# A Code Reuse Interface for Non-Programmer Middle School Students

**Paul Gross[1], Micah Herstand[1], Jordana Hodges[2], Caitlin Kelleher[1]**

Washington University in St. Louis[1]
One Brookings Dr., St. Louis, MO 63130
{grosspa, herstandm, ckelleher}@cse.wustl.edu

The University of North Carolina at Charlotte[2]
9201 University City Blvd, Charlotte, NC 28223
jwhodge1@uncc.edu

## ABSTRACT

We describe a code reuse tool for use in the Looking Glass IDE, the successor to Storytelling Alice [17], which enables middle school students with little to no programming experience to reuse functionality they find in programs written by others. Users (1) record a feature to reuse, (2) find code responsible for the feature, (3) abstract the code into a reusable Actionscript by describing object "roles", and (4) integrate the Actionscript into another program. An exploratory study with middle school students indicates they can successfully reuse code. Further, thirty-six of the forty-seven users appropriated new programming constructs through the process of reuse.

### Author Keywords

Code reuse, non-programmer, middle school, Looking Glass, Storytelling Alice.

### ACM Classification Keywords

H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

## INTRODUCTION

Middle school is a critical time when many students, particularly female students, decide whether or not they are interested in pursuing math and science based careers [7, 36]. At a time when the gap between male and female participation in undergraduate computer science is widening [37], the rarity of computer science teachers and opportunities to explore computing at the middle school level is unfortunate. Programming environments that provide a motivating and supportive context for learning to program may help increase the number of students interested in exploring computing.

Prior research on programming environments demonstrated that storytelling can provide a compelling context to learn computer programming in, particularly for middle school girls [17]. A formal study of girls' programming behavior found that users of Storytelling Alice were more than three times as likely to sneak extra time to program than users of a non-story based version of the same environment [18]. While encouraging, this study focused solely on users' first two to three hours of programming.

Enabling middle school students to learn new skills by reusing and adapting each others' code may enable more middle school students to explore computer programming longer term. Towards this goal, this paper introduces an interface for middle school students to reuse others' code without requiring they understand the program code.

Imagine that a user named Eva is creating an animation that tells a story about a girl, who develops super strength (Melly). Eva remembers seeing a story in which a secret agent character jumped into an evil doctor and the doctor toppled over. Eva wants Melly to jump into a house and knock it over.

To enable this, our code reuse tool guides users through the processes of *selecting* and *integrating* code [13]. Specifically users:

1. Record the execution of the program containing the functionality of interest.
2. Identify the beginning and ending of the functionality of interest.
3. Abstract the code by describing the roles that each character in the functionality plays.

This abstracted version of the code is saved as an Actionscript. To use the Actionscript, a user selects characters from the new program to act out the roles in the Actionscript.

To explore the potential for code reuse tools in a social learning environment, we conducted an exploratory study in the context of a summer science camp for at-risk middle school students. We found that users were able to successfully reuse code with our tool. Further, the process of selecting code for reuse helped some users to develop an understanding of new programming constructs, which they successfully used outside of the context of Actionscripts. A next step, we plan to generate tutorials that will guide users through creating the code in their new program.

## RELATED WORK

Our related work spans two areas: software reuse and end-user program sharing.


*CHI 2009*, April 4–9, 2009, Boston, Massachusetts, USA.

## Software Reuse

Researchers have identified a number of activities that are a part of code reuse [20, 26, 29]. Holmes [13] suggests that code reuse consists of three fundamental phases: *location*, *selection*, and *integration*. While much of the research in code reuse focuses on professional developers [35], some researchers have explored small-scale reuse [4]. Other work explores reuse through demonstration [10, 22].

### *Location*

During the location stage a user searches for software artifacts that may contain source code relevant to their task (not to be confused with searching in source code to determine if part of it is suitable for reuse). Many tools exist for assisting programmers in locating relevant source code (reviews in [14, 23, 27]). Some recent work focuses on finding code examples either on the web [11] or through tools integrated with an IDE [1, 8, 14, 31, 34].

### *Selection*

In the selection phase a user tries to identify the code responsible for functionality of interest, understand it, and determine its reusability.

The process of identifying and understanding the code responsible for particular functionality is difficult for non-programmers [9]. Systems for end user programmers, who typically have limited programming experience, employ a variety of techniques to help users identify and understand relevant code. FireCrystal [28] enables users to record web browser events and view the Javascript and CSS code that executed as a result of the events. The WYSIWYT spreadsheet environment [30] helps provides a visualization of cell dependencies. The Whyline [19] allows end-users to pose "why" and "why not" questions about program behavior and receive answers directly related to runtime information.

Program understanding tools typically employ program visualizations to help professional programmers grapple with feature and fault localization (e.g., [21, 24, 32, 33]). Effective use of these visualizations requires knowledge that non-programmers are unlikely to have.

### *Integration*

For the integration phase a user must insert the selected code into their own code and adapt it for their context. For Java developers, Jigsaw [4] evaluates structural and semantic information from a code source to manage dependencies and recreate missing functionality during integration to a new program. CReN [15] and CloneTracker [5] attempt to manage variable references in multiple cloned code locations.

Currently there are no tools for general, end-to-end software reuse [13]. However, d.mix, a recent web programming system, helps users with the selection and integration steps. Using d.mix, a user can identify reusable components on an arbitrary web page, select components to reuse, and integrate them into a working wiki page [10]. Like d.mix, our system supports non-programmers through the selection and integration steps, however d.mix generates static web API calls where our system can capture and reuse dynamic behaviors.

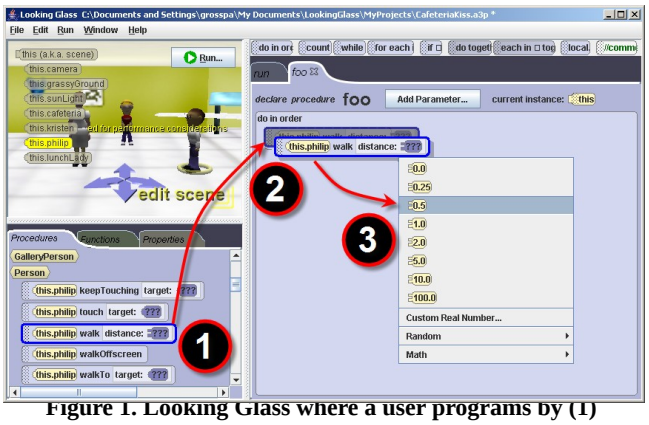



**Figure 1. Looking Glass where a user programs by (1) dragging a method, (2) dropping it into the code pane, and (3) selecting parameters.**

## End-user Program Sharing

There is a long history of designing programming environments for novice programmers [16]. To the best of our knowledge, none of these systems focus on enabling code reuse. Users of MOOSE Crossing could copy and customize or extend scripts written by other users [2]. Scratch [25] directly supports sharing through a web repository, but users share entire programs and there is no integration support for reusing sprite behaviors dependent on other sprites.

## LOOKING GLASS

We built our code reuse tools into Looking Glass, the successor to Storytelling Alice that is currently under development. Like Storytelling Alice, Looking Glass is designed to enable users to create interactive 3D animated stories. To prevent users from making syntax errors, Looking Glass users drag graphical tiles representing methods or programming constructs, drop them into a program editor area, and select parameter values through pop-up menus. Figure 1 shows the process of adding a line of code in Looking Glass. The environment supports common programming constructs such as loops and if-statements. Additionally, the do together construct enables users to have statements in Looking Glass execute concurrently.

## CODE REUSE TOOLS

To enable users to reuse functionality from programs created by others, we must support them in three basic activities:

1. Finding the code responsible for the target functionality.
2. Extracting the responsible code from its original program.
3. Integrating the responsible code into a new context.


**Figure 2. The History tool interface: (1) Time Slider for scrubbing through time, (2) Scene Viewer shows the scene at the selected time, (3) Current Actions Pane shows what actions characters did at the selected time, and (4) Annotated Code View selects the executing line of code, affords block and statement playback and navigational controls.**

For experienced programmers, an intelligent copy and paste system might be sufficient to enable code reuse. However, our goal is to enable new programmers to reuse code without requiring they understand how that code works. In an educational system, this may seem strange. Our eventual goal is to have users select functionality they want to reuse and then complete a tutorial guiding them through building that functionality in their own program. We believe the high level strategy of selecting code without fully understanding it and reconstructing the code to build understanding has two strong advantages:

1. If users build the selected functionality step by step, they will see the impact of each new line and each change. We believe it is easier for users to understand new functionality by constructing it rather than by exploring potentially complex, fully functional code.
2. Users are motivated to build their own stories [17]. We believe that enabling users to construct the new functionality in the context of their own story will align with their natural motivation.

To help users find the code responsible for functionality they want to reuse, we enable users to navigate the code based on observable output. Once users identify the begin and end points of the functionality they want to reuse, they can extract the identified function ality and integrate it into another program using wizard-style interfaces.

### Identifying Functionality for Reuse

Previous research on how new programmers approach the problem of identifying the code responsible for observable output found that new programmers employ a cyclic search process. New programmers (1) identify a search target in the output (or code), (2) search for potentially matching lines of code (or output actions) and (3) repeat (changing from output to code or vice versa) until there are enough matches to form a solution. This process was not very successful for non-programmers; users successfully found the code responsible for a specific output feature only 41% of the time [9].

The study identified several barriers contributing to non-programmers lack of success, including:

1. Users struggle to match their motion descriptions to method names and parameters in the code.
2. Users often fail to fully navigate relevant code.
3. Users do not recognize temporal relationships in programs containing constructs such as loops, do togethers (i.e., a concurrency block), or method calls.

We designed a history tool to address these barriers by helping users identify what code is executing and connect the code with visual output. This history tool is integrated into the code reuse process. The history tool interface includes four components: 1) the Time Slider, 2) the Scene Viewer, 3), the Current Actions Pane, and 4) the Code View Pane (Figure 2).

The *Time Slider* (1) enables users to scrub forward and backward through the program's recorded history.

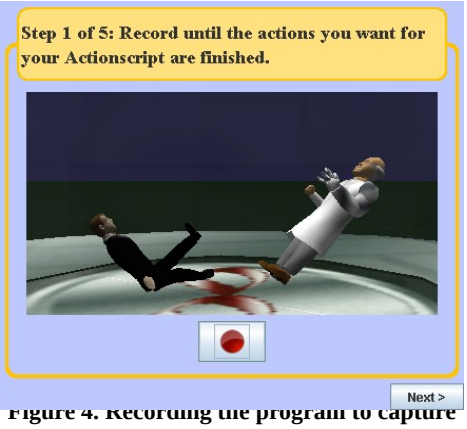



**Figure 4. Recording the program to capture a feature for reuse.**

The *Scene Viewer* (2) displays the scene's appearance at the selected time. To find a particular action of interest, a user can scrub through recorded history using the time slider until he or she sees the action of interest happening in the scene viewer.

The *Current Actions Pane* (3) shows all methods executing at the selected time, organized by character. To determine what a particular character (for example, the LunchLady) was doing at the selected time, the user can expand the "LunchLady's Actions". The expanded view shows which individual statements were executing, and the methods from which they were called. The user can navigate to a particular statement or method call by clicking on it. The Code View Pane will update to display the statement that the user clicked.

Finally, the *Code View Pane* (4) presents a view of the executed code in the latest run of the program. The statement that executed at the selected time is highlighted in green. A play button next to the executing statement enables users to play back all of the images captured while that statement executed. Buttons on the right side of the interface enable users to zoom in on block statements

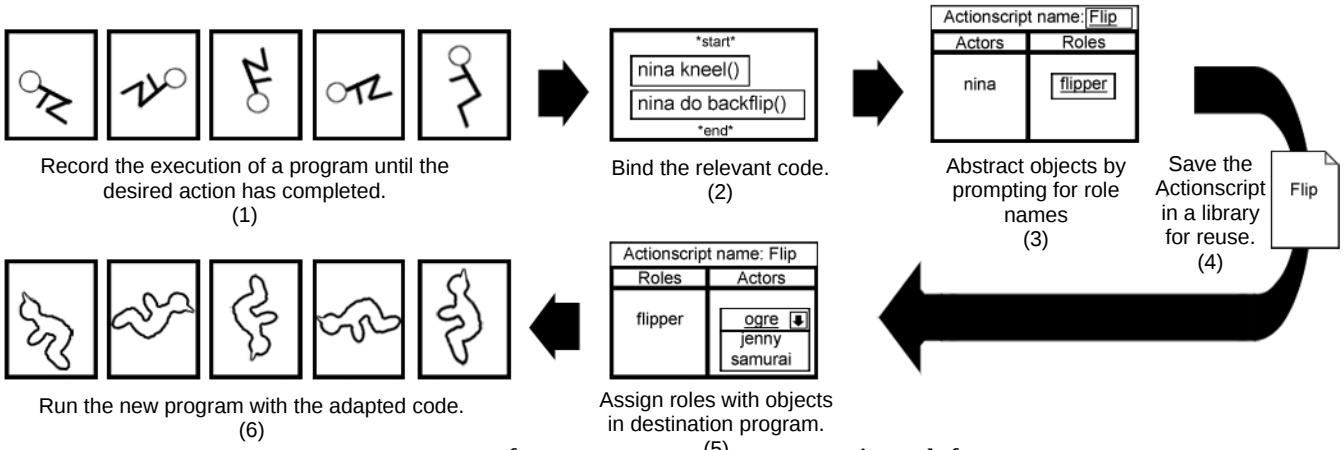



**Figure 3. Overview of process for selecting and integrating code for reuse.**

### Wizards for Extraction and Integration

A wizard style interface guides users through extracting the selected code. This extracted code is abstracted and saved into an intermediary form we call an Actionscript.

A second wizard style interface allows users to select which characters will perform the actions recorded in the Actionscript. After users assign characters to each role in the Actionscript, we generate the code necessary for the actions encoded in the Actionscript.

Figure 3 shows an overview of the process for identifying and reusing a section of code. In the remainder of this section we describe the process of reusing a section of code.

### Usage Scenario – Recording an Actionscript

In the introduction, we described a scenario in which a user named Eva wants to reuse an agent's jumping and knocking over an evil doctor in her super strong Melly story. Creating an Actionscript for the jump and knock over requires five steps that are presented through a wizard like interface

Step 1: Eva records the actions occurring in the agent's story (i.e., the tool constructs a dynamic execution trace) until after the agent's fall is complete. At the bottom of the record panel, Eva clicks a next button (see Figure 4) which loads the history tool to help her find where the agent's jump and fall begins and ends (see Figure 5).

Step 2: To find where the agent began to jump, Eva drags the time slider (Fig. 5a, circle 1) back until she sees the agent start to move up (Fig. 5a, circle 2). At the selected time, the current actions pane (Fig. 5b) indicates the agent and the doctor are active. Users' programs often contain simultaneously executing statements using the do together construct [9].

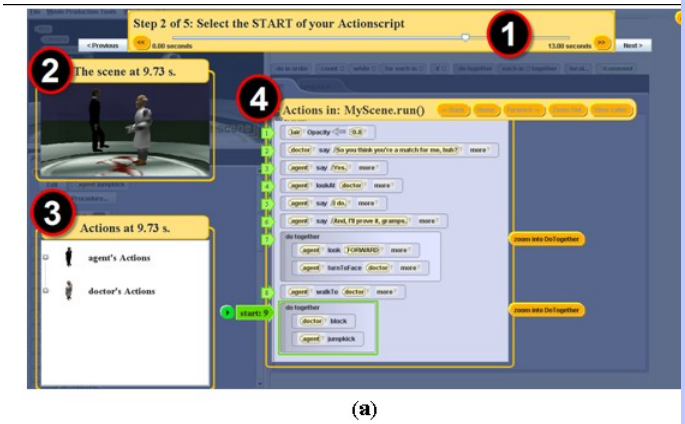
Because Eva is interested in the agent's actions, she expands "agent's Actions" in the Current Actions Pane. In this example, Eva clicks on move up. This action occurs within the method **jumpkick** which is not currently shown in the code view (Fig 5a, circle 4). When Eva clicks the link, the history tool opens the **jumpkick** method in the code view and selects the move up method in green (Fig. 5c). Eva clicks the play button to the statement's left which animates through the series of images recorded while the line of code executed.

At this point, Eva has found the beginning of the actions that comprise the jump and knock over. She selects the "start" arrow next to the move up action to indicate that the code to reuse begins with this action.

Step 3: To find the end action, Eva can scrub forward in time to find where the doctor falls. She can use a similar process to identify the final method call that is a part of the functionality to reuse.

Step 4: With the beginning and ending of the functionality she wants to reuse identified, Eva can play through the selected actions to confirm she selected the correct functionality (see Figure 6). If Eva decides that she prefers a different beginning or ending, she can return to the previous steps to make changes.

Step 5: In the final step, Eva names and describes her Actionscript (see Figure 7). The names and descriptions help her to remember the functionality in the Actionscript and what each character did. Eva names her Actionscript "Jump and Knock Down" and describes the roles each character played in the extracted code. In this example, she describes the agent's role as "Jumper" and the doctor's role as "Thing knocked over". Looking Glass saves the completed Actionscript in Eva's library for later use.




(a)


**Figure 5. (a) The history tool for selecting the beginning of a feature (b) After expanding the object's current actions to see a list of links. (c) After clicking an action link to show a new method in the code view and selecting a statement as the beginning.**

**Figure 7. Abstracting the selected code into an Actionscript by naming object's roles**


**Figure 6. Reviewing an Actionscript to ensure the feature is captured.**

### Usage Scenario-Using an Actionscript

To use her Actionscript, Eva opens the super strength Melly program and her Actionscript library (see Figure 8). She selects the "Jump and Knock Down" Actionscript she recorded earlier.

To incorporate the jump and knock down into her current program, Eva selects the character who she wants to jump and knock over another object (see Figure 9).

Once Eva assigns all roles and presses "Add Actionscript", Looking Glass generates all the code necessary for the chosen actors to perform the actions recorded in the script. The code is stored in a global method called "Jump_and_Knock_Down" which Eva can invoke or edit (see Figure 10). Looking Glass also adds an invocation of Jump_And_Knock_Down to the beginning of the program so Eva may immediately view it by running the program.

## IMPLEMENTATION

Our system for code selection and integration has four basic steps:

1. We capture a full execution trace enabling users to link program statements with observable output in the user interface.
2. Using the beginning and ending statements selected by the user, we prune the execution tree.
3. To enable later integration, we determine the types that can fill roles in the Actionscript.
4. When the user integrates an Actionscript, we generate new code based on which characters and scenery objects the user selects to play each role.

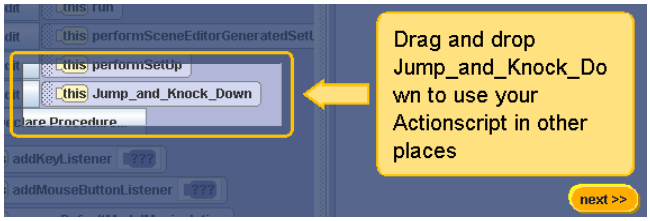



**Figure 10. The Actionscript is added as a global method**


**Figure 8. Choosing to an Actionscript to reuse.**

### Recording an Execution Tree

To enable users to select code responsible for observable functionality, we record a dynamic trace of the program execution. The trace is organized into a tree reflecting the hierarchical program execution. Code block executions and invocations of editable methods make up the internal nodes, while invocations of atomic, un-editable methods naturally form the leaves of the tree. Each node stores an executing statement, its execution period, its thread and its expression evaluations. The expression evaluations allow us to display the actual values used as parameters and the name of target objects in the user interface.

Additionally, we capture screenshots of the output continuously. These screenshots are indexed by the execution timestamp for use in the interface.

### Identifying the Relevant Code to Reuse

To identify the relevant code, we prune the tree based on time constraints and re-insert necessary declarations.

*Pruning with Time Constraints*


**Figure 9. Choosing objects to play the roles that were recorded in the ActionScript.**

Through the user interface, the user selects a start and end statement in the program. We identify the corresponding nodes in the execution tree and use their execution periods as a time range. Statements irrelevant to the user's desired functionality can execute concurrently during this time range (e.g., a character says something during the agent's jumpkick). To prevent the irrelevant statements inclusion,

we find a common ancestor of the start node and the end node. The shared ancestor's child nodes executing during the selected time range are marked as relevant.

If a code section exists as a child of the shared ancestor and does not execute (e.g., a conditional branch that does not execute), it is considered irrelevant. Using this condition ensures only code relevant to what the user observed will be copied. We relax this constraint if a user specifies a start and end nodes with no common ancestor. This is possible in other programming environments supporting user events.

*Re-Inserting Declarations*

Unresolved references can exist in the pruned relevant code. For instance, the first relevant statement may reference the variable speed which is declared before the relevant code. To account for this, we record any local declarations or modifications occurring in scope of the shared ancestor and before the first relevant statement. Then if a relevant statement references an undeclared local variable, we mark any recorded statements declaring or modifying the local variable as relevant and insert them in the Actionscript before all other relevant code. This ensures all local references are accounted for in the integration step.

**Determining Role Integration Types**

To determine which characters and scenery objects can safely fill roles in an Actionscript, we determine a role's lowest type reference. For instance, if the agent invokes a walk method then the agent's role cannot be filled by a table object because the table does not inherit from the class implementing the walk method. However, if the agent only invoked a turn method, a table can take the agent's role because it does inherit from the class implementing turn. We crawl the pruned tree to generate the full set of direct and indirect references to each of the characters and scenery objects in the relevant code. We use the results of expressions evaluated at runtime (e.g., function result, variable access) to resolve indirect references. The expressions' types determine a role's lowest type reference.

We make two exceptions, specific to Looking Glass, in determining a role's lowest type reference: getPart invocations, and user created methods and fields.

The getPart method is not an inherited or abstract method but is implemented in all character classes to get references to parts of a character (e.g., get the agent's head). Character classes are leaves in the class hierarchy tree and thus roles of these types would be unassignable to other objects.

User methods and fields are special because they do not actually exist in the Java code hierarchy. They exist only in the Looking Glass project they are used. However, we can create these methods and fields for any object filling a role. Type safety is ensured for code in user methods because it is also crawled for role references.

With crawling complete the extracted code is parameterized by the roles with all local and dynamic dependencies accounted for. This parameterized code is stored in a file as one parent method, corresponding to the shared ancestor, along with role information. This allows an Actionscript to be fully self-contained and reusable.

**Integrating an Actionscript**

When the user selects an Actionscript to use in a program, the user chooses objects to fill the roles and to resolve any problems that arise with the parts of objects.

Using the roles' lowest type information, we present users with a list of characters and scenery objects that can safely fill the role. Once all roles are assigned, we create a map of roles and the objects filling those roles. A copier takes the map, substitutes all references, and copies all code into the user's program.

Looking Glass tries to match object parts referenced in a getPart invocation by name (e.g., if a magician's hat is used and the user selects a samurai who has a hat to fill magician's role, then the samurai's hat is substituted). If no matching name is found, Looking Glass prompts the user to choose a substitute part. This prevents the program from generating a run-time error because the new object does not have the same body parts as the original object.

All copied code is placed into a user method declared by the Scene object (i.e., a global object in all Looking Glass programs) and given the name of the Actionscript. This avoids scoping issues as all objects in a Looking Glass program are fields of the Scene. By declaring the method in the Scene's class it essentially becomes a global method the user can invoke in any context. With all local and dynamic references previously managed by the crawler, the copying process is deterministic, creating user methods and fields in other classes as necessary.

**Implementation Limitations**

This approach has two major limiting factors: design choices focused on supporting middle school students and the need for visual feedback.

Because Looking Glass is designed for middle school students, we elected not to support full inheritance through the programming environment. Hence our implementation cannot resolve class hierarchy conflicts like other systems for integration [4]. For instance, iterating over an array of Samurais to invoke the Samurai class method backflip cannot be abstracted to a super class with a backflip method for two roles to inherit from. Thus we cannot assign two different types of objects to fill the Samurais' roles later.

Our intended audience for this tool is non-programmers and we assume they identify desired functionality by observing visual output. This assumption limits the generality of our approach to systems with visual feedback relating directly to code, such as dynamic web page behaviors.

## CODE REUSE EVALUATION

To evaluate the potential use for our code reuse tools, we conducted an exploratory study to answer several questions:

1. Can middle school users with little or no previous programming experience successfully reuse code?
2. Does the process of selecting and reusing code help middle school users extend their programming skills?
3. Will animations propagate through social networks?

### Participants

We conducted this study within the context of a class for the Exxon Mobil Bernard Harris Summer Science Camp held at Washington University. The camp provides opportunities to explore science and engineering for students with potential to succeed but who may be at risk due to limited academic opportunities in their school, family problems, or other issues. The camp works with St. Louis teachers to identify student who may benefit from attending. Campers are accepted based on teacher recommendations and an essay. The forty-seven students attending the camp this summer were rising sixth through eighth graders. The group was balanced by gender and predominantly African-American.

### Camp Course

During the two week camp we oversaw two one week classes: the first for twenty-four students and the second for twenty-three students. Each class was to include four two hour sessions. The first week was limited to three sessions due to a network outage that made the lab unusable.

#### *Evaluating Learning*

To explore the potential for students to learn new programming skills through reusing code, we intentionally limited the formal instruction we provided the students. During the first session we demonstrated adding a 3D character, making characters perform actions by dragging and dropping method invocations, and running the program. We also introduced students to our code reuse interface.

To enable students to teach themselves new programming concepts through reuse, we provided each student with three example programs on each of the first three days. We designed the three example programs to contain captivating animations to be reused in other stories, and illustrate a set of focus concepts for that day. The focal language concepts and constructs for each day are listed below in Table 1. Each day, the students had access to the programs for the current day and all the previous days for that session.

To ensure example code was students' primary learning resource, researchers did not provide assistance to students. Instead, researchers responded to help requests by suggesting a relevant program in the example library.

#### *Social Propagation*

We are also interested in whether program functionality may spread through different social groups. We created two different types of groups: working groups and presentation groups. While students actively built programs they sat with members of their working group. Toward the end of each session, we asked students to show the project they created that day to the members of their presentation group. Each working group contained eight students. Each presentation group contained six students.

On the first three days of each class, we asked each student to create a new program that incorporated two animations reused from other programs. Each day we created two example programs from which all working groups could reuse animations. We also provided a third example program, unique to each working group, intended to seed novel animations into the presentation groups.

Presentation groups contained two students from each working group. Students gathered in their presentation groups to view other student's programs constructed that day. We encouraged students to reuse animations they found captivating in their presentation group and offered prizes to anyone whose unique animation was reused more than once by other students.

#### *Data*

We collected three types of data: Actionscripts that participants captured, programs using those Actionscripts, and participants' performance on a programming quiz given during the final session of each camp class.

#### *Scripts and Programs*

Collecting participants' Actionscripts and programs enabled us to gather qualitative information about how users reuse code with our system, what kinds of actions they capture, and how they use those actions within their own creations. We are interested in the potential for users to learn new programming constructs or concepts through the process of reuse. To ascertain the extent to which this happens, we collected quantitative information about the constructs participants use through their Actionscripts and independently in their programs.

#### *Post-camp quiz*

At the end of the final session, participants took an eleven item forced-choice programming quiz in which we asked them to predict the behavior of short segments of Looking Glass code. Each question presented a small snippet of code and a series of four or five textual descriptions of how that code might behave. We asked participants to select the best description for each snippet of code. The questions on the quiz covered simple uses of sequential and parallel programming, count and while loops, iterating over a list, parameter passing, and method calls.

| | |
|---|---|
| Day 1 | User methods, count loop, do together |
| Day 2 | If/else, while loop, functions |
| Day 3 | Sequential blocks in do togethers, sequential iteration over a list, parallel execution over a list |

**Table 1. Concepts and constructs in daily example programs**

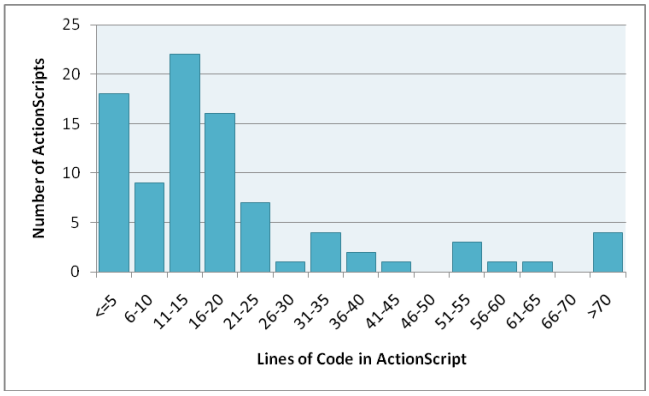



**Figure 11. Frequency of Actionscript size (lines of code)**

Based on an exploratory factor analysis of the quiz questions, we created a programming quiz scale that included six questions loading on the same factor (Cronbach's $\alpha = 0.60$). This factor reflects participants' ability to predict basic constructs behavior including sequential and parallel execution, parameter passing, loops, method calls, and iterating over lists. An additional two questions covered more advanced programming constructs: if statements and parallel execution over a list.

### Results

Forty-six of our forty-seven participants successfully captured and reused code. The majority of their Actionscripts, 77%, contain more than 5 lines of code (Figure 11) which we consider non-trivial functionality. Typical script content focused on a single character or an exchange involving a small group of characters. This content indicates a focus on capturing functionality which can be used in a story that may be unrelated to the source.

### Learning Through Reuse

We measured learning in two ways: 1) participants' modification or use of new programming constructs in their programs and 2) participants' performance on the post-workshop programming quiz.

*Modification and Use of Programming Constructs*

Thirty-six of our forty-seven participants either modified or independently used programming constructs which were only introduced through the process of reusing code.

Although our example programs used a variety of constructs, we saw a strong concentration of Do Togethers and Loops in the users' Actionscripts. This construct homogeneity limited the scope of learning. However, we observed evidence that participants used creating and modifying Actionscripts to learn new skills (see Table 2). Table 2 shows participants usage of programming constructs represented in the example programs.

We observed three levels of programming construct usage. At the most basic level, users created Actionscripts including a particular construct. In the next level users explored the code created by Actionscripts in their programs and modified one or more programming constructs. At the most advanced level, users created new sections of code which included programming constructs or techniques they discovered through reuse. Due to the emphasis on Do Together in the captured Actionscripts, we see the greatest number of modifications and independent uses of DoTogether statements. However, users also frequently learned how to call functions to request and animate the individual body parts of characters.

| Programming Group | Basic Constructs | Advanced Constructs |
|---|---|---|
| Low (n = 11) | 3.9 | 0.7 |
| High (n = 36) | 4.7 | 1.0 |

**Table 3: Average number of correct answers on Post-camp Quiz by Programming Group**

| Programming Construct | User Captured in Used Script | User Modified | User Added |
|---|---|---|---|
| Property Assign. | 31 | 4 | 2 |
| Function Call | 38 | 1 | 10 |
| Do Together | 40 | 10 | 28 |
| Count Loop | 25 | 1 | 1 |
| While Loop | 2 | 0 | 0 |
| For Each In Order | 3 | 0 | 0 |
| Each In Together | 1 | 0 | 2 |
| User Method Call | 39 | 0 | 2 |

**Table 2. Number of participants who captured a construct in an Actionscript they used, modified the construct in a program using the Actionscript, or added it independently.**

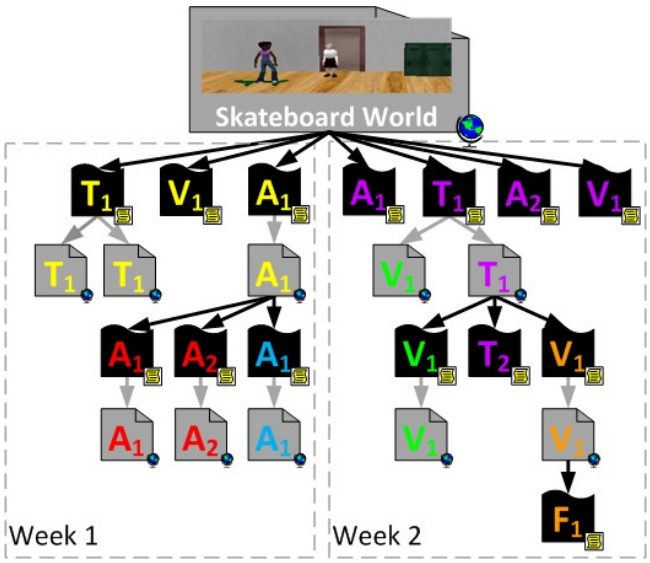



**Figure 12. Reuse originating from Skateboard World. Black wave shapes are scripts and grey shapes are worlds. Labels show creator with colors showing a user's working group and letters representing their presentation group.**

*Programming Quiz Performance*

We expect that learning will come primarily through modifying code from an Actionscript or using constructs discovered in an Actionscript to accomplish other goals. To investigate this, we divided our participants into two groups based on their programming behavior (see Table 3). The Low programming group (11 participants) includes participants who made no modifications and did not use new programming constructs elsewhere in their programs. The High programming group (36 participants) includes participants who either modified the code in Actionscripts or used programming constructs introduced through the Actionscripts in other places of their program. Par ticipants in the High programming group performed better on both the programming quiz scale ($r=.267$, $p < .1$) and the two questions on advanced constructs ($r = .295$, $p < .05$).

*Discussion*

Our results indicate a correlation between reuse and construct learning. However, we have no evidence of a casual relationship because construct learning is incidental with this reuse process. These results do support further investigation of reuse with tutorial reconstruction as an explicit learning mechanism.

### Social Propagation of Functionality

We did see animations propagate through the camp's working and presentation groups. Figure 12 shows how participants captured and reused functionality from one source program *Skateboard World*. During the first week of camp, three yellow working group users made Actionscripts containing animations from *Skateboard World*. Two of the three then created worlds using their captured Actionscripts. At the end of the first day, these users shared their programs with their presentation groups. During the second day, users from the red and blue working groups captured new Actionscripts based on a yellow user's world and used them in their own programs. We see a similar reuse pattern for the purple working group and this world during week two.

This social propagation of functionality may provide a social channel through which users of Looking Glass can teach each other new programming constructs.

### Study Limitations

Study participants came from a pre-selected pool of at-risk middle school students and took place in a highly motivated setting. We believe our results can generalize to middle school students of any ability level, but do require motivating circumstances. To use Looking Glass and the reuse process requires motivation, either extrinsic or intrinsic, to overcome the gradual learning curve.

## FUTURE WORK

Our interface enables middle school students to successfully select and integrate code. Further, the results of our exploratory study suggest the reuse process helps users to build their repertoire of programming constructs.

Currently, our system does not support any customization or modification of an existing Actionscript. However, several users asked about modifications such as including or excluding roles from an Actionscript or changing a property that is set within the Actionscript. To enable this type of interaction, we need to first understand what and how non-programmers want to modify and customize their Actionscripts. Enabling users to remove a role or modify a specific action may increase the reusability and utility of Actionscripts. The challenge lies in communicating with non-programmers about these modifications.

Participants in our study selected Actionscripts that included only a subset of the programming constructs represented in the example set. We are considering an interaction for watching and automatically recording from a stream of Looking Glass programs called Looking Glass

TV. A user simply stops watching when they see something they want and they can immediately begin the reuse process. By maintaining a history of programming constructs with which the user has experience, we may be able to select examples for Looking Glass TV that help to introduce new programming concepts.

While users seem to gradually increase their repertoire of programming constructs through reuse, we believe the process of constructing the code for an Actionscript may make the learning process faster and more effective. We plan to develop an automatic tutorial system that can guide users through building the code for a given Actionscript.

## REFERENCES


1. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. Example-Centric Programming: Integrating Web Search into the Development Environment. Stanford University Technical Report, CSTR-2009-01.
2. Bruckman, A. MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids. MIT Media Lab. Boston, MA., 1997.
3. Cottrell, R., Chang, J., Walker, R.J., and Denzinger, J. Determining detailed structural correspondence for generalization tasks. Proc. *SIGSOFT*, (2007), 165-174.
4. Cottrell, R., Walker, R.J., and Denzinger, J. Semi-automating small-scale source code reuse via structural correspondence. *Proc. SIGSOFT* (2008), 214-225.
5. Duala-Ekoko, E. and Robillard, M.P. Tracking Code Clones in Evolving Software. *Proc. ICSE* (2007), 158-167.
6. Fischer, G., Henninger, S., and Redmiles, D. Cognitive tools for locating and comprehending software objects for reuse. *Proc. ICSE* (1991), 318-328.
7. Gill, J. Shedding Some New Light on Old Truths: Student Attitudes to School in Terms of Year Level and Gender. *Proc. of the American Educational Research Association.* (1994).
8. Goldman, M. and Miller, R. Codetrail: Connecting source code and web resources. *Proc. VL/HCC* (2008), 65-72.
9. Gross, P. and Kelleher, C. Non-programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers. *Proc. VL/HCC* (2009), 75-82.
10. Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. Programming by a sample: rapidly creating web applications with d.mix. *Proc. UIST*, ACM (2007), 241-250.
11. Hoffmann, R., Fogarty, J., and Weld, D.S. Assieme: finding and leveraging implicit references in a web search interface for programmers. *Proc. UIST*, ACM (2007), 13-22.
12. Holmes, R., Walker, R., and Murphy, G. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Trans. On Soft. Eng.* (2006), 952-970.
13. Holmes, R., Cottrell, R., Walker, R.J., and Denzinger, J. The End-to-End Use of Source Code Examples: An Exploratory Study. *Proc. ICSM*, (2009), to appear.
14. Holmes, R., Walker, R.J., and Murphy, G.C. Strathcona example recommendation tool. *Proc. ESEC/FSE*, ACM (2005), 237-240.
15. Jablonski, P. and Hou, D. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. *Proc. OOPSLA*, ACM (2007), 16-20.
16. Kelleher, C. and Pausch, R. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv*. 37, 2 (2005), 83-137.
17. Kelleher, C. and Pausch, R. Using storytelling to motivate programming. *Comm. of ACM* 50, 7 (2007), 58-64.
18. Kelleher, C., Pausch, R., and Kiesler, S. Storytelling alice motivates middle school girls to learn computer programming. *Proc. CHI*, ACM (2007), 1455-1464.
19. Ko, A.J. and Myers, B.A. Designing the whyline: a debugging interface for asking questions about program behavior *Proc. CHI*, ACM (2004), 151-158.
20. Krueger, C.W. Software reuse. *ACM Comput. Surv*. 24, 2 (1992), 131-183.
21. Lanza, M. CodeCrawler-lessons learned in building a software visualization tool. *Proc. CSMR*, (2003), 409-418.
22. Leshed, G., Haber, E.M., Matthews, T., and Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise. *Proc. CHI*, ACM (2008), 1719-1728.
23. Lucredio, D., Prado, A., and de Almeida, E. A survey on software components search and retrieval. *Proc. Euromicro Conference* (2004), 152-159.
24. Lukoit, K., Wilde, N., Stowell, S., and Hennessey, T. TraceGraph: immediate visual location of software features. *Proc. ICSM*, (2000), 33-39.
25. Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., and Resnick, M. Scratch: a sneak preview [education]. *Proc. C*$^5$ (2004), 104-109.
26. Mili, H., Mili, F., and Mili, A. Reusing software: issues and research directions. Software Engineering, *IEEE Trans. on Soft. Eng.* 21, 6 (1995), 528-562.
27. Mili, A., Mili, R., and Mittermeir, R. A survey of software reuse libraries. *Annals of Soft. Eng.* 5, 1 (1998), 349-414.

28. Olney, S. and Myers, B. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. *Proc. VL/HCC*, (2009), 105-108.

29. Prieto-Diaz, R. Status report: software reusability. IEEE *Software* 10, 3 (1993), 61-66.

30. Ruthruff, J., Creswick, E., Burnett, M., et al. End-user software visualizations for fault localization. *Proc. SoftVis*, ACM (2003), 123-132.

31. Sahavechaphan, N. and Claypool, K. XSnippet: mining for sample code. SIGPLAN Not. 41, 10 (2006), 413-430.

32. Schafer, T., Eichberg, M., Haupt, M., and Mezini, M. The SEXTANT Software Exploration Tool. *IEEE Trans. on Soft. Eng.* 32, 9 (2006), 753-768.

33. Storey, M. and Muller, H. Manipulating and documenting software structures using SHriMP views. *Proc. ICSM*, (1995), 275-284.

34. Thummalapenta, S. and Xie, T. Parseweb: a programmer assistant for reusing open source code on the web. *Proc. ASE*, ACM (2007), 204-213.

35. Yongbeom Kim and Stohr, E.A. Software Reuse: Survey and Research Directions. *Journal of Management Info. Sys.* 14, 4 (1998), 113-147.

36. Zimmer, L. and Bennett, S. Gender differences on the California statewide assessment of attitudes and achievement in science. *Proc. of the American Educational Research Association*, (1987).

37. Zweben, S. 2007-2008 Taulbee Survey. *Computing Research News* 21, 3 (2009), 8-23.