KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
KANDIDATUURCENTRUM
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A – B-3001 Leuven

VISTO: A DECLARATIVE METHODOLOGY FOR GRAPHICAL USER INTERFACES, BASED ON HASKELL

Promotor:
Prof. Dr. ir. K. De Vlaminck

Proefschrift voorgedragen tot
het behalen van de graad van
doctor in de wetenschappen

door

Kris AERTS

mei 2001

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
KANDIDATUURCENTRUM
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200A – B-3001 Leuven

VISTO: A DECLARATIVE METHODOLOGY FOR GRAPHICAL USER INTERFACES, BASED ON HASKELL

Jury:
Prof. dr. B. Demoen, voorzitter
Prof. dr. ir. K. De Vlaminck, promotor
Prof. dr. ir. E. Duval, assesor
Prof. dr. ir. F. Piessens
Prof. dr. M.Sc. J. Vanderdonckt
(Université Catholique de Louvain)

Proefschrift voorgedragen tot
het behalen van de graad van
doctor in de wetenschappen

door

Kris AERTS

U.D.C. 681.3∗H52

mei 2001

D/2001/7515/4

ISBN 90-5682-292-6

Visto: A Declarative Methodology for Graphical User Interfaces, based on Haskell

Kris Aerts
Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract

Graphical user interfaces (GUI) are an important part of modern computer applications, requiring specific programming techniques. Although for the standard application programming declarative alternatives – functional and logic programming – have been developed, there is no such thing for graphical user interfaces. Starting from our experience in functional programming, Visto tries to fill in that empty space.

The end product of our research is a programming language with an operational implementation, with contributions in various fields of user interface implementation.

We first illustrate the historical trend in GUI toolkits to provide widgets with more and more default behaviour. Our first contribution, *selectors*, continues that path and offers an abstraction for a common subgoal of user interfaces, i.e. *selecting* a value. Different kinds of selectors, such as one-from-many and many-from-many, have been implemented in Fudgets[16] and TkGofer[88]. An important characteristic of these selectors is that they can be instantiated with various appearances, such as button rows, pull down menus, slider bars, ...

A second contribution, not directly in the field of user interfaces, is the development of Visto (Visual State Transforming Objects), an object oriented, prototype based system on top of the functional language Haskell[62], with different intuitive forms for deriving new objects from existing ones.

This object system is enriched with features for graphical user interfaces, such as the automatic redrawing of views.

We have also developed a variation of the *Model-View-Controller* pattern, which is typically applied to separate the different responsibilities in a GUI program. This variation is most apparent in the *Controller* part, where we present an alternative, GUI invokers, for the traditional call back functions. These GUI invokers, together with the selectors, describe the functionality of the user interface, i.e. they describe for each method of each object which GUI invokers can activate the method and which selectors provide the arguments for the method.

This can be considered as a mere description, but it can also be used as an implementation when the invokers and selectors are instantiated to a standard appearance as provided by Visto.

In the case study, where we implement a small application in a number of different GUI toolkits, we show that the Visto program is easy to develop and that it can swiftly handle changes to the user interface, more easily than the other systems can.

Contents

Preface

The booklet you are now holding contains the fruits of many years of work. As you read through the text, you will discover many of the interesting paths I walked upon during the development of this thesis.

Although this work, as most Ph.D.'s, has been solitary in many moments, I couldn't have completed this challenge without the help and encouragements of many people. We gladly make use of this preface to thank all these people.

Contrary to the rest of this text which is written in English, we prefer to use Dutch for the body of this preface, as words of respect and gratefullness can best be expressed in ones mother tongue . . .

Het behalen van een doctoraat is een werk van lange adem. Het is geen spurt, maar veeleer een bergrit met hoogten en laagten, met snelle afdalingen waar het onderzoekswerk met rasse schreden vooruitschrijdt, en tragere beklimmingen waar de resultaten pas na veel zwoegen zichtbaar worden. En op het einde wacht de hoogste berg, in wolkensluiers gewikkeld. Elke bocht ziet er uit als de laatste, maar verbergt in feite een nieuwe beproeving.

Wanneer dan, na de lange rit, de aankomst bereikt wordt, overheerst zonder twijfel, net als bij een echte bergrit, de voldoening en de zoete smaak van de overwinning. Op momenten als deze willen we deze vreugde delen met zoveel mogelijk mensen en vooral met allen die hieraan meegewerkt en meegeleefd hebben.

In de eerste plaats denk ik hierbij aan de leden van de jury, die allen met veel diepgang de voorlopige versies mijn proefschrift gelezen hebben, wat reeds wijst op een sterke interesse in mijn werk. Bovendien ben ik zeer blij met de samenstelling van de jury, waarin de verschillende specialisaties zoals user interfaces, object orientatie en declaratieve talen, de verschillende aspecten van mijn onderzoek goed vertegenwoordigen. Bovenop de algemene dankbetuigingen wil ik Jean Vanderdonckt danken voor zijn enthousiasme op CADUI, de grondigheid waarmee hij mijn werk gelezen heeft en daardoor een stempel heeft weten te drukken op de inhoud van dit proefschrift; Frank Piessens voor de manier waarop hij zijn gemeende interesse en enthousiasme ten toon weet te spreiden; Bart Demoen voor de snelheid waarmee hij mijn voorlopige teksten na kon lezen, voor de heldere opmerkingen en voor het feit dat hij mee probeerde de timing op te volgen; Erik Duval voor de gesprekken waardoor ik mijn onderzoek beter kon situeren in de rest van het user interface onderzoek en voor de diepere inzichten in mijn persoonlijk werk die ik daardoor kon verwerven; mijn promotor Karel De Vlaminck apprecieer ik vooral voor de vrijheid om dit proefschrift te mogen schrijven in mijn typische, verhalende stijl, en voor het geloof in mijn capaciteiten, als onderzoeker en als onderwijskundige.

Daarenboven heb ik met veel plezier kunnen samenwerken met de collega's bij het departement computerwetenschappen. Het MI-team wil ik dan ook danken voor de gezellige sfeer op de vele vergaderingen en het meeleven tijdens elkaars zware beklimmingen. Zonder dat ik de andere mensen minder wil bedanken, wil ik toch een extra woordje richten tot Dirk, Wim, Peter en Yvan die met mij in de A03.43 gezeten hebben. Zonder het misschien zelf te beseffen hebben ze mij in hun eigen stijl en met hun eigen karakter verder geholpen.

Natuurlijk mag ik ook de collega's monitoren niet vergeten. Zonder de welkome afwisseling van het onderwijs en de omgang met studenten had ik deze koers nooit kunnen uitrijden. Álle mensen die samen met mij op het kandidatuurcentrum werkten, hebben er samen voor gezorgd dat ik met veel plezier kan terugblikken op deze tijd. In het bijzonder wil ik Petra en Hilde bedanken voor de goten gezelligheid en de uitbabbelmomenten.

Vanzelfsprekend wil ik ook allen die nog bezig zijn met de vele beklimmingen op weg naar het doctoraat veel sterkte toewensen, en diegenen die voortijdig moeten afhaken geruststellen met de gedachte dat de sneeuw aan de andere kant van de berg niet noodzakelijk witter is.

Een bijzonder woord van dank gaat ook uit naar de mensen van de KHLim voor het getoonde vertrouwen en voor de kansen om in dit eerste jaar bij hen mijn doctoraat verder af te werken. Hierbij wil ik ook zeker Lieve Lemmens niet vergeten voor haar persoonlijke inspanningen om nog meer tijd voor mij vrij te maken en voor het warme onthaal in Diepenbeek.

Behalve de vele mensen in het professioneel leven mag ik zeker vrienden en familie niet vergeten. Hier hoef ik geen namen te noemen want zij weten beter dan geen ander dat al deze dankbetuigingen ook voor hen gelden, niet zozeer om de ondersteuning bij het doctoraat zelf, maar in de eerste plaats voor het samenleven in het leven buiten het doctoraat, en in de vele plezante momenten. Mijn ouders verdienen een extra woordje voor de impliciete en expliciete steun die ik mag genieten in al mijn ondernemingen.

Ten slotte wil ik een aparte alinea wijden aan mijn tweede ik, Liesbet. Als geen ander heeft zij meegeleefd in deze lange rit, als geen ander voelen wij samen wat wij moeten voelen, als van geen ander geniet ik haar steun, als geen ander hou ik van haar ... Dank u voor dit gelukkig samenzijn!

Chapter 1

Mission Statement

No Good Reason is
a Perfectly Good Reason

1.1 Introduction

In this thesis we present our *search for the holy grail of declarative programming* in the field of designing and implementing graphical user interfaces (GUI). The comparison with the *holy* search is adequate because

- there is no well agreed definition of *declarative*. It is based largely on *belief* and is subject to interpretation.
- The search never ends. GUI's can be very complex. We may never succeed in finding an ultimately declarative way for describing such complex behaviour. At the same time user interfaces are still strongly evolving because the enhanced hardware capabilities allow for features that could not be realised with older hardware e.g. real time dragging of entire window contents, or three dimensional interfaces in virtual 3D helmets, and because developers keep on building graphically more enticing implementations of existing ideas, along with the occasional new metaphor.
  So any general research, and certainly individual Ph.D students, will inevitably be unable to keep up with all the current advancements.

However we feel that we have revealed some interesting declarative paths which will be presented in the following chapters. And in the very least, just as the great movie by Monty Python, this (re)*search for the holy grail* provided us with pleasurable moments. In this thesis we merely present the results.

Overview

The style of this thesis reflects the personality of the writer. Therefore, from the second chapter on, the story of our research results is written down in a not too formal style. Unfortunately, as English is not our mother tongue, the final text is not as fluent and ad rem as we would have liked. Nevertheless we hope that you enjoy reading this text and are attracted by the different propositions that we make.

This first chapter is slightly different from the other chapters in the sense that the discussion in this chapter is slightly more formal and that it covers the assumptions and choices used throughout this thesis, or, put in marketing terms, this first chapter contains the *mission statement* of our thesis.

In the first section we present the main issues of this research project, its goals and its application domains. The second section briefly summarizes current trends in user interface research.

Then, in the third section, the basic assumptions of our research project are presented along with a discussion of the basic choices and its alternatives. Finally, in the last section of this chapter, we give a rough outline of our suggested methodology, and refer for a more detailed reading to the rest of this thesis while giving an overview of the main contents of each of the following chapters.

Throughout this entire thesis, and specifically in this chapter, we motivate our choices with more or less objective criteria. However note that in our opinion research should have some sense of unpredictability, often being influenced just as much by coincidence as by objective, motivated arguments. Therefore we feel that *no good reason is a perfectly good reason* as well. Researching something that 'should' be not searched is often disappointing, but may also lead to unexpected novelties.

1.2 Main Issues and Context

This thesis shifts continually between two positions: improving Haskell and improving user interface (UI) development. We started mainly in Haskell and ended mostly in the user interface community.

The improvement we aim to bring to Haskell is a broader application: programs in Haskell greatly benefit from the expressiveness of a functional, declarative language, resulting in shorter and more concise programs that can often be proved correct. However, programming a user interface in Haskell used to be more difficult than application programming because the necessary side effects are hard to express in a pure language, i.e. a language without side effects.

Therefore our Haskell objective is defining an extension of Haskell that allows for a smoother definition of user interfaces. The current solutions such as TkGofer and Fudgets fail in connecting the application with the user interface in a declarative way. They are more programming oriented than model oriented despite the fact that the UI community currently considers model based approaches the right thing to do. Although still offering a programming language, Visto, an acronym for **Vi**sual **S**tate **T**transforming **O**bjects, also implements a particular model oriented methodology and therefore brings the functional programming community closer to the UI model based community.

On the other hand, we try to improve research in the user interface community. This research is characterised by model based approaches. Being typically very high level, they are very expressive but often hard to master. They require specific craftmenship, often unavailable to the average programmer who stays with his traditional programming language.

Not so long ago automatic generation of user interfaces was the nec plus ultra, a feature that was often present in model based tools. In such tools the user specifies a model, and the tool generates a user interface from it. Relying solely on a single model has now been more or less left because most of the times the generated user interface is hard to change and often results in a rather static interface.

Visto implements a model based approach as well, but it is intentionally programmer oriented as we want to give the programmer more control over the actual user interface than he has in generated interfaces.

So we don't want to be classified in one extreme of the spectrum, that of automatically generated user interfaces, but neither do we want to be classified in the other extreme: that of traditional user interface programming using widgets and specific call back functions. Such call back functions suffer from being widespread across the program, often resulting in a real spaghetti of call backs.

Also, the call backs obfuscate the intention of the user interface: they state what happens when some widget is used, but do not clearly indicate the set of possible actions that the user interface can trigger, e.g. it is hard to see which widgets invoke the same action. The orthogonality of many user interfaces is poorly expressed by such call backs.

As we position ourselves somewhat between the extreme, purely model based approaches in which no programming remains present at all, and the extreme programming of the entire user interface, our intended application domain is in between as well.

We don't intend to define a revolutionary new domain theory, nor replace entire toolkits, but rather aim at basic interfaces and principles of user interfaces. We have worked broadly, providing both elements of (early) design and aspects of implementation: what widgets or abstractions over widgets to use, and how to place them on screen on the one hand, and on the other hand how to connect the application layer with the interface in a high level, descriptive, but nevertheless executable language.

Bringing those elements together in a whole system, we claim to have developed a declarative methodology for graphical user interfaces (GUI), based on Haskell.

Our main beneficiaries are the implementors of user interfaces. Even if Visto isn't used for the final implementation of the GUI system, that resulting system can still benefit from the clear description of the behaviour of the GUI. Although not (yet) as powerful as professional toolkits, the implementation of a user interface in Visto is straightforward and can be used for rapid prototyping, either to have a preview on a final system, or for systems that don't need super cool, ultra dynamic interfaces, such as software used in a research context, a situation where Haskell is a very nice choice just as well.

After having used Visto as a prototype, the Visto code can be retained as a formal description of the characteristics of the interface, helping in a better maintainability of the final program.

Summarised as bridging the gap between UI and Haskell research, these are in short and in rather general terms the main issues of our research project.

1.3 Current Research Trends

There exist many techniques to develop a (graphical) user interface. This section gives a broad overview of them, along with some other techniques that are related to our research. This certainly includes a discussion of the notion of *declarative* programming that is strongly advocated by this thesis.

1.3.1 User Interface Techniques

The tools for constructing user interfaces can be divided into five basic categories: interface builders, programming languages, demonstrational tools, user interface management systems (UIMS) and model-based tools. [79, 78]

Interface Builders

Interface Builders such as Visual Basic and the NeXT Interface Builder [55] are the most popular interface construction tools in the consumer market. They provide an easy to use, WYSIWYG interface for constructing interfaces where designers can draw the screens of the interface. They are specifically worthwhile for quickly developing small, individual interfaces, as they provide essentially a single modeling construct: instantiation. Designers can create instances of a predefined set of classes of interface building blocks, and set the value of their parameters.

This is at the same time their main disadvantage as they use a very low level description of the interface consisting mainly of the location, property and type of all the elements of the display. This is not enough information to support sophisticated analytic tools for design evaluation and critiquing, tools for retargetting the interface to a new platform, or tools for automatically generating help. Therefore designers who work in such systems are quickly faced with the limits of applicability of this approach.

Programming Languages

Programming languages, certainly the lower level programming languages such as C++, are the most flexible means for developing a UI. They allow programmers to create arbitrarily sophisticated interfaces by writing programs, mostly using UI toolkits, sometimes even from scratch. As a matter of fact, the programmer defines a procedural model of an interface rather than the conceptual idea of an interface.

Defining an interface in a traditional programming language is very difficult and prone to bugs. Visto is for a main part a programming language as well, but the interface is defined more conceptually. That way we hope to benefit from the conceptual advantages present in model-based tools (see below) and reduce the programming difficulty.

The other user interface systems for functional programming languages, besides Visto, all fall in this category, often relying on advanced programming techniques. These functional UI systems are introduced in the appendix and discussed and compared with Visto in the main text of this thesis, mainly in sec. 2.4 and 3.12.

Demonstrational Tools

Some research has been done in this field (e.g. Lapidary [54] and Druid[74]), but a major breakthrough hasn't been announced (yet) [53].

These systems try to use the power of demonstrations: they either demonstrate scenario's to the designer and user of the system who can then choose between those scenario's and let the system derive an interface for such a scenario, or either let the designer demonstrate the intended behaviour of the interface. After a number of such demonstrations the tool generates a generalisation of the examples which can then serve as a basis for the interface. However, the tools rarely help the designer understand the relationship between features of the examples and features of the generalisation. Hence, designers find it hard to understand the generalisation [78].

UIMS's

UIMS's [73] are user interface construction tools used mainly to construct the dialogue components of the interface. They use specialized languages to describe the dialogue (transition networks, grammars or events). Therefore they mostly require advanced training before they can be used.

Model Based Tools

Model-based user interface development tools trace their roots to work on the UIMS's from the previous section. In the early 90's the UIMS specification languages became more sophisticated, supporting richer and more detailed representations that allowed the systems to generate more sophisticated interfaces. Today's model based interface development tools use specifications (models) of the tasks that users need to perform, and require the developer to define data models that capture the structure and relationships of the information that applications manipulate, specifications of the presentation and dialogue, user models, ...

Early model-based systems supported only one or two of those models, but the currently most advanced systems integrate many of them. Even though model based tools are much more sophisticated than early UIMS's, they have not become popular in the commercial, mainstream sector. Most software developers still use interface builders, toolkits and a programming language with call back functions to build their user interfaces.

Visto incorporates model based features in a programming language, thereby hoping to slowly convince the programmers of (some of) the advantages of model based approaches.

An excellent description of characteristics of model based tools can be found in *Retrospective and Challenges for Model-Based Interface Development* [80]. The introduction of CADUI99 [86] contains an overview of the different models that are used.

The task model that describes the tasks the end-user has to perform is currently considered the main model from which a user interface development should be started. An extensive task model uses goals that specify what desired state must be reached, and methods that describe how to achieve such a goal.

Some more models, that are also partially used in Visto, are

- *the data model* that describes the semantics of the data that is to be manipulated throughout the UI,
- *the control model* that specifies the behaviour of an application. This model is mostly object oriented as objects capture the state of the entities in the program, and the methods capture the actions that are possible on those objects,
- *the presentation model* that specifies the appearance of layout of the different components of a user interface.

Systems stressing the model based approach more than Visto also define models for the behaviour of the system, the conversations, the environment, the platform, the user, ...

1.3.2 General Programming Techniques

Besides the user interface aspects we gave an overview of in the previous section, we also use prototype based object orientation and declarative programming.

Prototype Based Object Orientation

Object orientation as present in languages such as SmallTalk, C++ and Java, is probably by now a well known concept. Most mainstream object oriented languages are class based, i.e. the concept of *class* is the centre of attention. No object can exist without a class definition. Such a class definition describes the methods that can be applied to objects of the class, as well as the instance variables (data members) of the individual objects. When an actual object is constructed, the object's data members are properly instantiated and the object is then ready to receive messages (method invocations).

Because the methods are defined in the class, all objects of the same class behave identically, only depending on the values of the internal data members.

This identical behaviour is precisely the goal of the class based approach. It is aimed at systems in which a large number of identical objects exist, which is surely the case in many applications. A library for example will control a considerable amount of persons and books. A class based implementation will therefore define classes Person and Book, and have many instances of those classes.

On the other hand, when only very few instances of objects exist, the class based approach is not so practical, because it introduces an extra layer of indirection. When designing a new object, one must first move to the abstract level of a class, write a class definition, then instantiate it and test it, rather than remaining at one level, incrementally building an object (fig. 1.1).

It is also unlike our way of working in the real world. We don't think in terms of abstract artifacts, but rather use concrete building blocks. A cake is baked using real flour, eggs and sugar, not using instances of abstract ingredients.

And when Picasso painted his most famous master pieces, he wasn't first designing an abstract class MasterPiecePainting, but rather immediately painted on his canvas.

This is what prototype based object orientation does. We instantly create objects and derive new objects by copying and expanding from existing objects.

It's a bit like the difference between writing an assembly guide for an automobile (defining the class) or assembling the automobile. If we want to make lots of cars we'd rather first define a construction guide, perhaps build some moulds and specialised robots as well and then build thousands of cars according to the guide and using the moulds and robots. Conversely when we want to build a single, individual tool – a prototype – we'd rather build it at once instead of constructing complicated and expensive moulds.

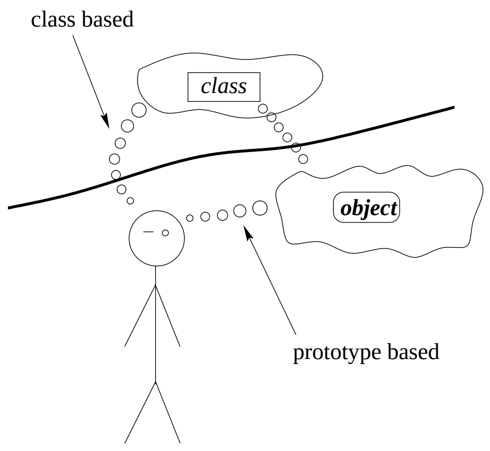


Figure 1.1: A More Direct Handling in Prototype Based Languages.

Both class based and prototype based languages offer mechanisms for re-use, the main difference being that class based languages reuse and extend class definitions, whereas prototype based languages directly reuse and extend objects.

Another interesting thing to mention is the fact that there are fewer differences within the categorie of class based languages than within the categorie of prototype based languages. Although they have the same foundations, there seem to be little agreement on the way in which to support features known in the class based world as inheritance, late binding and delegation.

Noted examples of prototype based languages are Self [81], NewtonScript [77], Garnet [52] and Kevo [83]. For another discussion about classes vs. prototypes, read Borning's paper [12].

Programming by Example

Like the demonstrational tools in user interfaces, there also exists a programming by example idiom in general application programming. Some tools, such as Eager [21] and work by Girard [30] have been developed that try to use the same principles as in demonstrational tools, i.e. by letting the programmer give some examples, or by monitoring whether the user performs a number of repetitive tasks, the tool tries to generalise these examples into concepts.

This principle is mainly used in abduction theory, a subdivision of logic programming, where it is rather successfully applied to find implicit relations. However we are sceptical about the chance of advanced results in the near future in the general programming field.

However, we think that humans often learn to program that way, i.e. by studying examples of programs and copying and manipulating the interesting parts of these examples. Therefore we think that a programming language that supports such learning principles may provide benefits to the average programmer.

Prototype based object oriented programming comes close to programming by example in the sense that programmers can create new objects directly from existing examples.

Declarative Programming

The great adagium of declarative programming is that a program should focus on the *what* instead of on the *how*. If the program reflects in a more concise way *what* is to be calculated, it will be easier to maintain and expand because we can more easily identify the purpose of the software. Describing *how* to calculate the desired result obfuscates the intended goal and may be something we would rather let the compiler find out.

There may be no clear definition that says whether a system is declarative, but we can compare two systems and say that one is *more declarative* than another when it focuses more on *what* should be done and the other more on *how* it should be done. So a system that may be considered today as extremely declarative may not be so declarative in the future.

This definition is rather subjective which makes it hard to argue about the declarativeness of a system. However, people seem to agree that both *logic* and *functional* programming are the two main constituents of the *declarative* programming research community.

These declarative languages can best be contrasted with the *imperative* programming style. In an imperative language the *state* of the program is made up by the current set of live variables. The state is modified by side effecting operations (commands), such as the assignment (e.g. *x := 3;*). As a consequence the order of those state changes is very important. Imperative languages generally have a notion of *sequencing* to permit precise and deterministic control over the state. This is, amongst others, present in the *begin* ... *end* construct and loops as the *for* and *while*.

Declarative languages typically put full emphasis on programming with *expressions* (or terms). They have *no implicit* state. The value of an expression is completely determined by its *arguments* and not by some implicit state variables.

This is similar to a spreadsheet[1] where one specifies the value of each cell in terms of the values of other cells. The focus is on what is to be computed, not how it should be computed. For example:

[1]An example from *http://www.haskell.org/aboutHaskell.html*

- we do not specify the order in which the cells should be calculated. Instead we take it for granted that the results computed by a spreadsheet depend only on the dependencies between cells and not on the order of computation.
- we do not tell the spreadsheet how to allocate its memory. Rather, we expect it to present us with an apparently unbounded plane of cells, perhaps allocating memory only to those cells which are actually in use.
- for the most part, we specify the value of a cell by an expression (whose parts can be evaluated in any order), rather than by a sequence of commands which computes its value.

Also striking – and often initially annoying for imperative programmers moving to or trying out the declarative world – is the *absence* of explicit loop operators. The way we deal with iteration is *recursion*, where a function or predicate calls itself and thus initiates iteration.

Take for instance the typical *append* example from Prolog.

$$\begin{array}{l} append([],X,X). \\ append([X \mid Xs],Ys,[X \mid Zs]) :- append(Xs,Ys,Zs). \end{array}$$

We first state the fact $append([],X,X).$: the append of the empty list [] with X is equal to X. Then, in the second line we have a list $[X \mid Xs]$ of a first element X, followed by the rest of the list, named Xs, and a second list Ys. The append of both lists will be the list $[X \mid Zs]$ (X as the head (i.e. the first element) of the list, followed by Zs), *if* $append(Xs,Ys,Zs)$ is true. Explained in another way: if we have to append the non-empty list $[X \mid Xs]$ and $[Ys]$, let Zs be the append of Xs and Ys and then put X in front of Zs to give $[X \mid Zs]$.

This can be read as an operational semantics as described in the second version, to calculate the append of two lists, but it can at the same time be used to test *if* a third list is the append of two other lists. This program thus expresses the *relation* between three variables of being each others append.

In this thesis we will stick with the 'other' declarative paradigm: *functional programming*. We more specifically use *Haskell* [62], named after the mathematician Haskell Brooks Curry. Although the foundations of functional programming go back to the 30's with the *lambda calculus* of Church [20], and Lisp, the prototypical ancient functional language, dates from the 50's, Haskell, the currently standard functional language, only saw the light as recently as 1987. After going through a number of revisions, the definition finally stabilised with the advent of *Haskell 98*. *http://www.haskell.org* contains a large collection of resources. Our appendix also contains a short introduction to Haskell.

If it was rather easy to contrast the declarative programming languages with the imperative ones, it is more difficult to claim in general when something is declarative.

We already mentioned the fact that declarativeness should always be considered in comparison with another system. We can then compare these systems and claim that one is more declarative than the other.

Table 1.1 contains a number of criteria for identifying whether a system is declarative. The most important criterion is that a declarative system describes more *the what* and the less declarative system more *the how*.

Therefore a declarative definition of a button behaviour states that by clicking the button this or that method is executed. A less declarative system might say that by pressing the button some specific event is triggered, and that we need to define a function that looks for such events, e.g. by watching at some specific bit in memory and executes a function when that bit is set.

not declarative	*declarative*
• description of how	• description of what
• procedural oriented	• goal oriented
• needs understanding of underlying mechanism	• don't need to know how it is done
• aimed at specific situation	• general

Table 1.1: Criteria for Declarativeness

This is often most apparent in the orientation of the system: a declarative system is more likely to state the goal instead of the procedures that must be followed to accomplish the goal.

Naturally it follows that users of a declarative system generally don't need to know how the goal is met; the system is free to implement it in any suitable way. In a less declarative system we have to precisely know what is happening, as we have to define all the individuals steps ourselves.

It is often also the case that the less declarative system is aimed at specific hardware, e.g. procedural languages such as Pascal and C are explicitly intended to be used on Von Neumann architectures. The declarative system mostly makes no assumptions about the hardware it is used on. Therefore it is more suitable for portable applications.

To conclude, for a system to be considered more declarative than another system, it can either score higher on some of the criteria or score on more of the criteria we just mentioned. So declarativeness is no absolute definition: a system can be more declarative than another on some criteria, but also less declarative on other criteria.

Therefore we must try to identify the individual merits of our system, as we will do in the following chapters of this thesis.

Specification Language

A specification language in the strict sense of the word is a language in which formal specifications can be developed. Specifications form an important part of software, as they specify formally the behaviour of parts (functions, modules, objects, ...) of a software application.

Some languages provide very broad support for specifications, including pre- and postconditions, but most mainstream programming languages support at best signatures or type definitions of the functions present in the program.

Good specifications clearly have a strong declarative aspect, as they describe the intended use of the functions, and not their implementation. However, a specification language in the strict sense doesn't contain implementations. Such specifications are not really declarative as they cannot be executed.

However, executable specification languages combine the best of both worlds. Code in Haskell, our base implementation language, often comes close to such executable specifications with its short, mathematically sound notation.

1.4 Assumptions and Choices

In this section we describe a number of important decisions that influenced the design of Visto. This is not a discussion of details such as the precise syntax of Visto commands, but rather about the very basics of our system, e.g. why combining functional and interface programming, why prototype based and not class based objects, ...

1.4.1 Why User Interface Research

This is the first, and most important question to answer. When we started this project, typical research in our functional programming group involved partial evaluation and related program analyses. At the same time the need for practical applications of functional programming started arising in an attempt to prove the world the fact that functional programming is a powerful (alternative) tool for general application development, and not just a research curiosum.

It is without question that user interfaces, and especially *usable* user interfaces form a very important field in computer science and thus are an important *must-have* if we want to develop powerful applications in Haskell. In a broader sense they are typically covered by the term Human-Computer-Interaction, HCI for short, which is the study of how humans interact with computers, and how to design computer systems that are easy, quick and productive to use by humans.

We can safely claim that better user interfaces brought computers closer to the human kind. More than the reduced costs for buying a computer, the enhanced accessibility because of more user-friendly interfaces accelerated the advent of the computer age. The computer could only find its way into nearly every home because it is so easy to handle – for most people –.

Graphical user interfaces are a lot easier to use than command line interfaces that require the user to know the precise grammar and syntax of all the possible commands. Therefore GUI's prevent a lot of these errors, resulting in the user feeling more in control of the application and more in control of the computer.

Moreover, as most interfaces are similar to each other, new GUI applications can be learned very easily, allowing users to perform many different and new tasks without the hurdle of a steep learning curve.

The reduced error-rates and increased productivity resulted in a greater task-satisfaction and less stress, although, on the other hand, employers often have high expectations of computer experts, which induces more stress again.

Anyhow, it is without question that because of increased computer power and better user interfaces, many people can now accomplish tasks, such as desktop publishing, that used to be available to expert users only.

Commercial aspects are important as well. The user interface of an application plays a vital part for consumers in deciding whether to buy this or that application. The user interface on its own has become one of the most important marketing tools, especially for the consumer market.

Conversely, creating a well designed user interface is an enormous task. Despite the huge number of papers related to HCI, it seems that user interface design and implementation is inherently difficult and that we will be stuck with this situation for a long time to come. Brad A. Myers [51] makes a convincing case for that claim.

As an immediate consequence creating user interfaces is very expensive: Myers has shown that over a wide class of program types, machine types and tools used, the percentage of the design time, the implementation time, the maintenance time and the code size devoted to the user interface was about 50%. Until very recently with the advent of *discount usability engineering*, even a single usability test could cost from a few thousands euro to easily over 100.000 euro.

One of the many reasons why GUI programming is so difficult is the diversity of the users. Their behaviour can be expected to be almost unpredictable. A programmer may build a function that only takes a fixed set of values, but the user interface should always be prepared to accept *any* value, and if necessary, warn the user in a proper and polite way, giving him another chance to enter a proper value. We should try to develop interfaces that prevent errors, and that are fool-proof (but we should also avoid interfaces that can only be used by fools).

Moreover, interactive systems are complex and dynamic structures that have many properties that make any system difficult to implement: multi-processing, robustness, undo, redo and abort actions, and several real-time requirements.

The declarative programming community has a growing number of user interface toolkits, both for logic and for functional languages. From our personal background we are more aware of the functional toolkits.

Unfortunately it turns out that most of these toolkits are not so *innovatively declarative*. As we will explain in our chapter on selectors, they provide declarative constructors for concrete widgets, but essentially keep on relying on those widgets and on the basic principles of traditional, imperative user interface programming, i.e. call backs. Therefore the systems that can be built using these functional toolkit suffer from the same problems as faced in typical imperative GUI-toolkits: the hundreds or thousands of widgets each have their own call back which makes maintenance very hard. These toolkits can often be considered rather direct translations – transfers – of imperative toolkits or as (straightforward) *bindings* to such an imperative toolkit, e.g. TkGofer [88].

To our knowledge Fudgets [16] is the only approach in functional programming – it certainly was the first – to offer an alternative for call backs. Call back mechanisms have actually disappeared from this toolkit. The connection between the user interface and the application goes through pre- and post-processors that are attached to input and output streams of fudgets. Unfortunately this poses important plumbing skills on Fudget programmers. Experiments of some of our graduate students showed that Fudgets are great for smaller applications, but rather complex for larger ones, which, on the other hand, – due to the inherent complexity of GUI design and implementation – may be quite normal. Anyhow, it also turns out that changes to a Fudget user interface can be quite difficult because of the probable necessary re-wiring of fudget streams.

Therefore we feel the functional programming community can still benefit from a better, more declarative user interface system.

1.4.2 A Declarative Approach for User Interfaces

Having discussed why we decided to define (yet) another UI system for functional programming, there remains a question about the added value of declarative user interface programming.

It seems that many powerful solutions to the problems of UI design have been found as very powerful user interfaces for a wide range of programs exist. The most popular approaches involve an *object oriented* framework with widgets (see chapter 2) and their *call back* functions that describe what should happen when the user operates the widgets.

However, certainly in the early days of user interface implementation, standard programming languages didn't provide an appropriate object system, as Apple invented ObjectPascal, and Motif and OpenLook for Unix hacked[2] together an object system *xtk* into C.

Although call back procedures are convenient for describing specific widget behaviour, they actually tend to make maintenance very hard. Usually, each widget (such as menus, scroll bars, buttons, ...) requires the programmer to supply at least one application procedure to be called when the user operates it. Each type of widget will have its own calling sequence for its call-back procedures. Since an interface may be composed of thousands of widgets, there are thousands of these call backs, which tightly couples the application with the user interface and creates a maintenance nightmare. At least two research papers [8, 50] mention this problem openly in their title. We do not discuss the results of these papers here, as they replace one technicality with another, but it proves that other researchers are worried about the problems associated with call backs just like we are.

One of the explicit goals of Visto is to get rid of these call back functions. They are a typical example of a solution that has been developed with the machinery in mind. The scenario is as follows: when a button is defined, this corresponds to an area on screen. Whenever the mouse button is clicked when the mouse pointer is over the window, an event is generated in the machine. Therefore we need a function (the call back) that reacts on such an event and triggers the function. This scenario, and the solution with call backs, is 100% computer oriented and not human oriented.

In fact, the user thinks in a completely different way, i.e. goal oriented: when he wants to do something, he tries to find out what is needed to achieve that goal. It is only then that he starts looking at the buttons, menu items and other widgets. The center of attention is on the action, not on the widgets.

Declarative programming is goal oriented programming, and thus more concerned about the action, and less about the precise widgets. If we can define our program in the same way as the user thinks, we increase the chances that our program behaves as the user expects. Then the program itself becomes documentation of the different uses of the software, whereas currently the program is no more than a technical, machine oriented translation of how the user must initiate the actions of the program.

If we can stress the user actions, this is a clear advantage of the enhanced declarativity. We force the programmer to think about the actions his system must be capable of. Then, by adding GUI invokers to those actions, we let him think about the actions that the user should be able to invoke.

So, on the one hand we want to concentrate on the actions that the user interface facilitates, on the *what*. However, naturally, *how* to invoke something is often very important in user interface design. The look-and-feel and the handling of the objects used in the interface are an important criterion when creating a GUI. So

[2]Annotation due to Brad A. Myers [51]

although we still prefer a declarative approach for GUI's, stressing *what* is to be done, we are faced with the difficult task of reconciling the declarative aspect with the often strong desire for being able to precisely describe *how* to do something.

If we *only* pay attention to the declarative aspects of the user interface, the result will not be as good looking as can be expected from modern user interfaces. We must try to make a system in which the actions of the UI are described declaratively, but that also allows the programmer to define in a very precise way precisely *how* something must be done, and in what visual manner. Combining these two contradicting concerns is a major challenge of declarative user interface research.

1.4.3 Declarative Programming: Functional or Logical?

The two main streams in declarative programming are functional and logic programming. A functional program is made up by composing different functions into one big function *main*. A logical program is made up of *predicates*. A result is built by choosing the predicates that succeed and instantiating the logic variables in the program.

In our opinion, it turns out that logical programming is more successful in certain specific domains, such as constraint logical programming and abduction, but not so much in general application programming, contrary to functional programming which is really intended to be used as a general application programming language. This observation was already a first indication for us to prefer functional programming.

Other advantages of functional programming include provable correctness, strong typing and concise, readable and maintainable code [37]. Especially the strong typing used to be absent in logical languages, although Mercury now supports it as well.

Moreover, in a functional program there is also a clear program flow. A function accepts *input* arguments and *outputs* return values. In a logical program there is no standard program flow: the evaluation mechanism supports largely on the instantiation of variables. We can either let the program calculate an instantiation (i.e. calculate a result) or test whether a certain instantiation is correct (i.e. test whether some result, e.g. the append of two lists, is correct). (cfr. the append function in sec. 1.3.2). This makes logic programming a good choice for a certain aspect of traditional imperative UI programming: when a value in an input field has been entered, the corresponding (mutable) variable is updated.

However, this is not what the user expects: he doesn't mind if a variable is set when he enters something or not; he'd rather expect the program to use it in an action he's initiating. Or, the users expects to initiate some program flow.

Modelling program flow is easier in functional languages than in logical languages[3], and that is a yet another reason why we prefer functional programming, the last one being that we are more familiar with and more fond of functional programming.

Having decided to use functional programming, the choice for Haskell was logical. It is after the all the standard, and probably the most used functional language. We might have been tempted to use impure languages such as Scheme or ML, as they could have made life easier by allowing side effects, an important characteristic of I/O, but we decided not to do that as we considered it cheating. If the pureness of Haskell is considered such an important aspect of Haskell, we should respect it as much as possible.

1.4.4 An Extension Language

If we want to express in our system the set of possible user actions, we spontaneously arrive in a model based system. Most model based systems automatically generate code for a user interface program. However, our model based system is a *programming* language. There is no separate phase of (automatically) deriving program code for the user interface.

As we use Haskell as a basis, a first choice is to write a library for Haskell containing our system. This means that our user interface will be implemented *in* Haskell. This is a good choice as we want to be able to define a GUI for existing Haskell programs without rewriting the functional core of the application. However, it turns out that the GUI libraries for functional languages, such as Fudgets for Haskell and the Clean I/O library for Clean, strongly influence the programming style of the underlying application layer.

This is something we definitely want to avoid. If the UI layer requires a programming style different from that for standard application programming, we'd rather define an extension *on top of* Haskell, than spoil the entire Haskell programming system.

This way the existing Haskell programs can be maintained and the UI defined in a really separate layer. The application layer of new programs can also be defined in the familiar programming style of Haskell; only for the UI one must learn Visto, which is a separate methodology. This is not necessarily bad: we already mentioned that user interface programming is intrinsically difficult, and any UI system requires specific programming skills.

Visto focuses on the user actions. We might be tempted to state that all functions from a Haskell program are user actions, but that isn't correct. From the large set of functions that has been defined in the Haskell program a lot will be auxiliary,

[3]As a matter of fact, since our decision to user functional programming, Mercury, the acclaimed successor of Prolog, has been developed. Mercury now contains support for specifying the information fbw.

not fulfilling direct needs of the user of the final program. So not all of them must be made available for the application user. We have to restrict the set of functions that can be used such that this set defines what functionality *can* be presented. We want the programmer to consider in a separate phase *what* actions he considers to be relevant to users of his program.

Haskell modules present some way for restricting the visibility of functions, but more in a library like way: they define which functions can be used by other programmers, not by *users* of programs.

They also don't solve the matter of distributed state. A user interface is meant to give the user control over the state of an application, but this application state is often distributed over separate processes and separate windows. So although it may be easier in functional programming to implement state transformation by passing a monolithic state around, it is actually better and more modular to have many local states, as long as we can make sure that they don't interfere.

Therefore we felt the need for a different approach apart from pure functional programming. In his paper on NewtonScript's user interface [77], Walter R. Smith already pointed out *the programming dichotomy*: programs with graphical user interfaces usually contain two very different components: the "model" of the data being manipulated, and the user interface that manipulates it. And "*Often, these components are best implemented using different styles of programming.*"

If Walter R. Smith talks here about the model of the data, we would like to restrict this to the actions on those data. After all, the UI must give the user access to those actions, more than to the data directly. Take for instance a web browser and a web design program: they both act on the same data, but as they support different actions, the UI must be different as well.

The idea that the UI and data are often best implemented using different styles of programming, invited us to take a closer look at the possibilities for defining Visto as a system external from Haskell. Staying within the programming community, we decided to consider object orientation first, also with the current widespread interest in it.

1.4.5 Why Object Orientation

Object orientation is advocated a lot. Therein one tries to model the world by objects that are supposed to match real world entities very closely. Because of this one believes that this methodology is easier to grasp, but it offers some objective [sic] advantages too, especially when encapsulating state and behaviour.

These are the key reasons why we have preferred an object oriented system for describing the user interface. The methods of an object form the exhaustive set of *what* an object can do, and when a method is executed the local state of the object is updated.

We want the programmer to first describe the objects that the user is aware of, and the methods that are relevant to the user. From that point on, and using that information the user interface will be constructed. We will explain this idea more thoroughly in remainder of this thesis.

Traditional abstract data types encapsulate state and behaviour as well, but typically object oriented systems also try to provide higher degrees of reusability by inheritance and polymorphism. Visto of course provides such features as well.

A *Functional* Object System

Traditional object systems are still built in an imperative way. The method definitions for each of the objects follow a clear imperative style with assignments and strict sequencing.

Because the goal of a method body is to calculate some result, we can use just as well a functional language for the method definitions instead of an imperative language. We even believe that it can possibly be a better solution as functional languages precisely excel when used for describing local calculations. The idea of objects that operate through their methods, is already clear to understand, but it can become even more understandable when the methods are defined in a more declarative way. In the end we hope to combine most of the advantages of both the functional and the object oriented world.

Side effects, an essential feature of user interfaces, are also something that must be dealt with. Most Haskell solutions prefer monads. With the convenient do-notation this certainly has become a lot easier than in the beginning days of monads, but it results in a mostly imperative coding style.

So, instead of effectively turning to the imperative – language wise or style wise – we build an object system on top of Haskell. Although one can build programs completely in Visto, thus considering it a separate methodology with side effecting operations on its state transforming objects, it is really intended to be used in a functional environment. Its main goal is defining a user interface for an (existing) Haskell program, or defining an object extension of a Haskell program.

The methods of our objects of course cannot be solely functional. They need some extra object features too, but its core – syntax, evaluation mechanism and usage of functions – is to remain Haskell. Visto tries to adhere to a maximal functional *look-and-feel*.

1.4.6 Prototypes versus Classes

As we explained in sec. 1.3.2 class based object orientation is best suited in a system with many identical objects. Prototype based is more suited when this is not the case.

Typically, object orientation is used in GUI systems to model the widget library, e.g. both a menu and a push button are specialisations of a more general class Button. In that sense a GUI consists of many identical interface objects.

However, the Visto objects are an implementation of specific action models. They are much higher level than the typical, often elementary widgets, and as a consequence have very few instantiations in a program. The actions for a web browser clearly are quite different from those for a spreadsheet. Therefore the Visto objects that model those actions are very different as well. Of course, the interfaces described in the Visto objects, contain a number of identical buttons and pull down menus, but those low level widgets are not the main issue of Visto. It's in the model based objects that we're interested in, and those really are mostly unique and certainly different enough to be placed in different kinds of objects (or classes in a class based approach).

So instead of defining many different classes of which only one or two instances exist, we prefer to use prototype based orientation in which we can directly implement our objects.

In just the same manner, the NewtonScript project [77], although skeptical about prototypes in the beginning, actually experienced the fact that this idea is ideal for user interfaces because creating unique objects is the most natural thing to do in a prototype-based programming environment. They stated that users of their systems found it easier to reason about and control the interactions of individual objects – the usual requirement for UI programming – when the objects themselves are being programmed directly. Being able to manipulate the objects themselves, instead of going through the indirection imposed by classes, gave the programmer a real sense of control, like being able to add some spices to a meal, without having to rewrite the recipe.

That is the main objective reason why we decided to make Visto prototype based.

Of course, there are also some subjective reasons, such as the fact that we like to go with the underdog – prototype based OO vs. class based OO – and our research curiosity in the unknown and unusual. It can as well be considered a fulfillment of the plea for more research on prototype-based languages and programming environments by Walter R. Smith from the NewtonScript project.

We also think that prototype based derivation can be more intuitive and especially flexible than class based because in the latter first classes have to be constructed before concrete objects can be made. The prototype based approach allows objects to be derived directly from other objects, just as people may do in every day life. Most people don't use abstract things like classes, or at least don't realise they do.

Philosophical Comments In this paragraph we present – with little personal comments – a further justification of our choice for prototypes, this time on less technical grounds.

Probably in the dark days in winter when nothing else could be done, Antero Taivalsaari from the Nokia Research Centre in Helsinki [82] wrote down some philosophical and historical observations on classes and prototypes, tracing back the philosophical origins of class based languages to Plato's distinction between classes and instances and Aristotle's interest in hierarchical classification. Prototype based languages on the other hand may have been inspired by Wittgenstein's criticism of classification – that it is difficult to say in advance what properties or hierarchies will be required – a critique we can only agree with when we notice the difficulties our students face when defining Java classes in our freshmen computer science course. Therefore concepts may be best defined by family resemblances between similar objects. This was developed into prototype theory by Eleanor Rosch in the mid seventies, who states that some particular members of families (prototypes) are better representatives of that family than others.

Antero then presents several consequences of this philosophical background. Because classifications cannot be determined in advance, there can be no optimum class hierarchy, and class library designs need to evolve over time. Class hierarchies evolve from the "middle out", in particular, classes high in the hierarchy will be very general and discovered after the more basic classes. Unfortunately, class based languages require that class hierarchies are defined top down, with the most abstract classes being written first. Prototype based languages which use family resemblances rather than class hierarchies are claimed to be able to avoid these problems.

1.4.7 A Particular Interest in Selectors

In the previous sections we have described a number of more high level decisions: why declarative and functional programming, why an external solution with prototype based orientation and so forth.

However, we have also worked on a lower level. After all, if the foundations of our system are not the most declarative, how can we expect our building to excel?

Therefore we also took a look at the basic building blocks of the user interface. It is no longer current practice to directly program a windowing system such as X Windows or the underlying windowing system of MS Windows, except for some exceptional programming requiring very low level events that are inaccessible otherwise, or for some sado-masochistic programmers. Such windowing systems offer very general and powerful means for defining window and screen operations, but they are by far too complicated for every day use as they require the programmer to control literally *everything*, from elementary drawing to event dispatching and handling.

These primitives are useful when very specialised and advanced, non standard user interfaces must be developed, but as modern user interfaces are being standardised more and more, most people prefer toolkits, such as the X Toolkit [65], or Microsoft Foundation Classes. Such toolkits provide common higher level elements such as buttons and text entries, for which they take care of a lot of the underlying repetitive tasks such as visualising clicking a button and drawing the menu when it is pulled down. When first presented, they were a huge step forward in declarativity and usability, perhaps even such a huge step forward that they are still seldomly questioned. But scientists continuously have to question the world around them, even the smallest and most trivial aspects of all.

So we searched the literature for declarative alternatives for the basic widgets and stumbled upon the *Selectors* of Jeff Johnson [40]. He shares our concern that widgets are very much involved with *how* things: the appearance and the layout of an individual widget, and the way in which the user handles the widget.

We also notice that as basic widgets are indeed very basic, constructing user interfaces from widgets only simply takes too much work.

Thirdly, *"Widgets do not mesh well with application semantics; they know nothing about the variables they control."* as J. Johnson states. The traditional UIMS approach (user interface management system) assumes that widgets know *nothing* about application data types. Therefore widgets in most toolkits only return generic indices or strings that must be converted or cast into valid application values. So whenever the application data type changes, the conversion must be adapted too, which is an error-prone process.

On the other hand this will cause quite some information duplication as well. For example, an application might have a Font variable that is an enumerated type (e.g. Times, Courier, Tekton, …). To display these options in the user interface, a programmer would have to supply labels for the different font types, mostly strings, e.g. "Times", "Courier", … , which essentially contains the same information in a different form.

Part of the design philosophy behind Selectors – a design philosophy we have adopted as well – is that there is much more to a good user interface than just nice looking widgets, and that higher-level aspects of the design (e.g. appropriateness of the widgets for what they control, task specificity of the interface and of the application) are more important than the lower ones.

Selectors are higher level elements that focus on a particular subgoal of user interfaces, i.e. *selecting* values, such as the size of the font to use, the name of a person, the date to pay a bill, etc.

These selectors abstract over the way in which the selection process takes place. From an operational point of view it doesn't matter whether the data are selected by a button row, a scroll bar or whatever. What matters is the fact that data are selected, and that we can constrain the set of values that can be selected, as in selecting a colour only from the set {red, green, blue}, or select a number between 3 and 11. Selectors allow us to make use of the semantics present in the data, such as the fact that the domain of the selectable values is an interval, or that we are selecting a time, or a money amount.

The concept attracted us as it was the first example of an interesting abstraction over widgets that is really goal oriented: selectors replace a set of widgets that have as their common goal the action of selecting something. Although the original Selectors have been designed for imperative languages, we wanted to prove that they can be applied to functional languages as well.

At the same time, we considered it a nice exercise in GUI and library programming, hopefully bringing us some new insights.

Another, more important reason is that selectors are an excellent domain in which the enhanced declarativity can make GUI programming easier. The goal of selecting something can be easily recognised. Declarative programming thus may be appropriate.

That is not always the case in GUI programming, where the actions of the end user and the visual feed back by the system often cannot be expressed simply in terms of the goal of the action, i.e. what the user seeks to accomplish. For such, often very specialised, user interfaces we think that declarative programming can hardly offer a solution. The same is true for algorithmic programming: sometimes we have to express precisely *how* something must be calculated, e.g. for performance reasons, and cannot simply state the *what*, e.g. the mathematical formula.

Performance is a criterion specific for algorithms, but for user interfaces other, more subjective criteria exist, such as beauty, usability and look-and-feel. These aspects are very important, but hard to quantify. They also don't involve explicit goals of a user interface such as selecting these or that data. Hence, they clearly cannot be expressed simply in terms of these goals.

This doesn't mean that selectors are useless. They follow the trend that is visible in most GUI toolkits, i.e. providing components with ever increasing amounts of default behaviour and a standard look-and-feel. This is due to the fact the ground layer of user interfaces has stabilised. The general principles of buttons, menu etc. have been accepted by the general public and most application interfaces adhere to these de-facto standards. Just as well the different O.S.'es resemble each other very much for those basic aspects. Therefore the UI toolkits can safely supply such larger components as they make life easier for the developers in developing standard user interfaces. Take for example the *JTable* component in the java swing library (sec. 2.7).

Selectors fit perfectly in this trend which present yet another reason why we chose to implement them. Another advantage is that they easily fit into existing systems. Although they can contain many widgets, they don't replace the entire library. Offering large, reusable structures they enhance the libraries without entirely removing the need for smaller, specific widgets.

These advantages apply as well to other research projects that enhance widgets, but we specifically chose to implement Selectors, not only because it was the first such tool that we encountered, but also because it fits rather nicely in the concept of functions. In some ways a selector can be considered a function in the sense that it returns a result: the value that was selected.

We started with an implementation in Fudgets because Selectors mix particularly well with Fudgets. Although the basic fudget is a functional abstraction for a widget, it can be considered an autonomous object that outputs values on its output stream whenever the user invokes some event. For a button this would be a button click, for an input field the entry of a value and so forth. The type of a fudget reflects this: $F\ a\ b$ is a fudget with values of type a on its input stream, and values of type b on its output stream. Essential is the fact that this stream behaviour can also be made to occur in Selectors. Whenever the user selects an appropriate value, we put that value on the output stream.

After having implemented Selectors for Fudgets, which were at the time the de-facto standard for programming GUI's in Haskell, we also implemented them in TkGofer, a language we considered easier to translate our Visto source code to.

A more pragmatic reason for only considering selectors is the fact that implementing selectors taught us that it doesn't suffice to define advanced libraries to create a really declarative GUI development system. An integrated methodology is needed. Therefore we decided to no longer examine other higher level widget libraries, but instead to devote our time to work on our methodology, the subject of the next section.

1.5 Suggested Methodology

1.5.1 Main Issues

Before we present our suggested itinerary for developing a declarative user interface using Visto, we remind you shortly of our main issues.

If declarativeness is the act of stating *what* is wanted and not *how* to accomplish it, declarative programming is writing programs that clearly express what the program must do, not how the machine does it. Therefore, when we want to declaratively program a UI, the program must state what the UI can do. In our view this is performing actions on the objects the users manipulate. The UI helps them in doing those manipulations and thus must be primarily concerned with the actions of the users. We definitely don't want to fall back on the call back functions, which are a machine oriented way of stating how and when the actions must be performed. We want to stress the actions of the user objects and show that the user interface only serves as a way of invoking those actions.

As it is often the case that the UI offers a number of alternatives to initiate the same action, we also want to clearly visualise this in the code. In the typical call back function approach this is not very visible. One must examine all the widgets and their call backs to find out the different UI-alternatives for invoking the same action. It is true that this is often expressed in the documentation of the program, but we prefer that the program code itself documents this. This makes it more readable, and declarative as the intention of being alternatives for the same action, a *what* aspect, is reflected in the code. It is no coincidental fact, but a true intention that deserves to be visible in the code. It also makes the code more easily maintainable.

A third goal is providing an abstraction over widgets such that the choice of the specific outlook of the UI can be postponed. We don't want to be involved from the very beginning with aspects of presentation and layout. The presentation is extremely important in a GUI, but we try to separate that from the functional specification of the UI. Using our model approach in which we focus on the actions of the user objects, we first define what the user interface must allow the user to do. When the developers have decided what actions (e.g. open file) the user interface will be capable of, the next question must be answered: in what way will the arguments for those actions be gathered (e.g. the file to open). This can be neatly handled by the Selectors approach, as these components deliver values that are selected by the user interface.

Finally, after having stated what selectors will be used, presentations for those individual selectors can be chosen and the layout and presentation of the entire interface finalised.

These ideas of stressing the actions and abstracting away the specific appearance of interface components are key issues in the development process that Visto promotes.

1.5.2 Stages of Development

The Visto development process can be divided into five stages. We present here a rough outline of these stages. Chapter 4 contains examples of the approach.

Functional Application Code

Although many ideas of Visto can be applied in any system, Visto itself is designed specifically for Haskell. Visto can be used for implementing a graphical user interface on top of (existing) Haskell programs. Its design philosophy therefore inherits many functional features, such as the requirement that every Visto method returns a return value and the use of a single state variable per object.

Although the GUI of the finished application will be event-driven, the Haskell code of the application layer doesn't have to take this into account. It can be safely developed using traditional software methodologies and techniques.

Visto Object Definition

As we will outline in our third chapter Visto has many object features. We do not expect the programmer to program all his logical objects in Visto. The stress is more on the visual aspect. We expect the programmer to define objects that correspond to visual *user* entities. In a word processor the *visual* model uses texts, whereas the application layer may prefer to use paragraphs, undo buffers and other logical entities hidden for the user.

We do not expect the user to work with these logical entities, but rather with *visual* objects. The Visto programmer will define a set of such objects, and, for each of these objects, a number of methods for actions on the objects. These methods constitute the complete set of actions the user interface can use. Screen refreshes are automatically performed using a **draw** method.

Choosing Invokers

Objects typically communicate with messages, invoking each other's methods. In Visto methods can be invoked by objects sending messages and by the GUI itself.

The methods defined in the previous stage constitute the complete set of actions that the GUI *may* invoke. The next stage is choosing which actions – which methods – will *effectively* be present in the user interface. This is done by associating invokers with methods. Note that we have first considered which actions on the different objects may be needed and only afterwards decided which ones will be present in the GUI.

We thus start from the methods, and select all the interesting ones. This way it is easier to ensure that the application provides all the needed functionality and that it doesn't miss some essential features. Then we can compose all the GUI invocations into a nice looking interface. If we had just focused on the outlook of the interface, it may look nice, but we have a greater risk of missing some functionality.

The programmer can associate a single method with *any* number of invokers. This allows different alternatives to be addressed in a simple way, e.g. a menu item and a button for the first time user or a key short cut for the expert user.

Associating the invokers with the methods instead of placing the call backs (‘methods’) with the widgets (‘invokers’) is more than changing ‘b is with a’ to ’a is with b’. The stress is put the other way around, on the methods instead of on the widgets. We find this a declarative advantage as it stresses *what* is to be done over *how* it is invoked.

Visto is also more goal oriented: "*To activate that method, use this or that invoker.*", whereas the call back mechanism simply describes an effect: "*When this widget is activated, that action is performed.*" Therefore Visto's code is closer to the user's mental model, who is seeking to fulfill a goal.

Naturally, *how* to invoke something is often very important in user interface design. The look-and-feel and the handling of the objects used in the interface are an important criterion when creating a GUI. So although we still prefer a declarative approach for GUI's, stressing *what* is to be done, we are faced with the difficult task of reconciling the declarative aspect with the often strong desire for being able to precisely describe *how* to do something. We return to this issue in the next section.

Another advantage of Visto is that the information is more concentrated. In a typical GUI application where each widget must have a call back procedure, those widgets may be widely spread across the source code and/or visual environment. So even if different widgets invoke the same call back – which may even be difficult because of different call back types for different types of widgets – those call backs will be located in many different places.

So it is not as explicitly documented in the code that they actually provide just different ways for invoking the same functionality.

Trying to solve that may be one of the reasons why Java uses object listeners that listen to actions on widgets and trigger some action in response. If we want different widgets to trigger identical behaviour we let the same object listen to those widgets. When the behaviour changes, we only have to change the implementation of the listener object.

This is a good design for changing the behaviour of a user interface, but as the listener listens to *events*, it is often difficult to see which *widgets* triggered those events. Therefore we find that listeners poorly reflect which widgets invoke the same actions. This is however an important design decision that is more clearly reflected in our system by the list of invokers associated with a method.

Changing the user interface also becomes quite simple in Visto, e.g. we can simply replace an invoker that uses a button by an invoker that uses a menu bar. This is quite different from the traditional approach where the type and parameter count of the call back function may depend on the widget used. So changing a widget may then incorporate quite some work.

As we already indicated, and as will be explained in more detail in chapter 2, we follow the example of J. Johnson [40] and provide an abstraction for actual widgets, Selectors [3]. In this technique we focus on the declarative aspect of the widgets: selecting the appropriate data. Some examples of selectors are:

- Selecting a single value from a range or from a discrete set,
- Selecting a specified number of values,
- Selecting between n_1 and n_2 values,
- Special interest selectors such as currency, time, ...

This stage thus offers two steps with an enhanced declarativity:

- focusing on the functionality over the widgets by adding invokers to methods and not vice versa,
- and abstracting away from the actual widgets by means of selectors.

Finalising the GUI

This final stage will be performed in a (not yet implemented) tool for drawing the user interface using direct manipulation. The tool will first analyse the Visto program and collect all the used invokers. These invokers will then constitute the precise set of things the programmer has to put in the user interface. It can be compared with a jigsaw puzzle where the puzzler has all the pieces in front of him and assembles them together until all pieces fit, being assured of the completeness of the result. The same applies to Visto: once all the selectors have been instantiated and inserted in the interface design, the programmer is assured that the interface provides all the desired functionality. This is different from traditional tools where the programmer has little or no guidance on which widgets to use and certainly not on which functionality the user interface should provide. Such an interface may then look good, but miss some essential functionality.

However, this optimistic scenario is only valid in an ideal world. A real jig-saw puzzle is designed such that it fills a full rectangle. The user interface must in the same way nicely fill its window. A good looking interface is more than the sum of the widgets. It must all fit together.

So, on the one hand, from a declarative point of view, we should design our interface such that it provides the right invokers and selectors. But on the other hand we also have to take care that the composition looks good. It would be an incredible coincidence if the initial set precisely fills the screen. If, for example, there is not enough space for a button row, we may prefer a pull down menu, or another more compact presentation. And when some empty space remains, we may add a few more invokers. The overall layout can influence the presence and instantiation of individual invokers and selectors.

Instead of a straightforward assembling of a jig-saw puzzle, the actual process is more back and forth. Nevertheless, also in this case the Visto *model-view-activator* methodology helps in facilitating this process. Instead of showing the (potentially very large set) invokers and selectors in a linear way, the visual GUI design tool should display them along with the Visto objects and their methods. This way the interface designer can always see in what way which methods can already be activated from within the user interface, and which methods are not present. And as Visto models represent the *user's* model of the program, the designer can see *what* the user can do with the user interface.

So whenever more or less items are needed in the interface, the designer can base his decision both on the presentation of the interface (what it looks like) and on its functionality (the methods that the interface controls). Both aspects are important and our tool will provide the necessary information about, and offer ways to manipulate both sides.

When placing the selectors on screen, the programmer chooses between various visual instances, such as for example an entry, a slider bar, button rows and columns, key short cuts, ... For some types specialised selectors may exist: we can select a time using a selector that allows us to manipulate the hands of a clock, or we can use generic range selectors to select the hour, minutes and seconds.

We prefer to have a specific tool to lay out the different selectors because it is a separate design phase. We should not mix this with the development of the actual program code. We may – or perhaps better *should* – outsource this to non-programmers, experts in Human Computer Interaction and/or user interface and lay-out artists.

Changing the GUI

A new interface can differ from an old one in many ways.

- Sometimes it's just a matter of layout. Then we keep the instantiations of the selectors as they were and only lay them out differently.
- Or a survey may have learned that we have the right number of ways to achieve an effect, but that some ways should be replaced with others. We then have to instantiate some selectors differently and restart layout.
- More probable is that we want to add – and perhaps also remove – some ways to invoke a certain action. We have to go back to the third stage and add or remove some invokers, move over to the fourth stage, instantiate the invokers and apply layout.

  The previous scenario's changed the functionality of the GUI, but not of the application. The same things can still be done, but just in a different way.
- Now suppose that some object methods weren't already associated with GUI invokers, for example because we expected them to be too difficult for the average user.

  If we really want the functionality of the user interface to change, to have it do things it couldn't do before, like that particular expert method, we again only have to add some invokers to the Visto objects in the third stage and take this along to the fourth stage. However the application doesn't really change if we have developed the objects and their methods well. The object will have been ready. Only an invoker has to be added.

If we had used call back procedures and started from the widgets, we would have to define the new call back procedure from scratch, inducing a larger workload.

- Of course such a dramatic change may also occur in Visto when really new functionality is added to the application itself. But then the application domain changes, and consequently and inavoidable the Visto objects and their methods must change as well.

Anyhow, we believe that this approach minimises some of the work in redesigning an interface and most certainly, the semantically different changes in the user interface (changing only the appearance or adding/removing functionality) take place in different stages of development, which is always a desired feature of a methodology: keeping separate decisions separate.

1.5.3 Translation in Haskell to TkGofer

To make Visto code executable, we follow the approach used by many compiler writers, i.e. compile to another high level language with an existing compiler. A lot of compiler writers use C as their intermediate language because a number of fine C compilers exist, and because C code is a lot easier to write than assembly, while still allowing hardware related optimisations. It then suffices to write a C preprocessor and leave the C compiler with the hard tasking of actually compiling to assembly.

We use TkGofer as our intermediate language as we can then simply pass the Haskell functions to the TkGofer interpreter. We must only translate the Visto objects and their object features to TkGofer. As a matter of fact, the translation of the Visto object code is fully Haskell compatible. It is only for the GUI elements that we use TkGofer specific features. And with the advent of TclHaskell [22] Visto could be fairly easily translated to full Haskell, but we have not yet done so, mostly because of a lack of time.

Besides the fact that choosing a Haskell-like language simplified our translator, we also wanted to prove that it is feasible to translate such high level GUI programs to Haskell, while still retaining comfortable execution speeds. We didn't actually measure the response speed because we didn't have the chance to experiment with large Visto programs, but in the very least the smaller examples we tested, all ran smoothly. We could have measured response times, but they would have been irrelevant because the test programs were simply too small.

The translator itself is written in Haskell as well. If we try to advocate the use of Haskell, because of all its nice features, it naturally would have been a denial of our arguments if we hadn't used Haskell as our implementation language.

The results are worthwhile. The Visto translator counts about 4000 lines of code, including comments. This makes it an Haskell application of considerable size, but it is still a modest size for an implementation of an original object based system with many user interface features.

Our approach, translating Visto by a Haskell translator to TkGofer, is a portable solution as well. Compilers for Haskell exist for practically every computer and TkGofer is available on a lot of platforms as well, e.g. Unix, Linux and MsWindows. A port of Visto to TclHaskell, that is very feasible, but has been delayed, only because of a lack of time, will make it available for every platform on which Tcl/Tk and Haskell are available, namely as good as any platform. Visto code can then be used on any platform without rewriting.

1.5.4 Overview of this Thesis

In some ways this thesis reads as a book: we begin modestly (with Selectors), but introduce, chapter by chapter, new elements until the solution is uncovered in one of the final chapters. Of course, this kind of reading is not necessary. The reader can easily skip to the chapter of his choice.

We begin this thesis with a closer look at the smallest building block of user interfaces: the widget. A widget is a common user interface element, such as a button, an entry or a pull down menu. We give on overview of a number of user interface toolkits and how they construct widgets. We will show their similarities and differences. They all provide widgets, but the widgets of the more recent toolkits have more and more default behaviour, contrary to the earlier toolkits in which the programmer had to define almost all characteristics of the widgets, such as the shape of a button. The trend for more standard structures is also present in a number of higher level libraries that provide larger standard components such as a requester or a table, but they are still restricted in the number of possible appearances, rather than focusing on the intended behaviour of the component. We then present the *selectors* approach, historically our first encounter with a goal oriented alternative for widgets. We discuss our fruitful implementation of this declarative idea, both for the Fudgets [16] library and in TkGofer [88]. These Selectors will help us in fetching the arguments for the actions the user initiates through the GUI.

In this second chapter we also present the *Model-View-Controller* pattern that separates the different responsibilities in a GUI program in three separate components.

A broader idea that goes further than the simple replacement of the basic widgets by our *selectors*, is then discussed in chapters 3 and 4. In **chapter 3** we present the object system, that is needed to be able to implement our ideas. It is a prototype based system, contrary to the more popular class based approach. Several ways for deriving new objects from existing ones – often referred to as *inheritance* – are presented. Several design alternatives are discussed. The model of the actions in the UI is to be implemented in this object system.

In **chapter 4** we show how to build a GUI with the objects from chapter 3. Once the initial – simplistic model – has been explained, some examples illustrate the approach. A slightly more formal overview of its possibilities is then given. Of course we compare our approach with several interesting existing research projects.

The next **chapter 5** is a *case study* in which we present another example more thoroughly, comparing it to implementations in other systems, showing how our program is more declarative and less influenced by changes to the user interface.

We then pay a second visit to Visto in **chapter 6**, discuss the language features shortly and present the compilation scheme from Visto to TkGofer, the implementation language of our choice.

The last chapter contains what any last chapter contains: *conclusions*, a summary of our contributions and an overview of future work.

As we start from these basic building blocks and continue with higher level elements, we use a bottom-up approach to present this thesis. We consider this the best way to experience the different fields in which we have been working: providing abstractions for common user interface patterns, designing and implementing an object oriented language, and finally bringing it together in a system for declaratively implementing a user interface. If we had presented it top-down, it might have been more difficult to understand the various facets of our research.

It also gives us a chance to first present some of the existing GUI systems, showing the reader some typical characteristics and trends in GUI development. Our first contribution to the field then follows eloquently from this observation.

An added bonus is that it reflects best the way in which we progressed. Hence, it corresponds to a correct historical evolution in our research. We started off looking at and testing some user interface toolkits for functional languages. This first investigation already showed that widgets are rather basic elements that often poorly express the actual goal of a combination of widgets, such as selecting some particular data. Also changing a group of widgets by another combination often requires a lot of programming. Combining these personal experiences with a reading of literature brought us, through the concept of selectors, to some initial improvements in the form of a more declarative abstraction for a typical combination of widgets with the goal of selecting data.

A strengthened understanding of user interface programming finally lead us to some more high level ideas that are discussed in the chapters that follow the next one about our approach on widgets.

Chapter 2

Selectors

2.1 Overview

In this chapter we mainly discuss the basic building blocks, the components that a user interface is composed of. We will illustrate that typically *widgets* are used.

However we first need to explain in rather general terms how a user interface is typically implemented. There is a general consensus to use the *Model-View-Controller* (MVC) paradigm which is presented in the next section. This presentation is necessary to show the role of the widgets in the overall implementation. After all, if we define a button – a typical example of a widget – we must also define a purpose for that button, and therefore need to connect it to the application. The Model-View-Controller idiom proposes a way in which to do this properly. There also exists a metamodel for the runtime architecture of interactive applications, the Arch model [84], but we prefer to use MVC here as it more commonly known. In chapter 4, when we compare Visto more thoroughly with MVC, we devote sec. 4.3.2 to a comparison of Visto with the Arch metamodel.

Presenting the various widgets without referring to MVC is like presenting them without context. We first present MVC and only then the widgets and their definition in a number of toolkits, because the MVC-idea is a common factor in most toolkits, whereas the construction and composition of the widgets is often different in different toolkits.

In the following sections we first present a number of commonly used GUI toolkits for imperative languages in a chronological order, and then do the same for functional languages. We will notice a trend for more declarative definitions for a widget, especially when we consider the widget libraries in functional languages.

This trend is amplified in some special purpose libraries, which is the contents of a following section.

Finally, in the notion of *selectors* a very high level of declarativeness is reached. We first present Jeff Johnson's paper [40] and what can be seen as a (partial) application in Java Swing. The *selectors* precisely express the goal of many user interface components or structures: *selecting* some data, from a range or another well defined set.

Then, the chapter reaches its end – and its main reason for existence – when we present our application of these *selectors* in both Fudgets and TkGofer. For each of the selectors we will define different ways in which to display them on the screen. The way in which that is done, e.g. as buttons or a pull down menu, should not influence the rest of the program, and the set from which to choose should be controllable at run time. We will show that our selectors fulfill those goals.

2.2 Widgets and MVC

The building blocks of user interface toolkits are typically called widgets which stands for *window gadgets*. According to FOLDOC [34] a widget is " *In graphical user interfaces, a combination of a graphic symbol and some program code to perform a specific function. e.g. a scroll-bar or button. Windowing systems usually provide widget libraries containing commonly used widgets drawn in a certain style and with consistent behaviour.* " This definition, however, still allows different degrees of granularity. Some people consider widgets as the extremely low elements, viewing e.g. the thumb and the arrows of the scroll bar as separate widgets. Other consider the scroll bar entity as a single widget, while other programmers even look at the scroll bar together with the window that is scrolled by the scroll bar as one single widget. Throughout this thesis we refer to widgets as the 'mid-size' elements, e.g. considering a pull down menu as one widget, and not as a set of menu item widgets.

Using such widgets, with a specific goal and presentation, is certainly more declarative than directly programming the underlying windowing system in the sense that we now specify what we want to define, a scroll bar or a button, instead of supplying all the drawing and event handling code that defines the behaviour for that particular widget. The toolkit itself defines e.g. the code that graphically gives the impression that the button really has been pressed, e.g. by inverting its borders.

However, defining a user interface is a lot more than simply defining and assembling a (large) number of widgets. In a proper implementation different aspects are defined in different structures. And in GUI programming special attention is paid to keeping the functional core independent of the user interface [7, 8, 14, 45]. The core of interactive systems is based on the functional requirements for the system, and usually is supposed to remain rather stable. User interfaces, however, are often subject to change and adaption, for example to support different look-and-feels, different expertise levels, etc. [15] Although this is in part *wishful thinking* as due to an often extremely short time to market, and ever changing requirements, the functional specification rapidly changes as well, the point remains valid. Separate issues should be separated, and it is also interesting to be able to define many different user interfaces for a same program, especially with all the (sales) talk of adding computers to all household equipment. Desktop PC's, WAP-mobile phones, internet set top boxes, the refrigerator, ... may all use the same application, but actually need different interfaces.

The pattern *Model-View-Controller* is a common way for accomplishing this.

- The **model** contains the core functionality and data.
- **Views** display information to the user. Each view obtains the data from the model and displays it in its own proper way.
- **Controllers** handle user input. When necessary they can initiate changes to the model. A change-propagation mechanism is used to make sure that all the views reflect the changes to the model.

The views and controllers together comprise the user interface. The model is no direct part of the user interface, and thus must be independent of specific output representations or input behaviour.

For a small user interface with a simple calculation, this *model-view-controller* paradigm resembles the simple, textual based I/O algorithm:

1. get input (*control*)
2. do calculation depending on input (*model*)
3. output the result of the calculation (*view*)

The fact of getting input and triggering the calculation is the task of the controller, doing the calculation is described in the model, and finally outputting the result is the responsibility of the view.

When several algorithms are involved things already start getting a bit more difficult. Moreover, because typically many different views on a single data model co-exist in more advanced graphical user interfaces, things get even more complicated. The model must now notify all views whenever its data changes. The views in turn retrieve the new data from the model and update the displayed information to reflect those changes.

At the same time, it is a good design to offer the end user several different ways for invoking some functionality, e.g. by clicking on a button, or by selecting an item from a pull down menu. For each of these alternatives, a controller must exist. Therefore, there are not only many views, but also many controllers.

In MVC there actually is a one-to-one relationship between controller and view. The controller, being closely associated with a view, must handle all events that occur when its view is active – in focus –. This includes triggering the model when the user performs actions that are intended to change the contents of the model, but it may also only refer to changing the view, e.g. when a scroll bar is dragged.

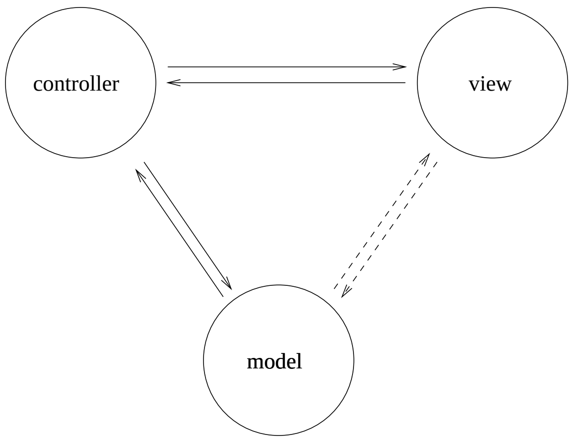


Figure 2.1: Model-View-Controller

To simplify the task of defining a controller, it is most often composed of common user interface elements – widgets – such as the button. The controller then must no longer take care that the borders of the button are inverted to give the impression of an indented button. It suffices to define what happens when the button has been pressed. This is done by adding a *call back function* to the widget that defines what to do when some particular event occurs in the widget, e.g. clicking or double clicking it.

Therefore widgets and their call back functions play a crucial role in the working of this MVC-scheme. The views make extensive use of widgets, and the controller must react on the events that follow from manipulating those widgets. Apart from the one-one relationship between views and controllers, this furthermore relates the controller to the view.

In fig. 2.1 the interconnection between the model, the controller and the view is depicted. The dotted arrows indicate dependencies that should be minimised as they break the independence between user interface and application. However it is often impossible to define complete independence as the interface intrinsicly depends on the application in the sense that the interface needs to know what kind of application it controls (e.g. a word processor or a web browser).

So for example if we must write code in the model that triggers the views when the model changes state, we loose the independence. However this is often considered acceptable when no more than a change event is signalled as in that case the views don't depend on the representation of the model, just on the fact that the model needs to say that is has changed its state. Nevertheless, we feel that a system that avoids such an explicit trigger of the views is preferable over a system with such triggers. As we will explain on p. 136 Visto contains no triggers in the model. Only the views must register themselves as views on a particular model.

In the following sections we present the definition and composition of widgets for several systems. As this chapter is rather *a view in the small*, looking at the differences and similarities for defining widgets in the various libraries, we don't discuss how to define model, view and controller in those systems, but put stress on the widgets.

We take a view *in the large* later on. At the end of section 4.3 on p. 133, when we have presented our alternative solution for defining graphical user interfaces, we will contrast our ideas and implementation with traditional MVC.

2.3 Commonly Used Imperative Widget Libraries

We first take a look at some of the popular libraries to explore some different widgets as well as different styles for structuring the library. In this chapter we are primarily interested in the construction and composition of widgets and not in the connection with the application or the architecture of a complete GUI system as the goal of this chapter is to present an alternative for those widgets. Presenting our alternative for connecting an interface with the application layer is postponed until chapter 4 and therefore we needn't discuss this issue here.

The first systems were intended for C. As the designers of these systems preferred to organise their library in an object oriented way, they were forced to implement their own object system into C. Later on, these libraries were transferred to C++, but often maintained their initial object system for compatibility reasons.

Their widgets were also extremely flexible, meaning that any sort of widget could be created. A button for example can have any shape and doesn't have to be rectangular. Hence, the constructors are also very powerful, but this implies at the same time that they are rather difficult to master as many parameters must be explicitly set.

Later on, the more recent libraries prefer having defaults for most standard features, which means that they become simpler to use. However they either provide alternate constructors or extra arguments for specifying the specific non-standard characteristics.

Of course, in the next section, we also take a look at functional language libraries. Although they often differ in style, both relating to their imperative brothers and in between, they follow the ideas of providing both simple, standard constructors and more complicated alternatives. However, different ways for composing the widgets together are used. We also take a look at those different compositions.

The selection of user interface toolkits we consider in this section is supposed to be chronologically representative over various operating systems. The *X Toolkit* and *OSF/Motif* are widely used in Unix environments and are the oldest well-known toolkits. We also take a look at Microsoft's *MFC*, which is a major commercial product. Finally, with java's awt we cover one of the newer language systems.

2.3.1 X Toolkit - Athena Widget set

X Windows [71] is a hierarchical system for the display of graphical user interfaces. The basic building block – the window – is not to be confused with the window that people use in their every day computer environment, the window that contains a Word file or web document with a title bar, buttons and contents, that can be dragged across the screen.

Such a window is already a very high level composed component in X Windows. An X window is just a designated area with a specific function. For a button we would create a window that reacts to mouse clicks and in response inverts its border to give the impression of really pushing the button. For a menu we would define a window that in response to mouse clicks creates and displays new windows which give the impression of a menu rolling down from the first window.

More complicated effects are achieved by placing such windows on frames and combining them into the higher level windows we already mentioned.

It is well-known that directly programming X Windows is a very complicated tasks. Libraries such as the X Toolkit [65, 26] make life easier by abstracting away lots of the tedious programming tasks such as visualising the action of pressing a button. All the programmer has to do is create the widgets with the right arguments, such as the text to appear on the button, assemble them together and write application-specific code that will be called in response to events in the widgets. The programmer only has to take care of the events that really matter for the specific widget, while the library hides the other events.

The core of the X Toolkit, the *Intrinsics*, consists of C-routines for using and building widgets, even for the most demanding applications. It remains very hard to study, probably because it is intended to be extremely flexible. XAW is the *Athena Widget set* and was primarily designed as a simple demonstration and test of the Intrinsics.

Figure 2.2: Typical Appearances of a Toggle Button.

Construction

Let's take a look at the definition of a toggle button. A toggle button is a button that keeps a boolean state: either true or false. Whenever the button is pressed, its state is inverted. This toggle button is mostly presented as a check box (fig. 2.2), but it can also be shown as a button that is displayed pushed down whenever the value is selected ('true') and in an upper state when it's not selected.

widget = **XtCreateWidget**(*name*, **toggleWidgetClass**, . . .)

Figure 2.3: Athena Toggle Button Creation.

This definition seems to be easy. The library certainly takes care of a lot of the work in the background such as reversing the foreground and background colours when the button is pressed, but the programmer is also left with quite some work.

Although (since) C does not provide an object system, the authors of Athena have implemented an object system themselves to be able to structure their library in an object oriented way. So naturally the programmer must know the class hierarchy of the widget, but that is not too difficult. A toggle button for instance is an instance of the *toggleWidgetClass* which is a subclass of *Command* that subclasses itself from *Label* and the programmer can therefore access all the methods from those classes. This class hierarchy, of which the programmer must be aware, may be quite big, but that is normal because the widget set is quite big as well.

More of a problem may be the fact that the programmer is offered a tremendous flexibility. A toggle widget will be mostly defined on a rectangular area, but as a matter of fact it can be defined on an area of any shape. This is very powerful, but may only be needed by a small number of programmers. Features as these increase the learning curve.

Other options include obviously the background and foreground colour, the width, the label, ... but also less trivial elements such as a list from event-to-action bindings that must be executed by this widget, even when the event occurred in another widget.

The argument list of the constructor contains this wide range of options. Those options that are not defined in the construction itself are retrieved from the local resource database.

Composition

We have already outlined that, in X Windows, windows can be placed on frames that then can serve as ‘higher order’ windows. The same applies to Athena. So one of the arguments of the constructor of a widget is the *window*, or frame, on which the widget should be placed. This creates higher and higher level windows, a recursive process that finally ends on a *real* window created by the procedure *XtInitialize*.

2.3.2 OSF/Motif

OSF/Motif [89] is another – more popular – widget set, mostly for use in C or C++. Its structure is largely identical to that of Athena, but it intentionally implements a particular user interface style and provides a set of style guide rules, which is in my opinion the main reason why it is more popular than Athena.

A button can be created in a way similar to that of the Athena widget set by using the *XtCreateManagedWidget* method, or by using the specialised method *XmCreatePushButton* (figure 2.4). Just as in the Athena set the argument list is used to control the various options.

- *button1* = **XtCreateManagedWidget(***name1*, **xmPushButtonWidget-Class**, *parent*, *argList*, *argCount*)
- *button2* = **XmCreatePushButton(***parent*, *name2*, *argList*, *argCount*)

Figure 2.4: Button Creation in OSF/Motif

2.3.3 MFC

The Microsoft Foundation Classes (MFC) are more recent. They provide an object oriented set of user interface elements for the Microsoft Windows operating system for C++. As the language that MFC supports already is object oriented, it must have been easier to create. The result, on the other hand, isn’t always easier to use, as often indeclarative data types are used, for example in the text edit field. This is a widget in which the user can type in any string. In MFC this is implemented by the *CEdit* class with support for both single and multi-line text. Besides direct typing, users can also cut, paste, copy, delete and clear text, either directly in the edit control or using the clipboard.

Construction

The control style of type *DWORD* is used to set the options, such as justification, case, single- or multi-line, . . . Although it may be a memory conservative approach

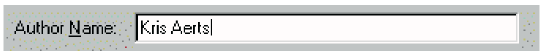

Author Name: Kris Aerts

Figure 2.5: A Text Edit Field.

```
BOOL Create(DWORD dwStyle,          // control style
            const RECT& rect,        // control area
            CWnd * pParentWnd,       // parent window
            UINT controlID );        // control ID
```

Figure 2.6: Signature of the Edit Field Constructor in MFC for C++

to use a *DWORD* – an unsigned 32-bit integer –, it is certainly not easier or more declarative.

Because objects are directly supported by the language, we no longer need to use a very generic *XtCreate(Managed)Widget* with an argument that specifies which widget to create. Nevertheless the constructor of the widget class doesn't suffice yet. We still need to use a *Create* method to actually and visually construct the widget. Other methods allow us to read or change the text in the edit field.

Composition

As can be seen in the signature of the *Create* method the widget needs a *parent window* to be drawn on. Of course several widgets can be drawn on a single window. The rectangular *control area* argument specifies *where* the widget should be drawn, and hence controls the layout.

2.3.4 java.awt

The Abstract Window Toolkit (awt) is a large collection of classes for building graphical user interfaces in Java, meant to be completely portable. The disadvantage is that it is just a kind of greatest common factor, but in the end most modern operating systems are close enough to another to have a very large common factor.

Construction

The construction of most widgets from java.awt is extremely simple as the constructor only needs the most essential arguments. Methods provide ways for setting optional features at will.

This construction is certainly a lot easier than in the previous cases, but after the construction the widget still has to be fine tuned. A number of methods are

- *Button() // Constructs a Button with no label*
- *Button(String label) // Constructs a Button with the specified label*

Figure 2.7: Two methods for constructing a method in java.awt

provided to be able to do so. This was also already the case in MFC, but the MFC *Create* method contained a lot more control arguments, which have completely disappeared in these Java constructors.

Composition

An example of such necessary fine tuning is adding the button to a frame to have it displayed. The *add* method does that. We can furthermore restrain the layout of the button by using the method *setBounds* or leave it to the frame's responsibilities.

We must also define what happens when the button is pressed. Java.awt uses an ActionListener object that can be instructed to listen to events coming from a specified widget.

So, although the construction itself is a lot easier in java.awt than in older widget libraries such as Athena and Motif as we mention directly that we want to construct a *Button*, after the construction we still have to fine tune the widget. The main difference is that distinct decisions are separated in the java code, which not only enhances readability but is also a good design practice.

javax.swing is a more recent GUI library for java, building on the success of java.awt. In section 2.7 we will take a closer look at the features of javax.swing that have reached a more declarative state.

2.4 Functional User Interface Systems

In this section, we take a look at functional systems. We consider only pure languages, i.e. languages that allow no side effects. Because a GUI needs side effects, not only for drawing and event handling, but also for updating the state of the program when some widget is used, these pure functional systems have to find a way for expressing these state changes in a referentially transparent way. How to do that, influences the design of the library and the programs that are written using the library. Although that is not the main focus of this section, we will present it when necessary.

We especially take a look at the solution for Clean [1] and Fudgets [16] as they are the two main GUI systems for functional languages that strongly influenced most of the other systems.

2.4.1 Clean

Clean [1] is the first of our examples from the functional programming community. However, when we explain the composition of a Clean interface, we will have to reveal parts of Clean's solution for managing state updates.

As most systems, also Clean has evolved. In this particular case it is worthwhile to consider both the newest version 1.3 and the older version 0.8 as it nicely depicts an interesting evolution.

Clean 0.8

Construction

The construction is fairly easy, but compared to java.awt we need more arguments again.

- DialogButton *buttonID Position label selectState callBackFunction*
- DialogButton 12 (Below 11) "down" Able decrease

Figure 2.8: A button in Clean 0.8

Most important for the dialogButton in isolation is the *label* that will appear on the button (*"down"*) and the *selectState* that indicates whether the dialogButton will be enabled or disabled.

At the same time we also have a call back function (*decrease*) and an ID for the button (12), plus an indication of where it should appear (*below widget 11*).

Composition

To be able to understand how widgets are placed together in a Clean program, we must understand how Clean solves the problem of embedding side effects in a pure functional program.

Clean uses the environment passing style (see also sec. A.3.2). Instead of changing variables in an imperative way, the environment passing style passes all variables around in the arguments of every function. Because the user interface must take the whole world into account, Clean programs pass the *entire world* around.

Because it is rather awkward to pass the world around in *all* functions, and because we definitely do not want to duplicate the world, Clean uses two techniques: its *uniqueness* features and dividing the world in autonomous parts (fig. 2.9). This way we can separate from the world the file interaction and the world of events that are generated by the user interface. The function *OpenEvents* in Clean 0.8 opens *Events*, the part of the world that takes care of the GUI. It contains all the events that are generated by user actions in the user interface.

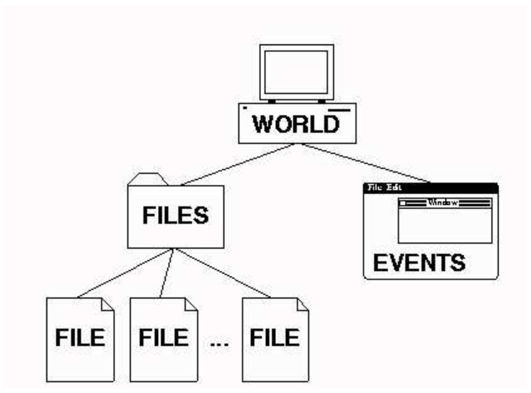


Figure 2.9: The World According to Clean.

So instead of the *entire world*, only this high level component *Events* must then be passed around in GUI functions. Each such GUI function takes an event stream and returns a new stream from which the events that have been handled, are removed.

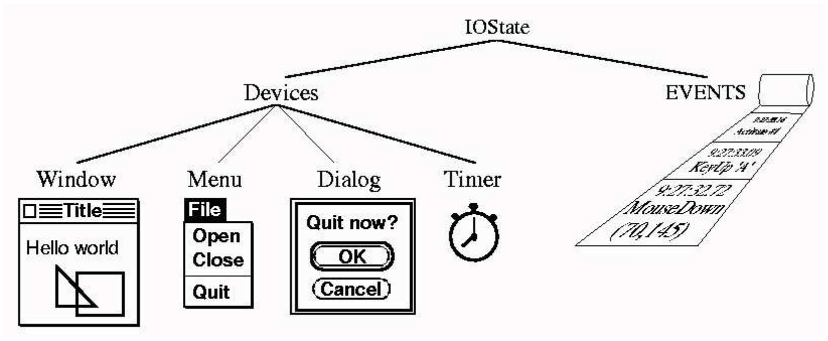


Figure 2.10: Clean's IOState.

Of course, the program and the GUI functions must also state which widgets they control. This too must be passed around in GUI functions. This is all contained in the environment of type (*IOState s*) (fig. 2.10), that is composed of

- the program state of type *s*, that is updated as users perform actions,
- *Devices*, the abstract devices – high level widgets – that participate in the interaction,
- and *Events*, such that the widgets can insert their events in the *Events* stream.

There are four kinds of devices:

- *windows*, that are the application windows in which application contents, such as a text document, graphics, a web page, ... can be displayed.
- *menus*, the pull down menus.
- *dialogs*, the command windows in which buttons, check boxes, input fields, ... can be displayed.
- *timers*, that can be used to periodically perform some action or simply to contain the current time.

The IOState can contain several of these devices, e.g. 3 menu bars, 7 dialogs and 18 application windows. Each dialog in turn can contain many *dialogButtons*, an example of which was already given in fig. 2.8. A *menu* contains *MenuElements* and so on: each device is composed from simple widgets, that have a call back function operating on such an *IOState*.

Finally, note that each widget needs a unique ID and that the layout is specified relative to other widgets. In the example widget 12 is placed below widget 11.

Clean Object I/O library version 1.3

Construction

In Clean 0.8 the definition of simple widgets was quite easy, but because of the need for unique ID's for each widget and because few primitives were provided to fine tune the defined widgets, the system seemed less easy for larger projects.

With the advent of version 1.3 the different widgets became classified in a more object oriented way. Clean 1.3 still adheres to the environment passing style but from now on smaller entities can be passed around, e.g. each web browser window must only take care of its own contents. It is no longer the case that the contents of *all* web browser windows must be passed around. The passed state value is now a real *local* process state and no longer a kind of global variable.

More important is the fact that simple widgets can now be created even more easily while providing at the same time more configuration options. Just as in the Athena and Motif widget sets, Clean widgets now also take a list of *ControlAttributes*.

varName = ButtonControl *label argList*

Figure 2.11: A button in Clean 1.3

While the call back function was a required item in Clean 0.8, it is now just one of the ControlAttributes. Other attributes include usual things as the width and perhaps the font, but also an ID as in the previous releases of Clean.

Composition

Composition is also entirely different from the older version. In Clean 0.8 the hierarchy World - Events - Dialog - DialogButton was rather fixed. Clean 1.3 now has on the lowest level simple widgets such as the button, the check box, labels and sliders, that are *Control* objects. A compound widget is then built from *Control* objects using a *CompoundControl* object.

Because a *CompoundControl* object is also a *Control* object, arbitrarily complex compound objects can be made.

Layout is done, either as previously by the *ControlPos* ControlAttribute that specifies layout positions relative to other widgets using their ID, or in a way similar to the CompoundControl by means of the *LayoutControl* object that takes a number of controls and ControlAttributes such as a horizontal (*ControlHMargin*) and vertical margin and the amount of space needed between items (*ControlItemSpace*).

An example application in Clean 1.3 is given in section 5.5.

2.4.2 Fudgets

The Fudgets library [16] (**Fu**-nctional Wi-**dgets**) was a major milestone as it introduced GUI programming to Haskell [62].

The fudget is the functional equivalent of the widget, but unlike most typical libraries it is not object oriented. Fudgets can in some way be considered objects in the sense that they encapsulate data and behaviour, but they are not structured using subclassing and inheritance.

Construction

Most simple examples are very straightforward, because the fudget library implements basic versions for the most widely used widgets, e.g. creating an integer input field is extremely easy as it is a standard fudget without arguments.

```
intInputF :: F Int Int
```

Figure 2.12: An input field in Fudgets

Most fudgets exist in two forms, an extremely simple one that has default behaviour and another one that takes a list of configurations just as in most other widget libraries.

Composition

Fudgets offer an original alternative for the call back functions found in most approaches, that at the same time specifies how fudgets are composed.

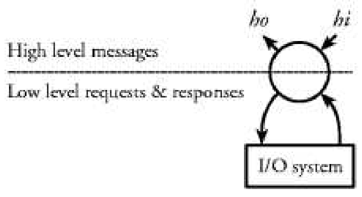


Figure 2.13: The Four Communication Channels of a Fudget.

A Fudget automatically has communication channels connected to it (fig. 2.13) [17], two that are visible to the programmer: one for input (hi – high in) and one for output (ho – high out), and two other channels that communicate with the window manager or operating system.

In an *intInputF* the value of type *Int* that has been entered will be sent to its output channel. A value of type *Int* on the input channel will be displayed in the edit field. So the type of an *intInputF* is *F Int Int* as *Int*'s travel on both the input and output channel.

When the value that is entered in some fudget should be used in other parts of the program, we have to connect the output channel of the fudget to the input channel of (an)other fudget(s).

We do this using combinators, a style that is more usual in functional programming than object oriented composition. The most simple connection $>==<$ sequentially connects two fudgets. A wide range of different connectors exists to accommodate for the most complex schemes.

Instead of call back functions that specify what happens when some widget is used, we have to connect a *stream processor* in between fudgets. Such a stream processor has a high level input and output stream but no visual representation. It is therefore often called a *abstract fudget*.

The activation mechanism therefore is really different. While a call back function directly transforms events into actions, a stream processor only transforms values to other values. However, as the output values of the stream processor are consumed by the fudget that is connected to its output stream, the stream processor can trigger that fudget and thus invoke actions. Hence, call back behaviour can also be defined with the right combination of abstract and concrete fudgets. However, the Fudgets approach cannot cope with global variables, whereas call back functions often can.

In fig. 2.14 we see the scheme for a counter. Whenever the button is pressed, a *Click* is emitted to the counter stream processor that updates its internal state and outputs that value to a display fudget. Notice, for this simple application, the similarity with MVC. The *increment* button is a controller, the *counter* abstract fudget a model, and the *display* fudget a view. This remains true for larger applications but then many models, views and controllers become densely intertwined, making it difficult to separate on a larger scale the different parts of the MVC-pattern.

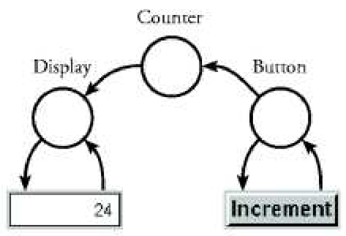


Figure 2.14: A Fudgets Scheme for a Counter Program.

The fudget combinators are not only used for connecting different input and output streams and for defining 'call back' behaviour, but also for defining the layout.

In an early version of the library special versions of the combinators existed, e.g. $>==\#<$ would take a fudget combined with layout information, such as *fud1* $>==\#<$ *(8, LAbove, fud2)* which would place fud2 8 pixels above fud1. This is a relatively easy way to specify layout in relation to other fudgets, but it is not so flexible because the layout is constrained by the structure of the program.

Today two different ways for defining layout are provided in the fudget library.

First of all the need for different combinators, with or without layout, is removed by the use of placers and spacers. One can first combine some fudgets using plain combinators such as $>==<$ and then apply *placerF* or *spacerF*, e.g. *verticalP* or *hCenterS* to the combination to specify a layout. This approach still suffers from being constrained by the structure of the program.

Therefore one can also use named layout in a way somewhat similar to the Clean approach with ID's for widgets. Using *nameF* a name can be given to the fudget. Layout can then be specified using those names and the *placeNL* and *spaceNL* combinators. To apply the layout to named boxes, *nameLayoutF* is used.

2.4.3 Haggis

Construction

Just as most functional widget libraries, Haggis [25] offers easy constructors. The label, for instance, which is a widget that is used to display textual information, only takes the string to display; the button constructor only needs a picture to display on the button and a value that is emitted whenever the button is clicked.

- $label :: String \rightarrow Component\,(Label, DisplayHandle)$
- $button :: Picture \rightarrow a \rightarrow Component\,(Button\,a, DisplayHandle)$

Figure 2.15: A label and button constructor in Haggis

The behaviour of emitting a value whenever the button is clicked is similar to Fudgets. However, a fudget button always emits *Click*, a fixed value. In Haggis we can choose ourselves what value will be emitted when the button is clicked. To achieve that in Fudgets we need a post processor to transform the *Click* into a suitable value.

Composition

Haggis treats the graphical user interface as a virtual I/O device just like other devices, e.g. file or printer. The type *Component (a, DisplayHandle)* in the definition in fig. 2.15 denotes such a virtual GUI device, but it is actually a type synonym for *DContext → IO (a, DisplayHandle)*. It must be opened using a function *wopen :: DContext → IO (a, DisplayHandle) → IO a* and thus forces a monadic style on Haggis programs.

Most of the functional widget libraries mentioned earlier on, used the *Continuation Passing Style* (CPS), but since the advent of monads this technique has become mostly obsolete.

It allows for a nice definition of side effects without having to sacrifice referential transparency, but it can easily result in an imperative coding style. Using the convenient *do*-notation we can sequence side effecting operations such as writing to mutable variables – commonly used in monadic style programs – or creating widgets – an example specific for a GUI toolkit.

A label that says "Nothing" can be created and opened using *wopen (label "Nothing") :: IO Label*.[1] In monadic style this returns a *handle* to a label. Using that handle and functions, dynamic behaviour can be added. The label has a function *setLabel :: Label → String → IO ()* to change what appears on the label. Other Haggis handle functions extract the values that were entered, or wait for a button click, and define call back behaviour. This is unlike Fudgets where values are automatically placed on the output channels. In Haggis they have to be explicitly extracted.

[1]Note how using η reduction and currying the display context *DContext* can be hidden instead of having to write *wopen (\ env → label "Nothing"env)*.

The use of the handles and the virtual I/O devices is a lot more flexible than the streams and channels of Fudgets because the handles and thus the values from the user interface can more easily travel around the program. It is also a lot more manageable than the wires approach from Gadgets [57].[2]

The virtual device handle controls the behaviour of the widget. The *DisplayHandle* on the contrary is used to define the layout. Functions as *vbox* , *hbox* :: *[DisplayHandle]* $\rightarrow$ *DisplayHandle* tile a set of components vertically resp. horizontally and return a new DisplayHandle that can be used in other layout combinators, e.g. *Space* that inserts white spaces between display components.

This is a good approach as it separates the construction of the widget from its layout. However, the display context *DContext* must now be made explicit which complicates the construction of the widgets and the structure of the program.

2.4.4 TkGofer

Being a rather direct translation of the imperative Tcl/Tk [60] GUI toolkit, TkGofer [88] relies at least as much as Haggis on the monadic style. So especially programmers familiar with Tcl/Tk will face little problems when programming a user interface in TkGofer.

Construction

$$\boxed{\mathit{checkButton} :: [\mathit{ConfCheckButton}] \rightarrow \mathit{Window} \rightarrow \mathit{GUICheckbutton}}$$

Figure 2.16: A check button in TkGofer

Again, TkGofer widget constructors take as one of their arguments a list of configuration options. The different widgets are described hierarchically using type classes such that configuration options can be shared between widgets.

The *cset* function, analogue to actual Tcl/Tk, is used to change widget options at run time.

Composition

All GUI actions (construction, modification, value extraction, ...) are monadic *GUI a* values and can be combined with other values (actions) of type *GUI a* using typical monadic constructs. This finally results in one value of type *GUI a* that contains the entire user interface. This value is then converted to a value of type

[2]We don't discuss gadgets here because its doesn't add anything to this overview : its construction is a lot like Fudgets and Haggis and its composition a bit in the middle of both.

IO () by the function *startIO*, which can then be used as the *main* function of a TkGofer program.

Also note that the constructors of a widget take a *Window* on which the widget should be displayed. Using the layout combinators that are precisely like their Tcl/Tk counterparts the different widgets can be laid out and *packed* on the window.

The main advantage of TkGofer – and the reason why we chose it as the implementation language of our system – is precisely the fact that it can serve as a flexible base to build more powerful – declarative – systems on, in a relatively easy way.

2.4.5 FranTk

FranTk [63] is one of the newer GUI systems. It builds on TclHaskell [22], which is a port of TkGofer to Haskell.

Construction

$$\boxed{\mathit{mkButton} :: [\mathit{ConfButton}] \rightarrow \mathit{Listener}() \rightarrow \mathit{Component}}$$

Figure 2.17: A standard button in FranTk

It still uses the same way for configuring widgets with an identical type class hierarchy as in TclHaskell/TkGofer.

Composition

Besides the typical configurations as colour and font type, FranTk adds extra configuration options, amongst others to implement call back behaviour.

The logical connection between widgets to manipulate state proceeds in yet another manner. Nice are the three concepts of *Behavior*, *Events* and *Listener*. We could already see a *Listener* appear in the definition of *mkButton*. Because a button emits *Events*, something has to receive those emissions. That is what a Listener does. A value of type *Listener a* is a function that given a value of type a, performs a side-effecting IO action with it.

Thirdly, a *Behavior* is a value that can change over time. For example an entry can have such a Behavior attached to it that contains the value of what is entered in the entry. Whenever a new value is entered, the *Behavior* variable is updated and vice versa.

Finally the layout: the TkGofer constructor for a button takes a *Window* in which the button is to be placed, but in FranTk the button constructor returns a *Component*. Components can then be combined using functions as *above* and *beside* and are finally rendered on a window, which is accomplished by the function *render* that takes a window and a component to render on the window.

Therefore the layout and the way in which the application is connected to the user interface differs greatly from TkGofer, but the construction of individual widgets not so much.

2.5 Comparing the Various Widget Systems

Obviously, the style of the various widget systems, i.e. naming conventions, argument types, … can be very different, especially when we compare imperative with functional toolkits.

More interesting is the evolution in the default functionality. The older toolkits provide very few defaults. The programmer has to fully instantiate his widgets. Later on, as typical interfaces started resembling each other more and more, the toolkits started providing more and more default behaviour. For instance, in XAW a button can have any shape, whereas in Java or the functional toolkits, this is by default a rectangular area. Another trend is dynamic layout. In the first toolkits the exact position of the widgets had to be defined. However, already as early as Tcl/Tk and later in e.g. Fudgets the position is often specified relative to other widgets, e.g. below or above. In these toolkits we also don't need to specify the size of the individual widgets. The toolkit makes sure that each button is large enough to contain its label.

Hence, as a matter of fact, these toolkits simply follow the trend in GUI development to follow the standards, set implicitly by the market leaders, or set explicitly in official guidelines. This trend continues in more advanced libraries, as presented in the next few sections.

Naturally this had its effect on the constructors for widgets. In the XAW toolkit we had one large constructor with an enormous set of options that could be used for any widget. Nowadays each constructor constructs a specific, very standard widget, such as a button or a pull down menu. These modern toolkits still support roughly the same amount of options, but instead of putting it all in the constructor, they provide methods to fine tune the widget. This is most apparent when comparing XAW with java.awt and Swing.

The results of this trend also appear in the functional toolkits. Here, all the libraries provide very simple widget constructors for widgets with a lot of default behaviour.

However, they don't agree as much on the concept of Model-View-Controller, which is very common in the imperative toolkits. All the imperative toolkits use call back functions to serve as a controller. With the exception of Fudgets that uses a different system with their in- and output streams, the controller parts of the functional toolkits all show that they originate from call back functions, but they apply it in different ways.

This is for a large part due to the pureness of the language. An imperative call back function can safely jump out of the context, using some global variables, but in a pure functional program the call back function has to contain all the arguments that influence the result or are influenced by it. This makes it a lot harder to actually define such a call back function.

Therefore traditional MVC, and its typical implementation, may not be suited as well for functional languages as for imperative, object oriented languages. Therefore we developed an alternative framework for GUI development, which we present in chapter 4. In this chapter we will first continue our discussion of the general trend in GUI toolkits to provide components with ever increasing amounts of default behaviour, and amplify this trend with our selectors as discussed in section 2.7 and later sections as well.

Common in all toolkits is the preference for defining specific widgets. The constructors clearly state which widget they construct. This is certainly more declarative than describing the actions that have to take place when certain events occur on certain places of the screen. It therefore makes things easier for the programmer when he only wants to construct that widget, but when he needs a (large) number of such widgets to achieve some desired larger effect this is no longer declarative. The programmer has to describe too much *how* the effect is accomplished by composing the widgets and programming the accumulated effects, instead of simply referring to the desired goal.

For instance the radio button[3] is directly supported by some toolkits, whilst others let the programmer define a set of buttons and require him to add a rather abstract widget, a *radio* that controls the different buttons and that makes sure that only one button is in a selected state.

This radio button is only a specific example, but, in general, the toolkits we discussed, support in very different ways precisely this process of combining separate widgets into more powerful entities such as requesters or command and information windows, both for logical composition and for combined layout. No system seems to be superior over others and thus research on this domain may still prove worthwhile. In chapter 4 we will offer yet another alternative.

Also note that because the libraries force us to choose specific widgets, the software is less adaptable to changes. If we want to change the way in which some

[3]A radio button is a set of buttons of which only a single one can be selected, like the buttons on a radio to select the channel.

option is chosen, e.g. by replacing a button with a poll for a key press, this may involve quite some changes, because the choice for the button was hard coded.

So we should try to make such changes easier by stressing more on the intended goal of the widget and less on the type of the widget. In the section on selectors , section 2.7, we will return to this point.

Because widgets are so commonly used in all the libraries, we feel that an enhancement to the widgets of one library should be easily transportable to other libraries. Our research in this context thus is to be widely applicable.

Another certain fact is that many user interface toolkits are very complex, a fact that has also been acknowledged by SUIT, the Simple User Interface Toolkit [61]. One of SUIT's goals is to provide a widget set that must be usable by undergraduates in under three hours. SUIT therefore tries to hide the (object oriented) hierarchy present in most toolkits and only provides very basic widgets.

We believe that the goal of providing a toolkit that can be quickly learned may also be fulfilled by providing more declarative abstractions for user interface goals.

2.6 More Advanced Libraries

Several more advanced libraries or systems already try to move the focus from individual widgets to more complex constituents of a user interface.

When using individual widgets the programmer still has a lot of work assembling and adapting the widgets to his intention. If we identify some structures that occur frequently in user interface designs, we can provide better abstractions for such structures. This enhances reusability, but it is also more declarative as the higher abstraction more clearly expresses the goal of the substructure than the composition of the individual widgets.

Such a structure can e.g. provide a number of edit fields and buttons and take care of the communication within the structure. The programmer will only need to instantiate the structure appropriately avoiding the tedious task of assembling and adapting the widgets.

2.6.1 Visual GUI Builders

Visual editors such as Visual Basic and those present in many programming environments certainly make life easier for the interface designers. They seldomly provide larger user interface structures themselves, but instead they help with the layout of the widgets, which is an important aspect when combining widgets.

By sketching the elements on a drawing pad, the layout can be designed in a *WYSIWYG* manner, already focusing more on *what* will be seen by the user and less on *how* the programmer defines the widgets. After drawing the widget the interface designer can often set a large number of options simply by selecting from a pull down menu of similar designer friendly operations.

The tool will then generate the code that constructs the user interface. The programmer has to supply more code to connect the application to this user interface.

This may be a suitable approach for rather static interfaces, but not so much for more dynamic ones as it's hard to draw such a dynamic interface on a static drawing board. The drawn design may still be useful as a starting base that is dynamically changed at run time, but it still suffers from the previous disadvantages as it mostly remains focused on individual widgets.

After all these visual editors only provide a solution for the problem of layout.

2.6.2 ReqTools for Amiga

The purpose of ReqTools [28] is to make it a lot quicker and easier to build standard requesters into a program. ReqTools is a runtime library for Amiga, designed with Commodore's style guidelines in mind, ensuring that all requesters have the look-and-feel of AmigaDOS Release 2. Requesters are rather complex if they have to be implemented from scratch, but they certainly share so much common behaviour that they can be easily put in a standard library.

ReqTools provides standard requesters such as confirmation boxes or *rtGetString* and *rtGetLong* that select a String or a long int, but also more specific requesters that ask for a screen mode [4], a file or a font. This idea is currently also applied in the Swing classes for Java (see also section 2.7).

- *ret = rtEZRequestA (body, gadfmt, reqinfo, argarray, taglist);*
- *rtEZRequest ("ReqTools Demo", "OK|Cancel", NULL, NULL);*

Figure 2.18: A confirmation box from ReqTools.

The *rtEZRequest* function puts up a requester and waits for a response from the user. If the response is positive, it returns the True value, otherwise False. This is a very typical requester that is used in many programs and ReqTools takes care of all the work behind the screens.

The *rtGetString* function and others such as *rtFileRequest* open up requesters that select a specified type of data. This is very declarative as it allows us to focus on what we want the user to select. The ReqTools library takes care of the construction, deconstruction and layout of the requester and makes sure that the input conforms to the criteria desired by the programmer.

[4] An application on Amiga can open its own virtual screen with a resolution different from the workbench. Well written programs ask the user for the resolution of the screen. That is precisely what the screen mode requester is meant for.

- *rtGetString* (*buffer*, 127, ”*Enter string* : ”, *NULL*, *TAG_END*)
- The return value indicates whether the user successfully entered a string. The variable *buffer* contains the entered string.

Figure 2.19: An example of a string requester from ReqTools.

Apart from the clear advantages, the fruitfulness of this approach is also proved by the fact that the majority of Amiga software, and certainly shareware software, effectively uses ReqTools.

But we can still do a lot better. ReqTools provides only solutions with requesters. If we want to select data in another way, we have to revert to the old way by selecting specific widgets and assembling them to get the desired effect.

2.7 Selectors

In sec. 1.4.7 of our mission statement we already outlined our special interest in the Selectors of J. Johnson. Selectors are higher level alternatives for widgets that focus on the *selection* of values, an important subtask of UI’s, abstracting away the specific appearance of individual widgets.

J. Johnson has analysed a number of archetypical user interfaces and the various types of interactive controls used in computer applications. Classifying them according to their semantics, a first distinction between two types of selectors has been made:

- Most widgets control application variables. Users can either freely set those variables (as in an edit fields) or choose between distinct values (as in radio buttons or menus). Typical is the fact that they set application *state*.

- Other widgets indicate available actions and allow users to invoke them. Such widgets trigger *commands*.

Shared is the behaviour of *selecting* something. That’s why their equivalent higher level structures are called *selectors*. The first kind is named *Data Selectors*, the other kind *Command Selectors*.

Data Selectors

This set is the most interesting to explore as it covers a rich domain. Applications have many task-specific data types. We don’t try to define data selectors for each of those types – it would be too time consuming and still remain a futile attempt –, but rather define generic data selectors.

After all, even then, independent of what type of value is selected, a lot of degrees of freedom remain:

- Some controls let the user select *one* value from a small, discrete set of values, e.g. the sex of person, either Male or Female.
- Other controls let you select a *number* of such values, e.g. the styles for a font: bold, italic, underline, none or a combination.
- Others specify that you should select *exactly 3* values, whilst others might indicate *upper and lower limits* to the number of values to choose (e.g. Pick at least two subjects for your master's thesis, but no more than five.)
- The set *from which* to choose from may also be different, selecting from a potentially large, continuous range of values (e.g. dollars to bet: $0 - $1000).

Summarised it seems that there are two factors :

- The number of values to select:
  - An exact number, e.g. 1, 3, or 17.
  - *Any* number.
  - A range is allowed, e.g. any number of values between 1 and 3, or between 0 and 4.
- From what range to select:
  - A discrete set of well known values, e.g. Black, White or Red.
  - A range of values, e.g. 0 - 1000, 10:00 - 17:00.
  - No such constraints, except for the type of the value, e.g. any integer will do.

Different combinations of those two degrees of freedom give different kinds of selectors and all of Jeff Johnson's Data Selectors fall into these categories. Because his Data Selectors encapsulate the semantics of *choice* and he considers choice to be like selecting from a fixed set, he does not support the category that poses no constraints on the values to be selected.

Although the Data Selectors cover most of the other categories, the internal representation of a data selector is very uniform. The domain – the values to select from – must be a set, either explicitly enumerated (e.g. {Jan, Piet, Joris, Corneel}), given by a range (e.g. [1..25]) or by a predicate function (e.g. isFemale(Person)).

The value selected by the Data Selectors is also a set, but it is restricted in size to a specified minimum and maximum. In the case of the Data Selector that selects a single value, minimum and maximum or both 1. Notice that the Data Selectors of J. Johnson do not support the empty set. They don't consider the choice of selecting nothing as a valid choice. They can only be used when something must be selected.

The richness in the number of data selectors is also present in the question in what manner to *display* the values to choose from. This is an entirely different question from the matter of defining what and how many values to select. The notion of Data Selector *Presenter* separates this design decision properly.

The type of the value that can be selected is a first issue. For a collection of basic data types (e.g. numbers, truth-values, colours, times and dollar amounts) Johnson's system defines a collection of presentations, e.g. a colour can be both represented as a rectangle filled with the specified colour or as a colour name.

Secondly, for each of the Data Selectors different Presenters are provided, e.g. a one-from-N choice can be represented as a set of radio buttons, a pop-up menu or a cycle button[5].

By selecting a Data Selector and by crossing these two presentation options, the toolkit claims to provide a much greater variety of interactive controls than most user interface toolkits can, which increases the likelihood that with little effort the right intention of the user interface (selecting some data) with the right visual presentation can be constructed.

It certainly is a lot more declarative because it forces the programmer to express precisely what data he wants the user to select, a design decision that is directly and precisely reflected in the code. This largely simplifies the task of changing the code to include other choices. We just have to change the *domain* of the data selector. In a typical user interface toolkit this would require quite some *how* code to define *how* to add and display the extra choices and *how* the user can select these values, and to a far lesser degree *what* code: the simple fact that some extra choices have been added.

As an extension we can easily consider more abstractions. Currently we are in the first place abstracting away the actual appearance of a selector, and rather focusing on the selection process. Later on we do define an actual Presenter, but one can easily imagine that this precise selection is abstracted away as well. Just as higher order functions that can contain a strategy how to calculate something, one can enhance this system by having the selectors visually instantiated by a particular strategy for choosing the right presentation. We could even abstract that away and define strategies that define the presentation strategy, depending on features such as the size of the targeted display or the expertise of the intended users. However this hasn't been expanded in our research project.

This data selector can be easily added to the system of Model-View-Controller. While the simple widget triggered events such as *ButtonClicked*, *WindowGotFocus*, ..., the selector can trigger events as *Selected 3*, *SelectedSet [1,2,4,17]*, ... The controller must then simply react on those events instead of on the more basic widget events.

[5]A cycle button is a button whose value changes whenever somebody clicks on it. The value cycles through a fi xed set, returning to the fi rst value after clicking on the last value of the set.

Command Selectors

The most typical form of a Command Selector is the button or drop down menu, used to invoke an application command.

Naturally any user interface library provides ample support for the button. Some libraries only allow the most typical form, a rectangular 3D-area, whereas others also allow buttons to be transparent such that they can be laid over other objects to make those objects mouse-sensitive.

Providing a separate category for this kind of command selectors may be a nice conceptual classification, but unfortunately it is less elaborated in Johnson's work.

Just as with Data Selectors, the notion of *choice* is supported, a choice that is present in the domain of the selector. The domain of a command selector is called a *Command List*. An example given in the paper is the pair of commands *OK* and *Cancel*. They might be represented as a pair of buttons, a menu with two entries or a text field that only accepts either "ok" or "cancel", but this is a bit an artificial example as the OK/Cancel pair is almost exclusively used as a button pair.

The second part of Selectors, providing *Presenters* is also rather poor for commands. However, this is quite normal as there is no natural way for depicting a command.

The distinction between selecting data and commands may be critical in a typical imperative setting, but, luckily for us, this is not the case in a functional language with functions as first class citizens, where commands can be considered just as other data.

Selectors in Java Swing

Although a number of research projects such as Jade [87], SEGUIA [85, 10, 11] and Genius [13] apply ideas similar to the selectors approach, the idea hasn't gained wide attention in the public field and it is present in only very few toolkits, at least some selector elements got their way into Java's swing classes. Although Eckstein, Loyd, Wood [23] call it the *next-generation GUI toolkit,* it only adopts parts of the selector ideas.

One of its main innovations is the fact that the *look-and-feel* can be changed at run-time. And instead of providing a smallest common divisor, it defines its own default look-and-feel (Metal).

It certainly is a next generation from java.awt. Components that were available in java.awt are also available in Swing, but the new classes are organised in a better way. It also offers new – more declarative – components such as tables, trees, progress bars, . . .

A disadvantage may be that programmers can still use the old style of GUI programming, using classes as *JButton* and *JFrame* to place buttons and other widgets on frames and adding objects that listen to events coming from those widgets to define call back behaviour.

On the other hand it may also be called an advantage because it may softly introduce the programmer to more declarative constructs such as *JtextField* that selects a text value or even larger ones such as a *JtabbedPane* that presents the user a set of tab browsable options, and *Jtable* that displays information in a table form. Other features such as the *JDialog* have already been discussed in previous sections (e.g. ReqTools [28]).

These classes are indeed declarative as they focus on desired behaviour instead of on the composition of widgets, but they still focus more on the actual visual presentation than on the semantic model.

Separating those is mostly achieved by the model-view-controller relationship. e.g. the *JList* that offers the user a selection from a list of options typically holds a *ListModel* that keeps the list of items to select from, and a *SelectedListModel* that keeps the selected list of items.

JList therefore, as well as e.g. *JFileChooser* are classes that focus the most on the selection of data, as in the Selectors approach of Johnson, although they provide fewer options for showing the user interface structure in different ways. At the very least Java Swing introduces the concept of selectors to the main stream programming community.

2.8 Implementing Selectors

We first present our implementation for Fudgets, beginning with a discussion of some general issues, and then talk about the work in TkGofer. Shared between these two systems and typical for a solution in Haskell is the fact we cannot use mutable variables connected with the selector to contain the variable that is selected. In many imperative toolkits we have such a variable that is updated whenever a value is selected. Programs written in such toolkits then simply must watch the variable and react on state changes. This process is rather easy to implement in imperative languages, but very hard in pure functional programs. That is therefore a difficulty we face, but it is also an opportunity as the typical solution doesn't excel in transparency. It is often hard to see in the code which variables are connected with which selectors. The connection can be declared far away from the declaration of the variable itself. If we can propose a new, more transparent alternative, the program itself can be become a better reflection of the intention of the programmer, thus improving in declarativity. Luckily, by following the conventions of either the Fudgets or the TkGofer library, this turns out to be the case, proving the advantages of the selectors approach and the general applicability of the idea.

2.9 Issues about Selectors for Fudgets

Before we present the selectors that we implemented using the Fudgets library, we have to discuss some general issues influencing the implementation.

We first an intuitive model of the fudget selectors. Next, we describe the general type of a selector, which includes a short preview of possible run time modifications. Finally, a more difficult issue that is discussed in greater detail concerns the visual appearance of a selector - especially the type that should describe the appearance.

2.9.1 Combining Selectors and Fudgets

It turns out that Selectors mix very well with Fudgets. Fudgets are defined such that they place values on their output streams as a reaction on user events. Selectors can do precisely the same: when a value is selected, it is placed on the output stream of the Fudget selector.

Our selectors are not just *more declarative* than custom-made fudget combinations. They are also a lot more *easy to use*. Remember that in a fudget program output streams of single fudgets have to be connected to input streams of other fudgets, resulting in a lot of wiring. As a selector is a lot higher level than simple widgets, our Fudgets selectors embed and replace many fudgets. In fig. 2.20 we can see that an important part of the program structure and wiring of in- and output streams is replaced by the selector, thus largely simplifying the design and implementation of Fudget programs. As a matter of speaking a selector can be considered a hyperfudget. The fact that this hyperfudget is composed from a set of fudgets $F_1 \ldots F_n$, connected in parallel, preceded by F_m amd followed by F_a, together with a feedback loop is hidden to users of the hyperfudget.

2.9.2 A Type for a Selector

An important consideration is of course the type of a selector. Because selectors are not intended to replace the *entire* Fudget library, we must take care that they mix well with the common fudgets. In a fudget program larger structures are created using general fudget combinators as $>==<$ and $> + <$. We want our selectors to be usable in such combinators as well. That way the selectors can replace complicated but general substructures of fudget programs.

Then, of course, our selectors must have the same type as all fudgets, i.e. *F a b* with values of type *a* on its input stream and values of type *b* on its output stream.

It is however an oversimplification to say that all fudgets have the *same* type. Their type always is *F a b* but the *a* and *b* depend on the specific fudget. A button for example will always have *Click* both as *a* and *b*. The *intInputF* has type *Int* on both streams. Others fudgets as e.g. *moreF* have different types for *a* and *b*.

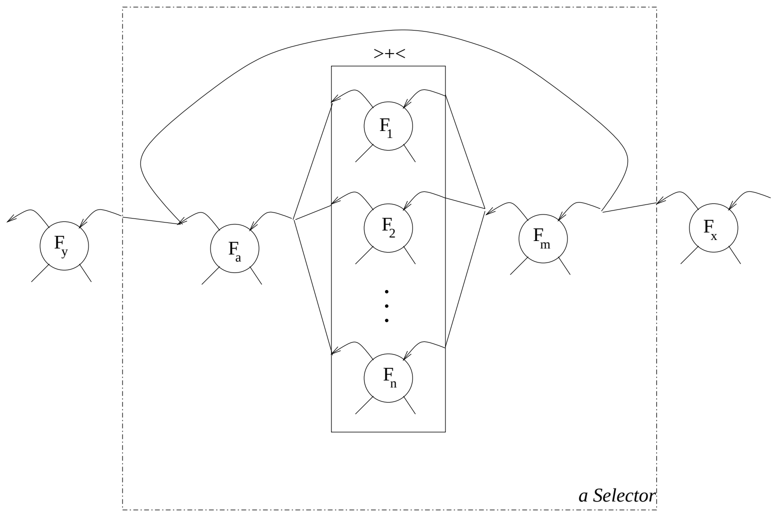


Figure 2.20: (Hypothetical) Structure of a Selector in Fudgets.

So although we use many different basic fudgets in our selectors, such as buttons, toggle buttons, menus or scroll bars that have very different types because they implement different behaviour, we have to hide that to the user of our selectors. Whether the selector is shown as a row of buttons, or as a pull down menu, should not influence the selector's type as that is precisely what we are trying to abstract away.

What really matters is the fact that the selector selects something (the *what* in terms of declarativeness), not the way in which it is displayed on the screen or *how* the user is to select the data.

So, if the goal of the selector is to select, it is fair enough to simply place the selected value on the output stream. We have selectors as *selectOneFromDiscrete* that select *one* value, so they will simply output *one* value of type *b*. Others such as *selectManyFromDiscrete* select *many* values, so they put *a list* of values, or a value of type [*b*], on the output stream.

What a selector can accept on its input stream is a less trivial question.

Buttons can accept *Click*s that they propagate to the output channel. Similarly, *menuF* is of type *F a a* with *a* the type of the menu items. Just as the button it propagates any input to the output.

Other fudgets really respond to their input. e.g. a toggle button accepts boolean values to switch the button on or off under program control. A radio button group takes an input to set the active button in the radio group.

It is such behaviour of being able to customise the selector under program control that we would like as well for our selectors. A more trivial aspect is that of changing the current selection, but that is just small matter. More important is changing the values from which the user can choose. It is a behaviour that occurs quite frequently in actual programs, but mostly involves a lot of work. Our selectors also take care of this work.

Other things that may change at run time are the order in which the choices are shown and the (temporarily) disabling of individual choices.

All this run time behaviour is controlled by putting specific values of type *SelectorData a b c* on the input stream of a selector. We come back to this discussion in section 2.11 when we will have seen some actual selectors.

Adding up, the type of selector that selects *one* value is $F\ (SelectorData\ a\ b\ c)\ a$. If the selectors selects *many* values, it is $F\ (SelectorData\ a\ b\ c)\ [a]$.

2.9.3 The Appearance of a Selector

Although we abstract away the appearance of the selector, both in its type and in its use, actually being able to define how it is to be displayed remains a crucial element. We already pointed out through Jeff Johnson's work that there are two separate issues: the way in which the user can select the values (through buttons, menu, . . .), and the way in which the selectable values are displayed: e.g. president candidates may be displayed using their name, their parties or using their photos (fig. 2.21).

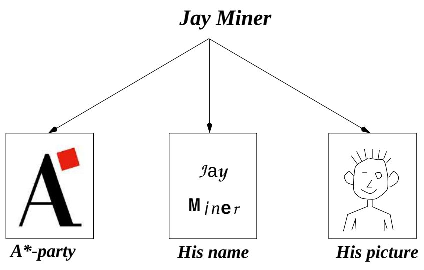


Figure 2.21: Three Views On a Person.

Displaying an individual value

In an object oriented language values can contain methods to describe how to display the value. In that case we would only have to supply the selector with the values ('objects') and apply the draw method.

Using the fudgets library we might indicate that the objects should be of the type class *Graphic a*. The class Graphic collects types whose values have graphical representations (just like the class Show collects types whose values have textual representations). Most basic types such as *String*, *Int* and *Bool*, belong to this class and the fudget library also provides some new types for *Graphic*.

This may look like a decent way for constraining the values that can be selected, but it is really a too strong constraint on the set of types that can be used in the selectors.

Some frequently used data types, such as the function, intrinsically have no visual representation that can be calculated at run time and it would be a shame to exclude those data types from use in a selector.

Also, if we used *Graphic*, we would only be able to display each value in *one* way. Providing different views on the same value would be extremely hard. Some kind of type cast may solve some of the problems, but will also certainly introduce others.

The solution we finally adopted is that when constructing the selector the programmer has to supply a list of *(data value, graphical representation value)* pairs. This allows for a flexible definition of the data and the way in which they are displayed in the selector.

Because of the strong typing of Haskell, in a single application of the data selector each data value must have the same type of graphical representation. We cannot mix e.g. strings with pictures. This can be circumvented by using an aggregate data type, but on the other hand it is also an advantage because it helps ensuring that all values are shown in the same way.

Selecting the values

Another question that the programmer has to answer when he constructs a selector, is in what way the user is to select the data.

For every selector we provide different ways, but not all possible ways turn out to be relevant for all selectors. Some layout schemes, such as the button row or the pull down menu are suitable for a large range of selectors, others, such as the scroll bar are only suited for a smaller set, still others may be usable for just one single selector.

In the end every selector has its own set of ways in which it can be displayed. Some of these are shared by a large number of other selectors, some are particular for one selector, others are shared with yet other selectors.

We need a safe way to express the way in which to display a particular selector, safe in the sense that the compiler should be able to signal when we're trying to instantiate a selector in the wrong way, and safe in the sense that shared appearances should have identical names.

This is not so easy in Haskell's type system and an extra complexity we have to deal with. We've tried several solutions:

1. We may be inclined to define a separate data type for each selector that defines how the selector can be displayed, e.g.

   ```
   data Presenter1 = ButtonRow
                   | ButtonColumn
                   | PullDownMenu
   ```

   This is a type safe approach. When the programmer supplies the selector with a wrong type (i.e. a way to display the selector that is not supported by the selector) a compile error is reported.

   But it is no intuitive approach. Although some graphical representations are shared between different selectors, we cannot use the same data value *ButtonRow*, because it can only appear in *one* data type. Therefore we must define a different data type (e.g. *Presenter*2) for each selector that has a different set of presentation forms. It forces us to supply different names for identical behaviour which will undoubtly confuse the user of our library.

   ```
   data Presenter2 = ButtonRow2
                   | ButtonColumn2
                   | PullDownMenuWithPreview
   ```

2. A better way is to use unified data types, e.g.

   ```
   data Buttons = ButtonRow
                | ButtonColumn

   data Menu1 = PullDownMenu
   data Menu2 = PullDownMenuWithPreview

   data Presenter1 = Presenter1_1 Buttons
                   | Presenter1_2 Menu1
   data Presenter2 = Presenter2_1 Buttons
                   | Presenter2_2 Menu2
   ```

   This way we can reuse the names of the different presentation forms such as *ButtonRow*, *ButtonColumn* and *PullDownMenuWithPreview*, but we have to precede them with an extra tag, e.g. *Presenter*2_1.

   It is still a type safe approach but the programmer can be easily misled between *Presenter*1_1 and *Presenter*1_2 resulting in superfluous compile errors caused by seemingly superfluous code.

3. An easier solution is to use *one* general data type.

   ```
   data Presenter = ButtonColumn
                  | ButtonRow
                  | ToggleButtonColumn
                  | ToggleButtonRow
                  | PullDownMenu
                  | PullDownMenuWithPreview
                  | HorizontalScrollBarWithPreviewAndBuffer
                  | HorizontalScrollBar
   ```

   Now all names can be shared between all selectors without the need for extra indirections. The *Presenter* type can be used as a parameter for any selector, despite the fact that not all *Presenter*s are valid for all selectors.

   So whenever a *Presenter* is used for a selector that does not support it a *run time* error is bound to occur.

   Such a system breaks the rule that *"Typed programs can't go wrong"*. This is clearly not such an ideal situation. It is always preferable to have compile time errors over run time errors, but in this case it is kind of bearable because the run time error will occur immediately when the user interface is initialised, which is most likely at the very start of the program run. It is therefore immediately noticed and fairly simply traceable, especially because it is bound to happen at every run of the program. It is not like a division by zero that only occurs at special occasions. The error is hard coded and *must* occur.

4. Undoubtly the better possibility is to use type classes. That feature allows a function to take different types and have distinct behaviour depending on the type used. A typical example is the type class *Eq*, as the equality function is defined on many types.

   This way we can define a type class for every selector. The instances of the type class will be those presentations that the selector supports.

   ```
   data Buttons = ButtonRow
                | ButtonColumn

   data SimpleMenu = PullDownMenu

   data PreviewMenu = PullDownMenuWithPreview


   class Selector1Presenter a where
     makeSelector1 :: a → Selector1Type
   ```

```
instance Selector1Presenter Buttons where
  makeSelector1 ButtonRow = ...
  makeSelector1 ButtonColumn = ...
instance Selector1Presenter SimpleMenu where
  makeSelector1 PullDownMenu = ...
```

The type of *makeSelector*1 then contains the fact that the Presenter must be suited for the selector: $(\mathit{Selector}1\mathit{Presenter}\ a) \Rightarrow a \rightarrow \mathit{Selector}1\mathit{Type}$.

Another selector that can be displayed using buttons, but not with the simple pull down menu, is to be defined as:

```
class Selector2Presenter a where
  makeSelector2 :: a → Selector2Type

instance Selector2Presenter Buttons where
  makeSelector2 ButtonRow = ...
  makeSelector2 ButtonColumn = ...
instance Selector2Presenter PreviewMenu where
  makeSelector2 PullDownMenuWithPreview = ...
```

This solution combines the advantages of the previous options: it is type safe and it allows for a reuse of the names of the presenters. It is therefore without a doubt the best solution, but at the cost of having to define type classes.

Conclusion Although an optimal solution exists, we have included this discussion because during the development of the library we used the not so type safe solution involving one aggregate data type for all presenters (as described in the third option).

We did this because although not flexible for the user, it is a flexible solution for the designer of the library. If we want to build a new appearance for a selector we can just add an item to the Presenter data type and implement that appearance for the appropriate selectors.

When using the type class approach, it may seem that we only need to add an instance of the type class – which is no more work than in the other approach –, but often this will also result in splitting up a data type that contained several presenters.

```
data Buttons = ButtonRow
             | ButtonColumn
```

In the example the data type *Buttons* contains two options, a *ButtonRow* and a *ButtonColumn.* Now suppose that for some obnoxious reason we want to supply a *ButtonRow* for a selector but not a *ButtonColumn.*

In such a case we would have to split up the data type *Buttons* into two distinct types that contain only one option. This would also mean that we have to create extra type class instances for all the selectors that support the button appearances.

It is clearly undesirable that the structure of existing program parts must change when new features are added. The only natural solution is to define only presenter data types that contain single values. This is OK in terms of compiler acceptance but it is unusual programming practice.

Together with the fact that we moved from an unfinished Fudget version to an incompatible TkGofer version, this explains why the inferior – because type unsafe – approach of defining one aggregate data type for all presenters of all selectors survived so long.

2.10 Selectors for Fudgets

Now that these introductory issues have been discussed, such as the type of a selector, $F\ (SelectorData\ a\ b\ c)\ a$ or $F\ (SelectorData\ a\ b\ c)\ [a]$, and the fact that the constructor needs pairs of $(data,\ presentation)$ that describe the choices and the way in which they will be shown, and a *Presenter* that describes how the items must be chosen (buttons, menu, ...), we can finally present some concrete selectors that we implemented for the Fudgets library.

Johnson made a big difference between data selectors and command selectors, but in Fudgets this is an irrelevant issue. On the one hand, a command in Haskell would be no more than a function. As this is a first class value in Haskell, there is no need to differentiate it from other data values.

On the other hand it may be true that a command selector in an imperative language triggers some action, contrary to the selection of data which will only update the state of a variable. Because of this difference, the two categories of selectors may be fine for imperative languages. In Fudgets the output stream of one fudget will be mostly connected to the input stream of another (abstract) fudget. The first fudget outputting some value will therefore trigger some action in the second fudget, both when 'data' or 'commands' have been selected. Again there's no real reason to differentiate between the selection of data or commands.

2.10.1 A 'command' selector – *commanderF*

Despite these convincing arguments for supplying only one kind of selectors, the *Data* Selectors from the section on Jeff Johnson's work, we have implemented one selector specifically suited for functions.

Remember from section 2.4.2 that a fudget can be considered a stream processing state transformer, stream processing because it accepts a stream of input values and produces a stream of output values; a state transformer because many fudgets keep an internal state that may be changed whenever a value is received on its input stream or when the user performs some action. For example, the toggle button will keep its state (on or off) and the radio button will remember the selected button.

Such state values are values that relate to the state of the user interface, but the developer of a fudget user interface will probably define some abstract fudgets [6] as well to contain application data values. As a matter of fact, such abstract fudgets attached to the output stream of a widget, are particularly used as an alternative to the typical call back functions. Whenever the widget is used, it puts something on its output stream and thus triggers the abstract fudget, which is precisely what a call back function is used for.

The data values present in the post processor abstract fudget can then be changed under influence of the user interface, for example by pressing a button. A typical introductory example is the counter (cfr. fig. 2.14) : a first button increases the value of the counter, and another one resets the value to 0.

This example obviously requires two buttons. But as fudget buttons only emit a *Click* whenever they are pressed, we also need some wiring to translate the Clicks to actions on the one hand and an abstract fudget on the other hand to contain the counter value.

This creates a specialised fudget combination for a simple counter example, but it is a behaviour that can be easily generalised. As a matter of fact, it is a pattern that occurs very frequently in fudget programs: a substructure that keeps an application value that can be changed because of actions in the user interface, such as key presses or button clicks.

- $\mathit{commanderF} :: (\mathit{Eq}\, a,\ \mathit{Graphic}\, b,\ \mathit{PresenterCommander}\, d) \Rightarrow$
  $[(a,\ c \rightarrow c,\ b)] \rightarrow c \rightarrow d \rightarrow F\ (\mathit{SelectorData}\, a\, b\, c)\ c$

- $\mathit{commanderF}\ [(1,(\lambda x \rightarrow x+1),\ \text{"}+1\text{"}),$
  $(2,(\lambda x \rightarrow x-1),\ \text{"}-1\text{"}),$
  $(3,(\lambda x \rightarrow 0),\ \text{"}=0\text{"}),$
  $(4,(\lambda x \rightarrow x*2),\ \text{"}*2\text{"}),$
  $(5,(\lambda x \rightarrow x*x),\ \text{"}\mathit{kwadrateer}\text{"})]$
  0
  PullDownMenu

Our *commanderF* selector implements this behaviour. At construction time, it is fed an initial state value of type c. The goal of this selector is to provide the

[6] An abstract fudget is a fudget without visual presentation. It is still a stream processing state transformer mostly used to connect two fudgets, e.g. with incompatible types.

user with a choice of commands that change that initial value. Each command corresponds to a function of type $c \rightarrow c$. Whenever such a command is selected, the function is applied to the state value and a new value is calculated. That value is put on the output stream of the selector and used as the new state value.

Also note that this selector is not only a special case because it is specifically aimed at commands – functions – and not at 'ordinary' data, but also because of the type of the constructor. Contrary to what we claimed earlier on, this specific selector doesn't need simple pairs $(data, presentation)$, but actually triples (*key*, $data$ (i.e. $function$), $presentation$). The *key* is needed to be able to identify the different choices. Especially when we want to dynamically change the set of choices and say e.g. we want $that$ item removed, we should be able to identify each choice. Unfortunately equality testing is not possible on functions, which is why we add a separate *key*, a unique identifier for the choice. In the example we simply numbered them from 1 to 5.

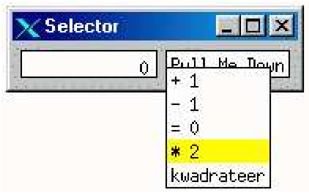


Figure 2.22: A *commanderF* as a Pull Down Menu.

The *PresenterCommander d*, the final argument of commanderF, indicates in which style the selector should be represented. So far we have implemented three styles:

- a pull down menu
- a column of buttons
- a row of buttons

In section 2.11 we discuss the commands that a selector can receive on its input channel to change the set of data the user can select from.

2.10.2 A Note on Data Selectors vs. commanderF

The difference between the 'ordinary' data selectors in the following sections and the previous *commanderF* 'command' selector is not the fact that the latter is aimed at functions. The selectors that will be discussed in this subsection let the user select *any* Haskell data type, thus also functions.

The difference is that the following selectors are ordinary in the sense that they only select some data and put them on the output stream of the fudget selector. They are unlike the *commanderF* that keeps an internal state and outputs the current state every time it changes. It is precisely that behaviour, and no more, that distinguishes the *commanderF* from the selectors in the following sections.

When we discussed the data selectors of J. Johnson, we summarised the two degrees of freedom on page 57. They relate to

- the number of values that is to be selected: precisely 1, an exact amount of values, or any number of values between a specified minimum and maximum.
- and to the values that can be selected from: from a discrete set, or from a continuous range.

The following sections present the selectors we implemented according to this classification.

2.10.3 selectOneFromDiscrete

Probably the most frequently used selector is the one that lets the user select *one* value from a *discrete set* of well defined choices. That was also the first selector we implemented.

- $\mathit{selectOneFromDiscrete} :: (\mathit{PresenterOneFromDiscrete}\ c, \mathit{Graphic}\ b, \mathit{Eq}\ a)$
  $\Rightarrow [(a,b)] \rightarrow c \rightarrow F\ (\mathit{SelectorData}\ a\ b\ d)\ a$
- *selectOneFromDiscrete*
  [(1, “*Een*,”), (2, “*Twee*”), (3, “*Drie*”), (4, “*Vier*”)]
  ButtonRow

Its first argument is the list of (*data*, *graphical representation*) pairs, the second is the presenter that specifies how to display the selector.

As can be seen in fig. 2.23, we have implemented for this selector only the most trivial appearances, namely,

- ButtonRow
- ButtonColumn
- PullDownMenu

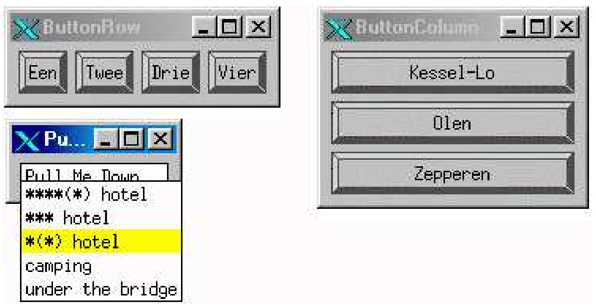


Figure 2.23: Different Appearances of *selectOneFromDiscrete*.

2.10.4 selectManyFromDiscrete

This selector also selects from a discrete set of values, but it is used to select more than one value, e.g. choose between 2 and 5 toppings for your pizza.

The type is like the type of *selectOneFromDiscrete* except that we also need a minimum n_1 and maximum n_2 for the amount of values to select. Naturally the type of values on the output stream is a list of values, because more than one value can be selected.

- *selectManyFromDiscrete*
  $:: \ (PresenterManyFromDiscrete\ c,\ Graphic\ b,\ Eq\ a) \ \Rightarrow$
  $Int \ \rightarrow \ Int \ \rightarrow \ [(a,b)] \ \rightarrow \ c \ \rightarrow \ F\ (SelectorData\ a\ b\ d)\ [a]$
- *selectManyFromDiscrete* 1 3
  [(*Sauna*, "*Sauna*"), (*Pool*, "*Pool*"),
  (*Jacouzzi*, "*Jacouzzi*"), (*Fitness*, "*Fitness*"),
  (*Solarium*, "*Solarium*")]
  PullDownMenu [7]

While the value that was selected in *selectOneFromDiscrete* is immediately placed on the output stream, we won't do so for *selectManyFromDiscrete*. Only when the user specifies that he has selected enough values, the resulting list will be output. No intermediate results are forwarded. One good reason to do so is that the program may rely on the correct amount of values to be output.

Suppose that we want to select between 2 and 5 values and that we output the list each time a new item is selected. After the first selection – on our way to the second – we would then output a list with *one* element. In that case the output of the selector would not conform to the constraints specified, namely selecting between 2 and 5 values. So we certainly don't want to do that.

We might specify that we output the selected values only if and as long as

[7]This example is of course only possible if a data type exists for the values *Sauna*, *Pool*, ...

the value count is in the correct range, but we don't do so because, just as with *selectOneFromDiscrete*, we only want to output a value (list) on *one* occasion.

Therefore the selector provides an acknowledge item that has to be selected by the user before the value list is placed on the output stream of the selector.

Instances of *PresenterManyFromDiscrete* are

- ToggleButtonRow
- ToggleButtonColumn
- PullDownMenu
- PullDownMenuWithPreview

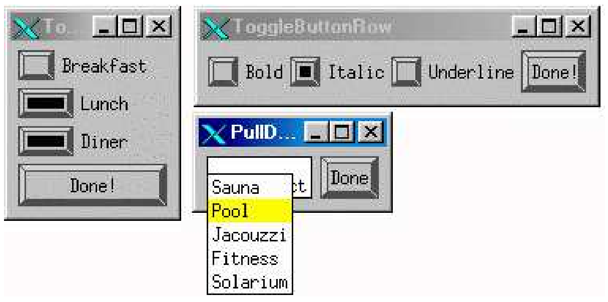


Figure 2.24: Different Appearances of *selectManyFromDiscrete*

Although the menu is commonly used in user interfaces, it is less advantageous when we have to select a specified number of values as it may be hard to keep track of the values that have already been selected. Some toolkits allow the user to add a symbol in front of the menu label to indicate a selected item. Unfortunately the fudgets library does not.

We developed a special pull down menu selector (fig. 2.25) which includes a preview on the set of data that has already been selected. Selecting a value in the right hand menu moves it to the left hand menu and adds it to the selection. Selecting an item in the left hand menu is actually a de-selection of the item and thus removes it from the selection.

2.10.5 selectOneFromInterval

Although this selector could be viewed as a special case of the *selectOneFromDiscrete* selector, it is not. The fact that it is from an interval that we are selecting can be used to its full potential, i.e. we can use a slider bar to visualise better the effect of selecting from a range.

This selector no longer takes the *full* list of possible values. As we're selecting from a range, we only have to specify the lower and upper bounds of the interval.

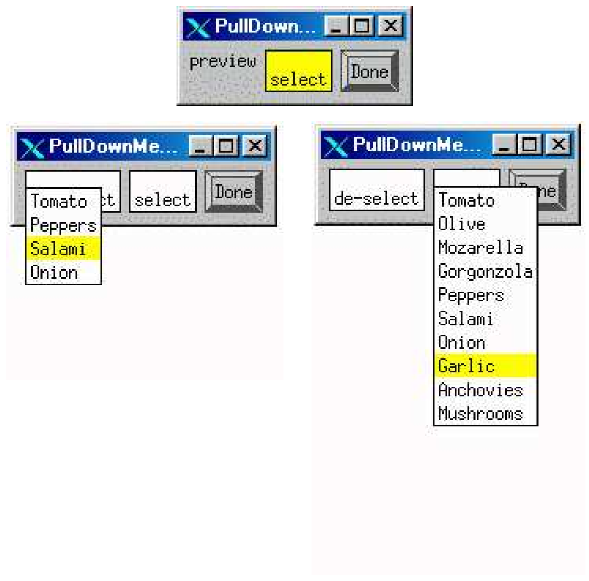


Figure 2.25: *selectManyFromDiscrete* with preview menu.

Of course the type of the selectable values now has to be an instance of the type class *Enum* [8].

- $selectOneFromInterval\ ::\ (PresenterOneFromInterval\ b,\ Graphic\ a,\ Enum\ a)$
  $\Rightarrow\ a\ \to\ a\ \to\ F\ (SelectorData\ a\ b\ c)\ a$
- *selectOneFromInterval* '*a*' '*f*' *ToggleButtonRow*

Another difference is that we no longer use a separate set of values that describe the graphical representations of the elements. Now the selectable values themselves have to be an instance of the type class *Graphic*. It can be cumbersome to take care that the range of data values matches the range of graphical representations, where our goal was to supply an easy to use abstraction over widgets. So we avoided this extra difficulty.

This constrains the usability of this selector a bit, but probably the values that are hard to include in the type class *Graphic*, such as functions, will also be hard to include in the type class *Enum* and therefore little is lost.

[8] In an older version of Haskell, the version that our selectors were implemented in, *Enum* was an instance of *Eq*. Although this is no longer the case in current Haskell, for simplicity's sake we have not included *Eq a* in the constraints for the type of *selectOneFromInterval*.

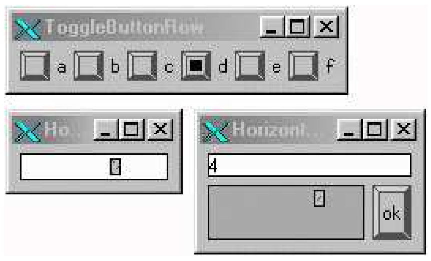


Figure 2.26: Different Appearances of *selectOneFromInterval*.

Although we could have reused some of the presenters for *selectOneFromDiscrete* we defined other presenters for *selectOneFromInterval* to get a wider range of the possible layouts.

- ToggleButtonRow
- ToggleButtonColumn
- HorizontalScrollBar
- HorizontalScrollBarWithPreviewAndBuffer

The scroll bar is the best way to visualise the fact that the user has to select from a pseudo-continuous range. It is specifically suited when the exact value of the selection is not so important, when we just want to express the fact that it is somewhere half way. A good example is an opinion poll where the subject is to express in what degree he agrees with some statement.

When we need a precise value it is better to display the current value of the selector, for example the amount of red in a colour (e.g. between 0 and 255). The scroll bar with a simple preview will continuously submit values to the output stream of the selector. This way the application can conveniently show the effect of selecting that value and display what changes when the selected value changes.

Sometimes, in the case of a time consuming algorithm, it can be annoying for the end user that the program continuously takes the currently selected value into account. In such a case one should use the *HorizontalScrollBarWithPreviewAndBuffer* presenter. This selector then buffers the value and only places it on the output stream when the user presses the acknowledgement button. An example of this presentation can be seen in the lower right corner of fig. 2.26.

2.10.6 Other selectors

Combining all the degrees of freedom from page 57 would give 3 * 3 = 9 selectors. As we changed the implementation language for our project after a year or two, we didn't finish the implementation in Fudgets of all the possible combinations.

On the one hand we have the selection of one single value without any restrictions on the value to select, except for the type of the value. As the fudget library already supports to a large degrees such fudgets, e.g. *intF* and *stringF*, fudgets that select an int or a string respectively, we did not implement our own versions of those generic fudgets.

On the other hand we have very complicated selectors, but also less frequently used ones like selecting one subrange from a specified interval, or just as we had select*One*FromDiscrete and select*Many*FromDiscrete, one can also imagine selecting a set of intervals within a given interval.

We did not implement such selectors because they represent rare selection patterns. The more likely selector, selecting a subrange from an interval, can be easily implemented using *selectManyFromInterval* 2 2 that lets the user select precisely two values from an interval. Those values can then be used as the lower and upper bounds of the interval. The other possibility, selecting a set of intervals, is more difficult to implement using existing selectors, but then it is also a complex selection. Until we will implement such a selector, it may be better to use a number of *selectSubInterval* selectors.

2.11 Modifying Fudget Selectors at Run Time

The previous section described the static initialisation of the selectors we implemented. However a static initialisation is a fine start, but more of a challenge is dynamically changing the selectors at run time, such as changing the set that can be chosen from, or the order of the elements to choose from. This is a lot harder to do in most toolkits and especially in Fudgets as it may involve rewiring new and existing elementary fudgets used in the implementation of the selector. We consider this an essential feature of a flexible toolkit and therefore devote a separate section to it.

To change a selector at run time one must place values of type *SelectorData a b c* on the input stream of a selector. For the *data* selectors only the *a*'s for the selected items, and the *b*'s for how they are shown, are relevant. For the *commanderF* we need all three *a*, *b* and *c*. *SelectorData a b c* is constrained by $(Eq\ a,\ Graphic\ b)$ because we want to test equality on the *a*'s, and use the *b*'s as graphical representation of the choices.

Besides the modifications suggested here, some further changes are possible, such as changing the presentation of an individual choice or changing the presentation of an entire selector e.g. from a button row to a pull down menu. We haven't implemented these. The first option is an extension that can be added rather easily as the Fudget library already supports commands for changing the graphical item in a widget, but the second option is a lot more difficult as this also has implications on the other UI elements such as defining a new layout and making sure that the application retains a consistent look-and-feel.

2.11.1 Changing the Order of Presentation

The first issue concerns the last degree of freedom for representing the data the user should choose from: the order of the individual elements.

When the user is selecting from a range, there is of course a natural order present in the range itself. When the set of data to choose from is entered as a list, at first we display the elements in the order of appearance in the list.

Nevertheless, it is not unlikely for the order to change under program control. We may prefer to order the items alphabetically or in reverse alphabetical order, or order cities by the number of its inhabitants, or by the distance to some central point.

This is one of the configuration commands that every selector will be able to receive on its input stream.

$$\begin{array}{lll}\textbf{data}\ \mathit{SelectorData}\ a\ b\ c & = & \mathit{Sorted}\ (a\ \rightarrow\ a\ \rightarrow\ \mathit{Ordering}) \\ & \mid & \mathit{NotSorted} \\ & \mid & \mathit{SortAgain} \\ & \ldots & \end{array}$$

The command *Sorted* takes as an argument the order in which the data values should be presented. *NotSorted* is used to suspend ordering such that new elements can be inserted at random places. Otherwise new elements are inserted in order. *SortAgain* then sorts the list again according to the latest ordering function entered.

Haskell has a type class *Ord a* that contains all types *a* for which an order has been defined, but we did not use that order for a number of reasons:

- It restricts the type of the data values that can be used in the selector. Sometimes we may want to select data that cannot be ordered and stick with the order as presented in the list that contains the selectable data values.

  If we had used *Ord* the data *must* be sortable.

- The type class *Ord* contains only one order. This would restrict the possible orders for the data elements to that order and its reverse order.

- The order present in *Ord* is fixed at compile time. By giving the order as an argument to *Sorted* we can use any order. We can even let the user input which order he wants.

Our approach certainly is a lot more flexible. Changing the order is only optional and when it is used, it can be used for any imaginable order.

2.11.2 Changing the Set of Elements to Choose From

This feature is undoubtly used a lot in actual programs, but it is also a feature that may involve a lot of work as it involves both removing, adding and replacing individual widgets.

Such dynamic behaviour is supported by the fudgets library, but only if the fudget is explicitly created as dynamic. Such a fudget takes a static fudget and has type: *dynF* :: *F a b* $\rightarrow$ *F* (*Either* (*F a b*) *a*) *b*. It has a split input stream. A value on the right branch is forwarded to the embedded static fudget, a value on the left branch (another fudget) replaces the existing static fudget with the new one.

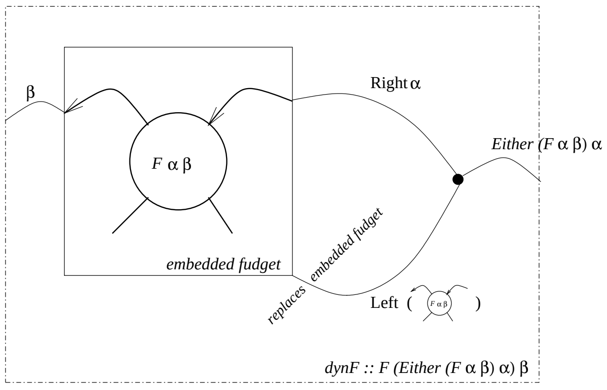


Figure 2.27: The *dynF* fudget.

To be able to support changing the set of elements the user can choose from, our selectors embed a *dynF*, but, as you already did *not* see in the type of our selectors, we nicely hid it to the users of our selectors library.

Discrete Selectors

The discrete selectors are used to select from a discrete, explicitly enumerated set of values. So we can either add or remove individual values. When we add a value, we have to mention both the data value and its graphical representation (a,b).

```
data SelectorData a b c = ...
                          | Add (a, b)
                          | AddAfter a (a, b)
                          | AddAt Int (a, b)
                          | Set [(a, b)]
                          | Remove a
                          ...
                          | SetAt a
```

Because the order in which the elements are shown matters, we can state where we want to insert the new item, at the end *Add*, or after a specific element *AddAfter*. This command takes a data value after which to insert the new item. We traverse the list and add the new element as soon as we encounter the data value after which to insert. Therefore we need the equality test on the data values, which is were the *Eq* constraint comes from. A third option is to add the new item at a certain index *AddAt*. To remove an item we just need the data value *a* to be removed.

The most drastic measure however is replacing the entire set of data with a completely new list. The command *Set* does so and takes a list of (*data value*, *graphical representation*) pairs.

Also note that we can change the currently selected item with the command *SetAt* which takes a data value *a*. Of course, this is only relevant when the current selection can be made visible, e.g. when toggleButtons or a scroll bar are used. So this is also valid for interval selectors.

Interval Selectors

An interval is completely defined by its bounds. Therefore we also need commands that change the bounds, either the lower or upper bounds, or both at once.

```
data SelectorData a b c = ...
                        | Begin a
                        | End a
                        | SetInterval a a
```

Command Selector

Finally, the command selector. That is a special case that is used to give the user a choice of commands – functions – to be performed on some data value. Whenever a function is chosen, it is applied and the new data value (‘state’) is placed on the selector’s output stream.

The run time commands that this selector can receive on its input stream are like the commands for a discrete selector, i.e. add and remove individual items or replace the entire set of choices.

However, just as with the construction of a command selector, only specifying the function and its graphical representation simply is not enough. We must be able to test on equality. How else can we know after which item we have to insert a new choice?

We add a third item: a *key* that identifies the item. A key is needed because we cannot have equality or order tests on functions and neither do we want to do so on the graphical representations as it is difficult to give a simple semantic interpretation to an order on pictures. The commands for the command selector therefore take triples: (a — the key, $c \rightarrow c$ — the function, b — the graphic).

```
data SelectorData a b c = ...
                        | Add3 (a, (c → c), b)
                        | AddAfter3 a (a, (c → c), b)
                        | AddAt3 Int (a, (c → c), b)
                        | Set3 [(a,c → c, b)]
```

The commands are like the ones for the discrete selectors, but they have a 3 appended, i.e. *Add*3 instead of *Add* as the arguments are now 3-tuples and not 2-tuples.

2.11.3 Temporarily Disabling Choices

Removing and adding choices as in the previous paragraphs may be a bit too permanent sometimes. e.g. if we have selected a *Quattro Formaggio* pizza, we can still select extra toppings, but no more cheeses. We do not want to physically remove the cheeses as that might give the unintended impression that the cheeses have left the menu.

We merely want to *disable* a choice.

```
data SelectorData a b c = ...
                        | Disable a
                        | Enable a
                        | ShowDisabled
                        | HideDisabled
```

By default the disabled choices are still visible but greyed or simply inactive. If the programmer wants them to be no longer shown, he can put the command *HideDisabled* on the input channel of the selector.

Although at first sight it may seem that these options are only relevant for the discrete selectors, it makes sense for interval selectors as well, e.g. if the user should select an exact hour for an appointment between 8 AM and 5 PM, 12.00 may be disabled because of lunch.

2.11.4 Known Problems

The data type *SelectorData* is too general. It contains the options for *all* selectors, although not all of them support all options. The discrete selectors have no use for the commands that have to do with interval selectors and vice versa.

The cause for this is similar as in the discussion on a data type for the presenters. The ideal solution there, type classes, cannot be used here because a type class can only be used if we want a function to be able to take different argument types, e.g. $==$ is defined on *Int*, *Bool*, *Char*, ... but always returns *Bool*. And a function *fun* that is defined in several instances of a type class isn't really *one* function. Although the name is shared, there are as many functions *fun* as instances of the type class. The compiler must have enough information to be able to know which specific function to use, otherwise an *unresolved ambiguity* error is signalled.

So suppose that we were to define code like this:

```
data OrderCommand   = Order (a → a → Ordering)
                    | NotSorted
                    | SortAgain
data ChangeDiscrete = Add (a, b)
                    | AddAfter a (a, b)
                    | AddAt Int (a, b)
                    | Set [(a, b)]
                    | Remove a


class Discrete a where
  selectOneFromDiscrete :: args → F a b
  selectManyFromDiscrete :: args → F a b
instance Discrete OrderCommand where
  selectOneFromDiscrete args = ...
  selectManyFromDiscrete args = ...
instance Discrete ChangeDiscrete where
  selectOneFromDiscrete args = ...
  selectManyFromDiscrete args = ...
```

Then apparently, we have implemented a suitable solution because we outlined that both the *OrderCommand*s and the *ChangeDiscrete* commands can be used for *selectOneFromDiscrete* and *selectManyFromDiscrete*. However, because the compiler must know which constructor is used, either the one from the instance *OrderCommand* or from the instance *ChangeDiscrete*, a specific discrete selector will *only* accept either *OrderCommand*s or only *ChangeDiscrete* commands. This is naturally not good because we *do* want to mix *OrderCommand*s with *ChangeDiscrete*s.

The best solution – if we want to reuse the names of the commands, which we do – is to stick with an aggregate data type. If a selector then receives an inappropriate command it is simply discarded as irrelevant, without having to resort to actual run time errors.

Another problem is the fact that the items that can be selected have to be instances of the type class *Eq* because we have commands that rely on equality testing. Unfortunately we cannot simply discard the commands that rely on equality (e.g. *AddAfter a* (a,b)) for the types that do not support equality, because the type constraint remains present in *SelectorData a b c*.

So our claim earlier on, that the *data* selectors can select *any* data, including functions, is not true because functions are not instances of *Eq*.

For users of our library that wish to use the discrete selectors to select functions, a first solution is to self-apply the work around we used for the command selectors: encapsulate the functions *fun* in tuples (key, fun) and create an instance of *Eq* that only tests on the *key* part of the tuple. Then select from the tuples (key, fun) and apply the post processor *snd* that takes the second element of the tuple.

Another solution is to use the rougher versions of the discrete selectors that we implemented with this purpose: they only take *one* command: replacing the entire set of data. We could have defined it such that it can take all commands that do not require the *Eq* type class, but we didn't do so because we would then need to make up slightly differing names for the 'shared' commands, which may confuse the user of our library. And, after all, if a rough version is needed, only a rough version is provided.

2.12 Selectors in TkGofer

Our work in Fudgets on the selectors was our first essential contribution to the field of graphical user interfaces. Later in the project we switched to TkGofer [88] as our implementation language because the way in which a user interface is built in TkGofer makes it easier to build our intended GUI system on top of it. The reasons why will become obvious in the following chapters, but are irrelevant here.

What matters is that we have also implemented some selectors in TkGofer. It was not just a matter of simply transferring code to a slightly different syntax. Fudgets and TkGofer's widgets are in many ways incompatible. They are constructed and laid out in very different ways, but we should be able to abstract that away in the implementation of our selectors. Both in Fudgets and in TkGofer the selectors should be structures that let the user select some value.

The main difference between the two approaches is what happens when a value is selected. In Fudgets the value is then automatically and immediately placed on the output stream. In TkGofer the value is only made *available* to the application, but the application still has to retrieve the value from the widget, using functions as *getValue* that takes an input widget and returns a *GUI v* with *v* the value in the widget.

It is without question that our selectors have to conform to the methodology present in the implementation language. Therefore our TkGofer selectors support a command to retrieve the value from the selector, just as all TkGofer widgets do.

In the following section we present the selectors that we have implemented in TkGofer. It was certainly not our intention to repeat our earlier findings and simply transfer the existing Fudget selectors and their presenters to TkGofer.

Instead we tested some new presentations for the generic selectors, especially the entry, and also implemented a specialised *getTime* selector.

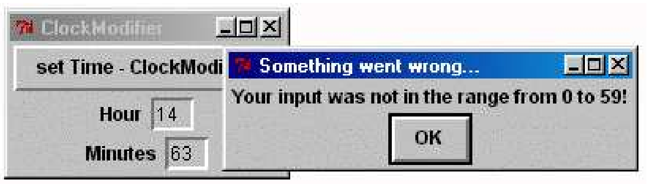
So although we finalised our work in TkGofer, this explains why we followed in this chapter our historical approach: first Fudget selectors and then TkGofer selectors. More work has been done for the Fudgets selectors and they more closely match the work of Jeff Johnson; and after all it *is* historically correct.

2.12.1 Some Generic Selectors

Using an Entry as Presentation

selectOneFromDiscrete. We are already familiar with the discrete selector. The previous presenters always showed the entire range of possibilities. Very often this range may be too large to be shown conveniently, or there may also be other reasons why we prefer not to show all choices, e.g. if in a form other values are to be entered using an entry field, it may be preferable to choose an entry field for the discrete selector as well in order to yield a uniform user interface.

We have implemented a *selectOneFromDiscrete* for TkGofer that uses an entry field. If the entered value is not in the list of acceptable values, a message box is displayed that tells the user an invalid value has been entered.



Your input was not in the range from 0 to 59!

Figure 2.28: An unacceptable value was entered in the entry.

conditionalEntry The idea of an entry that only accepts acceptable values also offers new possibilities. The previous selectors needed an exhaustive list of all choices, either as a list or as a range. This is a normal constraint if we want to display all choices, e.g. in a button row, but if we use an entry we can generalise the constraint of belonging to a certain exhaustive enumeration to that of conforming to a condition.

This has been implemented in a *conditionalEntry* selector. This selector takes a function $a \rightarrow Bool$ to check the entered value and a *String* that is displayed in a message box whenever the value does not conform to the condition.

The *selectOneFromDiscrete* using an entry actually has been implemented by means of this *conditionalEntry*.

selectOneFromInterval. Also the condition of belonging to a range can be easily expressed using a boolean function. Therefore also for *selectOneFromInterval* we have used *conditionalEntry*.

Other Presentations

Only for the *selectOneFromInterval* selector a large number of different presentations has been implemented. Original compared to the previous selectors is the use of a list box. Other presentations we're already familiar with are the set of radio buttons and the slider bar.

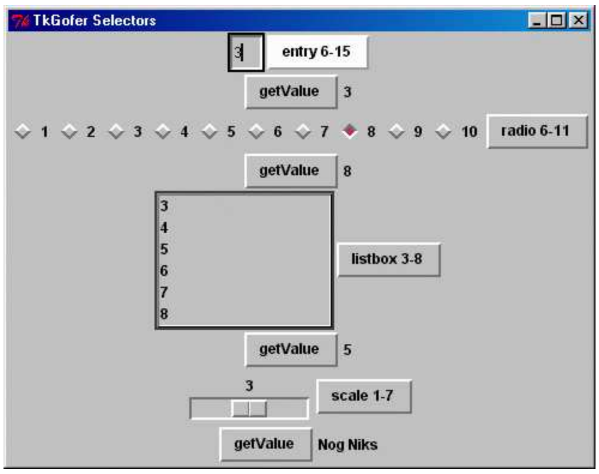


Figure 2.29: Four presentations of a TkGofer selector.

In fig. 2.29 all four representations are present, among with buttons that change the range from which to select and buttons that retrieve and display the value from the selector.

2.12.2 A Specialised Selector – *getTime*

Genericity is always a good thing, but sometimes specialisation can be more fun. Generic solutions often also fail in being able to provide well tuned solutions for specific situations. If we want to select a colour, we can use a generic pattern and provide three interval selectors that select the basic colour gradients green, red and blue between 0 and 255, but we could also define a user interface that creates colours as a painter would, by dipping a pencil in paint bowls with base colours and mixing the patches of paint.

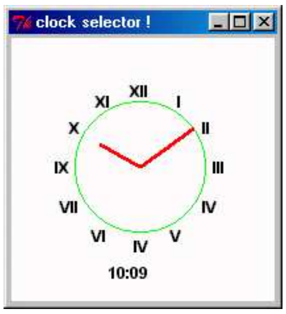


Figure 2.30: Selecting a time using a clock.

A more down to earth example is selecting a time by manipulating the hands of a clock. We effectively implemented such a selector, because it's both a good looking and very intuitive way for selecting a time (fig. 2.30), although it may lack a bit in handiness and preciseness. It is therefore only useful in situations where selecting the time is no principal task of the application that must be done frequently.

2.13 Conclusions

We have shown a historic evolution in the definition of individual user interface elements. Coming from the very small support present in windowing systems as X Windows, most toolkits took a great step forward by offering widgets, well defined user interface elements, such as a button or a menu (item) that take care of an important part of the communication and interaction with the operating system. Defining such widgets was a difficult task in the earlier toolkits as the toolkit provided very few defaults. The programmer had to define everything himself, such as the active area of his buttons. More recent toolkits prefer more standard instantiations of widgets with more default behaviour.

Although well known by now and often defined with a lot of defaults, widgets are still very basic components. It remains a lot of work to create an entire user interface with the tiny building blocks that widgets are. Modern toolkits acknowledge that problem and also supply larger structures. It is however striking that even then widgets retain a dominant position in almost all toolkits.

We also argued that widgets are far from declarative. They are chosen because of their appearance or because of the way in which the user has to manipulate them. Sometimes they badly express typical subgoals of a user interface such as selecting data or triggering commands. Besides defining *how* the user is to manipulate the screen elements, we should start with defining *what* the user interface should accomplish. Some specialised libraries as ReqTools and Java Swing provide (partial) solutions, but remain primarily focused on *how* the elements should be drawn.

Selectors as implemented for an imperative language by Jeff Johnson were the first really declarative building blocks for composing a user interface that we encountered on our quest through UI toolkits. Although they cannot handle all of the often complex tasks of a user interface, we decided to implement a new version of them. Selectors are in the first place useful for a rather static selection of data and not (yet) for the most advanced and dynamic direct manipulation interfaces and for displaying information in general. For those things new widgets remain very useful, just as for implementing new GUI ideas. Higher level descriptions of common tasks are worthwhile, but they should not hinder innovation.

We have implemented a number of selectors for two completely different systems for defining a user interface in a lazy functional language, the Fudgets library and the TkGofer programming language. We have shown that they precisely express the goal of selecting data and that the way in which they are drawn on screen does not influence the behaviour or the structure and definition of the rest of the program. Therefore changing the way the user interface is drawn involves marginal work as we only need to instantiate the selector with a different presenter.

The selectors are also very flexible in defining the set of data that the user has to choose from. They can be changed at run time and the changes (such as adding buttons or redefining a pull down menu or the control function behind an entry field) are performed transparently to the developer of the program.

We haven't had the time to implement an incredibly large amount of selectors, neither in Fudgets nor in TkGofer. We certainly implemented more forms for Fudgets. This is because we considered the Fudget selectors an initial goal that needed completion first, and thus deserved some more attention. At that time Fudgets were the standard and we wanted to explore the *selectors* idea broadly in Fudgets. However, we later found out that selectors alone are not enough for our intention of providing more declarative user interfaces, as we will discuss in the following chapters. In the process of that research we moved to TkGofer. The implementation of *some* TkGofer selectors is more like a final touch to the rest of the work, and not a goal on its own.

The successful implementation in the two systems proves the applicability of selectors. When we consider and add up the selectors for both user interface systems, we have implemented a rather broad range of different presentations for our selectors . And the expertise we built up during the development of these premier selectors, enables us to quickly develop, in a relative short span, more interesting selectors, both for Fudgets and for TkGofer.

We also feel that the selectors idea is applicable in other toolkits as well and therefore deserves further research.

Chapter 3

Visto: An Object System on Top of Haskell

3.1 Overview

As we outlined in sec. 1.4.6 we have developed a prototype based object system on top of Haskell.

Note that this object language has been developed in parallel with our idea for a GUI system. We started with a very simple system but because of the complex requisites posed by user interfaces, the system strongly evolved. The object language in its current form is a fully fledged OO system and provides many original language design features. Although developed in close relation with the GUI ideas, it can now be considered an accomplishment on its own and that is why we devote a separate chapter to Visto's object language.

Visto is a prototypical object language. One of the advantages mentioned by all designers of prototype languages is the flexibility with which new objects can be created. Along with maintaining as much as possible of Haskell's nice features as typing and higher order functions, flexible object creation was our major point of attention.

We continue this chapter with a quick look at a small example leading us through some typical characteristics of Visto.

We then give a quick and informal description of the different forms of derivation – often called inheritance. Visto provides several different ways for deriving a new object from an old one, depending on how close the two objects are to remain connected. We provide support for both completely independent objects as well as for dependent objects as in the model-view pattern [29].

Starting from a basic object definition, with message passing and state transformation, we move on to more complex structures that nevertheless conform to very intuitive concepts. That way we will see the initial, very simple system evolve to a powerful object oriented system with new and advanced features. All these concepts will naturally be illustrated with examples. Then we shortly show that this object language is geared towards UI programming.

We also take a look at some implementation details, discuss a number of restrictions in current Visto, and describe some design alternatives. In a final section related work is compared to Visto.

3.2 A Small Example

Tradition dictates the example of a counter:

```
interface Counter
  up  :: Int
  set :: Int

object PlusOne Counter Int 0 {
  up = let
           s' = state + 1
       in
       (s', s'),
  set = (state , 0)
}
```

This small example already shows some interesting aspects of Visto. A major feature can be seen at line 4: the keyword **object**. Visto is indeed object oriented, combining state and behaviour[1]. A program consists of a collection of Visto objects. In this very small example we have an integer state and two actions for the object *PlusOne*: increasing the value (the *up* method) and setting it to zero (the *set* method).

But before we can actually define the object in line 4 we have to define an **interface** the object has to implement. As promoted in the book *Design Patterns*[29] Visto strongly supports programming to an interface instead of to an implementation. An object implements the **interface** *Counter* if it has two methods, *up* and *set*, both returning an *Int*. This identifies a fundamental feature: every Visto method has a return value. Methods should resemble functions which always return a value as well.

In some ways this interface definition can be viewed as the definition of a type class. The functions in a Haskell type class are like the methods in a Visto interface.

[1]Other essential object features such as method invocation and inheritance are discussed in the next sections.

The header **object** *PlusOne Counter Int 0* defines an object *PlusOne* implementing the interface *Counter*. The type of the state is *Int* and its initial value *0*. One of our early design decisions was the use of *one* state variable for each object. This state variable can have any valid Haskell type, including self defined types. Visto provides the keyword **state** to ease access to this state value.

State transformation is not performed by some sort of assignment occurring at random places, but exactly when the method returns its value to the calling object. Visto methods do not only calculate a return value, but also a new state, therefore returning a tuple *(returnValue, newStateValue)*. Because the state type is hidden for the other objects, it doesn't appear in the **interface**.

The definition of the *up* method can now be easily understood: *s'*, the old **state** $+ 1$, is the return value of the method as well as the object's new state. *set* returns the old counter value and sets the object's state to *0*.

3.3 Different Derivation Forms

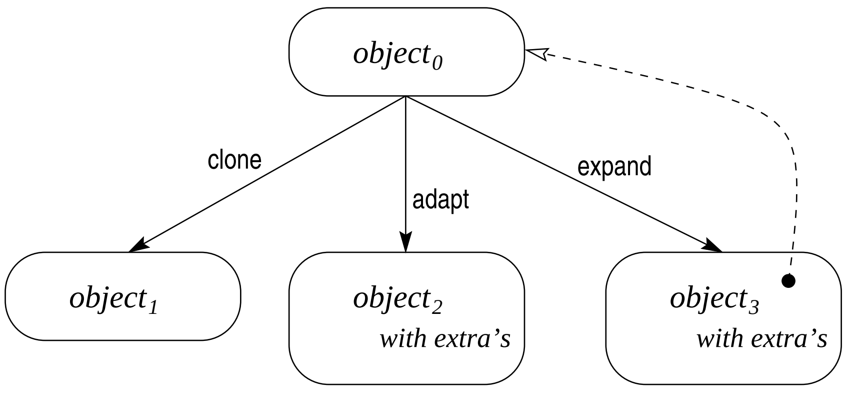


Figure 3.1: Three Derivations.

Prototype object orientation means that we can derive objects immediately from other objects. So far we have been able to distinguish three different forms. We quickly introduce these forms here. In the following sections we return to these derivations with a detailed discussion.

- A first application is the clone. Just like Dolly was cloned into identical but independent sheep, our *clone* derivation creates copies – clones – of an object inheriting all the state and behaviour from the first object. File copying is an example of this mechanism.

The instantiation in a class based system is much alike our cloning, except that we don't need a separate class definition. Instantiating a class creates identical but independent objects that share all their methods. They only differ in state.

In Visto we can do the same: we can clone an object and change the state of the clone by sending an appropriate message, or create a clone and give it a new state at once, e.g. clone a counter object and reset its counter to 0.

At the same time we can also create clones in Visto that provide a new implementation for some of their methods. So our cloning is actually a lot stronger than simple copying or instantiation.

- Sometimes – often – just cloning an object is not enough. We may need more powerful objects, e.g. a counter that cannot only count upwards, but also downwards.

  In such a case we would *adapt* an object. We create another object that inherits (most of) the behaviour of an existing object, but that also adds some extra methods or features. e.g. we clone a sheep, but at the same time take care that it can also bark like a dog. This object, just as cloned objects, lives independently from the originating object.

- The third derivation is the most advanced and at the same time the most "science fiction" if we continue our Dolly analogy.

  Like *adapt* we create an object that may both *inherit* methods and define *new* additional methods. Such an *expand*ed object can have a state that can change independently from the originating object, but on the other hand the object also contains a pointer to the originating object. This is contrary to the other derivations that have no such pointer: that offspring is entirely independent from the originating objects. An *expand*ed object has a pointer to its mother object, that is used for delegating messages (the dashed arrow in fig. 3.1). It can therefore be used to define some shared behaviour.

  For Dolly this would mean that we could have a sheep that lives independently but uses the brain of another sheep. An example more appropriate for computer science is that of the Decorator design. In such a design an object can be decorated with some feature, such as a scroll bar to a text window, or an undo function to an action object. In such a case the decorating object must know the object that it decorates and be able to forward requests to it. This, and even more, is what our *expand* derivation does.

3.4 A Basic Object Definition

3.4.1 The Interface

Visto demands explicit type signatures for every method. They are grouped in so called interfaces that can be partly seen as type classes. An interface defines which methods an object has to implement in order to belong to that interface. Any object may provide a different implementation for each of the methods as long as the type conforms to the interface.

```
interface Clock
  getTime :: Time
  setTime :: Int → Int → Int → Ordering
```

This defines an **interface** *Clock* with two methods, *getTime* returning a *Time*, and *setTime* taking 3 arguments and returning an *Ordering*.

The declared type of a method can be any Haskell type, but, as we remember from the introductory example, the actual implementation will return a tuple consisting of a return value and a new object state[2]. This is not mentioned in the interface because the state type is of no relevance to the other objects. Adding it to each of the method types would not only introduce redundant information, as the state type should naturally be identical for all methods, but is also very error prone. Even more, unless parametrised, this is impossible to express, as objects that share the same interface can have different state types.

Also note that this **interface** is no optional feature. Any object must implement an interface and also state which one, such that the Visto implementation can check whether the object conforms to that interface.

Some research (e.g. MIKE [59]) has been done for automatically generating a UI from such method signatures, but we consider this an inadequate situation because on the one hand we don't necessarily want every method to be present in the UI, and on the other hand some or many methods should be present in different forms in the UI. We fear that the programmer has too little control over the automatically generated UI, besides other disadvantages of generated UI's we mentioned on p. 3 of our mission statement.

Comparison with Class Definitions

This **interface** is not to be confused with a class definition. A class definition will often contain type signatures just as our interface does, but it also defines an implementation for each of the methods. Even more, providing an implementation for

[2]After consuming all of its arguments a method of type $a \to b$ will return a tuple of type $(b, \mathit{stateType})$.

the methods is one of the most important tasks of the class definition. It makes sure that all the objects that are instantiated from this class share the same behaviour.

Such a characterisation is *not* present in Visto. Objects that share an interface are *not* guaranteed to behave identically. Stating that they conform to a particular interface, only implies that they provide the methods of the interface, with the specified types.

A main feature of class based object orientation thus is the focus on the implementation. Current design directions prefer to focus on the interface [29], just as Visto does.

Of course, Visto objects need an implementation for their methods, just as in any other object language. When we define the object, we also define the methods. This is more flexible than in class based languages without interfaces, because objects that implement the same interface can have different implementations for particular methods.

Java also supports this concept of *interface*. As Java is class based, the interface doesn't contain implementations, but only describes which methods a class must have in order to comply with that interface.

Most current class based languages, amongst which Java, offer the notion of an *abstract* method: a method that is present in a class but only by its type signature, without providing an actual implementation. This way it becomes easier to program to an interface.

Because objects can only be derived from classes that are fully implemented, no objects can be directly instantiated from such an *abstract* class. First another class has to be constructed, inheriting from the abstract class all or most of the methods that *have* been implemented and providing genuine implementations for the abstract methods. Then, an actual object can be instantiated from the derived class.

Abstract methods may help in programming to an interface, but having interfaces as a concept, as is the case in Visto, of course remains a lot easier.

Another important difference between a class definition and an interface is that a class definition also contains the data members of the objects, i.e. the variables that represent the object's state. The interface does not contain anything like that. That is of course on purpose. An interface can be used for objects with possibly very different states. After all, the internal state of an object, and how it is represented, is of no importance to the other objects. What matters is how it reacts to messages.

It is precisely that, and no more, that our interfaces describe, making it easy to have two objects with a different internal representation conforming to the same interface.

Feature or Restriction

In the very beginning we introduced the interface simply to make our translation to Haskell easier. At translation time we need the types of the methods to be able to provide a type safe translation[3].

The idea was that we would use Haskell's type checker in the final implementation to get the actual type of the defined methods. Either we could actually include the entire type checker in our translator or simply wrap a program around Hugs[4] that feeds it the method definition, questions Hugs about the type and returns the replied type to our translator, just like an emacs-mode would. We considered that part to be feasible but rather tedious, and therefore started with requiring the Visto programmer[5] to explicitly state the type of the methods, to make our life as a translator easier. At that time it was considered a restriction rather than a feature, which would be removed at the time of the finalisation of our project.

As the system evolved, we realised that extensive documentation is a vital aspect of software. This is obvious for comments that explain the algorithms and the usage of the defined objects, but language features can also help imposing valuable documentation. The interface is such a vital part of the documentation. If we would let the system find out the type of the methods of an object, only the system would be aware that some objects – or perhaps many objects – share a common interface. Even if the system were to report that to the programmer, it would still give the impression that this is merely a coincidence.

However, in a properly designed object program it is *no* coincidence, but intentional. Such important design decisions *have* to be present in the program and therefore the presence of **interface**s is a desirable feature.

It also helps in facilitating "programming to an interface" as currently promoted in various books, among which the famous *Design Patterns* book [29].

3.4.2 The State of an Object

We have opted for only *one* state variable. This is completely unlike other object oriented systems, especially the imperative ones, that can have a large number of data members. Consider for example a point in Java:

```
public class Point {
   // data members
   protected int x,y;

   // methods
   ...
}
```

[3]Why that is so, is of no importance here, but will be discussed later in this thesis.

[4]Hugs is an interpreter for a language that is a close variant of Haskell.

[5]Which we were of course ourselves.

Here we have two state variables, x and y, which can be read and set in each of the methods. We preferred to go with *one* state variable because we find it more natural in a functional environment. In a simplistic view a Visto object can then be considered equivalent to a single Haskell value. A more correct view then depicts the object as a value combined with the methods that can be applied to the value, thus both clarifying and restricting the ways in which the value can be manipulated; clarifying because in the source code the methods are close to the value, restricting because the methods are the *only* way in which the value can be manipulated.

Consequently state changes become more natural. We only have to update the single state variable. We also want the state changes to be atomic, i.e. to happen at uninterruptable and well defined times. In the initial, extremely basic model the state changes only happen at *the end* of a method by letting the method return a new state value. Because Haskell functions cannot have more than one return value, it is no more than natural to have only one state variable[6]. Even now, we already have to tuple the return value of a method, because a method must also have a return value that is passed to the caller of the method.

If we had allowed many state variables, a method would have to return a $n+1$-tuple, with n the number of state variables. This can easily become unreadable, even without considering problems concerning the order in which the state variables should appear in the tuple. Another option in the case of multiple state variables is having assignment operators that modify particular state variables, but we deliberately refused to do so because that has no functional *look-and-feel*. Unless simultaneous assignments are used, a feature not supported by TkGofer, our implementation language, updating would also become incremental: first one variable is updated, a bit later another one and so forth. There would be no longer a clear view on the current state of the object.

That is another issue for which we wanted a clear functional look-and-feel. Throughout the body of a method we'd like to have a referentially transparent value for the object's state. That is yet another reason why we let the state change only occur at the end of the method, because that makes sure that the object's state remains unchanged during method evaluation.

Visto provides a new keyword **state** that refers to the state of the object at the beginning of the method evaluation. The value denoted by the keyword can be considered referentially transparent within a method call. During a later method evaluation the keyword may of course refer to a different value.

[6]As this single state variable can be any Haskell type, it can also be a tuple, thus being a composition of a number of 'separate' states into one state variable. The point remains that one reference suffi ces to retrieve and update the entire state of an object.

3.4.3 The Object Definition Itself

The definition of an object, e.g. *ClockModel*, consists of

- a name for the object – *ClockModel*,
- the interface it implements – *Clock*,
- a type for the state – *Time*,
- an initial value – *(Time 10 8 23)*, and
- the implementation for all the methods mentioned in the interface.

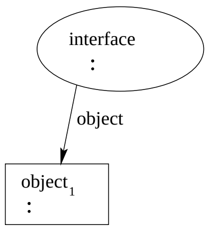


Figure 3.2: Object creation.

Conforming to an existing **interface** we can create an actual object with the keyword **object**:

```
object ClockModel Clock Time (Time 10 8 23) {
  getTime = (state, state),
  setTime = λ h m s → let
                          t = makeTime h m s
                        in
                        (compare t state, t)
}
```

This example shows that the methods return pairs and access the current state of their object through the keyword **state**. *getTime* returns **state** at both positions of the tuple. It returns the current time to the calling object and keeps its state. *setTime* makes a new Time *t*, returning the comparison to the calling object and using *t* as its new state value.

The definition of such a method is just like Haskell code. In practice, especially when we build a Visto program to compose a user interface on top of an existing Haskell program, we will define as little new code as possible in the method definition and use as much as possible existing functions from the underlying Haskell program layer.

That is also why we prefer just one state variable. If it is so important to have multiple state variables, the programmer should have supplied an appropriate data type in the Haskell part of his program. Remember that Visto is intended to build a user interface (and object program) *on top* of an (existing) Haskell program. So we should reuse as much as possible of the functions and datatypes present in the Haskell program.

Communication between objects

Visto objects are no solitary objects. They can communicate with each other and they do this using the well known message protocol. When an object sends a message to another, the receiver executes the associated method, returns a value to the caller and updates its state. As method bodies can call other methods we can quickly arrive in a message loop (fig. 3.3).

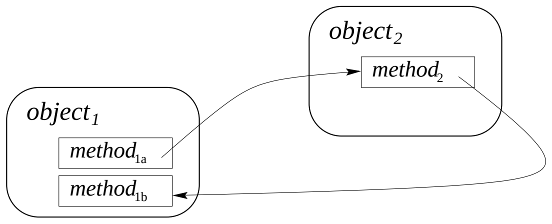


Figure 3.3: A Message Loop.

Instead of dynamically evaluating the **state** keyword, we have chosen to make the keyword **state** a static reference to the state of the object at the beginning of the method evaluation, while within the object itself the state can still change dynamically. Access to this dynamic state is only possible through methods.

So suppose that we would like to define a new method that accumulates the dynamic state changes of two other methods, we would then have to call those two methods by sending the appropriate messages, then explicitly extract the dynamic state, e.g. using a method *getState* and then finalise the method by returning the pair (*returnValue*, *newState*) as in the following code sketch:

```
object test ... {
  doBoth = let
               value1 = test!method1
               value2 = test!method2
               newState = test!getState
               returnValue = (value1, value2)
           in
           (returnValue, newState) }
```

The disadvantage is that **state** now no longer refers to the *current* state of the object, but only to the state of the object at the very beginning of the method evaluation. The advantage is that **state** then refers to the same value independent of the position in the method body. It therefore remains referentially transparant within a method evaluation. Of course, the next time the method is called we naturally look up the new value for **state**, but only once.

Visto evaluates messages strict, even if the return value isn't used. This is necessary because all the objects may change their state whenever a message is sent. We have to take that into account and therefore always evaluate the message at once, at least for the objects' state. Because of the laziness of the underlying TkGofer system, the return value itself will still be evaluated lazily and may be left in some non ground form.

The order of the messages is of course important too. To disambiguate this order, Visto only allows message broadcasts in the top let of a method body and evaluates the messages in the order of appearance in the let. In the right hand side we have the object and its message, in the left hand side the binding of the return value.

This means that we leave the declarative path a bit for the messages to other objects: we rely on the order of the commands, and give up on laziness.

The general syntax for this message sending and result value binding, and an ubiquitous example of a counter only adding in the morning, read as:
returnValue = *objectName*!*methodName [params]*.

```
object MorningCounter Counter Int 0 {
    up  =  let
              now  =  ClockModel ! getTime
              early  =  compare now (Time 12 00 00)
              s'  =  if early  ==  LT then
                         state + 1
                     else
                         state
           in
           (s', s'),
    set  =  (state, 0)
}
```

3.5 Object Derivation – the Simple Way

Visto provides two simple ways for deriving new objects from existing ones. A more complicated derivation follows later, in section 3.6.

3.5.1 Cloning

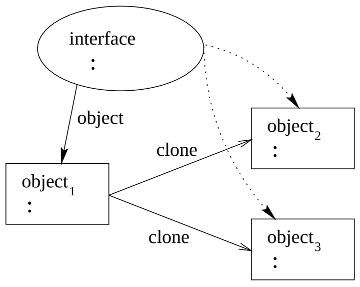


Figure 3.4: Cloning.

Creating a new object from another one while retaining the same interface is called *cloning*. This can be done by just defining a new state, or by redefining some of the methods, as in the following examples:

clone *HighPlusOne* **from** *PlusOne* { **state** = 1000 }

This example shows how to derive an object with just a new state while keeping the same methods as the original object. It's similar to a child saying: "I want the same toy (but with a different colour)." This style can be called *programming by example*. For programmers used to the class based approach it suffices to use this *cloning* technique to define many objects with identical behaviour but a different state. This allows them to easily simulate class based derivation, with the original object functioning as a kind of mother object, a 'class' .

```
clone Doubler from PlusOne {
  state = 1,
  up    = let
              s' = state * 2
          in
          (s', s')
}
```

Doubler is a slightly more advanced example in the sense that we also define a new definition for the *up* method. From an implementation point of view this object *Doubler* behaves distinctly different from the previous object *PlusOne*, but it still implements the same interface and that is what cloning is used for, just as indicated by the dotted lines in figure 3.4.

The state type of a cloned object can differ from the original object's state type. This is fairly useless when the two state types are not compatible at all – it would be better to define the new object from scratch instead of cloning an incompatible object –, but when the two state types are member of the same Haskell type class, the code will be largely reusable.

The **clone** syntax, as well as the other derivation syntaxes, provide support for this feature. Or, because Visto translates to TkGofer, it is actually the underlying system that supports this code reuse.

So, whenever the state type of both the original object and the clone belong to the same type class, and whenever in a method all the functions that use the **state** keyword are within that type class , the method can be reused in the clone. If this is the case for all methods, the entire object code can be fully reused.

Optimal code reuse in clones is thus possible, even for clones with different state type, when the Haskell program on which the Visto program relies, has been cleverly developed with the appropriate type classes.

3.5.2 Adapting

More powerful derivations may add methods, as in creating a counter that can also count down. Because such an object can no longer simply implement the old interface, we first create a new interface by *adapting* the old one.

```
adaptInterface TwoWayCounter from Counter
    down :: Int
```

This creates an interface *TwoWayCounter* with all the methods that were defined in the interface *Counter*, and an extra method *down* with return type *Int*.

This usage of the *adapt* derivation should be quite familiar to class based programmers. There, classes can be derived from other classes by adding extra methods, e.g. as in the *extends* of Java. From that view point, it is just like our **adaptInterface**, but again, because of the difference between prototype and class based, our *adaptInterface* only adapts the interface. A class based extension must at the same time provide implementations for the new methods. Our **adaptInterface** takes the two facets apart, defining a new interface and defining new methods.

The actual definition of an adapted object can start once the new interface has been defined, following the path suggested in **adaptInterface**.

```
adapt PlusMinusOne TwoWayCounter from PlusOne {
  down = let
             s' = state − 1
         in
         (s',s') }
```

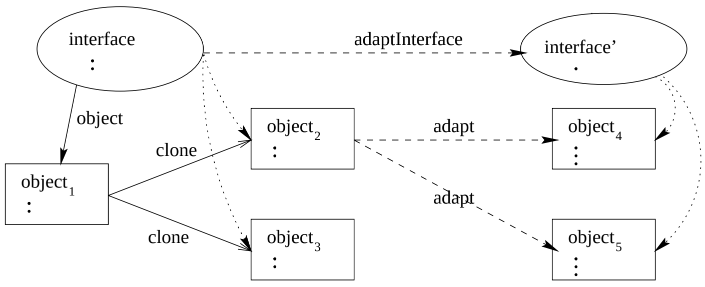


Figure 3.5: Adapting (as well as cloning).

We take as a basis an object *PlusOne* conforming to the old interface *Counter* and add a definition for the new method(s). *PlusOne* must only add one method, the method *down*.

The definition of an adapted object must of course include definitions for all new methods, but, just as in all other object languages, it may also provide fresh implementations for existing methods, thus refining the object's behaviour. The other methods, the ones that are not overwritten in the definition, are inherited from the originating object.

In the example we didn't define a new state for the new object *TwoWayCounter*, but we can. We can even define a new type for the state. In that case we must most likely define new implementations for the methods, but it may still be interesting to use the adapt feature to document the fact that we are re-using the ideas present in the first object.

Finally, note that despite the fact that we adapt the interface *TwoWayCounter* from *Counter*, this doesn't necessarily mean that *TwoWayCounter* objects *have* to be adapted from *Counter* objects. One may still create *TwoWayCounter* objects from scratch, but we *can* use the adapting mechanism.

Self contained objects. Quite contrary to many prototype based schemes as Self or NewtonScript, adapted Visto objects do not form a chain of objects. It is more like Kevo [83] where objects are logically stand-alone and typically have no shared properties with each other. The same is true for adapted Visto objects. The methods present in the first object are copied into the new object. This new object then contains *all* the method definitions.

When a method from an adapted object is then called, we only have to look in the object itself. In other systems this is more complicated. In Self we have to follow a chain of parent objects; in class based systems, this becomes a chain of classes following the class inheritance hierarchy, not mentioning the problems faced when dealing with multiple inheritance, a feature that is not present in Visto.

In NewtonScript we even have to alternate between the *_proto* and *_parent* relationship.

An analogue problem is encountered for the data members. If we take a typical example, a 3D point that is composed by adding a z-coordinate to a 2D point, we often have a first object – a 2D point – with slots *x* and *y*, and a second object with a slot *z* and a pointer to a 2D point. If a method in the second object needs the *y* slot, it is first looked up in the second object and then, following the upward pointer, in the first object.

This is not the case in Visto. The object's state is only contained within one object and can simply be accessed by the Visto keyword **state**. This is a solution that is easy to understand and easy to implement, but that also offers some disadvantages which we discuss in more detail in section 3.11 on design alternatives.

Open recursion/Dynamic binding. Another interesting feature of object orientation and its inheritance relationships is the aspect of open recursion. Suppose that in a first object two methods are defined, *methA* and *methB*, that are mutually recursive. If we derive a new object from the first, and also provide a new implementation for *methB*, will the call of *methB* in *methA* refer to the new method or the old one? In open recursion we use the new method. It is called *open* because the old recursion scheme can be reopened to include fresh implementations. This feature is often called *dynamic binding*.

In Visto the programmer has control over this feature himself. Remember that calling another method is handled by sending a message to an object. Sending a message within a single object is done in the same way. We still have to specify which object we want to send the message to. So, if we are within an object *objectA*, we can call *objectA*!*methA*. If the programmer always wants to use *objectA*, he uses this method call. He then chooses for closed recursion in the sense that the recursion is now fixed to proceed in *objectA*. This is also called *static binding*.

If he intends the recursion to be open, he should somehow be able to indicate the fact that he wants to call a method from the current object itself. Just as one of our inspirators, Self, we use the generic name *self* to refer to the current object. A message *self*!*methB* will then implement open recursion.

Downsizing Objects

So far we have only been able to *add* behaviour to objects. That is quite typical in object oriented systems. In the first place they are concerned with *incremental* designs: how can we add new functionality and more powerful objects and classes to existing designs?

That is indeed an important question, but just as in real life – which object orientation tries to adequately model – this is not always the best way to go. Many large companies already went bust because they were too much involved with up-scaling, and e.g. Unilever is currently largely reducing its number of consumer brands. And it is well known that two can do a lot more than one, but also that if you get too big, one spends more time in meetings than in productive moments.

Although the same human problems occur less easily in computer households, we found it worthwhile to invest some time in the downsizing of objects.

```
adaptInterface Counter3 from TwoWayCounter
    setAt  :: Int → Int
    _set
```

*Counter*3 shows an example of downsizing in Visto: by preceding the old method name with an underscore _, we state that we want to remove that particular method from the old interface. We can add methods to an interface, but we are also able of removing methods. In the example we remove the method to set the object state at a fixed value, but add one that sets it to an argument value.

It is a new and powerful enhancement, but it also means that we cannot use *adaptInterface* to implement subtyping, quite contrary to class based systems that do use inheritance to define subtyping. Our **adaptInterface** feature is only a means for reusing an existing interface. A Visto feature similar to subtyping is implemented by checking whether the needed methods occur in the referenced objects.

If we use **adapt** to create a new – possibly smaller – object, our implementation will automatically remove all the methods with a preceding underscore from the originating object. Unfortunately we haven't had the time yet to experiment extensively with this feature, but we believe it may be interesting to broaden this approach and implement more powerful constructs such as union, conjunction, ... as in a kind of Boolean algebra.

This may lead to a greater reuse of code, because typically classes (and thus objects) can only grow, where sometimes down-sizing may be more appropriate.

We also feel that this may introduce more flexible schemas. If we start from a given – and working – design and want to add a smaller class to the hierarchy, this may require re-implementing some of the inheritance relationships to fit this new class into the hierarchy. An objectB that used to inherit from objectA, may now be required to start inheriting from objectC, because objectC is smaller than objectB, but still larger than objectA (fig.3.6).

This would require changing existing code, which is something we'd rather avoid. Using *adaptInterface* with removal of methods we can just add the new object to the existing scheme without any code rework.

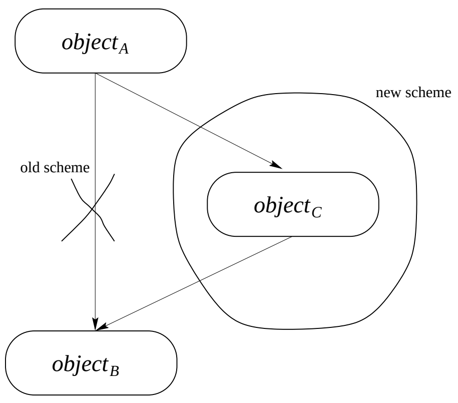


Figure 3.6: Changing the inheritance relationship to include a smaller class.

3.6 Object Derivation with Sharing

So far we have implemented the derivation mechanisms that are present in almost all prototype based languages. At the same time they can mimic the derivation mechanisms present in class based languages:

- *Instantiation*: we can use *clone* to create objects that behave identically but have different state values.
- *Inheritance*: *adaptInterface* and *adapt* can be used to implement incremental designs as present in inheritance schemes.

The big difference is that these previous derivations created autonomous objects that have no special relationship with each other, including the objects they originate from, except for the sharing of interfaces.

Even when two objects have identical methods Visto doesn't specify that the method must be shared. It is present in the structure for both objects. [7]

In this section we present a derivation mechanism that actually shares information. The data for an object will then be spread across different objects, but that is not really sharing: it is more spreading of information. This may be an interesting implementation feature, but it is irrelevant to the user of the system who only desires that he/she can access all the data in his objects, not whether it is concentrated in one place or distributed over many places.

[7]Although, more precisely, a pointer to the method is present in both objects. Of course, TkGofer, the actual implementation language can then share the method in the translated code.

Sharing is the feature where some information and/or behaviour is shared between objects, and where the programmer not only is aware of this sharing, but also exploits it.

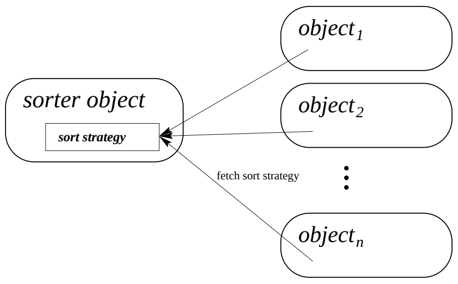


Figure 3.7: Sharing a sort strategy.

An example is a simple implementation of the strategy pattern [29] where an object contains a strategy for implementing some behaviour, e.g. sorting. A *sorter* object contains the sorting algorithm. Many different other objects can fetch the sorting routine from the same object. The sort strategy is *shared*.

To implement this behaviour we don't need extra language features. It can already be accomplished by simple message sending to a *sorter* object. Any time the strategy is needed, an appropriate message is sent. We prefer to call this sharing *passive*. The shared information is living passively in the shared object. If it is needed, it has to be retrieved. There is no active mechanism that automatically forwards any changes in the shared object to users of the object.

An active mechanism can be handy in the well known model-view relationship. We then have, on the one hand a model, e.g. the representation of a time, and on the other hand a number of views, e.g. an analogue and a digital clock. Of course we do not want to duplicate the time in both the analogue and the digital view and therefore only keep one instance of the model. We let the different views share the model. Active sharing mechanisms will inform the views when the value of the model changes like in the *observer* pattern[29], contrary to passive sharing that lets the views take care of this themselves. In such a situation the views must continuously poll the model.

Visto provides the *expand* derivation to implement such *active* sharing as well as distributed state. These features are the subject of this section. We first discuss how we handle distributed state – the easier part –, and then, in the section on message delegation and triggering, explain our approach of active sharing.

First, remember that the *expand* derivation is for a large part like *clone* and especially like *adapt*. We first take an existing interface and mold it to our needs:

we can both add and remove methods.

> **expandInterface** *AccessCounter* **from** *Counter*
> *count* :: (Int, Int)

An expanded object then *must* provide implementations for the new methods, and *may* do so for existing methods. These new or overridden methods can be defined like previous methods, but may also use some of the new features discussed in the following sections.

3.6.1 Distributed State

With simple message sending – the alternative for a more passive form of sharing – we can only access the methods of another object, not it's state.

Because expansion in our view is a very strong binding of the two objects, we also let the derived – expanded – object access the state of the original object. That is the easier part of the derivation. Visto provides the keyword **superState** for that purpose. We hereby follow the rule of most object oriented languages to refer with *super* to the *super*object, i.e. the object higher in the hierarchy, here the object that we expand from – the model.

The actual state of an expanded object is thus distributed among different objects. This is completely different from the previous *clone* and *adapt* derivations that created objects with a centralised state. It implies that the state of an *expand* object may change even when the *expand* object does not receive a message. This happens when the *super*object on its own receives a message and updates its state.

Just like **state**, **superState** is referentially transparent within one method evaluation and refers to the state of the superobject at the start of the method call.

3.6.2 Message Delegation and Triggering

A bit earlier on we called the redefinition of a method simply *overriding*. In fact it is more powerful. It is more than replacing the old definition with a new one.

As we will discuss in this section, the new implementation can still use the old implementation. They can both co-exist. Therefore the new method not simply replaces the existing implementation, but *expands* it. This explains why we call this derivation **expand**.

It should be clear that this situation is a lot more sophisticated. A small example will show when this technique may be needed: consider a two dimensional (2D) point and two possible expansions: a coloured 2D point that adds a colour, and a 3D point that adds a third dimension.

A *move* of a coloured point will simply execute the *move* of the 2D point. For a 3D point this is a bit more complicated: it has to move the 2D point and move along the third axis as well.

So an object that receives a message forwards that same message to its *super*object. This technique is called *message delegation*[8] : passing (delegating) the message from the derived to the original object, from the view to the model.

In some situations the message may also be passed from the original object to the derived objects. This situation will occur when the derived object has state components that need to be kept consistent with the state of the superobject. So whenever e.g. the method *up* is called within the superobject, the derived objects may wish to call their *up* method too. The superobject, the model *triggers* its views (fig. 3.8).

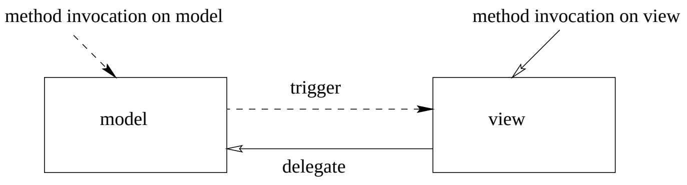


Figure 3.8: Simple Delegating and Triggering.

Especially in the decorator design pattern [29] it would have to be transparent to the other objects whether an object is decorated or not. So another object may choose to send a message to the undecorated (the *model*) or the decorated object (the *view*). If it is sent to the undecorated object, the decorator may need to be called as well.

It is clearly undesirable to let the model take care of the triggering because it doesn't need to be aware of its views. That would just clutter up the code, which really needs to remain identical. So we will let the expanded object decide whether it wants to be triggered.

However this combination of the superobject triggering its views and the views delegating to the superobject is potentially dangerous. When the view calls ('delegates to') the model and that model in return triggers all its views, which may again call the model and so on, we easily end up in a never ending loop: from $view_2$ to *model*, from *model* to $view_1$ and $view_2$, back to *model* and so on ... (fig. 3.9).

Therefore Visto provides the keyword **superValue** to take care of the message delegation. At the same time it avoids the endless loop. **superValue** is strict lazy [sic]. Strict for being always evaluated, lazy for being evaluated only once, even if different views require the superValue.

[8]Take a look at Garnet [52] for another discussion of message delegation.

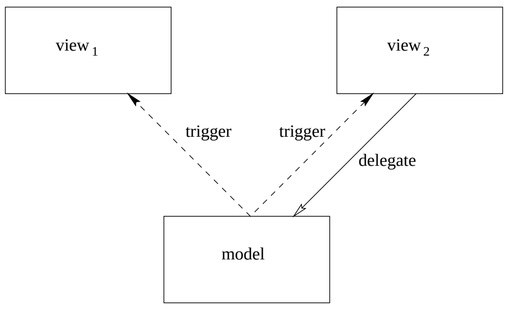


Figure 3.9: Multiple Delegating and Triggering.

Let's now take a look at an example that keeps a count of the number of times the methods *up* and *set* of some counter object are accessed.

```
expandInterface AccessCounter from Counter
    count :: (Int,Int)

expand AccessPlusOne AccessCounter
         (Int,Int) (0,0) from PlusOne {
    up =+ let
                returnThis = superValue
                (ups,sets) = state
                ups' = ups + 1
            in
            (returnThis, (ups',sets)),
    set =+ let
                returnThis = superValue
                (ups,sets) = state
                ups' = ups + 1
            in
            (returnThis, (ups',sets)),
    count = state }
```

This demonstrates that *AccessPlusOne* behaves just like *PlusOne* except that the access count can be read. The actual counter remains in *PlusOne*, so *AccessPlusOne* delegates its messages to *PlusOne* using **superValue**. Because we want the *up* and *set* methods to be activated in *AccessPlusOne* whenever they are called in *PlusOne*, the triggering mechanism must be activated. Therefore we use $=+$ instead of $=$ in the method definition.

This example also shows a derived object with a state type different from the original object, in this case *(Int,Int)* with initial value *(0,0)*.

It should be noted that clone and adapt are transitive actions for expand, that is, clones or adaptations of an expansion are expansions too. Therefore the use of the three new concepts in this section, the $=+$ binding and the **superState** and **superValue** keywords are also allowed within **clone** and **adapt**.

3.6.3 Another Expand Feature

The previous section described the technique of method delegation and method triggering. It allows methods to use their direct ascendants and/or descendants and is therefore a powerful technique, but it can also be considered limited because only the ascendants and descendants can be triggered. When delegating, a method X always delegates method X of its superobject; when triggering, a method X always triggers method X in all its expansions. It cannot trigger to method Y, nor can a method X delegate to a method Y. Such a language may be more powerful but also unintelligible. It would be very hard to understand and maintain applications built that way, trying to find your way in the labyrinth of unordered triggers.

Nevertheless, we did implement another technique that triggers the expanded objects whenever the superobject's state is updated, irrespective of which method was invoked. Because it is specifically aimed at user interfaces and the display of state, we will discuss it in the chapter on user interfaces in section 4.8.

3.7 Run Time Behaviour

3.7.1 Creation

So far we have discussed an interesting object schema where objects can be defined both loosely as well as very closely connected. This may already seem like quite an advanced object system, but when no objects can be created at run time, we are stuck with a static system. This may be appropriate for small or fixed applications, but not in general.

All the previous object derivations were aimed at compile time. We can use these syntaxes to create the starting set of objects, which as an aggregate constitute the system's state. This state can change as the objects change their state, but it remains fairly limited. Only when we can dynamically add new objects to a running system, we can implement really flexible programs.

Nevertheless flexibility is not a synonym of unlimited freedom. Typically the behaviour of all objects in the run time system is fixed at compile time. In class based systems this is true because the behaviour of each of the methods – and thus of each of the objects – is defined in the class. As only objects can be added at run time and not classes, the methods are fixed.

An analogue situation can be viewed in typed functional languages. Haskell provides a lot of features for user defined data types. This means that we can create new types, but just as in a class based language this can only occur at compile time, not at run time. Generic data types as the tree of *a*'s in the example below, provide instantiatable data types, but this still doesn't mean that we can create entirely new types at run time.

$$\begin{aligned}\textbf{data}\ (\mathit{Tree}\ a) &= \mathit{Leaf}\ a\\ &\mid \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\end{aligned}$$

Visto follows both conventions. Firstly, no new interfaces can be created as these directly correspond to a type in TkGofer (or Haskell for that matter). But also no new method definitions are allowed. The code for all the methods is fixed. At run time we only allow new states to be defined for the new objects. All the methods are inherited from the original object.

Naturally, because of Haskell's first class functions, any function can be part of an object's state and that way even the behaviour of the methods can change because of applying a different Haskell function that is contained in the object's state. This also offers the advantage that this possibly altering behaviour is clearly documented by the appearance of a function in the object's state.

This restriction is also imposed by implementation issues. So limiting the run time creation possibilities in this way is both based on other systems' limitations and analogies (cfr. classes and types fixed at compile time) as well as on Visto's limitations (because of the way in which objects and its methods are represented in and translated to TkGofer).

Visto essentially only allows *clones* to be created at run time. These clones may or may not define a new state value, but the *type* of that state must remain identical. One certainly cannot override method definitions. Visto introduces the keyword **addClone** with two syntaxes:

- **addClone** *newObjectName* **from** *oldObjectName*
  creates a new object with initially the same state as the old Object.
- **addClone** *newObject newStateValue* **from** *oldObject*
  creates a new object with a new state value.

With this syntax we can easily add as many objects as we like. This is an easy job for objects that are loosely coupled with others, but more difficult for tightly connected ones such as decorated objects, or model-view relationships. We would like to clone both the object and its decoration or the model and its view(s).

Starting from a single object we can easily construct an expansion hierarchy for all the objects depending on that object through the **expand** derivation (fig. 3.10). When we say we want to clone an object and its decorator(s) or a model and its view(s), we are actually referring to this expansion hierarchy. To ease the

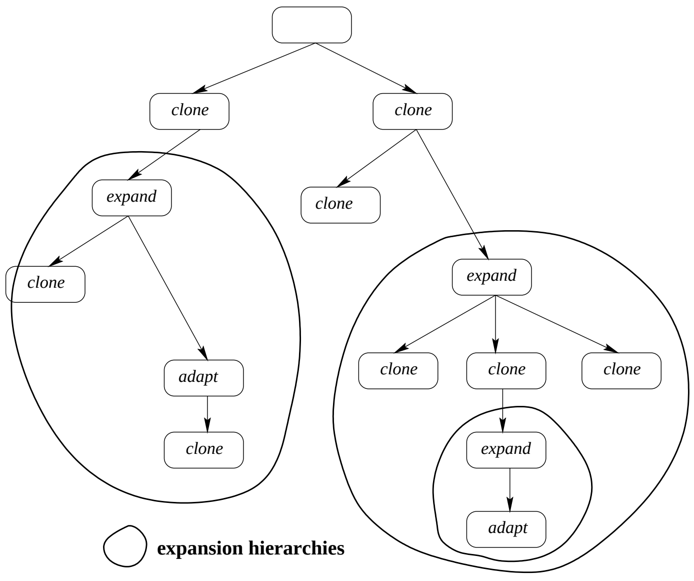


Figure 3.10: An Expansion Hierarchy.

cloning of such a hierarchy Visto provides a stronger alternative for *addClone*, **addClonePlus**.

The advantage is that the entire hierarchy is rebuilt, the disadvantage is that it is difficult to individually rename the depending objects or reset their states. Visto defaults to adding a suffix to the names and copying the state values. Only the ‘mother’ object’s state can be reset:

- **addClonePlus** *suffix* **from** *oldObjectName*
  creates a new object and depending expansion hierarchy. All the objects initially have the same state as the original objects. Their names are suffixed with *suffix*.

- **addClonePlus** *suffix newStateValue* **from** *oldObject*
  creates a new object and depending expansion hierarchy. The ‘mother’ object gets a new state consisting of *newStateValue* and all depending objects initially have the same state as the original objects. Their names are suffixed with *suffix*.

Run time object creation can appear where message sending commands can appear, that is in the top let binding of the method definition. From that point on the newly created objects can be referenced.

3.7.2 Removal

In functional programs data structures are usually garbage collected whenever they are no longer referenced. In our implementation of Visto we don't keep an eye on the references to objects. We prefer to keep all objects always alive.

The main reason lies in the implementation, but it is also a fact that it can be hard to tell when a Visto object is no longer referenced. There are two reasons for this: firstly Visto objects are *visualising* objects. They can be used to visualise information. [9] So although no explicit references may exist in the program, it may still be useful to display the information from the *visual* Visto object. We consider the user's eye as a permanent reference.

Secondly, Visto objects are also meant to describe the user interface in a declarative way [4]. The program itself may no longer reference the object, but the user interface may still be pointing to it, e.g. by a button that executes a method of the object whenever the button is pressed. This is again an external reference and we simply cannot predict when the user will stop using the object.

We could have taken that into account in our implementation, but for simplicity's sake, we preferred to use a stream lined system: no objects are garbage collected. Technically we inhibit garbage collection by continuously keeping a list of all Visto objects.

The programmer can however choose to remove some objects explicitly. This can be useful if the object displays some information or contains parts of the user interface, and the programmer wants to finish that part of the application and remove the interface defined in that object. We therefore supply a command, **killObject**, that removes an object from the system.

When that command is issued, we always remove the object even if other objects still reference it. We consider it the programmer's responsibility to check whether he can safely remove the object. He may do so, even with living references when he knows that the methods with the references will be no longer used.

Just as we did when creating objects at run time we have to take care when removing objects at run time. It would be rather dangerous to remove the model and keep all the views on that model, or to remove an object that is decorated while retaining its decoration, e.g. remove the document, but not its scroll bar.

Contrary to the creation where we had two alternatives, one that cloned just one object and the other that cloned an entire hierarchy, the **killObject** command will always kill an entire expansion hierarchy. It takes as an argument the name of an object and will remove that object and all its dependents.

In the actual implementation the object's visual aspects, the drawings and the interface elements will be explicitly removed by Tk commands, whereas the object's data structure will be dereferenced and actual garbage collecting deferred to the TkGofer system.

[9]They can have a **draw** method that is automatically executed whenever necessary. More about this feature in the following chapter.

3.8 Geared Towards UI Programming

Despite the fact that this chapter is devoted almost exclusively to the object oriented aspects of Visto and the fact that we only compare our project with other object oriented projects, Visto is actually geared towards UI programming. We wll return to this in the next chapter where we show how to add a user interface to a Visto program, but we can already prove some points.

However, before we start this discussion, we want to stress the fact that Visto is not about structing the widgets in an object oriented way. The objects in Visto are user oriented objects, i.e. the objects that users is working with such as the web document he is viewing or designing, the worksheet, ...

Central issues when programming a UI in a user oriented way, are features that allow the programmer to derive new objects in the same manner as the users of his application will do. The program can then more easily reflect the intentions of the user of the program.

Being able to define methods for each object, methods that the user will eventually trigger in the UI, is the most trivial aspect. It is no more than normal that Visto supports this; as a matter of fact, just as any object oriented language.

Beside that aspect of taking care that the user can manipulate existing objects, another user behaviour is that of opening a new window, either an entirely new, empty window, e.g. in a word processor, or a copy of the currently viewed window as e.g. in Internet Explorer. This corresponds to creating a new Visto object, either by cloning when no new actions are possible, or by adapting when new actions have become possible such as opening a web page in the designer part of the program. Visto can easily accommodate such actions.

Another thing often performed is opening a new view on an existing object. That is something that can be easily accomplished by the expand derivation as there a link to the original object is maintained. This expansion and model-view relationship is more a technicality that the user is not directly aware of, but it is of great importance to the programmer of the UI.

These three principle aspects of user behaviour are strongly supported by Visto, which proves our claim that Visto is geared towards UI development.

A fourth and final aspect is the separation of concerns in the original object and its expansion. We have seen that these two objects have a separate state. In a GUI context this is very useful as the model (the original object) can then contain the model related data, and the view (the expansion) the GUI related data, e.g. the font size. The view has access to the model data (to be able to depict it) as well as to its own data.

3.9 Restrictions

Although many object oriented features are present in Visto, we cannot claim that we have taken care of everything. Some features present in most other object languages are missing, mostly for simplicity's sake. This section discusses the two most important restrictions that are still present in Visto.

3.9.1 Visibility Modifiers

Visto adheres to a very simple philosophy: every method of every object is visible to every object. This is easy to explain and understand, but also a rather weak feature. Often, local methods are defined: they are useful as an auxiliary tool for methods *within* the object, but not for *other objects*.

Visibility modifiers tackle that problem. A *public* method is visible outside the object, a *private* method only within. Most often this basic system is extended with visibility modifiers that concern the inheritance hierarchy. After all, if an auxiliary method is needed in some object, it is probably needed in the derived objects as well. Declaring the method private is too strict – it remains invisible to derived objects –, declaring it public is too open, as it becomes visible to *all* objects, not only to derived objects.

Modifiers such as *protected* in Java fill this gap, making the protected method visible to the objects of the class itself and to all objects that inherit from the former class, while keeping it hidden to the objects of other classes.

Visto lacks such visibility modifiers. Although it is an important feature in professional software design, it does not change the set of programs that can be written in the language. So instead of spending our precious time on such well-known aspects, we preferred to explore new features. In the end, if Visto and its ideas prove to be successful, the visibility modifiers can be easily added to reach a more mature system.

Moreover, this is a less important issue in Visto than in *pure* object languages. In such a language methods are the only way to define 'procedures'. So if we want local procedures, we need a means to specify local – hidden – methods. The private modifier does precisely that. Visto defines an object system (that captures the GUI) *on top* of a Haskell program. So instead of using auxiliary *methods*, we can simply use auxiliary *functions* from our Haskell program.

However, this is still only *Ersatz*, an inferior work-around. Visibility modifiers are used for more than defining local methods. They are also useful in the context of inheritance and object derivation, and the solution of using auxiliary functions instead of auxiliary methods doesn't work here.

Nevertheless, it is only the current implementation that doesn't support visibility modifiers. Adding them to the definition of Visto doesn't pose any problems.

The intended visibility can already be documented. It is then only the programmer's responsibility to adhere to that documented visibility, without the support of the compiler checking and enforcing it.

3.9.2 First Class Objects

We feel that this is in some way Visto's strongest restriction: the fact that objects are not first class. This can be divided into three cases:

- variables cannot point to objects
- methods cannot return objects
- methods cannot take objects as an argument

Variables Cannot Point to Objects

With variable we mean in the first place the state variable of an object. Other variables in the definition of an object are either return values or arguments of a methods, which is covered by the two other cases.

The variables outside an object also cannot point to objects as we don't reveal the type of the object. This may however change in the future.

The fact that the state variable of an object cannot contain other objects means that no local objects are allowed. This restriction is due again to our attempts for making the initial system as simple to implement as possible. We considered the lack of local objects no vital loss as it is easy to have de-facto local objects. To mimic an object Y as being local to an object X, it suffices to make sure that only object X uses object Y. The disadvantage of course is that the programming system does not support nor enforce this locality. It is completely the programmer's responsibility. But, again, this notion of locality doesn't really influence the set of programs that can be written in Visto. Moreover, Visto is Turing-complete because Haskell is Turing-complete and any Haskell program can be easily turned into a Visto program.

More of a problem is that the contents of a real object variable can change more drastically than a local object as simulated above can. If we refer in the code to a pseudo-local object X, only the object's state can change. If we have a variable, it can change its reference to a completely different object. So we can start using an object X and end up using an object Y, without having to resort to changing the object X. This is clearly a lot more flexible than in the scenario with a simulated local object.

It is therefore one of our higher priorities after writing up this thesis to enhance Visto such that the object's state can contain other objects. We were reluctant to add this feature because we feared we would have to give away information about how a Visto object is represented in the translated code – and because it was rather cumbersome work in the original implementation.

This fear was inspired by the fact that in the beginning we thought of *interface*s as temporary (see section 3.4.1). Without them, it is hard to express the type of a prototypical object. In a class based system, this is simply the class, but not so in prototype based system. Nevertheless, now that the *interface* is really considered a feature, we can use that “prototypical” type. This makes us confident that it is feasible to add this feature without major reworks to the code and without having to reveal the translated type (or part of it), although the use of existential types may be necessary.

Methods Cannot Return Objects

When a method returns an object, a first option is to store the object in a variable within the Visto object. Because currently Visto objects cannot contain other objects, this is no valid choice.

The second option is to use the return value immediately to send a message to. However, in the current implementation we must know the precise name of the object to be able to find the type of the object. So using the second option is incompatible with the implementation. As we have not yet found a solution for that problem, we haven’t included this feature in the Visto definition.

This does not imply that methods cannot *create* new objects. They can, but until a place exists to store the newly created objects (i.e. in another object’s state), it can only be performed as a side effect using *addClone* as we discussed in section 3.7.1.

Methods Cannot Take Objects As an Argument

It is useful for a method to take an object as an argument because it can vary in that way the object(s) to which it sends messages. However, because we use type classes in the translated code, we must know the precise type of the object (including the type of its state variable) to be able to resolve the otherwise ‘unresolved ambiguity error.’

Because in the current implementation we precisely know which object we’re sending a message to, we also know the precise type of its interface and of its state type. We can then resolve the disambiguity and are not faced with this problem.

We expect that existential types may help in finally solving the problem. However, just as with the problem from the previous paragraph, we will not include this feature in the Visto standard until a viable implementation has been found.

3.10 Implementation Issues

In this section we discuss some of the important decisions in the implementation of Visto, such as the way in which Visto objects are represented in TkGofer, messages sent and results received, how Visto ensures type safety and very briefly some efficiency considerations. These are the most influential choices for the implementation of the object system, whereas the rest of the translation is either not so interesting or merely straightforward. Once entire Visto has been presented, we present in section 6.2 the scheme by which we translate Visto programs to TkGofer.

Although at first sight the most important aspect is the representation of Visto objects, we first deal with the way in which methods are represented, as it strongly influenced the object representation.

3.10.1 Method Representation

The syntax for sending a message is quite easy: just mention the name of the object to communicate with, the message to send (method to invoke) and the arguments for the method. The return value is then assigned to the left hand side of the let binding. Despite this apparent simplicity a lot is going on in the background. The object has to be fetched from memory and its method executed. But executing that method may not only change that particular object's state, it may in turn invoke other objects' methods, which may change their state and call yet other methods.

We cannot get *just* the object and pass the method its arguments, we also have to take into account the other objects, both as input to the method and as return value. Although we were hiding that in the Visto message sending, we can't do this any more in the translated functional TkGofer code.

Instead of defining an ad hoc function for every single method, we use a general dispatching function. Each Visto method will be translated to a standard TkGofer function and the general dispatching function will consume such a translated method. The consequence is that this function is to work on a large variety of very different methods, both in type and behaviour.

The first option, using a Haskell type class to accommodate for the different types, doesn't work. Haskell type classes are only appropriate for methods that differ in input arguments, not for different types of return value. Although it may be compatible with the Hindley-Milner type system [47], the dictionary based translation for type class functions is unable to cope with such a feature. Moreover, in Haskell it is also considered an undesirable feature.

Our solution is a *Univ* data type that combines all the return values of all the methods. It introduces some extra indirections though. Firstly, the result of a method invocation has to be packed in the *Univ* data type before it can be returned. Secondly, it has to be unpacked again when we want to use the actual return value in the method that sent the message. It is however the price we are prepared to pay for being able to use a global dispatching function, instead of many ad hoc functions with lots of code duplication.

Because Visto objects can also have both drawing features and an attached user interface (see the following chapter), we have to take extra monadic side effects into account. The type for such interactions in TkGofer is *GUI* (). A translated method will therefore also return a value of type *GUI*().

To complete the story of a method's type: it needs as arguments *all* the objects, and its own arguments. Because the method will certainly need access to its own object, especially for the state and possibly for other auxiliary methods, we conveniently add the object itself as an argument to each of its methods. Although it is also available in the list of all the objects, we find it handier to have it readily available.

As return value, packed in a 3-tuple, we have

- the actual return value (packed into *Univ*).
- *all* the objects
- side effects, in *GUI* ()

We make sure that we first supply the method with its *explicit* arguments [10]. This gives us a common type after the explicit arguments for the method:

$$
\begin{aligned}
&(arg_1 \rightarrow arg_2 \rightarrow \ldots \rightarrow arg_n \rightarrow) \\
&\qquad < \textit{the own object} > \rightarrow [Objects] \rightarrow (GUI\ (),\ Univ,\ [Objects])
\end{aligned}
$$

The actual type of the function is different from this type because of features needed for expanded objects. That discussion is postponed until section 3.10.3. Meanwhile we abbreviate the translated method's core type as *MethodType.*

$$arg_1 \rightarrow arg_2 \rightarrow \ldots \rightarrow arg_n \rightarrow Methodtype$$

3.10.2 Object Representation

Historically, we first settled on an approach for handling the method calls. As discussed in the previous section, one of the key features was the common core type for translated methods. In the same manner we want a uniform approach for objects, both for individual objects and for the aggregate object list.

We first discuss the representation of the Visto *interface*. It is an important feature of our object language. This remains valid in the translated code and therefore we have a direct one-to-one mapping between a Visto interface and a corresponding TkGofer data type. Objects state which interface they implement and so they will be translated to the corresponding data type. In this object type we will use the type of the methods as discussed in the previous section.

[10] We call the arguments that were mentioned in the object interface *explicit*. The *implicit* arguments are the list of all objects and the object itself.

The Visto interface

Because Visto is a prototype object system, it makes little sense to implement a class based approach where an object keeps a state and a pointer to its class. We could have as many ‘classes’ as objects.

So we prefer to keep all the object information in the object itself, including all the object’s methods, and not in some general class-like structure. This makes translated Visto objects self-contained.

We do not define the methods globally and let the objects just keep pointers to those methods. The methods are really part of the object and fill up one field in the record of the object. When cloning, we only have to either keep that record field when the method is inherited, or overwrite it when the method is overwritten. Remember that these method functions all have an identical type *MethodType* except for some possible method arguments.

It should be clear that this data type furthermore minimally contains the object’s name and state. Slightly less obvious is the fact that we also have fields for the name of the object this object may be expanding (i.e. the *model*) and for the names of the objects that expand this object themselves (i.e. the *views*). There is no such thing as multiple expanding (an object can only expand one object), so we keep a *Maybe String* for the first field and a [*String*] for the second as of course different objects may be expanding the same object: we can have many views on a model, but each view only views one model.

There are also some fields for the GUI. We don’t discuss them here as the GUI hasn’t been presented yet. This is postponed until section 6.2.1.

The type for the translation of a Visto interface is now almost fixed. The only variable remaining is the type of the object state. We purposely do not mention that type in the interface because it allows a broader reuse of the interface. How the object represents its internal state is irrelevant to the interface. It should not restrict the interface’s application. However, because it exists, it must be present in the translated type.

Fortunately, this is easy to express in Haskell. We can define parametric data types, e.g. *Tree a* is a tree of values of type *a*. The TkGofer data type for an interface is parametric in the type of the object’s state. This way both the original Visto interface and the translated data type can be used for objects with different state types.

An interface *Counter* is translated to a data type *Counter a* and thus can be easily instantiated to e.g. *Counter Int* and *Counter* (Int,Int), depending on the state type of a concrete *Counter* object.

```
data  < InterfaceName >  a
    = < InterfaceName >
              String a                — name of object, and type variable
```

$(Maybe\ String)$	— name of superObject, if present
$[String]$	— objects that expand this object
...	— A number of GUI related fields
$< MethodType >$	— Such a field per method
...	

The Aggregate Object List

The problem is that we have to form a list of all the objects. We might use existential types but that would imply that all the objects have to share the same constraints. Because of this and because existential types do not exist in TkGofer, we create a universal object data type.

Because the translated program uses this list throughout and we don't want unresolved type variables in the *main* function of the translated program, we cannot have a parametrised data type for each translated Visto interface. Instead we have to collect all the different state types for all the different objects. The resulting aggregate data type should look like the one on top and not like the lower one.

$$\begin{aligned} \mathit{data\ Objects} &= O3\ (\mathit{Counter\ Int}) \\ &\mid\ O2\ (\mathit{Counter}\ (\mathit{Int}, \mathit{Int})) \\ &\mid\ O1\ (\mathit{AccessCounter\ String}) \end{aligned}$$

$$\begin{aligned} \mathit{data\ Objects}\ \alpha\ \beta &= O2\ (\mathit{Counter}\ \alpha) \\ &\mid\ O1\ (\mathit{AccessCounter}\ \beta) \end{aligned}$$

We can then use [*Objects*] to create a list of all existing Visto objects. To accommodate for the state changes in the various objects we create a mutable variable capturing the list of objects. This mutable variable will then be continuously updated by the dispatching function that takes care of the method invocations.

This implies that Visto applies *whole program compilation*. All the objects must be present before our implementation can compile the program. Although this restricts modularity a lot, we haven't paid a lot of attention to that facet of modularity. We are however confident that it can be easily solved, o.a. by compiling each of the objects in a separate module, and by only adding the mutable list of all objects and the dispatching function when linking all the objects.

3.10.3 Delegating and Triggering

This section only applies to expanded objects or clones and adaptions of expanded objects.

We have already discussed the keyword **superValue** to delegate a message to the 'super' object. Although *superValue* may be used in different objects which are accessed because of the triggering, we want the *superValue* to be calculated exactly once. Therefore we cache the result in a value of type *Maybe Univ* and pass it as an extra argument to each method. Initially the value is *Nothing*, but as soon as the **superValue** keyword is used, it is calculated, instantiated to *Just Univ* and passed as argument to the triggered or delegated methods. Whenever the *superValue* is non-*Nothing*, it is simply reused instead of recalculated.

Although in Visto the objects that are expanded don't need to know whether they are expanded and by whom, the translated objects in TkGofer should rather know, so that we can easily retrieve the names of the expanding objects, fetch the objects themselves and trigger the methods.

This situation is still a simplified one as we keep the names of all objects who have at least *one* method that needs triggering. It doesn't necessarily mean that *all* methods should be triggered because triggering is defined on the level of methods and not globally on the level of the objects. So besides fetching the expanding objects, we still have to check whether the actual method needs triggering. We add a boolean value to the method function that indicates whether the method should be triggered or not.

The final type of the functions for translated methods is

$$\begin{aligned}(\mathbf{Bool}, \mathit{arg}_1 \;\to\; \mathit{arg}_2 \;\to\; \ldots\; \mathit{arg}_n \;\to < \textit{the own object} >\\ \to\; [\mathit{Objects}] \to\; (\mathbf{Maybe\ Univ})\\ \to\; (\mathit{GUI}\;(),\; \mathit{Univ},\; [\mathit{Objects}]))\end{aligned}$$

3.10.4 Type Safety

We strongly rely on the TkGofer type checker. Simple checks such as the names and the number of methods an object has to provide, and the number of arguments needed by a method, are performed during the Visto translation to TkGofer, but real type checking is deferred to TkGofer. This has the disadvantage that many error messages are postponed until the second compilation phase and that it may be difficult to relate these messages to the original code, but it has the strong advantage that *we* can simply rely on Gofer's well established type checking algorithm.

Checking e.g. that the type of the methods matches the interface definitions is done when loading the translated code into TkGofer.

One of the restrictions of Gofer's type checker is the possible disambiguity when using type classes. We use type classes a lot to implement the packing and unpacking of method return values in the *Univ* type. It turned out that the type checker could not disambiguate the intended type. This disambiguity is resolved by adding the type of the return value to the translated code when values are unpacked from the *Univ* data type. This information can be easily fetched from the interfaces, and results in a stronger type checking, ensuring that the correct type for method return values is checked at compile time.

Still, run time errors can occur when sending a message to an object that has not been created yet or already removed, but this can hardly be avoided in systems with explicit removal operators.

Anyhow, although Visto provides little type checking itself, we have experienced that the combination of Visto's minimal checking and TkGofer's type checker provides adequate type safety, despite the fact that it can be non-trivial to relate TkGofer error messages to the originating Visto program.

3.10.5 Effi ciency

Efficiency hasn't really been an issue during the development of Visto. Although we are well aware that there is a price penalty to pay for the packing and unpacking of method return values and objects, we didn't worry too much because we aim at user interfaces, in which the user typically responds magnitudes slower than the system. We cannot offer hard figures about this claim and even didn't start looking at the time and space complexity of our implementation, but the (small) test cases we ran showed seemingly satisfactory response times. And quoting from [46] *"Indeed many design patterns suffer similar overheads, but their popularity suggests that users are indeed more interested in design and extensibility considerations than in fine-grained efficiency."*

Various options may still increase performance. Amongst others we currently keep a flat list of objects which may introduce a long object seek time when having lots of objects. Using both a hash table and an expansion hierarchy of objects may reduce this seek time.

3.11 Design Alternatives

3.11.1 A Single State Variable

We currently have an atomic state variable, i.e. the state is represented as *one* variable, that is referenced by *one* keyword **state**. This is unlike most other systems that have many data members or slots, that can be referenced and updated individually. The main reason why we did not prefer that approach is because we want atomic state updating. A method can only change its object's contents when it returns its return value, as already outlined in previous sections.

We also wanted this state update to have a functional behaviour. A new state is made by returning a new state value, not by some imperative non-atomic state updating.

Currently all methods return 2-tuples containing the return value and a new state. Our initial implementation even lexically checked on the existence of a 2-tuple. That lexical check has since then been removed because of its inflexibility, but methods still return 2-tuples.

If objects were to have many data members and we were to follow the same methodology, some methods would have to return 3-tuples, other 10-tuples. This would make Visto a lot less uniform. It is also difficult to force an ordering of the state parts in the return tuple. Should the tuple be $(\textit{return value}, x, y, z)$ or $(\textit{return value}, z, y, x)$? That is why we do not consider such an approach viable.

We could use mutable variables as implemented in monads to define state updating with multiple state variables in a sane functional way, but we don't want to do so because it imposes a monadic and thus imperative style on the entire method definition, as mixing monadic and standard functional style is often difficult.

Nevertheless we genuinely doubt if the choice for an atomic state variable was the best. It has its advantages, but from a re-use point of view also rather strong disadvantages.

Consider e.g. an interface for a 2D point. The interface of a 3D point is likely to be an adaption of that interface. Actually setting the x coordinate in a 2D point, can even be precisely like setting the x coordinate in a 3D point. Unfortunately, if we were to adapt an actual 3D point from a 2D point, we would have to define a new state variable containing (x,y,z) instead of (x,y). The method for setting the x coordinate would have to be completely rewritten instead of being able to inherit it from the 2D point.

On the other hand, as we pointed out in the section about the implementation of the Visto interface, a Visto interface is currently independent of the way in which a Visto objects represents its state.

It is easy to implement if we only have one state variable. We can then define a parametric data type with a single type variable. When we would allow any number of state variables, this solution would no longer work. It would certainly be no longer straightforward to relate objects with identical interfaces to each other. We would either have to resort to ad hoc methods, or once again use a aggregate data type with the extra indirections it causes.

The least we can say, unfortunately, is that changing Visto to accommodate multiple state variables is not simple. So perhaps we should rather focus on its other innovative features instead of overstressing this atomic state variable.

3.11.2 Pairs (Return Value, New State)

Instead of forcing *every* method to return both an actual return value and a new state, we may offer a choice between:

- A method can behave like a pure function, having only a return value and no state update.
- A method can *only* do a state update and behave like an imperative procedure, except for doing it (lexically) in a functional way.
- Methods can as well continue behaving like current Visto methods, both having a return value and performing a state update.

- And why not, methods can do none of both options: nor return a value or update the state.

Because every Haskell function can only return values of a single type, this requires a new data type:

$$
\begin{aligned}
\mathit{data\ MethodReturn\ a\ b} \;=\; & \mathit{Return\ a} \\
\mid\; & \mathit{NewState\ b} \\
\mid\; & \mathit{ReturnAndState\ a\ b} \\
\mid\; & \mathit{Void}
\end{aligned}
$$

For a single method this makes it clearer what it actually does. Currently many methods do not actually return a new state value, they just copy the old value over. No longer insisting on pairs (*returnValue*, *newState*) would certainly increase the performance of Visto because we no longer have to replace an object with an identical object.[11] It also expresses precisely that the method is only a data retriever.

But it makes Visto less uniform. Different kinds of methods would exist, some that just return a value, others that only update a state, and others that would do both or even none. This would not only introduce the dreaded *Void*, but also introduce a far more complicated semantics with a four valued method classification.

3.11.3 Prototype or Class Based

An obvious design alternative is class based OO. It offers the advantage that it is more easy to express that a range of objects has identical behaviour. Our *clone* and *adapt* derivations can be easily applied in a class based system as well, but we see no reasons why it can be *better* expressed in such a system. For *clone* and *adapt*, we find each object system as powerful as the other.

The *expand* derivation on the other hand is a lot more difficult to accommodate in a class based system. Method delegation is trivial to accomplish, but method triggering not. We haven't yet found a class based system that implements a feature like our *expand*. Even if it were, we believe that it is best expressed in a prototype object system, because *expand* is such a specialised relationship that it can be hardly applied and expressed in the general context of a class definition.

Listeners as present in Java come the closest, but they enforce both objects to be aware of each other, where our expansion is such that the model is not aware of the views that are expanding it.

We feel that the flexibility of prototype based systems makes it a more natural place to experiment with such features.

[11]Although this may already take place because of compiler optimisations.

3.12 Comparison with other work

We can hardly be expected to make a full comparison of Visto's object system with *all* object languages. It would be like one of Hercules' works. Instead we pick the most interesting ones with a special focus on other exemplary prototype object systems, design patterns and functional object systems.

3.12.1 Prototype Systems

This project has been strongly inspired by Self [81] and NewtonScript [77], especially for the prototype based object orientation. However, when we consider Visto's current state, it bears more resemblance with Kevo [83] and Garnet [52] than with Self and NewtonScript.

All these systems typically make no distinction between methods and (state) variables. They just call them slots. This enhances representation hiding, as the users of these systems cannot see the difference between a method and a variable. So what might seem like a variable could as well be a method and vice versa. We however have opted to fully hide our (single) state variable and let the programmer only access the methods.

Unlike Visto where objects are fully self contained, Self objects always have a pointer to a *parent* object. If a method cannot be found in the Self object itself, the parent pointer is followed and the search continues there. That is how Self implements inheritance.

NewtonScript has a similar feature with its *_parent* slot, but it also adds a *_proto* slot. Actually the _proto slot has a higher priority than the _parent slot, but the main point remains the same. The difference is that the authors of NewtonScript try to provide a finer scale, embedding a refinement relation (_proto) within a containership relation (_parent).

In some ways the Visto derivations *clone* and *adapt* can be seen as such a refinement resp. containership relation, but they should not be restricted to that interpretation. An adaption does not necessarily *contain* the original object. It more specifically *uses* another object, but it is free to leave out or modify some (essential) parts of the original object. Visto's derivations are therefore more flexible.

Both Visto and Kevo have self-contained objects. Kevo is an attempt to prove that delegation-free prototype-based languages are indeed feasible and practical and the authors feel they have succeeded. Visto avoids delegation in the *clone* and *adapt* derivations, but, especially in the context of the model-view concept, we felt the need to provide yet another derivation that *does* provide delegation, the *expand* feature. At the same time it introduces objects that are not self-contained, because expanded objects keep track of their superobject.

At first sight Visto's closest match might be Garnet in the sense that it, just like Visto, has been designed to describe GUI applications. Garnet uses a prototype based approach as well, and even an extremely flexible one. Any slot name is always valid. If a slot doesn't exist, it is simply created. This of course makes it completely impossible to type check, and thus inappropriate in the context of a strongly typed functional language.

Interesting, however is Garnet's use of *constraints*, which is a convenient way of specifying special relationships that should be continuously maintained. This set of constraints is automatically maintained at run-time by a constraint solver. The strength of this feature perhaps outbalances the weakness introduced by Garnet's lack of type checking. It has however little to do with how it handles prototype based object orientation and therefore we will postpone a deeper discussion of Garnet until we have presented our GUI framework in the next chapter.

For their approach to inheritance, the developers of Garnet follow the ideas of Self to have a parent chain that is searched for the matching slot when a slot is accessed. However, they fear the performance penalty of having to follow a long chain of parent links and therefore cache slots. The result of this is that Garnet becomes closer to our approach where we keep *all* slots, not just the cached ones, within the object.

Summing up, it seems that these prototype systems all agree on their preference for prototypes over classes but not so much on how to implement this, especially when inheritance is concerned.

For its simpler expansions Visto is like Kevo, for its *expand* more like Self and Garnet, while we also add a number of new features, such as the Visto interface and the triggering present in the *expand* derivation. And just like all the authors of these system, of whom especially the authors of NewtonScript expressed their satisfaction despite initial scepticism, we are pleased with the expressiveness and flexibility of a prototype object system.

3.12.2 Design Patterns and Interfaces

In the area of design patterns and interfaces, a large number of papers can be found. We particularly mention Mocciola and Martínez López [48] who promote the search for design patterns usable in and aimed at functional programming.

Although Visto has been developed completely separately from Rapide [44] both systems share many features such as the ability to derive a new interface from an existing one by removing methods. A consequence is that interface derivation does not include subtyping. Rapide uses *structural subtyping*: an interface is a subtype of another interface if all the objects of the first interface conform to the second. This is also valid in Visto.

Interesting is the comparison between our interfaces, Baumgartner and Russo who implement signatures [6] for C++, and the Java interface feature. The latter two systems show that the typical class based approach is too strict and that it can only cope with difficulty with more flexible derivations. Still a programmer doesn't have to use interfaces. In both systems they are an optional feature. While Java insists that a class declares that it implements an interface, the compiler in [6] infers the structural conformance. That is clearly an advantage, but because Visto doesn't use classes to structure the program, we use the structure of the interfaces and therefore force objects to declare which interface they implement. In the end this is no disadvantage, because each system insists that each object/class conforms to some structure. In Java this is both class and interface, in [6] just the class with the compiler inferring all possible interfaces. In Visto this is one interface with the other interfaces implicitly inferred.

It is furthermore interesting to see how the combination of Visto's object system and its functional core simplifies the implementation of two important and typical design patterns from the *Design Patterns* book [29], the *Decorator* and *Strategy* patterns.

The decorator pattern is used to transparently decorate an object with some extra features, often user interface elements such as scroll bars. It should be transparent such that the decorated object behaves just like the undecorated object, i.e. the interface is to remain identical. The other objects will refer to the decorated object as if it were undecorated. Visto's **expand** feature does just that. Although we create an extra object, accessing one of both objects, either the decoration itself or the object to be decorated, gives the same effect because of the delegation and triggering.

The strategy pattern uses objects which encapsulate (part of) an algorithm, such that it can easily vary. Because of Haskell's first class functions this can be done per object individually by keeping an algorithmic function in the object's state. This may be a viable solution, but is far from perfect if we wish to change the algorithm for a large number of objects at the same time. Instead of changing the algorithmic function in each object, it may be better to define an extra object, a *strategy* object, that keeps the algorithm function and returns it after sending a message to it. This is easy in Visto because any Haskell type including a (higher order) function can be a method's return value. Changing the algorithm for all the objects that use this strategy object then only requires changing the algorithmic function in the strategy object.

These two examples show some of the possible advantages of the Visto combination of object orientation and functional programming. Especially the aspect of higher order functions is fruitful.

3.12.3 Functional Object Systems

Peter Achten and Rinus Plasmeijer have implemented a very basic object system [2] with objects consisting of just an internal state and a state transition function. Like Visto it needs an interpretation function to keep single versions of each object in scope. The authors' main achievement is proving the standard polymorphic Milner/Mycroft type system [47, 49] without type classes not powerful enough to assign a type to the interpretation function. As Visto methods not only change state, but also have return values of different type, this explains why we have to use a universal data type for the return values.

Fudgets [16] actually introduced a nice GUI library to Haskell. The system can also be seen as a first attempt at object orientation as the GUI components have a hidden local state and can accept messages. Alas, because the communication proceeds through direct coupling of fudgets, this can hardly be used for general object oriented programming.

FOOPS for Functional Object-Oriented Programming System [66] with a focus on formal aspects, UFO (United Functions and Objects) [70], which is mainly interested in parallelism, and CLAIRE [18], mostly aimed at combinatorial problems, are more like traditional object oriented languages with addition of functional features as functions (though not higher order in FOOPS), typing and state transformation based on copying, and have little in common with Visto, except for CLAIRE defining object methods as overloaded functions, just as in Visto. CLAIRE relies on Tk/Tcl for its user interface as well.

From the two Gofer [42] related object systems, both from southern Germany, GOS [69] is yet again mostly OO with the simple addition of some functional features. Object-Gofer [72] on the other hand, and Haskell++ [38] bear more resemblance with Visto, all translating to functional languages, Gofer, Haskell and TkGofer respectively. Haskell++ relies strongly on the Haskell compiler to syntax check the translated code. Visto also relies on its implementation language, TkGofer, but only for type errors, particularly for method return values.

Object-Gofer does not use an interpretation function like Visto. Instead it relies on monads for state transformation and destructive updating. Haskell++ provides no special support, but allows methods to return a new object.

In Haskell++ an object is just a value of an ordinary Haskell type. Therefore any Haskell function can access the complete structure of the object, also the fields that it should not access. Object-Gofer and Visto only allow the object to be accessed by its methods, thus having real object types, with a separate, hidden state.

Both Object-Gofer and Haskell++ are class based as opposed to Visto's prototype nature. Object-Gofer is more consistent than Haskell++, allowing inheritance only between classes. Haskell++ defines inheritance on objects and as containment inheritance: the old, inherited object has to be part of the new object. Haskell++ is the only language of the three that allows multiple inheritance but in just the same way as type classes do.

Objective Caml [67] is much like Object-Gofer, but performs destructive updating through its reference variables and its imperative features.

Finally, in his research on a GUI for Curry[31], M. Hanus [32] describes an object system which behaves a bit like Haskell's old I/O-system with lazy lists as the list of all the messages an object will accept. There is few state hiding, but quite unique is the way in which results of method invocations are passed. This approach uses logical variables that will be instantiated when the object finishes processing the request. This is quite different from the way Visto passes the result in a typical functional way as a function returning a value.

3.13 Conclusions

Visto seems to be the first functional object system that is *prototype based*. The envisaged flexibility is achieved by the three different forms to derive new objects from existing ones, and by the fact that such derived objects can also remove methods from their ancestor, contrary to traditional schemes where only methods can be added.

The really unique feature is the **expand** derivation. The trigger on state change and method application is vital for user interface design, but can also be of great use in 'normal' systems because it allows different objects to share some information and to keep track of changes to it and because it allows for an easy implementation of some important patterns.

The implementation of such patterns not only benefits from the prototype based nature of our object system, but also from functional features, such as the higher order functions and the use of pattern matching in the definition of auxiliary functions.

We are already quite satisfied with the broad range of object oriented features that Visto offers, although, apart from interfaces and the *expand* derivation, there are few features that make Visto fundamentally better than other prototype systems. In the very least, continuing research should focus on first class objects. Also modularity, which we didn't pay a lot of attention to, should be improved.

The choice for the single state variable and the way in which that state is updated, was established with an easy implementation in mind, but may not have been an optimal choice. In the same way, letting TkGofer type check the translated code may have been easy to implement, but certainly doesn't make the compilation errors easier to understand, as the relation with the original source code becomes obfuscated.

However, as we have been able to provide a working implementation we have proven the viability of a functional, prototype based object system. Future research may build on this experience to provide new systems with all the whistles and bells present in modern object languages.

Chapter 4

A More Declarative GUI

4.1 Overview

In this chapter we present the GUI features of Visto and describe how a user interface can be added to the objects that have been developed using the techniques of the previous chapter.

The end of this chapter is full of (smaller) examples that demonstrate in different steps how a user interface can be made using Visto, but before that we first outline the different stages of development that we have in mind for building an application in Haskell and equipping it with a user interface in Visto.

A somewhat larger example is for the following chapter – a case study – where we compare a Visto application with an implementation in other languages and their toolkits.

But first things first: we start by pointing out our research goals and then describe the initial – simplistic – model for a GUI in Visto, illustrated by a rough and ready example. We then present the stages of development and the previous examples we already refered to. We then take a closer look at the visualisation properties of Visto and at some more details on how to create the GUI.

Again – as is mostly the case – a comparison to other work and the conclusions are left to the end of the chapter.

Although we can present some nice findings of our research, we must also admit that this has been no easy task. There is a vast richness present in user interface research, not only in functional and declarative languages, but especially in general. From the very beginning much functionality was envisaged, but often considered unreachable because either computers were simply not fast enough, or users not patient enough. That's why NextStep tried to repaint the entire contents of a dragged window, but MacOS only drew the box outlines of the window. With the advent of more powerful computers the unreachable came within reach and the impossible possible. Naturally our individual research cannot touch the enormous richness present in accumulated interface research, but when appropriate we will point out some possible further directions.

4.2 Research Goals

Most GUI systems consider the user interface as a collection of widgets totally independent from the application. Their programming language works with widget combinators or provides a tool for drawing and positioning the widgets. Adding functionality is typically accomplished by attaching call back functions to the widgets. Changing the user interface then becomes a difficult task, requiring not only to replace the widgets, but also to define new call back functions, as their type often depends on the widget. This scenario explains that typical GUI programming is far from declarative. It focuses on the *how* in 2 ways:

- first one chooses a particular kind of widget (a button, a menu or a key short cut, . . .) instead of providing suitable abstractions,
- then one adds a call back function to the widget to provide the desired functionality.

In this sequence one first chooses *how* something is to be invoked (through which widget) and afterwards adds *what* is to be done (which call back function). This means that the stress is on what the user interface looks like and not on what it can do.

This is also potentially dangerous: the user interface that has been directly drawn or developed with only in mind how it should appear, may look good, but if it doesn't offer all the desired functionality, it's no good user interface. Only if we clearly identify the desired functionality of the application and know which functions should be present in the user interface, can we constitute a good interface. Visto must do precisely that. So instead of adding behaviour and functionality to a user interface, Visto insists that a user interface is added to the defined functionality.

Although this is more a matter of design than of implementation, Visto makes sure that the design decision is more properly reflected in the Visto source code than in the code of traditional systems.

Our main goals are

- a system that describes in a proper – declarative – way the *functionality* (i.e. behaviour) of the user interface,

- and that describes neatly how this functionality can be triggered from the user interface,
- support for easy modifications, making sure that they have little impact on the rest of the application,
- and, as far as this is possible, means for specifying layout independently from the rest of the application and from the connection between application and user interface.

4.3 A Simple Model

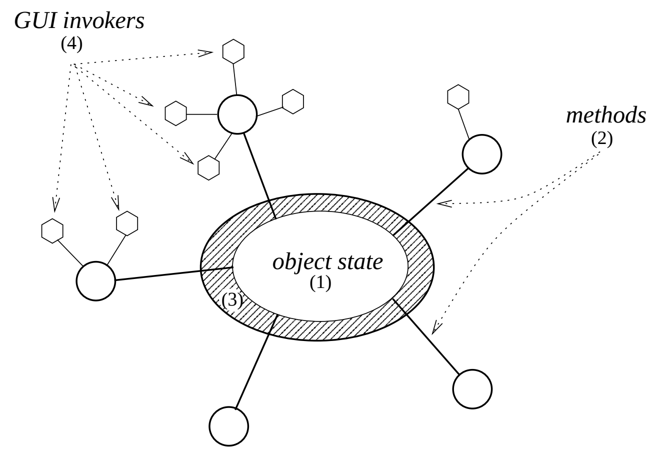


Figure 4.1: A Simple Model.

Our simple model is an incremental model.

- The first phase, before coding in Visto, is the development of a Haskell program that contains the core functionality of the application, or the reuse of an existing Haskell program.

  The programmer then switches to Visto and defines *visualising* objects. After all, Visto is an acronym for *VI*sualising *S*tate *T*ransforming *O*bjects.

  These objects have a state *(1)* of any arbitrary Haskell type. They correspond to visual entities that are relevant to the user of the application, not to the logical entities that the programmer chose to adequately represent the problem domain.

- Then the programmer defines the functionality of these visualising objects: he defines the methods *(2)*. He is supposed to do this independently from the way in which the user interface might be constructed. He can provide both methods that are directly relevant to the user of this program and methods that have no direct relevance, but that nevertheless describe functionality of the 'user' objects.

  The point is that the programmer tries to define the ultimate working set of methods for these objects, such that the chances decrease that this part of the implementation will have to be changed.

  And, just as in all object languages, these methods are the only way in which the object can be accessed and updated.

  These two first parts of Visto have been extensively discussed in the previous chapter on the object principles of Visto.

- The *visualising* part of Visto now comes into play. For each object we can define a supplemental **draw** method *(3)* that describes how the object has to be drawn on screen. Visto takes care that this *draw* method is executed whenever the object evaluates a method, which enforces that the information display is consistent with the objects' states as the methods are the only way in which the methods can change their state.

- Finally, the programmer decides which methods he will present directly to the user in the user interface, and in what way they will be presented: a button, menu, key short cut, ... Currently this is done in the program code, but we plan to develop an interactve environment for it.

  Not every method must have associated GUI invokers. It is more likely that only a subset will be needed; a small subset if we want a basic interface, a larger one for more complicated interfaces.

  A single method can be easily associated with multiple invokers, which is in general a good practice. A good user interface should provide different ways for accomplishing the same task, e.g. use the *save* item from the *File* menu, or press a Alt-S key combination.

Summarised, the objects in Visto and their methods describe the functionality of the application and form the start basis of any Visto project. In a second phase the programmer identifies which methods are to be called from within the user interface and adds extra 'GUI methods' to the Visto object definition. On the left hand side of such a method we have the name of the method to be invoked, preceded by the keyword **GUI**, and on the right hand the different GUI invokers, such as buttons, key presses and menu items. A selected method can be associated with any number of invokers to allow alternatives in the interface and to please both expert and non expert users. This also allows for easy modification: just change, add or remove some invokers and recompile!

The important matter of consistent information display is taken care of by the **draw** method. By following Visto's design philosophy the development of such a GUI application is thus automatically centred around the visualisation of information.

Through these various steps we hope to achieve an enhanced declarativity when defining user interfaces, focusing more on the *what* instead of on the *how*. We will compare this with traditional Model-View-Controller (MVC) after we moved to a more complex model.

This simple model couldn't survive long because in a real life application not all objects will necessarily visualise something, although at least one visual environment, Animation [19], also prototype based as Visto is, has taken the extreme approach of giving every object in the system a visual representation. Just as the *abstract Fudget* we can also have *abstract* objects in Visto.[1] So a **draw** method is no longer necessary. If an object provides one, it is considered a real visualising object and the *draw* will be automatically called.

Also the object oriented features amplified the possibilities for designing Visto interfaces. As we can have an object that expands another one, and whose methods trigger the methods from the original object, we can separate the definition of the user interface even more from the definition of the behavioural object and its methods.

In this more complex model (fig. 4.2) we have a model that contains the application data. It defines methods, but doesn't have a *draw* definition, nor GUI invokers. Two objects expand this model:

- a activator that adds GUI invokers to the methods, and that may have some different state that contains GUI related data.
- a view that provides a **draw** method and a state that can contain data concerning the way in which to display the model's state, e.g. the colour or font face.

4.3.1 A Comparison with MVC

This scenario is related with the traditional model-view-controller pattern (MVC), well known from SmallTalk. We share the aspect of separating responsibilities:

- A *model* contains the data of the application model. Methods describe the behaviour and may alter the state of the model.
- A *view* displays the model in a visually attracting way.
- A *controller* takes care of the GUI.

[1] 'Sto' objects if you like.

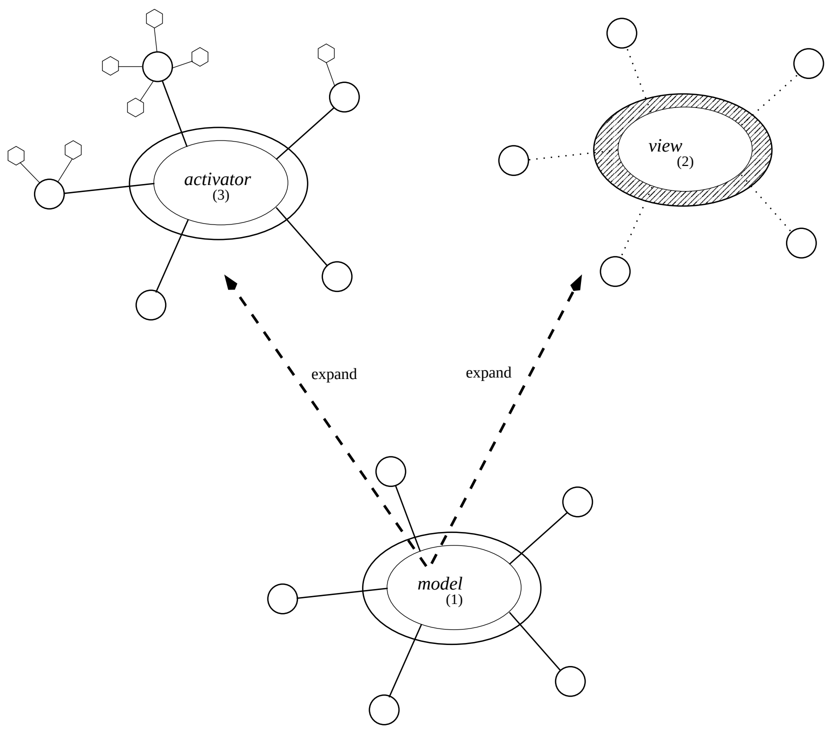


Figure 4.2: A More Complex Model.

We agree completely with the role of the view. In a properly programmed application the model should not contain references to how it must be displayed. This is undesirable as changes to how some object is displayed, should not effect the model. The behaviour of something doesn't change when its presentation changes.

Furthermore, we must be able to easily define multiple views for a single object. When the views are embedded in the model, this complicates and puts too much load on the definition of the model.

Both MVC and Visto support multiple, independently defined views on a model, facilitating a good, modular design of a program.

In principle we also agree with the idea of a model only modelling the application data and not the view, nor the GUI, but we have a different interpretation of *the application data*. We differentiate between the application model and the user model. The application model on the one hand is the model as perceived by the programmer of the application. This model can be different from the model perceived by the user of the application. E.g. in a flight booking system, the programmer can prefer to work with planes and companies, whereas the user may only be interested in destinations.

The model in Visto is the user model, and the methods of the model are the user actions. This is different from typical MVC, where the model contains the programmer's model. The user actions are defined in the controller and are *just* a composition of methods from the programmer's model.

That is where a more fundamental difference between MVC and Visto comes into play. The MVC-controller is directly related to a particular GUI-object, which is most often a widget. If we use a button, we need a controller for that button. It "interprets the mouse and keyboard inputs from the user" [14] on the active area of the widget. In the same manner we must define in Java a *Listener* for each widget. The listener listens to all the events coming from the widget. The process is thus entirely controlled by the events, as the controller decides which methods to use, depending on which events occur.

So although we choose a particular widget because of what it selects – in the selectors idea –, and because of its appearance and its visual behaviour, the implementation of the controller depends on the events that the widgets trigger. So if a button emits *Click*s, as in Fudgets, our controller must react on *Click*s and not on buttons. Therefore we find that the implementation of a controller is more technical than declarative, working on the events and not on what the user did: activating a widget, or even more abstract, activating a method.

That is also the viewpoint of Visto. We start with the goal of the GUI, i.e. calling a method and selecting arguments for the method. The user naturally intends to call methods of *his model* and not methods of *the programmer's model*. That is why we have a user model in Visto, separate from the application model in the Haskell program.

Our controller *expands* the user model using the expand object derivation. In principle it has the same methods as the user model, describing for each method how the user can invoke it. This must of course be done in a visually attracting way, but, working in declarative environment, we use invokers and selectors here, which we instantiate graphically only later on. So our controller contains a number of methods, and for each method, a number of ways in which the methods can be activated from the GUI. As we place the GUI with the method, we directly mention that the GUI activates methods from the user model.

This is not the case in traditional MVC. There we put the controller with the widgets, and only specify in the implementation of the controller what happens when the widgets are activated. This may be a method of the model, but that is not necessarily the case.

Therefore, if we believe that the models and their methods form the functionality of the system, the Visto scheme is more declarative because in Visto the

GUI can only invoke methods of user models. In traditional MVC controllers can simply do anything. That is of course more flexible and may be better for something flashy, such as a document that already starts to desintegrate when it approaches the trashcan, but, to make a blatant comparison, machine language is also more flexible than standard Haskell.

Because the Visto controller automatically contains the activation of methods, we prefer to call it an *activator*. The MVC controller is not active by default. It only becomes active when the programmer adds actions as reaction to events.

Another difference is the relation of the controller with the view. In SmallTalk there is a one-to-one relationship between controller and view, and also in Java they are strongly related as they contain references to each other. In SmallTalk we can only have a view without a controller when the view has as controller an element of the class *NoController*.

This is completely different from Visto, where view and controller (activator) are independent. We find this more appropriate as we prefer to separate these separate issues. On the other hand the separation complicates the design of strongly integrated view-controller items, such as the clock that can be both a view and a selector at the same time.

The reason why view and controller are so strong related in MVC, is that a controller is an object that controls a particular area of the screen, and that handles all the events that occur on that area. As a view occupies an area on screen, MVC requests a controller for the area. As our activator doesn't work that way, it doesn't need a view, and also vice-versa: a view doesn't need an activator.

We disagree with the strict relation "controller - active area on screen". The area relevant for a small area with the focus may be a lot larger than the active area. Consider for example a view on a clock. It may be a good idea to have a controller in MVC that updates the clock when its pointers are moved, but if we also want to be able to control the clock with a menu bar and a button row on a distant area of the screen, we must either enlarge the active area of the controller, or spread the clock control over various controller-view-pairs. The first solution is not adequate as the menu bar may be meant to control many different views. The second solution spreads the control over one logical entity over different controllers and may induce code and behavioural duplication.

So in Visto the activator doesn't control an active area of the screen, but rather contains a list of ways for activating methods.

Finally, Visto is also more flexible from an implementation point of view. Any Visto object can be a model. It doesn't have to be explicitly designed for use with MVC, contrary to SmallTalk and Java where this is the case.

In SmallTalk the model of MVC must inherit from the class *Model*; in Java from the class *Observable*. And when a method updates the object's state, it must call the methods *setChanged()* and *notifyObservers()*. The views (the *observers* of the model) react on those methods by invoking their *update()* method. The communication from model to view thus always goes through that single method *update()*, even if the model may activate its *notifyObservers()* in different methods. The *update()* may then become a very large method as it must take care of all possible activations of *notifyObservers()*.

As the triggering and delegation of Visto works on the level of methods (i.e. a method in the model triggers the method with the same name in each view), it is more easy to take that precise information into account. The view knows by which method it is triggered.

The conclusion is that Visto's model may be very similar to MVC at first sight, but that it is actually different in many places, especially in the concept of the controller vs. activator, and in the way in which the design must be implemented.

4.3.2 A Comparison with Arch Metamodel

In the previous section, and in sec. 2.2 we used MVC as our reference point. We did this because MVC is more commonly known, and because Visto contrasts more with MVC than with the Arch Metamodel [84]. That is something inherent to metamodels as they must try to accommodate as many models as possible.

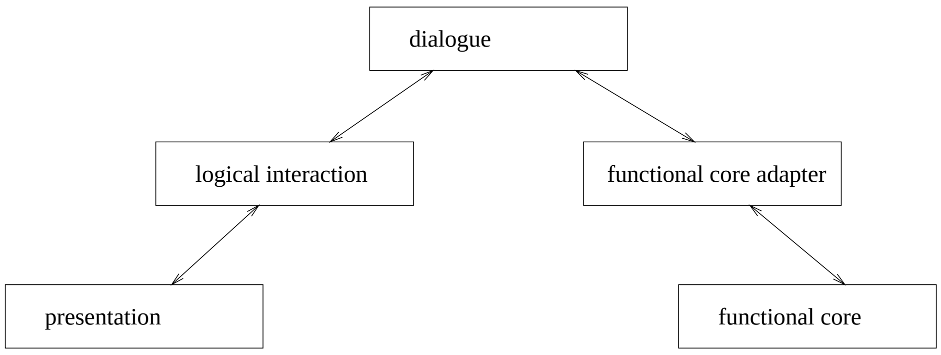


Figure 4.3: The Arch Metamodel

In fig. 4.3 we can notice five parts. These five parts can be found in Visto as well:

1. The Functional Core simply is the Haskell Application Layer.
2. The Functional Core Adapter is composed of the different Visto objects whose methods may use the functional core.

3. The Dialogue is present in the invokers and selectors.

4. The Logical Interaction between the dialogue and the presentation is part of the Visto internals and requires no extra programming efforts.

5. The Presentation is defined by the *draw* methods of the visualising objects.

The problem of course with such a high level metamodel is that it is subject to many interpretations. So, this Visto incarnation of the Arch Metamodel is probably only of one of the many possible incarnations.

The essential point is that Visto fits into the Arch metamodel. Although MVC fits in it as well, it is interesting to notice that we find a number of interesting differences between MVC and Visto if we compare them directly with each other, instead of going through the metamodel.

4.4 A Quick and Dirty Example

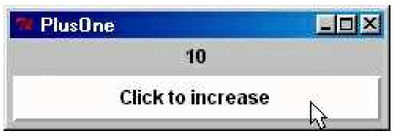


Figure 4.4: A Simple Counter.

Let's now complete the traditional counter to have a first look at a first complete GUI derived in Visto :

```
interface Counter
    up  :: Int


object PlusOne Counter Int 0 {
   draw   = Label
              text (show state),
   up     = let
                s′ = state + 1
              in
              (s′, s′),
   GUIup  = Button “Click to increase”; Key “ + ” }
```

4.4.1 Object aspects

From the knowledge acquired from the previous chapter, the object aspects of this example should be straightforward.

PlusOne is an object that conforms to the *interface* Counter, which says that it should implement one method *up* returning an int. It does so by increasing its state with 1, and using that new value as return value of the method.

4.4.2 Visual Output

This *PlusOne* object is also a *visualising* object, as it defines a ***draw*** method. Visto guarantees that this method (and the resulting screen refresh) is performed whenever needed. The application of **text** *(show state)* draws the current state of the counter on a **Label**, a one line text display.

4.4.3 Interaction

At the other end of the spectrum we have the need to *manipulate* our data, mostly through the user interface. To indicate which methods are to be called from within the GUI, we can add some extra equations to the object's definition, with on the left hand side the name of the method preceded by the keyword **GUI** and on the right hand side the different invokers, separated by a **;**.

The *PlusOne* example contains two invokers for the *up* method, a button with the label *"Click to increase"* and a key shortcut '+': both clicking the button or pressing the + sign will increase the value of *PlusOne*.

4.4.4 Quick and Dirty

We referred in the section title to this example as *quick and dirty*. This is true because we have mixed in one single object both behaviour and state of the object, the way we view that state and how it is called from within the user interface. Typically this should be separated. A means for doing so has already been presented and will be continued in the following sections, but Visto also allows for a quick and dirty approach which might be valid for early prototyping, when separation of user interface and application, and reusability may be less important.

4.5 Stages of Development

The various stages have already been presented in sec. 1.5.2, but we repeat them shortly here to enable a smoother reading of the development process of the two examples in the next section as well as of the case study in the next chapter.

1. Implement the application layer in Haskell.
2. Define Visto objects and methods representing the user's view on the application, and add *draw* methods for the visualising objects.
3. Choose which methods must be present in the UI (define invokers) and use selectors to gather the arguments for these methods.
4. Finalise the GUI by instantiating the invokers and selectors and by laying them out properly.

Of course, whenever the UI changes and as long as the current GUI isn't like it ought to be, one must refine the implementation by changing the steps made in stage 4 or by moving back to previous stages as described in sec. 1.5.2.

4.6 More Examples

The purpose of this section is to let these two small examples shed more light on how a user interface can be created in Visto with little effort.

In later sections we describe more extensively the features that are introduced and used intuitively in this section.

4.6.1 A Calculator

Yet another typical example is the calculator.

The Functional Core

Except for the standard numerical operators this example doesn't use any (self defined) Haskell functions. Nevertheless we can identify the functional core of a calculator. Given a current state, a calculator accepts functions, such as + and *. It then takes another number and applies the function to both values, which gives a new state for the calculator.

This is basically a state machine that accepts either a numeric value or a function with two arguments. Whenever the function is entered, the state machine uses the current numeric value as the first argument for that function, accepts another numerical value as the second argument and then awaits the message to apply the function.

This behaviour is first translated to a Visto *interface*:

```
interface BinaryTransformer
  addFunction
       :: (Int → Int → Int) → (Int → Int)
  addValue :: Int → Int
  applyFunction :: Int
```

The type for the return value of *addFunction* reflects the fact that we will use the current numerical value as the first argument for currying the function. *addValue* takes an int that serves as the second argument for the function, and returns an int. When *applyFunction* is applied, both arguments should have been supplied and therefore *applyFunction* takes no more arguments.

When we implement the object, we first have to choose a type for the object's state variable. We use the now well known technique from [88] and [32] with two elements in the state: the current numeric value and a one argument function.

```
object Calculator BinaryTransformer
      (Int, Int → Int) (0, λ x → 0) {
  addFunction   = λ fun → let
                            (value, _) = state
                            newFun = fun value
                          in
                          (newFun, (value, newFun)),
  addValue      = λ val → let
                            (value, fun) = state
                          in
                          (value, (val, fun)),
  applyFunction = let                                }
                    (value, fun) = state
                    newValue = fun value
                  in
                  (newValue, (newValue, fun))
```

In both *addFunction* and *applyFunction* we naturally return the new values. For *addValue* we prefer to return the old value to the caller of the method, so that that old value will not get automatically lost and might still be used by the calling object.

Note that we can use all the object features from the previous chapter. So we can easily create another object that prefers to return the new value when *addValue* is called. This can be done in an easy way by using the **clone** derivation.

```
clone Calculator2 from Calculator {
  addValue = λ val → let
                       (_, fun) = state
                     in
                     (val, (val, fun)) }
```

Or, almost similar to class based creation of objects, we can use the clone feature to create several different objects that differ only in their state. The default value for the function in our calculator is the one that returns zero. It may be preferable to use the identity function or one that reports an error.

> **clone** *CalculatorId* **from** *Calculator* $\{$**state** $= (0, id)\}$
> **clone** *CalculatorError* **from** *Calculator*2 $\{$
>   **state** $= (0, \lambda x \rightarrow$ *error* "*no function present yet*"$)$ $\}$

A User Interface

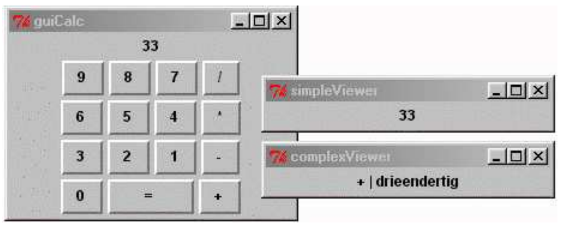


Figure 4.5: The Calculator with 2 Views.

The functional core works with complete values, e.g. 312, 4087, ... , not like a real calculator in which users enter value digit by digit. They individually enter 3, 1, and 2 to form 312. And submitting that value is not really a separate process as in the functional core, but an implicit one when the function symbol is pressed: this both submits the number and the function; or when we press the '=' symbol to get the result, we actually do two things: submit the number and ask for the result.[2] We have to create a new object for such a GUI, with the following object interface:

> **interface** *Activator*
>   *addDigit* :: *Int* $\rightarrow$ *Int*
>   *addFunction* :: $($*Int* $\rightarrow$ *Int* $\rightarrow$ *Int*$)$ $\rightarrow$ $($*Int* $\rightarrow$ *Int*$)$
>   *getResult* :: *Int*

We can notice that the interface for *BinaryTransformer* and *Controller* share the method *addFunction.* This is no problem as long as the method type in both interfaces is identical. This feature is similar to the polymorphism used in object oriented languages. It is translated using type classes and so in the end uses the polymorphism of both language systems.

[2] Better interfaces for the functional core of a calculator may exist, but we deliberately chose one that differs from the GUI object to show yet another design alternative for separating GUI from application, and to show an example of a user's model that is different from the programmer's model.

To implement the actual controller, which we will name *guiCalc*, we have to refer to the calculator. We do this by sending the appropriate messages:

```
object guiCalc Activator (Int, Bool) (0, True) {
  addDigit = λ dig → let
                       (x, bool) = state
                       newX = newDigit x dig bool
                     in
                     (newX, (newX, True)),
  addFunction = λf → let
                       (val, _) = state
                       okVal1 = Calculator!addValue val
                       okVal2 = Calculator!addFunction f
                     in
                     (okVal2, (val, False)),
  getResult = let
                (val, _) = state
                okVal = Calculator!addValue val
                newValue = Calculator!applyFunction
              in
              (newValue, (newValue, False)),
  ...      }
newDigit val digit True = (val * 10) + digit
newDigit _ digit False = digit
```

Up until now we have just defined which functionality *can* be present in the user interface. Now we have to define the functionality that *will* be present in the user interface. To do this, we add some GUI associations, associating methods (the *what*) with ways *how* to invoke those methods.

Basically we need buttons for each digit and for each function symbol, as well as a button for the '=' symbol and provide key short cuts for all these buttons.

The object definition is complemented with:

```
object guiCalc Activator ... {
  GUIgetResult = Button " = "; Key " = ",


  ...
  GUIaddDigit =Button "1" <<− 1;
               Button "2" <<− 2;
               Button "3" <<− 3;
               ...
               Key "0" <<− 0;
               Key "1" <<− 1;
               Key "2" <<− 2;
               ...
```

```
    GUIaddFunction =Button "+" <<− (+);
                    Key "+"    <<− (+);
                    Button "−" <<− (−);
                    Key "−"    <<− (−);
                    ...
}
```

In this example we associate the methods *addDigit*, *getResult* and *addFunction* with GUI-invokers. If they are methods without arguments (as *getResult*), it suffices to just have a number of invokers, in this case just a Button and a Key. So *getResult* will be executed whenever either the "=" button or the "=" key are pressed.

In the case of *addDigit* and *addFunction*, when the methods need one or more arguments, we still have to supply those arguments. They are separated by $<<-$ and in this case fixed: 0, 1, 2, ..., 9 or the functions (+),(-),(/),(*), but, of course, these arguments can be selected by selectors as well.

Visualisation

Once these associations of methods with GUI invokers have been defined, the user can control the calculator. Unfortunately there's no view on the data yet. At least two alternatives can be examined.

The first option is adding a **draw** method to the *guiCalc* object. This way the view on the state of *guiCalc* is updated every time one of its methods is executed, including whenever one of the buttons or one of the key short cuts is pressed. This is clearly a desirable behaviour and is indeed quite easy:

```
object guiCalc Controller ... {
   draw  = Label
             text (show (fst state)),
   ...
}
```

The other alternative creates a separate view on the calculator itself, whereas the first option only depicted the activator's state.

This model-view relationship is so special, certainly in user interfaces, that Visto provides a dedicated derivation format, **expand**.

This derivation tightly couples two objects. The derived object will be kept informed about state changes in the original object, the *super*-object. A first consequence is that its **draw** will be executed whenever the superobject receives a message. It can also access the superobject's state by the use of the keyword **superState**.

```
expandInterface BTViewer1 from BinaryTransformer
  _addFunction
  _applyFunction
  _addValue
```

By preceding a method name with a underscore (_), we indicate that we want this method to be removed from the derived object. As a view is not to change the model we remove in this example all methods. Although the derived objects cannot receive any methods on their own the object remains useful because of the visualising aspect.

```
expand simpleViewer BTViewer1 from Calculator {
  draw = Label
           text (show (fst superState)) }
```

Besides simply accessing the **superState**, the derived object can *delegate* its messages to the superobject, a technique well known and often used in object based languages. This is done using the keyword **superValue**, that returns the return value of the delegated message.

Interesting is also the technique of triggering which makes sure that whenever a method in the superobject is called that same method will also be called in all of the derived objects – if they want that feature to be enabled. This triggering is particularly useful when the derived object has some state that must be kept consistent with the superobject's state.

Consider this more advanced example in which we both try to guess which function is used[3] and display the calculator's numeric value in textual form.

```
expandInterface BTViewer2 from BinaryTransformer
  _applyFunction
  _addValue
  setLanguage :: String → String
```

Whenever the calculator is to use a new function, we want to be informed and thus we did not remove *addFunction* from the *BTViewer*2 interface. We also define a new method that sets the language in which the current numeric value is translated.

```
expand complexViewer BTViewer2 (String,String)
      ("Unknown function","Dutch") from Calculator {
  draw = Label
           let   (value, _) = superState
                 (guess,language) = state
           text  (guess ++ " | "
                   ++ (translate language value)),
```

[3]The defi nition of such a function *guessFunction* is left to the imagination of the reader.

```
    seLanguage =λlang → (snd state, (fst state, lang)),
    addFunction =+
        λfun → let
                    thisResul = superValue
                        — the delegated return value
                    thisGuess = guessFunction fun
                    newState = (thisGuess, (snd state))
                in
                (thisResul, newState)      }
```

We use =+ instead of = in the definition of *addFunction* to indicate that this method should be executed whenever the *addFunction* method of the Calculator is executed. That way we can transparently add an expansion (the *decorator* design pattern from [29]) and still be assured that the view's state is consistently updated.

4.6.2 A Switcher

Time now to describe a seemingly useless application but at least one that demonstrates some of the run time modifications that can be made to the user interface.

```
interface BooleanLike
    true :: Int
    false :: Int
    invert :: Int
```

The interface *BooleanLike* has three methods: two for setting the object's state either true or false, and another one that inverts the state. The return value of type *Int* is a count for the number of accesses.

The implementation of the methods is quite simple, but the interesting aspect is that we temporarily disable the GUI for the redundant methods, i.e. the method *true* will be disabled when the state already is *true*, and just the same for *false*.

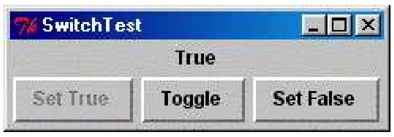


Figure 4.6: A Switcher.

```
object SwitchTest BooleanLike Switcher startSwitch {
    draw = Label
            text (show (onValue state)),
true = let
          disableMethod "true"
          enableMethod "false"
          newCount = (trueCount state) + 1
          newState = state{onValue = True,
                           trueCount = newCount}
       in
       (newCount, newState),
false = let
          disableMethod "true"
          enableMethod "false"
          newCount = (falseCount state) + 1
          newState = state{onValue = False,
                           falseCount = newCount}
        in
        (newCount, newState),
invert = let
          newBool = not (onValue state)
          ableMethod (not newBool) "true"
          ableMethod newBool "false"
          (newTrue, newFalse) = increaseBool switch newBool
          newState    = state {onValue = newBool,
                               trueCount = newTrue,
                               falseCount = newFalse}
         in
         (newTrue + newFalse, newState),
GUItrue =  Button "Set True"; Key " + ",
GUIfalse = Button "Set False"; Key " − ",
GUIinvert =Button "Toggle"
}
```

When the method *true* is executed, we first do a **disableMethod** of the method *true*. This means that all the GUI invokers of the method *true* will be disabled. It does not mean that the method *true* itself is disabled. Another method or object can still call *true*. However, it can be no longer invoked from within the user interface.

enableMethod, **disableMethod** and **ableMethod** have an effect on *all* the GUI invokers of the method. In the next section we present more fine grained operators for run time behaviour that affect only particular elements of the user interface, e.g. a particular entry, button or selector.

To complete the code for this example, a little bit more Haskell code is needed, i.e. for the data type *Switcher* that is used as the state for the object *SwitchTest*.

$$\begin{aligned}&\textbf{data}\ \mathit{Switcher} = \mathit{Switcher}\ \{\mathit{onValue} :: \mathit{Bool},\\ &\qquad\qquad\qquad\qquad\quad \mathit{trueCount} :: \mathit{Int},\\ &\qquad\qquad\qquad\qquad\quad \mathit{falseCount} :: \mathit{Int}\}\end{aligned}$$

$$\begin{aligned}&\mathit{startSwitch} = \mathit{Switcher}\ \{\ \mathit{onValue} = \mathit{True},\\ &\qquad\qquad\qquad\qquad\ \ \mathit{trueCount} = 0,\\ &\qquad\qquad\qquad\qquad\ \ \mathit{falseCount} = 0\}\end{aligned}$$

$$\begin{aligned}&\mathit{increaseBool\ switch\ True}\\ &\quad = ((\mathit{trueCount\ switch}) + 1, \mathit{falseCount\ switch})\\ &\mathit{increaseBool\ switch\ False}\\ &\quad = (\mathit{trueCount\ switch}, (\mathit{falseCount\ switch}) + 1)\end{aligned}$$

4.7 From Methods to GUI Invokers

One of the most innovative features of Visto is that it steps away from the typical call back function approach. The call back function is what is added to a GUI object (mostly a widget) in order to let it communicate with the functionality of the application. It is a *back* function as it is an argument of the GUI. *How* this functionality is called, seems to be considered more important than *what* is called. This is clearly not so declarative.

Visto's approach could be called *call forward*. Not only does it sound more positive, it is also more declarative as the use of the interface is stressed over the widget that invokes it. We define a method for an object, then state in the left hand side of the **GUI** $<$ *methodName* $>$ which says that this method will be invoked in the GUI, and finally state in the right hand side *how* this will be done and in what way it will be visualised.

In the introductory *quick and dirty* example the *up* method can be invoked either by clicking the *"Click to increase"* button or by pressing the + key. [4]

$$\textbf{GUI}\mathit{up} = \textbf{Button}\ \text{"}\mathit{Click\ to\ increase}\text{"};\ \textbf{Key}\ \text{"}+\text{"}$$

Defining a new interface now becomes extremely easy. Just change, add or remove some GUI invokers and recompile.

We can even separate individual design decisions: if we want to change the functionality of the user interface, we have to add or remove some methods from the GUI. If we want to change the way in which the methods are called, we change the invokers or selectors.

[4] The different invokers are separated by a **;**.

Invokers are user interface elements that invoke a method, such as buttons or key presses, whereas selectors provide arguments for the methods, such as entries and pull down menu's. These arguments are supplied after the invoker and are separated by $<<-$'s as in

$$\textbf{GUI}\mathit{addDigit} = \mathit{Button}\text{ “1” } <<- \ 1;$$

which says that whenever the button with label “1” is pressed the method *addDigit* will be called with argument 1.

Naturally the arguments for a method are not limited to fixed values. We can also use values that are selected by the user in the user interface. We might be tempted to provide concrete widgets such as an entry to select to those data, but of course, we wouldn't have discussed our successful selectors if we weren't to use them here. We have implemented a range of selectors [40, 3], such as *selectOneFromRange* that selects a value from a range of possible values e.g. [1..13], or *selectOneFromDiscrete* that selects from a discrete set of values.

For the invokers we currently only supply concrete widgets:

$$
\begin{array}{lll}
\textbf{Invoker} = & \textbf{Button} & \mathit{label} :: \mathit{String} \\
 & & \quad \text{— the label for the button} \\
\mid & \textbf{Button}' & \mathit{name} :: \mathit{String}\ \mathit{label} :: \mathit{String} \\
\mid & \textbf{Key} & \mathit{String} \\
 & & \quad \text{— the key to press} \\
\mid & \textbf{Key}' & \mathit{name} :: \mathit{String}\ \mathit{key} :: \mathit{String} \\
\mid & \textbf{Menu} & (\mathit{topLabel} :: \mathit{String}, \mathit{pos} :: \mathit{Int}) \\
 & & \mathit{label} :: \mathit{String} \\
 & & \quad \text{— the label for the menu item}^{5} \\
\mid & \textbf{Menu}' & \mathit{name} :: \mathit{String} \\
 & & (\mathit{topLabel} :: \mathit{String}, \mathit{pos} :: \mathit{Int}) \\
 & & \mathit{label} :: \mathit{String} \\
\mid & \textbf{NamedInvoker} & \mathit{name} :: \mathit{String}
\end{array}
$$

We have supplied both unprimed and primed versions, e.g. *Button* and *Button′* that provide either anonymous or named invokers and selectors. After all a value that is selected for the purpose of one invoker, can also be used in another invoker; or a button can be used to invoke two different methods.

The named invokers and selectors can then be easily reused by their name.

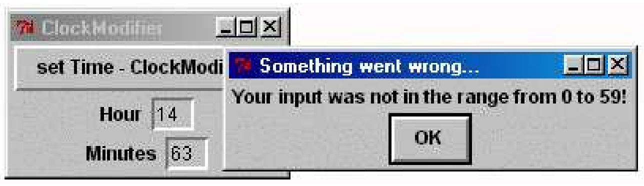
In the following example we define three invocations for a method *setTime* of type $\mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{Ordering}$. This method takes an hour, the minutes

[5] The menu item will appear under the *topLabel* menu at position *pos*. Visto collects all *Menu* invokers and builds the corresponding menu bar with separators inserted in missing places.

and seconds, and thus we must have three arguments (selectors) for each of the invokers to supply the hour, the minutes and the seconds.

```
GUIsetTime =
  Button "set Time"  <<- SelectOneFromRange' "SetHour" "Hour" 0 23
                     <<- SelectOneFromRange "Minutes" 0 59
                     <<- 0;
  Button "set Hour"  <<- NamedSelector "SetHour"
                     <<- 0 <<- 0;
  Button "Midnight"  <<- 0 <<- 0 <<- 0
```

We use the *selector* **SelectOneFromRange** to select a value from a continuous range. The first argument of *setTime*, the hour, ranges from 0 to 23; the second, the minutes from 0 to 59. The third is fixed at 0, as do all the parameters for the alternative of setting the clock at midnight.



Your input was not in the range from 0 to 59!

Figure 4.7: A selectOneFromRange with input out of range.

This **SelectOneFromRange** could be drawn with entry fields, slider bars, radio buttons, ... but this is abstracted out in the definition.

The tool for setting the layout is still unimplemented. Currently we can fine tune the layout by rewriting the compiled TkGofer code, but this can of course be no final solution. In the final version of the tool this will be done in a GUI builder as described in section 1.5.2.

Run Time Modifications

The set of associations of methods with GUI invokers defines the initial state of the user interface. Of course this is something that is bound to change in time. The most important changes in functionality are disabling or enabling part of the interface. Visto provides ways for doing that.

We consider three different levels:

- **ableMethod** On the first level we find the method. A modification (enable or disable) may be performed on an entire method. E.g. in the calculator example we may disable the function application as long as no valid number has been entered. Or any means for invoking the *save* method may be disabled as long as no changes have been made to the document.

- **ableAssociation** On the second level we may wish to disable a particular way of invoking the method, e.g. we may indicate that the key short-cut is no longer valid. This modification not only influences the use of the specific invoker, but also the selectors that provide the arguments for the invoker. So **disableAssociation** disables both the Button and the selectors that supply the arguments.
- **ableElement** Or we may, on the lowest level, just disable or enable a single element, e.g. a particular *SelectOneFromRange* or grey out a button without influencing the selectors that provide arguments. This is particularly useful to ensure that the correct arguments are entered before enabling the invoker.

All modifiers exist in three versions

- **disable***XXX*[6] “*name*”, **enable***XXX* “*name*” disables, or enables, the widget(s).
- **able***XXX bool* “*name*” takes an extra boolean argument that specifies whether to enable or disable the invoker or selector.

The argument *name* naturally depends on the primitive. In the case of *ableMethod* it is the method name, in the case of *ableAssociation* the name of an invoker and in the third case, *ableElement* the name of a selector or invoker. The *ableMethod* and *ableAssociation* primitives will of course include anonymous selectors that appear with the GUI for that method or after the invoker, but in the second and the third syntax the invoker and selector naturally cannot be anonymous.

4.8 Visto’s are Visualising Objects

Not only do we need to keep a local state and access it through the user interface, we should show it as well to the user. Visto puts strong emphasis on this visual output. The visualising part of Visto comes into play when the programmer defines a **draw** method for his object. This special method, that does not return any value, but only contains drawing (side effecting) operations, is executed whenever a message is sent to the object, thereby guaranteeing that the displayed information is always consistent with the present state.

The model - view methodology can then be simply applied in Visto by defining **draw** methods for the viewers, but not for the model. Drawing an analogue view of a clock can then be defined as:

[6] *XXX* is to be replaced with either **Method**,**Association** or **Element**.

```
expandInterface ClockView from Clock
  _setTime[7]

expand AnalogueClock ClockView String "green"
        from ClockModel {
  draw = Canvas (100,100)
           clear
           let time = superState
           setColor state
           line (50,50) (hourCoordinate (timeHour time))
           line (50,50) (minuteCoordinate (timeMin time))
}
```

The first line of the Visto **draw** method defines the 'widget' to draw on.

- **Label**: a single line containing a string, updated by the command **text** which takes as an argument the string to be displayed.
- **Canvas** *size :: Point* [8]: a window with a number of drawing operations (defined in the table below).

Drawing	*Setting Options*
line *Point Point*	setColor *color :: String* [9]
rect *Point Point*	setFillColor *color :: String*
oval *Point Point*	setFillColor *color :: String*
clear	setWidth *Int*

Figure 4.8: Operations on Canvas

Besides these operations any other valid TkGofer code is allowed (where reference to the current label or canvas can be made through **thisLabel** and **thisCanvas**). This way we keep simple things simple, while allowing expert users to create cooler and more complicated effects. Future versions of Visto will include more window and life time related primitives.

This **draw** method strictly localises the code responsible for displaying the state of the object. It therefore increases maintainability of the application. To create a different view on the data, it suffices to change the **draw** method.

This automatic *draw* feature is an original approach in functional object systems, but not in general. Its application in Java with the *paint*, *repaint* and *update*

[7] A view should not change its model. Therefore we remove the *setTime* method.

[8] A Point is a tuple (Int,Int).

[9] For exact color names consult the TkGofer manual.

methods proves its practical applicability. It certainly simplifies event handling a lot.

However, in Visto this is currently the only place where drawing can take place. This means that dynamic effects such as zoom, balloon help and the example of the document that starts to desintegrate as soon as it approaches the trashcan are impossible to implement.

This is of course a strong restriction of Visto. We used this strategy of a separate draw method because we wanted the actual method definitions to be side effect free, just as in an functional language.

The best solution to let drawing commands appear in method definitions is probable reverting to monadic style. However, from the very start we prefered not to confront the user of Visto with monadic style, because programming monads was still very difficult when we started our research.

Today, because of a better understanding of monads and more syntactic sugar, this certainly has become a reasonable option. We didn't explore this option any further because when we revert to monads, we may as well use monads for the state updating of our objects. This would fundamentally change Visto. We however prefered to continue our initial path to be able to make a firm conclusion about a functional GUI object system without monads.[10]

4.9 Three Views on one Clock

Assembling the previous clock related examples, adding a digital *clone* of the *AnalogueClock* and making an *expansion* for a versatile clock that can display the time in digital mode, in English and in Dutch and that can switch between these modes, we get 52 lines of Visto source code resulting in the screen shot in fig. 4.9. The source code can be found in the appendix of this chapter.

4.10 Future Extensions

The Visto framework is now firmly established, but that doesn't mean that all work is done. It *does* mean that adding extensions should be relatively easy. Such extensions are indeed necessary. Visto currently only implements a limited set of invokers and selectors. Particularly *direct manipulation* is missing from Visto. A good example is *drag and drop*. To print a document, the user grabs the document and drops it on the printer icon; to remove it, he drags it to the trashcan. Many modern programs use direct manipulation, e.g. to resize a rectangle by grabbing one of its corners.

[10]Although we use monads in the implementation of Visto, they are not visible to the user of Visto. That is what we mean with a system without monads.

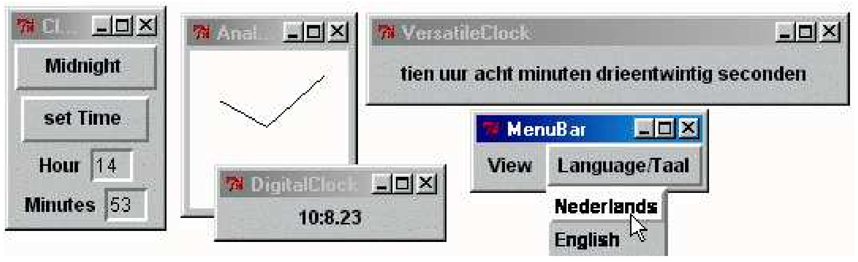


Figure 4.9: A Clock with different Views.

We started with the more simple widgets and our selectors, simply because they are easier to implement, and because they allow us to demonstrate that our selectors are indeed more easy to use. However, in the same way as our other invokers and selectors, more advanced features can be added to Visto.

Currently we have a distinct division between invokers and selectors. Selectors are those components that *select* data, such as *selectOneFromDiscrete*. Invokers are used to trigger the object method. When an invoker, such as the button or the pull down menu, is activated, the arguments are fetched from the selectors and fed to the method that then is executed.

Sometimes this should happen sooner. Consider for example a form of a web document. We can fill in all the entries and press the *Submit* button, but often it suffices to press enter in some field to submit the entire form. The selector functions as an invoker as well. Adding this to Visto should not be difficult. When the user enters a value in the widgets that constitute the selectors, an event is triggered. Currently we use that event only to check whether the user entered a valid value. To use selectors as invokers, we must react on those events, not only to check the validity of the entry, but also to trigger the method.

A more dynamic feature of user interfaces is the popping up of windows, e.g. when the *Print* option is selected, a window pops up to let the user specify options for the print job.

This doesn't change the core behaviour of the selector, *selecting* something. It only influences the visual presentation of the selector. A good place to make this decision is when the actual layout is defined, which occurs in the interface builder. Because the interface builder hasn't been implemented yet, the popup behaviour isn't present yet as well. We will however adequately add it when we implement the interface builder.

More difficult is *direct manipulation*. This is now typically implemented using call back functions attached to the mouse (or to the event of clicking a mouse button). The call back function can then check *where* the mouse was clicked, if the button has been released again (if not a *drag* is going on) and how many times the button has been clicked (as single and double or triple clicks make a great difference). It is easy as the call back function contains all the information about the mouse events, but unfortunately such a call back function will rarely be declarative because it contains many decisions and may call many different actions depending on how and where the mouse was used.

So we should definitely try to provide more declarative definitions for the actions behind a direct manipulation interface. We haven't done a lot of research in that direction yet, but we can already differentiate the following actions. We discuss these actions along with some problems that still have to be solved. However, regardless of those problems and of the actions that we discuss, the following direct manipulations are often no more than a particular *style* of selecting some data. E.g. when we drag a document to the printer icon, we basically select a document that must be printed. We can first define a method with as parameter the document that must be printed, and add an invoker that selects a document. This selector can then be instantiated to a file requester, a button row, or a drag and drop interface.

So we can consider these direct manipulations as no more than style arguments to our selectors and not an entirely new, special case that we are unable to deal with.

Also, remember that our idea of selectors and invokers is in the first place a more declarative description of the connection between GUI and application, to let the designer think in a more structured way about the functionality of his GUI. The examples of instantiations of our selectors should not be considered an extensive enumeration of all possibilities. They can be useful to adhere to a standardised look-and-feel, but should not limit the designer's intellectual freedom nor stop innovation.

Hence, the main issue of this section is showing that the idea of using selectors and invokers is not limited to the actual appearances we – incidently – implemented.

- **MouseClick.**

  This a of course a very familiar action. In this case we are not interested in mouse clicks to activate an edit field or a window, as they are already handled by those widgets themselves, but more in mouse clicks that are used to select data, e.g. a click on a map to select a city, or on a graphic window to select the starting point of a line to be drawn.

  Such mouse clicks select data and thus must be added to our selectors. We are not yet sure how to do this precisely. We can define selectors *MouseClickX* and *MouseClickY* that return the x- and y-coordinate of the position of the mouse click. An alternative is to let the programmer define specific areas of interest and associate a value with those areas. The MouseClick selector will then return the value of the area that is clicked. Or using a name for the area of interest we can define a selector and/or invoker *MouseClickOn* that takes as an argument the area, and that is triggered when a click occurs on that area.

- **MouseSelection.**

  Likewise, the mouse selection deals with the selection of typically a rectangular area. This area can then contain the objects that are to be grouped together in a desktop publishing application (DTP), or the text that is to be deleted, ... The same issues arise here as with *MouseClick*.

- **IconDrag** and **IconSelect.**

  This is a selector of a somewhat other kind. The two previous actions had a strong focus on the coordinates of where the mouse was. It may even be possible that these coordinates will remain present in the selector. However with the *IconDrag* we are not interested in the coordinates but in the *object* (or rather its *icon*) that is dragged or selected.

  This action can both serve as an invoker, e.g. to print a file when its icon is dragged to the printer icon, or as a selector, e.g. a graphical user interface for *tar*[11] or compressors as *lha* or *zip* may open a window in which users can drop files to be compressed. The IconDrag *selects* the files to be tar-ed or compressed and afterwards an invoker *invokes* the program.

4.11 Comparison with other work

Visto is different from other functional GUI systems because it implements an object system *on top of* Haskell to define the user interface instead of a library *in* Haskell like most other systems.

Because the program flow in a GUI is heavily *event* controlled, GUI systems must provide flexible means for reacting on those events. The easiest way would be that we can simply state which *action* must be executed when a particular event occurs. This action can be a procedure, a method, a function, ... , that changes the state of the program, or part of it. The problem is then the management of that program state, taking care of the different actions invoked by the long sequence of events. This is particularly difficult in functional programming, where we cannot have a set of variables that is (destructively) updated whenever an event occurs. The solutions adopted by the various functional GUI systems are shown when we discuss those systems, but unlike those systems we prefer an object system.

[11]tar is a Unix program that was originally intended for writing files to a *ta*pe drive, but is now generally used to created a big file containing many (smaller) files, which is then often compressed using gzip.

The state of our program is then contained in those objects. Because an object can only change its state by evaluating its methods, the methods constitute the actions that are allowed on the objects.

When an event occurs, we can now indicate which method must be executed. The management of the program state is then simply the management of an object program, which we find easier to use, and more flexible than the solutions of the other functional GUI systems. Still, functions are at the basis of our methods and throughout we try to keep a functional *look and feel*.

So although parts of the Visto philosophy, such as the selectors, can and have been implemented for use in a functional language (in Fudgets for Haskell, and in TkGofer), the object system is a vital feature for the enhanced declarativity.

Although we define a programming style outside of Haskell, this is no real objection, because all systems impose their own programming style. In the following subsections we compare the five main GUI systems for functional languages with Visto. Although there exist other GUI systems, most of them are derived from these five 'ancestors' and inherit most fundamental features.

4.11.1 Fudgets

This trend setting GUI system for Haskell is centred around the fudget [16], a stream processor with a separate input and output stream. The library contains fudgets for basic widgets as buttons and menus, but also abstract ones without visual form for the definition of objects with local state. The complete user interface is built by repeatedly applying fudget combinators.

These combinators take care of the layout and the communication between fudgets by connecting the streams of the composing fudgets. This communication is rather inflexible: a fudget can only receive messages from the fudget connected on its input stream and send messages to the one on its output stream. Messages between distant fudgets have to travel through many unrelated fudgets, inducing increased complexity and reduced flexibility. Although Gadgets [57] tries to solve this by adding wires, it doesn't change the system fundamentally. As in Visto an object can call any method of another object, our communication is a lot more flexible and doesn't have to pass through fixed channels or wires.

The structuring of fudgets relies on the X-Windows hierarchy and this shows. The final user interface definition corresponds more to this X-Windows hierarchy than to the logical structure of the program on which Visto, through its object orientation, really focuses.

Fudgets have proven to be very nice for assembling smaller user interfaces but they do not scale up as nicely for larger ones. We believe that Visto can do a better job in that area, because our objects can be built independently from the visual structure of the user interface, and because we have a more flexible communication scheme.

4.11.2 Haggis

Haggis [24] takes a completely different view on user interface building. The interactive components (such as the widgets) are viewed as *virtual I/O devices*. Haggis provides handles on such I/O devices and functions on handles to control their behaviour, such as specifying what to do on button clicks. They are embedded in IO monads to express the desired side effecting operations.

Visto replaces part of these handles and their functions by associating methods with GUI-invokers, defining what to do when the invokers are selected.

Haggis provides separate *display handles* to program the layout of the user interface in Haggis. Just like those display handles, our visual tool for specifying the layout of the GUI will contain references to each of the used GUI-invokers and selectors, but the user interface will be drawn in this interactive tool instead of being programmed.

4.11.3 Clean

In version 0.8 of its I/O library Clean [1] defined widgets by functions that take as one of their arguments a call back function. Instead of monads Clean uses the environment passing style. Essentially the entire world is passed around, although in Clean this is divided in independent parts. The call back function not only transforms the world, but also a global state. The authors seemed to be well aware of the disadvantage of this single global state. In version 1.3 of their new Object I/O library, which is a major rework, each widget can contain a local state. And because of the environment passing style used in Clean I/O the local state also contains a component *IOSt* to accommodate for a local process state. A Clean program can have different processes running at the same time and therefore there may be different local process states flowing through the program and its widgets.

Sharing information between a number of widgets used to be only possible through some *global* state, but now one should rather use *Receivers* to communicate in a message passing style. Because of the way in which Visto uses objects to define logical entities rather than user interface elements, this communication is a lot more flexible in Visto as we don't need explicit *Receivers*.

Both Clean's IO library 0.8 and the new 1.3 Object I/O library use the typical call back functions that configure the different user interface elements. This is different from Visto which defines data and behaviour in objects to which widgets can be added.

Apart from all the differences between Clean and Visto, there is also a common factor. The Visto draw primitives are strongly inspired by and reminiscent of those present in Clean.

4.11.4 TkGofer

TkGofer [88] offers typed versions of Tk/Tcl primitives, embedded in Gofer. As in Clean, call back functions are added to widgets. Because these are monadic functions and can contain any side effecting operation, no global variable as in Clean is needed.

This is good because it gives a high degree of freedom, but it also allows for badly structured user interfaces. It is suitable for using and defining loosely coupled components but provides no methodology for creating larger user interfaces. As we leave the path of the typical call back functions, and we use objects and their methods, Visto gives more support for a better structuring of the functionality of the user interface – which however doesn't mean that we cannot avoid badly structured interfaces.

We provide a number of predefined Visto selectors, invokers and widgets, but also leave a way through to TkGofer for experienced users.

4.11.5 FranTk

FranTk[63] builds on TclHaskell[22] and this is visible in the way it configures its widgets. Both supply a list of configurations to the constructor for a widget. Interesting is the fact that FranTk has configuration options that implement dynamic behaviour. E.g. the configuration option *textB* takes a *Behavior String*. The widget with this option will be redrawn whenever the Behavior changes state. This is a bit like Visto's *draw* method that is performed whenever the object receives a message, but more efficient as FranTk doesn't have to watch each method, just the Behavior.

This *Behavior a* data type is an essential feature of FranTk. The system uses mutable variables of type *BVar a* to take care of local state. Each *BVar a* can also be referred to by means of three other data types: the already mentioned *Behavior a*, *Event a* and *Listener a*. They roughly correspond to a container of data, a producer and a consumer of events. E.g. an entry contains data and can be examined in that way, or produces events when data has been entered and consumes events as well when other data is inserted by the program.

The flow of the events can then be controlled by attaching Behaviors, Events and Listeners to the correct widgets. Along with quite some extra primitives (such as *ifB* and *listB* that define behaviour familiar in functional programming but reprogrammed for user interface widgets) and new data types (such as the *Wire a* that connects widgets and/or event streams) a complex user interface can be built.

This is clearly an original approach with strong benefits because of the distinction between containers, consumers and producers. But because these variables can and will appear in many different places, the control over and the behaviour of some data (present in a mutable variable *BVar a*) is largely distributed across the program, quite contrary to Visto where the objects centralise the data and behaviour. The object's state conforms to FranTk's Behavior and the methods are consumers of events, whereas the GUI-invokers produce events, but then centralised around some data.

4.11.6 Hanus' System

Next, in his research on a GUI for Curry[31], M. Hanus [32] builds, just like TkGofer and TclHaskell, a library on top of Tcl/Tk[60]. Therefore the definition of the user interface shares a lot of similarities with those systems. It also has one similarity with our approach in that it uses objects that can accept a number of different messages, but their object system behaves a bit like Haskell's old I/O-system with lazy lists as the list of all the messages an object will accept. There is few state hiding, but quite unique is the way in which results of method invocations are passed. This approach uses logical variables that will be instantiated when the object finishes processing the request. This is quite different from the way Visto passes the result in a typical functional way as a function returning a value.

So, after all, because (logical) values are destructively updated and the typical call back functions are used to define behaviour for the widgets, this system has a rather imperative feel.

4.11.7 Garnet

Finally, we take a closer look at Garnet [52] of Brad A. Myers e.a. We already discussed it in chapter 3 about Visto's object system because of its prototype based character, but as it is intended, just as Visto, to be used for the development of user interfaces, we have to discuss it here again.

It shares an important part of the features of Visto, but it often uses them in different ways.

- The largest similar feature is indeed the *prototype based object orientation*. Although some critiques on our research question the use of prototypes, its successful application in Garnet proves the contrary – over 50 projects all over the world use or have used Garnet. Note however that Visto's and Garnet's object system differ in important ways (sec. 3.12.1).

- In both systems the *drawing mechanism* is largely automated. In Visto's initial model all objects were to be drawn on the screen, but this changed throughout the history of Visto. In Garnet *all* objects can be drawn on the screen, but this only happens when they are added to a Garnet window [12]. Consequently, to make it disappear, the Garnet object must be removed from the window.

[12] Actually, Garnet's graphical object system is called Opal, but for simplicity we named it Garnet here.

But, just as in Visto, the programmer never has to call the object's draw method directly. If any property of any object changes, Garnet automatically refreshes the screen and redraws the updated object. The difference with Visto is that Garnet reacts on changes of properties, whereas Visto reacts on method invocations, which is the only means for changing a property in Visto.

- Garnet uses *interactors* to encapsulate input device behaviors, such as menu's, dragging of objects through the mouse etc. In that way they can be considered a bit like our selectors as they can be very easily applied on different objects, e.g. only three lines of code are needed to select from a list of objects using a menu interactor.

  However, although the interactors are capable of handling different looks-and-feels, they still largely focus on their appearance (e.g. a menu) and less on a more general intention of selecting some data.

- *Functional based.* Finally, both Visto and Garnet have a functional underlying system. Visto uses TkGofer as its implementation language, Garnet uses Common Lisp. However, apart from the abundant presence of parentheses, this is not prominently visible in Garnet. Furthermore, constraints are used to define the behaviour of the objects as already discussed in the previous chapter.

It is striking that although Visto and Garnet have been developed completely independently from each other[13], they resemble each other so much precisely on two of Visto's innovative and at the same time controversial features, i.e. the prototype based OO and the automatic *draw* method.

This pleases us as it shows that our approach is supported by others, but it also makes our approach a bit less original.

As a final note, it is striking to see that with Visto – functional –, Hanus – logic – and Garnet – constraints – the three main directions in the declarative programming community are also present in user interface design.

4.11.8 Comparison with Model-Based Tools

During the active phase of our research we didn't look at and consequently haven't been inspired by model based tools [78], of which a wide range exists, e.g. UIDE [27], MASTERMIND, [79], Mobi-D [64], ACE [41] and TRIDENT [10, 11]. It now turns out that we have used some ideas present in model based tools as well, especially the focus on the user actions. Our Visto objects can be considered an implementation of the application or task model.

[13]Garnet was there before us and we became only aware of Garnet in the final stages of our research.

Visto certainly is not as powerful as modern model-based systems, who can derive a user interface from the models that have been defined. In Visto we still have to program the user interface, although this programming is being done on a very high, declarative level with relatively little work.

On the other hand defining the models in most model based systems takes a lot of work. Depending on the tool, not only the actions and interface components must be defined, but also pre- and post-conditions, operational constraints, entity-relationship-attribute diagrams, activity chaining graphs, ... Acquiring this large amount of information may be a large investment, but it is often worthwhile. Not only can the user interface be generated automatically from that information, but many tools support in the same effort automatic help generation and adaptive user interfaces that can refine themselves by performing analyses on the user action sequences that have been tracked by the system.

All of these facets are not present in Visto, but programming a user interface doesn't require as much information as in complex model-based tools. That is certainly an advantage.

Another aspect is that the generated code of many model-based tools is far from declarative. The generated UI code still relies on the call back functions and elementary widgets.

If we consider the Visto objects as *an implementation* of a task model, this opens some interesting perspectives. Visto could then be used as (a basis for) an implementation language for model-based tools, such that the generated code from such tools retains its declarative character.

However, this may only be a theoretical advantage. In the past, it was more the case that model based tools didn't generate flexible code. The generated UI was very standard and rather inflexible. So if one wanted a more professional looking interface, one had to manually recode the generated code. Nowadays the development of the professional UI can be completed by staying within the model based tool all the time. The generated code is only needed to have an executable file. One is not expected to refine the code. Therefore, even if we were to extend Visto such that it can be used as an implementation language for model-based tools, the benefit would be small as only very few people would take a look at the code (and see the beautiful Visto code).

4.12 Conclusions

The Visto methodology of concentrating on the visual objects, defining methods for manipulating that (displayed) information and supplying GUI-invokers for the methods opens a brand new perspective on user interface design. Especially the move from call back functions to GUI-invokers and selectors is an important shift. The automatic application of the **draw** method and the **expand** derivation for defining views are other innovations.

The design order of Visto – first objects and their methods, then the GUI invokers – fulfills the first two of our research goals. In a proper way the (potential) *functionality* of the user interface is described, and only when we actually need a method *in* the user interface, we add an invoker, which neatly describes how and when a method is triggered in the user interface, also because the GUI for the method is nicely concentrated in the invokers. Also, only user interface elements with a well-known goal (the method to invoke) are present in the interface. Useless elements cannot be present.

Only the invoker definitions and/or the layout have to be changed to change the user interface. The rest of the Visto objects can stay as they are, which meets the other research goals we outlined in the beginning of this chapter.

Nevertheless our work is not finished. More invokers and selectors should be considered as well as some more primitives for fine tuning the look of the user interface. Besides those enhancements, the user interface builder will be another milestone in the development of the final Visto system.

A more fundamental conclusion may be that a GUI needs so many dynamic features with side effects that the absence of monadic style in the Visto method definitions is felt more strongly here than when simply defining an object system as in chapter 3.

Of course the programmer can still use monadic style in his method definitions, but Visto doesn't provide special support for monads. It supposes standard functional style.

If reverting to monadic style in the method definitions by default would facilitate the definition of advanced GUI's, we may as well use the opportunity to its full potential and build a object system with monadic state updating, resulting in a very different Visto, suited for more dynamic interfaces. However, this will rather be the subject of a continuation of this thesis.

This doesn't mean that current Visto is useless. A large set of interfaces can be easily defined in Visto, especially those interfaces that don't need many ultra-dynamic features. Such interfaces typically need specific support from the application layer, e.g. to be able to show a progress indicator, or zoom functions. As a matter of fact, one of Visto's aims is facilitating the implementation of a user interface for existing Haskell programs. Those programs haven't been developed with an advanced GUI in mind and don't contain such GUI-related functions as showing intermediate results and displaying progress bars. Without a major redesign, they are only suited for more basic interfaces, precisely what Visto is already capable of. In that sense does Visto fulfill its goals.

4.13 Appendix - Source Code

This appendix reveals the source code for the example in figure 6. The count of 52 lines excludes Haskell code for the data type *Time* and translations. The described system only displays a time. It is no real clock that automatically advances every second.

```
interface Clock
  getTime :: Time
  setTime :: Int → Int → Int → Ordering
object ClockModel Clock Time (Time 10 8 23) {
  getTime = (state, state),
  setTime = λ h m s → let
                         t = makeTime h m s
                       in
                       (compare t state, t)
}


clone ClockModifier from ClockModel {
  setTime = λ u m s → let
                         t' = ClockModel!setTime u m s
                       in
                       (t', state),
  GUIsetTime =Button "setTime"
                       <<− SelectOneFromRange "Hour" 0 23
                       <<− SelectOneFromRange "Minutes" 0 59
                       <<− 0;
             Button "Midnight" <<− 0 <<− 0 <<− 0
}


expandInterface ClockView from Clock
  _setTime


expand AnalogueClock ClockView
      String "green" from ClockModel {
  draw = Canvas (100, 100)
         clear
         let time = superState
         oval (0, 0) (100, 100)
         setColor state
         line (50, 50) (hourCoordinate (timeHour time))
         line (50, 50) (minuteCoordinate (timeMin time))
}
```

```
clone DigitalClock from AnalogueClock {
  draw  = Label
            let   time  =  superState
                  h  =  show (timeHour time)
                  m  =  show (timeMin time)
                  s  =  show (timeSec time)
            text  (h ++ ": " ++ m ++ "." ++ s)
}


adaptInterface MultiClock from ClockView
  setLanguage :: Language − > Time
  setDigital :: Time
  setTextual :: Time


adapt VersatileClock MultiClock
     (Language, Bool) (NL, True) from AnalogueClock {
  setLanguage  =  λ l → let
                              newstate  =  (l, False)
                            in
                            (superState, newstate),
  setDigital  =  (superState, (fst state, True)),
  setTextual  =  (superState, (fst state, False)),
  draw =Label
           let   time = superState
                 h = show(timeHourtime)
                 m = show(timeMintime)
                 s = show(timeSectime)
                 (lang, digital) = state
                 digitalString = h ++ ": " ++ m ++ "." ++ s
                 string =  if digital then
                              digitalString
                           else
                              translateTimetimelang
           text  string,
  GUIsetDigital = Menu ("View", 1) "digital display",
  GUIsetTextual = Menu ("View", 2) "textual display",
  GUIsetLanguage =Menu ("Language/Taal", 1) "Nederlands"
                               <<− Nederlands;
                          Menu ("Language/Taal", 2) "English"
                               <<− English
}
```

Chapter 5

A Case Study

5.1 The Case

In this case study we implement an example in five different systems including Visto. This test will then show that some typical changes to the interface are easier to implement, i.e. requiring less changes to the interface and application code, in Visto than in the various other systems.

As can be expected, we have chosen this case such that it can be easily implemented in Visto. The aim is not to advantage Visto, but rather to test if Visto succeeds in what it is aimed at. Therefore the case is an example of an application in which the user selects data on which the application performs a calculation and displays the results. Although many more dynamic interfaces exist, this sketch is still typical for a very large set of GUIs.

Interesting is the definition of the initial interface, but also the various changes to the code as the interface grows more complex.

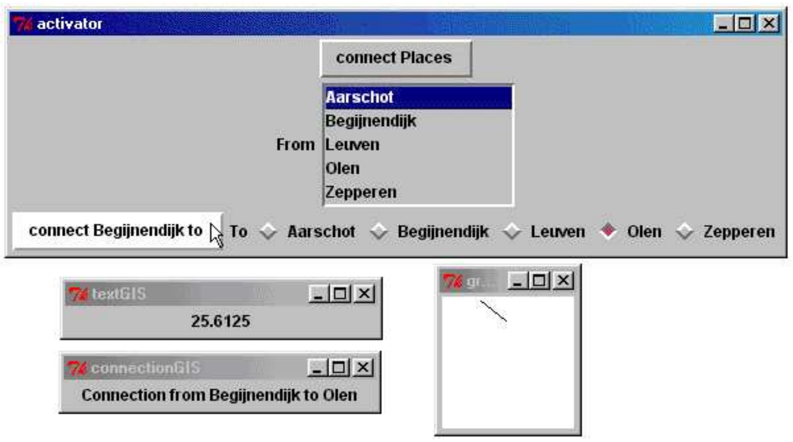
The case is a largely simplified *Geographical Information System*. In such an application, a user is guided from one to the other location. In this simplification the user selects two locations from a list of known places and the application then gives small bits of information about the connection.

So we have three components:

- the *interface* that lets the user select places and connect them,
- the *application value* that contains the connection,
- the *views* that display information about the application value.

In fig. 5.1 the final version of the Visto application is shown. Below the activator window we have three views: a graphical view on the right that draws a straight line between the two locations, and on the left two textual views that show the distance and the names of the locations.

In the activator window we can use the *connect Places*-button to connect the *From* location in the listbox with the *To* location in the button row. The *connect Begijnendijk to*-button connects Begijnendijk to the *To* location.



25.6125

Connection from Begijnendijk to Olen

Figure 5.1: The Final Version in Visto.

In the initial interface the user can only use the *connect Places*-button, and select from only three locations. Both the *from* and the *to* location are selected by a set of radio buttons. The connection is established when the user clicks the button.

This is gradually extended.

- First, another location is added to the selection.
- Then, the *from* location is to be selected from a list box. This is often preferable over a set of radio buttons when the set of choices gets larger. Hence, this represents a typical modification to a user interface. Nevertheless it doesn't substantially change the behaviour of the user interface (selecting a location), so the modification should minimally affect the rest of the application.
- Then another view is added. The view we actually add is very simple (we just mention the selected connection), but it sufficiently demonstrates the changes needed for adding a view to the application. Although the actual implementation of a view may be fundamentally different from this example, the way in which to add a view to a model remains mostly identical.

- Finally, we provide a second way to invoke the method connect. As a version of this application is aimed at inhabitants of Begijnendijk, we add a button that by default connects the *to* location to Begijnendijk.

We first implement this application in Visto. With the experience of the previous chapters its simplicity should be self explaining.

Next we use Tcl/Tk [60], a powerful script language that is still widely used. Although Visto indirectly builds on this toolkit through TkGofer[88], there appears to be no simple mapping between the Visto program and its Tcl/Tk counterpart.

We also implement this simple case in Java, a recent and much hyped member of the family of class based object oriented languages.

Of course, the functional community is visited as well. Both an implementation in Clean[1] and Fudgets[17] follows, after which we conclude this chapter.

5.2 Visto

5.2.1 The Basic Application

The Visto implementation reflects the three distinct tasks of a user interface:

1. Application objects that keep the application values and have methods for modifying those values (the *models*).
2. GUI objects that let the user invoke methods (the *activators*).
3. Objects that maintain a view on the application values (the *views*).

The two latter objects are implemented as *expansions* of the model object.

For the first implementation of the GIS application (fig. 5.2) we have one model *gisser*, one activator *activator*, and two views *textGIS* and *graphical*. *Graphical* is an *adaptation* of *textGIS* as it has an additional method for setting the *colour* in which the connection between the two places can be drawn.

Haskell code. As Visto can be used to develop a user interface on top of an existing Haskell program, we first have a bit of Haskell code [1] that contains a data type *Place* and a function to calculate the distance.

```
data Place = Place String (Int, Int)
place (Place x y) = x
coords (Place x y) = y
```

[1]As we use TkGofer as our implementation language, this is actually Gofer code. However, as almost all of Gofer is identical to current Haskell, and Gofer itself is no longer supported, we prefer to refer to this code as *Haskell code*. Haskell itself has some extra features that allow for an even more simple implementation of the *Place* data type and its associated functions.

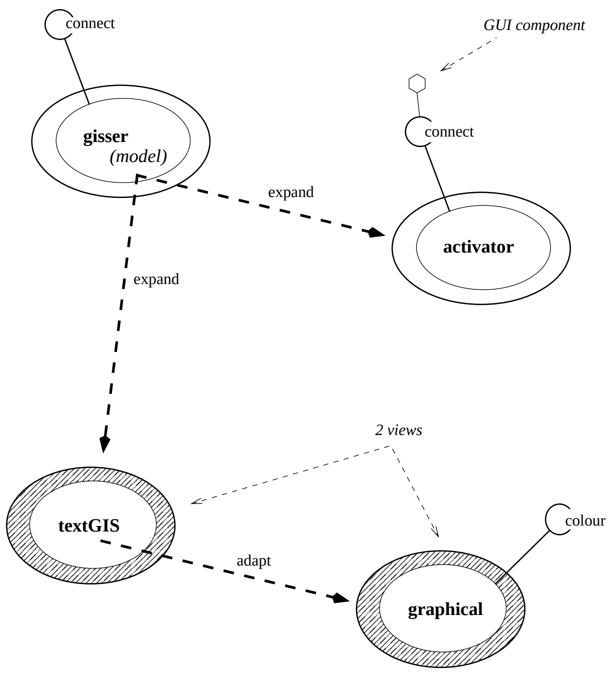


Figure 5.2: The Design in Visto.

$$
\begin{aligned}
&\mathit{distance}\ \mathit{place}1\ \mathit{place}2\ =\ \textbf{let}\\
&\qquad (x1, y1)\ =\ \mathit{coords}\ \mathit{place}1\\
&\qquad (x2, y2)\ =\ \mathit{coords}\ \mathit{place}2\\
&\qquad \mathit{xDist}\ =\ \mathit{fromInteger}\ (\mathit{sqr}\ (x2\ -\ x1))\\
&\qquad \mathit{yDist}\ =\ \mathit{fromInteger}\ (\mathit{sqr}\ (y2\ -\ y1))\\
&\quad \textbf{in}\\
&\quad \mathit{sqrt}\ (\mathit{xDist}\ +\ \mathit{yDist})\\
&\mathit{aarschot}\ =\ \mathit{Place}\ \text{“}\mathit{Aarschot}\text{”}\ (50, 50)\\
&\mathit{begijnendijk}\ =\ \mathit{Place}\ \text{“}\mathit{Begijnendijk}\text{”}\ (40, 42)\\
&\mathit{leuven}\ =\ \mathit{Place}\ \text{“}\mathit{Leuven}\text{”}\ (20, 90)
\end{aligned}
$$

The application object. This is only a small object. From a user's point of view, we can only connect two locations to each other. Therefore the interface *GIS* for the application object *gisser* only contains a *connect* method. It returns an *Int*, the approximation of the distance between the two places.

The state type of *gisser* is *Maybe* (*Place*, *Place*), which is either *Nothing* when no connection has been selected or *Just* (*origin*, *destination*), the tuple of selected places.

```
interface GIS
  connect :: Place → Place → Int

object gisser GIS (Maybe (Place, Place)) Nothing {
  connect = λ origin destin → let
                                dist = truncate (distance origin destin)
                              in
                              (dist, (Just (origin, destin))) }
```

The GUI object. Of course this object *gisser* must be triggered from the user interface. We therefore need another object that contains an activator for the *gisser* object. Such an object must be an *expansion*.

```
expandInterface GISAction from GIS
  —No new methods are defi ned here.

expand activator GISAction Int 0 from gisser {
  GUIconnect = Button "connect Places"
             << − SelectOneFromDiscrete [(aarschot, , "Aarschot"),
                                         (begijnendijk, "Begijnendijk"),
                                         (leuven, "Leuven")]
             << − SelectOneFromDiscrete [(aarschot, , "Aarschot"),
                                         (begijnendijk, "Begijnendijk"),
                                         (leuven, "Leuven")]
}
```

The definition **GUI***connect* explains precisely that to connect two locations, the user must press the *connect places*-button and select *one* location from the *discrete* set {*Aarschot*, *Begijnendijk*, *Leuven* } (i.e. *SelectOneFromDiscrete*) both for the *from* and *to* location.

The views. Finally we need two views on the data present in the *gisser* object. The first, *textGIS* is a view that displays the distance on a *label*, the second one, *graphical* is a graphical view that draws a straight line between the two places. Its interface contains an additional method to set the colour in which the line should be drawn.

```
expandInterface GISView from GIS
  _connect
```

```
expand textGIS GISView Int 0 from gisser {
  draw = Label
         let   (place1, place2) = fromJust superState
               distText = if isJust superState then
                              show (distance place1 place2)
                            else
                              "No places selected"
         text  distText}
adaptInterface GISColour from GISView
  colour :: Colour → Colour

adapt graphical GISColour Colour "black" from textGIS {
  draw = Canvas (100, 100)
         let  (place1, place2) = fromJust superState
              color = if isJust superState then
                        state
                      else
                        "white"
              (coords1, coords2) = if isJust superState then
                                     (coords place1, coords place2)
                                   else
                                     ((0, 0), (0, 0))
         line coords1 coords2,
  colour = λcolour → (state, colour)}
```

The three distinct parts of the application – the application object, the GUI activation and the views – are nicely separated. The application object *gisser* doesn't have to know – and doesn't know – what GUI object activates it, and which views display its information.

What we have hidden here is that we instantiated the **SelectOneFromDiscrete** as a set of radio buttons. In a final version of Visto this should be done in the GUI builder, but now we did it in the translated code in TkGofer.

5.2.2 Adding Another Place

This is very easy. We have to define the new place and add it to the selector in the GUI object *activator*.

```
olen = Place "Olen" (30, 4)

allPlaces = [ (aarschot, "Aarschot"), (begijnendijk, "Begijnendijk"),
              (leuven, "Leuven"), (olen, "Olen")]
```

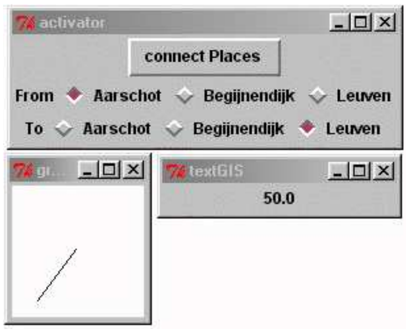


Figure 5.3: Screen shot of Visto program.

```
expand activator GISAction Int 0 from gisser {
  GUIconnect  = Button “connect Places”
                          << − SelectOneFromDiscrete allPlaces
                          << − SelectOneFromDiscrete allPlaces
  }
```

5.2.3 Moving from a Radio Button to a List Box

Almost as easy. We add another place (*Zepperen*) and instantiate the *from* selector differently. This must be done in the GUI builder when that part of Visto is finished, but currently was done in the translated TkGofer code by hand, requiring only changes to one line of code.

5.2.4 Adding a View

To add a view to some model, we must only create another view that is an expansion of the model. We therefore don’t have to change some controller object or the model itself, unlike traditional MVC where the controller must know the views.

In this case we can re-use the existing object *textGIS* to create another view using the **clone** construct.

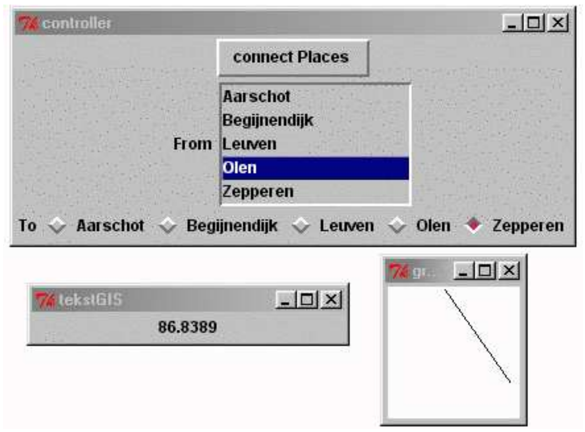


Figure 5.4: Using a list box.

```
clone connectionGIS from textGIS {
  draw = Label
         let  (p1, p2) = fromJust superState
              distText = if isJust superState then
                           "Connection from " ++ (place p1)
                           ++ "to " ++ (place p2)
                         else
                           "No places selected"
         text  distText
  }
```

5.2.5 A Short Cut for the Begijnendijk Connection

As we add an invoker for the method *connect*, we return to the *activator* object. The second selector is reused in the new invoker. Therefore we must give it a name, "*to*" and use that named selector as the second argument of the second invoker. The first argument of that invoker is *begijnendijk*.

This implementation nicely reflects the fact that the method *connect* can be activated in the GUI in two different ways.

```
expand controller GIScontrol Int 0 from gisser {
  GUIconnect = Button "connect Places"
                  <<− SelectOneFromDiscrete [(aarschot, , "Aarschot"),
                                             (begijnendijk, "Begijnendijk"),
                                             (leuven, "Leuven")]
                  <<− SelectOneFromDiscrete′ "to"
                     —As this selector is reused, it must get a name.
                                            [(aarschot, , "Aarschot"),
                                             (begijnendijk, "Begijnendijk"),
                                             (leuven, "Leuven")];
                 Button "connect Places"
                  <<− begijnendijk
                  <<− NamedSelector "to" }
```

5.3 Tcl/Tk

Visto is an object oriented language and we could therefore use nice data structures. Tcl/Tk however is a rather simple script language that basically stores *all* its values as strings. Our Tcl/Tk implementation therefore remains quite simple, but it works and shows interesting features typical for the family of basic script languages. One of the problems is the fact that the widgets only return strings as their selected values. We therefore often need functions mapping those strings on the actual application values that we're interested in. In Tcl/Tk associative arrays make this task a lot easier. We can index such arrays on strings, and thus on the values that the widgets return.

Although most typical imperative languages have a much richer type system than Tcl/Tk, most GUI toolkits for imperative languages still only return either a string or an (index) integer as the selected widget value. This is probably because they offer little or no support for polymorphism. We can thus safely consider this example as prototypical for the larger family of imperative languages.

5.3.1 The Basic Application

The activator window. Let's first take a look at the window that activates the application and in which we can connect the two locations. We start with the radio button that is used to select the *from* location.

```
label .fromFr.from −text "From : "
set from Aarschot
radiobutton .fromFr.aarschot −text Aarschot −variable from −value Aarschot
radiobutton .fromFr.begijnendijk −text Begijnendijk −variable from −value Begijnendijk
radiobutton .fromFr.leuven −text Leuven −variable from −value Leuven
```

```
pack .fromFr.from .fromFr.aarschot .fromFr.begijnendijk
.fromFr.leuven −side left −in .fromFr
```

We have to define separate radio buttons for each choice. When these individual buttons are all associated with the same variable, they behave as a radio group of which only one choice can be selected simultaneously. For the *from* place we use the *from* variable. The definition for the *to* place uses a *to* variable in an identical way.

Also, notice that Tcl/Tk only uses strings. Both the *text* on the radio button and the *value* that is placed in the variable *from* when that button is selected, are of type String.

Layout is done using *pack* commands. First we pack the radio buttons into a *fromFr* frame, which is packed again into a top level window, along with frames for the selection of the *to* place and a frame that contains the button to connect the two places.

The pack command given here is only an example. The other pack commands are left out because we are not interested in the matter of layout here.

```
button .connectFr.connect −text Connect −command connectCommand
```

The definition of the *Connect* button is largely similar, but it needs a *call back function* to make it really functional. This call back function has to retrieve the values that were selected in the radio buttons *and* trigger the views to display the information. We must therefore first know more about the views.

The views. The *distance* view is made easy by using a *label* that is associated with a text variable. Whenever the variable's state changes, the label is refreshed to reflect the state change.

The *graphical* view is a canvas on which we can draw lines and other geographical figures using Tk commands.

```
#the distance view :

toplevel .viewDist
set distance 0
label .viewDist.distance −textvariable distance
pack .viewDist.distance −in .viewDist


#the graphical view :

toplevel .viewGraphic
canvas .viewGraphic.c −width 100 −height 100
pack .viewGraphic.c −in .viewGraphic
```

The call back function. We first need to define some application data. More specifically, we need an associative array here that maps the locations on their x and y coordinates.

```
array set xcoords {"Aarschot" 60 "Begijnendijk" 50 "Leuven" 50}
array set ycoords {"Aarschot" 30 "Begijnendijk" 20 "Leuven" 90}

proc connectCommand args {
  global from to xcoords ycoords distance
  set x1 $xcoords($from)
  set y1 $ycoords($from)
  set x2 $xcoords($to)
  set y2 $ycoords($to)
  set distance [expr [calcDistance $from $to]]
  .viewGraphic.c create line $x1 $y1 $x2 $y2
  }
```

In the call back function *connectCommand* we use the values in the *from* and *to* variables as indices in the *xcoords* and *ycoords* associative arrays. We can then update *distance* which refreshes the *distance* view. We also create a line on the *viewGraphic* view.

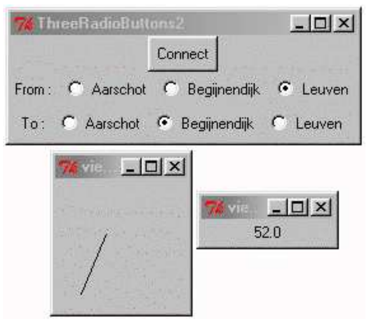

ThreeRadioButtons2
Connect
From : Aarschot Begijnendijk Leuven
To : Aarschot Begijnendijk Leuven
vie...
vie...
52.0

Figure 5.5: Screen shot of Tcl/Tk program.

An essential conceptual difference with Visto is that the call back function describes what to do when the connect button is clicked, whereas the Visto activator says what invoker must be used to activate a particular method. The Visto implementation thus primarily describes a goal, the purpose of the invoker, and the call back mechanism more a behavioral description of what happens when the widget is used.

Another difference is the fact that the call back function must trigger the views. As Visto's views are expansions, they automatically redraw themselves without explicit intervention from a controller object.

5.3.2 Adding Another Place

If we want to add the location *Olen* we need to define another radio button, and adapt the *pack* command for the two radio button frames that let the user select the *from* and *to* location.

```
radiobutton .fromFr.olen −text Olen −variable from −value Olen
```

We must also add a new item to the *xcoords* and *ycoords* associative arrays. We may change the previous array itself as a whole, or simply add an element.

```
array set xcoords {"Olen" 30}
array set ycoords {"Olen" 4}
```

So, because the widgets only return strings and Tcl/Tk has no user defined data types, we must modify the program at two different points to add a new location, i.e. change the user interface and extend the arrays.

5.3.3 Moving from a Radio Button to a List Box

Now things get a lot more complicated. The structure of the radio button group cannot be used for the list box. We must first create an empty list box and add the different items to the list.

```
listbox .toList
.toList insert end "Aarschot"
.toList insert end "Begijnendijk"
.toList insert end "Leuven"
.toList insert end "Olen"
.toList insert end "Zepperen"
```

Contrary to the radio button, a list box is *not* associated with a variable. To retrieve the selected value, one must use the procedure *.toList curselection* which returns a list of the indices of the currently selected list box items. Therefore we must define another array to map that integer index to the name of the location, or define such a procedure.

Neither solution is great, because it relies on the order in which the elements are displayed in the listbox, and because we must duplicate the information present in the listbox: the names of the locations must be exactly replicated, which is error-prone.

```
array set index2Location
        {0 "Aarschot" 1 "Begijnendijk" 2 "Leuven" 3 "Olen" 4 "Zepperen"}

proc getListSelection index {
   if { $index == 0} {
      return "Aarschot" }
   if { $index == 1} {
      return "Begijnendijk" }
   if { $index == 2} {
      return "Leuven"}
   if { $index == 3} {
      return "Olen" }
   if { $index == 4} {
      return "Zepperen" }
   return "Aarschot"
   }
```

At the same time we must change the call back function of the connect button to use the new style of finding the selected location.

```
proc connectCommand args {
   ...
   # new style
   set to [getListSelection [expr [.toList curselection]]]

   # or
   set to $index2Location([.toList curselection])
   ...
   }
```

Although the *what* of the user interface didn't change: we still want the user to *select* a location, the change in the *how* (a list box instead of a radio button list) turns out to be very intrusive, resulting in various program changes, contrary to Visto where the change was minimal.

5.3.4 Adding a View

Finally we add another view on the selected connection. Like the first view, we can use a label with an associated variable that is updated whenever a connection has been selected.

```
toplevel .viewConnect
set connection "no connection yet"
label .viewConnect.connection -textvariable connection
pack .viewConnect.connection -in .viewConnect
```

To do the update, we must change the call back function of the connect button again.

```
proc connectCommand args {
  ...
  # add another global variable
  global connection
  ...
  # new view
  set connection [concat "Connection from " $from " to " $to]
  }
```

This is not a lot of work, but we feel that the Visto approach is more natural. The connect button must be made aware of all the views, although it should do no more than connecting the two places. Why should something be aware of everything that is looking at it? A clock on the bell tower of a church doesn't need to know everyone who is looking at it, does it?

5.3.5 A Short Cut for the Begijnendijk Connection

We must of course add a button, but as the call back function defines the behavioral aspects, we cannot simply reuse the old call back function. Its behaviour is different as we don't need the value from the *from* location for the Begijnendijk connection.

In Visto we only had to change the interface object in a simple way.

```
button .toFrame.begijnen -text "Begijnendijk to : "
      -command connectBegijnenCommand

proc connectBegijnenCommand args {
  global from to xcoords ycoords distance index2Location
  global connection .viewConnect.connection
  # compared to connectCommand, this part has to change
  # we must no longer retrieve the $from variable
  set x1 $xcoords(Begijnendijk)
  set y1 $ycoords(Begijnendijk)
  set to [getListSelection [expr[.toFrame.toListcurselection]]]
  set x2 $xcoords($to)
  set y2 $ycoords($to)
  # hard code Begijnendijk
  set distance [expr [calcDistance "Begijnendijk" $to]]
  .viewGraphic.c create line$x1 $y1 $x2 $y2

  # new view with hard coded begijnendijk
  set connection [concat"Connection from Begijnendijk to " $to]
  }
```

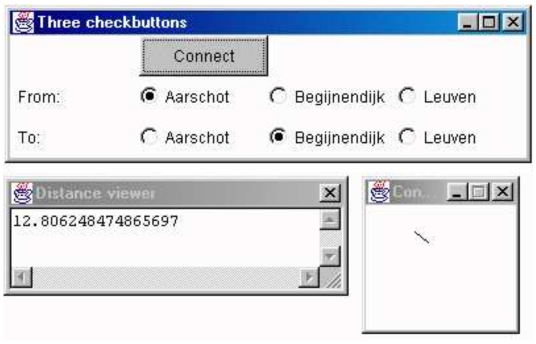
5.4 Java

The comparison of Visto with Tcl/Tk may be considered unfair because Tcl/Tk doesn't claim to be a modern, declarative solution for implementing graphical user interfaces. Java, on the other hand, is a modern programming language that is object oriented, just as Visto, and that currently attracts lots of attention. Can Visto still survive the comparison with Java?

5.4.1 The Basic Application

First, we defined a class **Place** with behaviour like our Haskell data type *Place*. The implementation is so straightforward that we don't include it here. The class **Connection** is as straightforward and contains the two connected places and a method to calculate the distance between the two places.

We also defined a graphical view – the class **ConnectionView** – with a method *void draw(Place from, Place to)* that draws a straight line between the two places, and a textual view using an output box.



```
12.806248474865697
```

Figure 5.6: Screen shot of Java program.

More interesting is the dialogue that lets the user connect two places. It is defined in the class **Venster**. From its definition we don't display the commands related to the layout.

```
import java.awt.*;

public class Venster extends Frame {

  protected Button connectButton;
public Venster(){
    connectButton = new Button ("Connect");
    // the button that connects two places
    add (connectButton);

    add (new Label("From : "));
    CheckboxGroup from = new CheckboxGroup();
    // the checkboxgroup with the 'from' place
    add (new Checkbox("Aarschot", from, true));
    add (new Checkbox("Begijnendijk", from, false));
    add (new Checkbox("Leuven", from, false));
    add (new Label("To : "));

    CheckboxGroup to = new CheckboxGroup();
    add (new Checkbox("Aarschot", to, true));
    add (new Checkbox("Begijnendijk", to, false));
    add (new Checkbox("Leuven", to, false));

    // and then we add the actionListeners
    connectButton.addActionListener(new ConnectionListener(from,to));    }
}
```

Instead of a call back *function* as in Tcl/Tk, Java uses a *ActionListener* object that listens to the events that are created by the button and/or check box groups. This object must have a method *ActionPerformed(ActionEvent event)* that defines the call back behaviour.

Let's take a look at that method in the class **ConnectionListener**.

```
public void actionPerformed (ActionEvent event) {

  String fromWhom = event.getActionCommand();

  if (fromWhom.equals("Connect")) {

      Place fromP, toP;
      fromP = AllPlaces.toPlace(from.getSelectedCheckbox().getLabel());
      toP = AllPlaces.toPlace(to.getSelectedCheckbox().getLabel());
      Connection connection = new Connection(fromP, toP);

      //and now trigger all views
      outputBox.clear();
      outputBox.printLine(connection.distance());
      outputBox.show();

      drawView.draw(fromP,toP);    }
  }
```

We first identify the name of the object that created the event. If it is the connect button, we retrieve the places that are selected in check box groups using the method *getSelectedCheckBox().getLabel()*, create a new connection and trigger all the views. We can also see here that the controller from the MVC pattern really controls the application. It must explicitly trigger the views. This is completely different from Visto, where the views are automatically triggered when the model changes.

Because the method *getLabel()* returns a String, we must define a mapping from the String to the actual object of type Place. Because Java is object oriented we only need a single mapping, contrary to Tcl/Tk where we needed such a mapping (an associative array) for every interesting value that each place contains. As Visto uses the rich type system of Haskell, we don't need any of such functions in Visto, because we can directly select a value of *any* type.

```
public class AllPlaces {

  public static Place aarschot = new Place("Aarschot", 50, 50);
  public static Place begijnendijk = new Place("Begijnendijk", 40, 42);
  public static Place leuven = new Place("Leuven", 20, 90);

  public static Place[] allPlaces = {aarschot, begijnendijk, leuven};

  public static Place toPlace(String name) {
     boolean success = false;
     Place found = new Place("noPlace", -1, -1);
     int i = 0;
     while ((i != allPlaces.length) && (!success)) {
        if (name.equals(allPlaces[i].place())) {
           success = true;
           found = allPlaces[i];
        }
        i++;
     }

     if (success) return found;
     else throw (new Error("Place doesn't exist."));
  }
}
```

We implemented the method *Place toPlace(String name)* as a class method of the class **AllPlaces** to make it available everywhere in the program.

We handily placed all the available places in an array to be able to find more easily and in a more general manner the Place we're looking for.

5.4.2 Adding Another Place

When we add another place, we have to re-implement our class **Venster**, adding the new place, both to the *from* and *to* check box group.

However, having learned from the *toPlace()* method, we can also define the constructor of that class in a general manner using the class variable *AllPlaces*.

```
public Venster() {
  Place[] places = AllPlaces.allPlaces;
  nrPlaces = places.length;

  ...
  CheckboxGroup from = new CheckboxGroup();
  // the checkboxgroup with the 'from' place

  for (int i = 0; i != nrPlaces; i++)
      add(new Checkbox(places[i].place(), from, true));
  ...
  // analogue for the 'to' check box group
}
```

Having done this, we only need to change the class **AllPlaces** to add a new selectable place.

```
public class AllPlaces {
  public static Place aarschot = new Place("Aarschot", 60, 30);
  public static Place begijnendijk = new Place("Begijnendijk", 50, 20);
  public static Place leuven = new Place("Leuven", 50, 90);

  //change for olen
  public static Place olen = new Place("Olen", 30, 4);

  // change for olen
  public static Place[] allPlaces = {aarschot, begijnendijk, leuven, olen};

  ...
```

This is as good as in Visto, but it requires a larger initial programming effort placing *allPlaces* in a static class and implementing the constructor of **Venster** genericly, which Visto avoids. The advantages of Visto originate here in the use of selectors. We believe that it should be possible to implement such a selector class in Java as well.

5.4.3 Moving from a Radio Button to a List Box

Just as with Tcl/Tk this is a large modification. First the construction of **Venster** changes.

```
public Venster() {
  ...
  // if we want to change this to a ListBox this has to change completely.
  //was: CheckboxGroup to = new CheckboxGroup();

  List to = new List(nrPlaces, false);
  //false because we only want one selection!

  for (int i = 0; i != nrPlaces; i++)
     //was: add(new Checkbox(places[i].place(), to, true));
     to.add(places[i].place());
  //Also new is that we have to add this list to the frame
  add(to);
  // and then we connect the actionListeners
  connectButton.addActionListener( new ConnectionListener(from,to));
  }
```

Although the usage of the constructor of *ConnectionListener()* doesn't change, its type changes and a new constructor must be defined. Additionally, retrieving the selection from a list box is different from doing it for a check box group. So that must change as well.

```
public void actionPerformed (ActionEvent event) {
  String fromWhom = event.getActionCommand();

  if (fromWhom.equals("Connect")) {
     Place fromP, toP;
     fromP = AllPlaces.toPlace( from.getSelectedCheckbox() .getLabel());

     //also the way in which to extract the value must be changed
     //was: toP = AllPlaces.toPlace(to.getSelectedCheckbox().getLabel());
     String toName = to.getSelectedItem();
     if (toName == null) {
        toP = AllPlaces.allPlaces[0];
        to.select(0); }
     else toP = AllPlaces.toPlace(toName);
     Connection connection = new Connection(fromP, toP);
     ...
```

Adding a new location could be done as easily as in Visto, but changing the widget with which the place is selected, is far from as easy as in Visto. Although some of the ideas of Visto such as the *expand* derivation may be not so easy to implement in Java, the *selectors* idea, which avoids the difficulties discussed in this section, certainly should be feasible.

It is however not a trivial task, requiring a definition that abstracts away the actual presentation of the selector. Furthermore a new set of more declarative events must be defined.

5.4.4 Adding a View

The ideas of Tcl/Tk survive here. We may create an independent object that displays another view of the connection, but the *ActionListener* object must still be aware of this object.

The advantage over Tcl/Tk however is that it is no longer the call back function of the widget itself that must know all the views, but a *controller* object that centralises the control flow between the application object, the views and the user interface.

```
public void actionPerformed (ActionEvent event) {
    ...

    //and now trigger all views
    outputBox.clear();
    outputBox.printLine(connection.distance());
    outputBox.show();

    drawView.draw(fromP, toP);

    // a new view
    System.out.println("Connection from " + fromP.place() + " to " + toP.place());
}
```

Nevertheless, it remains easier in Visto to add a new view because only the view must know that it is an expansion of a some other object, and we don't need to let that object or its controller know that a new view exists.

5.4.5 A Short Cut for the Begijnendijk Connection

The part where we add another button to select Begijnendijk as the starting point is easy. In the class **Venster** we have to add a new button to the frame.

We must also define a *Listener* for this button. This *Listener* object is the controller of the MVC-pattern and thus entirely controls the views. In this case we want to share the views for the default connection button and for the Begijnendijk button. However, we made the views private data members of the ConnectionListener. This was a good decision when we had just one view, but imposes some problems now.

Making the views public is no valid option, as we don't want to give full control over the views to the public. Instead we define *get* methods that return the different views, and use those methods in the *BegijnenListener* to make sure that it uses the same views as the *ConnectionListener*.

```
public Venster() {

   ...
   // we first add the button for Begijnendijk
   begijnenButton = new Button ("Begijnendijk");
   add (begijnenButton);

   ...
   // and then we add the actionListeners

   // The ConnectionListener can no longer be anonymous.
   ConnectionListener cl = new ConnectionListener(from, to);
   connectButton.addActionListener(cl);

   BegijnenListener bl = new BegijnenListener(to);
   // Let bl share the views with cl
   bl.getViewsFromConnectionListener(cl);

   begijnenButton.addActionListener(bl);
}
```

The changes to the class **ConnectionListener** are easy, but what matters it that we had to apply some changes. More important is that, although the class **BegijnenListener** shares a lot of the behaviour of the class *ConnectionListener*, it cannot reuse elements of *ConnectionListener*.

```
public class BegijnenListener implements ActionListener {
   // the 'selector' elements
   private List to;

   // the views. New is that the views are not initialised
   private MainWindow m;
   private OutputBox outputBox;
   private ConnectionView drawView;

   public MainWindow getMainWindow()
   { return m; }

   public OutputBox getOutputBox()
   { return outputBox; }

   public ConnectionView getConnectionView()
   { return drawView; }
```

```
    publicBegijnenListener(List_to) {
        super();
        to = _to;
    }

    public void getViewsFromConnectionListener(ConnectionListener cl) {
        m = cl.getMainWindow();
        outputBox = cl.getOutputBox();
        drawView = cl.getConnectionView();
    }

    public void actionPerformed (ActionEvent event){
        String fromWhom = event.getActionCommand();

        if (fromWhom.equals("Begijnendijk")){
            Place fromP, toP;
            fromP = AllPlaces.begijnendijk;

            String toName = to.getSelectedItem();
            if (toName == null){
                toP = AllPlaces.allPlaces[0];
                to.select(0);}
            else {toP = AllPlaces.toPlace(toName);}
            Connection connection = new Connection(fromP, toP);

            //and now trigger all views
            System.out.println(
                "Connection from" + fromP.place() + "to " + toP.place());

            outputBox.clear();
            outputBox.printLine(connection.distance());
            outputBox.show();

            drawView.draw(fromP,toP);
        }
        else {throw(new Error("Unexpected source of event."));}
    }
}
```

This extension of the interface requires even more changes than in the Tcl/Tk version, which already was a lot more work than in the Visto program. This time the advantage of Visto lies within the usage of invokers and the easy reuse of selectors using named selectors.

5.5 Clean

After these two examples from the imperative or object oriented world, we return to functional programming. In section 2.4.1 we already explained shortly how Clean handles I/O by splitting the *World* in autonomous parts. Here we can see a somewhat larger example of how the rather abstract idea is actually being applied in a ‘real’ program.

5.5.1 The Basic Application

We will not explain all details of the program, but essential is the fact that we continuously pass around a *Process State* (mostly *pst* in the program code). From this process state individual entities can be split by functions as *openWindow* and *openDialog*.

```
Start :: *World -> *World
Start world = startGIS (openIds 6 world)
```

Start is the *main* of a Clean program. We have already explained that identifiers are used for laying out the widgets, but they are also needed for accessing them. So we first let the function *openIds* create 6 unique identifiers. The function *startGIS* then really starts the program.

```
startGIS :: ([Id],*World) -> *World
startGIS ([viewId,connectId,fromId,toId,fromTextId,dialogId : _], world)

    = startIO SDI                          //SingleDocumentInterface
              Nothing                      //The initial LOCAL process state
                                           //No connection yet, thus ‘Nothing'
              initialise                   //The initialisation action
              [ProcessClose closeProcess]  //Attributes
              world
   where...
```

The only interesting part in this function – except for the names of the identifiers – is the function *initialise* that really initiates the I/O. In this function we define the different user interface elements and their behaviour. [2]

The code in the following fragment is already more interesting. In *initialise* we first open a *graphicWindow*, which is the graphical view on the connection. If this is successful we open the dialog that lets the user select the connection.

```
    ... where
```

[2]The # is a non-recursive let.

```
initialise pst
    # (error,pst) = openWindow undef graphicWindow pst
    | error <> NoError
        = abort "Clean GIS could not open window."
    # (error,pst) = openDialog undef dialog pst
    | error <> NoError
        = abort "Clean GIS could not open dialog."
    | otherwise
        = pst
graphicWindow = Window "Graphical View" NilLS
                                    [WindowId viewId
                                    ,WindowViewDomain ViewDomain
                                    ,WindowLook True (look Nothing)
                                    ]
```

Just as the graphical windows in Visto have a **draw** method, we must define a *WindowLook* if we want to draw something on the window. The function *look* is executed whenever the window's contents must be refreshed. It is written again in a environment passing style, both taking and returning a picture.

```
look :: (Maybe (Place,Place)) SelectState UpdateState *Picture -> *Picture
look Nothing _ _ picture = picture
look (Just (p1 =: coords = (x1,y1), p2 =: coords = (x2,y2))) _ {newFrame} pic
   # picture = drawLine {x = x1,y = y1} {x = x2,y = y2} pic
   = picture
```

The connection dialog.

```
dialog = Dialog "Select Places..."
              (ListLS
                  [(ButtonControl "Connect"
                           [ ControlId connectId
                           , ControlFunction connect])
                   :+: (RadioControl
                           [(pl,Nothing,id) \\ p =: place = pl <- allPlaces]
                           //list comprehension
                           (Rows 1) 1 [ControlId fromId])
                   :+: (TextControl "From : "
                           [ControlId fromTextId])
                   :+: (RadioControl
                           [(pl,Nothing,id) \\ p =: place = pl <- allPlaces]
                           (Rows 1) 1 [ControlId toId])
                   :+: (TextControl "To : " [])
                  ])
              [WindowId dialogId]
```

Again, the layout information which is normally specified using the *ControlPos* attributes and relative to other widgets, e.g. *ControlPos (Below fromId,NoOffset)*, has been removed from this program fragment.

As we have learned from the Java implementation, we use a list *allPlaces* containing all the places, and construct the *RadioControl* using a list comprehension $[(pl, Nothing, id) \backslash\backslash\ p =: place = pl \ \leftarrow allPlaces]$

The *ButtonControl* needs a call back function (*connect*) to activate it. This function, of type $(LocalState, ProcessState) \rightarrow (Localstate, ProcessState)$. is executed whenever the button is pressed,

```
connect (lst,pst)
    # pst = appPIO performSomeIO pst
    = (lst,pst)

fromP w = fi ndPlace (fromJust (snd (getRadioControlSelection fromId w)))
toP w = fi ndPlace (fromJust (snd (getRadioControlSelection toId w)))

performSomeIO io
    # (maybeDialog, io) = getWindow dialogId io
    # dialog = fromJust maybeDialog
    # newConnection = Just (fromP dialog, toP dialog)
    # io = setWindowLook viewId True (True, look newConnection) io
    = io
```

However, because we want to change the *WindowLook* of our graphical view to reflect the new connection, we actually want to perform some IO, and therefore must type cast our IO function *performSomeIO* to the right type using *appPIO*.

The functions *fromP* and *toP* use *getRadioSelection* to know which radio button has been selected. As this returns an index – just as in Java – we have also defined a function *findPlace* mapping such an index to the actual place.

```
fi ndPlace n = hd (drop (n − 1) allPlaces)
```

5.5.2 Adding Another Place

Just as with our Java program, it suffices to add another place to the constant *allPlaces*. The initial programming effort required to be able to do this is smaller than in Java, but still substantially larger than in Visto, where no such effort must be made.

5.5.3 Moving from a Radio Button to a List Box

A list box as known in the other toolkits doesn't exist in Clean, but instead we used the *PopUpControl* (fig. 5.7) which behaves a lot like the list box in the other toolkits.

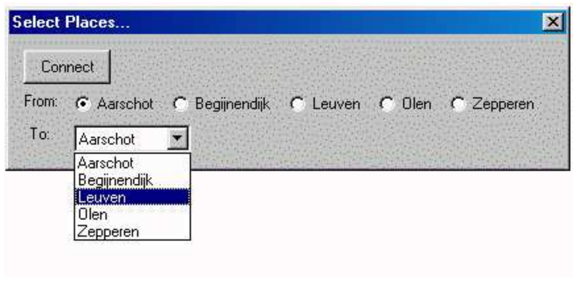


Figure 5.7: Clean PopUpControl.

To include this in our GIS program, we have to change the *RadioControl* to a *PopUpControl*.

```
...
//changed
:+: (PopUpControl
    [(place, id)
    p =: place = place ← allPlaces]
    1[ControlId toId])
...
//changed
toP w = findPlace (fromJust (snd (getPopUpControlSelection toId w)))
```

At the same time, we had to change the function *toP* to use the *getPopUpControlSelection* instead of the *getRadioControlSelection*. Luckily the type of this function is identical to the previous type, so we didn't have to change other functions.

We believe that the Clean people have done a very good job here. Without reverting to the *selectors* approach, this must be the smallest possible code change. However, if selectors *do* make life easier, why shouldn't we apply the idea?

5.5.4 Adding a View

The previous stages proceeded pretty successfully in Clean. Adding a view turns out to be substantially more work.

A first change is the fact that we need another ID. We must now open 7 ID's instead of 6. This influences both *Start* and *startGIS*.

```
Start :: *World -> *World
Start world
  = startGIS (openIds 7 world) //changed


startGIS ([viewId,connectId,fromId,toId,
           fromTextId, dialogId,connectWinId : _],world)
  = startIO MDI
            // Multiple Document Interface
            // had to change as well from SDI to MDI
            Nothing
            initialise
            [ProcessClose closeProcess]
            world
where
  initialise pst
     # (error,pst) = openWindow undef graphicWindow pst
     | error <> NoError
       = abort "Clean GIS could not open window."
     //new
     # (error,pst) = openWindow undef connectionWindow pst
     | error <> NoError
       = abort "Clean GIS couldn't open connection window"
     ...
     //new view
     connectionWindow
         = Window "The connection" NilLS
                  [WindowId connectWinId
                  ,WindowViewDomain ViewDomain2
                  ,WindowLook True (look2 Nothing)]
```

Next, we don't only have to open a new window (*connectionWindow*), but also change from a *SDI* (Single Document Interface) to a *MDI* (Multiple Document Interface).[3]

Of course, also a new *WindowLook* function must be defined.

```
//also completely new
look2 :: (Maybe (Place,Place)) SelectState UpdateState *Picture -> *Picture
look2 Nothing _ _ picture = picture
look2 (Just (p1 =: {place = pl1}, p2 =: {place = pl2})) _ {newFrame} pic
  # picture = drawAt {x = 5, y = 20}
                     ("Connection from " +++ pl1 +++ " to " +++ pl2) pic
  = picture
```

[3]That's the reason why our first Clean version only had *one* view. If we had started with two views, we wouldn't notice that change here.

This part is only slightly more work than in Visto, but, again like in Java, we must also update our call back function *connect* of the connect button to accomodate the new view. So we see again that the Visto views are really automatically updated and the views of traditional MVC are not.

```
performSomeIO io
  # (maybeDialog, io) = getWindow dialogId io
  # dialog = fromJust maybeDialog
  # newConnection = Just (fromP dialog, toP dialog)
  # io = setWindowLook viewId True (True, look newConnection) io
  // new view
  # io = setWindowLook connectWinId True (True, look2 newConnection) io
  = io
```

5.5.5 A Short Cut for the Begijnendijk Connection

Just as in the previous case where we needed another ID when a widget was added, this is again the case here. A more important change occurs in the dialog. There a new button must be added.

```
dialog = Dialog "Select Places..."
              (ListLS [(ButtonControl "Connect" ...)
               :+: (ButtonControl "Begijnendijk"
          [ControlId begijnenId
          , ControlFunction connectBegijnen
          , ControlPos (LeftOf toTextId, NoOffset)])
          //dialog layout is ugly, but is considered irrelevant here
```

A second change is the definition of a new call back function.

```
performBegijnenIO io
  # (info, io) = getWindow dialogId io
  # real = fromJust info
  # newConnection = Just (begijnendijk, toP real)
  # io = setWindowLook viewId True (True, look newConnection) io
  # io = setWindowLook connectWinId True (True, look2 newConnection) io
  = io
```

Although this is no difficult function if we know the definition of the previous *performSomeIO* call back function, what matters is that we have to define a completely new function, just as in the case of Tcl/Tk and Java. This is normal as Clean essentially provides a functional implementation of the MVC-pattern with call back functions, which requires a more if-then implementation: if that widget is selected, perform that action. Visto on the other hand supports a more goal-oriented implementation "*to activate that method, the user must select those widgets.*", which, at least in this case, turns out to be more easily extensible and maintainable.

5.6 Fudgets

5.6.1 The Basic Application

The previous implementations could be approached rather directly, explaining them incrementally. However, in Fudgets it is important to first get a general idea of the structure of the program.

Remember from section 2.4.2 that a fudget is stream processor: it accepts values on its input stream and can put other values on its output stream.

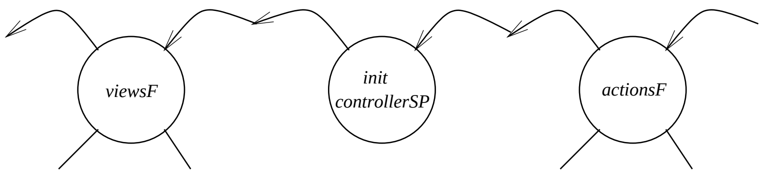


Figure 5.8: General scheme.

So our general scheme (fig. 5.8) has a dialog fudget (*actionsF*) on the right, the views (*viewsF*) on the left, and in the middle a controller (*initControllerSP*) that accepts inputs from the dialog, processes them and outputs appropriate values to the views.

actionsF. Naturally, the real scheme is a bit more complicated. In the dialog we actually have three components: two radio button groups that let the user select the *from* and *to* place, and a *connect* button that connects them.

$$\begin{aligned}&\mathit{placeSelectorF} \;=\; \mathbf{radioGroupF}\ [(p, \mathit{place}\ p) \mid p \leftarrow \mathit{allPlaces}]\ \mathit{aarschot}\\ &\mathit{allPlaces} \;=\; [\mathit{aarschot}, \mathit{begijnendijk}, \mathit{leuven}]\end{aligned}$$

$$\begin{aligned}\mathit{placesF} \;=\; &(\mathbf{labLeftOfF}\ \text{"}\mathit{From}\ :\ \text{"}\mathit{placeSelectorF})\\ &>+<\\ &(\mathbf{labLeftOfF}\ \text{"}\mathit{To}\ :\ \text{"}\mathit{placeSelectorF})\end{aligned}$$

In Fudgets we are able to fully re-use the definition of the selection of a place. Using $>+<$ we connect this *placeSelectorF* in parallel with itself. A value *Left Place* is a *from* place, a value *Right Place* is the *to* place.

```
connectF = buttonF "Connect"

  —actionsF :: F a (Either Click (Either Place Place))
actionsF = connectF >+< placesF
```

This *placesF* is once again connected in parallel with the connect button. A value *Left Click* comes from a click on the connect button, a *Right* value is a location that has been selected.

The controller must handle these values correctly.

initControllerSP.

```
initControllerSP = controllerSP aarschot aarschot
controllerSP from to = getSP (\ input → controlOutputSP input from to)
controlOutputSP (Left _) from to = putSP (from, to) (controllerSP from to)
controlOutputSP (Right (Left from)) _ to = controllerSP from to
controlOutputSP (Right (Right to)) from _ = controllerSP from to
```

controllerSP keeps two places, the origin (*from*) and the destination (*to*). When it receives a *Right* value, it replaces the appropriate place with the input, and outputs nothing. When it receives a *Left* value, the connect button has been clicked, and the views must be updated. *controllerSP* then tuples the connection and puts it on its output stream.

viewsF. We first take a look at the view that only displays the distance between the two places as it is a bit easier than the graphical view.

```
viewDistanceF = moreShellF "Distance between selected places"
                              >=^< distanceStrings
distanceStrings (p1,p2) = [show (distance p1 p2)]
```

We use a *moreShellF* that accepts values of type [*String*] and displays them in a window. We must precede this fudget, using the combinator $>=\wedge<$, with a function *distanceStrings* that maps the output from *controllerSP*, a tuple (*Place*,*Place*), to [*String*].

```
drawLine (p1, p2) =
  let
    (x1,y1) = coords p1
    (x2,y2) = coords p2
    lineCom = DrawLine (Line (Point x1 y1) (Point x2 y2))
  in
  replaceAllGfx (FD (Point 100 100) [lineCom])
```

$$emptyDrawing \ = \ \mathbf{replaceAllGfx}\ (FD\ (Point\ 100\ 100)\ [])$$

$$graphicalF \ = \ \mathbf{graphicsDispF} >{=}\wedge{<}\ drawLine$$

$$\begin{aligned}&viewGraphicalF\\ &\quad = \mathbf{shellF}\ \text{“}View\ connection\text{”}\ (graphicalF >{=}\wedge\wedge{<}\ \mathbf{putSP}\ emptyDrawing\ idSP)\end{aligned}$$

The graphical view is analogue, but a bit more complicated as we must supply complete drawings instead of just a list of strings. Furthermore we must initialise the graphics window with an empty drawing, and wrap it in a *shellF* to get in a top level window.

Putting it all together. We let the controller simply output the connection once, but actually it must be passed to both views. Therefore we connect the two views in parallel and use the *toBothSP* stream processor to duplicate the connection both to the *Right* and to the *Left*.

$$viewsF \ = \ viewDistanceF >{+}{<}\ viewGraphicalF$$

$$\begin{aligned}gisF \ = &(viewsF\ >{=}\wedge\wedge{<}\ toBothSP)\\ &>{=}{=}{<}\\ &(initControllerSP >\wedge\wedge{=}{<}\ actionsF)\end{aligned}$$

5.6.2 Adding Another Place

What we could do in Visto without effort, in Java with some effort, and in Clean with little effort, can be done in Fudgets as well. Only the constant *allPlaces* must be changed to add another place to the selection the user can choose from.

5.6.3 Moving from a Radio Button to a List Box

And yet again, this was little work in Visto, but in Fudgets a lot more, as the type of the list box (a *pickList* in Fudgets) is very different from the type of a *radioGroup*.

- First, *pickList* puts values of type *InputMsg* $(Int,\ Place)$[4] on its output stream instead of the selected element itself. If we want a type compatible with the other *radioGroup* selector, we must strip the *InputMsg* and take the second element of the tuple $(Int,\ Place)$.
- One of the arguments of *radioGroup* was the list of values to display in the radio group. *pickListF* accepts values of type *PickListRequest a* on its input stream to define the set to choose from, not by giving it as its argument when constructing the list box. So we must instantiate *pickListF* using a *putSP*.

[4]Actually this is polymorphic *InputMsg (Int, a)* with *a* the type of the selectable values. For simplicity we instantiated the type *a*.

- *radioGroup* expected tuples $(value,\ graphical\ representation)$ and used the graphical representation to display the value. *pickList* uses a function $a \rightarrow String$ to map each of its values to a string that is used in the list box.

The following code accomodates these requirements:

$$\begin{aligned}
&placeSelector2F \;=\; \textbf{pickListF}\; place >\!=\wedge\wedge< \textbf{putSP}\;(\textbf{replaceAll}\; allPlaces)\; \textbf{idSP}\\
&strippedPlaceF \;=\; stripper > \wedge =< placeSelector2F\\
&stripper\; inputmsg \;=\; \textbf{snd}\;(\textbf{stripInputMsg}\; inputmsg)
\end{aligned}$$

Just as we commented with Tcl/Tk, a lot of technicalities must be solved here simply to change the way in which the place is selected, which is actually programming the *how* of the program, something we avoid whenever possible in a declarative program.

5.6.4 Adding a View

Creating a new view on itself is not too difficult. However as we're now stuck with three views, the parallel connection is no longer viable. Instead we use the *listF* combinator. To send a value x to the n^{th} fudget of the list, we put a (n,x) on the input stream of the *listF* fudget. So, to send the output of the controller to all views, we use the *toAllSP* stream processor.

$$\begin{aligned}
&viewConnectionF\\
&\quad = \textbf{shellF}\; \text{``}Show\ connection\text{''}\;(\textbf{displayF}\; >\!= \wedge <\; connectionString)\\
&connectionString\;(p1,p2)\\
&\quad = \text{``}Connection\ from\ \text{''} +\!+ (place\; p1)\; +\!+ \text{''}\ to\ \text{''} +\!+ (place\; p2)\\
&\\
&views3F\; =\\
&\qquad \textbf{listF}\;[(1, viewGraphicalF),\\
&\qquad\qquad (2, viewDistanceF),\\
&\qquad\qquad (3, viewConnectionF)]\\
&toAllSP \;=\; \textbf{getSP}\;(\backslash\, x \rightarrow \textbf{putsSP}\;[(n,x) \mid n \leftarrow [1,2,3]]\; toAllSP)\\
&\\
&gis2F \;= (views3F >\!= \wedge\wedge < toAllSP)\\
&\qquad >\!=\!=\!<\\
&\qquad (initControllerSP > \wedge\wedge =< (\textbf{shellF}\; \text{``}actions\text{''}\; actionsF))
\end{aligned}$$

In Fudgets we got the desired effect of not having to change a controller or a call back function to accomodate for new views, but unfortunately we had to change the way in which the views are connected to each other. Close, but no cigar...

5.6.5 A Short Cut for the Begijnendijk Connection

To add a view, we changed the flow in the Fudgets program such that view commands reached all the views. Now we add an action widget. The *Click*s from this fudget must also be added to the flow of the program. Therefore we change the *actionsF* fudget and the stream processor *controlOutputSP* that receives the inputs from *actionF*.

$$\begin{aligned}&\mathit{begijnenF} \;=\; \mathbf{buttonF}\;\text{“}\mathit{Begijnendijk}\text{”}\\&\mathit{actionsF} \;=\; (\mathit{connectF} >\!+\!< \mathit{begijnenF}) >\!+\!< \mathit{places2F}\end{aligned}$$

Previously we could ignore the precise value of the input on the left stream of the stream processor *controlOutputSP*. Now we have to disambiguate between the input $(\mathit{Left}\ (\mathit{Left\ Click}))$ which comes from the general connect button and $(\mathit{Left}\ (\mathit{Right\ Click}))$ which originates from the Begijnendijk button.

$$\begin{aligned}&\mathit{controlOutputSP}\ (\mathit{Left}\ (\mathit{Left\ Click}))\ \mathit{from\ to}\\&\quad = \mathbf{putSP}\ (\mathit{from}, \mathit{to})\ (\mathit{controllerSP\ from\ to})\\&\mathit{controlOutputSP}\ (\mathit{Left}\ (\mathit{Right\ Click}))\ \mathit{from\ to}\\&\quad = \mathbf{putSP}\ (\mathit{begijnendijk}, \mathit{to})\ (\mathit{controllerSP\ from\ to})\end{aligned}$$

Luckily this turned out to be rather straightforward, but it can already be guessed that relative simple changes to the user interface can have a large impact on the way the streams are connected to each other, and on the stream processor that control these streams.

In Visto we could see that the changes were restricted to only the program parts that directly defined the interface and not in more distant parts of the program.

5.7 Conclusion

We have seen how in the Visto implementation the changes only affected the code that directly had to do with the changes. No other parts of the program had to be changed.

Tcl/Tk is the oldest and most basic of the systems we used, but it still is good at what it does. Although even simple changes had an impact on the program structure at two places: the definition of the interface itself and the associative arrays, this is mostly due to the absence of a modern type system in Tcl/Tk.

Moving to more recent and advanced programming languages as Java, Clean and Fudgets, things went easier for adding a new location. Especially with a bit of careful programming, this turned out to be easy in all systems. Visto still excells because it doesn't require the extra initial programming effort. This is thanks to the selectors approach that allows for an easy modification of the set to choose from. The advantage in code reduction may not be enormous but it exists and furthermore expresses this subgoal of the user interface well.

The change of a radio button group to a list box was more intrusive, because the two widgets are often constructed in a different way. The required code modification in Clean was minimal, but because the selected value in a listbox is retrieved differently from the radio button group, we needed a considerable amount of recoding in in Tcl/Tk and Java. In Fudgets this was due to different types that are put on the output stream of the fudget. In Visto however, we only had to instantiate the selector in a different way and correct the layout. This aspect of being able to easily alter the appearance of a selector has a greater impact than that of changing the set to choose from. Hence, defining a number of different user interfaces starting from a system with selectors like Visto should be a lot easier than using a traditional system. Nevertheless no system can ever provide all possible instantiations of a selector. So we must always support bypassing pre-implemented instantiations, but still the selectors remain useful for describing the selection process at a higher implementation level.

Fourthly, adding a new view, required changes to the *ActionListener* object in Java and to the call back function in Tcl/Tk and Clean. In those systems 'something' must really trigger each of the views individually. In Fudgets the controller stream processor doesn't need to know that a view has been added, but the structure of the Fudgets streams must be changed. Visto was once again easier and, as we feel it, more logical: only the view must register itself as a view of some other object. We don't have to change any other code outside the view, thus minimizing the impact of adding a new view. Therefore the *expand* object derivation together with the automatic draw feature prove to be an easy and reliable technique for views, at least when it suffices when the views are redrawn once for each method evaluation. For more dynamic views that wish to update their display more frequently and in a less predictable way, extra language features are needed, but Visto's technique can still be used as a starting basis and/or for prototype purposes.

Finally when we added a short cut for the Begijnendijk connection, we could experience a disadvantage of the call back function approach, making the changes to the Tcl/Tk, Java and Clean program a lot harder than the changes to the Visto program, as we couldn't reuse the previous call back function although it basically does the same for the Begijnendijk connection. We even had the most troubles in the newest language, Java. In Fudgets the problem was again in the connection of the streams of the various fudgets and the stream processor that must handle all those streams. Hence, our invokers and selectors are not only a more declarative description and implementation, but also one that allows a higher reuse and less recoding.

This case study thus has shown that Visto precisely fulfills the goals that were put forward in the beginning of this thesis: describing in a declarative way the interface (stress on the *what*, the goal of the interface) and minimizing the impact of changes to the user interface.

It is furthermore worthwhile pointing out the individual contributions of each of Visto's features, such as the two way abstraction of selectors (the set to choose from, and the appearance), the view technique and the invokers approach. Therefore other systems don't have to adopt all of Visto's ideas, but can already benefit from a partial application, such as defining generic selector classes for Java.

Chapter 6

Visto Revisited

6.1 Language Overview

In the previous chapters we have presented Visto in a story-wise manner. This is fine for explaining the conception and evolution of Visto, but as a side effect the characteristics of our language become spread widely over the text of this thesis.

Therefore we pay a second visit to Visto in this short chapter, which enables us to stress the essential features that are new in this work or different from most standard works.

As a guide line we use a simplified BNF-syntax, following a top-down approach. Elements in bold and between " **quotes**" are terminals. Non-terminals are written like $< \mathit{this} >$. $|$ denotes a choice and square brackets ($[$ and $]$) optional elements. Repetition of zero or more times is indicated by a *, repetition of at least one time by $^+$.

There are also some simple non-terminals that are not explained in the text, such as the various names, e.g.. *interfaceName*, *objectName*, ..., which are simple strings, and *stateValue* which is a Haskell value for the state of an object.

6.1.1 General Program Structure

$$\begin{aligned} < \mathit{VistoProgram} > \; = \; & < \mathit{interfaceDef} >^+ \\ & < \mathit{objectDefinition} >^+ \\ & [< \mathit{haskellCode} >] \end{aligned}$$

Visto is an object language, so we need a number of *object definitions*. As each object must implement a particular interface, we also need a number of *interface definitions*.

However, Visto is not a *pure* object language in the sense that *only* objects are allowed in a Visto program. We do need at least one object, but some plain Haskell code is allowed as well.

We did this because one of the goals of Visto is defining a graphical user interface for an (existing) Haskell program. So we don't want the programmer to fully retranslate his program into Visto, but rather let him reuse as much code of his Haskell program as possible and only have him define the Visto objects necessary to create the user interface.

This way we also have an enhanced functional feel of the Visto system. If we need some auxiliary functionality, we can just define standard Haskell functions. Hence, this is completely different from most object oriented languages.

6.1.2 Interfaces

The concept of interfaces is certainly not a concept introduced by Visto. New is its application in a prototype based language, as it is a notion more familiar in class based languages because there it is more likely that many objects will adhere to an interface. In prototype based languages each object can easily differ from every other object and thus have its own interface.

Therefore designers of prototype languages can be easily inclined not to use interfaces. To our knowledge no prototype based language uses interfaces. At least the two most important prototype based languages, Self and NewtonScript, don't use interfaces. We however have consistently used interfaces throughout Visto, which we can thus safely consider as a new contribution to the field. We insist on interfaces because we want to facilitate *programming to an interface* and find it valuable documentation.

$$\begin{array}{lcl} <\textit{interfaceDef}> & = & <\textit{simpleInterfaceDef}> \\ & | & <\textit{adaptInterfaceDef}> \\ & | & <\textit{expandInterfaceDef}> \end{array}$$

We have three kinds of interfaces. The simple interface defines a completely new interface. However, just as all object languages have constructs to derive new *members* [1] from existing members, we provide support for this in Visto as well.

Our approach lies between classes and prototypes. On the one hand, interfaces can inherit from existing interfaces, like in class based languages; on the other hand objects can inherit from other objects. By deriving interfaces from interfaces, we reuse types; by deriving objects from objects, we reuse implementation.

We first present $<$ *simpleInterfaceDef* $>$, the basic interface definition, and then $<$ *adaptInterfaceDef* $>$ and $<$ *expandInterfaceDef* $>$, the two ways for deriving a new interface from an existing one.

[1] In class based languages the *member* is the class, in prototype languages the object.

The simple interface definition contains a name for the interface and the name and type for each of the methods.

$$< \mathit{simpleInterfaceDef} > \; = \; \text{“}\textbf{interface}\text{”} \; < \mathit{interfaceName} > \\ < \mathit{methodType} >^{+}$$

$< \mathit{methodType} > = < \mathit{methodName} > \; :: \; < \mathit{haskellType} >$

The type of the method can be any Haskell type. The type present in the interface reflects the type of the arguments for the method and the type of the return value. Visto methods also have to return a new state value, but that is not visible in the interface, as discussed in section 3.4.1. Some alternatives for this mechanism were presented in section 3.11.2.

The second interface $< \mathit{adaptInterface} >$ is useful when we want to modify an existing interface, by adding and/or removing methods from the interface.

$< \mathit{adaptInterfaceDef} > =$
  “**adapt**” $< \mathit{interfaceName} >$ “**from**” $< \mathit{interfaceName} >$
    $(< \mathit{addMethod} > | < \mathit{removeMethod} >)^{*}$

$< \mathit{addMethod} > = < \mathit{methodType} >$

$< \mathit{removeMethod} > = < \mathit{underscore} >< \mathit{methodName} >$
$< \mathit{underscore} > =$ “_”

$< \mathit{expandInterfaceDef} > =$
  “**expand**” $< \mathit{interfaceName} >$ “**from**” $< \mathit{interfaceName} >$
    $(< \mathit{addMethod} > | < \mathit{removeMethod} >)^{*}$

If we adapt an interface, we in principle inherit all the methods from the ancestor interface. Just as in all other object languages we can also add new methods to the interface.

Original in Visto is the fact that methods can be removed from an interface as well. This is an interesting novelty that we included because we want to exploit to its full potential the flexibility offered by prototype languages. So as it is possible to remove elements from concrete things e.g. remove a drawer from a desk, we felt that it should be possible to remove methods from an interface and a derived object as well.

The third interface, *expandInterface*, has the same syntax as *adaptInterface*. From an interface point of view, the two first interfaces – the simple interface and the adapted interface – suffice. We can write completely new interfaces and/or reuse precisely those parts of the interface that we're interested, removing the unnecessary methods and adding new ones.

But interfaces are not the final viewpoint for our system. In a prototype based language objects are the key members. We chose to implement all objects as self-contained. This means that they contain all their method implementations and thus don't depend on other objects. This makes the implementation slightly more efficient as we don't have to follow a chain of parent object links when a method is not defined in the object itself, but it is also a bit restrictive.

In class based languages where an object of a derived class also contains a reference to an object of the super class – a super object –, we can often explicitly use methods of the super class. This is technically not possible in Visto as an object only contains its own methods, and no references to the overwritten method in the superobject.

A typical application where the method of the superobject is used, is the incremental definition of a method. We have already applied the incremental definition of an object: we start with an object with fewer methods and add some new methods. But also a method itself can be implemented incrementally: we (first) apply the method in the super object, and (then) do something extra in the method of the derived object.

A typical example is the three dimensional point that inherits from a two dimensional point. A move of the 3D point then is a move of the 2D point and a move among the third axis as well.

This pattern can be applied in user interfaces as well. If we have a restricted view on some larger object, implemented by a scrollable window, a change to the size of the underlying object also affects the scrollable window e.g. the thumbs of the scroll bar must be resized to correctly reflect the relative magnitude of the view.

As Visto is aimed at user interfaces, we found this pattern particularly interesting and implemented it using the expand derivation (section 3.6). If one wants to use this pattern, the $<$ *expandInterface* $>$ must be used along with the $<$ *expand* $>$ derivation of objects.

We didn't consider visibility modifiers for the methods, such as public, private and protected, because we don't find this aspect essential for an experimental object language (section 3.9.1).

6.1.3 Object Definitions

$$
\begin{array}{rcl}
< \mathit{objectDefinition} > & = & < \mathit{blankObjDef} > \\
 & | & < \mathit{cloneDef} > \\
 & | & < \mathit{adaptDef} > \\
 & | & < \mathit{expandDef} >
\end{array}
$$

The first possibility is to create an object from scratch. We must then provide a name for the object, mention which interface we implement, give a type for the state value and an initial state value, and finally provide implementations for each of the methods in the interface. A draw definition is optional, as are the GUI invokers.

$$
\begin{array}{rl}
< \mathit{blankObjDef} > = & \text{“}\textbf{object}\text{”} < \mathit{objectName} > < \mathit{interfaceName} > \\
& < \mathit{haskellType} > < \mathit{stateValue} > \\
& \\
& \text{“\{”} \\
& \quad [< \mathit{drawDefinition} >] \\
& \quad < \mathit{methodDefinition} >^{+} \\
& \quad < \mathit{GUIinvoker} >^{*} \\
& \text{“\}”}
\end{array}
$$

We have chosen a single state variable per object. The motivation for this decision can be found in section 3.4.2, a review of alternatives in section 3.11.1.

The idea behind method definitions is like in all object languages: to provide a part of the functionality of the object. We return to this in the following section 6.1.4.

Because flexibility is an important concept in prototype languages, it must be easy to create new objects that follow an existing example. That is the reason why we can also create clones. As a clone implements the same interface as the original object, we don't have to specify the interface for the cloned object.

We can use the clone mechanism in a restricted way, like the instantiations of a class in a class based languages to define objects that only differ in their state value, but our clone mechanism is a lot more flexible.

We try to minimise the constraints on deriving objects from existing ones, and we thus allow the state type of the derived object to be different from the original object. When that is the case, we must of course also supply a new state value. Otherwise we can still give the derived object a new state by using the $< \mathit{stateDefinition} >$.

$$
\begin{array}{rl}
< \mathit{cloneDef} > = & \text{“}\textbf{clone}\text{”} < \mathit{objectName} > \text{“}\textbf{from}\text{”} < \mathit{objectName} > \\
& \quad [< \mathit{haskellType} > < \mathit{stateValue} >] \\
& [\\
& \text{“\{”} \\
& \quad [< \mathit{drawDefinition} >] \\
& \quad [< \mathit{stateDefinition} >] \\
& \quad < \mathit{methodDefinitionExtra} >^{*} \\
& \quad < \mathit{GUIinvoker} >^{*} \\
& \text{“\}”} \\
&]
\end{array}
$$

The methods for which no new definition is provided are inherited from the original object.

The *adapt* derivation shares all the features of the *clone* like the fact that it can have a different state type and that methods can be inherited or overwritten. The greatest difference is that an adaption implements a different interface than the original object. Therefore we must mention the interface that the adaption implements.

If that interface removes some methods from the original interface, those methods are automatically removed in the adaption. If there are new methods in the interface, we must provide implementations for those methods.

$$
\begin{array}{lll}
<\mathit{adaptDef}> = & \text{“}\textbf{adapt}\text{”} & <\mathit{objectName}> <\mathit{interfaceName}> \\
& & [<\mathit{haskellType}> <\mathit{stateValue}>] \\
& & \text{“}\textbf{from}\text{”} <\mathit{objectName}> \\
& [& \\
& \text{“}\{\text{”} & \\
& & [<\mathit{drawDefinition}>] \\
& & [<\mathit{stateDefinition}>] \\
& & <\mathit{methodDefinitionExtra}>^{*} \\
& & <\mathit{GUIinvoker}>^{*} \\
& \text{“}\}\text{”} & \\
&] &
\end{array}
$$

Just like our clone can be compared with the instantiation of class based languages, our adapt can be compared with the extension of an existing class, as our adapt adds methods to and overwrites methods from an existing object, just like an extended class does with its original class.

Again Visto is more flexible. We already pointed out that besides adding methods we can remove methods as well. Another difference is that in a class based languages all objects of a derived class extend the original class in exactly the same way. That is normal as they are all instantiations of the same class. However, in our system two objects that adapt the same object according to the same adapted interface, can adapt the object in completely different ways: inherit different methods and implement the other methods differently.

This is a *free* feature, as it required no extra design effort. It was already present in the basic definition of an object, where two objects implementing the same interface can have different method implementations.

Finally, the definition of an expanded object. Just like $<\mathit{adaptInterface}>$ was syntactically identical to $<\mathit{expandInterface}>$, the syntax of the expand definition is identical to that of an adaption. The semantics however are fundamentally different as the expansion can delegate to, and be triggered by its mother object (section 3.6.2).

Together with the application of the concept of interfaces to prototype based languages, and the removal of methods from an interface, this is one of the really new features in the Visto object language.

A detailed discussion of expansions can be found in section 3.6.

```
< expandDef > =  "expand" < objectName > < interfaceName >
                          [< haskellType > < stateValue >]
                          "from" < objectName >
                 [
                  "{"
                      [< drawDefinition >]
                      [< stateDefinition >]
                       < methodDefinitionExtra >*
                       < GUIinvoker >*
                  "}"
                 ]
```

So far we have seen that the definition of an object can consist of many fields: a draw definition, a state definition, method definitions and GUI invokers. We have tried to provide a consistent syntax for the definition of these fields.

```
< drawDefinition > =  "draw" " = "  < drawBody >
< stateDefinition > =  "state" " = "  < stateValue >
< methodDefinition >
   = < methodName >  " = "  < methodBody >
< methodDefinitionExtra >
   = < methodName >  ( " = "  |  " = +")  < methodBody >
< GUIinvoker > =  "GUI" < methodName >  " = "  < GUIBody >
```

On the left hand side of the definition we always have the name of the field we're defining, and on the right hand side the actual definition. As standard in most programming languages, we use a equal sign to separate the left hand side from the right hand side. The only exception is the *< methodDefinitionExtra >*. Here we can use the **"=+"**-sign instead of **"="** to activate the triggering mechanism for that method (section 3.6.2).

At the places where these definitions can use Haskell values, the keywords **state** and **superState** can be used as well to refer to the state value of the object and that of the superobject, if such a superobject exists.

The draw definition contains on the first line the type of window to draw on, a *canvas*, a rectangular area on which lines, rectangles and ovals can be drawn, or a *label*, a one line text field. The actual $< DrawCommands >$ and their syntax are described in section 4.8.

$$
\begin{aligned}
< drawBody > \; &= (\text{ “}\textbf{Canvas}\text{” } < canvasSize > | \text{ “}\textbf{Label}\text{”}) \\
&\quad < DrawCommands >^{+} \\
< canvasSize > \; &= \text{ “(” } < width > \text{ “,” } < height > \text{ “)”}
\end{aligned}
$$

Our main motivation for this *draw method* was to centralise the code for the (re)drawing of an object, and to automate this redrawing, making sure that the information display is always consistent with the information present. It was only later that we discovered that Java has the same feature with its *paint* and *repaint* methods.

Initially we wanted all objects to visualise some information, and thus this draw method was obligatory. Later we noticed that this is too strict and doesn't allow a proper separation of issues (*model-view-controller pattern*) and made the draw method optional.

However, we also noticed that having it central, and *only* executing it whenever a method finishes its evaluation prohibits more dynamic interfaces, where the object's state is updated even during the execution of a method. And on the other hand, it may be too demanding for the system and the end user to redraw the object *every time* when a method finishes.

Therefore retaining this feature means that Visto is primarily aimed at more static interfaces, where simply redrawing each object when it executed a method suffices. In that case it does considerably simplify event handling.

In the other case we must give the programmer more control over how and when an object is redrawn. This means that we must add such features to the bodies of methods and thus furthermore 'contaminate' those methods in the sense that we add more side-effecting commands.

We currently feel that we should then consider moving to monadic style, which would probably mean developing an entirely different Visto, as we shortly discussed at the end of section 4.8 on p. 153.

The principle of the GUI invokers, an alternative for the typical call back mechanism, and that we find another important innovation of Visto, has already been explained in section 4.7 and several times throughout this thesis. An important motivation was the stress on expressing a goal (to activate that method from within the user interface, use that invoker with those selectors) versus describing an effect (when that widget is pressed, this and that action is performed).

$$
\begin{aligned}
< GUIBody > \; &= \; < GUIActivation > \\
&\quad (\text{ “;” } < GUIActivation >)^{*} \\
< GUIActivation > \; &= \; < invoker > \\
&\quad (\text{ “} << - \text{” } < selector >)^{*}
\end{aligned}
$$

Each method can be invoked in a number of different ways. Therefore the $<$ *GUIBody* can contain different $<$ *GUIActivation* $>$s. Each activation consists of an invoker, which is the GUI element that effectively initiates the evaluation of the method. When the method has parameters, the arguments are supplied by the $<$ *selector* $>$s.

6.1.4 The Method Body

So far it wasn't very visible that Visto is a functional object system, perhaps except for the fact that every method has a return value, just like functions have.

This is a good thing as we desired Visto to be an object language in the first place, with a functional implementation of the methods in the second place.

This section explains how we mixed the pure functional features with non-pure features, such as object communication and run time changes to the interface.

$$\begin{array}{lcl} < \textit{methodBody} > & = & < \textit{methodResult} > \\ & | & \text{“}\textbf{let}\text{”} \\ & & \quad < \textit{letBinding} >^{+} \\ & & \text{“}\textbf{in}\text{”} \ < \textit{methodResult} > \end{array}$$

The method result is a pair of the actual return value of the method and a new object state. This an original new approach. Imperative object systems naturally use destructive updating of state variables. Also Objective ML [67] uses that approach, but most modern functional object systems, such as Object-Gofer [72] prefer monadic style and monadic state variables.

Haskell++ [38] comes closest to our approach but returns a whole new object when it only needs to change its state. Sections 3.4.1 and 3.11.2 discuss these aspects in more detail.

$$\begin{array}{lcl} < \textit{letBinding} > & = & < \textit{HaskellLetBinding} > \\ & | & < \textit{messageBinding} > \\ & | & < \textit{superValueBinding} > \\ & | & < \textit{objectAction} > \\ & | & < \textit{interfaceAction} > \end{array}$$

As we explained in the beginning of this chapter, we want the programmer to be able to reuse Haskell code. Therefore one of the possibilities for a let binding is the standard let binding of Haskell, with on the left hand side the name of the local variable and on the right hand a Haskell expression.

The second possibility is applying a method from another object. We use the well known message sending protocol for that. Thirdly, we can add and remove some objects to the system, and fourthly we give the programmer the ability to dynamically change the user interface.

If we want even more dynamic features such as control over the redrawing of the object as discussed in the previous section, it is precisely here that we should add those features that we consider interesting on the level of methods.

And it is this *let* that would have to change to a *do* if we want to move from standard functional style to monadic style. This move would give us the opportunity to apply a number of other alternatives as well, resulting in a very different Visto (sec. 4.8).

$< \textit{HaskellLetBinding} > = < \textit{variableName} >$ "$=$" $< \textit{HaskellValue} >$

The *HaskellLetBinding* is the functional core of Visto.

$< \textit{messageBinding} > = < \textit{variableName} >$ "$=$" $< \textit{message} >$
$< \textit{message} > = < \textit{destination} >$ "!" $< \textit{methodName} >$ $[< \textit{argument} >]$
$< \textit{destination} > = < \textit{objectName} > \mid$ "**self**"
$< \textit{argument} > = < \textit{HaskellValue} >$

To apply the message passing mechanism, we must identify the object that we want to send a message to, and mention the name of the method that we want to call, and supply the method its arguments. Instead of using a fixed name for the object we send the message, we can also use the keyword **self** to refer to the object itself and to implement dynamic binding (see section 3.5.2, p.101).

As this message sending model implies that all objects can change their state (during the method evaluation the method may call other objects as well), we implicitly pass the list of all objects around. The semantics of this mechanism, and the complications are discussed in section 3.4.3.

$< \textit{superValueBinding} > = < \textit{variableName} >$ "$=$" "**superValue**"

As the keyword **superValue** can initiate a whole process of delegating and triggering a number of objects (section 3.6.2), we put it in a separate binding.

$< \textit{objectAction} >$
  $=$ "**addClone**" $< \textit{newObject} >$ $[< \textit{newState} >]$ "**from**" $< \textit{oldObject} >$
  $\mid$ "**addClonePlus**" $< \textit{suffix} >$ $[< \textit{newState} >]$ "**from**" $< \textit{oldObject} >$
  $\mid$ "**killObject**" $< \textit{objectName} >$

Next we can also create some clones at run time, and physically remove an object and its GUI contents (invokers, selectors and visualiser) from the system. They are the $< \textit{objectAction} >$s that are discussed in section 3.7.

$< \textit{interfaceAction} > =$ "**enableMethod**" $< \textit{methodName} >$
  $\mid$ "**disableMethod**" $< \textit{methodName} >$
  $\mid$ "**enableAssociation**" $< \textit{invokerName} >$
  $\mid$ "**disableAssociation**" $< \textit{invokerName} >$

$|$ "**enableElement**" $<$ *selectorName* $>$
$|$ "**disableElement**" $<$ *selectorName* $>$

Finally, we have the run time modifications to the interface. As we want the interface to change consistently, we have provided commands that let the programmers disable or enable at once all the 'widgets' involved in the application of the particular method.

This is an interesting innovation as it simplifies the task of the programmer. It is also more declarative as we can more easily identify why a programmer wants to disable a number of widgets: it is to disable the invocation of a particular method.

As performing the action on all the possible invocations of a method may be too strong, e.g. we may only want to disable the keyboard shortcut, we can also disable (or enable) an invoker, along with its selectors.

We can as well go to the lowest level and disable or enable a particular selector. More about these runtime modification can be read on page 150.

6.2 Compilation Scheme

We have already pointed out that Visto is translated to TkGofer. In section 3.10 we discussed some implementation issues that were relevant for the object system, but we didn't sketch the entire translation process yet. As we have now presented all the Visto features in a BNF-like manner, we can finally give an overview of the compilation from Visto to TkGofer.

There is a rather direct mapping from a Visto interface to a TkGofer data type, and from a Visto object to a TkGofer value. The Visto objects that implement a particular interface become TkGofer values of the corresponding data type.

6.2.1 From Interface to Data Type

An interface contains only the name of the interface and the types of the various methods. The name of the TkGofer data type is the name of the interface, and there are of course fields for the methods in their translated form, but we must also accommodate the name and the state of the object. As different objects of the same interface can have a different state type, we define a generic data type with a parameterised state type. This can then be instantiated according to the different concrete state types.

We furthermore must take care that we can correctly implement the delegation and triggering mechanisms. Therefore we also keep in the object the name of the object that it may expand, its superobject, and the names of the objects that expand it, for whom it is the super object.

Therefore, schematically, any interface is translated to

$data\ <InterfaceName>\ a$	
$= <InterfaceName>$	
$String\ a$	— name of object, and type variable
$(Maybe\ String)$	— name of superObject, if present
$[String]$	— objects that expand this object
$\ldots$	— Some GUI related fields
$<MethodType>$	— Such a field per method
$\ldots$	

This was already discussed in section 3.10.2, but we didn't present the GUI fields yet.

A first element is the fact that a Visto object can be a *visual* object. In that case we must have something on which the object can draw, i.e. a Label or a Canvas. Therefore we also have fields of type $(Maybe\ Visualiser)$ for the label or the canvas and $(Maybe\ (<interfaceName>\ a\ \rightarrow\ [Objects] \rightarrow\ Visualiser\ \rightarrow\ GUI()))$ for the draw method. The *Maybe* is present in both types because there *may* be such a *Visualiser* and draw method. The type of the draw method is a bit more complicated. Of course we need the object itself (the $<interfaceName>\ a$ parameter) to retrieve some information from the object, i.e. the data that must be displayed. We also need the list of all objects $[Objects]$, not to retrieve information from, but to redraw their state when they expand this object, and thus are registered as views of this object. We then recursively apply the draw method on these views and their views. The *Visualiser* argument is the window we draw on, and the result is $GUI\ ()$, a monadic action that redraws the window.

Second is the fact that we also provide commands for dynamically changing the interface, e.g. disabling all the invokers and selectors for some method. We thus must have access to these invokers and selectors and provide another field in the data type for the interface. This field has type *ObjectSelectors*.

We could define these GUI fields when we translate the Visto object to a TkGofer value. This is a good approach when the objects are final, but the problem is that they can be cloned at run time.

In that case we cannot copy the GUI elements over, as this would mean that the cloned object would redraw its state on the same canvas as the original object, and that the same GUI-action would have an effect on both objects, e.g. clicking on the *Bold* button in one window of a word processor would put the text in bold face in all windows.

This is naturally not the intention of cloning. Therefore we add another field in the data type that contains a function that initialises those GUI fields. We evaluate the function when the object comes alive, and whenever its clone is created. We then have identical, but independent visualisers, invokers and selectors.

The type of this initialisation function is $(String \rightarrow < interfaceName > a \rightarrow MVar\ [Objects] \rightarrow GUI\ (< interfaceName > a))$. It takes

- a name for the new object,
- the object to initialise the GUI fields of (in the case of run time cloning, this is already a clone of the original object, with default GUI fields),
- the mutable monadic variable containing all the objects, *MVar* [*Objects*], such that it can update the initialised object,
- and it returns $GUI\ (< interfaceName > a)$, a monadic action, that when executed, performs some GUI actions, such as drawing the visualiser and the invokers and selectors on the screen, and returns the initialised object.

Apart from all these fields, we also define functions for each interface that select individual fields from the data type and that update fields. This is now *free* in Haskell, but wasn't available in Gofer, and thus neither in TkGofer. While we're at it, we also define some convenient methods to add and remove views of an object to use when new clones are created or objects removed from the system.

As an example, the interface *BinaryTransformer* from the calculator example in section 4.6.1 is translated as follows:

```
interface BinaryTransformer
  addFunction :: (Int → Int → Int) → (Int → Int)
  addValue :: Int → Int
  applyFunction :: Int
data BinaryTransformer a
   = BinaryTransformer
      String a    —name and type variable
      (Maybe String)    —name of model (superObject), if present
      [String]    —shareHolders (objects that share this one)
      (Maybe Visualiser)
      (Maybe (BinaryTransformer a → [Objects] → Visualiser → GUI ()))
        —maybe Draw method
      (String → BinaryTransformer a
          → MVar [Objects] → GUI (BinaryTransformer a ))
        —GUI initialiser
      ObjectSelectors    —dynamic behaviour for GUI elements
        —and finally the methods
      (Bool, (Int → Int → Int) → BinaryTransformer a
          → [Objects] → Maybe Univ → (GUI (), Univ, [Objects]))
      (Bool, BinaryTransformer a
          → [Objects] → Maybe Univ → (GUI (), Univ, [Objects]))
      (Bool, Int → BinaryTransformer a
          → [Objects] → Maybe Univ → (GUI (), Univ, [Objects]))
```

A further complication is the fact that the name of a method can be shared by many interfaces. In that case we take care that the name is also shared in the translated code and define type classes for those shared methods.

6.2.2 From Object to Value

When we translate the object implementing a particular interface, we must provide values for all the fields of the derived data type.

Most of these fields are rather trivial, except for the translation of a method, to which we devote a separate section, and the GUI initialisation function which is too technical to present in this compilation overview.

The remaining fields are the name of the superobject, and the names of the subobjects (the views). The superobject is easy to retrieve as the definition of a expansion contains the name of the object it expands, but for the subobjects we must first parse the entire program to find all the subobjects for each object. Therefore the translation of a Visto program to TkGofer is a two-pass process.

6.2.3 Translating a Method

We already discussed in sections 3.10.1 and 3.10.3 that the type of a translated method is

$$\begin{aligned}(&\mathit{Bool}, \mathit{arg}_1 \to \mathit{arg}_2 \to \ldots \mathit{arg}_n \to <\textit{the own object}> \\ &\quad \to [\mathit{Objects}] \to (\mathit{Maybe\ Univ}) \\ &\quad \to (\mathit{GUI}\ (), \mathit{Univ}, [\mathit{Objects}]))\end{aligned}$$

When translating the method, we must of course provide all the extra arguments, but another important factor is that a lot of implicit information is present in the definition of a method. For instance the keywords *state* and *superState*, and the fact that when sending a message to another object all other objects must be taken into account.

Therefore we add the bindings of *state* and *superState* to the let of the method, before the original bindings of the Visto code.

At the end of the Visto code we must update the object with its new state and insert it in the list of all objects. And when the object has a draw method, this method must be executed just before returning from the method. Finally, the return value must be inserted in the universal data type *Univ*.

Hence, translating a simple method definition with only some Haskell bindings is not very difficult.

```
λ args → let
           bindings
         in
         (returnValue, newState)
```

```
$\lambda$ args objs lazySuper $\rightarrow$
  let
    state = getState < interfaceName > o
    superObject = < find object in list objs >
    superState = getState < superInterfaceName > superObject

    bindings

    < insert newState in object >
    < replace old object in objs with new object >
    thisdraw = < perform drawMethod >
  in
  (thisdraw, injectUniv returnValue, newobjects)
```

This is just a simplified sketch. We must also take care of the interface commands (p. 150). As they have a side-effect, they are also of type *GUI* (), and must be added to *thisdraw*, using sequential composition in the *do* command, with now a slightly more difficult translation.

```
$\lambda$ args $\rightarrow$ let
              bindings1
              interfaceCommand1
              interfaceCommand2
              bindings2
              ...
            in
            (returnValue, newState)

$\lambda$ args objs lazySuper $\rightarrow$
  let
    state = getState < interfaceName > o
    superObject = < find object in list objs >
    superState = getState < superInterfaceName > superObject

    bindings1
    $visto GUI'$ = interfaceCommand1
    $visto GUI''$ = interfaceCommand2
    bindings2
    ...

    < insert newState in object >
    < replace old object in objs with new object >
    thisdraw = < perform drawMethod >
    thisio = do $visto GUI'; visto GUI''$; thisdraw
  in
  (thisio, injectUniv returnValue, newobjects)
```

The message sending mechanism complicates this even more. In the translated code sending a message, is equivalent to evaluating the TkGofer function that corresponds to the Visto method, but apart from the normal arguments, we must also supply it with the current list of all objects and a value for the 'lazy' *superValue*. The return value in the translated code is different from the return value in the Visto code as well. Although in Visto there is only a simple return value, it actually has three components in the translated code:

- some monadic actions, such as redraws and changes to the interface,
- the actual return value, injected in the universal data type.
- and the updated list of all objects.

All this must be taken into account properly, as in the following sketch.

```
λ args → let
            bindings1
            return1 = object1!message1 args1
            bindings2
            return2 = object2!message2 args2
            ...
         in
         (returnValue, newState)
```

```
λ args objs lazySuper →
   let
     state = getState < interfaceName > o
     superObject = < find object in list objs >
     superState = getState < superInterfaceName > superObject

     bindings1
     object1' = < find object1 in list objs >
     (gui', return1xxx, objs') = message1 args1 object1' objs Nothing
     return1 = < retrieve from return1xxx (of type Univ) the actual return value >
     bindings2
     object2'' = < find object2 in list objs' >
     (gui'', return2xxx, objs'') = message2 args2 object2'' objs' Nothing
     return2 = < retrieve from return1xxx (of type Univ) the actual return value >
     ...
      < insert newState in object >
      < replace old object in objs with new object >
     thisdraw = < perform draw Method >
     thisio = do gui'; gui''; thisdraw
   in
   (thisio, injectUniv returnValue, newobjects)
```

In the same way, the dynamic creation and removal of objects is implemented. Just like sending a message, this changes the set of objects, and has a monadic side effect of type *GUI* () when the new objects are initialised, displaying their invokers, selectors and visualisers, or when these elements are removed when an object is removed.

Another matter is the *superValue* that is used when delegating a message to the superobject. As we want this value to be calculated exactly once, we added the parameter *lazySuper* to the translated method function.

Initially this value is *Nothing*. When the first method needs the *superValue*, it is calculated in the superobject. The superobject then returns the value to the calling object, but it may need to trigger some other objects as well. Their translated method functions are consequently evaluated, but instead of using *Nothing* as the *lazySuper* argument, we use *Just* < *real superValue* >.

When the method now needs the *superValue*, we can see that it has been calculated, as *lazySuper* is no longer *Nothing* and use the value instead of delegating to the superobject again (read also section 3.10.3).

The binding of *delegateResult* = *superValue* is translated to:

```
(gui', superValueUniv, objs') =
    if isJust lazySuper then
       (done, fromJust lazySuper, objs)
    else
       calculateSuper (< name of superobject), < name of method >) objs
superValue = < extract superValue from universal type >
delegateResult = superValue
```

Both in this calculation of the *superValue* and when handling a message to an object, we must extract an actual value from the universal data type *Univ*. Although we used type classes to write general extraction functions, they can only be applied when the compiler can find out the exact return type of the extraction function, otherwise a *Unresolved Ambiguity Error* is triggered. To avoid this, we insert the actual type of the return value and thus must help the type checker.

6.2.4 The Interpretation Function

Finally, after the interfaces and the objects have been translated, we must initiate the actions. Because we cannot have global variables with destructive updating in TkGofer, an interpretation function must maintain the list of active objects.

In this interpretation function we evaluate the initialisation function of all the objects, such that they can display their GUI elements. We also put the list of all objects in a monadic mutable variable and pass this variable to each object.

After that initialisation, all action is triggered by the user interface. Whenever some invoker is activated, we must evaluate the method that the invoker is associated with. TkGofer uses the call back mechanism, and thus the call back of the invoker must apply a message of some object. We have written a function *applyMessage* that does precisely that. It retrieves the list of objects from the monadic mutable variable, performs the necessary I/O and stores the updated list of objects back into the monadic variable.

Another important generic function is the function *calculateSuper* that delegates a message to the superobject. Its implementation is rather straightforward.

Finally, we must also take care of the inherited methods. The problem is that a method always contains a reference in its type to the interface that the method is used in. A method that is inherited thus cannot simply be copied to the inheriting interface, as that method would then contain a reference to the wrong interface. We need to convert it. The conversion is not too difficult and can be automated. The conversion function can even be written genericly, but TkGofer's type system cannot cope with the genericity of its type.

Therefore for each inheritance from one interface to another (*adaptInterface* or *expandInterface*), we must define a specific conversion function for that relationship. This specific function is still generic enough to be applied on every method that is inherited.

The part of the Visto program with the Haskell code of course must not be translated, but can be easily inserted in the translated code. Therefore it lives in the same name space as the translated object code. A programmer who is aware of the translation process thus can use Visto objects and methods in his Haskell functions, but that is not the intention of our system. Anything that uses object features must be in the object part of the Visto program. The functional Haskell code remains 100% functional.

A general comment about this translation scheme is that it is mostly lexical, inserting the right additions at the correct places with little checks on the semantic correctness of the code, except for some simpler tests as the existence of objects and methods that are used. It is only after loading the translated code into the TkGofer system that most type and semantic errors are signalled. This is of course a weakness for a commercial system, but we considered it acceptable for an experimental system that Visto still is.

6.3 Conclusion

All of Visto's features have been presented again in this chapter, but this time in a top-down manner, with references to sections in the thesis where the feature is discussed in more detail.

Again we could see that Visto has covered a broad spectrum of different fields within computer science: functional and declarative programming, prototype and class based object orientation, design (MVC and patterns) and GUI programming.

We also presented a general overview of how Visto code is translated to TkGofer. It is a rough and ready, basic translation, but it is functional (sic).

Chapter 7

Conclusion

In this thesis we presented the ways in which we think a user interface can be designed and developed in a more declarative way. Our work originates in the functional community, but many of our ideas can be applied in other domains as well.

We started with the smallest building block of a user interface in **chapter 2**: the widget, which is a typical user interface element, such as a button, an entry or a pull down menu. We have given an historic evolution of the definition of such widgets in various GUI-toolkits. These toolkits follow the strong trend of standardisation in user interfaces and thus put more and more default behaviour in their widgets, although – of course – they still allow the developer to adapt a widget to his personal needs.

However, we argued that using widgets as the sole constituents of user interfaces is far from declarative. We should first consider the separate subgoals of a GUI. One such goal is *selecting* values.

This idea has been stressed by Jeff Johnson in his selectors paradigm and applied by us with an implementation in Fudgets and TkGofer, two completely different libraries for defining a graphical user interface in a lazy functional language. Our selectors are very flexible as the set of data that the user has to choose from can be easily changed at run time.

Hence, we made a strong contribution to the process of defining more generic GUI components, as our selectors are generic, both in the set of data they can select, and – especially innovating – in the appearance of the selector, e.g. a button row, pull down menu or slider bar.

However, selectors are only a partial solution, as not all interface behaviour can be defined as selecting something. Other forms may even be quasi-impossible to define in declarative terms only. Furthermore no toolkit can ever provide all possible appearances for selectors. Therefore one must always allow the developer to define an alternative, personal style for each selector.

Nevertheless, we feel that such selectors should be implemented for all user interface toolkits as they precisely describe what values the user interface must select, and make life a lot easier for the developer when the standard appearances for these selectors are defined in the toolkit.

A very promising extension, however untouched, is that of defining strategies that decide which presentation forms of the selectors to use, or even defining strategies that define such a strategy depending on e.g. the speed of the underlying hardware, the size of targeted display, etc., as indicated on p. 58.

Next, in chapter 3 and 4 we presented our solution that goes beyond the simple application of *selectors*. It requires *an object system* that has been discussed in **chapter 3**.

Our object system is prototype based, contrary to the more popular class based approach. Visto has proven the feasibility of an extensive *prototype based* object system, based on functional programming. Several easy and intuitive forms for deriving new objects from existing ones – often referred to as *inheritance* – have been presented.

A really unique feature is the **expand** derivation. We presented its delegation and triggering principles. They are vital for user interface design, but can also be of great use in 'normal' systems because it allows the sharing of information between different objects interacting closely together, such as the view and the model.

We have shown how our object system allows an easy implementation of some important patterns from the Design Patterns book, where especially the combination of prototype objects and higher order functions proved interesting.

We also discussed a number of design alternatives and give an insight in some implementation details.

Because the final object system turned out to be very powerful and full of features, we devoted a separate chapter to it, although it has been developed in parallel with the GUI system for Visto.

Considered apart from user interface programming, the Visto object language seems all right. It does most things other object languages can, and introduces the concept of *interfaces* to prototype based languages, and the powerful *expand* derivation.

We however also outlined some of the weaknesses of Visto, such as the absence of higher order objects, that is, objects are currently no first class: objects cannot contain objects, and methods cannot take or return objects. Future work should definitely focus on that.

In retrospect we feel that the progress in our work and consequently the design of Visto has been driven too much by implementation aspects. Some features have influenced by whether it was difficult, or easy, to implement. The run time cloning of objects is an example of this as it was difficult to implement clones with a run time definition of the method definition. We can still motivate why we chose run time clones the way they are chosen, but nevertheless we admit that being driven by implementation aspects is in general no perfect approach. However this may be a typical problem of this kind of research where the implementation is developed along with the language design.

The GUI system was the subject of **chapter 4**. The programmer wishing to define a user interface for a Haskell program, must first identify the visualising objects that are relevant to the end user and define methods for those objects, using the object system from chapter 3.

That object program fully describes the potential of the user interface. It is then 'only' a matter of describing what parts will be presented in the user interface. We have illustrated how the Visto programmer can associate those methods with GUI invokers, indicating that this particular functionality will be presented in the user interface. The invokers use our selectors from the second chapter to gather the arguments for the methods. This way the GUI is defined in a goal oriented way: *"To activate this or that method, use this or that invoker."*, versus the call back approach that describes an effect: *"When this widget is used, that action takes place."*

This has a number of advantages. As the GUI is implemented in Visto through a more declarative, goal oriented description, we offer the developer and the end-user a better understanding of what (methods) the GUI is capable of.

Secondly, this description can be used to check the functional equivalence of user interfaces. The Visto program shows which methods can be activated from within the user interface. When two interfaces enable the same methods, they essentially offer the same functionality, although the invokers may be instantiated differently. This way it can become easier to quantify the preference of one interface over another, e.g. an interface might be less beautiful, but preferably because it offers more functionality, or conversely, when two interfaces offer the same functionality, it is clear that the visually more enticing and more usable interface is to be preferred.

The advantages of the descriptive nature of the final program are clear, but, although we presented a fine theoretical approach for developing such Visto programs, it can be difficult to apply this pattern to actual GUI's. The pattern states that the developer defines GUI-invokers for those methods that the GUI must support, and then instantiates the invokers. However, the set of methods supported by the GUI, depends on the set itself and on the actual instantiations, e.g. when the available space is limited, options must be removed, and vice versa, when an initial layout is not satisfactory, some options may be added, e.g. to make sure that the window is appropriately filled.

We outlined a strategy for solving this problem, but it hasn't been tested and thoroughly examined yet. This is another part of future work.

Our approach for defining the GUI with invokers and selectors also meant that we had to define an alternative for the traditional *Model-View-Controller* (MVC) mechanism used in most user interface toolkits. We agree on the role of the *view* as a separate object responsible for displaying information about the main object. Visto provides strong support for it through the *expand* derivation of the object system, and the presence of a *draw* method that is automatically executed whenever the visualising object is requested to execute a method.

There is also little discussion about the role of the model, although we tend to differentiate between the user's model and the programmer's model.

The real difference however lies in the controller. The controller of MVC typically is an object that controls all the events that occur on some particular area of the screen. We don't use such controllers, but rather *activators* that define in what ways methods of the model object can be activated. This is again the goal oriented description of the user interface that we referred to earlier in this conclusion.

We contrasted our ideas with the traditional call back approach and have shown that Visto describes in a proper – declarative – way the functionality of the user interface, that it points out neatly what functionality is triggered from the user interface, and that several typical modifications to the user interface have little impact on the rest of the application. Only the invoker definitions and/or the layout have to be changed to change the user interface. The rest of the Visto objects can stay as they are.

We feel that the Visto methodology of concentrating on the visual objects, defining methods for manipulating that (displayed) information and supplying GUI-invokers for the methods, using the very declarative *selectors*, opens a brand new perspective on user interface design.

However, as the *draw* methods are the only place where drawing commands can be used, and as the *draw* methods are only executed after evaluating a method in the object or in the superobject(s), it is rather difficult to define objects with very dynamic drawing features.

Future work that focuses on that aspect, will probably change the purely functional style of the method definitions to monadic style. Then a lot of Visto can change as well, such as the number of state variables and the way in which the object's state is updated. This will result in a major redesign of Visto. We seriously doubt whether this requires changes to other fundamental characteristics of Visto such as its prototype nature, the expand derivation, the invokers and selectors, ... We are confident that this new Visto will keep its declarative solutions to some of the problems we outlined, but of course that remains an (half-)open question until the future work is effectively implemented.

Another task for the future work force is the development of the GUI builder where the set of objects and their methods with invokers and selectors must be presented to the interface designer in a handy and efficient way. Little research has been spent on that area.

The practical advantage of all these concepts has been shown in **chapter 5**, a case study in which we implemented a small *Geographical Information system* in five systems: Visto, Tcl/Tk, Java, Clean and Fudgets. We applied some typical changes to each of the user interfaces and compared the impact.

The Visto program proved to be superior in all fields, with also a very simple to develop initial system. It shows that Visto precisely fulfills our research goals: allowing the declarative development of a user interface that can be easily changed.

Finally, in **chapter 6** we revisited Visto in a more technical manner, capturing a simplified BNF-syntax of Visto and explaining the translation process from Visto to TkGofer.

This overview of our solutions proves that we have worked on a broad problem domain, from standard functional programming to the development of a fresh prototype based object language on top of Haskell, and from simply comparing and understanding the various GUI systems for Haskell – and their problems – to the implementation of a more declarative alternative for call back functions, over advances in GUI toolkits with the higher level interface components that selectors are.

Despite Visto's weaknesses and the fact that we couldn't delve as deep into each path as we may have wanted, we at least hope that the declarative ideas of this research will influence other user interface system builders and increase the overall declarativeness of such systems.

Finally, we also know that user interface research will never end. The application domain is continually expanding as computers become capable of doing things previously unimagined. We may know when we started, but we'll never know when our *search for the holy grail of declarativeness* in the field of designing and implementing graphical user interfaces ends . . .

Appendix A

Introduction to Functional Programming

A.1 Related Material

Many fine and extensive introductions and tutorials to functional programming and Haskell exist. We do not pretend to give an extensive introduction here, but rather a first taste of what functional programming is about. For a more thorough understanding we refer to other papers:

- A very interesting paper about the *conception and evolution* of functional programming is given by Paul Hudak [35].
- Some questions & answers about functional programming and Haskell can be found on *http://www.haskell.org/aboutHaskell.html*.
- The *Introduction to Functional Programming using Haskell* [9] is a more thorough introduction for readers not familiar with functional programming at all.
- *The gentle introduction* [36] on the other hand is more for those who already know a bit about functional programming (e.g. Lisp, Scheme or ML) but not about Haskell specifically.
- The last of our selection is the *Haskell Companion* [75] which is structured around specific topics.

A.2 Functional Programming Concepts

Functional programming, as the name already suggests, is centred around *functions*. Each expression is a function call that only depends on its arguments and that returns one result. So in a pure functional language [1] as Haskell, different calls to the same function with the same arguments will always produce the same result.

This notion forms the base of the concept of *referential transparency* which states that *"The meaning of an entity is unchanged when a part of the entity is replaced with an equal part."* For example, consider the expression

$\ldots x + x \ldots$
where $x = f\ a$

As we state that x equals $f\ a$ we can safely replace $\ldots x + x \ldots$ in the expression by $\ldots (f\ a) + (f\ a) \ldots$ without changing the return value of the expression. The same cannot generally be said in an imperative language, because the application of $f\ a$ may rely on some global variables that can change between different applications of f. So replacing x by $f\ a$ is not a safe operation in a language that is not referentially transparent.

Although the notion of referential transparency may seem like a simple idea, the clean equational reasoning that it allows is very powerful, not only for reasoning formally about programs but also informally in writing and debugging programs. [35]

Let's now take a look at another example that shows how recursion is used in Haskell to implement 'loops'.

$\mathit{fac}\ 1 = 1$
$\mathit{fac}\ (n + 1) = (n + 1) * \mathit{fac}\ n$

This definition of the factorial function is an almost direct translation from the mathematical equation $n! = n * (n - 1)!$. So this can be considered very *declarative* as it expresses precisely *what* should be calculated without having to explain in what way which variables must be updated.

The evaluation mechanism is that of term rewriting and reduction strategies. The expression $\mathit{fac}\ 3$ is reduced to $3 * (\mathit{fac}\ 2)$ which is reduced to $3 * (2 * (\mathit{fac}\ 1))$ and $3 * (2 * 1)$ where the recursion ends and reduction continues to deliver 6 as the result of $\mathit{fac}\ 3$.

[1] A language is considered *pure* if it is free of side effects such as assignments. Languages as Scheme and ML are functional, but not *pure*.

An extra asset of Haskell, not automatically a feature of all functional languages, is the notion of *typeful* programming.[2] Although our factorial function didn't contain explicit type information, the type inference algorithm (derived from Milner/Mycroft's one [47, 49]) can infer the type for $\mathit{fac} :: \mathit{Int} \rightarrow \mathit{Int}$.[3]

Haskell is strongly typed, which means that the compiler must be able to give each function a fixed type. This makes types are very pervasive in Haskell programs. Because of the richness of the type system it is often said that "*Typed programs can't go wrong.*". It is at least true that the errors that remain in a program once it successfully passed the type checker, are almost always errors to the *logic* of the program such as a faulty algorithm, and not simple – time consuming – programming accidents.

Part of the *typeful* experience in Haskell are the *user defined data types*. It is extremely easy to introduce new data types for your own application domain, e.g.

$$\begin{aligned}&\textbf{data}\ \mathit{Human} = \mathit{Male}\ \mathit{String}\\ &\qquad\qquad\ \mid \mathit{Female}\ \mathit{String}\end{aligned}$$

A *Human* is either a *Male*, characterised by his name, which is a *String*, or a *Female* also characterised by her name. Haskell data types can also be generic, containing type variables that can be substituted for concrete types when the data type is actually used. If we want to define the type for a binary tree containing values of *any* type *a*, we can do this eloquently using a recursive type definition:

$$\begin{aligned}&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a\\ &\qquad\qquad \mid \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\end{aligned}$$

We may now be able to write programs that are extremely declarative and that directly correspond to mathematical definitions, but unfortunately the efficiency of an algorithm remains vitally important. Consider for example the trivial implementation of the Fibonnacci function, which 'describes' the population growth of a family of rabbits:

$$\begin{aligned}&\mathit{fib}\ 1 = 1\\ &\mathit{fib}\ 2 = 1\\ &\mathit{fib}\ n = \mathit{fib}\ (n - 1) + \mathit{fib}\ (n - 2)\end{aligned}$$

This example is correct and executable, but unfortunately runs in exponential time and space. The compiler cannot transform this into an efficient program. In such cases it may be better to consider an alternative algorithm, e.g. using an

[2]A phrase due to Luca Cardelli

[3]Actually, the type is $\mathit{fac} :: (\mathit{Num}\ a) \Rightarrow a \rightarrow a$. This says that fac can be applied to any type *a* that belongs to the *type class Num*. We will not introduce type classes here and thus simplifi ed the actual type of fac a bit.

accumulative parameter. The following *fib* program is a lot better in performance than the previous trivial program.

$$\begin{array}{l}
\mathit{fib}\ 1\ =\ 1\\
\mathit{fib}\ 2\ =\ 1\\
\mathit{fib}\ n\ =\ \mathit{fibAcc}\ n\ 1\ 0\\
\mathit{fibAcc}\ 1\ \mathit{acc}\ \_^{4}\ =\ \mathit{acc}\\
\mathit{fibAcc}\ n\ \mathit{acc}_1\ \mathit{acc}_2\ =\ \mathit{fibAcc}\ (n\ -\ 1)\ (\mathit{acc}_1\ +\ \mathit{acc}_2)\ \mathit{acc}_1
\end{array}$$

Functional programs are typically very powerful when it comes to list processing. Using *pattern matching* (e.g. in the *quicksort* example below we match the argument against either the empty list [] or $(x : xs)$, the list with x as the head of the list, and *xs* as the tail – the rest – of the list) and *list comprehensions*, we can give a definition of *quicksort* in Haskell that can be considered far more declarative than the equivalent Pascal or Java procedure. The imperative quicksort must extensively spell out *how* the ordered sequence is to be calculated through various variable manipulations, whereas the Haskell program for the most part says *what* to do: append the ordered sequence of smaller elements before the ordered sequence of larger elements.

$$\begin{array}{ll}
\mathit{quicksort}\ [\,] & = [\,]\\
\mathit{quicksort}\ (x : xs) & = \mathit{quicksort}\ [y\ \mid\ y\ \leftarrow\ xs,\ y\ <\ x]\\
 & \quad +\!\!+\ (x : (\mathit{quicksort}\ [y\ \mid\ y\ \leftarrow\ xs,\ y\ >\ x]))
\end{array}$$

List comprehensions come from set theory. They restrain the elements that should appear in the list, e.g. in $[y\ \mid\ y\ \leftarrow\ xs,\ y\ <\ x]$ the y's that appear in the list, come from the list *xs* and must be smaller than x. Another fine example is $[\,(x,y) \mid x\ \leftarrow\ [1\,..\,7],\ y\ \leftarrow\ [1\,..\,x],\ x+y\ <\ 10]$. This calculates all pairs of numbers whose sum is less than 10, and whose constituents are integers between 1 and 7. We find list comprehensions one of the elements that makes *fun*-ctional programming *fun* and at the same time very declarative.

Naturally *functional* programming is strong at *functions*. Just as simple as we can create new numerical values, e.g $x\ *\ y$, can we create new functions. Suppose that we want to define a function that takes an x and returns the square root of x, then we can do this using *lambda abstractions.* The function $(\lambda\, x\ \rightarrow\ x\ *\ x)$ maps any x on $(x\ *\ x)$.

The concept of *higher order functions*, the fact that functions are first class, is another very powerful mechanism present in all functional programming languages and thus also in Haskell. We have seen simple functions as *fac* and *fib* that take integers and return integers, but functions can also take other functions as arguments, and return functions as a result of their computation.

[4]We use an underscore _ to denote an argument whose value is of no importance in this defi nition.

Consider for example the function *doBoth* that takes two functions f and g, and a value x, and returns a tuple $(f\ x,\ g\ x)$. This function can be simply be implemented as $doBoth\ f\ g\ x\ =\ (f\ x,\ g\ x)$ and the inferer is able to find the matching *polymorphic* type $doBoth\ ::\ (a\ \rightarrow\ b)\ \rightarrow\ (a\ \rightarrow\ c)\ \rightarrow\ a\ \rightarrow\ (b, c)$: it takes a function from a to b, another function from a to c, a value of type a and returns a tuple of type (b, c). The type is polymorphic because it matches many different applications. The only constraint is that the type of the first argument of the functions f and g must be identical to the type of the value x, which is no more than could be expected, as both f and g are to be applied on x.

However note that f and g don't have to be functions with just *one* argument. Haskell functions are always considered to be *curried* functions. E.g. the function application[5] $(+)\ 3\ 17$ is considered to be an application of $(+\ 3)$, the function that adds 3 to its argument, on 17. So $doBoth\ (+\ 3)\ (+)\ 8$ is a valid application that returns the pair $(11,\ (+\ 8))$ with the integer value 11 on the first position of the pair and the function that adds 8 to its argument on the second position.

Higher order functions and currying are orthogonal features, but they often occur together. Most often higher order programming is used to exploit general patterns present in most programs, such as a function that is applied to every element of a list. e.g. the national tax administration will apply the tax calculation function on every resident of the country, and the university will apply an evaluation function on every student's record to calculate his/her results.

This *map* function can be easily defined in Haskell:

$$\begin{array}{ll} map & ::\ (a\ \rightarrow\ b)\ \rightarrow\ [a]\ \rightarrow\ [b] \\ map\ f\ [] & = [] \\ map\ f\ (x : xs) & = (f\ x) : (map\ f\ xs) \end{array}$$

Other popular higher order functions include

- $filter$ that takes a predicate function[6] and a list, and returns only those values of the list that conform to the predicate, e.g. *filter odd* $[1,2,3,4,5]$ returns $[1,3,5]$,
- $foldl$ and $foldr$ apply a function accumulatively on a list of values, e.g. *foldl* $(+)\ 0\ [1,2,3,4,5]$ adds up all the elements of the list and returns 15.

The Haskell Companion[75] describes a strong case in favour of higher order functions: " *In 1979 a researcher named Richard Waters did a study of the Fortran Scientific Subroutine Library which was a major package of many thousands of lines of code in common use at the time. He determined that if Fortran were capable of supporting three basic high-order functions: map, filter and fold, roughly 60 percent of the code in the library could be rewritten as calls to them. This would have significantly reduced the size of the library and also means that optimisations*

[5](+) is the prefi x notation of the infi x + function.

[6]A predicate is a function that returns a boolean value, either false or true.

in the code produced for just these functions would have widespread benefit. “

Finally, a feature present in Haskell, but certainly not in all functional languages, is *laziness* – call by need –. A language can be called *lazy* when an argument of a function isn't evaluated until it is actually needed, contrarily to a *strict* language that always evaluates its arguments, even when they're not needed.

```
f x y z = if x > 0 then
            x + y
          else
            y + z
```

The call $f\ 3\ 4\ (/\ 0\ 0)$ will succeed in a lazy language because only 3 and 4 are needed, but not in a strict language because that would really evaluate $(/\ 0\ 0)$ and end with a fatal *division by 0* error. It has been shown [9] that programs that successfully evaluate in a strict language, will also evaluate successfully in a lazy language with the same return value, so lazyness does add expressiveness and strength to a language.

Laziness becomes especially interesting when used with lists. Here it also means that the elements of a list will only be produced when some other function consumes those values. So we can safely write programs with (potentially) infinite lists when we only need a finite part, e.g. $\mathit{take}\ 5\ [x \mid x \leftarrow [100..],\ \mathit{odd}\ x]$[7] returns a list with the 5 first odd elements greater than 100.

A fine example that illustrates the expressive power of laziness is the implementation of a function *primes* that returns the prime numbers by using Erathothenes' *sieve*. Erathothenes noted that you can generate the list of primes by starting with the list of all integers greater than or equal to two, and then repeatedly removing the first element and sieving the remainder of the list through that prime (i.e. removing all multiples of the prime) (fig. A.1).

— primes	— remaining list
ϕ	[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...]
[2]	[3,5,7,9,11,13,15,...]
[2,3]	[5,7,11,13,...]
[2,3,5]	[7,9,11,13,...]
...	
[2,3,5,7,11,13,17]	[19,23,29,...]

Figure A.1: The *sieve* algorithm applied.

This is expressed in the same way in the following Haskell program. We start with $[2\ ..]$ the list of all integers starting from 2, and *sieve* that list. $\mathit{sieve}\ (a : x)$

[7] *take n xs* takes the fi rst *n* elements from the list *xs*.

takes a, and continues sieving with $[y \mid y \leftarrow xs, y \text{ `}mod\text{` } a > 0]$, [8] all elements y that come from xs and that are no multiple of a ($y \text{ `}mod\text{` } a > 0$).

$$\begin{aligned}&primes :: [Int]\\ &primes = sieve\,[2\,..]\\ &sieve\,(a : xs) = a : sieve\,[y \mid y \leftarrow xs, y \text{ `}mod\text{` } a > 0]\end{aligned}$$

These concepts, referential transparency, recursion, types, pattern matching, list comprehensions, higher order functions and laziness, and their examples finely illustrate the Haskell programming style. For a more detailed introduction we refer again to the literature (e.g. *www.haskell.org.* and [35, 9, 36, 75]).

A.3 Functional I/O and GUI

So far we have only presented a number of simple but powerful constructs. Of course, Haskell contains more features than we presented and it should certainly not be considered a toy language. It has been applied in a broad range of domains and functional programming has proven to be a valuable tool for developing general purpose software, not only in the field of research and education [33] but also for large scale applications such as telecommunication (Erlang [5]), natural language processing (LOLITA [39]) and year 2000 systems such as AnnoDomini and LS/2000. The web pages *http://www.haskell.org/practice.html* and *http://www.cs.bell-labs.com/∼wadler/realworld/* offer a more elaborate selection.

There are good reasons why these developers chose functional programming: through the course of history, the paradigm has grown mature, efficient compilers have been built and many special or general purpose libraries exist.

Also in the field of graphical user interfaces for Haskell many successful systems have been built. Fudgets [16] and the Clean IO-system [1] are the two most noticeable initiators. Apart from Haggis [24] and systems that rely on and translate to Tcl/Tk [60], such as TkGofer [88], TclHaskell [22] and Haskell-Tk [43], most later systems build upon Fudget features. In the chapter 2 we will give a short introduction to many of the functional GUI systems, and a somewhat longer introduction to Clean and Fudgets.

However, we start off with an historical overview of the different ways in which I/O has been expressed in functional languages. Although reading this is not absolutely necessary to understand this thesis, it gives an insight in some of the reasons why the inventors of Clean and Fudgets had to look for new ways for expressing GUI's, an advanced form of I/O.

Trying to categorise the two initiating GUI systems, we can say that Clean is like the environment passing style (sec. A.3.2); Fudgets uses elements of streams

[8] $x \text{ `}f\text{` } y$ is infi x notation for $f\ x\ y$.

(sec. A.3.4) and continuations (sec. A.3.3), although the implementation later moved to monadic style (sec. A.3.5).

Our research project Visto – for *VI*sual *S*tate *T*ransforming *O*bjects – will be discussed in the main text of this thesis and uses the object oriented technology. Although it could therefore be considered a separate methodology with side effecting operations, it has always been developed with the functional community in mind. It is intended to be used in a functional environment, defining a user interface for an (existing) Haskell program and expressing the non side effecting operations in Haskell. Furthermore, Visto code is translated to TkGofer and the translator itself is written in Haskell.

A.3.1 No Imperative I/O

On the first impulse we may be tempted to provide side effecting primitives similar to I/O functions in imperative languages such as Pascal or C:

$$\mathit{getChar} :: \mathit{Char}\ \mathit{putChar} :: \mathit{Char} \to \mathit{Void}$$

The *getChar* function waits for a key to be pressed and returns the corresponding character, while the function *putChar* simply takes a character and prints it on *stdout*, returning a value of type *Void*, actually indicating that it returns *nothing (interesting)*. However, such functions are not referentially transparent, which is not allowed in a pure language. Different calls of *getChar* may return different characters, thus breaking the rule that “equals can be replaced by equals”. $x == x\ \mathbf{where}\ x = \mathit{getChar}$ is not equal to $\mathit{getChar} == \mathit{getChar}$ (unless of course the user accidently pressed the same key twice), because each application of *getChar* will poll for a new key press.

A.3.2 Environment Passing Style

The problem with the previous approach is *implicity*. It is implicit in the function *getChar* that it accepts keys at different moments: the first key press is followed by a second one a few milliseconds later, and a third one perhaps minutes later, possibly as a response to feedback of the system.

Implicit is the fact that the *world* changed. Either because the user already pressed a key, or because the system told him some interesting news. Naturally the input of the user depends on this world on the ‘outside’ of the computer.

In a pure, referentially transparent functional language, the result of a function only depends on its arguments. So, if we want to take the world into account we have to add it to the function:

$$\begin{aligned}&\mathit{getChar} :: \mathit{World} \to (\mathit{Char}, \mathit{World})\\&\mathit{putChar} :: \mathit{Char} \to \mathit{World} \to (\mathit{Void}, \mathit{World})\end{aligned}$$

getChar now correctly reflects that it depends on the 'outside' world, returning the character that was pressed and a new state for the world. Of course, in the data type *World* we don't have to keep the *entire* world, but only the I/O devices of the computer, as they are the channels that connect the computer to the world.

This may seem like a clear and rather simple model, but it forces us to pass the world around in a single threaded and strictly sequential way from I/O function to I/O function such that the changes to the world are correctly described. This poses restrictions on both the programmer and the compiler that has to check whether the programmer follows the restrictions.

With a function $getList\ ::\ World \rightarrow ([a], World)$ that gets a list of user inputs, the definition of a function that appends two lists retrieved from the input, would be:

```
inputAndAppend world = (append list1 list2, world'')
  where
    (list1, world') = getList world
    (list2, world'') = getList world'
```

We certainly cannot duplicate the world in a program, because it is technically impossible to instantly duplicate the screen and keyboard. Neither can we throw away a new instance of the world, because we cannot return to the old state of the world, unless we can erase printed characters from our laser printers or undo the pressing of a key by the user, which is of course impossible. When retrieving the second list of user inputs, the *noGoodInputAndAppend* program fails to take into account that a first list has already been retrieved.

```
noGoodInputAndAppend world = (append list1 list2, world)
  where
    (list1, _) = getList world
    (list2, _) = getList world
```

The notion of *uniqueness* is adopted by Clean, that uses environment passing, to ensure that the world is not duplicated and only passed around uniquely and in a single threaded way. The compiler is then able to check that at any given time only a unique instance of the world exists.

A.3.3 Continuations

Another solution for making sure that the program continues in the right order is *continuation style*. This technique has once been very popular in functional programming to model state updating and I/O, and is still used a lot in partial evaluation. For I/O however it has been replaced by monadic style (sec. A.3.5).

Continuations are particularly well suited to tackle to problem of failures: what if the I/O can't proceed, for example because there is no printer attached, or because of communication problems.

The point is that instead of a simple function $f :: a \to b$ taking an a and returning a b, we give the function two continuation arguments of type $b \to c$: a *fail continuation* that will be used when something went wrong, and a *success continuation* for when things were OK.

$$f :: a \to (b \to c) \to (b \to c) \to c$$

The b is still being calculated, but instead of being returned directly, it is passed to one of the continuations, and the result of the continuation is returned.

Let's take a look at an example. Suppose there exists a continuation function $readInput :: (Message \to IO) \to (String \to IO) \to IO$ as well as a function $writeOutput :: String \to (Message \to IO) \to IO \to IO$. *readInput* reads a String from *stdin*. If something goes wrong, the message that explains what went wrong is passed to the fail continuation. When successful the string is passed to the success continuation. *writeOutput* takes a String and when successful it simply returns the success continuation, that doesn't take any arguments because *writeOutput* isn't supposed to return anything. Using the typical continuation style, a *main* function that reads from *stdin* and writes to *stdout* can now be written as:[9]

$$\begin{aligned}
&main :: IO \\
&main = readInput \quad errorExit\,(\lambda\, s \to \\
&\qquad writeOutput\; s\; errorExit\; (\\
&\qquad done\,))
\end{aligned}$$

Although this clearly reflects the order of evaluation, this style doesn't excel in readability.

A.3.4 Streams/Dialogues

Unfortunately worse exists. The streams solution is even more difficult to understand, and people often don't seem to trust the approach because it is so counter-intuitive. It is however one of the first solutions – if not the first – for handling I/O in a pure, lazy functional language, and therefore deserves a place here.

The idea behind the streams I/O is that the program and operating system (O.S.) communicate with each other. In this *dialogue* the O.S. receives and handles requests. Typical examples of requests are "Let the user enter a character on the keyboard." and "Display this string on the screen." The O.S. then forwards the responses from the user, e.g. "The user pressed F1." or "String successfully printed on the screen." The O.S. receives *Requests* and delivers *Responses*.

The program, the other participant in this dialogue, of course must do this the other way: it creates the *Requests* for the O.S. and accepts the *Responses* from the O.S. Therefore the type of the dialogue is:

[9] errorExit takes a *Message* and aborts the program with a error message. *done* simply ends the program.

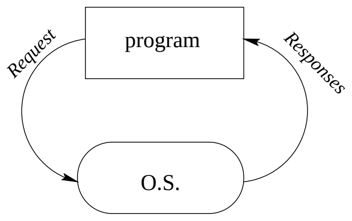


Figure A.2: The Streams Model.

type *Dialogue* $=$ [*Response*] $\rightarrow$ [*Request*]

In a strict language this would be quite complicated. We would need to define the responses before any requests are sent out. So laziness is here an absolute need. This way the program can already start creating the list [*Request*] without needing an element from [*Response*]. Both lists [*Response*] and [*Request*] aren't standard lists, but rather lazy *streams* to which elements are dynamically added as the program runs.

Suppose the following definitions:

```
data Request = ReadChan Chan
             | AppendChan Chan String
             | ReadFile File
             | WriteFile File String
```

```
data Response = Success
              | Str String
              | Failure Message
```

In a Haskell dialogue each response corresponds to one request. A very simple program that only reads from *stdin* then looks like

```
main :: Dialogue
main [Str s] = [ReadChan stdin]
```

In this program we already start from the assumption that we will receive a string from the request *ReadChan stdin* that we're sending to the O.S. This is OK if we make sure that the output is sent *before* the response is being received. Therefore we must use *lazy patterns*. By adding a $\sim$ before the pattern, we indicate that the match will always succeed (if accidently it doesn't, the program will abort with an error), and that matching must be postponed until something is available. Laziness is thus explicitly called for.

$$main\ \sim(Str\ s : \_) \ = \ [ReadChan\ stdin]$$

Let's conclude this quick overview of the stream/dialogue approach by an example that also outputs the string that was read:

$$\begin{aligned} main\ \sim(Str\ s :\sim(Success : \_)) \ = [\ & ReadChan\ stdin, \\ & AppendChan\ stdout\ s \\ ]& \end{aligned}$$

This program still relies on the *Success* of *AppendChan.* Other Haskell techniques can be used to accommodate for that. We will not discuss those here. The point is that this approach requires a thorough understanding of the lazy evaluation mechanism. Also, continuously having to use the lazy pattern syntax $\sim$ quickly becomes tedious. Finally, as the responses appear in the argument list of the function, and the requests in the result, response and request are always very distant from each other. This even deteriorates when there's more I/O as the distance then increases more and more.

Therefore soon other alternatives were presented. As a matter of fact, although we presented them in the other order, continuation style I/O came after this streams model, trying to solve some of the problems present in the streams approach, such as a better handling of failure and success.

A.3.5 Monads

None of the previous solutions was really satisfactory. Fortunately, the advent of *monads* was a big step forward. Although the monadic model relies on rather difficult mathematics, i.e. category theory, currently we don't have to be aware of that theory anymore when we only want to use monads for I/O. In that case it is certainly more easy to explain it without the theory of categories and basic monads, but rather as an abstract data type. After all, monads are a conceptual structure into which I/O *happens* to fit. It is no more necessary to understand monad theory to perform Haskell I/O than it is to understand group theory to do simple arithmetic. Note that it is no simplification from our side to leave out the category theory. All modern text books explain monads without it.

An important notion is the difference between the *return value* of I/O, and the *action* of doing some I/O. Whereas in *pure* functional languages, functions that return values from I/O cannot be defined as they break referential transparency, functions that define actions *can* be defined referentially transparent. After all, the return value may be different but the actions themselves are identical, e.g. *getChar* will always be an *action* that gets a character.

The point is that I/O is now defined as a composition of actions. If we want two applications of *getChar*, we have to combine them into a new action. This new action can then replace the equivalent two consecutive applications of *getChar* everywhere. Thus "equals can be safely replaced by equals" and referential transparency is maintained.

```
getChar :: IO Char
putChar :: Char → IO ()
```

getChar is a function that returns an action that, when performed, returns a *Char*. *putChar* is a function that takes a *Char* and returns an action that, when performed, doesn't return anything interesting. Because every function must return something, () is used.

The two important functions that can bind I/O functions (actions) and application functions are *bind* :: *IO a* → (*a* → *IO b*) → *IO b* and *result* :: *a* → *IO a*. *bind* takes a first action that, when performed, returns an *a*, and a function that given such a value of type *a*, calculates (prepares) another action that, when performed, returns a *b*. *bind* then combines those two actions to a new action that, when performed, finally returns a value of type *b*. *result* can be used to insert a normal value into an action that, when performed, returns that value.

```
main :: IO ()
main = getChar `bind` (λ c1 →
       getChar `bind` (λ c2 →
       putChar c2 `bind` (λ _ →
       putChar c1)))
```

This program reads two characters and writes them in reverse order on *stdout*. This program reflects the order of execution nicely but is still rather hard to read and maintain because of the abundance of lambda-expressions and parenthesis.

Luckily, the *do*-notation provides sweet *syntactic sugar* which allows the same program to be written in a nicer way:

```
main :: IO ()
main = do c1 ← getChar
          c2 ← getChar
          putChar c2
          putChar c1
```

Although this program is still a pure functional program, it now describes I/O in a manner that is familiar to imperative I/O. That is its strength, but at the same time also its weakness because some people feel that this still is like introducing imperativeness to functional programming, although it is wrapped in a purely functional thing, the monad.

We agree with the fact that it can be used to express I/O in a nice, sequential way, but also feel that if we introduce imperative style programming to Haskell, we can also introduce other styles, that may even be more appropriate. Simple textual I/O is rather sequential and imperative, so may be best suited with this monadic style, but for GUI that is more sequentially non-deterministic, we believe that object orientation might be a better alternative. This is what we have explored in our research and is discussed in the main text of this thesis.

Bijlage B

Nederlandse Samenvatting

Visto: Declaratievere Methodes voor GUI-ontwikkeling

Inhoudsopgave

1 Inleiding

Dit proefschrift verhaalt hoe we gezocht hebben naar een meer declaratieve beschrijving van grafische gebruikersgrensvlakken – ongetwijfeld beter bekend onder de Engelse term *Graphical User Interfaces*. In deze beknopte Nederlandse versie zullen we hiervoor steeds de afkorting *GUI* gebruiken.

Zelfs al is de *term* GUI bij het grote publiek redelijk onbekend, het concept is dat helemaal niet. De GUI is dé standaardmanier geworden waarop de gebruiker met een programma werkt. Zelfs nog extremer: veel hedendaagse gebruikers weten niet meer dat er ook zoiets bestaat als tekstuele interfaces, waarin alle commando's heel exact ingetypt moeten worden en waar de gebruikers dus de precieze syntax van al die commando's moeten kennen. Dit is zeker geen nostalgische terugblik naar *de goede, oude tijd*. De GUI kent veel voordelen. Via zijn aanschouwelijke en vrij intuïeve concepten met vensters, knoppen en menuutjes kennen programma's met een goede GUI een veel gemakkelijkere leercurve. Doordat de meeste programma's op een analoge manier werken – ze volgen een welbepaalde stijlgids – is het, ook voor minder ervaren computergebruikers, veel gemakkelijker om nieuwe programma's te leren.

Een innovatie die geïntroduceerd werd door de Xerox Star en snel navolging kreeg van Mac, Amiga, Atari en uiteindelijk ook de hedendaagse marktleider MS Windows, maakte zo de computer heel wat toegankelijker en versnelde op die manier de steeds toenemende informatisering.

Spijtig genoeg is het niet enkel rozegeur en maneschijn. De programma's met een GUI verkopen dan wel pakken meer dan concurrenten met minder of minderwaardige GUI, maar de ontwikkeling van een GUI is wel heel wat moeilijker en dus duurder. Bovendien duurt ze ook langer, wat dan weer een sterk nadeel is in een markt waar vernieuwing – zeker in marketing – het verschil maakt.

Die grotere complexiteit van GUI's maakt ook het onderhoud – een van de grootste kosten in software – zwaarder. Dit proefschrift probeert hieraan te verhelpen door te onderzoeken hoe men op een betere, want meer declaratieve manier GUI's kan ontwikkelen.

Hiertoe beschrijven we eerst wat we verstaan onder *declaratief*. Meer specifiek zullen we ingaan op onze vertrekbasis: het *functioneel programmeren*. We lichten ook even toe hoe men tegenwoordig – in monadische stijl – gewone in- en uitvoer beschrijft in Haskell[62], de standaard functionele taal. Een GUI is een meer geavanceerde vorm van in- en uitvoer en gebeurt dan ook niet gewoon in die monadische stijl. Desalniettemin zou dit alles een goed beeld moeten geven van de toestand zoals ze was toen wij aan ons onderzoek begonnen.

Het spreekt voor zich dat we ons in het eigenlijke onderzoek niet beperkt hebben tot het gebied van functionele talen maar een veel breder overzicht nastreefden.

Een eerste resultaat hiervan zijn *selectors*. Die worden in **de derde sectie** beschreven. Daarin vertrekken we van de elementaire bouwstenen waaruit GUI's bestaan, *widgets*. Typische widgets zijn een individuele knop, een invoerveldje, een menu, ... We bekijken hoe zoiets standaard geconstrueerd wordt in twee bibliotheken voor imperatieve talen en in Clean[1] en Fudgets [16], de twee grote voorgangers van GUI-bibliotheken voor functionele talen. We stellen vast dat widgets in recentere bibliotheken meer en meer standaard gedrag krijgen en tonen aan dat we nog een stapje verder gaan met de notie van *Selectors* zoals uitgewerkt door Jeff Johnson[40] en geïmplementeerd door ons in Fudgets en TkGofer[88].

Die verbetering volstaat echter nog niet. Ze legt wel nieuwe fundamenten, maar als daarop nog steeds hetzelfde gebouw gebouwd wordt, schieten we niet veel op. Aan de basis van onze (ver)nieuwe(nde) constructiemethode ligt een object geörienteerde taal. Die taal, Visto genoemd, van *VIS*ualiserende *T*oestands-veranderende *O*bjecten, is het onderwerp van de **vierde sectie**. Alhoewel klein begonnen is Visto dermate gegroeid dat we het nu kunnen presenteren als een aparte verwezenlijking.

In de **vijfde sectie** tonen we dan hoe aan een Visto object-programma een GUI kan toegevoegd worden. Dit gebeurt – zoals de bedoeling is – op een declaratieve manier, die niet alleen beter beschrijft wat de GUI moet doen, maar die ook gemakkelijk veranderingen aan de GUI opvangt zonder dat er veel aan de rest van het programma moet veranderen. Hiertoe hebben we een alternatieve versie van het traditionele *Model-View-Controller* patroon ontwikkeld, die we dan ook bespreken in deze sectie.

Aan de hand van een testgeval hebben we kunnen ervaren dat Visto het inderdaad beter doet dan Tcl/Tk[60], Java, Clean en Fudgets. We bespreken kort de resultaten van die steekproef.

Finaal volgen natuurlijk nog wat algemene conclusies die bevestigen wat we hier nu al beweren.

2 Declaratief Programmeren

In onze inleiding hebben we al verteld dat het moeilijk is om GUI's te ontwikkelen en dat o.a. het onderhoud een probleem is. Dit geldt echter niet alleen voor de GUI bovenop het programma. Ook het programma zelf is moeilijk om te ontwikkelen. Daarom bestaan er al langer verschillende onderzoeksrichtingen die allen nastreven om dat probleem te verhelpen. *Declaratief programmeren* is hierin een belangrijke vernieuwende factor.

Declaratief programmeren staat tegenover het traditionele *imperatieve* programmeren, waar met opdrachten gewerkt wordt. Die techniek sluit wel beter aan bij de Van Neumann-architectuur van de computer en is dus wel heel geschikt voor de computer, maar daarom is het nog niet de beste programmeertechniek voor mensen.

Nu is het wel moeilijk, zoniet onmogelijk, om een precieze en waterdichte definitie te geven van wanneer een taal al dan niet declaratief is. Gelukkig kunnen we wel talen onderling vergelijken en zeggen dat de ene taal *declaratiever* is dan de andere wanneer ze meer beschrijft *wat* er gebeurt i.p.v. *hoe* het gebeurt.

Neem bijvoorbeeld de *quicksort* methode. Het idee is dat we een lijst kunnen sorteren door de lijst op te splitsen in twee delen, een deel dat alle elementen bevat kleiner dan een bepaalde spilwaarde, en een deel met de grotere waarden. Die twee delen sorteren we dan (op dezelfde manier) en vervolgens plakken we de gesorteerde delen terug achter elkaar om zo de hele lijst gesorteerd te hebben.

Deze schets beschrijft precies *wat* er gebeurt. Een implementatie in eender welke imperatieve taal, Pascal, C, Java, ... zal dit doen via allerlei (ingewikkelde) manipulaties van variabelen met herhalingsinstructies e.d., zodat in zulke programma's uiteindelijk vooral beschreven wordt *hoe* het algoritme uitgewerkt wordt i.p.v. *wat* het conceptueel doet.

```
quicksort []       = []
quicksort (x : xs) = smaller ++ (x : larger)
  where
    smaller = quicksort [y | y ← xs, y < x]
    larger  = quicksort [y | y ← xs, y > x]
```

Figuur B.1: Quicksort in Haskell

Het Haskell programma in figuur B.1 beschrijft precies *wat* er gebeurt en valt dus terecht onder de term *declaratief*.

Een ander (typisch) voorbeeld is de faculteitsfunctie:

$$7! = 7 * 6 * 5 * \ldots * 2 * 1$$
$$n! = n * (n - 1)!$$

Ook hier zal een niet-declaratief, imperatief programma om tot het resultaat te komen variabelen manipuleren, o.a. een teller om alle factoren tussen 1 en n te overlopen, terwijl het Haskell-programma perfect de wiskundige gelijkheid beschrijft.

```
fac 1 = 1
fac n = n * fac (n - 1)
```

Figuur B.2: Faculteit in Haskell

Natuurlijk bevat Haskell nog een heel pak andere, mooie eigenschappen, maar die vallen buiten het bereik van dit beknopt overzicht van het proefschrift. Voor uw intellectuele nieuwsgierigheid verwijzen we in de eerste plaats naar de website *http://www.haskell.org* waar prima inleidende en gevorderde documenten en referenties te vinden zijn.

Voor de volledigheid verwijzen we ook nog even naar een andere tak van declaratief programmeren, het *logisch* programmeren. De benaming impliceert niet dat andere manieren van programmeren niet logisch zijn, maar eerder naar de onderliggende logica. Hierbij werkt men met *relaties* die waar of vals zijn en waaruit een uitspraak over een programma-opdracht (‘query’) wordt afgeleid. Zowel logisch als functioneel programmeren werkt op basis van expressies. Dit in tegenstelling tot imperatief programmeren, dat, zoals we reeds eerder vertelden, opdrachten gebruikt. Terwijl logisch programmeren relaties gebruikt, verkiest het functioneel programmeren *functies*.

Dit proefschrift dient gesitueerd te worden in de *functionele* wereld. Vertrekkende van het declaratieve aspect van het functioneel programmeren proberen we op een declaratieve manier een GUI toe te voegen aan een functioneel programma.

Maar vooraleer we proberen zo ver te springen, zullen we eerst de stap zetten naar gewone in- en uitvoer.

2.1 Monadische in- en uitvoer

Alhoewel er in de loop der tijden verschillende stijlen bestaan hebben om traditionele in- en uitvoer in functionele programma’s te programmeren, bespreken we hier enkel het eindresultaat zoals het nu in de meeste programma’s gebruikt wordt.

Onder traditionele in- en uitvoer verstaan we invoer van het toetsenbord en van bestanden en uitvoer naar scherm, printer en bestanden. De veel complexere GUI wordt slechts zelden direct geprogrammeerd in termen van ‘traditionele’ in- en uitvoer, alhoewel het natuurlijk *kan*. Vooraleer we de GUI bekijken, moeten we eerst inzien welke problemen er zijn bij de ‘gemakkelijke’ in- en uitvoer.

Als we even kijken naar de aanpak in imperatieve talen, kunnen we geneigd zijn om functies *getChar* and *putChar* te voorzien met volgende types:

$$\begin{aligned} &\mathit{getChar} :: \rightarrow \mathit{Char} \\ &\mathit{putChar} :: \mathit{Char} \rightarrow \mathit{Void} \end{aligned}$$

getChar is dan een functie zonder parameters die het teken teruggeeft dat de gebruiker op het toetsenbord indrukte; *putChar* neemt een teken en drukt dat af op het scherm. Het resultaattype, *Void* duidt er op dat er niks interessant als resultaat is. [1]

[1] Behalve natuurlijk het feit dat er iets op het scherm verschijnt, maar dat is een *neveneffect*. Met een resultaat bedoelen we iets dat we kunnen gebruiken in andere functies. Bijvoorbeeld, het resultaat van $3+5$, 8, kunnen we gebruiken om op te tellen bij 13, als in $(3+5)+13$. *putChar* heeft in die zin geen resultaat, *Void* dus.

Het probleem hiermee is dat Haskell een *pure* functionele taal is. Puur betekent dat het resultaat van een functie-oproep *enkel* mag afhangen van de parameters, zodat identieke functie-oproepen altijd identieke resultaten geven. Deze eigenschap noemt men ook *referentiële transparantie* en wordt sterk op prijs gesteld, o.a. omdat het correctheidsbewijzen gemakkelijker maakt en handig is om de compiler efficiënte transformaties uit te laten voeren.

getChar en *putChar* blijken echter *niet* puur te zijn. Een oproep van *getChar* zal niet altijd hetzelfde resultaat geven, omdat de oproep in feite impliciet extra informatie bevat, nl. het feit dat het over opeenvolgende toetsaanslagen gaat. Elke opeenvolgende toetsaanslag komt overeen met een volgende oproep van *getChar*.

Omwille van die impliciete informatie zijn deze functies taboe in pure talen. Essentieel in de oplossing die *monads* daarvoor biedt, is het onderscheid tussen een *resultaatwaarde* en de *actie* die erbij hoort. De resultaatwaarde van *getChar* mag dan wel verschillen van oproep tot oproep, de actie van het opvragen van een toetsaanslag blijft altijd dezelfde, en is enkel afhankelijk van haar argumenten. Zo'n monadische *actie* is dus *wel* referentieel transparant.

Monads zijn gebaseerd op categorie theorie, maar gelukkig moeten we die theorie niet kennen als we monads enkel gebruiken voor in- en uitvoer. Geen enkel modern tekstboek doet het nog op die manier.

$$\begin{aligned}
&\mathit{getCharAction} :: \mathit{IO}\ \mathit{Char}\\
&\mathit{putCharAction} :: \mathit{Char} \rightarrow \mathit{IO}\ ()
\end{aligned}$$

Het feit dat *getCharAction*[2] een monadische actie is, valt af te lezen uit het type *IO Char*. In het algemeen is *IO a* een in-/uitvoer [3] actie die, als de actie uitgevoerd is, een resultaatwaarde van type *a* zal bevatten.

getCharAction is dus een actie die, als ze uitgevoerd is, een *Char* teruggeeft. *putCharAction* neemt een *Char* en geeft een actie die, als ze uitgevoerd is, geen resultaat geeft, of beter gezegd een resultaat dat niet interessant is, nl. $()$.

Merk op dat we steeds praten over een actie die, als ze uitgevoerd wordt, een of ander resultaat zal geven. De vraag is wanneer en hoe zo'n monadische actie effectief zal uitgevoerd worden. Het antwoord is dat de *main* van een Haskell programma [4] van het type *IO a* is. $\mathit{main} = \mathit{getCharAction}$ is bijvoorbeeld een goede hoofdfunctie. Wanneer dit programma gecompileerd is en uitgevoerd wordt, geeft dat programma als resultaat de eerste toetsaanslag terug.

Het is dan eigenlijk enkel kwestie om in de *main* alle acties te steken die we willen doen. Dat betekent dat we primitieve acties zoals *getCharAction* en *putCharAction* moeten combineren tot nieuwe acties, die we opnieuw kunnen combineren met andere, tot we de uiteindelijke hoofdactie bekomen.

[2]Eigenlijk heet deze functie *getChar*, maar we gebruiken hier *getCharAction* om de verwarring met de hoger vermelde, niet-pure functie *getChar* te vermijden.

[3]In-/uitvoer is het Engels Input/Output en daarom wordt *IO* gebruikt.

[4]De *main* is die functie die uitgevoerd wordt wanneer het gecompileerde programma opgestart wordt.

Daartoe hebben we in de eerste plaats volgende twee functies:

$$\begin{array}{ll} \mathit{bind} & :: \mathit{IO}\ a \ \rightarrow\ (a\ \rightarrow\ \mathit{IO}\ b)\ \rightarrow\ \mathit{IO}\ b \\ \mathit{result} & :: a\ \rightarrow\ \mathit{IO}\ a \end{array}$$

Met de eerste functie *bind* kunnen we tweede monadische acties verbinden, en de resultaatwaarde van een eerste actie gebruiken in een tweede. We hebben bijvoorbeeld gezien dat *getCharAction* van type *IOChar* is, en dat *putCharAction* zo'n *Char* neemt en een actie van type *IO* () teruggeeft. Als we dan een toetsaanslag willen inlezen en die op het scherm afdrukken, moeten we de *Char* in de *IO Char* van *getCharAction* binden aan *putCharAction*. Precies dat doet *bind*.

$$\begin{array}{lll} \mathit{main} & = & \mathit{getCharAction}\,'\mathit{bind}'^{5}\ (\lambda\ \mathit{char}\ \rightarrow \\ & & \mathit{putCharAction}\ \mathit{char}) \end{array}$$

$$\begin{array}{lll} \mathit{main}2 & = & \mathit{getCharAction}\,'\mathit{bind}'\ (\lambda\ \mathit{char}1\ \rightarrow \\ & & \mathit{getCharAction}\,'\mathit{bind}'\ (\lambda\ \mathit{char}2\ \rightarrow \\ & & \mathit{putCharAction}\ \mathit{char}2\,'\mathit{bind}'\ (\lambda\ \_\ \rightarrow \\ & & \mathit{putCharAction}\ \mathit{char}1))) \end{array}$$

*main*2 is dan een programma dat twee toetsaanslagen inleest en die in omgekeerde volgorde op het scherm afdrukt.

result dient om een gewone functionele waarde in een monad te steken, zodat we die kunnen gebruiken in de monad-combinaties.

Het kan ondertussen al duidelijk zijn geworden dat deze monadische stijl niet uitblinkt in leesbaarheid. Daarom heeft men de *do*-notatie ontwikkeld die eigenlijk niet meer dan syntactische suiker is voor de vroegere schrijfwijze. Feit is wel dat programma's nu een pak leesbaarder worden:

$$\begin{array}{lll} \mathit{main} & = & \mathbf{do}\ \mathit{char}1\ \leftarrow\ \mathit{getChar} \\ & & \quad\ \mathit{char}2\ \leftarrow\ \mathit{getChar} \\ & & \quad\ \mathit{putChar}\ \mathit{char}2 \\ & & \quad\ \mathit{putChar}\ \mathit{char}1 \end{array}$$

Alhoewel dit programma nog altijd puur functioneel is, beschrijft het nu toch in een volledig imperatieve stijl in- en uitvoer. Dat is een voordeel omdat het een heel bekende stijl is, die voor in- en uitvoer wel degelijk geschikt is omdat ze de volgorde van uitvoering mooi beschrijft, maar tegelijkertijd vinden sommigen dat het 'wringt' om functionele en imperatieve dingen te mengen, zelfs als het verpakt is in in een puur functioneel iets zoals de monad. Of anders geparafraseerd: *"Timeo Danaos, donae ferentes"*.

[5] $a\ 'f'\ b$ is infi x-notatie voor $f\ a\ b$.

Wij zijn vooral van mening dat, als we dan toch al een nieuwe stijl introduceren in Haskell, we evengoed andere, misschien nog betere stijlen kunnen proberen. Eenvoudige, tekstuele in- en uitvoer wordt misschien best beschreven in zo'n sequentiële, imperatieve manier met monads, maar voor GUI's vinden we objectoriëntatie een beter idee. Dat idee hebben we dan ook in ons onderzoek uitgespit. Een neerslag hiervan is te lezen in de volgende delen van dit beknopt overzicht.

3 Selectors

In dit deel bekijken we eerst hoe widgets, zoals gewone knoppen, wisselknoppen[6], invulvelden, menu's, … geconstrueerd worden in typische bibliotheken voor imperatieve talen. We hebben gekozen voor OSF/Motif, een iets oudere bibliotheek voor C/C++ en java's awt, een modernere variant, omdat deze twee bibliotheken populair zijn en een goed beeld geven van het brede spectrum aan imperatieve bibliotheken. Daarna bekijken we hoe dit gebeurt in Clean[1] en Fudgets[16], de twee trendsetters voor een GUI in functionele talen.

Figuur B.3: Voorbeelden van het typisch uitzicht van een wisselknop.

Vervolgens lichten we de evolutie in deze bibliotheken toe en illustreren hoe selectors [40] een stap verder gaan in die richting. Daarna volgt een bespreking van onze implementatie voor Fudgets en TkGofer.

3.1 Imperatieve Bibliotheken

OSF/Motif

OSF/Motif[89] is een zeer populaire bibliotheek voor C/C++ voor de aanmaak van widgets voor X-Windows[71]. De widgets zijn op een object geörienteerde manier geklasseerd zodat ze heel wat gedrag met elkaar kunnen delen. Omdat er in C geen object systeem bestaat, hebben de auteurs zelf zoiets moeten toevoegen aan C.

Om een wisselknop te maken, kan men ofwel de algemene constructor *XtCreateManagedWidget()* gebruiken, ofwel een gespecialiseerde versie *XmCreateToggleButton()*.

Hierbij geeft men dan o.a. de *parent* op. Dit is het venster waarin de knop moet verschijnen. Op die manier komt men tot de hiërarchische structuur van X-Windows. De vensters van X-Windows komen immers niet overeen met de 'echte'

[6]Met wisselknop bedoelen we een knop die ofwel *aan* of *af* staat. Door de knop aan te klikken, wisselt de toestand van de knop van aan naar af, en omgekeerd.

- *button1* = **XtCreateManagedWidget(***name1*, **xmToggleButtonWidgetClass**, *parent*, *argList*, *argCount*)
- *button2* = **XmCreateToggleButton(***parent*, *name2*, *argList*, *argCount*)

Figuur B.4: De constructie van een wisselknop in OSF/Motif

sleepbare vensters zoals we ze kennen uit populaire besturingssystemen als MS-Windows. In X-Windows is een basisvenster niet meer dan een welbepaald gebied (‘venster’) op het scherm met een welbepaalde functionaliteit. Een menu is in die zin een venster dat reageert op een muisklik door een nieuw venster te tonen waaruit de gebruiker een keuze kan maken.

In het algemeen kunnen vensters op andere hogere-niveau-vensters geplaatst worden. De *parent* is hier zo’n hoger-niveau-venster waarop de wisselknop geplaatst wordt.

In de *args* zitten een heleboel opties. Dat is (een deel van) de kracht van OSF/Motif omdat daar echt heel veel opties ingesteld kunnen worden. Zo kan bijvoorbeeld een venster elke willekeurige vorm aannemen en hoeft dit dus niet beperkt te worden tot rechthoeken zoals we gewoon zijn. Het is tegelijkertijd ook een nadeel omdat er zodanig veel opties zijn dat de gebruiker met een hoge leercurve geconfronteerd wordt.

Java

De awt bibliotheek van Java lost dit op door een veel eenvoudigere constructor aan te bieden.

- **Button()** *// Maakt een knop zonder iets erop.*
- **Button(String label)** *// Maakt een knop met een specifiek label.*

Figuur B.5: Twee methodes om een knop aan te maken in java.awt

Natuurlijk kan Java gebruik maken van de object-oriëntatie die al in de programmeertaal ingebakken zit, maar los daarvan is het toch opvallend hoe eenvoudig de constructie geworden is.

Natuurlijk moet men in Java dan nog andere methodes gebruiken om de functionaliteit van de knop aan te passen aan de eigen behoeften, maar een standaard knop aanmaken is wel heel eenvoudig geworden. Dat is dan ook de kracht van awt. Het is heel gemakkelijk om standaard widgets aan te maken.

Daarna moeten we die standaard widget nog (lichtjes) aanpassen tot ze doet wat ze moet doen, maar het grote werk is dan al gedaan. Een voorbeeld van wat meestal nog moeten gebeuren, is het gebruik van de methode *add()* waarmee we een knop op een venster kunnen plaatsen.

Het voordeel is in ieder geval dat de constructie van individuele widgets in Java gemakkelijker is geworden en dat afzonderlijke ontwerpbeslissingen nu ook op afzonderlijke plaatsen in de code voorkomen, wat natuurlijk steeds een goede eigenschap is.

3.2 Clean

Clean[1] is een pure functionele taal die qua syntax wel wat afwijkt van Haskell, maar die er qua stijl heel sterk bij aansluit, behalve dan vooral voor de manier waarop Clean in- en uitvoer doet. I.p.v. met monads (sectie 2.1) te werken, verkiest Clean om *de volledige omgeving* mee te geven van de ene functie naar de volgende functie. Die omgeving bevat dan in principe alle informatie die een invloed uitoefent op het resultaat van de functieoproepen, of voor in- en uitvoer dus in feite de *hele wereld*.

Voor de functies *getChar* die een toetsaanslag inleest, en *putChar* die een letterteken op het scherm schrijft, wordt het type dan

$$\begin{aligned}&\mathit{getChar} :: \mathit{Wereld} \to (\mathit{Char}, \mathit{Wereld})\\&\mathit{putChar} :: \mathit{Char} \to \mathit{Wereld} \to \mathit{Wereld}\end{aligned}$$

Op die manier worden deze functies wél puur. Opeenvolgende oproepen van *getChar* of *putChar* krijgen immers verschillende werelden mee als parameter, en na elke oproep van *getChar* en *putChar* verandert de wereld.

In de praktijk zou het natuurlijk nogal omslachtig worden om overal de hele wereld mee te nemen. Clean 1.3 heeft dan ook functies die een stuk van de wereld afsplitsen. Zo zal bijvoorbeeld *openDialog* een dialoog afsplitsen. In zo'n dialoog worden allerlei widgets, zoals knoppen en invulvelden, in een venster gepresenteerd aan de gebruiker. Na *openDialog* is dan een stukje proces-toestand voor de dialoog afgesplitst van de wereld. De elementen in de dialoog moeten dan enkel rekening houden met hun eigen proces-toestand, wat het systeem heel wat beheersbaarder maakt.

Naast die hoog-niveau widgets zoals een dialoog en een venster, zijn er in Clean natuurlijk ook laag-niveau widgets die op de hoog-niveau widgets geplaatst worden.

varNaam = **ButtonControl** *label optiesLijst*

Figuur B.6: Een knop in Clean 1.3

Opnieuw maakt de constructor een widget met veel standaard gedrag, zodat alle knoppen er gelijkaardig uitzien en dezelfde look-and-feel hebben. Daarnaast zitten er in de *optiesLijst* nog dingen om het uitzicht precies bij te stellen, zoals de breedte van de widget, het lettertype, . . .

3.3 Fudgets

Fudgets [16], van *fu*-nctionele wi-*dgets*, is een bibliotheek voor Haskell. Ze dateert nog van voor de tijd van de monads. Bijgevolg hebben de auteurs een eigen oplossing bedacht voor het probleem van neveneffecten in pure functionele talen: stroom-verwerkers.

Dat betekent dat elke fudget stromen verwerkt: het ontvangt waarden op een invoer-stroom, verwerkt die, en zet (andere) waarden op een uitvoer-stroom. In feite zijn er zelfs twee invoer- en twee uitvoerstromen: hoog niveau, en laag niveau. De laag-niveaustromen zorgen voor de communicatie met X-Windows. Bijvoorbeeld als er op een knop geklikt wordt, ontvangt de laag-niveau-invoerstroom die boodschap, en stuurt de knop-fudget opdrachten op de laag-niveau-uitvoerstroom om de knop visueel te inverteren zodat ze er effectief uitziet als een ingedrukte knop.

In de praktijk dient de programmeur niet met die laag-niveaustromen te werken, enkel met de hoog-niveaustromen.

intInputF *:: F Int Int*

Figuur B.7: Een invoerveld voor gehele getallen in Fudgets

Het algemene type van een fudget is *F a b*, met *a* het type van de waarden op de invoerstroom, en *b* het type van de waarden op de uitvoerstroom. In dit concrete voorbeeld van *intInputF* kan de gebruiker gehele getallen ingeven in een invoerveld. Wanneer hij/zij op enter drukt, wordt de waarde in het invoerveld op de uitvoerstroom gezet. De waarden die op de invoerstroom binnenkomen, vervangen de waarde in het invoerveld. Bijgevolg is het type van de waarden op beide stromen *Int* en is het type van *intInputF* gelijk aan *F Int Int*.

De Fudgets bibliotheek bevat op soortgelijke manier fudgets voor de verschillende soorten widgets. Er bestaan telkens gewone versies, zoals *intInputF*, met heel standaard gedrag en alternatieve versies met een weglatingsteken, zoals *intInputF'*, die een lijst nemen met opties om het gedrag preciezer af te stellen.

In een volledig fudgets programma moet men dan de verschillende in- en uitvoerstromen van de verschillende fudgets aan elkaar koppelen zodat waarden van de ene fudget naar de andere kunnen reizen.

3.4 Selectors

Dit overzicht was weliswaar heel beknopt, maar toch is al vast te stellen dat de widgets in de modernere bibliotheken veel meer standaard gedrag meekrijgen dan in de oudste bibliotheek van deze steekproef OSF/Motif, waar de programmeur zelf nog bijna alles moet instellen.

Diezelfde trend is ook vast te stellen in het geheel van GUI's, waar ook een grote standaardisatie heeft plaatsgevonden. GUI's voor verschillende tekstverwerkers bv. gelijken, zeker qua gedrag en look-and-feel sterk op elkaar. In het verlengde hiervan ligt dan ook de swing klassen van java, waar nog grotere standaard componenten zoals een tabel (*Jtable*) of paneel (*JtabbedPane*) aangeboden worden.

Gemeenschappelijk in al deze bibliotheken blijft het feit dat ze zich direct bezig houden met heel concrete uitzichten zoals een knop, een menu, ... en minder met wat men met die componenten wil bereiken (het declaratieve doel). Als we een GUI zien, denken we niet in termen van de knoppen, menu's en invoervelden die er staan, maar wel in termen van de taken die we willen vervullen. Die taken zijn niet zozeer het drukken op knoppen, maar een belangrijk en veel voorkomend onderdeel van GUI's is wel het *selecteren* van iets, van opdrachten, of van argumenten voor die opdrachten. Als we bijvoorbeeld een uur moeten kiezen, is het belangrijk dat we een keuze moeten maken uit een waarde tussen 0 en 23, en niet zozeer of we op die of die knop moeten drukken.

De keuze van de widgets hierboven is vooral ingegeven door het uitzicht van die widgets, of door *hoe* ze gebruikers toelaten om dingen te kiezen, niet zozeer door *wat* ze kiezen. Immers, of we nu een opdracht selecteren uit een menu, of uit een knoppenbalk, maakt niet echt veel uit. Het is de opdracht die telt, en niet de wijze waarop we ze kozen. Toch beschrijven widgets in de eerste plaats hun uitzicht.

Om die reden zijn de traditionele widgets *niet* declaratief: ze beschrijven niet *wat* er geselecteerd wordt, maar wel *hoe*.

Dat is ook de vaststelling van Jeff Johnson die het idee uitwerkte in *Selectors* [40]. Na een studie van verschillende GUI's kwam hij tot het onderscheid tussen

- *data selectors*, de meest voorkomende vorm. Hierbij kan de gebruiker een waarde selecteren, zoals een getal tussen 10 en 100, of een plaatsnaam uit een lijst van bekende plaatsen.

- *commando selectors*. Uit een serie mogelijke commando's kiest de gebruiker een opdracht. De toepassing voert die opdracht dan uit.

Bij de data selectors zijn er twee vrijheidsgraden:

- Het aantal waarden dat geselecteerd moet worden:

  - Een precies aantal: 1, 4 of 17.
  - Een willekeurig aantal.
  - Of een aantal begrensd door een boven- en ondergrens, bv. tussen 3 en 8, of tussen 1 en 4.

- De keuzemogelijkheden

  - Een precies gedefinieerde verzameling, bv. zwart, geel of rood.
  - Een reeks, bv. 1 tot 100, of 10:00 tot 17:30.
  - Een willekeurige waarde.

Door hieruit te combineren, kunnen zowat alle mogelijke selecties voorzien worden. Deze declaratieve keuze staat dan ook centraal in de selectors-aanpak.

Natuurlijk moeten we daarna nog kiezen *hoe* die selectors precies getoond moeten worden. Dit is afhankelijk van twee dingen:

- De voorstelling van een individuele keuzemogelijkheid.

  Voor een aantal basistypes zoals getallen, booleans, kleuren, tijden, geldhoeveelheden, . . . heeft Johnson een aantal mogelijkheden voorzien. Zo kan een kleur voorgesteld worden als een gekleurd vierkant, of gewoon door zijn naam.

- De voorstelling van de selectiemanier.

  Om één waarde te kiezen uit *n* mogelijkheden kunnen we bijvoorbeeld een rij radioknoppen[7] gebruiken, of ook een menuutje.

Door die twee vrijheidsgraden qua voorstellingswijze te combineren, verkrijgen we opnieuw een heel uitgebreide reeks voorstellingswijzen.

Gecombineerd met de vrijheidsgraden qua keuze, toont dit aan dat selectors een heel breed gamma van selectiemechanismen dekt. Op een soortgelijke wijze worden ook de *commando selectors* behandeld, maar wel iets minder uitgebreid.

Natuurlijk zijn selectors niet het nec-plus-ultra van GUI's, maar wij vinden ze heel interessant omdat ze een subdomein van GUI's vormen waarvan het doel heel gemakkelijk declaratief beschreven kan worden, dit in tegenstelling tot vele andere subdomeinen van GUI's waar het precieze uitzicht (het 'flashy' zijn van de GUI) enorm belangrijk is. Om aan te tonen dat selectors algemeen toepasbaar zijn, hebben we ze geïmplementeerd in zowel Fudgets als TkGofer.

[7]Een rij radioknoppen is een rij knoppen waarvan er maar ´e´en tegelijkertijd ingedrukt kan zijn, zoals de knoppen op een radio om de voorkeurzender te kiezen.

3.5 Onze Implementatie van Selectors

In Fudgets

Het grootste deel van onze eigen selectors is geïmplementeerd in Fudgets. De manier waarop Fudgets werken sluit immers heel goed aan bij selectors. Een waarde die geselecteerd is in een *intInputF* wordt immers op de uitvoerstroom geplaatst, of als er op een knop geklikt is, wordt een *Click* op de uitvoerstroom geplaatst.

Op dezelfde wijze zullen wij de geselecteerde waarde van de Fudget-selector op de uitvoerstroom zetten. Op die manier kunnen onze selectoren probleemloos samenwerken met andere, bestaande Fudgets, en (ingewikkelde) substructuren vervangen. Ze zijn dus niet alleen declaratiever, maar ook gemakkelijker in gebruik.

commanderF Alhoewel er in een functionele taal met hogere orde functies [8] niet echt behoefte is aan de opdeling in commando selectors en data selectors, zoals Jeff Johnson wel doet, hebben we toch één selector gemaakt die specifiek voor functies dient.

Een veel voorkomend patroon is immers datgene waarbij ergens een lokale waarde wordt bijgehouden en de gebruiker uit verschillende opdrachten kan kiezen om die waarde te veranderen. Een goed voorbeeld is een teller, waarbij de gebruiker op knoppen kan drukken om die teller te verhogen, te verlagen of terug op 0 te zetten.

Dat soort patroon zit in *commanderF*.

- $\mathit{commanderF} :: (\mathit{Eq}\,a,\ \mathit{Graphic}\,b,\ \mathit{PresenterCommander}\,d) \Rightarrow$
  $[(a,\ c \to c,\ b)] \to c \to d \to F\ (\mathit{SelectorData}\ a\ b\ c)\ c$

- $\mathit{commanderF}\ [(1,(\lambda x \to x+1),\ \text{"}+1\text{"}),$
  $(2,(\lambda x \to x-1),\ -1\text{"}),$
  $(3,(\lambda x \to 0),\ \text{"}=0\text{"}),$
  $(4,(\lambda x \to x*2),\ \text{"}*2\text{"}),$
  $(5,(\lambda x \to x \wedge 2),\ \text{"}\mathit{kwadrateer}\text{"})]$
  0
  $\mathit{PullDownMenu}$

commanderF neemt een lijst van keuzemogelijkheden. Elke keuze bestaat uit

- Een *sleutel* van type *a*. Aan de hand van de sleutel kunnen we deze keuzemogelijkheid herkennen. Het is dan ook nodig dat we sleutels met elkaar kunnen vergelijken en daarom is *Eq a* vereist.

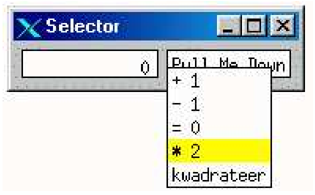
[8]Functies in Haskell zijn hogere orde, omdat functies andere functies als parameter kunnen nemen, en nieuwe functies als resultaat kunnen teruggeven. Een functie staat dus op dezelfde voet als andere data.

- Een functie $c \to c$, die dient om de interne waarde van de selector te veranderen.
- Een voorstellingswijze voor de functie. Die waarde van type b moet in de type klasse *Graphic* zitten.

Daarnaast neemt *commanderF* ook nog een startwaarde c voor de interne toestand, en een parameter d uit de type klasse *PresenterCommander* die specifieert hoe *commanderF* getoond moet worden. Voor *commanderF* hebben we de keuze uit een menu en een knoppenrij of -kolom.

Telkens een functie $c \to c$ gekozen is, plaatsen we de nieuwe interne waarde op de uitvoerstroom van de selector.

In een latere paragraaf vertellen we meer over *SelectorData a b c*, de waarden die op de invoerstroom van de selector mogen verschijnen.



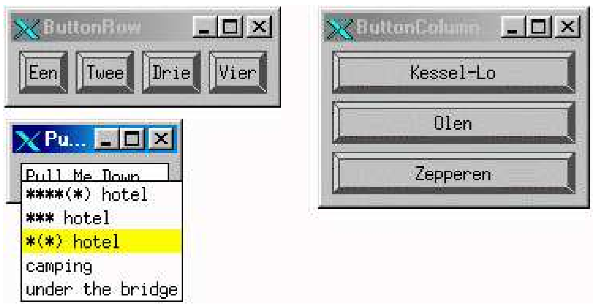
Figuur B.8: Een *commanderF* in de vorm van een menu.

selectOneFromDiscrete Dit is vermoedelijk een van de meest gebruikte selecties. Hierbij moet de gebruiker één waarde kiezen uit een discrete set mogelijkheden.

- $\mathit{selectOneFromDiscrete} :: (\mathit{PresenterOneFromDiscrete}\ c, \mathit{Graphic}\ b, \mathit{Eq}\ a)$
  $\Rightarrow [(a,b)] \to c \to F\ (\mathit{SelectorData}\ a\ b\ d)\ a$

- *selectOneFromDiscrete*
  [(1, "*Een*,"), (2, "*Twee*"), (3, "*Drie*"), (4, "*Vier*")]
  ButtonRow

De constructie gelijkt sterk op die van *commanderF*, maar is nog wat gemakkelijker omdat we geen startwaarde voor een interne toestand moeten opgeven, en vooral omdat we hier geen aparte sleutel moeten opgeven bij elk van onze keuzemogelijkheden. Hier kunnen we onze keuze identificeren aan de hand van de waarde a zelf. Bij *commanderF* kon dit niet omdat we in het algemeen niet kunnen testen of functies gelijk zijn aan elkaar.



Figuur B.9: Verschillende voorstellingen van *selectOneFromDiscrete*.

selectManyFromDiscrete Hierbij kiezen we *verscheidene* waarden uit een keuzeset. We moeten dan ook een boven- en ondergrens opgeven voor het aantal waarden dat geselecteerd dient te worden.

- *selectManyFromDiscrete*
  $:: (\mathit{PresenterManyFromDiscrete}\ c,\ \mathit{Graphic}\ b,\ \mathit{Eq}\ a) \Rightarrow$
  $\mathit{Int} \to \mathit{Int} \to [(a,b)] \to c \to F\ (\mathit{SelectorData}\ a\ b\ d)\ [a]$
- *selectManyFromDiscrete* 1 3
  [(*Sauna*, "*Sauna*"), (*Zwembad*, "*Zwembad*"),
  (*Jacouzzi*, "*Jacouzzi*"), (*Fitness*, "*Fitness*"),
  (*Solarium*, "*Solarium*")]
  PullDownMenu [9]

De uitvoer is nu een lijst van waarden *a* omdat er verschillende waarden geselecteerd worden. De selector controleert of het aantal geselecteerde waarden voldoende is.

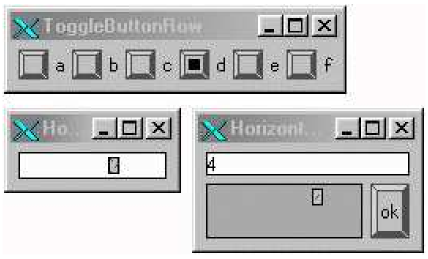
selectOneFromInterval Nu moet de gebruiker een waarde selecteren uit een interval dat gegeven wordt door zijn onder- en bovengrens. We kunnen van het interval dan natuurlijk ook een discrete verzameling maken zoals hierboven. Op die manier zou dit gewoon een speciaal geval zijn van *selectOneFromDiscrete*, maar toch is dat in de realiteit niet zo.

Het feit dat we uit een interval selecteren kunnen we namelijk ten volle uitbuiten, bv. door een schuifbalk te gebruiken. Die toont veel beter het feit dat we uit een interval selecteren.

- *selectOneFromInterval* $:: (\mathit{PresenterOneFromInterval}\ b,\ \mathit{Graphic}\ a,\ \mathit{Enum}\ a)$
  $\Rightarrow a \to a \to F\ (\mathit{SelectorData}\ a\ b\ c)\ a$
- *selectOneFromInterval* '*a*' '*f*' *ToggleButtonRow*

[9]Dit is natuurlijk enkel geldig als er een data type gedefi nieerd is voor *Sauna*, *Zwembad*, ...

Vroeger moesten we paren opgeven van $(waarde, voorstellingswijze)$, maar nu hoeft dat niet meer. Het volstaat om de onder- en bovengrens op te geven. We zullen die waarden tegelijk ook als hun voorstellingswijze gebruiken. De elementen *a* moeten dan wel zelf van de type klasse *Graphic* zijn. De belangrijkste reden waarom we dat hier zo doen, is dat het anders moeilijk kan zijn om te garanderen dat het interval van selecteerbare waarden overeenkomt qua lengte en volgorde met het interval van voorstellingswijzen.



Figuur B.10: Verschillende vormen van *selectOneFromInterval*.

I.p.v. de oude presentatievormen van *selectOneFromDiscrete* te hergebruiken, hebben we een aantal nieuwe vormen uitgeprobeerd. Zo is er de schuifbalk bijgekomen. Daarvan hebben we dan ook direct twee versies gemaakt: een gebufferde en een ongebufferde. De ongebufferde stuurt de gegevens van de schuifbalk direct door als die versleept wordt. Op die manier kan het programma onmiddellijk tonen welke invloed die keuze heeft. Als dat tonen echter rekenintensief is, kan het interessanter zijn om het selectieproces te bufferen. De gebruiker ziet dan wel welke waarde geselecteerd is, maar die wordt pas op de uitvoerstroom geplaatst wanneer hij/zij op de *OK*-knop drukt.

SelectorData a b c Al onze Fudget-selectors aanvaarden elementen van het type *SelectorData a b c* op hun invoerstroom. Hiermee kunnen we hun gedrag tijdens de uitvoering van het programma veranderen.

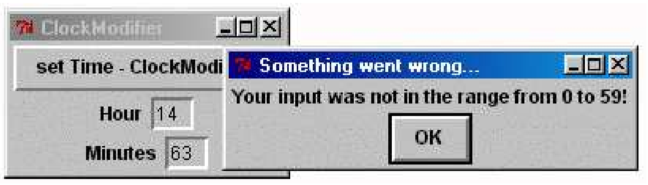
Een veel voorkomende verandering is het toevoegen of schrappen van bepaalde keuzes. Ze is echter niet altijd even gemakkelijk te implementeren omdat dit betekent dat er widgets dynamisch verwijderd en/of toegevoegd moeten worden. Onze selectors doen al dit werk transparant voor de programmeur/gebruiker. De Fudgets-bibliotheek voorziet immers wel de fudget *dynF* die toelaat om de ingebedde fudget tijdens de uitvoering van het programma te veranderen, maar het type en het gebruik ervan is redelijk ingewikkeld. Onze implementatie gebruikt die *dynF*, maar dan wel onzichtbaar, zoals reeds *niet* te zien was in het type van onze selectors.

Daarnaast kunnen we ook nog de volgorde veranderen waarin de individuele elementen getoond worden. Initieel is dit de volgorde zoals ze opgegeven wordt bij de constructie van de selector, maar dit kan veranderd worden in eender welke volgorde: alfabetisch, analfabetisch of bv. voor steden volgens de afstand tot een bepaalde plaats. Dit is volledig onder controle van de gebruiker van onze selectors.

In TkGofer

Na het grote(re) werk in Fudgets, hebben we ook nog een aantal selectors voor TkGofer gemaakt. Net zoals we er bij Fudgets voor zorgden dat onze selectors goed geïntegreerd kunnen worden met bestaande widgets, deden we dit ook in TkGofer. Het belangrijkste verschil is dat bij Fudget-selectors de geselecteerde waarde onmiddellijk op de uitvoerstroom van de Fudget gezet moet worden, maar dat die hier bij TkGofer ter beschikking moet gesteld worden. Pas het toepassen van de functie *getValue* geeft de geselecteerde waarde terug.

Onze aandacht ging in de eerste plaats uit naar nieuwe voorstellingswijzen voor de selectors. Concreet hebben we vooral het invoerveld gebruikt. Dat is immers een vorm die interessant wordt als er heel veel keuzes zijn. I.p.v. dan alle keuzes te tonen in een ellenlange rij knoppen, controleren we de invoer van de gebruiker in het invoerveld op correctheid.



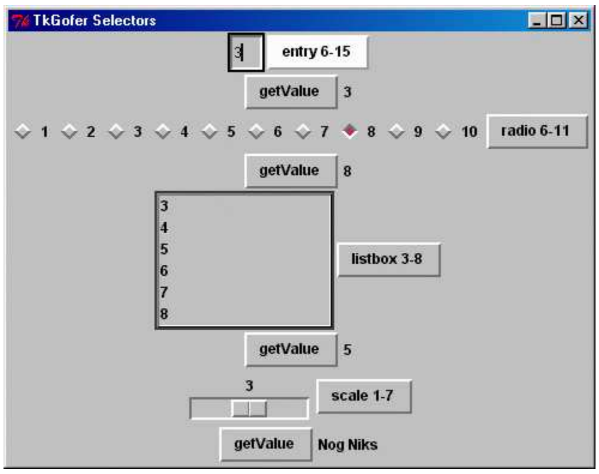
Your input was not in the range from 0 to 59!

Figuur B.11: Een ongeldige waarde in het invoerveld!

Het gebruik van het invoerveld opent tegelijkertijd ook nieuwe perspectieven voor het bepalen van de geldige set waarden. Vroeger moest die expliciet opgesomd worden, maar nu kunnen we ook een geldigheidsfunctie gebruiken, bv. *even* om enkel even waarden toe te laten.

In figuur B.12 zien we vier vormen. De stijl hier is duidelijk verschillend van de Fudgets stijl omdat TkGofer niet enkel onder X-Windows werkt, maar onder voor vele andere besturingssystemen.

Naast al deze algemene selectors, hebben we ook nog een specifieke selector voor *tijd* (figuur B.13) gemaakt. Hiermee kunnen we de tijd selecteren door de wijzers van de klok op het juiste uur te zetten.



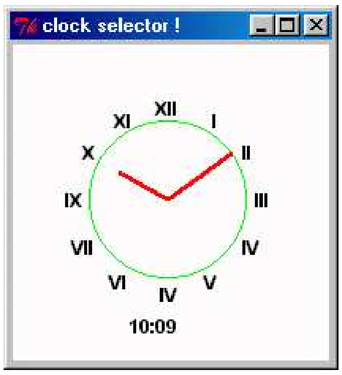
Figuur B.12: Vier vormen van een TkGofer selector.

3.6 Besluit

We hebben gezien dat, in GUI-bibliotheken voor zowel imperatieve als functionele talen de nadruk ligt op de constructie van heel concrete widgets. Toch zijn die niet de bestaansreden van GUI's, maar wel het feit dat we iets met de GUI kunnen verwezenlijken. Tijdens de zoektocht naar dingen die we met GUI's doen, zijn we op een belangrijk subdomein gevallen: het selecteren van gegevens (data) of opdrachten.

We hebben het idee van selectors naar voren geschoven en onze implementatie besproken in Fudgets en TkGofer. Voor beide systemen opgeteld hebben we uiteindelijk een heel pak verschillende selectors met verschillende voorstellingswijzen geïmplementeerd. Deze selectors gaan een stap verder in de trend om standaard componenten aan te bieden omdat ze een aantal veel voorkomende selectiemechanismen blootleggen en er handige GUI-componenten voor aanbieden. Bovendien kunnen onze selectors verschillende soorten gegevens selecteren en dit in verschillende presentatievormen (knoppenrijen, schuifbalken, . . .). Op die manier zijn ze heel breed inzetbaar.

Hiermee hebben we dan een nieuwe, declaratievere bouwsteen voor GUI's voorgesteld. In de volgende delen tonen we aan hoe we ook de bovenste lagen declaratiever kunnen opbouwen.



Figuur B.13: Een klok om de tijd te selecteren.

4 Visto's Objecttaal

4.1 Motivatie

Ons doel is niet louter een declaratievere GUI-*bibliotheek*, die we al in het vorige hoofdstuk bespraken, maar een *volledig declaratief systeem*. Zo'n systeem moet in een zo hoog mogelijke mate *wat*-dingen beschrijven: wat doet de toepassing, wat doet de GUI, welke waarden worden geselecteerd, en veel minder *hoe*-waarden: hoe wordt de toepassing gekoppeld aan de GUI, hoe roepen we dingen op, ...

Een eerste cruciale vraag hierbij is welke functionaliteit men vanuit de GUI moet kunnen oproepen. Omdat Visto toelaat een GUI te maken voor een (bestaand) Haskell-programma zou een eerste optie erin kunnen bestaan om *elke* functie uit het Haskell-programma oproepbaar te maken.

Dat is echter geen zinvolle optie omdat er veel te veel van die functies zijn. Ongetwijfeld zitten er tussen die functies heel wat *hulpfuncties* die we natuurlijk niet aan de eindgebruiker moeten aanbieden. We moeten die uitgebreide set functies dus op een zinvolle wijze beperken. Haskell-modules doen dit al, maar eerder vanuit de visie van een software-bibliotheek. De keuze wordt in de eerste plaats beheerst door het standpunt van de programmeur, niet door het standpunt van de eindgebruiker.

Object-oriëntatie slaagt daar wel beter in, en biedt bovendien het voordeel van een mooi gedistribueerde toestand. Het object dient alleen met zijn eigen toestand rekening te houden wat concreet dan kan overeenkomen met een apart sub-proces van het programma in een apart grafisch venster.

Walter R. Smith van NewtonScript [77] vernoemde reeds de *dichotomie van het programmeren*: programma's met een GUI bevatten meestal twee sterk verschillende componenten: het "model" van de gegevens die gemanipuleerd worden, en de GUI die de manipulaties uitvoert, en *"Dikwijls worden deze verschillende componenten best geprogrammeerd in verschillende programmeerstijlen."* Wij hebben er daarom het volle vertrouwen in dat voor een GUI object-oriëntatie goed moet kunnen samenwerken met de functionele stijl.

Uiteindelijk hebben we voor Visto – voor *Vis*ualiserende *T*oestandsver- anderende *O*bjecten – een *prototype* gebaseerd *object* systeem gekozen.

De belangrijkste redenen voor object-oriëntatie zijn hierboven al aangehaald, nl. het feit dat de methodes van een object precies beschrijven welke functionaliteit een specifiek object aanbiedt, en dat het enkel door die methodes is dat een object van toestand kan veranderen.

Traditionele object-talen gebruiken als basis een imperatieve taal. Door toekenningen aan variabelen en herhalingslussen wordt het gewenste programmaresultaat berekend. Visto onderscheidt zich hiervan door een functionele taal als basis te kiezen, nl. Haskell[62]. Uiteindelijk moeten de methodes een functieresultaat berekenen en dat kan even goed in een functionele taal – of zelfs beter – als in een imperatieve taal. Natuurlijk bevat Visto extra primitieven om objecten met elkaar te kunnen laten communiceren e.d., maar de basis en de *look-and-feel* moeten zo veel mogelijk functioneel blijven.

De meeste object-talen bevatten ook nog extra concepten zoals overerving en polymorfisme om op die manier een (nog) groter hergebruik van code te bewerkstellingen. We zullen tonen hoe Visto ook daarin niet moet onderdoen.

Een groot verschil is wel dat de meeste object-talen *klasse* gebaseerd zijn en Visto daarentegen *prototype* gebaseerd is. In klasse gebaseerde talen staan klassen aan de basis van alles. Vooraleer men een object kan maken, moet er eerst een klasse gedefinieerd zijn, waarin het gedrag van het object volledig beschreven staat – uitgezonderd de inhoud van de toestandsvariabelen van het object. Door de klasse te instantiëren, krijgen die toestandsvariabelen een concrete waarde en begint het object te leven. Omdat alle objecten die geïnstantieerd worden van dezelfde klasse zich identiek gedragen, is zo'n systeem heel handig als er (heel) veel gelijkaardige objecten bestaan in het programma.

Wij maken echter hoog-niveau GUI-objecten. Zulke objecten zullen er nooit veel zijn in eenzelfde programma. In zo'n geval is het veel handiger om rechtstreeks het object aan te kunnen maken en te testen zonder de omweg over de klasse te moeten maken. Object-talen die zo werken, noemt men *prototype gebaseerd*. Geïnspireerd door het succes van NewtonScript [77] en Self[81] hebben ook wij die piste bewandeld.

4.2 Een Eerste Voorbeeld

Laat ons even het archetypische *teller*-voorbeeld bekijken, met daarin een object waarbij een eerste methode de toestand van het object verhoogt, en een tweede methode die toestand terug op haar startwaarde initialiseert.

```
interface Teller
   hoger :: Int
   start  :: Int

object PlusEen Teller Int 0 {
   hoger  = let
                  s' = state + 1
               in
               (s',s'),
   start   = (state , 0)
}
```

Het is een klein voorbeeld, maar het toont toch al een aantal interessante eigenschappen van Visto's objecttaal. Op de vierde regel zien we het sleutelwoord **object** waarmee een object aanmaken.

Vooraleer we dat kunnen, moeten we wel een **interface** definiëren. Die *interface* is niet de klasse van het object. Ze beschrijft enkel de naam en het type van de methodes die een object moet voorzien als het die bepaalde interface wil implementeren.

Elk object moet verplicht een interface implementeren. Het legt de nadruk op het "*programmeren volgens een interface*", zoals gepromoot in het bekende Design Patterns book [29], in tegenstelling tot het (oude) "*programmeren naar de implementatie*" zoals (meer) beschreven in klasse-gebaseerde talen.

Ons object *PlusEen* implementeert de interface *Teller*. Deze interface bevat twee methodes

- *hoger*, die een *Int* teruggeeft,
- *start*, die ook een *Int* teruggeeft.

In de hoofding **object** *PlusEen Teller Int* 0 lezen we verder dat *PlusEen* als toestand een *Int* bijhoudt en dat die begint met de waarde 0.

Daarna volgen de implementaties van *hoger* en *start*.

- Voor *hoger* nemen we eerst de oude toestand **state**, verhogen die met 1 en noemen die s'. Het resultaat van de methode-oproep is dan het koppel (s', s'). In dat koppel staat steeds op de eerste positie de teruggeef-waarde van de methode, in dit geval s', en op de tweede plaats de *nieuwe toestand* voor het object, in dit geval ook s', zijnde de oude toestand verhoogd met 1.
- Voor *start* geven we gewoon de oude toestand als teruggeef-waarde (dus op de eerste positie in het resultaat-koppel), en als nieuwe toestand voor het object (op de tweede positie) de initiële waarde 0.

In de volgende onderdelen zullen we iets dieper ingaan op de *interface* en de aanmaak van een eerste object, en op verschillende manieren om uit bestaande objecten nieuwe te creëren.

4.3 De Aanmaak van een Object

De interface

Het doel van de interface is al vermeld in het inleidende voorbeeld: het opsommen van de methodes en hun types die een object moet implementeren om te voldoen aan die interface.

```
interface Klok
  geefTijd  :: Time
  zetTijd   :: Int → Int → Int → Ordering
```

Het type van zo'n methode kan eender welk geldig Haskell type zijn, m.i.v. zelf gedefinieerde types. Zo'n methode kan dus natuurlijk ook parameters nemen, zoals de methode *zetTijd* die een uur, minuten en seconden neemt.

De *interface* bevat enkel het type van de methodes. Verschillende objecten die eenzelfde interface implementeren mogen dus verschillende implementaties voor een methode geven, zolang het type maar voldoet aan dat gespecifieerd in de interface.

Merk ook op dat in de eigenlijke implementatie elke methode een koppel (*terugkeerwaarde*, *nieuwe objecttoestand*) dient terug te geven, i.p.v. enkel een terugkeerwaarde. Dit wordt echter niet weerspiegeld in de interface, ten eerste omdat het type van de object-toestand dan identiek in alle methodes zou moeten verschijnen, wat redundante informatie introduceert, en ten tweede omdat het type van de toestand van een object eigenlijk niet van belang is voor de interface. De gebruikers van een object kennen immers enkel de interface en zijn dus enkel geïnteresseerd in de teruggeefwaarde. Objecten die voldoen aan dezelfde interface mogen verschillende toestandstypes hebben, en zullen dat ook heel dikwijls effectief doen.

Dat is natuurlijk een belangrijk onderscheid met een klasse definitie. Die bevat dikwijls ook wel het type van elk van de methodes, maar tegelijkertijd ook de implementatie [10]. Bovendien bevat de klasse-definitie ook de toestandsvariabelen. Objecten van dezelfde klasse hebben dus ook dezelfde toestandsvariabelen.

Merk ten slotte op dat dit concept ook in andere talen, zoals bv. Java voorkomt.

De Aanmaak Zelf

Bij de aanmaak van het object zelf moeten we dan de *interface* opgeven die het object implementeert, maar daarnaast moeten we ook een type voor de toestand kiezen.

[10] Tenzij de methode *abstract* is, maar in dat geval kan er geen object afgeleid worden van die klasse.

Wij hebben er voor gekozen om slechts één toestandsvariabele te nemen, dit in tegenstelling tot de meeste imperatieve object-talen die verschillende toestandsvariabelen hebben.

De reden hiervoor is dat we een atomische toestandsverandering beogen, die we dan bewerkstellingen door te eisen dat methodes niet alleen een teruggeefwaarde definieren, maar ook een nieuwe toestand voor het object. De feitelijke resultaatwaarde van een methode is daarom een koppel met daarin die twee waarden, teruggeefwaarde en nieuwe object-toestand.

Indien we verschillende toestandsvariabelen hadden, moesten we ofwel een willekeurig groot n-tupel teruggeven, wat onhandig is, ofwel opeenvolgende toestandsveranderingen doorvoeren, wat we ongewenst vonden.

Tegelijkertijd maakt die unieke toestandsvariabele voor een object het gemakkelijk om in algemene termen te verwijzen naar de toestand van een object. We hebben hiervoor het sleutelwoord **state** gekozen, wat een referentieel transparante verwijzing is naar de toestand van het object *op het begin van de methode-evaluatie.*

Het spreekt voor zich dat de waarde van **state** kan verschillen tussen verschillende methode-oproepen, maar op zijn minst *tijdens* een methode-oproep is de waarde referentieel transparant.

```
object KlokModel Klok Time (Time 10 8 23) {
  geefTijd = (state, state),
  zetTijd  = λ u m s → let
                           t = makeTime u m s
                         in
                         (compare t state, t)
}
```

Het sleutelwoord **object** kondigt de definitie van een object aan. De definitie bestaat uit:

- een naam voor het object – *KlokModel*,
- de interface – *Klok*,
- een type voor de toestand – *Time*,
- een startwaarde voor de toestand – *(Time 10 8 23)*, en
- de implementatie van elk van de methodes vermeld in de interface. Hierin kan men dus gebruik maken van het sleutelwoord **state**.

Communicatie met Andere Objecten. Een essentiële eigenschap van objectsystemen is verder dat objecten met elkaar kunnen communiceren. Hiervoor wordt meestal het boodschappen-protocol gebruikt. Ook Visto doet dat.

Omdat methodes in Visto, net als functies in Haskell, steeds een teruggeefwaarde hebben, en omwille van implementatieredenen, dient zo'n boodschap naar een ander object steeds in een let te gebeuren. Aan de linkerzijde van de let-binding komt de variabele waarin de teruggeefwaarde wordt gestockeerd, aan de rechterzijde de naam van het object waarnaar we het bericht sturen, gevolgd door een uitroepteken en de naam van de methode die we willen oproepen, eventueel nog gevolgd door de nodige parameters:

*returnValue = objectName***!***methodName [params].*

Een teller die enkel 's morgens optelt, kunnen we dan als volgt neerschrijven:

```
object OchtendTeller Teller Int 0 {
   hoger  = let
                nu    = KlokModel ! geefTijd
                vroeg = compare nu (Time 12 00 00)
                s'    = if vroeg == LT then
                            state + 1
                        else
                            state
            in
            (s', s'),
   start  = (state, 0)
}
```

Een object kan van toestand veranderen als het een bericht verwerkt. In die verwerking van het bericht kan de methode opnieuw berichten versturen naar andere objecten, die dan ook van toestand veranderen. In principe kunnen dus alle objecten van toestand veranderd zijn wanneer een eerste bericht verwerkt is, inclusief zelfs het object dat het eerste bericht verzond.

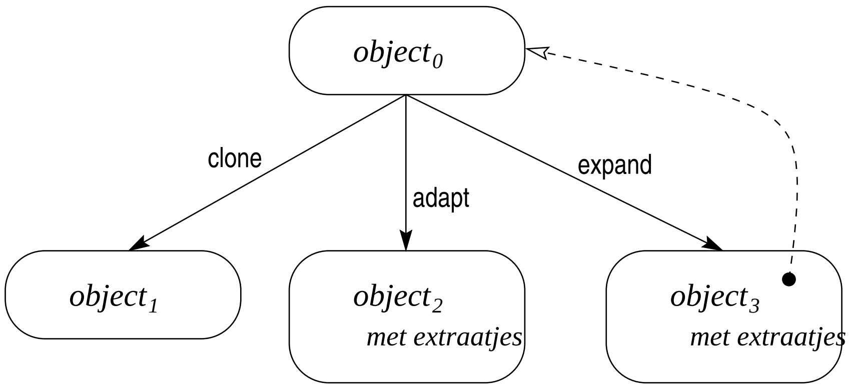
Desondanks blijft het sleutelwoord **state** verwijzen naar de toestand van het object *in het begin* van de evaluatie van de methode, zelfs als dat object tijdens de verwerking van de methode zelf berichten moeten verwerken en op die manier van toestand kan veranderen. **state** refereert dus eigenlijk naar een momentopname van de toestand, terwijl het object zelf intern nog steeds van toestand kan veranderen. **state** verwijst echter niet naar die interne, wijzigende toestand en blijft dus binnen een methode-evaluatie referentieel transparant.

Als men desondanks toch de huidige toestand van het object wil opvragen, moet men maar een methode voorzien die die toestand teruggeeft. Het spreekt voor zich dat men voorzichtig moet zijn met het publiek maken van die methode.

4.4 Nieuwe Objecten Afleiden van Bestaande

In het vorige onderdeel toonden we hoe we objecten vanuit het niets kunnen creëren. Dikwijls willen we ook andere objecten maken naar analogie met bestaande objecten. I.p.v. dat nieuwe object dan vanuit het niets te creëren is het beter om zo veel mogelijk het oude object te hergebruiken, niet alleen vanwege gezonde luiheid – het brengt minder werk mee –, maar ook omdat het zinvol is om expliciet te vermelden dat we iets naar een beeld en gelijkenis maken.

In traditionele klasse gebaseerde systemen wordt dit stelselmatig *overerving* genoemd. Wij doen dit niet om geen verwarring te stichten, aangezien onze afleidingsvormen niet helemaal hetzelfde doen. Overerving blijft natuurlijk wel een zinvol referentiekader.



Figuur B.14: Drie Vormen van Afleiding.

Clone

De eerste vorm is, zoals meestal, de meest eenvoudige. Net zoals in de ’natuur’ met bv. het schaap *Dolly* maken we een kloon die zelfstandig door het leven gaat, maar wel dezelfde eigenschappen heeft als het origineel.

Concreet in Visto betekent dit dat de kloon dezelfde interface implementeert als het origineel. De eenvoudigste kloon heeft ook identiek dezelfde methodes als het origineel, met enkel een andere interne toestand.

clone *PlusEenHoog* **from** *PlusEen* { **state** = 1000 }

PlusEenHoog doet identiek hetzelfde als *PlusEen*, maar vertrekt van een hogere toestand, nl. 1000. Dit is een heel intuïtieve afleiding, zoals een kind dat zegt: *“He, ik wil ook zo’n speelgoedje, maar wel in het blauw!”*

Dit voorbeeld komt ook het meeste overeen met het principe van *instantiatie* in klasse-gebaseerde objecttalen. Alle instanties van een klasse hebben dezelfde methodes, maar wel verschillende waarden voor de toestandsvariabelen.

Omdat onze **clone**-afleiding enkel stelt dat de *interface* moet behouden blijven, hebben we echter een veel krachtiger mechanisme. Het is ook toegelaten om bepaalde methodes te overschrijven:

```
clone Verdubbel from PlusEen {
   state  = 1,
   hoger  = let
                s' = state * 2
            in
            (s', s')
}
```

Het object *Verdubbel* voldoet nog steeds aan de interface *Teller*, maar verdubbelt wel telkens zijn toestand wanneer de methode *hoger* wordt opgeroepen.

De methodes die niet worden overschreven, worden overgenomen – *geërfd* – van het originele object.

Adapt

adapt is dan logischerwijze die afleiding waarbij de interface *niet* behouden blijft. We dienen dan ook eerst een nieuwe interface te definiëren:

```
adaptInterface TweeWegTeller from Teller
   lager  :: Int
   zetOp  :: Int → Int
   _start
```

De nieuwe interface, in dit geval *TweeWegTeller*, erft in principe alle methodes van de vorige interface, en kan er nieuwe methodes aan toevoegen. Dit is zoals de overerving in klasse-gebaseerde talen, maar origineel in Visto is het feit dat we ook methodes kunnen verwijderen, door ze met een lage liggende streep (_) vooraf te laten gaan, zoals hier *_start*.

Dit betekent dus dat objecten niet meer enkel nog incrementeel opgebouwd kunnen worden, maar dat ze soms ook kunnen kleiner worden, *downsizen*, een techniek die zich ook meer en meer doorzet *in het echte leven*, waar bedrijven zich af en toe beperken tot hun kernactiviteiten.

Verdergaand in het voorbeeld van Dolly zouden we een adaptie kunnen maken waarbij een schaap geen melk meer geeft, maar wel haar eigen wol kan breien.

Wanneer we dan effectief een adaptie maken, moeten we vermelden welke interface we hier implementeren en van welk object we vertrekken, gevolgd door de implementatie voor op zijn minst de nieuwe methodes, en eventueel ook de methodes die we willen overschrijven.

```
**adapt** *PlusMinEen TweeWegTeller* **from** *PlusEen*
  $lager$ $=$ **let**
              $s' = \textbf{state} - 1$
          **in**
              $(s', s')$,
  $zetOp = \lambda x \to (\textbf{state}, x)$
  }
```

Het nieuwe object moet niet noodzakelijk hetzelfde type toestand behouden als het originele object. We kunnen gerust een nieuw type en een nieuwe startwaarde definiëren. We moeten dan natuurlijk wel de implementaties van de methodes aanpassen, tenzij beide types compatibel zijn, bv. via Haskell type klassen.

Expand

Ten slotte is er nog de meeste geavanceerde afleidingsvorm, zeker als we het Dolly-voorbeeld volhouden. Het wordt dan zelfs “science-fiction”.

Immers, bij de expansies blijft er wel degelijk een band bestaan tussen het oorspronkelijke object en de afleidingen. Een deel van het gedrag en een deel van de toestand wordt gedeeld. Voor Dolly zouden dit kunnen betekenen dat we schapen maken die wel zelfstandig rondlopen en melk geven of hun wol breien, maar die toch gecontroleerd worden door een centraal ‘brein-orgaan’.

Meer toepasselijk voor computerwetenschappen is het *Decoratie*-patroon uit Design Patterns [29]. Hierbij kan een object gedecoreerd worden met extra eigenschappen, zoals bv. een schuifbalk aan een venster, of een *undo*-functie aan een of ander actie-object. In zo’n geval moet de decoratie, bv. de schuifbalk, een beroep kunnen doen op het oorspronkelijke object.

Dat alles is precies wat onze expansies doen. Vooreerst moeten we natuurlijk een nieuwe interface maken voor onze expansies, net zoals bij *adapt*, omdat een expansie ook methodes kan toevoegen en/of verwijderen.

Gedeelde Toestand Nieuw is alvast dat het nieuwe object nu ook toegang heeft tot de toestand van het origineel. Traditiegetrouw noemen we dat origineel het *superobject*. De toestand ervan kunnen we bijgevolg opvragen met het sleutelwoord **superState**, referentieel transparant zoals **state** in de zin dat het verwijst naar de toestand van het superobject bij aanvang van de methode-evaluatie.

Delegatie en Triggers Daarnaast willen we dat het object en het super-object van elkaar gebruik kunnen maken. Dat betekent alvast dat we *delegatie* moeten voorzien. Delegatie is het principe waarbij het afgeleide object een deel van de verwerking van een boodschap *delegeert* naar het super-object.

Het typische voorbeeld in klasse-gebaseerde talen is dat van een gekleurd punt, dat we bekomen door te erven van een gewoon punt. Als het gekleurd punt een boodschap *beweeg* ontvangt, stuurt het die door naar het gewoon punt.

Dat kan in Visto door gebruik te maken van het sleutelwoord **superValue**. Dat sleutelwoord verwijst naar de teruggeefwaarde van het bericht gedelegeerd naar het superobject.

Maar de medewerking kan ook omgekeerd gebeuren. Een expansie gebeurt transparant. Dat betekent dat een (bestaand) programma gebruik kan maken van het superobject zonder te weten dat er een afgeleide expansie van bestaat. Het kan dus veilig een bericht sturen naar het superobject zonder dat het systeem hierover lastig doet.

Dikwijls echter houdt het afgeleide object een toestand bij die coherent moet zijn met de toestand van het superobject. Als we dan enkel een bericht sturen naar het superobject, zonder dat er er iets gebeurt bij het afgeleide object, kan dat laatste in de problemen komen.

Daarom hebben we ook een eigenschap voorzien, triggers genaamd, waarmee de programmeur kan aangeven dat een afgeleid object geactiveerd wil worden wanneer het superobject een bericht ontvangt. Het superobject *triggert* dan de expansie.

We doen dat door in de methode-definitie **=+** te gebruiken i.p.v. een enkel gelijkheidsteken. Merk dus op dat de verantwoordelijk enkel bij het afgeleide object ligt. We hoeven niets te wijzigen aan het superobject om dit gedrag te bekomen, en dat is maar goed ook.

```
**expandInterface** *ToegangsTeller* **from** *Teller*
  $toegangen :: (Int, Int)$
  $leesTeller :: Int$
**expand** *ToegangPlusEen ToegangsTeller*
        $(Int, Int)\ (0,0)$ **from** *PlusEen* {
   $hoger =+$ **let**
              $returnThis = \mathbf{superValue}$
              $(hogers, starts) = \mathbf{state}$
              $hogers' = hogers + 1$
            **in**
            $(returnThis,\ (hogers', starts)),$
   $start =+ \langle$analoog voor hoger$\rangle,$
   $toegangen = \mathbf{state},$
   $leesTeller = \mathbf{superState}\ \}$
```

Dit voorbeeld toont aan dat *ToegangPlusEen* gebruikt kan worden in de plaats van *PlusEen* en omgekeerd. Een bericht sturen naar *ToegangPlusEen* of *PlusEen* geeft hetzelfde resultaat. Alleen kunnen we via *ToegangPlusEen* ook het aantal keren lezen hoeveel de teller verhoogd is en/of op 0 gezet, en kunnen we de teller lezen zonder die van waarde te veranderen.

4.5 Gedrag tijdens Uitvoering

M.b.v. al deze verschillende manieren om Visto-objecten aan te maken kunnen we een ruime set objecten aanmaken, maar als we dynamisch, tijdens de uitvoering van het programma, geen objecten meer kunnen bijmaken, blijven we wel met een behoorlijk statisch systeem zitten.

Daarom voorzien we ook de mogelijkheid om tijdens uitvoering nieuwe objecten aan te maken. Enige beperking is wel dat we enkel nieuwe klonen toestaan, maar dat is net als bij Haskell types, die we ook niet dynamisch kunnen bijmaken, en bij klasse gebaseerde talen waar we enkel instanties van bestaande klassen kunnen aanmaken.

De opdracht **addClone** maakt een kloon dynamisch aan. Er zijn twee mogelijkheden:

- **addClone** *nieuwObject* **from** *oudObject*
  maakt een nieuw object met dezelfde toestand als het oude object.

- **addClone** *nieuwObject nieuweToestand* **from** *oudObject*
  maakt een nieuw object met een nieuwe toestand. Het type van die toestand moet echter behouden blijven.

Dat is al zeker een interessante toevoeging, maar met de expansies zitten we dikwijls met sterk verweven structuren. Eén los element bijmaken en de rest van die structuur niet overnemen is dan heel gevaarlijk, bijvoorbeeld in het *model-view* patroon. De view overnemen zonder een model is nogal zinloos.

Daarom hebben we ook een extra krachtige versie van *addClone* die de hele expansie-hiërarchie meeneemt. Immers, klonen en adapties van een expansie zijn immers opnieuw expansies. Het volstaat dus niet om één object en één expansie over te nemen, we moeten een hele ketting aflopen. De variant **addClonePlus** doet dat.

- **addClonePlus** *suffix* **from** *oudObject*
  maakt een nieuw object aan, samen met een nieuwe expansie-hiërarchie (als die er is). Alle objecten hebben dezelfde toestand als hun voorvader en hun naam wordt achteraan uitgebreid met de *suffix*.

- **addClonePlus** *suffix nieuweToestand* **from** *oudObject*
  zoals *addClonePlus* met het verschil dat het eerste object een nieuwe toestand krijgt.

Behalve objecten toevoegen, kunnen we er ook expliciet verwijderen. Daarvoor gebruiken we de **killObject**-opdracht. Die verwijdert steeds de hele expansiehiërarchie, omdat we zeker geen wezenloze objecten willen overhouden.

4.6 Besluit

In dit hoofdstuk hebben we, zonder veel te verwijzen naar het GUI-systeem waarvoor Visto ontwikkeld is, de objecttaal van Visto besproken. In feite bewijst alleen al het feit dat we zo weinig verwezen hebben naar de GUI, dat Visto's objecttaal op zich een interessante ontwikkeling belichaamt, maar die vaststelling blijft ook puur objectief (sic) gezien overeind.

Voor zover ons bekend, is Visto het eerste prototype gebaseerde object-systeem bovenop een functionele taal te zijn. Verschillende intuïtieve manieren om nieuwe objecten uit oude aan te maken zijn gepresenteerd. Vooral de **expand**-afleiding met zijn delegatie en triggers is nieuw.

In het volgende hoofdstuk tonen we dan aan hoe we aan zo'n object-programma in Visto een GUI kunnen toevoegen.

5 Een GUI toevoegen aan Visto

5.1 Motivatie

GUI's, en dan vooral *bruikbare* GUI's vormen een belangrijk gebied binnen computerwetenschappen, niet enkel omdat er heel veel ontwikkelingstijd en dus kosten in gestoken worden, maar ook omdat het een doorslaggevend element is bij de bruikbaarheid van een programma.

Terecht wordt er dus veel in geïnvesteerd omdat een minder goed programma met een betere GUI pakken beter verkoopt dan een schitterend product zonder of met een slechte GUI.

Er zijn natuurlijk verschillende redenen waarom GUI's bouwen zo moeilijk is, zoals o.a. aangetoond door Brad A. Myers [51]. Komt daar nog eens bij dat meestal het onderhoud ook heel moeizaam verloopt . . .

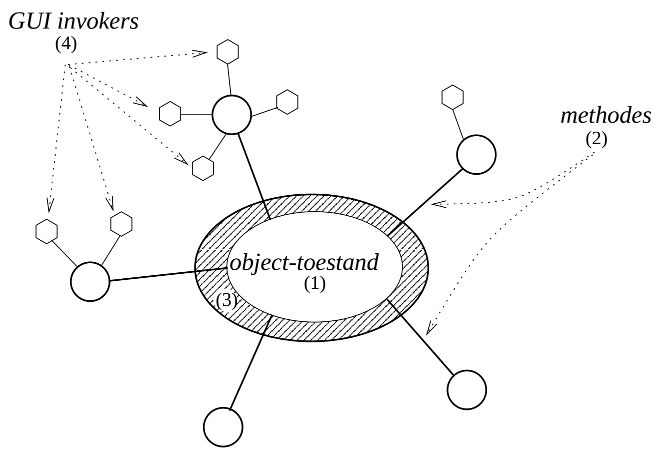
Volgens ons en verschillende andere auteurs [8, 50] is dit een gevolg van het overvloedig gebruik van *call backs*. Met zo'n call back functie wordt traditioneel het verband gelegd tussen de GUI en de toepassing. Elk individueel widget heeft een eigen call back functie die beschrijft wat er gebeurt wanneer dat widget geactiveerd wordt.

Omdat er echter dikwijls honderden, soms zelfs duizenden widgets zijn, verkrijg je dus een echte spaghetti van call backs, en alhoewel lekker in de handen van een goed Italiaans kok, wordt spaghetti-code terecht verfoeid in de informaticasector.

Op zoek dus naar een andere oplossing, zoals wij er een in dit hoofdstuk presenteren. De eerlijkheid gebiedt ons wel om te zeggen dat wij moeilijk een allesomvattende oplossing kunnen aanbieden. Immers, het onderzoek in GUI's is zo rijk en groeit nog elke dag zo sterk, o.a. omdat computers qua rekenkracht nu dingen kunnen die tot voor kort onmogelijk werden geacht. Van ons individueel doctoraatsonderzoek kan men echter moeilijk verwachten dat het alles kan omvatten.

We menen echter wel interessante paden bewandeld te hebben. In dit hoofdstuk verhalen we eerst het model, zoals we dat in een eenvoudige beginsituatie voor ogen hadden. Natuurlijk is het model in de realiteit beduidend groter en krachtiger. De rondleiding in dit ruimere model gebeurt aan de hand van een voorbeeld, waarna we tonen in welke mate dit ontwikkelingsmodel afwijkt van het traditionele *Model-View-Controller* patroon. Vervolgens presenteren we onze conclusie van een ruimere gevalstudie. Hierin hebben we een (zeer) eenvoudig GIS-systeem geïmplementeerd in vijf verschillende systemen waaronder Visto, een aantal typische wijzigingen aangebracht en de impact ervan geëvalueerd. Daarna volgen natuurlijk de conclusies.

5.2 Een Simplistisch Model



Figuur B.15: Een Eenvoudig Model.

Dit eenvoudig model is in de eerste plaats incrementeel.

- Vertrekkende van het puur functionele Haskell-programma maken we *visualiserende* objecten. Deze objecten visualiseren een toestand van het programma die interessant is voor de gebruiker, zoals bv. het tekstdocument dat hij/zij editeert, de webpagina die bekeken wordt, ...

Deze objecten hebben allereerst een interne toestand *(1)* die uit eender welk Haskell type kan bestaan. Ze refereert naar de visuele eenheden die relevant zijn voor de eindgebruiker en niet noodzakelijk naar de logische datastructuren die de programmeur heeft uitgekozen om zijn probleem handig te kunnen modeleren.

- Vervolgens definieert de programmeur de *functionaliteit* van deze objecten, zijnde de methodes *(2)*. Hier moet de programmeur nog niet denken aan de concrete GUI, maar eerder aan acties die (ooit) relevant kunnen zijn voor deze visualiserende objecten.

  Deze verzameling methodes dient beschouwd te worden als *de volledige functionaliteit* van de objecten, zodat het onwaarschijnlijk (of minder waarschijnlijk) wordt dat dit deel nog aangepast moet worden.

  Hoe we die methodes kunnen en moeten implementeren is beschreven in het deel over Visto's objecttaal.

- In de derde faze komt het visualiserende karakter van de Visto-objecten pas echt naar voren. We definiëren een **draw**-methode *(3)* die beschrijft hoe het object getoond moet worden. Visto zorgt ervoor dat deze *draw* wordt uitgevoerd elke keer een methode van het object wordt opgeroepen. Op die manier zijn we automatisch verzekerd van het feit dat de informatie die getoond wordt altijd consistent is met de informatie die aanwezig is in het object.

- Ten slotte moeten we nog bepalen welke methodes in de GUI voorkomen. Hierbij dienen we te kiezen uit de volledige functionaliteit zoals ze omvat is in de methodes van het object.

  De GUI invoceert dus methodes. Ze zal in feite een aantal invokers *(4)* voor de methodes bevatten. We hoeven wel niet voor *elke* methode een invoker te bepalen. We kunnen gerust beslissen dat een bepaalde methode niet echt nuttig is in *deze* GUI. Langs de andere kant kunnen we ook *verschillende* invokers koppelen aan een methode. Hierdoor bieden we alternatieven aan: een beginnende gebruiker zal een knop verkiezen, een meer ervaren gebruiker misschien een toetsencombinatie.

  Als allerlaatste beslissing moeten we dan nog kiezen hoe de invokers er concreet uit zullen zien (knoppen, menu-items, toetsencombinaties, ...) en dit alles samen in een mooie layout gieten.

Dit is natuurlijk een simplistisch model aangezien bv. niet elk object noodzakelijk gevisualiseerd dient te worden. Dit model is ook te rechtlijnig. Het is niet altijd zo dat we eerst kunnen kiezen welke methodes we (rechtstreeks) aanbieden in de GUI en pas dan bepalen hoe we die invokers instantiëren. Dat hangt dikwijls af van de andere instantiaties. Bv. als er te weinig plaats is, of omgekeerd, als de schermlayout er onaf uitziet, gaan we andere instantiaties kiezen en/of meer of minder invokers nemen.

Bovendien is het ongewenst om zowel de inhoud en het gedrag, het uitzicht en de eraan gekoppelde GUI in één object te steken.

Dergelijke afzonderlijke ontwerpbeslissingen worden best afzonderlijk behandeld, en dus ook in verschillende objecten beschreven. Dit kan natuurlijk in Visto, en we zullen dan ook in een volgende sectie op die manier een rondleiding doorheen Visto geven. Langs de andere kant *kan* het ook op de 'foute' manier in Visto, wat wel eens gemakkelijk kan zijn wanneer we snel een prototype willen maken.

5.3 Een Rondleiding met Voorbeeld

In deze rondleiding tonen we hoe we in Visto een programma met een GUI kunnen ontwikkelen. Hierbij zullen we ook (opnieuw) het nut aantonen van onze derivaties zoals de kloon, de adaptie en vooral de expansie.

We beginnen met een mogelijke implementatie van een eenvoudige rekenmachine. We zouden dit zo kunnen ontwerpen dat het logische implementatiemodel overeenkomt met het visualiserende GUI-model, maar dat doen we opzettelijk niet om een algemenere werkstructuur te kunnen tonen, en om te tonen dat zo'n onderscheid helemaal niet voor problemen zorgt.

Het functioneel gedeelte. We veronderstellen een rekenmachine met enkel binaire operaties. Elke operatie heeft dan twee argumenten. In dit model gebruiken we als eerste argument telkens de interne toestand van de rekenmachine. Daarna verwachten we, in infix-volgorde, een functie, een getal als tweede argument, en dan een bevel om de functie uit te rekenen.

Dit schema vinden we terug in volgende *interface*:

```
interface BinaireRekenaar
  voegFunctieToe
        :: (Int → Int → Int) → (Int → Int)
  voegWaardeToe :: Int → Int
  pasFunctieToe :: Int
```

Het type van *voegFunctieToe* toont al aan hoe we de interne toestand van onze binaire rekenaar gebruiken om de functie te curry-en [11] zodat er nog slechts één argument overblijft.

[11] In Haskell worden functies steeds gecurryd. Dat betekent dat elke functie in feite ´e´en argument neemt. Een functie met twee argumenten komt dan overeen met een functie die het eerste argument neemt, en als resultaat een functie teruggeeft die het tweede argument consumeert. Dit principe wordt op die manier uitgebreid tot functies met *n* argumenten.

In de implementatie van een object *RekenMachine* gebruiken we als interne toestand dan niet enkel een getal, maar ook een functie met één argument.

```
object RekenMachine BinaireRekenaar
     (Int, Int → Int) (0, λ x → 0) {
  voegFunctieToe = λ fun → let
                             (value, _) = state
                             newFun = fun value
                           in
                           (newFun, (value, newFun)),
  voegWaardeToe = λ val → let
                             (value, fun) = state
                           in
                           (value, (val, fun)),
  pasFunctieToe  = let
                     (value, fun) = state
                     newValue = fun value
                   in
                   (newValue, (newValue, fun))
}
```

In deze implementatie geeft *voegWaardeToe* de oude getalwaarde in de rekenmachine terug. Als we een alternatieve rekenmachine willen die de nieuwe waarde teruggeeft, kunnen we een *kloon* maken:

```
clone RekenMachine2 from RekenMachine {
  addValue = λ val → let
                        (_, fun) = state
                      in
                      (val, (val, fun))
}
```

Een Visualiserend Object De manier waarop een gebruiker effectief een rekenmachine gebruikt verschilt van bovenstaande methode. I.p.v. een getal in zijn geheel door te geven, typen we cijfer per cijfer in. Pas als het getal volledig is, geven we het door aan de rekenmachine. Dit doen we door op een functie-knop te klikken. We geven dan in feite zowel het getal als de functie door.

Daarna geven we een tweede getal in, dat we pas effectief doorsturen als we de "=" knop indrukken, wat onmiddellijk ook het functieresultaat opvraagt.

De *interface* van een echte rekenmachine verschilt dus van het bovengaande model. We hebben naast het *model* van de data een *activator* die eventueel zelf nog gegevens omtrent de GUI kan bijhouden (zoals de kleur waarin iets getoond moet worden of de grootte van het venster) en die het model *activeert*.

De drie vermelde acties: een cijfer toevoegen, een functie toevoegen en het resultaat opvragen vinden we terug in de *interface Activator*.

```
interface Activator
  voegCijferToe :: Int → Int
  voegFunctieToe :: (Int → Int → Int) → (Int → Int)
  bereken :: Int
```

Merk op dat de methode *voegFunctieToe* in twee interfaces voorkomt. Dat is echter geen probleem omdat de methode hetzelfde type heeft in beide interfaces.

```
object guiCalc Activator
        (Int, Bool) (0, True) {
  voegCijferToe
       = λ dig → let
                      (x, bool) = state
                      newX = newDigit x dig bool
                   in
                   (newX, (newX, True)),
  voegFunctieToe
       = λf → let
                   (val, _) = state
                   okVal1 = RekenMachine!voegWaardeToe val
                   okVal2 = RekenMachine!voegFunctieToe f
                in
                (okVal2, (val, False)),
  bereken
      = let
            (val, _) = state
            okVal = RekenMachine!voegWaardeToe val
            newValue = RekenMachine!voegFunctieToe
         in
         (newValue, (newValue, False)),
  ...
}

newDigit val digit True = (val ∗ 10) + digit
newDigit _ digit False = digit
```

De Toestand Visualiseren Vervolgens gaan we het visualiserende object effectief zichtbaar maken. De eerste mogelijkheid is dat we dat doen voor het eigenlijke GUI-object *guiCalc* zelf. We tonen dan het getal dan momenteel gevormd wordt.

Om dat te bereiken moeten we enkel een **draw**-methode toevoegen aan *guiCalc*.

```
object guiCalc Controller ... {
  draw = Label
         text (show (fst state)),
  ...
}
```

Een tweede mogelijkheid is dat we een *view* creëren voor het functionele *model*. Omdat de view automatisch moet veranderen als het model verandert, gebruiken we hiervoor de **expand**-afleiding. Visto zorgt er voor dat de **draw**-methode van een expansie wordt uitgevoerd telkens het model van toestand verandert.

We maken eerst een *interface* waaruit we alle methodes verwijderen. Het is immers niet de bedoeling dat een view zijn model verandert. We vermelden alle methode-namen met een lage liggende streep (_) ervoor.

```
expandInterface Viewer from BinaireRekenaar
  _voegFunctieToe
  _voegWaardeToe
  _pasFunctieToe


expand simpeleViewer Viewer from RekenMachine {
  draw = Label
         text (show (fst superState))
}
```

Het gebruik van een *draw*-methode omlijnt mooi de plaats waar het (her)tekenen van een object beschreven wordt en het vereenvoudigt op die manier het onderhoud. Deze aanpak is weliswaar origineel voor functionele systemen, maar wordt ook (gedeeltelijk) toegepast in Java met *paint, repaint* en *update* methodes.

Om dingen te tonen hebben we de keuze uit een **Label**, een éénlijnig tekst-veld, of een **Canvas** waarop we met volgende grafische commando's kunnen tekenen.

Tekenopdrachten	*Opties Instellen*
line *Point Point*	setColor *color :: String*
rect *Point Point*	setFillColor *color :: String*
oval *Point Point*	setFillColor *color :: String*
clear	setWidth *Int*

Figuur B.16: Operaties op een Canvas

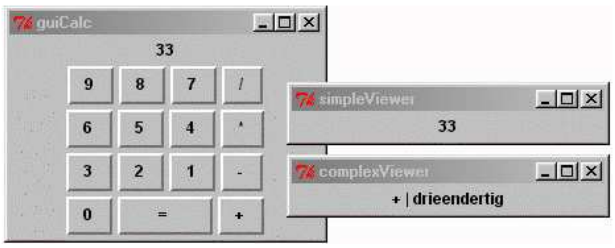
GUI-invokers Tot slot moeten we nog kiezen welke methodes we in de GUI presenteren, en in welke vorm(en) we dit doen.

Daartoe moeten we aan ons *Activator* object nog een aantal definities toevoegen. We herkennen dat het GUI-invokers zijn doordat we aan de linkerzijde eerst **GUI** plaatsen, gevolgd door de naam van de methode, en aan de rechterzijde de verschillende invocatie-manieren, van elkaar gescheiden door punt-komma's.

```
object guiCalc Activator ... {
   GUIbereken = Button " = "; Key " = ",

   ...
   GUIvoegCijferToe =Button "1" <<− 1;
                     Button "2" <<− 2;
                     Button "3" <<− 3;
                     ...
                     Key "0" <<− 0;
                     Key "1" <<− 1;
                     Key "2" <<− 2;
                     ...
   GUIvoegFunctieToe =Button " + " <<− (+);
                      Key " + "    <<− (+);
                      Button " − " <<− (−);
                      Key " − "    <<− (−);
                      ...
}
```

We zien hier dat de methode *bereken* wordt uitgevoerd wanneer er ofwel op de "="-knop wordt geklikt of wanneer de "="-toets wordt aangeslagen.



33

33

+ | drieendertig

Figuur B.17: De Rekenmachine met 2 views.

Als de methode parameters nodig heeft, vermelden we die achter de invoker, gescheiden door $<<-$. I.p.v. heel concrete waarden op te geven zoals hier, kunnen we ook gebruik maken van de selectors die we eerder besproken hebben.

Layout. De laatste faze is dan de layout van al deze invokers en selectors, maar dat gebeurt ook op een heel gestructureerde wijze. Immers, in een faze die nog niet geïmplementeerd is, worden alle invokers en selectors uit het programma verzameld.

Die gegevens worden dan doorgegeven aan een GUI-bouwer, een programma waarin de ontwikkelaar op een interactieve manier en met directe manipulatie zijn GUI kan samenstellen. Hij/zij kan dan knoppen e.d. op een venster plaatsen en zo op een visuele manier zijn GUI klaarstomen.

Dergelijke dingen bestaan al voor vele omgevingen, maar het verschil met onze versie is dat we de volledige set van invokers en selectors aanbieden die geplaatst moeten worden. Hierbij moeten we er wel zorg voor dragen dat deze potentieel heel grote set op een overzichtelijke wijze wordt gepresenteerd.

Los daarvan is de ontwikkelaar er wel zeker van dat op deze manier alle nodige elementen geplaatst zijn als alle selectors en invokers op zijn. Het is zoals een legpuzzel waar we alle stukjes krijgen en we die enkel op de juiste manier moeten samenvoegen.

Typische GUI-builders bieden de ontwikkelaar op dat gebied veel te veel vrijheid en zo'n GUI zal er dan meestal wel heel mooi uitzien, maar de kans is groot dat in die GUI opties vergeten zijn, of op een verkeerde manier aangeboden worden. De nadruk ligt immers vooral op het uitzicht en veel te weinig op de functionaliteit.

Samengevat. Bekijken we nog even de verschillende fazes waarin we een programma met GUI bouwen:

1. Eerst schrijven of gebruiken we een Haskell-programma met daarin de belangrijkste functies die we nodig hebben. Dit programma kan gemakkelijk ontwikkeld worden met de welbekende technieken omdat het geen (of weinig) rekening hoeft te houden met het interactieve karakter van een GUI.

2. Daarna ontwikkelen we de Visto-objecten die vertrekken van het standpunt van de eindgebruiker. We schrijven alle methodes die nuttig kunnen zijn voor die eindgebruiker. In die methodes gebruiken we de functies uit ons Haskell-programma.

3. Voor de objecten die zichtbaar moeten zijn schrijven we een visualiserende **draw**-methode, rechtstreeks in het object zelf of in een aparte *view*, wat we doen m.b.v de handige **expand**-afleiding.

4. Voor de GUI kunnen we nu kiezen uit de set van potentiële acties, nl. de methodes van onze objecten. In aparte *activator*-objecten associëren we methodes met GUI-invokers, waarbij we de argumenten van die methodes opvragen met onze declaratieve selectors.

5. Voor de layout instantiëren we alle invokers en selectors en passen die in elkaar als een legpuzzel, met natuurlijk die beperking, zoals beschreven op p. 273, dat de instantiatie van individuele selectors dikwijls afhangt van de volledige set instantiaties voor alle selectors. Uiteindelijk moet de interface immers een mooi geheel vormen, en is ze dus meer dan de optelsom van de individuele componenten.

   Zelfs dan blijft de set invokers en selectors en de Visto-beschrijving die vertelt hoe bepaalde methodes in de GUI opgeroepen kunnen worden een goede vertrekbasis voor de uiteindelijke layout. Wel zal in het reële proces deze vijfde faze niet louter *na* de vierde faze gebeuren, maar veeleer parallel.

Bespreking. Op die manier bekomen we een heel declaratieve beschrijving van de acties van de GUI. We definieren eerst *de functionaliteit* van de verschillende objecten, wat een heel sterke *wat*-bekommernis is. We moeten eerst weten *wat* er kan.

Daarna bepalen we *welke* methodes opgeroepen kunnen worden in de GUI, en pas helemaal op het einde kiezen we *hoe* dat gebeurt en op welke wijze we de verschillende invokers voorstellen. Uiteindelijk is het primordiaal eerst te beslissen *welke* waarden de gebruiker moet selecteren en pas daarna *hoe* de gebruiker dat moet doen, met een knoppenrij, of met een toetsencombinatie, . . .

Aanpassingen zijn nu heel gemakkelijk aan te brengen.

- Soms is het enkel een kwestie van layout. Dan moeten we enkel de laatste faze herhalen en kunnen we al de rest behouden.

- Als we langs de andere kant vastgesteld hebben dat we wel genoeg mogelijkheden hebben, maar dat we verkeerde keuzes voor het uitzicht van bepaalde selectors hebben gemaakt, dan moeten we die selectors anders instantiëren (bv. een invulveld i.p.v. een knoppenrij) en de layout herdoen.

- Meer waarschijnlijk is echter het feit dat we niet genoeg – of te veel – alternatieven hebben. Dan moeten we aan zo'n **GUI**-methode een paar invokers en selectors toevoegen of verwijderen, en ook weer overgaan naar de instantiatie ervan en ten slotte naar de layout.

  In traditionele systemen daarentegen, die opgebouwd zijn uit kleine widgets en samengesteld met vele call backs is dat beduidend meer werk, zoals blijkt uit de vergelijkende studie in de volgende sectie, alhoewel we in dit geval waarschijnlijk een groot deel van de call back functie kunnen hergebruiken.

- Tot zover veranderde de functionaliteit van de GUI niet echt. Enkel het uitzicht en/of het aantal alternatieven voor een bepaalde methode veranderde.

  De functionaliteit kan wel fundamenteler veranderen als we willen dat ze echt meer kan dan vroeger, bv. als we een bepaalde methode die we vroeger te moeilijk vonden voor onze gebruikers toch willen toevoegen aan de GUI. In dat geval hebben we een methode die nog niet geassocieerd is met een of andere invoker.

In Visto moeten we dan enkel zo'n associatie toevoegen, invokers en selectors kiezen, die instantiëren en de layout aanpassen. De objecten zelf moeten niet aangepast worden. Dit blijft dus relatief weinig werk.

In andere systemen moeten we dan echter een aantal nieuwe widgets samenvoegen en totaal nieuwe call back functies schrijven.

- Natuurlijk kan zoiets ook in Visto voorkomen als we dingen willen toevoegen waarvoor we zelfs nog geen methode hebben.

  Maar dan verandert in feite ook het toepassingsdomein en is het niet onlogisch dat de Visto-objecten en hun methodes wijzigen.

Een aantal kritieken op het simplistische model hebben we al kunnen wegwerken, maar het blijft zo dat de instantiaties van individuele invokers afhangen van de instantiaties van de volledige set invokers blijft. Daarom blijft ook het probleem dat we die instantiaties niet louter **na** de beslissing over de keuze van de invokers kunnen doen. Een wisselwerking tussen beide fazes blijft nodig.

Daarom dient de GUI-bouwer, waarin we de instantiaties kiezen, de set van invokers en selectors zodanig te presenteren dat we weten bij welke methodes en objecten ze horen. Bovendien moet hij er op voorzien zijn dat we die set van invokers en selectors nog kunnen wijzigen tijdens het tekenen en layouten van de uiteindelijke GUI. Dit moet de noodzakelijke wisselwerking tussen beide fazes mogelijk maken.

Een belangrijk voordeel blijft in ieder geval de beschrijving van de uiteindelijke GUI aan de hand van invokers en selectors. Dit uit zich ten eerste omdat het een declaratieve beschrijving is van de GUI: de invokers en selectors verduidelijken heel precies welke methodes met welke argumenten men kan oproepen vanuit de GUI.

Ten tweede kunnen we vanuit die informatie heel gemakkelijk afleiden wanneer twee GUI's functioneel equivalent zijn, nl. als ze dezelfde invokers en selectors aanbieden. Het feit dat die op een andere manier geïnstantieerd kunnen zijn, verandert natuurlijk het uitzicht en de bruikbaarheid van de GUI, maar niet wat men er mee kan doen.

Het vergemakkelijkt dan ook in zekere zin de keuze tussen verschillende GUI's: als de ene meer functionaliteit aanbiedt dan de andere, maar misschien iets minder mooi is, kan die toch nog verkozen worden omdat men er meer mee kan (als dan tenminste een belangrijk criterium is). Maar vooral omgekeerd is interessant: als eenduidig vast staat dat twee GUI's dezelfde functionaliteit hebben, hoeven we enkel nog te letten op bruikbaarheid en uitzicht. Het eventuele argument dat de minder mooie misschien meer functionaliteit aanbiedt, kan dan met zekerheid weerlegd worden, en enkel de intrinsieke kwaliteit van de GUI telt.

5.4 Een Alternatief voor Model-View-Controller

Een belangrijke doelstelling in software-ontwikkeling in het algemeen is onderhoudbaarheid. Dat is immers een van de grootste kosten van software, en zeker een van de meest onderschatte kosten. Een essentieel hulpmiddel om het onderhoud te vergemakkelijken is het opsplitsen van software in componenten met wel omschreven en afgescheiden verantwoordelijkheden. Ietwat overdreven uitgedrukt zal elke component precies één taak hebben.

Daarom is het geen goed idee, zoals we reeds eerder vertelden, om alles van de GUI in één object te steken, een object dat zowel de inhoud en het gedrag bevat, als de manier waarop het object getoond wordt, en de GUI zelf.

Model-View-Controller, kortweg MVC, is de meest gebruikte manier om de verschillende deeltaken van een GUI op te delen. Visto volgt in principe dezelfde opdeling, maar kent (lichtjes) afwijkende taken toe aan de verschillende delen, wat zich vooral uit in de implementatie.

- *View*

  Over de rol van de view bestaat geen discussie. De view dient om het object te tonen aan de eindgebruiker, niet meer, maar ook niet minder.

  Het is belangrijk om dit in een apart object te doen, zodat we enkel dat object moeten aanpassen als we het op een verschillende manier willen tonen, en niet het oorspronkelijke, grote object met alles er in gepropt.

  Bovendien kunnen we dan gemakkelijker verschillende views hebben voor eenzelfde iets, bv. een klok kunnen we zowel digitaal als analoog tonen. Dit is gemakkelijk omdat elke view afzonderlijk gedefinieerd wordt, en dus enkel zijn eigen taken dient te beschrijven.

- *Model*

  In essentie zijn we het ook eens over de rol van het model. Dit dient enkel de gegevens en het gedrag van het data-object bij te houden, en niks over hoe het getoond wordt, of over hoe het gekoppeld is aan de GUI. De *view* toont dus in feite het *model*.

  Zoals we reeds in de rondleiding hierboven konden zien, maken wij wel een onderscheid tussen het model, zoals de ontwikkelaar het voor ogen had, en het gebruikersmodel. Die twee kunnen immers sterk van elkaar verschillen; de ontwikkelaar heeft vooral oog voor hoe hij het probleem handig en logisch op kan splitsen, de gebruiker is eerder geïnteresseerd in wat wij noemden de visuele objecten. Ons model in dit MVC-patroon is in de eerste plaats het gebruikersmodel, terwijl dit in traditioneel MVC vooral het ontwikkelaarsmodel is.

- *Controller*

  Het is vooral in de controller waar wij van mening verschillen. In gewoon MVC is de controller een stukje software dat een welbepaald widget controleert. Bv. als we een knop gebruiken, hebben we een controller nodig voor die knop, die alle muisbewegingen en toetsenbordmanipulaties van de gebruiker interpreteert wanneer die knop actief is, en die indien nodig, via de call-back functie, een aantal acties uitvoert, zoals het oproepen van methodes van het model.

  In onze visie is dat geen goede optie, omdat het de klemtoon ligt op individuele widgets en de events uit dat widget (de muisbewegingen en toetsenbordmanipulaties), terwijl wij eerder de GUI zien als een *activator* van methodes van het model. Op die manier hebben wij dan ook onze GUI geprogrammeerd: *Om methode X te activeren, gebruik die of die invoker.* Onze beschrijving is dus doelgericht: öm een bepaald doel te bereiken, gebruik die invokers", eerder een *call forward* als je wil.

  Bovendien kan een controller als reactie op een event *eender wat* doen: een methode oproepen van het model-object, iets hertekenen, of ook iets totaal anders wat niet beschreven staat in de objecten die normaal participeren in MVC. Het voordeel hiervan is natuurlijk de grote vrijheid, omdat we dan kunnen doen wat we willen, bv. heel dynamische effecten creëren. Toch staan we dat soort vrijheid niet toe in Visto. Als wij vinden dat het model en zijn methodes precies de volledige functionaliteit van het systeem beschrijven, is het niet meer dan logisch om te stellen dat de controller, een activator in Visto, enkel zulke methodes kan en mag activeren.

  Dat is tegelijkertijd ook de kracht van Visto. Binnen onze activator schrijven we steeds **GUI***methodeX* = *invokerX* ...], wat betekent dat *methodeX* geactiveerd wordt wanneer de eindgebruiker *invokerX* gebruikt. Dit betekent dat zodra *invokerX* gedefinieerd is dat die dan ook onmiddellijk actief is. Dat is een scherpe tegenstelling met traditionele controllers waar een widget pas actief wordt als de ontwikkelaar acties toevoegt aan de controller. Dat is een extra reden om onze controller een activator te noemen om de actieve rol ervan te beklemtonen.

Een volgend verschilpunt is de samenwerking tussen de view en de controller. In MVC is er een één-één relatie tussen view en controller. Dat is voor een belangrijk stuk zo omdat de controller alle events van een widget moet opvangen. Omdat een view ook steeds een (samengesteld) widget is, bestaat er dus ook een controller voor elke view. Dat is handig omdat we dan een view tegelijkertijd ook kunnen gebruiken als selector, bv. als we de tijd tonen, kunnen we tegelijkertijd daarmee het uur aanpassen.

Toch is dat ook onhandig, want uiteindelijk zijn view en controller verschillende taken van de GUI. Visto scheidt ze dan ook volledig. Nu is het wel zo dat we ook in SmallTalk een view zonder controller kunnen hebben. We moeten doen als controller een instantie van de klasse *NoController* opgeven, maar dat is dus uiteindelijk niet meer dan een uitzondering op de regel.

De view heeft dus een controller nodig, maar de controller dient ook de view te kennen. De reden hiervoor is dat het de taak is van de controller om de view te verversen, bv. als iemand op een knop gedrukt heeft, zal de controller van de knop op dat event reageren en o.a. het model aanpassen en aan de view de opdracht geven om zich aan te passen aan die nieuwe toestand van het model.

Die omweg dient niet in Visto te gebeuren. Omdat een view een expansie is van het model, zal de view zich automatisch hertekenen (ttz. zijn *draw*-methode uitvoeren) wanneer het model van toestand verandert. Dat is dus een veel duidelijkere en gemakkelijkere situatie.

Ten slotte is Visto ook veel flexibeler. Eender welk Visto object kan een model zijn, omdat het zelf geen verzwijzingen bevat naar zijn views of controllers. Die beslissen zelf in welke mate ze gebruik maken van het model. Dat is opnieuw verschillend van SmallTalk en Java.

In SmallTalk moet het model erven van de klasse *Model*; in Java van de klasse *Observable*. Wanneer in zo'n klasse een methode de toestand van het object verandert, dient dat object dan de methodes *setChanged()* en *notifyObservers()* op te roepen om aan alle afhankelijke objecten te laten weten dat het van toestand veranderd is.

Het is dus in Visto veel gemakkelijker om van een bestaand object een model te maken uit het MVC-triumviraat zonder dat het model hiervoor specifiek ontwikkeld is, en dat is natuurlijk een voordeel.

Het besluit is dat het Visto-model op het eerste zicht heel gelijklopend met MVC mag zijn, maar dat het toch op verschillende manieren ervan afwijkt, vooral in het concept van controller versus activator.

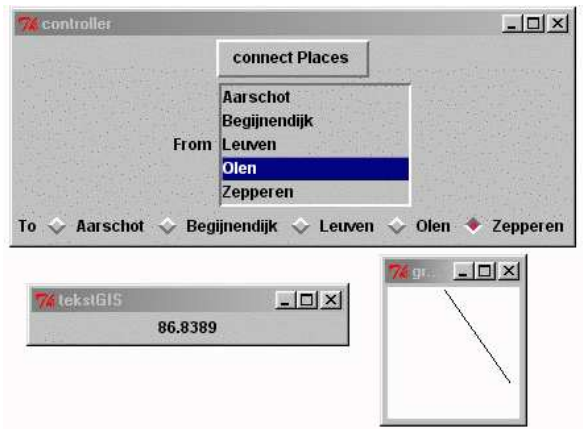
5.5 Een Vergelijkende Studie

Vanzelfsprekend hebben we ook de proef op de som genomen en zijn we via een vergelijkende studie nagegaan of ons Visto-programma inderdaad zoveel gemakkelijker te onderhouden is dan alternatieve GUI-systemen.

Daartoe hebben we een klein *Geografisch InformatieSysteem* (GIS) ontwikkeld, dat de gebruiker twee plaatsen laat selecteren en informatie toont over de verbinding tussen die twee plaatsen.

Dit programma is ontwikkeld in Visto, Tcl/Tk, Java, Clean en Fudgets. Aan elk van die programma's hebben we dan een aantal identieke veranderingen aangebracht en de impact ervan geëvalueerd.

- Eerst hebben we een keuzeplaats toegevoegd.
- Dan hebben we de selectiemanier voor de vertrekplaats veranderd van een knoppenrij naar een lijstselectie, waarbij de lijst in een venster getoond wordt en gekozen wordt door op de naam te klikken. Dit is een vrij typische verandering aangezien een knoppenrij onhandig wordt wanneer het aantal keuzemogelijkheden te groot wordt.
  Op zich is dit geen grote verandering, want het enige wat verandert is *hoe* de keuze moet gebeuren. De impact zou dan ook minimaal moeten zijn.
- Vervolgens hebben we nog een andere *view* aan de verbinding toegevoegd. Het is weliswaar een heel eenvoudige view, maar ze toont goed hoeveel (of hoe weinig) werk zoiets vraagt.
- Tot slot hebben we nog een tweede manier voorzien om een verbinding te leggen. Hierbij gebruiken we de aankomstplaats uit de vorige connectiewijze.



Figuur B.18: Uitzicht van het Visto programma.

Meer details over dit experiment staan te lezen in het Engelse gedeelte van dit proefschrift.

We hebben in ieder geval kunnen vaststellen dat de veranderingen in de Visto-code enkel betrekking hadden op die code die direct te maken had met de veranderingen. Geen afgelegen stukken code of andere objecten moesten aangepast worden.

Omdat Tcl/Tk geen krachtig type-systeem heeft, noopten de meeste wijzigingen tot veranderingen aan de code op verschillende plaatsen, maar daarbuiten deed

Tcl/Tk het niet echt slechter dan Java, Clean of Fudgets, alhoewel het de oudste van de vijf systemen is.

In zowel Java, Clean als Fudgets bleek, weliswaar met de nodige aandacht en een extra initiële inspanning, vooral het toevoegen van een extra keuzeplaats best gemakkelijk te zijn. Visto blinkt hierin nog altijd uit omdat die initiële inspanning overbodig is in Visto.

De knoppenrij veranderen in een keuzelijst bleek beduidend zwaarder omdat die twee widgets op een andere manier aangemaakt worden en omdat het opvragen van welke selectie er gemaakt is, heel anders gebeurt. In Clean was de aanpassing relatief klein maar zeker niet verwaarloosbaar. In Tcl/Tk, Java en Fudgets was de aanpassing nog gevoelig groter. In Visto volstond het om de selector op een andere manier te instantiëren en de layout aan te passen.

De derde aanpassing, een *view* toevoegen, betekende dat we in Java het *ActionListener* object moesten aanpassen, en in Tcl/Tk en Clean de call back functie. In die systemen is er effectief 'iemand' die de views op de hoogte moet brengen van wijzigingen aan het model. We moeten bijgevolg die derde partij ook aanpassen. Dat is niet het geval in Visto. Het volstaat om de view toe te voegen. Enkel de view zelf moet zichzelf aanmelden als view en Visto zorgt er dan voor dat die view hertekend wordt wanneer het model verandert. Op geen enkel andere plaats dan in de view zelf moet er nieuwe code komen. In Fudgets ging dat ook bijna, maar niet helemaal. De abstracte fudget die het systeem centraal beheert moest niet aangepast worden, maar het stroomschema van de gecombineerde Fudgets wel. Visto scoort hier dus opnieuw het beste, ook omdat het het meest logisch is: als we een view toevoegen, is dat enkel een kwestie van die view en dus moeten we niks anders aanpassen. Als plotseling meer mensen naar de Big Ben komen kijken, moet de Big Ben dat toch ook niet weten?

Ten slotte bleek uit het toevoegen van een nieuwe connectiewijze duidelijk de nadelen van de call back-aanpak. In Visto konden we probleemloos de selectie van de aankomstplaats uit de vorige selectie hergebruiken, maar in alle andere systemen moesten we een nieuwe call back-functie schrijven alhoewel die sterk op de vorige gelijkt. Bijgevolg blinken onze selectors en invokers niet alleen uit door het feit dat ze declaratief zijn, maar ook omdat ze een hoger hergebruik toelaten.

5.6 Besluit

De Visto-techniek om te concentreren op de visualiserende objecten, daarvoor methodes te schrijven en dan pas te kiezen welke van de methodes in de GUI aangeboden worden, opent een nieuw en declaratiever gezichtspunt op de definitie van GUI's. We vinden vooral de verschuiving van call back functies naar de GUI-invokers een belangrijke vernieuwing.

Tegelijk vereenvoudigt de automatische uitvoering van de **draw**-methode het consistent houden van de getoonde informatie en blijkt de **expand**-afleiding een handig wapen te zijn voor het gebruik van views.

De vergelijkende studie heeft uitgewezen dat Visto er inderdaad in slaagt om op een declaratievere wijze GUI’s te beschrijven die minder hoeven te vrezen van veranderingen aan de GUI en aldus het onderhoud vergemakkelijken.

6 Algemeen Besluit

In dit proefschrift hebben we uitgestippeld hoe volgens ons GUI’s op een declaratievere wijze kunnen beschreven worden.

Daartoe hebben we eerst de elementaire bouwstenen aangepakt. I.p.v. de traditionele widgets hebben we *selectors* voorgesteld. Deze hoger niveau-bouwstenen concenteren op de *wat*, nl. het feit dat men dingen wil selecteren: gegevens en/of commando’s. We hebben verschillende selectoren en verschillende presentatievormen geïmplementeerd zowel in Fudgets als TkGofer.

Om een stap verder te gaan, hebben we een objecttaal nodig. Deze is dermate gegroeid dat we in staat waren om een apart hoofdstuk aan deze objecttaal, Visto, te besteden. Visto is een *prototype gebaseerde* objecttaal met verschillende manieren om nieuwe objecten te maken vertrekkende van bestaande objecten, gaande van eenvoudige *klonen*, over *adapties* tot *expansies*. Innoverend hierbij zijn de delegaties en triggers.

Met de Visto-objecten en hun methodes kunnen we beschrijven welke functionaliteit aanwezig *kan* zijn in de GUI. We hebben dan aangetoond hoe we een selectie van deze methodes dan effectief aanbieden in de GUI door gebruik te maken van GUI-invokers.

De vergelijkende studie heeft uitgewezen dat het Visto-programma inderdaad heel gemakkelijk aan te passen was, zodat we durven stellen dat Visto inderdaad voldoet aan de eisen die we bij het begin van ons onderzoek gesteld hebben, nl. het beschrijven van de GUI op een declaratievere manier in een systeem dat zo min mogelijk gevoelig is voor aanpassingen.

Bibliography

[1] P. Achten, R. Plasmeijer *The Ins and Outs of Clean I/O* In Journal of Functional Programming. 5:1, (1995), 81–110. Also at `http://www.cs.kun.nl/~clean/`

[2] P. Achten, R. Plasmeijer: *Using Type and Constructor Classes to Interpret Object Structures* In Proc. of Workshop on the Implementation of Functional Languages, Båstad, Sweden, Chalmers Tekniska Hogskola, (1995) 142–156.

[3] K. Aerts, K. De Vlaminck: *A GUI on top of a functional language (abstract)* In Proceedings of the ACM SIGPLAN International Conference on Functional Programming, (1997), p.308

[4] K. Aerts, K. De Vlaminck: *Visto: A More Declarative GUI Framework* In J. Vanderdonckt, A. Puerta (eds.), Computer-Aided Design of User Interfaces II, Proceedings of the 3rd International Conference on Computer-Aided Design of User Interfaces CADUI'99, Kluwer Academics, (1999), p. 73–78.

[5] J. Armstrong, *The development of Erlang* In Proceedings of the ACM SIGPLAN International Conference on Functional Programming, (1997), 196–203 Also at `http://www.ericsson.se/cslab/erlang/`

[6] G. Baumgartner, V.F. Russo: *Implementing signatures for C++.*, In ACM Transactions on Programming Languages and Systems, Vol 19, No. 1, January 1997, 153–187.

[7] J. Bergin: *Building Graphical User Interfaces with the MVC Pattern* Pace University. From `http://www.wol.pace.edu/ bergin/mvc/mvcgui.html''` (May 9, 2000).

[8] T. Berlage: *Using Taps to Separate the User Interface from the Application Code* In Proceedings of the ACM Symposium on User Interface Software and Technology (UIST), Monterey, California, ACM Press, (1992) 191–198

[9] R. Bird: *Introduction to Functional Programming using Haskell.* Prentice Hall, New York, 1998.

[10] F. Bodart, A-M. Hennebert, J-M. Leheureux, J. Vanderdonckt: *A Model-based Approach to Presentation: A Continuum from Task Analysis to Prototype*, Proceedings of Eurographics Workshop on Design, Specification, and Verification of Interactive Systems DSV-IS'94, Springer-Verlag, Berlin, 77–94. Also available at `http://www.info.fundp.ac.be/cgi-publi/pub-spec-paper?RP-94-023`

[11] F. Bodart, A-M. Hennebert, J-M. Leheureux, I. Provot, J. Vanderdonckt, G. Zucchinetti: *Key Activities for a Development Methodology of Interactive Applications*, Chapter 7 in Critical Issues in User Interface Systems Engineering, Springer-Verlag, Berlin, 109–134, (1995). Also available at `http://www.info.fundp.ac.be/cgi-publi/pub-spec-paper?RP-96-025`

[12] A.H. Borning: *Classes versus prototypes in object oriented languages*. In Proc. ACM/IEEE Fall Joint Computer Conference, (1986) 36–40.

[13] H-J. Bullinger, K-P. Fahnrich, A. Weisbecker: *GENIUS: Generating Software-Ergonomic User Interfaces Articles* International Journal of Human-Computer Interaction, 1996 volume 8 n.2, 115–144

[14] S. Burbeck: *Applications Programming in Smalltalk-80: How to use Model-View-Controller* From `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html` (1987, 1992, Mar. 4, 1997)

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *A System of Patterns, Pattern Oriented Software Architecture*. John Wiley & Sons, ISBN 0 471 95869 7

[16] M. Carlsson, Th. Hallgren: *Fudgets - A Graphical User Interface in a Lazy Functional Language.* In Proceedings of FPCA, (1993) 321–330.

[17] M. Carlsson, Th. Hallgren: *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces*, PhD. thesis, available at `http://www.cs.chalmers.se/ hallgren/Thesis/`, (1998).

[18] Y. Caseau, F. Laburthe: *CLAIRE: Combining Objects and Rules for Problem Solving* In Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming, TU Berlin, (1996) Also: `http://www.dmi.ens.fr/~laburthe/claire.html`.

[19] B-W. Chang, D. Ungar: *Animation: from cartoons to the user interface.* In Proceedings of the ACM Symposium on User Interface Software and Technology (UIST), Atlanta, ACM Press, (1993)

[20] A. Church: *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, New Jersey, (1941)

[21] A. Cypher: *EAGER: Programming Repetitive Tasks by Example Programming by Demonstration*, Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems, 33–39, (1991)

[22] C. Dornan, M. Sage: *TclHaskell* Originally developed by Chris Dornan, but now available from Meurig Sage at `http://www.dcs.gla.ac.uk/~meurig/TclHaskell/`.

[23] Eckstein, Loyd, Wood: *Java Swing*, (1998).

[24] S. Finne, S. Peyton Jones: *Composing Haggis* In Proceedings of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics, Maastricht, September 1995, Springer Verlag. Updated info at `http://www.dcs.gla.ac.uk/fp/software/haggis/`

[25] S. Finne, S. Peyton Jones: *Composing User Interfaces with Haggis. Sigbjorn Finne and Simon Peyton Jones* Presented at the 1996 Summer school on advanced Functional programming, Olympia, WA, Aug 25-30 1996. Available from `http://www.dcs.gla.ac.uk/fp/software/haggis/`

[26] D. Flanagan: *X toolkit intrinsics reference manual*, O'Reilly Sebastopol, (1992).

[27] J. Foley, N. Sukaviriya: *History, Results, and Bibliography of the User Interface Design Environment (UIDE), an Early Model-Based System for User Interface Design and Implementation*, Interactive Systems: Design, Specification, and Verification, Springer, Berlin, 3–10, (1995).

[28] N. François, *Reqtools, Amiga runtime library for requesters*, Since 1995 maintained by Magnus Holmgren, available at `http://www.algonet.se/ lear/reqtools.html`.

[29] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patters, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, (1995).

[30] P. Girard, G. Pierra, J.C. Potier: *Customising by Demonstration Generic Systems to Specific Tasks*, 3rd ERCIM Workshop on 'User Interfaces for All' 1997 n.24 p.7 ERCIM, Also available at `http://ui4all.ics.forth.gr/UI4ALL-97/girard.pdf`

[31] M. Hanus (ed.) *Curry: An Integrated Functional Logic Language*, Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, (1999).

[32] M. Hanus *A Functional Logic Programming Approach to Graphical User Interfaces,* In Proceedings of PADL 2000, LNCS 1753, (2000), 47–62.

[33] P. Hartel, R. Plasmeijer (Eds), *Proceedings of Functional Programming Languages in Education*, Lecture Notes in Computer Science **1022** (1995).

[34] D. Howe (editor), *The Free On-line Dictionary of Computing',* available at `http://foldoc.doc.ic.ac.uk/`.

[35] P. Hudak: *Conception, evolution, and application of functional programming languages.* ACM Computing Surveys, 21(3): 359–411, 1989.

[36] P. Hudak, J. Peterson, J. Fasel, R. Thomas: *A Gentle Introduction To Haskell, version 98*. From `http://www.haskell.org/tutorial/`.

[37] P. Hudak, M.P. Jones: *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity* Available from `http://www.haskell.org/papers/NSWC/jfp.ps`, 1994, 16 pages

[38] J. Hughes, J. Sparud: *Haskell++: An Object Oriented Extension of Haskell* Haskell Workshop 1995, Orlando, Florida

[39] S. Jarvis, S. Poria, R. Morgan: *Understanding LOLITA: Experiences in teaching large scale functional programming,* In Proceedings of FPLE '95, Lecture Notes in Computer Science **1022** (1995), 103–121.

[40] J. Johnson: *Going Beyond User-Interface Widgets.* In CHI'92 Conference Proceedings, ACM conference on human factors in computing systems (1992) , 273–279.

[41] J. Johnson, B. Nardi, C. Zarmer, J. Miller: *ACE: A New Approach to Building Interactive Graphical Applications*, Communications of the ACM, April 1993, 41–55.

[42] M.P. Jones: *An introduction to Gofer (draft),* Included as part of the standard Gofer distribution, (1993).

[43] E.W. Karlsen: Haskell-Tk has been developed in the context of *The UniForM Concurrency ToolKit and its Extensions to Concurrent Haskell,* Glasgow Functional Programming Workshop, (1997).

[44] D. Katiyar, D. Luckham, J. Mitchell, S. Meldal: *Polymorphism and subtyping in interfaces.* Int ACM SIGPLAN Notices, Vol. 29, No. 8, IDL Workshop. (1994), pp. 22–34.

[45] G.E. Krasner, S.T. Pope: *A Cookbook for Using the Model-View-Controller User Interface Paradigm in SmallTalk-80* In Journal of Object Oriented Programming, August/September 1988, pp. 26–49.

[46] S. Krishnamurthi, M. Felleisen, D.P. Friedman: *Synthesizing Object-Oriented and Functional Design to Promote Re-Use*, Rice University Technical Report TR98-299, (1998), Available from `http://www.cs.rice.edu/CS/PLT/Publications/#tr98-299`

[47] R.A. Milner: *Theory of Type Polymorphism in Programming* In Journal of Computer Science and System Sciences, (1978), Vol. 17, no. **3**, 348–375.

[48] P.A. Mocciola, P.E. Martínez López: *Design Patterns for Functional Programming*, In online proceedings of the 3rd Latin-American Conference of Functional Programming, (1999), `http://www.di.ufpe.br/ clapf99/proceed.htm`.

[49] A. Mycroft: *Polymorpic Type Schemes and Recursive Definition* In Proceedings of the 6th International Conference on Programming, (1984) LNCS **167**, 217–228.

[50] B.A. Myers: *Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs* In Proceedings of the ACM Symposium on User Interface Software and Technology (UIST), Hilton Head, South Carolina, ACM Press, (1991) 211–220

[51] B.A. Myers: *Why are Human-Computer Interfaces Difficult to Design and Implement* Tech Report CMU-CS-93-183 Carnegie Mellon University

[52] B.A. Myers, D. Giuse, R.B. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, P. Marchal: *Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment*, IEEE Computer, Vol. **23**, No. **11**, 71–85, (1990).

[53] B.A. Myers, A. Cypher, D. Maulsby, D. Smith, B. Shneiderman (panel): *Demonstrational Interfaces: Coming Soon?*, Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems, 393–396.

[54] B.A. Myers, B. Vander Zanden, R.B. Dannenberg: *Creating Graphical Interactive Application Objects by Demonstration*, In Proceedings UIST'89, 95–104, (Nov. 1995).

[55] NeXT, Inc. : *NeXTStep and the NeXT Interface Builder*, NeXT Inc., 900 Chesapeake Drive, Redwood City, CA 94063, 1991.

[56] R. Noble, C. Runciman: *Functional Languages and Graphical User Interfaces – a review and a case study* Technical report YCS-94-223 University of York, (1994) Available from `ftp://ftp.cs.york.ac.uk/reports/YCS-94-223.ps.Z`

[57] R. Noble, C. Runciman: *Gadgets: Lazy Functional Components for Graphical User Interfaces (abstract)* In Proceedings of PLILP'95, Lecture Notes in Computer Science **982** (1995). 321–340.

[58] R. Noble, A. Taivalsaari (eds.): *ECOOP'96 Workshop on Prototype Based Object Oriented Programming*, Summary available at `http://www.mri.mq.edu.au/ kjx/proto/wkshop96/workshop-writeup.html`

[59] D.R. Olsen: *MIKE: The Menu Interaction Kontrol Environment*, ACM Transactions on Graphics 1986 volume 5, nr. 4, 318-344.

[60] J.K. Ousterhout: *Tcl and the Tk toolkit*, Addison Wesley, (1994).

[61] R. Pausch, N.R. Young II, R. DeLine *SUIT: The Pascal of User Interface Toolkits* In Proceedings of the ACM Symposium on User Interface Software and Technology (UIST), Hilton Head, South Carolina, ACM Press, (1991) 117–125

[62] S. Peyton Jones, J. Hughes (editors): *Haskell 98: A Non-strict, Purely Functional Language* Report on the Programming Language Haskell 98, Available from `http://www.haskell.org/` or at `http://haskell.systemsz.cs.yale.edu/onlinereport/`

[63] *FranTk: A Declarative GUI System for Haskell*, available from `http://haskell.cs.yale.edu/FranTk/`

[64] A.R. Puerta: *A Model-Based Interface Development Environment.*, IEEE Software July/August 1997, 40–47.

[65] R. Rao, S. Wallace: *The X Toolkit - The Standard Toolkit for X Version 11*. In Proceedings of USENIX Summer 1987 Conference, Phoenix, Arizona, (1997)

[66] L. Rapanotti, A. Socorro: *Introducing FOOPS* Oxford Technical Report, PRG-TR-28-92, (1992)

[67] D. Rémy, J. Vouillon, *Objective ML: An effective object-oriented extension to ML*. To appear in Theory And Practice of Objects Systems, (1998). Also at `http://pauillac.inria.fr/ocaml/`.

[68] J. Rosenberg e.a.: *X Toolkits: the Lessons Learned* In Proceedings of the ACM Symposium on User Interface Software and Technology (UIST), Snowbird, Utah, ACM Press, (1990) 108–111

[69] B. Rumpe: *GOS: Gofer Objekt-System, Imperativ Objektorientierte und Funktionale Programmierung in einer Sprache vereint* In Kolloqium Programmiersprachen und Grundlagen der Programmierung, MIP Bericht N.

9519 Universität Passau, (1995) Also: `http://www4.informatik.tu-muenchen.de/~rumpe/gos/`

[70] J. Sargeant, S. Hooton, C. Kirkham: United Functions and Objects: Key Ideas *UFO: Language evolution and consequences of state* In High Performance Functional Computing, (April 1995), 48–62 Also: `http://www.cs.man.ac.uk/arch/projects/ufo.html`

[71] R.W. Scheifler, J. Gettys: *The X Window System* In ACM Transactions on Graphics, Vol. 5, No. 2 (April 1986).

[72] W. Schulte, K. Achatz: *Functional Object-oriented Programming with Object-Gofer* In Informatik '97: Informatik als Innovationsmotor, Informatik aktuell, Springer (1997). Also in: Arbeitstagung Programmiersprachen. Arbeitsbericht der Universität Münster, Institut für Wirtschaftinformatik, Nr. **58**, (Sept. 1997).

[73] G. Singh, M. Green: *A High-level User Interface Management System*, Proceedings of SIGCHI'89, 133-138, (April 1989).

[74] G. Sing, C.H. Kok, T.Y. Ngan: *Druid: A System for Demonstrational Rapid User Interface Development*, In Proceedings of UIST'90, 157–177, (Oct. 1990).

[75] J. Skibinski: *Haskell Companion* – Most common concepts and definitions of functional language Haskell – Available at `http://www.numeric-quest.com/haskell/hcompanion/index.html`

[76] R.B. Smith, J. Maloney, D. Ungar: *The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity and Flexibility* In ACM Sigplan Notices, OOPSLA, (1995), Vol. 30, no, **10**, 47–60

[77] W.R. Smith: *Using a Prototype-based Language for User Interface: The Newton Project's Experience* In ACM Sigplan Notices, OOPSLA, (1995), Vol. 30, no, **10**, 61–72

[78] P. Szekely, P. Luo, R. Neches: *Beyond Interface Builders: Model-Based Interface Tools*, In Proceedings of CHI93, 383–390, Also available at `http://www.isi.edu/isd/Interchi-beyond.ps`.

[79] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy. E. Salcher: *Declarative Interface Models for User Interface Construction Tools: the MASTERMIND Approach*, in Proceedings of EHCI95, 120–150, Also available at `http://www.isi.edu/isd/Mastermind/Papers/ehci95.ps`

[80] P. Szekely: *Retrospective and Challenges for Model-Based Interface Development*, int Proceedings of DSV-IS96, 1–27. Also available at `http://www.isi.edu/isd/Mastermind/Internal/Files/DSVIS96/paper.ps.Z`, (1996).

[81] D. Ungar, R.B. Smith: *Self: The Power of Simplicity* In OOPSLA'87 Proceedings, ACM Press, (1987), 227–242

[82] A. Taivalsaari: *Classes vs. Prototypes Some Philosophical and Historical Observations* Nokia Research Center P.O. Box 45, 00211 Helsinki FINLAND, taivalsa@research.nokia.com, April 22, 1996, also available from `http://citeseer.nj.nec.com/did/48544`

[83] A. Taivalsaari: *Kevo – a prototype-based object-oriented language based on concatenation and module operations*. Technical Report DCS-197-1R, University of Victoria, B.C. Canada, (1992).

[84] The UIMS Workshop Tool Developers: *A Metamodel for the Runtime Architecture of An Interactive System*, SIGCHI Bulletin, 24, 1, 32–37, (Jan. 1992)

[85] J. Vanderdonckt: *Automatic Generation of a User Interface for Highly Interactive Business-Oriented Applications*, Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems, volume 2, 123–124, (1994)

[86] J. Vanderdonckt, A. Puerta (eds.) *Computer-Aided Design of User Interfaces II*, Kluwer Academic Publishers, ISBN 0-7923-6078-8, (1999).

[87] B. Vander Zanden, B. A. Myers: *Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces Constraint Based UI Tools* , Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems, 27–34, (1990)

[88] T. Vullinghs, D. Tuijnman, W. Schulte *Lightweight GUIs for Functional Programming* In Proceedings of PLILP'95, Lecture Notes in Computer Science **982** (1995). 341–356. Also at `http://www.informatik.uni-ulm.de/pm/ftp/tkgofer.html`

[89] D.A. Young: *The X Window System : Programming and Applications with Xt. OSF/Motif Edition* , Prentice Hall, (1990).