Software Visualisation of Java Programs in InspectJ

Rilla Khaled, James Noble
Victoria University of Wellington, New Zealand
{rkhaled,kjx}@mcs.vuw.ac.nz

Robert Biddle
Carleton University, Canada
{robert_biddle}@carleton.ca

Abstract

Visualisation is a powerful method for explaining how programs work. However, while it is advantageous in theory, it is not used as frequently as it might be. Patterns may be used to describe architectures, so in this paper we present a pattern language in two parts. The first part describes a set of patterns for approaching the development of program-specific visualisations. The second focusses on actual implementation of the visualisations using the InspectJ framework, which uses AspectJ to monitor programs at run time. The patterns step through the process of building a visualisation from start to finish, featuring a running example of the development of a visualisation for a simulation program. After reading these patterns, you should be able to use InspectJ to create visualisations for your own Java programs.

Introduction

Visualisation is a powerful method for explaining how programs work. Nonetheless, while it is advantageous in theory, it is not used as frequently as it might be. In particular, software developers seem particularly wary of developing their *own* visualisations for software they come across. This could be partly related to developers not wanting to expend unnecessary effort. Almost certainly however, another major reason why visualisation is not used as much as it could be is because many programmers just do not *know* how they might approach the task of developing a visualisation.

This paper presents a pattern language in two parts, the first part focussing on how to approach the development of visualisations, and the second part providing a set of implementation patterns for InspectJ, which is a visualisation

framework for building visualisations. After reading these patterns, programmers should be able to use InspectJ to develop their own visualisations for Java programs, or at least have ideas about how to develop visualisations in general. The patterns, the problems they address and the solutions they provide are summarised in Table 1. Figure 1 shows the relationships between the patterns: the boxes outlined in bold represent general visualisation patterns, while the others represent InspectJ implementation patterns. The directional arrows denote the path of usage, where a straight line signifies a concrete relationship while a dotted line signifies a more implicit relationship. Loosely based on the structure of the Coplien form (Coplien 1996), these patterns feature **Motivation**, **Example**, **Problem**, **Forces**, **Solution**, **Example Revisited**, **Resolution of Forces** and **Related Patterns** sections. These patterns are aimed at programmers who are sufficiently knowledgeable about object-oriented programming concepts and ideally reasonably experienced Java programmers. InspectJ makes use of AspectJ (Kiczales, Hilsdale, Hugunin, Kersten, Palm & Griswold 2001), so it is assumed that programmers have the necessary AspectJ tools, which may be obtained from the AspectJ website (`http://www.eclipse.org/aspectj/`).

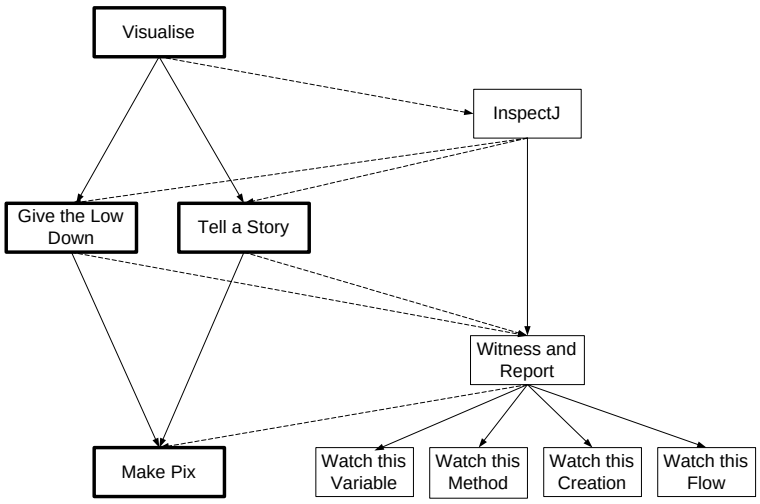

Figure 1: A map of the pattern language.

General Forces

Throughout the process of developing a visualisation for a program, there are certain forces that make themselves apparent again and again. These general forces are:

- The visualisation code should be well organised, to aid understandability. As a consequence, the effort and complexity of the task of visualisation can be controlled and reduced, thus making visualisation development a more viable option.

Pattern	Problem	Solution
1.1 Visualise	How can you improve your understanding of a running program?	Develop a visualisation of the program.
1.2 Give the Low Down.	What will your visualisation depict?	Give the low-down on raw run time information by doing the bare minimum to communicate this information.
1.3 Tell a Story	What will your visualisation depict?	Tell a story about the program by identifying characters and events that will make up a storyline.
1.4 Make Pix	How do you create pictures?	Make pix by breaking down the overall visualisation into small pieces.
2.1 InspectJ	How do you go about visualising a Java program?	Use the InspectJ framework to visualise the Java program.
2.2 Witness and Report	How can you structure monitoring code to track a range of features of a target program?	Divide the monitoring code into Witness and Report blocks.
2.3 Watch this Variable	How do you monitor modifications made to a variable?	Watch this variable by declaring a pointcut for any situation in which it changes.
2.4 Watch this Method	How do you monitor method invocations?	Watch this method by declaring a pointcut for situations in which the method gets called.
2.5 Watch this Creation	How do you monitor the instantiation of objects?	Watch this creation of a new object by declaring a pointcut that describes the situation in which any of the object's constructor gets called.
2.6 Watch this Flow	How do you monitor the flow of control?	Watch this control flow by using the `cflow` mechanism of AspectJ.

Table 1: Summary of the patterns

- The visualisation code should be well organised, to facilitate later maintenance. In certain systems, visualisation code is scattered throughout a range of program components, which makes later modification difficult as it potentially requires *all* of these components to be changed when the visualisation needs to be updated. Therefore the organisation of the code significantly affects the amount of time and effort needed for maintenance.

- The target program should be left unmodified. Certain visualisation approaches require modification of the target program source. This is undesirable as it is intrusive: in other words, it is necessary to change the *original* program code in ways that it was probably not designed to be extended in. Furthermore, changing the source for visualisation purposes may restrict how the program can be modified later on.

Running Example

These patterns feature a running example of the development of a visualisation for a Java simulation program based on the DESMO-J framework (Lechler & Page 1999). DESMO-J utilises container structures and events to model constraint and availability problems.

Currently, applications of this framework may provide textual traces of all program events. However, generated traces tend to be overly long and hard to follow. A visualisation might be more effective at explaining program events.

The steps involved in the development of the visualisation include deciding what to visualise, specifying what information needs to be monitored, defining visualisation behaviour and image rendering.

Suppose you have an application of this framework that models ships entering and leaving a port for the purpose of obtaining loading slots alongside a quay. There are a limited number of loading slots, so if a ship cannot obtain the required number as soon as it pulls into the port, it joins a waiting queue and waits for the required number of slots to become available.

General Visualisation Patterns

1.1 Visualise

Motivation You have a program that you would like to understand more clearly. There are many reasons why you may want to do this. Maybe it is a program that you developed yourself and you would like to explore exactly how it works. Perhaps this program code has been developed by someone else and you have been assigned the task of developing it further, debugging, or testing it. Or maybe you are working on a similar program and therefore you are interested in seeing how this program works. Then again, perhaps you are just curious about the program.

Problem How can you improve your understanding of a running program?

Forces

- The program code may be lengthy and unwieldy.
- There may only be certain features of the program you want to understand better.
- Learning can often be enhanced by using *various* explanatory devices.

Therefore: *Develop a visualisation of the program.*

Visualisation has been used to control and explain software complexity by providing mappings between program elements and cognitive representations of these elements. Noble defines it as the application of computer graphics techniques to computer programs, in the same way that scientific or engineering visualisation applies these techniques to scientific data or engineering artifacts (Noble 1995). Visualisation, which often presents a high level view of a system, can assist programmers in constructing, debugging and maintaining programs. It has also been found to be a useful teaching aid, particularly for describing data structures and explaining algorithms (Oechsle & Schmitt 2002, Najork 2001).

Typical existing visualisation systems range from providing general types of information views, for a wide range of programs, such as UML class and sequence diagrams, to very domain-specific information views for programs from a certain domain. The advantage of using the generic visualisation tools is that they can be used to visualise *many* types of programs, yet their visualisations are often fairly abstract. As for the domain-specific visualisation systems, while their visualisations may be more readily understandable, these systems are often difficult to customise. By developing your own visualisation system, you have control over what types of pictures you end up seeing. Furthermore, if you develop it with reuse in mind, you can make the code generic enough to suit other, similar applications.

Example revisited Applying the solution to the running example, to better understand the internal workings of the simulation framework, develop a visualisation for the port simulation application. This will involve extracting run time information about what the simulation application is doing, and depicting it graphically. A suitable technology needs to be chosen for the graphical component: for this application, we use a Tcl/Tk display, Tcl (Tool Command Language) being a scripting language that makes use of Tk, a graphical user interface toolkit (Ousterhout 1994).

Resolution of Forces By building a visualisation tool for the program you are interested in, you can avoid having to read through pages and pages of potentially confusing code. Furthermore, if you already know which behaviour you are interested in, or which program components you want to pay attention to, you can tailor visualisations to communicate *only* this information at a level of detail that is relevant to you. Finally, having a *different* perspective of the internal operations of the program (aside from the code) may clarify your understanding of the program, as visualisations are useful for teaching and learning (Oechsle & Schmitt 2002, Najork 2001).

Related Patterns If your target program is in Java, consider using **InspectJ (2.1)** as a visualisation system implementation. Independent of the visualisation implementation, you will need to decide what you want the visualisation to show. If you are interested in no-frills run time information, you could **Give the Low Down (1.2)**. Or perhaps you are interested in the causal relationships in the program, in which case you could use the **Tell a Story (1.3)** pattern.

1.2 Give the Low Down

Motivation Software visualisation is all about explaining information about the software through the use of pictures of some form. So to develop a visualisation for the target program, it is necessary to decide on what pictures you want to show.

Problem What will your visualisation depict?

Forces

- The visualisation needs to relay certain run time information.
- Aesthetic appeal is not of high importance.
- Time and development cost should be kept to a minimum.
- The visualisation should be easily understandable.
- Semantic meanings cannot be associated to the program because they would be too abstract.
- Attention needs to be focussed only on a portion of the program.

Identify raw run time information from the program that will be of interest to the viewer of the visualisation. Perhaps this might consist of a program trace, identifying flow of events. Perhaps it will be changes to certain variables as the program progresses.

Decide on a "bare-bones" way to communicate this information. Standard visualisation formats include UML diagrams and flowcharts. Simpler representations can be used as well. If text is chosen as the representation format, ways to illustrate that changes have taken place might consist of special formatting or changing text on a text pane. Another way to represent program information is by use of simple shapes. These shapes, which may be associated to any aspect of a running program, may be changed in order to communicate that the corresponding program aspect has also changed.

The idea that a largely data-driven display counts as a visualisation may seem counter-intuitive at first. If we consider however that visualisation presents a specific "view" of a system, displaying only certain numerical or textual information also counts as a view.

Example revisited Applying the solution to the running example, suppose you are interested in the average waiting time per ship, and whether congestion increases as time progresses. This will involve measuring waiting time for Ship objects within the target program waiting queue, which involves noting when the ship enters the queue and when it leaves the queue. Devise a simple numerical view that periodically outputs average waiting time for the last ten ships, along with a cumulative overall waiting time that averages over all ships so far. Figure 2 shows such a view.

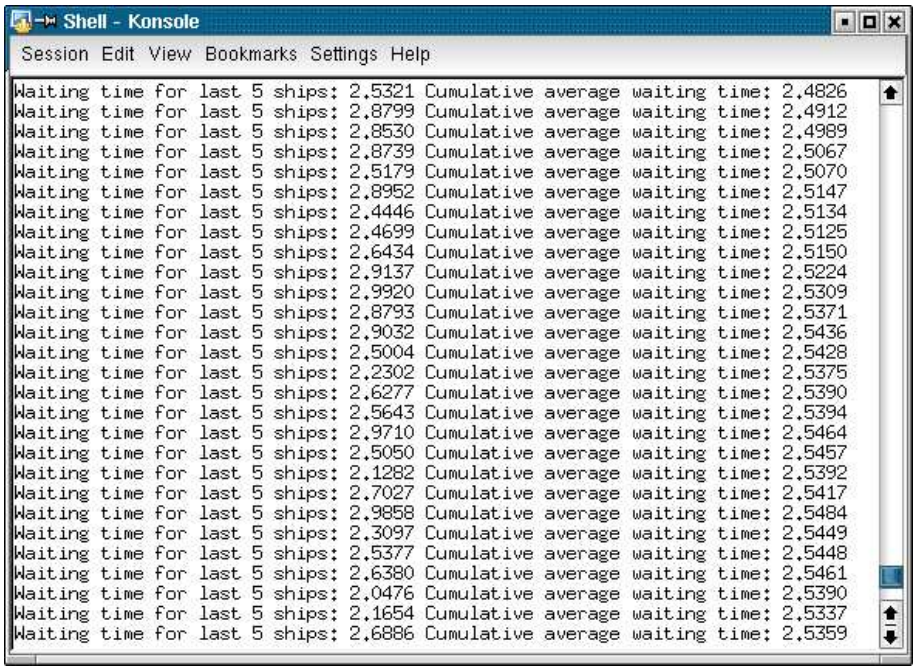

Figure 2: Numerical view of average waiting time for ships

Resolution of Forces The bare-bones visualisation format relays the relevant target program information in a simple and direct manner, without resorting to forcing it into a story form. The simple format usually also means that the time and effort spent are kept to a minimum. Since information about the overall program is not necessarily being communicated by the visualisation, it is possible to focus the desired amount of attention on arbitrary points of interest in the target program.

Related Patterns Next, program monitoring code needs to be written that will obtain relevant program information. If your target program is in Java, consider using the **Witness and Report (2.2)** pattern. This information can then be used to **Make Pix (1.4)**.

1.3 Tell a Story

Motivation Software visualisation is all about explaining information about the software through the use of pictures of some form. So to develop a visualisation for the target program, it is necessary to decide on what pictures you want to show.

Problem What will your visualisation depict?

Forces

- The visualisation needs to relay run time information.
- Object-oriented programs often relate to real world objects and interactions. However, visualisations rarely "semantically" depict these real world relationships.
- The visualisation should be easily understandable.
- Time and effort are not too limited.
- The visualisation should be interesting: it should present graphics as opposed to simple numerical views and these graphics should not be graph-based pictures or standard views.

Therefore: *Tell a story about the program components and their interactions with each other.*

Identify the *characters* in the program. Perhaps they are important recurring concepts in the program, or objects that carry significant responsibility. Maybe they have real-world counterparts. The characters should be vital components in

explaining what it is that the program does. Decide on some visual representation for each character.

Next, identify the *events* that happen to the characters. An event can be thought of as anything major that happens to a character, or anything major that the character causes. Decide on some way of representing the event such that it is clear how it affects the character or how the character causes it.

This combination of characters and their related events forms a story.

This approach is reminiscent of Rehearsal World, which is a visual programming environment which involves moving "performers" around "stages" (Finzer & Gould 1993).

Example revisited Applying the solution to the running example, the main characters in this story are ships, the ship waiting queue, the quay side and the port. Events that need to be modeled are ships entering the port, ships leaving the port, ships entering the queue, ships leaving the queue, ships obtaining loading slots and ships that have completed the loading process. You can use cartoon representations for the ships (see Figure 3) and divide the Tk canvas into port entry, port exit, loading and queuing areas (see Figure 4). Depict events by moving ships in and out of these areas.

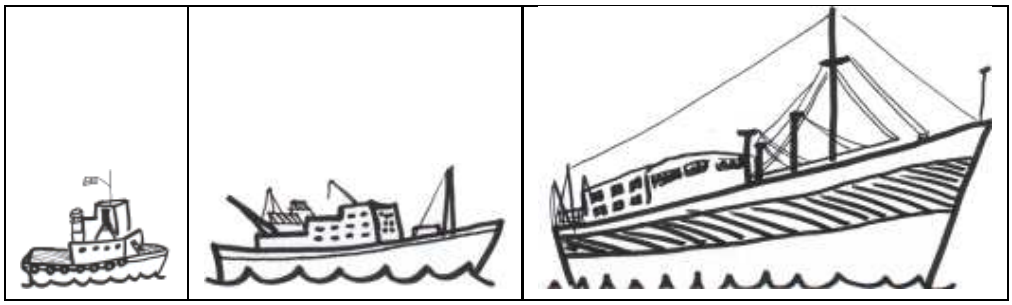

Figure 3: Planned ship representations for ships of different sizes

Resolution of Forces Run time events are represented by the story-line. By carrying through information about relationships between objects from the target program through to the visualisation, semantic meaning inherent in the target program is not lost. Story telling has been used for a long time to explain existing concepts: visualisation by way of story telling should therefore be easy to understand, since story telling is a conventional form of communication. While development of this type of visualisation might take slightly more time than a more data-oriented view, with experience it will take a *story-teller* less and less time to develop these visualisations, as they will establish good techniques and potentially reusable code. Finally, although raw data displays can be unquestionably useful sometimes, watching a visualisation that contains some type of plot is often more interesting than watching a raw data display!

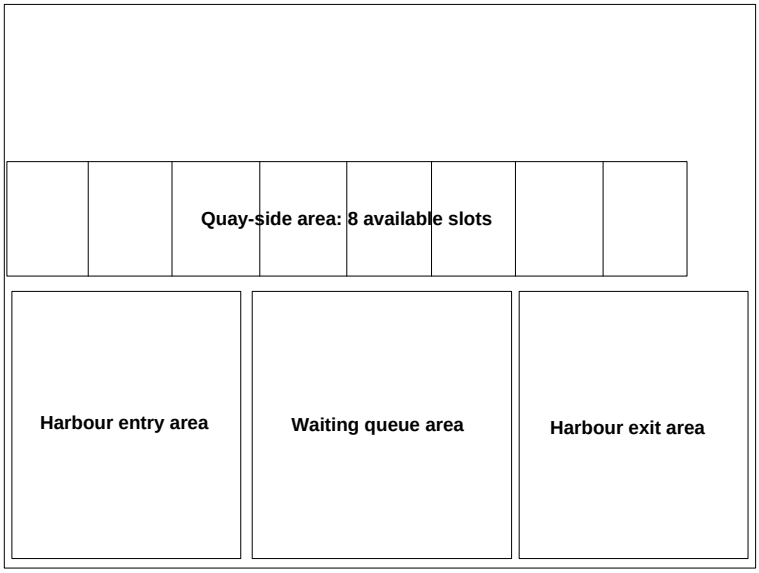

Figure 4: Areas of movement for the ships

Related Patterns Next, program monitoring code needs to be written that will obtain relevant program information. If your target program is in Java, consider using the **Witness and Report (2.2)** pattern. This information can then be used to **Make Pix (1.4)**.

1.4 Make Pix

Motivation After deciding what the visualisation will show, and writing the monitoring code to obtain the necessary information, the only step that remains is to develop the graphical side of the visualisation.

Example Carrying on with the port simulation example, the visualisation format chosen was "Tell a Story", and the important characters and events have been identified. Using "Witness and Report" blocks, monitoring code has been written that will collect information about these characters and events. This information needs to be displayed now.

Problem How do you create pictures?

Forces

- Each program element being monitored contributes different data to the overall picture.

- The picture needs to be updated regularly to reflect the changing state of the program.
- Time and effort expenditure should be kept to a minimum: taking the entire process of developing a visualisation into consideration, this stage is most likely to consume the *most* time and effort.

Therefore: *Make pix by breaking down the overall visualisation into small pieces.*

The "break-down" approach can be applied both to information acquisition and organisation tasks and also to drawing tasks. In other words, have methods or procedures which are responsible for receiving certain types of information. Their responsibilities include performing any final data translation and triggering drawing of this information.

One way to organise these procedures might be to align procedures with events that are related to characters: the procedures may take information about the event and related characters as arguments.

If the visualisation is small, the drawing triggered by the method or procedure might occur within the method or procedure itself. However, for more complicated visualisations, have methods or procedures which explicitly deal with the task of drawing specific elements of the visualisation. The "drawings" themselves may be created using the graphics facilities of the visualisation component, or they might be existing pictures that get changed somehow to illustrate program changes.

Example revisited Applying the solution to the running example, since the visualisation component is Tcl/Tk, write procedures that receive information about new ships and departing ships (responsible for depicting new ships entering the harbour and leaving the harbour respectively). Write procedures that deal with adding and removing ships from the waiting queue, and one that deals with redrawing the queue. Analogous to the waiting queue procedures, write procedures for adding and removing ships from the loading area, as well as a procedure that draws the loading area. Write a generic procedure that moves an object from one part of the canvas to another. Use hand drawn representations of ships as objects. Figure 5 shows a screen capture from this visualisation.

Resolution of Forces Applying the "break-down" principle, writing procedures that receive information about different monitored elements allows them to act independently. Since it is these information receival procedures that control the triggering of information drawing, visualisation redrawing does not have to occur all at once. Different parts of the picture change at different rates, according to relevant triggers. These triggers directly correspond to the occurrence of relevant events in the target program, so the overall picture does update to reflect the change of state in the program. In fact, if written generically enough, these procedures may be reused for other visualisations for different target programs.

Figure 5: DESMO-J visualisation

Finally, identifying exactly what needs to be done in the form of small tasks (to be achieved by procedures) should help maintain some sort of upper bound on the amount of work that needs to be done.

InspectJ Patterns

2.1 InspectJ

Motivation You have a Java program that you would like to understand more clearly. Live visualisation can be used to provide a graphical explanation of programs at run time, so you have decided to use visualisation to improve your understanding.

Problem How can you dynamically visualise a running Java program?

Forces

- Modification of the source is undesirable as it may introduce bugs into the source.
- Time and effort expenditure should be kept to a minimum since the visualisation is probably *not* the main goal to achieve.

Therefore: *Use InspectJ to visualise the Java program.*

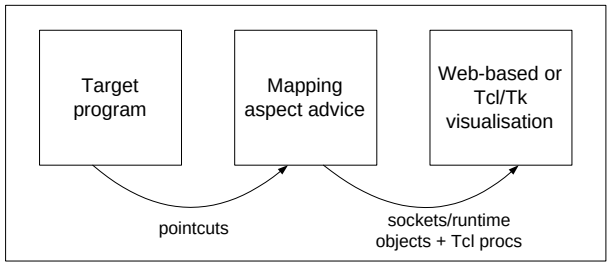

Figure 6: InspectJ

InspectJ (Khaled 2003) is a program visualisation framework that is based on a modified Program Mapping Visualisation (PMV) model (Jerding & Stasko 1994), which was developed by Stasko, Roman and Cox. The model is a fairly standard program visualisation conceptual architecture, and consists of three components: a Program component, which deals with representing the target program to the visualisation system, a Mapping component, which acts as an information filter and also as a translator, and a Visualisation component, which handles image rendering. The separation of the components means that reusability can be increased, since changes in one component will not necessarily cause changes in another.

It can however be time consuming to build multiple sets of three components. The InspectJ version of the PMV model, shown in Figure 6, harnesses the fact that AspectJ may be used to monitor running Java programs and extract run time information about it. This is achieved by combining AspectJ code with target program code at compile time.

While conceptually it consists of three components like the PMV model, since the AspectJ code is used in combination with the target program itself to obtain information, effectively the target program ends up playing the role of the Program component. Thus the main components to be developed consist of the Mapping and Visualisation component. The Mapping component is mostly AspectJ code, while the Visualisation component can be assembled from whichever platform best suits the visualisation needs of the application.

Example revisited Applying the solution to the running example, to better understand the internal workings of the simulation framework, use the InspectJ model to develop a visualisation for the port simulation application. This will involve using AspectJ to extract run time information about what the simulation application is doing, and passing this information on to a visualisation component. A suitable technology needs to be chosen for the visualisation component: for this application, we use a Tcl/Tk display, Tcl (Tool Command Language) be-

ing a scripting language that makes use of Tk, a graphical user interface toolkit (Ousterhout 1994).

Resolution of Forces By virtue of InspectJ being aspect-oriented (Kiczales, Lamping, Menhdhekar, Maeda, Lopes, Loingtier & Irwin 1997), the program source code itself is left unmodified. Since modification is avoided, introduction of modification-related bugs is also avoided. Visualisation code is constrained within visualisation-related aspects, which means that later modification of visualisation code is more feasible. If constructed well, entire visualisations can be reused for different applications, due to the insularity of the model and of aspects themselves. Assuming that the developer of the visualisation system is familiar with the Java programming language, as a consequence of AspectJ being similar to Java, development of visualisation code should not take long since technology learning time no longer needs to be factored in.

Related Patterns Decide what you want the visualisation to show. If you are interested in no-frills run time information, you could **Give the Low Down (1.2)**. Or perhaps you are interested in the causal relationships in the program, in which case you could use the **Tell a Story (1.3)** pattern. Once you have decided, you will need to use the **Witness and Report (2.2)** pattern to obtain relevant program information at run time.

2.2 Witness and Report

Motivation Visualisations portray information about a range of features of the target program. Depending on the type of visualisation on display, the program needs to be monitored for different information.

Example Continuing with the example from **Tell a Story (1.3)**, the port simulation program needs to be monitored to capture events dealing with ships, the queue, or the quay side. Additional visualisation-specific behaviour needs to be specified upon "seeing" these events.

Problem How can you structure monitoring code to track a range of features of a target program?

Forces

- Modification of the source is undesirable as it may introduce errors.
- Each feature may need to be monitored in different ways, with different resulting behaviour for each important event.

Therefore: *Divide the monitoring code into situations for information capture, and corresponding visualisation program behaviour in those situations: in other words, use Witness and Report blocks.*

A Witness and Report block specifies what to do if a certain situation in the target program occurs. For example, there may be a Witness and Report block specifying visualisation related behaviour if a certain method in the target program gets executed. Witness and Report blocks make use of AspectJ syntax — the "Witness" portion consists of the conditions in which to activate the special behaviour (otherwise known as a `pointcut`) while the "Report" portion, which describes the resulting action, corresponds to the AspectJ version of a method (otherwise known as `advice`, which gets triggered from appropriate pointcuts).

AspectJ pointcuts can be used to describe almost any well-defined point of execution within the target program (Kiczales et al. 2001). This means that "Witness" points can be set up for almost any point of execution in the target program. Pointcuts can be composed of other pointcuts, allowing reuse of "Report" behaviour for different points of execution.

These Witness and Report declarations are contained within a program monitoring aspect.

Example revisited Applying the solution to the running example, make a program monitoring aspect for housing Witness and Report blocks for the port simulation. Declare Witness pointcuts for the following situations: creation of a new Ship object, when a ship enters the port, when a ship joins the waiting queue, when a ship leaves the queue, when a ship obtains quay slots for loading, when a ship has finished loading, when a ship leaves the port. For each pointcut, define a corresponding Report: notify the visualisation component of the event. Figure 7 shows how Witness and Report blocks relate to target program source code.

Resolution of Forces Witness and Report blocks relate target program conditions and consequential actions together, thus making visualisation code understandable. Furthermore, since these blocks are separate from each other, they can be easily modified to specify new conditions or new consequential actions. However, in the event that visualisation code needs to be largely rewritten, since all Witness and Report blocks are constrained to one area by being grouped together within a program monitoring aspect, overall modification is still fairly straight forward. Finally, since aspects are completely external to the target program code, the target program source does not need to be altered at all.

Related Patterns Each kind of Witness and Report block achieves a specific monitoring task, that will help you **Make Pix (1.4)**. If variable watching is the goal, **Watch this Variable (2.3)** may be used. If the task is method monitoring, **Watch this Method (2.4)** may be used. **Watch this Creation (2.5)** is useful

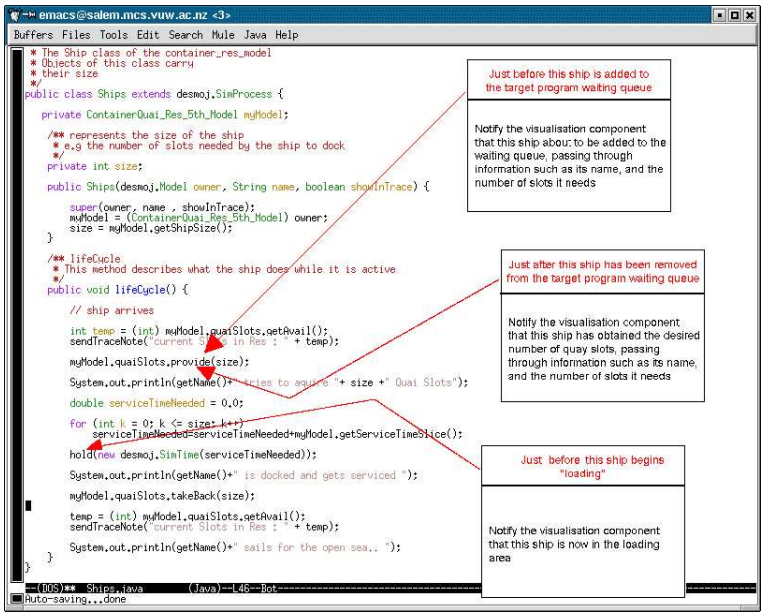

Figure 7: Code interspersed with Witness and Report blocks

if the task is monitoring object initialisation, while **Watch this Flow (2.6)** captures program control flow.

2.3 Watch this Variable

Motivation Variable state makes up program state. Consequently, a useful way to trace how the target program is changing is to track how its variables are changing. Some variables offer useful program run time information all on their own, while others give useful comparative information in relation to others. Depending on what the visualisation is going to show, different variables need to be focussed on.

Problem How do you monitor modifications made to a variable?

Forces

- The target program should be left unmodified, as modifications may introduce errors.
- Java variables may have an access level of public, protected or private.

Therefore: *Watch the variable by declaring a pointcut for any situation in which it changes.*

This involves defining a Witness and Report block that uses a pointcut for its

Witness portion that specifies the name of the variable, the type of object it belongs to, along with the new value of the variable.

```
before(int tns, SimProcess s): set(int SimProcess.totalNumShips) &&
                               args(tns) && this(s){
    writeToVisComp(tns);
}
```

In the above example, the `totalNumShips` variable is being monitored. Its new value gets captured in the `tns` parameter, which gets passed through to the visualisation component via the `writeToVisComp` method.

In some situations, the new value does not need to be immediately reported to the visualisation component but rather needs to be recorded for later use. If the variable is private, AspectJ `introduction` may be used to add "dummy" variables to objects for the purpose of holding mirror values. In the example below, `introduction` is used to add a private variable called `numShips` to the SimProcess class. `numShips` acts as a dummy variable for `totalNumShips`.

```
aspect SimTracker {

    private int SimProcess.numShips = 0;
    .
    .


    before(int tns, SimProcess s): set(int SimProcess.totalNumShips) &&
                                   args(tns) && this(s){
         s.numShips = tns;
    }

    .
    .
}
```

Resolution of Forces The Witness and Report block format facilitates understanding and later modification. By capturing whenever the variable is about to change, along with the new value it is about to receive, the access level of the original variable does not matter.

Related Patterns If you are interested in everything that happens to objects of a certain type, perhaps you should use **Watch this Method (2.4)** and **Watch this Creation (2.5)** on objects of the right type.

2.4 Watch this Method

Motivation Program methods embody program behaviour, so monitoring the execution of methods gives an idea of how the program is behaving. As with variables, depending on what the visualisation will show, sometimes certain methods will yield more relevant information than others.

Problem How do you monitor method invocations?

Forces

- The target program should be left unmodified, as modifications may introduce errors.
- Depending on the visualisation representation format, special method information may be needed either just before or just after the method has been called.

Therefore: *Watch the method by declaring a pointcut for situations in which the method gets called.*

The method call pointcut makes up the Witness portion of a Witness and Report block. To specify when a specific method gets called, usually it is necessary to specify the name of the method as well as the type of object it is being called upon.

If the visualisation component needs to be notified about the method call immediately *before* it has actually occurred in the target program, the pointcut may take the following form:

```
before(Ships s): call(void lifeCycle()) && target(s) {
    ..
}
```

If the visualisation component needs to be notified *after* the method call, the pointcut may be structured as follows:

```
after(Ships s): call(void lifeCycle()) && target(s) {
    ..
}
```

Resolution of Forces The Witness and Report block format facilitates understanding and later modification. The `before` and `after` AspectJ mechanisms allow for the Witness portion of a Witness and Report block to specify appropriate activation points for the corresponding Report advice.

Related Patterns While constructors are a form of method, AspectJ does not consider them to be methods, therefore constructor monitoring cannot be achieved via method monitoring. To monitor initialisation, use **Watch this Creation (2.5)**.

2.5 Watch this Creation

Motivation Objects are fundamental building blocks in object-oriented programming. They model concepts in the application domain, so monitoring their creation sheds some light on how the program works. Object creation information may be helpful in locating program bugs, or showing patterns of memory usage.

Problem How do you monitor the instantiation of objects?

Forces

- The target program should be left unmodified, as modifications may introduce errors.
- Depending on the visualisation representation format, special object creation information may be needed either just before or just after the object has been created.
- You only want to know about the creation of certain types of objects, not all objects.
- The class that the object belongs to may have multiple constructors.

Therefore: *Watch the creation of new objects by declaring a pointcut that describes the situation in which any of the object's constructors get called.*

This object constructor pointcut, which will make up the Witness portion of a Witness and Report block, gets called whenever new objects are created. Some objects may have multiple constructors, which are differentiated in the number of parameters they take. This means that one object may possess constructors with different signatures.

Declaring a pointcut to capture object initialisation involves specifying what class the object belongs to. To make this pointcut applicable for *all* constructors of this class, AspectJ "wildcard" syntax needs to be used. This allows constructor matching to occur for constructors of the given class with any number of parameters.

In many cases, it will be useful to be notified just after a new object has been created, which is specified using `after` syntax.

```
after(ContainerModel cm): (execution(* .new(..)) && target(cm)) {
    ..
}
```

The code featured above is an example of some `after` advice which captures the execution of all constructors where the targeted object is of type `ContainerModel`.

Sometimes, the visualisation requires some sort of preparatory work to be done for new objects. In these cases it is useful to be notified just before the object has been created. This is specified using `before` syntax.

```
before(ContainerModel cm): (execution(* .new(..)) && target(cm)) {
    ..
}
```

Resolution of Forces The Witness and Report block format facilitates understanding and later modification. Specification of the object type allows focus to be placed on the right type of object. The `before` and `after` mechanisms allow flexibility in object creation monitoring. The wildcard constructor syntax allows further flexibility in constructor signature matching.

Related Patterns If you are interested in a very specific object of the given type, **Watch this Variable (2.3)** is a reasonable alternative.

2.6 Watch this Flow

Motivation If the program is relatively small, then attempting to pay attention to its behaviour in an overall sense is a feasible goal. One way in which an overall view of behaviour can be gained is by obtaining a trace of control flow, or program event flow. Since object-oriented programming focusses on the interaction between objects, these events include constructor calls and method calls.

Problem How do you monitor the flow of control?

Forces

- The target program should be left unmodified, as modifications may introduce errors.
- Even for a relatively small program, since all methods and constructors are of interest, it quickly becomes unreasonable to use "Watch this Method" and "Watch this Creation" to trace events.

Therefore: *Watch the control flow by using the* `cflow` *mechanism of AspectJ.*

The `cflow` pointcut, which will form part of a Witness and Report block, picks out *every* well-defined point of execution in the target program. To narrow down the scope of the `cflow` declaration, part of the Witness pointcut should specify that only method calls and constructor calls should be looked at. This is specified using the following syntax:

```
call(* *(..)) || call(* .new(..))
```

As well as narrowing down the scope of the `cflow`, the Witness pointcut needs to specify *which* part of the program the control flow should start being monitored from, in other words, which method the control should be in before monitoring begins. This is specified as below, with *methodName* modified as appropriate:

```
cflow(withincode(* methodName(..))
```

`cflow` is a recursive mechanism: it gets called repetitively, from the point of current control. To avoid stack overflow, resulting from infinite looping caused by the aspect recursing on itself, the pointcut should exclude the aspect from the `cflow` declaration, by specifying that control must *not* be within the aspect definition.

```
!within(AspectName)
```

Below is a complete example of a Witness pointcut that uses `cflow` to capture all constructors and methods called from within the control of the `startExperiment()` method. The aspect that houses the pointcut definition is called `ProgramMonitor`.

```
pointcut cthisflow(Object o): cflow(withincode(* startExperiment(..)))
                              && ((call(* *(..)) || call(* .new(..)))
                              && target(o)) && !within(ProgramMonitor);
```

Resolution of Forces The Witness and Report block format facilitates understanding and later modification. Given that the AspectJ `cflow` mechanism provides a fairly generic way for "capturing" all run time events, it can be harnessed within a Witness pointcut to focus on constructor and method calls that occur from the control of any specified method.

Related Patterns This pattern is effectively **Watch this Method (2.4)** and **Watch this Creation (2.5)** tied together recursively. If recursing is to be avoided, either because multilevel recursing in unnecessary or the program needs to run faster, those patterns should be used instead.

Acknowledgments

I would like to thank my shepherd, Alejandra Garrido, and my ever-helpful supervisors James Noble and Robert Biddle, for helping me get this paper into a respectable form. I would also like to thank everyone who participated in my writers' workshop for providing useful comments and insights on the paper.

References

Coplien, J. (1996), *Software Patterns*, SIGS Books.

Finzer, W. F. & Gould, L. (1993), *Watch What I Do: Programming by Demonstration*, The MIT Press, chapter Rehearsal World: Programming by Rehearsal.

Jerding, D. F. & Stasko, J. T. (1994), Using Visualization to Foster Object-Oriented Program Understanding, Technical Report GIT-GVU-94-33, Atlanta, GA, USA.

Khaled, R. (2003), InspectJ: Program Monitoring for Visualisation using AspectJ, *in* 'Proceedings of the Twenty-Sixth Australasian Computer Science Conference (ACSC2003)'.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001), 'An Overview of AspectJ', *Lecture Notes in Computer Science* **2072**, 327–355.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997), Aspect-Oriented Programming, *in* M. Akşit & S. Matsuoka, eds, 'Proceedings European Conference on Object-Oriented Programming', Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp. 220–242.

Lechler, T. & Page, B. (1999), DESMO-J : An Object Oriented Discrete Simulation Framework in Java, *in* 'Proceedings of EUROSIM '99'.

Najork, M. (2001), Web-based Algorithm Animation, *in* 'Design Automation Conference', pp. 506–511.

Noble, J. (1995), Abstract Program Visualisation, PhD thesis, Victoria University of Wellington.

Oechsle, R. & Schmitt, T. (2002), JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI), *in* 'Software Visualization', pp. 176–190.

Ousterhout, J. K. (1994), *Tcl and the Tk Toolkit*, Addison Wesley.