Designing an API at an appropriate abstraction level for programming social robot applications

James Diprose[a], Bruce MacDonald[b], John Hosking[c], Beryl Plimmer[a]

[a]*Department of Computer Science, University of Auckland, New Zealand*
[b]*Department of Electrical & Computer Engineering, University of Auckland, New Zealand*
[c]*Faculty of Science, University of Auckland, New Zealand*

Abstract

Whilst robots are increasingly being deployed as social agents, it is still difficult to program them to interact socially. To create usable tools for programming these robots, tool developers need to know what abstraction levels are appropriate for programming social robot applications. We explore this through the iterative design and evaluation of an API for programming social robots. The results show that high level primitives, with a close mapping to social interaction, are suitable for programming social robot applications. However, the abstraction level should not be so high that it takes away too much control from programmers. This has the potential to enable programmers to produce high quality social robot applications with less programming effort.

Keywords: application programming interfaces, api, usability, design, cognitive dimensions, human robot interaction, social robot interaction, humanoid robot.

1. 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.

Figure 1 illustrates a range of different types of programmable social robots, including Keepon [1], Kismet [2] and the Geminoid series of robots [3, 4].

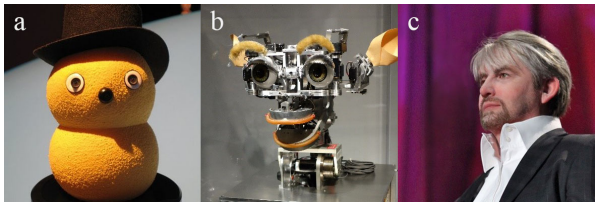

Figure 1: Social robots: (a) Keepon, (b) Kismet and (c) Geminoid-DK (c). Image sources (cropped): Keepon - [5]; Kismet - [6]; Geminoid-DK (background altered) - [7].

Kozima et al. [8] developed Keepon (Figure 1a) as a companion for interacting with children by displaying joint attention through gaze and expressing emotion by moving its body. Kismet (Figure 1b) was created by Breazeal & Scassellati [2] to explore the application of autonomous 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. Kismet has more realistic anthropomorphic features than Keepon. The Geminoid series of robots [3, 4] are replicas of real people (Figure 1c); their features and social interactions are very close to real humans. They can interact through speech, facial expressions and gestures [4] and were developed to study how life-like appearance and behaviour affects human-robot interaction. The Geminoids typically do not interact autonomously and are instead controlled by a human operator.

The overall objective of our work is to improve the usability of tools for programming social robots, such as Keepon [1], Kismet [2] and Geminoid [3, 4]. We began by developing a visual language to enable end users to program healthcare robot behaviour [9]. Whilst exploring this field, we discovered that it was hard for us (programmers) to program social robot applications; this appeared to be because the robot APIs we were using (NAOqi [10], Choregraphe [11] and ROS [12]) contained too many primitives with low abstraction levels. For example, a common social interaction task is pro-

gramming a robot to speak to a specific person by vocalising words and gazing at them [13]. Rather than simply specifying what the robot should say and who they should gaze at, we would have to give detailed commands for speech synthesis to make it speak, analyse results from a face detection algorithm to find the person's head and use joint control to make the robot gaze at the person's head.

According to the Cognitive Dimensions [14] it is important to choose an abstraction level carefully because it affects several other Cognitive Dimensions, including closeness of mapping, viscosity, visibility & juxtaposablity and hidden dependencies. Closeness of mapping can be improved or worsened by raising or lowering the abstraction level [14, pp. 38-39]; the ideal abstraction level depends on the task and audience being programmed for, some will need a lower levels and others higher. For instance, to write a mobile application, we use an API designed specifically for creating mobile applications (higher level) because it has a close mapping to the task, not assembly code or firmware (lower level) which has a distant mapping. Raising the abstraction level reduces viscosity and improves juxtaposablity, however, this also creates hidden dependencies because subroutines are now hidden from the user. Learnability, whilst not a Cognitive Dimension, is negatively affected if a programmer has to learn multiple abstractions to complete a task [14, p. 18].

In the field of social robot programming there is little evidence about what makes a good abstraction level for creating social robot applications. Tools used to program robot social interaction [11, 15, 16, 17, 18, 19, 20, 21, 22] do not give a clear indication of what abstraction level is appropriate as they collectively use a wide range of abstraction levels and some even mix different abstraction levels within the same tool (Section 2.2). Reported usability studies of these tools [15, 16, 18, 21] don't provide much data in the way of abstraction levels and how this affects the Cognitive Dimensions (Section 2.2).

This lead to the studies presented in this paper, which explore in more detail what abstraction level is appropriate for programming social interaction applications. Two research questions were used to guide this research:

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

RQ2. What are the trade-offs associated with using different abstraction levels to program robot social interaction?

To explore RQ1 & RQ2 in detail, we iteratively designed, implemented and evaluated an API for programming socially interactive robots. Each iteration examined the effect that different abstraction levels (RQ2) had on API usability and expressivity, resulting in an understanding of an abstraction level that is appropriate for programming socially interactive robot applications (RQ1).

The first iteration adopts a high abstraction level for programming a Nao humanoid robot in response to our experience with NAOqi [10] and ROS [12] described earlier. The second iteration shifts to a slightly lower abstraction level based on the results of a user study and stakeholder interview. It provides users with more control over the social interactions they can program.

The remainder of this paper is organized as follows: Section 2 provides an overview of related work, examining the abstraction levels found in tools for programming social robots and discussing their limitations. We then describe our approach to exploring what abstraction levels are appropriate for programming social robot applications (Section 3). Next, we describe the first iteration of this approach, the design and evaluation of a high level API for programming a Nao humanoid robot (Section 4). This lead to the second iteration, presented in Section 5, which describes the design and evaluation of a refactored API providing finer control social interaction and support for more robots. We then discuss the findings of this research and directions for future work (Section 6).

2. Background

This section reviews tools used to program socially interactive robots. It starts by providing an overview of the abstraction levels that can be used to program socially interactive robots (Section 2.1). Social interaction programming tools are then analysed (Section 2.2); this shows that there is no clear indication of what abstraction level is appropriate for programming social robot applications, as a wide range of abstraction levels are used and sometimes mixed within the same tool. It finishes with a discussion of the tools, their abstraction levels and implications for this research (Section 2.2.9).

2.1. Social interaction abstraction levels

To create a model of social interaction abstraction levels we decomposed robot social abilities into five abstraction levels based on a synthesis of robot software and hardware, including hardware primitives, algorithm primitives, social primitives, emergent primitives and

methods for controlling primitives. These five abstraction levels are illustrated in Figure 2 and explained and supported in the following paragraphs.

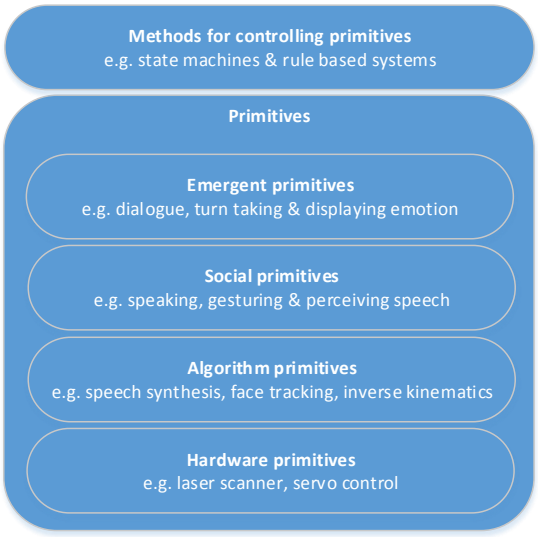

Figure 2: Abstraction levels of social interaction & examples

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 [23] and ROS [12].

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 [10] provides many primitives at this abstraction level, including interfaces to configure and retrieve data from each algorithm.

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. [20] 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 *methods for controlling primitives*, which are used to control primitives and create higher level behaviour, regardless of the specific type of primitive. For example, a finite state machine could be used to create dialogue between a human and a robot, by combining primitives for speaking and listening together. This is similar to the distinction that Siepmann [21] 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.

2.2. Social robot programming tools

A number of related tools and their abstraction levels are summarised in Table 1; detailed supporting data is available here [24]. The rest of this section discusses these tools and their abstraction levels.

2.2.1. Choregraphe

Choregraphe is an end-user visual programming environment for programming Nao humanoid robots [11], illustrated in Figure 3. It allows end users to program social interaction with a mixed set of abstraction levels, including hardware primitives, algorithm primitives, social primitives and methods for controlling primitives (Table 1). Users create social interaction by combining visual blocks together.

Most of Choregraphe's [11] 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 supported by Pot et al. [11] 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

Tool	Target audience	Notation	Abstraction levels					References
			Hardware primitives	Algorithm primitives	Social primitives	Emergent primitives	Methods for controlling primitives	
Choregraphe	Novice	Visual	✓	✓	✓		✓	[11]
Interaction Blocks	Novice	Visual				✓	✓	[15]
Interaction Composer	Novice & professional	Visual		✓	✓	✓	✓	[16]
TiViPE & its textual robotics command language	Novice & professional	Textual & visual	✓	✓	✓		✓	[17]
AIML	Professional	Textual			✓		✓	[19]
BML	Professional	Textual			✓		✓	[20]
BONSAI	Professional	Textual	✓	✓	✓		✓	[21]
Robot Behavior Toolkit	Professional	Textual			✓		✓	[22]

Table 1: Social interaction programming tools.

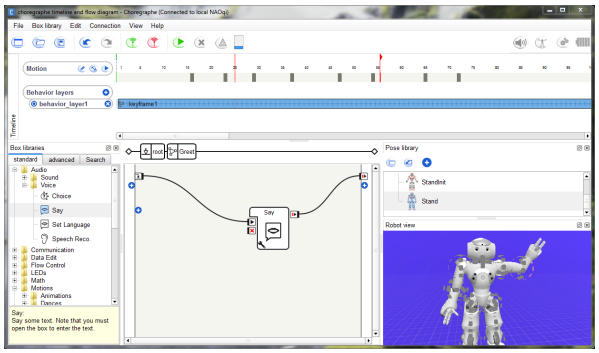

Figure 3: Choregraphe.

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 which include a timeline and a data flow diagram (methods for controlling primitives). The timeline is used to organise visual blocks in a linear fashion and to create animation content with a key-frame 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.

2.2.2. *Interaction Blocks*

Interaction Blocks is a visual end user social interaction programming tool designed to enable designers to prototype social interaction scenarios [15], illustrated in Figure 4. Interaction Blocks provides visual blocks with which the user authors social interaction, similar to Choregraphe [11] and Interaction Composer [16]. The main difference is that all of the visual blocks are presented at an 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 (methods for controlling primitives).

The results of a System Usability Scale [25] evaluation suggest that the Interaction Blocks environment and notation as a whole are usable [15]. 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 monologue-comment primitive and the individual monologue and comment primitives [15, p. 1445]; and that participants desired more primitives, e.g. to represent utterances that have the same underlying meaning [15, p. 1446].

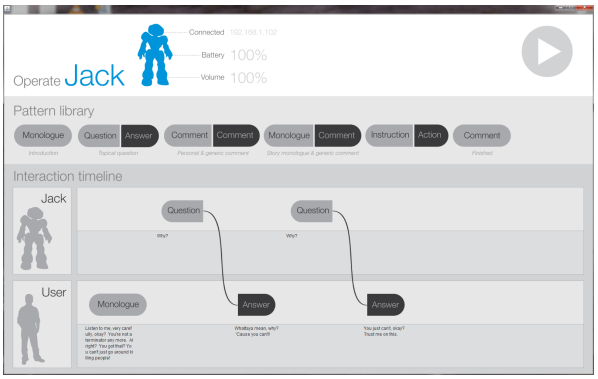

Figure 4: Interaction blocks.

2.2.3. *Interaction Composer*

Interaction Composer, illustrated in Figure 5, is a programming environment supporting collaboration between programmers and end users to create social interaction scenarios [16]. 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, including algorithm primitives, social primitives and emergent primitives (Table 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 (methods for controlling primitives).

An evaluation of Interaction Composer [16] 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).

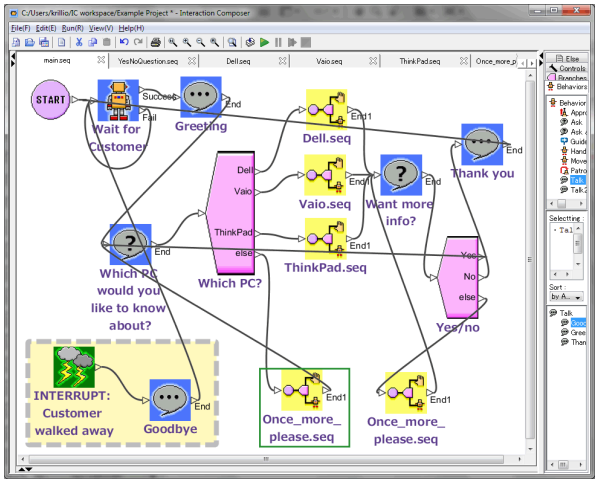

Figure 5: Interaction Composer. Used with permission of Dylan F. Glas [16].

2.2.4. *TiViPE textual robotics command language*

Lourens & Barakova [17] and Barakova et al. [18] present a "textual robotics command language" (Figure 6) for programming a Nao robot paired with the visual programming environment TiViPE [26]. 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 [16], 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. 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 (methods for controlling primitives).

Barakova et al. [18] 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.

```
walk(1.0, 2000) &
[ledto(LeftFaceLedGreen0, 1, 1000) |
 ledto(RightFaceLedGreen0, 1, 1000) |
 movem(HeadPitch 30, 700, HeadYaw -30, 700)] &
say(Hello, 50)
```

Figure 6: Example TiViPE textual robotics command language program.

2.2.5. *Artificial Intelligence Markup Language*

Artificial Intelligence Markup Language (AIML) is an XML based markup language for authoring the content of natural language dialogue systems [19]; most of its abstractions can be categorised as methods for controlling primitives. It is the technology behind the popular ALICE chat bot [27]. An example is illustrated in Figure 7. AIML has been applied as the dialogue engine behind physical robots including Valerie [28] and Pearl [29]. The most important of AIMLs primitives are illus-

trated in Figure 7, which 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 chat bot says what is in the template element (a social primitive for speaking). For example, in Figure 7, if the user types "Who are you?" the chat bot will say "I am a chat bot". AIML has many more features and its programs are typically orders of magnitude more complex than the example in Figure 7 [19].

```
<category>
  <pattern>WHO ARE YOU</pattern>
  <template>I am a chat bot</template>
</category>
```

Figure 7: AIML example

2.2.6. *Behaviour Markup Language*

Behaviour Markup Language (BML) is a textual domain specific language for specifying the actions of Embodied Conversational Agents [20], an example of BML code is shown in Figure 8. BML allows the definition of social interaction by providing XML interfaces at the social primitive abstraction level to control 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 [30, 31].

```
<bml xmlns="http://www.bml-initiative.org/bml/bml-1.0"
   character="Nao" id="bml1">
  <speech id="speech1" start="0">
    <text>Hello human</text>
  </speech>
</bml>
```

Figure 8: BML example

2.2.7. *BONSAI*

BONSAI is a software framework designed to help robot software developer's program household interactive robots [21], 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 method for controlling primitives is provided by using State Chart XML (SCXML), a generic XML based finite state machine language and engine [32]. BONSAI uses SCXML as the method for controlling primitives when implementing scenarios such as guiding a person around a house.

Siepmann [21] performed a number of different evaluations of BONSAI, one of which focuses on evaluating its usability [21, p. 82]. In this study, participants performed programming tasks and completed post task questionnaires. The question, "Has Bonsai provided functionality in a meaningful way?", aimed to assess how well users perceived BONSAIs abstraction level. From the answers to this question the authors concluded that BONSAIs abstractions were well chosen.

2.2.8. *Robot Behavior Toolkit*

Like BONSAI [21], the Robot Behavior Toolkit [22] 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 9. Social primitives are provided that enable the robot to speak and gaze at objects. Additionally, two methods for controlling primitives 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. 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. Huang and Mutlu [22] evaluated the effectiveness of the social interactions produced by the Robot Behaviour Toolkit, not its usability.

```
<behaviors>
  <channel type="gaze">
    <action startTime="0" endTime="300" target="listener" />
    <action startTime="300" endTime="1000" target="the cup" />
    <action startTime="1000" endTime="1500" target="unspecified" />
    <action startTime="1500" endTime="3500" target="sink" />
    <action startTime="3500" endTime="4000" target="unspecified" />
    <action startTime="4000" endTime="5000" target="listener" />
  </channel>
  <channel type="speech">
    Please put the cup into the sink sir.
  </channel>
</behaviors>
```

Figure 9: An example Robot behaviour toolkit program.

2.2.9. Discussion

The tools collectively have a mix of visual and textual notations. All of the tools targeted at novice programmers provide visual notations, including Choregraphe [11], Interaction Blocks [15], Interaction Composer [16] and TiViPE [18]. TiViPE also has a textual notation intended for use by a professional programmer in collaboration with an end user. The tools targeted solely at professional programmers only have textual notations, these include AIML [19], BML [20], BONSAI [21] and the Robot Behavior Toolkit [22].

The abstraction levels used by the tools show little consensus of a suitable abstraction level for programming social robot applications, as they collectively use all of the available abstraction levels.

Four of the eight tools were evaluated with user studies, but don't report much about abstraction level usability. These include TiViPE and its robotics command language [18], Interaction Blocks [15], Interaction Composer [16] and BONSAI [21]. These studies are discussed below.

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 [21] evaluation of BONSAI was the only study to explicitly examine the appropriateness of its abstraction level. From the answers to one question ("Has Bonsai provided functionality in a meaningful way?"), 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, the trade-offs caused by using different abstraction levels to program social interaction are not explored by the evaluations (RQ2).

Whilst the evaluations of existing social interaction programming tools do not explore the usability of their abstraction levels in detail, the Cognitive Dimensions of Notations framework [14] can be used to discuss possible usability trade-offs associated with using different abstraction levels to program robot social interaction.

Over half of the tools have mixed abstraction levels (excluding methods for controlling primitives), showing that their designers have not come to a conclusion about any single abstraction level being appropriate. These include Choregraphe [11], Interaction Composer [16], TiViPE [18] and BONSAI [21]. Mixing abstraction levels goes against the Cognitive Dimensions [14] principle of consistency, which states that once a user has learned part of a notation, they should be able to infer more of it. Also, having to learn multiple abstraction levels to complete a task negatively affects learnability [14], for instance needing to learn multiple lower level primitives to program a social interaction task.

BONSAI [21], Choregraphe [11], Interaction Composer [16] and TiViPE [17] have a large proportion of lower level hardware primitives and or algorithm primitives. The Cognitive Dimensions [14] 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 [14, 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 [14, pp. 38-39]. Lastly, the hardware primitives and algorithm primitives have a distant mapping to the problem domain of social interaction.

Almost all of the programming tools provide social primitives, which have a closer mapping to the social interaction domain than the previous two levels. Green & Petre [14] argue that giving a notation a close mapping to a problem domain should make it easier for users to solve problems in that domain.

Choregraphe [11], Interaction Composer [16], TiViPE [18] and BONSAI [21] provide social primitives for some aspects of social interaction, but often for functions that are easy to implement, e.g. speaking. The programmer then has to fall back to lower abstraction levels, even for common tasks such as gesturing and gazing at people. This could potentially reduce the consistency of the tool as abstractions are being mixed for similar tasks.

AIML [19], BML [20] and the Robot Behavior Toolkit [22] solely provide social primitives, which gives a higher level of consistency. However, all three lack social primitives for 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.

Interaction Blocks [15] and Interaction Composer [16] both contain emergent primitives, which retain a somewhat close mapping to the social interaction domain, but have a higher abstraction level than social

primitives. Interaction Blocks [15] only provides emergent 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 [14].

3. Our approach

We use an iterative research methodology inspired by design science to explore what abstraction levels are appropriate for programming social robot applications. Design science is concerned with discovering how the design and creation of artefacts influences people and their environment [33]. A high level overview of the methodology is illustrated in Figure 10.

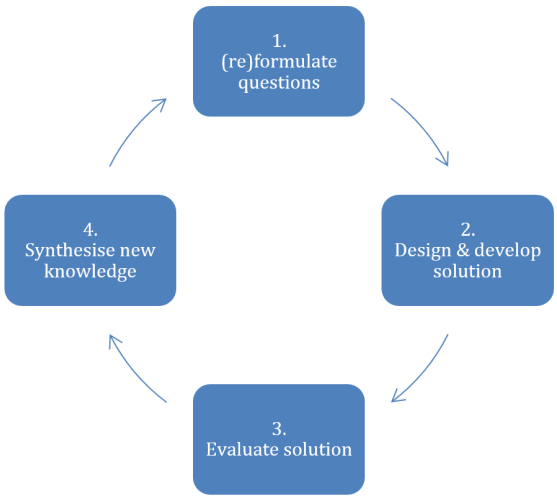

Figure 10: Iterative research methodology, based on process from [33, p. 58].

With this methodology the researcher (1) first formulates questions about a problem, e.g. research questions and then (2) designs an artifact as a solution to the problem [33]. This is not just engineering, the researcher uses his or her creativity to create a novel solution to a problem [34]. 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 re-formulate the questions and conduct further investigation.

The research questions discussed in the introduction (Section 1) were addressed through two iterations of API design, evaluation and synthesis. User-centred design [35] and the Cognitive Dimensions [14] were used throughout this process.

The Cognitive Dimensions [14] is a vocabulary of 13 'dimensions' intended as a design tool to help designers weigh up decisions that affect the usability of a notation. Each dimension describes a unique way that the structure of a notation can be altered to affect usability; changing one dimension will very likely affect another. For instance, raising the abstraction level of an API can increase hidden dependencies, but it could also increase or decrease the closeness of mapping of the notation depending on the target audience.

The Cognitive Dimensions [14] can also be used to evaluate an API. This typically begins by performing a usability study to collect data (e.g. [36, 37, 38, 39, 40, 41]). In the usability study, participants perform programming tasks (e.g. [36, 37, 41, 42]) and or write pseudo code (e.g. [39, 41, 43]). The participants are observed using the API while performing the tasks [36] and may complete pre or post task questionnaires (e.g. [36, 37, 44, 45]). The Cognitive Dimensions is then used to analyse the collected data [36].

In our studies, we use the Cognitive Dimensions Questionnaire (CD Questionnaire) [46] to collect data from participants. It is a standard, well accepted questionnaire that presents the Cognitive Dimensions [14] in a way that allows a programmer to provide Cognitive Dimensions based feedback without having previously been exposed to them. For example, Clarke [47] used the CD Questionnaire to gather data about the usability of the programming language C#, parts of which are technically an API.

4. Iteration 1: Social interaction for Nao

To begin exploring how to improve the usability of APIs for programming social robot applications through choosing an appropriate abstraction level, we designed an API with high level, domain specific primitives for programming the social interaction of a Nao humanoid robot. It contains primitives for making speech, performing gestures, modeling human features and understanding human speech.

4.1. Method

The API was designed with a user-centred design methodology [35], using the Cognitive Dimensions [14] as a design framework to weigh up the merit of various design choices.

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 [48, 49]. 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.

The APIs primitives were based on a taxonomy of primitives that we developed from researchers' descriptions of robot behaviour, this is presented in [50].

A lab based user evaluation [51] was then conducted to evaluate the APIs domain specific primitives and the imperative state machine API. The participants completed programming tasks to become familiar with the API and researcher observations and a post-task CD Questionnaire [46] were used to gather data. The Cognitive Dimensions [14] 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.

4.2. The API

The API is composed of a number of important classes that perform different tasks: *Environment*, *Object* (with subclasses *Robot* and *Person*) and *HriQuery*[1]. The rest of this section overviews these.

4.2.1. Environment

The *Environment* class encapsulates the objects the robot perceives in its environment, including the robot itself. These are represented by the *objects* and *robot* attributes. 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 robots 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.

4.2.2. Object

The *Object* class contains functionality common to all objects and is inherited by all classes that represent a particular type of object. The most important functions include: *distance_to(obj)*, which finds the distance between two objects; and standard functions for querying the spatial relationships of objects, including, *left_of(obj)*, *right_of(obj)*, *infront_of* and *behind(obj)* which return whether an object (caller) is left-of, right-of, in front or behind another object (obj) respectively (Figure 11).

```
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
```

Figure 11: *Object* function examples.

The API defines two *Object* subclasses for programmers: *Robot*, which represents the robot being programmed; and *Person*, which represents a human being and their body parts.

4.2.3. Robot

The main design goal of the *Robot* class was to provide high level functions for making speech, performing gestures and understanding human speech described. The *Robot* classes' action and understanding human speech functions are explained in the rest of this section.

4.2.4. Robot: Actions

The most important function for the *Robot* class is *say_to(text, audience)* (Figure 12), 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 features required for programming socially interactive robots, including the ability to 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.

```
robot.say_to("Hello", people)
robot.say_to("<wave> Hello </wave>", people)
robot.say_to("<point target={0}> Who's that?
        </point>", people, person1)
```

Figure 12: *say_to* action

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

[1] HriQuery stands for Human-Robot Interaction Query

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 allows the robot to synchronise gestures with speech and target the gestures at objects. The following list has example gestures (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 mark-up 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 [16] and BML [20] use a more diffuse syntax, e.g. *"<gesture type='wave'> </gesture>"*.

4.2.5. Robot: Understanding human speech

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

associate_utterances_with_meaning(meaning, utterances, context=None)[2]. This function is used to inform the robot of spoken words that mean the same thing. For example, with this function, the words *'hello'* and *'hi'* (utterances) could both be classified as a 'greet[ing]' (the meaning). Then when a person says 'hello' or 'hi', the robot recognises this as speech with a meaning of 'greet'.

An example is shown in Figure 13. A list of utterances are classified as greetings (*'hello'* and *'hi'*) and another list are classified as farewells (*'bye'* and *'see ya'*). The greetings that have recently been spoken can be queried from the Person class, described in Section 4.2.6.

```
meanings = Enum('greet', 'farewell')
robot.associate_utterances_with_meaning(
       meanings.greet, ['hello', 'hi'])
robot.associate_utterances_with_meaning(
       meanings.farewell, ['bye', 'see ya'])
```

Figure 13: Associating utterances with meanings.

An optional feature is that utterances can be assigned to a meaning based on a particular context (a unique key) - see Figure 14. This is useful when you don't want to treat two or more utterances as synonyms, because they can mean different things in different contexts used in your application.

```
meanings = Enum('greet')
robot.associate_utterances_with_meaning(
         meanings.greet, ['oi'], context='UK')
robot.associate_utterances_with_meaning(
         meanings.greet, ['mate'], context='NZ')
```

Figure 14: 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.

4.2.6. Person

The *Person* class represents a person that has been perceived by the robots perception system. As stated

[2]This function was called *add_meaning_to_utterances_map* in the version used in the evaluation

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) 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. [52]. 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.

4.2.7. HriQuery

The *HriQuery* class is used to filter objects from the robots 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 [53], 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 [54]. 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. Examples of queries are shown in Figure 15.

```
# Instantiate query
q = query(env.objects)

# Select all Person instances (people)
ppl = q.of_type(Person)

# Select people less than 2m from the robot
ppl.where(
    lambda p: p.distance_to(env.robot)) < 2)

# Order people by distance to robot: increasing
ppl.order_by(lambda p: p.distance_to(env.robot))

# Order people by distance to robot: decreasing
ppl.order_by_descending(
     lambda p: p.distance_to(env.robot))
```

Figure 15: Query examples.

4.3. 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 [46], 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.

4.3.1. 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 [46] (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 16) where the Nao 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. The winning team then verbally responds with the answer (understanding human speech). If the team's verbal response is correct, then Nao increases the team's score (dialogue). If the verbal response is wrong, Nao either gives 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.

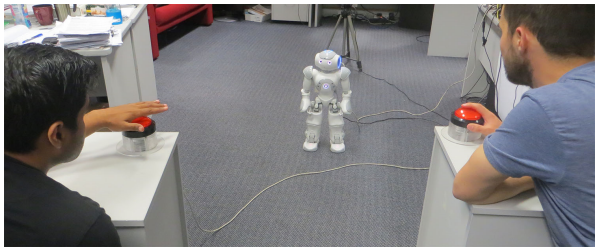

Figure 16: Game show setup.

After interacting with the robot, participants spent 20-30 minutes reading the API documentation and an example program. 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.

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 [46] to reflect on their experience using our API to program social interaction.

4.3.2. Results

Results analysis consisted of classifying questionnaire responses by whether they were positive, equivocal or negative; based on how Blackwell & Green [46] analysed responses to the CD 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 make a robot express emotion and empathy (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 an imperative state machine to orchestrate primitives into a social interaction scenario; over 60% of the specific negative responses (31 of 53) related to this. The second theme is discussed in a separate paper [50]; because it relates to a different research question than those in this paper [3]. The following paragraphs discuss how the APIs domain specific interfaces affected usability in the context of the Cognitive Dimensions of Notations.

Participants appreciated the domain specific interfaces of the API because of their visibility & role expressiveness, close mapping to social interaction and terseness.

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 also 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

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

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 [4] 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.

A number of other usability issues were also raised, including:

Participants were observed questioning why they needed to add queries 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).

[4] This refers to the functions described in Section 4.2.5; used in combination with the Person class (Section 4.2.6), the HriQuery class (Section 4.2.7) and the finite state machine (described in another paper [50]).

4.4. Stakeholder interview

Subsequent to the user evaluation, we performed an email based stakeholder interview with members of Hanson Robotics [55] and the OpenCog Foundation [56] 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 17. To support these robots, the API would need more specific abstractions for programming robot facial expressions.

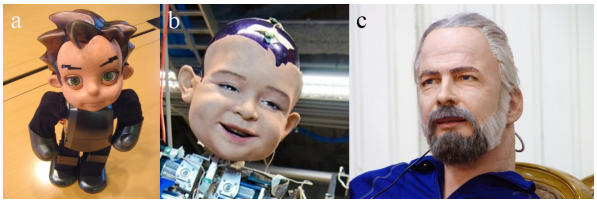

Figure 17: Hanson Robotics human-like robots: (a) Zeno, (b) Diego-San & (c) Philip K. Dick. Image sources: Zeno - [57]; Diego-San image used with permission of David Hanson; Philip K. Dick - [58].

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. Iteration 2: Social interaction refactored

To further explore how to improve the usability of APIs for programming social robot applications through choosing an appropriate abstraction level (RQ1 & RQ2), we performed a second iteration of API design and evaluation. Based on feedback from previous usability study and the stakeholder interview, the APIs abstraction level was refactored to provide finer control of social interaction.

5.1. Method

As with the previous iteration, the API was designed with a user-centred design methodology [35] and the Cognitive Dimensions [14] to weigh up the merit of design decisions.

A new scenario was adopted, the goal of which was to program the Nao [59], Zeno, Zoidstein robots to communicate with people emotionally. Zeno and Zoidstein (Figure 18) can express a wider range of social interaction than the Nao robot [59] due to their human-like faces.

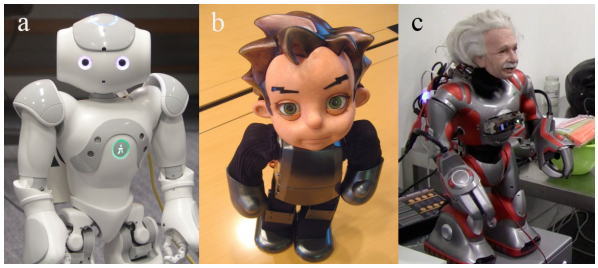

Figure 18: Humanoid robots: (a) Nao, (b) Zeno and (c) Zoidstein. Source for images: Nao - used with permission from Samar Pant; Zeno - (Jurvetson, 2008). Note that Zeno had a different body from the one pictured above.

The second API was evaluated with a case study [60] to gain in depth results with ecological validity. 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 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 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 [46]. The data was analysed against the Cognitive Dimensions framework [14].

5.2. The API

The second API iteration was driven by three high level objectives, listed below:

O1. Refactor the APIs abstraction level to provide finer control of social interaction.
O2. Broaden the aspects of social interaction the API can program by supporting new robots and scenarios.
O3. Address the usability issues of the previous API.

The main change to the API involved refactoring the *Robot* class to provide finer control of social interaction (O1). The abstractions are closer to the social primitive abstraction level than the previous API, which also had emergent primitives. Additionally, to broaden the aspects of social interaction that could be programmed (O2), extra primitives were added to the *Robot* class to support the facial expressions of the Zeno and Zoidstein robots (Figure 18).

The *Person* and *Query* classes were refactored and *World* class introduced to fix the usability issues discovered from the CD Questionnaire used in the previous usability study (O3).

The rest of this section describes these changes and their design rationale.

5.2.1. 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' with a lower case first letter. To fix this ambiguity, the *Object* base class was renamed *Entity* . All real world objects, including robots, are now derived from the *Entity* class.

5.2.2. Robot

The *Robot* class represents the robot being programmed. 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 Nao robot is represented by the *Nao* class. This is because each robot has its own unique body structure represented within its own class. 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 [14] 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.

5.2.3. 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. 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 [61].

The code in Figure 19 demonstrates a range of different robot actions. The robot begins by simultaneously saying "Hello" and gazing at a persons head. When these actions have finished, the robot asks them "Who are you?" whilst smiling and pointing at the persons torso. When the robot finishes speaking it lowers its arm and stops smiling. These actions are explained in the following paragraphs.

```
robot = Zeno()
person = Person(1)

ah1 = robot.say("Hello")
ah2 = robot.gaze(person.head)
robot.wait(ah1, ah2)

text = "Who are you?"
length = robot.say_duration(text)

ah3 = robot.say(text)
ah4 = robot.gesture(Gesture.PointLArm,
       target=person.torso, duration=length)
ah5 = robot.expression(Expression.Smile,
       duration=length)
robot.wait(ah3, ah4, ah5)

robot.gesture(Gesture.LowerLArm)
robot.expression_and_wait(Expression.Smile,
              intensity=0.0)
```

Figure 19: Demonstration of a range of robot actions.

say(sentence). Makes the robot speak. 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.

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 [62]. Each action function has a synchronous sister function.

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.

gaze(target, speed=0.5). The gaze action primitive is used to make a robot gaze at a *target* (any subclass of *Entity*). The speed that the robot gazes at is controlled by the speed parameter.

The gestures that a particular robot can perform are represented using Python enumerations, a collection of symbolic names (members) tied to distinct values [63]. Each gesture enumeration member contains the default duration the gesture should be played for. The Gesture enumeration members for the Nao robot are illustrated in Figure 20.

```
Gesture.LowerLArm
Gesture.LowerRArm
Gesture.WaveLArm
Gesture.WaveRArm
Gesture.SwipeLArm
Gesture.SwipeRArm
Gesture.PointLArm
Gesture.PointRArm
Gesture.HandsOnHips
```

Figure 20: Nao's gestures.

Alternative design decisions for gesture and expression representation included using a configuration file or hard coding 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.

The *gesture(gesture, target=None, duration=Default())* function is used to make the robot perform a gesture, e.g. wave. The gesture to perform is specified by the gesture parameter, a Python enu-

meration 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).

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 21 below, they make Zeno smile, frown his mouth, open his mouth and frown his forehead respectively.

```
Expression.Smile
Expression.FrownMouth
Expression.OpenMouth
Expression.FrownForehead
```

Figure 21: Zeno's facial expressions.

The *expression(expression, intensity=Default(), speed=Default(), duration=Default())* function 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).

In addition to calling functions to make the robot perform actions, objects can be created to represent the *say*, *gaze*, *expression* and *gesture* actions (Figure 22). Action objects are useful in situations where actions need to be pre-computed and stored for later use; a realisation of the object oriented command pattern [64]. 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 next.

```
SayAction("Hello I'm a robot")
GazeAction(person.head)
GestureAction(Gesture.WaveLArm, duration=6.0)
ExpressionAction(Expression.Smile)
```

Figure 22: Action objects.

5.2.4. Robot: Action controls

The second change to the *Robot* class was the addition of more action control primitives (Figure 23), which provide finer control of how actions are executed. The API provides four additional methods of controlling actions, which include the ability to wait for actions to complete, stop actions and run actions simultaneously or consecutively. These are similar in principle to fork/join frameworks, e.g. [65]. Action objects to represent action control primitives were not implemented, but would be a natural extension. They are explained below.

```
robot = Zeno()
person = Person(1)

ah1 = robot.say("Hello, I will eat you alive")
robot.wait(ah1)

ah2 = robot.gesture(Gesture.WaveLArm,
                    duration=10.0)
time.sleep(1.0)
robot.stop(ah2)

a1 = SayAction("Hello I'm a robot")
a2 = GazeAction(person.head)
a3 = GestureAction(Gesture.WaveLArm)
a4 = ExpressionAction(Expression.Smile)

ahs = robot.simultaneously(a1, a2, a3, a4)
robot.wait(ahs)

ahs = robot.consecutively(a1, a2, a3, a4)
robot.wait(ahs)
```

Figure 23: Action controls.

*wait(*action_handles)*. The wait control primitive is used to wait 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(*action_handles)* The stop control primitive is used to stop an action from running, e.g. to stop a robot from speaking, gazing, gesturing or showing an expression.

*simultaneously(*actions)*. The simultaneously control primitive is used to make a robot execute actions in parallel by directly passing action objects as parameters to the simultaneously function. This function is asynchronous returning an array of action handles; one for each action.

*consecutively(*actions)* The consecutively control

primitive is used to make a robot execute actions one after another by directly passing action objects as parameters to the consecutively function.

5.2.5. Robot: Action callbacks

The third change to the *Robot* class is the ability to subscribe to callbacks that fire during the execution of actions. There are two types of callbacks for each action type, including a feedback callback which indicates the current progress of the action, and a done callback that indicates when the action has finished. The standard ROS actionlib callback mechanisms are used here [62].

5.2.6. 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 24).

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 of the previous system. 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 registered 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.

```
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 24: Example listening program.

5.2.7. Person

The *Person* class models the human body, including the persons head, neck, torso, shoulders, elbows, hands, hips, knees and feet. It is used to program interactions with people, e.g. to make the robot look at a person's left arm.

The big 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.

5.2.8. World

The APIs *World* class provides access to the entities perceived by the robots perception system. It is iterable, meaning that it can be iterated over, just as a list or array can. Perception algorithms communicate with the *World* instance, populating it with instances of entities that represent what the robot is perceiving.

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 [66]. 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.

5.2.9. Query

The Query class provides a high-level interface to filter, sort and select entities perceived by a robot similar to Microsoft's LINQ [67] and based on Python-ASQ [53].

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. 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. [42] who found that constructors are easier to use than the factory pattern.

5.3. User evaluation

This version of the API was evaluated with a case study [60]. Four fourth year software engineering students, all taking an advanced human computer interaction course, used the API for a period of two weeks.

5.3.1. 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 [68]. These are explained in the following paragraphs.

Research questions. The objective of the case study was to explore how well the design decisions of the second API iteration address the research questions introduced in Section 1.

Case and subject selection. The case and subjects were selected to reflect a typical real world application, group of users and time spent with the API.

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 interview questions were based on questions from both the CD Questionnaire [46] and the questions Clarke [36] asks participants when measuring API usability.

The data was transcribed and then analysed by grouping quotes with similar themes together, akin to the open and axial coding techniques from grounded theory [69].

Validity procedures. The validity procedure involved a form of data triangulation: 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. The participants were fourth year software engineering students, one female and three male. They had previous experience with ROS [12] during a previous engineering course. The participants started by reading the API documentation 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.

5.3.2. Results

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, 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"

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; 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). These are explained in more detail below.

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, it should be noted that users can create new abstractions to represent a new robot or a new set of gestures or expressions.

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 [11]. 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 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 [14] 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 [14] 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).

The participants also commented about the APIs expressiveness, environment and documentation. Participants thought that they could have more control over social interaction, e.g. pausing speech and changing speaking volume (P1). The environment was the most difficult part of using the API (P1, P2, P3, P4), due to unreliable services that support the API and the need to use many command line arguments to start them. The participants appreciated the tutorial style documentation, however, they also desired traditional class and method documentation (P1, P2 and P3), the absence of which hurt APIs usability.

6. Discussion

We now discuss the results of the two iterations of API design and evaluation presented in this paper.

6.1. Methodology

The Cognitive Dimensions [14] is used frequently throughout this research as a design tool, a questionnaire [46] and a framework for analysing results from user studies.

The Cognitive Dimensions [14] 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, we 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 e.g. [41, 38], cognitive walkthroughs [70], or having an external researcher conduct a Cognitive Dimensions expert evaluation.

The CD Questionnaire [46] was used in both user evaluations. In Section 4 participants wrote responses to the questionnaire and in Section 5 the questions were incorporated into interview questions. Blackwell & Green [46] 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.

Whilst we found the CD Questionnaire [46] to be very useful, it does take a long time to answer because there are (deliberately) two to three very similar questions for each dimension (e.g. Figure 25). 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 [46] 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 25: CD Questionnaire progressive evaluation questions [46].

6.2. An appropriate abstraction level for programming social robot applications

The first API (Section 4) 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 & RQ2). This is because they provide good visibility and 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 & RQ2, because it implies that a lower abstraction level may be more appropriate for programming socially interactive robots.

This led to the second API iteration (Section 5) 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 refactoring. 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 [14].

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 [71] 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 (Section 4) 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 (Section 5) 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 (Section 4) 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 (Section 5) 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 everyday 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 (Section 5) 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 [72], 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 amount of 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.

6.3. Further exploration of APIs for programming social interaction

The APIs presented in this paper are dependent on current commercial robot platforms. This made it difficult to explore certain areas of social interaction, especially those related to human perception. For example, identifying who is speaking is an important part of social interaction, 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 [73] and Morse [74] are two popular simulators for ROS [12] 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. Inspiration for this interface can be taken from real time strategy games such as StarCraft II [75] 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.

Whilst this research has found that textual APIs can work well for programming particular aspects of robot social interaction, visual languages may be more appropriate for other tasks, such as *methods for controlling primitives*. For example, the social interaction API could be used to program individual robot actions and perceptions whilst a visual language could be used to orchestrate the higher level flow of behaviour of the application (e.g. dialogue). Visual language programming techniques from game industry, such as visual behaviour tree editors [76], may prove useful for such tasks.

7. Conclusions & future work

The key contribution of this research is a deep 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.

We found that high level, domain specific primitives have many benefits for the usability of social robot programming; these include both social and emergent primitives.

The evidence suggests that social primitives are more suitable for programming robot social interaction than emergent primitives, because they are more reusable and are still usable.

The benefits 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 [72] API documentation guidelines.

There are a number of avenues for future work. First, to encourage more detailed responses, the CD Questionnaire [46] could be shortened when used in lab studies by only providing one question per dimension. Second, the results regarding abstraction levels could be generalised to design APIs with appropriate abstraction levels for programming robot navigation and manipulation applications. Third, using a simulator rather than a real robot platform would help to design robot APIs faster and make them more future proof by removing dependence on current hardware and software. Last, textual

social robot APIs (such as the one described in this paper) could be used in conjunction with visual languages (e.g. visual behaviour tree editors [76]) to orchestrate primitives into higher level social interaction.

Acknowledgments

The authors thank the University of Auckland PhD Scholarship programme for financial support, the participants of the user studies, the University of Auckland HCI Research group for their feedback on previous versions of the paper (particularly Andrew Luxton-Reilly) and the University of Auckland Robotics Research Group.

References

[1] H. Kozima, C. Nakagawa, Y. Yasuda, Children–robot interaction: a pilot study in autism therapy, Progress in Brain Research 164 (2007) 385–400.

[2] C. Breazeal, B. Scassellati, How to build robots that make friends and influence people, in: Intelligent Robots and Systems, 1999. IROS'99. Proceedings. 1999 IEEE/RSJ International Conference on, volume 2, IEEE, pp. 858–863.

[3] E. G. Dougherty, H. Scharfe, Initial formation of trust: Designing an interaction with Geminoid-DK to promote a positive attitude for cooperation, in: Social Robotics, Springer, 2011, pp. 95–103.

[4] S. Nishio, H. Ishiguro, N. Hagita, Geminoid: Teleoperated android of an existing person, INTECH Open Access Publisher, 2007.

[5] Keysmaker, File:KeeponTophatNextfest2007.jpg - Wikimedia Commons, `commons.wikimedia.org/wiki/File:KeeponTophatNextfest2007.jpg`, 2007.

[6] Daderot, File:Kismet, 1993-2000, view 1 - MIT Museum - DSC03710.JPG - Wikimedia Commons, 2013.

[7] S. Jurvetson, Uncanny Valley | Flickr - Photo Sharing!, `www.flickr.com/photos/jurvetson/6818431732`, 2012.

[8] H. Kozima, M. P. Michalowski, C. Nakagawa, Keepon, International Journal of Social Robotics 1 (2009) 3–18.

[9] J. P. Diprose, End user robot programming via visual languages, in: Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on, IEEE, pp. 229–230.

[10] Aldebaran Robotics, NAOqi modules APIs NAO Software 1.14.5 documentation, `doc.aldebaran.com/1-14/naoqi/index.html`, n.d.

[11] E. Pot, J. Monceaux, R. Gelin, B. Maisonnier, Choregraphe: a graphical tool for humanoid robot programming, in: Robot and Human Interactive Communication, 2009. RO-MAN 2009. The 18th IEEE International Symposium on, IEEE, pp. 46–51.

[12] Open Source Robotics Foundation, ROS.org | Powering the world's robots, `www.ros.org`, n.d. Accessed: 2014-11-11.

[13] T. Fong, I. Nourbakhsh, K. Dautenhahn, A survey of socially interactive robots, Robotics and Autonomous Systems 42 (2003) 143–166.

[14] T. R. G. Green, M. Petre, Usability analysis of visual programming environments: a "cognitive dimensions" framework, Journal of Visual Languages & Computing 7 (1996) 131–174.

[15] A. Sauppé, B. Mutlu, Design patterns for exploring and prototyping human-robot interactions, in: Proceedings of the 32nd annual ACM conference on Human factors in computing systems, ACM, pp. 1439–1448.

[16] D. Glas, S. Satake, T. Kanda, N. Hagita, An interaction design framework for social robots, in: Robotics: Science and Systems, volume 7, p. 89.

[17] T. Lourens, E. Barakova, User-friendly robot environment for creation of social scenarios, in: Foundations on Natural and Artificial Computation, Springer, 2011, pp. 212–221.

[18] E. I. Barakova, J. Gillesen, B. Huskens, T. Lourens, End-user programming architecture facilitates the uptake of robots in social therapies, Robotics and Autonomous Systems 61 (2013) 704–713.

[19] R. Wallace, The elements of AIML style, Alice AI Foundation (2003).

[20] S. Kopp, B. Krenn, S. Marsella, A. N. Marshall, C. Pelachaud, H. Pirker, K. R. Thórisson, H. Vilhjálmsson, Towards a Common Framework for Multimodal Generation: The Behavior Markup Language, in: Intelligent Virtual Agents, number 4133 in Lecture Notes in Computer Science, Springer, 2006, pp. 205–217.

[21] F. Siepmann, Behavior coordination for reusable system design in interactive robotics, Ph.D. thesis, Universitätsbibliothek Bielefeld, 2013.

[22] C.-M. Huang, B. Mutlu, Robot behavior toolkit: generating effective social behaviors for robots, in: Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction, ACM, pp. 25–32.

[23] B. Gerkey, R. T. Vaughan, A. Howard, The player/stage project: Tools for multi-robot and distributed sensor systems, in: Proceedings of the 11th international conference on advanced robotics, volume 1, pp. 317–323.

[24] J. Diprose, jdddog/social-abst-level-data, 2016.

[25] J. Sauro, Measuring usability with the system usability scale (SUS) (2011).

[26] T. Lourens, Tivipe-tino's visual programming environment, in: Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, IEEE, pp. 10–15.

[27] A. L. I. C. E. The Artificial Linguistic Internet Computer Entity, `alice.pandorabots.com`, n.d. Accessed: 2015-03-02.

[28] R. Gockley, A. Bruce, J. Forlizzi, M. Michalowski, A. Mundell, S. Rosenthal, B. Sellner, R. Simmons, K. Snipes, A. C. Schultz, et al., Designing robots for long-term social interaction, in: Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on, IEEE, pp. 1338–1343.

[29] C. Torrey, A. Powers, M. Marge, S. R. Fussell, S. Kiesler, Effects of adaptive robot dialogue on information exchange and social relations, in: Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction, ACM, pp. 126–133.

[30] A. Holroyd, C. Rich, Using the behavior markup language for human-robot interaction, in: Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction, ACM, pp. 147–148.

[31] Q. A. Le, C. Pelachaud, Generating co-speech gestures for the humanoid robot NAO through BML, in: Gesture and Sign Language in Human-Computer Interaction and Embodied Communication, Springer, 2012, pp. 228–237.

[32] W3C Consortium, State Chart XML (SCXML): State Machine Notation for Control Abstraction, `www.w3.org/TR/scxml`, 2015.

[33] K. Peffers, T. Tuunanen, M. A. Rothenberger, S. Chatterjee, A design science research methodology for information systems research, Journal of management information systems 24 (2007) 45–77.

[34] D. A. Schön, Educating the reflective practitioner: Toward a new design for teaching and learning in the professions, San Francisco (1987).

[35] D. A. Norman, The design of everyday things, Basic books, 2002.

[36] S. Clarke, Measuring API usability, Doctor Dobbs Journal 29 (2004) S1–S5.

[37] E. Duala-Ekoko, M. P. Robillard, Asking and answering questions about unfamiliar APIs: An exploratory study, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, pp. 266–276.

[38] J. Stylos, B. A. Myers, The Implications of Method Placement on API Learnability, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, ACM, New York, NY, USA, 2008, pp. 105–112.

[39] J. Beaton, S. Y. Jeong, Y. Xie, J. Stylos, B. A. Myers, Usability challenges for enterprise service-oriented architecture APIs, in: Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on, IEEE, pp. 193–196.

[40] T. Scheller, E. Kuhn, Influencing factors on the usability of api classes and methods, in: Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on, IEEE, pp. 232–241.

[41] J. Brunet, D. Serey, J. Figueiredo, Structural conformance checking with design tests: An evaluation of usability and scalability, in: Software Maintenance (ICSM), 2011 27th IEEE International Conference on, IEEE, pp. 143–152.

[42] B. Ellis, J. Stylos, B. Myers, The Factory Pattern in API Design: A Usability Evaluation, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 302–312.

[43] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, R. Ehret, J. Karstens, A case study of API redesign for improved usability, in: Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on, IEEE, pp. 189–192.

[44] C. A. Brown, Usability analysis of the channel application programming interface, Ph.D. thesis, Monterey, California. Naval Postgraduate School, 2003.

[45] M. Piccioni, C. A. Furia, B. Meyer, An empirical study of api usability, in: Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on, IEEE, pp. 5–14.

[46] A. F. Blackwell, T. R. Green, A Cognitive Dimensions questionnaire optimised for users, in: Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group, pp. 137–152.

[47] S. Clarke, Evaluating a new programming language, in: 13th Workshop of the Psychology of Programming Interest Group, pp. 275–289.

[48] I. Kruijff-Korbayová, G. Athanasopoulos, A. Beck et. al., An Event-Based Conversational System for the Nao Robot, in: Proc. of the Paralinguistic Information and its Integration in Spoken Dialogue Systems Workshop, Springer, 2011, pp. 125–132.

[49] S. Moubayed, Furhat - a robot that plays quiz games - YouTube, `www.youtube.com/watch?v=yy7ccoJPy58\&feature=youtube_gdata_player`, 2013.

[50] J. Diprose, B. Plimmer, B. MacDonald, J. Hosking, A human-centric API for programming socially interactive robots, in: Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on, IEEE, pp. 121–128.

[51] J. S. Dumas, J. Redish, A practical guide to usability testing, Intellect Books, 1999.

[52] F. Grondin, D. Létourneau, F. Ferland, V. Rousseau, F. Michaud, The ManyEars open framework, Autonomous Robots 34 (2013) 217–232.

[53] R. Smallshire, asq - A Python implementation of LINQ to objects and Parallel LINQ to objects. - Google Project Hosting, `code.google.com/p/asq/`, n.d. Accessed: 2015-10-06.

[54] M. Fowler, Domain-specific languages, Pearson Education, 2010.

[55] Hanson Robotics, Hanson Robotics Inc Home - Hanson Robotics Inc, `www.hansonrobotics.com`, n.d. Accessed: 2014-10-13.

[56] OpenCog Foundation, OpenCog Foundation | Building better minds together, `opencog.org`, n.d. Accessed: 2014-10-13.

[57] S. Jurvetson, Zeno - Robotic Friend | Flickr - Photo Sharing!, `www.flickr.com/photos/jurvetson/2218864889`, 2008.

[58] R. Lerdorf, Philip K. Dick android | Flickr - Photo Sharing!, `www.flickr.com/photos/rlerdorf/6867549790`, 2012.

[59] Aldebaran - SoftBank Group, NAO robot: intelligent and friendly companion | Aldebaran, `www.aldebaran.com/en/humanoid-robot/nao-robot`, n.d. Accessed: 2014-10-13.

[60] R. K. Yin, Case study research: Design and methods (Fifth Edition), SAGE Publications, 2013.

[61] B. Goertzel, D. Hanson, G. Yu, Toward a Robust Software Architecture for Generally Intelligent Humanoid Robotics (2014).

[62] E. Marder-Eppstein, V. Pradeep, actionlib - ROS Wiki, `www.ros.org/wiki/actionlib`, n.d. Accessed: 2014-10-07.

[63] Python Software Foundation, 8.13. | enum Support for enumerations | Python 3.4.2 documentation, `docs.python.org/3/library/enum.html`, n.d. Accessed: 2014-10-15.

[64] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Education, 1994.

[65] D. Lea, A Java fork/join framework, in: Proceedings of the ACM 2000 conference on Java Grande, ACM, pp. 36–43.

[66] Y. Lin, J.-B. Michel, E. L. Aiden, J. Orwant, W. Brockman, S. Petrov, Syntactic annotations for the google books ngram corpus, in: Proceedings of the ACL 2012 System Demonstrations, Association for Computational Linguistics, pp. 169–174.

[67] E. Meijer, B. Beckman, G. Bierman, Linq: reconciling object, relations and xml in the. net framework, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, pp. 706–706.

[68] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical software engineering 14 (2009) 131–164.

[69] B. G. Glaser, A. L. Strauss, The discovery of grounded theory; strategies for qualitative research, Aldine Pub. Co., Chicago, USA, 1967.

[70] C. Wharton, J. Rieman, C. Lewis, P. Polson, The cognitive walkthrough method: A practitioner's guide, in: Usability inspection methods, John Wiley & Sons, Inc., pp. 105–140.

[71] D. Roberts, R. Johnson, Evolving frameworks, Pattern languages of program design 3 (1996).

[72] M. P. Robillard, R. DeLine, A field study of API learning obstacles, Empirical Software Engineering 16 (2011) 703–732.

[73] Open Source Robotics Foundation, Gazebo, `gazebosim.org`, n.d.

[74] LAAS-CNRS, ONERA, morse - Openrobots, `www.openrobots.org/wiki/morse/`, n.d.

[75] Blizzard Entertainment Inc, StarCraft II Official Game Site, `us.battle.net/sc2/en/`, 2015. Accessed: 2015-02-04.

[76] G. Robertson, I. Watson, A Review of Real-Time Strategy Game AI, AI Magazine 35 (2014) 75–104.