Visualization of Designs in an Object-Oriented Software Engineering Environment
Alvarez, X.; Dombiak, G; Prieto, M.; and Rossi, G. (*)
LIFIA - Departamento de Informática
Facultad de Ciencias Exactas - Universidad Nacional de La Plata. Argentina
(*) also CONICET and PUC-Rio (Brazil)
e-mail: [ xavi, gaston, maximo, grossi ] @ info.unlp.edu.ar

ABSTRACT
Complex or unknown designs, when implemented are hard to comprehend and use, not to mention re-use. Existing documentation is normally insufficient because it fails to represent the inherent dynamism of the objects involved.

We developed a set of tools that can be easily included in a Smalltalk environment to allow the visualization and animation of any set of objects and their inter-relations. It also provides the ability to analyze, by means of graphical representations, a given model at multiple levels of abstraction. As a consequence of this, the developer can use the same tools to verify or investigate the implementation of a given design. All the visualization and animation is done by means of non-intrusive techniques that allows on-line modifications to be reflected in the visualization.
1. INTRODUCTION

Traditional techniques of animation focus on static code structures, and are inappropriate for Object-Oriented designs and the required abstraction level that allows a coherent comprehension. Current animation and visualization Object-Oriented techniques consist of a model that plays the role of the subject and an application that shows the communication patterns between aspects of the model. This approach fails to adapt, on the fly, to the constantly changing visualization requirements. The definition process by which the animation is built, is manual, and heavily relies on the knowledge of the model by the person making it. This drastically limits the usefulness of such tools just to the post-mortem analysis and documentation of the models involved.

What we developed is a set of tools that enhance the environment by allowing the visualization of any object model. The configuration is done in a graphical manner in order to obtain the desired level of abstraction. All this is non-intrusive to the application and without any undesired side effects on the performance of the non-visualized aspects, with the benefit that if the user isn't animating, there's no performance cost or impact in the environment.

2. AN OVERVIEW OF THE VISUAL ARENA ENVIRONMENT

The Visual Arena environment is a set of tools that collaborate among them to allow the user, or composer, of the animation to present a set of objects interacting in a graphical interface. The environment is enhanced with special tools that facilitate the handling of these objects. These tools have many potential uses that aren't necessary related to the visualization, thus the environment as a whole is enriched.

This non-intrusive policy gives us the leverage to claim that if you don't use it, then you don't regret it. This not only goes to Smalltalk as a whole but also to the instance level. Let's say you're visualizing the interaction of some heavy used object like Point or Rectangle. On some animation schemas once you declared an area of interest in a given class then all the instances of that class are affected by the animator. This is probably because some methods have rewritten at a class level. In our schema only the instance of the class that you want to visualize had its behavior modified somehow.

2.1. Visualization and Animation

When this project started, the main problem was to obtain a good definition of what we meant by animation and visualization of objects. In our first discussions, each one of us had a mental picture of it, but of course, nobody was totally satisfied with the other persons' definition. As time passed, the idea matured in our heads. With the first prototypes we slowly begun to discover what we meant by these two words. Also, as we
explored the things that we wanted to visualize, we discovered many other things and concepts that could be, or should be, capable of being visualized.

Our first target for visualization and animation was the basic semantic of the Object-Oriented paradigm: object - message. This was relatively simple to detect but we placed some restrictions on how we would detect it that made it harder. So, the first thing we considered, was having two objects and wanted to detect a message-send between them. Once we solved how to intercept the message send, many other things started to pop-up in our minds. For example, lets say that one object is referenced by the instance variable of another. What would happen when a third object is assigned to that instance variable? All of a sudden my visualization could be off-synch with the objects that I'm trying to visualize. Should I visualize just objects or use other abstractions that allowed me to visualize the composition, or whatever, of my model.

The concepts we were handling when talking about visualization and animation were constantly growing and changing from our primitive mental conceptions. With this in mind, we are currently working with the following definitions:

**Animation** - Is the process by which an object is displayed on a graphics medium and its activity is represented.

**Visualization** - Is the process by which one manipulates a given animation to introduce abstract concepts such as sub-systems and object composition in order to better understand the underlying model.

2.1.1. Traditional Approaches

On some 'traditional' approaches, the process by which the animation is done relies on a pre-existing general animation framework or the ad-hoc creation of one. The first approach is usually intrusive and the parametrization is sometimes too rigid for dynamic modifications. The second alternative, is normally too expensive and the application is mangled with the related visualization code. In either case, the process is manual and it implies the modification of the application code to indicate the points of interest to the current animation. The exact placing of these 'visualization-breakpoints' greatly depends on the knowledge that the user has of the application. Therefore, the animation turns out to be a simple documentation or demo of the inner works.

Most of the explored techniques deal with a static parametrization -- through scripts, breakpoints, etc. This means that the application needs to be prepared especially for the animation. This has a big drawback in that you can't change the focus of the animation as you start to understand it and maybe decide to see the inner works of a subsystem, or if the animation started on a small subset, then go up in the level of abstraction and collapsing the current focus. Also, with static visualizations
you can’t see how a modification in a method starts other paths of communication, thus enabling you to dynamically see the different design approaches.

2.1.2. Temptative Designs

Stack Monitoring

In our exploration of the possibilities to detect the activity within a set of objects, we came across with the possibility of having a high priority process that would be constantly monitoring the process stacks to detect objects currently in focus and the context in which they were being used. Although this a non-intrusive approach, we decided that it would degrade the performance of the whole environment. Considerations were taken, mainly regarding the volume of objects involved in a particular animation. With few objects, the decision as to the relevance of the current object in the stack would take little time thus enabling the use of this approach. Unfortunately, as the volume of animated objects grows, the monitoring would be too time consuming.

Intrusive message sends

As said before, some of the techniques we found that allowed the visualization of behavior were code intrusive. At first we first evaluated what did they provide and how to avoid them, but still providing the same functionality. We finally developed a technique that allowed us the same functionality but without being code-intrusive and transparent in the application level.

We basically discarded this approach as it was hard to control and allows the user modify the application without getting tangled in the visualization added code.

Code Simulation / Interpretation

This is an approach based on the all praiseworthy debugger. Unfortunately, this approach is too costly in the performance aspect, but the most flexible in terms of capturing the desired events (message sends, variable assignments) even before they’ll happen. Although we explored it, with varying results, we decided to abandon this track due to the performance costs. Its major benefit was a better understanding of the environment itself.

Non-Intrusive / Automatic triggers

Given that this approach is the chosen one to develop the extensions mentioned in this paper, we’ll basically say that being non-intrusive is the capability of adding behavior to objects without modifying the application code. Although there are other definitions possible, we define the automatic triggers, as actions that will be taken when a given condition is reached or satisfied by a given object.

2.2. Available Tools

In order to have the necessary instances for doing a visualization, we were in desperate need of better mechanisms than the inspector or debugger. So, we developed tools that allowed us to handle the instances of an application. These tools let the user recover, store and manipulate them. A side effect of the
initial use of these tools is that most of them can be used independently of the visualization, thus enhancing the environment.

2.2.1. Full Inspector

Given the hierarchy of included inspectors in Smalltalk, their main drawback is the inability of seeing the whole internal state of an object at a glance. They are designed for showing only one aspect or variable of the inspected object. The FullInspector allows you to see the whole internal representation and state without having to select each one.

2.2.2. Instance Hunter

This interface allows the user to retrieve all the instances of a given class. Although this could be easily done with aClass allInstances, this is sometimes tedious. The InstanceHunter lets you select a class just like as if you were browsing the Smalltalk environment. Once the target class is selected, you may define a select or reject block that filters the given class instances.

After the instances were retrieved, they are presented in a list from where they can be manipulated. The InstanceHunter has a FullInspector incorporated, so any hunted-instance can be fully inspected.

2.2.3. Object Pools

This is a simple tool that allows the user to store objects for later retrieval. All the instances that are placed in the pool are identified with either a serial number or a user defined name. We developed two types, local and global. Local pools were designed to be inside other tools, and to provide the ability to reference the pooled object as a parameter to be used in sending them messages in a given context, much in the same way that you can reference parameters and temporal variables inside the debugger.

The main difference with the global pool, which is a singleton, is that the instances inside it, can be referenced in almost any context. The use of it should be careful and watchful, so that no references to them are kept inside compiled methods. We decided to include this global pool, in order to facilitate the sharing of objects that were of interest in many places to add a global access to a heavily used instance. We disclaim all responsibility of the possible mishaps due to the use of this tool.

2.2.4. Instance Workbench

The purpose of this tool is to provide most of the facilities that a user needs in order to interact with a given instance. This tool includes, most of the previously mentioned tools. It provides the facility to select a given selector that is implemented in the instance class, send it and capturing the result. Due to the well-known impracticality of creating the parameters necessary for the message send, the InstanceWorkbench has an incorporated localObjectPool. The included objects in this pool may be used as the parameters for the selected selector. A FullInspector on the instance currently in the workbench is also present. It also includes a log of all the messages sent to the instance, from this log. You can retrieve the parameters and results for
a more detailed analysis. Finally, there is a workspace that is linked to the local and global pools from which the user can prototype methods and finally include them in the class, as if it were a class browser.

This tool provides one of the most flexible mechanisms through which the user can manually trigger any communication patterns of the objects currently being visualized.

2.2.5. Lightweight Browser

Through this tool, you can view, add or modify, the behavior added to a particular lightweighted instance (to be explained later). The way this tool works is very similar to a ClassBrowser with the exception that you work at an instance level instead of a class level.

The main objective of this tool was for internal use, but since the ability to add behavior to any particular instance is very powerful we decided to include it as an extension of the environment.

2.2.6. Drag and Drop

Although the purpose of this tool is self-explaining, the reason for it is that without it the manipulation of the instances wouldn't be intuitive. For example, the passing of an instance from the InstanceHunter to the InstanceWorkbench.

2.2.7. Visual Arena

This is the central tool of the visualization environment. It is a graphical interface that allows the user to drop and place objects to see how they communicate. When a new object is added to the arena, the user may specify what protocol and or instance variables should be visualized. This will take on effect immediately and when the instance receives one of the tagged selectors, a line will be displayed between the sender and the receiver. These lines are called pipes and provide the user with certain information about the exactitude of the current visualization. It also shows the currently active stack of message sends between the objects in the arena.

Pipes provide a visual clue about the patterns of communication through the activity level. Each time a message send between two objects is detected, the pipe increments its intensity level; represented by the red component of its color. As time passes by, the color is slowly degraded towards the background color. This provides a simple but effective way of reminding what communications occurred while showing how strong was the collaboration in comparison to the present state of the model.

2.2.8. Class Activity Detector

Although this tool is very similar to the VisualArena, the kinds of visualizations that can be viewed differ mainly in that it's based on a class level instead of the instance level. As the application (or model) is running, all the classes of the instances involved will be added dynamically to the visualization. There is no possibility for the filtering of protocols. This means that any communication between the instances of the classes being visualized will be registered.
A subtle variation to this kind of visualization is that the user can configure the initial classes to be monitored thus reducing the initial complexity because only the classes that have some direct interaction with the present classes will be added dynamically.

Both of these kinds of visualizations can be set by the user at a class level. In the first case, every class that is present in the communication will be added to the arena as a 'direct' class. In the second one, only the classes that communicate to or with a 'direct' class of the visualization will be added, but as an 'indirect' visualized class. At any point of the visualization, the attribute of a class (direct or indirect) can be changed.

This allows to visualize a model at a pretty high level of abstraction without having to hunt the instances of the classes without previously knowing which are the classes that are relevant to the model.

3. THE VISUAL ARENA FRAMEWORK

Frameworks aren't easy to come by, and good ones are even harder. Although the claim that this is a framework might be overstated, we feel that it has the basics for the extensions and flexibility to allow it to become a full framework on its own right.

One of the main objectives of doing this animation and visualization tool was to be capable of visualizing almost anything pertaining to the Object-Oriented paradigm. Since the implementation of any given design is code, which is machine processable, we started designing an architecture that would allow us to visualize this code. Unfortunately, there are many other things that could be visualized or animated. With this in mind, we had to develop an architecture with three main components: the model, the detection and the visualization layers. Each of these layers is basically independent of each other and with clear cut communication pattern. This gives us the flexibility to replace any of them. For example, if we replace the model layer with an use-case and the interception layer with a simulator-player, we would be able to visualize and animate a use-case. On the other hand, we could also enrich or change the visualization layer by adding hypermedia capabilities.

3.1. The Object Domain Layer

The object domain layer is the model that provides the behavior of the application. This domain could be represented by many techniques (source code, use-cases, etc.). When the models are developed, they don't need any special considerations towards the animation and visualization tools; quite the contrary, they should be independent.

As we said, we are currently working with Smalltalk code. This implied that the whole visualization and animation design was slightly biased towards the necessity of animating it.

3.2. The Interception / Detection and Conceptualization Layer

The main purpose of this layer is the ability to detect and intercept the Object-Oriented
concepts, mainly the message sending between objects. Once the event is detected, proper actions should be taken such as notifying the visualization layer to show it.

Regarding the conceptual functional aspect; it's basically the component that regulates at what level of abstraction is the model being visualized. One of the many uses of it is the ability to describe some interaction as being the interaction of two sub-systems instead of two objects.

In our case, we developed a set of wrappers capable of intercepting message sends to objects and the monitoring of instance variable assignments. We define a wrapper as an object capable of encapsulating an object from the domain model, while detecting the message sends and variable assignments under normal circumstances.

As the complexity and diversity of things that we wanted to visualize and animate grew, we found ourselves incapable of allowing the user to visualize a model at different levels of abstraction (class, subsystem, component hierarchy, etc.). This was the seed that forced us to develop a hierarchy of specialized wrappers that allowed the conceptualization of these events and concepts.

3.2.1. Instance Wrappers

These were the first kind of wrapper we developed, and allowed us to wrap an instance. They keep the configuration of what messages are of any interest to the visualization.

3.2.2. Global Wrappers

Bad designs are possible, and the use of global variables is in almost any case a pretty bad decision. We have to differentiate between a globally accessible object from the global accessible repository of that object. This means that sometimes we are interested in detecting the assignment of the given variable while other times we are only interested in the current object held by the variable.

Global wrappers are the design solution for being able to monitor the access to the object in the global variable (repository). We should note that a class is an instance, thus if the interest is in the behavior of the class, it will be wrapped in an InstanceWrapper. Even if you could consider the class as being a global variable, what is global is the repository of it, and not the object referenced in it.

3.2.3. Conceptual Wrappers

Because the abstraction of a part-hierarchy isn't directly supported by the Smalltalk code, we had to provide a way of reconstructing and showing it. The main objective of this type of wrapper is to hide the complexity and regulate the level of abstraction.

It allows the user to group, in an arbitrary way, a set of objects in a concept, and treat them as if they were one. This enables the user to visualize the application at different levels of abstraction by effectively reducing the diversity and volume of the visualized objects.

3.3. The Visualization Layer

This layer is responsible for producing a
representation of the activity and behavior of the model, in a user-comprehensible way. In order to do this, it has to receive the notifications from the interception layer and show them in the current context of the visualization.

In order to obtain a better understanding of the model, the user may need a set of specialized visualizations. This means that several visualizations of a model have to be created but each one focused in different aspects. Therefore, each visualization needs a context. The context of a visualization means what objects and what aspects of them are being visualized. For example an object may be visualized in many different places, while the aspects being shown in each one are different and the objects that share the visualization with it, are not always the same.

In our case, we adopted a simple 2-D graphic approach, being the visualized concepts represented by icons, and the communication patterns by lines connecting the objects.

4. DESIGN AND IMPLEMENTATION DETAILS

The whole idea started basically from the visualization layer, that is, the idea of being capable of seeing how a set of objects interacted and be able to manipulate them in many ways. In order to be able to do this, we faced many problems with the 'technicalities' of how to intercept and detect all the things that we wanted to visualize.

The basic idea that drives this approach is the ability to detect the message sends at an instance level and not at a class level. This is totally different to the detection of these message sends to all the instances of a class and filtering only those that are of any interest.

Even though this approach has many extensions and could be used in many other ways, the main use we gave it was the ability to add behavior to a particular instance so as to be able to hook the visualization triggers in a totally automatic and transparent way.

Many things were used, designed and tried in order to obtain the so-called visualization and animation. Next we present a description of the objects that allow and conform the inner-works of any visualization.

4.1. Lightweight Class

In order to detect a message that was received by an instance we had to find a way in which that sole instance would notify this event. This implies that the instance should have an extended behavior that differentiates it from any other object or instance of the same class. The main idea for solving these 'technicalities' was derived from Bob Hinkle’s tutorial 'Reflective Smalltalk', although we tried several others and extended the overall scope.

The mechanics under which new behavior, in this case the visualization specific code, is added to a particular instance is the creation on-the-fly of a new subclass of the object’s class under which the new behavior is added. The
next step is to change the class of the object to that of the newly created subclass.

4.2. Wrappers

The main role of these objects is the preparation of the model to be visualized and the notification of the events that were defined to be detected. An object model isn’t limited to only one wrapper. This allows that many different wrappers, with different aspects of interest be defined for a model.

In our case, the wrappers are the link between the object (pertaining to the model domain) and the visual arena under which it’s being shown. When a domain object is ‘wrapped’, the object will be lightweighted (as explained above). In this lightweighting process, the user of the visualization will specify what protocol should be intercepted. For each specified selector that will be intercepted, the method will be overridden in the lightweight subclass in order to do three things:

a) notify the objects dependents of this new invocation to the intercepted selector,

b) invoke the inherited selector, and finally,

c) notify its dependents of the finalization of the execution of the selector.

In order to work the wrappers make themselves dependents on the object they are wrapping. When a wrapper receives one of these visualization specific notifications, it verifies that the protocol being detected is defined in its collection of protocols to display -- this permits to have several wrappers wrapping the same object but each wrapper belonging to different arenas and thus effectively showing different aspects of the same object in different contexts.

Since many things and concepts could be visualized, a series of different wrappers were defined for each case.

4.3. Watcher Processes

Given that we can’t guarantee that the objects in the model use double-encapsulation, there will be instance variables that can be assigned in many methods without any clear point of where the assignment will occur. Even worse, there may be methods in which the assignment will be conditional. To solve this problem, we adopted a monitor process. When the user wants to monitor an instance variable, all the methods that have the potential capability of changing the value are specialized to launch a process. This process, that will run at a high priority, will monitor the variable as long as the execution of the tagged message is running.

4.4. Stack Analyzers

This component is the one that provides the analysis of the fidelity and/or exactitude of the current visualization, to be later presented by the arena to the user.

One of the main capabilities of the visual arena is to show the message sends between objects that are included in it. Because the user is responsible for the inclusion of the objects, the visualization might be incomplete regarding what objects are really collaborating. For
example, let's say that the arena contains two objects, A and C. Suppose that A sends a message to object B, which results in B sending a tagged message to C. This communication will be shown as a line/pipe connecting objects A and C. Because B might be or not relevant to the current visualization, the arena can provide visual clues through the pipes (mainly in the color) about the amount of the missing objects that collaborated between A and C, like B in this case.

4.5. Pipes

As we mentioned above, all the communications between any two objects included in an arena are registered. Pipes are the objects in charge of logging this. The basic information that a pipe has is what objects/wrappers are linked by it, and what messages were sent in each direction. This message sends are timestamped so as to give a better understanding of the dynamism of the communication.

4.5.1. Simple & Full Pipes

Simple Pipes are the most basic kind of pipe. It only knows what object is its destiny and what selectors have been sent to it.

Full Pipes are made of two simple pipes that model the communications going from one object to another, being logged in one simple pipe, or the other, depending on the direction of the message flow.

4.5.2. Conceptual Pipes

Very similar to the full pipes, but are designed to handle the communications to and from conceptual wrappers. This gives place to some strange combinations; for example, a full pipe conformed by a simple pipe in one direction and a conceptual pipe in the other, which maps the communications between a concrete wrapper with a conceptual wrapper.

5. Extensibility

Due to the prototypical characteristics involved in the development of the visualization, many things were left out. It is our intention to further evolve or enrich many aspects.

Apart from allowing a visualization and animation tool that would not interfere with the model and be non-intrusive, one of the main ideas was to provide an extended environment that would be capable of being extended and modified by the user -- as all Object-Oriented designs should be.

5.1. Graphical

The current richness or level of detail that an object shows in the arena is a simple icon chosen by the user. Although this was a basic thing that it should have, many ideas came up in order to enrich this rather primitive representation. For example, we could think that the graphical representation should be customized for each object, taking in consideration many user-defined aspects such as, but not limited to, the internal state.

Another idea is to present a fish-eye view of the currently active object; this will present the other objects in the arena 'weighted'
against the object currently in focus. This would undoubtedly give a better understanding of the interrelations of the objects, by focusing primarily on the active objects.

The next step would be the representation, with or without the above extensions, in a 3D space. Even more, it would be interesting and almost wonderful, to be able to immerse it in a virtual reality space.

5.2. Textual & Hybrids

Sometimes, a graphical representation fails to provide exact values. For example, in a pie chart, we perceive the proportions, but not the exact values. On the other hand, a tabular display shows the exact values, but fails to provide the overall picture. So, although the visualization is mostly graphical, some aspects of it might be perceived with better detail in a textual or hybrid (text and graphics) representation.

6. CONCLUSIONS

Due to the nature of the Object-Oriented paradigm that models the reality in a faithful way, which is complex and constantly changing, system requirements grow in complexity and dynamism. The need to handle this ever increasing complexity is a reality that any developer meets sooner or later. In order to manage this complexity in human-understandable proportions we are forced to create new tools.

One such tool, is what we presented. The ability to encapsulate low level designs in a high level view, together with the capability of manipulating the application objects in a much more intuitive way allows the developers to make meaningful abstractions. Abstractions could represent many things and are not restricted to any level of comprehension or abstraction.

These newly 'visualized' abstractions could be the seed for future frameworks or the discovery of patterns within a given application.

7. REFERENCES


3Brian Foote, Ralph E. Johnson. Reflective Facilities in Smalltalk-80

4Rob Hinkle, Tutorial: Reflective Programming in Smalltalk, OOPSLA 1993