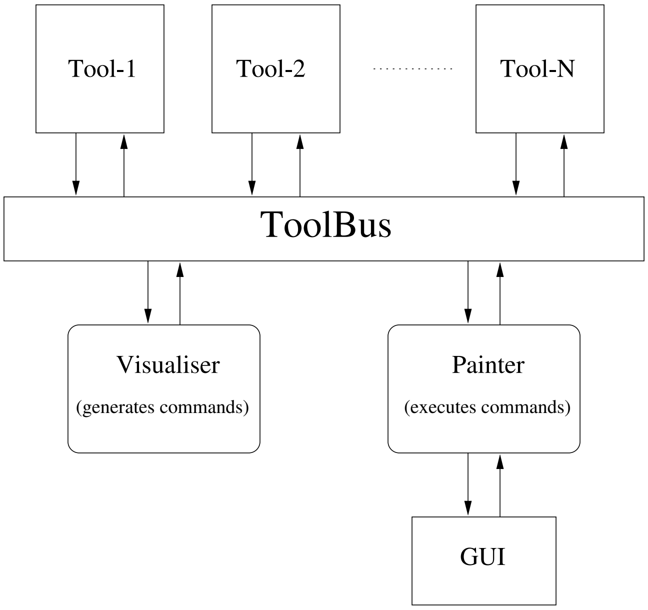
A Visualisation Framework for ToolBus Applications

Hayco de Jong

November 1999



Master's Thesis in Computer Science
Programming Research Group
Faculty of Mathematics, Computer Science, Physics and Astronomy
University of Amsterdam
Supervisors: prof. dr. P. Klint, dr. M.G.J. van den Brand, drs. P.A. Olivier

Contents

1 Introduction **5**
1.1 General 5
1.2 Application domain 6
1.3 Related work 6
1.4 Structure of this thesis 7
1.5 Acknowledgements 8

2 ToolBus in a nutshell **11**
2.1 Introduction 11
2.2 The ToolBus philosophy 11
2.3 The ToolBus architecture 12
2.4 Data exchange between ToolBus and tools 13
2.5 ToolBus-scripts 14
2.6 Example ToolBus-script 16

3 The Art of Debugging **19**
3.1 General 19
3.2 Code augmentation 20
3.3 Code augmentation hazards 20
3.3.1 Incorrect debug statements cause confusion 20
3.3.2 Overzealous removal introduces new bugs 21
3.3.3 Insertion of statements suppresses the bug 21
3.4 External debuggers 22
3.4.1 Using a dedicated debugger 22
3.4.2 Using a language independent debugger 22
3.5 External debugger hazards 23
3.5.1 Overhead involved in using a debugger 23
3.5.2 Optimisation and debugging are mutually exclusive 24
3.5.3 Implicit suppression of the bug 24
3.6 The hybrid approach 24
3.7 Summary and conclusions 25

4 Design of the Framework **27**
4.1 General 27
4.2 Breaking down the framework 27
4.2.1 The visualisation step 27
4.2.2 The painting step 28
4.2.3 Connecting the steps 28

4.3 Requirements issues 29
4.3.1 Coordinate conversions 29
4.3.2 Colour configuration 29
4.4 Visualisation instructions 30
4.4.1 Atomic animation elements 30
4.4.2 The visualisation canvas 32
4.5 Summary 33

5 Implementation of the Framework **35**
5.1 General 35
5.2 Implementation of the VISUALISER 35
5.2.1 Unique identification of expressions and instructions 36
5.2.2 Choosing a language for the VISUALISER 36
5.2.3 Starting a new visualisation 36
5.2.4 Updating the value of an expression 37
5.2.5 Requesting a list of all possible templates 37
5.2.6 Sending a visualisation instruction 38
5.2.7 Class hierarchy of the VISUALISER 38
5.3 Implementation of the PAINTER 38
5.3.1 Alternative visualisation packages 38
5.3.2 PainterTool.java 41
5.3.3 AnimCanvas.java 42
5.3.4 Class hierarchy of the PAINTER 44

6 Examples **45**
6.1 The GenericAnimator template 45
6.2 The ProgressAnimator 47
6.3 The PieAnimator 51

7 Concluding remarks **53**
7.1 Java implementation quirks 53
7.2 Summary 54
7.3 Conclusions 55
7.4 Future work 55

A The visualiser process **59**

B The painter process **61**

C The PainterTool class **63**

D The AnimCanvas class **69**

E The GenericAnimator class **79**

Chapter 1

Introduction

1.1 General

Typical ToolBus[1] systems have many dedicated subcomponents. An example of such a ToolBus system is the new ASF+SDF Meta-Environment [8]. It has amongst others an I/O component, a parse table generator, a parser, a compiler, an interpreter, and a graphical user interface (GUI).

As components of such ToolBus applications are often dedicated, performing specialised tasks, they may have the need to present data in a visual, user friendly manner, even though they are not well equipped to do this. Imagine for example a (simple) weatherstation with a thermometer and a pluviometer ("rain gauge"). Each of these meters is designed to measure and yield data (temperature and amount of rain). Suppose however, that instead of a *raw* data dump (17°, 5mm) we want a nice picture of a thermometer showing just the right amount of mercury depending on the current temperature and a picture of a measuring jug showing the current precipitation. The conversion of data offered by a component into a visual representation and subsequent rendering is called *visualisation*.

The example of the weatherstation uses physical instruments (a thermometer and a pluviometer) as examples of components that yield data. It is, however, just as imaginable that these instruments are software components. For example, we might want to monitor the progress of a batch of computations, or perhaps check the status of a mailbox to see if new mail has arrived.

Another example is the ToolBus Integrated Debugging Environment, called TIDE (see [13]). This environment can be used to debug heterogeneous applications with components written in different programming languages. Each language has a dedicated debugger that is linked to the ToolBus via a debug adapter. The user works in this debugging environment using several ToolBus-components, such as a *source-code viewer* to display the code under review and a *process viewer* showing the current state of all processes involved. One crucial component is not yet present in TIDE. TIDE lacks a component to display the values of variables and expressions used by the program being scrutinised during a debugging session.

[1]Chapter 2 presents a crash course in understanding the ToolBus-philosophy and reading ToolBus-scripts for users unfamiliar with the ToolBus coordination architecture.

This thesis presents a number of ToolBus-components which together with a ToolBus-script form a visualisation framework. An *expression* (current temperature, state of the mailbox) is visualised by passing its current value to a *visualiser* which issues instructions that are used to drive a *painter*. The visualiser and painter are separate components. They communicate with each other and with the component responsible for yielding the expression's current value by means of the ToolBus. This framework was designed to extend TIDE with a way to visualise data, while at the same time being generic enough not to be restricted to *only* work with TIDE.

This thesis does not aim to provide a complete set of templates for the visualisation of every possible expression. Rather, its aim is to present a user extensible visualisation framework, allowing the framework to be retrofitted with new visualisation templates as the need for them arises.

1.2 Application domain

This thesis presents a visualisation framework using ToolBus-components. It may therefore not be easy to identify uses outside the realm of ToolBus applications. The main purpose of this visualisation framework is to complete the caveat in TIDE by extending it with a generic way to view program variables and expressions. But part of the strength of ToolBus-components lies in their ability to be reused in other (heterogeneous) systems as well. The new ASF+SDF Meta-Environment [8] is a complex system using several ToolBus-components. It may very well benefit from the visualisation framework described in this thesis for some of its display activities, such as the values of rewriting of terms, error messages concerning parsing, and parse trees.

1.3 Related work

A most stimulating piece of related work was found in the Animation Designer's Packages Polka [17] and Samba [18]. POLKA offers a core of functionality to create animations from programs written in the `C++` programming language. SAMBA is an interpreter built on the POLKA core, which reads primitive animation commands from an ASCII file, and uses POLKA to create the actual animations. In fact, an attempt has been made to reuse these packages for the PAINTER component (Section 5.3). The approach taken in POLKA is to augment source code written in `C++` by inserting statments that drive a visualisation engine. To facilitate visualisation of programs *not* written in `C++`, the SAMBA interpreter is used and the visualisation is then driven by reading instructions from a file. The approach described in this thesis uses the ToolBus to drive the visualisations. It does not pose a restriction on the language used the way POLKA does, and it does not require the source code to drive the visualisation. It may need augmentation of the program, but only to extract the values of the expression being visualised, no visualisation specific instructions need to be introduced.

Another work on visualisation, specifically tailored towards visualisation of linked lists for debugging purposes, can be found in [15]. It shows how the output of an external debugger can be used to visualise (portions of) linked lists.

Specific techniques are used to improve readability of the visual representation of the list, at the same time condensing this representation to minimise the amount of "wasted" space. This approach can *only* be used in conjunction with the Unix `dbx` debugger, and *only* to visualise linked lists. It is a powerful, highly specialised tool for a single purpose. Our approach is much more generic and flexible, allowing visualisation of many different kinds of expressions. By extending the framework with new visualisation templates, the framework can be tailored to user specific needs.

An earlier package, called PROVIDE, is given in [12]. It describes a "Process Visualisation and Debugging Environment", presenting graphical representations such as pie charts and bar graphs for data in a process. Its visualisation efforts concentrate on usage in a debugging environment, and to this end, it builds a database of the values that variables take on during execution, allowing the user to "step back" in time and view previous values of an expression as well as current values. The PROVIDE package offers a *static* set of visualisation templates from which the user can choose. As stated before, this differs from our approach in the amount of flexibility offered. This thesis describes a system which can be modified and extended as the need arises, thus offering a more *dynamic* set of visualisation templates.

Part of the art of visualisation lies in defining how certain expressions should be visualised, such that the result is easy to interpret, and if possible, aeshetically pleasing as well. While it may be easy to think up a visualisation for a boolean variable (which is either *true* or *false*), visualising a graph is much more complex. We do not offer a fixed set of visualisation templates, but rather stimulate users to extend the framework by adding new templates. In [23], a discussion of achievements in the field of "Graph Drawing and Information Visualisation" is found, which offers theoretical background for those users trying to implement some of the more intricate visualisation templates.

1.4 Structure of this thesis

The remaining chapters of this thesis are structured as follows. Chapter 2 presents the ToolBus in a nutshell explaining the philosophy behind it as well as explaining how to read the ToolBus-scripts presented in this thesis. Following this is an introduction in the field of program debugging in Chapter 3. These introductory parts are necessary to understand the design of the visualisation framework presented in Chapter 4. With this design in mind, the implementation of the framework's components will be discussed in Chapter 5. Some examples of visualisation templates, including detailed sections of source code, can be found in Chapter 6. Finally, conclusions and future work are detailed in Chapter 7.

1.5 Acknowledgements

No man is an *Iland*, intire of it
selve; every man is a peece of the
Continent, a part of the *maine*; if a
Clod bee washed away by the *Sea*,
Europe is the lesse, as well as if a,
Promontory were, as well as if a
Mannor of thy *friends* or of *thine*
owne were; any man's *Death* dimin-
ishes *me*, because I am involved
in *Mankinde*; And therefore never
send to know for whom the *bell* tolls;
It tolls for *thee*.

John Donne, 1572–1631

Performing an intellectual task such as writing a Master's Thesis, is seldom a solitary enterprise. On the professional level a host of colleagues, peers and supervisors see to it that despite any hard times that inevitably pop up every now and then, the project is brought to a successful conclusion. In particular, I wish to thank Paul Klint, Pieter Olivier, Mark van den Brand, Jan Bergstra and Chris 'X' Verhoef for their support, motivation, criticism, and helpful remarks and gestures. Without their professional and social input, finishing this thesis would surely have become quite a challenge indeed.

On the social level, I've had the honour to be surrounded by so many dear friends and colleagues, that I could easily spend an entire section extending my gratitude to each of them. First and foremost however, I would like to thank my mother for her never failing support and love. Whether it was a linguistic, social, or even financial challenge, you were always there for me. Every son is supposed to disagree with his mom from time to time, but in the end, I would not want any other!

Next, I would like to thank Susanne Laane. We have had our share of laughs and tears, of love and pain. Even though we are no longer together, I look back at the three close years we shared with a smile in my heart and I thank you for your love and support during this period.

Of my closest friends, I would like to single out two people who have been of great importance to me during my "Thesis Years". First, I would like to thank Jean-Pierre, or JP as I more often call you. You were there for me when I needed you most. You found room, not only in your apartment, but also in your heart, to put up with my antics for six months while I was looking for a place of my own. Thanks! Second, I would very much like to thank Eggie van Buiten. A soon-to-be ex-colleague, but above all a true friend. Even though you are a relatively new friend, we have already shared many hours of "Fun 'n Games", as well as many interesting conversations about the more important things in life, such as "Ome Henk", "Pipo de Clown", and trying to get a cable telephone from the A2000 company.

Finally, I would like to thank all my dear friends that I can't spend an entire paragraph on. In no particular order, I would like to name Ruben, Meriam, Mike, Tracy, Arno, Maud, Richard, Elin, Timme, Alain, David, Katalin, Oma, Nanon, Wendy, Oki, Wim, Els, Cara, Sander and any friends that may have slipped from this list. In one way or another, each of you has had an impact on me and I thank all of you for your contribution to my life.

Chapter 2

ToolBus in a nutshell

2.1 Introduction

This thesis presents a number of ToolBus-components operating in a ToolBus environment. To fully understand the design issues as well as the code and interaction between the components, at least a basic understanding of ToolBus-programming is needed. This chapter presents an overview of the ToolBus architecture and gives an introduction to ToolBus-programming through the use of ToolBus-scripts.

Readers who are already familiar with the ToolBus can safely skip this chapter, or skim through it to freshen up their knowledge. Interested readers may find a complete description of the ToolBus in [4]. In fact, Sections 2.2, and 2.3 are derived from [4, Chapters 1 and 2]. Sections 2.4 and 2.5 are taken from [3], an earlier report on the ToolBus containing a more elaborate section on the definitions of *terms* and ToolBus *primitives*.

A number of established techniques have been applied to approach the design of the ToolBus at various levels of abstraction. Details can be found in [3] on the use of process algebra [2], algebraic specification using ASF+SDF [10], and C [11] to describe the ToolBus.

2.2 The ToolBus philosophy

We begin this Section by quoting the first paragraph of the introduction in [4]:

> Building large, heterogeneous, distributed software systems poses serious problems for the software engineer. Systems grow *larger* because the complexity of the tasks we want to automate increases. They become *heterogeneous* because large systems may be constructed by re-using existing software as components. It is more than likely that these components have been developed using different implementation languages and run on different hardware platforms. Systems become *distributed* because they have to operate in the context of local area networks.

As software systems become larger, it becomes less and less feasible to build them in a *monolithical* way. Often, individual subcomponents can be identified, which allows for specialised, re-usable implementations in a language best suited to these particular components. Also, the idea of being able to replace a single component or *tool*, as opposed to *patching up* the entire monolithical system, is much more attractive.

When a software system is broken up into individual subcomponents, these are going to have to communicate with each other in one way or another. Rather than having the components shout to each other like toddlers in a kindergarten, they should communicate sensibly if the software system is to be a success. This is where the ToolBus comes in. The ToolBus is perhaps best summarised by quoting the abstract from [3]. Through the use of the ToolBus, we can

> ... take control over all possible interactions between the software components ("tools"), by forbidding direct inter-tool communication. Instead, all interactions are controlled by a process-oriented `T` script that formalises all the desired interactions among tools. This leads to a component interconnection architecture resembling a hardware communication bus, and therefore we call it a "TOOLBUS".

2.3 The ToolBus architecture

The ToolBus defines the cooperation of a variable number of components, or *tools* T_i $(i = 1, \ldots, m)$ that are to be combined into a complete system. Each tool T_i can be thought of as a black box. Its *internal* behaviour is irrelevant to the ToolBus. We concentrate solely on the *external* behaviour of each tool. In general, an *adapter* will be needed for each tool to adapt it to the common data representation and message protocols imposed by the ToolBus.

The ToolBus itself consists of a variable number of processes P_j $(j = 1, \ldots, n)$. These processes will sometimes be called ToolBus-processes to avoid confusion with, e.g., processes at the operating system level.

The parallel composition of the ToolBus-processes P_j now represents the intended behaviour of the entire system. The tools T_i are external, computational activities. Tools come into existence in either of the following ways:

- The ToolBus takes the initiative and issues a command to execute the tool.
- Execution of the tool is initiated externally and the ToolBus explicitly allows an instance of this tool to connect.

The processes and tools are deliberately given unique indices i and j respectively. Although a one-to-one correspondence between tools and processes may seem desirable, the ToolBus does not enforce this and permits tools being under the control of multiple processes, as well as a single process controlling multiple tools.

The remainder of this section explains the different kinds of communication the ToolBus can handle.

Communication inside the ToolBus. Inside the ToolBus, there are two communication mechanisms available. First, a process can send a *message* (using `snd-msg`) which should be received by a single other process (using `rec-msg`). This type of communication is known as *synchronous message passing*.

Second, a process can send a *note* (using `snd-note`) which is broadcasted to other, interested, processes. The sending process does not expect an answer while the receiving processes read notes asynchronously (using `rec-note`) at a low priority. Notes are intended to notify others of state changes in the sending process. Sending notes amounts to *asynchronous selective broadcasting*. Processes will only receive notes to which they have *subscribed*.

Communication between ToolBus and tools. The communication between ToolBus and tools is based on handshaking communication between a ToolBus-process and a tool. A process may send messages in several formats to a tool (`snd-eval`, `snd-do`, and `snd-ack-event`), while a tool may use the formats `snd-value` and `snd-event` to a ToolBus-process.

Once more we emphasise that there is *no direct communication between tools*.

2.4 Data exchange between ToolBus and tools

Data can be exchanged between the ToolBus and tools, and data can be manipulated inside the ToolBus. For such an exchange or manipulation to take place, the data must be represented as prefix expressions called *terms*. Some examples of terms are: `true`, `add(3, mul(4, 5))` and `destination(flight("KL666"), "Terrormolinos")`.

The following definitions and examples are an almost verbatim copy of [3, Section 2.2 (p.8-9)], except for minor changes which are explained in the footnotes below.

The following *types* are defined:

- `bool` represents Boolean values.
- `int` represents Integer values.
- `str` represents String values (strings of characters).
- `list` represents lists whose elements may have arbitrary types.
- `list(`*Type*`)` represents lists whose elements are of type *Type*.
- `term` represents arbitrary terms.
- The single identifier *Id* represents a constant term with function symbol *Id*.
- $Id\texttt{(}Type_1, \ldots, Type_n\texttt{)}$ represents terms of the form $Id\texttt{(}Term_1, \ldots, Term_n\texttt{)}$, where $Term_i$ is of type $Type_i$.
- $\texttt{[}Type_1, \ldots, Type_n\texttt{]}$ represents lists[1] of the form $\texttt{[}Term_1, \ldots, Term_n\texttt{]}$, where $Term_i$ is of type $Type_i$.

The following definition of *terms* is used:

- A Boolean value B is a term.
- An Integer value I is a term.
- A String[2] value S is a term.
- A variable V is a term. The ToolBus enforces the convention that variables start with an uppercase character.

[1] The original text in [3] incorrectly states that this type represents *terms* instead of *lists*.
[2] As before, "String" refers to a string of *characters*.

- A result variable $V?$ is a term.
- A single identifier *Id* is a term. The ToolBus enforces the convention that identifiers start with a lowercase character.
- An application $Id(Term_1, \ldots, Term_n)$ is a term, provided that $Term_1, \ldots, Term_n$ are also terms.
- A list $[Term_1, \ldots, Term_n]$ is a term, provided that $Term_1, \ldots, Term_n$ are also terms.
- A placeholder <*Type*> is a term.

The *basic data types* Boolean, Integer, and String are considered standard. Elaboration on these types is beyond the scope of this thesis.

Variables appearing in ToolBus-scripts have to be declared to be of a certain *type*. The ToolBus enforces that only terms of the appropriate type will be assigned to variables.

The ToolBus distinguishes two kinds of *occurrences of variables*:

- *Value occurrences* of the form V whose value is obtained from the context in which they are used.
- *Result occurrences* of the form $V?$ who get a value assigned depending on the context in which they occur; this may be either as a result of a successful match with another term, or as a result of an assignment.

For example, in a context where variable `X` has value `3`, the term `f(X)` is equivalent to `f(3)`. When, on the other hand, the terms `f(X?)` and `f(3)` are matched, the value `3` will be assigned to variable `X` as a result of this successful match.

Placeholders are intended to define *term patterns* in which certain positions are marked with the required type at that position. For instance, the term `add(<int>,<int>)` defines the type of a function `add` with two arguments of type `int`.

After the ToolBus was developed, its implementation of *terms* based on the *make and match* paradigm was not only used in ToolBus applications, but quickly found its way around to internal projects which had nothing to do with the ToolBus in particular. Rather, the fact that it had terms *built in* made people use the ToolBus, just so they could work with the notion of terms. This term library was later separated from the ToolBus and recent developments in [7] have yielded a high-efficiency implementation of what are now called *Annotated Terms*, or *ATerms* for short.

2.5 ToolBus-scripts

The complete behaviour of a ToolBus system is described in a "ToolBus-script". Such a script consists of the parallel composition of a number of processes, each defined by a *process expression*. This Section describes only those process expressions actually used in this thesis. An exhaustive description of all process expressions available in ToolBus-scripts can be found in [4, Section 2.3 (p.9-14)].

- `delta`: the atomic process corresponding to inaction or deadlock (hence the name of the Greek letter δ). The notion of `delta` is familiar from process algebra [2].

Introduction of variables

- `let` Var_1`:`$Type_1$`,...` `in` P `endlet`: introduces new variables and their required type in process expression P.

Assignments

- V `:=` $Term$: assigns the result of evaluating $Term$ to variable V. Variables occurring in $Term$ are replaced by their current value.

Synchronous process communication (*messages*)

- `snd-msg` and `rec-msg`: used for sending and receiving messages between two processes using synchronous communication. A `snd-msg` can communicate with *exactly one* `rec-msg` that matches the `snd-msg`'s argument list. Both atoms will assign values to result variables (marked with `?`) appearing in their argument lists; these can be used later on in the process expression in which these atoms occur.

Asynchronous process communication (*notes*)

- `subscribe` and `unsubscribe`: subscribe or unsubscribe, respectively, to notes of a given form. A process will only receive notes to which it has subscribed.
- `snd-note`, `rec-note`: used for sending and receiving notes via asynchronous, selective broadcasting. A `snd-note` is used to send to all (i.e. zero or more) processes that have subscribed to notes of that particular form. Each process maintains a queue of notes that have been received but have not yet been read. In this way, notes can never be lost. A `rec-note` will inspect the note queue of the current process, and if the queue contains a note of a given form, it will remove the note and assign values to variables appearing in its argument list; these can be used later on in the process expression in which the `rec-note` occurs.

Execution and connection of tools

- `execute`: starts the execution of a tool.
- `rec-connect`: receives a request to establish a connection with a tool already executing outside the ToolBus.

Communication between ToolBus and tools

- `snd-eval`: requests a tool to evaluate a term. The first argument serves as the identification of the tool, while the second argument is the term to be evaluated.
- `rec-value`: receives from a tool the result of a previous `snd-eval` request.
- `snd-do`: sends a term to a tool without expecting a result term.
- `rec-event`: receives an event from a tool. The first argument of `rec-event` is a tool identification. The second argument serves as an identification of the source of the event. The remaining, optional, arguments give the details of the event in question. The ToolBus protocol dictates that after a tool sends an event of a certain type, it does *not* send another event of that type *until* the previous instance has been acknowledged.

- `snd-ack-event`: sends an acknowledgement of a previously received event to a tool.

Termination and disconnection of tools

- `snd-terminate`: terminates a currently executing tool.
- `rec-disconnect`: receives a request to disconnect a tool from the ToolBus (without terminating its execution).
- `shutdown`: terminates *all* currently running tools as well as all ToolBus processes.

The following 4 binary operators can be used to combine process expressions. They are presented in order of descending precedence:

- $P_1 * P_2$ (binary Kleene star operator): zero or more *repetitions* of process expression P_1 followed by process expression P_2.
- P_1 . P_2: the *sequential composition* of process expressions P_1 and P_2, i.e., P_1 followed by P_2.
- $P_1 + P_2$: the *choice* between process expressions P_1 and P_2.
- $P_1 \mid\mid P_2$: the *parallel composition* of process expressions P_1 and P_2 inside one ToolBus process. Note that no communication is possible between P_1 and P_2 as this is only permitted between process expressions appearing in *different* ToolBus processes.

2.6 Example ToolBus-script

Consider a conversion system consisting of a conversion tool capable of calculating currency conversions. That is, given a certain amount of money in one currency, the tool calculates the value in another currency. The system allows multiple *clients* to log on, ask for one or more conversion calculations, and log off. Finally, to keep the conversion rates up-to-date, a single *master* can log on to update the rates.

As the purpose of this example is to demonstrate some of the ToolBus primitives in an actual `T` script, the discussion of security issues is beyond the scope of this example. We first present the ToolBus code for the example, followed by a detailed explanation.

```
process Convert is
  let Cid : convert, E : str, V : int
  in
    execute(convert, Cid?) .
    subscribe(update(<str>, <int>) .
    ( rec-note(update(E?, V?)) .
      snd-do(Cid, update(E, V))
    ) * delta
    ||
    ( rec-msg(convert, E?) .
      snd-eval(Cid, convert(E)) .
      rec-value(Cid, V?) .
      snd-msg(convert, E, V)
    ) * delta
  endlet
```

```
tool convert = { command = "currency-converter" }

process ConnectClients is
  let  Cid : client, Pid : int
  in
    ( rec-connect(Cid?) .
      create(Client(Cid), Pid?)
    ) * delta
  endlet

process Client(Cid : client) is
  let E : str, V : int
  in
    ( rec-event(Cid, calc-conv-rate(E?)) .
      snd-msg(convert, E) .
      rec-msg(convert, E, V?) .
      snd-do(Cid, result-conv-rate(V)) .
      snd-ack-event(Cid, calc-conv-rate(E))
    ) * rec-disconnect(Cid)
  endlet

tool client is {}

process ConnectMaster is
  let Mid : master, E : str, V : int
  in
    ( rec-connect(Mid?) .
      ( rec-event(Mid, update(E, V)) .
        snd-note(update(E, V)) .
        snd-ack-event(Mid, update(E, V))
      ) * rec-disconnect(Mid)
    ) * delta
  endlet

tool master is {}

toolbus(Convert, ConnectClients, ConnectMaster)
```

The process `Convert` is responsible for starting the conversion tool as well as handling all communication with it. It begins by declaring three variables: `Cid` to hold the identification of the conversion tool, `E` to hold an expression representing a conversion request, and `V` to hold an integer value. The process then *executes* the conversion tool, receiving a unique identifier to use in all communications with the tool. Then, the process subscribes to *update*-notes.

Following the subscription is the parallel composition of two loops. The first loop is set to receive *notes* which tell the process that the conversion rates have been updated. After receiving such a note, the process tells the conversion tool to update its rates. This loop is set to repeat *forever* through the use of the `(..) * delta` construction. The ToolBus avoids inaction (`delta`) as long as there are other steps possible. The second loop first expects a *message* requesting a currency conversion. The conversion tool is then asked to *evaluate* this request. The process then *receives the value* and ultimately *sends a message* with the result of the query.

The string value given for `command` in the definition of the `convert` tool is the operating system level command needed to execute the tool. In this case, it tells the ToolBus to start the execution of the external command `currency-converter`.

The process `ConnectClients` allows multiple clients to connect, creating a dedicated delegate to handle each client's requests. It runs in an infinitely repeating loop, waiting for an external client to *connect*. It then creates a `Client` process for that particular client. Note the difference between creating an internal ToolBus process (here: `client`) and executing an external tool as done in the `Convert` process.

Each client gets its own `Client` process. This process repeats until the client disconnects. The `Client` process waits for an *event* from the client. When such an event (pertaining to a currency conversion request) occurs, a *message* is sent, which matches with the `Convert` process. After the result has been received through another message, the client is told the result of the request through a `snd-do`, and the event is acknowledged. Remember that it is a violation of the ToolBus protocol to send a second event of the same type if the previous instance has not yet been acknowledged.

As `client` tools are started externally and connect to the ToolBus (and are never executed by the ToolBus itself), the definition of the `client` tool is left empty.

The last process, called `ConnectMaster`, allows one (and only one) `master` tool to connect at a time. When such a `master` connects, the `ConnectMaster` process enters a state where it no longer accepts another connection request until the `master` has disconnected. In this state, the process accepts update-*events* from the `master` tool. It then sends out a note informing all subscribers that a currency conversion rate has changed. Then the event is acknowledged and the process is ready for another such event (or the disconnection of the `master`).

As with the `client` tools, `master` tools are started externally. Therefore their definition in the ToolBus script is left empty.

The last line shows one possible configuration of the ToolBus given the processes just described. Note that it would be possible to extend this configuration to start multiple `Convert` processes. Each of these processes would then execute its own, dedicated, conversion tool. As updates of currency conversion rates are sent through the use of *notes*, and all `Convert` processes subscribe to these notes, they will all get the information sent by a `master` tool.

Chapter 3

The Art of Debugging

3.1 General

From the Latin phrase "*errare humanum est*[1]" we learn that mankind has known for quite a while that it is not infallible. The very reason that a project such as TIDE is necessary, is because people tend to make mistakes in developing software as well. Even though many people place a firm belief in anything their computer *tells them*, up to the limit of amusing sillyness[2], they sometimes forget that their computer has been programmed by humans. Often these programmers have been under the *guidance* of a manager, which comes with its own range of peculiarities (See [1]).

The main focus of this thesis is on the development of the visualisation framework. As the need for this framework finds its roots in TIDE, which is all about debugging programs, a brief detour on the subject of debugging is in order. Without delving too deep into every possible technique, this chapter sheds some light on the sometimes dismal art of finding and removing software bugs.

Debugging a program is usually done in one of three ways:

- The erroneous source code is augmented, e.g. by inserting statements that print information about variables, or that prematurely abort the program if certain conditions are not fulfilled.
- An external debugger is used, allowing step-by-step execution of the program and monitoring of its state.
- A hybrid technique is applied using augmentation of source code in conjunction with an external debugger.

Section 3.2 explains how the technique of augmenting source code works. Section 3.4 details the use of an external debugger. Section 3.6 shows how these two techniques can be combined forming a hybrid approach to debugging.

[1]Meaning "to err is human", or for MS-DOS users: "ABORT, RETRY, FAIL?"

[2]The author distinctly remembers the answer his grandmother received from a social security employee, when asked why the payment of her social security benefit was suddenly discontinued: *Excuse me, Madam, but you have no right to be calling us. My computer tells me that you have passed away. Actually, I see on my screen that this is the second time you are trying to pull this stunt. You passed away last month as well!*

3.2 Code augmentation

The technique of augmenting source code by inserting debug statements is usually applied in an iterative manner. First an attempt is made to locate the section that is defunct, and the statements that are most likely wrong are inspected. By this time some hypothesis is formed based on the behaviour the program is displaying, combined with inspection of the source code. Then some feedback is needed to verify this hypothesis. The programmer inserts statements to print the value of variables that are used, or to abort the program when certain conditions are met. The program is then recompiled with these extra statements, and is rerun with the same parameters[3]. The values of the variables now displayed can be used to verify that indeed something is wrong. Usually the exact spot is not found in the first run. Therefore, the source code is again inspected and more print statements are inserted, or the previously inserted statements are refined to pinpoint a more specific situation. The program is then recompiled and rerun again. This procedure is iterated until the bug is found. Of course, after the bug has been found and the defunct statements have been repaired, the print statements which are now redundant have to be removed from the program.

It is relatively easy for a programmer to apply this debugging technique. Although tedious, it has the advantage that it does not require the programmer to be skilled with an external debugger. The programmer already knows how to use the programming language and how to display the contents of a variable or expression.

3.3 Code augmentation hazards

There are three important hazards to be considered when using this method of augmenting source code. The first two are related to the human tendency to make mistakes, the third is computer architecture related:

- Incorrect debug statements cause confusion.
- Overzealous removal introduces new bugs.
- Insertion of statements suppresses the bug.

3.3.1 Incorrect debug statements cause confusion

The very reason that debugging is necessary is because no human is infallible. Obviously, man's tendency to err does not cease when he is debugging. Although inserting debug statements is programming on a small scale, it is remarkably easy to insert an incorrect debug statement and spend a considerable amount of time figuring out what went wrong.

For example, suppose that a line of code meant to display a certain variable `i` is to be inserted. Unfortunately, a typographical error is made and instead the line of code (incorrectly) displays variable `j`. If variable `j` exists in the

[3] If this is at all possible. Sometimes the exact same situation cannot be recreated, e.g. when the program uses pseudo-random numbers and the initial state of the random number generator is unknown.

program, it may be perfectly legal to display its contents. When `i` and `j` are of the same type even a typechecker will not complain. Moreoever, in languages allowing *overloading*, the compiler may very well accept a variable of a different type, perhaps even silently converting the value to another data type before it is printed.

This sort of mistakes adds an extra level of difficulty to the debugging process. The programmer will now have to notice that strange values are being displayed. Then, he will have to deduce that there must be an error in the debugging code. Especially if the values displayed are not too ridiculous, but lie in the range of what is expected, one can easily be baffled by the output. Thus is quite conceivable that a programmer ends up *debugging* his debugging code.

3.3.2 Overzealous removal introduces new bugs

After the bug has been fixed, the debug statements necessary to isolate the bug have to be removed again. Sometimes a programmer, having finally found the bug, is so enthusiastic that he not only removes these debug statements, but accidentally removes one or more lines of the original program as well. Unless the statements happen to be irrelevant to the correct execution of the program (which admittedly is unlikely, but certainly not impossible), their removal most likely introduces a new bug, which does not necessarily show up immediately. The opposite occurs when a programmer *forgets* to remove all of the debugging statements he inserted. This could easily lead to, e.g., sudden abortion of the program, or to the display of unwanted information (debug output) which has little or no meaning to people other than the programmer.

This class of errors may be prevented by leaving the debug statements in place rather than removing them. The lines can be deactivated instead, e.g., by placing them between comment delimiters. Unfortunately, this tends to pollute the source code as more and more debug statements are added to the program. Therefore debug statements should ultimately be removed, perhaps at a time when the programmer is in a less ecstatic mood.

3.3.3 Insertion of statements suppresses the bug

A very annoying property of inserted debug statements is their ability to suppress the bug. In the running phase of the debugging session, the bug appears to have “magically” disappeared. This vanishing act is often caused by the fact that the contents of the program stack are subject to change when debugging statements (e.g. a call to a print function) are inserted.

Another cause of bug suppression through statement insertion is memory architecture related. In the runtime memory environment of a program, the data segment is usually placed after the program segment. The insertion of even a single statement could cause expansion of the program segment. The data segment is then moved up, thus beginning at a different memory address. If the bug is related to a specific class of memory addresses or their runtime contents, and the address of the variable producing undesired results is not in that class after the debug statements have been inserted, the bug may not occur at all.

In either case, removal of the inserted debug statements usually causes the bug to return, but little information is gained other than that the nature of the bug may be stack or memory address related.

3.4 External debuggers

The alternative to augmenting source code is the use of an external debugger. This Section introduces the notion of an external debugger by showing two different types. One is a *dedicated debugger*, which offers debugging support for a limited set of programming languages. The other can be called a *language independent debugger*, as it does not have this limitation, but rather allows debuggers for different programming languages to be adapted and connected to the debugging system. A more ambitious approach to *generate* source-level debugging tools from the specification of an interpreter for a given language can be found in [22].

3.4.1 Using a dedicated debugger

An external debugger can be a dedicated, single language debugger, or it can be a language independent debugging system. An example of a dedicated debugger can be found in the debugger integrated in Borland's Visual `C++` IDE. The "GNU debugger" `gdb` is slightly more than a single language debugger. According to [16], it can handle programs written in `C` and `C++`, with partial support available for Modula-2 and Chill as well. Some support for Pascal and Fortran is also present, although this support is perhaps best termed *limping*, since many important `gdb` features (e.g. entering expressions and printing values using the language's native syntax) are not supported.

External debuggers (such as `gdb`) often offer functionality which is not readily available to the programmer. Most notable are the ability to suspend program execution at certain predefined points (*breakpoints*), or whenever the value of a variable or expression changes (*watchpoints*). Also, the ability to *step* through the program, inspecting internal data structures as the execution goes along often yields more insight than seeing the program *run* and crash, or terminate with the wrong answer.

3.4.2 Using a language independent debugger

An example of a language independent debugger can be found in [13]. The "ToolBus Integrated Debugging Environment" described therein presents a single graphical user interface (GUI) to the user, regardless of the actual debugger employed for the low level debugging steps. For each language TIDE is to support, a debugger has to be found and a debug adapter has to be written so the debugger can be connected to TIDE.

At the time of writing of this thesis, TIDE already offered support for Java, `C`, Tcl/Tk, ASF+SDF, and ToolBus-scripts. For the `C` language, a `gdb`-adapter was written. For Java an adapter was created to connect SUN's JDPA[4] [21]. As the ToolBus and ASF+SDF projects were developed in our own research group, no "off-the-shelf, ready-to-use" debuggers were available, meaning dedicated debugging components had to be written. Finally, debugging support for Tcl/Tk

[4] Java Debugger Platform Architecture

was acquired by *on the fly* modification of the script to be debugged. This (automated) modification consists of padding each Tcl/Tk instruction with extra instructions, relaying control of execution to TIDE. This way *breakpoints* and *watchpoints* are simulated.

Using all these different techniques to subjugate particular debuggers to TIDE hints at both a strength and a weakness of the TIDE philosophy:

pro The strength of TIDE obviously lies in the versatility offered, allowing many different debuggers to be used at the cost of learning a single GUI. Virtually any debugger can be attached to TIDE using the approach most suitable to that particular debugger.

con In general, it is a non-trivial task to get a debugger adapted in such a way that most of its features are exploited to optimally implement all of TIDE's requirements. A profound knowledge of the debugger to be adapted as well as insight in TIDE's requirements is necessary. Even then, not all features might fit in the TIDE framework.

3.5 External debugger hazards

Using an external debugger, whether *dedicated* or *language independent*, comes with its own set of peculiarities. The following issues are discussed.

- Overhead involved in setting up the debugger.
- Optimisation and debugging are often mutually exclusive.
- Using the (external) debugger may implicitly suppress the bug.

3.5.1 Overhead involved in using a debugger

Each debugger usually has its own set of instructions which the user must master before being able to effectively benefit from the features the debugger offers. Some debuggers are commandline oriented, meaning the user must learn a set of commands to operate it. Other debuggers provide a graphical user interface, which may be more intuitive to comprehend, but which requires intensive mouse motion and point-and-click work.

Even after the programmer has mastered the debugger, there is still more work to be done before the debugger can be used. Compilers often require different invocation parameters to propagate debugging information. For example, if not explicitly asked to propagate line number information, the compiler usually throws it away. Therefore, the programmer must often recompile his program in a specific way to satisfy the demands of the debugger.

A major drawback to using external debuggers is that for each different language there is another debugger to master. Sometimes, even for the same language, different debuggers exist on different platforms. The Solaris Unix platform offers two debuggers for the `C` language alone: GNU's `gdb`, and `dbx` by SUN MicroSystems. Switching to the Windows platform, there are at least `Borland Visual C++` and `Microsoft Visual C++` to consider. For different programming languages a programmer usually has to get familiar with several debuggers. Although TIDE only presents a solution for the Unix platform at this moment, the concept of having a single debugger with one GUI, capable of debugging several different programming languages, feels like a fresh breeze on a hot summer's day indeed. This is especially true when one considers that more and more monolithic systems can be broken up using the ToolBus, with each component being written in the programming language most appropriate to solving the problem the component was designed for.

3.5.2 Optimisation and debugging are mutually exclusive

When a compiler is asked to optimise the instructions it generates while translating a program, it can be quite a challenge to find the relationship between the the original source code and the resulting output of the compiler. Moreover, many compilers use architecture specific *tricks* to move instructions around to come to a more efficient sequence. It is quite conceivable that after juggling instructions around to obtain an optimised version for a specific architecture, an execution path is created which differs from the one created when the compiler is asked to generate regular or even debug specific code. Sometimes, this makes it very hard indeed to track down a bug which only occurs in the optimised version, while being forced to use the debugger on the debug version of the compiled code.

3.5.3 Implicit suppression of the bug

Recall from Section 3.3.3 that insertion of debug statements may move up the data-segment, causing suppression of certain types of memory address related bugs. A compiler, when instructed to produce debug specific code, can suppress the bug in much the same way. As extra instructions needed for the debugger are inserted by the compiler, the program segment expands resulting in relocation of the data segment. If the bug only occurs when some variable is at a specific address, using the debugger may not prove to be of much help.

3.6 The hybrid approach

The techniques of augmenting source code and using an external debugger can be integrated to form a hybrid approach.

Inflexibilities in the debugger can sometimes be avoided by making a minor modification to the source code. For example, the `C` language allows the use of macros to avoid writing complex expressions over and over again. As these macro's are expanded by a pre-processor run before the actual compiler is invoked, these macro's no longer exist and will be inaccessible from the debugger. If the result of a macro expression is needed in the debugging session, the macro has to be looked up in the source code and meticulously copied into the debugger. An easy way to avoid such error-prone labour is to explicitly assign the result of a macro expression to a variable, as a debugger is often able to display values of variables.

Conversely, it may be convenient to use the debugger to halt execution of the program at a specific point. Continuing in step-by-step mode, the debugger can help scrutinise a section of code, something which is much more difficult to emulate by merely modifying the source code.

3.7 Summary and conclusions

This chapter explained three approaches to debugging programs. Two of these are distinct techniques: *augmenting source code* and using an *external debugger*, the third approach is a combination of these techniques. Some advantages and disadvantages associated with these techniques were shown: *augmentation* is easy to master, but actually changing the source code is dangerous; using a *debugger* requires the programmer to get passed an initial learning curve, which can be tedious as each language has (at least) its own debugger. In addition, both techniques allow for at least one hazard: changing the code, either manually or by means of having the compiler generate debug specific code, may suppress the bug the programmer is looking for.

From the previous Sections we draw the following conclusions:

- Neither the technique of augmenting the source code, nor using an external debugger solves all the hazards involved in debugging a program.
- An external debugger offers debugging functionality which cannot easily be mimicked by merely altering the source code.
- A language independent debugging system such as TIDE relieves the programmer of some of the burdens associated with using external debuggers. Only a single system needs to be mastered to be able to debug several different programming languages.

Chapter 4

Design of the Framework

This chapter discusses the design of the visualisation framework. Some general design issues are discussed in Section 4.1. Section 4.2 describes how the visualisation framework can be broken down into two distinct subcomponents. Given this breakdown, Section 4.3 discusses some requirements. Section 4.4 introduces the instructions needed for communication between the subcomponents. A summary is of this chapter is given in Section 4.5.

4.1 General

The acronym TIDE stands for "ToolBus Integrated Debugging Environment". In TIDE, several components, such as a source code viewer, a process viewer and several debug adapters can be attached to the ToolBus, depending on which components the user has activated and which program is being debugged. It naturally follows from the ToolBus philosophy (Section 2.2) that for the framework to be a success in TIDE, its implementation in the form of one or more ToolBus-components is a *must*. This will allow for easy replacement of the components in the event that better implementations are built. Implementing the framework as a set of ToolBus-components has the additional advantage that it can be used in other ToolBus projects, that also require visualisation of expressions.

4.2 Breaking down the framework

In this Section we break down the framework into subcomponents. This is done by recognising that visualising an expression can be done in two distinct steps: a visualisation step (Section 4.2.1) and a painting step (Section 4.2.2).

4.2.1 The visualisation step

First, we need to know *how* an expression is to be visualised. That is, we need some sort of *template* explaining what to do, given a specific value of the expression. Consider for example the visualisation of a boolean valued expression. One possible visualisation *template* could specify that depending on the value of the expression either the word `true` or the word `false` should be printed, perhaps even adding some colour by rendering `true` in red and `false` in green. Another template for boolean expressions could display a "thumbs-up" picture when the expression evaluates to `true`, and a "thumbs-down" picture when it is `false`. Neither of these visualisation templates does any *actual* drawing, they merely specify *how* the expression should be visualised. This component has no knowledge about the actual execution of the *animation commands* it is issuing. It only receives values for the expression it is visualising and emits animation commands according to a given template. We hereby baptise the component responsible for specifying how an expression should be visualised the VISUALISER. At this stage, the semantics of the data are converted into instructions to draw specific shapes or display specific pictures on the animation canvas. The user should be able to define visualisation rules for new data types as well as change the way existing data types are displayed. Depending on how radical these changes are, implementing them may be as simple as reconfiguring the colour at runtime or as intricate as rewriting (part of) the code that generates the visualisation commands.

4.2.2 The painting step

Second, the animation commands issued by the VISUALISER, have to be made visible somehow. Therefore, we need a component capable of executing the animation commands. As this component is responsible for the actual "painting" it is termed the PAINTER. Obviously, the PAINTER should follow the semantics implied by the VISUALISER. That is to say, it should not, e.g., print the string `false` in green when ordered to print the string `true` in red. Thus, the PAINTER is subordinate to the VISUALISER's orders. The graphical engine used in the implementation of the PAINTER should run on multiple platforms, or its usefulness will be limited to a single architecture.

4.2.3 Connecting the steps

The communication between the VISUALISER and the PAINTER, as well as the communication between these components and the "outside world" is handled by the ToolBus. Figure 4.1 shows a setup of the ToolBus, the VISUALISER, and the PAINTER. A number of other components, marked $\text{tool}_1 \ldots \text{tool}_n$ in the figure, are also attached, depicting the rest of the world. These could be a connection to TIDE, but the visualisation components can also be used in another ToolBus-context.

As can be seen in Figure 4.1, the VISUALISER is not directly connected to the PAINTER. Instead, it behaves like any ordinary ToolBus-tool, communicating with the PAINTER over the ToolBus. Separating the issuing of animation commands from their execution (e.g. by drawing on a display medium), introduces a level of abstraction. This abstraction allows other components to *catch* the drawing instructions and perform actions which the VISUALISER has no knowledge of. For example, a *logging* component could be constructed which does not actually execute the instructions by drawing on a canvas, but instead writes the instructions to a file. By reading this file at a later time, the visualisation can be repeated. Instead of the VISUALISER issuing the drawing instructions, some other component reads the instructions from file and feeds them to the PAINTER, again through communication over the ToolBus.

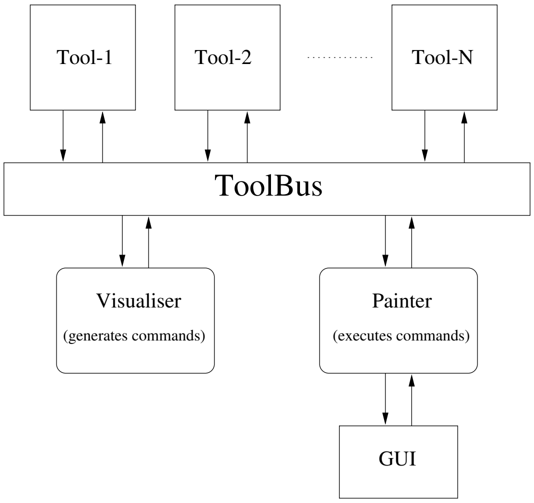


Figure 4.1: The Visualiser and Painter components.

4.3 Requirements issues

With the breakdown of the visualisation framework into a VISUALISER and PAINTER subcomponent in mind, we briefly discuss the following requirements. Section 4.3.1 details on coordinate conversions in visualisations. Section 4.3.2 introduces the notion of colour configuration by users of the visualisation framework.

4.3.1 Coordinate conversions

To facilitate the use of many different visualisation templates, users who program such a template should not be forced to use a rigid set of coordinates. Ideally, each animation should have access to its own set of coordinates. The PAINTER decides how to execute the visualisation commands, e.g. open a new window for each visualisation, or draw multiple visualisations on the same canvas. This means that somewhere a conversion must be made from the set of coordinates defined by the user for that particular template, to the set of coordinates that are eventually used to draw the animation on the canvas. The visualisation framework should take care of the coordinate conversion, relieving the user of this burden.

4.3.2 Colour configuration

As we all know, there is no accounting for tastes. If a visualisation template is designed that draws a pink rectangle, you can rest assured that soon after the template is set to use, someone demands that the rectangle be coloured purple instead. In an effort to avoid user complaints about the colours used in a visualisation, we require the visualisation framework to allow reconfiguration of colours. This prevents “hard wiring” of specific colours in the visualisation templates. Rather, the template should specify *default* colours. These colours should still "make sense", so that users are not forced to setup their own colours.

4.4 Visualisation instructions

This section discusses the basic set of instructions needed for communication between the VISUALISER and the PAINTER. Section 4.4.1 introduces the notion of atomic animation elements. Section 4.4.2 details on the visualisation canvas and its instructions.

4.4.1 Atomic animation elements

An *atomic animation element* (or *anim-atom* for short) is a part of a visualisation. Examples include a *line*, a *rectangle*, and an *arc*. The animation element is termed *atomic* because it cannot be broken down into smaller parts.

This section lists a set of animation atoms that should be supported by the framework, along with their specific properties. The set of properties for any of the atoms given here, is just *one* way to represent them. The properties listed for a particular anim-atom are in no way claimed to be superior to any other representation for the same atom.

For example, we introduce a *rectangle* by giving the coordinates of its top left corner and its width and height. Another representation could specify its bottom right corner instead of the width and height of the rectangle. For a *line* segment, the (r, ϕ) notation could be used instead of giving the segment's end coordinates as done here.

line A *line* segment as shown in Figure 4.2(a).

- (x_1, y_1) the start location of the line.
- (x_2, y_2) the end coordinates of the line.

rectangle A *rectangle* as shown in Figure 4.2(b).

- (x, y) the location of its upper left corner.
- (w, h) the width and height of the rectangle.

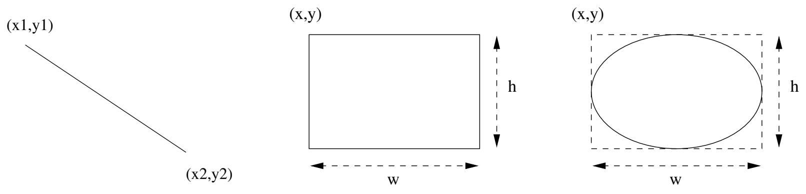
ellipse An *ellipse* as shown in Figure 4.2(c).

- (x, y) the location of the upper left corner of the bounding rectangle.
- (w, h) the width and height of the bounding rectangle.

The ellipse is defined by its bounding rectangle. Let (c_1, c_2) be the center of the ellipse. The points on the ellipse are given by the formula $(x, y) = (c_1 + \frac{1}{2}w\cos\phi, c_2 + \frac{1}{2}h\sin\phi)$, where ϕ is an angle in radians modulo 2π. A circle can be made by taking $w = h$.

arc An *arc* as shown in Figure 4.3.

- (x, y) the location of the upper left corner of the bounding rectangle.
- (w, h) the width and height of the bounding rectangle.

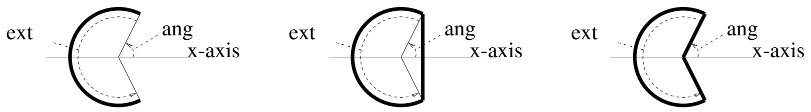


(a) Line (b) Rectangle (c) Ellipse

Figure 4.2: Examples of the *line*, *rectangle* and *ellipse* atoms.

- *ang* the start angle.
- *ext* the angular extent (or length of the arc).
- *type* the *closure* type (`open`, `chord`, or `pie`).

An arc is a partial section of an ellipse (defined by its bounding rectangle). The angles are specified in such a way that 45 degrees falls on the line from the center of the ellipse to the upper right corner of the bounding rectangle. The examples in Figure 4.3 were created with $w = h$, thus creating an arc that is part of a circle. This is no restriction however, and ellipsoid arcs can be constructed just as well. The arc's *closure* type dictates how the end points of the arc are connected. Closure can be one of *open* (Figure 4.3(a)), *chord* (Figure 4.3(b)), or *pie* (Figure 4.3(c)).



(a) Arc (open) (b) Arc (chord) (c) Arc (pie)

Figure 4.3: Examples of an *open*, *chord* and *pie* arc.

text represents a *string* of characters defined by

- (x, y) the location of the leftmost character in the string
- *str* the string of characters

Even though a character string may be thought of as consisting of smaller parts (the characters that make up the string), for purposes of the visualisation the string is considered atomic.

image can be any image, e.g. a GIF or JPEG picture. The example in Figure 4.4 shows a smiling face. Although this *smiley* can be seen as consisting of four subcomponents (a circle for its face, two dots for its eyes, and an arc for its smile), it is in fact *not* made up of any subcomponents. The entire picture is *one* (atomic) image.

- (x, y) the location of the top left corner of the image.
- *data* any (possibly binary) data making up the picture.

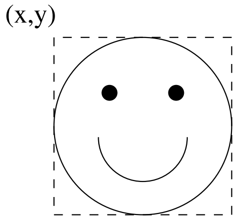


Figure 4.4: Example of an *image* atom.

Note that each atom has a (x, y) coordinate (called (x_1, y_1) in the *line* case to avoid confusion) that acts as its "hotspot". When the atom is moved, this (x, y) location moves to the new spot. The rest of the atom moves according to its relative position to this location.

Several instructions exist in the framework that have effect on the appearance of an *anim-atom*. These instructions deal with the atom's visibility, its position, its colour, and properties which differ depending on the type of the atom. After all, the *angle extent* property, which is present in an arc, makes no sense when drawing a rectangle.

set-colour sets the *anim-atom*'s colour to one of the previously registered colours.

set-location moves the *anim-atom* to a new location on the canvas.

set-solid sets the *solidity* of an *anim-atom*. For example, a *solid* circle appears "filled", as opposed to just having its border drawn.

set-visible sets the *visibility* of an *anim-atom*. A *visible anim-atom* appears on the canvas, an *invisible anim-atom* is (temporarily) hidden from view.

set-property sets a property specific to the type of the *anim-atom*.

4.4.2 The visualisation canvas

The VISUALISER should be able to allocate what we will refer to as a *canvas*. A canvas is a portion of "drawing space", that can be used for a visualisation. If the "painter" does not paint at all, but e.g. merely logs all instructions to file, the implementation of the allocation of a *canvas* may amount to opening a dedicated file to store the instructions in. If the painter *does* execute the instructions by actually drawing, it may do so on a portion of an existing window, or by creating a new window, dedicated to this particular visualisation.

The following instructions specific to working with a canvas are listed here.

create-canvas creates a new canvas for use in a visualisation.

range-x sets the range of X-coordinates (see Section 4.3.1).

range-y sets the range of Y-coordinates (see Section 4.3.1).

`reg-colour` register a colour with the canvas for later use. Instead of directly stating the colour of each animation element, a level of indirection is used. First all colours to be used are registered with a key. Later, the colour of an animation element can be set using this key. This simplifies the introduction of the colour-override mechanism as explained in Section 4.3.2.

`repaint` request the canvas to repaint all its (visible) *anim-atoms* after a series of *anim-atom instructions* have been issued.

`create-atom` request the creation of a new animation element on the canvas.

4.5 Summary

This chapter discussed the design and some requirements of the visualisation framework. We pointed out the need for the framework to consist of ToolBus-components (Section 4.1). Then the visualisation process was divided into two steps and the suggestion was made that a separate ToolBus-component be implemented for each step. The components responsible for these steps are the VISUALISER for issuing animation commands, and the PAINTER for the actual execution of these commands. The issues the issues of coordinate conversion and colour configuration were discussed in Sections 4.3.1 and 4.3.2, respectively. Finally, we presented visualisation instructions (Section 4.4) necessary for communication between these two components, introducing the notion of a *canvas* (Section 4.4.2) whereupon *anim-atoms* (Section 4.4.1) are drawn.

Chapter 5

Implementation of the Framework

5.1 General

This chapter describes the implementation of the Visualisation Framework. Section 4.2 discussed the breakdown of the framework into a visualisation and a painting step. The implementation described in this chapter adheres to the proposed subdivision by presenting two ToolBus components. Section 5.2 presents the implementation of the VISUALISER, Section 5.3 discusses the implementation of the PAINTER.

5.2 Implementation of the Visualiser

Recall from Section 4.2.1 that the VISUALISER is responsible for generating the visualisation instructions. The following distinct communications between ToolBus and VISUALISER are implemented. Note that the first three communications are sent by the ToolBus to the VISUALISER, whereas the last one travels in the opposite direction.

- A request from the ToolBus to the VISUALISER to start a visualisation of an expression.
- A request from the ToolBus to the VISUALISER to update the value of an expression currently being visualised.
- A request from the ToolBus to the VISUALISER for a list of all supported visualisation templates.
- A visualisation instruction from the VISUALISER to the ToolBus to be executed by the appropriate PAINTER.

Before discussing the implementation of these communications, we first explain how to uniquely identify elements of these communications, and select an implementation language for the VISUALISER.

5.2.1 Unique identification of expressions and instructions

The VISUALISER is able to handle multiple visualisations of multiple expressions. Visualisation instructions subsequently issued by the VISUALISER might all be executed by the same PAINTER, but they might just as well be delegated to different components. Therefore, unique keys are needed for the following issues.

First, each request to *visualise an expression* must contain a unique key in order to associate subsequent *updates* of the value of this *expression* with the correct visualisation.

Second, as the execution of visualisation instructions is delegated, e.g. to a PAINTER, it is reasonable for this delegate to expect incoming *instructions* issued by the VISUALISER to be uniquely identifiable as well.

The abstraction between the VISUALISER and the PAINTER described in Section 4.2 makes it a poor choice to use the same key for both identification issues. Instead, an *expression-key* will be used to uniquely identify updates of an expression, and an *instruction-key* to uniquely identify the destination of visualisation instructions.

5.2.2 Choosing a language for the Visualiser

The VISUALISER component is responsible for the bookkeeping of visualisations of several expressions. This means that it has to perform administrative tasks for each visualisation it is running. Every time the value of one of its visualisations is updated, it needs to send out the correct animation commands. These requirements mark the VISUALISER as a highly administrative component[1]. Given the set of contemporary programming languages (including e.g. `C` [11], `C++` [19], Tcl/Tk [14] and Java [9]), a language that is easy to use for bookkeeping is to be preferred. The Java programming language is a language that fits these demands. Not only is Java an object oriented language, its development kit (JDK) comes with extensive support for managing lists, trees, sets and the like. This relieves us of the burden of having to write our own data structures even for simple things such as a list. Another more pragmatic reason is the fact that TIDE was written in Java as well. Although it is not a necessity that TIDE and the VISUALISER be written in the same language, doing so does simplify the incorporation of the VISUALISER into TIDE.

5.2.3 Starting a new visualisation

Now that we are able to uniquely identify the necessary elements in the communication between ToolBus and VISUALISER, we concentrate on the implementation of these communications. The first communication we describe is a request from the ToolBus to begin visualising an expression. This request has the following parameters:

- A unique *expression-key* used to pass updates of the value of the expression from ToolBus to VISUALISER.
- A unique *instruction-key* used to pass painting instructions from VISUALISER to ToolBus.

[1] As opposed to, for example, the PAINTER, which is a highly graphical component.

- The *expression* to be visualised.

- A description, or *template*, telling the VISUALISER how to visualise the expression, e.g. *counter* to indicate the expression represents values of a counter.

The VISUALISER first performs a sanity check, enforcing that it is not already running a visualisation for the supplied *expression-key*, thus detecting duplicate keys as early as possible.

Then, the given *template* is checked against all known templates to find out how the expression should be visualised. An exception is raised if the *template* is unknown to the VISUALISER. Otherwise an instance of this template is created which will handle the visualisation.

Finally, the internal administration of the VISUALISER is updated, associating the instance of the template with the *expression-key*.

5.2.4 Updating the value of an expression

Whenever the value of an expression changes, the ToolBus-script notifies the VISUALISER by passing the *expression-key* and a representation of the new value of the expression. This representation can be either of:

- `value(<val>)`, with `<val>` a string representation of the value of the expression, e.g. "*42*"

- `error(<msg>)`, with `<msg>` a string representation of the error that occurred while evaluating the expression, e.g. "*variable i is out of scope*".

First, the VISUALISER checks its administration to see if there is indeed an animator associated with the given key. If this is not the case, the VISUALISER denies the request and raises an exception. Otherwise, the new value is inspected to see whether it encapsulates a legal value for the expression, or indicates that an error occurred during evaluation of the expression. Depending on the outcome, visualisation instructions are generated to either reflect the new value of the expression, or to display the error message. If the representation matches neither the `value` nor the `error` pattern something is definitely askew and the VISUALISER raises an exception notifying the user of the bogus request.

5.2.5 Requesting a list of all possible templates

The VISUALISER can display the value of an expression according to different templates. It makes little sense, for example, to display the items of a list using a template suited only to display a counter. It is useful if a ToolBus-component can obtain a list of all possible templates, so that it can present this list to its user before requesting the visualisation of an expression.

The implementation compiles all known templates into a list and consequently sends this list as the result of the query.

5.2.6 Sending a visualisation instruction

The exact nature of the visualisation instructions that are generated when a new visualisation is created and during updates of its expression depend heavily on the template used to visualise the expression. For example, simply displaying the value of an expression as a string generates instructions which differ significantly from those required to draw a fancy animation. Each instruction, or animation command, is encapsulated in a ToolBus-term, together with the *instruction-key* so that on the ToolBus level, each animation command is of the form `anim-cmd(<term>,<term>)`. For example, the instruction to tell a painter that the horizontal scale to be used ranges $0 \ldots 100$ is: `range-x(0,100)`. When the VISUALISER issues this command to a painter with key P, it is sent to the ToolBus as: `anim-cmd(`P`,range-x(0,100))`. Encapsulating animation commands in a pattern such as `anim-cmd(...)` allows for easy matching in the ToolBus-script, and gives access to both the key needed to address the corresponding PAINTER and the animation command itself.

5.2.7 Class hierarchy of the Visualiser

For the interested reader who is familiar with Java's *interfaces*, and (*abstract*) classes, the class hierarchy of the VISUALISER is shown in Figure 5.1. The figure shows three examples of a visualisation template, called GenericAnimator, ProgressAnimator and PieAnimator. These example templates are detailed in Chapter 6.

The middle portion of figure 5.1 shows how a PieAnimator generates AnimCommands which are handled by an AnimCommandHandler. This route is also present in the other templates, but was omitted from the figure, to keep it legible.

5.3 Implementation of the Painter

Recall from Section 4.2.2 that the PAINTER is responsible for the actual execution of the visualisation instructions. A practical implementation employs some sort of graphical engine, executing the instructions by drawing on a *canvas*.

Whereas the VISUALISER was a rather unique component, several graphical engines already exist which could possibly be re-used (partially) for the PAINTER.

5.3.1 Alternative visualisation packages

This section takes lists some of the available graphical engines, keeping in mind that (some of) the functionality they offer might be re-used for the PAINTER. In particular, two closely related animation designer's packages (POLKA and SAMBA) will be discussed. The scripting language Tcl/Tk and the Java Foundation Classes (JFC) are discussed as well. Finally, a motivation to chose the JFC for the implementation of the PAINTER is given.

The POLKA animation designer's package

From the POLKA manual [17] we learn that it

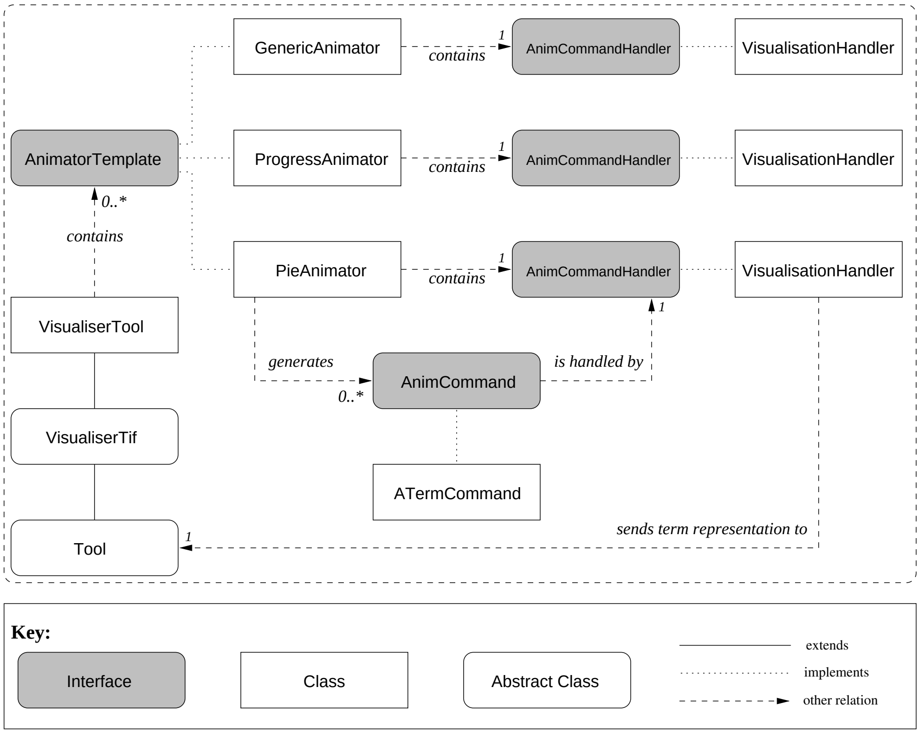


Figure 5.1: The Java class hierarchy of the VISUALISER.

> ... implements a structured graphics and animation design toolkit in `C++`. With it, you can create colour, smooth, $2\frac{1}{2}$-dimensional animations on top of the X11 Window System. It is particularly good for creating algorithm animations.

The remarkable $\frac{1}{2}$ dimension constitutes a notion of depth in between 2D and 3D. POLKA does not have 3D modelling primitives. In fact, animation objects only have an (x, y) coordinate pair, they have no z-coordinate. Still, it is possible for one animation object to lie *on top of* or *beneath* another.

As the manual states, POLKA is particularly good for creating algorithm animations. As POLKA is a `C++` library, any animation that wishes to use POLKA's functionality, has to be written in `C++`.

A feasible implementation of the PAINTER using POLKA would be to write a special ToolBus component in `C++`, which exports POLKA's functionality to the ToolBus. To avoid any accusation of a possible bias towards the Unix platform, we mention straight away that any implementation of the PAINTER using POLKA will *only* run on the X11 Window System, thus excluding, e.g., PCs using any of Microsoft's Operating Systems. Taking into account however that at the time of writing of this thesis, the ToolBus does not yet work under Windows, but rather requires PCs to run some sort of Unix anyway, using POLKA does not impose any additional constraints. As it is still conceivable that a ToolBus implementation for Windows-based computers will emerge, we should keep in mind that at that time using POLKA *will* pose a limitation on the platform independance of the visualisation framework.

The SAMBA animation designer's package

A disadvantage of POLKA is that you have to write all your animation code in a specific language (`C++`). To make the functionality in POLKA available to a more widespread field of languages, SAMBA was created. Its manual tells us that SAMBA

> ... provides an interpreted, interactive animation front-end to POLKA. Samba simply reads an ASCII file, one command per line, in order to acquire its directions for creating an animation. This is beneficial because you can have the output of any program, be it Pascal, C, Modula-2, etc., drive an animation.

In the early development stages of the framework, this ASCII front-end has been used to create a `samba-adapter`. Although SAMBA was created to read instructions from an ASCII file, the Unix OS makes no particular distinction between reading from an existing file or from redirected output. This enabled us to *cheat* SAMBA by "feeding" it instructions directly from the `samba-adapter`, rather than from a physically present file.

For this *trick* to work, we were forced to temporarily suppress one feature SAMBA offers: repeating an animation from start. This is because it is easy to re-read an existing file, but if an adapter is to repeat an animation, it would have to remember all instructions it sent to SAMBA during a session. Although it would certainly be possible to extend the adapter with this functionality, it was declared out of scope during the case study to see if SAMBA could be used for the PAINTER.

Unfortunately, it turned out that several of the atomic animation elements offered by POLKA were *not* supported in SAMBA. For example, the notion of a *pie* (part of a circle) which is present in POLKA, is found absent in SAMBA. Upon browsing through the source code of POLKA and SAMBA to find out if this error was easy to fix, it turned out that several of the basics "deep" inside the core of POLKA were implemented insufficiently versatile to later build a more generic interpreter on this core. As POLKA and SAMBA are third party software, no time or effort was invested in digging deep into the source code, trying to fix the problem. It is possible that future releases of POLKA and SAMBA do not have this problem, at which time it might be worthwhile to have another look at these packages, as some of the example animations that were shipped with them looked very promising indeed. But rather than wait for new releases, we decided not to use POLKA or SAMBA, preventing inheritance of their software maintenance problems. The development of the `samba-adapter` was aborted at this time.

Tcl and the Tk ToolKit

A popular contemporary scripting language is Tcl, which stands for "tool command language" and is pronounced "tickle". Quoting a portion of the FAQ's definition we learn that

> ... Tcl is a simple textual language, intended primarily for issuing commands to interactive programs such as text editors, debuggers, illustrators, and shells. It has a simple syntax and is also programmable, so Tcl users can write command procedures to provide more powerful commands than those in the built-in set.

Tcl itself has no graphical primitives, but Tcl comes with Tk, a toolkit that allows programmers to create graphical user interfaces by writing Tcl scripts. Tk has quite a number of graphical primitives and could easily support the anim-atoms we need in this thesis. A description of Tcl/Tk can be found in [14].

Tcl/Tk has been used in a previous approach to TIDE but was deemed too slow and inefficient at the time, which resulted in the reimplementation of TIDE in Java. Tcl has evolved and nowadays compilers are available that compile Tcl/Tk into bytecode, which has a positive effect on the performance of Tcl/Tk scripts. But, as Java fits TIDE's needs well enough, migrating back from Java to Tcl/Tk is not necessary. Tcl/Tk is certainly powerful enough and should definitely be considered as an option for an implementation of the PAINTER.

The Java Foundation Classes

The Java Development Kit 1.2 (later termed *Java 2* by SUN Microsystems) comes with many features. Amongst them are the *Java Foundation Classes* (JFC). Two components of the JFC that look interesting for the PAINTER are the *Swing Components* and *2D Graphics and Imaging.* Browsing through the on-line documentation offered by SUN Microsystems, we find that

> Swing is the part of the Java Foundation Classes (JFC) that implements a new set of GUI components with a pluggable look and feel. Swing is implemented in 100% Pure Java, and is based on the JDK 1.1 Lightweight UI Framework. The pluggable look and feel lets you design a single set of GUI components that can automatically have the look and feel of any OS platform (Windows, Solaris, Macintosh).

SUN's summary on Java's Graphics2D class also sounds encouraging:

> This Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-dimensional shapes, text and images on the Java platform.

As both the core of TIDE and the VISUALISER were already written in Java, at least some practical knowledge of the Java programming language was available. Moreover, the Java programming language is a relatively new language, and as such it has been the topic of several research efforts in our Programming Research Group ([5, 6]). One of the advantages of using Java over Tcl/Tk would be the fact that we could perform the visualisations *inside* the TIDE environment, rather than opening yet another window on the user's desktop. Java's suitability, along with some of these more pragmatic concerns helped make the decision to implement the PAINTER using Java and the JFC.

5.3.2 PainterTool.java

The first Java class we explain is PainterTool. A copy of the source code can be found in Appendix C. This copy has been modified to increase readability of the source code in this thesis. As a result, the layout of the code may not be consistent with the Java Coding Standard [20].

The PainterTool is responsible for handling all communication with the ToolBus. We use a separate Java Thread to *run* instances of this tool so that it does not need the main program thread. This simplifies the incorporation of the PainterTool in existing projects, such as TIDE which already have other classes that need to run in the main thread. To further this simplification, the PainterTool uses a given instance of Java's JDesktop instead of creating its own. This allows us, for example, to re-use the desktop used in TIDE.

Creating a new canvas

Upon receiving a request to create a new canvas, the PainterTool creates a unique identification number for the new canvas. We use a Java integer for this purpose, starting at zero and incrementing it each time a new identification is requested. We use this unique identification in further ToolBus-communication, to pinpoint the correct canvas.

We then create a PainterFrame which is an extension of Java's JInternalFrame. This extension consists of setting the frame's title, specifying its sizes and listening to user requests to close it. Given this frame, we can create an instance of the class AnimCanvas, which will perform the actual drawing inside this frame. We associate the unique canvas identification numbers we created with their canvases by putting them in a Java Map. In particular, we use a HashMap for this administration, ensuring an efficient one-to-one correspondence between the identification and the canvas. As identification numbers can only be used once in the lifetime of an instance of the PainterTool, using the identification numbers as direct indices in an array is not a feasible solution. Removing a canvas would result in creating a *gap* in the array which can never be re-used.

Finally, we make the frame visible on the desktop and pass the unique identification number to the ToolBus, as part of the result of the canvas creation request.

Executing animation commands

A request to execute an animation command is parameterised by the *identification number* of the canvas that needs to execute the command and the *command* itself. First, we lookup the identification number in our administration to find the canvas which we will delegate the command to. If no such canvas is found, we throw an Exception to notify the user.

We then match the command against all term patterns that make up an animation command. When a match occurs, we know which command to execute, as well as having extracted the necessary parameters for this particular command. We delegate the command by invoking the corresponding method in the canvas. At this point, the ToolBus specific representation of the animation command is removed, so the implementation of the canvas need not know that it is somehow connected to a ToolBus.

If, after trying all term patterns that make up an animation command, we still have not found a match, an Exception is thrown to indicate we received an unknown command.

Receiving a termination request

We handle the `rec-terminate` request from the ToolBus by stopping the thread that runs this PainterTool instance, not by an explicit invocation of the `exit` method in Java's System class because we have our own thread driving the PainterTool. Other classes can wait for our thread to finish, by calling our `join` method, which is Java's way to gracefully wait for a thread to finish.

5.3.3 AnimCanvas.java

A copy of the source code can be found in Appendix D. As with the PainterTool (5.3.2), this copy has been modified to increase readability in this thesis. The AnimCanvas class is extended from Java's JComponent, the base class from Swing Components. It is created inside a JInternalFrame and its job is to draw the anim-atoms that make up a visualisation.

The internal coordinate space used in an instance of AnimCanvas defaults to a range of $0 \ldots 100$ in both x and y direction. Of course, this coordinate space can be changed according to the requirement in Section 4.3.1 as we will explain below when discussing the `paintChildren` method of the AnimCanvas class.

The AnimCanvas uses several *rendering hints* supported by Java's Graphics2D class. We use the *anti-aliasing* option, both for drawing and for text, and also use prefer rendering quality over rendering speed. These choices influence performance in a negative way, preferring better *looks* over higher *speed*.

We surround the canvas on which we will be drawing by an instance of Java's JScrollPane ensuring that even drawings that would fall out of scope of the size of the canvas can be seen. The JScrollPane class conveniently introduces scrollbars in both horizontal and vertical direction as soon as they are needed. That is to say, if the drawing *fits* there are no scrollbars, but if the drawing is too large to be displayed in either direction, appropriate scrollbars pop up and the user can scroll over the image.

The AnimCanvas class also allows users to expand or condense the canvas in both x and y direction. Thus, the drawing can be stretched or squeezed together horizontally or vertically for optimal viewing results.

Creating a new anim-atom

A Java *interface* (called AnimAtom) was written, defining the functionality that should be present in all anim-atoms. Then, an abstract implementation of this interface was made (AbstractAtom). All anim-atoms were then implemented by extending this AbstractAtom class with specific code for that particular anim-atom.

When asked by the PainterTool to add an anim-atom to the canvas, an instance of the corresponding atom class is created. For example, when asked to add a line to the canvas, an instance of LineAtom is created. Again, a HashMap is used to maintain the bookkeeping of all anim-atoms created on a canvas. The key used to uniquely identify these atoms is the one that was passed by the PainterTool (which in turn got it by extracting it from a ToolBus-term).

Changing an anim-atom

Given a key to an anim-atom, several of its attributes can be changed. A separate method exists to change an atom's *location*, *colour*, *solidity*, and *visibility*. As explained in Section 4.4.1, each atom also has a set of attributes specific to that atom, such as for example the *width* and *height* of a rectangle. For all these atom specific attributes a single method exists that uses a String to identify the name of the attribute to be changed and an Object to hold its new value. The actual changing of these attributes is then delegated to the atom that needs to change, as only that particular atom *knows* if it is a valid request. The implementation of each atom accepts only those requests to change an attribute that make sense. If, e.g., an instance of LineAtom is asked to change its *radius*, it will be unable to comply because a line does not have a radius property. All atoms *complain* when asked to update an attribute they do not have, by throwing an IllegalArgumentException.

The actual painting of atoms

Whenever the surface of the canvas is exposed (for example just after its creation, or when a window covering the canvas is removed), Java invokes the `paintChildren` method. The job of the canvas is to override this method and perform any actions necessary to *paint* the atoms on the canvas.

If the default coordinate space (ranging 0..100 in both x and y direction) is used, no coordinate transformation is necessary and the canvas can immediately paint the relevant (visible) atoms. If, however, a different range of coordinates was requested, as should be allowed (see Section 4.3.1), the atoms need to be scaled accordingly. One of the reasons we chose Java for the implementation of the PAINTER, is because the Graphics2D object that is used for the actual drawing has built-in support for affine transformations. This enables us to scale, rotate and translate objects on the canvas while keeping properties such as parallelism of lines invariant. This greatly simplifies our task at hand, allowing us to simply translate the Graphics2D object over the correct vector and scale it according to the user's wishes. We then iterate over the list of atoms, check the visibility of the current atom and if it is indeed visible, request the atom to draw itself on the (translated and scaled) graphics object.

5.3.4 Class hierarchy of the Painter

Figure 5.2 shows the class hierarchy of the PAINTER as it is used in the Java implementation.

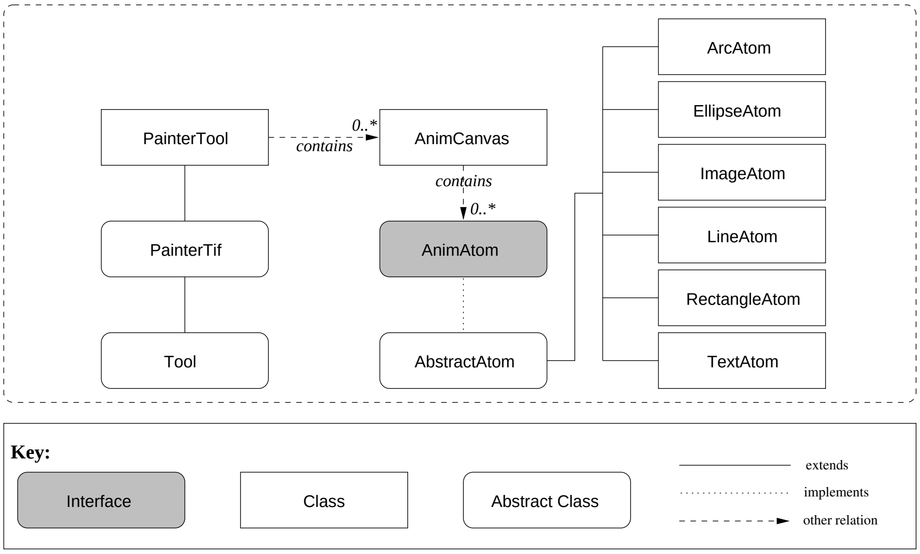


Figure 5.2: The Java class hierarchy of the PAINTER.

Chapter 6

Examples

This chapter presents some examples of visualisation templates. Section 6.1 details on a *generic* animator which presents a textual representation of the expression and its values. Section 6.2 shows how two RectangleAtoms can be combined to create a *progress indicator*. Section 6.3 shows an example of an EllipseAtom and an ArcAtom combined to create a *pie*. Several screenshots of the templates are shown throughout this chapter to give an impression of what the visualisation templates look like when used in a TIDE-session.

6.1 The **GenericAnimator** template

This section details on a simple visualisation template called GenericAnimator. A session using this template with TIDE is shown as well. The template shows a textual representation of the expression being visualised as well as its current value. In addition, it uses colours to distinguish between legal expression *values* and *error* messages.

The source code of the GenericAnimator can be found in Appendix E. The parts that deal with the actual visusalisation are copied and explained below.

All visualisation templates *implement* the AnimatorTemplate interface, which dictates the basic functionality that must be present in a template, and defines some constants for template states.

```
public interface AnimatorTemplate {

    public static final String STATE_UNKNOWN = "unknown";
    public static final String STATE_VALUE   = "value";
    public static final String STATE_ERROR   = "error";

    public String getState();

    public void setValue(String value);
    public void setError(String value);
}
```

The GenericAnimator is always in one of these three states: `unknown`, `value`, or `error`. The template is initialised to the `unknown` state, as no information about the value of the expression is available. The `value` and `error` states are used to represent the states mentioned in Section 5.2.4.

In addition to the state information, the GenericAnimator keeps track of the expression it is visualising and its value. At the time of creation, templates are given an instance of the class AnimCommandHandler which handles the low level sending of an AnimCommand to the ToolBus.

```
protected void initAnimator() {

    // Setup the default colors we will be using.
    colorMap = new HashMap();
    colorMap.put(STATE_UNKNOWN, new Color(100, 100, 255));  /* blueish */
    colorMap.put(STATE_VALUE,   new Color(  0, 255,   0));  /* green   */
    colorMap.put(STATE_ERROR,   new Color(255,   0,   0));  /* red     */
    registerColors();

    // Setup X- and Y-ranges.
    fireCommand(CommandFactory.cmdRangeX(0, 500));
    fireCommand(CommandFactory.cmdRangeY(0,  30));

    // Setup label to display the expression and value.
    labelID = dispenseID();
    fireCommand(CommandFactory.cmdCreate(labelID, "ATOM_TEXT"));
    fireCommand(CommandFactory.cmdSetLocation(labelID, 10, 20));
    fireCommand(CommandFactory.cmdSetColor(labelID, STATE_UNKNOWN));
    fireCommand(CommandFactory.cmdSetProperty(labelID, TextAtom.MESSAGE,
        expr + ": <unknown>"));
    fireCommand(CommandFactory.cmdSetVisible(labelID, true));

    // Request a repaint.
    fireCommand(CommandFactory.cmdRepaint());

    // Done with initialization, state of expr is unknown.
    state = STATE_UNKNOWN;
}
```

The method `initAnimator` begins by setting up the colours[1] it uses. Using a HashMap, a colour is associated with each of the three states.

```
private void registerColors() {
    Iterator iter = colorMap.keySet().iterator();
    while (iter.hasNext()) {
        String key = (String) iter.next();
        Color value = (Color) colorMap.get(key);
        fireCommand(CommandFactory.cmdRegisterColor(key, value));
    }
}
```

The CommandFactory mentioned in `registerColors` is a factory that contains the term patterns of all the commands. It is used to abstract from the ToolBus term level as soon as possible, allowing other classes to use the notion of an AnimCommand, rather than its term representation[2].

After setting up the colours, the template fires off two more commands to setup the X and Y ranges it intends to use. Then, the actual text atom is created. It is placed at coordinates $(10, 20)$, and initialised by setting its colour to the `unknown` colour and its text to the expression being visualised plus

[1] In Java, the American spelling of the word "color" is used, whereas this thesis uses the British spelling "colour". To keep consistency as local as possible, the American "color" is used in all source code, and the British "colour" is used in the text.

[2] e.g. `set-color(<int>,<str>)`

the verbatim "`: unknown`". The template then makes the atom visible. After everything is set up, the `repaint` command is issued.

Whenever the visualiser process receives an update of the value of the expression (at the ToolBus level), it invokes the `setValue` or `setError` method in the template, depending on whether a legal *value* or an *error* message was passed.

```
public void setValue(String newValue) {
    boolean needRepaint = false;

    if (state != STATE_VALUE) {
        state = STATE_VALUE;
        fireCommand(CommandFactory.cmdSetColor(labelID, STATE_VALUE));
        needRepaint = true;
    }

    if (newValue != value) {
        value = newValue;
        fireCommand(CommandFactory.cmdSetProperty(labelID,
            TextAtom.MESSAGE, expr + ": " + value));
        needRepaint = true;
    }

    if (needRepaint)
        fireCommand(CommandFactory.cmdRepaint());
}
```

The `setError` method is almost equal to the `setValue` method. The main difference is that the template's state as well as the atom's colour is set to `error` rather than to `value`.

The screenshots shown in Figure 6.1 and Figure 6.2 show the output of the commands issued by the GenericAnimator and executed by the PAINTER in TIDE. These screenshots were taken during a TIDE-session on a Tcl script. The script has a single procedure `fac{n}` which calculates the factorial of n. Present on the screenshot are a process list (on the left), two instances of the GenericAnimator showing the variables `i` and `result` used in the script, and the Source Viewer highlighting the current point of execution in the script.

Note that in Figure 6.1 the upper canvas displays an error condition (its text is painted in red). Variable `i` will not be declared until *after* this line of code is executed.

The toolbar at the bottom of the canvas shows five buttons. The first four allow condensing and expanding of the atoms on the canvas in x and y direction respectively. The fifth button (with four tiny, coloured squares on it) pops up a window allowing the user to interactively change the registered colours on the canvas.

6.2 The **ProgressAnimator**

A progress indicator is usually visualised by showing the outline of a rectangular frame, which is gradually "filled" by a another rectangle. Progress indicators are well-known from installation programs, but can be used for other tasks as well, e.g. to monitor the progress of a loop counter.

This section shows how two RectangleAtoms can be combined to create a progress indicator. Only specific portions of the source code are listed here for

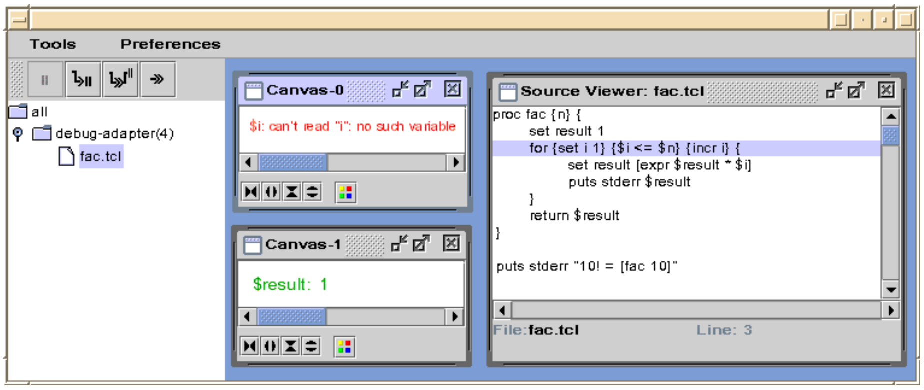


Figure 6.1: Two canvases of a PAINTER in TIDE executing commands from a GenericAnimator.

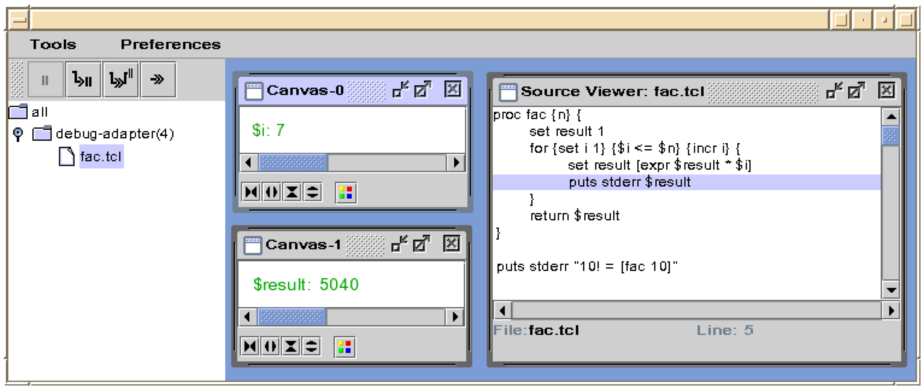


Figure 6.2: The canvases after several iterations of the loop.

reasons of brevity. Screenshots of a ProgressAnimator can be found in Figure 6.3 and Figure 6.4.

The example uses one RectangleAtom for the outer frame. This atom has its *solidity* set to false, which causes only its outline to show up when it is drawn. A second RectangleAtom is used for the body of the indicator. Its initial width is set to zero and it grows as the value of the expression increases. Until the first value of the expression arrives, the body of the progress indicator is hidden from view by setting its *visibility* to false. When the value of the expression exceeds a preset bound, the width of this rectangle is no longer increased, but instead its colour is changed to indicate it is out of bounds. As in the GenericAnimator, a textual representation of the value of the expression is also shown.

```
protected void initAnimator() {

    // Setup the default colors we will be using.
    colorMap = new HashMap();
    colorMap.put(COL_FRAME,      new Color(  0,    0,    0));
    colorMap.put(COL_ERROR,      new Color(255,  100,    0));
```

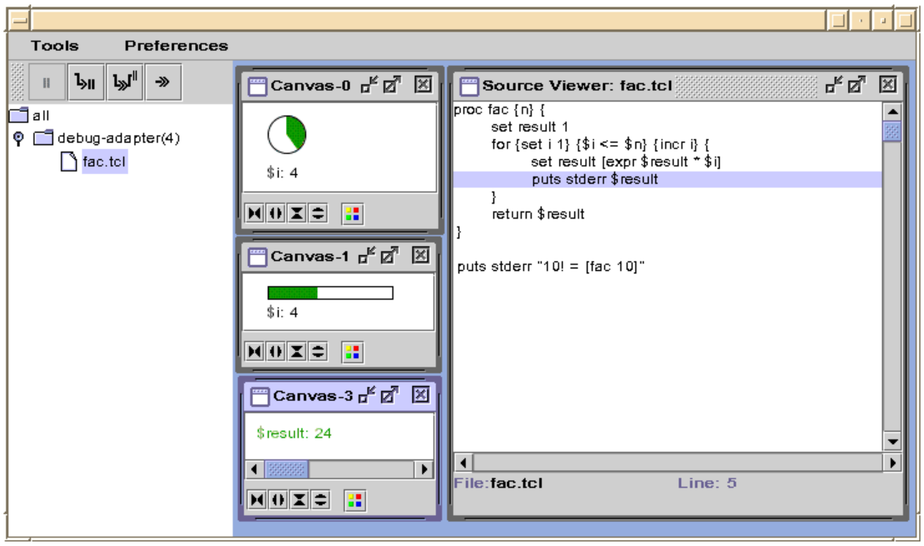


Figure 6.3: A Pie, Progress, and Generic animator in action.

```
colorMap.put(COL_IN_RANGE,  new Color( 50, 150,   0));
colorMap.put(COL_OUT_RANGE, new Color(255,   0,   0));
registerColors();

// Setup X- and Y-ranges.
fireCommand(CommandFactory.cmdRangeX(-10, 110));
fireCommand(CommandFactory.cmdRangeY(0,  40));

// Setup indicator.
indicatorID = dispenseID();
fireCommand(CommandFactory.cmdCreate(indicatorID, "ATOM_RECTANGLE"));
fireCommand(CommandFactory.cmdSetLocation(indicatorID, 10, 10));
fireCommand(CommandFactory.cmdSetSolid(indicatorID, true));
fireCommand(CommandFactory.cmdSetColor(indicatorID, COL_FRAME));
fireCommand(CommandFactory.cmdSetProperty(indicatorID,
            RectangleAtom.HEIGHT, "10"));
fireCommand(CommandFactory.cmdSetProperty(indicatorID,
            RectangleAtom.WIDTH, "0"));
fireCommand(CommandFactory.cmdSetVisible(indicatorID, false));

// Setup frame.
frameID = dispenseID();
fireCommand(CommandFactory.cmdCreate(frameID, "ATOM_RECTANGLE"));
fireCommand(CommandFactory.cmdSetLocation(frameID, 10, 10));
fireCommand(CommandFactory.cmdSetSolid(frameID, false));
fireCommand(CommandFactory.cmdSetColor(frameID, COL_FRAME));
fireCommand(CommandFactory.cmdSetProperty(frameID,
            RectangleAtom.HEIGHT, "10"));
fireCommand(CommandFactory.cmdSetProperty(frameID,
            RectangleAtom.WIDTH, "100"));
fireCommand(CommandFactory.cmdSetVisible(frameID, true));

// Setup textlabel.
labelID = dispenseID();
fireCommand(CommandFactory.cmdCreate(labelID, "ATOM_TEXT"));
```

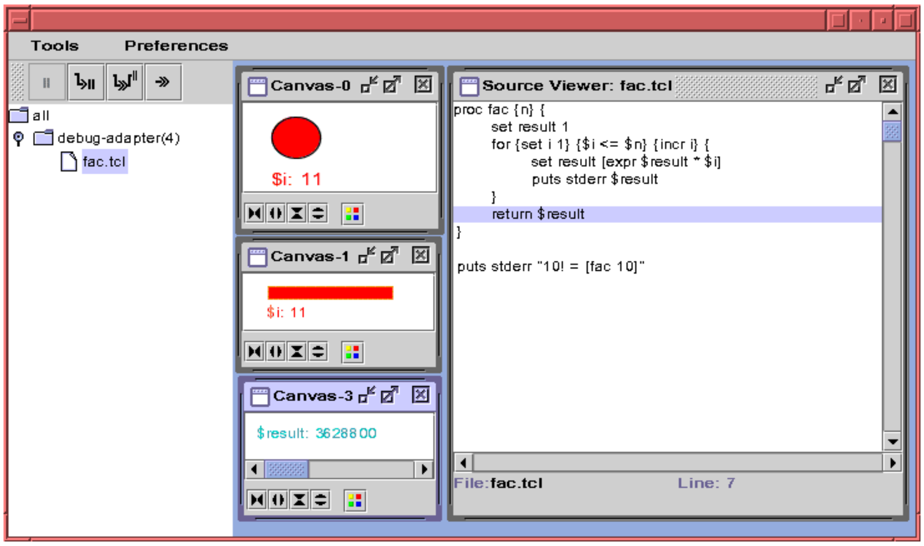


Figure 6.4: The Pie and Progress animator showing values that are out of range.

```
    fireCommand(CommandFactory.cmdSetLocation(labelID, 10, 35));
    fireCommand(CommandFactory.cmdSetColor(labelID, COL_FRAME));
    fireCommand(CommandFactory.cmdSetProperty(labelID,
                TextAtom.MESSAGE, strExpr + ": ?"));
    fireCommand(CommandFactory.cmdSetVisible(labelID, true));

    // Request a repaint.
    fireCommand(CommandFactory.cmdRepaint());

    // Done with initialization, state of expr is unknown.
    state = STATE_UNKNOWN;
}
```

The source code for the `setValue` method is listed next. To minimize the number of animation commands that are sent to the PAINTER, some administration has to be done. If this bookkeeping is not done, the PAINTER would be told to make the indicator visible *each* time a new value for the expression is set. Although this would be a legal way, there is no need to keep telling the PAINTER to make something visible that is already visible. The same holds for the colour of the atoms we want to change. The template could just tell the PAINTER to change the colour of the indicator, but there is little sense in constantly "changing" the colour of an atom to the colour it already has.

```
public void setValue(String newValue) {
    if (newValue != strValue) {

        // Remember if previous state was also a legal value.
        boolean wasValue = isValue;

        // Convert new value from String to int.
        int value = Integer.parseInt( newValue );

        // Update internal state
        strValue = newValue;
```

```
        isValue  = true;
        inRange  = (value >= minValue && value <= maxValue);

        // If previous state wasn't a value state, make indicator
        // visible and change color of frame and label.
        if (!wasValue) {
            fireCommand(CommandFactory.cmdSetVisible(indicatorID, true));
            fireCommand(CommandFactory.cmdSetColor(labelID, COL_FRAME));
            fireCommand(CommandFactory.cmdSetColor(frameID, COL_FRAME));
        }

        fireCommand(CommandFactory.cmdSetColor(labelID,
                    inRange ? COL_FRAME : COL_OUT_RANGE));
        fireCommand(CommandFactory.cmdSetColor(indicatorID,
                    inRange ? COL_IN_RANGE : COL_OUT_RANGE));

        // Make sure value does not fall out of frame.
        value = Math.min(maxValue, Math.max(minValue, value));

        // Update width of the indicator.
        int width = (100*value)/(maxValue-minValue);
        fireCommand(CommandFactory.cmdSetProperty(indicatorID,
                    RectangleAtom.WIDTH, ""+width));

        // Update text on the label.
        fireCommand(CommandFactory.cmdSetProperty(labelID,
                    TextAtom.MESSAGE, strExpr + ": " + strValue));

        // Issue a repaint.
        fireCommand(CommandFactory.cmdRepaint());
    }
}
```

The code for the `setError` method is not listed here, but could be considered an exercise for the interested reader. All it does is change the colour of the label and the frame to the registered *error* colour, hide the indicator from view and update the textual representation of the value of the expression.

6.3 The **PieAnimator**

This section contains a concise explanation of a *pie* animator. A PieAnimator shows values of an expression that lie within a given range by drawing a segment of a circle. Screenshots of the PieAnimator can be found in Figure 6.3 and Figure 6.4.

The following source code is taken from PieAnimator's `initAnimator` method. It shows how to setup an ArcAtom and an EllipseAtom. The arc is used for the "body" of the pie, the ellipse is used to draw the outline. As the bounding box (defined by `width` and `height`) is square, the resulting arc and ellipse are circular in shape. As with the ProgressAnimator, the body starts out invisible, until a value for the expression is known.

```
indicatorID = dispenseID();
fireCommand(CommandFactory.cmdCreate(indicatorID, "ATOM_ARC"));
fireCommand(CommandFactory.cmdSetLocation(indicatorID, 30, 10));
fireCommand(CommandFactory.cmdSetSolid(indicatorID, true));
fireCommand(CommandFactory.cmdSetColor(indicatorID, COL_FRAME));
fireCommand(CommandFactory.cmdSetProperty(indicatorID, ArcAtom.WIDTH, "30"));
```

```
fireCommand(CommandFactory.cmdSetProperty(indicatorID, ArcAtom.HEIGHT, "30"));
fireCommand(CommandFactory.cmdSetProperty(indicatorID, ArcAtom.START, "90"));
fireCommand(CommandFactory.cmdSetProperty(indicatorID, ArcAtom.EXTENT, "0"));
fireCommand(CommandFactory.cmdSetProperty(indicatorID, ArcAtom.TYPE, ""+Arc2D.PIE));

frameID = dispenseID();
fireCommand(CommandFactory.cmdCreate(frameID, "ATOM_ELLIPSE"));
fireCommand(CommandFactory.cmdSetLocation(frameID, 30, 10));
fireCommand(CommandFactory.cmdSetSolid(frameID, false));
fireCommand(CommandFactory.cmdSetColor(frameID, COL_FRAME));
fireCommand(CommandFactory.cmdSetProperty(frameID, EllipseAtom.WIDTH, "30"));
fireCommand(CommandFactory.cmdSetProperty(frameID, EllipseAtom.HEIGHT, "30"));
fireCommand(CommandFactory.cmdSetVisible(frameID, true));

fireCommand(CommandFactory.cmdRepaint());
```

The angle extent is calculated by linear interpolation of the current value of the expression between 0° for the minimum possible value and 360° for the maximum possible value of the expression. By using a negative angle, the effect of clockwise "motion" is achieved.

```
// Update indicator.
int extent = - (360*value) / (maxValue-minValue);
fireCommand(CommandFactory.cmdSetProperty(indicatorID, ArcAtom.EXTENT, ""+extent));
```

Chapter 7

Concluding remarks

This chapter looks back upon the work presented in this thesis. We discuss some practical experiences, summarise the chapters, detail on ideas and future work and present several conclusions.

7.1 Java implementation quirks

During the implementation phase of the PAINTER we came across several quirks in Java, ranging from amusing things like a typo in a method name to more serious issues. In this section, we list some of the weird things we encountered when working with the JFC.

WYSIWYG: What You Set Is What You Get?

When implementing the LineAtom class, we wanted to use Java's Point class to hold the coordinates of the two endpoints of the line segment. This Point class has two attributes x and y of type `int` that hold the actual coordinates. However, the corresponding methods to retrieve these attributes, `getX()` and `getY()` return x and y in `double` precision. Even worse, Point has a method `setLocation()` accepting `double` values, which are silently converted to `int`s (which are used in the internal representation), loosing the extra precision without warning.

Fortunately, the Point class was reimplemented in the `java.awt.geom` package. This package offers the abstract notion of a 2D point in the Point2D class and supplies two concrete implementations, descriptively called Point2D.Float and Point2D.Double respectively. The documentation of the older Point contains no reference to these classes, however.

Semantical difference through operator overloading

The reason we want to use Java's Graphics2D class is because it supports easy translation, rotation, etc. When looking up the documentation on how to invoke the method for translating an object, we made the following, rather peculiar, discovery. There are *two* versions of `translate`. The first takes two integers as parameters for the translation, the second takes two doubles. This by itself is nothing but *operator overloading*, a well known Object Oriented programming technique. However, when we read the documentation, it turns out that this is not just a case of operator overloading, but instead there is a semantic difference:

`translate(int x, int y)`:
Translates the origin of the Graphics2D context to the point (x, y) in the current coordinate system. Modifies the Graphics2D context so that its new origin corresponds to the point (x, y) in the Graphics2D context's former coordinate system. All coordinates used in subsequent rendering operations on this graphics context are relative to this new origin.

`translate(double x, double y)`: Concatenates the current Graphics2D Transform with a translation transform. Subsequent rendering is translated by the specified distance relative to the previous position. This is equivalent to calling transform(T), where T is an AffineTransform represented by the following matrix:

```
[   1    0    tx  ]
[   0    1    ty  ]
[   0    0    1   ]
```

In other words, if we (by accident) pass the translation vector using integers, we get something *similar* to using doubles, but not quite the same.

Swing: *festina lente*[1]

The current implementation of Swing (the package that is used to create the desktop, frames, windows, buttons, menus, etc.) is very slow. Even when working on an otherwise unloaded, *powerful*[2] SUN Workstation (ULTRA SPARC-5, 270 MHz with 256 Mb of memory), the speed at which a JInternalFrame can be dragged across a JDesktop is at best called *crawling*. Swing is still under development though, and the performance of hardware still increases considerably each year. Therefore we expect this performance problem to gradually disappear.

7.2 Summary

In this thesis, we have presented a generically applicable framework for visualisation of expressions in ToolBus applications. The need for this visualisation framework found its roots in TIDE, a "ToolBus Integrated Debugging Environment". We gave a crash course in ToolBus understanding (Chapter 2) and presented an introduction to debugging software (Chapter 3).

Given the constraint that the visualisation framework had to fit in an existing ToolBus application (TIDE), we discussed the design of the framework in Chapter 4. We divided the framework into two separate components: the VISUALISER, and the PAINTER. These components work together by communication through the use of a set of instructions defined on atomic animation elements. Both components have specific responsibilities and areas of expertise, which are closely related to these responsibilities.

The VISUALISER knows the expression to be visualised, receives updates of the value of the expression and is responsible for the generation of animation commands. The VISUALISER has no knowledge about the actual execution of the animation commands.

The PAINTER receives animation commands and is responsible for the semantically correct execution of these commands. We say semantically correct

[1]from Latin, meaning *make haste slowly.*
[2]powerful at the time of writing of this thesis, that is.

to prevent the PAINTER from drawing, e.g. a line when asked to draw a rectangle. The PAINTER has no knowledge about the expression it is visualising, it does not even know that it is in fact visualising the values of expressions. The PAINTER only knows how to execute a given set of animation commands and does just that.

Given the requirements, we looked into finding a good way to implement the components. For both the VISUALISER and the PAINTER we motivated why we opted for the Java Programming Language for their implementation and then highlighted some of the implementation details of both components.

7.3 Conclusions

- Use of the ToolBus stimulates thinking in terms of *generic solutions* for *specific problems*. Instead of creating an *ad hoc* visualisation for some recurring debugging expressions, we created a generically applicable visualisation framework for ToolBus applications.

- The extra level of indirection introduced when using the ToolBus to *glue* together a set of components obfuscates debugging both the individual components and the system as a whole. When the system is not responding, it is often unclear whether a component is malfunctioning, whether the communication between component and ToolBus is askew, or whether communication between processes inside the ToolBus has gone off track.

- Java's use of *interfaces* to describe the functionality of data types (e.g. List, Map, and Set) is convenient during program development. We can just pick one implementation and focus on what we really want to create. This contrasts the routine when programming in, e.g. `C`, where much time is needed to write basic data structures again and again.

- Design and implementation details of one ToolBus component *can* influence the implementation of other ToolBus components. By specifying more and more specific demands in the VISUALISER, we can make it virtually impossible to implement a matching PAINTER. Therefore, even when we are developing one component of an integrated system, we still have to keep the project as a whole in mind.

7.4 Future work

During the development of a project, many ideas, fantasies and Utopias bubble up. We ended up with a generic visualisation framework for use in ToolBus applications. Some work remains to be done to seamlessly integrate the framework into TIDE, most notably the creation of several *templates* for the visualisation of recurring debugging expressions, such as counters, lists, and perhaps even structures or classes.

We have extended TIDE with a selection mechanism, allowing users to enter the expression and select a visualisation template from a list, but perhaps there is a better way for this from the TIDE viewpoint. For example, driving the visualisation directly from a source code viewer may be more intuitive to a programmer than selecting a menu item, followed by manual entering of the expression.

Both the Java and the Tcl/Tk candidates for the PAINTER offer a more sophisticated means to layout components on a window than the basic coordinate mechanism used in our framework. These layout or geometry managers allow more abstract placing of objects by allowing the programmer to give constraints such as "place A north of B", instead of having the programmer pass explicit coordinates. It may prove useful to exploit these features, allowing, the VISUALISER to say, e.g. "draw a rectangle to the left of this circle".

Bibliography

[1] S. Adams. *The Dilbert Principle.* MacMillan Publishers Ltd, 1996.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[3] J.A. Bergstra and P. Klint. The Discrete Time ToolBus. Technical Report P9502, Programming Research Group, University of Amsterdam, 1995.

[4] J.A. Bergstra and P. Klint. The discrete time ToolBus — a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[5] J.A. Bergstra and M.E. Loots. Empirical Semantics for Object-Oriented Programs. Technical Report 007, Onderwijsinstituut CKI - Utrecht University, July 1999. Artificial Intelligence Preprint Series.

[6] J.A. Bergstra and M.E. Loots. Software Mechanics for Java Multi-Threading. Technical Report 005, Onderwijsinstituut CKI - Utrecht University, July 1999. Artificial Intelligence Preprint Series.

[7] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Accepted for publication in Software – Practice & Experience*, 1999.

[8] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and Implementation of a New ASF+SDF Meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, November 1997. Springer Verlag.

[9] J. Gosling, B. Joy, and G. Steele. *The Java language specification.* The Java series. Addison-Wesley Publishing Company, 1996.

[10] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The Syntax Definition Formalism SDF, Reference Manual.* Programming Research Group, University of Amsterdam, 1992.

[11] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Software Series. Prentice Hall, 2nd edition, 1988.

[12] Thomas G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.

[13] Pieter A. Olivier. Debugging distributed applications using a coordination architecture. In D. Garlan and D. Le Metayer, editors, *Coordination Languages and Models (COORDINATION)*, volume 1061 of *LNCS*, pages 98–114. Springer Verlag, 1997.

[14] J.K. Ousterhout. *Tcl and the Tk ToolKit*. Professional Computing Series. Addison-Wesley Publishing Company, 1994.

[15] Takao Shimomura and Sadahiro Isoda. Linked-list visualization for debugging. *IEEE-Software*, 8(3):44–51, May 1991.

[16] R.M. Stallman and R.H. Pesch. *Debugging with GDB, the GNU Source-Level Debugger*, seventh edition, February 1999. For GDB version 4.18.

[17] J.T. Stasko. *POLKA Animation Designer's Package, Release 1.25*. Graphics, Visualization, and Usability Center, College of Computing, Georgia Insitute of Technology, December 1997. URL: `ftp://ftp.cc.gatech.edu/pub/people/stasko/polka.tar.Z`.

[18] J.T. Stasko. *SAMBA Animation Designer's Package, Release 1.25*. Graphics, Visualization, and Usability Center, College of Computing, Georgia Insitute of Technology, December 1997. Shipped with the POLKA package [17].

[19] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 3rd edition, July 1997.

[20] Sun Microsystems, Inc. *Code Conventions for the Java Programming Language*, 1999. Available for as online HTML and for downloading. URL: `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`.

[21] Sun Microsystems, Inc. *Java Platform Debugger Architecture*, 1999. Available only as online HTML documentation. URL: `http://java.sun.com/products/jdk/1.3/docs/guide/jpda/index.html`.

[22] F. Tip. *Generation of Program Analysis Tools*. PhD thesis, Institute for Logic, Language and Computation, Amsterdam, March 1995. ILLC dissertation series; 1995-5.

[23] Ioannis G. Tollis. Graph drawing and information visualization. *ACM Computing Surveys*, 28(4es):19, December 1996.

Appendix A

The visualiser process

```
process VISUALISER is
let
    V           : visualiser,       %% Generates animation commands.
    P           : painter,          %% Painter that creates canvases.
    D           : debug-adapter,    %% Responsible for the evaluation.
    T           : debug-tool,       %% The tool requesting the visualisation.
    From        : term,             %% Source of an event from D.
    To          : term,             %% Target of an event from D.
    Event       : term,             %% Actual event D is sending.
    Expr        : str,              %% The expression to be visualised.
    Tmpl        : str,              %% Defines how expr is visualised.
    Proc        : term,             %% Process asking for visualisation.
    Rid         : int,              %% Rule ID of watchpoint.
    Cid         : int,              %% Canvas ID.
    Cmd         : term,             %% Animation command.
    Templates   : term              %% Possible visualisation templates.

in
  %% Startup the visualiser tool.
  execute(visualiser, V?) .

  %% Subscribe to events pertaining to watchpoints.
  %% These events tell us the value- and errorstates of the expression.
  subscribe(event(<term>, <term>, watchpoint(<int>,value(<term>)))) .
  subscribe(event(<term>, <term>, watchpoint(<int>,error(<term>)))) .
  (
    %% Handle a request to visualize an expression.
    %% First construct a key containing all elements that make
    %% this request unique. Then pass the key, the expression and
    %% the template to the visualiser.
    rec-msg(visualize(Proc?, Rid?, Expr?, Tmpl?)) .
    snd-msg(create-canvas) .
    rec-msg(canvas-created(P?,Cid?)) .
    snd-do(V, visualize(vis-key(Proc,Rid), canvas-key(P,Cid), Expr, Tmpl))
    +

    %% Handle event from visualiser to send an animation command.
    rec-event(V, anim-cmd(To?,Cmd?)) .
```

```
    snd-msg(To, Cmd) .
    snd-ack-event(V, anim-cmd(To,Cmd))
    +

    %% A watchpoint is informing us about one of our expressions,
    %% so extract the key and pass the event to the visualiser.
    rec-note(event(From?, To?, watchpoint(Rid?, Event?))) .
    snd-do(V, update(vis-key(From, Rid), Event))
  ) * delta
  ||
  (
    %% Respond to requests asking for all the possible visualization
    %% templates we can handle. Let the visualiser answer the question
    %% and return the resulting list of templates to the caller.
    %% Do this in parallel to handling the more intricate visualization
    %% instructions, as it does not interfere with those.
    rec-msg(get-visualization-templates) .
    snd-eval(V, get-templates) .
    rec-value(V, templates(Templates?)) .
    snd-msg(visualization-templates(Templates))
  ) * delta
endlet
```

Appendix B

The painter process

```
process PAINTER is
let
    P       : painter,               %% The painter tool.
    Cid     : int,                   %% Unique canvas ID per painter.
    Cmd     : term                   %% Animation command to execute.
in
  rec-connect(P?) .
  (
    %% Handle a request to create a new canvas. Pass the request
    %% on to the painter, receive a unique ID associated with
    %% this canvas and send that as the result.
    rec-msg(create-canvas) .
    snd-eval(P, create-canvas) .
    rec-value(P, canvas-created(Cid?)) .
    snd-msg(canvas-created(P,Cid))
    +

    %% Handle request to destroy a canvas.
    rec-msg(destroy-canvas(canvas-key(P,Cid?))) .
    snd-do(P, destroy-canvas(Cid))
    +

    %% Handle request to execute an animation command on a canvas.
    rec-msg(canvas-key(P,Cid?), Cmd?) .
    snd-do(P, execute-command(Cid, Cmd))
  ) *
  rec-disconnect(P)

endlet
```

Appendix C

The PainterTool class

```
package tide.tools.animviewer;

import aterm.*;
import aterm.tool.*;

import java.awt.*;
import java.awt.event.*;

import java.io.*;
import java.net.*;
import java.util.*;

import javax.swing.*;
import javax.swing.event.*;

import tide.tools.*;
import tide.debug.*;

public class PainterTool extends PainterTif
{
    /** Name of the painter used in ToolBus communication.      */
    private static String PAINTER_NAME = "painter";

    /** Used to generate unique identification numbers.         */
    private int canvasID;

    /** Mapping to hold all canvases this painter knows about.  */
    private Map canvasMap;

    /** Desktop the canvases should be created on.              */
    private JDesktopPane desktop;

    /** State of the tool, is it running?                       */
    private boolean isRunning;

    /** Thread that runs this tool                              */
    private Thread thread;
```

```
/**
 * Create a new PainterTool object.
 *
 * @param desktop JDesktopPane that holds created canvases.
 * @param addr Address of the ToolBus to connect to.
 * @param port Port of the ToolBus to connect to.
 */
public PainterTool(JDesktopPane desktop, InetAddress addr, int port)
    throws UnknownHostException
{
    // Let parent setup the name, address and port.
    super(PAINTER_NAME, addr, port);

    this.canvasID  = 0;
    this.canvasMap = new HashMap();
    this.desktop   = desktop;
    this.isRunning = true;
    this.thread    = new Thread(this);

    // Start the thread running this painter.
    thread.start();
}

/**
 * Allow another thread to join our thread.
 * The parent that created us may wish to wait for us to terminate
 * gracefully. This can be done by <code>join</code>ing us.
 *
 * @exception InterruptedException
 * @see Thread#join
 */
public void join() throws InterruptedException {
    thread.join();
}

/**
 * Run method is called from our thread.
 *
 * While we are in the running state, let our parent do its work.
 */
public void run() {
    while (isRunning) {
        super.run();
    }
}

/**
 * Dispense unique canvas identifiers.
 *
 * Implemented to give the next (unused) <code>int</code> in the
 * ascending range <code>0 through Integer.MAX_VALUE</code>.
 *
 * @exception RuntimeException when the next canvas ID would
 * exceed <code>Integer.MAX_VALUE</code>.
```

```
 */
private synchronized int dispenseCanvasID() {
    if (canvasID < Integer.MAX_VALUE) {
        return canvasID++;
    } else {
        throw new RuntimeException("PainterTool: ID-pool exhausted!");
    }
}

/** Handle request to create a new canvas.  */
ATerm createCanvas() {
    // Get a unique ID for this canvas.
    int cid = dispenseCanvasID();

    // Create the frame and the canvas.
    PainterFrame frame = new PainterFrame(cid);
    AnimCanvas canvas = new AnimCanvas(frame);

    // Put the canvas in our administration.
    canvasMap.put(new Integer(cid), canvas);

    // Pack and install the canvas.
    frame.pack();
    desktop.add(frame, new Integer(1));
    desktop.moveToFront(frame);

    // Return the ID.
    return ATerm.make("snd-value(canvas-created(<int>))", new Integer(cid));
}

void executeCommand(int cid, ATerm command) {
    Vector matchResult;

    System.err.println("#P# executing " + command);

    AnimCanvas canvas = (AnimCanvas) canvasMap.get(new Integer(cid));
    if (canvas == null) {
        throw new RuntimeException("No such canvas: " + cid);
    }

    matchResult = command.match("create(<int>,<str>)");
    if (matchResult != null) {
        int id = ((Integer)matchResult.elementAt(0)).intValue();
        String type = (String)matchResult.elementAt(1);
        canvas.createAnimAtom(id, type);
        return;
    }

    matchResult = command.match("range-x(<int>,<int>)");
    if (matchResult != null) {
        int lowX  = ((Integer)matchResult.elementAt(0)).intValue();
        int highX = ((Integer)matchResult.elementAt(1)).intValue();
        canvas.setRangeX(lowX, highX);
        return;
```

```
}

matchResult = command.match("range-y(<int>,<int>)");
if (matchResult != null) {
    int lowX  = ((Integer)matchResult.elementAt(0)).intValue();
    int highX = ((Integer)matchResult.elementAt(1)).intValue();
    canvas.setRangeY(lowX, highX);
    return;
}

if (command.match("repaint") != null) {
    canvas.repaint();
    return;
}

matchResult = command.match("set-location(<int>,<int>,<int>)");
if (matchResult != null) {
    int id = ((Integer)matchResult.elementAt(0)).intValue();
    int x  = ((Integer)matchResult.elementAt(1)).intValue();
    int y  = ((Integer)matchResult.elementAt(2)).intValue();
    canvas.setAtomLocation(id, x, y);
    return;
}

matchResult = command.match("set-color(<int>,<str>)");
if (matchResult != null) {
    int id = ((Integer)matchResult.elementAt(0)).intValue();
    String colorKey = (String) matchResult.elementAt(1);
    canvas.setAtomColor(id, colorKey);
    return;
}

matchResult = command.match("set-solid(<int>,<bool>)");
if (matchResult != null) {
    int id = ((Integer)matchResult.elementAt(0)).intValue();
    boolean vis = ((Boolean)matchResult.elementAt(1)).booleanValue();
    canvas.setAtomSolidness(id, vis);
    return;
}

matchResult = command.match("set-property(<int>,<str>,<str>)");
if (matchResult != null) {
    int id = ((Integer)matchResult.elementAt(0)).intValue();
    String key = (String)matchResult.elementAt(1);
    String value = (String)matchResult.elementAt(2);
    canvas.setAtomProperty(id, key, value);
    return;
}

matchResult = command.match("set-visible(<int>,<bool>)");
if (matchResult != null) {
    int id = ((Integer)matchResult.elementAt(0)).intValue();
    boolean vis = ((Boolean)matchResult.elementAt(1)).booleanValue();
    canvas.setAtomVisibility(id, vis);
```

```
            return;
        }

        matchResult = command.match("register-color(<str>,<str>,<int>,<int>,<int>)");
        if (matchResult != null) {
            String path = (String) matchResult.elementAt(0);
            String key  = (String) matchResult.elementAt(1);
            int r  = ((Integer)matchResult.elementAt(2)).intValue();
            int g  = ((Integer)matchResult.elementAt(3)).intValue();
            int b  = ((Integer)matchResult.elementAt(4)).intValue();
            canvas.registerColor(path, key, new Color(r, g, b));
            return;
        }

        matchResult = command.match("config-path(<str>)");
        if (matchResult != null) {
            String configPath = (String) matchResult.elementAt(0);
            canvas.setConfigPath(configPath);
            return;
        }

        throw new RuntimeException("Unknown command: " + command);
    }


    /**
     * Handle request to terminate this tool.
     *
     * As this tool is run by a thread, we don't do a System.exit,
     * but deactivate the responsible thread instead.
     *
     * @param desc describes nature of termination request.
     */
    void recTerminate(ATerm desc) {
        // Stop the thread running this tool.
        isRunning = false;
    }

    class PainterFrame extends JInternalFrame {
        PainterFrame(final int cid) {
            super("Canvas-"+cid, true, true, true, true);
            addInternalFrameListener(new InternalFrameAdapter() {
                public void internalFrameClosing(InternalFrameEvent e) {
                    destroyCanvas(cid);}});
        }

        public Dimension getPreferredSize() {
            return new Dimension(200, 110);
        }
    }
}
```

Appendix D

The AnimCanvas class

```
package tide.tools.animviewer;

import tide.tools.*;
import tide.debug.*;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.tree.*;


public class AnimCanvas extends JComponent {

    public static final String ARC       = "ATOM_ARC";
    public static final String ELLIPSE   = "ATOM_ELLIPSE";
    public static final String IMAGE     = "ATOM_IMAGE";
    public static final String LINE      = "ATOM_LINE";
    public static final String RECTANGLE = "ATOM_RECTANGLE";
    public static final String TEXT      = "ATOM_TEXT";

    /** Eheu! "List" ambigu est: java.awt.List vs java.util.List */
    private java.util.List animAtoms;

    private int startX;
    private int startY;
    private int endX;
    private int endY;

    private int zoomPctX;
    private int zoomPctY;

    private Map renderingHints;
```

```
private JToolBar toolBar;
private JScrollPane scrollPane;
private JInternalFrame parent;
private Container contentPane;
private String configPath;

/** Mapping to hold all registered colors.  */
private Map colorMap;

/** Create a new AnimCanvas object.  */
public AnimCanvas(JInternalFrame parent) {
    this.animAtoms = null;

    // Default coordinates: 0..100
    this.startX = 0;
    this.startY = 0;
    this.endX   = 100;
    this.endY   = 100;

    // Default zoom is at 100%
    this.zoomPctX = 100;
    this.zoomPctY = 100;

    // Create a mapping of the RenderingHints for later reuse.
    this.renderingHints = new HashMap();
    this.renderingHints.put(RenderingHints.KEY_ANTIALIASING,
                            RenderingHints.VALUE_ANTIALIAS_ON);
    this.renderingHints.put(RenderingHints.KEY_RENDERING,
                            RenderingHints.VALUE_RENDER_QUALITY);
    this.renderingHints.put(RenderingHints.KEY_TEXT_ANTIALIASING,
                            RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

    this.toolBar = createToolBar();

    this.parent = parent;
    // Surround ourselves by a scroll pane.
    this.scrollPane = new JScrollPane(this);

    // Use a hash map to hold all registered colors.
    this.colorMap = new HashMap();

    // Extract content pane.
    this.contentPane = parent.getContentPane();

    this.configPath = "";

    // Install a BorderLayout.
    contentPane.setLayout(new BorderLayout());

    // Put scroll pane in the center, toolbar below.
    contentPane.add(scrollPane, BorderLayout.CENTER);
    contentPane.add(toolBar, BorderLayout.SOUTH);

    // Start with a white background.
```

```
    setBackground(Color.white);

    // Curtain up.
    setVisible(true);
}

private JButton createColorButton(final JPanel panel, final Object key) {
    Color curColor = (Color) colorMap.get(key);
    final JButton b = new JButton("Change", new ColorIcon(curColor));
    b.setFont(new Font("Helvetica", Font.PLAIN, 12));
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            Color newColor = JColorChooser.showDialog(panel,
                "Select color", (Color) colorMap.get(key));
            if (newColor != null) {
                // Update color on this very button
                b.setIcon(new ColorIcon(newColor, 12, 12));
                // Update color map.
                colorMap.put(key, newColor);
                repaint();
            }}});
    return b;
}

private void customize() {
    JPanel treePanel = new JPanel();
    treePanel.setBorder(new CompoundBorder(
        new EmptyBorder(2, 2, 2, 2),
        new TitledBorder(new EtchedBorder(), "Persistency",
        TitledBorder.DEFAULT_JUSTIFICATION,
        TitledBorder.DEFAULT_POSITION,
        new Font("Helvetica", Font.PLAIN, 10))));
    CustomizeTree tree = new CustomizeTree();
    treePanel.add(tree);
    JScrollPane treeScroll = new JScrollPane(treePanel);
    treeScroll.setBorder(null);
    tree.addCustomPath(configPath);

    JPanel colorPanel = new JPanel();
    colorPanel.setLayout(new BoxLayout(colorPanel, BoxLayout.Y_AXIS));
    colorPanel.setBorder(new CompoundBorder(
        new EmptyBorder(2, 2, 2, 2),
        new TitledBorder(new EtchedBorder(), "Colors",
            TitledBorder.DEFAULT_JUSTIFICATION,
            TitledBorder.DEFAULT_POSITION,
            new Font("Helvetica", Font.PLAIN, 10))));
    Iterator iter = colorMap.keySet().iterator();
    while (iter.hasNext())
        colorPanel.add(createColorButton(colorPanel, iter.next()));
    colorPanel.add(Box.createVerticalGlue());

    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
    p.add(treeScroll);
```

```
        p.add(colorPanel);

        JInternalFrame frame = new JInternalFrame("Customize", true, true, true, true);
        frame.getContentPane().add(p);
        frame.pack();

        JDesktopPane desktop = (JDesktopPane) parent.getParent();
        desktop.add(frame, JLayeredPane.PALETTE_LAYER);
        desktop.moveToFront(frame);
    }


    private Icon createCustomIcon() {
        return new Icon() {
            public void paintIcon(Component c, Graphics g, int x, int y) {
                Color oldColor = g.getColor();
                g.setColor(Color.yellow);
                g.fill3DRect(x+1,y+1,4,4,true);
                g.setColor(Color.red);
                g.fill3DRect(x+7,y+1,4,4,true);
                g.setColor(Color.green);
                g.fill3DRect(x+1,y+7,4,4,true);
                g.setColor(Color.blue);
                g.fill3DRect(x+7,y+7,4,4,true);
                g.setColor(oldColor);
            }
            public int getIconWidth()  { return 12; }
            public int getIconHeight() { return 12; }
        };
    }

    private JToolBar createToolBar() {
        JButton bAction;

        JToolBar toolBar = new JToolBar(JToolBar.HORIZONTAL);
        toolBar.setFloatable(false);

        bAction = toolBar.add(new
            AbstractAction("condense X",
                           new ImageIcon("images/condense-x.gif")) {
                public void actionPerformed(ActionEvent e) {
                    zoomPctX -= 10;
                    repaint();
                }
            }
        );
        bAction.setText(null);
        bAction.setToolTipText("Condense X");
        bAction = toolBar.add(new
            AbstractAction("expand X", new ImageIcon("images/expand-x.gif")) {
                public void actionPerformed(ActionEvent e) {
                    zoomPctX += 10;
                    repaint();
                }});
```

```
    bAction.setText(null);
    bAction.setToolTipText("Expand X");
    bAction = toolBar.add(new
        AbstractAction("condense Y", new ImageIcon("images/condense-y.gif")) {
            public void actionPerformed(ActionEvent e) {
                zoomPctY -= 10;
                repaint();
            }});
    bAction.setText(null);
    bAction.setToolTipText("Condense Y");
    bAction = toolBar.add(new
        AbstractAction("expand Y", new ImageIcon("images/expand-y.gif")) {
            public void actionPerformed(ActionEvent e) {
                zoomPctY += 10;
                repaint();
            }});
    bAction.setText(null);
    bAction.setToolTipText("Expand Y");
    toolBar.addSeparator();
    bAction = toolBar.add(new
        AbstractAction("customize", createCustomIcon()) {
            public void actionPerformed(ActionEvent e) {
                customize();
            }});
    bAction.setText(null);
    bAction.setToolTipText("Customize");

    return toolBar;
}

public Dimension getPreferredSize() {
    return new Dimension(endX - startX, endY - startY);
}

public void setRangeX(int low, int high) {
    this.startX = low;
    this.endX   = high;

    int unitInc  = (endX-startX)/100; // equals 1% of area
    if (unitInc > 0) {
        scrollPane.getHorizontalScrollBar().setUnitIncrement(unitInc);
    }

    int blockInc = (endX-startX)/20;  // equals 5% of area
    if (blockInc > 0) {
        scrollPane.getHorizontalScrollBar().setBlockIncrement(blockInc);
    }

    scrollPane.revalidate();
}

public void setRangeY(int low, int high) {
    this.startY = low;
    this.endY   = high;
```

```
    scrollPane.revalidate();
}

private void addAtomImpl(int animID, AnimAtom atom) {
    if (animAtoms == null) {
        animAtoms = new ArrayList();
    }
    animAtoms.add(animID, atom);
}

private void delAtomImpl(int key) {
    animAtoms.remove(key);
    if (animAtoms.isEmpty()) {
        animAtoms = null;
    }
}

protected void setAtomsVisibility(boolean visible) {
    if (animAtoms == null || animAtoms.isEmpty()) {
        return;
    }

    Iterator i = animAtoms.iterator();

    while (i.hasNext()) {
        ((AnimAtom) i.next()).setVisible(visible);
    }

    repaint();
}

public void showAll() {
    setAtomsVisibility(true);
}

public void hideAll() {
    setAtomsVisibility(false);
}


public void paintChildren(Graphics g) {
    // First let parent do its work.
    super.paintChildren(g);

    // Early exit if we have nothing to do.
    if (animAtoms == null || animAtoms.isEmpty()) {
        return;
    }

    // Clear background.
    g.setColor(getBackground());
    g.fillRect(0, 0, getSize().width, getSize().height);
```

```
    // Advance to the more sophisticated Graphics2D object.
    Graphics2D g2d = (Graphics2D) g;

    // Set the preferred rendering hints.
    g2d.setRenderingHints(renderingHints);

    // Translate over vector (-startX, -startY) to reflect
    // origin offset in userspace.
    g2d.translate(-startX, -startY);

    // Scale according to specified zoom factors.
    g2d.scale((double)zoomPctX/100.0, (double)zoomPctY/100.0);

    // Iterate over all animAtoms.
    Iterator i = animAtoms.listIterator();
    while (i.hasNext()) {
        // Get the next AnimAtom.
        AnimAtom curAtom = (AnimAtom) i.next();

        // If current atom is visible, draw it.
        if (curAtom.isVisible()) {
            Color color = (Color) colorMap.get(curAtom.getColorKey());
            curAtom.draw(g2d, this, color);
        }
    }
}


/**
 * Create a new animation atom on the painter.
 *
 * @param animID the unique ID of the atom.
 * @param type the type of atom to create.
 *
 * @exception IllegalArgumentException if type is illegal.
 *
 */
public void createAnimAtom(int animID, String type) {
    AnimAtom atom;
    if (type.equals(ARC))            atom = new ArcAtom();
    else if (type.equals(ELLIPSE))   atom = new EllipseAtom();
    else if (type.equals(IMAGE))     atom = new ImageAtom();
    else if (type.equals(LINE))      atom = new LineAtom();
    else if (type.equals(RECTANGLE)) atom = new RectangleAtom();
    else if (type.equals(TEXT))      atom = new TextAtom();
    else throw new IllegalArgumentException("Unimplemented atom: " + type);
    addAtomImpl(animID, atom);
}

private AnimAtom getAtom(int atomID) {
    AnimAtom atom = (AnimAtom) animAtoms.get(atomID);

    if (atom == null) {
        throw new IllegalArgumentException("no such AnimAtom: " + atomID);
```

```
    }

    return atom;
}

public void setAtomLocation(int atomID, int x, int y) {
    getAtom(atomID).setLocation(x, y);
}

public void setAtomColor(int atomID, String colorKey) {
    getAtom(atomID).setColorKey(colorKey);
}

public void setAtomSolidness(int atomID, boolean solid) {
    getAtom(atomID).setSolid(solid);
}

public void setAtomVisibility(int atomID, boolean visible) {
    getAtom(atomID).setVisible(visible);
}

public Object getAtomProperty(int atomID, String propertyName) {
    return getAtom(atomID).getAnimProperty(propertyName);
}

public void setAtomProperty(int atomID, String propertyName, Object newValue) {
    getAtom(atomID).setAnimProperty(propertyName, newValue);
}

public void registerColor(String path, Object key, Color defaultColor) {
    colorMap.put(key, defaultColor);
}

public void setConfigPath(String configPath) {
    this.configPath = configPath;
}

class ColorIcon implements Icon {
    int w;
    int h;
    Color color;
    public ColorIcon(Color c) {
        this(c, 12, 12);
    }
    public ColorIcon(Color c, int w, int h) {
        this.color = c;
        this.w = w;
        this.h = h;
    }
    public void paintIcon(Component c, Graphics g, int x, int y) {
        Color oldColor = g.getColor();
        g.setColor(color);
        g.fill3DRect(x, y, getIconWidth(), getIconHeight(), true);
        g.setColor(oldColor);
```

```
        }
        public int getIconWidth()  { return w; }
        public int getIconHeight() { return h; }
    }
}
```

Appendix E

The GenericAnimator class

```
package tide.tools.animviewer;

import aterm.*;
import tide.debug.*;
import tide.tools.*;
import java.awt.Color;
import java.util.*;

public class GenericAnimator implements AnimatorTemplate
{
    /** Unique ID of each animation object we create. */
    private int animID;

    /** The AnimCommandHandler that handles our animation commands. */
    private AnimCommandHandler acHandler;

    /** The expression this animator is visualizing. */
    private String expr;

    /** The current state this expression is in. */
    private String state;

    /** The current value of the expression. */
    private String value;

    /** The colors used throughout the different states. */
    private Map colorMap;

    /** The ID of the label we use to display the expression and its value. */
    private int labelID;

    public GenericAnimator(String expr) { this(expr, null); }

    public GenericAnimator(String expr, AnimCommandHandler acHandler) {
        this.animID     = 0;               // first ID to dispense is 0.
        this.acHandler  = acHandler;
        this.expr       = expr;
        this.state      = STATE_UNKNOWN;
```

```
    this.value      = null;
}

public String getState() { return state; }

protected synchronized final int dispenseID() { return animID++; }

protected void fireCommand(AnimCommand cmd) {

    // Early exit if there is no acHandler.
    if (acHandler == null) return;

    // Issue the animation command.
    acHandler.issueAnimCommand(cmd);
}

private void registerColors() {
    Iterator iter = colorMap.keySet().iterator();
    while (iter.hasNext()) {
        String key = (String) iter.next();
        Color value = (Color) colorMap.get(key);
        fireCommand(CommandFactory.cmdRegisterColor(key, value));
    }
}

protected void initAnimator() {
    // Setup the default colors we will be using.
    colorMap = new HashMap();
    colorMap.put(STATE_UNKNOWN, new Color(100, 100, 255));  /* blueish */
    colorMap.put(STATE_VALUE,   new Color(  0, 255,   0));  /* green   */
    colorMap.put(STATE_ERROR,   new Color(255,   0,   0));  /* red     */
    registerColors();

    // Setup X- and Y-ranges.
    fireCommand(CommandFactory.cmdRangeX(0, 500));
    fireCommand(CommandFactory.cmdRangeY(0,  30));

    // Setup label to display the expression and value.
    labelID = dispenseID();
    fireCommand(CommandFactory.cmdCreate(labelID, "ATOM_TEXT"));
    fireCommand(CommandFactory.cmdSetLocation(labelID, 10, 20));
    fireCommand(CommandFactory.cmdSetColor(labelID, STATE_UNKNOWN));
    fireCommand(CommandFactory.cmdSetProperty(labelID, TextAtom.MESSAGE,
        expr + ": <unknown>"));
    fireCommand(CommandFactory.cmdSetVisible(labelID, true));

    // Request a repaint.
    fireCommand(CommandFactory.cmdRepaint());

    // Done with initialization, state of expr is unknown.
    state = STATE_UNKNOWN;
}

public void setValue(String newValue) {
```

```
        boolean needRepaint = false;

        if (state != STATE_VALUE) {
            state = STATE_VALUE;
            fireCommand(CommandFactory.cmdSetColor(labelID, STATE_VALUE));
            needRepaint = true;
        }

        if (newValue != value) {
            value = newValue;
            fireCommand(CommandFactory.cmdSetProperty(labelID,
                TextAtom.MESSAGE, expr + ": " + value));
            needRepaint = true;
        }

        if (needRepaint)
            fireCommand(CommandFactory.cmdRepaint());
    }

    public void setError(String newValue) {
        boolean needRepaint = false;

        if (state != STATE_ERROR) {
            state = STATE_ERROR;
            fireCommand(CommandFactory.cmdSetColor(labelID, STATE_ERROR));
            needRepaint = true;
        }

        if (newValue != value) {
            value = newValue;
            fireCommand(CommandFactory.cmdSetProperty(labelID, TextAtom.MESSAGE,
                expr + ": " + value));
            needRepaint = true;
        }

        if (needRepaint)
            fireCommand(CommandFactory.cmdRepaint());
    }
}
```