dautenhahn, &amp; Ji, 2011; Dautenhahn et al., 2009)</td>
</tr>
<tr>
<td>Keepon</td>
<td>(Kozima et al., 2007)</td>
</tr>
<tr>
<td>Nabaztag</td>
<td>(Klamer &amp; Ben Allouch, 2010)</td>
</tr>
<tr>
<td>Nao</td>
<td>(Shamsuddin et al., 2012)</td>
</tr>
<tr>
<td>Paro</td>
<td>(Wada, Shibata, Musha, &amp; Kimura, 2005)</td>
</tr>
<tr>
<td>RIBA</td>
<td>(Mukai et al., 2010)</td>
</tr>
<tr>
<td>Robot0a</td>
<td>(Robins, Dautenhahn, Te Boekhorst, &amp; Billard, 2005)</td>
</tr>
<tr>
<td>Twendy-One</td>
<td>(Iwata &amp; Sugano, 2009)</td>
</tr>
<tr>
<td>Zeno</td>
<td>(Ranatunga, Rajruangrabin, Popa, &amp; Makedon, 2011)</td>
</tr>
</tbody>
</table>

Figure 7.1: Example robots: (a) AIBO, (b) Charlie, (c) ASIMO, (d) Keepon, (e) Nabaztag, (f) Nao, (g) Paro and (h) Zeno. Images cropped, sources: AIBO - (Jordan, 2012), Charlie – used with permission of Healthbots, ASIMO - (S. Sowden, 2005), Keepon -(Keysmaker, 2007), Nabaztag - (Eussen, 2007), Nao – used with permission of Samar Pant, Paro – used with permission of Healthbots and Zeno - (Jurvetson, 2008).

### 7.1.2 Analysis

The data was analysed using a methodology inspired by grounded theory (Glaser & Strauss, 1967; Wolfswinkel, Furtmueller, & Wilderom, 2013). With grounded theory the
researcher builds up a theory by iteratively identifying concepts and relationships that fit the underlying data. Eventually, a conceptualisation of the data emerges.

Research questions

The research question addressed by the data analysis is the third research question of this thesis: “what primitives are needed to program robot social-interaction at the appropriate abstraction level?” The goal of the third research question is to derive an implementation independent taxonomy of primitives at an appropriate abstraction level for programming socially interactive robots. These primitives are at the social primitive abstraction level, which as discussed in Chapter 3.2.2, have an abstraction level above hardware primitives and algorithm primitives, but lower than emergent primitives.

Analysis

The analysis followed this general pattern:

1. The data was open coded.
2. The codes were axially coded.

I began by open coding data (Glaser & Strauss, 1967). Open coding is where a word, line or sentence is conceptually summarised by a keyword. As more and more data is coded, a bigger picture emerges that is grounded in the source data. This was accomplished by grouping data with similar meanings together and representing them with a single code. This is an iterative process of constant refinement that continues until the codes and supporting data make sense. A snapshot of the open coding process is shown in Figure 7.2a. In later stages codes were transferred to a spreadsheet.

The last stage involved axially coding the data (Glaser & Strauss, 1967). Axial coding is where relationships are formed between the codes that were created during the open coding stage. This was done by iteratively regrouping the codes given to each group of sentences to form categories, subcategories and other types of relationships. A snapshot of the axial coding process is shown is shown in Figure 7.2b. In later stages codes were grouped together in a spreadsheet.
Over and under generalisation has to be considered during this process. I focused on analysing uniquely social behaviours as opposed to those that could be produced by a generic software system. For instance, a robot copying a person’s movements was considered uniquely robotic, however, a robot measuring blood pressure and displaying a video was not. This is neatly summarised by the following analogy: imagine you are programming social-interaction of a person - what critical concepts and relationships would you need in order to program them? The next section describes the taxonomy.

7.2 The taxonomy

I aimed to create the taxonomy’s primitives at an abstraction level appropriate for programming robot social-interaction. During the second API iteration, it became clear that this should be the social primitive abstraction level. This is the abstraction level the second API was iteratively refined to, to increase reusability whilst maintaining usability. Social primitives have a higher abstraction level than hardware primitives or algorithm primitives, but a lower abstraction level than emergent primitives. The aim is to give them a closer mapping to the social-interaction domain than hardware primitives or algorithm primitives, without abstracting them too far to ensure that they remain reusable across a wide range of use cases. Three important categories emerged from the qualitative analysis of the data: Object, Person and Robot. The following subsections describe the categories in more detail. Explanations and examples of actions and perceptions are given next, along with supporting data.

7.2.1 Object

Object is the most general category that encompasses all other categories in a hierarchical fashion through inheritance. Examples of Objects and their inheritance
relationships are illustrated in Figure 7.3. The two most important types of Object are Person and Robot, which are explained in the next two subsections. Derived Objects can also have composition relationships with other objects, for example in Figure 7.3, Robot is composed of a Head and a LeftHand.

![Diagram of object relationships](image)

**Figure 7.3.** Examples of objects and their relationships: arrows are inheritance relationships, dotted lines are composition relationships. Objects with composition relationships also inherit from Object; this is not shown visually for brevity.

### 7.2.2 Person

Person is the main Object that a social robot interacts with. Hence, a Robot has many specialised primitives for interacting socially with a Person. The taxonomy assumes that a robot can interact with one or more Person instances.

### 7.2.3 Robot

A Robot is a special type of Object that represents the robot we are programming to interact socially. A Robot has the ability to both perform actions and perceive the world. The robots actions and perceptions are illustrated in Table 7.2 and Table 7.3. The actions and perceptions are grouped by primitive (e.g. making speech) and each primitive contains a number of features that are required to create social-interaction with that primitive.

#### Actions

Social robots communicate with people using five main actions: speech, gaze, gestures, facial expressions and touch (Table 7.2). Each action has a number of features that are required to be implemented for the full potential of the action to be realised, these are defined in Table 7.2. Gestures (body movements) and facial expressions (face
movements) could have been merged into a single category, however, gestures have an extra feature – “who to gesture at” so it was decided to keep them separate. The four actions and the reasoning for their features are explained below.

### Table 7.2. Robot social actions, see Appendix D for more supporting data

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Feature</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speaking</td>
<td>What to say?</td>
<td>“Hello human”</td>
</tr>
<tr>
<td></td>
<td>How to say it?</td>
<td>Volume, pitch, tempo, rhythm</td>
</tr>
<tr>
<td></td>
<td>Synchronise mouth with speech</td>
<td>Make mouth move in time with phonemes</td>
</tr>
<tr>
<td>Gazing</td>
<td>Who to gaze at?</td>
<td>Own body part, person’s body part or another object</td>
</tr>
<tr>
<td></td>
<td>What body parts to use?</td>
<td>Eyes, head, eyes and head, eyes, head and neck etc</td>
</tr>
<tr>
<td></td>
<td>How to gaze?</td>
<td>Speed, acceleration, deceleration</td>
</tr>
<tr>
<td></td>
<td>Blend with gestures</td>
<td>Nod and gaze simultaneously</td>
</tr>
<tr>
<td>Gesturing</td>
<td>Who to gesture at?</td>
<td>Own body part, person’s body part or object</td>
</tr>
<tr>
<td></td>
<td>What to gesture?</td>
<td>Wave, poke at, arm raise, lip touch, peek-a-boo</td>
</tr>
<tr>
<td></td>
<td>How to gesture?</td>
<td>Magnitude, speed, repeat, duration</td>
</tr>
<tr>
<td></td>
<td>Perform multiple gestures simultaneously</td>
<td>Shake head and point simultaneously</td>
</tr>
<tr>
<td></td>
<td>Blend with other gestures</td>
<td>Blend wave and point on same arm</td>
</tr>
<tr>
<td>Performing facial expressions</td>
<td>What to express?</td>
<td>Surprised, happy, sad, smile, anger, frown, blinking, neutral</td>
</tr>
<tr>
<td></td>
<td>How to express?</td>
<td>Magnitude, speed, repeat</td>
</tr>
<tr>
<td></td>
<td>Perform multiple facial expressions simultaneously</td>
<td>Frown and smile simultaneously</td>
</tr>
<tr>
<td></td>
<td>Blend with other facial expressions</td>
<td>Blend pucker and smile expressions</td>
</tr>
<tr>
<td>Touching</td>
<td>Who / what to touch?</td>
<td>Robots body part, a person’s body part</td>
</tr>
<tr>
<td></td>
<td>What type of touch?</td>
<td>Stoke, hit, pat, slide, scratch, kick etc</td>
</tr>
<tr>
<td></td>
<td>How to touch?</td>
<td>Pressure, duration</td>
</tr>
</tbody>
</table>

**Speaking.** The speaking action enables the robot to vocalise words in a particular order (Doupe & Kuhl, 1999, p. 571; Fong et al., 2003, p. 152). For example, the robot Cody was described as saying “I am going to rub your arm. I am going to clean you. The doctor will be with you shortly” (Chen et al., 2011, p. 460). The first feature is the ability to specify what the robot should say. The second feature is the ability to control how the robots vocalised words are being spoken, including the ability to control the volume (Fong et al., 2003, p. 152), pitch, rhythm and tempo (Doupe & Kuhl, 1999, p. 571; Fong et al., 2003, p. 152) of the robot’s voice. The third feature is the ability to synchronise the robots mouth with speech (Li, MacDonald, & Watson, 2009, p. 5012), which is especially important for robots with lifelike faces such as Zeno (Ranatunga et al., 2011).

**Gazing.** The gazing action enables the robot to direct its gaze at objects. For example, the robot Keepon was described as being “… capable of expressing attention by orienting
its gaze ...” (Kozima et al., 2007, p. 398). The first feature is the ability to control who the robot gazes at, for example, the robot Keepon was described as being able to change its gaze “between a child’s face, the caregiver’s face and sometimes a nearby toy” (p. 391). The second feature is the ability to control what body parts are used to gaze (mindmakers.org, n.d.), for example, the robots could use its eyes or head to gaze (Dautenhahn et al., 2009, p. 376; mindmakers.org, n.d.; Ranatunga et al., 2011, p. 1; Sakagami et al., 2002, p. 2480). The third feature is the ability to control how the gaze is performed, including the speed and acceleration of the gaze. The last feature is the ability to blend the gaze action with gestures, for example, a robot should be able to gaze at an object whilst performing a nodding gesture (explained next).

**Gesturing.** The gesturing action enables the robot to move a “part of the body, especially a hand or the head, to express an idea or meaning” (Oxford University Press, n.d.-b), for example, the robot Keepon responds to stimulus by “bobbing up and down” (Kozima et al., 2007, p. 398). The first feature is the ability to control to whom the robot gestures. This is based on an understanding of how primates, including humans, gesture. Primates use both dyadic and triadic gestures which involve the targeting of a gesture at an individual or an external object respectively (Liebal & Pika, 2005, p. 6). The second feature is the ability to control the type of gesture that is performed, for example, wave, poke at and arm raise (Dautenhahn et al., 2009, p. 377; Liebal & Pika, 2005, p. 6). The third feature is the ability to control how the gesture is performed, including its speed, magnitude and duration. The ability to control the duration of the gesture is particularly important so that speech and gesture can be synchronised over time (McNeill, 1992; H. Sowden, Clegg, & Perkins, 2013, p. 922). The fourth feature is the ability to perform multiple gestures simultaneously, for example, a robot should be able to shake its head and point simultaneously. The last feature enables spatially and temporally overlapping gestures to be blended together, for example, a waving and pointing gesture performed on the same arm could be blended together.

**Performing facial expressions.** The performing facial expression action enables a robot to perform facial expressions, for example, the “Actroid-F … can exhibit various facial expressions such as smile, anger and surprise.” (Yoshikawa et al., 2011, p. 2378). The first feature is the ability to control what type of facial expression is performed, examples include smiling, blinking, and surprise (Dautenhahn et al., 2009, pp. 376, 387; Yoshikawa et al., 2011, p. 2378). The second feature is the ability to control how the
facial expression is performed, including the facial expressions magnitude, speed, duration and rate. For instance, the magnitude and rate of the robot KASPARs blinking can be controlled (Dautenhahn et al., 2009, p. 376). The third feature enables multiple facial expressions to be performed simultaneously, for example, a robot should be able to smile and frown simultaneously. The last feature enables spatially and temporally overlapping facial expressions to be blended together, for example, a smile and pucker gesture could be blended together.

**Touching.** The touching action primitive gives a robot the ability to touch objects, for example, the robot Cody was described as having “autonomously reached out, touched the participant's arm, moved across their arm, and then retracted” (Chen et al., 2011, p. 457). The first feature of the touch action is the ability to control what to touch, for example, the “participants arm” in the above example (p. 457). The second feature is the ability to control the type of touch that is performed, for example, the robot Cody programmed to wipe a person’s arm (p. 457), but it could be programmed to perform other types of touch such as patting or scratching. The third feature is the ability to control how the touch is performed, including its pressure and duration (p. 459).

**Perceptions**

Social robots need to be able to perceive five types of information to socially interact with people, these include human features, speech, gestures, facial expressions and touch (Chen et al., 2011; Fong et al., 2003, pp. 155–156). Each perception primitive has a number of features that are required to be implemented for the full potential of the perception ability to be realised.

The robots perceptions are presented from an egocentric perspective (Klatzky, 1998, p. 2), however, they could also be described and implemented from an allocentric perspective (p. 2). For example, rather than stating that the robot can perceive a person’s gestures, the taxonomy could state that a Person is able to produce gestures and implicitly assume that a robot can sense them. The perception primitive’s and their features are defined in Table 7.3 and are explained below.

**People’s features.** A first part of interacting socially with people is the ability to perceive their physical features (Fong et al., 2003, p. 155). This includes a person’s body parts, such as their head, torso, arms and legs; and their facial features, such as their eyes, mouth and nose (p. 155). This is important because many social actions, including gazing,
gesturing and touching are dependent on the robot being able to perceive people’s features.

Table 7.3. Robot social perceptions, see Appendix D for more supporting data.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Feature</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>People’s features</td>
<td>Body parts</td>
<td>Head, torso, arms, legs etc</td>
</tr>
<tr>
<td></td>
<td>Facial features</td>
<td>Eyes, mouth, forehead, chin etc</td>
</tr>
<tr>
<td>People’s speech</td>
<td>Who spoke?</td>
<td>A specific person</td>
</tr>
<tr>
<td></td>
<td>Who were they speaking to?</td>
<td>The robot or a specific person</td>
</tr>
<tr>
<td></td>
<td>What did they say?</td>
<td>“Hello robot”</td>
</tr>
<tr>
<td></td>
<td>When did they say it?</td>
<td>Onset time, end time, duration</td>
</tr>
<tr>
<td></td>
<td>How was it said?</td>
<td>Volume, pitch, tempo, rhythm</td>
</tr>
<tr>
<td>People’s gaze</td>
<td>Who is gazing?</td>
<td>A specific person</td>
</tr>
<tr>
<td></td>
<td>What are they looking at?</td>
<td>The robot, a specific person or an object</td>
</tr>
<tr>
<td></td>
<td>When were they gazing?</td>
<td>Onset time, end time</td>
</tr>
<tr>
<td></td>
<td>How were they gazing?</td>
<td>Quickly, slowly, for a long time</td>
</tr>
<tr>
<td>People’s gestures</td>
<td>Who gestured?</td>
<td>A specific person</td>
</tr>
<tr>
<td></td>
<td>Who were they gesturing to?</td>
<td>The robot, or a specific person</td>
</tr>
<tr>
<td></td>
<td>What type of gesture was it?</td>
<td>Poke at, arm raise, lip touch, handshake, hand circling etc</td>
</tr>
<tr>
<td></td>
<td>When was it performed?</td>
<td>Onset time, end time</td>
</tr>
<tr>
<td></td>
<td>How was it performed?</td>
<td>Magnitude, speed, acceleration, deceleration, duration</td>
</tr>
<tr>
<td>People’s facial expressions</td>
<td>Who expressed it?</td>
<td>A specific person</td>
</tr>
<tr>
<td></td>
<td>Who were they expressing it to?</td>
<td>The robot or another person</td>
</tr>
<tr>
<td></td>
<td>What type of expression was it?</td>
<td>Smile, frown, pucker etc</td>
</tr>
<tr>
<td></td>
<td>When was it performed?</td>
<td>Onset time, end time</td>
</tr>
<tr>
<td></td>
<td>How was it performed?</td>
<td>Magnitude, speed, acceleration, deceleration, duration</td>
</tr>
<tr>
<td>People’s touch</td>
<td>Who touched me?</td>
<td>A specific person</td>
</tr>
<tr>
<td></td>
<td>What type of touch was it?</td>
<td>Stoke, hit, pat, slide, scratch, kick etc</td>
</tr>
<tr>
<td></td>
<td>Where was I touched?</td>
<td>On a particular body part, e.g. forehead, belly, back etc</td>
</tr>
<tr>
<td></td>
<td>When was I touched?</td>
<td>Onset time, end time</td>
</tr>
<tr>
<td></td>
<td>How was I touched?</td>
<td>Pressure, duration</td>
</tr>
</tbody>
</table>

**People’s speech.** The second perception primitive enables the robot to perceive human speech, for example, the robot Coco starts to play a quiz game when it hears “Quiz” (Racky et al., 2011, p. 1091) and ASIMO is able to understand natural language sentences (Sakagami et al., 2002, p. 2478). The first and second features enable the robot to understand who is involved in the sharing of information via speech; this includes both the ability to understand who is speaking (Fong et al., 2003, p. 156) and also who is being spoken to. The latter is important for reducing social faux pas caused by the robot thinking that a person is speaking to it when the person is really speaking to another person. This
occurs frequently enough for it to be a problem, such as when Kozima et al. describe children both talking “to Keepon” (Kozima et al., 2007, p. 394) and “other children” whilst they are in Keepon’s vicinity (p. 398). The third and fourth features include the abilities to understand what was spoken (Fong et al., 2003, p. 156) and when it was spoken. The last feature enables the robot to understand how the person vocalised the words (p. 156), which is important for understanding the emotional content of language (p 152). This feature includes the ability to understand each word’s volume, pitch, rhythm and tempo (Doupe & Kuhl, 1999, p. 576; Fong et al., 2003, p. 152).

**People’s gaze.** The third perception primitive enables the robot to perceive people's gaze, which is important for realising joint attention (Fong et al., 2003). The first feature is the ability to perceive what people are gazing at, which can include the robot or other objects (Robins, Dautenhahn, Nehaniv, et al., 2005, p. 112). The last two features include the ability to perceive when and how the gaze is being performed, including the onset time, speed and duration of the gaze.

**People’s gestures.** The fourth perception primitive enables the robot to perceive people's gestures. The first and second features enable the robot to perceive both who performed a gesture, and who they were performing it to; this is a basic part of perceiving dyadic and triadic gestures (Liebal & Pika, 2005). The third feature enables the robot to perceive the type of gesture being performed, examples include: waving, touching one's lip and raising an arm (Kozima et al., 2007, p. 394; Liebal & Pika, 2005, p. 6). The last two features enable the robot to perceive when and how the gesture was performed, including the onset time, magnitude, duration and speed of the gesture (Dautenhahn et al., 2009, p. 380).

**People’s facial expressions.** The fifth perception primitive enables the robot to perceive people's facial expressions. The first and second features enable the robot to perceive both who performed a facial expression, and who they were performing it to. This is illustrated by Kozima et al. who describe a situation where “S asked Keepon ‘Is this scary?’ showing bizarre facial expressions to the robot” (Kozima et al., 2007, p. 395). Keepon needs to understand (1) that S is performing the facial expression and (2) that the expression is being performed to Keepon and not someone else. The third feature enables the robot to perceive the type of expression performed, examples include smiling, frowning and puckering (Wada et al., 2005, p. 2). The fourth and fifth features enable the robot to perceive when and how the expression was performed; including the onset time,
magnitude, speed and duration of the expression (Dautenhahn et al., 2009, p. 380; Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112).

**People’s touch.** The *last* perception primitive enables the robot to perceive touch, a very common perception channel for social robots, including: Nao, Huggable, Keepon, Paro, Charlie and KASPAR (Amirabdollahian et al., 2011; Jayawardena et al., 2010; Kozima et al., 2007; Shamsuddin et al., 2012; Stiehl et al., 2006; Wada et al., 2005). The *first* feature of touch perception enables the robot to understand who is touching it, which is important when the robot responds or associates touch with a specific person. For instance, Kozima et al. describe situations where it was important that Keepon knew different children touched it such as when “S violently kicked Keepon” (Kozima et al., 2007, p. 395). The *second* feature enables the robot to perceive the type of touch being performed, examples include: stroking, patting and scratching (Jayawardena et al., 2010, p. 5991; Kozima et al., 2007, p. 397; Stiehl et al., 2006, p. 1290). The *third* feature enables the robot to perceive where on its body it was touched, for instance, Huggable has a “full body ‘sensitive skin’” which can localise touch to individual body parts (Stiehl et al., 2006, p. 1291) and Kozima et al. describe children touching different parts of Keepon’s body such has its head and belly (Kozima et al., 2007, pp. 397, 394). The *fourth* and *fifth* features enable the robot to perceive when and how it was touched; including sensitivity to onset time, pressure and duration (Amirabdollahian et al., 2011, p. 5350; Kozima et al., 2007, pp. 395, 398).

### 7.3 Summary

This chapter presented a taxonomy of primitives for programming social-interaction, which aims to address the third research question of this thesis: “What primitives are needed to program robot social-interaction at the appropriate abstraction level?” The taxonomy was derived throughout this thesis from relevant data with a technique based on grounded theory (Glaser & Strauss, 1967). The data included descriptions of robot behaviour, data gained through the design & evaluation of the APIs in Chapters 5 & 6, other social-interaction programming interfaces and robot, human and animal social-interaction literature. The taxonomy should inform robot social-interaction stakeholders of important primitives involved in creating social-interaction.

The next chapter describes the software architecture that supports the final social-interaction API.
The previous chapter presented an implementation independent model of the primitives needed to program robot social-interaction, at the social primitive abstraction level. The taxonomy was derived throughout this thesis using a methodology based on grounded theory (Glaser & Strauss, 1967) using descriptions of robot behaviour, data from the two iterations of API design and evaluation, other social-interaction programming interfaces and robot, human and animal social-interaction literature.

This chapter presents the software architecture designed to address research question four of this thesis (RQ4): “what is a suitable software architecture to support the API with primitives at the abstraction level appropriate for programming robot social-interaction? Iterative requirements engineering and development (Cao & Ramesh, 2008, pp. 63–64) was used to define the objectives and design goals of the software architecture. Cao & Ramesh (2008) argue that this is suitable when system requirements aren’t clear at the beginning of a software development project. The software architecture evolved as the first (Chapter 5) and second (Chapter 6) social-interaction APIs were developed, the current version supports the API discussed in Chapter 6. The objectives used to design the software architecture are discussed next:

O1. Provide mechanisms that enable the API to be extended to support a variety of use cases.

O2. Provide mechanisms that allow the API to be vendor agnostic at the social primitive abstraction level, enabling integration with a variety of social robot platforms.

There are three main reasons I focus on these objectives as opposed to other architectural objectives such as maintainability, reliability and performance. First, both
objectives were important for exploring the usability of primitive abstraction levels (RQ1) and primitive control mechanisms (RQ2). Objective 1 (O1) allowed the API to be extended to support the exemplar use cases used during the design of the APIs. Objective 2 (O2) enabled the usability of a variety of primitives to be explored because robot platforms with different capabilities could be integrated with the API. Second, objective 2 (O2) encourages reuse of existing algorithms because it means that direct mechanisms are available to integrate them with a high-level API. Third, both objectives make the API more useful for other developers by allowing them to support their own scenarios and robots.

The first objective (O1) is met by enabling users to extend the API through the definition of new entities, robots, gestures and expressions. Other robotics architectures also provide extensibility mechanisms, for example, BONSAI (Siepmann, 2013) provides a generic interface for reading sensor data, including both high and low-level data such as perceived people and laser scanner data respectively. This chapter’s architecture differs from BONSAI because it provides extensibility for common features required when programming social-interaction. This includes the ability to define custom object-oriented representations of entities, their relationships and linking them directly to perception data generated by robot middleware.

The second objective (O2) is met by defining social action & perception interfaces. These interfaces abstract away vendor specific details, enabling the API to communicate with robot platforms at a higher, social primitive abstraction level, e.g. an interface for controlling robot gestures. Existing robot middleware solutions such as ROS (Open Source Robotics Foundation, n.d.-a) typically provide vendor agnostic interfaces, however, these are at lower abstraction levels, e.g. a laser scanner interface from a hardware abstraction layer. The architecture presented in this chapter complements existing robot middleware solutions by providing interfaces at the social primitive level, rather than at hardware or algorithm primitive abstraction levels. These interfaces are termed ‘social action & perception interfaces’.

Other architectures also provide interfaces for social actions & perceptions, including BML (Kopp et al., 2006) and BONSAI (Siepmann, 2013). BML provides vendor-agnostic interfaces for sending social-interaction actions to embodied characters, including virtual characters and robots. However, BML is an XML definition, not an interface integrated with an execution engine. BONSAI provides interfaces that span a
number of abstraction levels. These include hardware primitive abstractions, e.g. LaserSensor; algorithm primitive abstractions, e.g. SlamSensor; and social primitive abstractions, e.g. SpeechActuator. The architecture in this chapter differs from BONSAI because it focusses solely on providing interfaces at the social primitive level and doesn’t duplicate functionality already provided by robot middleware (hardware abstraction).

This rest of this chapter starts by describing and explaining the design goals behind the APIs software architecture (Section 8.1). The chapter then moves onto describing how the API extensibility design goals were implemented (Section 8.2) and how the action & perception abstraction design goals were implemented (Section 8.3).

8.1 Design goals

A number of design goals were used to guide the design and development of the software architecture; they are separated into two categories based on the two architectural objectives: those related to enabling the API to be extended (O1) and those related to enabling the API to be vendor agnostic at the social-interaction abstraction level (O2). These design goals are discussed in more detail in the rest of this section; the API extensibility design goals in Section 8.1.1 and the social action & perception abstraction design goals in Section 8.1.2.

8.1.1 API extensibility design goals

The API extensibility design goals are discussed in the following paragraphs.

DG1. Provide a standard method to define entities and the relationships between them

Those extending the API need a standard way to define new entities and the relationships between them. This is to enable the small number of core social-interaction specific entities to be defined (Robot and Person) as well as giving users the ability to extend the API by defining their own entities. Common relationships that need a standard way to be defined include inheritance and composition relationships. For example, a person is a subtype of an entity (inheritance) and is also composed of a head, arms, torso and legs (composition).
To enable the API to be aware of the position of entities, programmers need a standard way to link the entities they define to perception data related to those entities, such as the position of an entity. For example, the Person entity and its body parts are linked to perception data detailing the position and orientation of each of its body parts. This mechanism can be used for any Entity.

**DG2. Provide a standard way to define new robots and their body parts**

When support for a robot is added to the API, it needs to be symbolically represented so that users can control it. Robots also have different body structures which need to be symbolically defined for each robot. To enable the API to be aware of the position of robots and their body parts, programmers need a standard way to link the classes representing a robot’s body parts to perception data sent by localisation algorithms and joint position drivers.

**DG3. Provide a standard method of defining new gestures, expressions and their default parameters**

The gestures and expressions that a robot can perform are typically authored by people, e.g. animators or API users; these need to be symbolically represented in the API so that interaction programmers can make use of them. Additionally, the default parameters of each gesture and expression need to be specified (e.g. duration, speed and intensity) to save the interaction programmer from having to specify them every time.

### 8.1.2 Social action & perception design goals

The social action & perception design goals are discussed in the following paragraphs.

**DG1. Provide social action interfaces**

Social action interfaces allow robot actions to be controlled at a high, social-interaction abstraction level. This provides vendor agnosticism because the API can interface with robot platforms at the social-interaction abstraction level, rather than with robot specific algorithms. For example, when a robot.gesture(Gesture.Wave) API call is made, a message is sent through the social action interface. The robot Nao would receive the message and then gesture using its NAOqi (Aldebaran Robotics, n.d.) platform; whereas the robot Zeno would receive the same message and execute the gesture with its blender based animation controller. Thus, social action interfaces provide a common, high-level
means of for the API to interface with the actual algorithms that make a robot perform a particular action.

DG2. Provide default social action interface implementations

Default implementations of the social action interfaces should be provided to save developers from having to implement common action interface communication code every time they implement the social action interface for a particular algorithm.

DG3. Provide social perception interface

The social perception interface allows robot platforms to send high-level perception data to the API in a vendor agnostic manner, including the type, position and orientation of different objects. This data is then represented by the API as high-level objects. For example, to perceive people, Nao uses a face tracker from the NAOqi framework (Aldebaran Robotics, n.d.), whilst Zeno its own open source face tracker. Despite the platforms using different perception algorithms, they both send their perception data through the same social perception interface to the API.

DG4. Provide a method to configure the perception system

Developers need the ability to configure the perception system so that they can choose what perception algorithms are used on a particular robot platform. In particular, they need to be able to specify what perception algorithm is used to track a particular type of entity.

8.2 API extensibility implementation

The architecture was implemented with the ROS (Open Source Robotics Foundation, n.d.-a) and the Python programming language (Python Foundation, n.d.). The software architecture enables several components of the API to be customised, including the definition of new entities, robots, gestures and expressions. Examples of the customisable aspects of the API are coloured orange 5 in Figure 8.1 and are explained in more detail in Sections 8.2.1 to 8.2.3.

5 White if printed in black and white.
The rest of this section explains how the extensibility features of the software architecture were implemented.

8.2.1 DG1: Provide a standard method to define entities and the relationships between them

New entities are defined by inheriting from the Entity class. For example, the two default entities provided by the API - the Robot and Person classes, both inherit from Entity (Figure 8.2).

Hierarchical entities are composed by creating a top-level class to represent the entity (e.g. Person), classes to represent their children (e.g. a Head and Torso) and assigning the children to attributes of the parent class.

The coordinates of entities are accessed via the ROS Transform Library (Foote, 2013), which stores coordinates in a tree structure and identifies them with a global string...
identifier called a frame id. The coordinate frame values are populated by individual perception algorithms (Section 8.3.3). A simplified visualisation of the coordinate frames for a robot and a person are illustrated in Figure 8.3. The robots coordinate frames include odom, torso, head_yaw_link and camera_frame; whilst the persons coordinate frames include person1_head, person1_torso, person1_left_hand and person1_right_hand.

Each Entity contains a frame id attribute which is used to retrieve the entities coordinate from the ROS Transform Library (Foote, 2013). Frame ids of entities in a hierarchy are local to the entity they represent; the global frame id that uniquely identifies an entities coordinate is retrieved by concatenating the local frame ids of all its parent entities together. For example, to calculate the global frame id of a person’s head, the local frame ids of the Person instance (‘person1’) and the Head instance (‘head’) are concatenated to give ‘person1_head’. This uniquely identifies the coordinate of this particular person’s head in the ROS Transform Library (Foote, 2013).

![Figure 8.2. Defining entities.](image-url)
An example that illustrates how to define a new hierarchical entity is shown in Figure 8.4. To create a new hierarchical entity, define a top-level class to represent the entity and also define classes to represent the entity’s children. Each class should inherit the `Entity` class (e.g. `Person` line 1, `Head` line 14, `Torso` line 18). Then within each class’s constructor, call the `Entity` class’s constructor from within (lines 2-3, 15-16 and 19-20).

```python
1  class Person(Entity):
2      def __init__(self, local_id):
3          Entity.__init__(self, 'person' + str(local_id))
4          self.head = Head(self)
5          self.torso = Torso(self)
6
7      @classmethod
8      def make(cls, local_id):
9          return Person(local_id)
10
11  def default_child(self):
12      return self.head

14  class Head(Entity):
15      def __init__(self, parent):
16          Entity.__init__(self, 'head', parent)

18  class Torso(Entity):
19      def __init__(self, parent):
20          Entity.__init__(self, 'torso', parent)
```

Figure 8.4. Defining a hierarchical entity.
The **Entity** classes’ constructor is explained below:

```python
__init__(local_frame_id, parent=None,
frame_scheme=FrameScheme.FlattenedHierarchy)
```

The `local_frame_id` parameter is where the frame id local to the current entity is specified. The `parent` parameter is `None` by default and is used to tell the entity if it has a parent instance or not. For example, the `Person` instance doesn’t have a parent, however, it is the parent of the `Head` and `Torso` instances (lines 4-5). Lastly, the parameter `frame_scheme` specifies the naming scheme used to calculate global frame ids, which is by default a flattened hierarchical scheme as discussed earlier.

The **Entity** class has two unimplemented methods that need to be overridden if the **Entity** is at the root of a hierarchy, e.g. the **Person** class. These are explained below:

```python
make(local_id)
```

This method is used by the perception system to create an **Entity** instance from perception data. The `local_id` parameter is an integer that uniquely identifies an entity from all entities of the same type, e.g. all **Person** instances. An example is shown in Figure 8.4 for the **Person** class (lines 7-9).

```python
default_child()
```

The root entities of hierarchies often don’t have a coordinate frame, for example, a tracked person has coordinate frames for their body parts, but not for the top level concept of a person (Figure 8.3). This method returns a child instance to represent the root entity in a hierarchy during spatial calculations. For example, to calculate the distance between a robot and a person using the function `robot.distance_to(person)`, children of robot and person need to be used as frames of reference for the calculation. In Figure 8.4 we use the frame id of the persons head (line 11-12).

### 8.2.2 DG2: Provide a standard way to define new robots and their body parts

New robots are defined in much the same way as other entities are, with a couple of differences. The top level class which represents the new robot type should inherit the **Robot** class instead of **Entity** (Figure 8.5); example code is illustrated in Figure 8.7.
The robots body parts should be represented by creating classes that inherit `Entity` (lines 12-15 and 17-20), and assigning their instances to the top level class (lines 6-7).

The coordinates of the robots body parts are retrieved using the ROS Transform Library in the same way that the coordinates of other entities are retrieved. However, robot body part coordinates are broadcast into the ROS Transform Library using a set of standard ROS nodes provided by the robots ROS package. The nodes and their relationships are illustrated in Figure 8.6.

![Figure 8.5. Architecture of defining new robots.](image-url)
Chapter 8 – Software architecture

Figure 8.6. The ROS robot state publisher architecture. See these references for more details:
(Sucan, n.d.-a), (Sucan, n.d.-b) and (Foote, 2013).

Additionally, robot frame ids use a different naming scheme than other entities. Each
robot body parts local_frame_id corresponds directly to the robot's frame ids in the
ROS Transform Library. For example, the local_frame_id values specified in lines
13 and 18 of the example code (Figure 8.7) directly map to the frame ids shown in the
ROS Transform visualisation for the Nao robot (Figure 8.3). The direct mapping naming
scheme is specified by setting the frame_scheme parameter in the Entity constructor
to the Scheme.DirectMapping enumeration member (lines 14 and 19).

```python
class Nao(Robot):
def __init__(self):
    Robot.__init__(self)
    self.head = Head(self)
    self.torso = Torso(self)
def default_child(self):
    return self.torso

class Head(Entity):
def __init__(self, parent):
    Entity.__init__(self, 'gaze', parent, 
                   frame_scheme=Scheme.DirectMapping)

class Torso(Entity):
def __init__(self, parent):
    Entity.__init__(self, 'torso', parent, 
                   frame_scheme=Scheme.DirectMapping)
```

Figure 8.7. Code to define a new robot.
8.2.3 DG3: Provide a standard method of defining new gestures, expressions and their default parameters

New gestures and expressions are defined by deriving the IGesture and IExpression classes respectively (Figure 8.8). IGesture and IExpression are Python enumerations (Python Software Foundation, n.d.) and have fields for specifying the default values of each gesture and expression enumeration member.

An example from the Nao robot about how to create a new set of gestures is shown in Figure 8.9. The Gesture class inherits IGesture; members for the different types of gestures are defined within. The default parameters for each enumeration member are specified in a tuple, which are unpacked and passed to the IGesture constructor. To
make a gesture run forever, or until a stop command is issued, the **default_duration** should be set to positive infinity (line 9).

The expressions for a robot are defined in much the same way as gestures; however they have more default parameters than gestures (Figure 8.10). As well as being able to specify a **default_duration**, each expression member also has a **default_intensity** and a **default_speed**. An example of how to define a set of expressions is shown in Figure 8.10. To make an expression run forever, or until a stop command is issued, the **default_duration** should be set to positive infinity (line lines 10-13)

```python
1 class IGesture(Enum):
2     def __init__(self, default_duration):
3         self.default_duration = default_duration

4 class Gesture(IGesture):
5     LowerLArm = (2.0)
6     WaveLArm = (6.0)
7     PointLArm = (float('inf'))

Figure 8.9. Defining new gestures.
```

```python
1 class IExpression(Enum):
2     def __init__(self, default_duration, \  
3         default_speed, default_intensity):
4         self.default_duration = default_duration
5         self.default_intensity = default_intensity
6         self.default_speed = default_speed

7 class Expression(IExpression):
8     Smile = (float('inf'), 0.8, 0.5)
9     FrownMouth = (float('inf'), 0.8, 0.5)
10    FrownForehead = (float('inf'), 0.8, 0.5)
11    OpenMouth = (float('inf'), 0.8, 0.5)

Figure 8.10. Defining new expressions.
```

### 8.3 Social action & perception interface implementation

Social action and perception abstraction enables the API to interface with a variety of robot platforms at the social primitive abstraction level. The action interfaces enable the
The perception interface facilitates the conversion of algorithmic perception data into high-level objects. This also has the potential to allow a large number of existing perception algorithms to be reused and integrated with the social-interaction API.

The social action interfaces (Figure 8.11) are implemented with the ROS actionlib API (Marder-Eppstein & Pradeep, n.d.). Commands are received by vendor action drivers, which execute the actions. Vendor action drivers also inform the API about the current state of actions by sending feedback messages back to the API via the action interfaces. The say, gaze, gesture and expression action primitives each have a customised social action interface, discussed in more detail in Sections 8.3.1 and 8.3.2.

The social perception interface is composed of two systems (Figure 8.11); the Perception Synthesiser and the ROS Transform Library (Foote, 2013). The Perception Synthesiser instructs the World instance to create high-level objects to represent perception data, whilst the ROS Transform Library stores the coordinates of tracked entities.

The rest of this section defines the social action & perception interfaces and explains how they were implemented.
8.3.1 DG1: Social action interfaces

Social action interfaces (Figure 8.12) are used by the API to communicate with the drivers which perform robot actions (action drivers). The API provides four action interfaces, which are used to: synthesise speech, target robot body parts at entities, perform gestures and show expressions. The action interfaces are implemented with the ROS actionlib API (Marder-Eppstein & Pradeep, n.d.), which was chosen because it is designed to command long running tasks that may need to be pre-empted and that require feedback (Marder-Eppstein & Pradeep, n.d.).

ROS actionlib definitions are split into three parts: the definition of the goal, the definition of the result sent back once the goal is completed (or pre-empted) and the definition of the feedback message sent whilst the goal is being performed (Marder-Eppstein & Pradeep, n.d.). The following paragraphs describe the four ROS actionlib definitions.

![Figure 8.12. Action interface.](image)

The Say action interface is used to synthesise speech (Figure 8.13). The goal definition contains one variable, text, which denotes the text to be synthesised. The result message definition is left empty because we only need to know when the synthesis has finished. Lastly, the feedback message contains the variable current_word_index, an integer that tells us the index of the current word being spoken.
The Target action interface is used to make a body part point at an entity, for example, to make a robot gaze at a person’s head (Figure 8.14). The goal definition includes parameters to specify the targets frame id (target_frame_id) and the speed and acceleration the body part should move at. The result definition doesn’t contain any parameters because all we are interested in is whether or not the target action has reached its goal. The feedback message contains a distance_to_target parameter, which tells us how far the target is from the end effector frame the 2 dimensions that end up facing the target (y, z).

```
# Define goal
string text # Text to be synthesised.
---
# Define result
---
# Define feedback
int32 current_word_index # Current word being spoken.
```

Figure 8.13. Text to speech ROS action definition.

The Gesture action interface is used to make a robot perform a gesture (Figure 8.15). The goal definition includes a gesture parameter which specifies the type of gesture to be executed (the string representation of a gesture enum), the duration the gesture should be performed for and the target that the gesture should be targeted at (the targets frame id). The result definition contains no parameters because we are only interested in knowing when the gesture has finished. The feedback message contains the distance_to_target parameter to give us the distance from the limb to the target.

```
# Define goal
string target_frame_id # The target to target end effector at
float32 speed # Speed to gaze at
float32 acceleration # Acceleration of gaze
---
# Define result
---
# Define feedback
float32 distance_to_target
```

Figure 8.14. Target ROS action definition.
The Expression action interface is used to make a robot perform facial expressions (Figure 8.16). The goal definition includes: a parameter that indicates the type of expression to perform (the string representation of an expression enum); an intensity value which represents the strength of an expression, e.g. a smile with an intensity of 1.0 is the largest smile possible (values normalized from 0.0 – 1.0); a speed value, which is how fast the expression is performed at, e.g. a smile expression with a speed of 1.0 is performed at the fastest possible speed (values normalized from 0.0 – 1.0); the last parameter is the duration that the expression lasts for (seconds).

8.3.2 DG2: Using the social action interface

Vendor action drivers communicate through the action interface by implementing the abstract class for the action type they perform (Figure 8.17). For example, to create an action driver that uses the festival speech synthesis framework, you would implement the ISayServer abstract class, overriding the abstract methods with function calls to the festival speech synthesis framework that synthesise, stop and send feedback about speech. Abstract action server definitions are provided for all four action interfaces, each of which is described below.
Say

The ISayServer abstract class is used to integrate speech synthesis drivers with the social-interaction API. It has three abstract methods that need to be overridden and implemented, including:

\textit{start\_speaking}(text)

Synthesise the sentence with a text to speech synthesiser.

\textit{stop\_speaking}():

Stop the text to speech synthesiser if it is currently active.

\textit{say\_duration}(text, start\_word\_index, end\_word\_index)

Return how long it takes (in seconds) to synthesise a subset of a text string. The subset of the text string is given via the parameters: \texttt{start\_word\_index} and \texttt{end\_word\_index}. This is used for synchronising other actions, such as gestures, with speech.

As well as abstract methods, ISayServer also provides methods that need to be called during various stages of speech synthesis, including:

\textit{send\_feedback}(current\_word\_index)

Call this method when the current word being synthesised changes.
send_done()

Call this method when the text to speech synthesiser has finished speaking the current sentence.

Target

The ITargetServer abstract class is used to integrate action drivers that make robot body parts target objects, e.g. make a head gaze at something. It currently has one abstract method:

execute(target_goal)

Implement your targeting algorithm within this method to make an end effector target a particular entity. Use the is_preempt_requested method from the actionlib API to find out when the target algorithm is pre-empted.

As well as the above abstract method, ITargetServer provides methods that need to be called while the targeting driver is running:

send_feedback(distance_to_target)

Call this method to update the action client with the current distance of the end effector from the target.

send_done()

Call this method to update the action client with the current distance of the end effector from the target.

Gesture

The IGestureServer abstract class is used to integrate action drivers that make the robot perform gestures. It differs from the previous two action servers because it can execute multiple goals simultaneously, which is important for running multiple gestures at once. It has two abstract methods that need to be implemented, including:

start_gesture(gesture_goal)

Start the gesture specified in the gesture_goal.

stop_gesture(gesture_goal)

Stop the gesture associated with this gesture_goal.
As well as the above abstract methods, `IGestureServer` provides methods that need to be called while a gesture is being performed:

\[
\text{send\_done}(\text{gesture\_goal})
\]

Call this method when a gesture has finished being performed; specify the gestures `gesture_goal` to tell the action server which gesture has succeeded.

\[
\text{send\_feedback}(\text{gesture\_goal}, \text{distance\_to\_target})
\]

Call this method to update the action client with the latest distance to the target, specify the `gesture_goal` to tell the action server which gesture this value relates to.

**Expression**

The `IExpressionServer` abstract class is used to integrate action drivers that make the robot perform expressions. It is similar to the `IGestureServer` because it allows multiple expressions to be executed at once. It has two abstract methods that need to be implemented, including:

\[
\text{start\_expression}(\text{expression\_goal})
\]

Start the expression specified by the `expression_goal`.

\[
\text{stop\_expression}(\text{expression\_goal})
\]

Stop the expression associated by the `expression_goal`.

### 8.3.3 DG3: Social perception interface

The social perception interface allows `Entity` instances (e.g. `Person` instances) and the `World` instance to communicate with vendor perception algorithms (Figure 8.18). The Perception Interface is composed of the Perception Synthesiser and the ROS Transform Library (Foote, 2013). The Perception Synthesiser populates the `World` with entity instances that represent perceived objects. The ROS Transform Library acts as a shared interface for storing and retrieving entity coordinates.

Vendor perception algorithms send two types of data via the Perception Interface: the entities they are currently tracking and the coordinates of each entity. A high-level overview of the perception system is given in Figure 8.19.
Tracking entities

The first data type perception algorithms send through the perception interface are the entities the algorithms are currently tracking. This is accomplished by publishing an array of the tracked entities local ids to the Perception Synthesiser via a ROS topic. A local id is an unsigned integer, unique within the scope of the perception algorithm. In this example (Figure 8.19), the openni tracker node, which tracks human skeletons, has published the array of local ids [1, 3] to the topic `/openni_tracker/entity_list`. This indicates that the openni tracker node is currently tracking two person entities: person 1 and person 3.

When the Perception Synthesiser receives new data from a perception algorithm, it updates the World instance, instructing it to create entities to represent the perception data. For example, in Figure 8.19, when a new person is detected by the openni tracker ROS node, the perception synthesiser tells the World instance to create a new Person instance to represent that person. The perception synthesiser will also indicate if the entity has gone in or out of view, which sets a visibility flag to true or false respectively.
Broadcasting coordinates

The second data type perception algorithms send through the Perception Interface are the coordinates of the entities they are tracking. This is accomplished by broadcasting the entities coordinates to the ROS Transform Library (Foote, 2013). To ensure that the API can retrieve the correct coordinates for a particular entity, the frame id naming scheme needs to be the same scheme used in the class definition of the entity (discussed in Section
8.2.1. The Entity class makes use of the coordinate data published ROS Transform Library (Foote, 2013).

An example of where the coordinates of an entity are used is illustrated in Figure 8.19, where the `robot.distance_to(person)` function is called. This finds the distance between the robot and person entity instances. To calculate this, the coordinates for each entity are retrieved using the ROS Transform Library, which looks up the coordinates using the frame ids of each entity. In this example, the Nao robot would return the frame id “torso” and the person object in this example would return “person1_head”. The distance calculation is then made and returned.

8.3.4 DG4: Configuring the perception system

The perception algorithms used by the Perception Synthesiser are configured with a YAML configuration file. An example is shown in Figure 8.20 which configures the perception synthesiser to use the openni skeleton tracker algorithm.

```
people_source:
  entity_list: openni_tracker/users
entity_type:
  module: hri_api.entities
  class: Person
```

Figure 8.20. Perception synthesiser yaml configuration file.

The `entity_list` key/value pair specifies the topic where the perception algorithm publishes its list of tracked entities. The `entity_type` dictionary contains two key-value pairs (module and type), which together indicate what entity class the perception algorithm creates. The Python module name of the entity class is assigned to the `module` key. The Python class name of the entity class is assigned to the `class` key. When the Perception Synthesiser sends an AddEntity service call to the World instance, to instruct it to create a new entity, it specifies a module and class name which indicate what type of entity the World instance should create.

8.4 Summary

This chapter presented the software architecture designed to address research question four (RQ4): “what is a suitable software architecture to support the API with primitives at the abstraction level appropriate for programming robot social-interaction?” It began
by discussing the objectives for the development of the software architecture. The first objective was to provide mechanisms that enable the API to be extended so that programmers can define new entities, robots, gestures and expressions. The second objective was to provide interfaces at the social primitive abstraction level. The chapter then discussed the design goals for each objective. The design goals for the first objective were to enable new entities, robots, gestures and expressions to be defined. The design goals for the second objective were to create social action and perception interfaces so that the API can communicate with multiple vendors robot platforms at the social primitive abstraction level. This was followed by a detailed discussion of how the design goals for each objective were implemented, which is important for those doing similar work and programmers adding a new robot to the social-interaction API.

The next chapter discusses the findings of the research presented in this thesis.
This chapter discusses the findings of the research presented in this thesis, which was directed by a high-level objective, to investigate methods for improving the usability of tools for programming socially interactive robots. This objective was realised through a number of research questions:

RQ1 What abstraction level is appropriate for programming social robot applications?
   RQ1.1 What are the trade-offs associated with using different abstraction levels to program robot social-interaction?
RQ2 How should primitives, regardless of their abstraction level, be orchestrated to create higher level social-interaction such as dialogue and non-verbal behaviour?
RQ3 What primitives are needed to program robot social-interaction at the appropriate abstraction level?
RQ4 What is a suitable software architecture to support the API with primitives at the abstraction level appropriate for programming robot social-interaction?

The chapter starts with a chronological summary (Section 9.1) summarising the work already presented in this thesis. This is followed by a discussion of the different aspects of the research and the contributions made by this work (Section 9.2).

9.1 Chronological summary

The first set of findings begins in Chapter 2, which presented the preparatory study of Ruru, a visual language designed to program tasks from the Learning Computing with Robots course book (Kumar, 2009) and the Player robot middleware examples (T. H. Collett et al., 2005). Ruru worked well to create typical programs that teach novices
principles of programming via robotics as well as simple robot middleware example programs.

However, when exploring how Ruru could improve the usability of complex robot social-interaction programs, I decided that visual languages were not the answer to this problem as they were not well suited to expressing such complexity. In fact, I found usability issues with robot social-interaction programming tools, whether they were visual languages, APIs or domain-specific languages. The usability issues originated because the tools either had very low abstraction levels, or they lacked the ability to express social-interaction adequately. The research conducted with Ruru and the limitations of existing robot programming tools led to the next investigation (Chapter 5).

The next line of investigation was to explore what abstraction level is appropriate for programming social robot applications (RQ1) and how primitives, regardless of their abstraction level, should be orchestrated into higher level social-interaction (RQ2). To begin this investigation I designed, implemented and evaluated an API for programming the social-interaction of a Nao humanoid robot. An API was chosen because the stakeholders that create robot social-interaction are mostly hobbyists and professional programmers who typically use textual programming languages, not visual languages.

There were two main components to this API: high-level domain-specific primitives and an imperative state machine implementation to orchestrate the behaviour of Nao. The APIs primitives were based on a taxonomy of primitives that I derived from researchers’ descriptions of robot behaviour (discussed later). A high-level, domain-specific abstraction level was chosen based on an analysis of existing programming tools with the Cognitive Dimensions (Green & Petre, 1996) (Chapter 3.2.5). The imperative state machine implementation was inspired by ROSs SMACH (Bohren & Cousins, 2010).

The API was then evaluated, which found usability benefits caused by high-level, domain-specific primitives, but also discovered usability issues caused when imperative state machines are used to orchestrate robot behaviour. After the user evaluation, I performed a stakeholder interview with potential users of the API. The main finding of the stakeholder interview was that the social-interaction domain may have been abstracted too far. This was important for continuing exploration of what abstraction level is appropriate for programming social robot applications (RQ1). There were some limitations to this study, in particular: it had only been performed on an API designed for one social-interaction scenario and one robot, each user only spent an hour with the API.
and users divided their time between using the social primitives and the imperative state machine.

This led to the next study (Chapters 6), which further investigated what abstraction level is appropriate for programming social robot applications (RQ1); by designing, implementing and evaluating a second iteration of the social-interaction API. To address issues with the previous abstraction level, the API was refactored to provide finer control of social-interaction (social primitives). To broaden the capabilities of the API, it was iteratively refactored to support two additional humanoid robots, the main difference being that they had animatronic faces. A new scenario was also used as an exemplar use case during the design and development of the API. The evaluation focused on giving users more time and freedom with the API as well as focusing the investigation on how social primitives affect usability. The results indicate that the slightly lower level, domain-specific primitives still had a positive effect on usability (noticeably high abstraction level, terse notation, good role expressiveness). It also revealed negative effects on progressive evaluation, caused by low remote visibility.

The next chapter presented a taxonomy of the primitives that are needed to program robot social-interaction at an appropriate abstraction level (RQ3). It was initially derived as a taxonomy of the primitives used to program healthcare robot behaviour before the first API iteration. It was then refined throughout the iterations described in Chapters 5 & 6. The final taxonomy specifies both the actions and perceptions of socially interactive robots, as well as the individual requirements of those actions and perceptions. The taxonomy is designed to inform social-interaction stakeholders of important primitives involved in creating social-interaction.

The last piece of the puzzle is Chapter 8, which presents the software architecture that evolved as the first (Chapter 5) and second (Chapter 6) social-interaction APIs were developed (RQ4). The software architecture was designed to be both extensible and vendor agnostic. This allows API users to customise it to their needs (e.g. by defining new entities and robots) and provides the infrastructure to enable a variety of robot platforms to be programmed by the social-interaction API (via the social action & perception interfaces).
9.2 Discussion

This section reviews the research conducted in this thesis. Section 9.2.1 discusses the strengths and limitations of the methodologies used in this thesis, providing advice for others doing similar work. Section 9.2.2 discusses the findings related to discovering an appropriate abstraction level for programming socially interactive robots. This is followed by Section 9.2.3, which discusses the limitations of designing APIs for real robots and provides guidance for those conducting further explorations into robot social-interaction APIs. Section 9.2.4 discusses the findings related to orchestrating primitives into social-interaction and promising directions for exploring this topic in the future. Section 9.2.5 discusses the taxonomy of social primitives. Lastly, Section 9.2.6 discusses future directions for exploring social-interaction software architectures.

9.2.1 Methodology

The Cognitive Dimensions (Green & Petre, 1996) was used frequently throughout the research presented in this thesis: as a design tool, a questionnaire (Blackwell & Green, 2000) and a framework for analysing results from user studies.

The Cognitive Dimensions (Green & Petre, 1996) was useful as a lightweight design tool for weighing up the merit of various design choices because it makes it explicit how the design decisions affect the notation. However, in practice, even with such a tool it is hard to judge the relative importance of these effects, especially before a user evaluation is performed. Whilst it is unlikely that user evaluations can be replaced, they are resource intensive and often occur once a working prototype has been developed. To understand the strengths and limitations of a particular design sooner, I recommend others undertaking similar research use a light weight evaluation method that doesn’t require a complete prototype, early in the design process. Examples include pseudo code evaluations, cognitive walkthroughs (Wharton et al., 1994), or having an external researcher conduct a Cognitive Dimensions expert evaluation.

The CD Questionnaire (Blackwell & Green, 2000) was used in both user evaluations. In Chapter 5 participants wrote responses to the questionnaire and in Chapter 6 the questions were incorporated into interview questions. Blackwell & Green designed the questionnaire to present the Cognitive Dimensions in general terms so that users with no prior knowledge of the Cognitive Dimensions can provide Cognitive Dimensions based
usability feedback. A big advantage of this method is that analysis is easier because the data is already organised by each Cognitive Dimension.

Whilst the CD Questionnaire (Blackwell & Green, 2000) was very useful, it does take a long time to answer because there are (deliberately) two to three very similar questions for each dimensions (e.g. Figure 9.1). This is appropriate when a respondent has several days to reflect on the questions and provide answers, as one group of participants did in Blackwell & Green’s (Blackwell & Green, 2000) pilot study of the questionnaire. However, it appeared to overwhelm lab study participants who had already spent an hour on a programming task. This meant that instead of providing detailed feedback they at times gave yes or no answers for the first question (of a dimension) and then skipped the rest of them. To address this, when used in a lab study, the questionnaire could be shortened by only providing one question per dimension. This gives participants more time to provide detailed, quality feedback. Participants could also be explicitly asked to provide detailed feedback and avoid yes/no responses in the questionnaire instructions.

- How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not? •
- Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not? •
- Can you try out partially-completed versions of the product? If not, why not?

**Figure 9.1 CD Questionnaire questions for progressive evaluation** (Blackwell & Green, 2000).

The think-aloud protocol was not explicitly used (Lewis & Rieman, 1993, p. 83) in the first user evaluation, however, some participants thought aloud on their own initiative. This was useful because it gave me an opportunity to step in and ask questions to clarify what they were thinking. Those undertaking similar evaluations would benefit from explicitly using the think aloud protocol. It would also serve as a good source of data triangulation combined with a post-task questionnaire like the CD Questionnaire (Blackwell & Green, 2000).
9.2.2 An appropriate abstraction level for programming robot social-interaction

The first research question investigated “what abstraction level is appropriate for programming social robot applications?” To explore this, I iteratively designed, implemented and evaluated two APIs for programming robot social-interaction (Chapters 5 & 6).

The first API (Chapters 5) was designed with high-level, domain-specific primitives. It primarily contained social primitives and emergent primitives. For example, the function `say_to` is an emergent primitive, whilst the `Person` class is a social primitive. The evaluation of the first API revealed that high-level, domain-specific primitives show promise as an appropriate abstraction level for programming robot social-interaction (RQ1). This is because they provide good visibility & role expressiveness, have a close mapping to social-interaction and are terse. However, participants in a stakeholder interview desired a finer control over social-interaction. This is important for RQ1 because it implies that a lower abstraction level may be more appropriate for programming socially interactive robots.

This led to the second API iteration (Chapters 6) where some primitives were refactored, giving them slightly lower abstraction levels and more control of social-interaction. For example, `say_to` became the four methods: `say`, `gaze`, `gesture` & `expression`. A user evaluation of the API revealed both positive and negative effects of the domain-specific primitives for programming robot social-interaction. Despite the changes between the two APIs, the key advantages still held. Participants perceived the notation as: having a close mapping to the social-interaction domain, having a high abstraction level, being terse and having a good role expressiveness. The main negative effect is reduced remote visibility, which makes progressive evaluation harder. The following paragraphs discuss these effects in more depth.

The first positive effect on usability is a close mapping to the social-interaction domain. Participants in the first study found that the close mapping kept the API terse whilst participants in the second study found that it helped them understand the API, presumably because they could use their domain knowledge to do so.

The second positive effect on usability is caused by a high abstraction level, which hides lower level implementation details. This is a particularly important attribute for
improving the usability of social robot programming tools because social robots rely on a plethora of complex hardware, middleware and algorithms that take a long time to understand and master. Despite the second API having a lower abstraction than the first, those who participated in the second API evaluation still perceived it as being high. This, coupled with the APIs close mapping to the social-interaction domain, suggests that social primitives are more appropriate than emergent primitives for programming robot social-interaction because they are more reusable and still retain positive usability attributes as viewed through the Cognitive Dimensions (Green & Petre, 1996).

The process of lowering abstraction levels between API iterations illustrates that there is a fine balance for choosing an abstraction level, even when creating domain-specific APIs. Roberts & Johnson (1996) provide a pattern (fine-grained objects) in their Evolving Frameworks Pattern Language that could help a designer address this problem. They suggest breaking objects (in this case methods) into smaller units that encapsulate a single behaviour.

The third positive effect on usability is caused by having a terser notation. Participants in the first study (Chapter 5) indicated that the domain-specific aspects (closeness of mapping) of the API had positive effects on the notations terseness. Additionally, participants in the second study (Chapter 6) found that the APIs terseness minimized the amount of code they had to write, especially in comparison with their experiences with ROS. This is probably because the social primitives have a close mapping to the social-interaction domain, allowing programmers to express what they want with fewer primitives than a general robot programming framework.

The last positive effect on usability is caused by an increase in visibility and role expressiveness. Participants in the first study (Chapter 5) found that the domain-specific aspects of the social primitives increased the visibility of the API, making it easier to find parts of the notation because it organized it in a clear and concise way. This is very closely related to role-expressiveness, which was high in both APIs. For example, in the second study (Chapter 6) participants indicated that it was very obvious how the methods in the API relate to each other and that it was easy to understand what each part of the API did. This is probably because users can use their existing domain knowledge of social-interaction to understand what each social primitives does and how they relate to each other.
The main negative effect of domain-specific API primitives is that they reduce remote visibility, making progressive evaluation harder because programmers write programs at a high abstraction level, but still have to debug their programs at a lower abstraction level when things go wrong. This became apparent in the second evaluation (Chapter 6) where participants commented that it was hard to explore the internal workings of the API; hard to find the sources of errors, e.g. not supplying a parameter and the method fails; and hard to understand the internal state of method calls when debugging at runtime. Presumably, a large part of this frustration is caused because when debugging, the programmer has to understand what is going on at the lower level before progressing.

The negative effects of reduced remote visibility could be addressed by following the API documentation guidelines proposed by Robillard & DeLine (2011) which are discussed in Chapter 3.1, in particular, providing detailed documentation about the internal workings of the API (document factors affecting penetrability). These can include but are not limited to performance properties, error handling behaviour and details if the abstraction does more than one operation. The goal is to minimise the time developers spend inspecting the inner workings of an API but to give them enough details to do so if they need to.

These same benefits may be seen when applying domain-specific API primitives to other areas of human-robot interaction, for example, to enable people to program applications that involve navigation or manipulation. They may even be useful when programming robots that are not designed to interact with humans, for example, industrial robot arms.

9.2.3 Further exploration of APIs for programming social-interaction

The APIs designed in Chapters 5 and 6 are dependent on current commercial robot platforms. This made it difficult to explore certain areas of the taxonomy, especially those related to human perception. For example, identifying who is speaking is an important part of the taxonomy but hard to implement on small humanoid robots; this made it difficult to explore how to embody this feature into an API. In the short to medium term, a social-interaction simulator would remove these artificial constraints on social-interaction API design, thus allowing the designer to create future proof API designs that
are not dependent on current hardware and software. In the long term, better algorithms need to be developed.

Gazebo (Open Source Robotics Foundation, n.d.-c) and Morse (LAAS-CNRS & ONERA, n.d.) are two popular simulators for ROS (Open Source Robotics Foundation, n.d.-a) which could be altered for this purpose. They would require an interface that enables an API designer to control or script multiple simulated people interacting socially with a simulated robot. The inspiration for this interface can be taken from real-time strategy games such as StarCraft II (Blizzard Entertainment Inc, 2015) which allow players to control multiple characters in real time from an isometric-like 3D perspective. In a similar manner, API designers could control the interactions of multiple simulated people, including their speech, gestures, expressions, touch, navigation and manipulation. This would allow much quicker prototyping and testing of API designs and algorithms that control robot social-interaction.

In the long term, improvements made to the API will need to be realised on real robots, which will require a large amount of work improving open source perception algorithms. I learned this the hard way by trialling many perception algorithms when developing the APIs presented in Chapters 5 and 6. Many of them didn’t work as advertised, or had major limitations when used for robot social-interaction. Speech recognition algorithms can’t be run while the robot is speaking because the robot will recognise its own voice. This creates an awkward interaction because the person has to say the same thing multiple times so that the robot hears it. To make matters worse, closed vocabulary speech recognisers such as Pocketsphinx (Carnegie Mellon University, n.d.) generally don’t reject out of vocabulary speech, meaning that background noise and speech is detected by the recogniser. Sound source localisation and tracking systems (e.g. Hark) can be used to associate people’s speech with tracked faces. Hark (HARK Support, n.d.), an open source system, appears to work well in demonstration videos (“HARK: Selectable Sound Separation - YouTube,” 2010), however, the example programs don’t work nearly as well in real life. Lastly, open source face detection algorithms don’t actually track people’s faces between frames, which is required to represent people as high-level objects.

Programmers would benefit from having programming based tools for authoring content. For example, participants in the second study (Chapter 6) wanted to be able to programmatically create new gestures and expressions rather than using an animation tool such as Choregraphe (Pot et al., 2009). This is because they perceived using an animation
tool to create content as slowing them down - programming is what they are used to. Meeting this goal would require a taxonomy of atomic gestures and expressions that could be combined in a large variety of ways creating an endless variety of new gestures and expressions for a range of robots. This would give programmers the ability to create new gestures and expressions in a way that they are familiar with. Some of the APIs methods may need changing to fully support this, but most of the effort would occur in the software architecture so that multiple gestures and or expressions can be blended together.

9.2.4 Orchestrating primitives into social-interaction

The second research question investigated “how should primitives, regardless of their abstraction level, be orchestrated to create higher level social-interaction such as dialogue and non-verbal behaviour?” This question focuses on finding usable mechanisms for programmers to easily author autonomous social-interaction with pre-defined logic (as introduced in Chapter 1.1, page 29). In Chapter 5, this question was explored by evaluating the effect that imperative finite state machines have on usability when used to program social-interaction. This is important because imperative state machines are a popular method of orchestrating primitives into robot behaviour, the most notable example is SMACH (Bohren & Cousins, 2010) the de facto method for authoring robot behaviour in the popular ROS (Open Source Robotics Foundation, n.d.-a).

The evaluation in Chapter 5 suggest that imperative finite state machines may have a number of usability problems, which include: being very diffuse; requiring premature commitment, because the programmer has to define the state machine in a certain order; requiring hard mental operations, for instance, participants commented that they needed to sketch out the design beforehand (and indicator of hard mental operations (Green & Petre, 1996, p. 11)); and they are error-prone, e.g. misnaming state identifiers is common. The main limitations of these findings are that the study was only performed with one implementation of an imperative finite state machine and the example programs the participants built with the state machine were relatively simple. However, the state machines design was influenced by SMACH (Bohren & Cousins, 2010), so the findings may be reproducible with other state machine notations.

Nevertheless, the results still indicate that further work should explore how to improve usability when orchestrating primitives into robot social-interaction. They also suggest that imperative state machines may not be the best answer to this question, despite their
popularity. Insights into this question may come from tools used to author character behaviour in commercial video games. In fact, the ‘artificial intelligence’ algorithms that drive the behaviour of most commercial game characters are actually human authored scripts (Robertson & Watson, 2014), which is a very similar problem to human authoring of robot behaviour. Additionally, the game industry has invested far more time and money into tools than those who use and create social robots.

A paradigm from game industry that may prove useful for programming social robot interaction are visual behaviour tree editors, a very popular means of authoring game character behaviour (Robertson & Watson, 2014). Many game engines provide visual behaviour tree engines, for example, Unreal Engine 4 (Epic Games Inc, n.d.). Figure 9.2 illustrates how a visual behaviour tree editor could be used in conjunction with the social-interaction API to author robot behaviour. The design is similar to modern user interface designers, e.g. the Windows Presentation Foundation Designer for Visual Studio (Microsoft, n.d.). The behaviour tree in this example makes the robot say “Hello new person!” to every new person it sees. Programmers create the behaviour tree logic using a visual editor (Figure 9.2a) and an XML representation (Figure 9.2b). Double clicking on a behaviour tree node automatically creates a method stump that will be called when the behaviour tree node is executed; the programmer fills the method stump with social-interaction API code (Figure 9.2c). The link between behaviour tree nodes and their method calls are encoded in the XML representation (Figure 9.2b).
9.2.5 A taxonomy of social primitives

The third research question investigated “what primitives are needed to program socially interactive robots?” To accomplish this, over the course of my thesis I derived a taxonomy of primitives at an abstraction level appropriate for programming socially interactive robots. This process began by searching through the literature for how researchers had described robot behaviour and used grounded theory (Glaser & Strauss, 1967) to form an overall theory of the features required to program this behaviour.

This was a good way to start developing an API because it provided a medium to think about the core features required for programming tasks without considering implementation details. The process of systematically searching the literature could be replaced by a more lightweight method, e.g. just find enough source data to build up a taxonomy. There comes a point where you can no longer think about the features for an API in such an abstract way and you must implement the taxonomy as a real API. This is echoed by Roberts & Johnson (1996) in their Evolving Frameworks Pattern Language, where they argue that determining the correct abstractions for a framework involves...
implementing a working system. This can be seen by comparing the original taxonomy presented in (Diprose et al., 2012) to the final taxonomy presented in Chapter 7. They are significantly different, however, it is easy to see how one evolved from the other as there are still common threads that link them. For example, the higher level categories of objects (People and Robots), actions and perceptions exist in both, but the finer details became more refined as the APIs were iteratively designed.

The same technique could be used as a lightweight method for beginning the design of APIs for other domains, especially when the designer doesn’t have a clear idea about what features to include in the API. Then later on, the designer can refine the abstractions in their taxonomy by implementing it as an API.

The taxonomy is designed to be used as a discussion tool for social robotics stakeholders to communicate because it provides a standard set of primitives and features for describing social-interaction. That is, it is similar in purpose to the Cognitive Dimensions (Green & Petre, 1996), but for the domain of robot social-interaction. The major categories defined by the taxonomy include objects, people and robots; the actions a social robot can perform; and what a social robot can perceive. It could be used by different stakeholders in a number of ways, including helping robot manufacturers identify important sensors and actuators for social-interaction applications; informing algorithm developers of what algorithms are useful for social-interaction applications; and highlighting what primitives and features are important for social robot programming tool developers.

Social robot programming tools could produce more complex applications if they implemented a greater number of primitives and features from the taxonomy of social primitives. However, tool designers need to be careful to implement the features of each primitive, because these are what enable the expression of nuanced social-interaction. For example, the perceiving people’s gestures primitive has a feature to identify who performed the gesture. An algorithm that simply returns what gestures are detected is not powerful enough because they aren’t associated with an individual person. This would limit the ability of a social robot programming tool to produce complex, nuanced social-interaction.

Lastly, the taxonomy could be expanded to provide detailed implementation independent requirements for other robotics domains, including robot navigation and manipulation.
9.2.6 Further exploration of social-interaction architectures

The fourth research question focused on investigating “what is a suitable software architecture to support the API with primitives at the abstraction level appropriate for programming robot social-interaction?” This research question was addressed in Chapter 8, which describes the software architecture that evolved through the development of the first (Chapter 5) and second (Chapter 6) API iterations.

The architecture presented in Chapter 8 is designed to be extensible and vendor agnostic at the social primitive abstraction level. Extensibility is enabled by allowing API users to define new robots, entities, relationships, gestures and expressions. Vendor agnosticism is realised through the definition of social action and perception interfaces; allowing the architecture to interface with a variety of robot platforms at a social-interaction abstraction level. The social action and perception interfaces correspond directly to the social primitives implemented in the second iteration of the social-interaction API (Chapter 6). There are a number of ways that this architecture could be further developed in the future.

The first way to improve the software architecture is to remove the APIs dependence on ROS (Open Source Robotics Foundation, n.d.-a). Various parts of the API that programmers directly interact with use ROS, for example, the `World` object uses the ROS messaging system to remain updated about the entities that are currently perceived by the robot and all entities request their coordinates via a ROS service. This makes it difficult to port the API to more operating systems and programming languages because ROS is only officially supported on Ubuntu (Foote, n.d.) and only has client libraries officially implemented in C++ and Python (Open Source Robotics Foundation, n.d.-d). To be clear, the underlying architecture that runs on the robot would still be implemented in ROS, however, the programmer facing components of the architecture should communicate with some other means, for example via a representational state transfer (REST) interface that is easier to port to other operating systems and languages.

Second, if the API was ported to multiple languages, a more scalable means of defining robots, their body parts and entities would be needed. This is because it would be too laborious to manually define every robot and entity in every language. This could be as simple as a tool that generates code to represent a robot from its URDF (Sucan, n.d.-b) definition file.
Last, the biggest changes to the architecture would come as more primitives are developed for the API in conjunction with a human-robot interaction simulator (as described in Section 9.2.3). An example is the perception synthesiser, which should be developed so it can integrate multiple perception sources and associate them with high-level concepts. For instance, if a person’s face is being tracked by a face tracker, their speech recognised by a speech recogniser and the location of their voice detected by a sound source localiser, the perception synthesiser should be able to associate all of this data as originating from the same person. The algorithms behind the robot BIRON’s perception system (Lang et al., 2003) are a potential way forward.
Conclusions & future work

This thesis presented an investigation into how to improve the usability of tools designed to program socially interactive robots. Despite the investigation being constrained to social-interaction, the results are likely applicable to other areas of robotics, such as robot navigation and manipulation. This chapter presents a summary of this research, outlining the key contributions made to the area of social robot programming as well as summarising directions for future research.

10.1 Contributions

The key contributions of this thesis are as follows:

1. An API with high-level, domain-specific primitives for programming socially interactive robots.

2. An in depth understanding of what abstraction level is appropriate for programming social robot applications. This includes knowledge of the effects that primitives of different abstraction levels have on usability when used to program robot social-interaction.

3. An investigation into how primitives, regardless of their abstraction level should be orchestrated to create higher level social-interaction. This revealed factors that may negatively affect usability when orchestrating robot social-interaction with textual, imperative finite state machines.

4. A taxonomy of primitives at the appropriate abstraction level for programming socially interactive robots.

5. The design and implementation of a software architecture to support social robot programming APIs with primitives at the appropriate abstraction level. The
architecture is extensible, vendor agnostic and implemented with the Robot Operating System (ROS), a popular robot middleware solution.

The first contribution of this thesis is an API with domain-specific primitives for programming socially interactive robots, which has practical value for social robot programmers. The development of this API also serves as an exemplar of the use of user-centred design principles and the Cognitive Dimensions (Green & Petre, 1996) for iteratively designing, implementing and evaluating an API.

The second contribution of this thesis is an in depth understanding of what abstraction level is appropriate for programming social robot applications and the effects that different abstraction levels have on usability in this context. The investigation found that high-level, domain-specific primitives have many benefits for usability; these include both social and emergent primitives. The evidence suggests that the social primitive abstraction level is most appropriate for programming robot social-interaction because it has higher reusability and still has many positive effects on usability. The positive effects of social primitives include a close mapping to the social-interaction domain, which helps people learn the notation; a high abstraction level, which hides lower level implementation details; a terse notation; and lastly high local visibility and good role expressiveness. The main negative effect is low remote visibility which can make progressive evaluation hard. This is a trade-off that occurs when choosing a high abstraction level; it could be addressed to some extent with Robillard & DeLine’s (2011) API documentation guidelines.

The third contribution is an investigation into how primitives, regardless of their abstraction level, should be orchestrated into higher level social-interaction. This led to an understanding that imperative finite state machines may have poor usability when used to orchestrate primitives into robot social-interaction and specific reasons why this might occur. This is important because finite state machines are a common means of creating robot dialogue and imperative finite state machines (a subset of finite state machines) are a common tool available to robotics developers, e.g. ROS’s SMACH (Bohren & Cousins, 2010). Specific reasons why textual, imperative finite state machines can be hard to use include: they are diffuse, using many lines of code; their use involves hard mental operations and premature commitment; and they are error prone, for example, misnaming state identifiers is a common occurrence.
The fourth contribution of this thesis is an implementation independent taxonomy of the primitives required to program robot social-interaction at the appropriate abstraction level. The taxonomy was iteratively derived over the course of this thesis using a methodology based on grounded theory (Glaser & Strauss, 1967). It specifies important objects for robot social-interaction; the actions and perceptions that social robots perform; and the individual features of each action and perception. The taxonomy is designed to act as a discussion tool for social robotics stakeholders to communicate and think about robot social-interaction. These include robot manufacturers, algorithm developers and social robot programming tools designers.

The fifth contribution is the design and implementation of an extensible, vendor agnostic architecture that is designed to support APIs for programming robot social-interaction. The architecture enables social-interaction API users to define new robots and entities, as well as the relationships between a robot and its body parts and entities and their parts. The architecture is also vendor agnostic, which is realised through the definition of social action and perception interfaces. This has the potential to allow a large variety of vendor’s robots to be programmed by the social-interaction API because they can interface with the architecture at a social-interaction abstraction level.

10.2 Future work

Lightweight API evaluation methodologies could be used early in the API design process, including pseudo code evaluations, cognitive walkthroughs (Wharton et al., 1994) and Cognitive Dimensions (Green & Petre, 1996) expert evaluations.

To improve the results from the CD Questionnaire (Blackwell & Green, 2000), it could be shortened by removing duplicate questions and participants could be reminded to provide detailed answers where possible.

Choosing appropriate domain-specific primitives for programming social-interaction has many usability benefits; the same benefits may be seen in other areas of robotics by choosing API primitives with appropriate abstraction levels. For example, do APIs and domain-specific languages for programming navigation & manipulation tasks have appropriate abstraction levels?
The main negative effect of social primitives is reduced remote visibility, which makes progressive evaluation harder. This could be addressed to some extent by following Robillard & DeLine’s (2011) API documentation guidelines.

Social-interaction API designs could be explored faster and in more detail by using a human-robot interaction simulator that is capable of controlling the social-interaction of multiple human characters in real time, similar to how characters are controlled in real time strategy games. In the long term, perception algorithms will need to be significantly improved to make the simulated social-interaction scenarios work on real robots.

The methodology of developing a taxonomy of primitives as a part of the API design process could be applied to other domains. An API that implements more primitives and features from the taxonomy of social primitives would be able to produce more nuanced, expressive social-interaction. The taxonomy could be expanded to include other areas of robotics, including navigation and manipulation.

Programmers need programming tools to help them author content because they can’t always rely on artists and don’t want to use tools designed for artists. This could be achieved by creating a set of atomic gestures and expressions, which when combined create a large variety of unique gestures and expressions.

To improve the usability of tools for orchestrating primitives into social-interaction, inspiration can be taken from the tools used to program game characters. The ‘artificial intelligence’ behind many game characters are actually human authored scripts (Robertson & Watson, 2014) and the video game industry has invested more time and money developing their tools than the social robotics community.

As the capabilities of APIs for programming social-interaction are being developed further (e.g. in conjunction with a social-interaction simulator), so too should the architectures, algorithms and platforms for realising these features on real robots.
References

References


References


References


References


References


Spiza, S., & Hanenberg, S. (2014). Type names without static type checking already improve the usability of APIs (as long as the type names are correct): an empirical study. Proceedings of the of the 13th international conference on Modularity (pp. 99–108). ACM.


Appendix A
Related work
A.1 Robot programming tools

Table A.1 and Table A.2 provide source data for the abstraction levels found in different social-interaction programming tools and general robotics programming tools, as discussed in Chapter 3.1.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Target audience evidence</th>
<th>Hardware primitive evidence</th>
<th>Algorithm primitives evidence</th>
<th>Social primitives evidence</th>
<th>Emergent primitives evidence</th>
<th>Primitive control method evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choregraphe</td>
<td>Novice: “It is a way for non-expert developers to avoid the complexity of ‘post’ subtlety or ‘subscribeOnDat a’ functions” (Pot et al., 2009, p. 50).</td>
<td>✓ See Choregraphe’s visual blocks: e.g. LED control, sonar sensors, and accelerometer. Also: “Technically Choregraphe is just a graphical representation of NaoQi’s functions but practically it is much more” (Pot et al., 2009, p. 50).</td>
<td>✓ See Choregraphe’s visual blocks: e.g. Speech Reco, Face Reco and Sound tracker.</td>
<td>✓ See Choregraphe’s visual blocks e.g. Say, Stand up and Sit Down.</td>
<td>✓ Flow-based programming editor (Figure 5) and a timeline editor (Figure 6) (Pot et al., 2009, p. 50).</td>
<td></td>
</tr>
<tr>
<td>Interaction Blocks</td>
<td>Novice: “designers will need materials and tools that will enable them to explore and prototype a range of interactions that robots will offer in these settings” (Sauppé &amp; Mutlu, 2014, p. 1439).</td>
<td>✓ See Figure 3: introductory monologue, question-answer, comment exchange, monologue-comment, instruction-action, closing comment, wait (Sauppé &amp; Mutlu, 2014, p. 1443).</td>
<td>✓ Table I: lookForFace, isFaceDetected, isSpeechResult (Glas et al., 2012, p. 6).</td>
<td>✓ Table I: talk (Glas et al., 2012, p. 6).</td>
<td>✓ A timeline, see Figure 4 (Sauppé &amp; Mutlu, 2014, p. 1444).</td>
<td></td>
</tr>
<tr>
<td>Interaction Composer</td>
<td>Novice &amp; professional: “Roughly speaking, we can categorize the main developers of a robot application into ‘programmers’ and ‘designers’”.</td>
<td>✓ Table I: lookForFace, isFaceDetected, isSpeechResult (Glas et al., 2012, p. 6).</td>
<td>✓ Table I: talk (Glas et al., 2012, p. 6).</td>
<td>✓ Table I: ask (Glas et al., 2012, p. 6).</td>
<td>✓ Figure 2, a visual representation of imperative programming (Glas et al., 2012, p. 4).</td>
<td></td>
</tr>
<tr>
<td>Tool</td>
<td>Target audience evidence</td>
<td>Hardware primitive evidence</td>
<td>Algorithm primitive evidence</td>
<td>Social primitive evidence</td>
<td>Emergent primitive evidence</td>
<td>Primitive control method evidence</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------</td>
<td>----------------------------</td>
<td>------------------------------</td>
<td>----------------------------</td>
<td>-----------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>TiViPE</td>
<td>Novice &amp; professional: “allows a scenario designer to decide what blocks are needed and in collaboration with a developer to construct a set of useful graphical … robot behaviors” (Lourens &amp; Barakova, 2011, p. 218)</td>
<td>✓ Figure 2: ledto, ledset, move, movem, stiff, flush (Lourens &amp; Barakova, 2011, p. 215).</td>
<td>✓ Figure 2: walk, walks, walka, walkto, walkd (Lourens &amp; Barakova, 2011, p. 215).</td>
<td>✓ Figure 2: say (Lourens &amp; Barakova, 2011, p. 215).</td>
<td>✓ For the textual language a</td>
<td>b and d &amp; e notation as shown in (Lourens &amp; Barakova, 2011, p. 214). Visual interface in Figure 3 (Lourens &amp; Barakova, 2011, p. 217).</td>
</tr>
<tr>
<td>AIML</td>
<td>Professional</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓ XML based dialogue management system (Wallace, 2003).</td>
</tr>
<tr>
<td>BML</td>
<td>Professional: targeted at “ECA researchers” (Kopp et al., 2006, p. 205).</td>
<td></td>
<td>✓ Table 1: head, torso, face, body, legs, lips general actions: gaze, gesture &amp; speech (Kopp et al., 2006, p. 213).</td>
<td></td>
<td></td>
<td>✓ Figure 5: XML based event system, e.g. event and wait elements (Kopp et al., 2006, p. 214).</td>
</tr>
<tr>
<td>BONSII</td>
<td>Professional: “The focus of this work is to provide a framework for developers of interactive robot systems that perform in domestic environments” (Siepmann, 2013, p. 4)</td>
<td>✓ Table 1: sensors - laser, camera, odometry, position, map, speed; actuators: camera, screen (Lohse, Siepmann, &amp; Wachsmuth, 2014, p. 128)</td>
<td>✓ Table 1: actuators: navigation (Lohse et al., 2014, p. 128)</td>
<td>✓ Table 1: sensors: Person, Object; actuators: Speech, Arm (Lohse et al., 2014, p. 128)</td>
<td>✓ Listing 5.6: SCXML based finite state machine markup language (Siepmann, 2013, p. 61)</td>
<td></td>
</tr>
</tbody>
</table>
## Appendix A – Related work

### Abstractions

<table>
<thead>
<tr>
<th>Tool</th>
<th>Target audience</th>
<th>Hardware primitive evidence</th>
<th>Algorithm primitives evidence</th>
<th>Social primitives evidence</th>
<th>Emergent primitives evidence</th>
<th>Primitive control method evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robot Behaviour Toolkit</td>
<td>Professional: &quot;The Toolkit offers … an open-source Robot Operating System (ROS) [24] module that integrates the behavioral specifications … into an interaction model that supports human activity&quot; (Huang &amp; Mutlu, 2012, p. 26)</td>
<td>✓ Figure 3: participants, objects (Huang &amp; Mutlu, 2012, p. 25). Figure 5: gaze and speech channels (Huang &amp; Mutlu, 2012, p. 28).</td>
<td>✓ Figure 5: sequential timeline (Huang &amp; Mutlu, 2012, p. 28). Activity model + cognitive system + behaviour system (Huang &amp; Mutlu, 2012, pp. 27–28).</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table A.2 Abstraction levels found in general robot programming tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Target audience</th>
<th>Primitives control method evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gostai studio</td>
<td>Professional: “Gostai Studio is a complete IDE solution for developers or hobbyists to graphically create behaviors for any robot or complex system.” (Gostai, n.d.)</td>
<td>✓ “You can easily create hierarchical final state machines with an intuitive editor” (Gostai, n.d.)</td>
</tr>
<tr>
<td>SkillGUI</td>
<td>Professional: “Instead of going along the lines of programming the behaviours graphically (as RoboCupers usually are experienced programmers), skills need to be coded” (Niemüller et al., 2010, p. 248).</td>
<td>✓ “In this paper we propose a behaviour engine for this middle layer which, based on formalism of hybrid state machines (HSMs), bridges the gap between high-level strategic decision making and low-level actuator control.” (Niemüller et al., 2010, p. 240).</td>
</tr>
<tr>
<td>SMACH</td>
<td>Professional: Designed for ROS programmers, who are professionals. Also refer to programmers explicitly on page 17. (Bohren &amp; Cousins, 2010)</td>
<td>✓ “we introduce an approach based on nested state machines that has proven very effective at building real-ROS applications.” (Bohren &amp; Cousins, 2010, p. 18)</td>
</tr>
<tr>
<td>XABSL editor</td>
<td>Professional: “The language has been successfully applied on many robotic platforms, mainly in the domain of RoboCup robot soccer” (Loetzsch et al., 2006, p. 1)</td>
<td>✓ Figure 3 (Loetzsch et al., 2006, p. 3)</td>
</tr>
</tbody>
</table>
A.2 API usability

Table A.3 summarises the papers found in the systematic literature review from Chapter 3.1.

<table>
<thead>
<tr>
<th>Title</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>A case study of using HCI methods to improve tools for programmers</td>
<td>(Faulring et al., 2012)</td>
</tr>
<tr>
<td>A case study of API redesign for improved usability</td>
<td>(Stylos et al., 2008b)</td>
</tr>
<tr>
<td>A field study of API learning obstacles</td>
<td>(Robillard &amp; DeLine, 2011)</td>
</tr>
<tr>
<td>A modular implementation framework for code mobility</td>
<td>(Lobato et al., 2006)</td>
</tr>
<tr>
<td>An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. groovy with eclipse</td>
<td>(Petersen et al., 2014)</td>
</tr>
<tr>
<td>An Empirical Study of API Usability</td>
<td>(Petersen et al., 2014)</td>
</tr>
<tr>
<td>API design matters</td>
<td>(Henning, 2007)</td>
</tr>
<tr>
<td>API usability peer reviews: a method for evaluating the usability of application programming interfaces</td>
<td>(Farooq &amp; Zirkler, 2010)</td>
</tr>
<tr>
<td>Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study</td>
<td>(Duala-Ekoko &amp; Robillard, 2012)</td>
</tr>
<tr>
<td>Authorization enforcement usability case study</td>
<td>(Bartsch, 2011)</td>
</tr>
<tr>
<td>Automated measurement of API usability: The API Concepts Framework</td>
<td>(Scheller &amp; Kühn, 2015)</td>
</tr>
<tr>
<td>Automatic evaluation of API usability using complexity metrics and visualizations</td>
<td>(Souza &amp; Rentolía, 2009)</td>
</tr>
<tr>
<td>Building more usable APIs</td>
<td>(McLellan et al., 1998)</td>
</tr>
<tr>
<td>Constrid: data-centric access control for android</td>
<td>(Schreckling et al., 2012)</td>
</tr>
<tr>
<td>Developing a Usable API for Multi-Surface Systems</td>
<td>(Burns, 2013)</td>
</tr>
<tr>
<td>Do static type systems improve the maintainability of software systems?</td>
<td>(Kleinschmager et al., 2012)</td>
</tr>
<tr>
<td>Documenting and Evaluating Scattered Concerns for Framework Usability</td>
<td>(Hou et al., 2008)</td>
</tr>
<tr>
<td>Effective Java: Programming Language Guide</td>
<td>(Bloch, 2001)</td>
</tr>
<tr>
<td>Empirical analysis of GUI programming concerns</td>
<td>(Mijailovi &amp; Miliev, 2014)</td>
</tr>
<tr>
<td>Evaluation of a programming toolkit for interactive public display applications</td>
<td>(Cardoso &amp; José, 2013)</td>
</tr>
<tr>
<td>Experiences with User-Centered Design for the Tigres Workflow API</td>
<td>(Ramakrishnan et al., 2014)</td>
</tr>
<tr>
<td>Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET libraries</td>
<td>(Cwalina &amp; Abrams, 2008)</td>
</tr>
<tr>
<td>Heuristic inspections for documentation--10 recommended documentation heuristics</td>
<td>(Purho, 2000)</td>
</tr>
<tr>
<td>How do API documentation and static typing affect API usability?</td>
<td>(Endrikat et al., 2014)</td>
</tr>
<tr>
<td>How Programs Represent Reality (and how they don’t)</td>
<td>(Ratiu &amp; Deissenboeck, 2006a)</td>
</tr>
<tr>
<td>How to design a good API and why it matters</td>
<td>(Bloch, 2006)</td>
</tr>
<tr>
<td>Improving Documentation for eSOA APIs through User Studies</td>
<td>(Jeong et al., 2009)</td>
</tr>
<tr>
<td>Improving software API usability through text analysis: A case study</td>
<td>(Watson, 2009)</td>
</tr>
<tr>
<td>Influencing Factors on the Usability of API Classes and Methods</td>
<td>(Scheller &amp; Kühn, 2012)</td>
</tr>
<tr>
<td>InterState: Interaction-Oriented Language Primitives for Expressing GUI Behavior</td>
<td>(Oney et al., 2014)</td>
</tr>
<tr>
<td>Libtrace: a packet capture and analysis library</td>
<td>(Alcock et al., 2012)</td>
</tr>
<tr>
<td>Measurable Concepts for the Usability of Software Components</td>
<td>(Scheller &amp; Kühn, 2011)</td>
</tr>
<tr>
<td>Measuring API Usability</td>
<td>(Clarke, 2004)</td>
</tr>
<tr>
<td>Methods towards API Usability: A Structural Analysis of Usability Problem Categories</td>
<td>(Grill et al., 2012)</td>
</tr>
<tr>
<td>Title</td>
<td>Reference</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>35. Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers’ Newsgroup Discussions</td>
<td>(Hou &amp; Li, 2011)</td>
</tr>
<tr>
<td>37. Programs are Knowledge Bases</td>
<td>(Ratiu &amp; Deissenboeck, 2006b)</td>
</tr>
<tr>
<td>38. Some structural measures of API usability</td>
<td>(Rama &amp; Kak, 2013)</td>
</tr>
<tr>
<td>39. Structural conformance checking with design tests: An evaluation of usability and scalability</td>
<td>(Brunet et al., 2011)</td>
</tr>
<tr>
<td>40. The API Walkthrough Method A lightweight method for getting early feedback about an API</td>
<td>(O’Callaghan, 2010)</td>
</tr>
<tr>
<td>41. The concept maps method as a tool to evaluate the usability of APIs</td>
<td>(Gerken et al., 2011)</td>
</tr>
<tr>
<td>42. The Design of Distributed Hyperlinked Programming Documentation</td>
<td>(Friendly, 1996)</td>
</tr>
<tr>
<td>43. The Factory Pattern in API Design: A Usability Evaluation</td>
<td>(Ellis et al., 2007)</td>
</tr>
<tr>
<td>44. The implications of method placement on API learnability</td>
<td>(Stylos &amp; Myers, 2008)</td>
</tr>
<tr>
<td>45. The Reality of Libraries</td>
<td>(Ratiu &amp; Jurjens, 2007)</td>
</tr>
<tr>
<td>46. Type names without static type checking already improve the usability of APIs (as long as the type names are correct): an empirical study</td>
<td>(Spiza &amp; Hanenberg, 2014)</td>
</tr>
<tr>
<td>47. Unit tests as API usage examples</td>
<td>(Nasehi &amp; Maurer, 2010)</td>
</tr>
<tr>
<td>48. Usability analysis of the channel application programming interface</td>
<td>(Brown, 2003)</td>
</tr>
<tr>
<td>49. Usability evaluation for enterprise SOA APIs</td>
<td>(J. K. Beaton et al., 2008)</td>
</tr>
<tr>
<td>50. Usability Evaluation of Configuration-Based API Design Concepts</td>
<td>(Scheller &amp; Kühn, 2013)</td>
</tr>
<tr>
<td>51. Usability Implications of Requiring Parameters in Objects’ Constructors</td>
<td>(Stylos &amp; Clarke, 2007)</td>
</tr>
<tr>
<td>52. Usability challenges for enterprise service-oriented architecture APIs</td>
<td>(J. Beaton et al., 2008)</td>
</tr>
<tr>
<td>53. Useful, But Usable? Factors Affecting the Usability of APIs</td>
<td>(Zibran et al., 2011)</td>
</tr>
<tr>
<td>54. Using concept maps to evaluate the usability of APIs</td>
<td>(Gerken et al., 2010)</td>
</tr>
<tr>
<td>55. Using the cognitive dimensions framework to evaluate the usability of a class library</td>
<td>(Clarke &amp; Becker, 2003)</td>
</tr>
<tr>
<td>56. What Makes APIs Hard to Learn? Answers from Developers</td>
<td>(Robillard, 2009)</td>
</tr>
</tbody>
</table>
Appendix B

Iteration 1: Social-interaction for Nao
B.1 API Documentation

This section details the documentation given to users in the first APIs user evaluation (Chapter 6), it has been reformatted to fit in this thesis.

The interfaces in the API are separated into 5 parts: Environment, Objects, Queries, StateMachine and Events.

The Environment is used to get the objects in the robot's environment. The Robot object has functions that can be used to make it perform actions, whilst other objects have functions that can be used to get information. To filter these objects, we use Queries, for example we could filter objects by type or what they say to other objects. The StateMachine is used to tie Queries and the robot's actions together by using events that are driven by Queries.

The rest of this page lists the functions used in this API, gives examples of their use and explains them.

B.1.1 Environment

The Environment object contains the objects in the robot's environment, including the robot itself. These are encapsulated by the two attributes: objects and robot. The objects attribute references a constantly maintained list of objects in the robot's environment. The robot attribute references a Robot instance that encapsulates the actions of a robot.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>robot</td>
<td>Robot</td>
<td>A robot instance.</td>
</tr>
<tr>
<td>objects</td>
<td>list of Objects</td>
<td>The list of objects in the robot's environment</td>
</tr>
</tbody>
</table>

env = Environment()
robot = env.robot
objects = env.objects

B.1.2 Objects

These are the objects in the robots environment, including the robot itself.
Object:

The base Object to subclass when creating new types of Objects. It contains core functions that are inherited by subclasses.

<table>
<thead>
<tr>
<th>Function</th>
<th>Distance_to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>other (Object)</td>
</tr>
<tr>
<td>Returns</td>
<td>float</td>
</tr>
<tr>
<td>Example</td>
<td><code>robot.distance_to(objects[0])</code></td>
</tr>
<tr>
<td>Explanation</td>
<td>Finds the distance from one object to another, e.g. the distance from the robot to another object.</td>
</tr>
</tbody>
</table>

Robot:

Represents a robot and encapsulates the actions it can perform. A subclass of Object. See here (next heading) for the list of gestures the robot can perform and how to markup the text it speaks with gestures.

<table>
<thead>
<tr>
<th>Function</th>
<th>say_to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>text (String), audience (Object, Queryable)</td>
</tr>
</tbody>
</table>
| Example                         | `robot.say_to('Hello', people)`  
|                                 | `robot.say_to('<wave> Hello </wave>', people)` |
| Explanation                     | Makes the robot speak to a person or a group of people. Once the robot has made eye contact it will begin synthesizing the text given by the text parameter. If more than one person is supplied by the audience parameter, then whenever a new sentence is reached, the robot will change its gaze to another person.  
|                                 | The text supplied to the text parameter can optionally be marked up with gesture tags. A gesture tag denotes where a particular gesture starts and where it ends with respect to the text the robot will synthesize. |

<table>
<thead>
<tr>
<th>Function</th>
<th>add_meaning_to_utterance_map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>meaning (Enum), utterances (list), context (None, object)</td>
</tr>
</tbody>
</table>
### Example

```
meanings = Enum('greeting', 'insult')
robot.add_meaning_to_utterance_map(meaning.greeting, ['hello', 'hi', 'howdy'])
```

### Explanation

Associates a particular meaning to a list of possible utterances that all have the same meaning.

---

### Gestures:

The robot can perform a number of gestures. The following list details the gestures the robot can perform and the gesture tags used to markup the text it will synthesise.

- Wave: `"<wave> Hello human </wave>"`
- Put hands on hips: `"<hips> I am angry with you </hips>"`
- Point arm to right: `"<point-right> look at that over there </point-right>"`
- Point arm to left: `"no that <point-left> thing looks more interesting </point-left>"`
- Change eye colour to red: `"<red-eyes> I am the start of the robopocalypse </red-eyes>"`
- Change eye colour to blue: `"<blue-eyes> maybe not </blue-eyes>"`

---

### Person:

Represents a person and encapsulates the attributes we can query about them. A subclass of Object.

<table>
<thead>
<tr>
<th>Function</th>
<th>said_to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>meaning (Enum), other (Object)</td>
</tr>
<tr>
<td>Returns</td>
<td>bool</td>
</tr>
</tbody>
</table>
| Example   | meanings = Enum('one', 'two')
            | person.said_to(meanings.one, robot) |
| Explanation| Identify if a person said an utterance with a specific meaning to someone else (a robot or another person). In theory this is detailed enough to filter out: if a person said an utterance to another person (not the robot) and the robot talking at the same time as the people and its speech being detected. |
### B.1.3 Queries:

Queries are used to filter the list of objects in the robot's Environment. An API based on Microsoft’s LINQ called Python-ASQ is used to write queries. Typical queries include: finding objects of a particular type, ordering objects by distance to the robot and finding the specific person/people that said an utterance with a particular meaning.

Some important query functions and examples of their use in HRI are shown below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>iterable (iterable)</td>
<td>query(objects)</td>
<td>Creates a query, e.g. a query that can be run on the environments objects list and returns a Queryable object</td>
</tr>
<tr>
<td>of_type</td>
<td>classinfo (cls)</td>
<td>people = query(objects).of_type(Person)</td>
<td>Selects elements if they are of a certain type, e.g. selecting only people. Deferred execution.</td>
</tr>
<tr>
<td>where</td>
<td>predicate (lambda)</td>
<td>query(objects).where(lambda o: o.distance_to(robot) &lt; 1)</td>
<td>Selects elements if they match a predicate, e.g. finding objects within 1m of the robot. Deferred execution.</td>
</tr>
<tr>
<td>order_by</td>
<td>key (lambda)</td>
<td>query(objects).order_by(lambda o: o.distance_to(robot))</td>
<td>Orders elements by a key, e.g. ordering all objects by their distance to the robot. Deferred execution.</td>
</tr>
</tbody>
</table>
### Appendix B – Iteration 1: Social interaction for Nao

#### B.1.4 StateMachine

The StateMachine class is used to tie the robot’s actions and the queries together to form autonomous human robot interaction.

<table>
<thead>
<tr>
<th>Function</th>
<th>init</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>environment (Environment)</td>
</tr>
<tr>
<td>Example</td>
<td><code>sm = StateMachine(env)</code></td>
</tr>
<tr>
<td>Explanation</td>
<td>Constructor. Takes an environment object as a variable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>start</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td><code>sm.start()</code></td>
</tr>
<tr>
<td>Explanation</td>
<td>Starts a StateMachine.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>add_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>state (State), first = False (bool)</td>
</tr>
</tbody>
</table>
| Example | `state = Hello()`  
`sm.add_state(state)` |
| Explanation | Adds a State to the StateMachine. |

#### B.1.5 State:

Subclass State to create your own States. Override the state’s execute method and put your robot actions there to make the robot perform actions.

<table>
<thead>
<tr>
<th>Function</th>
<th>init</th>
</tr>
</thead>
</table>
| Example | `greet = Greet()`  
`listen = Listen()` |
<table>
<thead>
<tr>
<th>Explanation</th>
<th>Constructor.</th>
</tr>
</thead>
</table>

**Function**: add_transition  
**Parameters**: event (Event), destination (State), id = None

**Example**
```
ev = QueryEvent(objects)  
greet.add_transition(ev, listen)  
listen.add_transition(ev, greet, id = 1)
```

**Explanation**: Creates and adds an event based Transition to a State. For example, the first add transition creates a transition to listen that transitions when `ev` fires, the second add transition creates a transition to greet that transitions when `ev` fires - it is given a custom id of 1 to identify it.

**Function**: execute  
**Example**: e (EArgs)

**Explanation**: Override this method and include robot actions here.

**B.1.6 Events:**

The events used by a StateMachine.

**QueryEvent:**

<table>
<thead>
<tr>
<th>Function</th>
<th>init</th>
</tr>
</thead>
</table>
| **Parameters**: query (Queryable), guard = lambda data: len(data) > 0 (lambda)
| **Example**: QueryEvent(objects)  
QueryEvent(objects, guard = lambda data: len(data) > 2)

**Explanation**: A query driven event that fires when the data returned by the query makes the guard lambda expression return True. For example, the first query event will fire when the query returns more than one result is returned whereas the second will fire when more than 2 items are returned.
B.2 API Tutorial

This section details the tutorial given to users in the first APIs user evaluation (Chapter 6), it has been reformatted to fit in this thesis.

B.2.1 Overview

For this tutorial we are going to use a pre-defined scenario. In this scenario, the robot - Nao, waits until he sees a person, greets them, listens and responds based on what the person said. You can either greet, insult or ignore Nao; he will respond differently in each case.

Here is the code for the scenario:

```python
#!/usr/bin/env python
import roslib
roslib.load_manifest('nao_apps')
from hri_api.apps import StateMachine,
State, TimeoutEvent, Transition, QueryEvent
from hri_api.objects import Person, query
from hri_api.servers import Environment
from enum import Enum

'' Create Environment: it contains the Robot and all other Objects ''
env = Environment()
robot = env.robot
objects = env.objects

'' Associate meanings with utterances ''
meanings = Enum('greet', 'insult')
robot.add_meaning_to_utterances_map(meanings.greet,
["hello", "hi"])
robot.add_meaning_to_utterances_map(meanings.insult,
["stupid robot", "shut up"])''

'' Create & add object queries ''
people = query(objects).of_type(Person)
env.add_query(people)

'' Create StateMachine ''
sm = StateMachine(env)

'' Define States ''
class WaitForPerson(State):
    def __init__(self):
        State.__init__(self)

    def create_transitions(self, next_state):
        one_or_more_people = QueryEvent(people)
        self.add_transition(one_or_more_people, next_state)

class Greet(State):
```
```python
def __init__(self):
    State.__init__(self)

def create_transitions(self, next_state):
    self.next_state = next_state

def execute(self, e):
    robot.say_to("Hey you there!", people)
    return Transition(self.next_state)

class Listen(State):
    def __init__(self):
        State.__init__(self)

def create_transitions(self, next_state):
    p_greeted_query = people.where(lambda p:
        p.said_to(meanings.greet, robot))
    p_insulted_query = people.where(lambda p:
        p.said_to(meanings.insult, robot))
    p_greeted_event = QueryEvent(p_greeted_query)
    p_insulted_event = QueryEvent(p_insulted_query)
    p_timeout_event = TimeoutEvent(4.5)
    self.add_transition(p_greeted_event, next_state,
        id = "person_greeted")
    self.add_transition(p_insulted_event, next_state,
        id = "person_insulted")
    self.add_transition(p_timeout_event, next_state,
        id = "timeout")

class Respond(State):
    def __init__(self):
        State.__init__(self)

def create_transitions(self, next_state):
    self.next_state = next_state

def execute(self, e):
    if e.id == "person_greeted":
        robot.say_to("Hello, what are you doing loitering around me?", people)
    elif e.id == "person_insulted":
        robot.say_to("What a nasty person!", people)
    elif e.id == "timeout":
        robot.say_to("Why so silent!", people)
    robot.wait(3.0)
    return Transition(self.next_state)

''' Initialize States '''
wait_for_person = WaitForPerson()
greet = Greet()
listen = Listen()
respond = Respond()
```
B.2.2 Running the code

Execute the following command from the terminal to see how it makes Nao behave:

```
user@user-1:~$ roslaunch nao_apps tutorial.launch
```

B.2.3 Step by step explanation

This part of the tutorial explains what the above code does. We first initialise an Environment object, which maintains a list of objects in the robot's environment (stored in the objects attribute). It also has a reference to a Robot object, which you can use to make Nao perform actions.

```
''' Create Environment: it contains the Robot and all other Objects '''
env = Environment()
robot = env.robot
objects = env.objects
```

Nao can understand and interpret the things people say. To enable this we associate specific meanings with potential utterances people could say to Nao. An Enum represents the meanings the robot can understand. In this case, Nao can understand two things: a greeting and an insult. We then associate the meanings with utterances using the add_meaning_to_utterances_map method. For example, in this case we have classified "hello" and "hi" as a type of greeting and "stupid robot" and "shut up" insults.

```
''' Associate meanings with utterances '''
meanings = Enum('greet', 'insult')
```
A basic part of human robot interaction is knowing what objects are people so that the robot can interact with them. To accomplish this we use a Python-ASQ Language Integrated Query (Smallshire, n.d.) that selects all objects of type Person. The attribute people has been assigned the query that will find all of the people the robot can interact with. Lastly, queries need to be added to the Environment with the add_query function.

```python
''' Create & add queries '''
people = query(objects).of_type(Person)
env.add_query(people)
```

A StateMachine is defined to tie the objects, queries and robots actions together to form autonomous human robot interaction.

```python
''' Create StateMachine '''
sm = StateMachine(env)
```

Next, States are defined. The first State in this scenario is called WaitForPerson which has been programmed to transition to another State when one or more people are detected. This logic is setup in the create_transitions method (called at the end of the file). A transition to next_state is added, it will transition the StateMachine to next_state when a QueryEvent fires. The QueryEvent in question, one_or_more_people, fires when the query people returns one or more item.

```python
''' Define States '''
class WaitForPerson(State):
    def __init__(self):
        State.__init__(self)

    def create_transitions(self, next_state):
        one_or_more_people = QueryEvent(people)
        self.add_transition(one_or_more_people, next_state)
```

The Greet State is where we make Nao say "Hey you there!" using the Robot objects say_to function. say_to has two parameters: text and audience. Nao will synthesize the
speech given by the text argument and gaze at the people returned by the audience query. If the text argument has more than one sentence and the audience query returns more than one person, Nao will change his gaze from one person to another when he reaches a new sentence.

Robot actions are always specified in an overridden execute method, as shown below. You should also note that transitions are not always event based, in this example, the StateMachine transitions when a Transition is returned at the end of the execute method.

class Greet(State):
    def __init__(self):
        State.__init__(self)
    
def create_transitions(self, next_state):
        self.next_state = next_state
    
def execute(self, e):
        robot.say_to("Hey you there!", people)
        return Transition(self.next_state)

The Listen State encapsulates the logic for listening to what people say. Two queries are used to find the people that said a phrase to the robot with a particular meaning. p_greeted_query finds the people who said a 'greeting' to the robot, whilst p_insulted_query finds the people that said an 'insult' to the robot. The queries are passed to QueryEvents, which are used to fire particular transitions.

A special event type, a TimeoutEvent is used to transition the StateMachine out of the current state if no response is heard for a certain period of time. Note that the three transitions in Listen, all transition to the same State - next_state. To distinguish between them, each transition is given its own id.

class Listen(State):
    def __init__(self):
        State.__init__(self)
    
def create_transitions(self, next_state):
        p_greeted_query = people.where(lambda p:
                                           p.said_to(meanings.greet, robot))
        p_insulted_query = people.where(lambda p:
                                          p.said_to(meanings.insult, robot))

        p_greeted_event = QueryEvent(p_greeted_query)
        p_insulted_event = QueryEvent(p_insulted_query)
        p_timeout_event = TimeoutEvent(4.5)
The last State, Respond, processes the results from the Listen State. The robot says a different thing depending on what the Listen State detected they said. The id of the transition that fired to reach a State is stored in the 'e' parameter of the execute method. This is used to identify what the people said so the robot can say something different based on what they said.

```python
class Respond(State):
    def __init__(self):
        State.__init__(self)

    def create_transitions(self, next_state):
        self.next_state = next_state

    def execute(self, e):
        if e.id == "person_greeted":
            robot.say_to("Hello, what are you doing loitering around me?", people)
        elif e.id == "person_insulted":
            robot.say_to("What a nasty person!", people)
        elif e.id == "timeout":
            robot.say_to("Why so silent?", people)

        robot.wait(3.0)

        return Transition(self.next_state)
```

After the States are declared, they are instantiated:

```python
''' Initialize States '''
wait_for_person = WaitForPerson()
greet = Greet()
listen = Listen()
respond = Respond()
```

Once the States have been instantiated, their create_transitions methods are called to wire the transitions between each State together.
''' Create State Transitions '''
wait_for_person.create_transitions(greet)
greet.create_transitions(listen)
listen.create_transitions(respond)
respond.create_transitions(wait_for_person)

Lastly, each State is added to the StateMachine, a starting State is specified (given by the variable first) and the StateMachine is started.

''' Add States to StateMachine '''
sm.add_state(wait_for_person, first = True)
sm.add_state(greet)
sm.add_state(listen)
sm.add_state(respond)

''' Start StateMachine '''
sm.start()

B.3 Tasks

This section details the tasks given to users in the first APIs user evaluation (Chapter 5), it has been reformatted to fit in this thesis.

B.3.1 Overview

The set of tasks below progressively alter the scenario described in the Tutorial () to make a game where Nao asks a person to guess what number he is thinking of.

A brief overview of how the scenario should work is as follows: Nao greets a person and asks them what number he is thinking of. You can generate a random number from 1-3 to represent this number. Nao then listens for what the person says (one, two or three). Compare the randomly generated number to what the person said. If the numbers match then the person they were correct, if the numbers don’t match tell them they were wrong. Always talk to the closest person.

B.3.2 Task 1: Make Nao speak

In the Greet State, after Nao says "Hey you there!", add another command that makes him say "What number am I thinking of?" to all people:
robot.say_to("What number am I thinking of?", people)

Test your program, you will need to stand in front of Nao for him to speak to you. Nao should look at you, say "Hey you there!" and then say "What number am I thinking of?". Run the following command to start the program:

user@user-1:~$ roslaunch nao_apps tasks.launch

B.3.3 Task 2: Make Nao speak & gesture simultaneously

Nao can gesture and speak simultaneously. Edit the command in the Greet State where Nao says "Hey you there!" so that he waves for the duration of the speech:

robot.say_to("<wave> Hey you there! </wave>", people)

Test your program. Nao should look at you and wave while he says "Hey you there!". Run the following command to start the program:

user@user-1:~$ roslaunch nao_apps tasks.launch

Nao can perform more gestures, see gestures for more information.

B.3.4 Task 3: Write a query

After the query that finds the people in the robots environment, add a query that finds the person closest to the robot:

closest_person = people.order_by(lambda p:
    p.distance_to(robot)).take()

Change all of the say_to commands so that the robot now only ever talks to the closest_person.

robot.say_to("<wave> Hey you there! </wave>", people)

too:
Test your program. Run the following command to start it:

```
user@user-1:~$ roslaunch nao_apps tasks.launch
```

### B.3.5 Task 4: Change the utterances Nao can hear

People should be able to respond to Nao's statement "What number am I thinking of" by speaking a number from 1 - 3. Users can say different words to communicate the same idea, e.g. a user could say "one" or "number one" to communicate his intent to guess the number 1. You need to add meanings for each number Nao should listen for, and associate each meaning with utterances people can say to communicate them.

For instance, to associate the utterances "one" and "number one" with the meaning "one", modify your code like so:

```python
meanings = Enum('greet', 'insult', 'one')
robot.add_meaning_to_utterances_map(meanings.greet, ['hello', 'hi'])
robot.add_meaning_to_utterances_map(meanings.insult, ['stupid robot', 'shut up'])
robot.add_meaning_to_utterances_map(meanings.one, ['one', 'number one'])
```

Add meanings and utterances for the rest of the numbers people can guess.

### B.3.6 Task 5: Add queries, events and transitions to process what Nao hears

Alter the Listen State to transition to another State when people answer with numbers from 1-3. For example, the following code illustrates how to make a StateMachine transition on an event. In this case a QueryEvent searches for people that said one to the robot, if more than one person is found a Transition will fire to next_state.

```python
one_query = people.where(lambda p: p.said_to(meanings.one, robot))
one_event = QueryEvent(one_query)
sel.add_transition(one_event, next_state, id = 1)
```

Add a queries, events and transitions for the rest of the numbers people can guess.
B.3.7 Task 6: Make Nao respond to people’s answers

Edit the Respond State so that Nao replies to people’s answers. The only requirements are:

- Nao must inform the user if they guessed correctly.
- Nao must inform the user if they guessed incorrectly.

Tip 1: you can find out what Event/Transition fired with the e.id parameter in the execute function.

Tip 2: this is how you choose a random number between 1 and 3 in Python:

```python
answer = random.randint(1, 3)
```

B.4 Questionnaire

This section details the post task Cognitive Dimensions questionnaire given to users in the first APIs user evaluation (Chapter 5), it has been reformatted to fit in this thesis.

B.4.1 Section 1 – Background information

Q1. What is your working/academic background? ..............................................................
...........................................................................................................................................

Q2. Do you have any experience programming robots? (If Yes proceed to Q3, if No, proceed to Q6)

☐ Yes
☐ No

Q3. If you chose yes for Q2, how long have you been programming robots for?...............
...........................................................................................................................................

Q4. If you chose yes for Q2, do you consider yourself proficient at programming robots?..
...........................................................................................................................................
Q5. If you chose yes for Q2, what robotics development software have you used? (choose multiple) .................................................................
...........................................................................................................................................................................................................................................................................................................
...........................................................................................................................................................................................................................................................................................................

☐ Choregraphe
☐ Microsoft Robotics Developer Studio
☐ NAOqi
☐ Player
☐ ROS
☐ Other(s), please specify:

Q6. What other forms of software development do you have experience in? ......................
...........................................................................................................................................................................................................................................................................................................
...........................................................................................................................................................................................................................................................................................................

Q7. How long have you been developing software for? .................................................................
...........................................................................................................................................................................................................................................................................................................
...........................................................................................................................................................................................................................................................................................................

Q8. Do you consider yourself proficient at software development? ....................................................
...........................................................................................................................................................................................................................................................................................................
...........................................................................................................................................................................................................................................................................................................

Q9. Do you have any experience using Python to write software? (If Yes proceed to Q10, if No, proceed to Q11)

☐ Yes
☐ No

Q10. If you chose yes for Q9, do you consider yourself proficient at programming with Python? ...........................................................................................................................................................................................................................................................................................................
...........................................................................................................................................................................................................................................................................................................
...........................................................................................................................................................................................................................................................................................................

Q11. What other programming languages do you use? ........................................................................
...........................................................................................................................................................................................................................................................................................................
B.4.2 Section 2 – Definitions

You might need to think carefully to answer the questions in the next sections, so we have provided some definitions and an example to get you started:

<table>
<thead>
<tr>
<th>Product</th>
<th>The <em>product</em> is the ultimate reason why you are using the notational system – what things happen as an end result, or what things will be produced as a result of using the notational system. This event or object is called the product. Any product that needs a notation to describe it usually has some complex structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation</td>
<td>The <em>notation</em> is how you communicate with the system – you provide information in some special format to describe the end result that you want, and the notation provides information that you can read. Notations have a structure that corresponds in some way to the structure of the product they describe. They also have parts (components, aspects etc.) that correspond in some way to parts of the product. Notations can include text, pictures, diagrams, tables, special symbols or various combinations of these. Some systems include multiple notations. These might be quite similar to each other – for example when using a typewriter, the text that it produces is just letters and characters, while the notation on the keys that you press tells you exactly how to get the result you want. In other cases, a system might include some notations that are hard for humans to produce or to read. For example when you use a telephone the notation on the buttons is a simple arrangement of digits, but the noises you hear aren’t so easy to interpret (different dialling tones for each number, clicks, and ringing tones). A telephone with a display therefore provides a further notation that is easier for the human user to understand.</td>
</tr>
</tbody>
</table>

To review how we intend to use these terms, consider the example of typing business letters on a word processor. The *product* of using the word processor is the printed letter on paper. The *notation* is the way that the letter looks on the screen – on modern word processors it looks pretty similar to what gets printed out, but this wasn't always the case.
B.4.3 Section 3 – Questions about the main notation

<table>
<thead>
<tr>
<th>Question</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>How easy is it to see or find the various parts of the notation while it is being created or changed? What kind of things are more difficult to see or find? If you need to compare or combine different parts, can you see them at the same time?</td>
<td>Why?</td>
</tr>
<tr>
<td>When you need to make changes to previous work, how easy is it to make the change? What kind of things are more difficult to see or find? Which ones?</td>
<td>Why?</td>
</tr>
<tr>
<td>Does the notation a) let you say what you want reasonably briefly, or b) is it long-winded? What sorts of things take more space to describe?</td>
<td>Why?</td>
</tr>
<tr>
<td>What kind of things require the most mental effort with this notation? Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?</td>
<td>Why?</td>
</tr>
<tr>
<td>Do some kinds of mistake seem particularly common or easy to make? Which ones? Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?</td>
<td>Why?</td>
</tr>
<tr>
<td>How closely related is the notation to the result that you are describing? (Note that in a sub-device, the result may be part of another notation, rather than the end product). Which parts seem to be a particularly strange way of doing or describing something?</td>
<td>Why?</td>
</tr>
<tr>
<td>When reading the notation, is it easy to tell what each part is for in the overall scheme? Are there some parts that are particularly difficult to interpret? Which ones? Are there parts that you really don’t know what they mean, but you put them in just because it’s always been that way? What are they?</td>
<td>Why?</td>
</tr>
<tr>
<td>If the structure of the product means some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies are hidden? In what ways can it get worse when you are creating a particularly large description? Do these dependencies stay the same, or are there some actions that cause them to get frozen? If so, what are they?</td>
<td>Why?</td>
</tr>
</tbody>
</table>
How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not?
Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?
Can you try out partially-completed versions of the product? If not, why not?

Is it possible to sketch things out when you are playing around with ideas, or when you aren't sure which way to proceed? What features of the notation help you to do this?
What sort of things can you do when you don't want to be too precise about the exact result you are trying to get?

When you are working with the notation, can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first?
If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?

Where there are different parts of the notation that mean similar things, is the similarity clear from the way they appear? Please give examples.
Are there places where some things ought to be similar, but the notation makes them different? What are they?

Is it possible to make notes to yourself, or express information that is not really recognised as part of the notation?
If it was printed on a piece of paper that you could annotate or scribble on, what would you write or draw?
Do you ever add extra marks (or colours or format choices) to clarify, emphasise or repeat what is there already?

Does the system give you any way of defining new facilities or terms within the notation, so that you can extend it to describe new things or to express your ideas more clearly or succinctly? What are they?
Does the system insist that you start by defining new terms before you can do anything else? What sort of things?

Do you find yourself using this notation in ways that are unusual, or ways that the designer might not have intended?
If so, what are some examples?

After completing this questionnaire, can you think of obvious ways that the design of the system could be improved?
What are they? Could it be improved specifically for your own requirements?
Appendix C

Iteration 2: Social-interaction refactored
C.1 Tutorial

This section details the documentation given to users in the second APIs user evaluation (Chapter 6), it has been reformatted to fit in this thesis.

These tutorials describe how to get started programming human-robot interaction for the Nao humanoid robot. Make sure that Nao is on, connected to the network and in a stable standing pose. All of the examples described in this tutorial are located in the folder nao_hri/scripts.

Contents

C.1.1 Starting hri_api
C.1.2 Making Nao speak (Python)
C.1.3 Making Nao gesture (Python)
C.1.4 Making Nao gaze at people (Python)
C.1.5 Making Nao speak and gaze at a group of people simultaneously (Python)
C.1.6 Querying the World for objects Nao can sense (Python)
C.1.7 Listening to what people say (Python)

C.1.1 Starting hri_api

A number of background services need to be started before you can run your human-robot interaction programs on Nao. First, open a terminal window and start the rosmaster service:

```
user@pc:~$ roscore
... logging to /home/user/.ros/log/3a034616-42a2-11e4-969d-5
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://pc:54570/
ros_comm version 1.11.9

SUMMARY
======

PARAMETERS
* /rosdistro: indigo
  * /rosversion: 1.11.9
  ...
```
Then, in a new terminal tab, start the hri_api background services:

```
user@pc:$ roslaunch nao_hri nao_interaction_sim.launch
... logging to /home/user/.ros/log/d29dcda4-42ae-11e4-b129-b8ca3a81
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

**SUMMARY**

```
PARAMETERS
* /armature_name: Armature
* /blender_target_controllers: /home/user/catki...
* /launch_blender/blend_file: /home/user/catki...
* /launch_blender/python_script: /home/user/catki...
* /launch_blender/use_game_engine: False
* /nao_gaze_action_server/action_server_name: gaze
* /nao_gaze_action_server/axes: yz
```

---

**C.1.2 Making Nao speak (Python)**

Now that the background services have been started, lets learn how to make Nao speak.

**The code**

This is an example script that makes Nao speak. You can also find it in the nao_hri package under: nao_hri/scripts/hri_say_examples.py

```
#!/usr/bin/env python
# license removed for brevity
from nao_hri import Nao; import time

robot = Nao()

ah1 = robot.say("Hello")
robot.wait(ah1)

robot.say_and_wait("I'm a crazy robot")

ah2 = robot.say('A really annoying robot')
time.sleep(1)
robot.cancel(ah2)
```

**The code explained**

Now, let's break the code down.
#!/usr/bin/env python

Every hri_api script must have this declaration at the top. The first line makes sure your script is executed as a Python script.

from nao_hri import Nao

The nao_hri import is so that we can use the Nao robot class to program Nao.

robot = Nao()

This line defines a Nao robot instance.

ah1 = robot.say("Hello")
robot.wait(ah1)

This section of code makes Nao say "Hello" and then wait until it has finished speaking. The robot.say(text) function is what makes Nao speak. It is asynchronous and returns an action handle to uniquely identify the speaking action being performed. In line 7, we use the robot.wait(*action_handle) function to wait until Nao has finished speaking. The functions sans “and_wait” are useful to run multiple actions on the robot in parallel.

robot.say_and_wait("I'm a crazy robot")

This line makes Nao say “I’m a crazy robot”. The difference between the previous method of making Nao speak is that it blocks until the sentence has finished.

ah2 = robot.say('A really annoying robot')
time.sleep(1)
robot.cancel(ah2)

This section of code demonstrates how to cancel actions that the robot is performing. You simply pass the action handle returned by an asynchronous function to the
robot.cancel(*action_handles) function, which imediately stops the robot performing the action tied to the action handle.

C.1.3 Making Nao gesture (Python)

Now that we've made Nao speak, let's learn to make him gesture.

The Code

This is an example script that makes Nao gesture. You can also find it in the nao_hri package under: nao_hri/scripts/hri_gesture_examples.py

```
#!/usr/bin/env python
# license removed for brevity
from nao_hri import Nao, Gesture; import time

robot = Nao()

robot.gesture_and_wait(Gesture.HandsOnHips)

g1 = robot.gesture(Gesture.MotionLeft)
g2 = robot.gesture(Gesture.MotionRight)
robot.wait(g1, g2)
robot.gesture_and_wait(Gesture.WaveLarm, duration=10.0)
g3 = robot.gesture(Gesture.LarmDown)
g4 = robot.gesture(Gesture.RarmDown)
time.sleep(1)
robot.cancel(g3, g4)
```

The Code Explained

Now, let's break the code down.

```
#!/usr/bin/env python

As mentioned previously, every hri_api script must have this declaration at the top. The first line makes sure your script is executed as a Python script.

from nao_hri import Nao, Gesture
```
This time, as well as importing a Nao class, we have also imported a Gesture class. This is an enum that contains a list of all of the gestures Nao can perform.

```python
5 robot = Nao()
```

This line defines a Nao robot instance.

```python
7 robot.gesture_and_wait(Gesture.HandsOnHips)
```

This line makes Nao put his hands on his hips and wait until it has finished. This is enabled with the `robot.gesture_and_wait(gesture)` function, which takes a Gesture enum that specifies the type of gesture to perform.

```python
9 g1 = robot.gesture(Gesture.MotionLeft)
10 g2 = robot.gesture(Gesture.MotionRight)
11 robot.wait(g1, g2)
```

This section makes Nao put both of his arms up either side at the same time; using a combination of the asynchronous `robot.gesture(gesture)` and `robot.wait(*action_handles)` functions.

```python
13 robot.gesture_and_wait(Gesture.WaveLarm, duration=10.0)
```

This line makes Nao wave using his left arm for 10 seconds (specified with the `duration` parameter). In the previous examples, no duration was specified, so the gesture's were performed with their default durations. The default durations are specified in: `nao_hri/src/nao_hri/nao.py`

```python
15 g3 = robot.gesture(Gesture.LarmDown)
16 g4 = robot.gesture(Gesture.RarmDown)
17 time.sleep(1)
18 robot.cancel(g3, g4)
```

Lastly, this section shows how to cancel gestures. In this example Nao begins to put his arms down by his side, then after waiting for a second we cancel the two running gestures with the `robot.cancel(*action_handles)` function.
C.1.4 Making Nao gaze at people (Python)

We now learn how to make Nao gaze at things, in particular parts of a peoples bodies.

The Code

This is an example script that makes Nao gaze at different parts of a peoples bodies. You can also find it in the nao_hri package under: nao_hri/scripts/hri_gaze_examples.py

```python
#!/usr/bin/env python
# license removed for brevity
from hri_api.entities import Person
from nao_hri import Nao
robot = Nao()
person1 = Person(1)
person2 = Person(2)
person3 = Person(3)
g1 = robot.gaze(person1.head)
robot.wait(g1)
robot.gaze_and_wait(person2.torso)
robot.gaze_and_wait(person3.head, speed=0.8)
g2 = robot.gaze(person1.head)
time.sleep(1)
robot.cancel(g2)
```

The Code Explained

Now, let's break the code down.

To make Nao interact with people, we need to import a Person class from the hri_api.entities module.

As usual we create a Nao robot instance.
This snippet creates three instances of the Person class. In section 1.1 when you ran the command `roslaunch nao_hri nao_interaction_sim.launch`, it started ROS nodes that broadcast simulated coordinates for these three people.

```python
person1 = Person(1)
person2 = Person(2)
person3 = Person(3)
```

We now make Nao gaze at the first person's head with the `robot.gaze_and_wait(target)` function.

This function returns when the robot has succeeded gazing at the specified target. In this example, Nao first gazes at person1's head, then at person2's torso and lastly at person3's left hand.

As can be seen on line 15, you can also control the speed that the robot gazes at via the speed parameter. This is a normalized value between the range: $0.0 < \text{speed} \leq 1.0$.

### C.1.5 Making Nao speak and gaze at a group of people simultaneously (Python)

In this section we learn how to make Nao speak and gaze at people simultaneously. It is particularly useful if you have long sentences for the robot to speak.

#### The Code

This is an example script that makes Nao speak and gaze at a group of people simultaneously. You can also find it in the `nao_hri` package under: `nao_hri/scripts/hri_say_to_examples.py`

```python
#!/usr/bin/env python
# license removed for brevity
from hri_api.entities import Person
from nao_hri import Nao
import time
```
The Code Explained

Now, let's break the code down.

As usual we create an instance of the Nao robot class. We then create a Person and a list of people.

The say_to function is used to make the robot gaze and speak to a person or a group of people. When there is one person in the audience, the robot first gazes at that person (e.g. person1); when it has made eye contact it speaks (e.g. s’aying Hello who are you?’ in this example). This particular example is done through a combination the asynchronous function robot.say_to(text, audience) and the blocking robot.wait(*action_handles) function.
16 'in this decade, not because its easy, but '
17 'because its hard, because that goal will '
18 'serve to organize and measure the best of '
19 'our energies and skills, because that challenge '
20 'is one that we are willing to accept, one we are '
21 'unwilling to postpone, and one which we intend to '
22 'win.', people)

Lines 15 - 22 illustrate a slightly more complex example, which is where we want to make the robot speak to not just one person, but a group of people. This is also accomplished with the robot.say_to(text, audience) or robot.say_to_and_wait(text, audience) functions. The key variable is the audience parameter of the say_to functions, which can be a single Person object, a list of Person objects or a Query that contains Person objects (explained later on). In this case, Nao will initially make eye contact with the person in the people list that is closest to itself. Once Nao has made eye contact with that person, it will start speaking. Whenever Nao reaches a new sentence (denoted by punctuation: .,?!), it chooses a random person from the list and gazes at them.

C.1.6 Querying the World for objects Nao can sense (Python)

In the previous examples, we created the Person objects that Nao interacted with. This is perfect for testing human-robot interaction scripts, but when real people interact with Nao, the Person objects and their coordinates need to be created dynamically via the robots perception systems. The following examples assume that you run two launch files, which are different to what we used in the previous examples:
The Code

This is an example script that blocks until at least one person is detected. Then we find the person closest to Nao and make Nao greet that person and wave at them. You can also find it in the `nao_hri` package under: `nao_hri/scripts/hri_perception_examples.py`

```python
#!/usr/bin/env python
# license removed for brevity

import rospy
from hri_api.entities import Person, World
from hri_api.query import Query
from nao_hri import Nao, Gesture

world = World()
robot = Nao()
people = Query(world).select_type(Person)
closest_person = people.sort_decreasing(lambda p: p.distance_to(robot)).take(1)
rate = rospy.Rate(2)
while not rospy.is_shutdown():
    result = closest_person.execute()
    if len(result) > 0:
        break
    rate.sleep()

ah1 = robot.say_to('Hello I can see you!', closest_person)
ah2 = robot.gesture(Gesture.WaveLarm)
robot.wait(ah1, ah2)
```

The Code Explained

Now, let's break the code down.

In this example we import several modules and classes we haven't seen yet, including: `rospy`, which contains Python helper classes and functions for ROS; the `World` class,
which contains the objects detected in the robots environment and the Query class, which is used to filter objects in the world.

```python
8   world = World()

This line creates a World object.

10  people = Query(world).select_type(Person)

This line is the first example of how to write a Query. Queries are similar in concept to Microsoft's LINQ and are a fork of the library Python ASQ. Queries are created by chaining methods together to create complex yet readable queries.

This particular Query selects all of the Person objects from the world. Queries are initialised with the Query(iterable) constructor, which takes an iterable as input (e.g. a list). We then call the method .select_type(cls) which selects objects that are instances of a particular class, in our case, Person objects. Queries use deferred execution, i.e., they are only executed when the .execute() method is called.

```python
11  closest_person = people.sort_decreasing(lambda p: p.distance_to(robot)).take(1)
```

The next query builds on the query from the previous line by sorting all of the people objects by the distance of each person to the robot. This is accomplished with the .sort_decreasing(key) function, which sorts the objects in decreasing order. The order of the items is determined by the key parameter, an anonymous inline function (lambda). The lambda function we use to order the people objects is lambda p: p.distance_to(robot), it takes a person object as input (p) and returns the distance of that person to the robot (lambda p: p.distance_to(robot)). The values returned by these function calls are used as the keys for ordering the list of people. The last part of the query is the .take(n) function, which takes the first n items from the resulting query, in this case 1 item (.take(1)).

```python
13  rate = rospy.Rate(2)
14  while not rospy.is_shutdown():
```
This section of code loops at a rate of 2 times a second, executes the closest_person query (line 15) and breaks if the closet_person query returns 1 or more items (lines 17-18). In other words, we continue when 1 or more people are detected by Nao.

Finally, we make Nao gaze and say 'Hello I can see you!' to the closet_person via the .say_to(text, audience) function and also make Nao wave its left arm simultaneously with the robot.gesture(gesture) function. We wait for both actions to finish with the robot.wait(*action_handles) function.

C.1.7 Listening to what people say (Python)

In this last example we look at how to listen to the things people say when they talk to Nao.

1.7.1 The Code

This is an example script that makes Nao say back to you what he just heard. You can also find it in the nao_hri package under: nao_hri/scripts/hri_listen_examples.py
from nao_hri import Nao
import rospy
from threading import RLock

lock = RLock()
robot = Nao()

def i_heard_that(speech):
    with lock:
        if speech is not None:
            robot.say_and_wait(speech)
        else:
            robot.say_and_wait("I didn't hear you, speak up!")

robot.register_listen_callback(i_heard_that)

The Code Explained

This function makes Nao respond to speech. It takes a string as input (speech) and only executes the rest of the function when the lock object is free (line 11). This is to prevent the function being executed more than once simultaneously. When Nao detects speech, but doesn't recognise what was said, the result is None. Hence, we check to see if the speech string is None. If it isn't, we make Nao say the text in the speech variable. If it is None, we make Nao say "I didn't hear you, speak up!".
This line is very important, it registers the function we just created (i_heard_that) with the robot class as a listen callback. This means that whenever Nao hears speech, the function i_heard_that is called - the parameter speech contains the text representation of what Nao heard.

\begin{Verbatim}
18 rospy.spin()
\end{Verbatim}

This last line stops our program from exiting, until ctrl-c is pressed.

**C.2 Interview questions**

This section contains the interview questions used in the case study, it has been reformatted to fit in this thesis. Two of the questions are phrased using Clarke’s Cognitive Dimensions (Clarke, 2004) rather than the original Cognitive Dimensions (Green & Petre, 1996). The mapping between the definitions are: domain correspondence -> closeness of mapping; and work-step unit -> diffuseness and terseness.

**Improvements**

In what ways does the API take control away from you? In what ways would you like more control? *E.g. being able to change the volume, make pauses in speech.*

**Progressive evaluation**

What information is missing when you stop the program to check your progress? *E.g. error messages.*

**Error proneness**

Do you often write code and something unexpected happens? Do you often find yourself making small errors by accident?

**Domain correspondence**

How closely related is the API to the result you are describing? Why? Which parts of the API seem like a strange way of expressing something?

**Premature commitment**

When using the API, can you go about the job in any order you like, or does the API make you think ahead and make certain decisions first? If so, what decisions do you need to make in advance? What problems does this cause you?
Work-Step Unit
Does the amount of code required to program a scenario seem about right, too much or too little? What sorts of things take more space to program?

Role expressiveness
When reading code that uses the API, is it easy to tell what each section of code does, or are some parts difficult to interpret? Why?
Is it easy to know what classes and methods to use when writing code, or is it unclear what parts of the API to use when writing code?

Abstraction gradient
Does the API insist that you define new classes and functions before you can proceed? What types of things?
Are there particular parts of the API where being able to extend it would make programming your scenario easier? What are they?

Viscosity
When you need to make changes to previous work, how easy is it to make the change? Why?
Are there particular changes that are difficult to make? Which ones?

Visibility
How easy is it to see or find the various parts of the notation while it is being created or changed? What kinds of things are difficult to see or find?

Hidden dependencies
Do changes to one part of the API have unexpected consequences in another part?
Are some parts of the API hard to see or find?

Consistency
After learning part of the API, how much of the rest can you successfully guess? What parts?
Are there some parts of the API that do similar things, but are programmed in a different way?

Hard mental operations
What things require the most mental effort to use in the API?
When using the API, do some things seem difficult to work out in your head?
Appendix D
A taxonomy of social primitives
D.1 Supporting data

Table D.1 and Table D.2 provide supporting data for the abstraction levels found in different social-interaction programming tools and general robotics programming tools, as discussed in Chapter 3.1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Example</th>
<th>Source data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speech</td>
<td></td>
<td></td>
</tr>
<tr>
<td>What to say</td>
<td>“I am going to rub your arm. I am going to clean you. The doctor will be with you shortly” (Chen et al., 2011, p. 460)</td>
<td></td>
</tr>
<tr>
<td>How to say it</td>
<td>“Human speech contains three types of information: who the speaker is, what the speaker said, and how the speaker said it” (Fong et al., 2003, p. 156)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“Speech contains three types of information: who the speaker is, what the speaker said, and how the speaker said it” (Fong et al., 2003, p. 156)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“Language contains “prosody (the pitch, rhythm, and tempo of an utterance)” (Doupe &amp; Kuhl, 1999, p. 571)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“Parameters that govern the emotional content of speech are loudness, pitch (level, variation, range), and prosody” (Fong et al., 2003, p. 152)</td>
<td></td>
</tr>
<tr>
<td>Lip sync</td>
<td></td>
<td>(Li et al., 2009, p. 5012)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaze</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Body parts to use</td>
<td>Eyes, head</td>
<td>“These interactions can include conversation, eye contact and facial gestures.” (Ranatunga et al., 2011, p. 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Eyes: Pan and tilt, possibly supporting mutual gaze and joint attention” (Dautenhahn et al., 2009, p. 376)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Sakagami et al., 2002, p. 2480)</td>
</tr>
<tr>
<td>How to perform</td>
<td>Speed, acceleration</td>
<td></td>
</tr>
<tr>
<td>Where to target</td>
<td>Own body part, person’s body part or another object.</td>
<td>“The robot thus (1) alternates its gaze between a child’s face, the caregiver’s face, and sometimes a nearby toy” (Kozima et al., 2007, p. 391)</td>
</tr>
<tr>
<td>Blend with</td>
<td>E.g. nod and gaze simultaneously</td>
<td>“hand to face [and] looking down [and] no smile” (Dautenhahn et al., 2009, p. 380)</td>
</tr>
<tr>
<td>gestures</td>
<td></td>
<td>“looking to right [and] closed smile [and] hand to chin” (Dautenhahn et al., 2009, p. 380)</td>
</tr>
</tbody>
</table>

Continued…
### Appendix D – A taxonomy of social primitives

<table>
<thead>
<tr>
<th>Feature</th>
<th>Example</th>
<th>Source data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gesture</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type of gesture</td>
<td>Wave, poke at, arm raise, lip touch, peek-a-boo</td>
<td>“producing gestures such as waving, peek-a-boo etc.” (Dautenhahn et al., 2009, p. 377)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Thus, a sender directs a certain gesture, such as poke at, arm raise or lip touch, towards a particular recipient.” (Liebal &amp; Pika, 2005, p. 6)</td>
</tr>
<tr>
<td>How to perform</td>
<td>Magnitude, speed, repeat</td>
<td>“socially powerful movements such as slightly tilting the head (important for expressing more subtle emotions/personality traits such as shyness, cheekiness etc.)” (Dautenhahn et al., 2009, p. 377)</td>
</tr>
<tr>
<td>Where to target</td>
<td>Own body part, person’s body part or object</td>
<td>Dyadic and triadic gestures in primates (Liebal &amp; Pika, 2005, p. 6)</td>
</tr>
<tr>
<td>Synchronise with speech</td>
<td>Shake head while saying “you’re a naughty human”</td>
<td>Gesture and speech are synchronised with time (H. Sowden et al., 2013, p. 922)</td>
</tr>
<tr>
<td>Multiple simultaneously</td>
<td>Shake head and point simultaneously</td>
<td>“(e.g. smiles when KASPAR imitated human drumming)” (Dautenhahn et al., 2009, p. 386)</td>
</tr>
<tr>
<td>Blend overlapping</td>
<td>Blend wave and point</td>
<td></td>
</tr>
</tbody>
</table>

| **Facial expressions** | | |
| Type of expression | Surprised, happy, sad, smile, anger, frown, blinking, neutral | “different facial expressions (e.g. ‘surprised’, ‘happy’, ‘sad’ etc.)” (Dautenhahn et al., 2009, p. 387) |
| | | “facial expressions (e.g. smile …)”(Dautenhahn et al., 2009, p. 389) |
| | | “Actroid-F … can exhibit various facial expressions such as smile, anger and surprise.” (Yoshikawa et al., 2011, p. 2378) |
| | | “Eyelids: Blinking (full or partial, at various rates)” (Dautenhahn et al., 2009, p. 376) to allow [human] to individually interpret the expressions as ‘happy’, ‘neutral’, ‘surprised’ etc Dautenhahn et al., 2009, p. 376) |
| How performed | Magnitude, speed, repeat | “Eyelids: Blinking (full or partial, at various rates)” (Dautenhahn et al., 2009, p. 376) |
| Multiple simultaneously | Frown and smile simultaneously | |
| Blend overlapping | Blend pucker and smile | |

Continued…
## Appendix D – A taxonomy of social primitives

<table>
<thead>
<tr>
<th>Feature</th>
<th>Example</th>
<th>Source data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Touch</td>
<td>“a robotic nurse autonomously reached out, touched the participant's arm, moved across their arm, and then retracted.” (Chen et al., 2011, p. 457)</td>
<td></td>
</tr>
<tr>
<td>Type of touch</td>
<td>“instrumental touch) or to provide comfort (affective touch). (Chen et al., 2011, p. 457)</td>
<td></td>
</tr>
<tr>
<td>How to touch</td>
<td>Pressure, duration</td>
<td>“The robot then immediately moves its end effector downward until the force sensor on the wrist measures a force magnitude 2 N, indicating the end effector has made contact with the arm.” (Chen et al., 2011, p. 459) “We designed the arm trajectory so that the &quot;Initial&quot; action completed within approximately 7 seconds when tested on a lab member's arm.” (Chen et al., 2011, p. 459) “A bang-bang controller attempts to keep the force magnitude measured by the force sensor on the wrist between 1 and 3 N.” (Chen et al., 2011, p. 459) “As a safety precaution, the robot terminates the touching behavior if the force magnitude exceeds 30 N.” (Chen et al., 2011, p. 459) “Third, the robot attempted to keep the magnitude of the force against the participant's arm lower than 3 N.” (Chen et al., 2011, p. 460)</td>
</tr>
<tr>
<td>What to touch</td>
<td></td>
<td>“autonomously reached out, touched the participant's arm, moved across their arm, and then retracted” (Chen et al., 2011, p. 457)</td>
</tr>
</tbody>
</table>
### Table D.2 Robot perception primitive source data.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Example</th>
<th>Source data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model human features and their states</td>
<td>Skeleton model</td>
<td>(Fong et al., 2003)</td>
</tr>
<tr>
<td></td>
<td>Face model</td>
<td>(Fong et al., 2003)</td>
</tr>
<tr>
<td></td>
<td>Gaze (head and eyes)</td>
<td>Gaze: “[childs] Eye gaze (when directed at the robot).” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112)</td>
</tr>
<tr>
<td></td>
<td>Model multiple people</td>
<td>“Voice control: Coco can be operated by voice control. For example, the words “Quiz”, “Play”, or “Game” trigger the quiz function.” (Racky et al., 2011, p. 1091) “Finally there is high-level identifiable data like human Face, gesture, posture, staircase, door, and natural language sentences” (Sakagami et al., 2002, p. 2478)</td>
</tr>
<tr>
<td>Speech</td>
<td>What was said</td>
<td>“Human speech contains three types of information: who the speaker is, what the speaker said, and how the speaker said it” (Fong et al., 2003, p. 156)</td>
</tr>
<tr>
<td></td>
<td>Who spoke</td>
<td>A person or another robot “‘M vocalized non-words to Keepon’” (Kozima et al., 2007, p. 394)</td>
</tr>
<tr>
<td></td>
<td>How spoken</td>
<td>Volume, pitch, tempo, rhythm “Human speech contains three types of information: who the speaker is, what the speaker said, and how the speaker said it” (Fong et al., 2003, p. 156) Language contains “prosody (the pitch, rhythm, and tempo of an utterance)” (Doupe &amp; Kuhl, 1999, p. 571) “parameters that govern the emotional content of speech are loudness, pitch (level, variation, range), and prosody” (Fong et al., 2003, p. 152)</td>
</tr>
<tr>
<td></td>
<td>Directed at</td>
<td>The robot or another person “‘M vocalized non-words to Keepon’” (Kozima et al., 2007, p. 394) “In S19, after playing with Keepon for a while, IZ/m asked other children nearby, ‘Please take care of Keepon.’ ” (Kozima et al., 2007, p. 398)</td>
</tr>
<tr>
<td>Gaze</td>
<td></td>
<td>Gaze: “[childs] Eye gaze (when directed at the robot).” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) “M began acting exploratively with Keepon, such as looking into its eyes, waving her hand at it, and listening to its sound” (Kozima et al., 2007, p. 394) “However, N often glanced at the robot” (Kozima et al., 2007, p. 394)</td>
</tr>
</tbody>
</table>

Continued…
### Appendix D – A taxonomy of social primitives

<table>
<thead>
<tr>
<th>Feature</th>
<th>Example</th>
<th>Source data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Gesture</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type of gesture</strong></td>
<td>Poke at, arm raise, lip touch, handshake, hand circling, bye-bye, hand swing, high-hand, come here call</td>
<td>“Thus, a sender directs a certain gesture, such as poke at, arm raise or lip touch, towards a particular recipient.” (Liebal &amp; Pika, 2005, p. 6) “Recognized gestures include handshake, hand circling, bye-bye, hand swing, high-hand and come here call.” (Sakagami et al., 2002, p. 2481) “Some [children] mimicked [Keepons] emotive expressions by rocking and bobbing their own bodies” (Kozima et al., 2007, p. 391) “M began acting exploratively with Keepon, such as looking into its eyes, waving her hand at it, and listening to its sound” (Kozima et al., 2007, p. 394)</td>
</tr>
<tr>
<td><strong>Who gestured</strong></td>
<td>A specific person or another robot</td>
<td>“KASPAR just repeated the beats produced by the human partner” (Dautenhahn et al., 2009, p. 386) “When N performed a movement (bobbing, rocking, or bowing), soon Keepon mimicked her; then N made another, and Keepon mimicked her again.” (Kozima et al., 2007, p. 395)</td>
</tr>
<tr>
<td><strong>How gestured</strong></td>
<td>Magnitude, speed</td>
<td>“seems to encourage [the children] to reply with emphasised or bigger expressions in return, i.e. with a bigger smile, and bigger hand movements in imitation games etc.” (Dautenhahn et al., 2009, p. 380)</td>
</tr>
<tr>
<td><strong>Directed at</strong></td>
<td>The robot or another person</td>
<td>“M began acting exploratively with Keepon, such as looking into its eyes, waving her hand at it, and listening to its sound” (Kozima et al., 2007, p. 394) “when another child behaved violently with Keepon, S often hit or pretended to hit the child, as if he were protecting Keepon.” (Kozima et al., 2007, p. 395) Primates use both dyadic and triadic gestures which involve the targeting of a gesture at an individual or an external object respectively (Liebal &amp; Pika, 2005, p. 6).</td>
</tr>
<tr>
<td><strong>Facial expressions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type of expression</strong></td>
<td></td>
<td>“S asked Keepon, ‘Is this scary?’ showing bizarre facial expressions to the robot.” (Kozima et al., 2007, p. 395)</td>
</tr>
<tr>
<td><strong>Who expressed</strong></td>
<td>A specific person</td>
<td>“S asked Keepon, ‘Is this scary?’ showing bizarre facial expressions to the robot.” (Kozima et al., 2007, p. 395) “However, when Paro was given to her, she smiled and was willing to stroke Paro.” (Wada et al., 2005, p. 2)</td>
</tr>
<tr>
<td><strong>How expressed</strong></td>
<td>Magnitude, speed</td>
<td>“seems to encourage [the children] to reply with emphasised or bigger expressions in return, i.e. with a bigger smile, and bigger hand movements in imitation games etc.” (Dautenhahn et al., 2009, p. 380) “The investigator was able to recognize even subtle expressions of the child” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112)</td>
</tr>
<tr>
<td><strong>Directed at</strong></td>
<td>The robot, another person</td>
<td>“S asked Keepon, ‘Is this scary?’ showing bizarre facial expressions to the robot.” (Kozima et al., 2007, p. 395)</td>
</tr>
</tbody>
</table>

Continued…
## Appendix D – A taxonomy of social primitives

### Feature | Example | Source data
--- | --- | ---
**Touch**<br>**Type of touch**<br>Stroking, hitting, patting, sliding, scratching, kick | “allowing not only the identification of touch onset, but also its type and pressure” (Amirabdollahian et al., 2011, p. 5350)<br>“stroking [Keepons] head gently.” (Kozima et al., 2007, p. 397)<br>“from hitting it to feeding [Keepon]” (Kozima et al., 2007, p. 397)<br>“it can show a friendly gesture when patted on the back.” (Jayawardena et al., 2010, p. 5991)<br>“it is possible to detect sliding motion on the tactile sensor” (Mukai et al., 2010, p. 5999)<br>“[an animal] is able to understand that it is being petted, scratched or touched” (Stiehl et al., 2006, p. 1290)<br>“S violently kicked Keepon” (Kozima et al., 2007, p. 395) | “to permit the measurement of human contact with Paro” (Wada et al., 2005, p. 2)<br>“The robot, which was equipped with tactile sensing capabilities, could respond autonomously when touched” (Amirabdollahian et al., 2011, p. 5348)<br>“FS/m and TA/m, strongly hit Keepon’s head a couple of times” (Kozima et al., 2007, p. 398)<br>“allowing not only the identification of touch onset, but also its type and pressure” (Amirabdollahian et al., 2011, p. 5350)<br>“S violently kicked Keepon” (Kozima et al., 2007, p. 395)<br>“stroking [Keepons] head gently.” (Kozima et al., 2007, p. 397)<br>“it is being designed with a full body ‘sensitive skin’” (Stiehl et al., 2006, p. 1291)<br>“capacitive sensors … located on its forehead, divided into three segments.” (Shamsuddin et al., 2012, p. 190)<br>“when the child touched any part of the robot” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) |
**How touched**<br>Pressure, onset time | “NR/m hit Keepon’s head several times” (Kozima et al., 2007, p. 397)<br>“it is being designed with a full body ‘sensitive skin’” (Stiehl et al., 2006, p. 1291)<br>“capacitive sensors … located on its forehead, divided into three segments.” (Shamsuddin et al., 2012, p. 190)<br>“when the child touched any part of the robot” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) | “FS/m and TA/m, strongly hit Keepon’s head a couple of times” (Kozima et al., 2007, p. 398)<br>“allowing not only the identification of touch onset, but also its type and pressure” (Amirabdollahian et al., 2011, p. 5350)<br>“S violently kicked Keepon” (Kozima et al., 2007, p. 395)<br>“stroking [Keepons] head gently.” (Kozima et al., 2007, p. 397)<br>“it is being designed with a full body ‘sensitive skin’” (Stiehl et al., 2006, p. 1291)<br>“capacitive sensors … located on its forehead, divided into three segments.” (Shamsuddin et al., 2012, p. 190)<br>“when the child touched any part of the robot” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) |
**Where touched**<br>Head, forehead, belly, arm, leg, back | “NR/m hit Keepon’s head several times” (Kozima et al., 2007, p. 397)<br>“it is being designed with a full body ‘sensitive skin’” (Stiehl et al., 2006, p. 1291)<br>“capacitive sensors … located on its forehead, divided into three segments.” (Shamsuddin et al., 2012, p. 190)<br>“when the child touched any part of the robot” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) | “FS/m and TA/m, strongly hit Keepon’s head a couple of times” (Kozima et al., 2007, p. 398)<br>“allowing not only the identification of touch onset, but also its type and pressure” (Amirabdollahian et al., 2011, p. 5350)<br>“S violently kicked Keepon” (Kozima et al., 2007, p. 395)<br>“stroking [Keepons] head gently.” (Kozima et al., 2007, p. 397)<br>“it is being designed with a full body ‘sensitive skin’” (Stiehl et al., 2006, p. 1291)<br>“capacitive sensors … located on its forehead, divided into three segments.” (Shamsuddin et al., 2012, p. 190)<br>“when the child touched any part of the robot” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) |
**Who touched**<br>A specific person | “NR/m hit Keepon’s head several times” (Kozima et al., 2007, p. 397)<br>“it is being designed with a full body ‘sensitive skin’” (Stiehl et al., 2006, p. 1291)<br>“capacitive sensors … located on its forehead, divided into three segments.” (Shamsuddin et al., 2012, p. 190)<br>“when the child touched any part of the robot” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) | “FS/m and TA/m, strongly hit Keepon’s head a couple of times” (Kozima et al., 2007, p. 398)<br>“allowing not only the identification of touch onset, but also its type and pressure” (Amirabdollahian et al., 2011, p. 5350)<br>“S violently kicked Keepon” (Kozima et al., 2007, p. 395)<br>“stroking [Keepons] head gently.” (Kozima et al., 2007, p. 397)<br>“it is being designed with a full body ‘sensitive skin’” (Stiehl et al., 2006, p. 1291)<br>“capacitive sensors … located on its forehead, divided into three segments.” (Shamsuddin et al., 2012, p. 190)<br>“when the child touched any part of the robot” (Robins, Dautenhahn, Te Boekhorst, et al., 2005, p. 112) |
Appendix E
Forms
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


Parts used in Chapter 7 of thesis.

Nature of contribution by PhD candidate: Work and writing.

Extent of contribution by PhD candidate (%): 85%

CO-AUTHORS

Name | Nature of Contribution
--- | ---
Beryl Plimmer | Supervision + editing
Bruce MacDonald | Supervision + editing
John Hosking | Supervision + editing

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td></td>
<td>15/06/2015</td>
</tr>
<tr>
<td>Bruce MacDonald</td>
<td></td>
<td>15/06/15</td>
</tr>
<tr>
<td>John Hosking</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last updated: 25 March 2013
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


Used in Chapter 5 of thesis.

<table>
<thead>
<tr>
<th>Nature of contribution</th>
<th>Work and writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>85%</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td>Supervision + editing</td>
</tr>
<tr>
<td>Bruce MacDonald</td>
<td>Supervision + editing</td>
</tr>
<tr>
<td>John Hosking</td>
<td>Supervision + editing</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beryl Plimmer</td>
<td></td>
<td>15/06/2015</td>
</tr>
<tr>
<td>Bruce MacDonald</td>
<td>Click here</td>
<td>17/06/15</td>
</tr>
<tr>
<td>John Hosking</td>
<td>Click here</td>
<td></td>
</tr>
</tbody>
</table>

*Last updated: 23 March 2013*
# Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Acknowledgements.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


**Used in Chapter 2 of thesis.**

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Work + writing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>85%</td>
</tr>
</tbody>
</table>

## CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bruce MacDonald</td>
<td>Supervision + editing</td>
</tr>
<tr>
<td>John Hosking</td>
<td>Supervision + editing</td>
</tr>
</tbody>
</table>

## Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bruce MacDonald</td>
<td>[Signature]</td>
<td>17/06/2015</td>
</tr>
<tr>
<td>John Hosking</td>
<td>[Signature]</td>
<td>Click here</td>
</tr>
</tbody>
</table>

*Last updated: 25 March 2013*