Proposal: Concreteness Fading and Visual Programming in Teaching Object-Oriented Programming

Andy Jiang, Michael Mauer, and David Li

1 Problem Definition

Much effort has gone towards methods to teach programming as an overall concept, with systems like Alice, Scratch, and CodeSpells demonstrating how visual programming can successfully introduce students to this field. Our goal is to teach the more specific topic of object-oriented programming to novice programmers using these same techniques, focusing on how to abstract and represent ideas such as inheritance, polymorphism, and interfaces in such a framework. Additionally, to reinforce these concepts to an audience already somewhat familiar with programming, we will introduce concreteness fading to the system, transitioning students from visual programming to directly writing code. This will facilitate the learning of these specific higher-level concepts and abstractions within computer science, which is important to effectively educate and train the next generation of computer science and software development students.

2 Approach

Our approach is to develop a game based on visual programming, where the immediate objective is to manipulate various objects within a world to accomplish certain goals. Tentatively, the main theme will be controlling a robot to gather resources, build new robots (teaching object instantiation, design patterns), writing code for new robots (teaching inheritance, encapsulation, etc.), or terraforming the environment. Students will control the robot and other objects via a block-based visual programming interface akin to Scratch. However, one novel idea is that the interface will be designed to express object-oriented concepts, making clear ideas such as method invocation, object instantiation, and polymorphism. Additionally, the other novel idea is to use concreteness fading to demonstrate the link between the abstracted representation and the underlying code: the system will translate the visual representation to Java code, showing its execution in tandem with the symbolic execution of the blocks and the effect of the code in the world. The system will highlight the visual block being executed, as well as the corresponding line of code. Eventually, the system will fade the visual representation, asking students to directly write more and more of the code itself.

Specifically, we intend to integrate various visual metaphors for object-oriented concepts within the visual programming interface. For example, class definitions would be represented as “blueprints” containing lists of methods, which students would drag

into “tell” blocks that represent the message-passing style of OOP. Students would also add an object reference to the “tell” block, which the system would translate into method invocation on the instance. Each class would have an associated pictorial representation, which would be attached to object reference blocks in the editor, and which would also be used to represent instances of that object in the game world grid. Later on, this representation could be faded by removing the picture and/or morphing the tell block into the dot syntax used by Java.

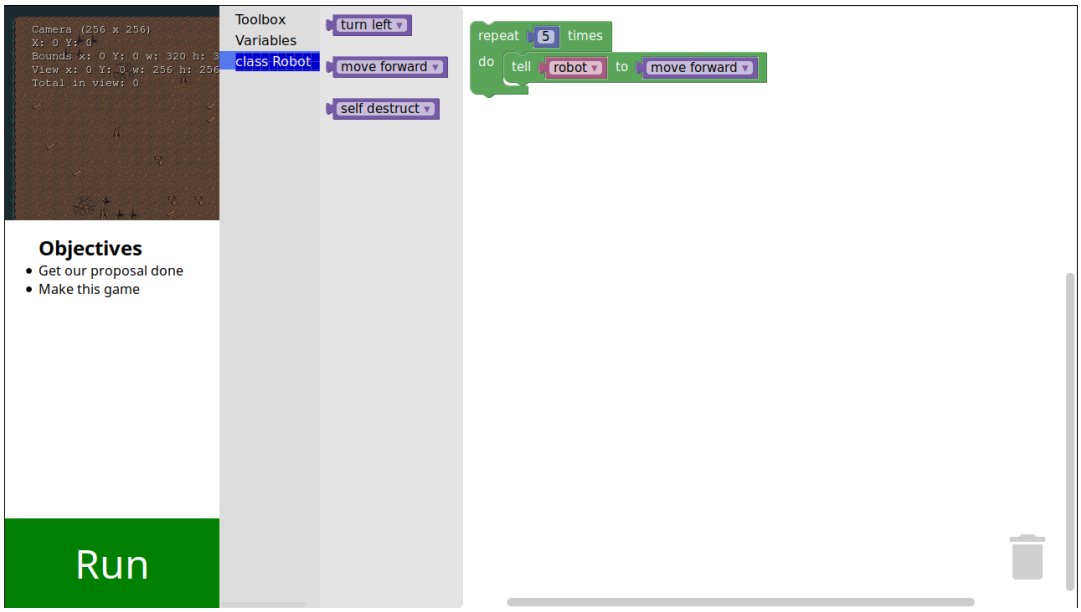

Figure 1: The main coding interface.

Based on feedback from the paper prototype, the user interface has been redesigned. There are two main activities: editing code and executing code. When editing code, the user will have access to the toolbox of unlocked blocks, an area to assemble them, and a small view of the world map and objectives. Once the student is done, they will run the code, at which point the toolbox will shrink, the map will expand, and the game will begin stepping through their code while simultaneously displaying their side effects on the world map. We removed the interactive Java debugger, as students would likely ignore it. Instead, we plan to fade Java code into the blocks themselves: at first some blocks will begin displaying code on them; later, some blocks will be translated into code while executing.

Here the “blueprint” metaphor shows the student what methods are available, and the object instance they are controlling is highlighted on the map. The `self` block (`this`

Figure 2: The interface while code is running.

in Java) is annotated with the same picture as the blueprint and map.

2.1 Level Design and Progression

Our rough concept progression can be found in figure 3.

The game will begin and feature a series of interactive tutorials. When introducing a concept, students will be guided through using the new idea to accomplish their objectives. In particular, most block types and capabilities will be slowly introduced, based on feedback from the paper prototype, where the function and intent of many blocks was unclear when presented without some sort of accompanying explanation.

The actions the player can take, in terms of game mechanics, are simple: gathering resources, instantiating robots, constructing buildings in their base, and interacting (via code) with other environmental objects. Gathering resources allows them to instantiate more robots/build buildings; buildings may increase the number of robots they can support or grant access to new types; environmental objects may advance them in the game's storyline. Overall, the mechanics here are not intended as an end unto themselves—this is not a game about gathering minerals and gas. Instead, a set of objectives will be packaged along with a tutorial and guidance into a self-contained level, with each level building upon the last.

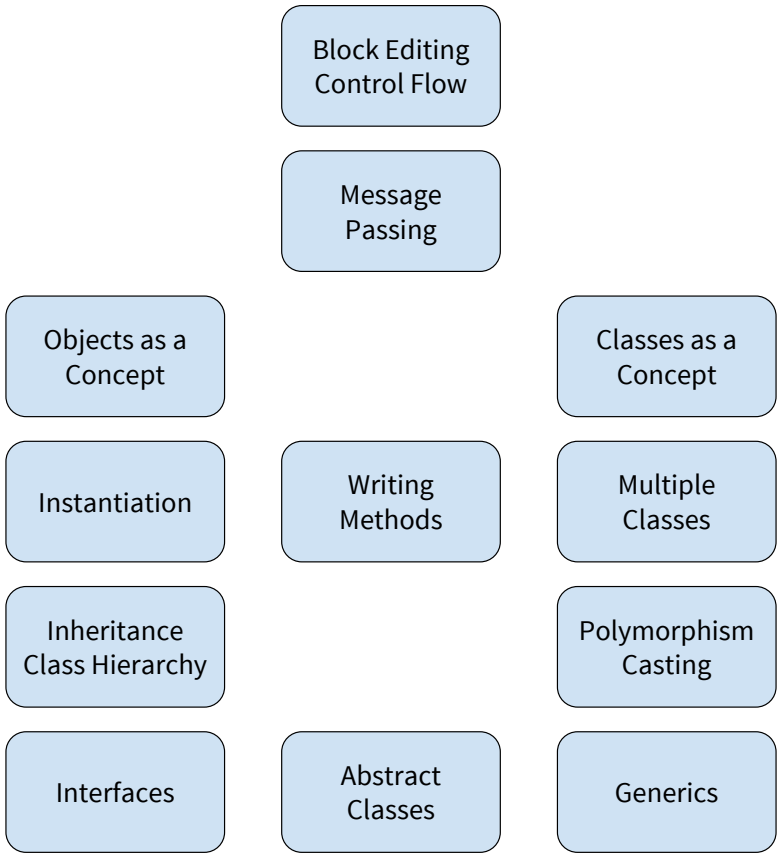

Figure 3: The concept progression. Items within a row do not necessarily depend on each other, but each row depends on the concepts in the previous row.

In terms of editing code, the main actions the user can take are writing code in the main controller class, which issues commands to the various robots; defining new methods on robots to implement various required functionality (e.g. being able to control a robot arm); and defining new robot classes, to make specialized robots for certain objectives.

As for the challenges themselves, the basic flow will be to:

1. Instantiate (or be given) new robot(s) in the Controller class
2. Give commands to the robot(s) to:
    - Pick up resources
    - Activate tools to extract resources/terraform

- Bring things back to base/a target location
- Activate tools to construct buildings
- Avoid obstacles/traps

3. Implement new methods and classes, to fill in missing functionality or specialize a robot for a task. For instance, the robot may not be able to make use of a drill until it is subclassed and methods are added to power on the drill, control its speed, power it off, etc.

Some example tasks, with difficulty levels and rationale, are given below. In general, more difficult tasks involve at least one of the following: more concepts, more objectives, and less scaffolding/tutorial material.

Task	Description	Difficulty
Gathering Iron	Behind a hill and out of sight of the player is located a deposit of iron. Command the robot to search around the base of the hill for the iron and bring it back to base.	Easy—this is primarily about control flow, with method invocations necessary to command the robot.
Two Left Wheels	One of the robots you have can only turn left. Implement a `turnRight` method for it.	Easy—this is a very basic method definition.
Robot Rescue	One of your robots has broken down; instantiate a new robot, send it over, call a method to get a reference to the broken robot, and reboot it.	Medium—students need to work with multiple objects, instantiation, and understand that methods can return object references.
Fracking for Gas	Coordinate a pump robot and a collector robot to extract natural gas from a deposit.	Medium—students have to manipulate two objects of distinct classes, and must understand why methods are available on one but not the other.
General-Purpose Robot	Given a `Tool` superclass, design a robot that is given some `Tool` and can activate and use it without knowing the actual subclass. Use these robots with various tools to mine resources and build a rocket.	Hard—students have to understand objects as representing abstract concepts.

Our system simply requires the student to accomplish the given objectives, using the tools given to them. Judging object-orientedness is not a goal of our system: instead;

our system essentially only allows object-oriented programming. The only non-OOP blocks are control flow blocks, and levels will be designed such that students will be required to implement methods, create subclasses, and so on. Of course, we will have interactive tutorials for each concept introduced as well, to walk students students through the new blocks and ideas.

3 Prior Work

Compared to prior systems like Scratch[1] and Lightbot[2], our system will explicitly model object-oriented concepts. For instance, LightBot provides a fixed set of actions that implicitly operate on the robot and space for a single subroutine. In contrast, our system would allow user-defined methods on multiple classes, as well as multiple different types of objects and control over multiple objects simultaneously. Compared to Scratch, while our block system is (intentionally) modeled after theirs, again, our system explicitly shows the objects and methods being invoked, while in Scratch scripts are attached to a particular sprite and implicitly act on that sprite. In other words, while Scratch has all actions take place on `this`, our system will show `this` as well as other objects and allow the user to manipulate them at will.

Google's Blockly Games[3] demonstrates our concreteness fading idea: as the game progresses, blocks first have code appear on them, then the blocks are removed entirely, leaving only code. Our system differs in that we intend to make this transition between blocks and code gradually, migrating the blocks to show code on them slowly, then having some tasks (e.g. the implementation of some methods) be code-only, before ending with code-only. In contrast, Blockly Games makes the jump suddenly; when the user advances to the next set of levels, the blocks suddenly contain code on them, with no explanation. However, Google does have good design advice[4] in the layout of the editor and the format of interactive tutorials (essentially: make them mandatory and only dismissable by completing the relevant objective, otherwise, students will close them on instinct).

4 Evaluation

We intend to test the system with a group of students in a class such as CS 2110, Advanced Placement Computer Science, or similar class involving Java and object-oriented principles. Students could be given a pre- and post- test asking about various concepts from this paradigm. As noted from feedback on the paper prototype, finding an

[1] `https://scratch.mit.edu/`

[2] `http://lightbot.com/hour-of-code-2015-flash.html`

[3] `https://blockly-games.appspot.com/about`, `https://github.com/google/blockly-games`

[4] `https://developers.google.com/blockly/hacking/mistakes`

appropriate target audience is difficult—we need students who have some programming basics, but who have not fully learned OOP. Students from other universities and high schools could be recruited; alternatively, we could target students in classes like CS 1110, revamping the system to use a language besides Java.

As for other data to collect, we can record basic statistics such as the amount of time spent in the game, the amount of time per lesson, the number of tries per lesson, and so on. We could also record their intermediate attempts at a solution.

5 Milestones

5.1 Alpha

The minimum playable product would require:

- Block editor
- World map
- Code execution
- The first lesson

Note: multiply all given values by π, because estimation is hard.

Task	Priority	Person	Hours
Integrate Google Blockly as the block editor	Must have	David	4
Set up Blockly with relevant block types	Must have	David	8
Set up Blockly to generate Java code	Must have	David	2
Find art assets for the world map	Must have	David	2
Render the map using Phaser.js	Must have	David	4
Implement functions to be injected via Blockly, which update the execution state	Must have	Andy	8
Implement “shim” functions to be called by user code, which update the game state	Must have	Andy	8
Launch a JVM and execute user code	Must have	Andy	4
Implement the game state (server-side)	Must have	Michael	12
Implement the game state (client-side)	Must have	Michael	6
Relay the game state via WebSocket to the client	Must have	Michael	2

5.2 Beta

- Level creation tools (to develop content and lessons quickly) & level definition format, i.e. for a given level, creating a file format that defines what blocks are

available, what map to use, what tutorial popups to show when, the victory conditions, etc.

- Class design/creation interface
- Implement concreteness fading: new block types/code editor
- Tutorial/hint capability (to guide players through a lesson)
- More lessons
- Savegame capability

Task	Priority	Person	Hours
Level definition format	Must have	Andy	12
Interpreting level definitions	Must have	David	8
Class hierarchy interface	Should have	Michael	15
Concrete-faded block definitions	Must have	Andy	2
Writing tutorials demonstrating the block fading	Should have	Andy	6
Integrated code editor	Should have	David	4
Tutorial/popup library	Must have	David	8
Savegame capability	Must have	Michael	3
UI polishing	Could have	Anyone[5]	Spare time
Misc. backend features as needed	Should have	Michael	2
Writing more levels (up to writing methods)	Should have	Everyone	10
Storyline	Could have	Michael	Spare time

5.3 Final

- All the lessons, covering up to subclassing/class hierarchy (our target concept)
- Data collection

[5] Pull requests appreciated.

Task	Priority	Person	Hours
Pre/post test	Must have	Andy	10
Server-side data collection	Must have	Michael	4
Client-side data collection	Must have	David	4
Inheritance/hierarchy lessons	Must have	Michael	16
Revamping earlier lessons from Beta feedback	Must have	Andy/David	25
Polymorphism/more advanced lessons	Could have	Anyone	Spare time
UI polishing	Could have	Anyone	Spare time
Storyline	Could have	Michael	Spare time