A Case Study of Cartoon Visualisation using AspectJ

Rilla Khaled **James Noble** **Robert Biddle**

School of Mathematical and Computing Sciences
Victoria University of Wellington
Wellington, New Zealand
{rkhaled,kjx,robert}@mcs.vuw.ac.nz

Abstract

In this paper, we present a real-time visualisation of a port simulation program. It makes use of *cartoon visualisation* and was created using InspectJ, our AspectJ-driven visualisation framework. AspectJ is an aspect-oriented extension to Java that allows concerns that cut across a program to be addressed explicitly in aspect components, thereby supporting a novel approach to visualisation control. InspectJ renders visualisations on a Tk canvas. The combination of using AspectJ for program monitoring and the Tk canvas made creation of the visualisation reasonably fast and fairly straightforward. We discuss interesting issues related to the visualisation of frameworks, as well as future directions for framework visualisation, such as the web-based visualisation creation systems. We also discuss cartoon animation as a means of visualisation, along with its potential applications.

Keywords: Program visualisation, simulation, AspectJ

1 Introduction

Typical 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. In this paper, we present a real-time visualisation of a port simulation program which was put together over a single weekend. It makes use of *cartoon visualisation* and was created using InspectJ, our AspectJ driven visualisation system. AspectJ is an aspect-oriented extension to Java that allows *cross-cutting* concerns, such as error handling or event logging, to be addressed explicitly in aspect components. We regard monitoring for visualisation as one such concern, and so we suggest AspectJ supports a novel approach to visualisation control.

The rest of the paper is structured as follows. Section 2 reviews some background concepts related to the simulation program, namely Simula, DEMOS and DESMO-J. Section 3 discusses the architecture and components of the InspectJ visualisation system. Section 4 describes the design of the visualisation, as well as some implementation details. Section 5 discusses the issues relating to the technology, in particular aspect-oriented visualisation, frameworks, and the the visualisation technique. Finally in Section 6, we present some conclusions.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Australian Symposium on Information Visualisation, Adelaide, February 2003. Conferences in Research and Practice in Information Technology, Vol. 24. Tim Pattison, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Background

Simula The Simula programming language was designed and built by Kristen Nygaard and Ole-Johan Dahl in the 1960s. Although the language was originally intended to be used for the specific purpose of modelling discrete events, it was later expanded into a full scale general purpose programming language, and is considered to be the original object-oriented language. As well as introducing important object-oriented concepts such as classes and inheritance, a strength of Simula is in its preservation of the underlying model of the system under study.

DEMOS In 1979, Graham Birtwhistle developed DEMOS (Discrete Event Modelling on Simula), a Simula based framework for discrete simulation. Discrete simulations simulate systems that change states at discrete points in time. Typical types of concerns to model involve queuing, availability checking and storage strategies.

A framework is a set of classes created for solving problems from a certain domain (Johnson 1997). The generic framework provides a starting point from which applications to solve particular tasks can then be developed. The major advantage of using frameworks is that developers need only extend or redefine some classes in the framework according to their application needs, as opposed to having to entirely create the application from scratch. As a consequence, development time of applications is greatly reduced.

DEMOS was used in simulation courses and was found to be particularly effective in explaining object-oriented programming and simulation concepts. DEMOS has had various successors: in 1989, graduate students at the computer science department of the University of Hamburg developed DESMO in Modula-2. Since then, the DESMO framework has been implemented in Oberon, Smalltalk and C++. In 1998, Thomas Schniewind implemented DESMO-C, and added extra modelling concepts to the original framework.

DESMO-J DESMO-J (Discrete Event Simulation and MOdelling in Java) is based on DEMOS: it is a simulation framework for modelling discrete event simulations (Lechler & Page 1999). In a framework, common modelling concerns are already implemented

in "skeleton" classes which may be tailored to suit specific applications.

While DESMO-J was developed with the intention of being used for educational purposes, its capacity for modelling in either an event-oriented or process-oriented style, along with its ability to reuse models inside more complex models, makes it a fairly flexible framework with sufficient simulation power.

Currently DESMO-J outputs results of simulation experiments by generating statistics pages at the end of each simulation run. In other words, apart from the textual information accessible at the end of each experiment, there is no other form of live visualisation for each simulation.

We decided to create program visualisations for DESMO-J in order to explore the issues that arise while visualising framework applications. The information depicted in the visualisation was to correspond to events triggered within DESMO-J programs, namely method calls and data structure modifications. Being a simulation framework, DESMO-J program states closely correspond to the "real-world" situation being modelled, consequently there is not much difference between the program state and the simulated scenario. Therefore, one way to communicate program state information is to depict the corresponding real world scenario.

We set out to achieve this using InspectJ, our aspect-oriented visualisation framework. Another reason why visualisation of DESMO-J makes sense is related to DESMO-J serving as an educational tool: it has been claimed by some that program visualisation is helpful for explaining how programs work. Visualisation of DESMO-J provides another view of simulation concepts.

3 Overview of InspectJ

To create the visualisation, we used InspectJ (Khaled 2002). InspectJ is an AspectJ-driven program visualisation system. In keeping tradition with Stasko's PMV model (Jerding & Stasko 1994), it is loosely based on the idea of having three components (Program, Mapping and Visualisation) that work together. Its architecture can be seen in Figure 1.

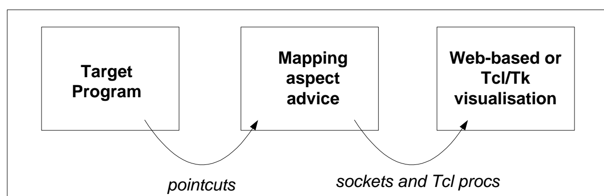

Figure 1: The architecture of the InspectJ system

3.1 Aspect-oriented programming and AspectJ

The goal of aspect-oriented programming is to present developers with mechanisms to cleanly deal with *cross-cutting concerns*, where a cross-cutting concern is a programming concern composed in a certain way that orthogonally cuts across other programming concerns. Examples include tasks such as adding error-handling across a system, adding pre and post condition checking, system logging and program monitoring. In AOP terms, a property that cannot be cleanly encapsulated in a generalised procedure is called an *aspect*. The goal of AOP is to give programmers a mechanism to compose *components* (properties that can be cleanly encapsulated in a generalised procedure, which may include objects, methods or procedures) and *aspects*, while retaining separation from each other.

AspectJ (Kiczales, Hilsdale, Hugunin, Kersten, Palm & Griswold 2001) is a Java-based aspect-oriented language which has existed since 2001. It is an extension of the Java programming language, equipped to deal with cross-cutting concerns: while it has the capacity to deal with regular Java classes, the special AOP "cross-cutting" type of class is called an *aspect*. It was designed to be used in conjunction with Java and can be used with existing Java programs.

The creators of AspectJ claim that by separating out the two different types of concerns, it should be significantly easier for programmers to write component programs in such a way that their structures clearly (and cleanly) represent the properties they were meant to be embodying, and write aspect programs to carry out whichever cross-cutting concern need be addressed. They also claim that this separation should aid understanding of various system parts by the programmer, as well as simplify development and lessen development time (Kiczales, Lamping, Menhdhekar, Maeda, Lopes, Loingtier & Irwin 1997, Kiczales et al. 2001).

AspectJ works by combining the aspect programs with the component programs. In order to specify at which point aspect programs should be combined into the component program, programmers need to specify the exact places in the component program where the aspect programs should be combined. This place is generally referred to as a *join point* (see Figure 2).

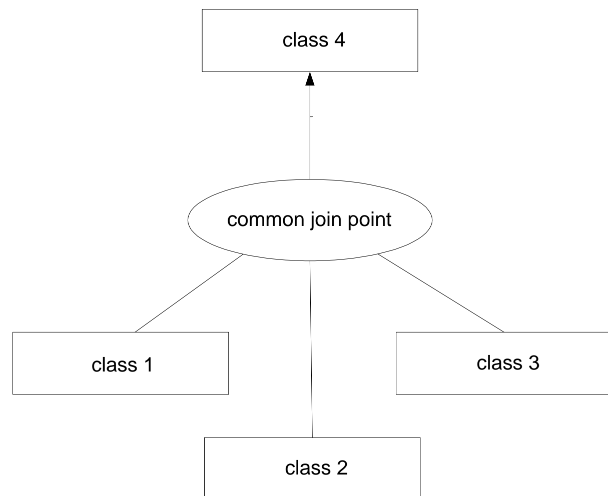

Figure 2: Dealing with cross-cutting concerns using AOP

3.2 Visualisation is a cross-cutting concern

Program monitoring for visualisation involves capturing calls to methods of interest or changes to certain data structures, so that information of these events can be relayed visually. Typically, these events occur *everywhere* in the target program: they cannot be isolated to a specific area, otherwise they would not reveal useful information about the running of the program as a whole. Even if only one type of event is to be visualised, cross-cutting still occurs.

Therefore it seems logical to use AspectJ for program monitoring since AspectJ is a realisation of AOP, which specifically deals with modularising

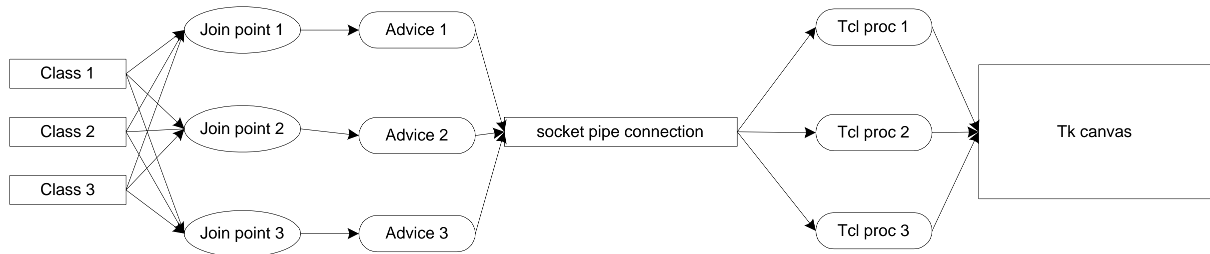

Figure 3: Information flow in InspectJ

cross-cutting concerns. This is especially useful when visualisation systems get updated, because changes are localised within aspects. Therefore, not only does updating become a lot easier, development time is lessened.

3.3 InspectJ

Visualisation systems range from being specific to a certain task, e.g. specialising in the visualisation of sorting algorithms, to being fairly generic, e.g. specialising in the visualisation of a multitude of features of running Java programs.

InspectJ can currently be used to generate live visualisations for various different programs. For each program, different visualisations are created: these visualisations range from sequence diagrams to communicate program trace information, to standard sorting and stacking algorithm visualisations, and also to domain specific visualisations relating to certain programs.

InspectJ uses AspectJ aspects to define additional visualisation pertinent behaviour, with respect to the current *target program*, which is the computer program that is to be visualised. One of the advantages of using aspects to define additional program behaviour is that it is not necessary to directly modify the original program. In other words, it is possible to obtain information about the target program, purely through the use of AspectJ aspects, without changing anything about the target program or supporting software and hardware. While the Java Platform Debugger Architecture also allows target program events to be tracked, the process of specifying event capture conditions using the the Java Debug Interface (JDI), which is a layer of the JPDA, is more involved than writing an equivalent join point in AspectJ. This can be attributed to the fact that AspectJ was *designed* to deal with cross-cutting concerns, and therefore specification of visualisation monitoring points in programs is straightforward.

Another immediate advantage of using AspectJ for visualisation is that since it is essentially Java, as long as programmers are already familiar with Java, they are not faced with a steep learning curve.

Figure 3 contains a broad view of information flow within InspectJ. The AspectJ compiler compiles target program classes and particular visualisation-specific aspects together, so that whenever a set of join point conditions is satisfied, target program information is passed through to aspect *advice*, which are like methods for aspects. The advice performs any mapping transformation and then sends transformed data through a socket connection to a Tcl program, created specifically for the particular visualisation, which renders information on a Tk canvas.

3.4 Program component

The program component is the part of the visualisation system that traditionally has output information about the target program. In terms of InspectJ, the program component can be characterised as the join points and advice that work together to release target program information at certain points of interest. The developer of the visualisation system needs to be knowledgeable of the target program source in order to be able to specify the relevant points of interest. As for the advice that gets activated at the join points, its main responsibility is to collect program information from the interesting points and report it to the mapping component.

3.5 Mapping component

In the PMV model, the mapping component has typically been the component that receives information from the program component, applies any necessary transformations to it and then sends it off to the visualisation component. In InspectJ, the mapping component is composed of advice and/or methods declared within AspectJ aspects. Since the mapping component is responsible for reporting to the visualisation component and the visualisation component is *not* directly powered by AspectJ, the mapping component needs to open some sort of communication pipe to the visualisation component. After opening this channel, the mapping component updates the visualisation component whenever it receives target program information. For InspectJ, this communication pipe consists of a client/server socket connection between the mapping and visualisation components, where the mapping component sends update information and requests to the visualisation component.

3.6 Visualisation component

One of the strengths of the PMV model is that it is easy to completely change the visualisation format, e.g. change the visualisation component, and leave the rest of the visualisation system unaffected: this is because the separation of the components provides insulation against change. As a consequence, there is a lot of flexibility available in choosing what sort of "front-end" the visualisation system is to have. The choice of what to use as the front-end is an important one because the front-end is what the users of the visualisation system will see: the drawing/display capabilities have to be able to represent the information passed through from the mapping component. Yet at the same time it is desirable, from the point of view of the developers of the visualisation system, to choose some technology that is easy to learn and/or easy to use, to reduce development time. For the port simulation example, Tcl/Tk canvasses (Ousterhout 1994) were the main type of visualisation component we

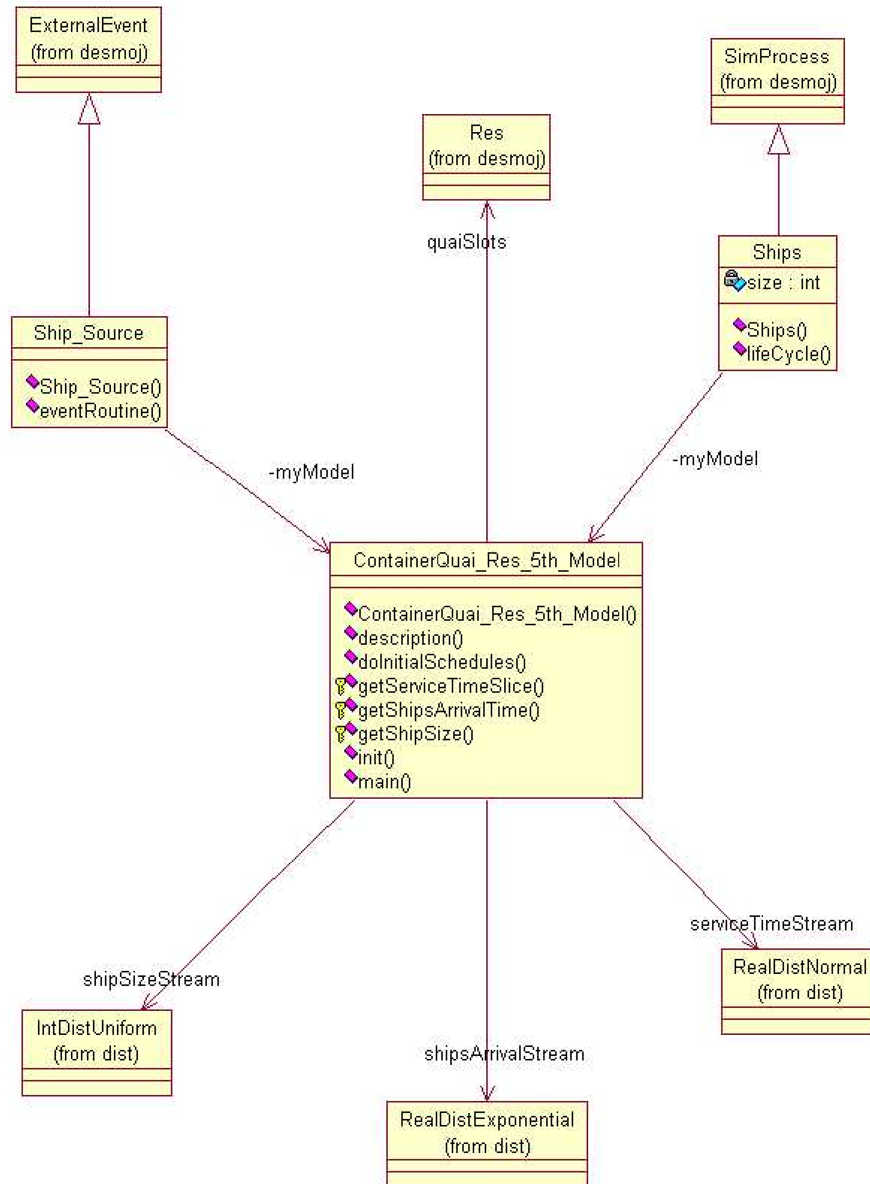

Figure 4: DESMO-J port simulation class diagram, showing inheritance from framework components

used. With Tk canvasses, it is easy to quickly create helpful looking images: it is capable of drawing standard shapes and rendering existing images. It is also straightforward to change Tk object coordinates. For these reasons, we chose to use it.

4 Creating the visualisation

4.1 The target program

The target program, which is a part of the DESMO-J tutorial, is a program modelling the water side of the quay of a port (Neidhardt 1999). This follows a tradition of program examples dating back to Simula and DEMOS. The quay has 8 quay slots on which incoming ships can be served. Ships entering the port to be loaded and unloaded require 1 to 3 quay slots, depending on their size.

Quay slots are a limited resource: when a ship enters the port, it tries to obtain contiguous slots. If there is enough free space for the ship, it moves into the slots, docks, loads, then undocks and leaves the port. However, if enough free space to accommodate the ship cannot be obtained, the ship joins a queue and waits for quay slots to become free.

While the simulation experiment is running, details of ships arriving, leaving and queuing get output to the standard output stream. After the experiment has finished, various simulation statistics can be viewed from HTML report pages that get generated at the end of each experimentation run.

Figure 4 features a UML class diagram of the main classes involved in the simulation.

4.2 Design of the visualisation

For certain applications, visualisation is often a visual form of narrative. Typical narrative-based visualisation systems take a lot of effort to create. In order to create narrative-based visualisation systems quickly, a lightweight approach must be adopted, as some quality will have to be sacrificed in order to save creation time. The combination of these three factors, namely visualisation, narrative, and lightweightness seem to point to an obvious solution: cartoon animation. By cartoon animation, we mean a kind of iconic visualisation, where the icons are simple (cartoon) repre-

sentations of objects in the application domain, and these icons are animated simply to represent change and interaction as is common in cartoons.

Although cartoon animation has not been used much in the past as a way of software visualisations, cartoon animation seems like it could be a way to create visualisations quickly. This seems particularly appropriate for visualisation that involves representing activity in a tangible application domain, such as the port simulation. It would be less appropriate for visual representation of a more abstract or complex application domain, or for software internals.

Cartoons on television typically depict stories, which are effectively event traces that describe interactions between things. Cartoon visualisation can be used to explain software in the same manner: if certain images are associated with objects, modules or methods, then every time the object, module or method gets acted upon or acts upon something else, appropriate pictures can be drawn.

Our aim was to create cartoon visualisations for this application using InspectJ. Specifically, the goal was to create cartoon animations that mirrored data structures, program concepts and program events directly. Since DESMO-J was created specifically to model simulations of events that take place in the real world, the classes and objects created in it can be mapped almost directly to real world objects. In other words, the DESMO-J program objects are sufficiently concrete that they can be easily represented by cartoon depictions of their real-world counterparts.

The major conceptual entities in the program are the ships, the quay slots and the ship waiting queue. These needed to be depicted.

The activity in the simulation involves new ships entering the port at a range of time intervals, as specified by a random number generator. Program activity also includes ships joining the waiting queue until there is free space to dock, ships docking in the harbour for as long as it takes for them to be loaded, and ships leaving the port: this activity needed to be communicated visually also.

Figure 5 shows a picture taken from the DESMO-J website, relating to the port simulation. While this picture is contained in the tutorial to explain the concepts in the program, currently no visualisations exist for it.

As well as representing target program activity and concepts, we wanted to provide some statistical information about the simulation of the sort contained in the generated HTML reports, since these simulations are often used to model real world situations, in order to better understand them or make predictions about real world behaviour. These statistics include average waiting queue length, maximum queue length, current queue length, total number of incoming ships, average number of slots needed per ship, as well as other slot usage details.

4.3 Responsibilities of the program and mapping components

The primary task of the PMV program component is to capture calls made to methods which correspond to modelled events. A secondary task is to use this method call information to gather new statistical information. Given that InspectJ provides a non-intrusive way of monitoring target programs, these tasks are achieved without modifying the target program source.

The mapping component transforms the target program information in a manner that will be readable by the visualisation component and then sends it to the visualisation component via a socket connection.

4.4 Responsibilities of the visualisation component

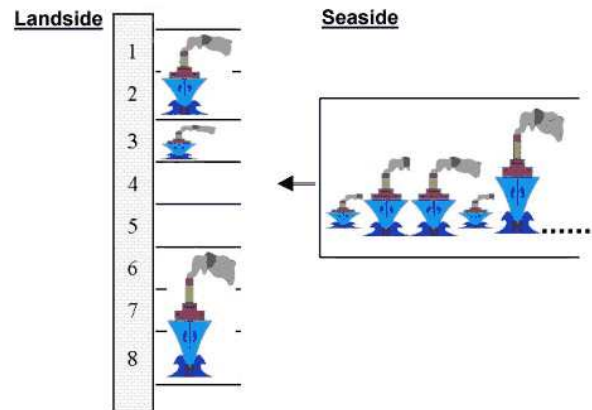

Figure 5: DESMO-J port picture

When the simulation program is launched, the Tk canvas displays an event-based view, which commences by showing a default background photographic image of an empty harbour.

4.4.1 Movement areas in the event-based view

The event-based view canvas is split up into regions within which specific event information is depicted. These regions consist of the harbour entry area, the waiting queue area, the quay-side area, and the harbour exit area. Figure 6 shows how the canvas is divided into these areas.

Every time the program creates a new ship, a small cartoon depiction of a ship "sails" into the harbour entry area. Different cartoon representations are used for the ship depending on how many slots it needs: a ship needing only 1 slot is represented as a tugboat, a 2 slot ship is represented as a bigger boat which is double the length of the tugboat and finally a 3 slot ship is represented as a cargo ship, 3 times the length of the tugboat.

When new ships arrive, they immediately join the end of the waiting queue, in the waiting queue area. This area is for ships that cannot acquire enough slots to dock: it corresponds to the internal queue in the simulation program and contains appropriate ship depictions, names and space requirements. The queue "grows" downwards. Since there is no limit to how many ships there might be waiting in the queue (this number is determined by the ship creation rate which can be altered in the simulation program), if the queue appears to be of a larger size than the default canvas size will display, the canvas size may be extended to display the "hidden" ships.

If there are enough free slots in the quay to accommodate the ship waiting at the front of the queue, the ship gets removed from the queue and placed in however many slots in the quay-side area that it needs. The rest of the ships in the queue get moved up one space. Ships stay in quay-side slots until they have finished loading.

Once loading has completed for a given ship, it is removed from the quay-slot area and redrawn in the harbour-exit area, at which point it "sails" away.

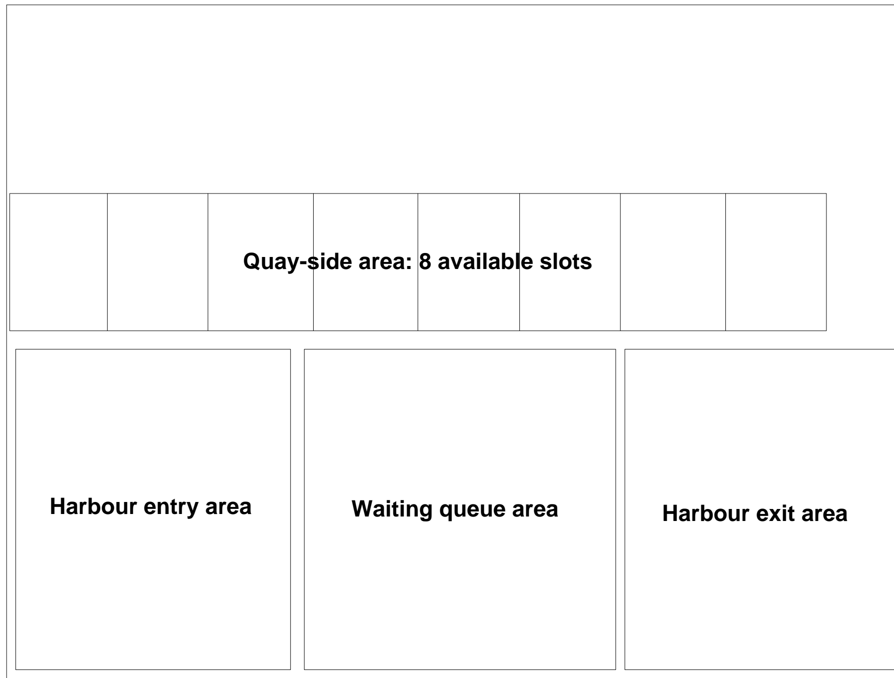

Figure 6: Event-view canvas area divison

Figure 7 is a screenshot from the visualisation corresponding to one simulation run: it shows 4 ships docked in the harbour, 3 ships waiting to be docked and a new ship that has just arrived.

4.4.2 Specifying movement in the event-based view

Smooth continuous animation results in gradual updates of visualisations. An advantage of such gradual updates is that people's visual systems can easily perceive and understand the changes that have taken place (Stasko 1998).

For this reason, ships entering and leaving the port are depicted using a recursive smooth animation method: they are redrawn using a frame system such that their movement seems to be continuous. This method takes a ship type, "before" coordinates and "after coordinates". It then creates the smooth animation by drawing and redrawing the ship, by calling itself, at points nearer and nearer to the end position. It stops calling itself when the current position and the end position have converged. Changing object coordinates and redrawing images is easy to do with the Tk canvas.

Defining movement in this manner has an advantage: since the method is fairly generic, it can be used to specify movement for any object from any start point to end point.

The view can be switched to the statistics view by clicking on a canvas button.

4.4.3 The statistics view

The statistics view gets constantly updated by every method notification sent through the pipe. The statistics view provides a numerical explanation of the experiment while it is in progress, much like the HTML reports the simulation provides, *after* the experiment has finished. Figure 8 shows the statistics view at the point in the simulation when 15 ships have entered the port.

5 Discussion

5.1 The aspect-oriented approach to visualisation

Visualisation is a cross-cutting concern, which means that events that are being monitored cannot be restricted to just one area of the target program. The aspect-oriented approach to program monitoring for visualisation addresses this issue, and allows for natural separation of cross-cutting concerns from component code. This separation not only aids maintenance of code clarity, it also prevents introduction of annotation-related bugs.

Another major advantage that the aspect-oriented approach provides for program monitoring is that it is non-intrusive: it is unnecessary to modify existing software and hardware, and only visualisation aspects need to be written.

5.2 Frameworks

The main problems encountered while developing this visualisation were to do with the fact that the DESMO-J package is a blackbox framework.

Two linked lists needed to be created in the InspectJ program aspects to store both ships currently in port and ships in the waiting queue: this is a case of data structure duplication, as these queues are already contained within the framework code. These duplicate data structures needed to be created because the framework hides the target program representations of these and therefore cannot be directly accessed by aspects.

Populating these duplicate data structures also proved to be somewhat problematic, again due to the target program being built from framework code:

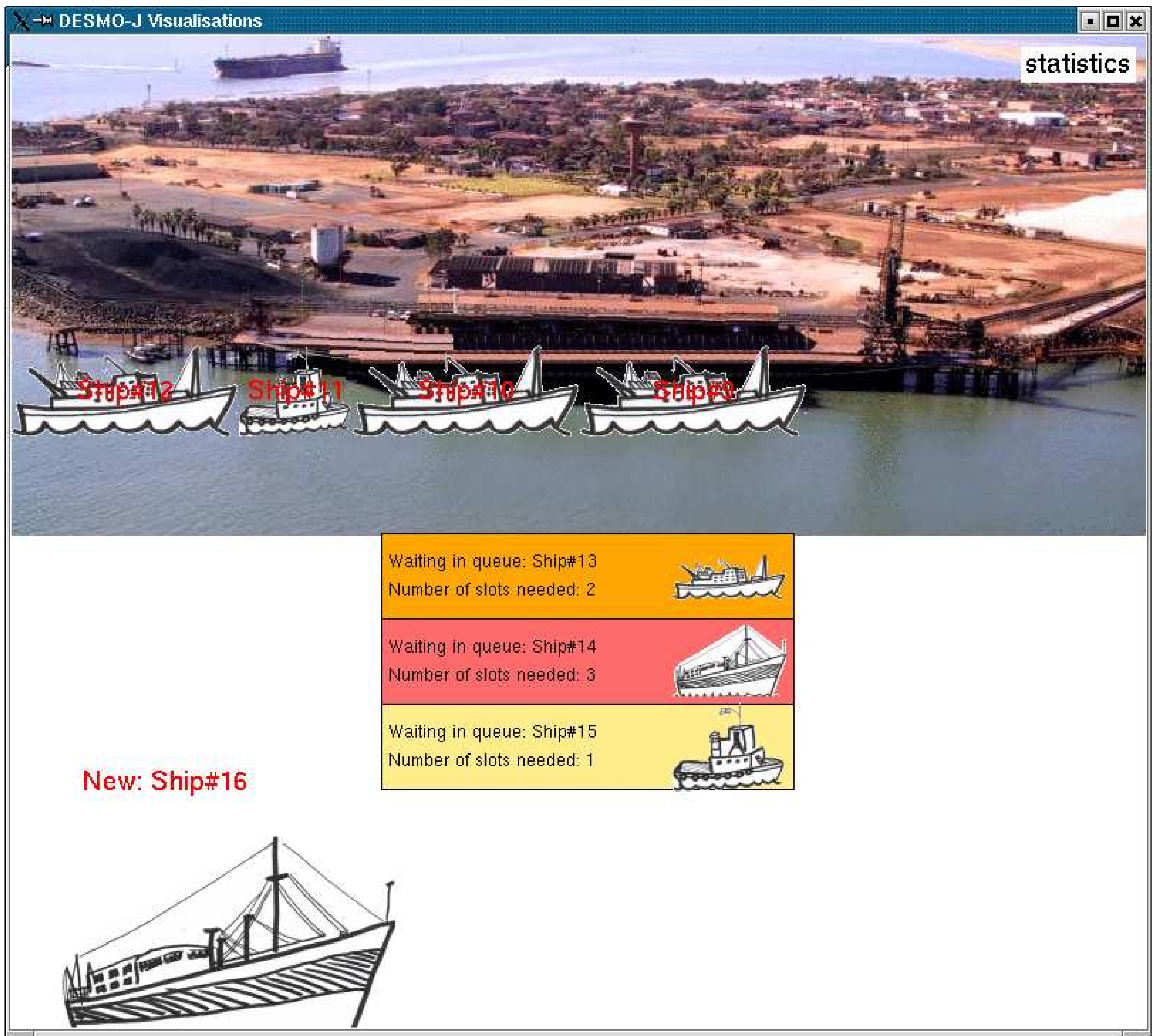

Figure 7: DESMO-J visualisation

the nature of frameworks means that target program events might be "hidden" in abstract classes, which makes it difficult to discern which events need to be captured, since many relevant ones may be invisible as far as developers are concerned. Examples of important events that needed to be captured, but could not directly be captured, included when ships were added to the waiting queue and when ships were released from the queue. We had to define the program component aspect advice in such a manner that they worked *around* the events that we were certain *did* occur, then using appropriate information we were able to populate the duplicate data structures.

Generally it is considered to be advantageous that classes of blackbox frameworks are like pluggable black boxes, with mysterious insides that somehow achieve the task at hand. This means that it is not possible to see code contained within abstract classes, which concrete program objects are often inherited from. In other words, concrete program objects depend on abstract framework class code which cannot be seen by developers. This is a problem when the visualisation is dependent on target program events because the events may be triggered from within the unseen code, and therefore developers of the visualisation are unaware of *which* events they should be monitoring at all.

While for the simulation example, the events were straightforward enough to approximate what the abstract classes were doing based on input and output parameters for methods, for a larger system this ambiguity of knowledge of internal events will almost certainly cause problems. Note that with whitebox frameworks, also known as glassbox frameworks, it *is* possible to see how class behaviour implemented, so the ambiguity problem is less relevant.

Visualising frameworks has advantages as well: if a generic visualisation is created for the framework, the visualisation, too, can be customised to suit specific applications. This could be achieved by defining default representations for typically used framework components, e.g. queues and bins. Then, to specialise the generic visualisation to the specific application, the developer of the visualisation need only define representations to model the application-specific concepts.

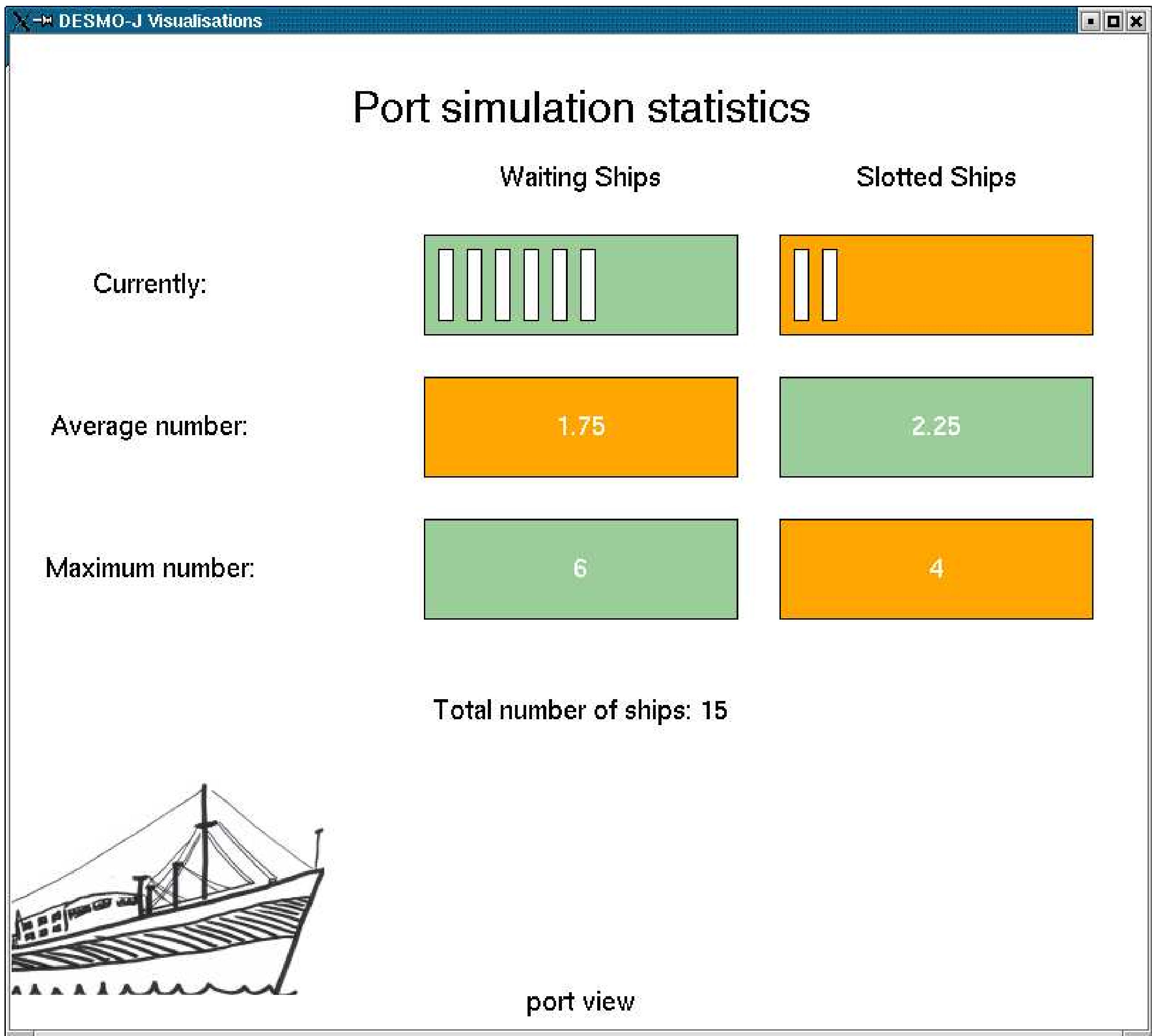

Figure 8: Port simulation related statistics

5.3 Visualisation technique

The cartoon animation approach to visualisation for the simulation, which is a lightweight approach, proved to be easy and fast to create. After drawing the cartoon representations of the ships and defining Tk canvas object movement procedures, little remained to be done. Yet its lightweight nature did not prevent it from being informative of program events: events in the target program were able to be clearly represented by the cartoon animations in a manner that was easy to understand without much prior explanation being needed.

The cartoon approach could be effective psychologically in making systems seem less complex: not only would the program object/cartoon image association provide a metaphor for the developers, since many people associate cartoons with children, they would be biased towards believing that the interaction they were watching was not overly complicated. It remains to be seen how helpful cartoon visualisations would be for complex systems however.

The main disadvantage to the cartoon visualisation approach is in coming up with images to associate to events, as for larger systems many would have to be defined. One way to get around this problem is to use existing clip-art, of the type freely available online. While clip-art might be a little less customised then desired, it reduces custom visualisation creation time even more significantly.

6 Conclusion

In this paper we presented a case study of the visualisation of a DESMO-J port simulation using InspectJ and cartoon visualisation. InspectJ is our visualisation framework, which is based on the PMV model and driven by AspectJ, an aspect-oriented language. The advantages of using an aspect-oriented approach to visualisation are that since visualisation is a cross-cutting concern, the nature of aspect-oriented programming caters well to the task of program monitoring for visualisation, since program events worthy of monitoring cannot be isolated to certain regions within the target program. Furthermore, additional code needed to create visualisations is separate from the target program: it is not necessary to alter the target program. This separation of concerns makes the task of visualisation development simple and fast.

DESMO-J is a simulation framework: we created a visualisation for one application of the framework, the

shipping port simulation. The visualisation, which made use of a Tk canvas, contained both an event-driven view and a statistics view. Advantages of using the Tk canvas are that it is easy to draw and redraw objects, as well as move them around.

The combination of using AspectJ for program monitoring and the Tk canvas for rendering the visualisations made creation of the visualisation reasonably fast and fairly straightforward.

We discussed some interesting issues that arose while trying to visualise framework applications, such as the difficulties encountered while visualising an application based on a blackbox framework. We also discussed cartoon animation, which is a lightweight approach to visualisation which seems particularly useful for representing activity in tangible application domains.

References

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

Johnson, R. E. (1997), 'Frameworks = (Components + Patterns)', *Communications of the ACM* **40**(10), 39–42.

Khaled, R. (2002), 'InspectJ: AspectJ for visualisation', BSc Honours Report, Victoria University of Wellington.

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'.

Neidhardt, O. (1999), 'DESMO-J Tutorial', `http://asi-www.informatik.uni-hamburg.de/personen/stud/1neidhar/tutoria%l/homepage.html`.

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

Stasko, J. (1998), *Software Visualisation*, MIT Press, chapter Smooth, Continuous Animation for Portraying Algorithms and Processes.