Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

# The Episode Framework

Exporting visualization tools to the web

**Marco Primi**

**supervised by**
Prof. Dr. Michele Lanza
Marco D'Ambros

# Abstract

Many tools exist nowadays to help software developers in the process of analyzing a software system. Software visualization applications represent an evergreen branch of software engineering. The common objective of the tools in this category is to help software architects gaining a better knowledge of the system under analysis with extensive the use of graphical representations.
Another characteristic is common to those tools: the difficulty to retrieve, install, configure and use such applications.
In this work we propose a framework that can be used for reworking existing tools to export them on the web, enhancing their accessibility.
The Episode framework, presented here, provides an environment to export existing software visualization tools to the web without the pain of rewriting them from scratch. During this conversion process we also introduce the concept of portability into the produced visualization, transforming it into an portable, interactive object, that can ”live on its own” and it’s easy to save, share and archive.

# Acknowledgments

Alla mia famiglia, per avermi sempre incoraggiato a proseguire gli studi.

Thanks to Marco and Michele for your precious time.

Thanks to my Sleeping Beauty

Thanks to the FSM, for creating electronics, computers as well as anything else...

# Contents

# Chapter 1

# Introduction

Software Visualization (SV) is an active research field in software engineering that uses typography, graphics, and animations to illustrate different aspects of software. SV systems can be utilized in program development, research, and teaching to help programmers and learners understand the structure, abstract and concrete execution as well as the evolution of software.

## 1.1 The problem

Today quite a lot of software visualization tools is available for various purposes. What all those programs share is that they are standalone tools, designed for a software engineer or developer sitting in front of his screen.
In most of the cases using one of such programs involves quite a lot of work, downloading, installing and configuring it, specific training may be needed, OS, libraries or middleware incompatibility may occur, etc..

Even if the panorama of software visualization programs is very rich, the number of applications that can be used via web or collaboratively is really small.
This is a major limit to the benefit we can get from those tools, for example a team of developers willing to go though a system analysis together, have no other solutions other than sitting in front of the same screen (which is not always possible).

Another issue that we want to address is that software visualization is not generally considered something *portable*. It is more the result of a session using some program, that disappears when the program quits.

This work is based on the assumption that all branches of software visualization, from large scale enterprise system analysis to algorithm animation for educational purposes, can experience great benefits from the advent of the web era.

## 1.2 Our approach

By introducing the concept of portability for visualization artifacts, it is our aim to give new life to existing software visualization programs, increasing their accessibility and utility by porting them to the web.
In this work we present Episode, a framework that can be used to create software visualization web-applications on top of existing programs transforming ”views” into interactive, animated, exchangeable objects.

## 1.3 Structure of this document

In Chapter 2, we briefly analyze some existing visualization tools, focusing on the approach they have towards their goals, then we explain the motivation for thisic work.
In Chapter 3 we describe the idea behind Episode, a framework that can export existing tools in a semi-automatic way while ensuring the portability of the visualization.
In Chapter 4 we validate this design by showing how the Episode Framework can enhance the value of confirmed tools such as the Evolution Radar[DLL06] and how it is feasible to create from scratch new graphic-intensive web application such as Ivory.
In Chapter 5 we discuss positive feedback of this experience as well as the limits of such approach, drawing our conclusions.
The appendix contains more details on some technologies involved in this project, such as Seaside and SVG.

# Chapter 2

# Software visualization

## 2.1 Introduction to Visualization

Visualization is in general the process of transforming information into a visual form, enabling the viewer to observe, browse, make sense, and understand the information. Visualization is not a new phenomenon, for thousands of years it has be useful to improve understanding of the data being presented, for example in maps, scientific drawings, and data plots. A common example often associated to visualization is the famous map of Napoleon's Russian campaign drawn by Minard (Figure 2.1). The data under analysis in this case is the size of Napoleon's army along the way to Moscow. This image clearly shows how sometimes a picture is worth thousand words.

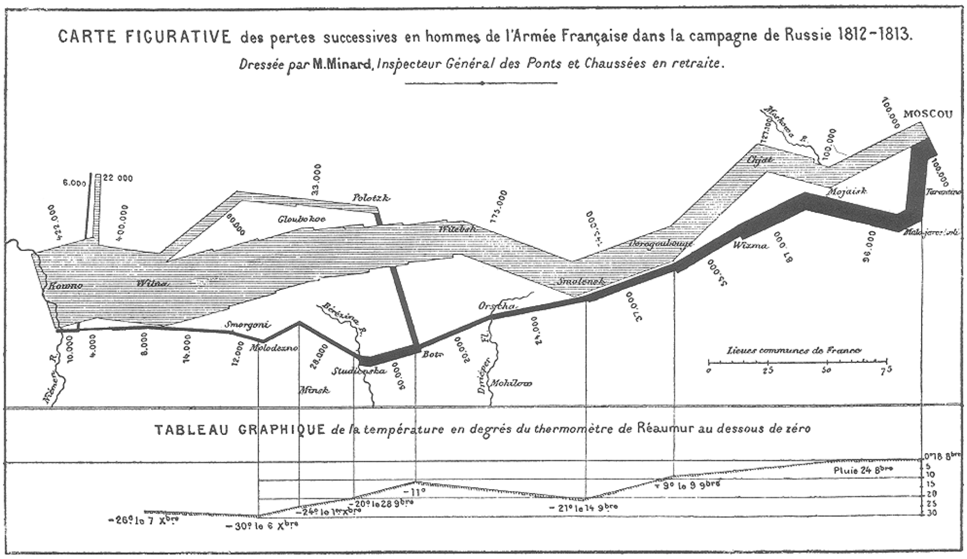
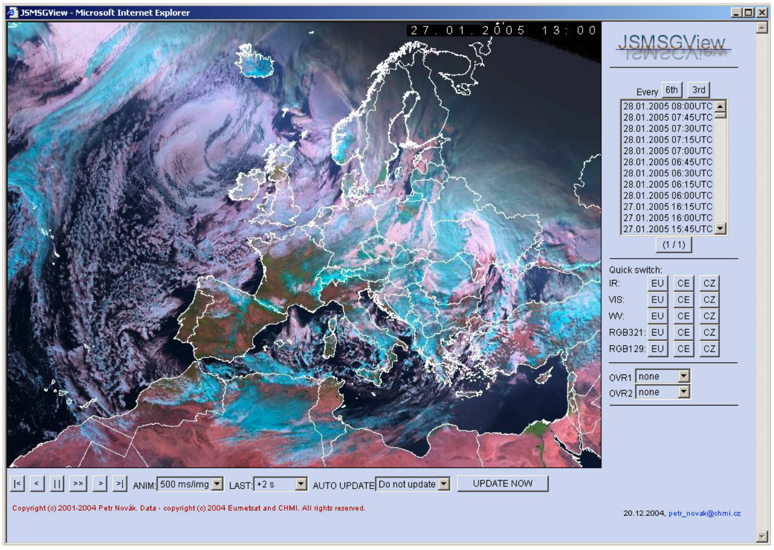


Figure 2.1: Famous map from Minard that shows the size of Napoleon's army shrinking along the way to Moscow

**Scientific visualization** In those days we are used to the fact that computer graphics, together with computer simulations, radically changed the way other disciplines are conceived.

Experiments that could take years can be simulated by computers in hours, huge quantities of data can be condensed together in a more understandable way by visualization software to aid reasoning, hypothesis building and cognition.
A clear example of that is meteorological data (see Figure 2.2) which combines the incoming data from thousands of various kinds of sensors to forecast the weather, a process that is impossible for a human without some visualization tool on support.


JSMSGView - Microsoft Internet Explorer

27.01.2005 13:00

28.01.2005 08:00UTC
28.01.2005 07:45UTC
28.01.2005 07:30UTC
28.01.2005 07:15UTC
28.01.2005 07:00UTC
28.01.2005 06:45UTC
28.01.2005 06:30UTC
28.01.2005 06:15UTC
28.01.2005 06:00UTC
27.01.2005 16:15UTC
27.01.2005 16:00UTC
27.01.2005 15:45UTC

OVR1 none
OVR2 none

ANIM: 500 ms/img LAST: +2 s AUTO UPDATE Do not update

Copyright (c) 2001-2004 Petr Novák. Data - copyright (c) 2004 Eumetsat and CHMI. All rights reserved.

20.12.2004, petr_novak@chmi.cz

Figure 2.2: A screenshot of JSMSGView by Petr Novák

**Software visualization** Software visualization implies the use of computer graphics and animation to help illustrate and present computer programs, processes, and algorithms. Software visualization systems can be used in teaching to help students understand how algorithms work, and they can be used in program development as a way to help programmers understand their code better.
Software visualization [Die02] is concerned with the static or animated 2-D or 3-D [MFM03a] visual representation of information about software systems based on their structure [MFM03b], size [LD03], history [KLS00, VRD04], or behavior [JSB97].
Visualization is inherently not a method for software quality assurance but can be used to manually discover anomalies similar to the process of visual data mining [Kei02, SDSD02]. The objectives of software visualizations are to support the understanding of software systems (i.e., the structure) and algorithms (e.g., by animating the behavior of sorting algorithms) as well as the analysis of software systems and their anomalies (e.g., by showing classes with high coupling).
Tool for software visualization are used to visualize source code and quality defects during software development and maintenance activities. Their target is the automatic discovery and visualization of quality defects in software systems and services.
Those tools, often designed as a plugin for an IDE (e.g., eclipse[1]), visualize the direct relationship between components of a software artifact, helping the programmer to navigate and build a better cognition of the entire system.

[1] Eclipse is an open-source itegrated development environment (IDE) written in Java *http://www.eclipse.org*

**Example: CodeCity** CodeCity[WL07a, WL07b] implements a visualization approach built on the city metaphor. An object-oriented software system is represented as an interactive city that can be explored in three dimensions. The buildings in the city (representing software artifacts) are positioned according to well defined rules, and thus facilitate the establishing of visual orientation to gain familiarity with the system. The goal is to give the viewer a sense of locality to ease system comprehension.

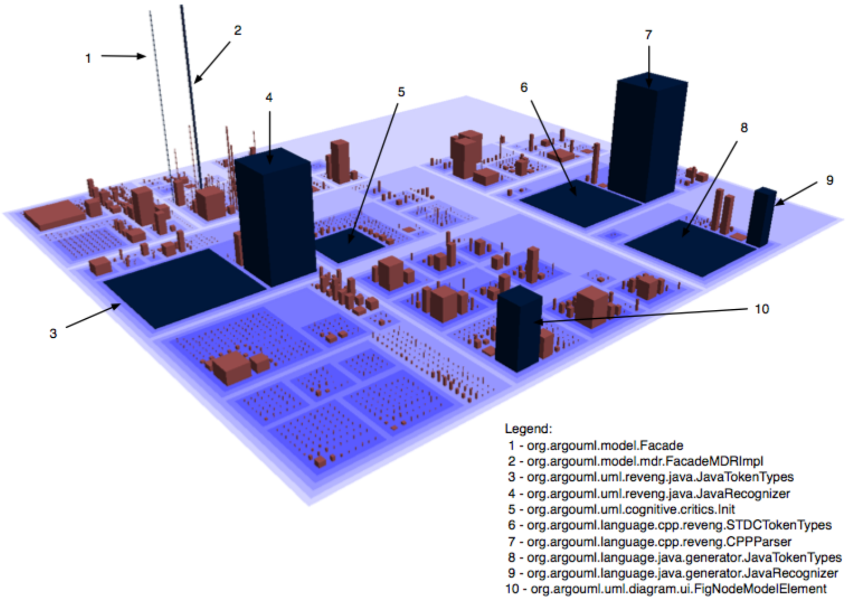

1
2
3
4
5
6
7
8
9
10
Legend:
1 - org.argouml.model.Facade
2 - org.argouml.model.mdr.FacadeMDRImpl
3 - org.argouml.uml.reveng.java.JavaTokenTypes
4 - org.argouml.uml.reveng.java.JavaRecognizer
5 - org.argouml.uml.cognitive.critics.Init
6 - org.argouml.language.cpp.reveng.STDCTokenTypes
7 - org.argouml.language.cpp.reveng.CPPParser
8 - org.argouml.language.java.generator.JavaTokenTypes
9 - org.argouml.language.java.generator.JavaRecognizer
10 - org.argouml.uml.diagram.ui.FigNodeModelElement

Figure 2.3: A picture of CodeCity representing ArgoUML, a Java tool for UML modeling

CodeCity is a tool for system analysis, not just a beautiful visualization application. In Figure 2.3 we see a screenshot of CodeCity in which some anomalies are graphically highlighted.

Classes are the cornerstone of the object-oriented paradigm, and together with the packages they reside in, the primary orientation point for developers. Therefore within CodeCity, classes are displayed as building and packages as city districts.

## 2.2 State of art

In the following section we go through some examples of validated software visualization tools publicly available, trying to understand what what they have in common.

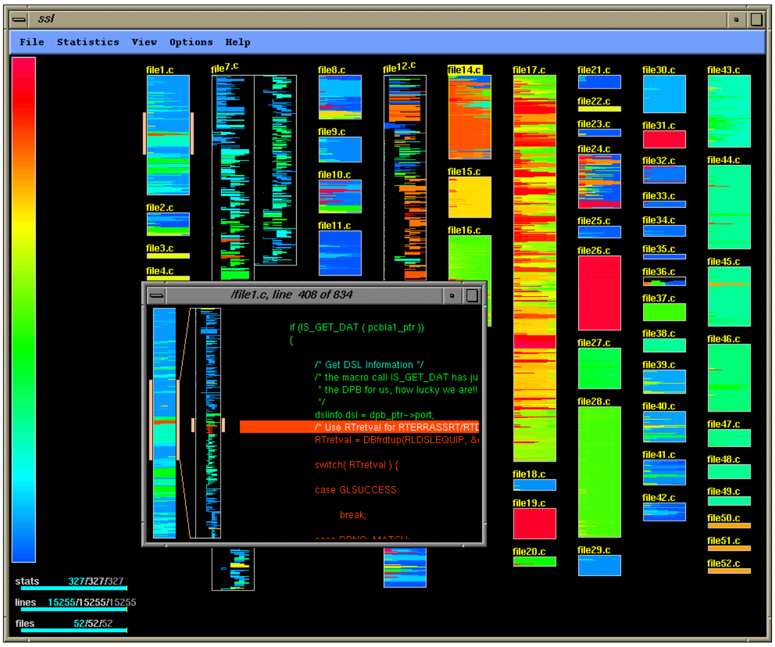
### 2.2.1 "Classical" visualization tools

What characterizes this group of tools is that they offer the minimal functionality based on the analysis on which they focus, for example a visualization of a particular attribute or small set of attributes.

The GUI generally contains a canvas in which we have the visualization and some controls (sliders, buttons) to interact, the canvas itself may be interactive too.

Those applications must be downloaded, maybe compiled, installed, configured, etc., a consistent difficulty for a casual user willing to try the programs, that in some case may also discourage an expert developer.

### Example: SeeSoft



```
if (IS_GET_DAT ( pcbla1_ptr ))
{

        /* Get DSL Information */
        /* the macro call IS_GET_DAT has ju
         * the DPB for us, how lucky we are!!
         */
        dslinfo.dsl = dpb_ptr->port;
        /* Use RTretval for RTERRASSRT/RTD
        RTretval = DBfrdtup(RLDSLEQUIP, &

        switch( RTretval ) {

        case GLSUCCESS:

                break;
```

Figure 2.4: A screenshot of SeeSoft

SeeSoft (see Figure 2.4) is one of the earliest tools that visualized program history from version control systems data [ESEE92]. It is a generic tool for visualizing statistics associated with lines in text files. It creates a visual representation out of a large amount of code for the purposes of code exploration and project management.

The key characteristic of Seesoft is its line-based visualization that maps each line of source code into a thin row on the screen, with files comprising the system arranged in columns across the screen. The colors can represents different values such as as age or developer who authored it.

SeeSoft is interactive, a user can select a subset of information to be displayed (for example, to color only the lines, across all files, that were created together). It is possible to brush: simply moving the mouse over an entity in the view to get a more detailed view.

### Example: CVSgrab

CVSgrab (see Figure 2.5) is a visual tool for exploring the evolution of industry size projects [VT06].

It is based on a dense pixel visualization combined with interactively built layouts and makes it possible to navigate and assess large code repositories. For example, we can get a comprehensive overview of the complete evolution of the VTK project (2700 files, 40 developers, over 11 years, about 100 versions for active files) in five screens, with quite little interaction.

CVSgrabs main strength comes when one does not know where (and why) to zoom in,

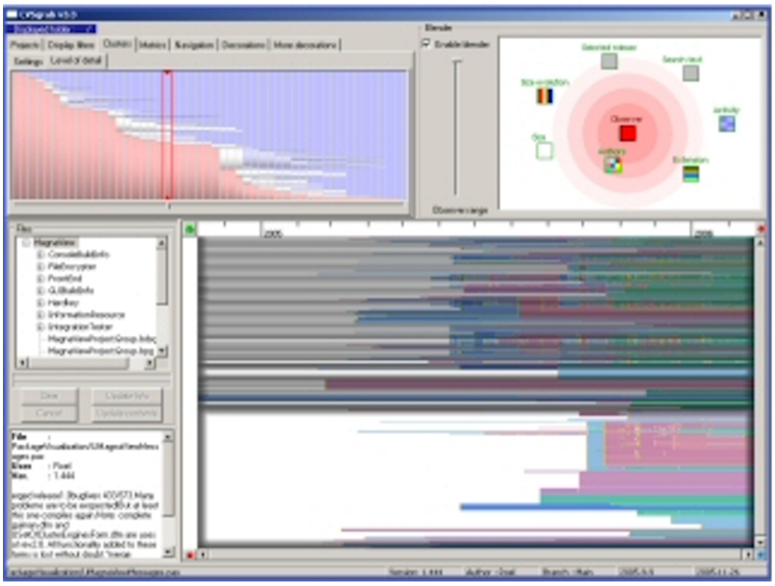


Figure 2.5: A screenshot of CVSgrab in action

given a large software project of many versions.
Secondly, the evolution-based similarity sorting and clustering proposed here can be effectively used to discover relations between files in a project that are not apparent, without needing to use more the complex, slower, language-specific parsing of the files contents.

**Example: Softwarenaut**

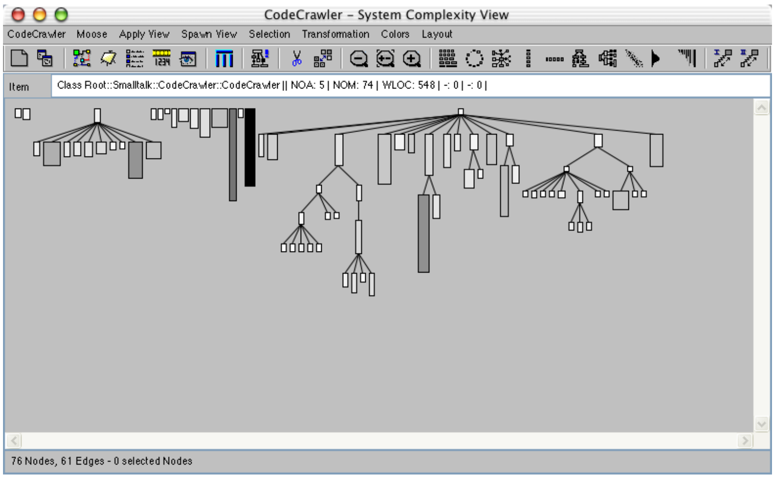
Softwarenaut[LL06] provides an interface to interactively explore the hierarchical clusters given by the classes that use similar terms. Low level code visualization (e.g., class attributes and methods) does not scale for industrial-size systems, for this reason hierarchical clustering is applied to the artifact.
Softwarenaut consists of an environment for the interactive, visual exploration of any hierarchical decomposition of a software system. For example on top of Hapax, a semantic analysis framework.
The user can interact with, and navigate the visualizations of the semantical clusters, aided by complementary lower level information about the properties and interconnections between the components of the clusters.

### 2.2.2 Software visualization frameworks

This software generally evolves out of a standalone tools like the one presented previously, the program grows beyond the views originally included. It becomes more like an environment in which visualization techniques can be ”plugged in”, transforming the tool in a complete, customizable toolkit.
This architecture has a lot of advantages, the development cost shrinks because we can focus on the visual technique instead of spending time developing a stable application from scratch, since common functionalities are already offered by the framework.
The kind of API offered may vary: when developing in such environment we have to

adapt to the functionalities offered and this can be limiting in some cases, especially for what matters interaction, the simplest example is that it would be difficult to implement a 3D navigation in a framework for 2D graphic. If no framework fits our desires we may be forced to write our own tool anyway.

### Example: CodeCrawler



Class Root::Smalltalk::CodeCrawler::CodeCrawler || NOA: 5 | NOM: 74 | WLOC: 548 | -: 0 | -: 0 |

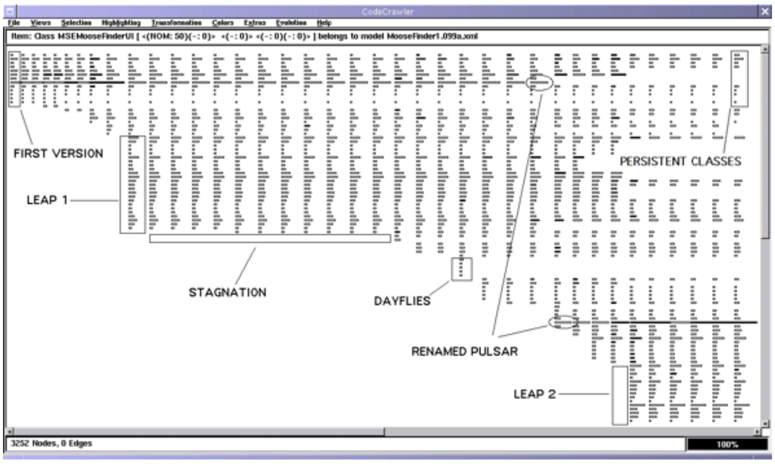
76 Nodes, 61 Edges - 0 selected Nodes

Figure 2.6: CodeCrawler displaying itself with a System Complexity View

CodeCrawler (see Figure 2.6) is a language independent reverse engineering tool which combines metrics and software visualization. CodeCrawler is based on Moose[DLT00], an implementation of the FAMIX metamodel which is described later.
It is mainly targeted at visualizing object oriented software, and in its newest implementation it has become a general information visualization tool. It has been validated in several industrial case studies. It strongly adheres to lightweight principles: CodeCrawler implements and visualizes polymetric views, lightweight visualizations of software enriched with semantic information such as software metrics and source code information.
Polymetric views [LD03] are based on two-dimensional nodes representing entities and edges representing relationships, we enrich these simple visualizations with up to 5 metrics on these node characteristics: width, height, color, X and Y coordinates.

### Example of CodeCrawler view: Evolution Matrix

Evolution Matrix is an approach to visualizing software evolution [Lan01]. It is a specialized view within the previously described CodeCrawler. Evolution Matrix uses program analysis to calculate various metrics based on a set of releases of the software.
The key characteristic of Evolution matrix is its presentation of the software as a 2D matrix with classes arranged on the Y-axis and time-ordered releases on the X-axis. Cells of the matrix encodes values of two of the classs metrics, one for each dimension. The tool displays the number of methods and number of instance variables in the class, and identifies several typical patterns of class evolution (see Figure 2.7 for an example). A user can change the metrics that are being visualized, as well as presentation parameters such as the mapping of colors to data values.



Item: Class MSEMooseFinderUI [ <(NOM: 50)(-: 0)> <(-: 0)> <(-: 0)(-: 0)> ] belongs to model MooseFinder1.099a.xml

3252 Nodes, 0 Edges

Figure 2.7: A screenshot of an Evolution Matrix view

### Example: SHriMP

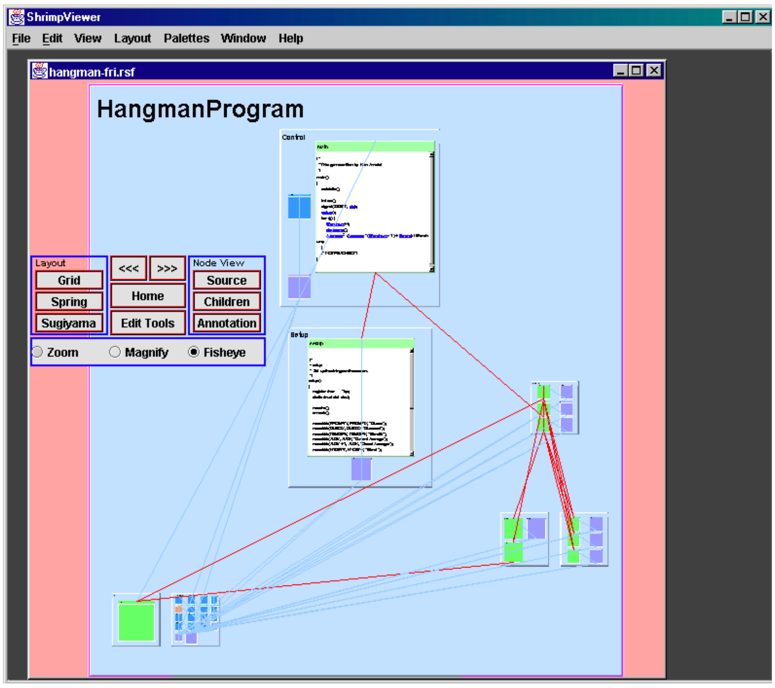


Figure 2.8: A screenshot of SHriMP

SHriMP, Simple Hierarchical Multi-Perspective [SM95, MADSM01] is a tool for visualizing and analyzing hierarchical data (see Figure 2.8).
Though mainly used for analyzing and visualizing large software systems it is not limited to this domain. The tool is graph-based and represents nodes as rectangles and edges as lines. SHriMP supports nesting, meaning that it is possible to embed an entire visualization inside a node as opposed to embedding a graph inside a node.
The authors have implemented several layouts. Especially the SpringLayout and the RadialLayout produce aesthetically pleasing results.

A interesting feature that SHriMP offers is semantic zooming [BH94]. This is particularly interesting when exploring data in an interactive way, since we can define several levels of detail for our data, exploiting application nesting.
The standalone version of SHriMP loads its data from a file with a graph-like structure (e.g., GXL, RSF, XML, XMI). SHriMP does not work on the model directly. This means that we have to export our model each time we perform changes on it. The need to export and import the data leads to a very slow iteration loop. Semantic zooming only works if we export all the necessary data. It also seems that SHriMP is not meant to be easily extended with custom figures. So, as a typical user, we are limited to the basic shapes that SHriMP offers.

### 2.2.3 Web-oriented tools

This is a relatively new category which still not very popular, there is no tool, known to the authors, with full support via web.
There are indeed tools which are making a step in this direction for example by exporting data in a format that can be rendered in a web browser, like SVG[2].
Traditionally a visualization exists only within the program that generates it, it is just an internal representation painted on the screen for the user.
Web-oriented tools instead forces a clear separation between the drawing and it's rendering, the visualization is created in a format (for example XML) in which it cannot be observed directly. The visualization becomes portable, we can use external programs to render it (e.g., a web browser), and it is easier to eventually reinterpret or apply new computation it in a later stage.
The Flash SWF[3] format, as well as Java Applets[4] can be used to develop widgets or small applications that runs inside web pages and in fact we find full-fledged online games based on Flash. Nonetheless Flash and Java are not suitable alternatives for us since they are not portable in the strict sense. Most of the web browser support them but a Flash object as well as an Applet is a compiled binary object, their wide support has not to be confused with portability.
Java Applets can be extremely useful for simple interactive applications such as algorithm animation, however they do not scale well and for sure are not suitable for a complex software visualization tool.

#### Example: Evolutionary Storyboard

The Evolution Storyboard (see Figure 2.9) provides dynamic views of the evolution of a softwares structure [BH06]. The visualization technique implemented is based on the principle that a representation that provide only static views does not capture the dynamic nature of software evolution.
An storyboard consists of a sequence of animated panels, which highlight the structural

[2] Scalar Vector Graphics is an XML language proposed by WC3 to represent 2D graphic objects. Refer to the appendix for more details on SVG

[3] *"`SWF` is a proprietary vector graphics file format produced by the Flash software from Adobe (formerly Macromedia). Intended to be small enough for publication on the web, SWF files can contain animations or applets of varying degrees of interactivity and function. The Flash program produces SWF files as a compressed and uneditable final product, whereas it uses the `.fla` format for its editable working files."* source:Wikipedia.org 25/07/07, *http://en.wikipedia.org/wiki/SWF*

[4] A Java applet is an applet delivered in the form of Java bytecode. Java applets can run in a Web browser with the help of a plugin an any system where a Java Virtual Machine (JVM) is installed.

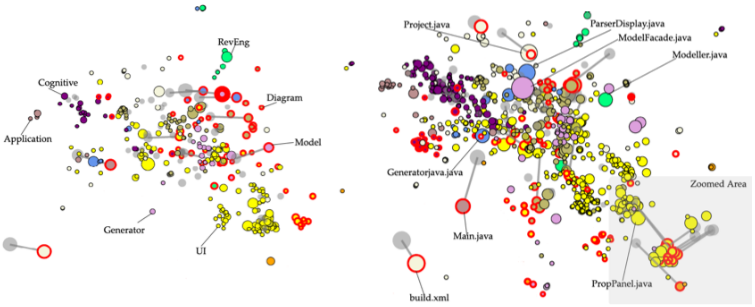


Figure 2.9: Start and end of an animation showing the evolution of ArgoUML

changes in the system, one panel for each considered time period.
It is possible to do basic interaction such as moving across panels, zooming, pausing the animation or changing the view.
The graph layout is currently provided in VRML format, in SVG format, in a standard text format, or directly drawn on the screen.

**Example: White Coats**

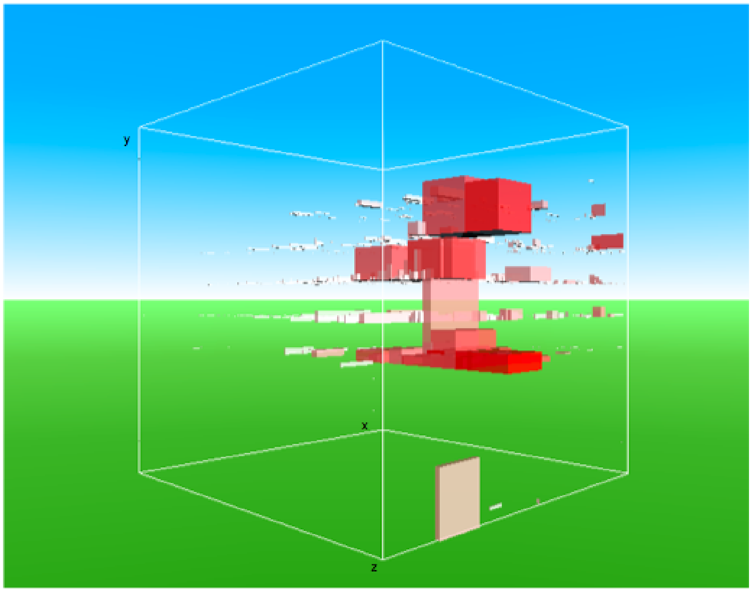


Figure 2.10: A White Coats view displaying file activity of the Azureus project

White Coats[ML05] is a service to create 3D visualization via web in VRML format[5]. It has been used for example to visualize the evolution of a software system by analyzing CVS[6] repository metadata.

[5] VRML (Virtual Reality Modeling Language, originally known as the Virtual Reality Markup Language) is a standard file format for representing 3D interactive vector graphics, designed particularly with the World Wide Web in mind.

[6] The Concurrent Versions System (CVS) is an open-source version control system invented and developed by Dick Grune in the 1980s. CVS keeps track of all work and all changes in a set of files, typically the imple-

The viewer can easily navigate, zoom, filter, fly through, interact with, inspect, etc. any of the visualized entities.
A screenshot of White Coat in action is shown in Figure 2.10

## 2.3 Motivation

By analyzing the variety of software visualization tools presented in previous section, we now have a more precise idea of what characterize software in this category. Some aspect such as the interaction are more or less homogeneous in our sample selection.
We can identify some common aspect among all of the tools:

- The tools are designed to run on a host machine, where one or more developers, in front of the same screen, can explore the system;
- Often the authors mention the fact that is their objective to help developers work more closely together, however none of the tools allows more than a single user at the same time;
- The interactivity required is generally not too complex, we have hovering effects, live update of panels or statistics based on clicks, controls such as sliders, contextual menus and so on;
- Graphical representations are usually composed by a number of simple geometrical objects;
- We can summarize architectures of the tools in a common way: generally there is a stage of data acquisition, a stage of construction of the system model and the final presentation to the user, who can then customize or refine the view with interaction;
- The final user view is not considered portable, it is the result of the activity of the user with the tool. It exists only within the program, and disappears when the program quits.
- Visualizations are generally static, meaning that the tool allows interaction and changes on the view but the items are not animated or moving around;

### 2.3.1 Design tradeoffs

Software visualization tools are a wide variety and anyone of them takes a different approach to the problem, presenting a unique visualization that tries to help developers get more insight of the system under analysis.
However from a structural point of view, they all have a very similar approach. This is natural and means that the respective authors made similar design choices.
During the development of a new software visualization tool it is important to consider the tradeoffs described below:

mentation of a software project, and allows several (potentially widely-separated) developers to collaborate.

### Interaction vs. portability

Making views portable can have a lot of benefits, it may be useful to share with others some aspect of the system, since most of the tools are designed for this purpose.
However it becomes evident that this portability comes with a major drawback, if we chose a well supported format (for example jpg) we have to renounce to something else e.g., interaction in this case. We may implement a tool designed to take specific detailed views and export them as bitmaps, but at this point is more convenient to develop an interactive tool and take a picture of the screen when needed...
Instead if we can create a new portable abstraction for software visualization views, it becomes easier to share results, for example with a online database of case studies that can be used to benchmark a new visualization technique.

### Web vs. standalone

The availability of a software is of course a benefit. However, since we are talking about graphic-intensive applications, the alternatives are limited, Java applets or Flash basically, recently also SVG became more supported.
Given that the tool is not web-oriented we can pick a standard graphical library (such as Swing[7] or GTK+ [8]), the limit of this approach is producing a tool that needs to be installed/configured and used on a single machine. Exporting a tool to the web will completely take out those difficulties.
Especially for software engineering tools the number of web-aware applications is really small, while it could be useful for developers, to be able access to the same unique instance of a visualization application, running server-side.

### Collaborative vs. single user

One of the goals that many of the tools have on common is their effort to make developers work together better, by making them aware of each other with a global system view.
Collaborative visualization software is not a new idea [FBDJK07], and is already reality in different fields such as medical visualization, simulations, etc..
Nonetheless none of the software visualization applications available nowadays offers support for multiple users. People working in the same project or analyzing the same system have to send each others screen captures if they want to share a particular view.
A web software may facilitates the development of collaborative application since we have a single server-side application and many clients.
This unique instance of the application running communicates on top of HTTP using a web server. This saves the effort of developing a complex peer-to-peer protocol to coordinate distributed user actions.
With a web server implementing some features becomes trivial, for example being able to save an analysis session to resume it later or to share it within a group of developers.

[7] Swing is a UI library for Java, see *http://java.sun.com/j2se/1.5.0/docs/guide/swing/* for more details

[8] GTK+ (the GIMP toolkit) one of the most popular widget toolkits for creating graphical user interfaces for the X Window System, *http://www.gtk.org/*

### 2.3.2 Goals of this project

It is our objective to contribute to the discipline of software visualization by filling the gaps presented in the previous part. We think that the recent growth of the technologies offered via web can have positive benefits on the field of software visualization.
Our aim is to be able to treat visualizations as standalone portable objects that can be exchanged and published on internet without losing their interactivity.
Another important objective is that we do not want to leave a distinction between web and non-web tools, we want to be able to transform any existing application into a web application in a semi-automatic way and ideally with no modification of the original program.
For those reasons we pose ourselves the following goals:

**1.** Design an architecture suitable for semi-automatic conversion or of existing software visualization tools providing an alternative user interface via web.

**2.** Create a framework that implements this architecture, with the following abilities:

- Export existing software visualization applications to the web;
- Provide an environment to develop from scratch graphic-intensive web applications;
- Allow user basic and common user interaction capabilities;
- Ensure the portability of the visualization;

**3.** Validate the framework and the architecture, by porting an existing Smalltalk tools, the EvolutionRadar

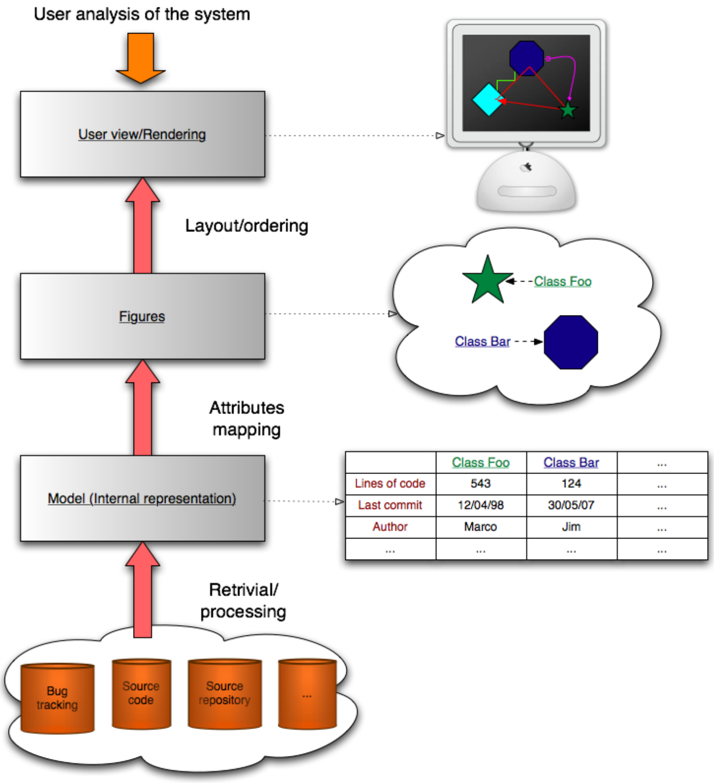
# Chapter 3

# The Episode framework

## 3.1 Architecture for web-based software visualization

### 3.1.1 Structure of a software visualization tool

Software visualization tools are very different one from the other but at the same they have numerous structural similarities. An abstract architecture for software visualization programs is presented in Figure 3.1 and discussed in the following section.



| | Class Foo | Class Bar | ... |
|---|---|---|---|
| Lines of code | 543 | 124 | ... |
| Last commit | 12/04/98 | 30/05/07 | ... |
| Author | Marco | Jim | ... |
| ... | ... | ... | ... |

Figure 3.1: The steps of a software visualization tool processing a software artifact to create a visualization

Depending on the analysis it carries on a SV application will need to import raw data about software the system, such as source code, repository metadata, bug tracking reports, etc.. In this stage, the program processes data in the original format and builds an internal representation of the system.

**Example** A tool that builds a hierarchical view of the classes of a project, is interesting in building an internal representation of the hierarchy, by linking classes to superclasses until the tree is complete.
As opposite, a tool that generates graphical statistics about the contribute on a project of different developers, will query the repository and build arrays of values, for example the lines of code to assign to each developer.

Once the data acquisition is finished and the internal representation is complete, the tool has full knowledge of the system under analysis: we call that representation a *model*. Which attributes of the model are interested in a particular situation depends on the kind of analysis.
The next phase consists of associating figures to objects in the system as their representation. A *view* which is the implementation of a visualization technique, provides a set of mapping between the object attributes and the shape attributes. Once finished, we have the representation of the model.

**Example (cont.)** The class hierarchy visualization tool can transform each class in a tridimensional cube, whose size depends upon the lines of code of the original source file. The developers statistics tool instead could focus more on mapping big squares to programmers who contributed more, and hottest colors to who contributed more recently.

The last stage of this process is presenting data to the user, possibly in the simplest and more intuitive way, this is a crucial task. The operations involved may include reordering, filtering, scaling figures, the criteria for that stage are likely to depend upon the view and/or on the layout.
Once the view is composed, it can be presented to the user, this usually happens though some graphic library that may depend upon platform and programming language.
The interface generally allows interaction. The user can navigate through the system, refine the visualization by changing various parameters or choosing another view or layout, inspecting the entities, (e.g., reading the source code, or commit-related informations, etc.).

**Example (cont.)** The class hierarchy visualization tool composes the class tree with alignment based on inheritance depth, when the user clicks one of the cubes, the visualization fades out the upper part of the hierarchy focusing on the subclasses of the object clicked.
In the developers statistics visualization instead, we may have a view where developers are represented with squares in proportion to the code they produced. Another view may consider each single file of the project and color it with a blend of colors of its developers, mixed in proportion to the activity.

### 3.1.2 Model-Figure relation

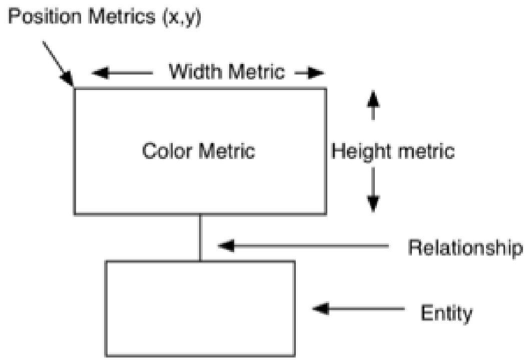


Figure 3.2: The principle of a polymetric view

Further inspection of this structure, reveals another major similarity between the tools. The model under analysis exists within the framework and so does the user visualization. However since the view needs to be reworked later, we can assume that the program stores the model and its entities separated from the view. The model will be useful later to calculate new views while the representation of the object is just a drawing on the screen that will disappear later.
The coupling between an Entity and the figure temporarily representing it may be high but a separation between the two exists. This mechanism can be exploited since it allows to maintain the original model while associating a new view, a web view in our case, to the entity.
For an example refer to Figure 3.2, the schema, taken from *CodeCrawler - A Lightweight Software Visualization Tool*[Lan03] explains the relation between an entity and its representation in a polymetric view, where the attributes of the figure such as positions, dimensions and color are mapped to attributes of the model. It is likely that those graphical properties are recalculated specifically for each view while the entity remains unchanged.

## 3.2 Framework architecture

### 3.2.1 What is episode

Episode is a framework implemented to validate our idea, it aims to introduce portability of visualizations on top of existing tools.
Episode is written within Seaside[1], a Smalltalk[2] toolkit for creating web applications. The goal of Episode is to provide an environment for designing web interfaces on top of an existing non-web tools, as well as creating new graphic-intensive web application from scratch.

[1] Seaside is a framework for developing sophisticated web applications in Smalltalk, refer to Section A.1 in the Appendix for more details

[2] Smalltalk is an object-oriented, dynamically typed, reflective programming language. It dates back to 1969. *www.smalltalk.org/*

This works tries to help software engineers who are going to develop a software visualization application. With Episode it is possible to create software visualization tools via web with the same effort required by any other graphical library, also the level of interaction and the complexity of the figures generated is comparable.

The current Episode implementation is written in Smalltalk on top of Seaside, it is suitable to export for Smalltalk tools. The same concept can apply to other programming languages and web tookits.

### 3.2.2 Main components

Since we want to export existing applications without possibly changing the original code, we need to offer an API similar as much as possible to the classic API of a GUI package. It follows that we need to have counterparts to objects contained in such libraries. Some of those are present present in Seaside as part of the HTML specifications.
The classes added by Episode are oriented instead to graphical objects that cannot be recreated with just HTML.
We describe some of those components in more details.

#### Canvas

The canvas is the outermost container for our graphical objects. We can choose wether to have a single central canvas or many small canvases around the page, depending on the application.
This class tries to be as friendly as possible to the developer, providing a set of general purpose functionalities. For example it automatically keeps elements organized in three separated groups: background, contents and GUI. This way when we want to clean the GUI elements (such as menus) we only need to flush the appropriate group.
Another example is the canvas area navigation, which is built in the framework. By triggering the appropriate action the canvas reveals four directional arrows used to move around the drawing and it is possible to zoom in and out, reset the view etc..
The canvas can be saved at any time as a standalone SVG object, this it may or not preserve its interactivity since just the scripting included in the canvas itself is preserved. Pusing all the interactions in the canvas requires requires a grater effort (since we can't use the Seaside components) but it is possible.
The canvas, just like any other component can be sub-classed to override and/or extend its behavior.

#### Shapes

The SVG specification contains a small set of geometrical objects, consisting of circle, rectangle, ellipse and line. In Episode we have transpose those figures in order to treat all of them as 'shapes', following the principle of polymorphism.
For example, in the SVG specification, the position of a square is respect to its upper-left corner while for a circle is the position of its center. In episode it is possible specify the positions of all of the shapes with both of the methods (center or top-left corner), internally we calculate the proper attributes for the specific shape (e.g., based on width and height).

#### Texts

Texts are a particular kind of shapes that represents strings of characters. Texts programming interface tries to be as close as possible to shapes.
The SVG specifications does not contain rules for text manipulation (e.g., inserting line breaks when the text goes beyond a certain limit). With the functionality offered by Episode it becomes trivial deal with this kind of issues.

#### Composite objects

So far we discussed components related to a specific figure. As a framework, in Episode is possible to create new components, their view can be an aggregate of many simpler figures.
An simple example of a composite present in Episode is the *button*. The view of a button is composed by a rectangle, a text and a CSS[3] property (which is an object too). When creating a button we can decide some layout options for example wether is the button size that needs to fit the text or the opposite. The CSS provides a highlighting effect when the user cursor hovers the button area. The button is generally assigned an action to execute when clicked (we refer to this action as a *callback*).
It is also possible to create components that embed their own canvas, just like small widgets that can be placed in pages.

#### Special purpose/Utilities

The components offered by Episode do not consist of only shapes. Some helper objects are present, for example a *colors converter* to traduce RGB format used in most of the graphic toolkits into HTML hexadecimal colors. Another example is the *canvas view helper* used to move the observer point of view around the canvas and zoom.

### 3.2.3 A web-GUI library

The abstract structure for software visualization tools presented in the previous section is obviously an ideal model. In reality the distinction between the layers of processing of the architecture are much more fuzzy difficult to spot.
What is important for us at this point is the uppermost layer, the final visualization to the user and the interaction.
In fact it is difficult to find a tool that implement it's own graphical drawing functions. It makes more sense, in most of the cases, to pick a widely-used, widely-tested, GUI package (such as Swing for Java, HotDraw[4] for Smalltalk and etc.). This is an advantage for us because if the library is external to the application, the coupling between the two is lower. It becomes easier for us to recreate the interface without modifying the rest of the program.

The APIs provided by most of the GUI libraries are very similar, we generally find frames, containers, tables, buttons and so on, we can assign values and actions to be executed to them and probably customize their look.

[3] Cascading Style Sheets (CSS) is a simple mechanism for adding style (e.g. fonts, colors, spacing) to Web documents. *http://www.w3.org/Style/CSS/*

[4] HotDraw is a two-dimensional graphics framework for structured drawing editors that is written in VisualWorks Smalltalk, *http://st-www.cs.uiuc.edu/users/brant/HotDraw/*

As it turns out, with recent improvements of web browsers both in terms of new formats support and dynamic web technologies, it is finally possible to design a GUI library with the special ability of creating web interfaces.
Episode contains functionality that makes it suitable as a choice for a user interface. Our UI are created for for web applications, requiring no download, installation, or configuration for the end user. This is not a totally new concept, for example the OpenLaszlo[5] project tackles the same goal, allowing to create windows, buttons and canvases inside web pages. The drawback of Laszlo is that this is obtained by decomposing the view into a number of small compiled Flash objects, with limits portability and dynamicity of the application.
Our goal is not *just* to create a web UI library with fancy effects, so the approach proposed by Laszlo (as well as other technologies such as Java Applets for example) is not feasible for creating a general purpose framework that also produces portable visualizations.

The Episode framework provides functionality to create full-fledged graphic-intensive user interfaces. Episode is built within the Seaside web toolkit so it *adds* functionalities to the set of standard HTML objects offered by Seaside (such as select list, checkboxes, radio buttons, popup windows etc.). Those objects alone are sufficient to create forms but cannot really be compared to a real graphical user interface.

For that reason Episode generates SVG inlined directly as part of the page. This produces results really similar to what we obtain with Flash, except that SWF Flash objects are compiled, while SVG[6] is XML, just text. In Figure 3.3 we see an example of SVG object rendered in a web browser.

Interaction is a critical factor for a user interface, Episode can create UIs rendered through a browser with very similar effects and API to other GUI package.
Any object can be assigned an action, for example we want to open a contextual menu for the specified object when it is clicked. In this case what we do is assign to the object itself a *callback*, the corresponding action is to show the menu. The *menu* is one of the objects present in the Episode classes, it consists of different simpler SVG shapes combined, we see an example menu in Figure 3.4. With the help of scripting we can obtain also other kind of interaction events such as mouse hovering, dragging, dropping etc., functions presents in most of GUI packages.

### 3.2.4 An environment to export existing applications

Offering an API to create graphical web application may be considered a good result by itself, however it is not our main objective, what we seek is to being able to create web interfaces on top of existing tools, without the need to rewrite them from scratch.
Episode provides an environment to export the existing user interface of a tool with relatively small effort by the developer. We can quickly create a partial web porting of the tool, ideal for example to create an online demo, with only part of the functionality to give an idea of the application.

[5] OpenLaszlo is an open source platform for creating zero-install web applications with the user interface capabilities of desktop client software. OpenLaszlo programs are written in XML and JavaScript and transparently compiled to Flash and, with OpenLaszlo 4, DHTML. The OpenLaszlo APIs provide animation, layout, data binding, server communication, and declarative UI. *http://www.openlaszlo.org/*

[6] Refer to the appendix for more details on SVG

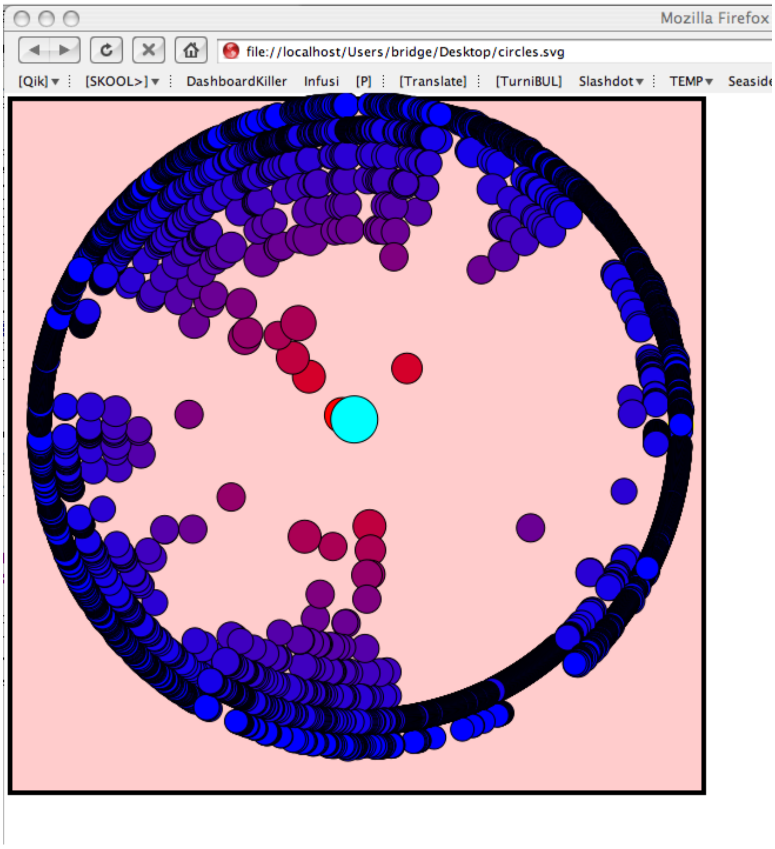


Figure 3.3: An SVG object rendered in a web browser, this is one of the first experiments with SVG

To obtain that, all we need is to to create a mapping between the tools internal figure objects and the Episode figures, specifying the relations between attributes that we want to maintain. An example of this concept is shown in Figure 3.5 where with a simple function we grab the current screen of the EvolutionRadar application and we convert each figures, maintaining their position, size and color.

It is possible to extend the framework, for example by implementing our own shapes and GUI elements, by subclassing and combining other objects, and to integrate the graphical canvas with traditional html controls available in Seaside.
The tool interface can be gradually pushed toward the web recreating UI components and assigning actions to figures and so on.
We discuss about this process in Chapter 4 where we present Episode in action. The schema in Figure 3.6 outlines how Episode's architecture fits the purpose of recreating a web interface by converting an existing one. Our work did not focus on full automatic conversion of user interfaces since it goes beyond the scope of this project, instead we implement in Episode functionalities to make this process straightforward as much as possible for a (human) developer.

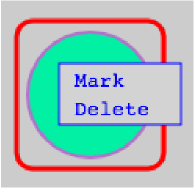


Figure 3.4: A simple Episode contextual menu that shows up when clicking the circle

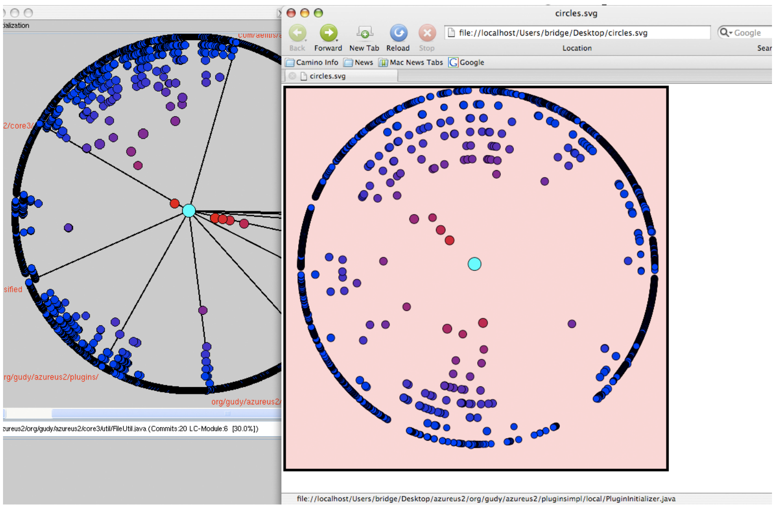

ialization
com/aelitis/
2/core3/
sified
rg/gudy/azureus2/plugins/
org/gudy/azureus2/
zureus2/org/gudy/azureus2/core3/util/FileUtil.java (Commits:20 LC-Module:6 [30.0%])
circles.svg
Back
Forward
New Tab
Reload
Stop
file://localhost/Users/bridge/Desktop/circles.svg
Location
Google
Sear
Camino Info
News
Mac News Tabs
Google
circles.svg
file://localhost/Users/bridge/Desktop/azureus2/org/gudy/azureus2/pluginsimpl/local/PluginInitializer.java

Figure 3.5: A conversion from a EvolutionRadar view to standalone SVG object

### 3.2.5 A portable visualization representation

As discussed before, we don't want the visualization produced by a tool to live only within the scope of the program itself. We need to force an intermediate representation in a more portable format. This schema is visible in Figure B.1

The SVG graphics generated by episode is *inlined* directly in the web page. A SVG object by itself can carry much more information than just figures. It supports for example embedded CSS, scripting (Javascript or ECMA) and animations.

This opens a whole new range of possibilities. Embedding interaction directly in the object allows us to produce self contained visualizations that can be saved, redistributed or included in other web pages.

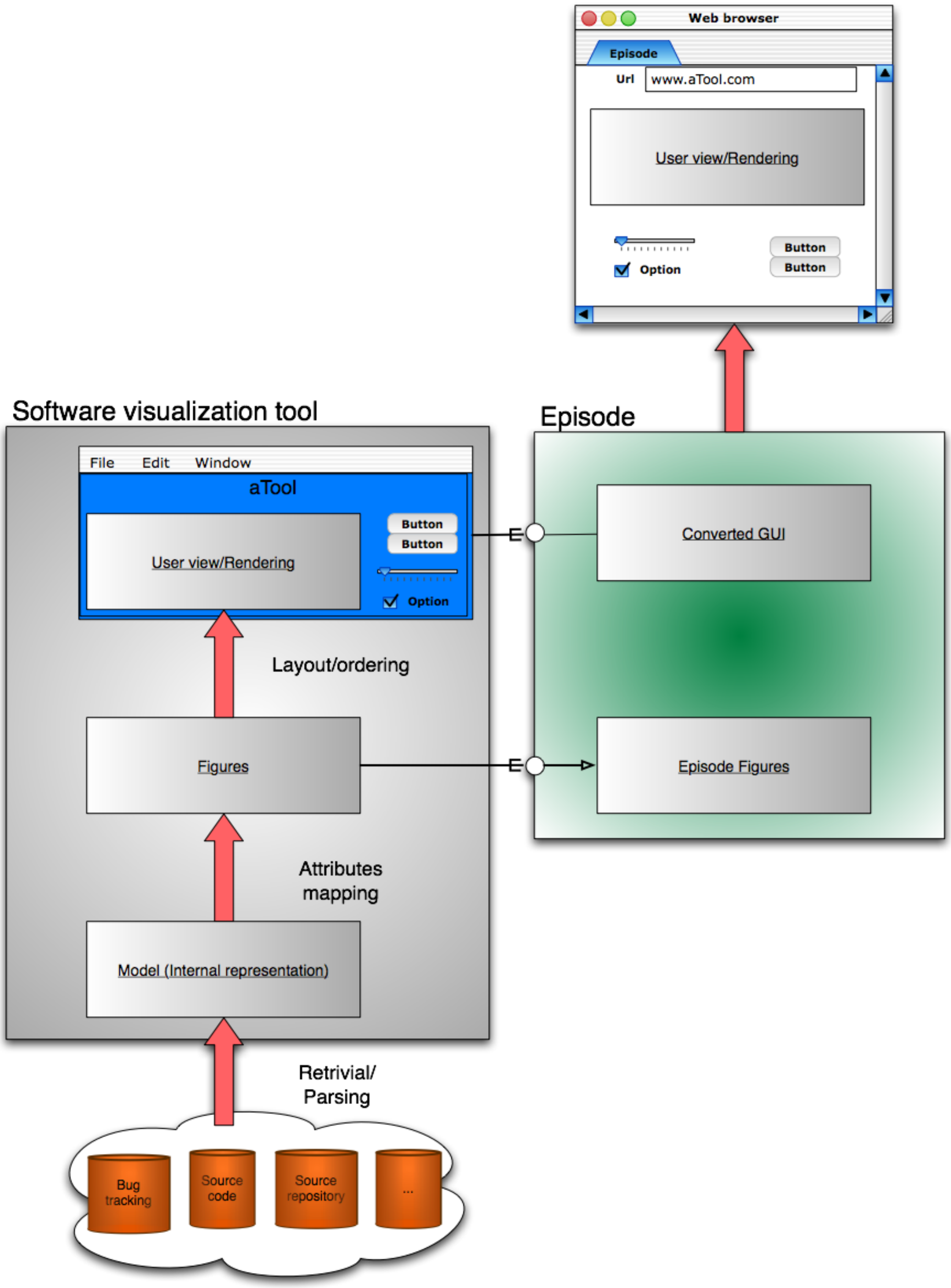


Figure 3.6: A schema that summarizes mechanism of web-converting existing tools with Episode

# Chapter 4

# Episode in action

In this chapter we validate the idea behind the Episode framework by exporting an existing software visualization tool written in Smalltalk, the EvolutionRadar.
In addition to that we present Ivory, an application that produces interactive visualizations via web based on scientific publications data.

## 4.1 Web-porting a tool: the EvolutionRadar

### What is the EvolutionRadar

In software evolution research logical coupling has extensively been used to recover the hidden dependencies between source code artifacts. They would otherwise go lost because of the file-based nature of current versioning systems.
Previous research has dealt with low-level couplings between files, leading to an explosion of data to be analyzed, or has abstracted the logical couplings to module level, leading to a loss of detailed information. The EvolutionRadar [DLL06, DL06] proposes a visualization-based approach which integrates both file-level and module-level logical coupling information, extracting data from source repositories.
This not only facilitates an in-depth analysis of the logical couplings at all granularity levels, it also leads to a precise characterization of the system modules in terms of their logical coupling dependencies.

#### Example usage session

To use the EvolutionRadar we need to have access to a source repository that hosts it and a working VisualWorks[1] image of Cincom Smalltalk.
We start by creating a model in this example we focus on Azureus[2] a Java Peer-to-peer client for BitTorrent. After the download, the graphical user interface appears and we start the model creation calling an sample action present in EvolutionRadar.

```
azureus := ChroniaProjectBuilder new createAzureusModel
```

[1] *http://www.cincomsmalltalk.com/*

[2] http://azureus.sourceforge.net/

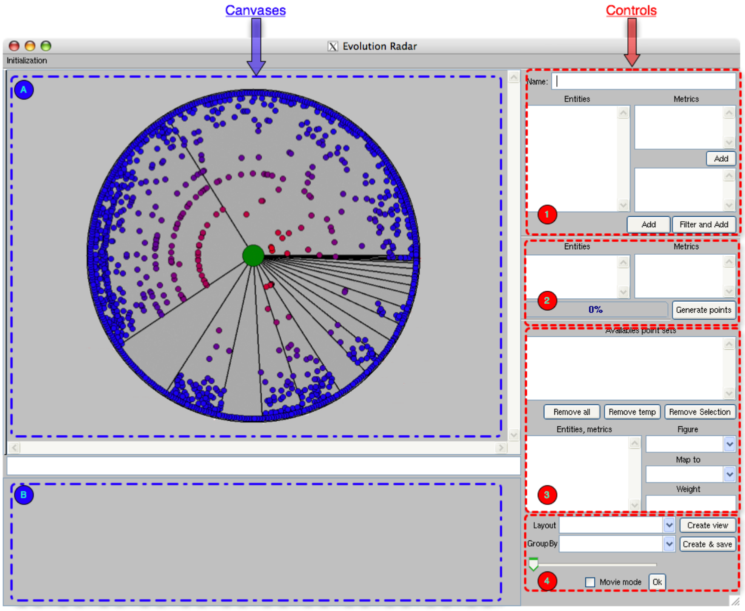


Figure 4.1: The user interface of EvolutionRadar after a fresh install

```
azureus gpkName: 'AzureusModel'
```

The program starts querying the Azureus source repository getting information considered relevant regarding the past evolution of the system.
After that we can import the model from the UI to begin the visualization creation.
The following steps are accomplished by acting on the corresponding controls marked in Figure 4.1.

**1** We pick a name for a new visualization and we select the main entities to visualize, e.g., source files. Each entity may have different metrics, an example is the number of times a two source code files were committed together. The EvolutionRadar uses this kind of metadata to reconstruct the logical coupling between entities.

**2** Once we have a set of entities to analyze with metrics specified, we generate the pointset. EvolutionRadar will extract information from data previously downloaded from the repository.
We can repeat steps 1 and 2 to generate more pointsets.

**3** In the top panel of the corresponding area, we see the list of pointsets created previously, selecting one of those will show a tree view of the entity with the metrics analyzed as leaf. On the right we select the figure that will represent the entity (e.g., a circle). We also select how to represent visually the attributes inspected (for example we map logical coupling with the distance from the center of the radar).

**4** We can finally select a layout (e.g., the evolution radar layout) and generate the view, that will appear on the main canvas. With the slider we can go trough the history of the system and get an insight of its evolution.

**Canvas interaction** The entities are now visible in the main canvas, it is possible to select a subset of them to spawn a new radar visualization.

### 4.1.1 Quick conversion

We begin our demonstration with a small conversion that can be useful to generate a portable snapshot of the current view.
By digging a little bit in the source code of EvolutionRadar with the author, we find a first entry point: each entity generates its view by associating to itself a figure with certain attributes. In the case of EvoloutionRadar, entities are grouped into *PointSets*, a pointset stores all the entities and their status at different time intervals.
At runtime, the `PointSetManager` class is responsible of storing the different pointsets, when a view is generated all the figures for the corresponding entities at a given time are computed. After the layout is applied, the view is presented to the user through the GUI.
Our first application will just generate a web snapshot of the current user screen, we proceed as follows:

**1.** Create a new Episode application, let's call it EvoRadarWeb, this is done by subclassing `WASvgApplication`, this is ready to host a new web application based on a single canvas.

**2.** Implement a method that grabs the current view and makes a quick conversion from the current figures classes to the shapes provided by the framework. The pseudo code for such method, that only converts circles and texts may look like this:

```
EvoRadarWeb >> ImportCurrentView
  //get current precomputed view
  ps := EvolutionRadar:PointSetManager.getCurrentPointSet;

  //iterate on elements of the pointset
  foreach entity in ps do {
    if(entity.figure.class == EvolutionRadar:Circle) { //only circles
        oldCircle := entity.figure;
        newCircle := new Episode:Circle;
        //copy the attributes we want to maintain
        newCircle.x = oldCircle.x;
        newCircle.radius = oldCircle.radius;
        newCircle.color = oldCircle.color;
        ...
        //add the new figure
        self.mainCanvas.add(newCircle);

    } else if (entity.figure.class == EvolutionRadar:Text) { //only texts
        oldText := entity.figure;
        newText := new Episode:Text;
        newText.content = oldText.content;
```

```
          newText.x = oldText.x;
          ...
          self.mainCanvas.add(newText);
      } else {
          //We discard other graphic elements for now
      }
    }
```

There may be problems during this process, for example the coordinate system may not coincide with ours, or (an this is a real example) the shapes position may be referring to their top left corner, while in our case refers to the center. Beside those minor issues, this process is linear.

**3.** We are ready to do a first conversion, we just need to invoke the above method, we do that by placing a link to this action in the application view (we talk of links since at this moment the application can be reached with a web browser). Since we are editing the user interface we add another component: a 'Save as SVG' button is already implemented by the framework, we need just to include it in the application layout.

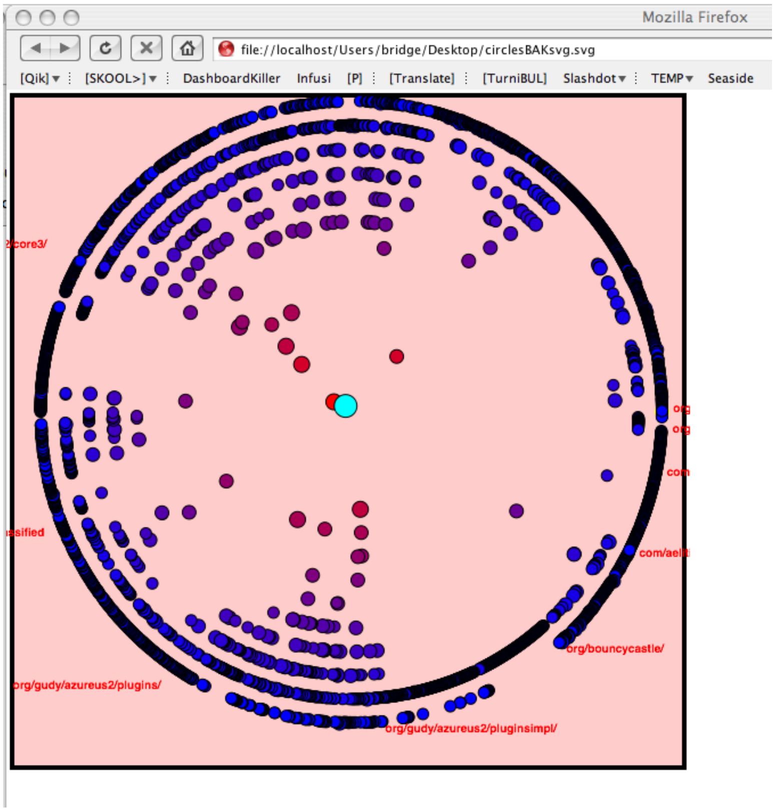


Figure 4.2: An EvolutionRadar snapshot generated by our quick conversion

**4.** We use EvolutionRadar as normal but once we find a view that we want to save we switch to EvoRadarWeb and click on the ImportCurrentView link. If everything worked

we should find the the current radar view in our web browser.
The canvas in which we see the visualization now is included by default by the Episode application. Another inherited behavior lets us already click the circles to bring up a contextual menu in which, for example, we can mark in red some of the figures.
If we are satisfied with what we see, we may click the 'Save as SVG' and what we get is a standalone SVG object like the one in Figure 4.2. This object is nothing more than an XML file, however it is easy to exchange, for example including it in a web page, also modifying it to highlight some particular aspect is straightforward since it is a text file.

### 4.1.2 Reproducing the UI controls

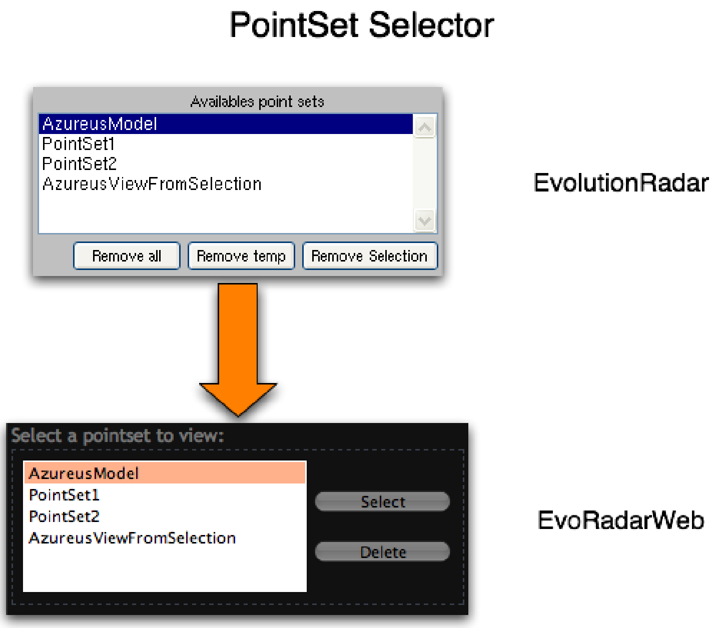


Figure 4.3: The pointset selector in EvolutionRadar and its counterpart in EvoRadarWeb

We already discussed the main controls present in the user interface in Section 4.1, let's now see how can we export them to the web.
We choose as example a piece of the EvolutionRadar interface: the PointSetSelector (see Figure 4.3). This component is responsible for making the user select one of the pointsets previously generated on the model.
Converting this panel is straightforward:

**1.** We create a new Episode component in `EvoRadarWeb`, let's call it `PointSetSelectorPanel`.

**2.** The view of this object is an html form with the following contents:

1. A list of pointsets, we already seen that we can obtain that from the `PointSetManager` class, in html this corresponds to a 'select' element

2. A 'select' button, to confirm the selection (notice that if we enable Javascripts in our application this button is not needed, since just by clicking on the list element we update the selection)
3. A delete button, since it is present in the original panel and we want to convert it. Just like the previous one can be traduced into an HTML 'button' element.

**3.** Next we associate actions to our active components, the action for the 'Select' can be summarized as follows:

```
//Callback action of Select button
selectedPS := self list.selectedElement;
EvolutionRadar:PointSetManager.setCurrentPointSet(selectedPS);
```

We may need to dive a in the EvolutionRadar source code to discover exactly what are the messages to send, however this is trivial in most of the case, since we can inspect the original panel and copy the same functionality.
The delete button is very similar, only it will delete the selected pointset from the PointSetManager, also in this case we need to find up which is the correct method to invoke on it.

**4.** We include `PointSetSelectorPanel` in the previously created `EvoRadarWeb` application, so that it appears in the view (e.g., in the web browser).

**5.** As it is, our `PointSetSelectorPanel` is pretty useless, what it does is just modify some inner attribute of the PointSetManager, we have not even a visual feedback for that.
What we can do, for the sake of this demonstration is change a little bit the action of our select button. We have seen that in EvolutionRadar each pointset contains different "samplings" in time of the model, it would be trivial to implement a selector for a time interval using the methods above described so we don't go through the details of that. Instead we hard-code a choice: automatically select the first time interval, this is only to speed up this demonstration.
We add the following lines to the callback action of the select button:

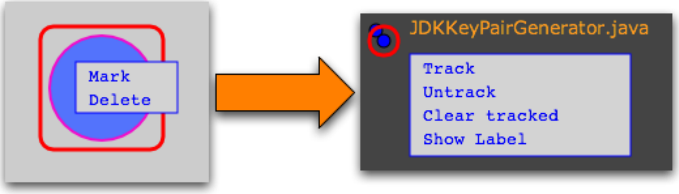
```
selectedTimeInterval = selectedPS.timeIntervalsArray[1];
EvolutionRadar:PointSetManager.setCurrentTimeInterval(selectedTimeInterval);
EvolutionRadar:PointSetManager.generateCurrentView;
//we call the function implemented in the first example
self.ImportCurrentView;
```

What happens now if we select a pointset via web is that the view gets updated in EvolutionRadar but it is also automatically exported to the canvas in the web interface.

### 4.1.3 Introducing graphics interaction

We already discovered the canvas component in Episode, it appears by default as a grey area in which the visualization is drawn. This class is central in our design and has a lot of behavior already implemented, ready to be reused. For example it organizes its content in three groups, *background*, *content* and *GUI*. If we add figures without specifying a group (like in the first example of this section) they are redirected in the **contents**.

What is special about the contents group is that we assume the user will interact with the figures contained, for this reason Episode tries to associate a callback to them (an action that gets executed when we click on the corresponding object). So far, for our experiments, we used `Episode::Circle` as it is, the default behavior for this class in case of click consists of opening a contextual menu (as the one in Figure 4.4).



Default click action for Circle:
contextual menu, and bounding box

EvoRadarWeb Circle override:
different actions and a label

Figure 4.4: We override the default action associated to the click event

This menu is an another object present in the framework, it is a little more complex since it is composed of rectangles and clickable texts, it includes behavior to automatically scale to fit the size of contents, etc..

In EvolutionRadar, clicking one of the elements brings up a contextual menu too, we find for example a function to track a particular entity, so that later we can generate a new view from the selection. Let's implement this functionality in EvoRadarWeb.

**1.** We decide to subclass `Episode::Circle` creating our own circle class, let's call it `ERCircle`. Such class can safely inherit all behavior, the only thing we want to change is the action for the click event.

**2.** We override the `defaultOnClickAction` currently implemented in `Episode::Circle`, this way all of our `ERCircle`s will have this new action.

Note that another solution could have been to dynamically change the click action, by assigning it a block to execute when clicked. This would have saved us a class, however, while this is straightforward in Smalltalk, it may not be so easy to do that with another language, so we stick with the subclassing solution.

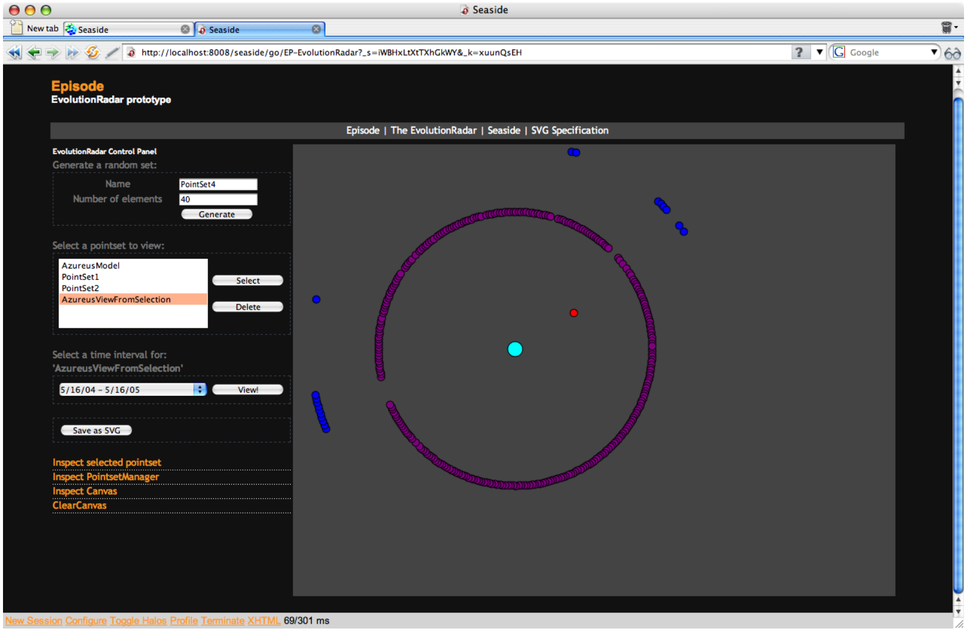
The content of the new action for our circle will look like that:

```
ERCircle >> defaultOnClickAction
  //we display a label with the item entity name
  self.canvas.addElementToGUIGroup(new Text(self.entityName));
  menu := new Episode::Menu;
  //we add ourself to a list of tracked entities
  menu.addAction("Track", self.application.markAsTracked(self));
  self.canvas.addElementToGUIGroup(menu);
```

Clicking on a circle now brings up a contextual menu with a single entry ”Track” that adds the entity to a list of tracked figures managed by the application (we don't go in the details of how this is implemented).

We have seen how easy is to reproduce the functionality of objects that are rendered inside the canvas.

### 4.1.4 The result


Seaside
New tab
Seaside
Seaside
http://localhost:8008/seaside/go/EP-EvolutionRadar?_s=iWBHxLtXtTXhGkWY&_k=xuunQsEH
Google
Episode
EvolutionRadar prototype
Episode | The EvolutionRadar | Seaside | SVG Specification
EvolutionRadar Control Panel
Generate a random set:
Generate
Select a pointset to view:
AzureusModel
PointSet1
PointSet2
AzureusViewFromSelection
Select
Delete
View!
Save as SVG
Inspect selected pointset
Inspect PointsetManager
Inspect Canvas
ClearCanvas
New Session Configure Toggle Halos Profile Terminate XHTML 69/301 ms

Name: PointSet4
Number of elements: 40

Select a time interval for: 'AzureusViewFromSelection'
5/16/04 - 5/16/05

Figure 4.5: Taking the EvoRadarWeb conversion further

In Figure 4.5 we show EvoRadarWeb with a minimal set of components that makes it usable from the web. We still need to use the classical interface to operate on the model and generate pointsets, but we can navigate and explore as well as create new views from selection. EvoRadarWeb could be used for example as an online demo of EvolutionRadar, in which the pointsets are pre-generated.

## 4.2 Building a new graphical application: Ivory

*IVORY: Visualizing Scientific contributions* is a Project by Anja Guzzi implemented with an early version of Episode. Unlike previous examples, this application is not a conversion of an existing tool, it's a brand new visualization web-application.

IVORY explores visualizations in the field of scientific publications, crawling informations from DBLP online databases[3]. *The DBLP server provides bibliographic infor mation on major computer science jour nals and proceedings. Initially the server was focused on DataBase systems and Logic Programming (DBLP), now it is gradually being expanded toward other fields of computer science. You may now read "DBLP" as "Digital Bibliography & Library Project". The server indexes more than 900000 articles and contains several thousand links to home pages of computer scientists.*[4]
However, navigating the DBLP page, we are soon cluttered by hundreds of tables, references, coauthors, citations, etc.. At the point that is really difficult to get an insight of how much a person has contributed.

[3] DBLP is currently maintained by Michael Ley, http://dblp.uni-trier.de/
[4] From the DBLP welcome page, June 2007)

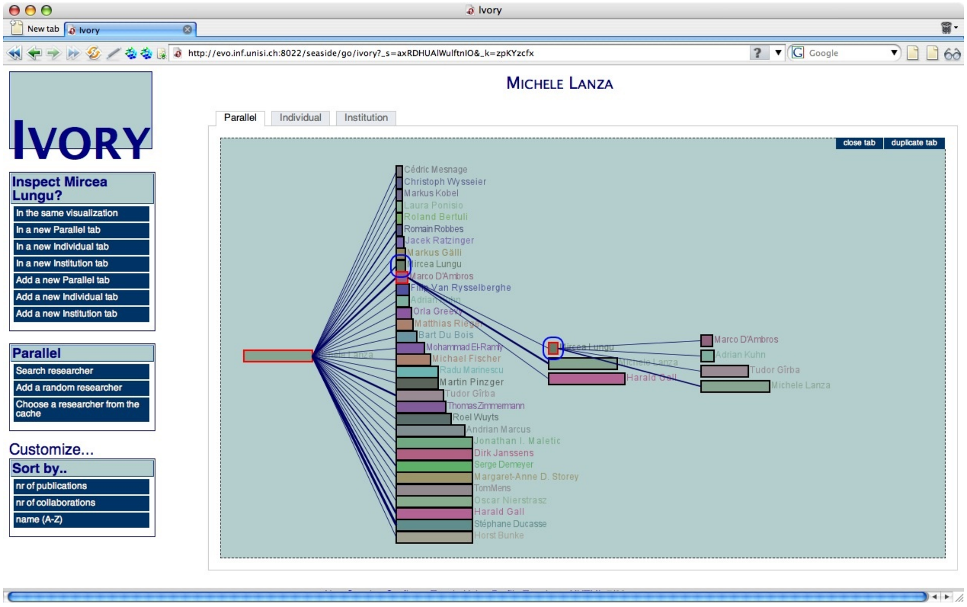


Figure 4.6: A screenshot of IVORY: parallel view

IVORY applies visualization to informations like collaborations and number of publications. The central object of interest are the researchers so in general the size of the figure on the screen, representing a paricular person, is proportional to the amount of works published.

Different views are available: *Parallel visualization* (see Figure 4.6) allows recursive inspection of collaborations rendered as a tree. *Individual visualization* (see Figure 4.7) focuses on a single researcher, producing graphs about collaborators and activities in time. *Institution Visualization* (see Figure 4.8) is ideal to inspect a pool of researcher, the graph edges in this case can represent publications, collaborations or composition and management of the institution.
Researchers are interactive, in the sense that clicking on a person updates the visualization, for example focusing on the selected author.

With Episode creating such application is straightforward and requires little or no knowledge of the SVG specification.
The programmer can focus on the model (the researcher for IVORY) and, like in this case, put together simple and effective visualization web-application.

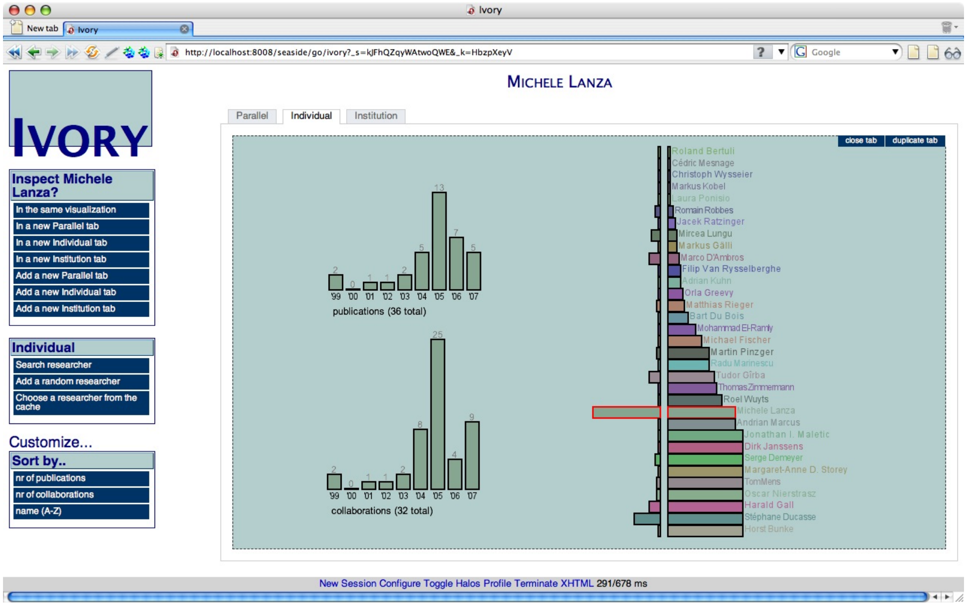


Figure 4.7: A screenshot of IVORY: individual view

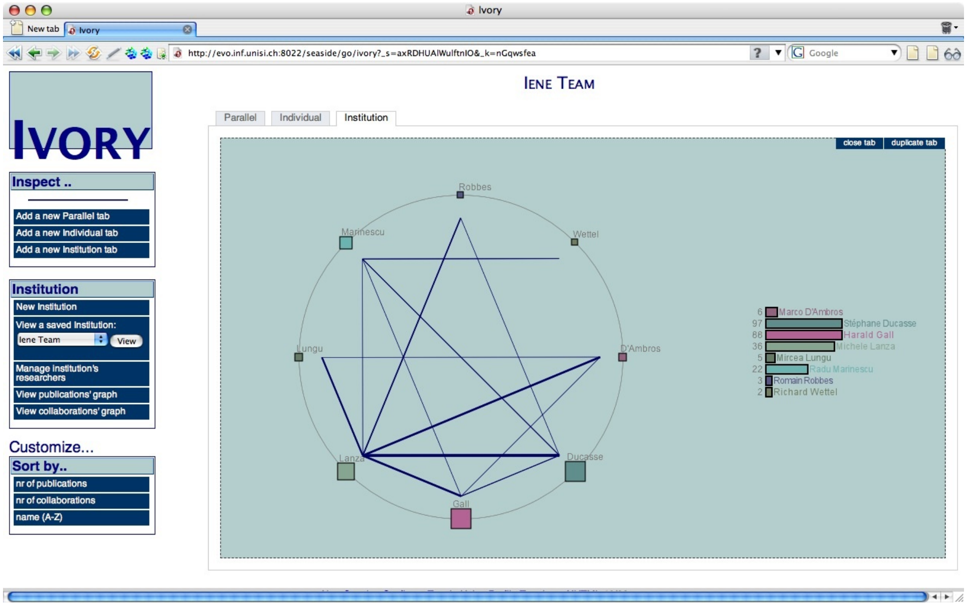


Figure 4.8: A screenshot of IVORY: institution collaborations view

# Chapter 5

# Conclusions

## 5.1 Discussion

The experience of implementing Episode tells us that the web technologies available nowadays makes possible to create web-applications also for graphic-intensive programs like software visualization tools.

The main difficulty encountered during the development and the validation of Episode concerns browser support for SVG.
Most of the browser don't support SVG, Firefox (2.*), Opera (9.*) and Safari (3.*) were used mainly during development. Among those, only Opera manages to be fast enough to be usable. Firefox is really slow when we have an high number of figures to render. Since every interaction we make requires a complete redraw of the page, with Firefox and Safari becomes really boring after a while.
To overcome this problem, we tried to implement the AJAX client-server interaction paradigm, this would allow us to make partial modifications to the canvas without rendering the whole page again.
However different difficulties were encountered in this process, mainly because the support for Javascript within SVG inlined into HTML is not uniform, e.g., we the same action produced different results on different browsers. For this reasons, AJAX scripting is not currently part of Episode.

We think Episode can be useful to export existing tools, however the difficulty of that process varies a lot depending on the tool itself. The more the GUI is highly coupled with the tool, the more difficult it to reconstruct it with Episode. If in the tool we have a clear distinction between the user interface and the underlying functionality, then the exporting process is straightforward. Instead if the GUI design is messy and the distinction between it and the tool itself is fuzzy we may have difficulties.
It follows that is generally much more easier to convert a tool for the author himself, because of a better knowledge of the application architecture.
For example, during the validation of this framework we ported the EvolutionRadar, doing that without having the program author available for questions would have required much more time.

## 5.2 Conclusion

I this document we have presented Episode, a novel approach for building software visualization applications via web.
We discussed similarities of a variety of software visualization tools, trying to abstract a common structure present in all of them, in order to create a framework able to work on top of them, creating a web interface.
The Episode framework addresses mainly two issues: how to make visualizations more portable and how to enhance existing visualization applications without having to rewrite them from scratch.
Episode can be used on top of a software visualization tool to export it to the web, increasing its accessibility with a minimal effort.
Episode produces a visualizations that are standalone objects, introducing the concept of portability for software visualization artifacts.
We presented the web-counterpart of EvolutionRadar, a software visualization tool, by creating EvoRadarWeb. We shown Ivory, an example of application written from scratch within our framework.
This is a positive feedback for on our initial intuition and a good motivation for future work.

## 5.3 Future Work

Our implementation, at the moment is still unripe, it has been validated with a single tool, and we are sure that converting more applications will raise other issues, giving us the chance to expand and improve our framework.
As discussed before we decided not to include any Javascript library at the moment and focus more on structural issues. Indeed AJAX is the only alternative in some case, for example to implement drag and drop or effects, or multiple selection with click and drag etc..

The SVG specification provides functionality to create full-fledged animations, we do not actively exploit this ability in Episode. The number of tools making use of animations is really limited, however it may be the case. For example this would be extremely useful for a specific branch of software visualization: algorithm animations for learning purposes.
Other elements of the SVG specifications were superfluous for the scope of this project so were not considered.

Although someone considers full automatic GUI conversion science fiction, we believe that some other steps can be taken to make the conversion process more automatic. For example, once we know the library originally used (Swing, JHotDraw, etc.) we may be able to associate automatically Episode components counterparts.

## 5.4 Epilogue

In this work we investigated alternative ways to increase the accessibility of software visualization applications, a category of application used in software engineering to an-

alyze systems by means of graphical metaphors.
The Episode framework, result of this project, can be used to enhance the effectiveness of existing tools by creating web-interfaces on top of them, with minimal effort for the developer.
We think that the lesson learned here can be useful in a future where visualization tools are easier to access for everyone, instantaneously. We also think that a server-side application encourage experimentation in the field of collaborative software visualization.
The portable representation of graphics based on XML adopted in this project, can stimulate the dialog on software visualization techniques and contribute to a better teamwork, since it makes easier the sharing of ideas with a more appropriate support.

# Appendix A

# Technologies

## A.1 Seaside

[1] Seaside is a framework for developing sophisticated web applications in Smalltalk. Seaside provides a layered set of abstractions over HTTP and HTML that let you build highly interactive web applications quickly, reusably and maintainably. Seaside includes:

- Programmatic HTML generation. A lot of markup is boilerplate: the same patterns of lists, links, forms and tables show up on page after page. Seaside has a rich API for generating HTML that lets you abstract these patterns into convenient methods rather than pasting the same sequence of tags into templates every time.

- Callback-based request handling. Why should you have to come up with a unique name for every link and form input on your page, only to extract them from the URL and request fields later? Seaside automates this process by letting you associate blocks, not names, with inputs and links, so you can think about objects and methods instead of ids and strings.

- Embedded components. Stop thinking a whole page at a time; Seaside lets you build your UI as a tree of individual, stateful component objects, each encapsulating a small part of a page. Often, these can be used over and over again, within and between applications - nearly every application, for example, needs a way to present a batched list of search results, or a table with sortable columns, and Seaside includes components for these out the box.

- Modal session management. What if you could express a complex, multi-page workflow in a single method? Unlike servlet models which require a separate handler for each page or request, Seaside models an entire user session as a continuous piece of code, with natural, linear control flow. In Seaside, components can call and return to each other like subroutines; string a few of those calls together in a method, just as if you were using console I/O or opening modal dialog boxes, and you have a workflow. And yes, the back button will still work.

Seaside also has good support for CSS and Javascript, excellent web-based development tools and debugging support, a rich configuration and preferences framework,

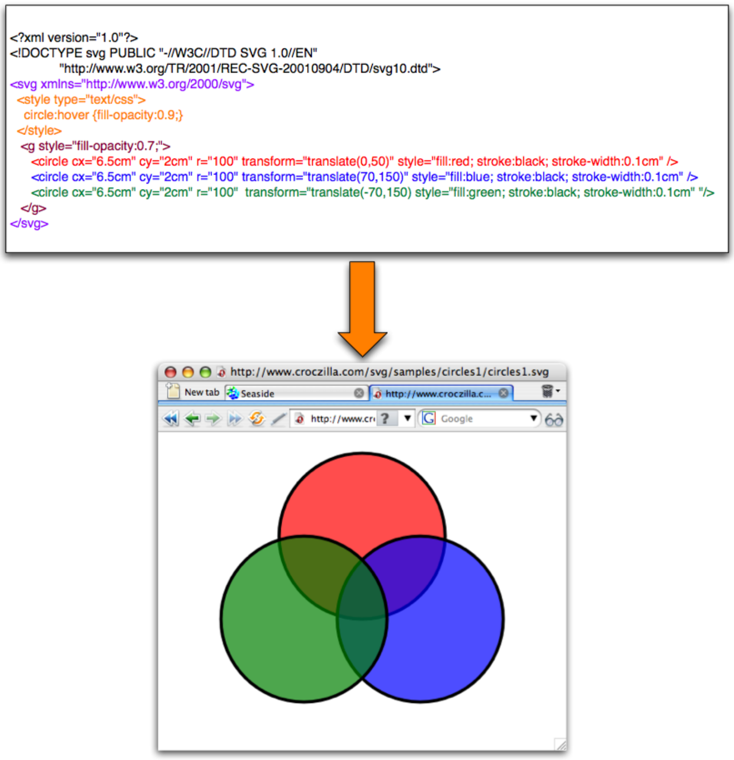
[1] http://seaside.st

and more.

Seaside is currently developed and supported by Avi Bryant, with the help of the Seaside community.

## A.2 SVG

SVG stands for Scalar Vector Graphics. It consists of a XML language for describing 2D graphics. In fact it is a W3C[2] recommendation in terms of graphics applications for the web.
Key features include shapes, text and embedded raster graphics, with many different painting styles. It supports scripting through languages such as ECMAScript or JavaScript and has comprehensive support for animation.
SVG can be compared to Flash in terms of what we can create, however SVG has the main advantage of being XML, as opposed to Flash which is embedded into pages as a binary compiled object. An SVG object can be inlined directly in the html of a page (like in our case) and it supports CSS styling.
SVG is used in many business areas including Web graphics, animation, user interfaces, graphics interchange, print and hardcopy output, mobile applications and high-quality design.
In Figure A.1 we propose a simple SVG example, hovering the circles with the mouse



```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
          "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg">
  <style type="text/css">
   circle:hover {fill-opacity:0.9;}
  </style>
   <g style="fill-opacity:0.7;">
     <circle cx="6.5cm" cy="2cm" r="100" transform="translate(0,50)" style="fill:red; stroke:black; stroke-width:0.1cm" />
     <circle cx="6.5cm" cy="2cm" r="100" transform="translate(70,150)" style="fill:blue; stroke:black; stroke-width:0.1cm" />
     <circle cx="6.5cm" cy="2cm" r="100"  transform="translate(-70,150) style="fill:green; stroke:black; stroke-width:0.1cm" "/>
   </g>
</svg>
```

http://www.croczilla.com/svg/samples/circles1/circles1.svg
New tab
Seaside
http://www.croczilla.c...
http://www.cr
Google

Figure A.1: A simple SVG example, the source code and how it is rendered

[2] W3C is the World Wide Web consortium *http://www.w3c.org*

in a browser changes the opacity of the element.
Browser support for SVG is still uncomplete, at the moment only Opera (since version 9.1) supports a full specification, `SVG 1.1 Basic`.

# Appendix B

# Implementation

Episode is an extension to Seaside, a Smalltalk toolkit to create ”object-oriented” web applications, during this section we discuss how Episode is implemented so some familiarity with Seaside may help in understanding. Refer to Section A.1 for a brief overview of Seaside.

The first modification we make to Seaside regards configuration of the session maintained with an HTTP connection from server to web browser.
Most of the browsers won’t render SVG unless they are aware of it from the beginning of the session.
Changing some of the HTTP headers is a requirement to get SVG inline (directly mixed with html, as opposed to embed an external object which can be done without special configuration).
Another required operation is changing the document type produced by the web server, by default Seaside creates `xhtml` pages. By hacking a little more into the session, we change it to generate pure XML maintaining all the functionalities of Seaside working fine. (See Figure B.1).

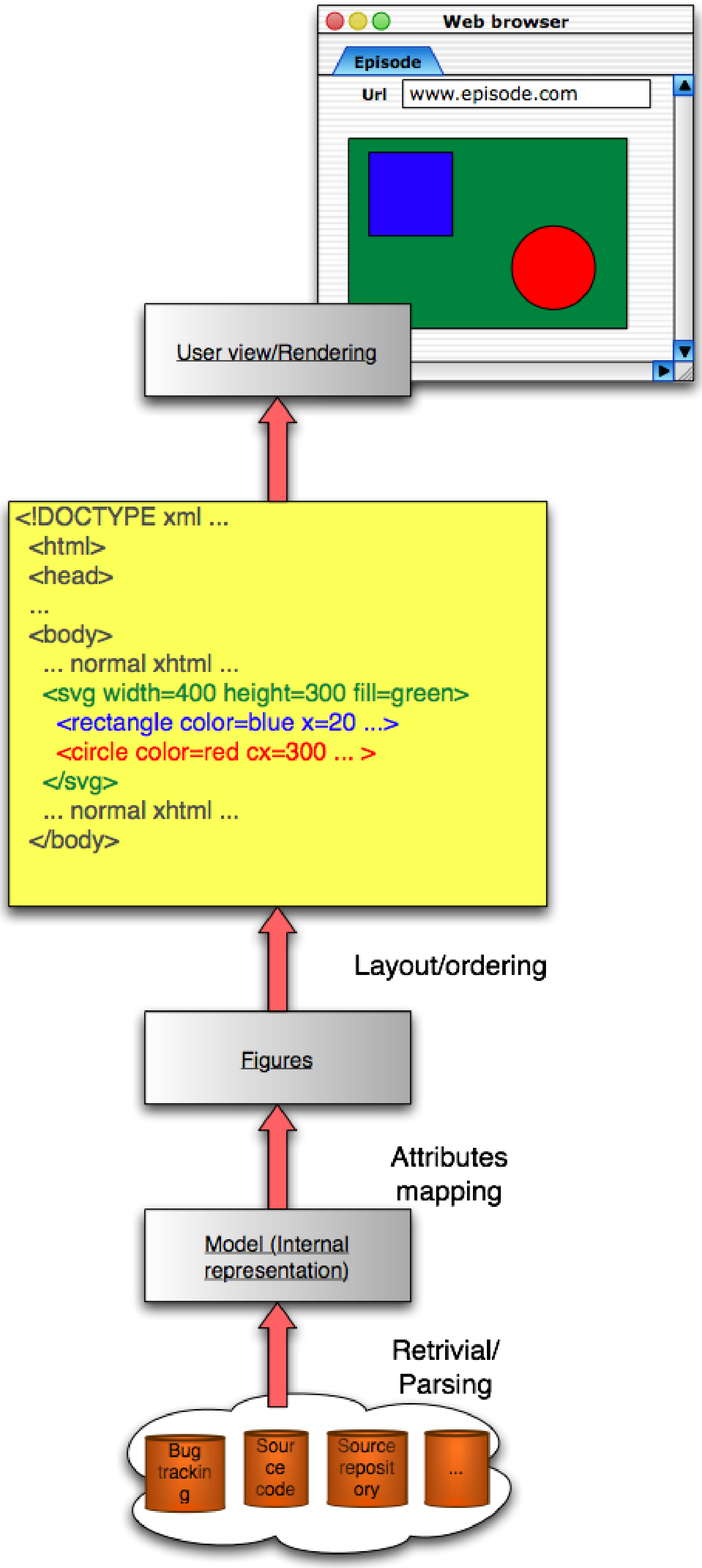
A general principle in Seaside is that each component has three responsibilities: maintaining UI state, reacting to user input, and displaying itself as HTML. What we do in our Episode components is replacing HTML with SVG.
When a object needs to be rendered, it receives a `render` message with the current html canvas as argument. The canvas consists in reality of a stream, into which objects write their html representation.
For example an html table when rendered will write to the canvas a new `tableTag` (”`<table>`”), then will recursively invoke the rendering on its content (the rows in this case). When the children are done, the `tableTag` will close (”`</table>`”).
The same mechanism can work for SVG so we extend the library of tags with what we need. Extending Seaside in this way is optimal since we are ”transparent”, in practice this means that we inherit all of the functionalities, since our objects are not different at all from Seaside components.

The next steps consists in creating a library of objects, since we need to interact directly with the object (while for example in a table we may interact with the *content*). So it makes sense to, for example, create the `Circle` object, this way a lot of functionalities

can be pushed into it, to be shared and reused.
This takes the object-oriented paradigm directly on the web browser, when we click on a circle we are really sending a message to the circle itself.
For example a circle instance with no associated action will just render as circle element in the XML. Instead a circle which has an action associated to the click, when drawing will wrap itself in a link element, to become clickable. (Actually we have many other ways of doing that, but this is the most similar to Seaside and helps a clearer inheritance).

During the validation of Episode we used it to convert existing application, this process enriched a lot the library of functionalities offered by the components. For example it becomes necessary to divide the canvas contents into tree main groups, background (without actions, may be updated if we move the navigate), GUI (elements like contextual menu are separated from the rest so we can get rid of all of them in a single action when we update the view) and contents (which contains the actual contents).

In Figure B.2 we see the polymetric view of the Episode namespace (notice that other classes that extends Seaside functionality (for example SVG tags) are implemented in the Seaside namespace and thus not visible there), in Figure B.3 an we see the main classes in a hierarchical tree with root in WAComponent, the base class for Seaside applications, extending this class allows us to inherit all behavior of Seaside objects.



```
<!DOCTYPE xml ...
  <html>
  <head>
  ...
  <body>
    ... normal xhtml ...
    <svg width=400 height=300 fill=green>
      <rectangle color=blue x=20 ...>
      <circle color=red cx=300 ... >
    </svg>
    ... normal xhtml ...
  </body>
```

Figure B.1: The intermediate form adopted by the visualization

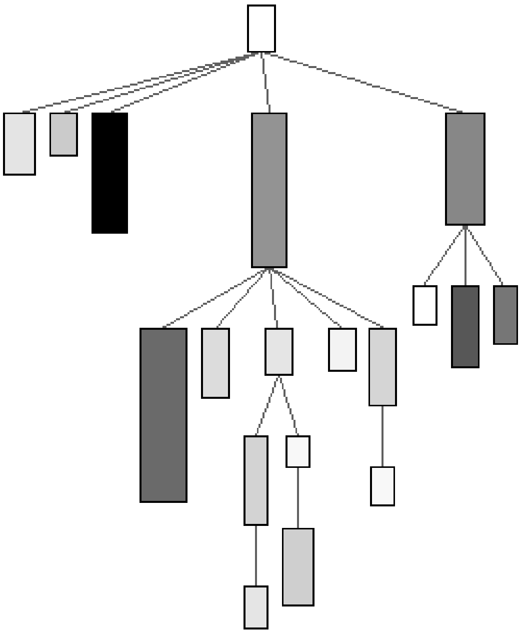


Figure B.2: Polymetric view of the Episode main classes

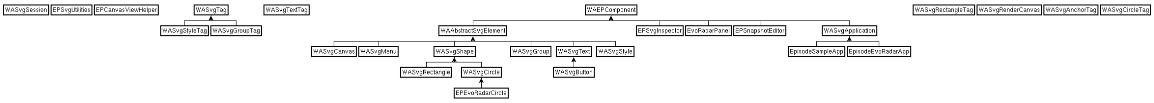


Figure B.3: UML class diagram of the Episode namespace

# List of Figures

# Bibliography

[BH94] Benjamin B. Bederson and James D. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, New York, NY, USA, 1994. ACM Press.

[BH06] Dirk Beyer and Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. *wcre*, 0:199–210, 2006.

[Die02] Stephan Diehl, editor. *Software Visualization*. Springer, 2002.

[DL06] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pages 189 – 198, 2006.

[DLL06] Marco D'Ambros, Michele Lanza, and Mircea Lungu. The evolution radar: Integrating fine-grained and coarse-grained logical coupling information. In *Proceedings of MSR 2006 (3rd International Workshop on Mining Software Repositories)*, pages 26 – 32, 2006.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.

[ESEE92] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[FBDJK07] Wim Fikkert, Torsten Bierz, Marco D'Ambros, and TJ Jankun-Kelly. Interacting with visualizations. In Andreas Kerren, Achim Ebert, and Joerg Meyer, editors, *Human-Centered Visualization Environments*, volume 4417, chapter 3, pages 77–161. Springer LNCS, june 2007. in press.

[JSB97] Dean J. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of ICSE '97*, pages 360–370, 1997.

[Kei02] Daniel A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 08(1):1–8, 2002.

[KLS00] Matthias Kreuseler, Norma Lopez, and Heidrun Schumann. A scalable framework for information visualization. *infovis*, 00:27, 2000.

[Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[Lan03] Michele Lanza. Codecrawler – a lightweight software visualization tool. In *Proceedings of VisSoft 2003 (2nd International Workshop on Visualizing Software for Understanding and Analysis)*, pages 51–52. IEEE CS Press, 2003.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[LL06] Mircea Lungu and Michele Lanza. Softwarenaut: Exploring hierarchical system decompositions. *csmr*, 0:351–354, 2006.

[MADSM01] C. Best M.-A. D. Storey and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[MFM03a] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

[MFM03b] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, New York, NY, USA, 2003. ACM Press.

[ML05] Cdric Mesnage and Michele Lanza. White coats:web-visualization of evolving software in 3d. In *Vissoft'05*, 2005.

[SDSD02] Tom Soukup, Ian Davidson, Tom Soukup, and Ian Davidson. *Visual Data Mining: Techniques and Tools for Data Visualization and Mining*. Wiley, May 2002.

[SM95] M.-A. D. Storey and H. A. Muller. Manipulating and documenting software structures using shrimp views. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 275, Washington, DC, USA, 1995. IEEE Computer Society.

[VRD04] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, September 2004. IEEE Computer Society Press.

[VT06] Lucian Voinea and Alexandru Telea. Mining software repositories with cvsgrab. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 167–168, New York, NY, USA, 2006. ACM Press.

[WL07a] Richard Wettel and Michele Lanza. Program comprehension through software habitability. In *Proceedings of 15th International Conference on Program Comprehension (ICPC 2007)*. IEEE Computer Society, 2007.

[WL07b] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Proceedings of 4th IEEE International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT 2007)*. IEEE Computer Society, 2007.