Lost in Object Space

Thomas Cleenewerck
Programming Technology Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussel, Belgium
thomas.cleenewerck@vub.ac.be
http://prog.vub.ac.be/

May 13, 2003

Abstract

The functionality in object-oriented programs is fragmented in its objects. Composing the whole functionality of such a program out of the functionality of the objects is not a trivial task. Furthermore, the role that objects play during the execution is for many students unclear. Consequently many novice students suffer from the "*lost in object space*" phenomenon. In this paper we present an environment that assists students to gain insight in the fragmented functionality, to help them understand the dynamic behavior of a program as well as the roles the objects are playing by visualizing the program while executing.

1 Introduction

Writing programs in an object-oriented programming language is not a trivial thing to do. But teaching someone how to program is an even more complex task. The main reason that causes this is that we are not capable of stating and defining the act of programming. Therefore it is hard to explain to someone else what programming is all about. What we do instead, is describing it in terms of the goal of programming, the various qualities of a program and the programming concepts e.g. looping and non-looping conditionals, classes, inheritance, etc. When teaching object orientation to starters the latter is in most cases the only subject one teaches about. Most students are perfectly capable to understand and reason about each of these concepts in solitude. The trouble begins when students have to put these together. In this paper we will focus more precisely onto the specific object orientation concepts like classes and objects. During observations of students using the think-aloud protocol [BL90] we discovered that students get lost in the amount of objects that are created and in the messages that are send. We call this phenomenon "*lost in object space*" which is similar to the lost in hyper-space phenomenon [DE93]. It is rooted in the poor insight in the dynamic behavior of the code. Previous attempts to tackle this problem using UML diagrams [Gro03], describing the dynamics in a *static* way e.g. sequence diagrams and collaboration diagrams weren't quite successful. Despite the complexity of all the diagrams, students still complained about the difficulty to imagine and trace the execution of a program. Furthermore students were not able to experiment and play with the software to learn on their own because the documentation e.g. collaboration diagrams didn't reflect the changes made to the software. In this paper we propose a new environment assisting the students to learn *independently* the dynamic behavior of *any Java*-program. The environment takes a program as input and visualizes the objects in the program while it is executing. On important configurable events like a method-call, reading and altering the value of local datamembers, etc the execution can be halted or paused. Giving the student the necessary time to consider the event and inspect the new state of the program.

2 Understanding object-oriented programs

To determine the problems the students are facing when trying to comprehend object-oriented software, students of various courses have been observed during the guided development of (little) programs. In this setting the think-aloud protocol gave us the most reliable and useful results. It allows us to follow the thoughts of the participants: What information are they searching in order to understand the program? How do they proceed to find and interpret that information? When do they abandon a trail of taught ? When do they give up? What did they missed?

Object-oriented programs consist of various aspects like problem knowledge, syntax, semantics of the programming concepts, algorithmic structure, design decisions, static structure and collaborations consisting of the construction of objects and message sends In order to fully understand a program, students need to be informed of all the issues listed above. OO-programs introduce the notion of am object or a class as an abstract entity which responds to certain messages. The functionality of OO-programs is *scattered* over many tiny objects, each capable of performing little subtasks on their own. Moreover, the objects or classes designed aren't always that obvious let alone the abstraction that led to the design of the object or class. Some of them originate from the problem domain, some of the choice of an algorithm, some of the various qualities a program must adhere to, etc. Composing the whole functionality of such a program out of the functionality of the objects is thus not a trivial task.

Documentation is the most commonly used mechanism to describe a program. We observed that students quite well understand a class described in class diagram and even the interaction between several objects in a collaboration diagram. There are two problems with documentation. Firstly for most of the students, the diagrams and other documentation doesn't suffice to reach a full understanding of the *role* that each object or class fulfills in the program. Secondly the students can not experiment with the program. Since the models are separated from the code, changes in the code are not always reflected in the models.

To help them comprehend the role of each object or class we often reverted to manual execution i.e. a human who executes the program one step at a time. But very soon, students were getting lost in all the objects that have been created so far, the current object which is executing, the previous objects and their methods which are currently on the execution stack, etc. We refer to this state of mind with the verb phrase "*lost in object space*". It is quite similar to the lost in hyper-space problem. A situation in which the user is disoriented, where he or she does not know where exactly he or she stands in the hypertext structure, how far it is until the end, how much of the whole is already read, etc.

Let us map hyper-spaces to OO-programs. A hyper-space consists of pages which are interconnected by links, whereas an OO-program consists of respectively objects and message sends. The solutions for the lost in hyper-space problem proposed in the research area of hypermedia is to provide at all times the necessary context information by clearly naming the pages, by using label and design elements as context cues, by creating a breadcrumb trial [Ber99], undo actions, etc. When mapping the solutions back to OO-programs we observed that the current page corresponds to the currently executing object and the breadcrumb trail to the execution stack.

3 Visualization Environment

To tackle the lost in object space phenomenon we developed a visualization environment that visualizes the execution of OO-programs written in Java.

3.1 What

Object-oriented programs are perceived as big state machines where the state is defined as the state of the objects currently alive in the heap, its execution stack and instruction counter. The visualization environment visualizes each state of a program and must clearly indicate which event triggered the state change and what has been changed against the previous state.

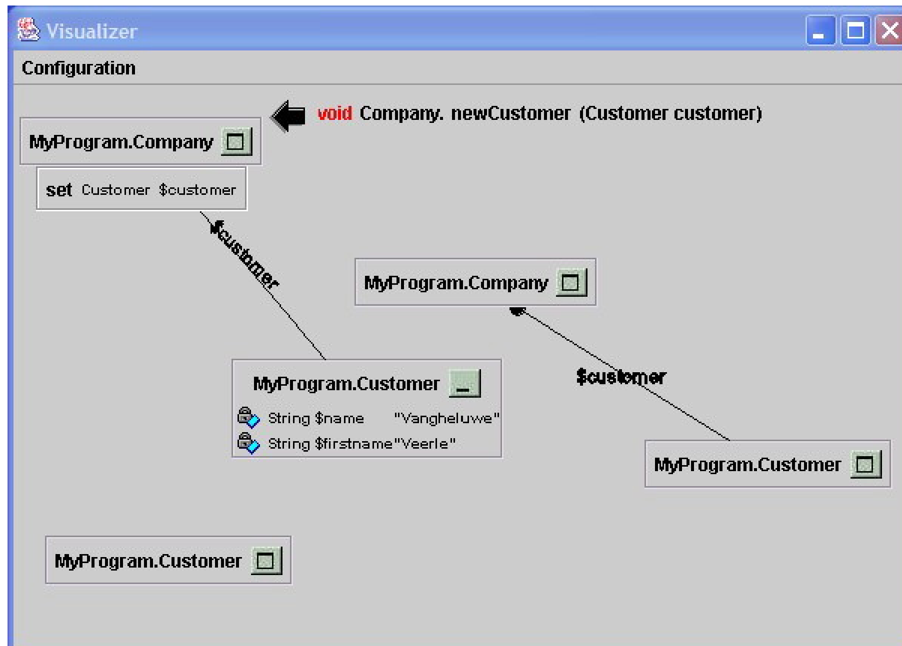

Figure 1: Overview of the visualisation of a small OO-program.

In this setting, not all the states of a program can to be visualized since it would clutter up the visualization. Most of these states do not contribute to the understanding of the program in terms of object orientedness e.g. states where an address of a pointer is retrieved out of the memory or even states where an if-statement jumped to its failed code block. Therefore the visualization environment only shows interesting states and state transitions based on the lifecycle of an object: object creation, datamember changes, method calls, unreachable object and object destruction. Many OO-languages, as is the case with Java, are garbage collected. In other words, objects are collected and destroyed whenever the garbage collection algorithm sees fit. However the point in time where the state of the object lifecyle moves from alive to unreachable is important information to understand where and when in the code some of the objects become obsolete or are being discarded.

In each state that is visualized, the currently executing object and its active method is shown. After each method-call state transition, the currently active object and method is possibly changed. This change is reflected in the environment. Early tests showed that it is very hard to keep track of what the program is doing since the context of each method-call is lost. Therefore we divided the method-call transition into a method-entry transition and a method-exit transition. Now the student can observe when a method execution has finished and a previous method-entry has been resumed.

Research in the area of hypermedia showed that providing context information is crucial to aid people orient themselves in a set of hyperlinked documents. The same problem arises with the visualization of the states and state transitions of a program. Hypermedia introduced the notion of breadcrums to show the hierarchical path from the root document to the current document. Mark that it is not always the path that was taken by the user to reach his document. This mechanism will also be used in the visualization environment to provide the necessary context information in a slightly different way. We will not use the hierarchical static context information e.g. package, class, innerclass, method, etc. but the hierarchical dynamic context information of nested method calls i.e. the execution stack. The execution stack shows us all the method-entries from the start-method up to the currently executing method. In other words it contains the necessary dynamic context (history) information to determine why and by which other method, a

method was invoked.

3.2 How

The visualization of an OO-program is conceived as a graphical interactive presentation of the state changes described in the section 3.1. Interactive because the student or teacher can tweak the presentation to meet their needs and can at all times pause, resume and terminate the presentation. For most parts of the visualization we choose an entirely graphical and intuitive presentation to reduce the cognitive overhead to interpret what is begin visualized. The visualization is composed out of the visualization of objects and their relations, indicators and context providers.

In figure 1 the visualization of the object space is shown of an example program about companies and customers. There are two company objects each with one customer attached to it (relation with the label `customer`). One company object is executing the current method `newCustomer(...)`. During its execution the `$customer` attribute has been set (denoted by the box below this object). The value of this attribute is a customer object which has been expanded. The values for the attributes name and firstname are respectively `Vangheluwe` and `Veerle`.

3.2.1 Object and Object Relation Visualization

Figure 2: This figure shows the a newly created customer object using the standard visualization. The current executing method (denoted by the arrow) is showing the constructor method (init). The customer's first name has been initalized.

The core of the visualization is the one that visualizes the object space. Upon creation of an object in the program, a new object visualizer component is created and added to the object space. A standard object visualization is available which can visualize every object (figure 2). Additionally, custom visualization for certain objects can be added as a plugin to the environment. The standard visualization uses the UML notation to display the objects. Primitive datamembers are visualized using their string representation (figure 2), non-primitive datamembers by arrows associating the objects with each other (figure 1).

The visualized object can be minimized (figure 1), hiding the datamembers. There are several reasons for this. Firstly the visualized objects occupy less space on the screen. The size of a graphical screen is nowadays still limited. Secondly the type and value of datamembers are not always of interest. Test runs proved that it is important to be able to carefully select the information that is provided/presented to avoid overwhelming the student which would reduce the quality and effectiveness of the presentation and the students attention and focus. To assist the students using the minimized visualization, an object is automatically expanded/restored to full size when an the value of one of its datamembers changes.

3.2.2 Indicators

When the state of the program changes, the student should not ransack the new state in search for the differences. The primary purpose of indicators is to avoid this situation and thus to guide the students through the presentation. They are responsible for handling the transitions: datamember-change, method-entry, method-exit and unreachable.

For each transition the teacher or student can configure whether the execution

- should continue without any interruption or basically ignore the event. Visualizations are often run/viewed a couple of times where each time the focus is on different aspects/parts of the visualization. Ignoring a transition is often used to reduce the amount of information offered and rendered by the visualization.
- should be delayed for a period of time allowing the student to prioritize the information offered and rendered by the visualization.
- should be interrupted with an explicit notification of the event to clearly draw his or her attention. The notification pauses the execution of the program giving the student the necessary time to interpret the information and learn. In figure 3 a notification of a method entry is shown.
- should be a timed interrupt like the former but the execution resumes automatically after a period of time. This option is mainly for convenience reasons.

When a datamember is changed, both the old value and the new value are shown. When a method-call is initiated the caller method is placed above the callee method (the new method) with an arrow facing down (figure 3). On the termination of a method, the same is shown but with an arrow facing up.

Figure 3: This figure shows the a notification message that is shown when chosing for a interruption when a method is called.

The arrow is a symbolic notion that is easy to understand and is quickly interpreted by the students hereby reducing significantly the cognitive overhead compared to a textual description. In each state the currently executing object is shown. When methods are initiated and terminated, the change of the currently executing object is reflected in the environment. When a object becomes unreachable it is removed from the presentation.

3.2.3 Context providers

In section 2 we have shown that more information is needed besides objects and state transitions to be able to follow the presentation and situate the current state in the course of the execution of the program. Before the visualization starts, the student should be informed of the name of the program, a brief explanation of the abstractions behind the objects used in the program and how their originated (section 2). This information must be supplied, since it can not be extracted out of the (running code). The context providers in the visualization environment visualize the execution stack and the method history. The execution stack is visualized as a list of all the methods currently on the stack. There is however no graphical representation yet, showing the execution trail in the visualized object space. The method history component is presented in a similar way.

4 Future work

When visualizing large programs or only a certain part of the program is the object of study, not all the objects and their accompanying events are relevant. Currently custom visualization

can be plugged into the visualization environment. This mechanism can be used to collapse the visualization of a whole set of objects to a single visualizing component. But events associated with these inner objects are still visualized. Future work is needed to collapse the events as well.

Another problem with large program is the lay-outing algorithm to position the visualized objects on the screen. Currently a simple grid lay-outing strategy is used. But when lots of objects are created this doesn't suffice at all. Therefore another lay-outing mechanism is required, one that takes into account the time they were created, the static structural properties (package, class), the correspondence with problem domain concepts, to which other objects this object is referring to, etc.

In Java, as in many garbage collected languages, the point in time where the objects are collected is not the point in time where the objects become unreachable. When objects become unreachable they must be removed out of the visualization environment.

The execution stack is now a simple list of methods. A more graphical approach would be more desirable.

The method history component is still in a testing phase. We are experimenting with some filters which can extract some more useful information and equip the list with some special browsing facilities to increase the component's functionality.

5 Conclusion

The environment is able to visualize the creation of objects, method calls, and datamember changes of any java-program. The visualization of object-oriented programs is conceptually mapped to a set of hypermedia documents. This enabled us to use the solutions form the hyper-space research area by offering additional context information, which proved indispensable in the visualization in the form of context providers and information indicators which help to situate the state of a program in its execution trail and guide the student through the visualization. With this environment students are capable of learning on their one, the dynamic behavior and the roles objects play during the execution of an OO-program. Furthermore experimentation is encouraged, since they are able to change their program and visualize the new program.

This work is still in progress. Students are very positive about the environment although the environment is still under construction. However a more thorough and detailed evaluation and validation is required before we jump to conclusions.

References

[Ber99] Mark Bernstein. Structural patterns and hypertext rhetoric. *ACM Computing Surveys (CSUR)*, 31(4es):19, 1999.

[BL90] Kathleen Gomoll B. Laurel. *The Art of Human-Computer Interface Design*, chapter Some Techniques for Observing Users, pages 85–90. Addison-Wesley, 1990.

[DE93] L. Hardman D.M. Edwards. *Hypertext: Theory into Practice*, chapter Lost in hyperspace: Cognitive mapping and navigation in a hypertext environment, pages 90–105. Intellect. Oxford, 1993.

[Gro03] Object Management Group. Unified modelling language, 2003.