Green: a pedagogically customizable round-tripping UML class diagram Eclipse plug-in

Carl Alphonce
Department of Computer Science & Engineering
University at Buffalo, SUNY
Buffalo, NY 14260-2000
alphonce@cse.buffalo.edu

Blake Martin
Department of Computer Science & Engineering
University at Buffalo, SUNY
Buffalo, NY 14260-2000
bcmartin@cse.buffalo.edu

ABSTRACT

UML class diagrams are used quite commonly in CS1-CS2 courses and textbooks. The benefits of using these diagrams include providing a programming-language independent way of communicating program design, in an industry standard language. While drawing diagrams by hand is in itself useful, beginning students do not always perceive the benefit of designing before coding, and create these diagrams only if they have to, and then only as an afterthought.

We have found that students are much more receptive to using UML class diagrams as an integral part of their development if they see immediate benefits from doing so. This paper describes *Green*, a simple to use yet flexible and extensible UML class diagramming tool. *Green* (an Eclipse plug-in) provides complete round-tripping between code and class diagram. This capability makes it easy for students to alternate between a detailed code-level view and a more abstract design view of their projects. With this capability students see creating class diagrams not as a separate and tedious activity, but as an easy way to turn designs into code and to discover the design of existing code.

Green's distinguishing features when compared to similar tools are that it has been developed to meet the needs of CS1-CS2 students, the semantics of its relationships are customizable, additional class relationships can be defined and it is integrated with Eclipse, a mature development environment.

1. INTRODUCTION

While programming-first approaches to the introductory CS curriculum continue to be popular [1], there is interest in teaching more than simply coding skills in CS1-CS2. Introductory courses with a broader software development perspective are increasingly common, and incorporate such topics as pair-programming [12, 13], test-driven development [6, 7], design patterns and an iterative coding and design process [3, 4, 5, 10].

Understanding object-oriented design is facilitated by the use of UML class diagrams [2], since these offer a programming language-independent way of communicating design ideas. UML class diagrams are frequently employed in CS1-CS2 courses and textbooks for this reason. Beginning students do not always perceive the benefit of designing before coding, and typically resist drawing class diagrams; they are seen as a distraction from the "productive" work of coding. If a diagram is a static artefact which is disconnected from the coding process, then this perception is understandable. We have all too often seen students sketch a class diagram prior to coding, but never consult it during the coding process, or create a class diagram as an afterthought once coding is complete.

This paper describes a simple-to-use UML class diagramming tool, developed at the University at Buffalo, and designed primarily for use by students in CS1-CS2 courses. The tool is an Eclipse plug-in. It relies on and manipulates the Java Development Tools' (JDT) Java model. In this way it provides a "live" UML class diagram view of a project: the class diagram view listens for changes to the JDT Java model and therefore stays current with changes made in the code base; likewise, any changes to the diagram which result in modification of the JDT Java model are immediately reflected in the code base.

While the tool is designed with the CS1-CS2 population of students in mind, its architecture is flexible and extensible, to allow the functionality of the tool to be tailored to fit the needs of its users: the semantics of the UML class relationships (and indeed the supported set of such relationships) are customizable.

Section 2 discusses some of pedagogical reasons for developing yet another class diagramming tool. Section 3 lists desirable features of a tool for CS1-CS2, makes a comparison of *Green* to a selection of similar tools, to help situate it in a broader context, and explains the tool's architecture in more detail.

2. PEDAGOGICAL ISSUES

In the following subsections we discuss our primary motivators for using class diagrams and a round-tripping class-diagram tool in CS1-CS2.

2.1 Why use UML class diagrams?

At the University at Buffalo (UB) the introductory computer science course sequence (CS1-CS2) is strongly object-oriented.[3, 4] Objects are introduced from the very first lecture and discussion of object-oriented concepts such as inheritance, polymorphism and encapsulation take precedence over discussion of control structures and language-specific syntactic details. In this environment it is natural for students to learn object oriented design and programming using design patterns.

Our goals in these introductory courses therefore include (i) an early introduction to program design, (ii) a requirement that students design and code incrementally, and (iii) a timely presentation of good design solutions in the form of design patterns.

We claim that students must have a programming-language independent vocabulary to communicate program design, including design patterns. Using programming language constructs alone is inappropriate for several reasons: it forces students to make discussions unnecessarily concrete and make them worry about language details irrelevant to the design issues, and it ties their thinking of design and patterns to a specific programming language, making it more difficult for them to transfer their understanding when working with different programming languages.

We therefore introduce students to UML class diagrams and make use of them in lectures and laboratory work. Students learn to think and reason about their programs at a more abstract level than the programming language code they write. This reduction in complexity facilitates greater adherence to design patterns, improved code reusability, and less coupling between the components.

Furthermore, UML class diagrams are industry standard. By teaching students to express their designs in this language they acquire not only the abstraction skill but also knowledge of a language they will likely encounter in their future careers. That said, UML is a very large and complex language, and it is important not to overwhelm beginning students. To this end we employ only class diagrams in our CS1 course, and even then we restrict the notation to suit our educational goals: we use only five class relationships, with very constrained semantics. We return to this in subsection 2.3, below.

2.2 Why use a round-tripping tool?

We believe there are many benefits to using a round-tripping tool, both to students and instructors.

2.2.1 Student issues

As noted, students resist drawing diagrams without perceiving a direct and immediate benefit to their time and effort. We believe that students are both short-sighted and pragmatic in their behavior. They are pragmatic because creating diagrams which do not stay up-to-date with their code does not provide enough benefits to make the extra time invested in creating them worthwhile. At the same time they are shortsighted because their focus is only their current assignment. They do not heed the pronouncements of faculty that as their projects become larger the need for good design and design tools becomes increasingly important. If students do not learn good design and a means to communicate design early, it will be more difficult for them to learn this later on. Instead they become conditioned to simply code without forethought.

Use of a tool to draw class diagrams, generate code from diagrams and reverse engineer class diagrams from code, provides a value-added incentive to students to use such diagrams in their software development. Instead of providing a static snapshot of code at a particular point in time, a round-tripping tool makes the diagram come alive for students. Rather than being a static artefact which is disconnected from the coding process, the class diagram becomes an alternate view on their code, which students can consult at any time simply by switching their view (as easily as viewing a source code file in an editor window).

2.2.2 Instructor issues

When teaching design, especially at the CS1-CS2 level, we feel it is critical that faculty do more than give lip service to design. In other words, faculty must practice what they preach: (1) their own code must be well-designed, even when developed live in lecture, and (2) design must be checked and accounted for in grading schemes. We feel that use of a tool such as *Green* addresses both these issues.

Having a round-tripping tool available during lecture enables a teacher to effectively show students an iterative design and coding process, and obviates the need to prepare static class diagrams (which may easily become out-of-synch with the code) prior to lecture. Class diagrams can adapt to the flow of the lecture.

The second point is especially important: if students realize that the design work they do does not count directly toward their grade, they simply will not do it, or it will become an added burden without meaning, done as an afterthought rather than an integral part of the software development process. Of course grading for design is difficult. A round-tripping tool lets teaching assistants (TAs) more easily determine the structure of student code. Without a reverse-engineering tool at their disposal, TAs are left with two options:

- Class diagrams can be made a required part of all submissions. TAs must then rely on students' static class diagrams when evaluating the design of their code, without knowing for sure whether the code truly matches the diagram.
- TAs can attempt to reconstruct student designs manually from their submitted source code. This is a time-consuming, tedious and error-prone process, one that in theory will get the job done, but which in practice will likely not occur. Moreover, with large enrollment classes it is infeasible due to time constraints.

With a tool such as *Green* TAs can inspect and explore the design of students' actual submitted code via the tool. Rather than view a static class diagram which may not accurately reflect the current state of a student's code, they can quickly and easily view an accurate class diagram. This makes it significantly more likely that a student's design grade will actually reflect the quality of their design.

2.3 Why use a customizable tool?

We discovered the need to have a customizable tool during *Green*'s development. We designed *Green* to support very simple semantics for relationships such as composition, association and dependency: the tool supported exactly the semantics we spelled out for these relationships in lecture.

For example, the only "dependency" relationship we discuss early in CS1 involves assignment of a new instance of a class to a local variable, as shown in the constructor below:

```
public class Driver {
    public Driver() {
        ...
        GUI gui = new GUI();
        ...
}    }
```

We teach our students that there is a dependency relationship between the class `Driver` and the class `GUI`. This goes both ways (this code implies the relationship, and the relationship implies this code).

While this definition of the dependency relationship works well at the start of CS1, it does not work well in CS2, and may not be appropriate at all for other instructors and their students. The tool's architecture allows any relationship to be redefined; not even the set of supported relationships is predetermined. More details of this are given in section 3.3.

Other tools that we have found and evaluated do not allow the semantics of relationships to be customized. It is pedagogically important for the tool to be able to adapt to the ever-changing needs of students as they mature in their command of the subject. This idea is not new. The same basic principle is at work in development environments which support language levels [8, 9, 11]. The students' environment should behave exactly as they expect it to behave, based on their current knowledge. As students grow more sophisticated, so should the tool.

While *Green* does not support anything as sophisticated as the "language levels" of the DrScheme, DrJava and ProfessorJ environments, this feature of allows an instructor to tailor the semantics of the tool to suit their course and students.

3. TOOL COMPARISON

This section first outlines desirable features of a UML class diagram tool, and then presents the results of a preliminary comparison of a selection of such tools.

3.1 Desirable features

With regard to the pedagogical use of class diagramming tools, there are two major concerns. One is the level of student comfort with the tool; the other is the ability of the tool to simplify the instructor's task of conveying the main concepts of object-oriented programming to the students. Many desired properties follow from these two goals.

Of paramount importance for a tool aimed at CS1-CS2 students is ease-of-use. Beginning students are generally not able to use industrial-strength tools effectively, and may become bewildered and frustrated trying to do so.

The following is a list of features that we considered important in our evaluation:

Restricted Drawing: Does tool enforce semantic constraints during drawing? Different tools impose different restrictions on what can be drawn in a diagram. Some tools allow unrestricted drawing, even if the result is nonsensical (e.g. having an interface extend a class). Others prevent users from constructing ill-formed diagrams.

Code Generation: Does tool support code generation from a diagram?

Reverse Engineering: Does tool support reverse engineering of a diagram from code?

Extensibility: Does tool allow extension or customizability of its functionalities?

Run-time interaction: Does tool provide run-time interactivity? Some tools let users manipulate object properties at run-time, giving beginners a hands-on experience of how objects behave.

Refactoring Support: Does the tool support refactoring through the diagram? Refactoring of code is an important feature an environment can provide to aid in the incremental development of a good design.

Set of Relationships: Does the tool support a reasonably wide range of binary UML class relationships?

3.2 A comparison of selected tools

There are many tools available which support the drawing of UML class diagrams. While we have cataloged the functionality of a selection of these, our list is by no means exhaustive. The selected tools are intended to give a sense of the range of tools available. We have classified each as being intended primarily for an educational setting (*educational* tools) or as being intended primarily for developers (*developer* tools). We discuss the main differences between the educational and developer tools, and present a summary of differences between individual tools in figure 1.

3.2.1 Developer tools

We considered four tools which we classify as being developer tools: *ArgoUML* (`argouml.tigris.org`), *Omondo* (`www.omondo.com`), *Umbrello* (`uml.sourceforge.net`), and *Visual Paradigm* (`www.visual-paradigm.com/vpuml.php`). As a group these tools support a wide range of UML diagrams, have sophisticated code generation and reverse engineering facilities (sometimes for multiple languages), and more generally incorporate features to support large-scale projects. Developer tools are generally not free, except for functionality-limited or time-limited versions.

As far as supporting introductory students any one of these tools is more than capable of providing the necessary functionality. In fact, the problem lies with the number of features implemented; developer tools have so many capabilities that navigating them would be quite a task for most students. Rather than supporting the educational goals of a course these tools are likely to be a distraction.

3.2.2 Educational tools

We also considered four tools which we classify as being educational tools: *BlueJ* (`www.bluej.org`), *Green* (`green.sourceforge.net`), *Violet* (`www.horstmann.com/violet`), and *Vortex* (`www.studentcentredlearning.com/vortex`). As a group these tools have been developed with students in mind. Their focus is more narrow than that of the developer tools. Their user interfaces are relatively simple in order to make them easy to learn and to use. They are designed to aid students in their learning of important concepts, rather than to be an everyday workhorse of their jobs. The educational tools are generally (but not always) free.

A very popular introductory-level environment for learning OOP, BlueJ is light-weight educational tool with a fairly intuitive interface that employs drag-and-drop diagramming. Code can be compiled and methods can be tested during the design process. It also has the capacity for expansion by way of plug-ins.

Perhaps BlueJ's most interesting feature is its object interaction facility. This interactive environment gives a visual representation of each object instantiated, and allows the use to manipulating and interact with object at runtime. While it is a very useful environment for a CS1 course, its capabilities are limited. For example, only inheritance and dependency are supported in diagram drawing, and it has limited code generation and reverse engineering capabilities. BlueJ was specifically designed for beginning students and is an excellent environment for this purpose, but does not provide more general UML class diagram functionality.

Green, our tool, is a simple UML class diagramming tool intended primarily for beginning students. We have

	Developer				Educational			
Diagram types	Argo	Omondo	Visual Paradigm	Umbrello	VorteX	BlueJ	Violet	Green
Class diagrams	•	•	•	•	•	•	•	•
Sequence diagrams	•	•	•	•	-	○	•	-
State diagrams	•	•	•	•	-	-	•	-
Desirable features	Argo	Omondo	Visual Paradigm	Umbrello	VorteX	BlueJ	Violet	Green
Restricted Drawing	○	○	○	○	○		-	○
Code Generation	•	•	•	•	•	○	-	•
Reverse Engineering	•	•	•	○	•		-	•
Extensibility	•		•	-	-	○	-	•
Run-time interaction	-			-		•	-	-
Refactoring Support	-	•		•		-	-	•
Set of Relationships	•	•	•	•	•	○	•	•

•Supported ○Somewhat supported -Not supported (blank) Not tested

Figure 1: Feature comparison

addressed their needs by building a tool with a single focus: UML class diagrams. Green supports whatever relationships are installed, provides live linking between the class diagram and the underlying code via the JDT, and allows incremental exploration of code. More details are given in the next section.

Violet is a simple diagramming tool with a very intuitive interface that is easy to master in a short amount of time. The environment is very unrestricted: diagrams can be drawn as the user wishes, whether the semantics of the diagram are valid or not. Violet also supports many different diagram types (see figure 1). A shortcoming of Violet is that it provides no connection between the diagrams and implementation in code. For example, the methods and fields of a class diagram's boxes can be customized to display any text, whether meaningful or not. Violet is very good at what it does, but is limited as a general-purpose class diagram tool.

VorteX is a tool which is geared specifically at the educational market, and provides both UML class diagramming capabilities as well as a collaboration environment. It is a commercial product, and does not appear to offer any free version. The VorteX environment, while very elaborate, appears inappropriate for use on solo projects. The additional features are likely to overwhelm an introductory level student. For them, simplicity and an intuitive interface will pay off better than an elaborate environment which contains features that they will never use.

3.3 Green

A difficulty in teaching object-oriented programming is conveying the semantics of the relationships between classes and their implementations in code. In order for such a tool to be of definite instructional value it must assist rather than hinder the educational process. In Green, the class diagram is linked directly to the code, providing full round-tripping. Users can *add* elements (types, relationships and notes) to a diagram (in which case they are simultaneously generated in code), *delete* elements from the diagram (in which case any code supporting the element is removed), *hide* types from the diagram (in which case they are not deleted but simply not shown in the diagram), or they can *display* existing types (classes and interfaces) in the diagram. Green displays only those relationships that exist between types which are currently displayed in the diagram (hence the utility of displaying or hiding types).

Green also provides an *incremental exploration* feature, which allows a user to select any type currently displayed in the diagram and have Green find and display all types in relationships which originate with the selected type. Incremental exploration is a useful tool to explore the design of an unknown piece of code.

Users can also quickly access the piece of code corresponding to a relationship, a class, or something as minor as a method. For example, double-clicking on a method name opens the relevant definition in the Java editor. Green uses the Eclipse icons for displaying things such as visibility and access modifiers.

Functionality available through JDT is available through the class diagram view, as appropriate. *Green* makes it easy to change the source code you are working with, whether you want to add to the code you have, modify existing code, or observe the interactions between classes. Refactoring provides quick movement of methods between superclasses and subclasses, as well as renaming of existing elements, including updating their references.

The basic architecture of *Green* is shown in figure 2. Since the tool is constructed as an Eclipse plug-in it derives much of its functionality from existing functionality available within Eclipse. The user interface of the tool has been developed using the Graphical Editing Framework (GEF), while the underlying model of the Java code of a project is maintained by the Java Development Tool (JDT) Java model.

The basic tool in fact consists of several plug-ins. The core tool provides a diagram editing framework with extension points for relationships and help. Currently five relationship plug-ins are provided: (1) generalization, (2) realization, (3) composition, (4) association, and (5) (instantiation) dependency. The semantics provided are the semantics used in our CS1 course. Other semantics can be provided at any time. The semantics define how to generate code for a relationship once drawn in the diagram, how to remove code for a relationship once deleted from the diagram, and how to recognize a relationship in the code. The help plug-in provides the help documents for the tool. These plug-ins are packaged as a feature and can be easily installed through the Eclipse software update facility.

We have been using Green in our CS1 course this (fall 2005) semester, and in our CS2 course in the spring 2005 semester. Student reaction has generally been positive. Much improvement in the tool occurred over the summer, to address concerns raised by the CS2 students. Instructor and undergraduate teaching assistant reaction has been very positive: the tool has proved useful and easy to use in classroom demonstrations, and also

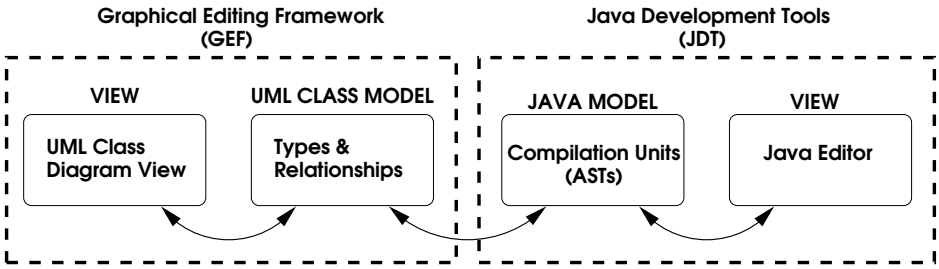

Figure 2: Architecture

speeds the development of classroom examples and laboratory exercises.

4. CONCLUSION

The *Green* UML class diagramming tool is a simple yet very functional class diagramming tool intended for use by beginning students. Its usefulness extends beyond the students in such classes to the instructors and graders of the course. Instructors benefit by being able to easily demonstrate iterative design and coding during lecture. Graders can easily recover the design associated with a student's submission by using the reverse engineering capability. It thus becomes feasible to grade student projects on their actual design, something that is difficult without a reverse engineering tool. Use of a tool such as this can also facilitate the iterative development process of methodologies such as extreme Programming by providing an easy way to move between code and class design views of a project.

5. FUTURE WORK

There are several directions in which to extend our current project. Our current review of UML class diagramming tools is limited, both in scope and depth. We would like to broaden our review of UML class diagramming tools in order to provide a useful resource.

We are exploring developing a simplified system for specifying relationship semantics. The current architecture requires that relationship semantics be specified as a visitor on the JDT abstract syntax tree. We are investigating developing a simple relationship specification language from which the tool can generate an appropriate AST visitor automatically.

In the longer term we intend to add design pattern support. Such support should enable students to quickly and easily manipulate their designs at a pattern level by doing things such as creating a skeletal pattern (e.g. a State pattern), adding new states to the pattern, creating a Composite state.

6. ACKNOWLEDGMENTS

We gratefully acknowledge financial support received from the UB Educational Technology Grant and IBM Eclipse Innovation Grant programs, and the contributions of numerous students to this project since 2001.

7. REFERENCES

[1] Computing curricula 2001. *Journal of Educational Resources in Computing*, 1(3es):1, 2001.

[2] *Unified Modeling Language Specification*. Object Management Group, 2003.

[3] C. Alphonce and P. Ventura. Object-orientation in CS1-CS2 by design. *ACM SIGCSE Bulletin*, 34(3):70–74, 2002.

[4] C. Alphonce and P. Ventura. Using graphics to support the teaching of fundamental object-oriented principles in CS1. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 156–161, New York, NY, USA, 2003. ACM Press.

[5] J. Bennedsen and M. E. Caspersen. Revealing the programming process. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 186–190. ACM Press, 2005.

[6] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 148–155. ACM Press, 2003.

[7] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30. ACM Press, 2004.

[8] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The DrScheme project: an overview. *SIGPLAN Not.*, 33(6):17–23, 1998.

[9] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177. ACM Press, 2003.

[10] S. Grissom and H. Dulimarta. An approach to teaching object oriented design in CS2. *J. Comput. Small Coll.*, 20(1):106–113, 2004.

[11] J. I. Hsia, E. Simpson, D. Smith, and R. Cartwright. Taming Java for the classroom. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 327–331. ACM Press, 2005.

[12] N. Nagappan, L. Williams, M. Ferzli, E. Wiebe, K. Yang, C. Miller, and S. Balik. Improving the CS1 experience with pair programming. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 359–362. ACM Press, 2003.

[13] L. Williams and R. L. Upchurch. In support of student pair-programming. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 327–331. ACM Press, 2001.