

**Finch 4 Alice:
A Visual Interface for
Programming the Finch Robot**

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin-La Crosse

La Crosse, Wisconsin

by

Brad Fisher

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

April, 2017

Finch 4 Alice: A Visual Interface for Programming the Finch Robot

By Brad Fisher

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

David Riley
Examination Committee Chairperson

Date

Kasi Periyasamy
Examination Committee Member

Date

Kenny Hunt
Examination Committee Member

Date

ABSTRACT

FISHER, BRAD, C., Finch 4 Alice: A Visual Interface for Programming the Finch Robot, Master of Software Engineering, April 2017, 100pp, David Riley, Kasi Periyasamy, Kenny Hunt.

The project described in this manuscript investigates the concept of incorporating an API for controlling the Finch robot into a visual programming environment, specifically that provided by Alice 3.

Several initial approaches are outlined to accomplish the task, with one approach chosen for implementation. Details of several of the technical challenges encountered are provided, along with approaches and techniques employed to address them.

The final implementation resulted in the open source project Finch 4 Alice, which can be found at <http://finch4alice.com>.

ACKNOWLEDGEMENTS

I wish to thank Dr. David Riley who provided the project concept and acted as project advisor. Many thanks also to all of the individuals responsible for providing the Master of Software Engineering program at the University of Wisconsin – La Crosse.

Appreciation is also extended to Carnegie Mellon University, the Carnegie Mellon University CREATE Lab, and the Alice Project Team, for producing Alice and the Finch robot.

I also extend heartfelt gratitude and dedicate this project to my wonderful wife, Nancy, and my sons, Brayden and Orion, who have all been very patient and supportive.

To Alanna – our little angel in Heaven.

TABLE OF CONTENTS

ABSTRACT.....	iii
ACKNOWLEDGEMENTS.....	iv
TABLE OF CONTENTS.....	v
LIST OF FIGURES.....	viii
LIST OF TABLES.....	xi
GLOSSARY.....	xii
1. Introduction.....	1
1.1 Alice 3.....	1
1.2 The Finch.....	2
2. Project Background.....	4
2.1 Option 1 – Acquire access to the Alice 3 source code.....	4
2.2 Option 2 – Utilize a different visual programming environment.....	4
Finch Dreams.....	5
CREATE Lab Visual Programmer for Finch.....	7
Scratch & Snap!.....	7
Blockly.....	8
2.3 Option 3 – Reverse-engineering of Alice 3.....	9
3. Through the Looking Glass: Hacking Alice 3.....	10
3.1 ZIP Open a JAR.....	12
3.2 Overriding and Enhancing Classes in Java.....	14
Bytecode Weaving with AspectJ.....	15
Replacing a Class in an Existing JAR File.....	16
Provide a new JAR.....	17
3.3 Decompilers Provide a Source.....	18
JD-GUI.....	18

CFR (Class File Reader)	26
3.4 Compiling the Generated Sources	28
Determining the Compilation Class Path.....	28
File Organization and Script for Compilation	30
Effectiveness of Decompilers for Integrating Finch Support	31
3.5 It's a Swinging Interface.....	33
Launching Alice 3 with Swing Explorer	33
Runtime Inspection of the Alice 3 GUI	36
4. Enhancing Alice.....	38
4.1 Exposing New Procedures and Functions	38
Annotations, Exposed	49
Identifying a Suitable Test Subject.....	52
4.2 Communicating with the Finch	53
USB Debugging with USBPCap	55
Down the Rabbit Hole	56
4.3 The BirdBrain Robot Server.....	57
Operating System-specific Issues	60
Unsupported Finch Functionality	61
4.4 Supporting New Releases of Alice 3: Augmenting the classpath	62
5. Finch 4 Alice Deployment	64
5.1 Supporting Multiple Operating Systems	64
5.2 Build Automation with Gradle	64
The Only Manual Dependency is the JDK	65
The Gradle Wrapper	65
Configuration through Code	65
Dependency Management.....	66
Plugin Support	66
Simple Command Line Interface	66
5.3 Cross-Platform Graphical Installer	67

5.4	Platform-Specific Installer Options	70
	Windows Executable Wrapper	71
	Shell Script Wrapper.....	71
5.5	API Documentation	71
5.6	Automated Builds and Release Artifact Publishing	72
5.7	Acquiring Finch 4 Alice	72
6.	Future Work	74
6.1	Use in Introductory CS Courses	74
6.2	Maintenance of the Finch 4 Alice Open Source Project	74
6.3	Enhancements to the Finch Representation in Alice 3	75
6.4	Enhance Alice with Functionality Beyond Finch.....	75
	REFERENCES	77
	Appendix.....	83
	Disclaimers	83
	Finch 4 Alice BSD 2-Clause License	83

LIST OF FIGURES

Figure 1: The Alice 3 visual programming interface.....	2
Figure 2: The Finch robot	3
Figure 3: Finch Dreams	6
Figure 4: CREATE Lab Visual Programmer for Finch.....	7
Figure 5: Snap! with Finch block definitions loaded.....	8
Figure 6: Blockly example interface.....	9
Figure 7: Alice 3 program folder contents.....	10
Figure 8: Folder view of “ast-0.0.1-SNAPSHOT.jar” opened in 7-Zip	13
Figure 9: Zip listing of “croquet-0.0.1-SNAPSHOT.jar”	14
Figure 10: Examining Alice 3 classes with JD-GUI.....	18
Figure 11: Result when bytecode cannot be interpreted.....	20
Figure 12: Example of duplicate variable declarations.....	20
Figure 13: Example of an improperly decompiled for-each loop.....	21
Figure 14: Corrected Figure 13 code with proper for-each loop usage	21
Figure 15: Example of omission of required typecast	22
Figure 16: Example of missing return keyword	23
Figure 17: Example of invalid initialization of static properties	23
Figure 18: Example of incorrect assertion decompilation	24
Figure 19: Code from Figure 18 with corrected assertion	24
Figure 20: Example of incorrectly decompiled enum with abstract methods	25
Figure 21: Corrected decompilation of the code from Figure 20	25
Figure 22: CFR output for a method it could not decompile.....	27
Figure 23: Example of unintuitive code generated by CFR	28
Figure 24: JD-GUID output for same for-each loop illustrated in Figure 23	28
Figure 25: Windows batch syntax for deriving class path.....	30

Figure 26: Bash script syntax for deriving class path.....	30
Figure 27: Windows batch file used to compile generated source files	31
Figure 28: The Swing Explorer interface.....	33
Figure 29: Java arguments embedded in Alice 3 native launcher executable	34
Figure 30: Batch file used for launching Alice 3.3 under Swing Explorer.....	36
Figure 31: Swing Explorer with SportsCar say method selected	37
Figure 32: Subtree of Swing view hierarchy for procedure elements	37
Figure 33: Source of isInstanceFactoryDesiredForType from StoryApiConfigurationManager	38
Figure 34: Decompiled source of the org.alice.ide.member.views.MethodsSubView class	40
Figure 35: org.alice.ide.member.MethodsSubComposite subclasses in Alice 3	41
Figure 36: Decompiled org.alice.ide.member.MemberTabComposite's getSubComposites method.....	43
Figure 37: Decompiled source of the org.lgna.project.ast.JavaType methods initialization	45
Figure 38: Decompiled source of org.lgna.project.ast.JavaType's handleMthd method .	46
Figure 39: Decompiled source of org.lgna.project.ast.AbstractMethod's "isFunction" ...	47
Figure 40: Decompiled source of org.lgna.project.ast.AbstractMethod's "isProcedure" .	48
Figure 41: Decompiled source of org.alice.ide.member.UserProceduresSubComposite's "isAcceptable" method	48
Figure 42: Decompiled source of org.alice.ide.member.MemberTabComposite's "isInclusionDesired"	48
Figure 43: Decompiled source of org.lgna.project.ast.JavaMethod's "getVisibility"	49
Figure 44: Example of a MethodTemplate annotation used to prevent method exposure	51
Figure 45: Example of using Visibility.TUCKED_AWAY to prevent method exposure	51
Figure 46: Example of MethodTemplate annotation indicating method is to be exposed	51
Figure 47: Original decompiled source of org.lgna.story.STransport class	53
Figure 48: Example of Finch method added to STransport class	53

Figure 49: Wireshark capturing communication with a Finch robot using USBPcap.....	55
Figure 50: The BirdBrain Robot Server user interface.....	58
Figure 51: Example of a Windows install4j vmoptions file	63
Figure 52: Example of a Linux install4j vmoptions file.....	63
Figure 53: Example of executing Gradle 'createExe' task	67
Figure 54: The Finch 4 Alice graphical installer	68
Figure 55: GitHub Release artifacts of Finch 4 Alice v0.4.....	72

LIST OF TABLES

Table 1: Sampling of Alice 3 external library dependencies	11
Table 2: Listing of Alice 3 internal libraries.....	12
Table 3: Java arguments extracted from the Alice 3 launcher exe	35
Table 4: Annotation classes defined in the Alice codebase.....	50
Table 5: BirdBrain Robot Server Finch Sensor Value Services.....	60
Table 6: BirdBrain Robot Server Finch Control Services	60
Table 7: Gradle plugins used by Finch 4 Alice.....	66

GLOSSARY

3D (Graphics)

In computer graphics, the term 3D refers to the technique of projecting a virtual object or scene onto a computer display in a way that presents the illusion of three-dimensionality.

Abstract Syntax Tree (AST)

A representation of the individual elements of a computer program often implemented as a tree of nodes. Each node in the tree describes a specific piece of program syntax, such as variable declarations, assignments, logical and mathematical operations, function declarations, or method calls. An AST is often generated by a code parser and may be generated as an intermediate step in compilation or interpretive script execution.

Accelerometer

A sensor for measuring acceleration forces. Accelerometers are commonly incorporated into electronic devices such as cell phones or video game controllers, and are often used to determine device orientation or if the user is shaking or tapping the device.

Alice 3

A freeware educational programming language with an integrated development environment (IDE), produced by Carnegie Mellon University. Alice 3 uses a drag and drop environment to create computer animations using 3D models, and places an emphasis on object-oriented concepts and facilitating a full transition to the Java programming language. [16]

Application Programming Interface (API)

A set of routine definitions, protocols, and tools for building software and applications. An API specification can take many forms, but often include specifications for routines, data structures, object classes, or variables. [68]

Decompiler

A decompiler, or reverse compiler, is a program that attempts to perform the inverse process of the compiler: given an executable program compiled in any high-level language, the aim is to produce a high-level language program that performs the same function as the executable program. [21]

Finch

A small robot designed to inspire and delight students learning computer science by providing them a tangible and physical representation of their code. [7] The Finch robot was designed at the CREATE Lab at Carnegie Mellon University, and is currently marketed by BirdBrain Technologies, LLC.

Graphical User Interface (GUI)

A graphical user interface utilizes the graphical capabilities of a computer to provide a user interface for computer software. GUIs often employ images, symbols and visual cues to facilitate user interaction in a point-and-click manner. Common methods for providing user input include the use of keyboards, mice, pen devices, touch screens or other devices for physical interaction.

Hyper Text Transfer Protocol (HTTP)

An application-level data transmission protocol used for exchanging data between client and server applications. Both HTTP requests and responses consist of a request method, a resource identifier, headers, and optional data.

Integrated Development Environment (IDE)

A suite of tightly-integrated development tools used for software development, often consisting of a combination of an assistive code editor (with syntax-highlighting, code completion and correction, or other features), compilers, static code analysis tools, runtime debugger, and performance and memory profilers.

JAR

The JAR (**J**ava **A**Rchive) format is typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform. [50]

Java

A popular, general-purpose, cross-platform programming language first released by Sun Microsystems in 1995 and currently maintained by Oracle Corporation. Code written in the Java language is typically compiled to class files containing Java bytecode, which can be executed by a Java Virtual Machine.

Java Bytecode

The instruction set executed by the Java Virtual Machine. Java bytecode is similar in principal to the instruction sets understood by most physical computer processors.

Java Classloader

A part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine.

Java Runtime Environment

A software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the core Java classes.

Java Virtual Machine

An implementation of the Java Virtual Machine Specification [53], typically in software form, for executing Java bytecode. A Java Virtual Machine exposes core system facilities to Java applications in a consistent manner, regardless of the host operating system type or CPU architecture, simplifying cross-platform development.

Light Emitting Diode (LED)

A semiconductor diode that emits light when conducting current. LEDs are often preferred over incandescent lights in electronic devices due to their lower power consumption and reduced heat output for comparable light intensity.

Open Source Software

Open Source Software (OSS) is software distributed with a license allowing access to its source code, free redistribution, the creation of derived works, and unrestricted use. [1]

Reverse Engineering

Chikofsky and Cross [20] define Reverse Engineering as the process of analyzing a software system to:

- Identify the system's components and their interrelationships, and
- Create representations of the system in another form or at a higher level of abstraction

For the context of this paper, it refers to attempting to divine the purpose, intent and operational parameters of the Alice 3 software to determine a means to extend it to support the Finch.

Swing

Swing is a subset of the Java Foundation Classes (JFC) which provides a framework for creating graphical user interfaces. Several common component implementations, such as buttons, text labels, and file selection dialogs are provided, and custom components can be implemented using the Java language. Swing components are considered lightweight, since

they are implemented in Java and do not require native implementations for different operating systems or hardware.

Visual Programming Language

A language which uses some visual representations to accomplish what would otherwise have to be written in a traditional one-dimensional programming language. [56] Programming is accomplished with visual expressions, generally spatial arrangements of text and graphic symbols.

1. Introduction

Students often understand abstract ideas more effectively if they can relate them to their own experiences with real-world objects. When teaching programming to entry level students, it can be beneficial to present the concepts in a way that enables students to easily visualize code execution. A desire to incorporate a more concrete link to the physical world into the curriculum for CT100 at the University of Wisconsin – La Crosse has prompted interest in developing a visual environment for programming and interacting with a device capable of providing a physical representation of software execution.

Previous courses have successfully used Alice 3 [18], a free visual programming environment created by Carnegie Mellon University, to introduce programming concepts. Its straightforward graphical interface enables students to easily create syntactically correct programs without requiring the students to be aware of the syntactical rules themselves. By providing an environment designed to build 3-dimensional animations manipulated through code, along with a migration path to the Java language, Alice 3 is suitable for entry to intermediate-level programming students.

The CREATE Lab [19] at Carnegie Mellon University has also developed a simple, interactive robot called the Finch [7]. Specifically designed for Computer Science student education, the Finch can be used to provide a robotic device that is programmable by students. It offers several sensors, output options, and mobility as a means of bridging the gap between the virtual and physical realms.

By using both Alice 3 and the Finch robot together, the goal is to increase engagement and remove some of the hurdles traditionally faced by students when learning to program.

1.1 Alice 3

The Alice 3 project offers a visual programming interface for programmatically animating objects within a 3D rendered virtual environment. Figure 1 illustrates Alice 3,

which compared to its predecessor, Alice 2, provides a similar graphical environment and visual code editor, but is designed to be much more object-oriented. Alice 3 also uses a more Java-like language syntax, and supports features for exporting projects to the Java language. As a result, Alice 3 is well-suited for use in introductory postsecondary programming courses to familiarize students with common concepts before switching to Java instruction. [24]

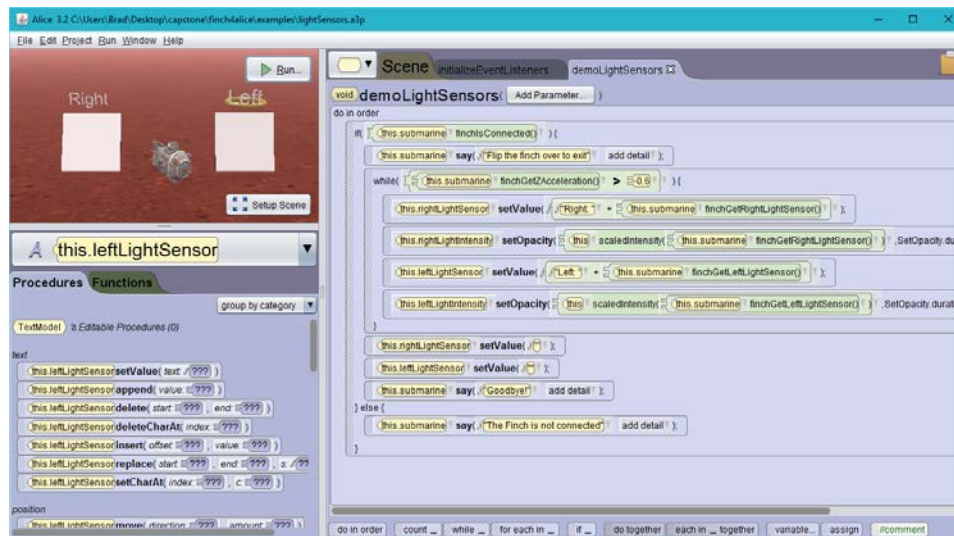


Figure 1: The Alice 3 visual programming interface

Through the use of simple drag-and-drop, interactive dialogs, and context sensitive menu selections, Alice 3 provides a great deal of guidance to novice programmers. The interface helps students avoid many common programming mistakes by ensuring code can only be constructed in a syntactically correct manner. Removing some of these common barriers, Alice 3 can be an effective tool to streamline the learning process [23,46].

1.2 The Finch

The Finch robot was designed to be a simple, cost-effective tool for educating students in the art of programming [15,41,42,48]. Several Application Programming Interfaces (APIs) are available to control the Finch from common programming languages, including Java, Python, C/C++ and others. An HTTP-based server, known as the Birdbrain Robot

Server [8], is also available to allow integration with languages which do not have direct API support, but do provide the ability to communicate over HTTP.

The Finch robot includes several on-board sensors, feedback mechanisms, and motors (See Figure 2). Among its array of sensors are an accelerometer, a temperature sensor, light sensors, and obstacle detection sensors. The Finch also supports output capabilities via a full-color LED, an internal buzzer, and two independently controlled wheels to provide mobility.

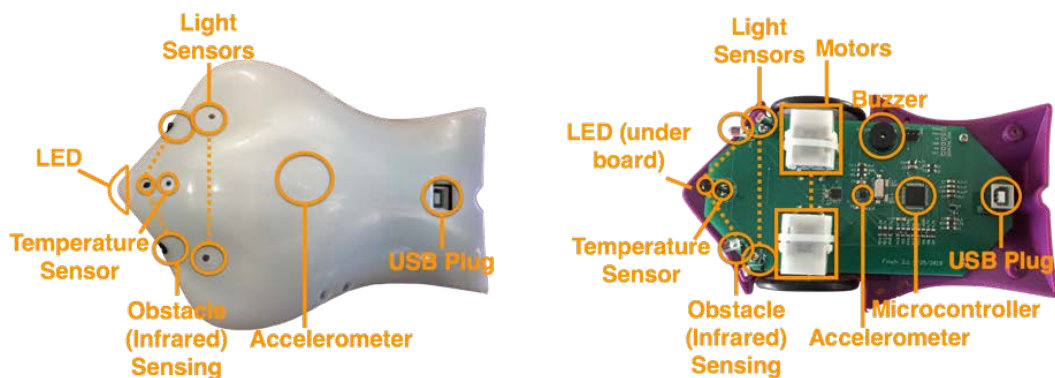


Image © BirdBrain Technologies, LLC.

Figure 2: The Finch robot

By combining the Finch's capabilities, students can write computer software to interact with the physical world in a rich and intuitive manner.

2. Project Background

To achieve the goal of integrating the Finch API [5] within a visual programming environment, three main options were considered:

1. Acquire access to the source code and API documentation for Alice 3 from the Alice team at Carnegie Mellon University.
2. Identify an alternative UI that has visual code editing capabilities similar to those provided by Alice 3 and is extensible to support the Finch API.
3. Attempt to reverse-engineer the Alice 3 program to determine a way to integrate the Finch functionality, without direct access to the source code.

2.1 Option 1 – Acquire access to the Alice 3 source code

While several sources [16,25,67], including the Alice 3 license [17], imply that Alice 3 is open source software, there is no known public source code repository from which to acquire the Alice 3 source code for review or modification. Unsuccessful attempts had been made prior to the commencement of this project to gain official access to the Alice 3 source code from the Alice team at Carnegie Mellon University. As part of the initial phases of this project, a new request was submitted to the team, again receiving a negative response. Since we were unable to convince the Alice team to grant access to the source code for Alice 3, this option was not a viable route.

2.2 Option 2 – Utilize a different visual programming environment

Several existing visual programming environments were evaluated as part of the requirements-gathering phase of the project, with emphasis on the following criteria:

- The software must be open source and freely available for student use.
- The source code must be available through a public source code repository.

- The software must run under the Windows, Macintosh OS X, and Linux operating systems.
- The software must provide a drag-and-drop visual programming interface.
- If Finch support is not yet available, the software must be extensible to add support through an external plugin or by updating available source code to interface with an existing Finch API [5] or the BirdBrain Robot Server [8].
- It must support standard object-oriented programming concepts, such as:
 - Encapsulation through classes with properties and methods
 - Inheritance of behavior and properties
- Support for intermediate to advanced language concepts should be provided.
 - Lists / Arrays
 - Recursion
 - Strict data typing
 - Parallel/concurrent execution paths
- The software must be capable of serving as a gateway or introduction to the Java programming language. The ability to export projects directly to Java source code is preferred.

Several software alternatives were considered during this phase, including Finch Dreams, Finch Visual Programmer, Scratch/Snap! and Blockly.

Finch Dreams

Finch Dreams [6] is a variation of Alice 2.2 which has been extended to include a Finch 3D library object which can be added into the scene and that exposes methods to access the Finch's sensors and output devices. Figure 3 illustrates with a sample Finch Dreams GUI.

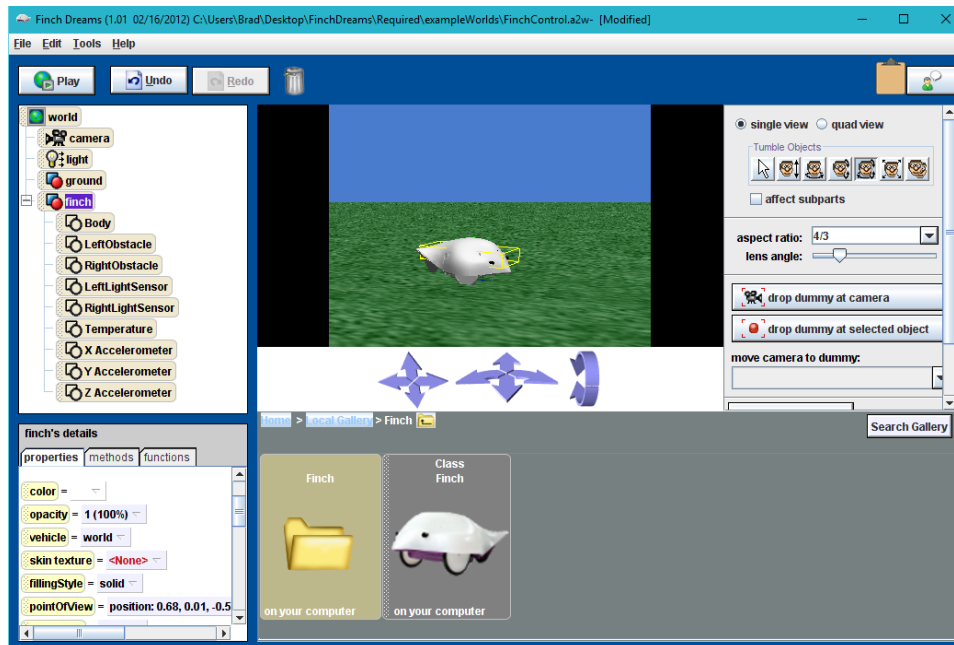


Figure 3: Finch Dreams

Finch Dreams was the closest of all the evaluated software in terms of meeting the evaluation goals and expectations. However, the Alice 2 environment on which Finch Dreams is based primarily targets K-12 students and does not provide access to many of the more advanced programming concepts that are desired for students in an entry level postsecondary environment. Finch Dreams neither supports exporting projects into Java code, nor does it expose students to object-oriented features such as inheritance and method overloading.

Utilizing Finch Dreams as a source of insight for the project described in this paper was considered, but was found to be of limited value. The Finch Dreams source code, like the source code for Alice 3 itself, is not readily available for examination. Also, since Finch Dreams is based on an old (circa 2011-12) version of Alice 2, it was deemed very probable that the code base shared between it and Alice 3 had diverged substantially in the meantime.

During the course of the project research, source code [58] was found for Storytelling Alice [38], a different fork of the Alice 2.2 source. Examination of the Storytelling Alice source revealed a very large number of implementation differences between its source code and that of Alice 3. The mechanism which Storytelling Alice uses to determine the methods

exposed to users for use in their own programs was found to be quite different than that for Alice 3. Due to these great differences observed between Storytelling Alice and Alice 3, it is very unlikely that the source code for Finch Dreams would be of much use for integrating the Finch into Alice 3, even if it could be obtained.

CREATE Lab Visual Programmer for Finch

The CREATE Lab Visual Programmer (see Figure 4) for Finch [14] is an open source visual programming tool developed by the CREATE Lab at Carnegie Mellon University, the organization responsible for the development of the Finch itself.



Figure 4: CREATE Lab Visual Programmer for Finch

Using the CREATE Lab Visual Programmer for Finch, one can indeed create a program to control a Finch to interact with its environment. However, the design of the Visual Programmer software is targeted at novice programmers, and as such does not provide access to several desirable programming features. Lack of user-defined variables, an inability to translate projects easily into Java code, and many other limitations prevented this tool from being useful for the desired purposes.

Scratch & Snap!

Scratch [43] and Snap! [66] are both free, browser-based visual programming languages. Scratch is a project of the Lifelong Kindergarten Group at the MIT Media Lab. Snap! is presented by the University of California at Berkeley, and is a reimplementations of Scratch

with additional features added, such as procedures, lists, and continuations. Figure 5 demonstrates a Snap! screen capture with Finch block definitions.

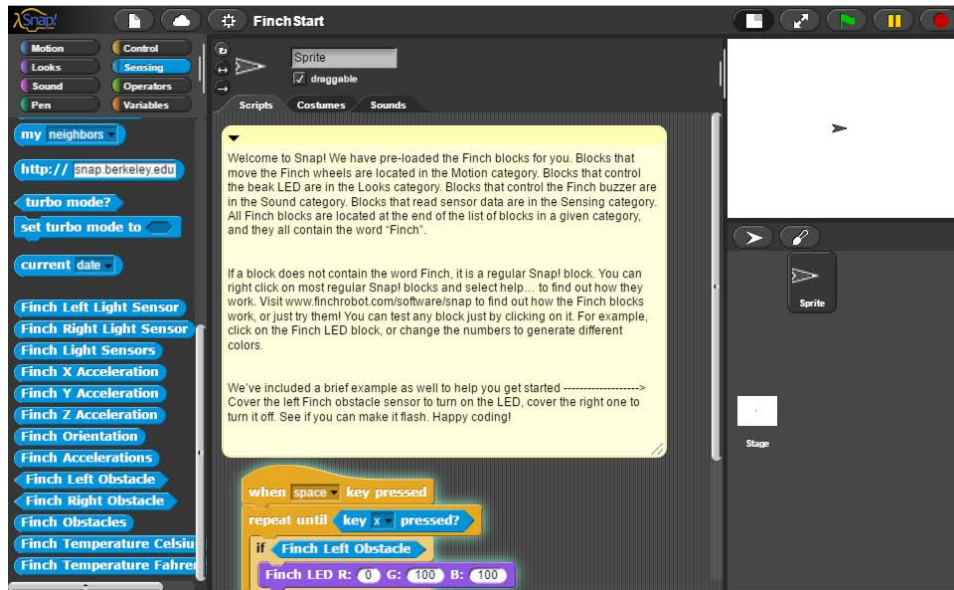


Figure 5: Snap! with Finch block definitions loaded

Both Scratch and Snap! provide very similar environments in which to develop programs. They both also have existing extensions for supporting interaction with a Finch. However, neither provides a strictly-typed, object-oriented language, nor are they similar enough to Java to ease the transition from the visual environment to a text-based IDE.

Blockly

Blockly [33] is a library created by Google for building visual programming editors. It allows a developer to embed a visual programming interface into a web page or Android application. A sample programming interface window from Blockly is shown in Figure 6. The Blockly editor provides facilities to generate code from the visual representation into several popular scripting languages, including Javascript, PHP, Python or Lua.

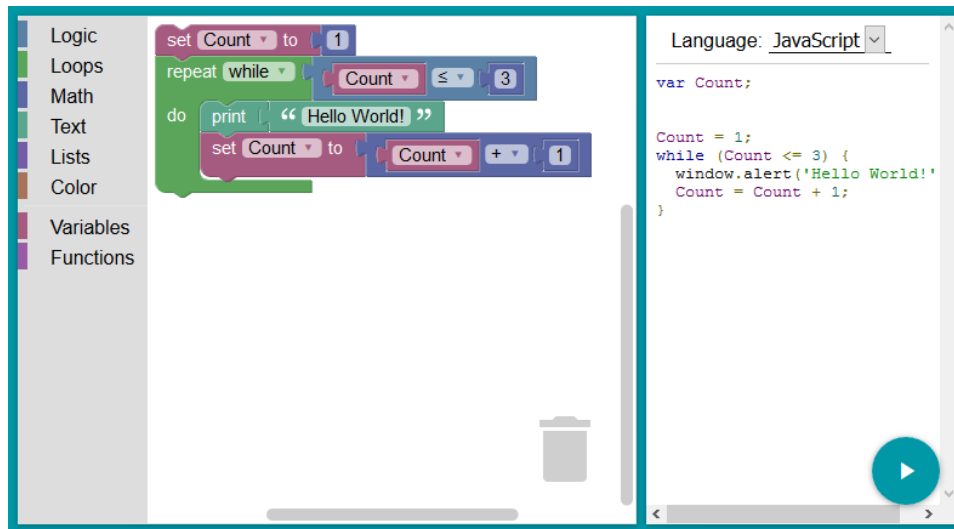


Figure 6: Blockly example interface

While the Blockly library appears to have some promise as a drop-in visual programming editor control, and its browser-based implementation provides wide platform support, it doesn't support the object-oriented programming styles desired for the project. It also appears to be focused more on supporting loosely typed scripting languages at this point, and doesn't provide typed variable declarations. For these reasons, Blockly was deemed insufficient to meet the needs of the project.

2.3 Option 3 – Reverse-engineering of Alice 3

Since the source code for Alice 3 was not easily available through standard channels, and no acceptable alternative software was found that met the desired requirements, only one of the identified options remained. Some method of incorporating functionality into Alice 3 to access the Finch would need to be determined. Attempting to modify the software would require reverse-engineering and analysis using various decompilation and introspection techniques to determine if and how the desired functionality could be added. Chapter 3 details the approaches and tools used to accomplish this task.

3. Through the Looking Glass: Hacking Alice 3

The first task to determine the viability of enhancing Alice 3 with Finch support was to gain an understanding of its internal workings. This was crucial to determining viable options for incorporating new functionality into the existing program. Since the original source code is not publicly available, more indirect methods of examining and understanding the construction of the software was required.

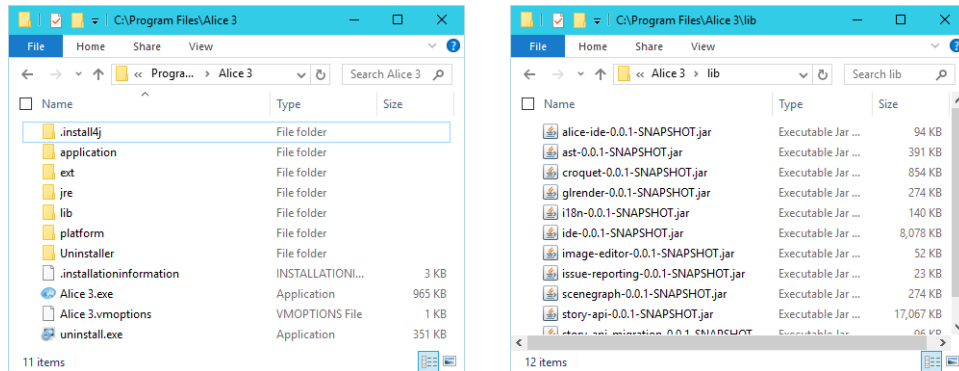


Figure 7: Alice 3 program folder contents

Alice 3 is implemented in the Java language and its executable components are packaged in a manner similar to most other Java projects. The source code is compiled to class files containing Java virtual machine bytecode, and packaged into Java Archive (JAR) files. Figure 7 shows the basic structure of the Alice 3 files.

Inside the Alice 3 program folder, these JAR files are organized into folders denoting whether they are external 3rd party library dependencies, or were generated from the original Alice 3 source code. External dependencies are placed in an “ext” subfolder, while libraries specific to Alice 3 are deposited in a “lib” folder. An abbreviated selection of the external library dependencies in the “ext” folder for Alice 3.3 are listed in Table 1.

Library Name	Description
Atlassian Jira SOAP client	For submitting issue reports to the Alice 3 Jira issue tracker
Google Guava	Google Guava contains functionality such as collections, caching, primitives support, concurrency libraries, common annotations, string processing, I/O, and so forth. – https://github.com/google/guava
JMF MP3	Java MP3 player – http://www.oracle.com/technetwork/java/javase/download-137625.html
Jogl2	Java OpenGL API binding – http://jogamp.org/jogl/www/
MiG Layout	Layout manager for Java Swing, SWT, and JavaFX 2 – http://www.miglayout.com/
Mmsc	MM's Computing open source library for PPM, PGM, and PBM image serialization, and color quantization algorithms
Vlcj	Java bindings to VLC media player – http://capricasoftware.co.uk/#/projects/vlcj

Table 1: Sampling of Alice 3 external library dependencies

Inside the “lib” folder are several other JAR files which contain Alice 3-specific classes. Table 2 contains a full listing of these internal libraries, including the number of individual class files contained within each. Based on only a quick analysis of the file names, these libraries appear to include functionality for internationalization (i18n-0.0.1-SNAPSHOT.jar), generating IDE views and components (alice-ide-0.0.1-SNAPSHOT.jar and ide-0.0.1-SNAPSHOT.jar), graphics rendering (glrender-0.0.1-SNAPSHOT.jar), and issue reporting (issue-reporting-0.0.1-SNAPSHOT.jar). However, the purpose of some of the libraries is not as clearly expressed in their names. For example, the name “ast-0.0.1-SNAPSHOT.jar” only provides three letters with which to divine its purpose, and “croquet-0.0.1-SNAPSHOT.jar” is a curious name as well.

JAR File Name	File Size	Number of Classes
alice-ide-0.0.1-SNAPSHOT.jar	98 KB	34 Classes
ast-0.0.1-SNAPSHOT.jar	400 KB	204 Classes
croquet-0.0.1-SNAPSHOT.jar	857 KB	457 Classes
glrender-0.0.1-SNAPSHOT.jar	277 KB	128 Classes
i18n-0.0.1-SNAPSHOT.jar	173 KB	0 Classes
ide-0.0.1-SNAPSHOT.jar	6,845 KB	1,775 Classes

image-editor-0.0.1-SNAPSHOT.jar	53 KB	12 Classes
issue-reporting-0.0.1-SNAPSHOT.jar	23 KB	11 Classes
scenegraph-0.0.1-SNAPSHOT.jar	280 KB	203 Classes
story-api-0.0.1-SNAPSHOT.jar	18,986 KB	1,238 Classes
story-api-migration-0.0.1-SNAPSHOT.jar	101 KB	21 Classes
util-0.0.1-SNAPSHOT.jar	860 KB	583 Classes
Total	28,953 KB	4,666 Classes

Table 2: Listing of Alice 3 internal libraries

Since Alice 3 incorporates features of a code editor, compiler, and execution environment, it might be guessed that “ast” refers to the abstract syntax trees commonly used within compilers and interpretive execution environments to represent the structure of the code. As far as “croquet” goes, a good guess would probably be that it is a reference to chapter 8 of Lewis Carroll’s “Alice in Wonderland”, in which Alice plays a croquet game with flamingo mallets and hedgehog balls.

While the above guesses may not be too illuminating, one great benefit of the fact that Alice 3 is implemented in Java is that there are several readily available tools to facilitate the examination of the JAR files and the class files contained therein. These tools can provide much more insight than mere file name analysis. One such tool is the humble ZIP file viewer.

3.1 ZIP Open a JAR

The JAR file format is a specialization of the format employed by the ZIP file format, which is designed to provide lossless data compression of files and directory structure. A JAR file has a specific organization to the compressed files within, but it’s still essentially a ZIP file, and can be examined using the same tools that can be used on a ZIP. The next analysis step was to use the program 7-Zip [54] to examine the contents of the JAR files to see the classes and other files that are bundled within.

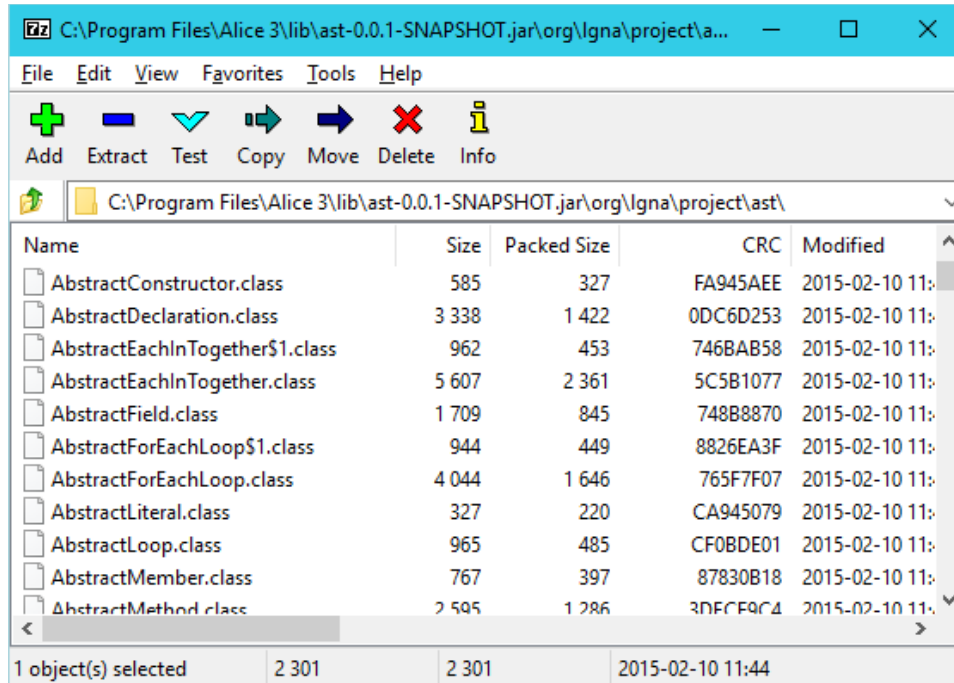


Figure 8: Folder view of “ast-0.0.1-SNAPSHOT.jar” opened in 7-Zip

By using 7-Zip to examine the “ast-0.0.1-SNAPSHOT.jar” file, as shown in Figure 8, there is further evidence that it contains classes which, based on their file names, appear to represent various aspects of the language syntax. These are exactly the kind of classes one would expect to see in an abstract syntax tree implementation, and strengthens the supposition that this JAR file contains functionality for an in-memory representation of the source code in an Alice project. The library also appears as though it may provide for an execution engine for evaluating the syntax tree as well.

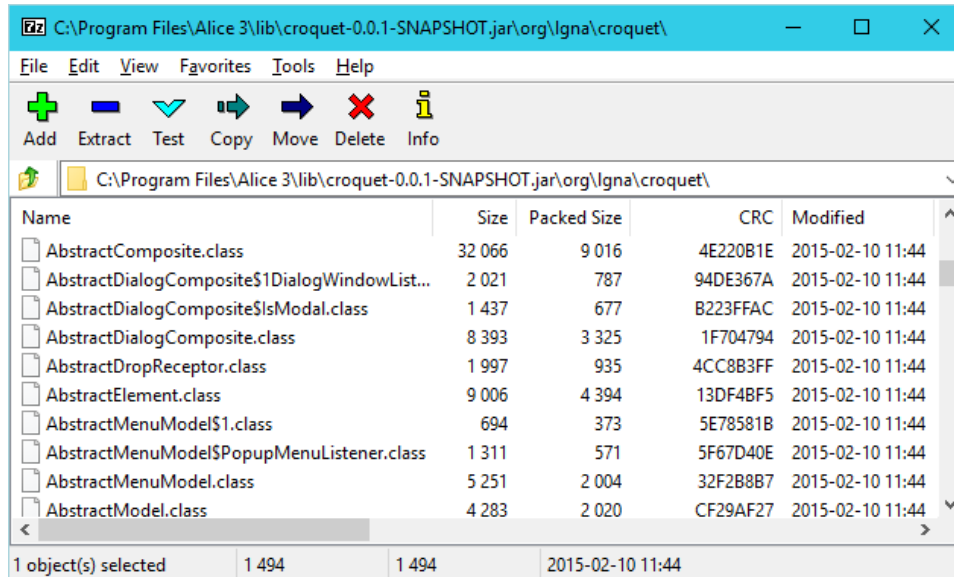


Figure 9: Zip listing of “croquet-0.0.1-SNAPSHOT.jar”

As shown in Figure 9, examination of the “croquet-0.0.1-SNAPSHOT.jar” library also reveals more of its secrets. In this library there appear to be classes related to various user interface elements, such as dialogs, menus, single and multi-select lists, and many others. While the listing of the classes don’t provide much insight into how these interface components are used, it nonetheless provides a much higher level of detail into what the croquet library is, and the inner workings of the Alice 3 user interface.

Using a tool such as 7-Zip to work with a JAR file’s contents not only exposes a great amount of detail, it can also be used to update a JAR file to include new file or directory entries, or to modify or remove existing files or directories. Such a technique is one of the options detailed in section 3.2.

3.2 Overriding and Enhancing Classes in Java

Section 3.1 illustrated that it is relatively easy to inspect the contents of the JAR files and gain access to the class and resource files contained therein. This provides a way to gain a fair amount of insight into the organization of the Alice 3 application that would be useful in modifying it to suit the project’s goals. In particular, it leads the way toward identifying classes which may be useful targets for modification to incorporate new functionality.

However, to take advantage of that knowledge, we must also determine how such modification can be best performed. The way in which the modifications are applied must be easy for students to do on their own, and must not adversely affect the standard Alice 3 experience. Three different methods for incorporating changes into the Alice 3 runtime environment were considered: Bytecode weaving, replacing classes in standard JARs, and providing an additional JAR containing the substitute class implementations.

Bytecode Weaving with AspectJ

One powerful method identified as a candidate for modifying the behavior of an existing Java class is to apply the concept of bytecode weaving. AspectJ [63] is an aspect-oriented programming [39] (AOP) language and compiler suite that can be used to apply bytecode weaving to an existing class. Two methods of bytecode weaving are supported by AspectJ:

- 1) Load time weaving [62]: Dynamically enhancing a class at runtime when the class is loaded.
- 2) Binary weaving [61]: Performing bytecode weaving to the binary contents of an existing class file, replacing the file with the modified version.

Applying bytecode weaving at load time requires either a Java agent to be loaded when starting the JVM or that a custom class loader is used by the application. Specifying an agent merely requires a new command line parameter be provided when launching Java (e.g. `-javaagent:path/to/aspectjweaver.jar`), and thus can be done without modifications to the Alice 3 classes. Likewise, the latter option of using a custom class loader is also possible, as long as the new class loader is registered before any of the Alice 3 classes are loaded. Such registration sequencing can be accomplished through tweaks to the Java command line arguments by specifying a custom classpath to `aspectjweaver.jar` and overriding the system class loader using the `"java.system.class.loader"` system property. The `"aj.bat"` script installed with AspectJ provides one example of how that can be achieved, though it would require modifications to support Alice 3.

While load time weaving does appear to be a possibility, it also imposes additional requirements and complexity to implementation and deployment that could be avoided.

Performing binary weaving of a class's bytecode removes the need to specify a custom agent or class loader, since the class's bytecode is modified prior to being loaded. However, regardless of the method used to implement bytecode weaving with AspectJ, any classes which are modified require access to the AspectJ runtime classes found in "aspectjrt.jar". Because of this dependency, the runtime must be provided in the classpath for the modified application to operate correctly.

In addition to requiring the availability of AspectJ runtime classes, several other factors led to bytecode weaving not being used in this project. AspectJ itself is a superset of the Java language that adds the concepts of aspects, which are composed of pointcuts and advices. These constructs provide a means to target specific points in a program and apply new actions that should be taken. While these can be powerful in the right context, they can also lead to code that may be difficult to maintain and understand. In particular, because aspects allow the execution of new actions where they were not previously performed, they can have an adverse impact when applied to an application that was not originally written with the knowledge that such a modification would be made. So, despite having the capability of incorporating the desired behavior into Alice 3, bytecode weaving was determined to incur too high a cost and was not the final method used.

Replacing a Class in an Existing JAR File

As mentioned in 3.1, a utility such as 7-Zip can be used to update a JAR file and replace files within it. Such a replacement could easily be used to incorporate one or more modified class files into existing Alice 3 JARs to change or extend the existing functionality. The approach itself is fairly straightforward and simple to achieve, and was used in the early phases of the project. However, it does have some drawbacks, rendering it unsuitable for long-term use.

One drawback is that any JAR that is modified to contain new classes would need to be redistributed. Since each JAR file typically contains many classes, and most of them may not require modification, it results in a larger dependency between the Alice 3 distribution and the output of this project. Redistribution and maintenance of such JAR is likely to incur

a higher cost, as it would be more tightly coupled with the specific version of Alice 3 that the original was obtained from.

It is possible that an existing JAR from an Alice 3 installation could be modified as part of the installation process for the customized classes. Since the original JAR would contain the unmodified classes, such an approach would permit distribution of only the classes which were modified. However, it would also require that the installation process be capable of modifying a JAR file, to either add classes or replace existing classes. A simpler method that doesn't require modification of any of the files distributed with Alice 3 is preferable.

Provide a new JAR

During the course of the project, it became clear that the simpler the method for incorporating new behavior was, the easier it would be for students to utilize the enhancements and for a maintainer to keep the project up-to-date with future Alice releases.

The third method for updating a class's behavior within Alice 3 accomplishes this by leveraging the way in which the classpath is utilized when classes are loaded. When loading a class for the first time, the standard Java class loader searches each individual component of the classpath in the order they are listed. The first class file found that matches the desired class's package and name will be loaded into memory and used from that point forward. Any class files in other locations within the classpath will be effectively ignored.

Since only the first matching class file is loaded, it is possible to "replace" an existing class within a Java application by ensuring that the location containing a replacement class file is injected into the classpath before the original. Thus, all that is necessary is to bundle all classes that provide new behavior into a single JAR file and modify the classpath that Alice 3 applies when it launches. Since this method limits the scope of the modifications needed within the Alice 3 installation to a much smaller footprint than the other two methods, it was the method chosen for implementation. While it does require updating the classpath, it doesn't require use of a non-standard compiler or set of runtime classes like

bytecode weaving does. It also does not require modification to existing Alice 3 JAR files, thus reducing concerns with distribution and installation.

With a way to incorporate new behavior, the next step in the process is to determine exactly which classes will need to be modified, and apply updates to them. To facilitate such modification, source code must first be derived from the binary class files, using a decompiler.

3.3 Decompilers Provide a Source

A Java decompiler is a program capable of reading a Java class file, interpreting the Java bytecode contained therein, and deriving a facsimile of the original source code that was compiled to create the bytecode. The source code generated by the decompiler should then be capable of being compiled with the Java compiler, once more generating bytecode equivalent to the original class file. Thus a decompiler works in almost the exact opposite manner of a compiler.

JD-GUI

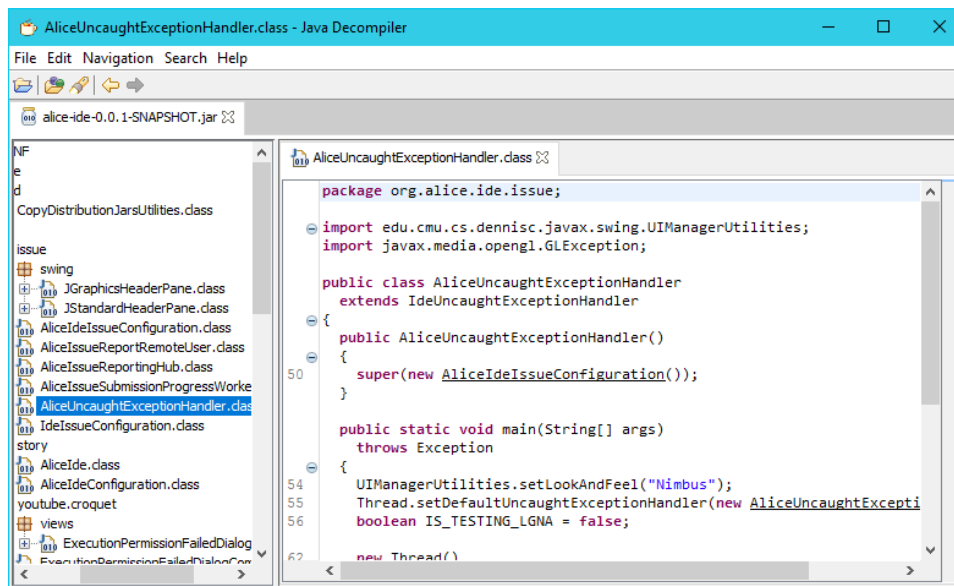


Figure 10: Examining Alice 3 classes with JD-GUI

One such tool, JD-GUI, is a freely available open source user interface to the closed source JD-Core Java decompilation library. JD-GUI provides a simple-to-use interface (see Figure 10) for examining the decompiled source of classes within a JAR file or directory tree. In addition to viewing decompiled source, JD-GUI also provides a simple method for saving the decompiled source for all classes it finds. These properties seemingly position JD-GUI as an ideal tool for deriving the Alice 3 sources from its class files. However, quite a few shortcomings became apparent after attempts to use it for that purpose. Details of several of these limitations are described in the following:

- Some bytecode could not be interpreted, resulting in error output in the generated source. Figure 11 contains an example from the class `org.lgna.project.virtualmachine.VirtualMachine`. Unfortunately, JD-GUI does not generate a report summary file detailing where such instances occur, so they are not apparent without searching through the generated source. Such unprocessed bytecode can go unnoticed when compiling, since only comments are output for these situations. The code in Figure 11 is also missing a close square bracket (]) in the array index for the “runnables” variable. It is unknown whether the missing bracket is a side-effect of the decompilation errors or a separate issue.

```

runnables[i = new Runnable() {
    /* Error */
    public void run() {
        // Byte code:
        // 0: aload_0
        // 1: getfield 16
        org/lgna/project/virtualmachine/VirtualMachine$3:this$0
        Lorg/lgna/project/virtualmachine/VirtualMachine;
        // 4: aload_0
        // 5: getfield 18
        org/lgna/project/virtualmachine/VirtualMachine$3:val$owner
        Lorg/lgna/project/virtualmachine/Frame;
        // 8: invokevirtual 30
        org/lgna/project/virtualmachine/VirtualMachine:pushCurrentThread
        (Lorg/lgna/project/virtualmachine/Frame;)V
        // 11: aload_0
        // 12: getfield 16
        org/lgna/project/virtualmachine/VirtualMachine$3:this$0
        Lorg/lgna/project/virtualmachine/VirtualMachine;
        ... More bytecode details follow, but were omitted for brevity ...
    }
}

```

Figure 11: Result when bytecode cannot be interpreted

- JD-GUI appears to have issues determining if a variable is already defined in the current scope, and often declares variables multiple times within the same scope. This can lead to a time-consuming process of removing the duplicate declarations to enable the decompiled source to be compiled. Figure 12 contains an example from the `org.lgna.project.ast.Element` class in which the “value” variable is declared multiple times within the same variable scope.

```

Object value;
Object value;
if (valueClsName.equals("")) {
    value = null;
} else {
    Class valueCls = ReflectionUtilities.getClassForName(valueClsName);
    if (valueCls.isArray()) {
        Object value;
        if (boolean[].class == valueCls) {
            value = binaryDecoder.decodeBooleanArray();
        } else {
            Object value;
            if (byte[].class == valueCls) {

```

Figure 12: Example of duplicate variable declarations

- Certain bytecode sequences result in the generation of improperly decoded for-each loops, as exhibited by the code from the org.lgna.project.ProgramTypeUtilities class shown in Figure 13. The outer loop is decoded improperly, while the first two inner loops appear correct. The third and final inner loop seems to have been intermingled and confused with the outer loop. Also note the invalid sequence “???” used in situations where a variable name would typically occur. A corrected version of the generated code is shown in Figure 14.

```

public static void sanityCheckAllTypes(Project project)
{
    for (
        Iterator localIterator1 =
            project.getNamedUserTypes().iterator();
        localIterator1.hasNext();
        ???.hasNext()
    ) {
        NamedUserType type = (NamedUserType)localIterator1.next();
        for (NamedUserConstructor constructor : type.constructors) {
            assert (constructor.getDeclaringType() == type) : type;
        }
        for (UserMethod method : type.methods) {
            assert (method.getDeclaringType() == type) : type;
        }
        ??? = type.fields.iterator();
        continue;
        UserField field = (UserField)???.next();
        assert (field.getDeclaringType() == type) : type;
    }
}

```

Figure 13: Example of an improperly decompiled for-each loop

```

public static void sanityCheckAllTypes(Project project) {
    for (NamedUserType type : project.getNamedUserTypes()) {
        for (NamedUserConstructor constructor : type.constructors) {
            assert (constructor.getDeclaringType() == type);
        }
        for (UserMethod method : type.methods) {
            assert (method.getDeclaringType() == type);
        }
        for (UserField field : type.fields) {
            assert (field.getDeclaringType() == type);
        }
    }
}

```

Figure 14: Corrected Figure 13 code with proper for-each loop usage

- In some instances, such as methods overloaded multiple times, some required typecasts are omitted. Figure 15 shows example code from the org.lgna.project.ast.AbstractType class illustrating a missing typecast. The first getDeclaredField method invokes a two-parameter overload of getDeclaredField with a null value for the first parameter. Since multiple overloaded methods of that name taking two parameters exist, and they only differ by the type of the first parameter, a compiler could not unambiguously resolve the reference without an explicit cast.

```

public F getDeclaredField(String name) {
    return getDeclaredField(null, name);
}

public F getDeclaredField(AbstractType<?, ?, ?> valueType, String name) {
    F rv = null;
    for (F field : getDeclaredFields()) {
        if (
            (field.getName().equals(name)) &&
            ((valueType == null) || (field.getValueType().equals(valueType)))
        ) {
            rv = field;
            break;
        }
    }
    return rv;
}

public F getDeclaredField(Class<?> valueCls, String name) {
    return getDeclaredField(JavaType.getInstance(valueCls), name);
}

```

Figure 15: Example of omission of required typecast

- When an anonymous class instantiation is performed as part of a return statement, JD-GUI omits the return keyword. This is illustrated in Figure 16, which contains an example from the code JD-GUI generates for the org.lgna.project.ast.JavaConstructor class. The typecast expression within the if statement is not a valid Java statement as defined in section 14.8 of the JLS. It is clear from the remaining code that the method should return the result of the expression, and the return keyword should have been specified before the typecast.

```

public static JavaConstructor getInstance(
    ConstructorReflectionProxy constructorReflectionProxy
) {
    if (constructorReflectionProxy != null) {
        (JavaConstructor)mapReflectionProxyToInstance
            .getInitializingIfAbsent(
                constructorReflectionProxy,
                new InitializingIfAbsentMap.Initializer() {
                    public JavaConstructor initialize(
                        ConstructorReflectionProxy key
                    ) {
                        return new JavaConstructor(key, null);
                    }
                }
            );
    }
    return null;
}

```

Figure 16: Example of missing return keyword

- In some cases where a static property declaration includes an initializer, the initialization code will be missing. As seen in the Figure 17 code sample from the `org.lgna.project.ast.JavaConstructor` class, the static property declaration for `mapReflectionProxyToInstance` is incomplete since it is missing an expression for computing the initial value following the equal sign. The correct declaration should be assigned to the result of `Maps.newInitializingIfAbsentHashMap()`.

```

public class JavaConstructor
    extends AbstractConstructor
{
    private static final
        InitializingIfAbsentMap<ConstructorReflectionProxy, JavaConstructor>
        mapReflectionProxyToInstance = ;
}

```

Figure 17: Example of invalid initialization of static properties

- Certain assertions are not decoded properly, as illustrated by Figure 18 from the class `org.lgna.project.reflect.ClassInfo`. Instead of generating code using the `assert` keyword, JD-GUI outputs an `if` statement referencing a synthetic class property named `$assertionsDisabled` that was generated by the Java compiler when the original source code was compiled. Since the property is synthetic and not defined within the code, the Java compiler will emit an undefined variable reference error

for the decompiled code. A correct interpretation of the assert is shown in Figure 19.

```
this.cls = Class.forName(this.clsName);  
  
if ((!$assertionsDisabled) && (this.cls == null)) {  
    throw new AssertionError(this.clsName);  
}
```

Figure 18: Example of incorrect assertion decompilation

```
this.cls = Class.forName(this.clsName);  
  
assert (this.cls != null) : this.clsName;
```

Figure 19: Code from Figure 18 with corrected assertion

- If a `java.lang.Enum` subclass is encountered with abstract methods, each instance of the subclass (the enumeration values) must implement all of the abstract methods. With JD-GUI, such abstract enumerations are not decompiled properly. This is shown in the example code of Figure 20, extracted from the class `org.lgna.project.ast.ArithmeticInfixExpression`. The decompiled code declares the `Operator` enum as abstract, which is invalid per section 8.9 of the Java Language Specification (JLS) [52] which states that enumerations are implicitly final. Section 8.9.2 of the JLS also requires that enumeration members implement the abstract methods declared in the enumeration definition, yet the example code contains no such implementations. Figure 21 illustrates how the code should have been decompiled. The `abstract` keyword is no longer used in the enum definition and each of the individual members provide definitions for the abstract methods declared by the enumeration subclass.


```

public static abstract enum Operator {
    PLUS,
    MINUS,
    TIMES,
    REAL_DIVIDE,
    INTEGER_DIVIDE,
    REAL_REMAINDER,
    INTEGER_REMAINDER;

    public abstract Number operate(Number n1, Number n2);
    abstract void appendJava(JavaCodeGenerator generator);
}

```

Figure 20: Example of incorrectly decompiled enum with abstract methods

```

public static enum Operator {
    PLUS {
        @Override
        public Number operate(Number leftOperand, Number rightOperand) {
            assert (leftOperand != null) : this;
            assert (rightOperand != null) : this;
            // ... Implementation of the method here ...
        }

        @Override
        void appendJava(JavaCodeGenerator generator) {
            generator.appendChar('+');
        }
    },
    MINUS {
        // ... Implementation of abstract methods here ...
    },
    // And so on for and TIMES, REAL_DIVIDE, INTEGER_DIVIDE,
    // REAL_REMAINDER, and INTEGER_REMAINDER...
};

public abstract Number operate(Number n1, Number n2);
abstract void appendJava(JavaCodeGenerator generator);
}

```

Figure 21: Corrected decompilation of the code from Figure 20

- When decompiling a for loop with an unknown control variable name, JD-GUI often uses the generic variable name “i”, even if a variable of the same name is already declared in the same scope. This prevents compilation of such code due to the redefinition, and in some cases may also result in undesired updates of the original variable.

- JD-GUI will generate import statements in the same source file for multiple inner classes that have the same class name. Multiple imports for classes with the same name are not permitted by the Java compiler, since such code would cause the imported classes to be ambiguous. An import for the enclosing class should be used instead. In the cases where such import statements are generated by JD-GUI, it appears they can be safely removed, since the actual references to the inner classes are qualified with the enclosing class's name.

CFR (Class File Reader)

Since so many deficiencies were identified with the source code generated by JD-GUI, alternative decompilers were sought as alternatives. Despite there being several Java decompiler projects in existence, only a small number of them have been maintained over the years with support for newer versions of Java. One of the most promising of those projects is the CFR [4] (short for Class File Reader) project by Lee Benfield. However, like JD-GUI, several issues were encountered using CFR as well, some of which are detailed below:

- Some bytecode sequences could not be decompiled by CFR. This is illustrated in the code from the `org.lgna.project.ast.StaticAnalysisUtilities` class shown in Figure 22.

```

private static String getConventionalIdentifierName(
    String name, boolean cap
) {
    rv = "";
    isAlphaEncountered = false;
    N = name.length();
    i = 0;
    while (i < N) {
        c = name.charAt(i);
        if (!Character.isLetterOrDigit(c)) ** GOTO lbl20
        if (!Character.isDigit(c)) ** GOTO lbl14
        if (isAlphaEncountered) ** GOTO lbl15
        rv = String.valueOf(rv) + "_";
        rv = String.valueOf(rv) + c;
        isAlphaEncountered = true;
        ** GOTO lbl21
lbl14: // 1 sources:
        isAlphaEncountered = true;
lbl15: // 2 sources:
        if (cap) {
            c = Character.toUpperCase(c);
        }
        rv = String.valueOf(rv) + c;
        cap = Character.isDigit(c);
        ** GOTO lbl21
lbl20: // 1 sources:
        cap = true;
lbl21: // 3 sources:
        ++i;
    }
    return rv;
}

```

Figure 22: CFR output for a method it could not decompile

- Some code generated by CFR appears unnecessarily complex or unintuitive. This may be a result of potentially literal translation of certain compiler optimizations, as the example from the `org.lgna.project.ast.AbstractType` class in Figure 23 shows. A few items appear to be suspicious with this translation, including: use of a mix of `||` and `&&` in the same logical expression without explicit grouping, multiple logical negations in an expression that could be expressed more succinctly if negated, and use of `break` outside an `if` statement. Figure 24 lists the more readable version of the same code produced by JD-GUI. Compared to code from Figure 23, the code shows a more natural grouping and use of logical operations and the `break` is used inside an `if` statement.

```

for (F field : getDeclaredFields()) {
    if (
        !field.getName().equals(name) ||
        valueType != null &&
        !field.getValueType().equals(valueType)
    ) {
        continue;
    }
    rv = field;
    break;
}

```

Figure 23: Example of unintuitive code generated by CFR

```

for (AbstractField field : this.getDeclaredFields()) {
    if (
        (field.getName().equals(name)) &&
        ((valueType == null) || (field.getValueType().equals(valueType)))
    ) {
        rv = field;
        break;
    }
}

```

Figure 24: JD-GUID output for same for-each loop illustrated in Figure 23

- CFR also adds extraneous imports for classes from the decompiled class's package. Since the Java compiler automatically imports classes from the same package as the class being compiled, such imports are superfluous and simply clutter the code.

3.4 Compiling the Generated Sources

After decompiling all classes in a JAR, the resulting source files should be able to be compiled back to class files with the same functionality. The compiled class files and other associated resource files from the original JAR can then be repackaged into a new JAR file that can be used as a drop-in replacement for the original. However, that task is easier stated than accomplished.

Determining the Compilation Class Path

Most, if not all, of the Alice 3 JAR files contain classes that are interdependent with other classes from different JAR files. These dependencies may be with classes from an external 3rd party library or from an internal Alice library. To properly compile a source file, those

other classes must be available to the Java compiler. This is accomplished by providing the compiler with a class path – a listing of the directories and JAR files that contain the classes the compiler should be aware of when compiling. Determining the correct class path to use for compiling the decompiled sources required more detective work.

For version 3.1 of Alice, this class path could be found in a file named “Alice.ini” found in the root Alice 3 program folder. This file contained multiple entries of the form “classpath.1=some path” with the number following “classpath.” being incremented for each entry, and “some path” referring to a path or JAR to add to the classpath. Each entry could be extracted from the file and combined manually to create a class path for the Java compiler.

With the release of Alice 3.2, the application directory structure was reorganized, a new installer, install4j [28], was used, and the class path was no longer exposed in such an obvious manner. Along with using install4j, its associated launcher exe4j was used to generate a native executable for launching Alice. This new launcher embeds the class path into the executable itself, requiring it to be manually extracted. Luckily, the class path is embedded in clear text, and can be manually identified and extracted using a text editor.

An alternative method to determine a quick and dirty class path is to simply pass the paths of all of the JAR files contained with the “ext” and “lib” subdirectories in the Alice application folder. While this method will not guarantee the JAR files will be loaded in the same order as is used when launching Alice 3 normally, it should not be a problem unless more than one of the loaded JAR files define classes with the same package and name. Since that is uncommon, and does not occur within the libraries Alice 3 utilizes, it is a generally safe method to employ. Examples of how to construct such a class path are illustrated in Figure 25 as a Windows batch file and as a Bash shell script in Figure 26.

```

SETLOCAL EnableDelayedExpansion
SET ALICE_DIR="C:\Program Files\Alice 3"

SET CLASSPATH=

FOR /f "tokens=*" %%F in ('dir /b /s %ALICE_DIR%\ext\*.jar') DO (
    SET CLASSPATH=!CLASSPATH!;%%F
)
FOR /f "tokens=*" %%F in ('dir /b /s %ALICE_DIR%\lib\*.jar') DO (
    SET CLASSPATH=!CLASSPATH!;%%F
)

SET CLASSPATH=%CLASSPATH:~1%

```

Figure 25: Windows batch syntax for deriving class path

```

# Set the following as appropriate for the target operating system
# This example uses the default path for a Linux installation
ALICE_DIR=~/.Alice3

CLASSPATH=
for FILE in `find "$ALICE_DIR"/{ext,lib} -name "*.jar"`; do
    CLASSPATH=$CLASSPATH:$FILE
done
CLASSPATH=${CLASSPATH:1}

```

Figure 26: Bash script syntax for deriving class path

File Organization and Script for Compilation

After decompiling with JD-GUI, all of the decompiled source files were placed in a “src” subdirectory, retaining the original package directory hierarchy. A “resources” directory was also created, and populated with all non-class files from the original JAR. The resource files include any image, license, configuration or other files originally packaged in the JAR that did not have the “.class” extension, and which were not placed under the top-level META-INF folder.

After this directory structure was created, a script was derived to simplify invoking the Java compiler. Since this portion of the project was performed using a Windows PC, a batch script was created for this task. The script first cleans any existing output class files from a previous run, then constructs a listing of source files by finding all Java files under “src”. Next, it builds a class path from the listing of JAR files within the “lib” and “ext” subfolders in the Alice program directory. The discovered source files are then compiled by “javac” using the derived class path, with the compiled classes placed into a subfolder

named “_classes”. Finally, the resource files found in the “resources” folder are copied into the “_classes” folder, and the result is then assembled into a JAR file by invoking the “jar” command.

```
@ECHO OFF
SETLOCAL EnableDelayedExpansion
SET ALICE_DIR="C:\Program Files\Alice 3"
SET JAR_NAME=ast-0.0.1-SNAPSHOT

%~d0
cd %~dp0

IF EXIST _classes rmdir /S/Q _classes
mkdir _classes

IF EXIST source_files rm source_files
dir /s /b src\*.java > source_files

SET CLASSPATH=
FOR /f "tokens=*" %%F in ('dir /b /s %ALICE_DIR%\ext\*.jar') DO (
    SET CLASSPATH=!CLASSPATH!;%%F
)
FOR /f "tokens=*" %%F in ('dir /b /s %ALICE_DIR%\lib\*.jar') DO (
    SET CLASSPATH=!CLASSPATH!;%%F
)
SET CLASSPATH=%CLASSPATH:~1%

javac -d _classes ^
    -encoding UTF-8 ^
    -sourcepath src ^
    @source_files 1>javac-output.txt 2>&1

type javac-output.txt

xcopy resources\* _classes /E /Q /Y

jar cf %JAR_NAME%.jar -C _classes .
```

Figure 27: Windows batch file used to compile generated source files

Effectiveness of Decompilers for Integrating Finch Support

By utilizing a decompiler such as JD-GUI or CFR, a great deal of insight can be gained about the structure and organization of the classes and code within a Java program such as Alice 3. However, since such decompilers are often unable to properly decompile certain bytecode sequences, they are of somewhat limited use. While it would in theory be possible to decompile all of Alice 3’s internal library JARs, such decompilation would require quite a large effort.

In the course of the project, the single Alice 3 JAR file named “ast-0.0.1-SNAPSHOT.jar” was decompiled using JD-GUI, and all of the syntax issues introduced by the decompiler were corrected. Additionally, all of the methods which could not be decompiled by JD-GUI were decompiled using CFR and re-integrated into the sources generated by JD-GUI. In the end, all of the issues preventing compilation within the selected library were found to be correctable, a new JAR was created from the generated classes, and it was found to work as a replacement for the original library within Alice 3.

The result of the experiment to create a JAR from the modified decompiled files served as a proof of concept. However, there are more considerations that need to be made. For example, what would the time investment be to completely recompile Alice 3, and is it necessary to do so in order to accomplish the goals of the project? Given that the chosen library was one of the smaller libraries in Alice 3, containing only 207 classes, and the fact that it required nearly five hours to successfully resolve all of the compilation issues within that one library, it was projected that it would take at least an additional 150 hours to successfully decompile, correct and rebuild all 4,666 of the internal Alice classes. In addition, it is not known whether there would be more decompilation problems encountered with the other libraries which cannot be so easily resolved.

Luckily, it is not necessary to decompile all of Alice to integrate new functionality into it. It would suffice to update a small number of classes, perhaps as few as a single class, as long as the modified classes can be successfully loaded into Alice along with the rest of the original classes and the modifications result in the exposure of Finch functionality within the Alice 3 interface. However, before any classes can be modified it must first be determined which classes would be the most appropriate to target. One method to determine the appropriate classes is through direct analysis of the source code generated through decompilation. Another method is to utilize a different type of utility that allows one to examine the internal workings of a Java GUI application while it is executing.

3.5 It's a Swinging Interface

The next step in the reverse engineering process was to examine the structure of the actual user interface and how its components are composed together. The Alice 3 GUI is constructed using the Java Swing framework, a fact which is advantageous to determining its inner workings. Several tools exist to facilitate debugging and examining the logical component hierarchy within Swing GUI programs. The tool used for that purpose during the course of this project is Swing Explorer [44,70], which provides a simple to use graphical interface of its own (illustrated in Figure 28).

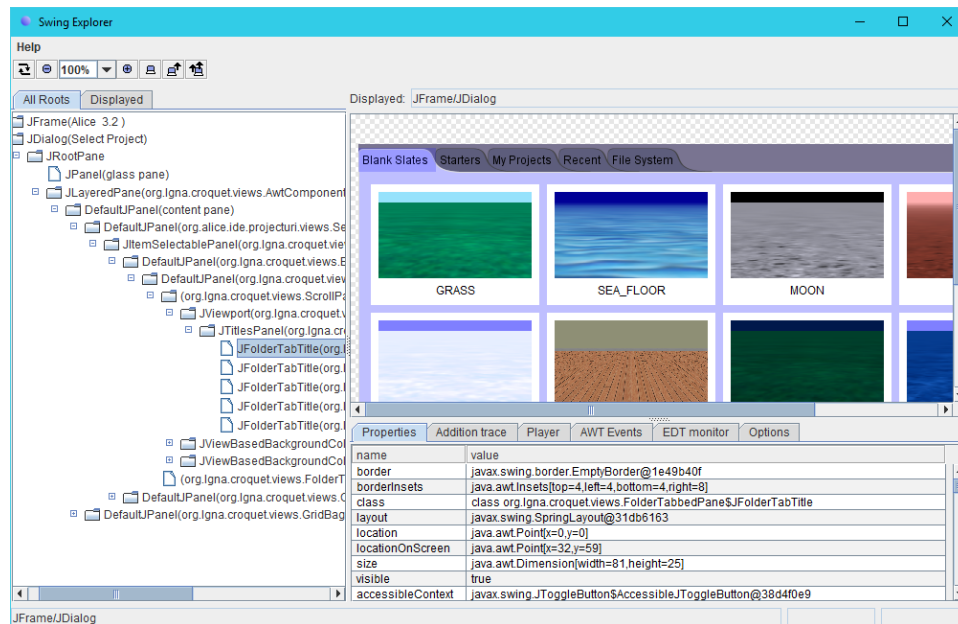


Figure 28: The Swing Explorer interface

Launching Alice 3 with Swing Explorer

Swing Explorer is a tool for inspecting the internal structure of a Swing-based GUI application. However, there are a few caveats when using Swing Explorer to examine the Alice 3 application. Swing Explorer works by utilizing a custom Java agent, which must be loaded by providing the path to the agent JAR file using the `javaagent` command line argument and adding the path to the agent JAR to the bootclasspath, when launching the `java` executable to inspect. In addition to specifying the agent, Swing Explorer also requires

an interface JAR be added to the classpath and that the class `org.swingexplorer.Launcher` be specified as the main class to execute. The main class of the application to be inspected must be provided as the first argument passed to the Launcher class, along with any other parameters that would typically be provided to the original main class.

All of the above conditions are usually accomplished by specifying the appropriate command line arguments to the java executable distributed with the Java Runtime Environment (JRE). However, since Alice 3 is launched by a separate native executable launcher, and not using the java executable, the proper arguments to provide to the java executable to launch Alice without the custom launcher are not immediately obvious. As mentioned previously in section 3.4, the Alice 3 launcher executable contains information required to launch Alice, such as the classpath. The executable also contains the name of the main class and command line arguments passed when the launcher starts Alice 3. That information can be found in close proximity to the classpath information, and is most easily located by opening the launcher executable in a text editor and searching for the word “alice”.

```

C:\Program Files\Alice 3\Alice 3.exe - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Alice 3.exe
r;A1.\lib\ide-0.0.1-SNAPSHOT.jar;A1.\lib\image-editor-0.0.1-SNAPSHOT.jar;A1.\lib\
issue-reporting-0.0.1-SNAPSHOT.jar;A1.\lib\scenegraph-0.0.1-SNAPSHOT.jar;A1.\lib\
story-api-0.0.1-SNAPSHOT.jar;A1.\lib\story-api-migration-0.0.1-SNAPSHOT.jar;A1.\l
ib\util-0.0.1-SNAPSHOT.jar;z[REDACTED]org.alice.stageide.EntryPoint{
[REDACTED]>[REDACTED]-ea -Xmx1024m -Dorg.alice.ide.rootDirectory=.
-Dswing.aatext=true -Djogamp.gluegen.UseTempJarCache=false
"-Dinstall4j.launcherId=24"
-Dinstall4j.swt=false|[REDACTED]; [REDACTED]SOHNUL[REDACTED]1~[REDACTED]
[REDACTED]1.6|[REDACTED]e[REDACTED]RS[REDACTED]B.\jre;Y;EJAVA
HOME;EJDK_HOME;[REDACTED]4ix9kwt4oh4lwy%86dc,[REDACTED]SOHNUL[REDACTED]
Normal text file length: 988160 lines: 1934 Ln: 1917 Col: 3123 Sel: 191 | 1 Macintosh ANSI INS
  
```

Figure 29: Java arguments embedded in Alice 3 native launcher executable

Figure 29 shows the embedded java command line arguments (selected text) from the native Windows Search launcher executable for Alice 3. Several arguments for the java executable are used, as detailed in Table 3.

Parameter Value	Purpose
-ea	Enables assertions
-Xmx1024m	Specifies the maximum size, in bytes, of the memory allocation pool.
-Xorg.alice.ide.rootDirectory=.	Instructs the Alice IDE where to locate files.
-Dswing.aatext=true	An old and undocumented method of enabling anti-aliased font rendering in JDK 1.5 and earlier. See https://bugs.openjdk.java.net/browse/JDK-6391267
-Djogamp.gluegen.UseTempJarCache=false	Disables automated native library loading behavior in JOGL (A Java OpenGL API binding). See http://jogamp.org/jogl/doc/userguide/#automatednativelibraryloading
-Dinstall4j.launcherId=24	Launcher Id value used by install4j.
-Dinstall4j.swt=false	Instructs install4j that Alice 3 does not use the Standard Widget Toolkit (SWT) for its GUI.
org.alice.stageide.EntryPoint	The main class that launches the Alice 3 IDE.

Table 3: Java arguments extracted from the Alice 3 launcher exe

Based on the information extracted from the Alice 3 launcher, a batch file was constructed to simplify inspection using the Swing Explorer. Figure 30 provides a listing of the batch file. To operate correctly, the Swing Explorer agent and GUI JAR files (swag.jar and swexpl.jar, respectively) must be present in the same parent folder as the batch file itself. The classpath to use when launching Alice 3 is determined by scanning the “ext” and “lib” subfolders of the Alice 3 installation folder for JAR files, as described in section 3.4 above.

```

@ECHO OFF
SET ALICE_DIR=C:\Program Files\Alice 3

SETLOCAL EnableDelayedExpansion
SET BASE=%~dp0

%ALICE_DIR:~0,2%
cd "%ALICE_DIR%"

SET CLASSPATH=%BASE%\swexpl.jar
FOR /f "tokens=*" %%F in ('dir /b /s .\ext\*.jar') DO (
    SET CLASSPATH=!CLASSPATH!;%%F
)
FOR /f "tokens=*" %%F in ('dir /b /s .\lib\*.jar') DO (
    SET CLASSPATH=!CLASSPATH!;%%F
)

@ECHO ON
java -javaagent:%BASE%\swag.jar -Xbootclasspath/a:%BASE%\swag.jar ^
    -ea -Xmx1024m -Dswing.aatext=true ^
    -Dorg.alice.ide.rootDirectory=./ ^
    -Djogamp.gluegen.UseTempJarCache=false ^
    "-Dinstall4j.launcherId=24" -Dinstall4j.swt=false ^
    org.swingexplorer.Launcher org.alice.stageide.EntryPoint %1

```

Figure 30: Batch file used for launching Alice 3.3 under Swing Explorer

Runtime Inspection of the Alice 3 GUI

With the proper command line arguments determined and a simple method to launch Alice 3 to be inspected with Swing Explorer, it is possible to inspect the internal GUI. The primary goal was to determine how the GUI elements were structured for the procedures and methods exposed for the various objects within Alice. To accomplish that task, Alice was launched under Swing Explorer, a new Alice project based on the “Grass” theme was created, and a “SportsCar” object was added to the scene. Within the Swing Explorer window, the entry titled “JFrame (Alice 3.2)” was selected for display from the hierarchy pane. Then the procedure entry GUI element for the SportsCar’s “say” method was selected in the Swing Explorer display frame (see screenshot in Figure 31 and relevant portion of view hierarchy in Figure 32).

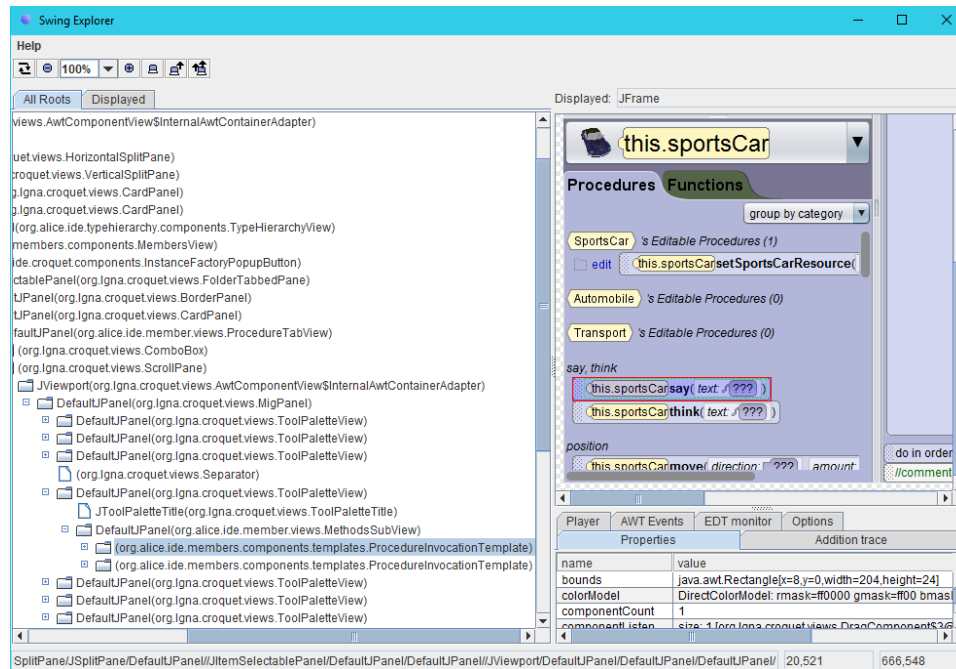


Figure 31: Swing Explorer with SportsCar say method selected

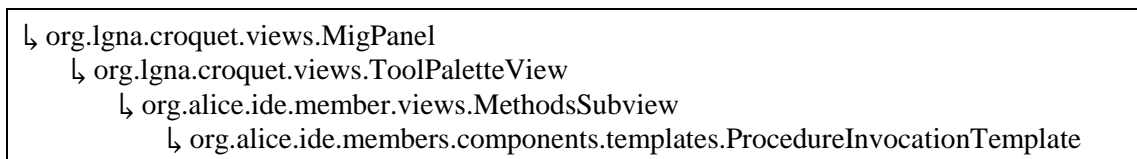


Figure 32: Subtree of Swing view hierarchy for procedure elements

By examining the tree displayed within Swing Explorer’s view hierarchy pane, the classes which implement the various composed views become evident. Given those class names, it was a simple matter to locate the implementations within the various Alice 3 JAR files. The package names of the classes provide a good indication of which JARs to inspect. For example, the JAR file named “croquet-0.0.1-SNAPSHOT.jar” is a very likely place to look for classes under the org.lgna.croquet package, and “ide-0.0.1-SNAPSHOT.jar” is likely to contain the classes under the org.alice.ide package. In both cases, these initial guesses proved correct.

Once the Swing view classes were located, the JAR files containing them could be decompiled, and the resulting source files inspected to determine how the individual procedure entries were populated. The details of that task are provided in chapter 4.

4. Enhancing Alice

Chapter 3 detailed methods for examining the package structure and classes within each JAR, deriving decompiled source for the classes within those JARs, and directly determining which classes are utilized within the Alice 3 GUI to present the lists of procedures and functions available. This chapter will expand upon those findings, with the first task being to determine how new procedures and functions can be incorporated to allow interaction with a Finch robot.

4.1 Exposing New Procedures and Functions

Alice 3 only allows user programs to invoke methods on certain types of objects. The types that are exposed for invocation are determined by calling a method of the `org.alice.stageide.StoryApiConfigurationManager` class named “`isInstanceFactoryDesiredForType`”, listed in Figure 33. In general, the exposed types include the Scene object itself, the single Program instance for the project, or an object which is placed within the 3D scene such as the Camera, a character, a vehicle, a prop, or a joint property of an object. More specifically, only classes that inherit from `org.lgna.story.SProgram` or `org.lgna.story.SThing` (but not also `org.lgna.story.SMarker`) are considered.

```
public boolean isInstanceFactoryDesiredForType(AbstractType<?, ?, ?> type) {
    if (type.isAssignableTo(SThing.class)) {
        if (type.isAssignableTo(SMarker.class)) {
            return false;
        }
        return true;
    }
    if (type.isAssignableTo(SProgram.class)) {
        return true;
    }
    return false;
}
```

Figure 33: Source of `isInstanceFactoryDesiredForType` from `StoryApiConfigurationManager`

The exposed objects have associated methods that can be invoked from a user's program to trigger a behavior of the object or retrieve information about the object. Several built-in methods are predefined for each object type, and user-defined methods can also be defined as a part of an Alice 3 program.

There are two classifications of methods in Alice 3. Methods which do not generate a return value are called *procedures*. Calling a procedure will generally result in changing the object's state, such as altering its position, orientation or size in the scene. When a method returns a value, it is called a *function*. Most Functions are typically reserved for retrieving information describing the object's state, and are nearly always side-effect free and do not cause any change to the state itself.

As described in 3.5, the structure of the Alice 3 GUI was examined using Swing Explorer to identify the classes that implement the representation of those procedures and functions. That examination revealed that the `org.alice.ide.members.components.templates.ProcedureInvocationTemplate` and `org.alice.ide.member.views.MethodsSubview` classes may be good starting points.

```
package org.alice.ide.member.views;

import edu.cmu.cs.dennisc.java.awt.font.TextAttribute;
import org.alice.ide.declarationseditor.DeclarationTabState;
import org.alice.ide.member.MethodsSubComposite;
import org.alice.ide.members.components.templates.TemplateFactory;
import org.lgna.croquet.views.AwtComponentView;
import org.lgna.croquet.views.BoxUtilities;
import org.lgna.croquet.views.LineAxisPanel;
import org.lgna.croquet.views.PageAxisPanel;
import org.lgna.project.ast.UserMethod;

public class MethodsSubView<C extends MethodsSubComposite>
    extends PageAxisPanel
{
    public MethodsSubView(MethodsSubComposite composite) {
        super(composite, new AwtComponentView[0]);
        this.setMaximumSizeClampedToPreferredSize(true);
        this.setBorder(javax.swing.BorderFactory.createEmptyBorder(0, 8, 12, 0));
    }

    public C getComposite() {
        return (C)super.getComposite();
    }
}
```

```

protected void internalRefresh() {
    super.internalRefresh();
    MethodsSubComposite composite = this.getComposite();
    this.removeAllComponents();
    composite.updateTabTitle();
    for (org.lgna.project.ast.AbstractMethod method : composite.getMethods()) {
        org.lgna.croquet.views.SwingComponentView component;
        org.lgna.croquet.views.DragComponent dragComponent =
            TemplateFactory.getMethodInvocationTemplate(method);
        if (method instanceof UserMethod) {
            UserMethod userMethod = (UserMethod)method;
            DeclarationTabState tabState =
                org.alice.ide.IDE.getActiveInstance().getDocumentFrame().
                    getDeclarationsEditorComposite().getTabState();
            org.lgna.croquet.Operation operation =
                tabState.getItemSelectionOperationForMethod(method);
            org.lgna.croquet.views.Hyperlink hyperlink =
                operation.createHyperlink(new TextAttribute[0]);
            hyperlink.setClobberText("edit");
            component = new LineAxisPanel(
                new AwtComponentView[] {
                    hyperlink,
                    BoxUtilities.createHorizontalSliver((int)8),
                    dragComponent
                }
            );
        } else {
            component = dragComponent;
        }
        this.addComponent(component);
    }
}

```

Figure 34: Decompiled source of the org.alice.ide.member.views.MethodsSubView class

The decompiled source for the MethodsSubview class, shown in Figure 34, contains a likely method named “internalRefresh” which appears to retrieve the available method list and populate the view with template instances for each match. The “internalRefresh” method retrieves the list by invoking “getMethods” on the org.alice.ide.member.MethodsSubComposite value originally provided to the constructor when the MethodsSubview instance was created.

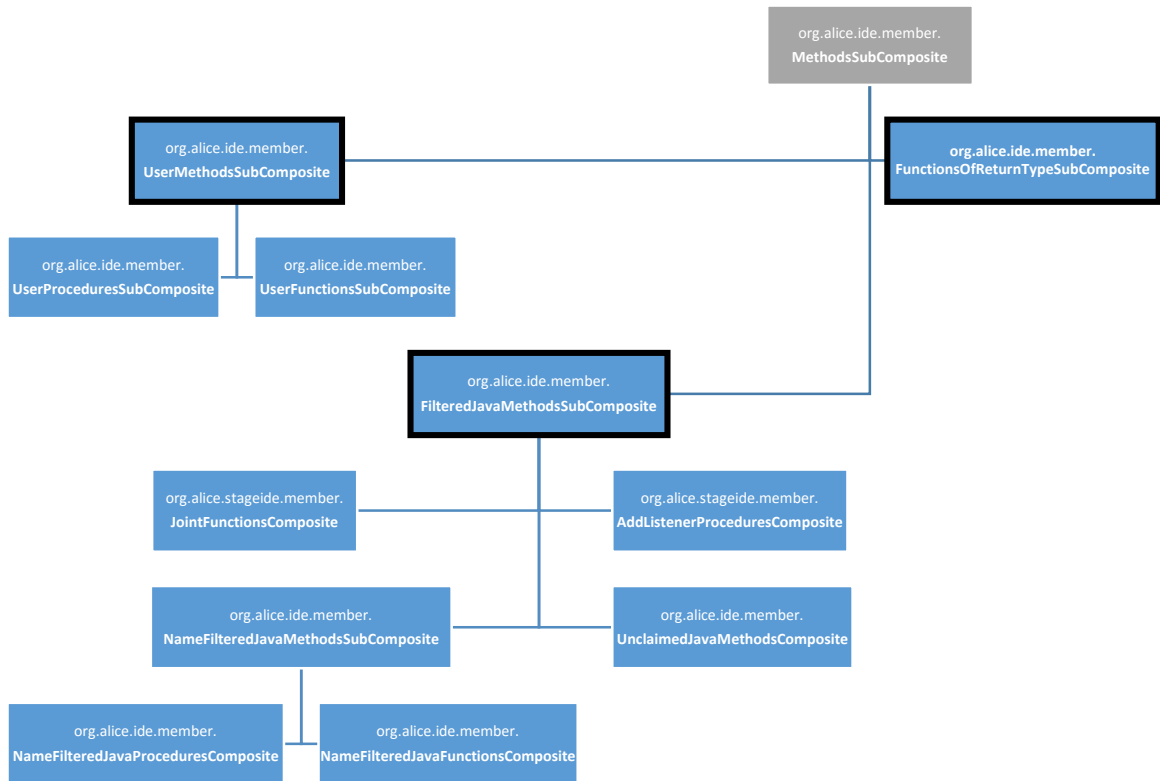


Figure 35: org.alice.ide.member.MethodsSubComposite subclasses in Alice 3

Figure 35 lists the MethodsSubComposite subclasses identified within Alice 3. Of particular interest are the classes outlined with a black border, as it is only those three which contain concrete implementations of “getMethods”. The first class, org.alice.ide.member.UserMethodsSubComposite, appears to be specifically related to user-defined methods, which limits its relevance, since any methods added for Finch functionality should be exposed as pre-defined methods. The implementation of “getMethods” in the second class, org.alice.ide.member.FunctionsOfReturnTypeSubComposite, merely returns a list which was set previously by calling “setMethods”. The third class, org.alice.ide.member.FilteredJavaMethodsSubComposite, uses a similar implementation, though with the returned list being set by a call to “sortAndSetMethods”.

A quick search through the code exposed only a single call to “setMethods”, invoked from the “getByReturnTypeSubComposites” method from the org.alice.ide.member.FunctionTabComposite class. Performing a similar search for “sortAndSetMethods”

revealed three calls, all from the “getSubComposites” method of org.alice.ide.member.MemberTabComposite. Since inspection of the FunctionTabComposite class showed it to be a subclass of MemberTabComposite, it was decided to focus first on the MemberTabComposite base class and its “getSubComposites” method, the code of which is listed in Figure 36.

```

public List<MethodsSubComposite> getSubComposites() {
    LinkedList<MethodsSubComposite> rv = Lists.newLinkedList();
    LinkedList<JavaMethod> javaMethods = Lists.newLinkedList();

    InstanceFactory instanceFactory =
        IDE.getActiveInstance().getDocumentFrame().
            getInstanceFactoryState().getValue();
    if (instanceFactory != null) {
        AbstractType type = instanceFactory.getValueType();
        while (type != null) {
            if (type instanceof NamedUserType) {
                NamedUserType namedUserType = (NamedUserType)type;
                UserMethodsSubComposite userMethodsSubComposite =
                    this.getUserMethodsSubComposite(namedUserType);
                rv.add(userMethodsSubComposite);
            } else if (type instanceof JavaType) {
                JavaType javaType = (JavaType)type;
                for (JavaMethod javaMethod : javaType.getDeclaredMethods()) {
                    if (
                        !this.isAcceptable(javaMethod) ||
                        !MemberTabComposite.isInclusionDesired(javaMethod)
                    ) continue;
                    javaMethods.add(javaMethod);
                }
            }
            if (!type.isFollowToSuperClassDesired()) break;
            type = type.getSuperType();
        }
    }
    if (rv.size() > 0) {
        rv.add(SEPARATOR);
    }
}

```

```

if (!"sort alphabetically".equals(this.getSortState().getValue())) {
    for (
        FilteredJavaMethodsSubComposite potentialSubComposite :
        this.getPotentialCategorySubComposites()
    ) {
        LinkedList<JavaMethod> acceptedMethods = Lists.newLinkedList();
        ListIterator<JavaMethod> methodIterator =
            javaMethods.listIterator();
        while (methodIterator.hasNext()) {
            JavaMethod method = methodIterator.next();
            if (!potentialSubComposite.isAcceptingOf(method)) continue;
            acceptedMethods.add(method);
            methodIterator.remove();
        }
        if (acceptedMethods.size() <= 0) continue;
        potentialSubComposite.sortAndSetMethods(acceptedMethods);
        rv.add(potentialSubComposite);
    }
}

LinkedList<MethodsSubComposite> postSubComposites =
    Lists.newLinkedList();
for (
    FilteredJavaMethodsSubComposite potentialSubComposite :
    this.getPotentialCategoryOrAlphabeticalSubComposites()
) {
    LinkedList<JavaMethod> acceptedMethods = Lists.newLinkedList();
    ListIterator<JavaMethod> methodIterator = javaMethods.listIterator();
    while (methodIterator.hasNext()) {
        JavaMethod method = methodIterator.next();
        if (!potentialSubComposite.isAcceptingOf(method)) continue;
        acceptedMethods.add(method);
        methodIterator.remove();
    }
    if (acceptedMethods.size() <= 0) continue;
    potentialSubComposite.sortAndSetMethods(acceptedMethods);
    postSubComposites.add(potentialSubComposite);
}

if (javaMethods.size() > 0) {
    UnclaimedJavaMethodsComposite unclaimedJavaMethodsComposite =
        this.getUnclaimedJavaMethodsComposite();
    unclaimedJavaMethodsComposite.sortAndSetMethods(javaMethods);
    rv.add(unclaimedJavaMethodsComposite);
}
rv.addAll(postSubComposites);
return rv;
}

```

Figure 36: Decompiled org.alice.ide.member.MemberTabComposite’s getSubComposites method

The “getSubComposites” method builds a list of MethodsSubComposite instances through four separate phases. The first phase determines the lists of user-defined methods and pre-defined methods which can be considered. The user-defined methods are added to the result immediately, while the pre-defined methods are filtered further in the following

steps. The second phase of the process only occurs if not sorting alphabetically, and filters the pre-defined methods to consider by calling “isAcceptingOf” on each item returned by the abstract “getPotentialCategorySubComposites”. Each sub-composite which matched one or more pre-defined methods is then added to the result. The third phase, which always executes, is similar to the second, but iterates over the result of calling “getPotentialCategoryOrAlphabeticalSubComposites” and the matched sub-composites are added after the fourth phase is completed. If any unclaimed pre-defined methods remain, the fourth phase simply calls “getUnclaimedJavaMethodsComposite” and populates the resulting composite with the remaining methods.

Since only the first phase of the process involves actually determining which methods are considered for the remaining steps, the methods it invokes must be followed further. If the value type retrieved from the instance factory derives from org.lgna.project.ast.JavaType, “getDeclaredMethods” is called to determine the type's pre-defined methods. The identified methods are ultimately those exposed for the type through the Alice user interface. Inspection of the JavaType class reveals the value returned by “getDeclaredMethods” is evaluated lazily, as shown in Figure 37.

```

this.methods = new Lazy<List<JavaMethod>>() {
    protected List<JavaMethod> create() {
        Class cls = JavaType.this.classReflectionProxy.getReification();
        if (cls != null) {
            LinkedList methods = Lists.newLinkedList();
            HashSet<Method> methodSet = null;
            List<MethodInfo> methodInfos = ClassInfoManager.getMethodInfos(cls);
            if (methodInfos != null) {
                methodSet = new HashSet<Method>();
                for (MethodInfo methodInfo : methodInfos) {
                    try {
                        Method mthd = methodInfo.getMthd();
                        if (mthd == null) continue;
                        JavaType.handleMthd(mthd, methods);
                        methodSet.add(mthd);
                        continue;
                    }
                    catch (RuntimeException mthd) {
                        // empty catch block
                    }
                }
            }
        }
    }
}

```

```

Method[] arrmethod = cls.getDeclaredMethods();
int mthd = arrmethod.length;
int n = 0;
while (n < mthd) {
    Method mthd2 = arrmethod[n];
    if (methodSet == null || !methodSet.contains(mthd2)) {
        JavaType.handleMthd(mthd2, methods);
    }
    ++n;
}
for (JavaMethod method : methods) {
    Method sttr;
    ValueTemplate valueTemplate;
    JavaMethod setter;
    Method mthd3 = method.getMethodReflectionProxy().getReification();
    GetterTemplate propertyGetterTemplate =
        mthd3.getAnnotation(GetterTemplate.class);
    if (
        propertyGetterTemplate == null ||
        (setter =
            ((sttr = PropertyUtilities.getSetterForGetter(mthd3)) != null
             ? JavaMethod.getInstance(sttr).getLongestInChain()
             : null)
            ) == null ||
        (valueTemplate = mthd3.getAnnotation(ValueTemplate.class)) == null
    ) continue;
    JavaMethod m = setter;
    while (m != null) {
        JavaMethodParameter parameter0 = m.getRequiredParameters().get(0);
        parameter0.setValueTemplate(valueTemplate);
        m = m.getNextShorterInChain();
    }
    return Collections.unmodifiableList(methods);
}
return Collections.emptyList();
};

```

Figure 37: Decompiled source of the org.lgna.project.ast.JavaType methods initialization

The process of initializing the method list for the JavaType involves utilizing the org.lgna.project.reflect.ClassInfoManager class to read cached method information in a custom binary format from a ZIP named “/org/alice/stageide/apis/org/lgna/story/classinfos.zip” embedded in the ide-0.0.1-SNAPSHOT.jar and then also using reflection to determine additional instance methods available from the class. The first step, while interesting, seems unnecessary and overly complicated since it will never add any method information beyond that determined in the next step. The second step of the process performs direct runtime reflection on each public and private method of the class itself, adding any additional methods that are found. Since all of the class’s defined methods are examined by this process, replacing an existing class with a new implementation that provides additional methods will result in those newly defined methods being recognized by Alice.

Regardless of the method enumeration process used, each method is filtered through “handleMthd”, listed in Figure 38, to determine whether it should be included in the result. The primary test performed ensures that the method has either public or protected access. If either condition is met, the method will be added to the list of reported methods for the JavaType instance.

```

private static void handleMthd(Method mthd, List<JavaMethod> methods) {
    int modifiers = mthd.getModifiers();
    if (
        JavaType.isMask(modifiers, Modifier.PUBLIC) ||
        JavaType.isMask(modifiers, Modifier.PROTECTED)
    ) {
        JavaMethod methodDeclaredInJava = JavaMethod.getInstance(mthd);
        if (mthd.isAnnotationPresent(MethodTemplate.class)) {
            MethodTemplate methodTemplate = mthd.getAnnotation(MethodTemplate.class);
            if (
                methodTemplate.visibility() == Visibility.PRIME_TIME &&
                !methodTemplate.isFollowedByLongerMethod()
            ) {
                JavaMethod longer = methodDeclaredInJava;
                Method _mthd = mthd;
                while ((_mthd = JavaType.getNextShorterInChain(_mthd)) != null) {
                    JavaMethod shorter = JavaMethod.getInstance(_mthd);
                    if (!_mthd.isAnnotationPresent(MethodTemplate.class))
                        continue;
                    MethodTemplate shorterMethodTemplate =
                        _mthd.getAnnotation(MethodTemplate.class);
                    if (!shorterMethodTemplate.isFollowedByLongerMethod())
                        break;
                    longer.setNextShorterInChain(shorter);
                    shorter.setNextLongerInChain(longer);
                    longer = shorter;
                }
            }
        }
        methods.add(methodDeclaredInJava);
    }
}

```

Figure 38: Decompiled source of org.lgna.project.ast.JavaType’s handleMthd method

Additional processing may also be performed when the method has an annotation of type org.lgna.project.annotations.MethodTemplate. If the annotation specifies a “visibility” value of Visibility.PRIME_TIME and its “isFollowedByLongerMethod” flag is false, then a chain of methods with the same name will be constructed, each of which must also be assigned a MethodTemplate annotation and match the parameter signature of the previously matched overload minus the last parameter. The method chain processing hints that multiple method overloads with the same name may be exposed within Alice 3 in some way, however, that behavior was not exploited for this project.

Once the methods are determined for the JavaClass instance, processing continues within the MemberTabComposite “getSubComposites” method (Figure 36). Each method returned is examined, and two additional tests are performed to determine whether the methods will actually be exposed through the UI. The first test is to invoke the “isAcceptable” method declared by the MemberTabComposite class. Since “isAcceptable” is abstract in MemberTabComposite, the implementing subclasses were examined to determine what the actual behavior will be. Four subclasses of MemberTabComposite exist which provide an implementation of the “isAcceptable” method:

1. org.alice.ide.member.FunctionTabComposite
2. org.alice.ide.member.ProcedureTabComposite
3. org.alice.ide.member.UserFunctionsSubComposite
4. org.alice.ide.member.UserProceduresSubComposite

The first two subclasses are specific to pre-defined functions and procedures, and the last two pertain to user-defined functions and procedures. Both FunctionTabComposite and UserFunctionsSubComposite provide the same implementation of the “isAcceptable” method, simply returning the result of calling “isFunction”, shown in Figure 39, on the AbstractMethod instance being tested.

```
public boolean isFunction() {
    if (this.getReturnType() != JavaType.VOID_TYPE) {
        return true;
    }
    return false;
}
```

Figure 39: Decompiled source of org.lgna.project.ast.AbstractMethod's "isFunction"

The ProcedureTabComposite implementation of “isAcceptable” is very similar, in that it simply returns the result of calling “isProcedure”, shown in Figure 40, on the AbstractMethod instance being tested.

```

public boolean isProcedure() {
    if (this.isFunction()) {
        return false;
    }
    return true;
}

```

Figure 40: Decompiled source of org.IGNA.project.ast.AbstractMethod's "isProcedure"

In UserProceduresSubComposite, the “isAcceptable” implementation is very slightly different. It contains an additional check to reject any static method named “main”, as illustrated in Figure 41.

```

protected boolean isAcceptable(AbstractMethod method) {
    if (method.isStatic() && "main".equals(method.getName())) {
        return false;
    }
    return method.isProcedure();
}

```

Figure 41: Decompiled source of org.alice.ide.member.UserProceduresSubComposite’s “isAcceptable” method

Returning to the MemberTabComposite “getSubComposites” method (Figure 36), if “isAcceptable” evaluates to true, the next filtering step invokes the “isInclusionDesired” static method. The “isInclusionDesired” method, listed in Figure 42, rejects all static and non-public methods or fields, as well as members whose “getVisibility” methods return an org.IGNA.project.annotations.Visibility enumeration instance that is not equal to Visibility.PRIME_TIME.

```

protected static boolean isInclusionDesired(AbstractMember member) {
    if (
        !(member instanceof AbstractMethod && ((AbstractMethod)member).isStatic()) &&
        !(member instanceof AbstractField && ((AbstractField)member).isStatic())
    ) {
        if (member.isPublicAccess() || member.isUserAuthored()) {
            Visibility visibility = member.getVisibility();
            return (
                visibility == null ||
                visibility.equals((Object)Visibility.PRIME_TIME)
            );
        }
    }
    return false;
}

```

Figure 42: Decompiled source of org.alice.ide.member.MemberTabComposite’s “isInclusionDesired”

The values passed from “getSubComposites” to “isInclusionDesired” are always instances of org.lgna.project.ast.JavaMethod. Since JavaMethod always returns false from its “isUserAuthored” method, only methods with public access are accepted. The implementation of “getVisibility” for JavaMethod, shown in Figure 43, retrieves the visibility value from any associated MethodTemplate annotation of the method.

```

public Visibility getVisibility() {
    Method mthd = this.methodReflectionProxy.getReification();
    if (mthd != null && mthd.isAnnotationPresent(MethodTemplate.class)) {
        return mthd.getAnnotation(MethodTemplate.class).visibility();
    }
    return null;
}

```

Figure 43: Decompiled source of org.lgna.project.ast.JavaMethod’s “getVisibility”

Annotations, Exposed

Throughout the course of the investigation, it became apparent that the Alice code utilizes several different annotations, listed in Table 4, to indicate how particular classes, methods and fields should be exposed and manipulated within the Alice 3 interface. Since the MethodTemplate annotation was also encountered earlier when examining org.lgna.project.ast.JavaType’s “getDeclaredMethods” (in both cases it was used to filter method visibility), it was further investigated to determine how and where the annotation is applied within the Alice code.

Annotation classes in package “edu.cmu.cs.dennisc.java.lang”	
Class Name	Use
ParameterAnnotation	Applied to a method parameter to indicate that the parameter is a varargs style array. Only two uses of this annotation exist in the Alice 3.3 source. One in org.lgna.story.Font’s constructor, and the other in org.lgna.common.DoTogether’s “invokeAndWait” method.
Annotation classes in package “org.lgna.project.annotations”	
Class Name	Use
AddEventListenerTemplate	Applied to a method to indicate that the purpose of the method is to add an event listener. Only methods within the org.lgna.story.SScene class have this annotation applied to them. This annotation should be applied in addition to a MethodTemplate annotation.

ArrayTemplate	Applied to an array producing method to specify the length of the returned array. While code exists to extract the length if such an annotation is assigned, no methods in Alice 3.3 have this annotation applied to them.
ClassTemplate	Applied to a class or interface to provide additional information regarding how the class or interface should be utilized, such as whether or not to expose a parent class within the Alice 3 interface. For example, parent class enumeration is disabled for org.lgna.story.event.AbstractEvent, org.lgna.story.SProgram, and org.lgna.story.SThing.
ConstructorTemplate	Applied to a constructor method to specify constructor visibility. This annotation is only applied to constructors within the org.lgna.story.AudioSource class.
FieldTemplate	Applied to class fields to indicate visibility within the Alice 3 interface and provide a method name hint. Appears to be primarily applied to joint properties.
GetterTemplate	Applied to a getter method to specify whether the value returned is persistent or not. In Alice, a getter method takes no parameters, returns a value, and has a name starting with “get” or “is” (if returning a boolean). This annotation should be applied in addition to a MethodType annotation, and does not appear to be required for all getters.
MethodTemplate	Applied to a method to indicate visibility within the Alice 3 interface and to specify whether the method should be considered for addition to an overload chain of same-named methods. No uses of the latter functionality were found in the Alice 3 code.
ResourceTemplate	Applied to an interface to specify the model class type of the resource described by the interface, in addition to whether it describes a top-level resource (Biped, Flyer, Prop, Quadruped, Slitherer, Swimmer, or Transport). For example, the org.lgna.story.resources.TransportResource specifies a modelClass of org.lgna.story.STransport.
ValueTemplate	Applied to either a method parameter or getter to specify an org.lgna.project.annotations.ValueDetails enumeration. Used in the following classes in the org.lgna.story package: AudioSource, Color, SGround, SModel, SRoom, SScene, STurnable.

Table 4: Annotation classes defined in the Alice codebase

In each case where the MethodTemplate annotation is applied to a method within the Alice codebase, it is currently used for one of three purposes. The first use, in which the visibility attribute is set to Visibility.COMpletelyHidden, is to mark a method as not exposed to the user for use in their Alice program. The org.lgna.story.SScene class contains several examples of this type of usage, such as the “removeSceneActivationListener” method shown in Figure 44.

```

@MethodTemplate(visibility=Visibility.COMpletelyHidden)
public void removeSceneActivationListener(SceneActivationListener listener)
{
    this.implementation.removeSceneActivationListener(listener);
}

```

Figure 44: Example of a MethodTemplate annotation used to prevent method exposure

The second application of the MethodTemplate annotation, in which the visibility attribute is set to Visibility.TUCKED_AWAY, applies the same logic as the first, essentially indicating a method should not be displayed to the user. The difference in the visibility attribute may be an indication that these methods may be exposed indirectly via a separate interface in Alice, such as the scene setup interface. An example of this use is illustrated by the “setName” method of the org.lgna.story.SThing class, shown in Figure 45.

```

@MethodTemplate(visibility=Visibility.TUCKED_AWAY)
public void setName(String name) {
    this.getImplementation().setName(name);
}

```

Figure 45: Example of using Visibility.TUCKED_AWAY to prevent method exposure

The final application of the MethodTemplate annotation either specifies a visibility of Visibility.PRIME_TIME or doesn’t specify a visibility value at all, since Visibility.PRIME_TIME is the default visibility value. In this case, it indicates the method should be directly exposed in the Alice GUI for use within user programs. This application of the annotation is used on several methods, such as the “setRadius” method of the org.lgna.story.SCylinder class, illustrated in Figure 46.

```

@MethodTemplate
public void setRadius(Number radius, SetRadius.Detail... details) {
    this.implementation.radius.animateValue(
        radius.doubleValue(),
        Duration.getValue(details),
        AnimationStyle.getValue(details).getInternal()
    );
}

```

Figure 46: Example of MethodTemplate annotation indicating method is to be exposed

Identifying a Suitable Test Subject

Once the mechanisms by which Alice determines which object types and methods to expose for use within a user program was understood, it was time to identify a class within Alice that could be easily extended. Since the intent is to expose new functionality for user programs to invoke, such a class would need to be a subclass of either `org.lgna.story.SProgram` or `org.lgna.story.SThing` (but not a subclass of `org.lgna.story.SMarker`). It must also be possible for the user to easily create new instances of the extended class within their project code.

While it may seem reasonable to augment the `SProgram` or `SThing` classes themselves, there are some concerns with such an approach. Those classes define several internal properties and methods, which would need to be re-implemented in any overriding class. `SProgram` defines two properties and seven methods to override, and `SThing` defines 12 methods of its own. Secondly, each of those classes currently have a `ClassTemplate` annotation attached which may also need to be replicated identically on the replacement class. The existing complexity of `SProgram` and `SThing` would increase the probability that future updates to Alice 3 may change their properties in a way that would make the substituted class incompatible, and require updates to it.

Since it seemed that overriding the `SProgram` or `SThing` classes themselves may not be the most appropriate approach, an alternative subclass needed to be identified. The ideal class would define no properties or methods of its own, and thereby reduce the risk of future incompatibility. Alice doesn't define any subclasses of `SProgram`, so only descendants of `SThing` needed to be examined to find a suitable match.

The `org.lgna.story.STransport` class was identified as a promising starting point to try inserting custom Finch-specific methods. It is a base class inherited by all transport object classes within Alice 3 (e.g. `Aircraft`, `Automobile`, `Watercraft`, etc.). As such, methods added to `STransport` can be easily accessed within the Alice 3 interface by simply adding any type of transport object into the scene. The original implementation of the `STransport` class, shown in Figure 47, is very short and adds no new methods to its base class of `org.lgna.story.SJointedModel`. These properties are ideal for isolating newly added methods from

existing code, thus reducing the interdependency between Alice 3 code and any methods added for manipulating a Finch robot.

```
package org.lgna.story;

import org.lgna.story.SJointedModel;
import org.lgna.story.implementation.TransportImp;
import org.lgna.story.resources.TransportResource;

public class STransport
    extends SJointedModel
{
    private final TransportImp implementation;

    public STransport(TransportResource resource) {
        this.implementation = resource.createImplementation(this);
    }

    @Override
    TransportImp getImplementation() {
        return this.implementation;
    }
}
```

Figure 47: Original decompiled source of org.lgna.story.STransport class

As detailed in “Annotations, Exposed”, the Alice code uses a metadata annotation of type org.lgna.project.annotations.MethodTemplate to decorate methods which should be exposed to users within Alice. For the purposes of this project, the parameter-less form was used to ensure the methods were fully exposed to users. An example of a Finch method as added to the STransport class is shown in Figure 48.

```
@MethodTemplate
public void finchSetWheelVelocities(int leftVelocity, int rightVelocity) {
    synchronized (finchLock) {
        getFinch().setWheelVelocities(leftVelocity, rightVelocity);
    }
}
```

Figure 48: Example of Finch method added to STransport class

4.2 Communicating with the Finch

The initial approach to interacting with the Finch from Alice 3 involved updating STransport to directly access the Finch Java API [5] distributed as a JAR file.

Modifications were made to the STransport class to incorporate several additional methods for interacting with a Finch, modelled on the API exposed by the Finch library. A private `getFinch()` method was added which managed a singleton instance of the `edu.cmu.ri.createlab.terk.robot.finch.Finch` class, and that single Finch instance was then used for all communication with the attached Finch robot.

While this seemed like a straightforward approach, several drawbacks were identified, including the need to distribute an additional 3rd party JAR library and to inject that JAR into the Alice 3 classpath. Additional issues were also encountered when attempting to communicate with the Finch. Any call to a method which interacted with the Finch caused Alice to become unresponsive.

The initial thought was that there was an issue with improper thread synchronization being performed with the Alice modifications when accessing the Finch API. However, no such problems could be identified. The `getFinch()` singleton initialization code uses a simple null check inside a synchronized block using a dedicated internal lock object, and each call to a Finch API method was also synchronized using the same lock object.

It was quickly ruled out that the problem was unique to the Alice modifications, as all of the sample programs included with the Finch software exhibited the same behavior of hanging on startup. This behavior was observed for both 32-bit and 64-bit JVMs on the Windows 7 laptop used as a test machine, although the same modifications run on a Linux host appeared to work as expected.

The Finch sample programs can be configured to create a log file containing debugging information regarding the Finch hardware detection process and communication error results. Examining the log generated when running those programs in Windows revealed that they were experiencing some sort of issue when attempting to write to the USB port, since a “Bad File Descriptor” error was being reported. Strangely, it seemed that the Finch was responding to the first command sent to it, indicating that sending the command to the Finch actually succeeded, although it was unknown whether the confirmation response from the Finch was being received properly.

USB Debugging with USBPCap

As part of troubleshooting the issues encountered with using the Finch API under Windows, the raw communication over the USB port between the host computer and the Finch robot was examined. This was accomplished using a USB packet interception library for Windows called USBPCap [45] in combination with Wireshark [69] (Figure 49).

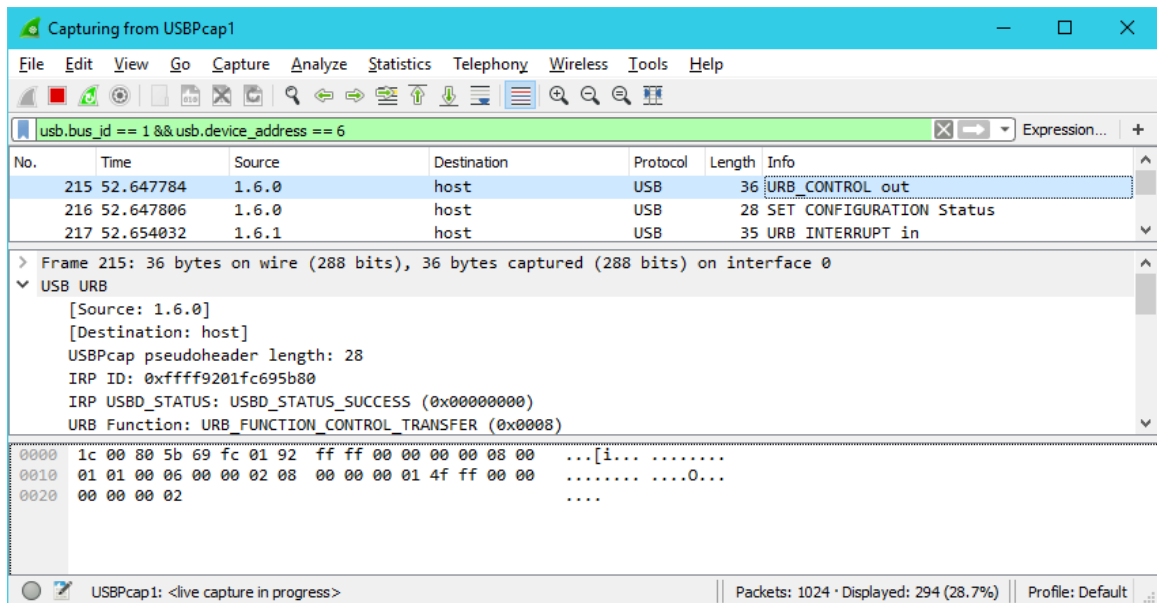


Figure 49: Wireshark capturing communication with a Finch robot using USBPCap

The USB architecture from the perspective of the host computer is arranged as a multi-leveled star topology containing hubs and other devices, and a modern host computer has several built-in and commonly attached USB devices necessary for property operation (for example, Wi-Fi adapter, keyboard, mouse, touchpad, audio and video devices, etc.) Since several devices are attached to the USB bus, and many of those produce copious amounts of traffic on the bus, utilizing Wireshark to analyze the traffic from a single device like a Finch posed challenges of its own.

In order to reduce the clutter from all of the other devices, a property of the USB protocol was leveraged to identify the location of the Finch on the bus. When a new USB device is attached, it broadcasts a device descriptor packet containing product and vendor IDs which uniquely identify the type of device. Since a Finch robot identifies itself in the device

descriptor exchange with a vendor ID of 0x2354 and a product ID of 0x1111, those ID values were used to define a Wireshark filter expression to reduce the displayed packets to only descriptor packets from a Finch. Once those packets were identified, the specific USB address to which the Finch was connected could be determined by extracting the bus ID and device address from the headers of the descriptor packets received when the Finch was attached to the host computer.

With the correct bus and device address in hand, a Wireshark packet filter expression was applied to filter all applicable packets, and it was verified that the initial command was indeed sent to the Finch successfully, and that the Finch was returning the proper response. In fact, analysis of the USB packets being transferred between the host computer and the Finch robot showed no sign of any issues, as the structure of the packets matched exactly with the published USB communication protocol [9] of the Finch robot. Data was observed both being sent to the Finch, and responses being returned back to the host computer.

Down the Rabbit Hole

Since the communication issues did not appear to be a result of hardware failure, the next logical place to investigate was the Finch Java API itself. Through examination of the Finch API source code [13], the trail led to an embedded JAR (create-lab-commons-usb-hid.jar) which is managed under a separate project, Create Lab Commons [12]. Within the Create Lab Commons JAR, the actual communication with the Finch device on Windows is performed by a class named `edu.cmu.ri.createlab.usb.hid.windows.WindowsHIDDevice`.

The `WindowsHIDDevice` class invokes methods of the `edu.cmu.ri.createlab.usb.hid.windows.Kernel32Library` class which uses JNA (Java Native Access) to call native Windows kernel functions directly to open, read from, and write to USB devices. After each operation, the code inspects the operating system error code returned by the “`getLastError`” function of the `com.sun.jna.Native` JNA class. If the returned result indicates a non-zero error code, the Finch API reports the communication as failed.

According to the JNA documentation [51], using “`getLastError`” to retrieve error status may be unreliable on some operating systems. The recommended method to determine

whether an error was returned is to update the JNA import definitions for the kernel functions being called to declare them to potentially throw a `com.sun.jna.LastErrorException`. Unfortunately, when this method was applied to the `Kernel32Library` class, and the JAR file was updated, the same “Bad File Descriptor” errors were thrown. Thus it seemed it wasn’t the use of that specific API that was to blame.

More testing revealed that while the “`getLastError`” method returned an error code after writes to the Finch, the Windows API “`writeFile`” call itself was actually returning a result indicating success. If the Finch API code was modified to only inspect the last error code when “`writeFile`” returned success, then writes to the Finch succeeded as expected. Unfortunately, once writes were working better, the same issue occurred when reading the responses returned by the Finch. The Windows API “`readFile`” call would return success, yet the Finch code still checked “`getLastError`”, received a “Bad File Descriptor” error, and the program would hang. After applying the same updates to the reading code, communication with the Finch proceeded reliably and the hanging issue was resolved.

The end result of the debugging and trial-and-error was that two classes within the Create Lab Commons JAR required updates: `edu.cmu.ri.createlab.usb.hid.windows.WindowsHIDDevice` and `edu.cmu.ri.createlab.usb.hid.windows.Kernel32Library`. A modified version of the Finch API JAR file was generated which could be used instead of the JAR file distributed directly from the Finch website. However, maintaining and distributing a separate fork of the Finch API would increase the complexity of the project. Even overriding the specific classes within the Finch JAR in the same manner as being used for Alice could incur a fair amount of maintenance overhead, due to the complexity of the two classes which required modifications. As such, an alternative Finch communication method was also examined to determine if it would be more suitable to the task.

4.3 The BirdBrain Robot Server

An alternative to using the Finch API JAR directly is the BirdBrain Robot Server, which is maintained and distributed by the producer of the Finch robot, BirdBrain Technologies,

LLC. The BirdBrain Robot Server implements an HTTP server which exposes web service endpoints to use for communicating with a Finch robot connected to the same computer the server is running on. Since the HTTP protocol is used for all communication with the server, it can be interacted with from any environment which provides an HTTP client, for example Snap! and Scratch which use it to interact with a Finch.

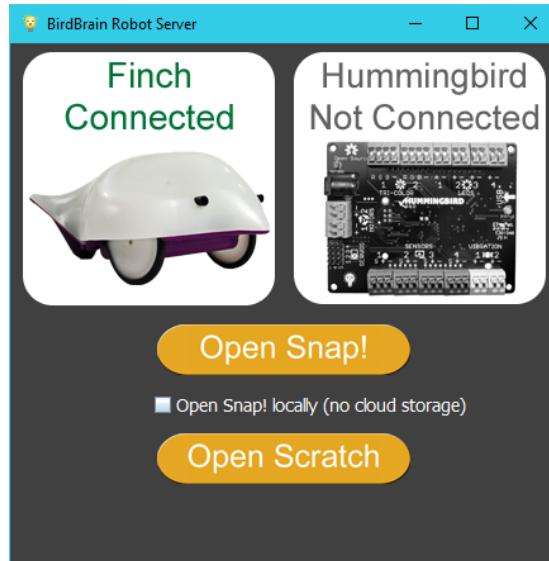


Figure 50: The BirdBrain Robot Server user interface

There are some drawbacks to using the BirdBrain Robot Server. It requires a separate manual installation by users on the machine which the Finch is connected to. Additionally, the server must be running and responding to requests for an application to communicate through it to the attached Finch.

There are also several advantages to utilizing an HTTP-based communication proxy implementation, such as the BirdBrain Robot Server. One such advantage is that there is no need to distribute the Finch API JAR directly, which simplifies the project packaging and installation. The Robot Server approach also opens up new possibilities, such as communicating with a Finch robot connected to a remote computer, instead of being limited to a direct connection. In fact, were multiple servers available on a network, a single program could potentially interact with several Finch robots simultaneously. Conversely,

it also permits the possibility of multiple client programs accessing sensor information from or controlling a single Finch robot simultaneously.

While using the BirdBrain Robot Server expands the ability to communicate with a Finch robot to many applications that have built-in support for performing HTTP requests, Alice 3 does not provide any such innate capability to user programs. To facilitate the HTTP communication with the server, a new class named FinchHTTP was created and loaded into Alice via a custom JAR, and the org.lgna.story.STransport class was overridden as before. The new class exposes an API similar to that provided by the standard Finch Java API to minimize the modifications required within STransport. Under the hood, the new class utilizes the standard java.net.HttpURLConnection class for executing the HTTP requests to the server, which means no additional 3rd party libraries are needed.

The HTTP responses received are parsed by the FinchHTTP class and converted to appropriate Java values, depending on the type of request performed. The BirdBrain Robot server provides service endpoints for both sensor input (see Table 5) and controlling output devices on the Finch and the host computer (see Table 6). In general, sensor value responses are returned as single items or space-separated lists containing either a numeric, boolean or string value.

BirdBrain Robot Server Finch Sensor Value Services	
Service Path	Value Returned
/finch/in/accelerationX	X-axis acceleration value read from the Finch's accelerometer, in g's.
/finch/in/accelerationY	Y-axis acceleration value read from the Finch's accelerometer, in g's.
/finch/in/accelerationZ	Z-axis acceleration value read from the Finch's accelerometer, in g's.
/finch/in/accelerations	accelerationX SPACE accelerationY SPACE accelerationZ (Example: "0.0 0.0 1.07")
/finch/in/lightLeft	Value read from the Finch's left light sensor, ranging from 0 to 100.
/finch/in/lightRight	Value read from the Finch's right light sensor, ranging from 0 to 100.
/finch/in/lights	lightLeft SPACE lightRight (Example: "55 50")
/finch/in/orientation	"Beak Up" "Beak Down" "Level" "Upside Down" "Left Wing Down" "Right Wing Down"
/finch/in/obstacleLeft	"true" "false"
/finch/in/obstacleRight	"true" "false"

/finch/in/obstacles	obstacleLeft SPACE obstacleRight (Example: "true false")
/finch/in/temperature	Value read from the Finch's temperature sensor, in degrees Celsius.
/poll	Listing of all current sensor values.
NOTE: All "/finch" services may return "null" if the server does not detect a connected Finch robot.	

Table 5: BirdBrain Robot Server Finch Sensor Value Services

BirdBrain Robot Server Finch Control Services	
Service Path	Parameters Expected
/finch/out/motor	leftSpeed "/" rightSpeed Values range from 0 to 100. (Example: "/out/motor/100/-100")
/finch/out/buzzer	frequencyHz "/" durationMillis (Example: "/out/buzzer/440/250")
/finch/out/led	red "/" green "/" blue Values range from 0 to 100. (Example: "/out/led/100/0/0")
/reset_all	Turns off all motors, buzzers, and LEDs on the connected Finch.
/speak	"The text to output using speech synthesis" (Example: "/speak/I am a Finch")
NOTE: All "/finch" services may return "null" if the server does not detect a connected Finch robot.	

Table 6: BirdBrain Robot Server Finch Control Services

Operating System-specific Issues

After implementing the new FinchHTTP class, and retrofitting the org.lgna.story.STransport class to use it, the issues previously encountered with Finch communication were no longer experienced within Alice on Windows. However, while the BirdBrain Robot Server resolved issues experienced under Windows, its use presented new challenges on another operating system. On OS X systems, it responded very slowly to requests and caused sporadic behavior within Alice when the BirdBrain Robot Server was not the currently active application. This was identified as a result of a power-saving feature built into the operating system called App Nap introduced in OS X Mavericks.

App Nap causes background applications to be scheduled for very little and infrequent CPU time, with the assumption that if they are not the foreground application, they will not require fast response times or have large processing requirements. For a server application like the BirdBrain Robot Server, which should serve requested content quickly, this is an undesirable behavior. Fortunately, OS X provides the ability for users to disable App Nap

on a per-application basis. When App Nap is disabled for the BirdBrain Robot Server, the delays and unreliability previously experienced when Alice was communicating with it were no longer a problem.

Unsupported Finch Functionality

While the BirdBrain Robot Server does provide support for a significant portion of the functionality provided by the `edu.cmu.ri.createlab.terk.robot.finch.Finch` class from the Finch Java API, there are some minor features which have not been implemented. These missing features include the ability to read the shaken or tapped status from the Finch accelerometer, as well as support for playing an audio tone or sound file on the computer the Finch is connected to.

The capability to produce sound through the computer's speakers is already incorporated into Alice. As a result, the server's lack of support was not considered a large deficiency. However, supporting the detection of when a Finch is tapped or shaken is an unfortunate loss of functionality. Without that ability users have fewer possibilities to leverage the Finch an interactive controller than they otherwise would.

One hypothesis for why this support was not included is that it may be related to how the BirdBrain Robot Server continually polls the Finch for status updates, even when no client is requesting data. Since the Finch's internal accelerometer will reset its shaken and tapped event flags after each poll, the status is very transient could be easily lost. Were the status of these events to be stored in internal server state as Boolean values as they are returned by the Finch, a similar behavior of resetting the state would likely be required each time the values were read from the server. Such a side-effecting operation for a state read operation would be inappropriate for a server which may potentially have multiple simultaneous clients that desire to know whether a shake or tap occurred since the last time they queried the status.

In an attempt to address this limitation, a pull request [32] was submitted to the BirdBrain Robot Server project on GitHub. The pull request adds the capability to report shaken and tapped events for a Finch in a manner which fully supports multiple clients

accurately reading the event. To accomplish this, timestamps are updated in the server state each time that polling of the Finch's status indicates that a shake or tap event occurred. When a client queries one of the services provided by the server for the status of one of the events, the server responds with the timestamp of the most recent event of that type. The client can then compare the received timestamp to the timestamp it received from a prior request for the event status. If the two timestamps differ, then the requested event has occurred and the client can react accordingly. As of the time of this writing, the submitted pull request has not yet been accepted into the master branch of the BirdBrain Robot Server.

4.4 Supporting New Releases of Alice 3: Augmenting the classpath

During the course of the project, several new releases of Alice 3 were made available. When the project began, Alice 3.1.81 was the most recent release. Alice 3.1 utilized a file named "Alice.ini" in the root installation folder to specify the classpath entries provided to the JRE when launching Alice. To augment the classpath in Alice 3.1, it was as simple as updating the "Alice.ini" file to contain the desired entries.

With the release of Alice 3.2, the graphical installer was switched to use install4j. Along with the new installer, the method for updating the classpath when launching Alice 3 also changed. With install4j, the default classpath is embedded differently for each supported operating system. For example, on Windows, the classpath is embedded in the launcher executable, while on Linux it is constructed by the shell script used to launch Alice. Fortunately, install4j supports a mechanism to adjust the classpath at runtime beyond that specified when the installation package was constructed, without bypassing the install4j launcher. Sadly, the manner in which the install4j support is implemented varies between each supported operating system.

On Windows and Linux, install4j will look for a file named "Alice 3.vmoptions" in the root installation path. If the file is found, it will be parsed and entries within it can append or prepend items into the classpath before Alice 3 is launched. Of primary interest is the ability to prepend entries to the classpath, since that permits overriding of classes. Prepending an entry can be accomplished by specifying "-classpath/p" directive in the file.

While this seems simple enough, the interpretation of entries varies between the two different operating systems. On Windows, `install4j` supports expansion of an environment variable named `EXE4J_EXEDIR` containing the absolute path to the launcher executable. Using that environment variable, the absolute path to the injected JAR file can be specified, regardless of the current working directory. An example of a Windows `vmoptions` file is shown in Figure 51.

```
-classpath/p ${EXE4J_EXEDIR}ext\finch4alice\finch4alice.jar
```

Figure 51: Example of a Windows `install4j` `vmoptions` file

On Linux, `install4j` does not support expansion of the `EXE4J_EXEDIR` variable, and there is no equivalent. Fortunately, when launching Alice 3 on Linux, the working directory always appears to be set to the directory containing the launcher script, so using a relative path for the Finch 4 Alice JAR works as desired. A Linux `vmoptions` file is illustrated in Figure 52.

Another difference between Windows and Linux is that the file path separators differ. For Windows, the semicolon character is used to separate different path elements, while for Linux, the path separator is the colon character. For this project, the difference in separator character wasn't significant, since only one item was prepended to the classpath. If multiple JARs needed to be injected, for example if the Finch Java API JAR had been used, a path separator would be needed.

```
-classpath/p ext/finch4alice/finch4alice.jar
```

Figure 52: Example of a Linux `install4j` `vmoptions` file

To further complicate matters, `install4j` on OS X operates slightly differently as well. Instead of looking for a file named “Alice 3.`vmoptions`”, it will search for one named “`vmoptions.txt`”. Luckily, the OS X version of `install4j` accepts the same format as the Linux version, so the file name change is all that is necessary.

5. Finch 4 Alice Deployment

Finch 4 Alice [29] is an open source implementation of the concepts explored in the preceding sections of this report. It is a simple extension to Alice 3 that adds methods for controlling a Finch robot to all STransport subclass instances. Cross-platform support is provided out-of-box for the Microsoft Windows, Apple Macintosh OS X, and Linux operating systems. The Finch 4 Alice source code, documentation and binary downloads are publicly available on GitHub and licensed under the permissive BSD 2-Clause License [64]. During the implementation of Finch 4 Alice, several additional tools were utilized and new challenges were encountered, as detailed in the remainder of this chapter.

5.1 Supporting Multiple Operating Systems

One of the goals of the project was to provide support for all of the operating systems officially supported by Alice 3. Those include Microsoft Windows, Apple Macintosh OS X, and varieties of Linux providing a full desktop environment. To accomplish this goal, care was exercised in the selection of software build and validation tools to ensure that they provided strong cross-platform support. Automation of complex tasks was also emphasized in an effort to facilitate contribution and simplify direct use of the project source, if desired. Additionally, an attempt was made to reduce the amount of software, beyond the base operating system, that is required to be manually installed on the user's computer as prerequisites to successfully build the Finch 4 Alice project.

5.2 Build Automation with Gradle

The primary build automation tool used by the Finch 4 Alice project is Gradle [34], which provides a flexible, cross-platform, code-configured alternative to other popular

Java build tools such as Ant [59] and Maven [60]. Gradle has several attractive features that led to its selection as the build tool of choice for this project.

The Only Manual Dependency is the JDK

Since Gradle is implemented using JVM languages and packaged in JAR format, it will run in any environment which has a supported JRE installed. This was particularly useful for targeting all of the desired operating systems, because a single build tool with a single configuration can be used, simplifying the implementation of build automation. Some of the Gradle plugins used by Finch 4 Alice require access to additional tools provided by the JDK, such as the Java compiler. Since the JDK itself is not available through the automated dependency mechanism, it must be installed by the user prior to building Finch 4 Alice. Since the JDK comes with a bundled JRE, it is the only manual dependency that must be installed by the user.

The Gradle Wrapper

One of the most desirable features of Gradle is the Gradle Wrapper [35], a shell script that automates the task of downloading a project-specific version of Gradle for executing the build. Utilizing the Gradle Wrapper relieves the user from the tasks of downloading and installing Gradle manually and guarantees that the correct version of the build tool is used. Gradle Wrapper scripts are included in Windows batch format and in unix shell format for both Linux and OS X.

Configuration through Code

The configuration of Gradle is done through code, which allows for a more expressive build configuration than XML-configured tools like Ant or Maven. Several reasonably complicated tasks are performed by the Finch 4 Alice Gradle configuration, such as detection of the Alice 3 installation path and support for direct installation of Finch 4 Alice into Alice 3.

Dependency Management

Gradle also provides rich dependency management, and takes advantage of existing infrastructures and public Maven and Ivy repositories to retrieve external dependencies. Several large public repositories are currently available and offer access to many thousands of JAR packages which can be configured as project dependencies and automatically downloaded as part of the project build process.

Plugin Support

A wide variety of Gradle plugins are available to support complex tasks and automate external tools. Finch 4 Alice utilizes several of these plugins to support operations such as producing a graphical izpack installer JAR, generate a Windows executable wrapper for the installer, publish updates to the project GitHub pages, and generate API documentation. Table 7 provides a listing of the Gradle plugins used by Finch 4 Alice.

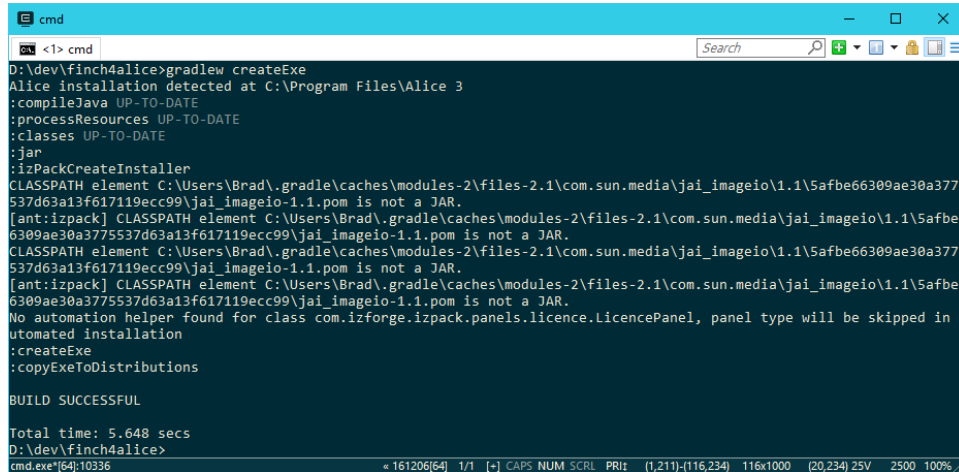
Plugin Identifier	Version	Description
java		Standard built-in plugin supporting Java projects
com.bmuschko.izpack	2.1	Automates IzPack installer creation tool
edu.sc.seis.launch4j	2.1.0	Generates a Windows EXE wrapper from a runnable JAR.
org.ajoberstar.github-pages	1.6.0	Publishing automation tasks for updating GitHub Pages Git repositories.
com.github.ben-manes.versions	0.13.0	Automates the task of checking for new versions of configured Gradle plugins.

Table 7: Gradle plugins used by Finch 4 Alice

Simple Command Line Interface

When it comes to hands-free build automation and scripting build tools, command line tools are best suited for the task. They work well in text-based, non-graphical environments as are often encountered in remote servers with limited memory or CPU requirements. Executing Gradle tasks is accomplished through short commands, and while several

granular tasks are defined for different steps of the build process, a selected task will automatically execute all prerequisite tasks needed to complete, as illustrated in Figure 53.



```
cmd
D:\dev\finch4alice>gradlew createExe
Alice installation detected at C:\Program Files\Alice 3
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar
:izPackCreateInstaller
CLASSPATH element C:\Users\Brad\.gradle\caches\modules-2\files-2.1\com.sun.media\jai_imageio\1.1\5afbe66309ae30a3775537d63a13f617119ecc99\jai_imageio-1.1.pom is not a JAR.
[ant:izpack] CLASSPATH element C:\Users\Brad\.gradle\caches\modules-2\files-2.1\com.sun.media\jai_imageio\1.1\5afbe66309ae30a3775537d63a13f617119ecc99\jai_imageio-1.1.pom is not a JAR.
CLASSPATH element C:\Users\Brad\.gradle\caches\modules-2\files-2.1\com.sun.media\jai_imageio\1.1\5afbe66309ae30a3775537d63a13f617119ecc99\jai_imageio-1.1.pom is not a JAR.
[ant:izpack] CLASSPATH element C:\Users\Brad\.gradle\caches\modules-2\files-2.1\com.sun.media\jai_imageio\1.1\5afbe66309ae30a3775537d63a13f617119ecc99\jai_imageio-1.1.pom is not a JAR.
No automation helper found for class com.izforge.izpack.panels.licence.LicencePanel, panel type will be skipped in automated installation
:createExe
:copyExeToDistributions
BUILD SUCCESSFUL

Total time: 5.648 secs
D:\dev\finch4alice>
```

Figure 53: Example of executing Gradle 'createExe' task

5.3 Cross-Platform Graphical Installer

Finch 4 Alice can be bundled into a cross-platform graphical installer application to simplify distribution to end users. To accomplish this, it utilizes the open source IzPack installer generation tool to generate a runnable JAR file containing the Finch 4 Alice assets. By default, the installer JAR executes as a windowed, graphical application (as shown in Figure 54) that walks users step-by-step through the installation process.

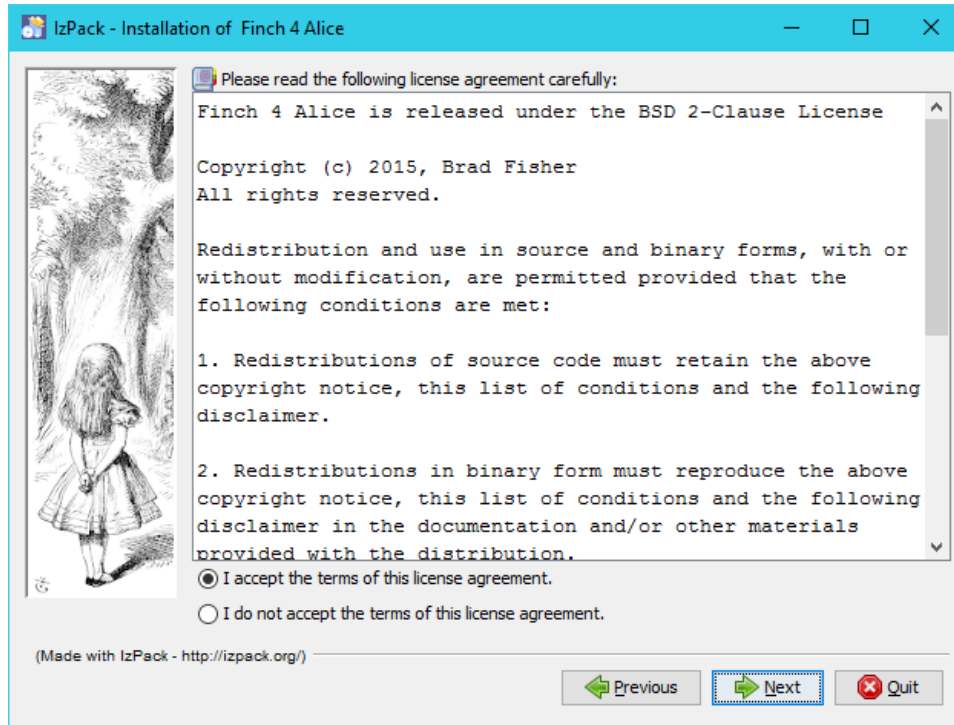


Figure 54: The Finch 4 Alice graphical installer

The JAR generated by IzPack can be executed directly on all supported operating systems by using the 'java' program directly on the command line, and can also be launched from a graphical file-browser application in operating systems that have an appropriate file type mapping that associates JAR files with the Java executable. Such a file type association is currently configured by default on Macintosh OS X and Ubuntu Linux, but must be manually associated on Microsoft Windows. Other Linux distributions may require an association to be configured before the JAR can be launched through the GUI. Another caveat on Linux systems is that the JAR must have the executable bit set to permit execution via a file association.

Besides IzPack, several other options were considered for use in generating the installer, including the open source VAInstall [3], Packlet [36], and Open Installer [49]. However, none of those alternatives have been actively maintained for several years, nor did any of them appear to have clear documentation describing their use. As far as Gradle integration, there were also no existing plugins available to facilitate their use.

The commercial products `install4j` by ej Technologies and `InstallBuilder` [11] from BitRock Inc. were also considered. Free licenses for open source projects can be requested from both vendors [10,26]. Of the two, only `install4j` has an available Gradle integration plugin [27], although `InstallBuilder` can be automated by executing a command line program directly from within Gradle. Both products provide excellent documentation, and a wide variety of examples for each can be found by searching the Internet.

`IzPack` was chosen over the evaluated commercial products for several reasons. Since `Finch 4 Alice` is open source software itself, it was deemed important that the entire build process was comprised of only open source software. `IzPack` is licensed under the permissive Apache License, version 2.0, which allows its use in both commercial and non-commercial software.

A Gradle plugin is available for automating `IzPack`, and, unlike the commercial alternatives, the `IzPack` plugin retrieves all of the software necessary directly from a dependency repository such as Maven Central [57] or JCenter [37] without needing any external installation. This helps maintain the simplicity of the build process, by reducing manual dependencies and processes. In addition, `IzPack` has been written in Java with a focus on cross-platform support, targeting all of the operating systems specified by the `Finch 4 Alice` requirements. Finally, an active community of developers maintain `IzPack`, reasonable documentation is provided to assist with configuration, and multiple configuration examples and tutorials can be found through an internet search.

Some challenges were encountered using `IzPack`, however. While the product itself has a long history, issues were encountered when attempting to use version 5, the most recent major release. A large amount of the existing documentation and examples were for the preceding version 4. While version 5 maintained much of version 4's configuration syntax, some breaking modifications were introduced that made it somewhat difficult to determine the correct organization to use.

The most difficult challenge with using `IzPack` was a result of bugs that prevented documented features from working as advertised. The Gradle `IzPack` plugin [47] used by `Finch 4 Alice` is a wrapper around the Ant integration task provided as part of `IzPack`. That

Ant plugin provides a null configuration value to IzPack, which was accepted as valid by version 4, however in version 5.0.6 and earlier a null pointer exception would occur.

A second issue was encountered when attempting to use a configuration with compound logical condition. While the configuration syntax was documented as supporting logical AND and OR operations to combine conditions, in practice it resulted in null pointer exceptions. These errors were due to an oversight in the IzPack code whereby the context data used by the conditional operator was not assigned before it was read and utilized.

To resolve these issues and permit use of the newest IzPack version, it was decided to attempt to analyze and contribute corrections back to the IzPack project. The task of determining the root cause of the encountered bugs was complicated somewhat by the structure of the IzPack code itself. IzPack makes heavy use of PicoContainer [55], an inversion of control library for managing dependency injection and object instance initialization and construction. One of the main tenants of PicoContainer is the idea of registering instances of particular classes within a container, which will then be provided indirectly when using that container to construct instances of other classes which depend on one or more objects of the registered types for initialization. The use of PicoContainer within IzPack seemed to introduce additional code complexity and reduce code clarity, causing it to be more difficult to identify and derive a proper solution to the problems faced. Despite the complexity of the IzPack code, corrections were submitted through two GitHub pull requests [30,31], and they were reviewed and merged very quickly by the maintainers. Release 5.0.7 incorporated the submitted updates, and no more major issues were encountered with the use of IzPack by Finch 4 Alice.

5.4 Platform-Specific Installer Options

In order to facilitate installation of Finch 4 Alice in specific operating systems, two additional installation packages may be produced.

Windows Executable Wrapper

As described in 5.3, IzPack creates an installer in the form of a JAR file. Since Windows may not be configured to launch JAR files automatically when double-clicked, the Finch 4 Alice project provides a task to create a native Windows executable wrapper using Launch4j [40]. Like IzPack, Launch4j is invoked through a Gradle plugin, and all dependencies are retrieved automatically. Despite generating a binary that can only be executed on Windows, creating the file is supported on all of the targeted operating systems.

Shell Script Wrapper

As an alternative to the JAR-format installer for Linux and OS X, a task has been provided for producing a shell script wrapper around the JAR. When executed, the script locates the installed java binary, through which it then launches the installer JAR embedded within itself. While this method still requires the user to launch the script through a shell, it provides a more automated alternative for installing in environments where a file association is not defined, since it doesn't require the user to know where the java executable is installed or how to invoke it.

5.5 API Documentation

Documentation for the Finch 4 Alice API is generated using Javadoc. Gradle tasks are provided for generating the documentation locally, and for publishing documentation updates directly to the project's website. Publishing of the API documentation for releases is accomplished using the `org.ajoberstar.github-pages` Gradle plugin, which automates cloning the GitHub Pages Git repository, copying the API documentation files into place, and pushing the updates back to GitHub.

5.6 Automated Builds and Release Artifact Publishing

The Travis CI [65] continuous integration build service was utilized to ensure that each update to the Finch 4 Alice project's code results in a working distribution on supported operating systems. Whenever a new commit or tag is added the Finch 4 Alice codebase, Travis CI executes the build in both Linux and OS X environments, and the resulting build status is displayed as a badge on the Finch 4 Alice website and GitHub page.

Release versions are identified by tagging a specific commit to produce a release from. After Travis CI successfully builds a release tag, it will publish the generated artifacts back into GitHub as a release and generate and publish the API documentation to the project's GitHub Pages site. As illustrated in Figure 55, the currently published release artifacts include the generic installer JAR, Windows installer EXE, shell wrapper script, API documentation, and archived snapshots of the source code.

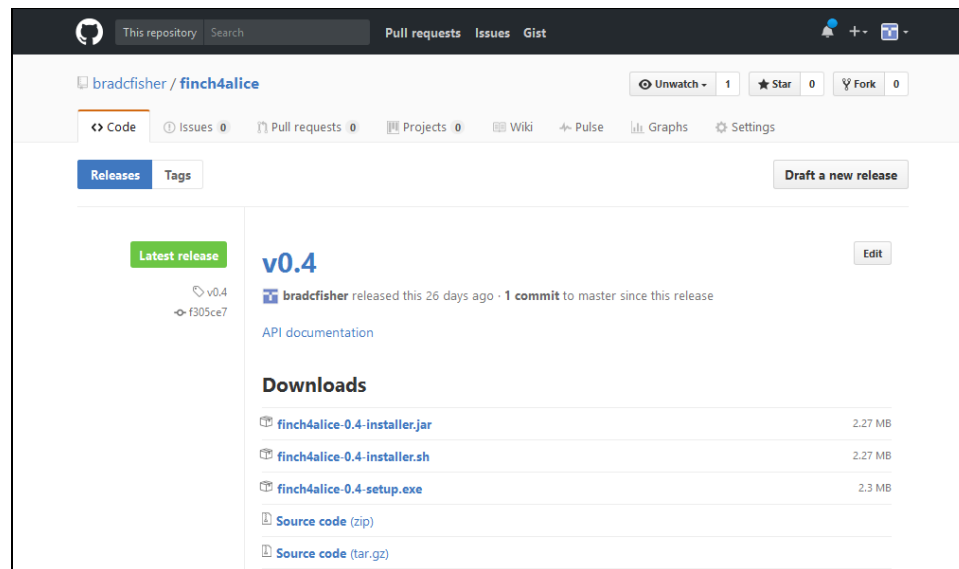


Figure 55: GitHub Release artifacts of Finch 4 Alice v0.4

5.7 Acquiring Finch 4 Alice

Both source code and release binaries of Finch 4 Alice can be obtained from the project's GitHub project, which is located at <https://github.com/bradcfisher/finch4alice>. The main project readme provides a wealth of documentation, including compilation instructions for

each of the supported installers. A project website, <http://finch4alice.com>, was setup using GitHub pages and provides additional content, including generated API documentation for each release. To acquire the latest Finch 4 Alice release, visit <https://github.com/bradcfisher/finch4alice/releases/latest>.

6. Future Work

During the course of the project, some potential opportunities for future work were identified, some of which may be appropriate as undergraduate projects. This chapter details several such opportunities.

6.1 Use in Introductory CS Courses

One of the original driving forces behind the conception of the Finch 4 Alice project was a desire to incorporate more interactive and engaging technologies, such as the Finch, into existing introductory coursework. Since Alice 3 was already being used for teaching students programming concepts, the idea of extending its capabilities to the physical world was compelling.

Now that Finch 4 Alice has made interacting with a Finch robot from an Alice 3 program a reality, all that remains is to adapt and augment existing curriculum to incorporate its use. The development of this new classroom material, along with trial implementation and analysis of its effectiveness could be an opportunity for one or more enterprising students to undertake.

6.2 Maintenance of the Finch 4 Alice Open Source Project

To remain relevant, the Finch 4 Alice project itself will need to be maintained. This may be necessary for several reasons. For example, as materials are developed for use in the classroom, the capabilities of Finch 4 Alice may require tweaking to accommodate new ideas and interaction paradigms. Addressing feature requests and correcting bugs in an open source project can provide valuable experience to students.

One future enhancement that could be undertaken would be the implementation of a continuous integration method with Windows support. The current CI solution, Travis CI,

only supports Linux and OS X, and does not currently provide Windows support. As a result, automatic build and test validation is not currently performed in a Windows environment when a release build is performed. The use of another CI implementation that supports Windows builds, such as AppVeyor [2], would help ensure Windows builds receive the same level of validation as Linux and OS X builds currently do.

Other future maintenance possibilities include ensuring compatibility with future releases of Alice 3. A future release of Alice may include modified internals that result in an incompatibility with Finch 4 Alice. Any number of changes could be made to Alice which would result in Finch 4 Alice not working as expected. For example, since Swing is essentially deprecated technology, it is conceivable that the existing Swing UI of Alice could be replaced with a JavaFX implementation at some point. Such an update would likely involve sweeping updates to Alice that would almost certainly break compatibility.

6.3 Enhancements to the Finch Representation in Alice 3

One of the first thoughts that came to mind when evaluating the idea of integrating Alice and the Finch robot was that having the on-screen representation of the Finch react and move in response to commands and/or the orientation of the Finch. The initial thought is that there could be one or more options that could control this behavior to enable or disable it as desired by the user.

Another tweak to how the Finch is represented within Alice could be to inject a custom model to represent the Finch instead of extending all STransport subclasses. This same concept was implemented in Finch Dreams, a fork of Alice 2. With Finch 4 Alice, the idea would be to provide a lighter-weight way of extending base Alice with a new Finch model instead of attempting to distribute a competing Alice version.

6.4 Enhance Alice with Functionality Beyond Finch

Another possibility that could be explored could be that of extending the concepts explored by the Finch 4 Alice project to create a generic mechanism for loading new capabilities into Alice 3 via injecting new JAR files into the classpath and exposing new

objects and methods in Alice. There are many limits to what Alice programs can currently do, and enhancing those capabilities could provide potential benefit to the Alice community. Some possible uses for such a mechanism could be to provide support for an HTTP communication library, enable interaction with user input devices beyond keyboards and mice, or even as a means to create and distribute new Alice 3 models beyond the ones shipped in the base product.

This type of enhancement to Alice is unlikely to be simple or intuitive to accomplish with the current release of Alice 3. The ability to invoke such functionality is limited to the methods exposed in the Alice interface. As the Finch 4 Alice project has shown, Alice currently only exposes the methods associated with objects that have a representation in the scene. While overriding an existing Alice class to expose new methods is possible, as shown by Finch 4 Alice, it is not likely to be a viable approach when the concept is extended to multiple custom extensions that may all expect to override the same classes with differing implementations.

REFERENCES

- [1] Stephanos Androutsellis-Theotokis, Diomidis Spinellis, Maria Kechagia, and Georgios Gousios, "Open Source Software: A Survey from 10,000 Feet," *Foundations and Trends® in Technology, Information and Operations Management*, vol. 4, no. 3-4, pp. 187-347, 2011.
- [2] Appveyor Systems Inc. (2017, March) Continuous Integration and Deployment service for Windows developers - AppVeyor. [Online]. <https://www.appveyor.com/>
- [3] Axel von Arnim and VAINSTALL Contributors. (2016, December) VAINSTALL homepage. [Online]. <http://vainstall.sourceforge.net/>
- [4] Lee Benfield. (2016, September) CFR - yet another java decompiler. [Online]. <http://www.benf.org/other/cfr/>
- [5] BirdBrain Technologies, LLC. (2016, August) Finch (Finch API). [Online]. <http://www.finchrobot.com/javadoc/edu/cmu/ri/createlab/terk/robot/finch/Finch.html>
- [6] BirdBrain Technologies, LLC. (2016, July) Finch Dreams | The Finch. [Online]. <http://www.finchrobot.com/software/finch-dreams>
- [7] BirdBrain Technologies, LLC. (2016, August) Finch Robot | The Finch. [Online]. <http://www.finchrobot.com/>
- [8] BirdBrain Technologies, LLC. (2016, July) Github - BirdBrainRobotServer. [Online]. <https://github.com/BirdBrainTechnologies/BirdBrainRobotServer>
- [9] BirdBrain Technologies, LLC. (2017, February) USB Protocol | The Finch. [Online]. <http://www.finchrobot.com/learning/usb-protocol>
- [10] BitRock Inc. (2016, December) BitRock InstallBuilder : Open Source Licenses. [Online]. <https://installbuilder.bitrock.com/open-source-licenses.html>
- [11] BitRock Inc. (2016, December) Cross-Platform BitRock InstallBuilder: Multiplatform Installer Tool. [Online]. <https://installbuilder.bitrock.com/index.html>

- [12] Carnegie Mellon CREATE Lab. (2017, March) CMU-CREATE-Lab/commons-java: Handy, general purpose Java classes. [Online]. <https://github.com/CMU-CREATE-Lab/commons-java>
- [13] Carnegie Mellon CREATE Lab. (2017, March) CMU-CREATE-Lab/finch: Java libraries and software for the Finch robot. [Online]. <https://github.com/CMU-CREATE-Lab/finch>
- [14] Carnegie Mellon CREATE Lab. (2016, July) CREATE Lab Visual Programmer. [Online]. <http://artsandbots.com/visualprogrammer/>
- [15] Carnegie Mellon University. (2016, September) A BirdBrain Idea. [Online]. <http://www.cmu.edu/homepage/computing/2011/spring/finch.shtml>
- [16] Carnegie Mellon University. (2016, July) Alice 3 Software - Carnegie Mellon University. [Online]. <https://www.cmu.edu/homepage/computing/2009/winter/alice-3-software.shtml>
- [17] Carnegie Mellon University. (2016, July) Alice Support/Help / Alice 3 EULA. [Online]. <http://alice3.pbworks.com/w/page/28830524/Alice%203%20EULA>
- [18] Carnegie Mellon University. (2016, July) Alice.org. [Online]. <http://www.alice.org/>
- [19] Carnegie Mellon University. (2016, September) Community Robotics, Education and Technology Empowerment Lab (CREATE Lab). [Online]. <http://www.createlab.ri.cmu.edu/>
- [20] Elliot J. Chikofsky and James H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13-17, January 1990.
- [21] Cristina Cifuentes and K. John Gough, "Decompilation of Binary Programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811-829, 1995. [Online]. <https://en.wikipedia.org/wiki/Decompiler>
- [22] Creative Commons. (2016, July) Attribution-ShareAlike 3.0 United States (CC BY-SA 3.0 US). [Online]. <https://creativecommons.org/licenses/by-sa/3.0/us/>
- [23] Tebring Daly, "Influence of Alice 3: Reducing the Hurdles to Success in a Cs1 Programming Course," Denton, TX, USA, Ph.D. Dissertation 2013.
- [24] Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba, and Steve Cooper, "Mediated Transfer: Alice 3 to Java," in *Proceedings of the 43rd ACM Technical*

Symposium on Computer Science Education, Raleigh, North Carolina, USA, 2012, pp. 141-146. [Online]. <http://doi.acm.org/10.1145/2157136.2157180>

- [25] Dr. Dobb's Journal. (2016, August) Innovative Alice 3 Educational Software Released. [Online]. <http://www.drdoobs.com/innovative-alice-3-educational-software/219400476>
- [26] ej Technologies. (2016, December) ej Technologies: install4j Open Source Licenses. [Online]. <https://www.ej-technologies.com/buy/install4j/openSource>
- [27] ej Technologies. (2016, December) install4j Help : Using install4j With Gradle. [Online]. <http://resources.ej-technologies.com/install4j/help/doc/cli/gradle.html>
- [28] ej Technologies. (2016, October) Multi-Platform Java Installer Builder - install4j. [Online]. <https://www.ej-technologies.com/products/install4j/overview.html>
- [29] Brad Fisher. (2016, December) Finch 4 Alice. [Online]. <http://www.finch4alice.com/>
- [30] Brad Fisher. (December, 2016) IZPACK-1170 - fix issues with Izpack ant task not working - Pull Request #408. [Online]. <https://github.com/izpack/izpack/pull/408>
- [31] Brad Fisher. (2016, December) IZPACK-1301 - Ensure that installData is set on nested conditions before use - Pull Request #409. [Online]. <https://github.com/izpack/izpack/pull/409>
- [32] Brad Fisher. (March, 2017) Pull Request #3 · BirdBrainTechnologies/BirdBrainRobotServer. [Online]. <https://github.com/BirdBrainTechnologies/BirdBrainRobotServer/pull/3>
- [33] Google. (2016, July) Blockly | Google Developers. [Online]. <https://developers.google.com/blockly/>
- [34] Gradle Inc. (2016, December) Gradle Build Tool | Modern Open Source Build Automation. [Online]. <https://gradle.org/>
- [35] Gradle Inc. (2016, December) The Gradle Wrapper. [Online]. https://docs.gradle.org/current/userguide/gradle_wrapper.html
- [36] Michael Hartmeier. (2016, December) Packlet installer tool. [Online]. <http://packlet.sourceforge.net/>

- [37] JFrog Ltd. (2016, December) Bintray jcenter - Maven, Gradle, Ivy, SBT, Groovy, Clojure central repositor. [Online]. <https://bintray.com/bintray/jcenter>
- [38] Caitlin Kelleher. (2016, September) Storytelling Alice. [Online]. <http://www.alice.org/kelleher/storytelling/index.html>
- [39] Gregor Kiczales et al., "Aspect-oriented Programming," in *ECOOP'97 — Object-Oriented Programming: 11th European Conference Proceedings*, Jyväskylä, Finland, October 1997, pp. 220-242.
- [40] Grzegorz Kowal. (2017, January) Launch4j- Cross-platform Java executable wrapper. [Online]. <http://launch4j.sourceforge.net/>
- [41] Tom Lauwers, "Aligning Capabilities of Interactive Education Tools to Learner Goals," Carnegie Mellon University, Pittsburgh, PA, PhD Thesis CMU-RI-TR-10-09, 2010.
- [42] Tom Lauwers, Illah Nourbakhsh, and Emily Hamner, "CSbots: Design and Deployment of a Robot Designed for the CS1 Classroom," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, Chattanooga, TN, USA, 2009, pp. 428-432.
- [43] Lifelong Kindergarten Group, MIT Media Lab. (2016, July) Scratch - Imagine, Program, Share. [Online]. <https://scratch.mit.edu/>
- [44] Jussi Malinen. (2016, October) Github - Swing Explorer 1.6.0. [Online]. <https://github.com/robotframework/swingexplorer>
- [45] Tomasz Moń. (2016, January) USBPCap. [Online]. <http://desowin.org/usbpcap/>
- [46] Barbara Moskal, Deborah Lurie, and Stephen Cooper, "Evaluating the Effectiveness of a New Instructional Approach," in *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA, 2004, pp. 75-79. [Online]. <http://doi.acm.org/10.1145/971300.971328>
- [47] Benjamin Muschko. (2016, December) bmuschko/gradle-izpack-plugin: Gradle plugin that provides support for packaging applications for the Java platform via IzPack. [Online]. <https://github.com/bmuschko/gradle-izpack-plugin>

- [48] Illah Nourbakhsh and Tom Lauwers, "Designing the Finch: Creating a Robot Aligned to Computer Science Concepts," in *Proceedings of the First Symposium on Educational Applications of AI*, Atlanta, Georgia, 2010, pp. 1902-1907.
- [49] Open Installer Contributors. (2016, December) Open Installer framework for building cross platform installers. [Online]. <https://java.net/projects/openinstaller/>
- [50] Oracle Corporation. (2016, July) JAR File Overview. [Online]. <https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html>
- [51] Oracle Corporation. (2017, March) Native (JNA API) - Project Kenai. [Online]. <https://jna.java.net/javadoc/com/sun/jna/Native.html>
- [52] Oracle Corporation. (2016, September) The Java Language Specification. [Online]. <http://docs.oracle.com/javase/specs/jls/se8/html/index.html>
- [53] Oracle Corporation. (2017, April) The Java Virtual Machine Specification. [Online]. <https://docs.oracle.com/javase/specs/jvms/se7/html/index.html>
- [54] Igor Pavlov. (2016, July) 7-zip.org. [Online]. <http://www.7-zip.org/>
- [55] PicoContainer Contributors. (2016, December) PicoContainer - home page. [Online]. <http://picocontainer.com/>
- [56] Nan C. Shu, *Visual Programming*. New York: Van Nostrand Reinhold, 1988.
- [57] Sonatype, Inc. (2016, December) The Central Repository. [Online]. <http://search.maven.org/>
- [58] teamalice. (2016, September) Sourceforge - Storytelling Alice Modification. [Online]. <https://sourceforge.net/projects/storyalice/>
- [59] The Apache Software Foundation. (2016, December) Apache Ant. [Online]. <http://ant.apache.org/>
- [60] The Apache Software Foundation. (2016, December) Maven - Welcome to Apache Maven. [Online]. <https://maven.apache.org/>
- [61] The Eclipse Foundation. (2016, October) AspectJ - Bytecode weaving, incremental compilation, and memory usage. [Online]. <https://eclipse.org/aspectj/doc/next/devguide/bytecode-concepts.html>

- [62] The Eclipse Foundation. (2016, October) AspectJ Configuration - Load-Time Weaving. [Online]. <https://eclipse.org/aspectj/doc/next/devguide/ltw-configuration.html>
- [63] The Eclipse Foundation. (2016, October) The AspectJ Project. [Online]. <https://eclipse.org/aspectj/>
- [64] The FreeBSD Project. (2016, December) The 2-Clause BSD License. [Online]. <https://opensource.org/licenses/BSD-2-Clause>
- [65] Travis CI, GmbH. (2016, December) Travis CI User Documentation. [Online]. <https://docs.travis-ci.com/>
- [66] University of California at Berkeley. (2016, July) Snap! (Build Your Own Blocks) 4.0. [Online]. <http://snap.berkeley.edu/>
- [67] Wikipedia Contributors. (2016, July) Wikipedia - Alice (Software). [Online]. [https://en.wikipedia.org/wiki/Alice_\(software\)](https://en.wikipedia.org/wiki/Alice_(software))
- [68] Wikipedia Contributors. (2016, July) Wikipedia - Application Programming Interface. [Online]. https://en.wikipedia.org/wiki/Application_programming_interface
- [69] Wireshark Contributors. (January, 2016) Wireshark - Go Deep. [Online]. <https://www.wireshark.org/>
- [70] Maxim Zakharenkov. (2016, October) Github - Swing Explorer Source Code. [Online]. <https://github.com/brocchini/swing-explorer>

Appendix

Disclaimers

The Finch 4 Alice project is not affiliated with either the Finch or Alice 3 projects or their respective intellectual property holders.

The Finch robot is produced by BirdBrain Technologies LLC. For more information visit <http://www.finchrobot.com/> and <http://www.birdbraintechnologies.com/>.

Finch 4 Alice interacts with the BirdBrain Robot Server which is covered by a Creative Commons Attribution-ShareAlike 3.0 Unported License [22].

Alice 2 and Alice 3 are developed by Carnegie Mellon University. More information can be found at <http://www.alice.org/>.

Finch 4 Alice BSD 2-Clause License

Finch 4 Alice is released under the BSD 2-Clause License

Copyright (c) 2015, Brad Fisher

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.