First steps in creative computational thinking with natural language programming and Lego Mindstorms

A dissertation submitted in partial fulfilment of the requirements for the MSc in Learning Technologies

by Geoffrey Falk

London Knowledge Lab, Birkbeck College and Institute of Education, University of London

September 2013

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme booklet and the School's website.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

In this project, an Android application was developed to help promote the development of creative computational thinking in the Key Stage 1 and 2 classroom. Based on natural language and Lego Mindstorms robots, the application was particularly focused on engaging children and teachers who would not naturally be interested in programming. The application was developed over a three month period in several iterations, and the finished product was tested with groups of children at local schools. The approach gained lots of interest from students and teachers alike, with one Principal sufficiently impressed to want to start investing in Mindstorms for his school. This report summarizes the stages of development for the application, its design, implementation and test results. The report concludes with a comparison of other programming applications, and suggests ways forward for the development of the application.

A video demonstration of the application together with some clips from user testing sessions can be found here: http://vimeo.com/user13559532/roboliterate

1. Chapter 1: Aims and Objectives 7
1.1. Overall aims 7
1.2. Pedagogical background 7
1.3. Technical background 9
1.4. Objectives 9
2. Chapter 2. Approach 11
2.1. Natural language programming 12
2.2. LCP Bluetooth control 13
3. Chapter 3. Methodology 14
3.1. Requirements gathering - June-July 14
3.2. Prototyping - July 14
3.3. Ontology development - July 14
3.4. Two phase development - July-August 14
3.5. Integration testing - August 14
3.6. System testing - September 14
3.7. User testing - September 14
4. Chapter 4. Requirements Gathering 15
4.1. Interviews 15
4.2. Document analysis 16
4.3. Prototyping 16
5. Chapter 5. Requirements Specification 20
5.1. User Requirements 20
5.2. Technical requirements: 22
5.3. NLP specification 23
6. Chapter 6. Software Design 24
6.1. Overall design 24
6.2. UI and UI Controller Layers 24
6.3. RLit Layer 25

6.4. Robot communication architecture 28

7. Chapter 7. Software Implementation....30

7.1. UI Design 30

7.2. UI Layer technical implementation 38

7.3. UI Controller layer 39

7.4. RLit Model layer 41

7.5. Interaction of UI Controller layer and RLit model layer 45

7.6. RLit Interpreter and Instruction class package 46

7.7. Storage layer 48

7.8. Bluetooth connection implementation 48

7.9. Robot Communication Layer 49

8. Chapter 8. Software testing 56

8.1. Prototype iterative test driven development 56

8.2. Integration testing 57

8.3. System testing 58

8.4. User evaluation 60

9. Chapter 9. Results 62

9.1. Story writing 63

9.2. Program execution 64

9.3. General impressions 66

10. Chapter 10. Analysis 67

11. Chapter 11. Critical comparison 68

11.1. Engagement 68

11.2. Promoting computational thinking 69

12. Chapter 12. Conclusion 71

13. Bibliography 72

14. Appendix A: RLit Ontological framework 75

14.1. RLit Story 75

14.2. RLit Sentences....76

14.3. RLit Phrases....77

15. Appendix B: RLit Phrase Maps 79

16. Appendix C: Table of RLit Phrases and their values 81

1. Chapter 1: Aims and Objectives

1.1. Overall aims

The aim of this project is to develop an application that can help introduce creative computational thinking into the Key Stage 1 and 2 classroom. A new Computing programme of study is being introduced into the National Curriculum from September 2014, in broad recognition that computational skill is now a basic skill that is required by everyone. It means that, for the first time, primary teachers will be required to plan and deliver lessons that teach basic programming skills (D.f.Education 2013).

There are a burgeoning number of products on the market which can be used to help teach such skills, including Bee-bots, floor turtles that follow in the tradition of Logo (Terrapin, 2013) , and a range of tablet applications, for example 'Move the Turtle (Turtle, 2013). However, for beginner programmers the overriding bias is placed on developing skills rather than inspiring creativity (see the critical review in Chapter 10), and this may not be enough to engage the imaginations of teachers and pupils who are less enthused by the process of programming itself than by the creative things that can be done with it. This application aims to help redress that balance.

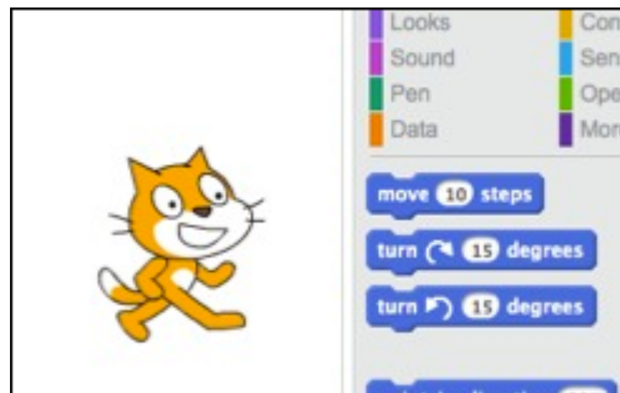
1.2. Pedagogical background

The project is inspired by the work of Seymour Papert and others at MIT Media Lab who since the 1980s have repeatedly shown how computational thinking can be encouraged in children. A recurring approach has been to connect abstract programming concepts with the physical world, via Logo turtles in the 1980s, physical micro-worlds and latterly Lego Mindstorms robots (Alimisis & Moro, 2007; McNerney, 2004; Papert, 1980). As a consequence, many schools and after-school clubs have starting running robotics courses, and there are competitive leagues to motivate children to become better programmers, for example RoboCup Junior(Eguchi, 2012). However, as discussed in the project proposal, these advances are taking place against the background of declining numbers of children studying computer science and science and maths-based subjects at higher levels(Falk, 2013). It must be the hoped that the new Computing curriculum will change this.

There must be a concern, however, that taking a isolated skills-based approach to teaching computing, even if it uses robotics, may not have the desired effect for some students. In their paper 'New Pathways into Robotics: Strategies for Broadening Participation', Rusk et al. quote this research finding from the American Association of University Women (2000):

"Girls and other nontraditional users of computer science – who are not enamoured of technology for technology's sake – may be far more interested in using the technology if they encounter it in the context of a discipline that interests them...computation should be integrated across the curriculum, into such subject areas and disciplines as art, music, and literature, as well as engineering and science' (Rusk, Resnick, Berg, & Pezalla-Granlund, 2008).

Rusk et al.(2008) propose that, for many learners, instead of focussing on design challenges, like building cars or moving robots through obstacle courses, structuring workshops around shared themes and storytelling would be more effective at engaging a wider diversity of students from early childhood onwards. Their work with programming tool Scratch has shown this to be true at least for older learners. As reviewed in the project proposal, Scratch allows users to create stories, art and games via a block programming interface, and now hosts more than 3 million projects.



Scratch, (MIT, 2013a)

It has been a major success, however according to the statistics on MIT's site, and shown in Figure 1.1, the vast majority of users are of middle school age and above, peaking at age 15 (MIT, 2013b). No programming environment has yet appeared for younger children that can inspire the creativity and dedication of Scratch users.

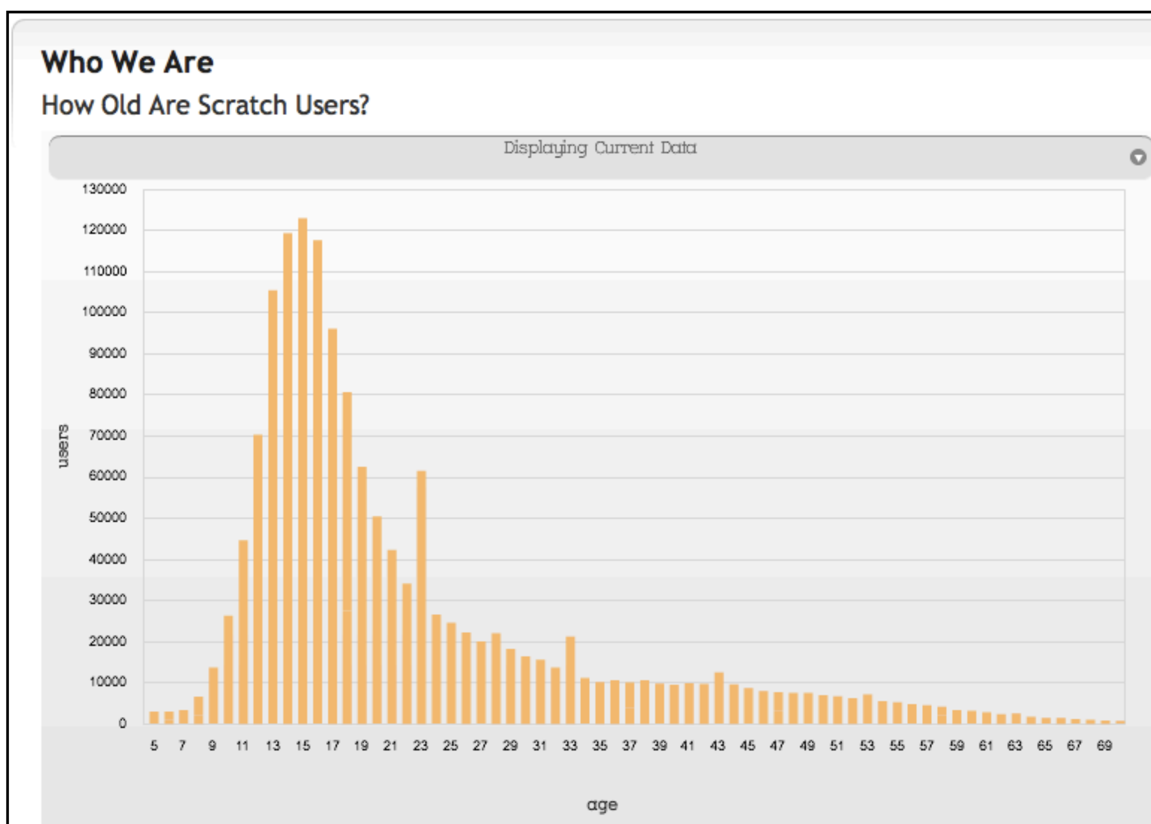


Fig 1.1 Table showing total users of Scratch, sorted by age, September 2013 (http://stats.scratch.mit.edu/community/)

As Ken Robinson says, 'focusing on skills in isolation can kill interest in any discipline...(t)he real driver of creativity is an appetite for discovery and a passion for the work itself.' (Robinson,

2013). This is clearly true based on creativity and passion shown by the Scratch community (Resnick, 2012).

At this critical time, it is vital that younger and less confident programmers can be inspired in the same way as 'Scratchers' are, and not be turned off by endless 'recipe-style' lessons (Alimisis, 2012) in order to meet the new curriculum requirements. If this happens, many more children may be driven away from IT and the sciences. But for busy teachers who are non-specialists in computing, getting it right will require lots of support. This project aims to make a small contribution towards this.

1.3. Technical background

From a technical point of view, the project proposal outlined the rationale for basing the project on Lego Mindstorms and Android. Young pupils have been found to engage easily with the robots, possibly because they are comfortable with playing with Lego bricks, and doing so reminds them of toys rather than assignments (Klassner & Anderson, 2003). They are also very powerful, with complex behaviours made available through coordinated use of the motors and sensors. This makes them more appropriate for context rich activities such as storytelling than the more basic Lego WeDo range, for example (Lego, 2013c).

As for Android, the platform is becoming rapidly more ubiquitous than iOS (IDC, 2013), and unlike iOS, Android devices can communicate via Bluetooth with the Mindstorms NXT model. In general, young children feel very comfortable using handsets, with a huge variety of games and educational applications available on Google Play for 4 year olds and upwards (Google, 2013f).

The original plan was to develop the software for the new Mindstorms EV3 model, thus being at the forefront of mobile development for this platform. In the end, the EV3 was never delivered, so the project would deal solely with the legacy NXT model, which has been on the market since July 2006 (Lego, 2013b). The technology for communication between Android devices and NXT via Bluetooth is well established, and has been integrated into products and platforms such as MindDROID (Lego, 2012), Cellbots (Cellbots, 2011), and leJOS ((leJOS, 2011). However none have arguably provided the range of Bluetooth control that this application provides, making comprehensive use of Lego's Bluetooth Developer's kit (Lego, 2006) to do much more than simply controlling the NXT's movement. By pushing NXT to its limits, the technical aim of the project is to lay the groundwork for future development with EV3. Making space for this would be one of the guiding principles of the system design for this project.

1.4. Objectives

In summary, the objectives of the project were to develop an Android application that controls a Mindstorms robot, and that does the following:

1.4.1. Promote computational thinking

The application should be able to be used to teach objectives from the new National Curriculum Computing Programmes of Study for Key Stage 1 and 2. More specifically, the application should be shown to convey some of the key computational thinking concepts and practices identified by Brennan & Resnick (2012) in their paper 'New frameworks for studying and assessing the development of computational thinking.'

1.4.2. Possess a fast learning curve

The application should provide an interface that young students and teachers can pick up and use instantly, without the need for a skills-based tutoring system. In other words, the interface should be 'intuitive' enough to allow users to start creating without them first needing to go through lots of recipe-style lessons to learn a new language of symbols.

1.4.3. Appeal to the 'dramatists'

As mentioned in the proposal, Bers (2008) discusses how young children start out by being 'both little storytellers and little engineers", but a division happens later on in schooling as a some children lose confidence in the new abstractions in the curriculum that they cannot tie to concrete experience. In another study, researchers at Harvard's Project Zero (Shotwell et al. 1979) show how children can be classified at an early age as either 'patterners' or 'dramatists' (Rusk et al., 2008). The objective of this application will be to engage the dramatists as well as the patterners.

A participant in RoboCup Junior's 'Dance' category (RoboCupJunior, 2013)

1.4.4. Be future ready

The application should work with Mindstorms NXT but be designed and implemented in such a way that it will be relatively easy to adapt to the EV3 in the future.

Lego Mindstorms EV3 - next generation of robot (Lego, 2013a)

2. Chapter 2. Approach

After the proposal was submitted in June, the approach to development had to evolve in response to requirements gathered in the early stages of the project.

The approach has remained closely aligned to using storytelling as a method for engaging children in programming. Technically, the project has kept the proposed approach of using an Android device to remotely control a NXT 'Robot Educator' Mindstorms robot (Falk, 2013). As requested by the tutor, the latest Android OS was used, JellyBean.

The major change has been with interface design, and the use of 'macro-behaviours'. These were defined in the proposal in a similar way to Logo's 'procedures', which can be broken down, reused and adapted by advanced users. The intention in the proposal was to provide a two-layered approach, with beginners sequencing the macro-behaviours along a simple timeline to tell their story, as shown in Figure 2.1, whilst the advanced students could drill down and adapt the procedures/behaviours with a Scratch-style interface, as shown in Figure 2.2.

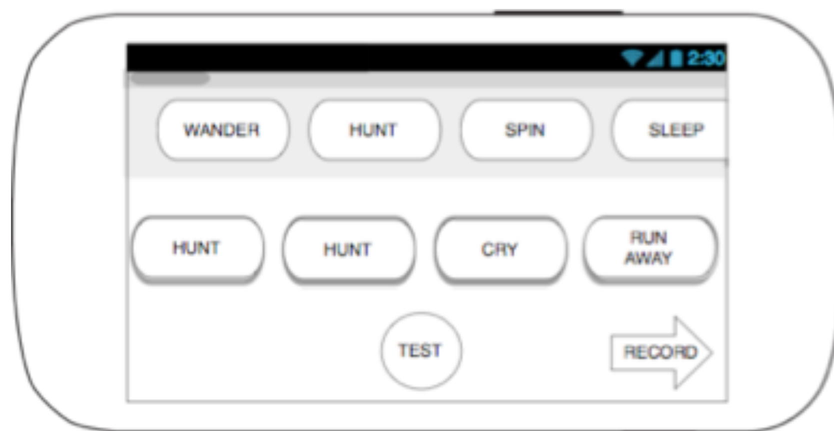


Figure 2.1 The 'basic' interface from the proposal

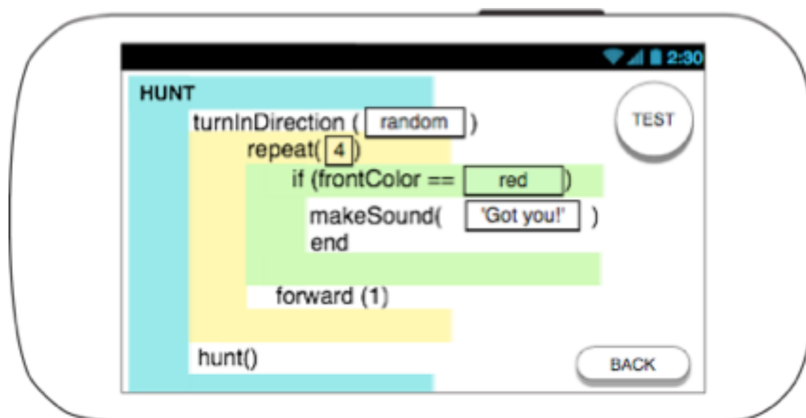


Figure 2.2 The 'advanced' interface from the proposal

After some paper testing and several iterations of the prototype it became clear that neither level would sufficiently meet the objectives of the project. The basic interface was simply not powerful enough to allow users to tell a story - with this interface it is impossible to answer important questions like '**What** is the robot hunting?', '**Who** is speaking?', '**Who** or **what** is he

running away from?' - questions that are central to building a meaningful story. As well as having no sense of the actors, no dialogues could be built. Finally, it was clear after building the prototype that the rigid movements of the robot and quiet speakers could not adequately convey these simple behaviours without more information being provided to the user.

The advanced 'Scratch'-style interface had different issues - like Scratch, the blocks and drop downs were simply too complicated for the younger users.

An alternative 'middle-way' approach had to be found that would provide enough detail for meaningful stories, but in a straightforward way. At this stage, it was decided that if the application was to hook 'storytellers', it was worth trying an interface that was familiar and comfortable to them - an interface that used natural language.

2.1. Natural language programming

The language of 'sentences' , 'words' and 'texts' is often used when describing the parts of a programming language - for example, in Logo these are all considered sentences:

```
[200 50]
[forward 100 right 90]
```

(Logo Foundation, 2012)

However, in a natural language sense, these sentences are more like a list of words. 'A natural language program' (NLP) is differentiated in that it has a set of ontological rules that govern how sentences can be constructed and interpreted to form 'real' readable sentences that reflect written language(Wikipedia, 2013b)(Wikipedia, 2013a)(Wikipedia, 2013b).

The use of natural language in computing was first popularised in the late 70s and 80s with text adventures such as Colossal Cave and The Hitchhikers Guide to the Galaxy (BBC, 2008). There still exists a strong interactive fiction (IF) community that uses engines such as Inform7 to author IF games using natural language programming (Inform7, 2013) . In the commercial world, sEnglish is a NLP that is used to control robots and manufacturing processes .

One statement in particular on the sEnglish website indicates how NLP could be useful in the educational setting:

"NLP does not separate programming from the application meaning. ***The program is the manual and the manual is the program"*** (SysBrain, 2007).

This suggests NLP could be of benefit when teaching computational thinking in the classroom. Teachers could potentially discuss programming concepts and practices without ever having to leave the language of the program itself. Another benefit highlighted on the website is that NLP can be shared with both fellow coders and non-specialists alike. - as an NLP program reads like an English document, anyone can potentially understand it. Again, for the objectives of this project, this seemed a very good fit - to have a programming language that could be easily shared between groups of students, and understood by both non-specialist teachers and beginning pupils.

Other potential benefits of using NLP:

- it is interface neutral, allow children to write programs even if they are not actually with a device. This would be important for classrooms with a pressure on resources

- Programs could be written on the board, on handouts, or cut up for paper assembling exercises. Sentences could be segmented into flip books and used to collaboratively write programs as a whole class or in groups
- NLP offers a much more accessible way to program for the blind and hard of seeing than other visually based programming interfaces
- It is firmly cross curricular. Aside from introducing literacy into computing, it could work the other way round with program writing becoming a valid activity during Literacy time

In choosing NLP as an approach, there was a worry that it would just be too challenging to achieve in the timespan of the project, with the demands of creating a whole new ontology-based language for the interface, however I thought it was an affordable risk considering the potential benefits.

2.2. LCP Bluetooth control

Through several iterations of prototyping, it became clear that all control needed to reside with the Android device, and that Bluetooth communication was the best means of achieving this. If the application was going to be telling users' stories, then the power of Android to control both audio played by the device and the movements of the robot was needed.

Approaches that were considered in the proposal, such as replacing the NXT firmware with Java-based leJOS (leJOS, 2011), were rejected because in tests it was discovered that leJOS only provides a small fraction of the control over NXT via Bluetooth than what is provided by the Lego Communication Protocol (LCP). To get the maximum amount of functionality out of the Bluetooth connection, it was considered better to deal directly with LCP, and its component byte code that underlies all Bluetooth communication between the device and Android. Lego has very clear documentation on all the byte code calls and this was used extensively (Lego, 2006), along with examples taken from the MindDROID application, (Lego, 2012)

3. Chapter 3. Methodology

As outlined in the proposal, a phased-development approach was followed, preceded by a phase of rapid, iterative prototype development. The prototypes ensured that by the time the main development cycles started in August, the scope of the project was clear.

The first phase in June completed the requirements gathering that began with the proposal, and in July a series of prototypes were developed and tested. After a final specification was developed, the final development occurred in two phases, beginning with development of the UI and model components, and ending with the robot control component in August. After integration testing, the finished application was tested with children and teachers in local schools.

3.1. Requirements gathering - June-July

Interviews and observations were carried out with three children over a two week period using pre-existing software developed for Android and Mindstorms, and other tablet software that covers objectives related to the new Computing programme of study. The results of the latter study are contained in the Critical comparison chapter at the end of the report.

During the same period, there was extensive document analysis, during which time I learned Android and Lego Communication Protocol.

3.2. Prototyping - July

Four iterations of prototype were built in rapid succession that explored in increasingly depths the scope of control that Android could have over a NXT robot using LCP. The results of the prototypes would define the final requirements of the application

3.3. Ontology development - July

The ontology for the ‘RLit’ language was developed and paper tested.

3.4. Two phase development - July-August

The main application was developed in two phases, the first focussing on the model and UI layer, and the second focussing on the robot control layer. The latter phase was heavily based on the learning gained from the prototypes

3.5. Integration testing - August

Finally, the different components were combined and tested extensively, entailing a lot of refactoring of all components

3.6. System testing - September

The application was tested as a system in September with the help of three children, and further refactoring was needed

3.7. User testing - September

The final application was tested with three groups of four children at a local school in Jaffa, Israel.

Detailed descriptions of each phase follow.

4. Chapter 4. Requirements Gathering

Aside from the background research outlined in the project proposal and summarised above, three methods were used to gather requirements for the project - Interviews, Document Analysis and Prototype development and testing.

4.1. Interviews

Three children were interviewed and observed over a two week period using various pieces of software for both developing general programming skills and for controlling a Mindstorms robot. The latter included MindDROID (Lego, 2012), and NXT Remote Control (Fedor, 2011), both of which offer basic remote control over the robot's motors. Aiden's experiences with the programming software is covered in more detail in the Critical review section at the end of this report.

Aiden (aged 9) performs near the top in his class in English and Maths, and is quick to master computer interfaces. He is a member of Club Penguin and visits it on average once every week, mainly to play the games. He has been given a Wordpress site, which he sporadically updates it with performances and puppet shows. He plays the piano and shows aptitude in all subjects, but especially keen on creating music and writing stories. He could be classified as a 'dramatist'.

He quickly lost interest in the various remote controller applications available for Mindstorms.

He showed interest in writing stories for the robot to act out, but when he was shown the proposed interfaces for this project, he wasn't sure what either one did. When told that they were to help him tell a story with the NXT, he couldn't say how he would achieve this. It was clear from his reaction that serious thought needed to be put into both interfaces from the proposal.

When asked about what he would like to control the robot to do, he mentioned that he'd like to create an adventure story with the robot.

Leo (6) is beginning to write, but struggles in school with Maths in particular. His interests are street dancing, art and football. He is not confident using the computer, and generally plays games on CBBC and Cbeebies, and getting print outs to colour in.

He enjoyed using the MindDROID application, and was skilled at moving the robot around the room. When asked what he would like to get the robot to do, he mentioned 'chasing people' and 'you tell it do something and it does it'. He liked the fact that the proposal's interface offered the chance to 'HUNT' and 'CRY' but when asked how he would make a story using these elements, he wasn't sure.

Llla (5) got on really well with NXT Remote Control and MindDROID applications. Although her reading level wasn't at a level where she was confident to read the words in the 'basic' interface, when it was explained what she would have to do, she said she would like to tell a story about a fairy robot who 'does magic', but couldn't explain what she would have to press to achieve this with the interface.

Early requirements gathered from this stage therefore reinforced the research that storytelling could be an good way to engage the 'dramatists', and that it was important for the application to allow for complex interactions making use of both motors and sensors. However it was also clear that the proposed interfaces were not sufficiently rich or user-friendly to achieve this, and major reworking would be needed.

4.2. Document analysis

As an Android beginner, I had to start with the basics and work my way through the various training modules available on the Android Developers' site (Google, 2013a). Then, to understand BluetoothAdapter API for Android, I analysed the sample Android application 'Bluetooth Chat' that demonstrates how to search for and connect with Bluetooth devices (Google, 2012). To understand the byte-code calls involved in Lego Communication Protocol, I analysed the online Developer's API documentation (Lego, 2006), which was also served as a constant reference throughout development. Lego's MindDROID application for Android provided important examples for how to apply LCP in a working application (Lego, 2012).

4.3. Prototyping

4.3.1. Prototype 1- Testing connections

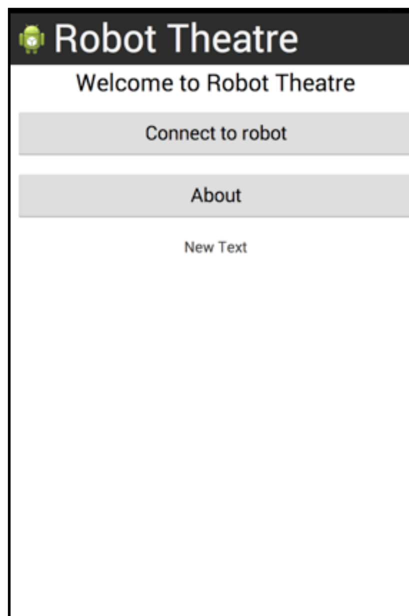


Figure 4.1 User Interface for Prototype 1

Based on the results of the interviews, and document analysis, a series of prototypes were then developed to test the technology and get reactions from the interviewees.

The first prototype, shown in Figure 4.1, was developed to test connection via Bluetooth with the Mindstorms NXT robot. The code relied heavily on the implementation code of Google's Bluetooth Chat sample application, and few problems were experienced making this connection.

Once a connection was successfully made, a new Activity was created which would serve as the main Activity for controlling the robot. This formed the basis for the second prototype.

At this point, the application was to be called 'Robot Theatre'.

4.3.2. Prototype 2 - Controlling the robot via Bluetooth

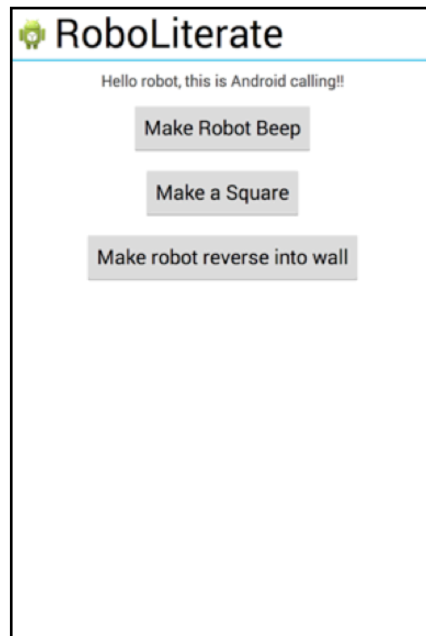


Figure 4.2 User interface for Prototype 2

The next prototype was built in rapid iterations, aiming to test increasingly complex sequences of robot actions to see what would be possible via Bluetooth. The completed prototype is shown in Figure 4.2.

All interactions with the robot were coded on the main UI Thread, so the application would pause during each interaction. The purpose of this was to quickly identify glitches in the communication code.

Step 1: Send a simple instruction

The prototype sent a simple instruction to the robot, not requiring any reply - to beep at a certain tone and duration.

This first test was successful, and I was able to begin planning an architecture that would allow sequences of instructions to be executed.

Step 2*: *Send a sequence of instructions

The second step was to perform a series of instructions sequentially - “Make a Square’ - with the following code.

```
program.addInstruction(Move.getInstance(RobotInstruction.NO_DELAY,10,50,true));
program.addInstruction(Turn.getInstance(RobotInstruction.NO_DELAY,90,50,true));
program.addInstruction(Move.getInstance(RobotInstruction.NO_DELAY,10,50,true));
program.addInstruction(Turn.getInstance(RobotInstruction.NO_DELAY,90,50,true));
                                   etc.
```

In this step, difficulties were experienced maintaining an accurate ongoing measurement of distance and angle of turn of the robot, due to Bluetooth reliability issues - high and jittering latency, limited throughput and general stability issues (Gobel, 2011). This was combined with the NXT latency in sending replies back to Android - up to 100ms for each command (Lego,

2006). Instead of abandoning Bluetooth, which would have resulted in a complete change of approach, I decided to adjust the architecture so that only a minimum number of 'reply' commands would be needed for each action. So, for drawing the sides of a square, instead of constantly polling the NXT for the tacho count (rotations) of the motors, a fixed distance to travel was sent the robot. When the motor speed was detected to be 0, the next command (Turn) was sent, and so on. Although this still didn't provide the smoothness and accuracy needed for a mathematically exact square, it was considered good enough to accomplish the aims of this project.

Step 3: Sequence of instructions with sensor listening

The next stage in the prototype was to build a sequence of instructions that included the monitoring of a sensor. For this purpose, the simplest test was to use the Switch sensor because it is a passive sensor that has a simple boolean ON/OFF value. The test was to reverse the robot until it hits a wall, whereby the switch sensor would be pushed, and the robot would stop moving.

An event listener class was introduced which continually monitored the switch sensor in a *while* loop until it detected a positive boolean. Once this was detected, the sequence continued to the next instruction.

This stage proved successful, meaning it would be possible to include event listeners in the final specification.

4.3.3. Prototype 3: Complex behaviours

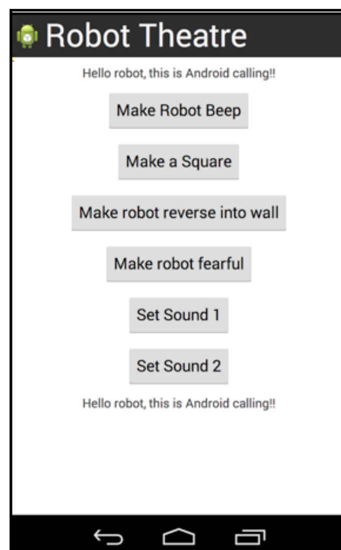


Figure 4.3 User interface for Prototype 3

To push things further, I tried to 'make the robot fearful'. This involved constructing a sequence of actions that included using the ultrasonic sensor to detect the proximity of an object. This provided more challenges since the ultrasonic sensor is a digital sensor with its own microprocessor, and requires a special set of LCP calls utilising the sensor's I²C (Inter-Integrated Circuit) communication bus (Lego, 2006).

Another step was to test how sounds could be uploaded to the robot - I wanted the robot to say 'Leave me alone!' as part of the 'Be fearful' sequence. This was implemented through document

analysis of LCP and the example used in the MindDROID application, and a separate class was developed to do this. A variety of different sounds were tested.

This also proved successful. At this stage, I decided there were enough interactions to provide rich choices to build stories.

4.3.4. Prototype 4: Service architecture

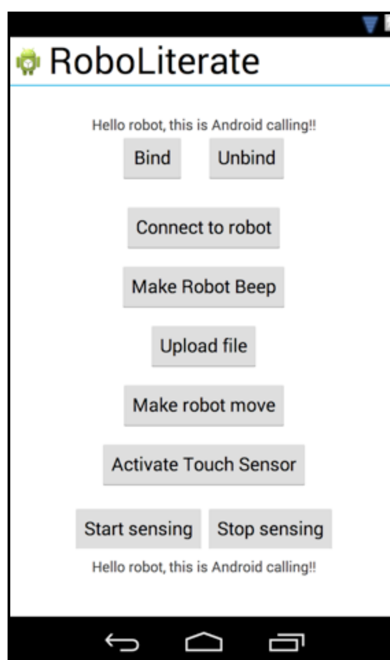


Figure 4.4 User interface for Prototype 4

The final stage in prototyping was to test how robot control could be achieved via a background Service rather than on the main UI Thread. This was achieved through analysing Google's documentation on Bound Services (Google, 2013b) and other projects that had implemented service architecture to communicate with Bluetooth devices (for example, see Gobel 2011)

With all Bluetooth communication moving from the UI thread to a background service thread, the application no longer 'hanged' whilst communicating with the robot.

During this phase, in discussion with another primary school teacher, the application name was finalised as 'RoboLiterate', since the application deals with robots and the only prerequisite for users would be that they were literate in basic English. The name also emphasises that the programming language that is 'understood' is literal English.

5. Chapter 5. Requirements Specification

Enough details had now been gathered to define the final requirements. The following sections summarise the user and technical requirements.

5.1. User Requirements

Users should be able to write and run programs using complete English sentences via an NLP interface. The interface should be straightforward enough for a literate 6 year old to use, and allow users to do the following:

- write a program, or 'story', in full English sentences on an Android device
- adapt a 'story' written by another user
- save their 'story' for later
- run and test their 'story' on a connected NXT Mindstorms robot

Figure 5.1 summarizes the use cases for these overall requirements.

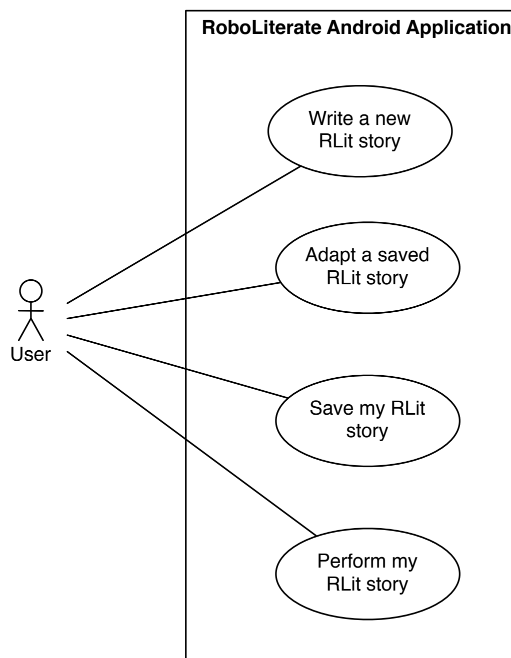


Figure 5.1 Summary use case diagram for RoboLiterate application

5.1.1. Use Case 1: Write a new RLit Story

Users should be able to author stories, sentence at a time. In building a sentence, users sequence phrases together, which then form a sentence and the sentence in turn becomes part of the story. Users should also be able to reedit sentences or delete them once they have been added to the 'story'.

Figure 5.2 shows a use case for the above requirements.

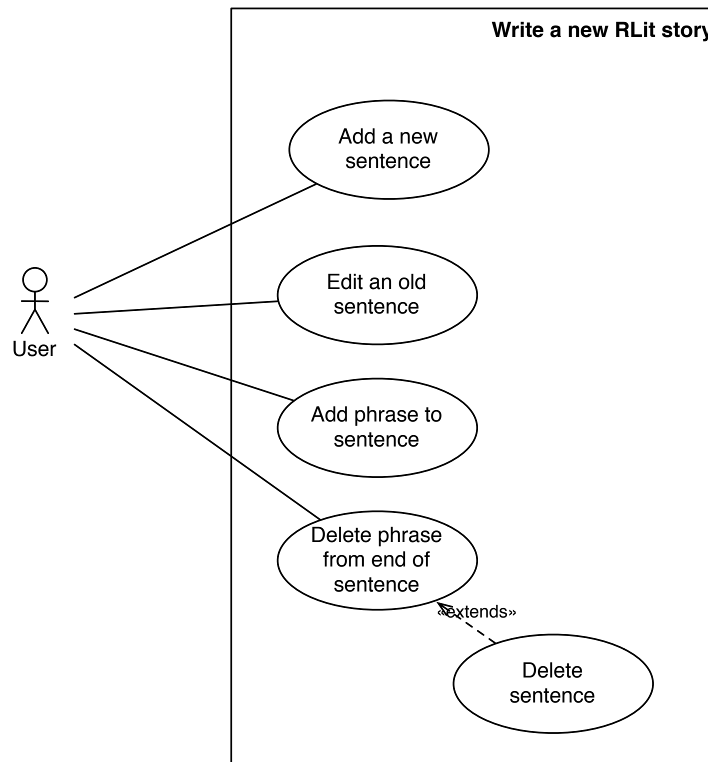


Figure 5.2 Use case diagram for writing RLit stories

5.1.2. Use Case 2 and 3: Adapt and save stories

Since the ability to reuse and adapt other people's work has been identified as a key practice for developing computational thinking (Brennan & Resnick, 2012), it was important to include in the user requirements the ability to adapt stories written by other users which have been saved on the system. This would hopefully help users get inspired and not always begin with a 'blank slate'.

This requirement entails the third use case, which is the ability to save one's story once it is written.

5.1.3. Use Case 4: Run and test stories whilst connected to a Mindstorms robot

Once written, the stories are interpreted by the application into code-readable Instructions that are executed in sequence by the robot and Android device.

Before a story can be performed, the user should be able to scan for, select and connect with a compatible Mindstorms robot via Bluetooth and configure the ports so they match the onscreen display. This should be achieved in as simple a way as possible.

Whilst the story/program is being executed, the user needs to be able to monitor on the Android screen what is happening, including having information on:

- file upload status
- any errors

- Information on when the program has begun and ended
- What stage in the program execution they are at, so that they can pinpoint errors such as infinite loops, hanging event listeners, etc.

Figure 5.3 shows a use case diagram for these requirements.

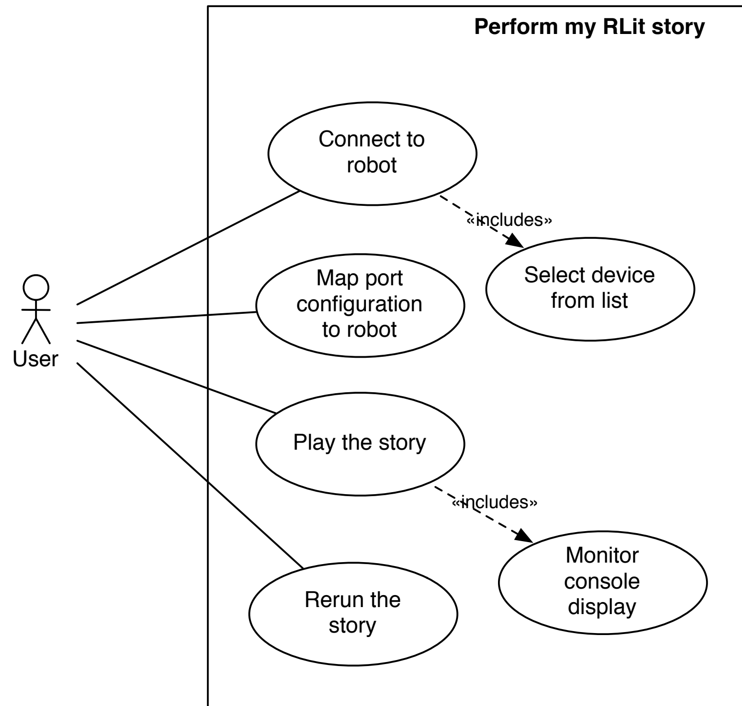


Figure 5.3 Use case diagram for running and testing RLit programs.

5.2. Technical requirements:

The main technical requirements were:

- to implement a JellyBean Android application that remotely controls an NXT robot
- to coordinate the robot's actions with Android sounds and music
- to map this coordination to the sequence of instructions defined by the user
- to translate the instructions from NLP sentences as defined by defined RLit ontology, specified below
- to upload sounds to the NXT
- to read information from the following NXT sensors: ultrasonic, sound, light, switch
- to read the 'tacho count' (motor rotations) from each NXT motor port
- to direct the NXT to move in a direction, rotate, play sounds and beep

- to ensure the application can work with the EV3 in the future
- to work on both handset and tablet size screens.

5.3. NLP specification

Part of the specification stage was to define in detail the ontological rules governing the NLP language, ‘RLit’. These rules needed to be in place before system design could begin.

As this ended up assuming the workload of a separate project in its own right, the fine details of RLit and design decisions taken along the way are contained in Appendix A. What follows is the main points as they relate to the main user and technical requirements of the project:

- The language needed to provide a range of sentence constructions that would allow the user to control a Mindstorms robot via Bluetooth, including setting motor actions, reading sensors, and playing sounds and notes.
- The language would also need to allow users to coordinate with this music tracks and spoken dialogue from the Android device.
- The language would not need to provide users with the fine grained control over the motors and sensors that, for example., the Lego NXT-G software does (Griffin, 2010). So, for this project, it would be sufficient to have a sentence like ‘ROBOT moves forward a little and steadily.’
- The language should however retain the potential to be developed in this direction in future iterations to allow for controlling variables such as speed, distance, degrees of turn and sensor thresholds.
- The language would need to include the ability for looping statements, and event handling. This was essential in order to cover some of the programming objectives in the Key Stage 1 and 2 Computing curriculum
- The language would need to support concurrency, so that more than one event could happen at the same time, for example moving the robot around whilst playing music from the Android device. This would bring a freedom to the user’s programs and also allow for interesting situations that would require the user to debug. Again, this is a stated objective in the Key Stage 2 curriculum.

6. Chapter 6. Software Design

The following sections outline in detail the system design of each tier with further explanation as to the design decisions made at each stage.

6.1. Overall design

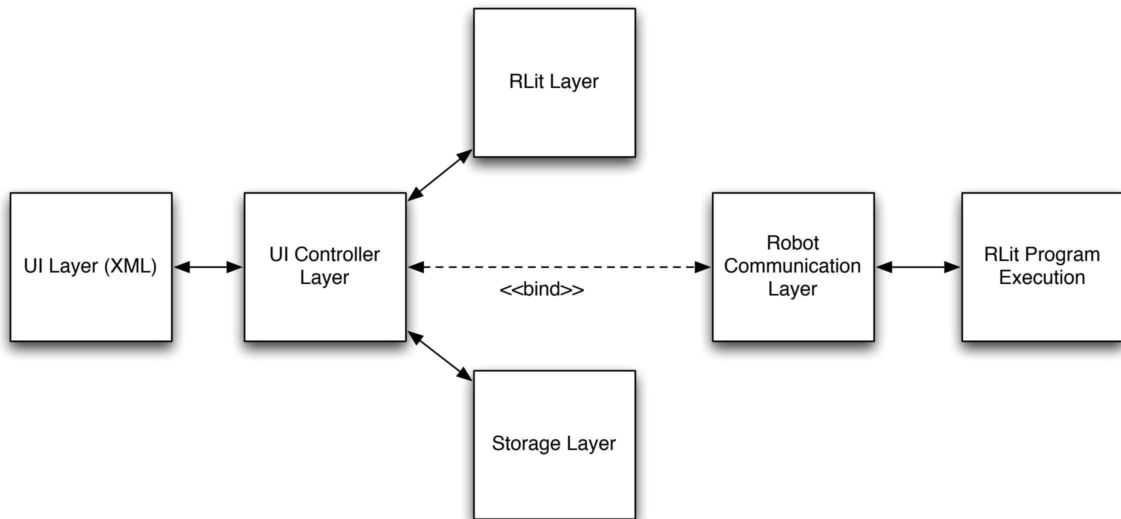


Figure 6.1 Overall architecture of RoboLiterate

RoboLiterate is designed as a multi-tier application, based on Android's MVC (Model View Controller) architecture (Google, 2013a). A summary of the architecture is shown in Figure 6.1.

6.2. UI and UI Controller Layers

The UI layer is composed solely of XML and resource files which are accessed and inflated by Activity and Fragment classes in the UI Controller layer. The controller layer also provides access to the RLit model layer, which contains classes implementing the ontology of the RLit natural programming language, as detailed in Appendix A. The controller layer connects with a storage data layer that is composed of an SQLite database, allowing users to save and load stories and sounds.

After the controller layer connects via Bluetooth with the robot device, all subsequent communication with the robot is managed by a background Service, which the controller layer binds to. The controller sends via Messenger objects all the information required for the service to communicate with the robot, including configuration choices and RLit program instructions. Robot and other program events are then relayed back to the UI controller layer in the same way.

This separation of concerns provides maximum flexibility for expansion and adaptation in the future. By separating out the UI layer from the controller layer, any number of different permutations of design can be used without affecting the rest of the application. Similarly, the RLit model can be used independently in a different application environment.

6.3. RLit Layer

6.3.1. Overall structure of RLit

'RLit' is designed as a natural programming language (NLP) specifically for the purposes of this project.

Like other NLPs, RLit's foundation unit is the 'sentence'. Each sentence represents a distinct instruction for controlling either the robot or Android device. The phrases within each sentence modulate how and when these instructions are performed. In turn, a set of sentences make up a 'story'. The 'story' defines how the sentences are executed relative to each other. Once written, an interpreter analyses the story and converts it into a 'program' that can be subsequently executed. Figure 6.2 illustrates the mapping between RLit components and Robot program components.

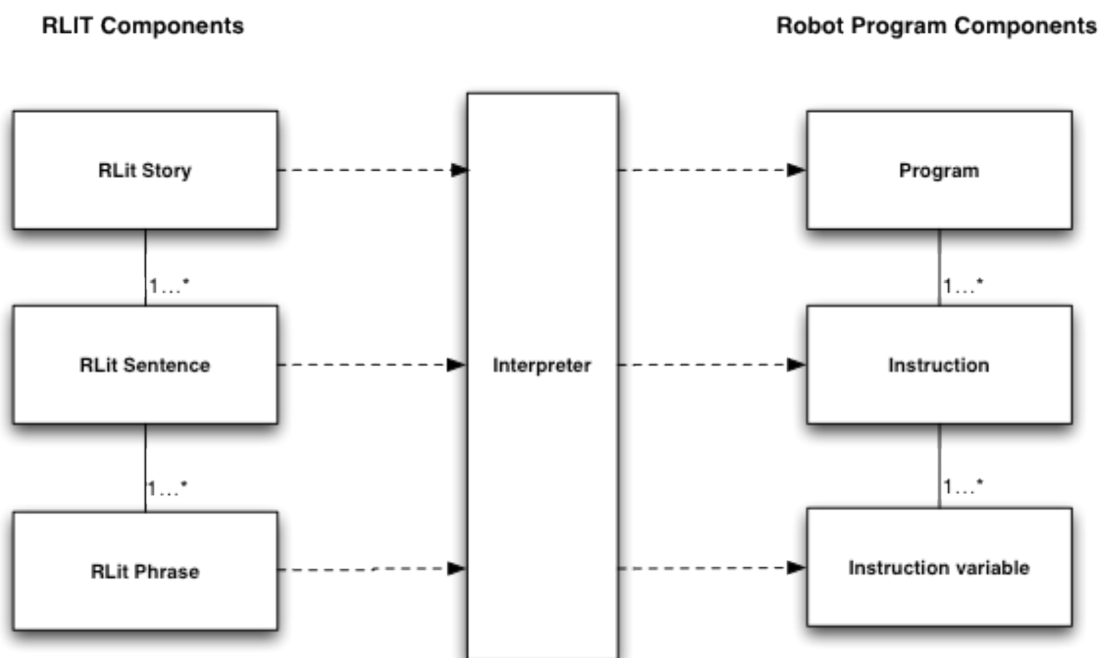


Figure 6.2 Relationship between RLit elements and program instructions

Unlike a fully developed NLP, the user is guided through writing an RLit sentence, step by step. This ensures that the process is as simple as possible, especially for Key Stage 1 pupils.

6.3.2. RLit Phrase hierarchy

Phrases are organised into hierarchical structures defining their exact ordering in sentences. Figure 6.3 shows a fragment of this structure. Two possible routes for RLit phrase combinations are shown, one for combinations that instruct the robot to move, and the other to instruct the robot to wait for a target reading from the ultrasonic sensor. Each group of phrases has an ID, and their children are shown by the arrows. So if the phrases 'First' then 'ROBOT' are selected, all phrases that have the parent ID of ROBOT (ID 10), are displayed to the user. If the phrase 'moves forward' is selected, then all phrases with parent ID 105 are shown. This process continues until all possible phrases are exhausted.

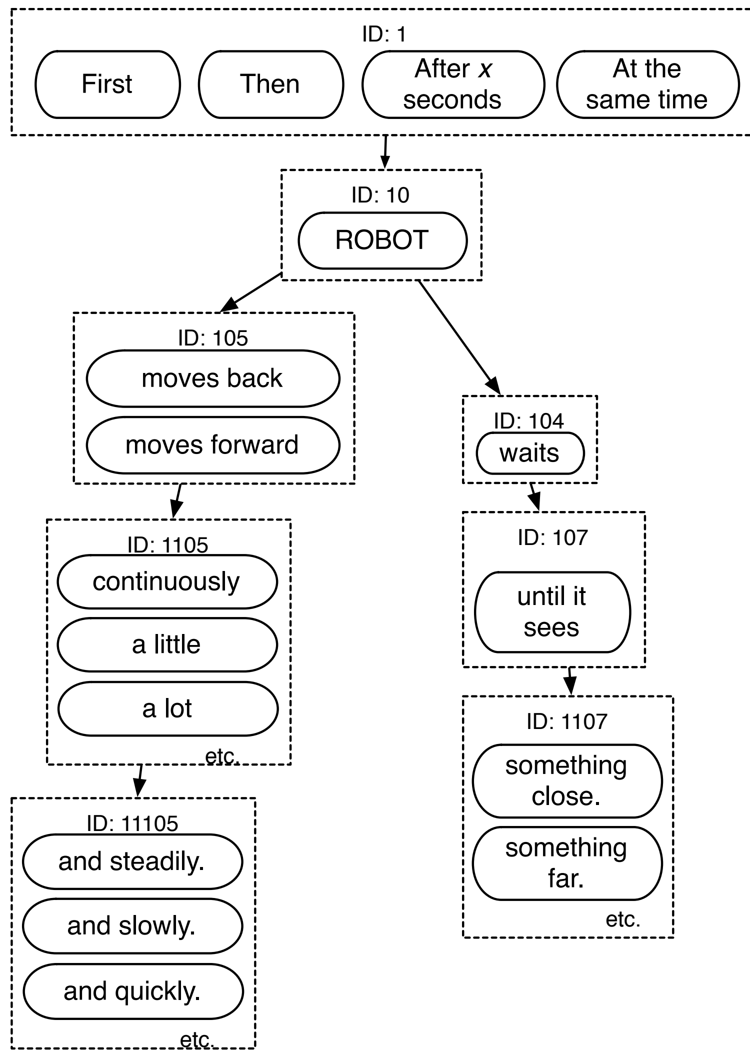


Figure 6.3 Fragment of RLit's hierarchical design

RLit was developed according to an ontological framework which provides enough power for users to program the robot in ways that can only otherwise be achieved using Lego's NXT-G software, RobotC (RobotC, 2012), etc. However, fine grained control was deliberately left out (e.g. instead of saying 'Move Forward 10 units', 'Move forward quite a bit') to provide a soft storytelling environment for users rather than a mathematically exact one. However, this functionality can be added at a later date.

Figure 6.4 shows a sequence diagram illustrating how users build their stories through adding phrases to make sequences of sentences. The figure also shows how the story that users create needs to then be interpreted by the system into a list of instructions for the robot and Android device to perform.

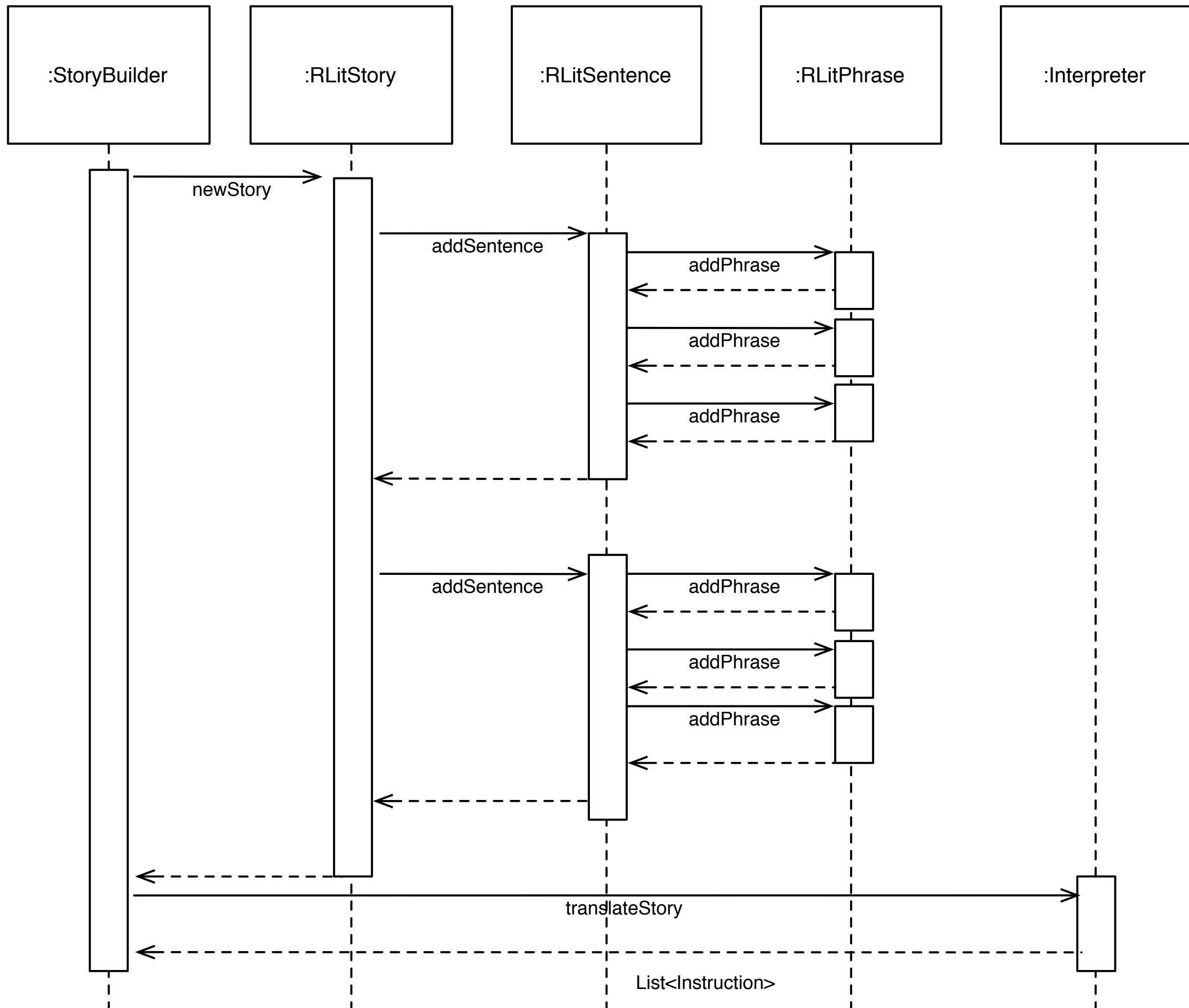


Figure 6.4 Sequence diagram showing the process of story authoring and interpretation by RoboLiterate

Appendix A contains further details on the ontological structure of RLit.

Appendix B contains a map of RLit's complete phrase structure.

6.3.3. Translating RLit into Instructions

The role of the interpreter in RoboLiterate is to take the Stories constructed in RLit and convert them to sets of instructions that the robot and Android device can execute.

The interpreter ascertains which Instruction type to create based on the InstructionID provided by the verb phrase in each sentence.

Each Instruction is represented by a unique ID, and several RLit verb phrases can share the same Instruction ID if they describe variations on the same Instructions. For example, the RLit verb phrases 'move back' , 'move forward', and 'stop moving' all share the Instruction ID INSTRUCTION_ROBOT_MOVE. The verb phrases 'turn left' and 'turn right' share the Instruction ID INSTRUCTION_ROBOT_TURN. On the other hand, there's only one way to beep, so the verb phrase 'BEEP' is unique in being the only verb phrase to have the ID INSTRUCTION_ROBOT_BEEP.

Once the Instruction type is extracted, it is instantiated by the Interpreter with parameters that are defined by an analysis of the field values of the totality of phrases in a sentence. Any particular phrase field value means something unique depending on the Instruction to which it is attached. For example, arg2 in the RLit phrase 'in a full circle' refers to the degree of turn for the instruction INSTRUCTION_ROBOT_TURN, whilst arg2 in the phrase 'something dark' indicates the upper threshold for the light sensor in the instruction INSTRUCTION_ROBOT_WAIT_LIGHT_SENSOR.

Once the Instructions are built, they are saved as a 'Program' - a simple list of Instructions that in turn contain all the data necessary for the Robot control classes to use during program execution.

6.4. Robot communication architecture

The design had to allow for the later substitution of new models of Mindstorms robots in place of the NXT device. In addition, the design needed to allow for future versions where multiple devices can be controlled, or where multiple Android devices could work together. For this reason, Android's Bound Service architecture was chosen, since it allows for multiple applications to connect to the same service (Google, 2013b). Using this architecture would also allow for the loose coupling required to allow new robot models be substituted in without affecting the other components of the application.

Through iterative development of the prototypes, and analysis of Lego's LCP documentation, the MindDROID app, and other documentation, five different families of robot commands were identified:

- direct commands that control movement and sound
- system commands to upload files
- 'reply' commands that obtain readings from the motor ports
- reply commands that obtain readings from passive sensors (e.g. switch) and active sensors (e.g. sound)
- specialist commands that send and receive information from digital sensors (e.g. the ultrasonic sensor)

Although I had no access to EV3 documentation, I assumed that the EV3 would behave differently in each of these cases, possibly drastically. In this case, it was necessary to design a system that encapsulated these behaviours in separate classes in a Strategy pattern, as shown in Figure 6.5.

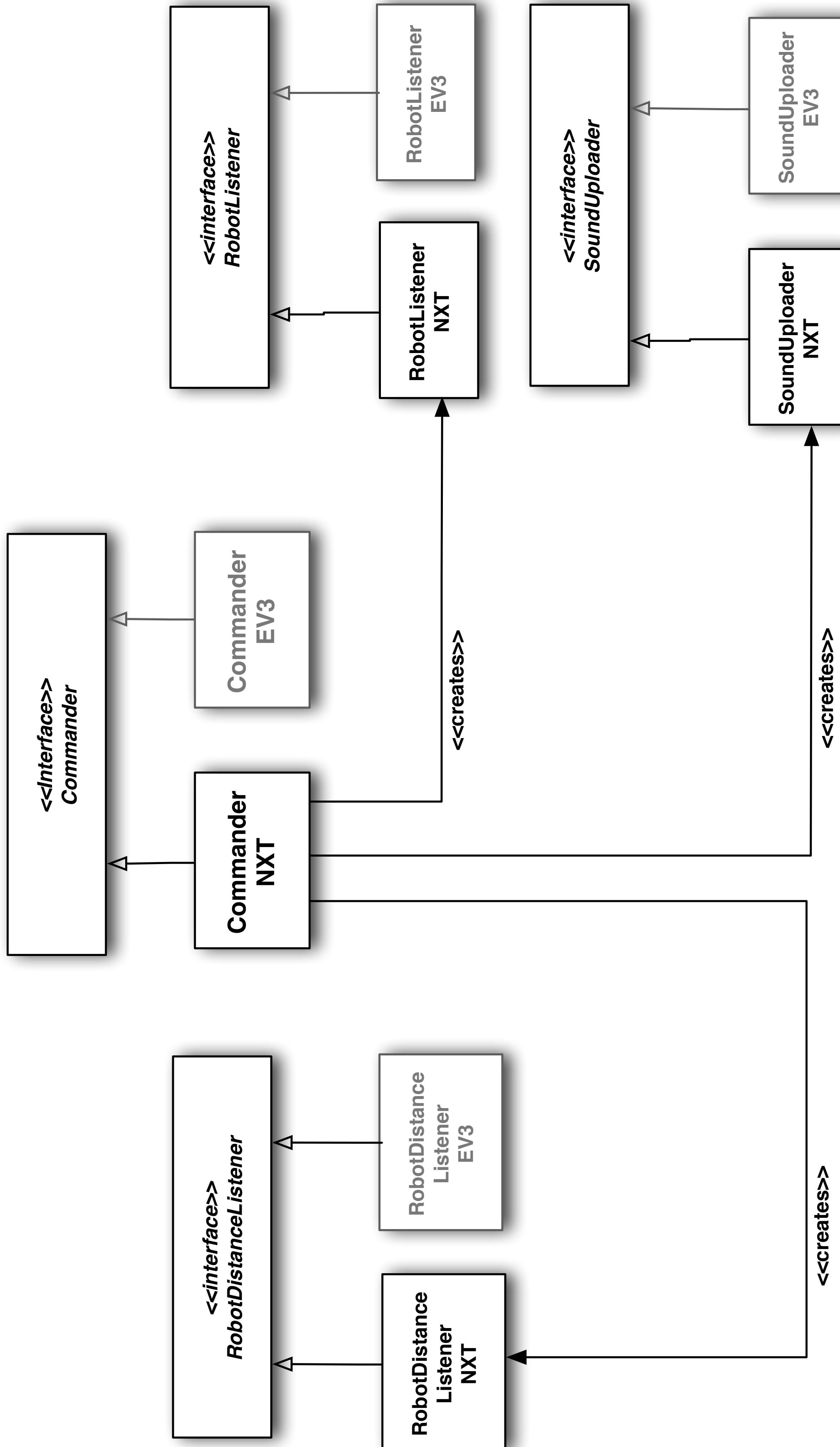


Figure 6.5 System design for robot control classes

7. Chapter 7. Software Implementation

The software was implemented and tested in four stages:

- Phase 1: Implementation of UI Layer and RLit model
- Phase 2: Implementation of robot service layer, based on prototype
- Phase 3: Integration of UI Layer, RLit model and Service layer
- Phase 4: Refinement of UI layer and RLit model based on outcome of user testing

7.1. UI Design

The following subsections detail each of the major user interface activities.

7.1.1. Launch activity

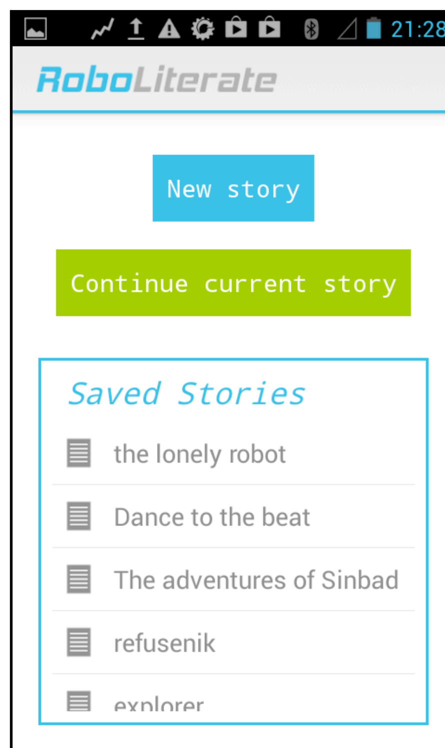


Figure 7.1 Opening screen of RoboLiterate

RoboLiterate's welcome screen allows users to start a 'New Story', and if they have returned to this screen from editing their own story, they have the option to continue with their current story. There are also a list of stories that have been saved to local storage which they can open, play and adapt.

As with the rest of the app, the design is kept minimal and functional, using the default colours of Android's Holo Light template. This is to keep the experience familiar and comfortable and avoid adding anything that might detract from the phrases and sentences.

7.1.2. Story builder activity - unpopulated

The aim of the interface design is to allow for an gentle learning curve which a confident literate year 2 child would find easy to pick up on his or her own.

Because of this, the design provides users with a limited number of choices at each step, and tries to make these choices very clear. So, in the opening screen of the Story building interface shown in Figure 7.2, there is what appears to be an empty sheet, and one main button, 'Add sentence'.

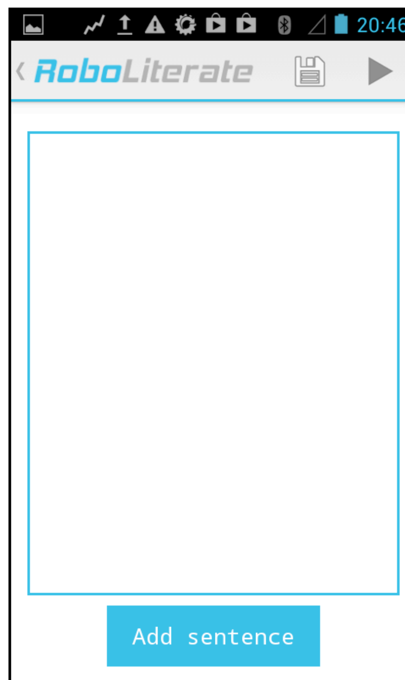


Figure 7.2 Opening appearance of Story Builder activity

The two buttons on the Action Bar allow users to Save their story and to Play it.

7.1.3. Sentence builder activity

Once 'Add sentence' is pressed, the Sentence builder screen opens, which again aims to be very clear about where to start. All stories begin with the phrase 'First', so this is the only option presented, apart from the Back button next to the editing window. This layout is shown in Figure 7.3.

As the user adds phrases, a new set of phrases appears at the bottom of the screen, as shown in Figure 7.4. Once a sentence has been built, users have the option to Accept the sentence or delete what they've done, phrase by phrase, shown in Figure 7.5

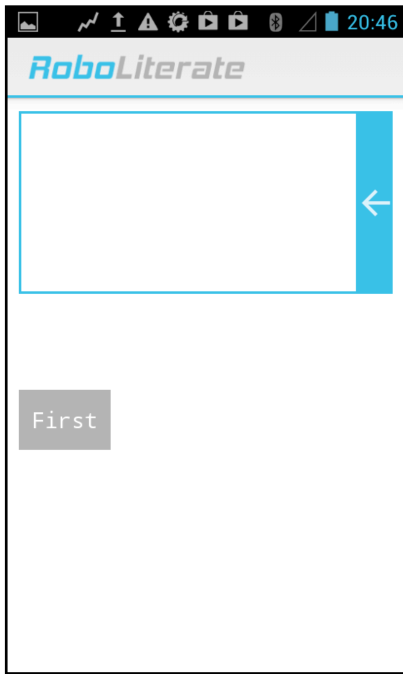


Figure 7.3 Opening appearance of Sentence builder activity

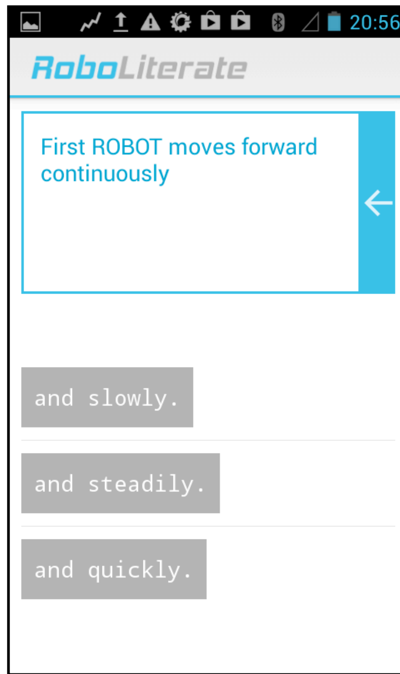


Figure 7.4 Sentence builder activity, part-way through a sentence

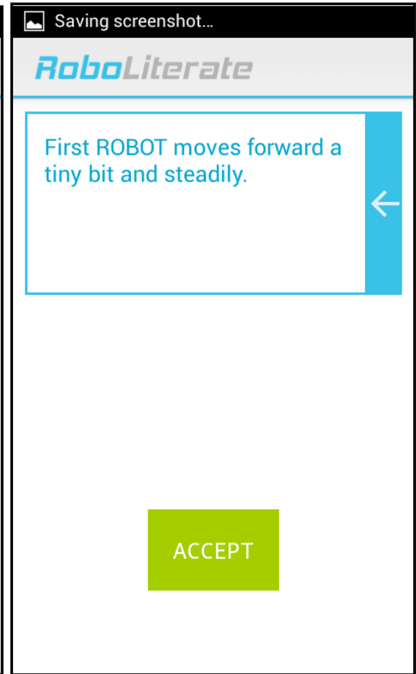


Figure 7.5 Sentence builder with a sentence completed.

The user also has the ability to record their own audio. This option appears in the ActionBar at the relevant stage of sentence construction, when the user has already selected the phrases 'ANDROID' and 'says' for their sentence.

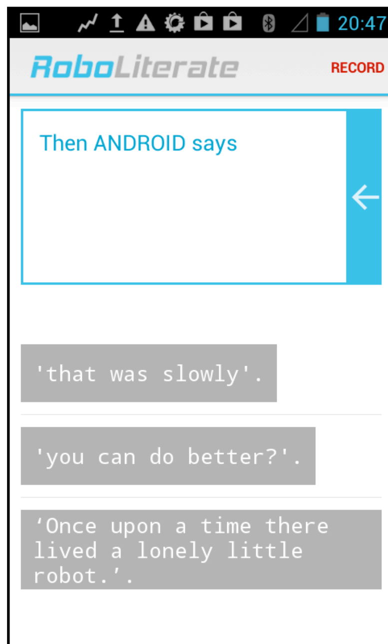


Figure 7.6 Option to RECORD in the Action Bar

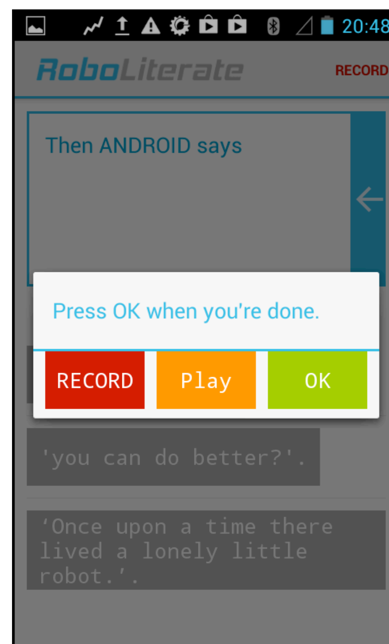


Figure 7.7 Record dialog

Once this feature is discovered, users can record their audio, review it, then name it. Figure 7.7 shows the record dialog popup that appears when users click RECORD. The name they give their audio then appears within quote marks as an option in the phrase list. The intention here is to pool the phrases that everyone records on one device, allowing the different users to utilise the sounds that others make for their own 'stories'/programs.

7.1.4. Story builder activity - populated

Once a sentence has been constructed in the previous screen, the Story Builder reloads with the completed sentence as part of the growing 'story', with each sentence beginning on a new line. Users can re-edit any sentence by clicking on it. This takes them back to the sentence building screen with their chosen sentence in the editing window.

Figure 7.8 and Figure 7.9 show examples of stories that were built in testing. The first programs the robot to 'explore' the room for 30 seconds, moving it forward until it detects a wall, then turning 90 degrees. The second story is more of a conversation between the Android device and the robot.

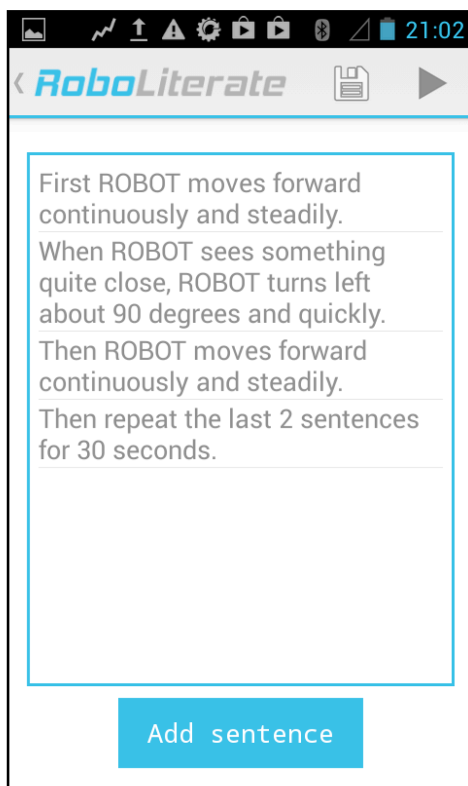


Figure 7.8 Story which gets robot to 'explore' the room

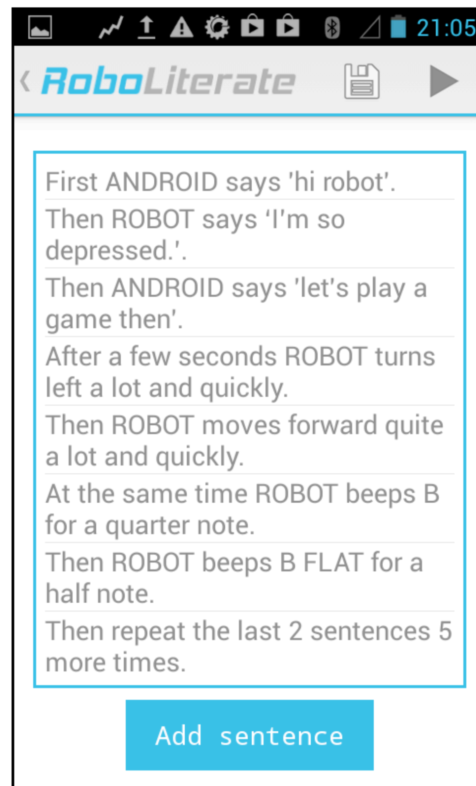


Figure 7.9 Story with a conversation between Android and robot

Once users are happy with their story, they can click the Play button in the ActionBar to start the program execution sequence, or the Save icon to save their story. As shown in Figure 7.10, users are prompted to name their story before saving.

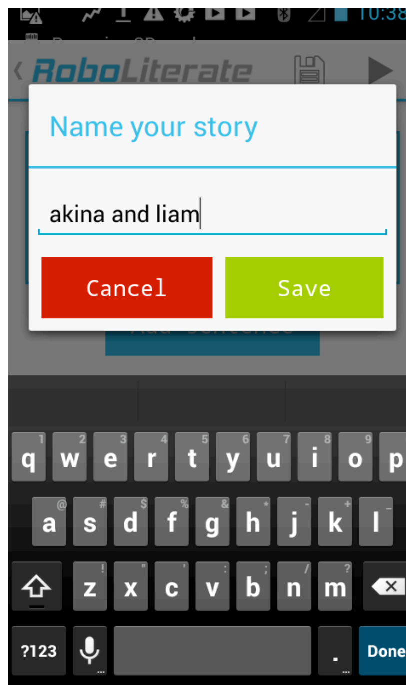


Figure 7.10 Save dialog on Story builder screen

Once a story has been saved once, the story is auto-saved again when the user presses the Play button, or simply re-saved if the Save button is pressed again. If the user presses Play and hasn't yet saved their story, they are prompted to do so.

This functionality was added to ensure that a record was kept of all stories during testing and no data was lost.

7.1.5. Device Connection Activity

After the user presses Play on the Story Builder Activity, then a Bluetooth socket is opened and the application presents a choice of paired devices to connect to, as shown in Figure 7.11. There is also an option to scan for new devices. This closely follows the flow of Google's Bluetooth guide, as exemplified in their Bluetooth Chat application.

Different icons for NXT and EV3 were developed so that if users have access to both types of robot, they can easily distinguish between them.

Once a device is selected, the application attempts to connect . If the device cannot be detected, the user is notified that there was a problem connecting and the device selection dialog opens again. If connection is successful, the user is asked to configure their robot.

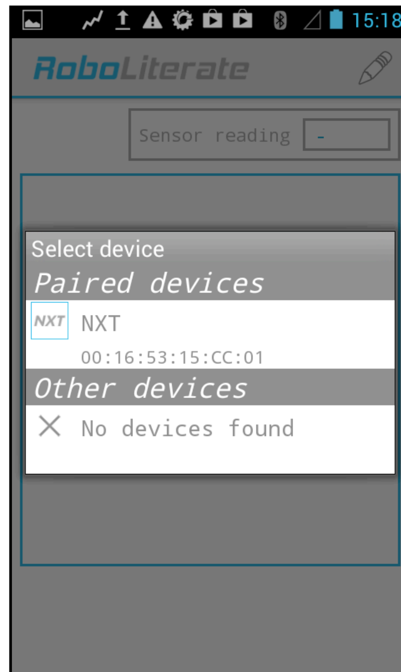


Figure 7.11 Connection dialog

7.1.6. Port Configuration Activity

When a connection is made, the user is presented with a dialog that requires them to check how they have configured their robot's ports, shown in Figure 7.12.

The default ports match the 'Robot Educator' model of the NXT, which is the basic model that ships with the Education Set of the NXT model.

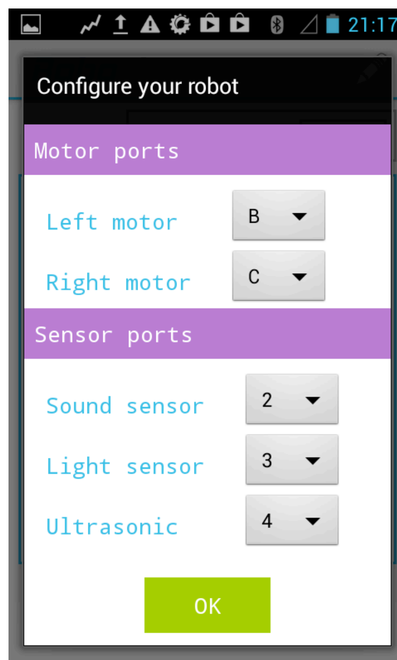


Figure 7.12 Port Configuration activity

In testing, this interface was skipped over after users were asked to ignore it, as in the testing sessions the robot was prebuilt and checking port configuration was not part of the test. After an explanation, all pupils had few problems skipping past this screen, however its presence means that in future versions this application can present more flexibility than others, at least for the NXT model.

7.1.7. Story Execution Activity

Once the ports have been selected, the Story Execution Activity is launched.

Like the Sentence builder and Story builder activities, this Activity has a 'clean' design with few distractions apart from the essential information, as shown in Figure 7.13. There are two windows on the Execution screen, one that shows Sensor readings, when appropriate, and the main window which behaves in a similar way to a Console window in an IDE. Messages appear informing users of the status of the program as it runs, including which lines have been executed, and when the program has begun and ends. It also displays error messages, for instance if the robot has run out of memory.

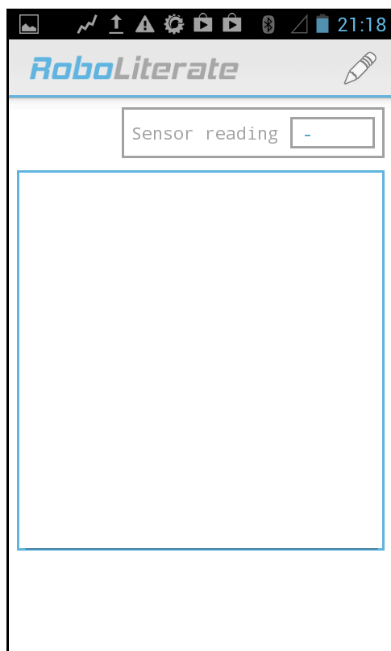


Figure 7.13 Program Execution Activity on launch

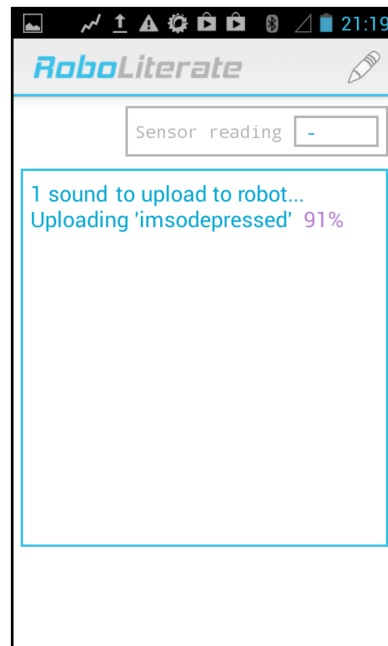


Figure 7.14 Activity showing upload information

The aim here was to provide users with the kind of environment that professional programmers would use, so that they get the end-to-end experience of first writing, then observing execution and debugging their programs.

If the user decided to insert robot-centric audio into their stories, then the application uploads these sounds to the robot. In Figure 7.14, the program is showing the upload status of sounds in the user's story.

As soon as all the sounds are uploaded, the program begins and the sentences appear as they are executed, emulating traces in a programmer's debugging console. The user's sentences appear in green, general information appears in blue (for example, 'Starting program', 'Uploading sound') and errors in red. Figures 7.15 and 7.16 show the console at different stages of program execution.

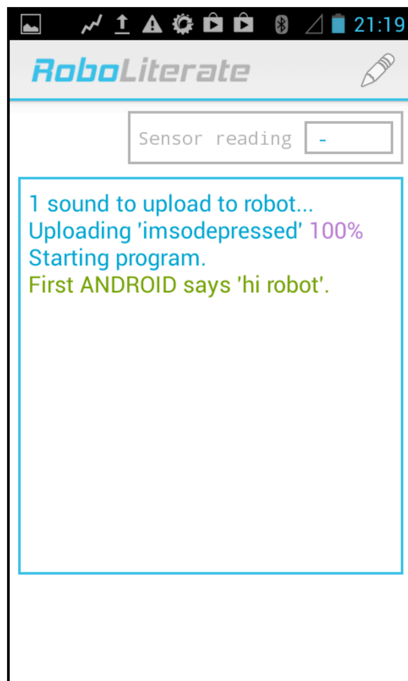


Figure 7.15 Activity showing trace of first line of program execution.

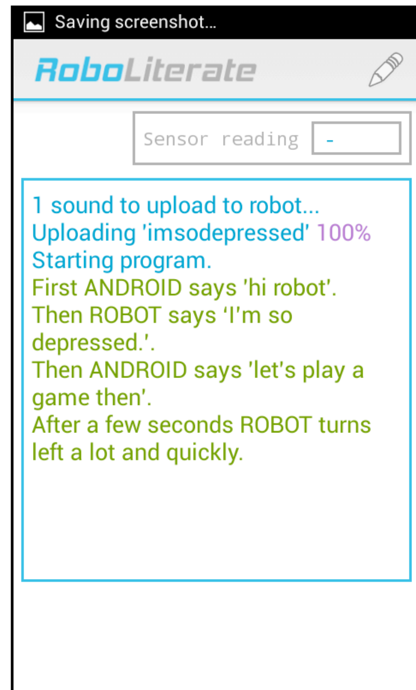


Figure 7.16 Activity after several lines have executed.

When an 'event listener sentence' is being executed, as shown in Figure 7.17, the 'Sensor reading' window wakes up and the live reading from the sensor is shown. When the target is reached, the reading turns orange and the program continues (Figure 7.18).

In future iterations, this precision will allow for more exact RLit sentences and measurements, for example having sentences such as 'When ROBOT sees something up to 30 cm away...' However, as the focus of this project was more on storytelling than on scientific measurement, this was not implemented. As noted before, the latency in bluetooth communication was also an issue, making the reading slightly behind, so this would also need to be addressed somehow.

Fig 7.18 illustrates how the console shows looping statements. The user has programmed the robot to 'explore' the room in three sentences. First, the robot moves forward continuously, and turn left when it detects an obstacle close by. This is then repeated 4 times. In the console window, the sentences that are repeated are rewritten, as they would in a programmer's IDE console window.

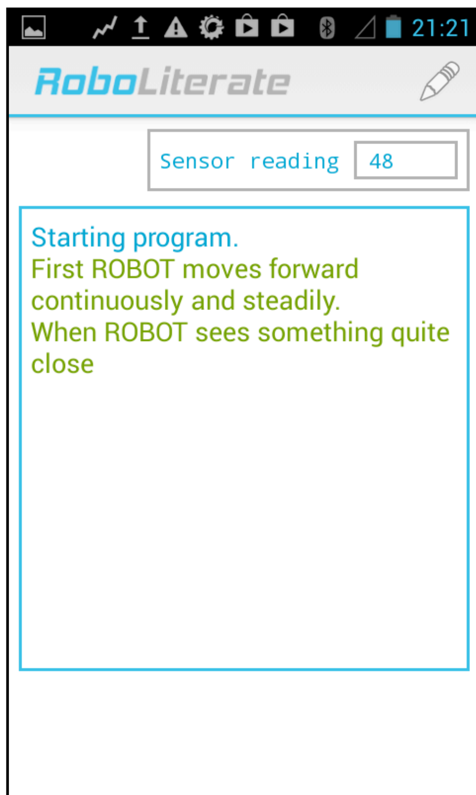


Figure 7.17 Activity showing trace of event listening sentence, and sensor reading.

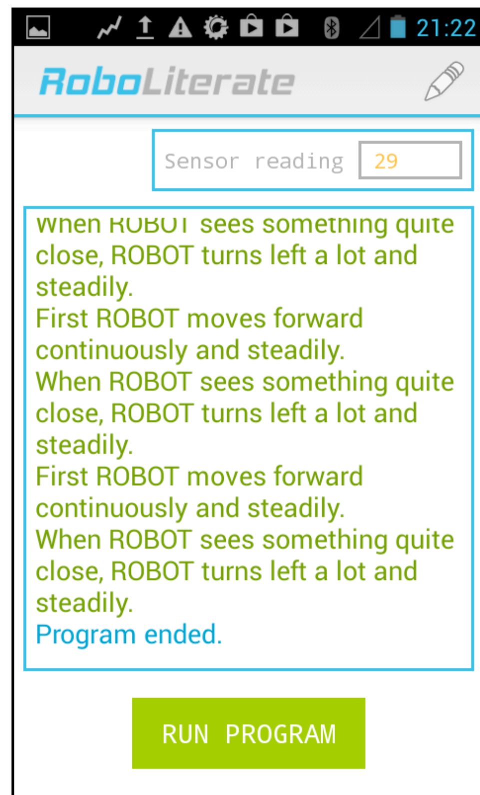


Figure 7.18 Console readout showing that a target has been reached, and the program continues.

At the end of program execution, the user is informed and a button appears at the bottom of the screen allowing users to re-run the program.

Users can end the program and return at any point to edit their RLit Sentences by pressing the Pencil icon in the ActionBar.

7.2. UI Layer technical implementation

The UI layer follows the design principles of Google's Android platform, with all layouts defined in XML files in the application's resources folder.

There are many benefits to following this architecture. The design layout of screens is separated from the Controller layer, meaning that updates can easily be applied without affecting the code. As Android is meant to run on a wide variety of devices, separating the UI files out into a separate layer means that one can have different layouts for different screen sizes and Android OS versions.

Although I was not so concerned with providing layouts for different devices in this project, or supporting Android versions other than JellyBean (as directed by my tutor), with this design it will be possible in the future.

The same follows for icons and bespoke graphics, or 'drawables'. Although this application has minimal original design elements apart from its logo, these are stored in the resources folder in separate folders according to screen resolution (hdpi, mdpi, xhdpi etc.)

All sound files are stored in the ‘raw’ resource directory, along with the dictionary text file for the RLit language. During development, having the RLit framework saved in one text file facilitated rapid iterations of improvements to RLit, including the ability to update labels and values on the fly without affecting the rest of the application.

String values are also all the UI layer, allowing for localisation at a later date, without disturbing any of the controller code.

7.3. UI Controller layer

Adhering closely to the recommended Android architecture for JellyBean (Google, 2013a), the controller classes were built making use of Android’s ActionBar class (the menu bar at the top of the screen) and Fragment classes. Fragments are classes that can be used in combination to create different configurations of design layout depending on the device’s screen size and orientation (Google, 2013c).

This project was built using a rooted Samsung Galaxy S, and later tested on a Nexus 7, and there is currently only one set of UI configurations used. The application forces portrait mode, and doesn’t change based on screen size. However, by using Fragments, the application is built flexibly enough to allow different UI configurations to be implemented relatively easily.

Figure 7.19 shows a class diagram for the UI controller layer.

The overall architecture is based on four Android Activity classes which users move between. All sessions start with the LaunchActivity, from where the StoryBuilder activity launches with either a new story or a saved story, loaded via the DatabaseHelper data access object located in the storage layer.

Fragments are linked to their parent Activities through interfaces declared inside each fragment. These interfaces define the contract which parent activities must implement to communicate with their child fragments. This architecture follows the Observer pattern, and ensures that Fragments behave as modularised components. This will become especially useful once the application is developed for different screens and devices, and composite UI configurations are used.

The ExecuteStory Activity communicates with the robot through binding to an Android Service. This is described further in Section 7.6

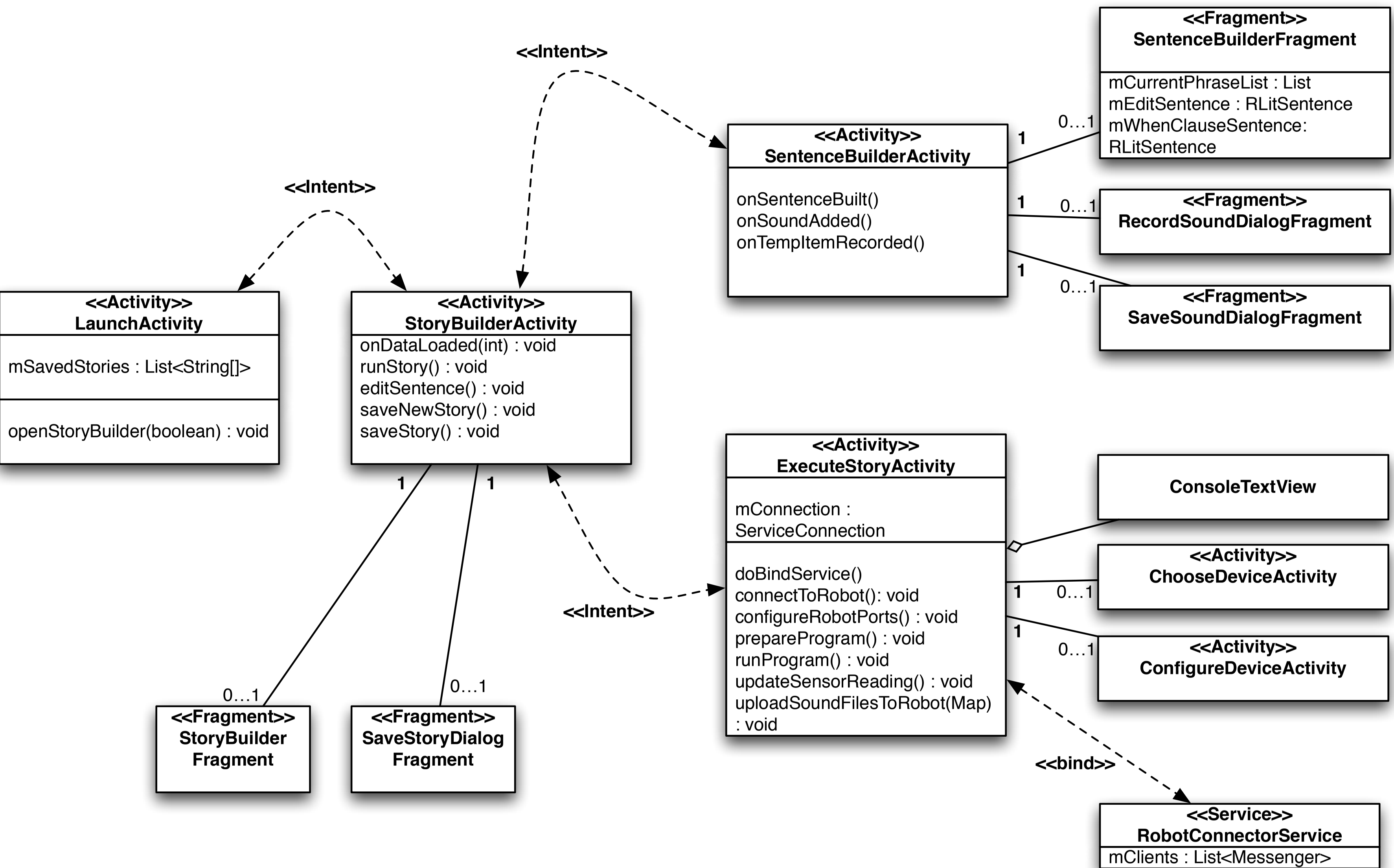


Figure 7.19 Class diagram for UI Controller layer

7.4. RLit Model layer

The RLit entity classes - RLitStory, RLitSentence and RLitPhrase - are contained in the RLit package. Also in the RLit package are the utility classes RLitDictionary, which holds all the available RLitPhrases, and RLitDictionaryLoader, which populates the RLitDictionary using data from the resources directories.

Fig 7.20 shows the relationship between these entity and utility classes.

7.4.1. RLitStory Entity class

At any point, there can only be one instance of RLitStory, namely the story that is currently being edited or executed. This is therefore a static instance and accessed through the RLitStory.getStory() method. RLitStory is instantiated the first time getStory() is called.

An RLitStory is a basic class composed of a specialist ArrayList of RLitSentences, and a cursor position which keeps track of which sentence is currently being added or edited. If the user chooses to load a saved story, then the sentence list is populated by the RLitSentences stored in the SQLite database, otherwise a fresh sentence list is instantiated. In both cases, the list is instantiated as a SentenceArrayList.

An inner class, SentenceArrayList ensures that when any change is made to the list, this is flagged to the parent RLitStory class. This will allow the UI to know whether an RLitStory has been updated and whether it needs to be saved.

Once a story has been started, or an old story is loaded, the RLitStory singleton is prepared, and LaunchActivity can then launch the StoryBuilder Activity. This then prompts the application to populate the RLitDictionary with phrases.

7.4.2. RLitSentence Entity class

Following the ontological rules of RLit, there are three types of sentence:

DIRECTION SENTENCES – Sentences that are translated into simple instructions. These instructions can appear anywhere in a Story e.g. '*First ROBOT moves forward a little and slowly*'

EVENT LISTENER SENTENCES – these 'sentences are translated into instructions that monitor readings from the robot sensors, and pause the flow of the Story until stipulated conditions are met (either a target reading is met or a fixed time period elapses). e.g. *'After that ROBOT waits until it hears something loud.'*

EVENT RESULT SENTENCES – these sentences occur after *EVENT LISTENER* sentences.

The sentence type is set when a 'verb'-like phrase is added through the method addPhase(RLitPhrase). See Appendix A for more details on this.

There are two varieties of EVENT LISTENER sentences - those that contain the phrase 'When', and those that contain the phrase 'wait'. Each of these varieties has an effect on how they are displayed by the Story Builder and Sentence Builder activities. 'Wait' sentences are displayed as separate sentences, whereas 'When' sentences are always displayed connected with an EVENT RESULT sentence, which begin with a comma.

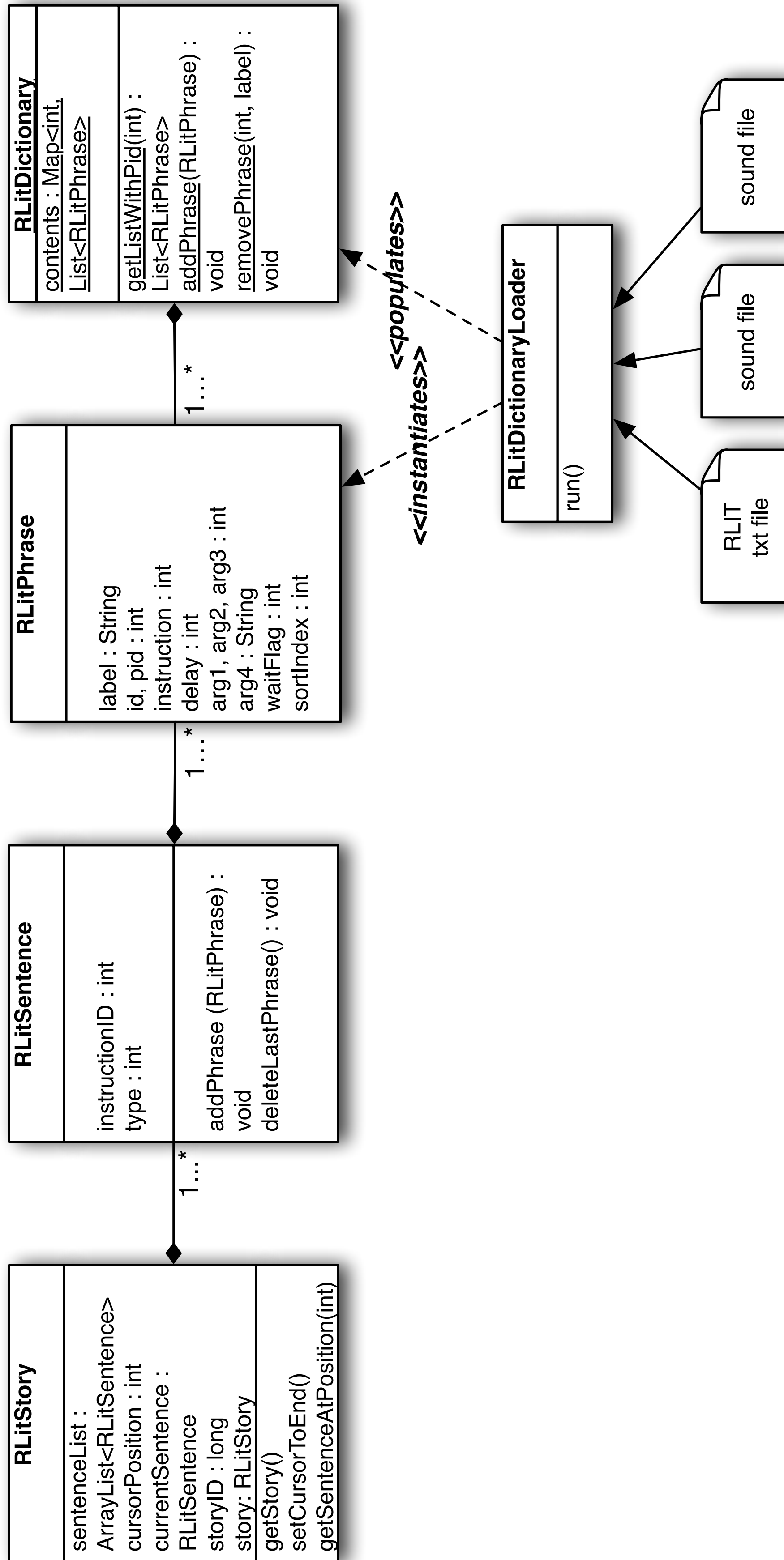


Figure 7.20 Class diagram showing relationship between RLit Entities, and Dictionary and Loader utility classes

7.4.3. RLitPhrase Entity class

Each RLit Phrase has the following fields:

Label	the human readable label for the phrase.
ID	The ID group that the phrase belongs to.
PID	The ID of this phrase's parent phrase i.e. the phrases that will always come before this one in a sentence.
InstructionID	for 'verb' phrases, the ID of the Instruction that maps to this verb.
Delay	the default waiting time before the sentence in which this phrase occurs will execute. This is relevant for 'sequencer' phrases such as 'After 3 seconds'.
Arg1, Arg2, Arg3	integer arguments that modify an Instruction's parameters, for example angle of turn, or speed of movement.
Arg4	String argument that modifies an Instruction's parameters, for example filename for a Sound Instruction.
WaitFlag	indicates whether the program looper needs to complete executing the previous sentence before executing the sentence with this phrase. For example, the Sequencer phrase 'After that' has a wait flag of 1, whilst 'At the same time' has a wait flag of 0.
SortIndex	used by compareTo method to sort phrases in lists. This is used to ensure that the more common and useful phrases always feature near the top of the list.

Appendix C contains a table with all the phrases and their argument values featured in RLit. It is the RLitDictionaryLoader's role to populate the RLitDictionary with all these phrases before the user can begin assembling sentences.

7.4.4. RLit Dictionary Loader class

The RLitDictionary is populated on the first launch of the StoryBuilder Activity. The contents of the dictionary are stored in a HashMap and as there is only one dictionary, the map is static and available to all Activity and Fragment classes, in the same way as the RLitStory static instance.

There are two stages to the process of populating the dictionary with RLit phrases. First, the general phrases are assembled from a CSV text file, 'rlit_dictionary.txt' in the raw resources directory. Each line of the CSV document maps to a single phrase and its fields.

Once the dictionary loader has completed processing the general phrases, it adds user-specific phrases based on the sound files found in the user's Shared Preferences.

When the loader has finished populating the RLitDictionary, it notifies listeners who have registered via a LoadDataListener interface - in this case the StoryBuilder Activity - that it has done so.

The sequence diagram in Figure 7.21 summarises this process.

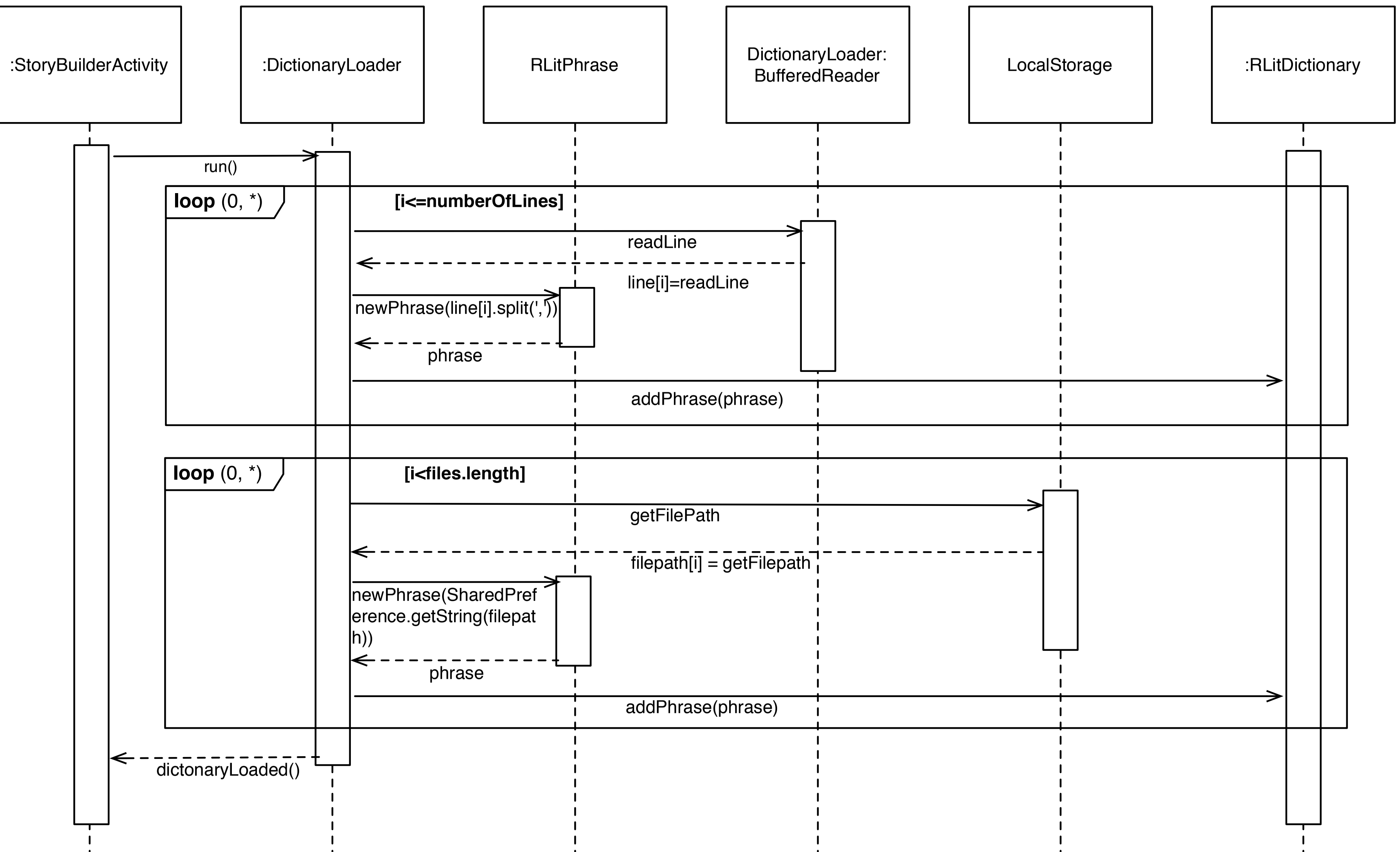


Figure 7.21 Sequence diagram showing how the RLit Dictionary is populated with RLit phrases from an external file

7.5. Interaction of UI Controller layer and RLit model layer

The sequence diagram in Figure 7.22 shows how the UI layer works with the RLit model layer to build RLit sentences.

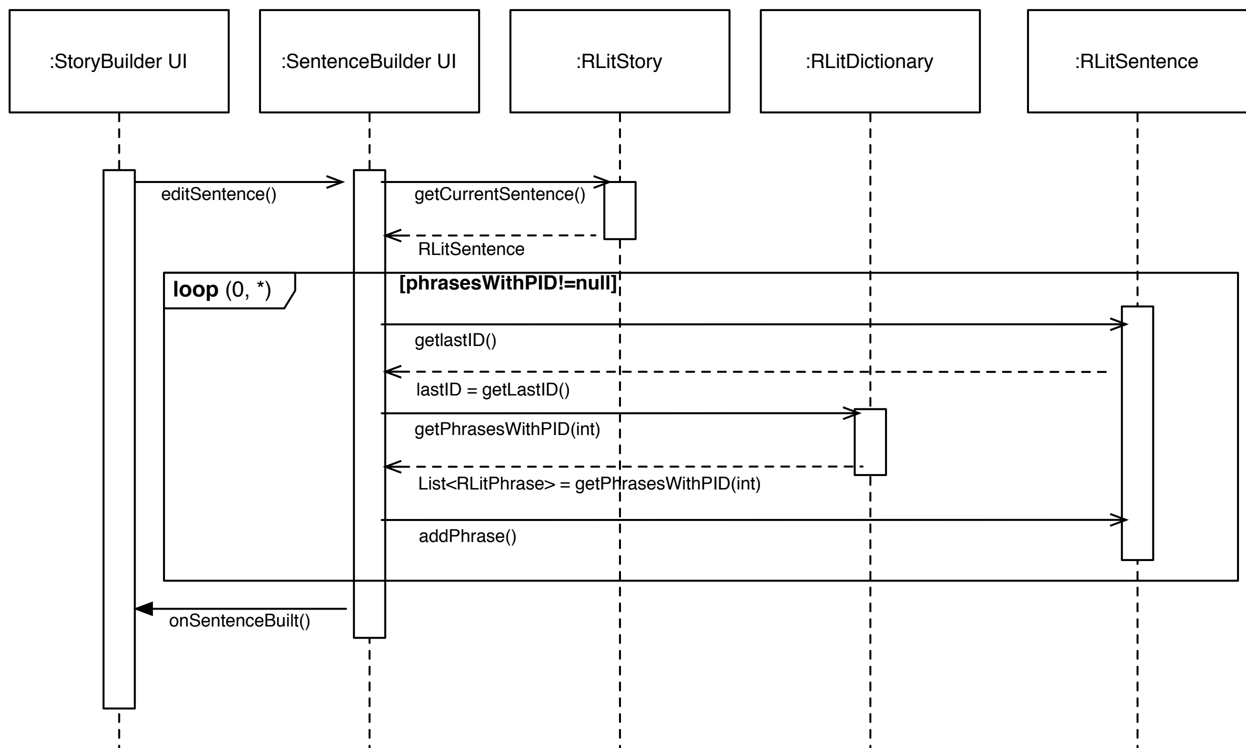


Figure 7.22 Sequence diagram showing the flow of information between UI and RLit model layers during sentence construction.

Before SentenceBuilderActivity is launched, RLitStory's cursor is set to the position of the sentence being built. SentenceBuilder can then get the contents of the current RLitSentence to populate the interface. In the case of adding a new sentence, this will be an empty sentence.

SentenceBuilder Activity has encoded the ontological rules that govern which phrases can follow each other in an RLit Sentence, mainly based on the RLitPhrase PID field. The arrows that connect the phrases in the RLitPhrase map (Appendix B) show the PID relationships between RLitPhrases.

A list of phrases is obtained by polling the RLitSentence for the ID of the last phrase in the sentence. If the sentence is empty, then the phrase with PID -1 ('First') is used. Otherwise, as long as special rules relating to RLitSentence type and program flow are not in effect, the phrases in the PID group that match the last phrases's ID are displayed.This process continues until all phrases are exhausted (i.e. when the *getPhrasesWithPID(x)* call to RLitDictionary returns null). If the user then accepts the sentence, SentenceBuilder activity is closed and a return call (*onSentenceBuilt())* is sent to the StoryBuilder activity.

Once all RLitSentences have been completed. the RLitStory can be interpreted into an ArrayList of Instruction objects that is later sent in a 'Program' wrapper class to the Robot Communicator Service, ready for execution by the Robot Commander.

7.6. RLit Interpreter and Instruction class package

The RLitInterpreter implementation's role is to convert the list of sentences contained in a RLitStory into a 'Program' - a wrapped list of Instructions that can be executed by the ProgramLooper class. To do this, the Interpreter iterates through the RLitSentences, and for each sentence, analyses its contents (i.e. all the RLitPhrases) to first extract an Instruction ID. The Instruction ID defines which Instruction to instantiate. The remaining phrases contain the information needed to populate the parameters for that Instruction's constructor.

Most of the Instructions are POJOs (Plain Old Java Objects) with fields specific to the Instruction they are representing. For example, the Move Instruction has fields related to throttle and distance, while the Beep Instruction has fields relating to tone frequency and duration. All Instructions share a set of fields which are defined in the *AbstractInstruction* class, which they all inherit. This is to ensure there is no unnecessary repetition of code. One specialist Instruction, the *Repeat* Instruction, also implements the *ProgramControlInstruction* interface, since it is uniquely concerned with program flow. As all method calls are defined in Interfaces, the Instructions are loosely coupled with the implementation classes in the Robot control package.

To execute an Instruction, the robot commander class must implement the *InstructionExecutor* interface. Only objects of type *InstructionExecutor* can access an Instruction's execution method. In this way, Instruction classes implement the Visitor pattern to safeguard against wrong calls being made, and to enable any number of different implementations can access the Instructions, including eventually EV3 implementations.

There remains some work to be done with the AndroidNarrative and AndroidMusic Instructions since there is some repetition of code shared between the two classes relating to sound playback. In addition, these are the only Instructions that contain implementation code, so it would probably be a better design to keep all Instructions as POJOs and move the implementation code elsewhere. However, there was not time to complete this for this project.

Figure 7.23 shows a class diagram with the connections between these classes.

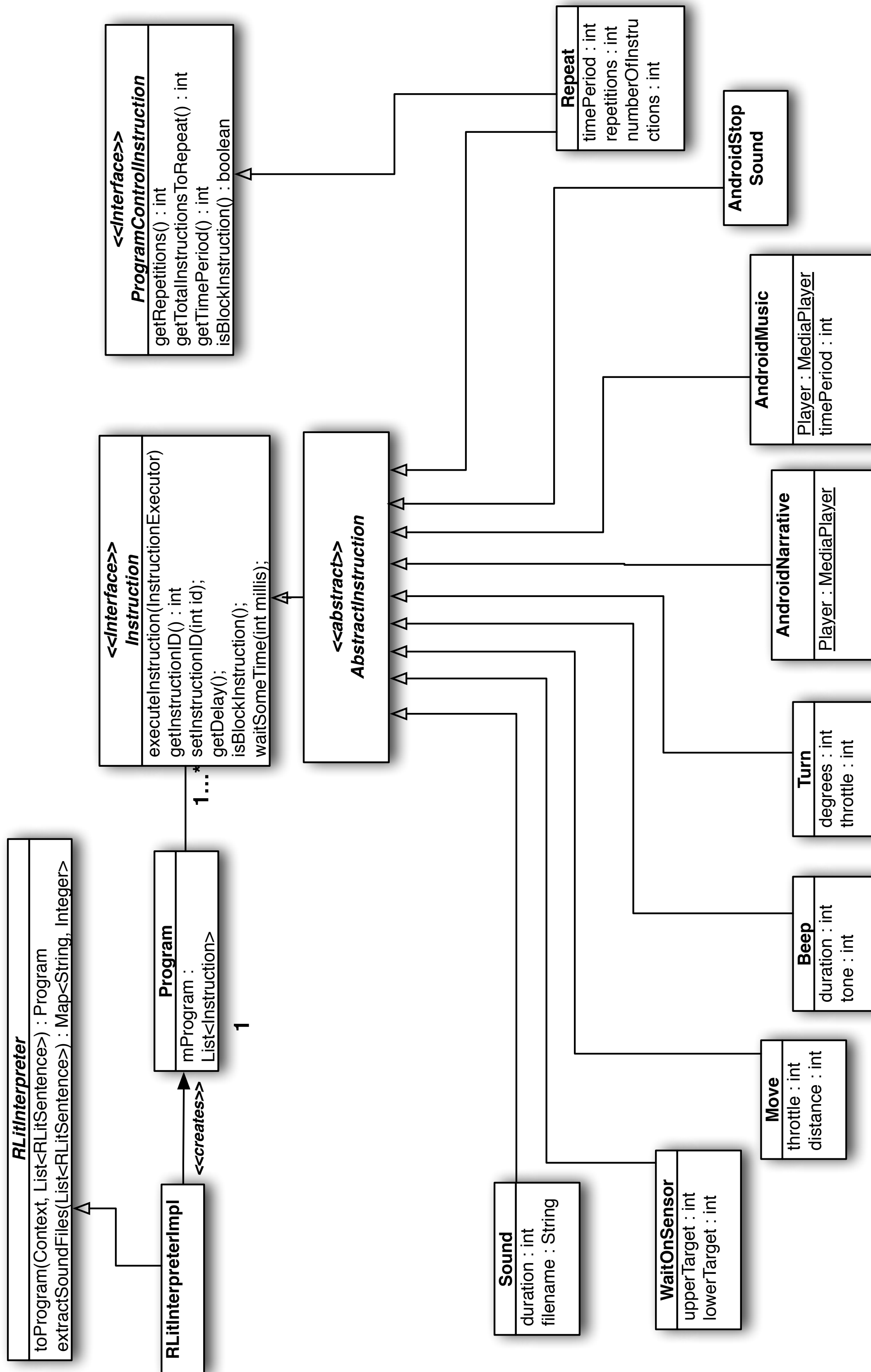


Figure 7.23 Class diagram showing Interpreter and Instruction classes.

7.7. Storage layer

Story data is stored as an SQLite database in the Android device's local storage. Using SQLite allows every RLitStory to have an unique auto-incremented ID which serves as its primary identifier. This is important since several saved stories might possess the same title, or sentence list. Other storage mechanisms, like using Shared Preferences, will not allow this kind of repeating data. Figure 7.24 shows how the components of the storage layer work together.

The schema for the RoboLiterate database is formalised by the RLitDbStoryContract class. The class contains constants that define the names for the tables, columns and URIs, and ensures that these names are consistently used throughout the application. The database for the application required just one table and, as recommended by Android developer documentation (Google, 2013e), the table and column names are defined in an inner class called RLitStoryTable.

The DatabaseHelper class was created as a data access object to provide all the methods for creating, reading, updating and deleting table entries. To save lists of RLitSentences, they first needed to be converted into a format that SQLite recognises, and Google's GSON library was chosen to do this. GSON is a Java library that converts objects into JSON (JavaScript Object Notation) strings, and it was chosen because it does not require annotations and it supports Java Generics (Google, 2013d).

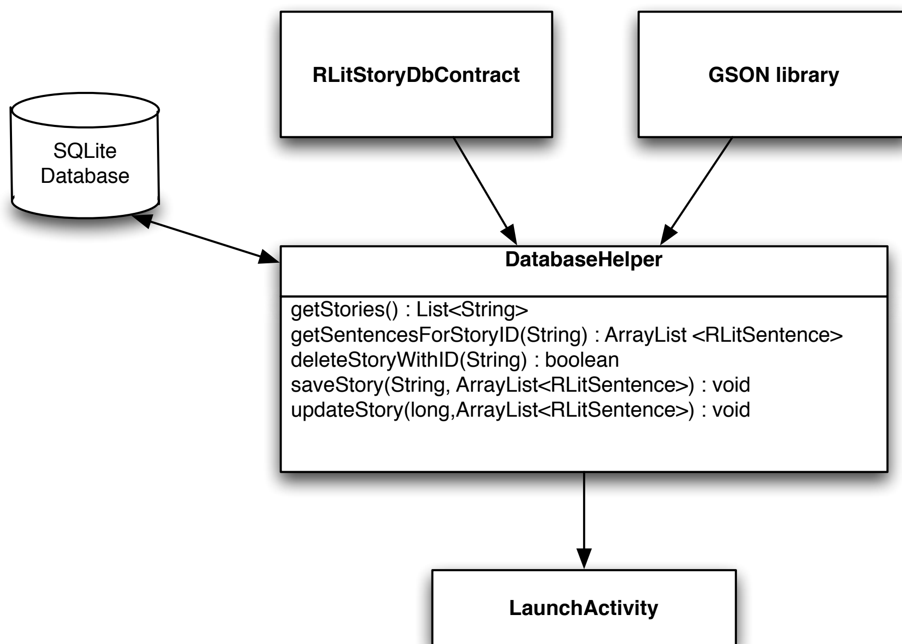


Figure 7.24 Components of Storage layer

7.8. Bluetooth connection implementation

The method of Bluetooth connection to the robot follows Google's sample code exemplified in its open source Bluetooth Chat application. The steps taken are as follows:

In *DeviceChooseActivity.java:*

- The class obtains a Bluetooth Adapter

- The class checks that bluetooth is available on the device, if it is not already. This requires sending the intent *BluetoothAdapter.ACTION_REQUEST_ENABLE*

- The system is queried for paired devices

- If the user presses 'Scan for robot', then the system discovers new devices, and registers a BroadcastReceiver for the ACTION_FOUND intent to receive information about each new device.

- The user selects a device from either the paired or new device list.

The selected device's name and MAC address are then passed by the UI layer in a message to the RobotConnectorService object, to which it is bound. From this point, all communication with the robot is managed by the service layer.

- The Service acts as a server for the Lego device and therefore first obtains a BluetoothServerSocket by calling *listenUsingRfcommWithServiceRecord (UUID)* using Lego specific UUID, which was obtained from Lego's MindDROID application. Then the application starts listening to the device by calling *accept()*.

- Finally, the input and output stream of the Bluetooth connection are obtained, and passed to the *BTCommunicator* class, which provides static methods to read and write data to the robot, encapsulated in one place.

7.9. Robot Communication Layer

The architecture for the robot communication layer was finalised during the prototype development phase, when most of the work went into researching and testing how to control the NXT from Android.

The overall architecture of this layer is shown in Figure 7.25.

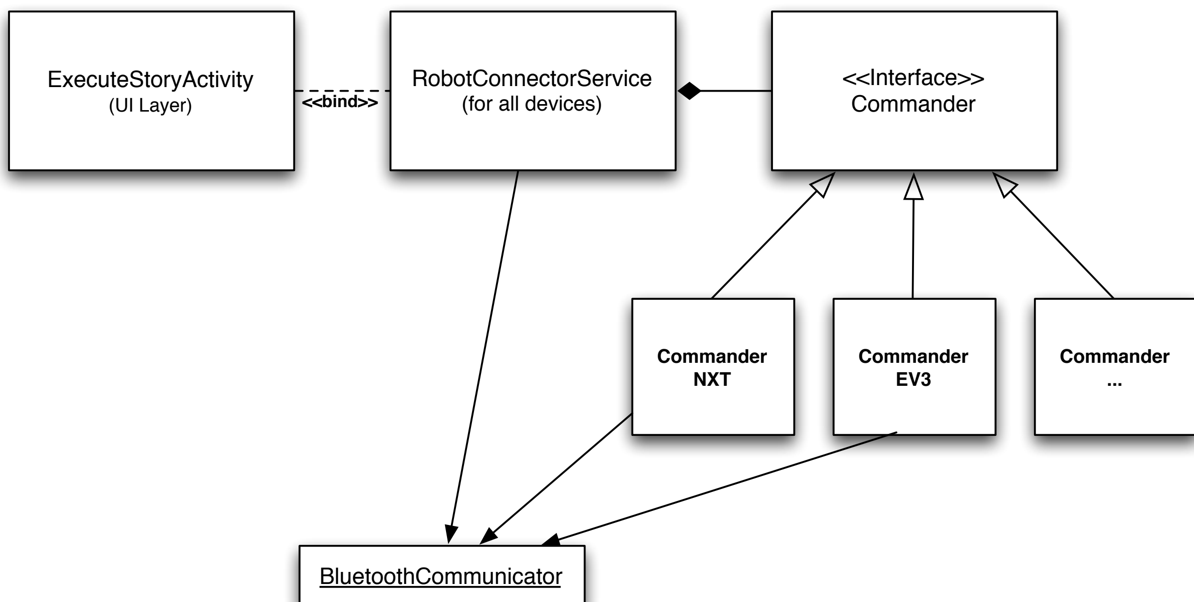


Figure 7.25 Class diagram showing general relationship between robot control classes

The UI layer is responsible for finding the name and MAC address of the robot device. The RobotConnector class then connects with the robot. The RobotConnector class also checks for the version of the robot, NXT or EV3. Depending on the version, it will then instantiate the relevant Commander class. Currently there is only one Commander implementation, CommanderNXT. All classes communicate with the robot via static methods in the BluetoothCommunicator class. A summary of this relationship is shown in Figure 7.25. A more detailed class diagram of the Robot Communication layer is shown in Figure 7.26.

7.9.1. Robot Connector Service class

The RobotConnectorService implementation contains methods that manage the robot connection, and through implementing the *CommanderListener* interface, it also receives update messages from the Commander object. These updates are then passes back to the UI layer as a Messenger object.

Two options were tested for communicating with the robot - opening a new thread, or running a separate Service layer.

Choice 1: Opening a new thread:

The 'simplest' method would have been to simply run all the robot communication code on a new thread running parallel to the main UI thread. This strategy is followed by applications such as MindDROID and NXT Remote Control (Fedor, 2011; Lego, 2012). There are a number of advantages - it is the simplest approach, in that no Messenger object needs to be created, making handling messages between threads easier to implement. It would also ensure that the thread stays open for only as long as needed, ensuring processing load would be kept to a minimum. The downside would be that communication could only take place from within the same application, and without careful planning, from the same Activity. This would make the application less extensible if one wanted to extend it to, for example, controlling the robot from another application. It would also require very careful thread management if multiple threads were needed (as is the case with this application).

Choice 2: Starting a service / Bound service

Alternatively, all robot communication could take place from a Service running in parallel to the UI thread. The advantage here is the Service could always be running no matter what Activity is on screen, until the Service is actively stopped. The disadvantage is that communication is restricted to send messages via a Messenger object and Handlers on either side.

The second approach was chosen, to maintain a clear separation of concerns between the UI layer and the robot communication layer and to keep open the possibility in future versions for multiple robot devices to be run from one Android device, or indeed have multiple Android devices working with one robot. Having a bound service would allow multiple activities and devices to bind and unbind from the service without affecting the Bluetooth connection itself.

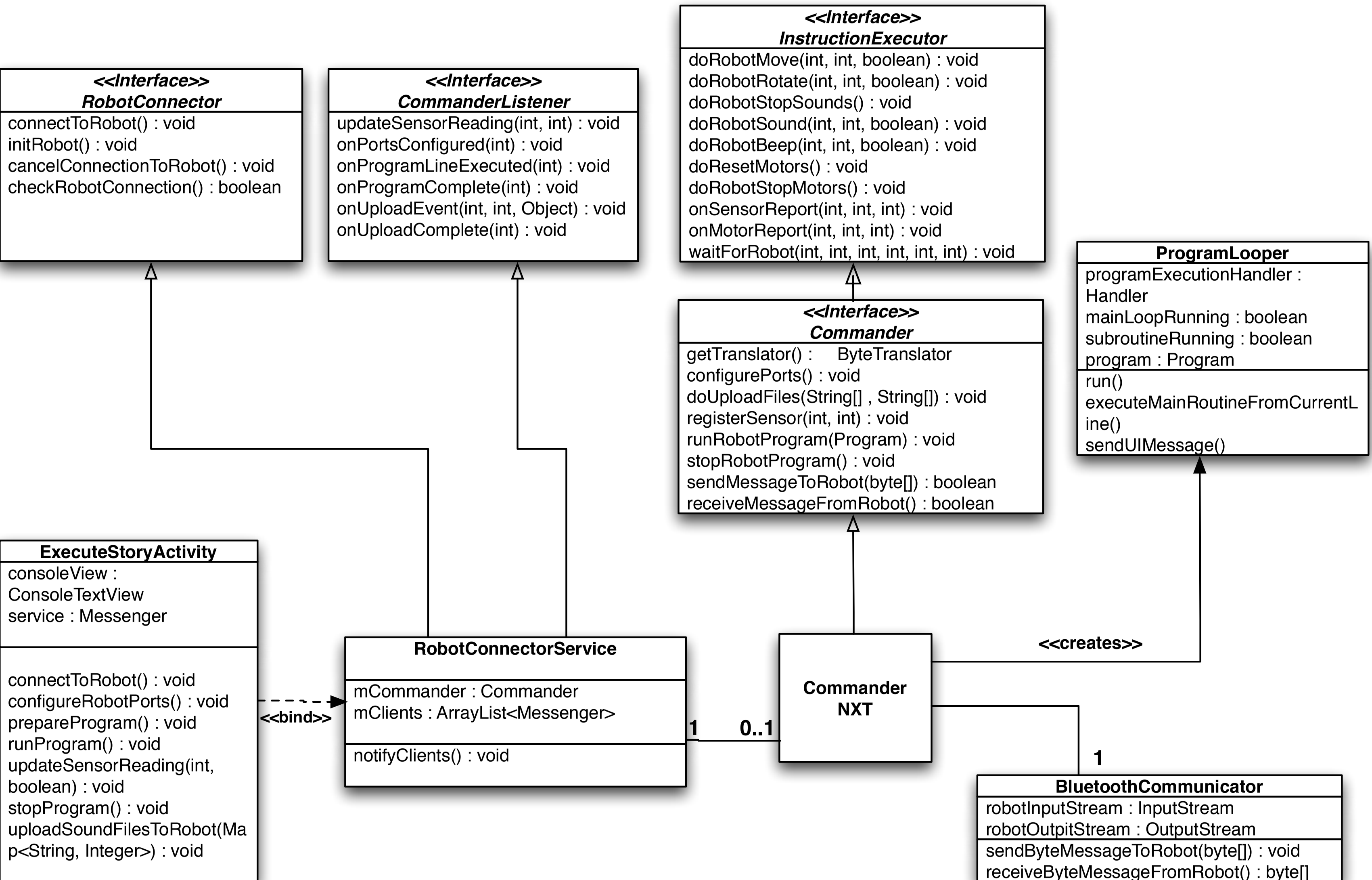


Figure 7.26 Detailed class diagram showing overall architecture of robot control classes

7.9.2. Robot Commander classes

The Commander classes control the messages sent to and received from the robot during the lifetime of the application. A Strategy pattern was used to ensure that the architecture would work with the new version of Mindstorms, EV3, although the current implementation only works with NXT.

The Commander class implements the *InstructionExecutor* interface defined in the RILt Instruction package, enabling it to be the 'Visitor' class that runs the 'executeInstruction' method within each Instruction. The timing and sequence of execution of these instructions is controlled by a ProgramLooper class, which the Commander class instantiates. Figure 7,26 details these relationships.

The Commander implementation class also instantiates a set of four utility classes specific to the device it are controlling:

ByteTranslator classes - these classes translate commands into byte code understood by the robot device. The translator includes methods for both DIRECT and SYSTEM calls , and those that require a REPLY and NO REPLY. The ByteTranslator class used in this project is based on a similar class used in the Lego's MindDROID application, however in that application, a very small fraction of the methods are used, whereas in RoboLiterate, the capabilities of LCP Bluetooth communication are pushed much further.

Methods were also added that interpret the contents of byte messages returned by the robot, for example sensor readings and file upload status. Thus all interpretation of byte code is encapsulated within ByteTranslator classes, hidden from the rest of the application, providing a better separation of concerns for developing for the EV3 later on.

SoundUploader classes - these classes specialise in the complex task of uploading files to robot. Again, the class used in this project is based on the file uploader class developed for the MindDROID project, where it is used to upload programs to the NXT. In this project, a similar method is used to upload .rso sound files (.rso is the unique extension used by the NXT device).

RobotListener classes - these utility classes specialise in listening to sensor readings sent by the robot device via Bluetooth until a target reading has been met. In these classes, the sensors are polled in a separate thread for readings in a *while* loop, and the loop ends once a target reading is met.

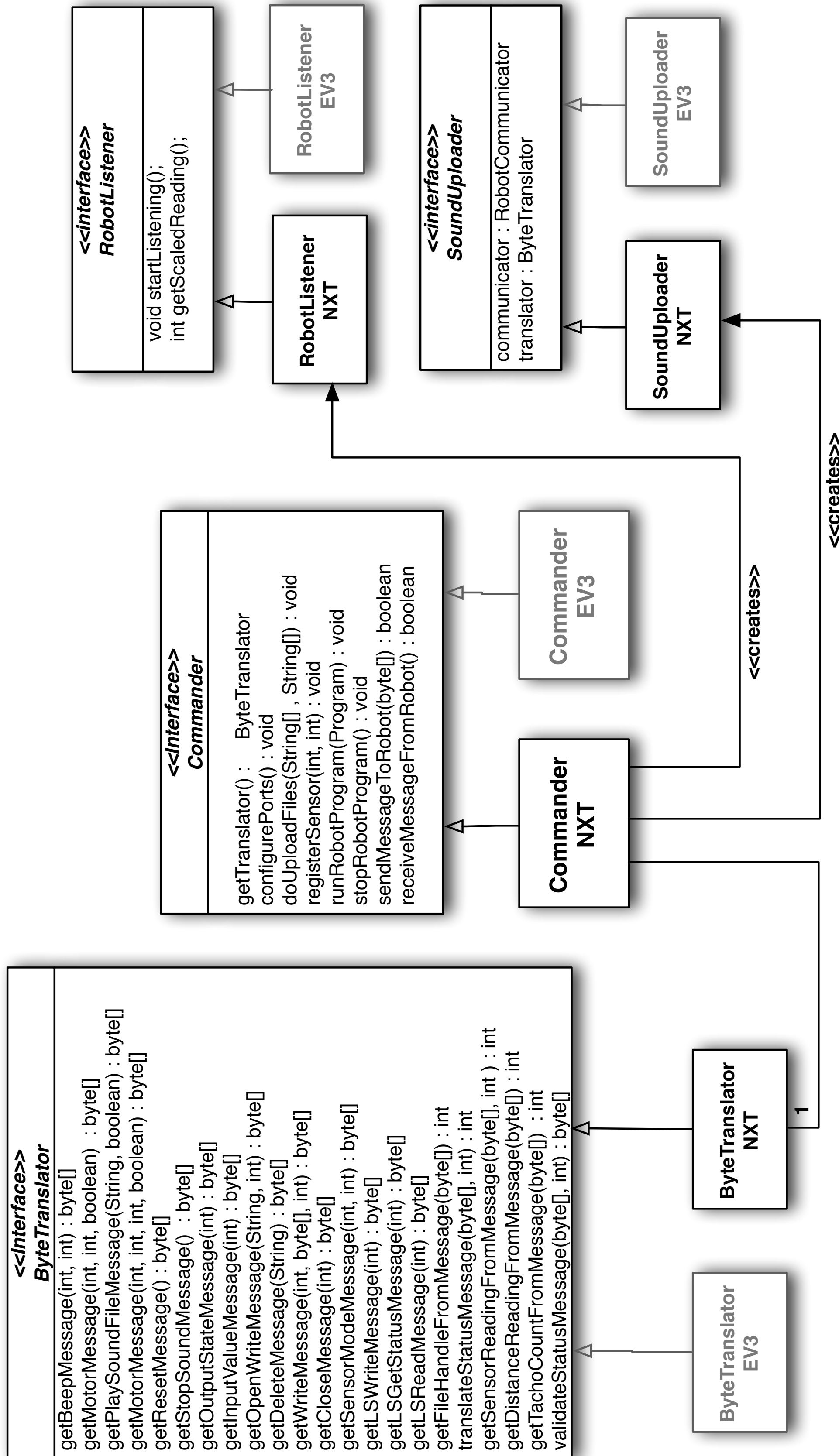


Figure 7.27 Class diagram showing the Strategy Pattern for Commander and related classes

7.9.3. Program Looper class

The ProgramLooper class governs the sequence and timing of execution of program instructions. It runs in a separate thread and is instantiated by the relevant Commander implementation class.

Once started, the thread will cycle through the list of Instructions within the Program object passed to it by the Commander. Each instruction is first examined to see if it has a delay, and then whether it is an instance of a *ProgramControlInstruction*, in which case it is a looping instruction.

If the former, then the Looper will execute the Instruction on the main program thread. This is achieved by passing the Commander object as a parameter to the Instruction's *executeInstruction()* method. If it is a looping instruction, then a new 'subroutine' is constructed, looping the required number of Instructions the required number of times. Then a new 'subroutine' thread is started and the main program loop is paused until the subroutine thread sends a message to the Program Looper Handler inner class notifying that it has completed.

If the subroutine is meant to run concurrently with the next instruction, then the main loop is not paused but continues.

Figure 7.28 contains an Activity diagram illustrating this behaviour.

Once the program is complete, the UIController classes are notified, and the user has the option to repeat the program loop or return to edit the story again.

That concludes a brief explanation of the implementation. Fuller details are included as comments in the code of each class.

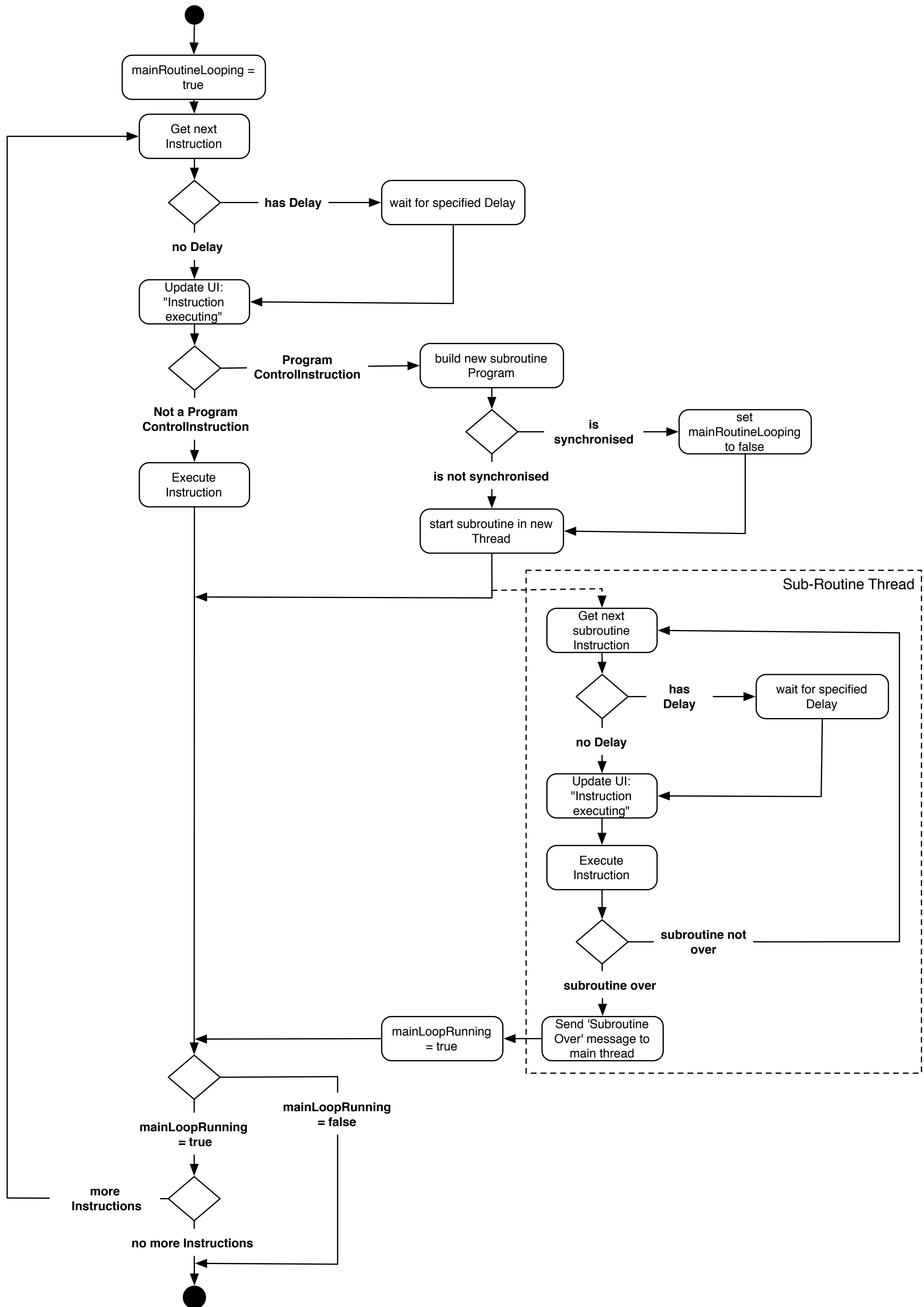


Figure 7.28 Activity diagram showing the flow of the ProgramLooper class

8. Chapter 8. Software testing

The software was developed and tested iteratively throughout its development, starting at the prototype stage, and progressing through test case scenarios to integration testing, system testing and finally user evaluation. The following sections describe in more detail each of these stages.

The Android Studio IDE was used throughout development, which remains in beta and does not yet have integrated support for JUnit testing. To overcome this, the early stages of the project were planned so that functionality could be added and then tested incrementally, with clear test criteria at each step. Each increment included a set of tests, followed by a refactoring and repeat test.

8.1. Prototype iterative test driven development

At the prototyping stage, features were added incrementally and tested in rapid iterations, as described below. In each cycle, the code was first developed and tested. If successful, the code was then refactored and encapsulated as a stand alone 'Instruction' before being tested again.

8.1.1. Test cycle 1: Make a Beep

The first tests involved successfully connecting to the robot and making it beep.

Test	Success criteria
To connect successfully with the robot.	bluetooth symbol on robot shows a connection
To send a message commanding robot to beep	Beep emitted by robot

This was followed by a refactoring to create a 'Beep' instruction, with configurable parameters for delay, frequency and duration. This allowed further testing with different parameter settings:

Test	Success criteria
Beep.getInstance(delay, frequency, duration)	Beep emitted by robot

8.1.2. Test cycle 2: move Robot in a Square

Subsequent tests followed the same pattern as Test cycle 1, but with more complex instructions. In test 2, the objective was to move the robot in a square:

Test	Success criteria
Send a message commanding robot to move forward a set amount	Robot moves set amount
Receive robot messages with 'tacho' count from motor port B until count returns to 0	Trace of motor sensor
To send a message commanding robot to turn 90 degrees	Robot moves 90 degrees

Then refactoring and repeating test:

Test	Success criteria
Move.getInstance(RobotInstruction.NO_DELAY,10,50,true)	Robot moves set amount
Turn.getInstance(90,50,true)	Robot turns 90 degrees
Move.getInstance(RobotInstruction.NO_DELAY,10,50,true) Turn.getInstance(90,50,true) Move.getInstance(RobotInstruction.NO_DELAY,10,50,true) Turn.getInstance(90,50,true) Move.getInstance(RobotInstruction.NO_DELAY,10,50,true) Turn.getInstance(90,50,true) Move.getInstance(RobotInstruction.NO_DELAY,10,50,true) Turn.getInstance(90,50,true)	Robot moves in a square

The same process was repeated for all prototype actions up to and including the most complex envisioned for the application - 'Make robot fearful'. As a result, a complete set of Instructions and robot control methods were developed, and tested, providing the backbone for the robot control layer

8.2. RLit development and testing

At this stage, the RLit ontology and model layer were developed. This was initially done on paper with the objective being to provide a language that could accomplish all of the complex sequences of instructions developed in the prototype. The chart in Appendix B and table in Appendix C shows the result of this stage, with the test being that every instruction in the prototype, including Instruction class and all constructor parameters, could be constructed on paper using combinations of RLit phrases.

The RLit ontology was then exported from the format in Appendix C into a text file, and the next phase was to create and test an RLitDictionary and RLitDictionaryLoader class that could populate a Map of RLitPhrases with the data set.

Two Android ListActivity UI classes were then developed that enabled construction of sentences from the data set - StoryBuilder and SentenceBuilder. The test cases at this stage were to construct sentences that matched every possible combination represented at the paper stage.

8.3. Integration testing

An RLitInterpreter class was then developed that converted the sentences into Instructions. This permitted the first integration tests for the RLit model component and robot control component developed for the prototype.

The test was that all the instantiation calls used at the Prototype phase could be dynamically assembled from the RLitModel and two related Activity classes, SentenceBuilder and StoryBuilder.

For the integration test, each test case in Section 8.1 was repeated, using this Interface, as exemplified in this table:

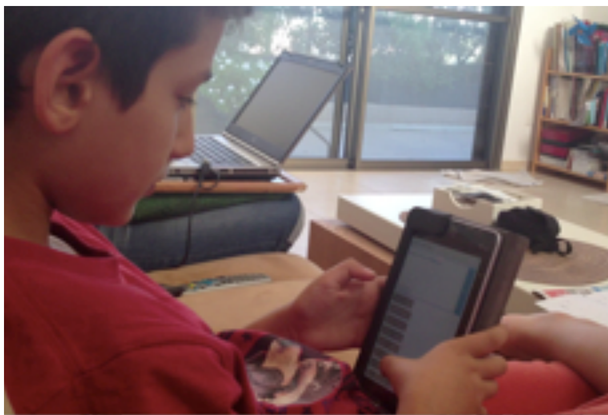
UI input	Success criteria
First ROBOT moves forward a little and steadily.	Robot moves set amount
Then ROBOT turns right 90 degrees steadily	Robot moves 90 degrees
Then ROBOT waits until it sees something close	Trace shows distance sensor reading and stops when target is reached

After successful integration of the RLit model and robot control classes, a Looper class was developed to test a list of Instructions run sequentially, and the tests were rerun, but this time not individually but together to test how the threading worked.

8.4. System testing

At this stage, the first round of system testing occurred, based on testing tasks that may be set as programming assignments for Key Stage 2. Three Key Stage 2 children, Liam, Nitzan and Alon assisted in the testing. They were asked to complete the following tasks:

1. Program the robot to draw a square
2. Program the robot to move around the room for 30 seconds without hitting a wall
3. Program a four line conversation between robot and Android
4. Program the robot to change speed in time to the Android's music



Alon, 10, testing RoboLiterate, experiences success with his program

8.4.1. Problems with Bluetooth connection

Problems were experienced generally with the Bluetooth connection, including the application crashing if the robot closed its connection (e.g. if it was turned off), or if one jumped repeatedly between the StoryBuilder and ProgramExecutor Activities.

Not all these issues could be solved in the timeframe, since the conditions under which these crashes happened were quite unpredictable and will need deeper analysis. As a short term solution to solve these problems, the bluetooth connection was opened and closed each time the user moved to and from the Executor Activity. In addition, more Bluetooth connection checks were inserted throughout the execution code.

Other issues relating to Bluetooth latency and robot performance meant that the robot sometimes behaved unpredictably, for example turning and moving at different rates when the same instruction was sent from the application. Further work will need to be done to solve these issues, however as these issues were not considered critical to the success of the project, they were deprioritised for later development.

Two major new problems were identified with the UI Controller and Rlit Model layers. The first related to Event Listener sentence combinations using 'When', and the other related to the operation of looping statements using 'Repeat'.

8.4.2. Problem with 'When' event listener

In the UI Controller layer, compound sentences starting with 'When' appear as one sentence, for example,

'When ROBOT sees something close, ROBOT turns left 120 degrees'.

This is to ensure that the sentences flow in exactly the way they would as regular English sentences. In the Rlit Story layer, however, this is stored as two separate RLit sentences:

When ROBOT sees something close,

and

, ROBOT turns left 120 degrees.

In the above example, the comma is treated the same way as the sequencing phrase 'After that'. In tests, this broke when the user deleted or re-edited these sentence. To fix this bug, two avenues were considered - either changing the RLit ontology to accept compound sentence types, or adjust the UI Controller layer to make sure all instances of insertion and deletion of 'When' sentences would work. The latter path was chosen since changing the entire ontology of RLit was deemed too risky.

8.4.3. Problem with looping behaviour

Issues arose when the user created a overlapping loop in their program, for example:

First ROBOT moves forward a little and slowly.

Repeat the last sentence 2 times.

Then ROBOT turns right 90 degrees and slowly.

Repeat the last 2 sentences 2 times.

The result was an infinite loop in the ProgramLooper class code. To solve this problem, two approaches were considered - either changing the Looper class to allow for nested loops, or block their usage at the UI level. Because of the lack of time, and lack of experience in creating threaded looping structures, it was again decided to localise the changes to the UI Controller layer, in the Sentence Builder Activity, because it has fewer dependencies. Now in the final application, the user is unable to create overlapping loops, as the UI controller code checks where the last Repeat statement occurs and ensures the user does not have the opportunity to overlap subsequent Repeat sentences.

8.4.4. RLit phrase ambiguities

Further issues were discovered with the phraseology of RLit, in this testing stage and during the subsequent user evaluation. The issues that arose were mainly to do with ambiguities regarding sequencing and looping.

It was not clear to users if the sequencing phrases 'Then' and 'After a few seconds' indicated concurrent action or sequential action. For example:

First ANDROID plays adventurous music for 30 seconds.

Then ROBOT moves forward continuously and quickly...

One user assumed the second sentence would execute immediately after the first sentence, whereas during the actual execution, he was surprised that it only occurred after 30 seconds. In other words, the user did not differentiate between 'Then' and 'At the same time'. In the final version, the phrase 'Then' has been changed to 'After that' to try to avoid this ambiguity. The user approved this change, however further testing is needed to see whether the confusion remains with others.

All the testees assumed 'After a few seconds' indicated delayed but concurrent action, whereas in the ontology it actually indicated delayed *and* sequential action. This required just a small change in the RLit text file to fix this.

Another ambiguity related to looping. For instance, the following story moves the robot in a square:

First ROBOT moves forward a little and steadily.

After that ROBOT turns left 90 degrees and steadily.

Repeat the last 2 sentences 4 times.

However in testing, users wrote as the final sentence the following:

Repeat the last 2 sentences 3 times.

In other words, they assumed that the repetition of the last two sentences would happen *after* they had first been executed once, so they only need to ask the robot to repeat 3 more times. Again, this ambiguity arose from using natural language as a programming medium rather than using something symbolic, for example brackets. A decision needed to be made as to whether to introduce devices such as brackets, but I chose to stick firmly to NLP, because adding brackets would mean adding a conceptual layer that would need learning, and the objective of the application was to have basic English literacy as the only prerequisite. The solution in this case was simple - in deference to how the users understood the Repeat sentence, it was changed to the following, with the change shown in bold:

Repeat the last 2 sentences 3 ***more*** *times.*

Now using this sentence will get the robot to move in a square path.

8.5. User evaluation

The final application was tested over a two week period at two different international schools in Jaffa, Israel.

Twenty children aged between 7 and 10 were selected by the Principals on the basis of their English speaking skills. The children were also identified according to their interest in English and art. The majority of the children were of Arab or Jewish descent, and did not speak English as their first language, but had at least basic English literacy skills.

Five 30 minute sessions were conducted, each with 4 students and a teacher observer. The teacher observers included the Principal of one school, an IT coordinator and an Art teacher. There were two testers present at each session, one who ran the session (myself) and the other recording with a camera.

The structure of the sessions were as follows:

(1) The students were shown a short demonstration story with the robot. The story was about a lonely robot, who turns around and moves away when asked by Android to play. Then Android offers to play some happy music to cheer the robot up. When the music starts, the robot starts spinning around and beeping.

(2) The children were then told that it was their opportunity to play with the robot and Android. They were introduced to the application RoboLiterate, however they were given no instructions on how to use it. Instead they were told they could ask for help if they got stuck. The children worked in pairs, one pair with an Android Galaxy S, and one pair with a Google Nexus 7 tablet.

(3) The children were observed using the application for 10 minutes. When they were ready to test what they had written, they were guided through the steps of connecting to the robot.

(4) They were then observed as the program executed, and asked questions such as 'What is happening now?', 'Is it what you expected?', 'Why is this happening?', 'What went wrong?', and 'How would you change it?'

(5) Each pair was then given another chance to go back to their story, make changes, and run the program once again. If time, they did this in a third cycle.

(6) As a plenary, they discussed what they liked and didn't like about the application, and how they would like to change it.

In the next section, results of the user evaluation are discussed.

Liam (aged 9) starts his demonstration

9. Chapter 9. Results

The testing went well and most of the pupils were able to write a story and get the robot and Android to perform as they intended. At the end of the 30 minute sessions, the children were generally very keen to continue, and although there wasn't enough time to fully develop story ideas and implement them, several children had begun drafting ideas that they wanted to turn into stories.

There was also evidence of learning linked to the objectives in the Key Stage 1 and 2 Computing programmes of study. The three teachers who observed the sessions showed a great deal of interest in the program and all said after the sessions that they would now be interested in investing in Mindstorms for their school.

Interestingly, the Art teacher was excited about the potential of the application to get children interested in literature and language, a reversal of the original objectives of the project. He suggested that IT-minded children could write stories based on the books they are studying in language class, for example having the robot and Android act out scenes from a Shakespeare story - 'RoboShakespeare'.

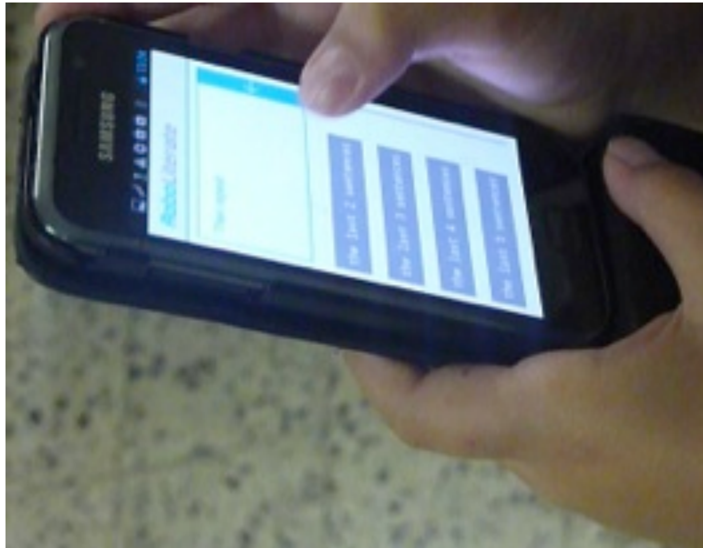
The IT coordinator was excited about the potential of bringing computing into the classroom and away from the IT suite. He was very keen to invest in NXT equipment after the visit. The Principal of Tabeetha school in Jaffa later wrote: "From what I saw the children were having an extremely good time...(they) learned how to solve problems and think logically to find solutions. Also they had [the] opportunity to practice making decisions and putting theories into practice. We are now seriously thinking of getting an NXT system into our school!"

Children testing their stories, with the school principal looking on.

The following sections present the results relating to the two main phases of the sessions - authoring stories and then testing them on the robot.

9.1. Story writing

Most children had little difficulty using the RoboLiterate interface, and once they got past writing the first sentence, they were very quickly writing long stories.



Using the SentenceBuilder interface

About half the children asked for confirmation that they needed to press ‘Start a new story’ on the launch screen, and after that all but one pair quickly recognised that they needed to press ‘Add sentence’ next. Surprisingly, nearly half the children experienced problems with the next Sentence Builder screen (Figure 7.3) and needed to be shown the ‘First’ button. Some children thought they needed to press on the empty text window to begin. However, after they were told to press ‘First’, and more phrases subsequently appeared in the lower part of the screen, all children quickly understood what to do next. In a future iteration, an adaptive feedback layer could indicate to users that they need to press ‘First’ to start their first sentence

The speed with which children picked up the story-writing interface meant that some pairs started building sentences without giving much thought to the story they were constructing. Emboldened by the easy interface, they churned out long essays, stringing phrases together to make a long list of attractive sentences, without thinking how they would work together during program execution.

This was actually beneficial, because for these students, when they ran the program, they came across many problems - endless loops, long pauses, event listeners with no events. It was then that the questioning could start, and the children had to think about the effect of the sentences they had generated. For these students, when they came to build another story, they did it with a lot more thought. In other words, this was their first lesson in coding, debugging and refactoring.

On the SentenceBuilder screen, some children missed the Backspace button and didn’t know they had to press this to delete phrases. This was especially problematic when they were returning to this screen to re-edit a sentence they had previously built. In a future redesign, the Backspace button should be made clearer, possibly by changing the icon to something more Android-centric, or by re-introducing the label ‘Delete’.

No child independently found the RECORD button, which had to be pointed out to them. This was far from ideal because it hid away a very powerful feature, which was very popular when discovered. In future iterations, this feature should be made more salient, possibly with an

adaptive feedback layer highlighting this button if the user hasn't used it after a certain number of visits.

The same issue was found with all ActionBar icons, with no child recognising the 'Save' button and more than half the children needing to have the 'Play' button pointed out to them. Again, in a future iteration, an adaptive feedback layer could show the user the button once he or she has constructed a number of sentences and has not yet found how to play a story.

One critical missing feature was the inability to insert sentences before previously authored sentences, and rearrange the order of sentences. One of the older students, Liam, said it was very frustrating having to delete many sentences as a means of inserting a new one in the middle of his story. In a future iteration, this would be an important feature to add.

Another important issue was children were initially unclear about *what* they could write, as phrase choices only appear as they typed. This meant, at least in their first two iterations of coding and testing in the session, the children were making up their stories as they went along, with little story planning. It was only after about two write-and-test cycles that they started to understand the limits of the language and what they could achieve. Two older children suggested that it would be useful to have a list of all the possible words in one place so they would know what options were available. Another suggested that recommended words would be highlighted to the user to show her a good example of what she can write.

Another interesting suggestion was to include symbols next to the phrases, for example having a curved arrow next to 'turn left'. Others suggested making the interface much brighter with different colours and animations. When questioned as to why they wanted this, the reply was that it would make it look 'more interesting', 'more fun', 'I'd use it more'.

9.2. Program execution

Running the robot program

While the robot and Android performed the story, all children were eagerly monitoring the Android screen to see how their program was executing. One issue was the length of time it took to upload sound files before the programs commenced, however the gradually incrementing percentage readout proved important in letting the children know that something was indeed happening as they waited for the program to begin.

When their programs began, the console proved successful in holding children's attention and provided a good basis for the testers to question the children about what was happening and why. It was at this point that many of the Key Stage 1 and 2 Computing learning objectives came into play.

For example, Mustafa, 9, had written the following:

First ANDROID plays scary music for 1 minute.

After that, ROBOT moves forward a little and steadily.

When he ran his program, he was surprised to find that the second line was not executing. When he was asked why it was happening, he was not sure at first, but when the tester pointed at the words 'After that', he understood. When asked how he would change his program, he already knew - 'Change it to 'At the same time''.

Another child, Nettie, 10, had written the following as her final two sentence:

After a few seconds, ROBOT turns left continuously.

At the same time ANDROID says 'goodbye'.

At the end of her program, the robot didn't stop turning round in circles. When asked why she thought this was happening, her first reply was that the robot had broken, but when told that it was something to do with her program, she looked at the console screen and identified the word 'continuously' as the issue.

The younger children had some problems with the English, and one pair, Ines and Steven, 8, were concerned that their story had 'frozen'. When they went back to edit it, they asked a tester for some help. The problem was that they had inserted an event listener which was never resolved:

After a few seconds, ROBOT waits until it sees something very close.

After that, ROBOT moves forward a little and quickly.

Testing the robot's distance sensor, one step at a time..

The tester encouraged them to try the program again and see what happened. When the program stopped at the first sentence, the tester asked them why nothing was happening. It was clear that English was a problem here, so another child, Liam, helped. He spotted that the

robot was waiting. Now understanding the problem, the tester asked Ines how she planned to fix it. She knew what to do and walked carefully towards the robot, and when it sprang into life, the group were excited and amused to see the result.

This experience exemplifies an issue with the interface that would need to be addressed in future iterations.This was that the users could often not remember what they had programmed to come next, and this was important when the program appeared to ‘hang’, as it is did with Ines and Steven. In other cases, when asked whether they thought their program had ended, or whether something was supposed to happen next, they sometimes couldn’t say. In a future iteration, it will be important to show the whole program on the screen, possible in grey, and highlight the lines as they execute.

9.3. General impressions

It was interesting to note that it was the younger group of children (the 8 year olds) who showed the most fearlessness and joy in using the application. Their shouts of delight, also noted by the Principal, showed that they were experiencing real excitement at getting their robot to perform.

The older children were more hesitant, maybe because they were more wary of getting things wrong, however through their more tentative approach by the end of the half hour some were beginning to think more about the stories they wanted to write, including:

- a story about “Robot has no friends. Then Robot makes friends with Android” (Nadine,8)
- a story about “a robot chasing people”. (Nikol, 10)
- “I want to write an adventure” (Alin, 9)

There was some disappointment that the robot couldn’t do more things, and the younger children couldn’t quite understand why it couldn’t do things like jump or do exactly what you tell it, however this stimulated an interesting plenary discussion about what it means to program a robot. For example, when we were discussing Nikol’s idea about making the robot chase things, we brainstormed about how to do it using RoboLiterate language. That got the children thinking about how to use its sensors to detect distance, and Samir (9) quickly deduced a problem if the person its chasing turns to the side. Although we only had a very short time, this discussion point easily could have been expanded into a whole project, and all based on Nikol’s story idea.

The use of audio added a very important enriching layer to the experience, meaning that the children could personalise their story. It was unfortunate therefore that the NXT’s speaker is so quiet, meaning that unless there was complete quiet, no one could hear it. Hopefully with the EV3 the speaker quality has been improved

10. Chapter 10. Analysis

From these initial tests, there are promising signs that this approach could provide a method for introducing computational thinking into the classroom in an entertaining and engaging way for children most interested in performance and telling stories. Evidence of learning was found by the testers and reinforced by the teacher observers, that matches many of the stated objectives of the new Computing programme of study for Key Stage 1 and 2.

It had already been found during earlier user testing that the application can be used to teach Key Stage 2 objectives, in that they were engaged in “designing, writing and debugging programs to accomplish specific goals with physical systems” (KS2 objective). However, the critical test was to see if computational thinking could happen in a classroom environment where children are not being led ‘recipe-style’ through scripted learning points, but through exploration based on their own imaginations and creativity.

Initial results based on this user evaluation show that this application’s approach is a method that can be used to facilitate this, serving as a basic introduction to ‘creative computational thinking’. There was no explicit goal given to the evaluation groups at the beginning of the session; after they were shown an example program, they were simply encouraged to help fun and explore the application themselves. It was emphasised that there was going to be no ‘right’ or ‘wrong’ answer. By the end of the thirty minutes, many children were already engaged in aspects computational thinking and self-setting goals for the next story they wanted to write.

Some of the observed learning included:

- learning that program execute by following precise and unambiguous instructions
- practice creating and debugging their own programs
- using logical reasoning to predict the behaviour of their programs, and investigating why their program was not behaving as expected.

These are stated objectives of the new Key Stage 1 curriculum.

All the groups were to differing degrees using sequence, selection and repetition in their programs, and using logical reasoning to explain how their programs worked, and detecting and correcting errors. These are all stated objectives in the Key Stage 2 curriculum.

Moreover, the initial tests provided evidence that the application itself is very quick to pick up, with children writing complex programs speedily, and with appropriate questioning, being led into computational thinking within minutes. Most of the children were highly engaged and excited by the activity, as noted by the Principal of Tabeetha school.

Several issues remain which will need to be solved to improve the user experience and make the application more attractive to the ambitious dramatists who wanted to see the robot do many more things right from the start. For example, the ‘macro-behaviours’ that were originally proposed need to be integrated into the RLit language, so that children can begin creating complex behaviours right from the start. In their comments, users wanted the robot to ‘chase’ things, ‘dance’ and ‘jump’. A way of integrating these high-level procedures into the language needs to be developed.In general, to help children to go further, much more work will need to be done to make the language powerful enough to allow children to realise their artistic vision.

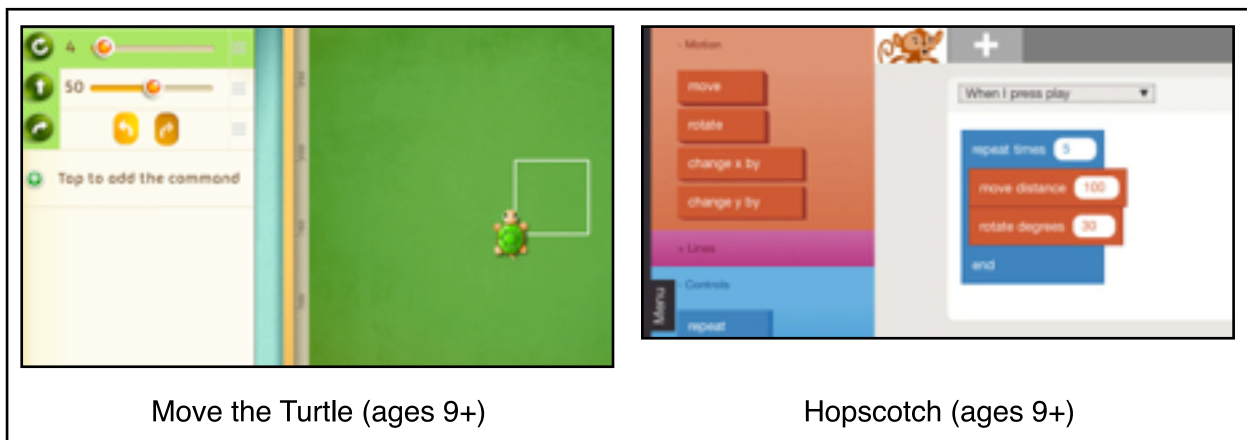
11. Chapter 11. Critical comparison

11.1. Engagement

Having conducted the user evaluation, I wanted compare these findings with results from testing other applications that teach computational thinking to Key Stage 1 and 2 children. In particular I wanted to focus on engagement in learning, and whether RoboLiterate can stimulate the dramatists to learn more than other applications can.

Unfortunately within the time frame I couldn't perform a fair comparative test between these applications and RoboLiterate with any one child, however I had the early data from testing tablet applications with Aiden, 9, and so could perform a loose comparison between that and results from the user evaluation.

Aiden would probably classify himself as a dramatist, as noted before, in that he loves music and acting. However, he is also very quick on the computer and iPad, and was eager to test the programming apps I downloaded for him. I chose two of the most visible applications for the iPad - 'Hopscotch' and 'Move the Turtle', which are both aimed at children aged 9-11 years old. Hopscotch is developed by Hopscotch Technologies and is 'heavily inspired by MIT's Scratch', allowing children to build program routines that control cartoon characters by dragging blocks onto the screen (Hopscotch, 2013). Move the Turtle is similarly aimed at 9-11 year olds, and draws its inspiration from the original Logo, getting children to program a turtle to move and draw patterns on the screen (Turtle, 2013). Screenshots of these applications are shown in Figure 9.1.



Move the Turtle (ages 9+) Hopscotch (ages 9+)

Fig 9.1 Two current iPad applications for teaching programming

Both Hopscotch and Move the Turtle engaged Aiden's interest for a while, and he returned to them 'a few' times over the two week period. It appeared that both applications had different incentives for him to return, however neither were strong enough for him to want to keep on returning to for long compared to, say, Garage Band, which he was constantly revisiting, making and tweaking his musical compositions.

'Move the Turtle' has a very strong tutoring system which leads children step by step from programming the turtle to move, to navigating a maze, to drawing fractals. Aiden enjoyed the reward system and was proud that he had already finished the first eight mini-levels and had reached Level 2. It appeared that achieving a score was the main motivating factor for this application, and he wasn't so interested in the movements of the turtle, because having got to Level 2, he didn't continue. Hopscotch was more interesting for him - after a week, he had managed to move three 'characters' across the screen, painting wavy trails behind them, and

was keen to show me what he had done. In the next week, however, he had not picked up the application again.

With ‘Hopscotch’, he was pleased with the results of his artwork achieved through programming, but there was a ceiling of interest that was reached quite quickly before he moved on. The issue with ‘Move the Turtle’ was that the rewards were not enough to hold his interest. This may be an issue with many applications on the market, which focus on narrow skills-based programming tasks but do not do enough to stimulate the minds of users and empower them to use programming for their own purposes.

It was not possible to test RoboLiterate with Aiden because of geographical separation, but it would have been interesting to see if this would have engaged his interest for longer. The the ideas and reactions that were gathered from the children here in Jaffa indicate that an application such as RoboLiterate might have much more potential to engage the creative mind of Aiden. RoboLiterate gives much more free reign to be creative, to program movements, sounds and dialogues, and to do so in a tangible physical space that gives the feel of a real performance. Clearly, more work will have to be done to see whether an approach like RoboLiterate can engage creative minds longer than these applications but from the results of the short user evaluations, the signs are hopeful.

11.2. Promoting computational thinking

In additional to testing engagement, the applications were analysed together with RoboLiterate to see how well they covered the key concepts and practices that form Brennan and Resnick’s framework for computational thinking (Brennan & Resnick, 2012). Table 9.1 summarises the concepts covered by each application, and shows quite an even coverage:

Concepts	Hopscotch	Move the Turtle	RoboLiterate
Sequences	✔	✔	✔
Loops	✔	✔	✔
Events	✔	x	✔
Parallelism	✔	x	✔
Conditionals	x	✔	✔
Operators	x	✔	x
Data	x	✔	x

Table 9.1 Table showing coverage of computational thinking concepts (Resnick & Brennan 2012)

It is hard for an application on its own to encourage what Brennan and Resnick (2012) identify as computational thinking practices. These practices are: *being incremental and iterative*, *testing and debugging*, *reusing and remixing*, and *abstracting and modularizing*. It is the environment that is built around a product, the context within which it is used, that must be at least as important in influencing the development of these practices. One of Scratch’s great strengths is its online community that encourages the reusing and remixing of other members’ projects.

The products in this study don’t get near Scratch in terms of being able to encourage these practices, however Move the Turtle, Hopscotch and RoboLiterate go some way to achieving it.

Move the Turtle encourages incremental and iterative programming indirectly through its excellent, step-by-step tutoring layer. Hopscotch allows users to share their project via email, so they can work on each other's projects. RoboLiterate does not yet have this functionality, but since all RLit stories are saved as JSON files, this would be easy to implement in future versions. At present, stories can be saved and shared with others who use the same device.

A great strength of RoboLiterate is how its design encourages debugging. This was witnessed in the user evaluation. It explicitly shows and updates the status of a running program in its console window, similar to a debugging window in an IDE. Many children used this to discuss what was happening with their program and why it wasn't behaving as expected.

12. Chapter 12. Conclusion

The application developed for this project introduces a different way to approach programming and computational thinking for Key Stage 1 and 2 children. It has the potential to open up programming to children and teachers who feel more comfortable with words and stories than they do with patterns and symbols. It could provide a means for children to explore theme-based programming that is cross-curricular and not tied to a specific skills-led agenda. At the same time it has been shown in testing to be rigorous enough that in a short 30 minute session, many aspects of computational thinking are touched upon, covering several aspects of the new Computing curriculum.

In general it has met the objectives of the project:

- Promoting computational thinking: RoboLiterate gives students experience in sequencing, looping, events and other concepts, and introduces them to the practice of creating and debugging simple programs, and sharing work with others
- Possess a fast learning curve: in testing, all children could master the interface quickly, and most of these children had English as a second language. However, more work will need to be done to introduce children to the range and depth of what they can create, and not just how.
- Appeal to the dramatists: from the evaluation, there was much excitement and interest, with children already planning what stories they wanted to tell. More work will need to be done to help dramatists tell their stories, however, with further examples for them to work with, and richer functionality and behaviours for the robot.
- Be future ready: the aim of the system design was to ensure that the application could be adapted easily for the EV3 version of Mindstorms. Only time will tell if this has been the case.

Lots of further work has been identified which will need to be done to improve RoboLiterate so that it can be more stable, educational, user-friendly and rewarding for the dramatists. Crucial improvements include the following:

- Re-introducing ‘procedures’, in the spirit of the proposal, but designed to fit with the RLit interface. This will allow for complex behaviours to be used right from the beginning, and later adapted and refined by users
- Develop the language and interface to allow for control over variables and data
- In parallel, fixing the Bluetooth issues so that the connection is more stable and the robot movements are better calibrated with the language
- Add adaptive feedback layers, that not only help users with the interface, like identifying ActionBar buttons, but also give guidance on how the explore the full range of actions available.

This project has been a huge undertaking considering the time and resources available and I feel it has just scratched the surface of all the possibilities in this area. However, as Computing becomes a central part of the new curriculum, I feel strongly that as many different approaches needed to be trialled as possible to ensure that this curriculum is a success and that *all* students and teachers will be kept on board.

13. Bibliography

Alimisis, D. (2012). *Robotics in Education & Education in Robotics: Shifting Focus from Technology to Pedagogy.* Paper presented at the Proceedings of the 3rd International Conference on Robotics in Education, Charles University in Prague.

Alimisis, D, & Moro, M. (2007). Robotics & constructivism in education: the TERECoP project.

BBC. (2008). The Hitchhiker Adventure Game. from http://www.bbc.co.uk/radio4/hitchhikers/game_nolan.shtml

Bers, M. (2008). Engineers and storytellers: Using robotic manipulatives to develop technological fluency in early childhood. In O. S. B. Spodek (Ed.), *Contemporary Perspectives on Science and Technology in Early Childhood Education*: Information Age Publishing.

Brennan, K, & Resnick, M. (2012). *New frameworks for studying and assessing the development of computational thinking.* Paper presented at the American Educational Research Association (AERA) annual conference.

Cellbots. (2011). Cellbots: Using Cellphones as Robotic Control Platforms. from http://www.cellbots.com/robot-platforms/lego-mindstorms/

Education, Department for. (2013). *National curriculum in England: computing programmes of study*. Retrieved from https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study.

Eguchi, A; Hughes, N; Stocker, M; Shen, J; Chikuma, N. (2012). RoboCupJunior - A Decade Later. In T. R. e. al. (Ed.), *RoboCup 2011* (pp. 63-77): LNCS 7416.

Falk, Geoffrey. (2013). *Controlling Next Generation Mindstorms Robots using Android Devices.* (Masters Project Proposal), Birkbeck College and Institute of Education.

Fedor, J. (2011). NXT Remote Control Android application. from https://play.google.com/store/apps/details?id=org.jfedor.nxtremotecontrol

Foundation, Logo. (2012). The Logo Programming Language. from http://el.media.mit.edu/logo-foundation/logo/programming.html

Gobel, S; Jubeh, R; Taesch, S-L; Zundorf, A. (2011). *Using the Android Platform to Control Robots*. Paper presented at the 2nd International Conference on Robotics in Education.

Google. (2012). Bluetooth Chat. from https://play.google.com/store/apps/details?id=com.blong.bluetoothchat&hl=en

Google. (2013a). Android Developers' training site. from http://developer.android.com/training/index.html

Google. (2013b). Bound Services. from http://developer.android.com/guide/components/bound-services.html

Google. (2013c). Fragments - Android Developers API Guide. from http://developer.android.com/guide/components/fragments.html

Google. (2013d). google-gson: a Java library to convert JSON to Java objects and vice-versa. from http://code.google.com/p/google-gson/

Google. (2013e). Saving Data in SQL Databases.

Google. (2013f). Top Paid in Education.

Griffin, Terry. (2010). *The Art of Lego Mindstorms NXT-G Programming*.

Hopscotch. (2013). Hopscotch Technologies. from https://www.gethopscotch.com

IDC. (2013). Top Smartphone Operation Systems, Shipments, and Market Share, 2013 Q3. from http://www.businesswire.com/news/home/20130807005280/en/Apple-Cedes-Market-Share-Smartphone-Operating-System

Inform7. (2013). About Interactive Fiction. from http://inform7.com/if/interactive-fiction/

Klassner, F, & Anderson, S D. (2003). LEGO MindStorms: Not Just for K-12 Anymore. *IEEE Robotics & Automation Magazine*.

Lego. (2006). Lego Mindstorms NXT Bluetooth Developer Kit. from http://mindstorms.lego.com/en-us/support/files/default.aspx

Lego. (2012). MINDroid application. from https://play.google.com/store/apps/details?id=com.lego.minddroid

Lego. (2013a, 22 January 2013). Announcing Lego Mindstorms EV3. from http://mindstorms.lego.com/en-us/News/ReadMore/Default.aspx?id=476243

Lego. (2013b). Comparing EV3 with NXT. from http://www.legoeducation.us/eng/misc/comparingEV3andNXT.cfm

Lego. (2013c). Lego WeDo range. from http://education.lego.com/en-gb/preschool-and-school/lower-primary/7plus-education-wedo/

leJOS. (2011). Using leJOS with Android. from http://lejos.sourceforge.net/nxt/nxj/tutorial/Android/Android.htm

McNerney, Timothy S. (2004). From turtles to Tangible Programming Bricks: explorations in physical language design. *Personal and Ubiquitous Computing, 8*, 326-337.

MIT. (2013a). Scratch. from http://scratch.mit.edu

MIT. (2013b). Scratch - Explore up-to-date statistics about the Scratch Online Community. from http://stats.scratch.mit.edu/community/

Papert, S. (1980). *Mindstorms: children, computers, and powerful ideas*. NY: Basic Books.

Resnick, M. (2012). *Mother's Day, Warrior Cats and Digital Fluency: Stories from the Scratch Online Community.* Paper presented at the Constructionism 2012, Athens, Greece.

Robinson, Ken. (2013, Friday 17 May). To encourage creativity, Mr Gove, you must first understand what it is, Comment piece, *The Guardian*. Retrieved from http://www.theguardian.com/commentisfree/2013/may/17/to-encourage-creativity-mr-gove-understand

RoboCupJunior. (2013). RoboCup Junior website. from http://rcj.robocup.org/dance.html

RobotC. (2012). RobotC. from http://www.robotc.net/teachingmindstorms/

Rusk, N, Resnick, M, Berg, R, & Pezalla-Granlund, M. (2008). New Pathways into Robotics: Strategies for Broadening Participation. *Journal of Science, Education and Technology*(17), 59-69.

SysBrain. (2007). Why sEnglish? , from http://www.system-english.com/?page=why

Terrapin, Software. (2013). Bee-Bot and Bee-Bot Support Materials. from http://www.terrapinlogo.com/bee-bot.php

Turtle, Move the. (2013). Move the Turtle. from http://movetheturtle.com

Wikipedia. (2013a). Colossal Cave Adventure. from http://en.wikipedia.org/wiki/Colossal_Cave_Adventure

Wikipedia. (2013b). Natural language programming. from http://en.wikipedia.org/wiki/Natural_language_programming

14. Appendix A: RLit Ontological framework

An initial version of the ontology was developed and then tested. After comments, revisions were made to make it as 'natural' and powerful as possible. The names of the self-standing entities in RLit adopt the nomenclature of grammatical language, such as 'sentence', 'verb', 'phrase', however this does not mean that these entities are synonymous with their linguistic namesakes,. Rather they share commonalities, and using these labels helps make sense of RLit's structure. All subsequent uses of these words refer to them within the context of RLit, rather than grammatical syntax, unless explicitly stated otherwise.

14.1. RLit Story

The Story is the 'document' that forms the basis for the executable program. The Story is composed of a sequence of 'sentences', of which there are 3 types:

DIRECTION SENTENCES – Sentences that issue simple instructions. These instructions can appear anywhere in a Story.

EVENT LISTENER SENTENCES – these Sentences monitor readings from the robot sensors, and pause the flow of the Story until stipulated conditions are met (either a target reading is met or a time period elapses)

EVENT RESULT SENTENCES – these sentences occur after EVENT LISTENER sentences that start with 'When'

RLitStories all need to begin with a Sentence which contains the 'Sequencer' phrase 'First'. Subsequent sentences all begin with a variety of other 'Sequencer' phrases, depending on the composition of the immediately preceding sentence(s).

An initial decision was to restrict the language to 'simple' sentences i.e. no compound sentences, for example 'when…do' sentences. This is because I wanted to keep the language as simple as possible, and make sure each sentence mapped directly to an instruction, whilst still providing a feature set that would make full use of the robot's sensors.

For example, in the first iteration, the following Story instructed the robot to move around a room, avoiding walls:

1. First ROBOT moves forward continuously.

2. At the same time ROBOT waits until it sees something close

3. Then ROBOT turns left 120 degrees.

4. Repeat the last three sentences for 1 minute.

From a programmer's point of view, this makes some kind of sense. Sentence 1 sets out the initial state, Sentence 2 is an event condition, and sentence 3 is the result of the condition being met. The final sentence surrounds the previous three sentences in a loop. This is indeed how the instructions are sent to the robot.

However, when tested it was apparent that this wasn't how users would ever explain the sequence orally. Firstly, it is linguistically awkward – how can a robot move continuously and wait at the same time? If the word 'waits' was substituted with 'senses', 'observes', it still didn't feel comfortable. This is because the separation of condition and result into separate sentences (2 and 3) is not what we do in natural language - we use if... and when... clauses instead.

For this current Masters project, I initially felt the demands of developing the ontology and subsequent implementation to include compound sentences would go beyond the practical scope of this project.

In the end, a solution was found - to maintain the underlying RLit ontology of just using simple sentences, but change the UI layer so from the user's perspective, compound sentences are now admissible. For example, users can type the following and it will have the same result as the story above:

1. First ROBOT moves forward continuously.

2. When ROBOT sees something close, ROBOT turns left 120 degrees.

3. Repeat the last two sentences for 1 minute.

Final testing provided confirmation that this was well worth the extra work. Users were technically still able to write stories using either of the methods above. Not one user thought of using the first method – everyone naturally used the second approach i.e. using When.

This did have some knock on effects on the ontological structure of RLitStory, in particular with the 'Repeat' sentence, and extra rules governing ordering of sentences. For this reason the ontology needed to include the three-fold classification of sentences and ensure that within RLit Story entities, EVENT LISTENER sentences that begin with 'When' are always followed by EVENT RESULT sentences, that begin with a comma. An example of this comes later.

14.2. RLit Sentences

The 'sentences' are the basic building blocks of RLit. A sentence is defined as a structure that contains at least one 'phrase', and this phrase must be of type 'verb' (which has many similarities but is not synonymous with the grammatical definition of verb).

Every 'verb' maps to a specific Instruction, which is identified through the verb's InstructionID. Through its verb phrase. each Sentence is therefore mapped to one Instruction. A sentence can never map to more than one Instruction.

In other words, each Sentence contains one and only one verb. In addition, every Sentence in RLit (so far) has a Noun, expect for sentences that govern program flow, such as the verb 'Repeat'.

All other phrase types are optional within an RLit sentence, however if they occur they must assume the following order:

Sequencer phrase + NOUN phrase + Auxiliary phrase + Verb phrase + Modifier phrase + Quantifier modifier.

Only one phrase of each type can occur in one sentence. For example, a sentence cannot contain two modifiers, although it can contain one modifier and one quantifier.

The following are examples of acceptable sentences within RLit:

1. First ROBOT moves back a little and quickly.
2. After 3 seconds ANDROID says 'What's up'.
3. When ROBOT sees something very close
4. , ROBOT turns left about 90 degrees and slowly.
5. At the same time ROBOT beeps C for a whole note.

Although all these are admissible RLit Sentences, only examples 1, 2 and 5 are sentences in an everyday sense. Examples 3 and 4 show how RLit divides compound linguistic structures into separate entities/'sentences'.

In the application, sentences are built via the UI by sequentially adding phrases to a growing sentence, beginning in all cases with ‘Sequencer’ type phrases, apart from ‘Repeat’ which is classified as a verb phrase. The phrase selections that are made available are governed by the ID of the immediately preceding phrase in the sentence (the ‘parent’ phrase). Appendix B show the complete ontological map of phrases, with all possible phrase combinations.

14.3. RLit Phrases

RLit Phrases are the subunits of every RLit Sentence.

Each phrase consists of a set of String and integer fields that dictate how they work together in a Sentence, and this orchestration is led by the sentence’s Verb phrase.

All phrases include a human-readable label, which is what appears in the UI, and a sort index, which defines where in the list of phrases they appears. This is important for usability issues, so that the most important phrases can promoted higher up on the list than more obscure, or challenging phrases. For example, ‘quickly’ appears higher than ‘very quickly’, and ‘moves back’ appears higher than ‘says’ (getting the Robot to say things involves a long sound upload procedure).

Aside from this, phrases can utilise up to four variables (three integers and one string), a delay variable, and for Verbs only, an InstructionID. Finally, each phrase has an ID, and a parent ID. The parent ID defines how phrases can follow each other in RLit sentences. Appendix B shows this relationship. Each phrase is boxed with a certain ID, and the arrows show the parent-child relationships between the phrases.

There are six types of phrase that can be included in a sentence:

SEQUENCERS

Phrases that control the timing of execution of a sentence. These phrases will tell the interpreter whether the sentence should be executed immediately after the previous sentence (‘Then’), after a delay (‘After x seconds’), or concurrently (‘At the same time’). A special Sequencer is “<<comma>>” which appears as a regular comma in the UI, but which affects program flow in the same way as the Sequencer phrase ‘Then’. However, as mentioned before, this Sequencer can only appear in sentences that immediately follow a ‘When’ type Event Listener sentence.

‘When’ is also a Sequencer, and has the same value as ‘At the same time’.

NOUNS

Phrases that define the object on which the sentence is executed on. At this initial stage, the two objects are ROBOT and ANDROID, although there is no reason why this cant be extended as the application is developed to control a wider range of devices in the future.

AUXILIARIES

These phrases are a vestige of the first iteration of the ontology, when event handling was addressed using the structure ‘wait until...‘ As this entails having two ‘verb‘ like structures in one sentence, this type was needed to differentiate between them. The only member of this type therefore is ‘wait (until)’. In future iterations, however, the ontology could be developed to use Auxiliaries in other ways, for example through introducing probabilistic (might, will probably) or frequency control (usually, sometimes).

VERBS

The central phrase in any sentence, the verb specifies the required action that the Noun will execute. Each verb is mapped to an Instruction object that in turns executes the program on the specified Noun, modulated by the other phrases in the sentence.

MODIFIERS

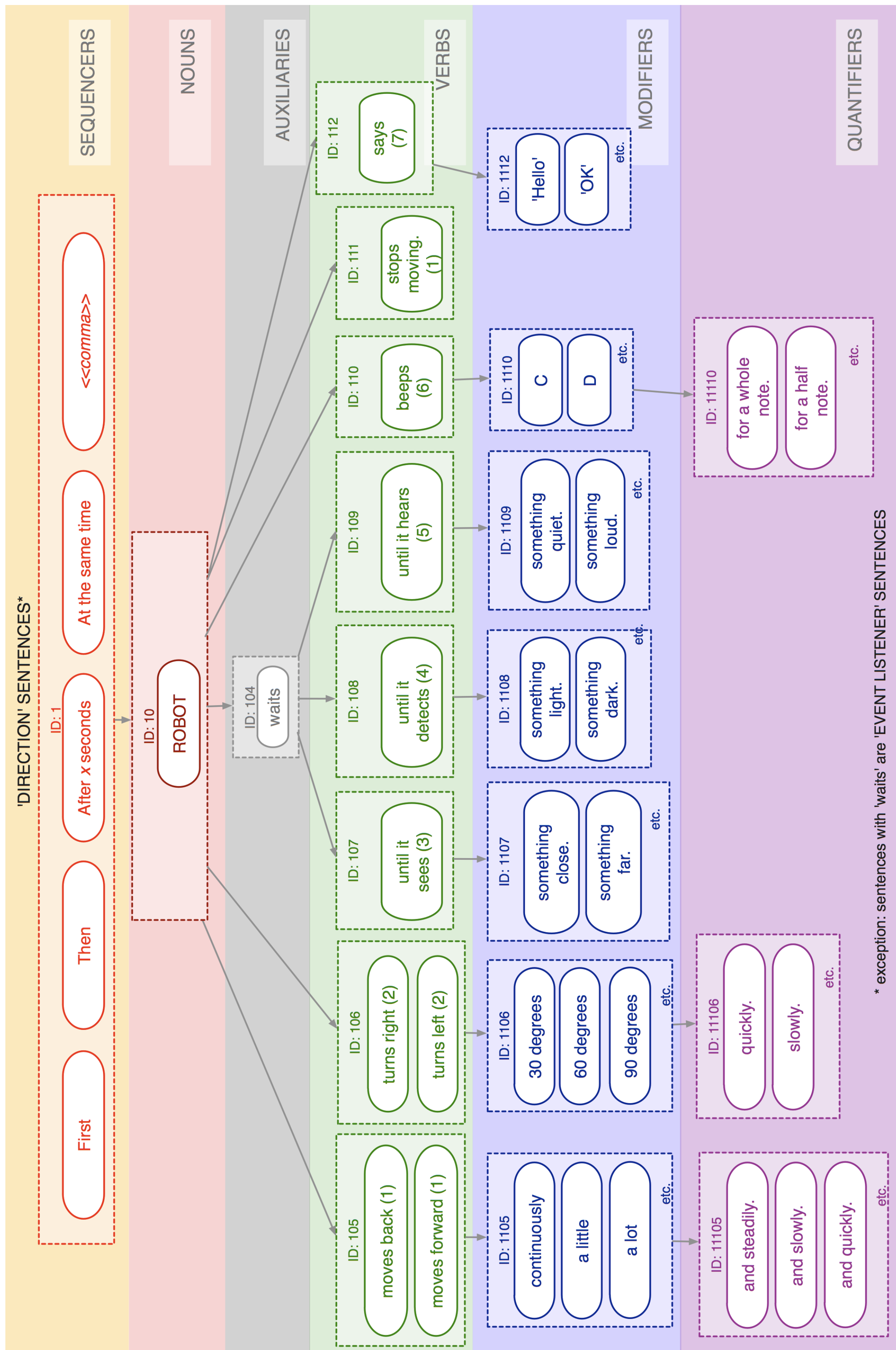
Modifiers are phrases that add parameters to the verbs, usually in terms of adding measures such as distance, degree, colour, sound frequency, or in the case of user generated sounds, filenames.

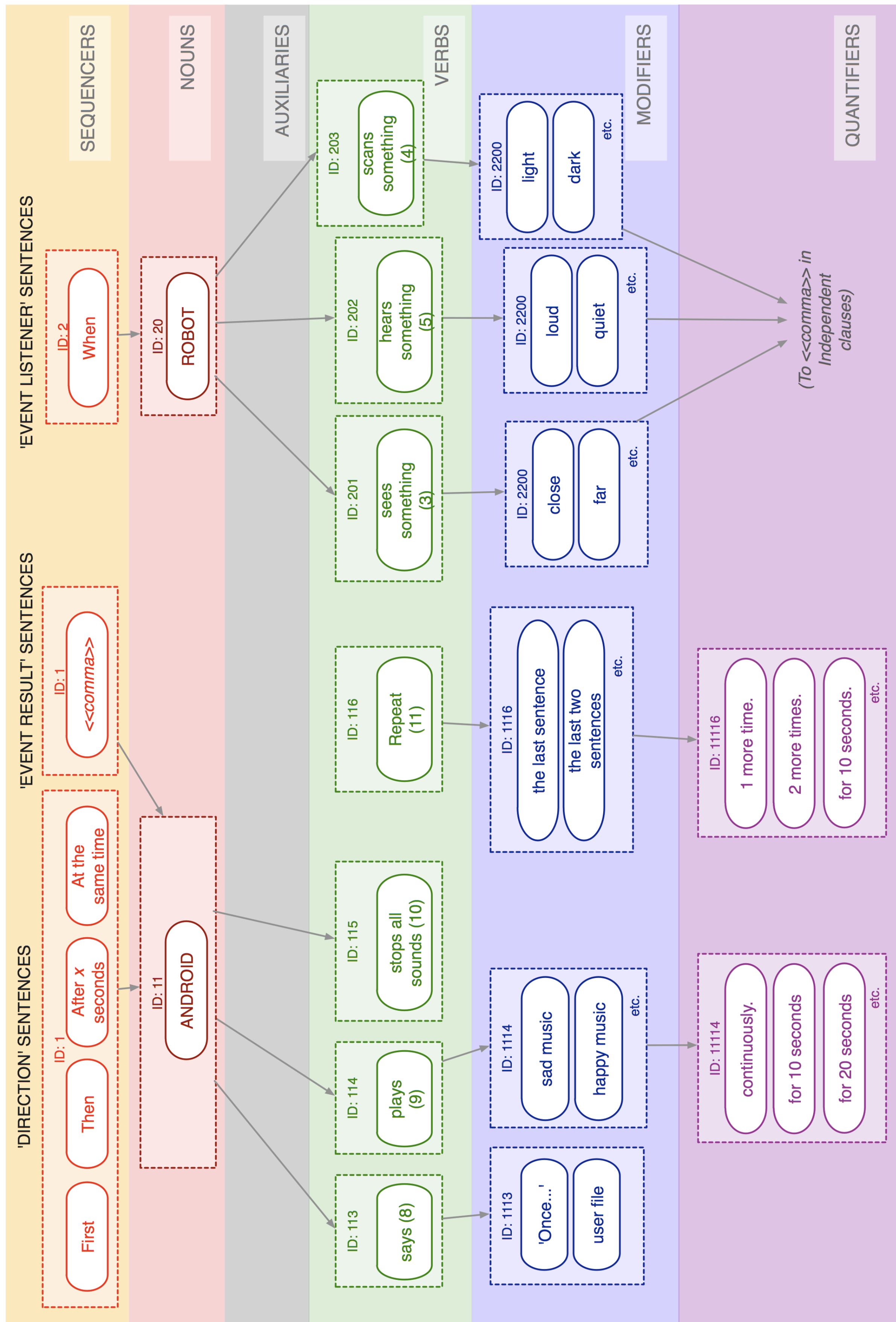
QUANTIFIERS

As the type name suggests, Quantifiers add a time or speed measure, for example ‘slowly’, ‘quickly’ , ‘for 10 seconds’, ‘continuously’.

In the development of the ontology, both the modifiers and quantifiers went through several changes based on the results of iterative testing. It is perfectly possible for this ontology to allow users to finely control the movements of motors and analysis of sensor readings, in the way other robot programming languages for Mindstorms allow (for example, RobotC, leJOS). This includes providing exacting control over the amount of degrees you want the robot to turn, and wheel rotations to cover a certain distance, depending on the configuration of the robot. However, this was not implemented for the purposes of this project

15. Appendix B: RLit Phrase Maps




'EVENT LISTENER' SENTENCES
'EVENT RESULT' SENTENCES
'DIRECTION' SENTENCES
SEQUENCERS
NOUNS
AUXILIARIES
VERBS
MODIFIERS
QUANTIFIERS
ID: 2
When
ID: 1
<<comma>>
ID: 1
First
Then
After x seconds
At the same time
ID: 20
ROBOT
ID: 11
ANDROID
ID: 203
scans something (4)
ID: 202
hears something (5)
ID: 201
sees something (3)
ID: 116
Repeat (11)
ID: 115
stops all sounds (10)
ID: 114
plays (9)
ID: 113
says (8)
ID: 2200
light
dark
etc.
ID: 2200
loud
quiet
etc.
ID: 2200
close
far
etc.
ID: 1116
the last sentence
the last two sentences
etc.
ID: 1114
sad music
happy music
etc.
ID: 1113
'Once...'
user file
(To <<comma>> in Independent clauses)
ID: 11116
1 more time.
2 more times.
for 10 seconds.
etc.
ID: 11114
continuously.
for 10 seconds
for 20 seconds
etc.

16. Appendix C: Table of RLit Phrases and their values

EnglishLabel	ID	Parent ID	InstructionID	Delay	Arg 1	Arg 2	Arg 3	Arg4	Wait Flag	Sort Order
First	1	-1	0	0	0	0	0		0	1
After a few seconds	1	0	0	3	0	0	0		0	5
After 5 seconds	1	0	0	5	0	0	0		0	6
After 10 seconds	1	0	0	10	0	0	0		0	7
At the same time	1	0	0	0	0	0	0		0	2
After that	1	0	0	0	0	0	0		1	1
When	2	0	0	0	0	0	0		0	4
comma	1	2200	0	0	0	0	0		1	1
ROBOT	10	1	0	0	0	0	0		0	1
ANDROID	11	1	0	0	0	0	0		0	2
ROBOT	20	2	0	0	0	0	0		0	1
waits	104	10	0	0	0	0	0		0	6
moves forward	105	10	1	0	1	0	0		0	1
moves back	105	10	1	0	-1	0	0		0	2
turns left	106	10	2	0	0	-1	0		0	3
turns right	106	10	2	0	0	1	0		0	4
until it sees	107	104	3	0	0	0	0		0	1
until it detects	108	104	4	0	0	0	0		0	3
until it hears	109	104	5	0	0	0	0		0	2
beeps	110	10	6	0	0	0	0		0	7
stops moving.	111	10	1	0	0	0	0		0	8
says	112	10	7	0	0	0	0		0	5
says	113	11	8	0	0	0	0		0	1
plays	114	11	9	0	0	0	0		0	2
stops all sounds.	115	11	10	0	0	0	0		0	3
Then repeat	116	0	11	0	0	0	0		1	3
sees something	201	20	3	0	0	0	0		0	1
hears something	202	20	5	0	0	0	0		0	2
scans something	203	20	4	0	0	0	0		0	3
a tiny bit	1105	105	0	0	0	10	0		0	2
a little	1105	105	0	0	0	20	0		0	3
quite a lot	1105	105	0	0	0	45	0		0	4
a lot	1105	105	0	0	0	60	0		0	5
continuously	1105	105	0	0	0	0	0		0	6
30 degrees	1106	106	0	0	30	0	0		0	2
90 degrees	1106	106	0	0	90	0	0		0	4
180 degrees	1106	106	0	0	180	0	0		0	5
360 degrees	1106	106	0	0	360	0	0		0	6
continuously	1106	106	0	0	0	0	0		0	7
something very close.	1107	107	0	0	5	20	0		0	1
something quite close	1107	107	0	0	21	30	0		0	2
something quite far.	1107	107	0	0	31	100	0		0	3

EnglishLabel	ID	Parent ID	InstructionID	Delay	Arg 1	Arg 2	Arg 3	Arg4	Wait Flag	Sort Order
something dark.	1108	108	0	0	0	200	0		0	1
something bright.	1108	108	0	0	800	1000	0		0	2
something quiet.	1109	109	0	0	300	500	0		0	1
something loud.	1109	109	0	0	501	700	0		0	2
something very loud.	1109	109	0	0	701	1000	0		0	3
A	1110	110	0	0	440	0	0		0	1
B FLAT	1110	110	0	0	466	0	0		0	2
B	1110	110	0	0	493	0	0		0	3
C	1110	110	0	0	523	0	0		0	4
C SHARP	1110	110	0	0	554	0	0		0	5
D	1110	110	0	0	587	0	0		0	6
E FLAT	1110	110	0	0	622	0	0		0	7
E	1110	110	0	0	659	0	0		0	8
F	1110	110	0	0	698	0	0		0	9
F SHARP	1110	110	0	0	740	0	0		0	10
G	1110	110	0	0	784	0	0		0	11
A flat	1110	110	0	0	831	0	0		0	12
‘Go away.’.	1112	112	0	0	2	0	0	goaway.rso	0	0
‘I’ll be back.’.	1112	112	0	0	2	0	0	arniewillbeback.rs	0	0
‘I’m so depressed.’.	1112	112	0	0	2	0	0	imsodepressed.rs	0	0
‘Oops!’.	1112	112	0	0	1	0	0	oops.rso	0	0
‘Stop it!’.	1112	112	0	0	1	0	0	stopit.rso	0	0
‘Will you be my friend	1112	112	0	0	2	0	0	willyoubemyfriend	0	0
‘You’re mine!’.	1112	112	0	0	2	0	0	youremine.rso	0	0
‘Armed.’.	1112	112	0	0	1	0	0	armed.rso	0	0
‘Awesome!’.	1112	112	0	0	1	0	0	awesome.rso	0	0
‘Bang!’.	1112	112	0	0	1	0	0	bang.rso	0	0
‘Burrrp!’.	1112	112	0	0	1	0	0	burp.rso	0	0
‘Collecting!’.	1112	112	0	0	1	0	0	collecting.rso	0	0
‘Come closer.’.	1112	112	0	0	2	0	0	comecloser.rso	0	0
‘Confirmed.’.	1112	112	0	0	1	0	0	confirmed.rso	0	0
‘Danger!’.	1112	112	0	0	1	0	0	danger.rso	0	0
‘Dooh!’.	1112	112	0	0	1	0	0	dooh.rso	0	0
‘Got it.’.	1112	112	0	0	1	0	0	gotit.rso	0	0
‘Gotcha!’.	1112	112	0	0	1	0	0	gotcha.rso	0	0
‘Hello Dave.’.	1112	112	0	0	2	0	0	hellodave.rso	0	0
‘Help!’.	1112	112	0	0	1	0	0	help.rso	0	0
‘Hot!’.	1112	112	0	0	1	0	0	hot.rso	0	0
‘I feel happy.’.	1112	112	0	0	2	0	0	ifeelhappy.rso	0	0
‘I’m hungry!’.	1112	112	0	0	2	0	0	imhungry.rso	0	0
‘Identify.’.	1112	112	0	0	1	0	0	identify.rso	0	0
‘Is it ok?’.	1112	112	0	0	2	0	0	isitok.rso	0	0
‘It is OK.’.	1112	112	0	0	2	0	0	itsok.rso	0	0
‘Lost object.’.	1112	112	0	0	1	0	0	lostobject.rso	0	0
‘Love.’.	1112	112	0	0	1	0	0	love.rso	0	0
‘Me.’.	1112	112	0	0	1	0	0	me.rso	0	0

EnglishLabel	ID	Parent ID	InstructionID	Delay	Arg 1	Arg 2	Arg 3	Arg4	Wait Flag	Sort Order
‘Motors Forward.’.	1112	112	0	0	2	0	0	motorsforward.rso	0	0
‘Motors Reverse.’.	1112	112	0	0	2	0	0	motorsreverse.rso	0	0
‘Move away!’.	1112	112	0	0	2	0	0	moveaway.rso	0	0
‘Negative.’.	1112	112	0	0	1	0	0	negative.rso	0	0
‘O Sole Mio!’	1112	112	0	0	4	0	0	osolemio.rso	0	0
‘Object detected.’.	1112	112	0	0	1	0	0	objectdetected.rs	0	0
‘OK?’.	1112	112	0	0	1	0	0	ok_question.rso	0	0
‘OK!’.	1112	112	0	0	1	0	0	ok_statement.rso	0	0
‘Oomph!’.	1112	112	0	0	1	0	0	oomph.rso	0	0
‘Oops!’.	1112	112	0	0	1	0	0	oops.rso	0	0
‘Perfect.’.	1112	112	0	0	1	0	0	perfect.rso	0	0
‘Phew!’.	1112	112	0	0	1	0	0	phew.rso	0	0
‘Positive.’.	1112	112	0	0	1	0	0	positive.rso	0	0
‘Shoot!’.	1112	112	0	0	1	0	0	shoot.rso	0	0
‘Atchoo!’.	1112	112	0	0	1	0	0	sneeze.rso	0	0
‘Thanks.’.	1112	112	0	0	1	0	0	thanks.rso	0	0
‘That was easy.’.	1112	112	0	0	2	0	0	thatwaseasy.rso	0	0
‘Touch me.’.	1112	112	0	0	1	0	0	touchme.rso	0	0
‘Tracking object.’.	1112	112	0	0	2	0	0	trackingobject.rso	0	0
‘Turning left.’.	1112	112	0	0	2	0	0	turningleft.rso	0	0
‘Turning right.’.	1112	112	0	0	2	0	0	turningright.rso	0	0
‘What’s up?’.	1112	112	0	0	2	0	0	waazuup.rso	0	0
‘Wow!’.	1112	112	0	0	1	0	0	wauw.rso	0	0
‘You.’.	1112	112	0	0	1	0	0	you.rso	0	0
‘Yawn!’.	1112	112	0	0	1	0	0	yawn.rso	0	0
‘Yeehah!’	1112	112	0	0	2	0	0	yeehah.rso	0	0
‘Once upon a time the	1113	113	0	0	2	0	0	onceuponatime.m	0	1
sad music	1114	114	0	0	0	0	0	music_sad_sombı	0	3
happy music	1114	114	0	0	0	0	0	music_goodbye_l	0	1
adventurous music	1114	114	0	0	0	0	0	music_monster_fa	0	4
chasing music	1114	114	0	0	0	0	0	music_mr_tiny.mp	0	5
quirky music	1114	114	0	0	0	0	0	music_night_is_da	0	6
scary music	1114	114	0	0	0	0	0	music_arabian_thi	0	2
the last sentence	1116	116	0	0	1	0	0		0	1
the last 2 sentences	1116	116	0	0	2	0	0		0	2
the last 3 sentences	1116	116	0	0	3	0	0		0	3
the last 4 sentences	1116	116	0	0	4	0	0		0	4
the last 5 sentences	1116	116	0	0	5	0	0		0	5
the last 6 sentences	1116	116	0	0	6	0	0		0	6
the last 7 sentences	1116	116	0	0	7	0	0		0	7
the last 8 sentences	1116	116	0	0	8	0	0		0	8
the last 9 sentences	1116	116	0	0	9	0	0		0	9
the last 10 sentences	1116	116	0	0	10	0	0		0	10
very close	2200	201	0	0	5	20	0		0	1
quite close	2200	201	0	0	21	30	0		0	2
quite far	2200	201	0	0	31	100	0		0	3

EnglishLabel	ID	Parent ID	InstructionID	Delay	Arg 1	Arg 2	Arg 3	Arg4	Wait Flag	Sort Order
quiet	2200	202	0	0	300	500	0		0	1
loud	2200	202	0	0	501	700	0		0	2
very loud	2200	202	0	0	701	1000	0		0	3
dark	2200	203	0	0	0	200	0		0	1
bright	2200	203	0	0	800	1000	0		0	2
and slowly.	11105	1105	0	0	30	0	0		0	1
and steadily.	11105	1105	0	0	60	0	0		0	2
and quickly.	11105	1105	0	0	90	0	0		0	3
and slowly.	11106	1106	0	0	0	25	0		0	1
and steadily.	11106	1106	0	0	0	50	0		0	2
and quickly.	11106	1106	0	0	0	80	0		0	3
for a whole note.	11110	1110	0	0	0	600	0		0	1
for a half note.	11110	1110	0	0	0	300	0		0	2
for a quarter note.	11110	1110	0	0	0	150	0		0	3
for 1 minute.	11114	1114	0	0	0	60	0		0	4
for 30 seconds.	11114	1114	0	0	0	30	0		0	3
for 20 seconds.	11114	1114	0	0	0	20	0		0	2
for 10 seconds.	11114	1114	0	0	0	10	0		0	1
continuously.	11114	1114	0	0	0	0	0		0	5
for 10 seconds.	11116	1116	0	0	0	10	1		0	4
for 20 seconds.	11116	1116	0	0	0	20	1		0	5
for 30 seconds.	11116	1116	0	0	0	30	1		0	6
1 more time.	11116	1116	0	0	0	0	1		0	1
2 more times.	11116	1116	0	0	0	0	2		0	2
3 more times.	11116	1116	0	0	0	0	3		0	2
4 more times.	11116	1116	0	0	0	0	4		0	2
5 more times.	11116	1116	0	0	0	0	5		0	3