Completely Customizing Modern GUIs Through Command-Driven Interfaces

by

Jeff Dicker

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

An ideal user interface accommodates the requirements and abilities of its users, and every user has a specific set of needs that must be fulfilled in order for an interface to be useful. This thesis concentrates on using the post-deployment tailoring technique of customization in order to ensure that an interface meets a user's needs and abilities in a final, user-driven design step. To this end, the more entirely a UI can be customized, the more perfectly it can be transformed into a state that best suits its user. Very few systems offer *complete customization*: allowing the entirety of an interface to be customized, baring change to its interaction style. While a few systems do offer complete customization, no fully customizable system exists that is built using modern widget-based GUI's. This is the goal of the architecture described in this thesis, the *Interface Manager*. It uses interface building techniques to make cosmetic customizations and a command-driven style similar to that of Unix shells to make functionality customizations. This system allows interfaces to become well suited to their user, but it also offers open questions about user-initiated innovation in software and the scaling of visual interface design tools.

Acknowledgements

First, and foremost, I would like to thank Bill Cowan for academic, financial and moral support. Few share his passion for widening academic discourse.

I thank my readers, Mike Terry and Charlie Clarke for giving their time to ensure my success. If it wasn't for Mike's determination and assistance, I would have likely taken another semester to finish this thesis.

I would also like to thank my family: Katy, Jerry and Jill. Without them, I certainly would not be writing this. CGL members also helped keep me sane, especially Élodie, Cam, Curtis, Eoghan, Vlad, Andrew and Alex.

Finally, I owe this entire journey to Alan Paeth. His belief in my ability is the reason that I am here.

Last, but not least: go Grad House FC!

Contents

List of Figures

Trademarks

The following trademarks are used in this thesis.

- Linux is a registered trademark of Linus Torvalds
- Microsoft and Microsoft Word are registered trademarks of the Microsoft Corporation in the United States and/or other countries
- Apple is a registered trademark of Apple Incorporated in the United States and/or other countries
- Adobe is a registered trademark of Adboe Systems Incorporated in the United States and/or other countries
- Unix is a registered trademark of The Open Group
- CORBA is a registered trademark of the Object Management Group in the United States and/or other countries
- LabView is a trademark of National Instruments in the United States and/or other countries

All other products mentioned in this thesis are trademarks of their respective companies. The use of general descriptive names, trademarks, etc., in this publication, even if not identified, is not to be taken as a sign that such names may be used freely by anyone.

Chapter 1

Introduction

An ideal user interface accommodates the requirements and abilities of its users. Every user has a specific set of needs that must be fulfilled in order for an interface to be useful, and a specific set of abilities or disabilities, such as enhanced motor ability or impaired eye-sight. Designing an interface that is perfectly suited to a hypothetical representative user does *not* produce an interface well-suited to each individual user. Instead, post-deployment tailoring of interfaces is required as a final design step. Post-deployment tailoring can produce an interface both with which a user is more comfortable and which has a dramatic effect on efficiency. For example, Gajos, et al. were able to make large gains (between 8.4% and 42.4%) in usage efficiencies of interfaces by tailoring them post-deployment to suit the abilities of each motor-impaired user [11]. In this case, tailoring was done using the technique of *interface adaptation*: modifying an interface automatically on behalf of a user. This thesis concentrates on *interface customization*, the other primary, complimentary post-deployment technique. The more entirely a UI can be customized, the more perfectly it can be transformed into a state that best suits the requirements and abilities of its user.

Following from the desire to customize as much as possible, customization tech-

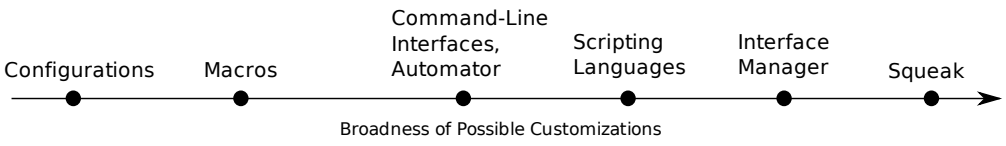


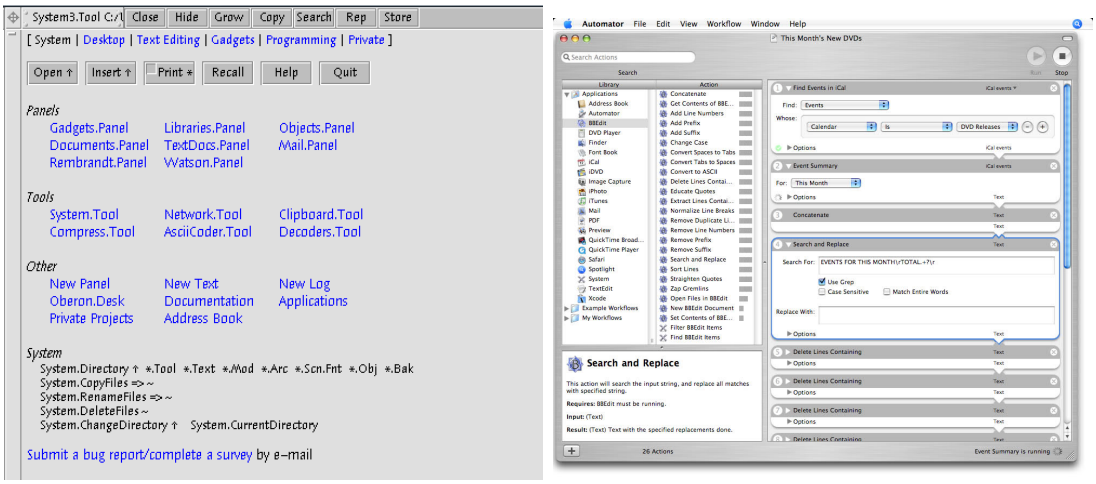
Figure 1.1: One way to view where a particular kind of customization sits on the spectrum of customizations is in terms of broadness: how much can be customized through particular technique?

niques can be placed onto a spectrum in terms of completeness: how much of an interface can be customized. The most narrow customizations are *configuration customizations*, which include such customizations as choosing a font size, a desktop background, and prefered applications. Macros allow users to *combine* commonly executed sequences of commands into new atomic commands. This process is where the spectrum starts to move into the realm of *end-user programming*. Command-line interfaces (CLIs) also allow users to combine sequences of existing functionality into new functionality, but provides the further advantage of allowing users to make *compositions*. A CLI composition is made when the input to a command-line application is redirected to be the output from another application. Momentarily skipping over scripting languages, the Squeak environment [9], using the Morphic [18] GUI and Squeak programming language, provides complete customization facilities.

A system that provides *complete customization* allows every part of its interface to be customized, with the exception of the interaction style. Details of the interaction style may be modified, such as single verses double clicks for opening files, but the actual interaction style, such as a WIMP GUI, will stay the same.

Squeak provides complete customization by providing a live, direct manipulation customization system (through Morphic) and facilities to modify the Squeak source code (through the Squeak programming language), all of which is available to end-users. However, except for the most simple customizations, knowledge of the Squeak programming language is required to make customizations, so this system has a particularly steep learning curve. Thus, while it is possible to tailor post-deployment the entirety of Squeak to be well-suited to any user, the process is as laborious as customizing an open-source system by editing its source code.

This is where we arrive at scripting languages, as they are used to alleviate this problem, lowering the amount of programming knowledge required to make complex customizations. They allow users to make a wide variety customizations by programming, and have a far shallower learning curve. The reason is that while most scripting languages do resemble (and often *are*) traditional programming languages, they have a feature set restricted to a narrow API. Modification and reuse of existing application functionality is their emphasis, and they are constructed so that API access takes priority over usability of the programming language. Both scripting languages and Squeak allow end-users to customize through programming, but the *guided* programming provided by scripting languages offers assurance that users can access application functionality quickly, and restricts the environment and usage so as to disallow operations detrimental to the actual software. This



(a) Oberon (b) Automator

Figure 1.2: Oberon presents commands as a first-class citizen of its GUI, but provides a programming language as the only tool that operates on them. Automator, on the other hand, presents a combination and composition tool on top of Apple Events, but MacOS does not promote Apple Events as first-class citizens of its GUI.

is a fundamental trade-off between power and ease of use: the limited API that can be used with scripting languages also limits how much customization can be performed. This philosophy is contrary to Squeak, which freely allows incautious users to destroy key functionality of both applications and system software.

This thesis proposes to bridge the gab in the customization spectrum between scripting languages and Squeak. Squeak does not guide customization enough, while scripting languages guide customization too much. To find a suitable middle-ground, the concepts of combination, composition and commands, as used in CLIs, are introduced. In UNIX shells, for example, combination is performed through order of commands or the `&&` operator, composition is performed through data flow via the `|` (pipe) operator and commands are manifest as small, re-usable applications that form the building blocks of a UNIX system. The platform described in this thesis, the *Interface Manager*, bridges the gap between scripting languages and Squeak by using these concepts to build the functionality of a GUI. There are existing GUIs that use combination and composition of commands, but none that allow all three to be used in interface customization. For example, the Oberon operating system's GUI is built entirely out of textual commands, as shown in Figure 1.2(a), but the only tool provided to utilise the command system is the Oberon programming language [30]. As a result, using Oberon's GUI often amounts to

writing programs in Oberon, which offers too little to a novice user.

Another example, Apple Events, encapsulates interface functionality in MacOS. There are various customization tools that sit on top of Apple Events. Apple Script is a scripting language that utilises Apple Events as API calls, providing a broad method of customizing as with any scripting language, but across all applications that use Apple Events. Automator, shown in Figure 1.2(b), is an application that allows Apple Events to be combined and composed exactly as in a CLI. It promotes combination and composition to first class citizens, but only for Automator's GUI, not MacOS's. The goal of the Interface Manager is to provide a GUI that presents commands, combination, and composition as first-class citizens in order that customization of functionality can be made directly to interface elements, instead of in between as in Automator.

In doing so, the Interface Manager provides guided customization, as do scripting languages, to every part of a UI, as does Squeak. Guided customizations of functionality are made on an Interface Manager based interface (a *managed interface*) by specifying combinations and compositions using a visual editor which more resembles an interface builder than it does Automator. The reason for using an editor similar to an interface builder is to allow customization of each piece of an interface, as Morphic does. Thus, an interface can be completely customized both cosmetically and functionally, through a technique simpler than editing source code. And, if required, additional commands can be created using traditional programming languages. Combination and composition under the Interface Manager is limited in scope only to the commands provided, as with Apple Events, which gives the advantage of allowing functionality to be shared system-wide, increasing the likelihood that existing functionality can support a desired customization. Full examples of usage scenarios are provided in Chapter 4.

More importantly, managed interfaces are mutable, user-owned objects. This has important implications to how customization happens within a community of users. Users can continue to use and modify their own interfaces in a virtuous cycle regardless of updates to underlying software, an improvement over the way open source software modification works. This includes novice users because of the lower barrier to customization provided by the Interface Manager. Furthermore, designers can examine user modified interfaces in order to produce better default interfaces, requesting developers implement the commands required. This is a new channel of communication between users and developers much richer than bug reports or focus groups: users can, through their own customization efforts, demonstrate to application developers what they want from an interface. Thus,

the Interface Manager creates a system of user-initiated innovation in software that does not currently exist, even in the open source community. Detailed discussion of this idea is provided in Chapter 5.

Squeak is not the only system to provide complete customization, so other completely customizable systems will be examined in Chapter 2. Furthermore, providing system-wide customization of interfaces separately from underlying applications can be seen as allowing for *interface evolution*. The method by which CLIs facilitate interface evolution will be analysed in depth in Chapter 3. Chapter 4 uses these techniques to produce the technical requirements for building an Interface Manager, the implementation of which is described in full, including various examples of its usage. Finally, discussion of the potential role of an Interface Manager in computing and drawbacks of the prototype design is found in Chapter 5.

Chapter 2

Background

A user interface allows users to interact with software. Having an interface to software immediately asks the question: what is a good interface? Some of the many measurements used in determining the benefit of an interface are relatively objective, such as efficiency, time required to become proficient and whether or not it allows relevant tasks to be accomplished. Other measurements are more subjective, such as the interface's aesthetic quality or how enjoyable it is to use. What is common to all of these measurements is that they vary both across interfaces and across users. The goal of interaction design is to produce, of all possible interfaces, the one with the most qualitative and quantitative utility according to these metrics. This thesis focuses on a related goal: how an existing interface, designed to be maximal in various metrics, can change in order to deal with the variability among users.

Design trade-offs exist because of the variability of users. For example, an interface that is aesthetically pleasing to one user may be hard to read for another user with poor eye-sight. An expert user may be held back by an interface that is very easy to learn, but lacks expert features. Not all users perform the same tasks with a word processor, for example, so it may be deficient in features for some users, while perfectly satisfying others. A change making able-bodied users more efficient may be a serious drawback for motor-impaired users.

A variety of solutions have been proposed for these problems. A brute force solution would be to ship software with a multitude of static interfaces, one for each user. In this limit the solution may seem ridiculous, but many interfaces implement two separate interfaces: a fully featured one for expert users that has a steep learning curve, and an easy-to-learn, but less powerful interface for novice users. Unfortunately, this solution does not scale, so making the interface more

Level of Customization	Example
Configuration	preference dialogs, rc files
Composition	macros, programming by demonstration
Enablement	scripting languages, extensions
Complete Customization	CLIs, Oberon, Squeak

Figure 2.1: The levels of customization.

dynamic may be tried. Joerg Beringer describes multiple levels of customization [4]: *configuration*, composition and enablement. The simplest of these are configuration tools such as rc files or preference dialogs, that appear in almost all interfaces. They are augmented by system level configurations for decisions like font choices. Configuration is a tool that provides many possible static interfaces from a single dynamic one. Solutions of this type are very successful: configuration is often *the* solution employed to provide interfaces that are well suited to a variety of users.

Suppose, however, a user needs a special feature that is not provided by configuration tools. In order to add new features to an existing interface, enablement is required. Enablement allows functionality that was omitted during design and development to be added post-deployment. One of the simplest ways of providing it is to provide a scripting facility. For example, Emacs can be extended by writing LISP scripts [32] and Microsoft Word can be extended by writing VBA scripts [12]. Using a scripting language, users can author, or hire somebody else to author, new features for an existing interface.

But enablement is not always enough. What can be done for a motor-impaired user who loses efficiency in trade for another user's aesthetic improvement? A configuration tool for choosing an alternate layout of interface elements could be provided to improve efficiency for this user, but no two users have exactly the same kind of impairment. This problem is similar to that of providing an impossibly large number of static interfaces: interface designers simply cannot provide enough configurations to suit the customization needs of every user, not to mention the increasing difficulty of configuration. Such modifications require a deeper level of customization, that which I call *complete customization*. A completely customizable system allows users to tailor all parts of all interfaces post-deployment, system-wide if desired. Note that a system need not allow interaction style to be modified in order to provide complete customization: a CLI may provide complete customization though input and output is all necessarily performed through lines of text, and a WIMP GUI may provide complete customization though it will continue to use

windows, icons, menus and pointing.

The next section further details the merits of customization, outlining research showing the desirability of customization by users. Following this is in depth discussion of completely customizable systems. These systems include Unix shells and other shell-like UI's, command-driven systems like Oberon and Acme, and, finally, Squeak. Each of these systems contains tools for providing complete customization that have not been adopted by modern widget-based GUIs, and provides necessary background for developing the Interface Manager.

2.1 Motivation for Customization

Any user would rather use an interface perfectly tailored to their requirements and abilities than one that is not. But user-made customizations can be flawed: a user can easily make customizations detrimental to their efficiency and satisfaction when using an interface. Providing a solution that guides users to make effective customizations is outside the scope of this thesis, and readers interested in this topic are instead refered to other works [6]. Herein, it is assumed that users understand which customizations will provide interfaces better suited to them.

The desirability of customizations that alter an interface to suit a user's abilities is fairly obvious. But such customizations are often not available: difficulties caused by this missing feature is demonstrated in a study conducted by Gajos, et al. [11]. Motor impaired users were found to be much less efficient than their able-bodied counterparts when using modern interfaces, such as print dialogs. Their efficiency increased greatly when using specially tailored interfaces, provided by SUPPLE [10]. SUPPLE is an adaptive platform that can automatically tailor interfaces written on top of it in various ways, such as accommodating screen size and type. In the study done by Gajos, et al., SUPPLE was used in conjunction with motor and eyesight tests to automatically make widget choices that accommodate deficiencies in users. It showed great success in increasing performance for motor impaired users, but SUPPLE does not solve the problem of allowing existing interfaces to have the required post-deployment tailoring performed, as special, non-functioning interfaces running on top of it were authored in order to conduct the study. Few systems perform customization complete enough to perform the tailoring that SUPPLE does, and without the ability to modify existing interfaces in arbitrary ways, these automated tailorings are only able to customize contrived interfaces of little to no value.

Also, while it is clear that customizations allowing an interface to suit users needs are useful, do users desire the ability to customize the functionality of interfaces? Arguments against this come naturally, as the desktop metaphor so prevalent in modern interfaces was created on the principle that users simply want to get work done, if possible without introspecting about the interfaces they use [7]. Furthermore, HCI research often judges the utility of an interface largely on the basis of untrained use, because the success of novices is perceived to be indicative of intuitiveness. Intuitiveness is a laudable goal, and this thesis recommends that designers should continue seeking it. The desirability of post-deployment tailoring is largely the result of good design. A tailor does not waste time modifying a t-shirt, and users are unlikely to customize a truly inadequate interface.

However, users *do* wish to customize their interfaces, as research has shown. For example, in 1996, a group of 101 diverse WordPerfect 6 users were surveyed, and 92% were found to have made customizations to its interface [23]. Another study conducted by MacLean, et al. at EuroPARC shows that a tailoring culture can be introduced into an office environment through training. Office workers, the end-users most often described in literature, were provided with rich, scriptable desktop "Buttons." A desktop Button is an object on the desktop interface representing LISP code that could be executed by a mouse click. The LISP code for each Button is readily available and can be modified. Eventually, with training, even the most novice users *demanded* customizable Buttons [17]. Perhaps most importantly, this study found that users took ownership of their interfaces. When the study began users spoke of Buttons on their desktop as though they were foreign, but eventually made statements such as "I don't know what I'd do without my Buttons" [17]. This use of possessive nouns shows, perhaps better than anything else, the desirability of customization. It may take time, but users ultimately desire the ability to customize.

Even if MacLean, et al. had shown customization can not be taught to end-users, it is a poor assumption that end-users alone will be engaged in customizing. All end-users borrow and modify customizations made by co-workers and other acquaintances. Co-opting the customizations of peers normally occurs within a tailoring culture and is a reasonable way to take advantage of customization and to learn about making them. The sharing of customizations among users has been observed in studies of real-world instances of user-initiated innovation, such as in the sport of wind-surfing [37]. As wind-surfers started to build more advanced equipment through customization of their current equipment, many different customized designs started to appear. The surfers helped each other with their customizations,

and pushed the sport of wind-surfing to its current state.

In short, customization is desired by users as a tool for tailoring interfaces post-deployment. Users benefit in terms of efficiency, and are more engaged with their computers when they can modify their interfaces to suit their needs and abilities.

2.2 Completely Customizable Systems

Several completely customizable systems exist, all varying in both interaction and customization style. Older systems that are completely customizable like UNIX and Oberon do not provide widget-based GUIs as modern systems do. Squeak provides complete customization of a widget-based GUI, but has its own drawbacks.

2.2.1 Unix CLIs

The command-line interface (CLI) provided by Unix systems is restricted by the era in which it was created. The primary method of input and output in Unix is through per-line text-based input followed by textual output. However, the level of customization it provides is likely the reason that Unix-style shells have such longevity. To be clear, the CLIs discussed in this thesis refer to this interaction style and not the applications that are executed by a CLI. A CLI-based application can present its own UI with quite a different interaction style, and this is not what CLI is intended to describe here. This property does reveal the most important customization tool in Unix, though: the text editor. Under the Unix paradigm, text editors play a role more diverse than a word processor, as they allow scripts of command-line input to be collected into a single, executable file. When these files are placed into a central location searched by the *command interpreter* for executables, they are virtually indistinguishable from binary applications compiled from C code. The multiple levels of abstraction provided by script files is one of the things that makes Unix CLIs so customizable.

However, the real key to Unix's customizability comes from its use of a command interpreter. CLIs use a variety of *shells*, the applications that takes a line of input (or a script) and transform them into a series of command calls, normally allow commands to be linked together through various means. The default language provided by nearly all shells is a *combination and composition* style programming language: functionality from different pieces of software can be composed together and executed in sequence automatically. Combination is performed through sequence,

as commands can be executed in order and composition is performed through the data-flow language of pipes. With the command interpreter between a query, such as a combination and composition of commands, and the applications that those commands represent, the input to the command interpreter can form programs. Furthermore, customization of how the command interpreter modifies input given to it allows CLI interaction to be customized. Such tools are commonly called built-in commands. For example, modern shells have an `alias` command that allows one string of text to be substituted for another. Its standard usage is to make short macros for longer commands, such as `alias ll='ls -l'`. With this in the alias table, entering `ll` into a the shell will execute `ls -l`. More complex usage of aliases exists. One example of more complex usage is performed by modern Linux distributions, where the alias `ls='ls --color=auto'` is inserted into the alias table by default so that the command `ls` will give coloured output by default.

Combination and composition is an excellent system of end-user programming and will be further examined in Chapter 3, but Unix CLIs could provide complete customization without it. The Bourne again shell (bash) [29] is a popular Linux shell that descended from the Bourne shell (sh) [5]. They both rely greatly on a combination and composition language. The C Shell (csh) [1], on the other hand, provides a shell with a grammar similar to the C programming language so that it can have more utility to C programmers. Even programming languages like Python [2] have been known to be used as glue between applications when large amounts of custom processing between them is required.

This leads to the crutch of Unix's complete customizable CLI: the scope of its applications. Each application is supposed to be small, performing a single task, such that it forms an atomic command. Not only does this allow for combination and composition, but users that create their own applications, whether they use a compiled or interpreted language or a shell script, and then integrate them with existing applications. This is an effective way of providing customization through enablement, and it works so well due to the nature of Unix. Most GUIs do not use a command system, so they cannot provide enablement in this way.

But applications in Unix are not necessarily small composable tools. Keeping applications interoperable is only a guideline, and many modern applications that can be executed from a CLI break this rule and run a different UI (sometimes a GUI) by default. This is the greatest weakness of CLIs: as soon as applications break the guidelines of interoperability, the whole system falls apart. The composability of applications on CLIs relies upon the fact that they can all supply ASCII input and output in order to allow composition to work.

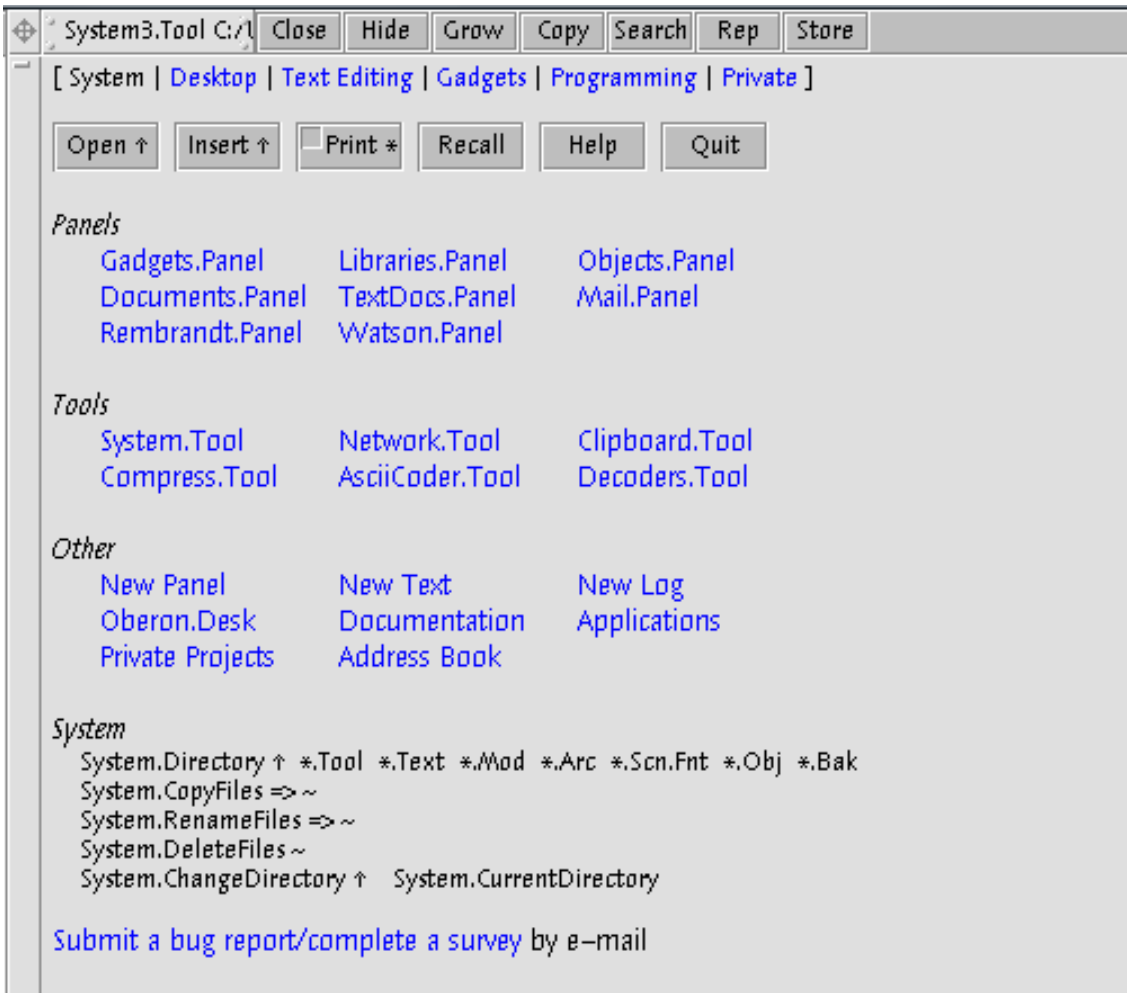


Figure 2.2: A screenshot of Oberon's interface.

Oberon and other single-language environments improve upon this flaw by specifying more clearly what commands are. In addition to this, Oberon provides a GUI. As such, it is a desirable upgrade, in terms of customization, to CLIs.

2.2.2 Oberon

In 1985 Niklaus Wirth and Jürg Gutknecht began to develop an operating system and programming language called Oberon [30]. While Oberon had many noteworthy features, its most overlooked feature was an unusual and novel GUI. The entire Oberon interface consists only of text. All text is editable, and any text can be executed as a command. The result is an interface that has been coined a "full-screen user-editable menu." The Acme editor written by Rob Pike [27] is heavily influenced by this style and provides a similar full-screen editable menu interface.

This unorthodox GUI style is shown in Figure 2.2, a screenshot of Oberon.

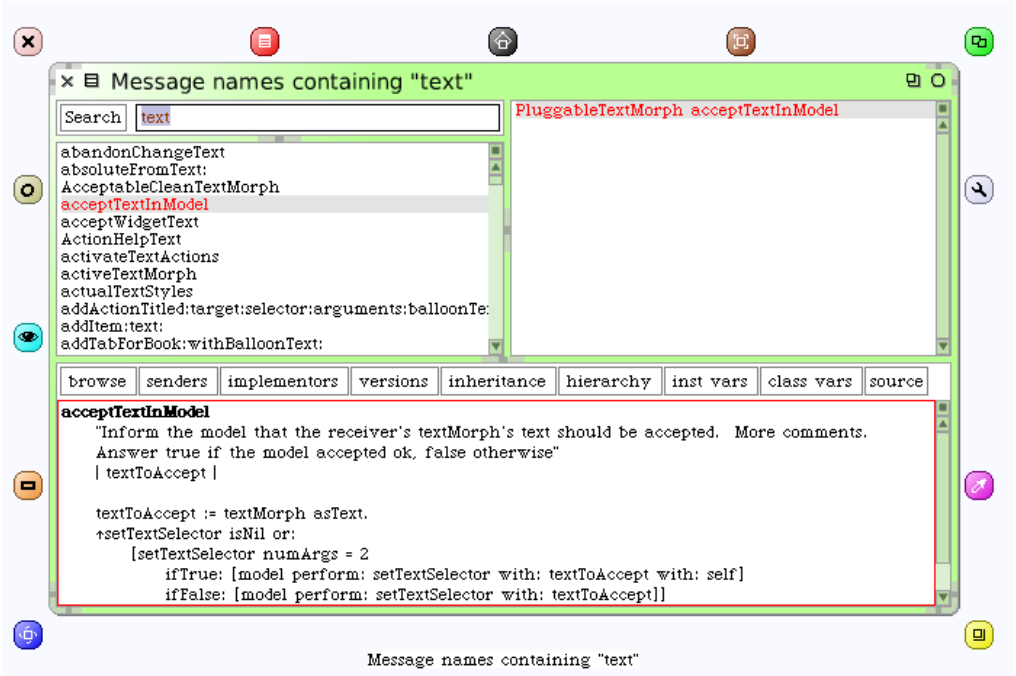
The most important feature of Oberon's all-text GUI for this thesis is command execution. A mouse button is bound to execute text underneath the cursor as a command, no matter what that text may be. For example, the text can be a command within a title bar, a demonstration command in a manual or a command in a list of frequently used operations. A command in Oberon's UI is any function exported by an Oberon module, where a module is an atomic piece of software in the Oberon programming language. The text used to execute a command is given in the form "Module.functionname." Thus, when the user presses execute over text that represents a command, a function from a module is called. The advantage that this has over a CLI is in scoping: a command is well defined as a single function. Programming language functions are also naturally limited in size by maintainability. It is desirable to write a single function for an atomic task in order to increase understandability and maintainability of code. This feature keeps the scope of commands reasonable for end-user programming, without requiring as much discipline in understanding what the limits of a command given to the UI should be. Instead, a programmer need only think about the common programming problem of whether a function should be internal or external.

While Oberon has some weaknesses in its implementation, its key weakness is that it requires its user to be a programmer. This design decision was intentional, in order that a full-screen customizable menu could be provided, but it means that novice users without programming experience are left in the cold. Its command style is better defined than a CLI's, and it provides a GUI, but it fails to support the same guidance in customization that the CLI's combination and composition language does.

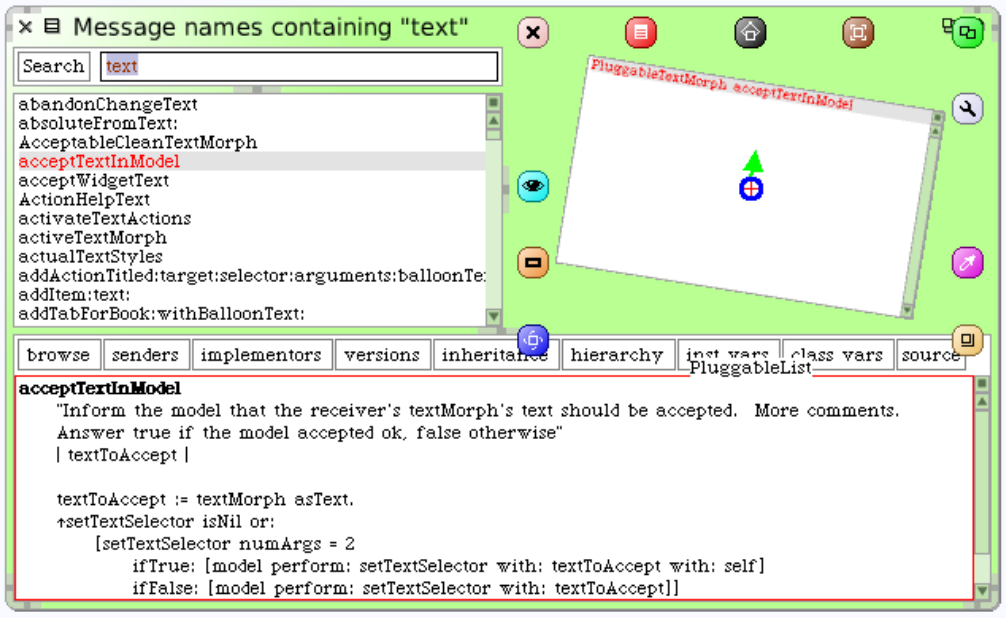
Also, Oberon does not provide its customizations to a modern widget-based GUI, the most popular modern interaction style. This is not a core problem with Oberon, but instead one of the reasons why it is important to examine Morphic, the only completely customizable widget-based GUI.

2.2.3 Squeak with Morphic

The Self programming language [36] was initially conceptualized and developed at Xerox PARC, but in 1990 Sun Microsystems began funding the Self team to create a complete environment based on the language [14]. In 1995 Self 4.0 was released, containing a user interface called Morphic [18]. Morphic was later adopted by an



(a) A screenshot of Morphic.



(b) A submorph being rotated.

Figure 2.3: The Morphic widget toolkit provides advanced functionality not found in modern systems. The window shown here is Squeak's method inspector. It has been selected to edit by using the middle mouse button, so a ring of possible commands is shown around it. In 2.3(b) the middle mouse button has been clicked again over top of a submorph list to select it and a rotation was performed.

open source Smalltalk derivative titled Squeak, and is currently easily downloadable in that form [9]. Note that the persistant interface provided by Morphic is similar to that provided by Smalltalk [16], among other systems. In such UIs, each interface component is a malleable, user-modifiable object. Morphic uses this property of Squeak in order to provide directness and liveness. No system yet has met it in terms of these goals: Morphic provides complete customization to all users, for any interface element, at any time, through direct manipulation.

In Morphic, liveness is attained by allowing all editing to be done at run-time, and it provides no distinction between running an interface and editing it. Directness is achieved by allowing direct manipulation of every component of an interface. In order to facilitate both of these features, a specified mouse button is used as an edit button. Each widget in Morphic, referred to as a morph, is selected by clicking the edit button while the mouse is over it. This method of editing is more live than any other interface builder: it requires no modifier key and can be done post-deployment. A selected morph has a ring of icons around it representing various operations that can be performed on it, such as modification of layout and rotation as shown in Figure 2.3(a). Each subsequent press of the edit button while over top of a morph allows the user to drill down and select the currently selected morph's children (provided the mouse is over a child morph), as shown in Figure 2.3(b). This allows for live modification of both layout and structure - Figure 2.3(b) shows a submorph that has been rotated.

The power of modifying a morph is not limited to Morphic's live editing system. Each morph has a body of Squeak code underneath it that can be both examined and modified at run-time. This is where traditional programming knowledge becomes required to make edits. A lot of work can be done through Morphic, but advanced customizations will require a user to program in Squeak.

While Morphic is still ahead of its time in terms of liveness and directness, it has not been adopted widely. One of the problems with adoption is that it runs on top of Squeak. With the notable exception of the Squeak based Seaside web server, it is difficult to find "real world" applications written in Squeak. It is usually thought of as a toy environment to be used for teaching programming concepts, since that was largely the reason for its inception [14]. Further flaws with Morphic are more trivial, including an outdated look and feel. This is partially a problem for adoption from an aesthetic standpoint, but also in terms of interaction, since modern widget-based GUIs have become fairly standardized in affordance and operation over the years in ways Morphic has not changed to accommodate. There are however, more important problems with Morphic.

A more critical flaw in Morphic, unrelated to its incredibly steep learning curve is the way Morphic implements live editing. On the design of Morphic, John Maloney states that purposefully avoiding the common "run/edit distinction ... has several advantages," including removal of need for frequent mode changes and freeing cognitive burden required in remembering the current mode [18]. It is difficult to justify these claims. It is not expected that a user will customize an interface so frequently that mode changing will provide significant overhead. There are only two modes, so indicating when a system is in edit mode through visual cues should not be problematic. In fact, Morphic actually raises problems through its insistence on liveness in this way. Maloney states himself that "modeless editing of potentially mouse-sensitive composite morphs poses a number of interesting problems." The problems he cites are distinguishing editing gestures from operating gestures, disambiguating spatial references, identifying the operands of an operation, and manipulating submorphs in place. These problems are indeed interesting, but they are *all* introduced as a trade-off simply to avoid the perceived overhead of mode switching, which produces a far less severe penalty. Disambiguating gestures to afford editing is hard, and in the best case, users might accidentally use a morph that they wish to edit. Far more grave is the reverse problem: since there is ambiguity between editing and using, users might accidentally edit a morph they intend to use. The consequences in this mistake could be critical, as a user could go as far as accidentally removing features required for a task in progress.

These issues, however, could be fixed with tweaks to Morphic. The biggest problem with Squeak, as a system of customization, is that it requires users to program to perform complex customizations. The amount of customization it allows provides a goal to reach for in terms of providing complete customization, but novice users are left without an easy way to customize functionality.

2.3 Conclusion

Previous research into completely customizable systems yield promising concepts not yet fully explored. First, it is clear that users desire customization, even if it requires programming. Assistance may be required for novice users to perform desired customizations, but tailoring cultures have been shown to be a natural extension of the availability and desirability of customization. Along this line of thought, it is also true that an interface can be better tailored post-deployment if more of it can be customized.

Thus, completely customizable systems exist: they allow customization of all interface properties, but not interaction style. Unix shells provide a powerful combination and composition language, but rely on applications being atomic operations, an attribute that is not necessarily true. Oberon and Acme present UI's with spacial interaction that retain the concept of interface commands. Almost as importantly, Oberon better defines the scope of commands to be single functions in the Oberon programming language. However, both of them require that their user is a programmer to an extent even further than Squeak, a large barrier to adoption by novice users. Squeak, along with its GUI Morphic, provides a completely customizable widget-based GUI. Unfortunately, it has a very steep learning curve, mostly due to the requirement of learning Squeak, partially due to the complexity of Morphic's customization facilities.

Nearly all modern systems employ various programming languages either compiled to machine code or interpreted at run-time, using event-driven widget-based GUI toolkits as a UI. With the exception of CLIs, all of these completely customizable systems exist within a single programming language. This is undesirable in a modern system. And while Morphic is an event-driven widget-based toolkit, it is far less elegant than modern toolkits (it is, after all, over a decade old).

In an attempt to solve these problems, The Interface Manager introduced in this thesis provides a completely customizable, modern widget-based GUI that can use functionality from nearly any programming language. Its visual interface editor is moded, as to make dangerous operations more difficult to perform, and it uses a CLI style combination and composition language to reduce the initial learning curve for modifying functionality. This allows novice users to make compositions without requiring them to be programmers. In this way, existing functionality can be reused through composition and, unlike existing systems like scripting languages, applied directly to interface elements. This system is described in the following chapter.

Chapter 3

Designing a Command-Driven GUI

The architecture described herein, the Interface Manager, provides a completely customizable, modern widget-based GUI. Beneath this GUI, it uses a command system to allow functionality to be modified more easily than through editing source code, which is required to make functionality changes in systems like Squeak. This chapter describes precisely why the Interface Manager is required to provide complete customization for a modern widget-based GUI, and also the pieces that are required to build one.

3.1 Introduction

To make the desirability of customization more clear, some interfaces benefit from customization more than others. For example, the interface to a nuclear reactor revolves around the tasks that the reactor performs. The user of the interface is a steward of the reactor, performing tasks specific to its operation so as to achieve a goal known prior to deployment of the interface. The interface style is quite different from that of desktop applications, such as word processors. They allow many shallow operations, like adding a character at the cursor. The user applies them as needed to produce a document, the contents of which cannot be known in advance. A reactor must have a rigid interface in order to be safe. Combining functionality, for example, can have unfortunate effects unpredictable to users. But the word processor's interface should allow a user to perform tasks specific to the user's requirements and abilities, which are known only to that user. Making frequently

combined actions a new affordance is both beneficial and appropriate in this case.

Thus, when customization is desired, the more extensive the possible customization the better. Only a few systems do offer complete customization of a GUI, in the sense that all elements of their GUI can be customized by the end-user. Squeak is the best example of such a system, as it allows modification of every piece of its interface.

To allow customization, Squeak provides Morphic as its UI. However, Morphic is unsatisfactory for several reasons: its learning curve is steep, because knowledge of the Squeak programming language is required to make complex customizations; it can only run a system using a single programming language; its widget toolkit is outdated; and it requires that all applications make their source code available. Like Squeak, the Interface Manager, which embodies the concepts explored in this thesis, facilitates complete customization, but avoids the problems. In short, it alleviates the above shortcomings through flexible attachment of a customization component to a modern widget-based GUI toolkit.

In modern event-driven GUI's, interfaces are tightly coupled to applications by event systems, with every user action connected to an entry point in application code by an *event handler*. Thus, the operation performed when an event occurs, such a button click, cannot be modified without modifying source code, which stops arbitrary changes to application functionality. Scripting languages are often used to provide enablement customizations in interfaces, but restrict what new features can be added to an interface by the API available. Furthermore, special tools are usually required to graft new functionality created in scripting langauge onto an interface. For example, most modern word processors have a menu and toolbar customization dialog that allows existing functionality or new scripts to be assigned to menu or toolbar elements. This method has two drawbacks. First, only menu items and toolbars can be customized. A user cannot, for example, customize the menu customization dialog itself in this way. The second drawback is that only existing functionality or functionality authored using the application's scripting language may be assigned to a menu or toolbar element. With a system that decouples interfaces from applications, any part of the word processor's interface could have new features added, including the menu and toolbar customization dialog.

How can interface and application be decoupled? Various methods have been used to abstract event-handler calls from widgets. For example, user interface description languages (UIDLs) are a good way of abstracting an event-driven widget based GUI, but they do not mitigate the problem of tight coupling. They con-

veniently abstract widget layout and event handler calls, but fail to loosen the connection between events and event handlers. Thus, even the most advanced event-driven GUI implementations still tightly couple application and interface. This key problem is examined in detail and solved in section 3.3.

Decoupling must go beyond abstracting event handler calls by name. To make complete customization of event-driven widget-based interfaces possible, interfaces must be completely separate from applications. Instead of being coupled to an event handler that exists inside application code, widget events should be coupled only to descriptions of commands that are interpreted by the Interface Manager using functionality provided by the application, which is analogous to providing sequences of commands to a CLI. To allow descriptions of functionality to be used instead of names of event handlers, the Interface Manager uses a UIDL like any other, except that event-handling descriptions contain combinations and compositions of commands instead of event handler function names. This decoupling produces a *command-driven, widget-based GUI*, with the Interface Manager acting as the command interpreter. This structure allows interfaces to evolve without having to change the application beneath them. In other words, there is no particular piece of software supporting an interface in this paradigm, only a set of commands that exercise the functions of the application. For example, the customization dialog used to modify menu items in a word processor would connect its widgets to event handlers such that different commands are executed when its "cancel" button is clicked. Thus, this button could be modified to call a different command that warns about unsaved changes that would be lost upon execution. This is just one example of why decoupling an interface from its application is desirable; more examples are given in section 3.4.

Allowing interface functionality to be modified in this way does not solve all the problems of Squeak. By the end of this chapter, this style of decoupling will be shown to solve Squeak's requirement for both open source code and having that code come from a single programming language. Also, since modern widget toolkits are being used by the Interface Manager, it also solves the problem of Morphic's out of date widgets. However, it does not lower the steep learning curve of using Squeak. To this end, a distinctive aspect of the Interface Manager is the use of a combination and composition language that connects interface components to application functionality. A combination and composition language is not required for the command system: any intermediate description of event functionality decouples events from application code. Thus, other types of languages could be used to play this role. A clear example is provided by UNIX shells. The Bourne shell [5],

one of the earliest UNIX shells, provides little more than combination and composition of commands, while the rc shell [8] provides a well-defined programming language including the combination and composition offered by the Bourne shell. Either shell, or any other for that matter, is acceptable for providing an abstraction layer between application binaries and the CLI above them, but different levels of knowledge are required to use each effectively. Thus, the Interface Manager can use any kind of programming language to describe the actions taken when an event is fired. The reasons for using a combination and composition language are explained below in section 3.4.2.

The next requisite of a command-driven, widget-based GUI, is a command system. Modern applications are written in many programming languages: for inter-application work the different programming languages must export commands in a standard form. Furthermore, the commands must be well enough defined that commands can interact without ambiguity. To understand this problem, past systems of command encapsulation including CLIs, Oberon and inter-application communication (IAC), are discussed in section 3.5.

Event-driven toolkits with couple interfaces to applications too tightly. Yet the Interface Manager must be compatible with modern GUI toolkits. To examine this issue, a selection of GUI toolkits is discussed in the next section. Two essential aspects of choosing a toolkit style are the existing abstraction between events and event-handlers and the interface construction tools available. The more loosely events are coupled to event handlers, the simpler it is to decouple them completely. Interface construction tools, or *interface builders*, are important in providing WYSIWYG methods of interface modification, and are currently only available for use at design time. Morphic makes no distinction between editing and use in providing toolkit-level visual editing at run-time. The Interface Manager instead takes inspiration from interface builders to provide moded customization of an interface's widget choice and layout at run-time.

3.2 GUI Toolkits and Interface Builders

The Interface Manager uses the widgets of a GUI as the interaction controls of its command system. UI toolkits have existed for decades and are now a mature technology for providing modern GUI's for applications. Most relavant to this thesis are widget toolkits that can be accessed not just through source code, but also by *interface builders*, and user interface description languages (UIDLs). An

Figure 3.1: Model of widget toolkit based systems. Creation of the widget-based interface only happens once. User input and ouput is governed by the widget-based interface, which calls application code through an event system.

interface builder is a WYSIWYG GUI for building an interface using a particular toolkit. It normally employs direct manipulation to allow users unfamiliar with programming languages, such as designers, to build prototype interfaces. Though intended for designers, the ability of interface builders to support interface building by novice users is what the Interface Manager must offer to end-users. Complete post-deployment modification of interfaces by end-users is possible when editing interfaces. Morphic is successful at providing complete customization because its users have access to an interface builder based on it widget toolkit. The Interface Manager does the same, but with moded editing and modern widget toolkits. To this end, a UIDL is an important abstraction for event-driven widget toolkits. The Interface Manager needs a representation for command-driven GUI's, and source code representations are not accessable enough for easy modification by end-users.

3.2.1 Event-Driven Widget Toolkit

The most popular style of interface design and interaction language yet devised, event-driven widget toolkits will prove to be as enduring as UNIX shells. Modern widget toolkits have evolved incrementally since the release of the Xerox STAR in 1981. They kill two birds with one stone: they provide consistency and learning transfer and they make incorporating these properties easy for developers. They are tangible, re-usable interface objects that have very obvious metaphors in both user affordance and also in API. For example, once a user has learned to drag in a scroll bar (which should be natural due to visual affordances), they will be able to drag in any scroll bar. Conversely, once a programmer has learned how to place a scroll bar in an interface, they can implement scroll bars in any situation requiring one. Widget use coincides with both the adoption of desktop computers by a larger audience and the adoption of object-oriented programming (OOP) by developers, and this is not a coincidence.

Event-driven widget toolkits have some specific features that make them attractive to programmers. For one thing, they use a model-view-controller (MVC) style of encapsulation. The application provides the model, while the widget toolkit provides a presentation layer incorporating both output and input, in the view and controller, respectively. Programmers need only create widget objects and place them, then attach events to event handlers in order to provide users with an interface. As such, using a UI toolkit is much simpler than previous methods of programming interfaces, like creating a UIMS description [26]. In most toolkits, programming an event call is highly abstracted. For example, implementing "when the mouse is over this label, call `mouseOverLabelHandler`," so that `mouseOverLabelHandler` receives the control flow during interface operation is very simple, with the details abstracted away. This practice adheres to a principle that is important in designing end-user programming systems: describe *what* something does, not *how* it does it [22]. To be sure, application code is required to set up the event system, but almost all widget code is encapsulated within an API.

Another advantage of widget toolkits is the availability of WYSIWYG interface builders. For example, the simple user interface toolkit (SUIT) is an early example of a widget toolkit with visual design capabilities [25]. Its interface builder explicitly created UI code, a feature imitated by modern interface builders, like Glade 2 [34]. Other interface builders, like Visual Basic, implicitly create UI code. Unfortunately, for end user programming code generation is undesirable. When code is generated, re-compilation is required to produce working changes which lengthens the edit/test cycle, and requires access to this source code. This problem is common because interface builders are intended for design, not for post-deployment modification. However, because interface builders edit interfaces using WYSIWYG direct manipulation techniques they are said to have a much lower threshold than editing source code: it is much easier for novice users to modify an interface using an interface builder than by modifying source code [19]. Furthermore, interface builders provide the most complete customization possible: they are used to create an interface, so they can also customize any part of it.

However, using a widget toolkit's interface builder to perform customizations is unsuitable because of tight coupling between event-driven toolkits and application code. The toolkits reside in the same layer of software as applications, and are coupled to event-handlers in the application using callbacks, as shown in Figure 3.1. The reverse is also true: it is completely possible to take direct control of interface elements created in an interface builder through source code. Even when an application does not use this feature, the problem persists because control is

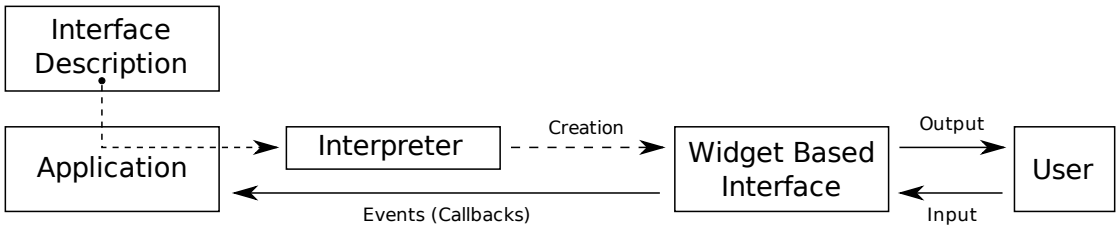


Figure 3.2: Model of interface description language based systems.

still *necessarily* available. The interface must be compiled from source code to an executable format, so there remains the possibility of interface modification from source code, not though an interface builder. The end result is that a user cannot make a change to widget based interface using an interface builder that generates source code unless they have access to the original development tools. Fortunately, user interface description languages help remove this problem.

3.2.2 User Interface Description Languages

The drawback of early interface builders was code generation. If the interface builder produces source code, decoupling model from presentation is possible only be modifying the source code. A solution to this problem modifies the process so that the interface builder creates a description (often XML based), which is interpreted at run-time. User interface description languages (UIDLs) pre-existed widget toolkits as a component of user interface management systems. Here, they are associated with widget toolkits. For widget-based interfaces, UIDLs describe three things: what widgets are chosen from the range of compatible possibilities, how widgets are configured and laid out, and what actions they initiate when their events are triggered. Complete customization of a widget-based interface can be provided by allowing these three things to be customized.

One of the reasons that UIDLs are so desirable for describing interfaces, as opposed to using an intermediate system like Python as glue between a GUI and an application, is that they *only* describe all the properties of an interface. Using a programming language like Python to glue an interface to an application allows detrimental operations to be performed, a problem in Squeak that is being avoided here. Furthermore, it has a steeper learning curve, since the scope of the language is far wider than creating interfaces, an activity mostly goverened by a toolkit API. UIDLs, however, are crafted specifically to describe interfaces and are only as broad as the capabilities of the interfaces they describe. In essence, they are nothing but

the toolkit API another language would need to use in order to display an interface, which makes them far more desirable for this purpose.

Aside from this, there are two primary reasons to use an interface description language with event-driven driven toolkit. First, UIDL based toolkits explicitly enforce loose coupling between application and UI. Second, they provide platform independence. Loose coupling by itself has enough benefit to warrant the implementation of description languages for event-driven widget toolkits. For example, Adobe claims "in sampling one of every 500 bugs in Photoshop's 20,000-bug database, roughly half ... of the bugs fell into the interface layer that the property model targets [24]." The property model is the part of Adobe's Adam and Eve system that manages all interactions between model properties and the presentation layer. Thus, a perfect implementation of the Adam and Eve UIDL and its interpreter would reduce bugs in Photoshop by *50%*. Even if perfection is not easier to obtain in implementing a UIDL instead of fixing bugs, each bug fixed in a poor UIDL implementation will provide a multitude of fixes in the interfaces that use it. This is simply in agreement with the advantages of writing modular software. In the case of the Interface Manager, the UIDL abstracts widget choice, layout and event actions. The biggest gain made by this is in allowing a command system to be used to describe event actions instead of event handlers. Furthermore, changes to any of these properties does not require recompilation or source code to produce a new interface.

Using a UIDL to perform customization by design is already possible, though it is unclear if end-user customization using a UIDL was ever intended. For example, .nib files for Mac OS interfaces created by the Interface Builder may be freely opened and edited by end-users using the Interface Builder. Other interface building systems, like QT Designer, provide similar functionality. However, these systems are still inadequate for customization. The most immediate barrier to this kind of customization is that a user needs to know that a .nib file exists for an interface, what it is and how to use an application to edit it in order to make customizations this way. Once this customization facility is discovered, the biggest barrier becomes the restriction of functionality that can be used on the interface. Only event handlers that exist inside of compiled source code, as written pre-deployment of the interface, may be used to handle events. This is equivalent to limiting interface functionality to an API. Completely customizable systems like Squeak were intended to be customized through design tools, so more visible tools that operate on intermediate descriptions are available. This is what the Interface Manager does, but it goes one step further to solve the problem of event coupling

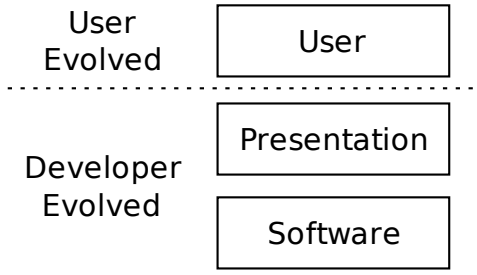


Figure 3.3: The evolving layers of software: the current model.

by underlying its UIDL with a command system, which replaces event handlers normally called from event-driven UIDLs.

3.3 Decoupling Event-Driven Interfaces

With event-driven toolkit paradigms, user interfaces must take into account three co-evolving entities, shown in figure 3.3. Note that users evolve according to research done in the field of Psychology. Software evolution is also a field of its own and will not be examined in any detail. The focus this thesis has on the evolving layers is the relationship between software evolution and interface evolution.

Event-driven interfaces evolve only when the software implementing them evolves, despite current customization technologies, mostly shallow, that they make available to the user. Possible configuration customizations are determined entirely pre-deployment, since a configuration in its nature is a designer specified choice that is left up to users. Users can change them, but not add to them. Even enablement of new features (e.g., through scripting languages) don't allow for arbitrary changes to be made in an interface: enablement tends to be provided through a feature extension dialog. For example, scripts can be written and executed from a list hard-coded into an application's UI, the elements of which can be neither changed nor replaced by the end-user. If any part of the interface is to evolve far enough to perform new functionality, the application code that responds to user input must be modified, and only decoupling can relax this constraint.

Event-driven widget toolkits implement the model-view-controller (MVC) design pattern. The model resides in the algorithms of an application, and the controller is melded with the view to form the presentation. For example, a button appears on screen as the output of a widget (view) and accepts input by mouse clicks (controller). This paradigm assumes that there is separation between the

model and presentation layers and the presentation layer contains only a thin shell of input/output code encapsulating the behaviour of the widget. The model resides within the application where all the real work is done. However, *something* must connect the two layers: event-driven widget toolkits connect the model to the presentation by an event-system.

In early toolkits, attaching event handlers to events is done bottom up: in source code a function pointer is added to an event's list of event handlers. When an event, such as a button click, is triggered, the event dispatches a signal to all event handler callbacks connected to it. This style of controller is shown in Figure 3.4(a).

Modern toolkits are a bit better, as coupling is usually performed top to bottom where event handlers in code are specified by name at the interface level for each widget event. One way to build this kind of event system is through a user interface description language (UIDL), which are described in full later. While UIDLs offer a convenient way to attach events to handlers by function name, they do not loosen the coupling between application and interface: event handlers must still be named for each event type. An interface builder using this style of event system is Glade 3, shown in Figure 3.5, which depicts steps in the editing of a button widget, labeled "My Button." The `clicked` event has been given the name of one event handler, "myBtnHandler" to call when the `clicked` event is triggered.

The problems associated with specifying event handlers by name in interface descriptions were recognized at least as far back as SUIT. During discussion of SUIT's interactive design facilities [25], a section on the drawbacks of using an event system on top of a compiled application appears:

> "Interactive widget creation is only for demos and decoration. While always impressive in a demo, the ability to create new buttons and sliders on the fly is really not very useful. Items that act merely as decoration, such as labels, can be added nicely, but most items have attached functionality that must be specified. Because SUIT is based on C, run-time binding would be much harder than in a LISP-based system like Garnet [21]; SUIT would need to compile and link code on the fly-something that a LISP-based system need not worry about. If the user creates a button interactively, how does he or she attach a callback function to it? ...
>
> "At present, the only way to do this is by hand: the programmer opens up the C source code file and makes a function call that attaches a function pointer to the button, referring to the button by its

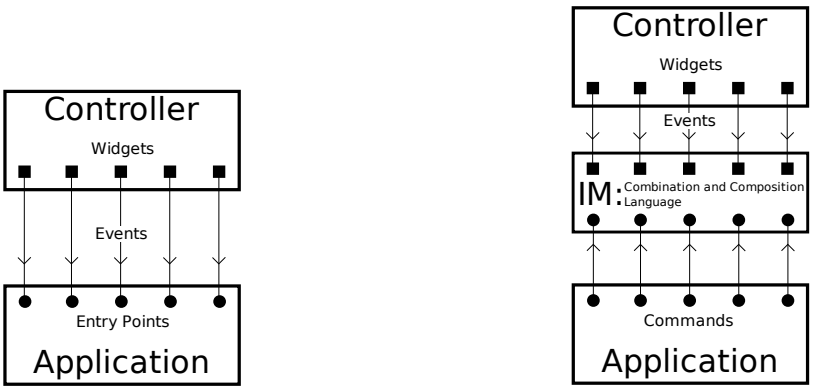
unique string name provided when the widget was created interactively. Spelling the name incorrectly results in a fatal run-time error."

As mentioned, previous systems have solved such coupling problems pimarily by not using a compiled language. Notably, Brad Myers' Garnet and, earlier, Gilt [20] provide widget functionality abstractions to reduce coupling through callbacks, but they only manage to do so by leveraging LISP. This passage states exactly the customization problem that the Interface Manager solves, as it can use compiled, closed source code for without trouble.

One solution to decoupling these layers is offered by command interpretation. On a CLI, a command can be run directly by giving the command interpreter a command name. This is analogous to calling an event-handler by name, except that the scope is at a system level, not an application level (separate per command flags can be used to specify which application functionality is to be called). The advantage of a CLI is that the command interpreter mediates the interaction between the interface and commands, so input more complex than single command names may be given. For example, the command name itself could be aliased such that it runs a different command entirely, or a composition of two different commands can be performed. In this way, the Interface Manager is inserted between widget events and application commands, offering an intermediate combination and composition language. In accordance with the MVC design pattern, the Interface Manager acts as a programmable controller, as shown in Figure 3.4(b). When an event is fired under this new paradigm, commands are interpreted by the Interface Manager instead of calling application code directly. This style of UI is a command-driven, widget-based GUI, and it performs the decoupling required to allow for complete customization of an event-driven widget-based GUI.

3.4 Command-driven, Widget-based GUI's

In a widget-based GUI modified to be command-driven, there are four layers that co-evolve as shown in figure 3.6. In this paradigm, the presentation layer no longer depends directly on the application. Instead, a given interface need only know the names of commands, nothing of their implementation. While complete customization of interface layout and widget choice on an event-driven widget-based GUI can be achieved with a run-time interface builder, complete customization of functionality is not possible. Features that are not already available in the application would



(a) Static controller provided by event-driven widget toolkits

(b) Programmable controller provided by the Interface Manager

Figure 3.4: Event-driven widget toolkits implement the controller from the model-view-controller paradigm through events, whereas an interface manager is a command-driven programmable controller that affords a combination and composition language to end-users.

require re-programming of the application to be made available. Command-driven widget-based GUI's allow interfaces to evolve separately from application code so that their functionality can also be completely customized.

To highlight the differences between an event-driven controller and a command-driven controller in practice, consider a user who is trying to print a web page on a printer for which drivers are not locally installed. With an event-driven GUI, such a task is virtually impossible, because the print button on common printing dialogs is connected to an event handler that can only print to an installed printer driver. If sending commands to a remote machine that has the driver installed has not been supplied as a configuration customization, it is impossible to change the interface so that it does so. Using a command-driven controller, on the other hand, a workaround can be written, using a combination of commands. This example is illustrated using commands available in a Linux shell. Normally printing this webpage would be done by the following command combination:

1. **jadicker$** `html2ps` *`webpage.html ./webpage.ps`*
2. **jadicker$** `lpr -P`*`printer webpage.ps`*

On line 1, the web page is converted to PostScript, a readily printable format.

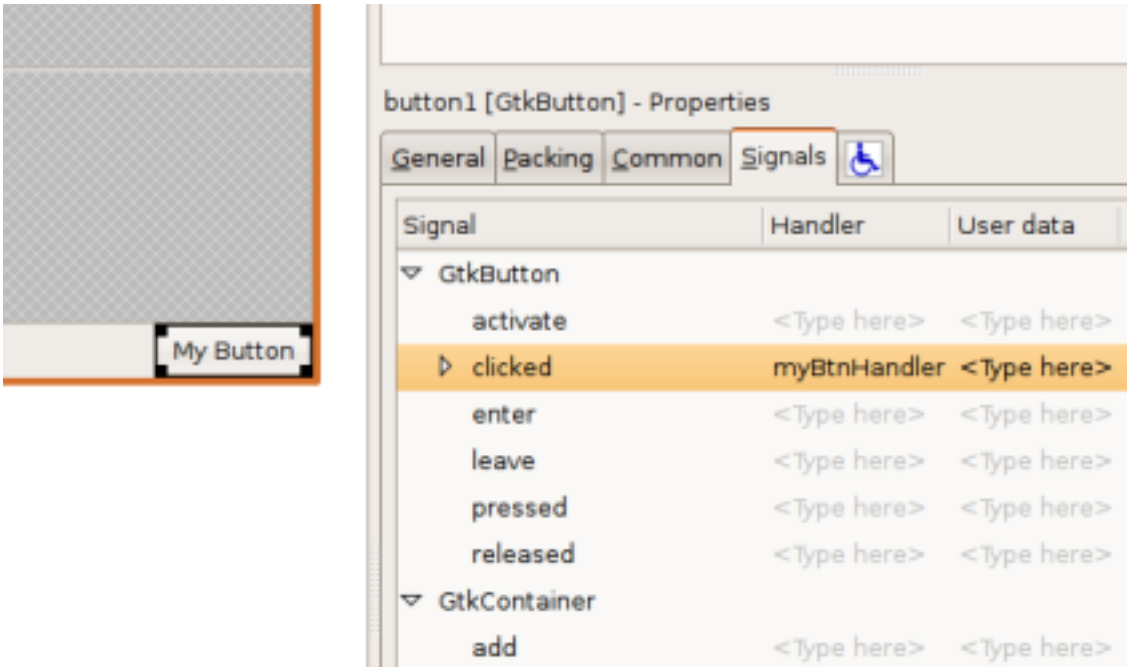


Figure 3.5: The Glade 3 signal editor. Glade 3 will create a separate interface description file that is interpreted through its API, libglade, but it still provides direct ties to software through event-system callbacks.

But with no printer installed, the `lpr` command on line 2 fails and the page is not printed. This problem is solved by the following combination:

1. **jadicker$** `html2ps` *`webpage.html`* `/tmp/webpage.ps`
2. **jadicker$** `scp /tmp/webpage.ps` *`remoteserver`*`:/tmp/webpage.ps`
3. **jadicker$** `ssh` *`remoteserver`*
4. **jadicker$** `lpr -P`*`printer`* `/tmp/webpage.ps`
5. **jadicker$** `exit`

This is a fairly complex combination. The web page is converted to PostScript on line 1, then copied to a remote computer on line 2. On line 3, the user logs in to the remote computer, then prints the web page on line 4. Finally, `exit` returns to the local computer. This combination requires five different commands in addition to knowledge about how to print on an accessible remote computer. By contrast, it is *possible*, though not necessarily easy, to achieve such a workaround using a command-driven system. Printing using commands is not tied to the application by event handlers, so additional steps can be added to the printing process. Therefore, the Interface Manager can solve this problem by allowing a print button's `onclick`

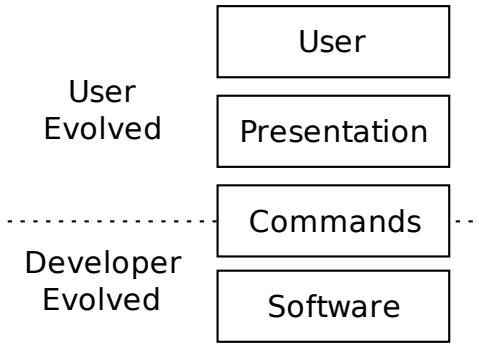


Figure 3.6: The evolving layers of software, the command-driven model.

event to contain, instead of the name of an event handler, the first CLI combination given above. Thus, the print button's `onclick` event can be modified to contain the second combination, if need be.

For the sake of demonstrating compositions, the original operation and its workaround can be alternatively performed by composition of commands. In this form, the original is:

jadicker$ `html2ps` *`webpage.html`* `| lpr -P`*`printer`*

and the workaround is:

jadicker$ `html2ps` *`webpage.html`* `| ssh` *`remoteserver`* `lpr -P`*`printer`*

Workarounds created by modifying interface functionality can be applied anywhere. The Interface Manager controls interfaces system-wide, so presentation is decoupled from model *in general*. Interfaces are thus tangible, user-owned parts of the system. Thus, whether an application has closed or open source code, its interface connects to it through the Interface Manager, and the functionality of its interface can be changed by the end user. The method by which an application exposes functionality as commands is examined in section 3.5.

As mentioned above, combination and composition languages are only one possibility. However, using them has many benefits, none more important than handling the range of possible users of the Interface Manager, from novices to experts. The reasons why combination and composition languages scale in terms of power and ease of use is given in the section following the upcoming description of target users.

3.4.1 Target Users

Who are the target users of the Interface Manager? As discussed in Chapter 2, the Interface Manager is designed to deal with the variability of users in terms of goals and skills. Therefore, interfaces based on the Interface Manager treat all users equally: there is no distinction at the interface level between designer, developer and end-user. Developers and designers still build software and design interfaces, but any user also has the option of introducing new commands and modifying interfaces.

The Interface Manager is likely to be managed differently by different users. Take, for example, replacing the find operation of a text editor with a new, externally-defined, find operation. How would different kinds of users make this modification? There are three different depths of customization that can be used to solve this problem. First, if the user is a programmer, the new find operation can be written in a traditional programming language, exported to the Interface Manager as a command, which replaces the old command. Second, if the user is an expert at using the system, but not a programmer, an existing find command from a different application can be used. Finally, if the user knows nothing about how to use the customization tools, an experienced friend can send them an improved version of the interface via email. Third party developers also play a role here, as they can be hired to replace any command in the application without having to do application-specific implementation.

Given that this wide spectrum of users should be able to benefit from the Interface Manager, it should use tools that scale from experts' knowledge to novices' lack of skill, which is why a combination and composition style command language is used to link interface functionality to application code.

3.4.2 Combination and Composition of Commands

The principle characteristic of a CLI is that it offers a wide variety of small applications that can be combined and composed in order to accomplish a variety of tasks. Combination and composition languages are a powerful customization tool and are surely responsible for the continued success of CLIs among expert users. There are real-world roots to the richness and naturalness of such languages.

An example of this comes from a user-initiated innovation anecdote given by Eric von Hippel regarding high-performance wind surfing [37]. In the 1970's, wind

surfers in Hawaii began to jump from the tops of waves. In action, this is a sequence of wind surfing, jumping, then wind surfing again, which is just a combination. There is also composition involved: the act of jumping into the air is performed *while* wind surfing. Problems introduced by jumping drove further composition. Surf boards would frequently leave surfers' control, as they would fall to the water without the surfer's feet. In order to accommodate this composition of actions, wind surfers latched their feet to the board by composing foot straps with the surf board.

Another real-world example of combination and composition is fishing. There is a combination required to fish: cast out the line, make the bait look appetizing to a fish (shake it around a little), then reel in the line when a fish bites. There is also a composition: bait and line combined together is the input to casting. Furthermore, this example presupposes that there is a mechanism to reel in the line - that mechanism is, in fact, composed with an initial fishing rod and line.

These two concepts apply equally as well to customization in a software environment. For example, there was a time when users were provided with only two of three tools that modern systems use to move text: cut and paste, but not copy. UNIX shells, for example, normally offer `ctrl+K` and `ctrl+Y` keyboard shortcuts for cut and paste, respectively. The pattern of `ctrl+K, ctrl+Y, ..., ctrl+Y` emerges fairly quickly. The first piece, cut then immediately paste, seems like it could be an atomic action on its own. The copy command was born from this combination, and is an early example of the usefulness of interface evolution. In practice, the evolution was performed by developers and released to users in a new version of the software. The Interface Manager, as is demonstrated in chapter 4, allows this evolution happen without requiring a new version of software.

What of composition? Mathematically, composition occurs when a function takes another function as a parameter. For commands, the result of executing one command is used as a parameter of another command. For example, a user who wants an application that shows the current CPU temperature graphically can base it on an existing temperature measurement command. This command is composed with a new, user created command that takes the temperature as a parameter and returns an appropriate image. The image is further composed with an image display widget, creating the desired functionality through two compositions.

Combination and composition have strong natural metaphors in action sequences and modular construction, which makes them scale understandably [31]. Once coordinated inter-muscle sequences, in an arm for example, are made atomic

by practice, they can themselves be combined into more complex sequence that carry out actions such as casting out a fishing line. With more practice, casting a fishing line itself becomes atomic and can be combined with chewing gum or daydreaming. In software this role is filled by scripts of existing commands. Combination and composition allow layers of increasing complexity to be stacked on top of one another, abstracting well-understood operations in order to allow more complex operations to be defined. In a software combination and composition language, as a user increases in understanding and skill, creation and use of new abstractions is possible, unloading cognition from the user to the computer. Because of this, combination and composition is an invaluable end-user programming tool that can be used by novices and exploited by experts.

How should the Interface Manager use combination and composition to provide a GUI above it and a command system beneath it? Choice of a GUI toolkit has been thoroughly explored in section 3.2, and the decision to use inter-application communication with a UIDL-based widget toolkit are justified in the next section.

3.5 Command Encapsulation

The Interface Manager requires a way of encapsulating application functionality into commands. Commands provided to the Interface Manager need to be system-wide, so examples of systems which already have system-wide command encapsulation are given in this section. Each is examined with respect to two primary issues: how are commands encapsulated, and how are they composed?

3.5.1 Applications as Commands Versus Functions as Commands

CLIs build their command system using applications as commands. This decision was made for many reasons, including performance restrictions during the early days of UNIX, but the power of this system remains relevant today. The real trick to using applications as commands is that the applications themselves must be small, atomic operations that do a single thing, and do it well. This is the UNIX philosophy, and it is essential for the Unix command system. When programmers fail to follow this convention, work is duplicated and applications do not contribute to their own re-use. For example, `ls | grep 'text'` searches the directory listing for the term "text." Writing an application that performs this operation in one step

is not useful, because it is easily written as a single command composition: `alias lsg='ls | grep'`. A problem with this convention, however, is that modern CLI commands provide their own UI, which makes it unclear how commands should be combined and composed. This makes the structure of a command ambiguous, a major drawback of the applications as commands paradigm.

A solution, possible for command systems in single language environments like Oberon, is to use functions as commands. With functions as commands, the structure of each command is well defined: it takes a set of parameters and produces a return value. A single function can indeed have its own UI, but it is easier for programmers to follow the convention of writing a single function than of writing an application that acts as one. Partly, it may be the presence of a type system for catching errors. Enforcing a type system allows every composition to be more guided. If types do not match, composition produces no result. Type checking is a property of Microsoft's Windows Power Shell, where .NET objects are passed between commands. It is also a property of Apple's Automator, in which Apple Events are combined when composition violates type checks. These issues are described in detail in the next section.

Overall, functions as commands is more useful than using applications as commands. But many systems, like Oberon, that use functions as commands allow only a single programming language to be used. To allow multiple languages an inter-application communication protocol is needed.

3.5.2 Inter-Application Communication

Inter-application communication (IAC) is the best way of encapsulating commands system-wide. It allows applications to share functionality regardless of their programming languages, and commands can be functions in a programming language. In effect, an IAC protocol allows developers to export application functions as commands through a software "bus" so that other developers (or users) are able to call them over the bus. For example, Apple Events are commands sent between applications using the Apple Event Manager. IAC differs slightly from its lower-level cousin inter-*process* communication (IPC). IPC is more general, as it is designed for low-level communication that can be used for more complex tasks such as parallel programming. General IPC is ignored here, as high-level access to existing application functionality falls within the scope of IAC.

IAC protocols are numerous, to say the least. On Windows, OLE, COM,

COM+, ActiveX (to an extent), and DCOM can be used as IAC protocols. On MacOS, Apple Events and AEOM are designed as IAC protocols. Linux systems have a particularly large list, but currently use DCOP and D-Bus. Finally, CORBA is arguably the most all-encompassing and complex IAC system of them all. This is merely a sample of such protocols: how to create a universal IAC protocol is an unsolved problem, and may always be because new standards will certainly emerge over time. The Interface Manager runs on Linux, so D-Bus, the new standard Linux desktop IAC protocol, is most desirable of these protocols.

Using an IAC protocol like D-Bus offers solutions to the problems presented in other command systems. A type system is enforced, so data other than text can be passed to and from commands. Multiple language bindings are available, so the Interface Manager is not locked into using commands from a single language. IAC is also robust: if a command fails or crashes it does not affect the system making the call as happens in Oberon. The only problem with IAC is that it is complicated. Plan 9's Plumber is perhaps the best attempt to make IAC simple, as it infers the type of data and which application handles that data based on internal resolution mechanisms [28]. In general, IAC tools are not so elegant, and without mature tools to abstract the complexity of dealing with a sophisticated communication bus, developing an application that exports commands can be difficult, as writing support for the Interface Manager itself is. This problem exists in any heterogeneous system, and it is a straightforward decision for the Interface Manager to deal with commands using IAC.

Apple's Automator can be seen as a compelling proof of concept that a combination and composition system can be built on top of IAC. Apple Script has traditionally been used to access Apple Events exported by applications, but Automator has recently been introduced as a graphical tool offering a combination and composition language for doing so. It guides data flow one step beyond CLIs by disallowing invalid compositions. For example, a shutdown command takes no input, therefor it cannot be combined with another command. On a Unix shell, the command `ls | shutdown now` is allowed, but `ls` produces no output: its output is piped to shutdown which ignores it. Automator disallows the composition, and attempts to produce it instead make the combination `ls && shutdown now` in Unix terminology. Automator provides command automation, as its name suggests. The Interface Manager similarly takes advantage of IAC, but uses combination and composition to build entire interfaces, not just automation scripts.

The Interface Manager decouples presentation from model in event-driven widget toolkits by modifying the paradigm to a command-driven one. Thus, the func-

tionality of an interface can be freely customized. In addition, an interface builder style visual editor is used for customization of widget choice and layout. Together, they enable complete customization of modern widget-based interfaces through a command-driven GUI. Beneath the interface, an IAC protocol allows applications to export functions as commands, accommodating multiple programming languages to be used. But how exactly is such a system implemented? The prototype, written using C++ and gtkmm, along with example usage, is described in the next chapter.

Chapter 4

Interface Management

The Interface Manager provides a command-driven abstraction for software beneath it, and both an interaction and design language to the representation layer above it. This construction is purposefully architected to facilitate complete customization by decoupling interface from software so that each can evolve separately. This chapter describes the C++, gtkmm based prototype.

First, the four components of the prototype are described in detail: the pointer table, the Interface Manager controller, the interface representation layer and the visual editor. The overall architecture is shown in Figure 4.1. Three implemented usage scenarios follow these description in section 4.2.

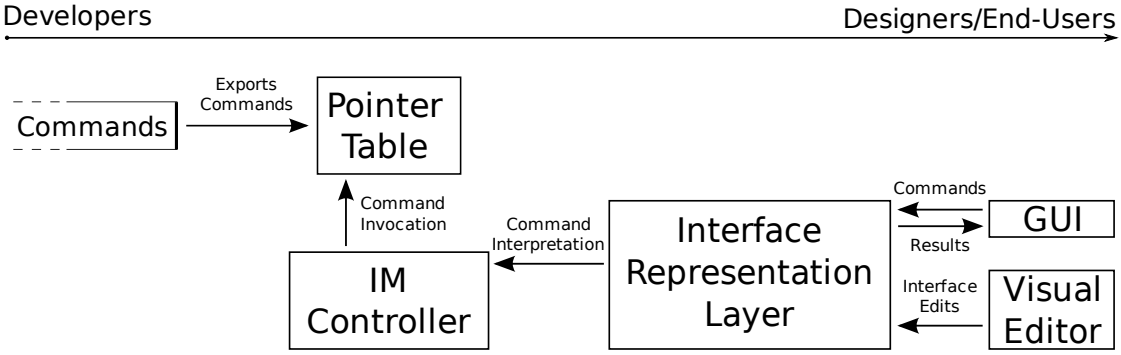


Figure 4.1: How the prototype's components work together

4.1 Components

What, is the Interface Manager? It is not a base window system, nor a window manager nor a widget toolkit. However, it should be *some* of all these things. User interface description languages (UIDLs) offer an initial solution to this identity crisis: the Interface Manager applies a command-driven UIDL to a pre-existing widget toolkit. It also offers a design language, inspired by existing interface builders. The choice of design language depends on the choice of widget toolkit. An interface manager also must provide a command layer. The command layer is the real backbone of the implementation: it is straight-forward to provide using a background process that negotiates inter-application communication (IAC).

In building the prototype, four primary components were implemented, as shown in Figure 4.1. The *visual editor* makes changes to interface widget choice, widget layout and functionality under control of the user. It provides a visual combination and composition language for functionality customization, because it is expressions, recursive combinations and compositions of commands, that the *interface representation layer (IRL)* uses to describe functionality. The IRL presents users with standard gtkmm-based interfaces; it also handles all events and serializes interfaces into an XML UIDL format. The action performed for each event is described in the IRL as an expression. When an event, such as a button press, is triggered, the IRL handles it by giving the expression to the *IM controller*. The IM controller then interprets it, calling commands through the *pointer table*. The pointer table invokes commands by executing the function pointers to shared library functions mapped to the command name given. If a command produces a result, the IM controller gives it to the IRL so that the widget takes the resulting value. These four components are described in detail in this section through the following scenario: a user loads an existing interface wanting to add a command written in C++ to it.

4.1.1 The IM Controller

The first thing that must be done to load a managed interface is to start the Interface Manager process, which runs in the background. This process and all communication with it is governed by the IM controller, shown in Figure 4.2. The user executes three Python scripts that connect to the IM controller through its D-Bus inter-application communication (IAC) backend. They are as follows:

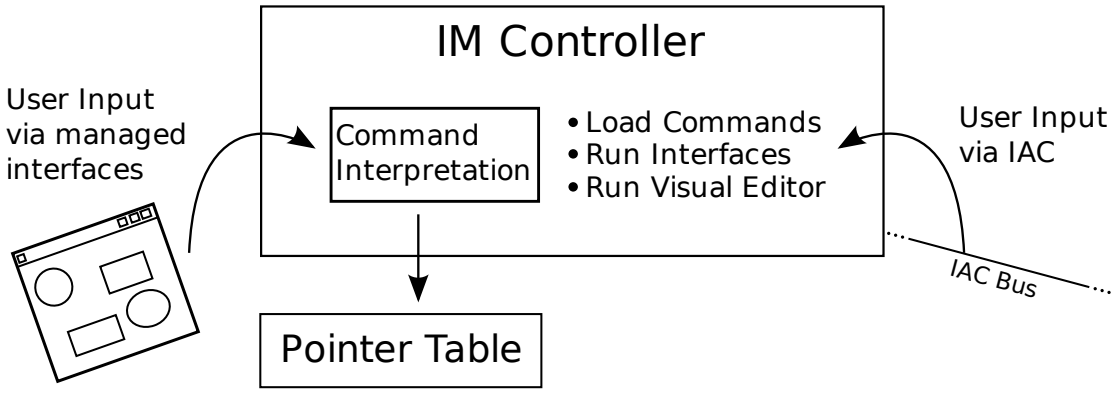


Figure 4.2: The IM Controller

1. **jadicker$** `loadlibrary commandlibrary.so`
2. **jadicker$** `runinterface interface.xml`
3. **jadicker$** `visualeditor`

The first line runs a script that loads a shared library binary file, `commandlibrary.so`, into the pointer table. This library exports functions as commands to make them available for use by interfaces. The details of this process are given in section 4.1.2. For the purposes of this scenario, assume that `commandlibrary.so` contains a C++ function called `myFunction` which is exported as a command, `myCommand`.

Line 2 runs an existing managed interface, `interface.xml`. Its XML file contains a description of widgets, their layout and functionality. The interface appears on screen as any other gtk interface would, but it is managed by the interface representation layer (IRL). This architecture is described in section 4.1.3.

After line 1 executes, the command `myCommand` becomes available for use in the visual editor, which is executed in line 3. The visual editor is a separate window used for editing and constructing interfaces. A screenshot of it is shown in Figure 4.7, and it is described in section 4.1.4.

This final script, `visualeditor`, above all the others, should make it clear that the IM controller does little work: its role is to govern user interaction with the prototype itself, in the form of running interfaces, loading libraries and executing the visual editor. All of the real work is done by the other three components of the prototype. The only other role the IM controller plays, that of a command interpreter, is explained in section 4.1.3.

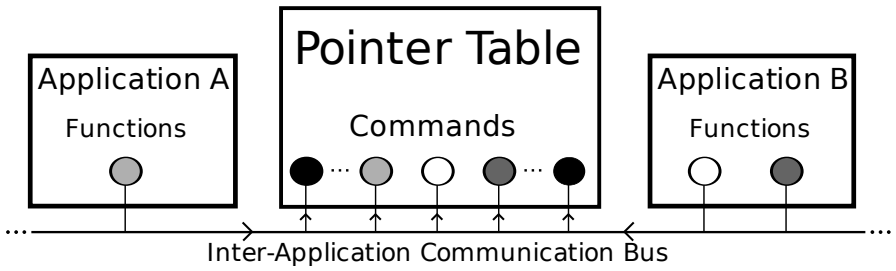


Figure 4.3: The Pointer Table

4.1.2 The Pointer Table

The pointer table owns the commands available to managed interfaces. A command in the pointer table consists of the following structure:

```
Command Structure
--------------------
application name
category name
function name
C++ function pointer
```

Thus, the command loaded from `commandlibrary.so` has the fully qualified name `myApplication::myCategory::myCommand`. As stated above, invocation of this command is actually execution of the C++ function `myFunction`, which is dynamically loaded from the shared library `commandlibrary.so`.

The actual process by which `myCommand` is dynamically loaded from the shared library consists of 3 steps. First, the user executes the script `loadlibrary` to tell the IM controller to tell the pointer table to load all exported commands from `commandlibrary.so`. Second, the pointer table calls a special function, `im_entry`, implemented by the library's author, as an entry point for the pointer table. This function returns a list of function names to the pointer table, one function name for each library function exported as a command. In the final step, the pointer table uses the list of names to dynamically load all library functions and store the function pointers in the command structures.

The previous paragraph omits a few details of dynamic loading. To perform the dynamic loading, the pointer table uses the POSIX dlsym library call, which uses

a shared library's symbol table to find and dynamically load its functions into the pointer table's address space. dlsym requires the name of a function to dynamically load, and returns a function pointer to the dynamically loaded function, if it was found. This means that the type system is enforced at this step: if a function in the shared library that is exported as a command does not have the appropriate function signature, it cannot be called by the pointer table. The required function signature takes as a parameter a list of `IMObject`'s, the super class of the prototype's polymorphism-heavy type system. The function signature also requires an `IMObject` to be returned. This means that a shared library function that is exported as a command can only use `IMObject`'s as parameters and return values. The final ommitted detail is that every C++ function name in the list returned from `im_entry` must have a corrosponding application, category and function name.

In this way, the pointer table is populated with commands. In what follows, the term *command name* refers to the *fully qualified command name*, of form `myApplication::myCategory::myCommand`. A command name, with its parameters, can now be given to the pointer table, which then returns the result of executing its corrosponding C++ function pointer.

The decision for the prototype to use dynamic library loading rather than the D-Bus IAC protocol was made to limit unnecessary complexity. Dynamic loading has low programming overhead compared with IAC protocols. Because a single programming language, C++, was used to author the prototype and all commands, dynamic loading was also adequate for language bindings. The drawbacks of this design decision are discussed in section 4.3. The decision to use only three levels names for a fully qualified command name was also made in favour of simplicity: it too has drawbacks, discussed section 4.3.

Commands must be complemented by interfaces that use them. The next section describes the interface representation layer, the component that owns the managed interfaces.

4.1.3 The Interface Representation Layer

When the script `runinterface` tells the IM controller to execute the managed interface `interface.xml`, the IM controller tells the interface representation layer (IRL) to perform this operation. `interface.xml` is a file authored (by hand or with the visual editor) in a user interface description language (UIDL). An example interface in the UIDL used by the prototype is shown in Figure 4.5, with its corrosponding

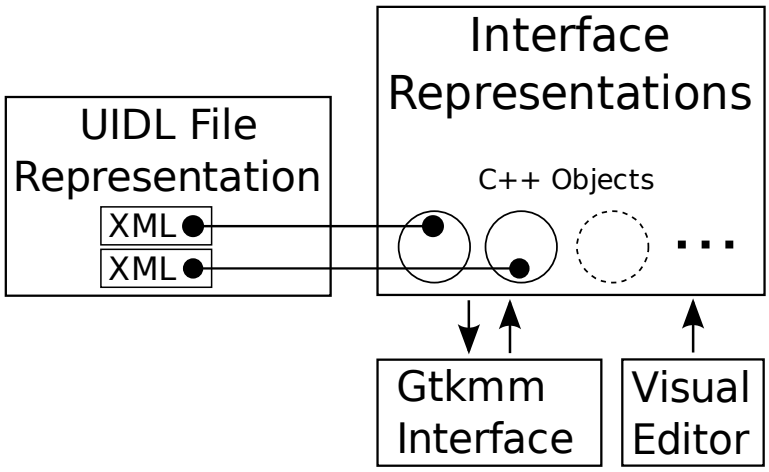


Figure 4.4: The Interface Representation Layer

interface in Figure 4.6. Research in UIDLs is complementary to my research on the Interface Manager: any UIDL including widgets, layout and event handling can be used by the IRL.

The first responsibility of the IRL is to load XML UIDL representations of managed interfaces from disk. The XML file is parsed, and a C++ object representation of the managed interface created. This C++ object contains the same information as the XML file: the choice of widgets, their layout and the combinations and compositions of commands to be executed when events are fired. The C++ object is the managed interface, the XML file a serialization of it. The one thing the C++ object contains that its XML file does not is a gtkmm-based interface.

Gtkmm is a C++ library that wraps the gtk C library. Thus, each C++ object representing a managed interface in the IRL presents the user with a gtk interface. An example is shown in Figure 4.8(a). Different from other gtk interfaces, a managed interface is mutable. Upon a request from the visual editor to the IRL that a running, managed interface is to be edited, the interface is toggled into edit mode, shown in Figure 4.8(b). In edit mode, the gtk interface is no longer operational as an interface. Instead, clicking on a widget will select it for editing by the visual editor. Furthermore, a variable pane is added to the managed interface when in edit mode. The variable pane holds temporary data that can be used in making compositions. This is done by placing the result of one command into a variable widget in order to use it as the parameter for another command. The variable pane is not visible during use.

In this way, any modification that can be made to a managed interface must

```
<interface name="factorial" serialize="true">
   <window name="Fact Win" width="200" height="150"
             title="Factorial">
     <widget name="mainBox" type="vbox">
       <widget type="label" name="fact_lbl">Factorial of:
       </widget>
       <widget type="entry" name="fact_txt">5</widget>
       <widget type="label" name="is_lbl">is:</widget>
       <widget type="entry" name="result_txt">120
       </widget>
       <widget type="button" name="compute_btn">Compute
       <!-- the main vbox contains no information -->
       </widget>NULL
     </widget>
     <vars>NULL <!-- nothing in the variable pane -->
     </vars>
   </window>
   <bindings>
     <bind name="fact_bind" event="onclick" src="compute_btn">
       <action app="Test App" category="Math" funcname="uim_fact"
             target="result_txt">
         <param srctype="widget" src="fact_txt" datatype="int"/>
       </action>
     </bind>
   </bindings>
</interface>
```

Figure 4.5: An example XML file in the prototype's UIDL. This particular interface computes the factorial of the integer in one text box and places it in another.

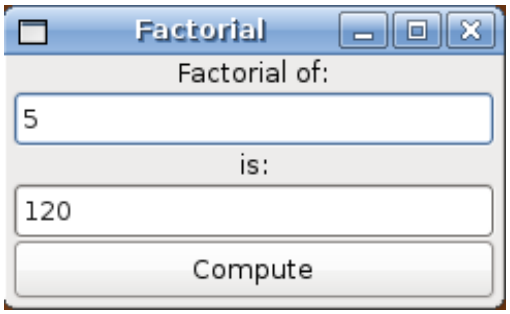


Figure 4.6: The interface produced by the XML in Figure 4.5

be explicitly made possible through modifying its C++ object representation. For example, when a widget is selected by clicking while in edit mode, the IRL provides the visual editor with a set of possible modifications for that widget. In other words, various editing systems could be used to modify an interface, but the modifications they make must be performed through IRL governed modifications to the C++ object representation of interfaces.

Before the next section, an explaination of why the IM controller is the command interpreter is given. When an event, such as a button press, triggers a sequence of commands, they are excecuted by the IM controller. The IM controller either places the result of each command into a target widget, or executes built-in functionality, such as closing the current window. In CLIs, the shell executes built-in functionality. For example, the Bourne shell provides aliases as a built-in. Composition is done by placing a result in a variable widget used by next command call, but other implementations are possible. The list of commands to call is passed atomically to the IM controller from the IRL, so composition can be done without using variable widgets: they are used to ease the implementation of composition visualization in the visual editor. Using variable widgets also makes natural the tee operation supported by most Unix shells.

4.1.4 The Visual Editor

The `visualeditor` script tells the IM controller to execute the visual editor, which opens the window shown in Figure 4.7. The visual editor consists of three parts: an edit button, a widget inspector, and an action inspector. The edit button toggles a managed interface between use and edit modes, as discussed in the previous section. This "use/mention distinction" [18] contrasts with the design of Morphic, eliminating one of its flaws, described in section 2.2.3. The widget inspector is a

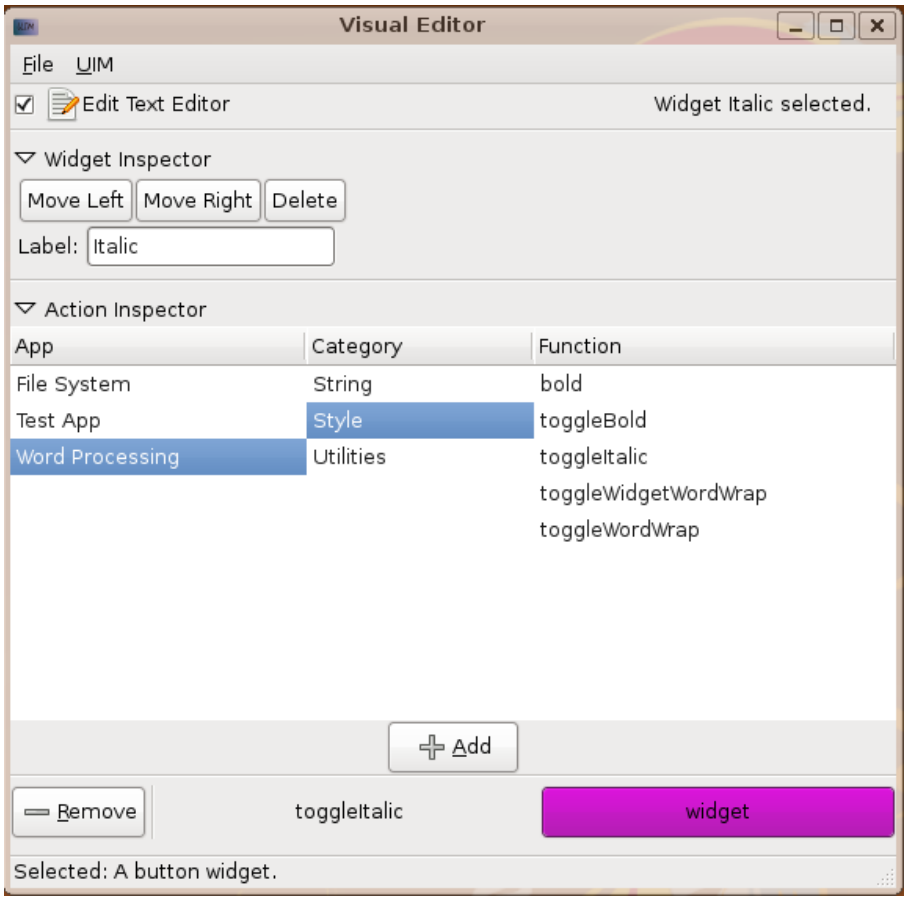
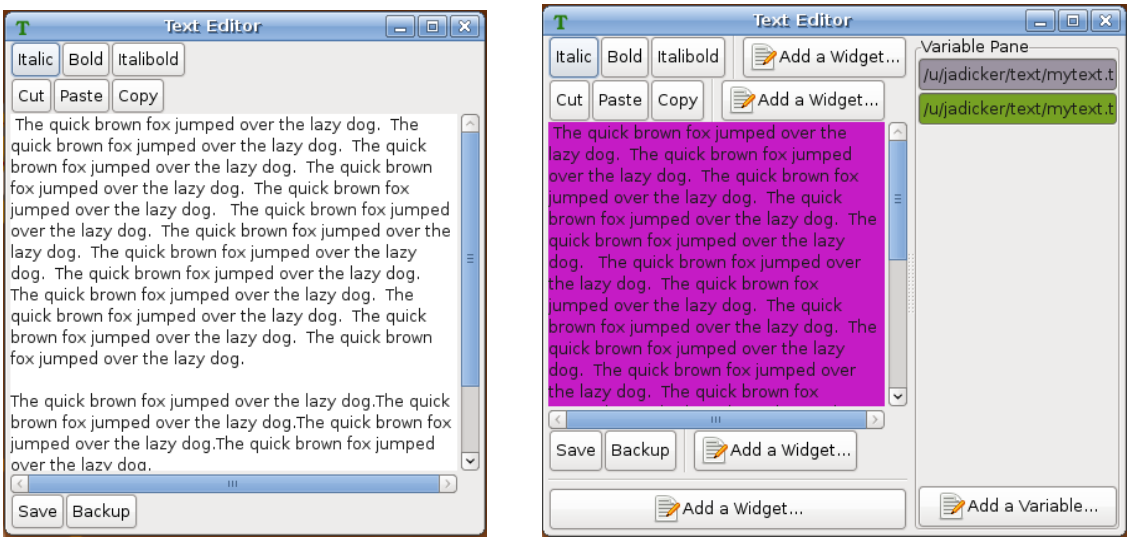


Figure 4.7: The Visual Editor. It consists of an edit button for toggling the mode of a managed interface, a widget inspector to change properties of widgets and an action inspector to explore available commands.

vanilla widget property editor. Properties such as widget text can be changed, and widgets can be moved within their parent container using up/down or left/right buttons.

The novel aspect of the visual editor is the action inspector. Available commands can be explored by selecting application and category in the list boxes, shown in Figure 4.7. This functionality is inspired by Squeak's (and Smalltalk's) system of method inspection, shown in Figure 2.3(a). When a button is selected when its interface is in edit mode, the action inspector displays the sequence of commands that are executed when it is clicked. Commands can be removed by clicking the `Remove` button to the left of the command, and they can be added by selecting a new command through the listbox interface, then clicking the `Add` button. For every command in a sequence, buttons are given for each parameter and the return value. After clicking a parameter, a user must click on the interface



(a) A managed interface, in use

(b) A managed interface, being edit

Figure 4.8: While being edited, managed interfaces reveal a pane containing variable widgets where information can be stored temporarily. This is a simple method of providing a visual data-flow programming language.

widget that will provide a value for the parameter. The return value is defined similarly. Colour is used to denote which parameter is attached to which widget. Each widget has a unique colour, and a parameter using a widget as an argument assumes that widget's colour. Any event can be modified in this way, but only button click events are mutable in the prototype.

Finally, the user adds the new command `myCommand` into the UIDL description of the interface, `interface.xml`, by making a button call the command. To do so, with the managed interface in edit mode, the user selects the button that will call the new command, then uses the action inspector to navigate through `myApplication` and `myCategory`, to find `myCommand`. With `myCommand` selected, the `Add` button adds it to the list of commands called when the selected button is clicked. In this way, entirely new, user authored functionality is added to an existing interface using the prototype.

4.2 Examples

In this section, three examples are given of customization using the dataflow language used to allow combinations and compositions of commands in the visual

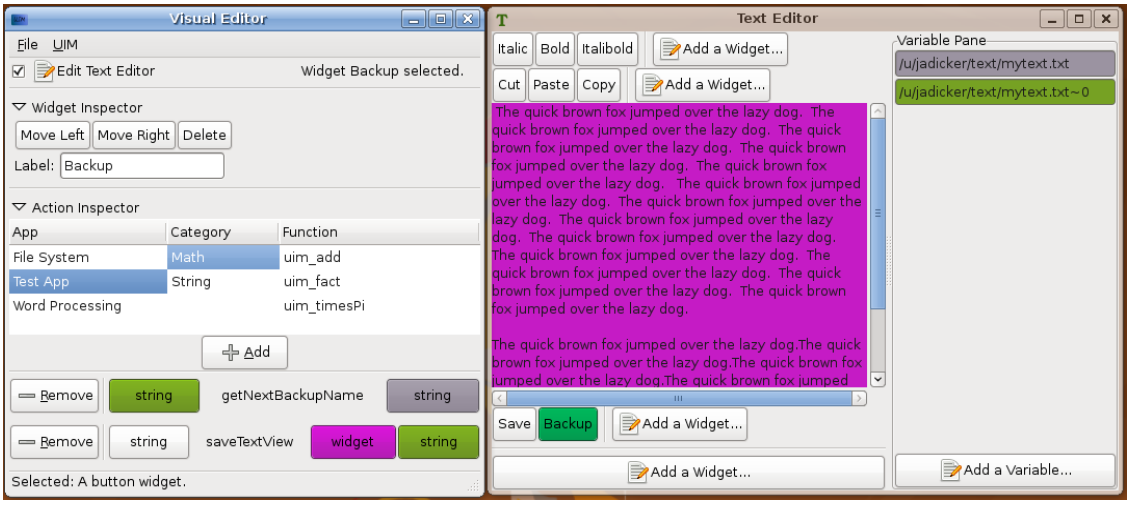


Figure 4.9: The prototype's visual editor with a text editor interface. The text editor is in edit mode, so it is augmented with buttons to add new widgets and a pane for variable widgets.

editor. The interface customized in these examples is that of a simple text editor. It consists of a text area widget and buttons that italicize, make bold, cut and copy selected text within the text widget. There is also a save button that writes the text area's contents to a text file. When the interface is in edit mode, a widget may be selected for customization. In Figure 4.9, the `Paste` button is selected, as denoted by its green highlight colour. Inside the visual editor window, the widget inspector allows the button to be moved, deleted or re-labelled. Below the widget inspector, the action inspector allows the expression executed on an event to be changed. As described earlier, the visual editor allows exploration of the commands available to interfaces, including the text editor's. They are available in the action inspector, labeled by `Word Processing`. Below the lists of available commands is the sequence of commands that is executed when the selected button is clicked. To the left of each command name are buttons for return values, to the right for parameters, which are used to choose argument widgets and to place return values. In edit mode, the text editor interface has an `Add a Widget...` button for each container, which adds a new widget from a menu to that container. For the variable pane, an `Add a Variable...` button allows the user to add additional variable widgets.

4.2.1 Application of Multiple Styles

Suppose that a user frequently makes text bold, then italicises it. Activating the combination frequently he may wish to make it the affordance of a single button.

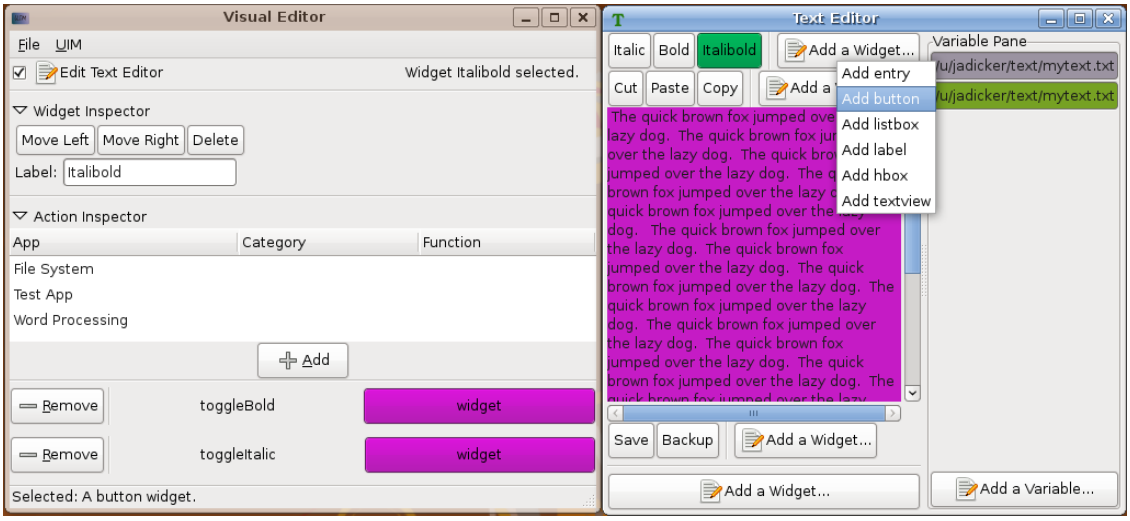


Figure 4.10: A screenshot of how example 1 is performed.

To do so, the text editor interface is toggled into edit mode by clicking `Edit` as shown in Figure 4.10. The user making this modification may not know what commands are activated by the `italic` and `bold` buttons. The answer is revealed in the action inspector when they are clicked. Using this information, the user can combine the two commands into one action. A new button is added by clicking `Add a Widget...`, then selecting `New button`. Clicking on the new button selects it for editing. The button is given the label, *italibold*, and moved to its preferred position. To create its action, the two commands `toggleBold` and `toggleItalic` are combined by selected them sequentially from the table of commands and clicking the `Add` button. Both commands act on text selected in the text area widget, toggling first the bold style, then the italic style. The new button executes `toggleBold` and `toggleItalic` in the order they were added. In this case, the order is irrelevant because the two commands commute. Though possibly trivial, this button is a new affordance, created through customization of the interface's functionality. It is also easy enough to imagine and implement to be the entry point to customization for a novice user.

What internal changes are triggered by the editing sequence? Modifications to button actions, properties and position change the managed interface's C++ object in the IRL. The XML representation is modified when the object is serialized. Once the edit is complete and the text editor interface is put back into use mode, on clicking the `italibold` button, the `toggleBold` and `toggleItalic` commands are sent from the IRL C++ object to the IM controller, which interprets them as commands and executes them using the pointer table. The remaining examples are

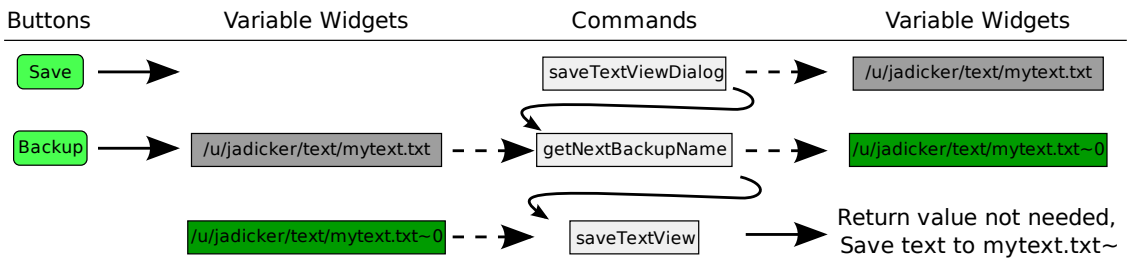


Figure 4.11: Flow diagram for example 3.

executed similarly.

4.2.2 Making Copy from Cut and Paste

A less trivial example adds `copy` functionality, constructed by combining `cut` and `paste`. `copy` can be achieved by cutting the selected text to the clipboard, then pasting it back at its original location, while remaining on the clipboard for pasting at a new location. When the cut and paste commands are combined in a single button, copy can executed with one button press. To do so, the user adds a new `Copy` button in the text editor interface, then uses the action inspector to add the `cut` command, followed by the `paste` command. Each takes the text widget, with its selected text, as a parameter. This customization is superficially similar to the first example, but more interesting for two reasons. First, there is a historical context. Copy was not a feature of early text editors. The omission was corrected in the implementations of subsequent software releases. Here the feature is added by evolving the interface rather than the software. Second, the components of the `copy` operation do not commute. This example shows that the combination and composition language is imperative in style. The next example shows this more clearly.

4.2.3 Save with Backup, an Example of Composition

The third example illustrates how composition works. In the prototype's combination and composition language, a hidden pane containing variable widgets is included in every application window. While editing, the variable pane is visible and can be used to program by side effect. Information returned from one command can be temporarily stored in a variable widget of the hidden variable pane, and are thus available as arguments of future commands.

In this example, backup functionality is added to the text editor. A more imaginative user will notice that when files are accidentally erased, a backup copy could replace the lost work. A more skilled user will realize that a new command can be combined with the two existing commands for saving the text area in order to produce a backup feature. The first of the two commands is the `saveTextViewDialog` command, which presents a dialogue in which the user provides the filename. The other command, `saveTextView`, normally bound to the `Save` button, provides no dialog and takes the filename as a parameter. Both commands return the filename, which is placed into a variable widget in the variable pane. This variable widget is initialized to the name of a file upon its first save through use of a dialog. Suppose that the user creates a C++ function and imports it as a new command, `getNextBackupName`, as described in section 4.1.2. This command takes a filename as a parameter and produces the next unique backup filename based on existing backup files matching the filename. To create backup functionality, the user starts by creating a `Backup` button. Composition is required to perform a backup. `getNextBackupName` takes as its argument the content of the variable widget where the filename has been temporarily stored by `saveTextViewDialog`, and stores the unique backup name it returns in another variable widget. Next, the command `saveTextView` takes the widget containing the backup filename as an argument, saving the text to a backup file. In doing so, the `getNextBackupName` command has been composed with the `saveTextView` command. When the `Backup` button is pressed, a backup is saved to the next unique backup filename.

This is an awkward method of performing composition, but it demonstrates the power of a language with composition. Commands supplied as part of a text editing application are used together with a C++ function written post-deployment to modify the functionality of an interface at run-time.

Furthermore, the new backup affordance could replace the existing save affordance. Instead of creating a separate button to execute a backup, the user may prefer that the `Save` button automatically makes a backup copy. To do so, the two commands required to perform a backup can be combined directly after the `saveTextViewDialog` command used by the `Save` button. It is also notable that this composition is itself a new command. It is not currently possible to name a command created in the visual editor so that it could be re-used, but having such a feature would be of great benefit.

There are a few other shortfalls of this implementation, and they are discussed in the following section.

4.3 Inadequacies of the Prototype

This prototype is not an ideal implementation of an Interface Manager. A few features need to be upgraded in a production implementation, including how commands are exported, how commands are looked up by users and the widget toolkit used.

Ideally, IAC (specifically D-bus) would be used to implement command export to the pointer table. This would make the command system more resistant to crashes, remove the dependency on C++ and improve cross-platform support, since dlsym depends on POSIX. Furthermore, if D-bus was used for IAC, the GObject type system would be enforced as the standard type system of the interface manager. The prototype currently uses an ad-hoc, OOP dependent type system, which hinders non-C++ language bindings, not to mention being inefficient and inelegant. Using D-bus with GObject would allow robust bindings to nearly any programming language, with full type system support.

Another problem is how commands are stored and looked up in the pointer table. Ideally, the pointer table includes meta data for each command. For example, author, date created, and comments on usage are useful information for finding, selecting, managing, and sharing commands. The prototype has only application, category and function name to identify commands, which is not enough. With a wider selection of meta data, a command can be found by searching this data and a unique identifier assigned to each command can be used to invoke it.

Toolkit choice is pivotal to any implementation of the Interface Manager. Specifically, the choice of widget toolkit heavily influence the design of the visual editor. For example, most widget toolkits include interface builders: both consistency and code re-use argue in favour of using it as the visual editor. The prototype uses the Gtk widget toolkit, which has Glade 3 as an interface builder. Because both are open source, Glade seems the obvious choice for the visual editor. However, Glade was *not* chosen for the prototype for reasons that are instructive. Glade does not use the Gtk widget hierarchy, but instead a doppelgänger Glade widget hiearchy, with a parallel GladeWidget for every GtkWidget. Glade *must* do this because Gtk widgets cannot contain arbitrary meta data. Meta data attached to widgets is important for many reasons: among them, Glade needs extended information about event calls to facilitate editing. The Interface Manager similarly requires such data, as each widget needs to store command names for its events. Thus, a solution similar to Glade's parallel widget hierarchy was required to implement the

visual editor, which is much less elegant that using a widget toolkit that supports meta data.

In short, it is feasible to build a completely customizable command-driven, widget-based GUI using existing event-driven widget toolkits and IAC protocols. Explanations of all components and usage examples have been given. This prototype is not complete, but a complete system is easily within reach, though it would require a large amount of routine work. The biggest open questions about the Interface Manager are not about implementation, but about how it would change interface development, whether or not it scales, and how novice users acquire expertise naturally while using it. These questions are discussed in the next chapter.

Chapter 5

Discussion

This thesis began by investigating the shortage of deeply customizable systems. For example, while Squeak allows everything to be customized, in doing so it lets dangerous operations be performed too easily, it requires knowledge of traditional programming languages, and has a steep learning curve, even in its visual editing system. More guided styles of customization, such as the combination and composition command language provided by command-line interfaces, provides complete customization, but in an interaction style that is no longer considered state of the art.

It then showed how modern event-driven, widget-based GUIs can be modified so as to decouple the model layer from the presentation layer. The modification inserts a command interpreter, the Interface Manager, between events and commands so that a programmable controller is provided to users. The result is a command-driven, widget-based style of GUI that is completely customizable. Furthermore, non-cosmetic customization of the interface is achieved by way of a combination and composition language that should make customization easy enough to bootstrap novice users. Description of a working implementation and examples of its use were given in Chapter 4.

This chapter discusses the significance of the Interface Manager in terms of user-initiated innovation. First, users can take ownership of managed interfaces, since they are malleable objects. This promotes a virtuous cycle of customization. Second, allowing interfaces to be customized like this can promote a user-driven interface tailoring culture similar to existing cultures, such as Linux script sharing. Third, the work between developer and designer is split such that developers make commands and designers design interfaces. This means that the user community

can provide innovation to designers in the form of modified interfaces, allowing for user-initiated innovation not yet seen in computing.

Finally, the chapter concludes with possible improvements to be made. These include implementaion improvements, the possibility of a longitudinal study, and discussion of security concerns.

5.1 Summary

Squeak, with Morphic, shows that it is possible to provide a completely customizable GUI. The approach Morphic takes to customization, however, is unsatisfactory for several reasons: its learning curve is steep, because knowledge of the Squeak programming language is required to make complex customizations; it requires a system implemented in a specific programming language; its widget toolkit is outdated; and it requires that all applications make their source code available. This thesis provides a solution to these serious difficulties.

The Interface Manager lowers the learning curve of systems like Squeak by guiding customizations using an interface builder for cosmetic customizations and a combination and composition language for deep customization that modifies or adds functionality. This allows existing commands to be re-used as they are in CLIs. It also allows any programming language to be used, provided that its implementation uses an IAC protocol for the pointer table. Any widget toolkit can be used in an implementation of the Interface Manager, though some toolkit-based difficulties may be encountered during implementation, as discussed in section 4.3. Finally, closed source applications work with the Interface Manager. As long as the application exports commands, it can execute applications compiled from closed source code.

The main contribution of the architecture is demonstrating the utility of inserting a command interpreter between events and event-handlers. In widget-based systems using the MVC pattern, this decoupling increases the encapsulation of applications and interfaces. The significance of modifying a system's architecture so that events are handled by expressions created from commands are far-reaching. They are discussed next.

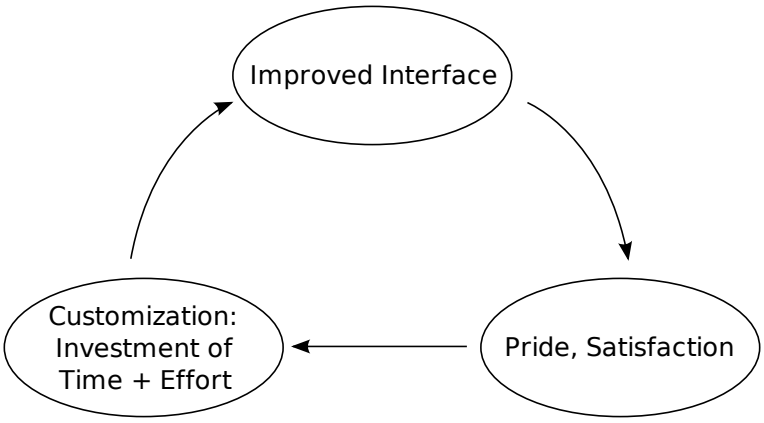


Figure 5.1: The virtuous cycle that occurs with user-owned interfaces: the more customizations a user makes, the more satisfaction they receive, the more customizations they make.

5.2 Significance

The significance of the Interface Manager goes further than giving a single user the tools to completely customize an application's interface. The architecture it provides changes the evolution of applications and of the communities that use it. As the first step, the Interface Manager liberates interfaces from applications. An application that uses the Interface Manager does not have a specific interface, but an interface created or chosen by the user from a large set of possible interfaces. Thus, interfaces are user owned. The second step is initiating change in the user community. Since interfaces are user owned objects, they can be shared, as shell scripts are, allowing propagation of interface customizations among users and jump starting a tailoring culture. The final step occurs in the development community. Splitting development between designers and developers, as managed interfaces allow may solve a few development problems, but, more important, it offers a wide bandwidth over which designers and developers can receive suggested changes to an application. This leads, in the end, to user-initiated innovation in software, a phenomenon that economists consider to be an important source of innovation, but that rarely occurs in the software world.

5.2.1 Ownership of Interfaces

The Interface Manager presents users with interfaces as malleable objects. This changes who owns interfaces. Widget-based interfaces are currently owned by ap-

plication developers: evolution occurs only with the release of new versions of the software, as described in Chapter 3. The Interface Manager turns the tables by decoupling interfaces from applications: the user begins with a default interface owned by the developer, and gradually assumes ownership as it evolves. Thus, ultimately users own managed interfaces. User ownership of interfaces produces a virtuous cycle: users invest time and effort into modifying interfaces, which deepens understanding of the application and interface and produces a feeling of ownership and satisfaction that, in turn, drives users to invest more time and effort into making modifications. This cycle is depicted in Figure 5.1. Customization exists to tailor an interface post-deployment to suit the requirements and abilities of a specific user, and continuous improvement both feeds and requires this virtuous cycle. This is the first advantage in terms of user-initiated innovation provided by the Interface Manager.

5.2.2 Tailoring Culture

The virtuous cycle described above lies within an individual user. How does it get boostrapped? Clearly, very few novice users come to a managed interface with the skills required to customize it, which is undesirable, since users cannot realise the virtuous cycle of investment and reward until they have enough skill to make modifications to interfaces. To be sure, the Interface Manager lowers the skill level required to change how an interface functions because it uses a combination and composition language, which has roots in real-world activities that all users understand. But, as the word "novice" implies, novice users have not the skill of expert users. No customization technique is easy enough that there is no skill barrier to using it.

This is why the "tailoring culture" described by MacLean, et al. [17] is so important: expert users provide pre-packaged customizations and assistance to novice users who share them as part of the community. Such a community came into existence in MacLean, et al.'s EuroPARC study, and occurs widely in Internet communities. For example, expert users often post Linux shell scripts to web forums as a solution to problems posed by novice users. In communities like this, novices take one of two paths: some are content only to borrow customizations made by other community members; others develop their skills through experience, increasing their expertise because even the most difficult skills to learn will be much easier to master within a community context. Thus, the most successful tailoring communities are not composed only of expert users, only an adequate supply of

experts and novices willing to learn to maintain the dynamism of the community.

5.2.3 Development Community

There is a second community invested in managed interfaces: application developers and designers. When interfaces are managed designers can design interfaces and script their functionality using commands produced by developers. Both groups are more skilled even than expert users, so the commands and interfaces they produce are of a higher quality than those produced by skilled users, even though they are likely to be less close to users' actual needs. Relations between developers and designers is often fraught with misunderstanding because they normally have disjoint skill sets. With interface design separated from applications by a command system, designers can focus on producing interfaces exactly how they wish, instead of asking developers to make changes. This also allows designers to become more directly involved with user-initiated innovation, as they will have the abilities to analyse and perfect user made changes to an interface.

5.2.4 User-Initiated Innovation

As von Hippel describes it, wind surfers jumping waves off the beaches of Hawaii discovered that commercial boards offered inadequate control when airborne [37]. Their response was to improvise straps to fix their feet to the board. This innovation spread among the wind surfers of Hawaii, with improvements to the design spreading through the community until the innovation was more or less perfected. At this point, board manufacturers incorporated the innovation into their products, mastering how to manufacture a new feature economically and reliably.

von Hippel observes, quite rightly, that *user-initiated innovation*, where manufacturers notice the innovations of users and subsequently incorporate them into later versions of their products, is very common among firms that gain their competitive advantage by innovation. It is, however, largely absent from application software and especially from user interfaces. The Interface Manager, among other benefits, changes this situation. To see how it does so, notice that user-initiated innovation has three essential requirements. First, the object of innovation, the board, must be designed and manufactured in such a way that modification is possible. Second, the skills of the users, drilling screwing and cutting, must be satisfactory for making workable modifications. Finally, there must exist a community

of users, the wind surfers of Hawaii, who use, test and improve the modifications of others.

Command line interfaces, like Linux shells, in communities like system administrators communicating through web forums, possess all these features. Not surprisingly, user-initiated innovation is common among such groups, and indeed many of their modifications and inventions have found their way into Linux distributions. As explained below, the Interface Manager opens up the application and its interface to modification by moderately-skilled users, making user-initiated innovation possible. First, however, it is necessary to refute recent claims that the open source software development process has the same effect.

Open source software development is *not* a good model of user-initiated innovation. Of the three requirements for user-initiated innovation, it satisfies one only: there is a community of skilled developers that improve open software. These developers guru programmers [], and there is little room for novices to explore and hone their skills. Experts have a body of knowledge about a huge code base, libraries, development practices and more. It takes a long time to enter into this development process. Even sharing modifications is difficult, which hinders the entry of novice users into a user-driven community. For example, a development environment needs to be set up to produce an application from modified source code, where the modifications are usually applied through patches. These are just two of the barriers presented to a novice attempting to adopt a pre-packaged modification to open source software.

Perhaps the biggest problem with cultivating user-initiated innovation in open source software is that triggering a virtuous cycle is lengthy and unlikely, for even the most determined user. Time and effort spent making changes to an open source application only contribute to a feeling of ownership when they are merged into the software. Contributions that are not accepted by the software's maintainers and incorporated into the "official" code disappear when a new version is released. Private copies of user made modifications often need to be re-written for the new version. Thus, a virtuous cycle is only produced in open source software when contributions are accepted into the software. Naturally, this limits the appeal of exploration and expertise creation that users undertake, because exploratory changes rarely persist for long. More importantly, there is no sense of ownership and the satisfaction and pride that come with it, which reduces the drive to make modifications to software. Solving this problem is the first of the improvements made by the Interface Manager.

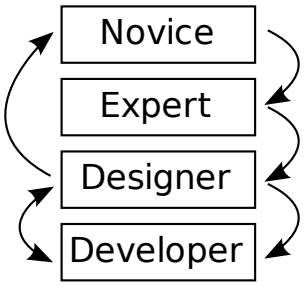


Figure 5.2: How software flows between four categories of user when software is based on the Interface Manager. First, on the left side, developers and designers collaborate to create a new application with its interface, which passes to novice users, who explore and experiment with it, until they develop expertise. Expert users then have a new channel for communicating with designers: their own designs. Finally, developers can implement new commands that have their origin in the practice of expert users.

Separating interface from application makes interfaces malleable objects that users can modify. In fact, novice users use interfaces, not software, so it is incorrect to assume that they even understand what an open source application *is*, let alone whether they know enough C to modify it. This satisfies the first require in cultivating user-initiated innovation: users can customize interfaces at run-time using the Interface Manager, exploring however they wish, knowing that their changes both produce immediate results and persist. The interfaces on their systems belong to them, so the virtuous cycle of investment into customization is possible.

Communities of users follow quickly. Co-located users, as McLean, et al. observed, form communities easily, Internet-connected users more slowly. Within them, novice users receive instruction and example interfaces from expert users and share both amongst themselves in the process, tailoring their interfaces and learning about them. While tailoring cultures have been observed for software, they are poorly supported for modern widget-based GUIs. Decoupling the interface from the application so as to make the interface a user modifiable artifact achieves this goal, and allows such a community to flourish. But, the most important benefit of these communities is the channel they create from users to developers.

New designs created by users from old ones can be incorporated into applications by designers and developers as user-initiated innovations. The process is shown in Figure 5.2. Normally, innovation in software flows downstream from developer to user. Designers and developers collaborate on an application and its interface, and then provide the software to users. Users communicate upstream only through lim-

ited options such as bug reports and focus groups. The Interface Manager allows for user-initiated, or upstream innovation to occur. Users who have invested time and effort in improving an interface can upload it to designers. They can then generalize and tune the design of the interface and improve its functionality by suggesting new commands for developers to implement. Once the developers and designers are satisfied with the improved interface and commands, the new interface is released back to the user community, satisfying needs that the community expressed by developing new versions of the previous interface. This is another virtuous cycle. Users are pleased, as the original interface was modifiable such that it was made to suit users better through customization, and they can submit new designs to the original designers in order to communicate it. The application developers are also pleased, because they receive feedback polished by users within the tailoring community and can thereby produce a better design. Because of this, the improved design created based on feedback that they release in a polished form is a product desired by users. Thus, application developers provide a product they know there is a market for, and users consume a product that has been and can further be modified to suit their needs.

There is a final requirement that the Interface Manager must meet in order to cultivate user-initiated innovation: users need to have the skills required to make customizations. With the Interface Manager, it is probable that most users have these skills, but it is not certainly so. Using a combination and composition language along with a visual interface builder should make customization as easy as possible. Certainly it is easier to customize an interface by using the Interface Manager than it is by modifying source code. However, the visual editing system built for the prototype is not adequately featured. In order to make interface customization as easy as possible, improvements should be made, as discussed in the next section.

5.3 Possible Improvements

While the prototype implementation of the Interface Manager demonstrates that implementation is possible, it is far from fully-featured. In this section, additional features required for a complete implementation are described. The first area for improvement is the visual programming system used for customizing interfaces. The second is the packaging of interfaces. The last improvement is the packaging of commands are packaged.

Finally, an improvement to the research, if not the interface, lies in the assumption that novices can and will become experts. Examining this question requires a large longitudinal experiment.

5.3.1 Implementation Improvements

The component that falls most short of full functionality is the visual editor, which needs to be extended in several directions. First, the visual editor must be carefully judged as an interface: powerful enough for expert designers, safe and easy enough for novices. Providing such a visual programming system is difficult, but the field of visual programming offers a large, well researched body of work [13].

Specifically for the visual editor, two kinds of editing capability are needed: the ability to make cosmetic modifications to the interface, such as widget layout and choice, and the ability to modify interface functionality. Modification of interface cosmetics is now routine thanks to the maturity of interface builders. Nearly all interface builders offer a direct manipulation editing system that allows designers to size and place widgets on an interface using drag-and-drop, a good way to deal with cosmetic customization. It is more difficult to customize of how an interface functions. Functionality customizations are performed on managed interfaces by modifying expressions built from commands. The current technique for doing so requires the variable widgets which allow programming by side effect. I suggest, as shown in Figure 4.11, that a dataflow visual programming language be used instead. Combination and composition are naturally expressed through dataflow, and the power of such languages can be demonstrated by their wide adoption. Pipes, for example, are very successful and are implemented in most CLIs. They are a central concept of dataflow. Commercial software, such as LabVIEW [15], use visual dataflow programming to give novice users a more intuitive method of programming for application domains in which data is successively filtered. Interface builders such as the QT Designer [35] and Interface Builder [3] allow widgets to be linked to functionality using simple dataflow semantics. Such systems can provide inspiration to future visual editor designs.

The second feature needed by a full implementation of the visual editor is a well-organized collection of default interface descriptions for each application. This is not only the interface to an application as a user first sees it, but a fallback in case detrimental customizations are performed to an interface. Furthermore, some system of version control needs to be provided. Version control can provide an

undo/redo stack, and allow branching of interfaces so that multiple versions co-exist gracefully. If small customizations need to be undone, an interface can be rolled back a little bit, but the default interface will always be available as revision zero, in case a complete rollback is required.

A third missing feature has to do with sharing of customizations. It should be possible to take a piece from one interface and place it into another. Internally, this should not be too difficult to implement: it requires that an XML node and its children be copied from one description file and added to another. Externally, such a feature can be difficult to design. An example of such a feature comes from the Façades system [33], where selection rectangles on screen space can be used to combine elements from various interfaces into one window. With Façades, these interface "slices" cannot interact, but they can with the Interface Manager. Thus, the problem is an old end-user programming problem of deciding the intention of the interface combination in order to link functionality correctly. Perhaps the solution is to make the user link functionality explicitely, but the ability to slice one interface and place its elements into another is desirable regardless.

The final improvement needed in the visual editor is a well-designed namespace for expressions and commands so that users can create and re-use functionality easier. This is analogous to naming and making executable a CLI script on a Unix system. It is essential for re-use of user created commands and sharing of customizations. Without it, users cannot easily share new commands with other users. With it, both complete interfaces and individual commands built from expressions of existing commands and can be shared throughout the community, new commands built from expressions of existing commands.

5.3.2 Longitudinal Study

Another potential problem with the Interface Manager, and also the visual editor, is its users. In the previous section, I claimed, on scanty evidence, that novice users can explore and learn until they become expert users. Is it possible for every novice user to become an expert? In order to find out how well novices learn to use such a system, a longitudinal study is required. Ideally, it would measure how a community of users evolves as skills for interface customization develop and diffuse, and as novice users become expert users. Such a long-term, difficult to manage, study is beyond the scope of this thesis, and indeed there is little consensus on how to do such a study at a reasonable cost. Such a study is a useful goal, along with an improved visual editor, for future research.

5.3.3 Security Concerns

There are certainly security concerns with a platform that allows a community to share customizations of the entirety of interfaces. However, these issues already exist within the Linux community, and some have been solved.

Some interface modifications will create dependencies for new commands: how can it be guaranteed that new commands are not malicious? Since Interface Manager commands are library functions, a solution can be offered from how Linux distributions deal with dynamic linking dependencies. For example, in Debian-based distributions, `aptitude` resolves dynamic libraries dependencies for a desired software install by installing them first. Security is performed at this step through authentication by a signing authority, normally the distribution's owner. Can unofficial functionality be made safe like this? In short, no. This is a trust problem that cannot be solved other than by using a signing authority, the solution offered above.

However, when an unsigned customization makes degrative modifications to an interface there should be some way to insure that it is not catastrophic. The power of the Interface Manager to cultivate a tailoring community is reliant on the ability to share customizations, after all, and not all users have the knowledge to sign customizations. The first step to take in securing the application of customizations is to sandbox them to a single interface. With sandboxing in place, a malicious customization can only destroy a single interface. The second step is to ensure that a version control system is being used to keep track of interface descriptions. This way an interface can be rolled back to its previous state if it gets damaged by a downloaded customization. These security measures should insure that both signed and unsigned customizations can be downloaded without requiring users to take too much risk.

5.4 Conclusion

Customization is desirable for post-deployment tailoring of interfaces. As the final phase of interface design, it allows users to adapt interfaces to their individual requirements and abilities. Clearly, the more an interface is customizable, the more perfectly it can be tailored to the user. But, although *complete customization* is clearly desirable, it has been offered to end-users by very few systems. Squeak, using Morphic, is unique in providing it, in the form of a completely customizable

widget-based GUI. However, Morphic is plagued by several problems. Difficult to use, it has been so little adopted that its potential is impossible to assess. The Interface Manager supplies equally complete customization by offering a combination and composition language for creating expressions from commands, a more accessible method than Squeak's full-featured programming language. The deployment of a *command-driven* interface style lowers the threshold for customizing functionality, which allows even novice users to use existing features as components of new ones. Furthermore, the Interface Manager uses a modern widget toolkit so that it integrates better with modern computing environments: it accepts closed and open source applications from any programming language to export commands to it. For these reasons, it is an improvement over existing completely customizable systems and a novel approach to supplying customization.

References

[1] *An Introduction to the C Shell, in Unix User's Manual, Supplementary Documents*. University of California at Berkeley, 1986. 11

[2] *The Python Language Reference Manual*. Network Theory Limited, 2003. 11

[3] Apple. Interface builder. http://developer.apple.com/tools/interfacebuilder.html, 2008. 62

[4] Joerg Beringer. Reducing expertise tension. *Commun. ACM*, 47(9):39–40, 2004. 7

[5] S. R. Bourne. An introduction to the unix shell. *Bell System Technical Journal*, 1978. 11, 20

[6] Andrea Bunt, Cristina Conati, and Joanna McGrenere. Supporting interface customization using a mixed-initiative approach. In *IUI '07: Proceedings of the 12th international conference on Intelligent user interfaces*, pages 92–101, New York, NY, USA, 2007. ACM. 8

[7] Harvey G. Cragon and W. Joe Watson. A retrospective analysis: The ti advanced scientific computer. *IEEE Computer*, 22(1):55–64, 1989. 9

[8] T. Duff. Rc – a shell for Plan 9 and Unix systems. In *UKUUG. UNIX - The Legend Evolves. Proceedings of the Summer 1990 UKUUG Conference*, pages 21–33 (of xi + 260), Buntingford, Herts, UK, 1990. UK Unix Users Group. 21

[9] The Squeak Foundation. Squeak homepage. http://www.squeak.org, 2008. 2, 15

[10] Krzysztof Gajos and Daniel S. Weld. Supple: automatically generating user interfaces. In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 93–100, New York, NY, USA, 2004. ACM. 8

[11] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1257–1266, New York, NY, USA, 2008. ACM. 1, 8

[12] Ken Getz and Mike Gilbert. *VBA Developer's Handbook*. Sybex, Indianapolis, IN, USA, 2001. 7

[13] Ephraim P. Glinert. *Visual Programming Environments: Paradigms and Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990. 62

[14] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, 1997. 13, 15

[15] Gary W. Johnson. *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*. McGraw-Hill School Education Group, 1997. 62

[16] Alan C. Kay. The early history of smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 69–95, New York, NY, USA, 1993. ACM. 15

[17] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. User-tailorable systems: pressing the issues with buttons. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182, New York, NY, USA, 1990. ACM. 9, 57

[18] John H. Maloney and Randall B. Smith. Directness and liveness in the morphic user interface construction environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28, New York, NY, USA, 1995. ACM. 2, 13, 16, 45

[19] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000. 23

[20] Brad A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *UIST '91: Proceedings of the 4th annual ACM symposium on User interface software and technology*, pages 211–220, New York, NY, USA, 1991. ACM. 28

[21] Brad A. Myers, Dario Giuse, Andrew Mickish, Brad Vander Zanden, David Kosbie, Richard McDaniel, James Landay, Matthews Golderg, and Rajan Pathasarathy. The garnet user interface development environment. In *CHI '94: Conference companion on Human factors in computing systems*, pages 457–458, New York, NY, USA, 1994. ACM. 27

[22] Bonnie A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993. 23

[23] Stanley R. Page, Todd J. Johnsgard, Uhl Albert, and C. Dennis Allen. User customization of a word processor. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 340–346, New York, NY, USA, 1996. ACM. 9

[24] Sean Parent, Mat Marcus, and Foster Brereton. *Adobe Open Source: Overview*. Adobe Software Technology Lab, http://stlab.adobe.com/group__asl__overview.html, April 2007. 25

[25] Randy Pausch, Matthew Conway, and Robert Deline. Lessons learned from suit, the simple user interface toolkit. *ACM Trans. Inf. Syst.*, 10(4):320–344, 1992. 23, 27

[26] Randy Pausch, Robert DeLine, and Matthew Conway. Suit: the simple user interface toolkit. In *CHI '92: Posters and short talks of the 1992 SIGCHI conference on Human factors in computing systems*, pages 29–29, New York, NY, USA, 1992. ACM. 23

[27] Rob Pike. Acme: a user interface for programmers. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 18–18, Berkeley, CA, USA, 1994. USENIX Association. 12

[28] Rob Pike. Plumbing and other utilities. Technical report, Murray Hill, NJ, USA, 2000. 36

[29] Chet Ramey and Brian Fox. *GNU Bash Reference Manual*. Network Theory Limited, Bristol, UK, 2006. 11

[30] Martin Reiser. *The Oberon system: user guide and programmer's manual*. ACM, New York, NY, USA, 1991. 3, 12

[31] Herbert A. Simon. *The sciences of the artificial (3rd ed.)*. MIT Press, Cambridge, MA, USA, 1996. 33

[32] Richard M Stallman. *GNU Emacs Manual*. Free Software Foundation, Boston, MA, USA, 2007. 7

[33] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. User interface façades: towards fully adaptable user interfaces. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 309–318, New York, NY, USA, 2006. ACM. 63

[34] The Glade Team. Glade user interface builder. http://glade.gnome.org, 2008. 23

[35] Trolltech. Gui builder - trolltech. http://trolltech.com/products/qt/features/tools/designer, 2008. 62

[36] David Ungar and Randall B. Smith. Self. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 9–1–9–50, New York, NY, USA, 2007. ACM. 13

[37] Eric von Hippel. *Democratizing Innovation*. The MIT Press, Cambridge, Massachusetts, 2005. 9, 32, 58