A Formalized Object Structure For Graphical User Interface Frameworks

Fredrik Öhrström
d92-foh@nada.kth.se
http://www.nada.kth.se/~d92-foh

December 7, 1998

Abstract

This thesis identifies three important structures in object oriented programs with graphical user interfaces; the compositional, the interactional and the window structure. These structures have previously not been separated, which has resulted in code and user interface flexibility problems.

By separating these structures it is possible to create a uniform hierarchy of models and their user interfaces from low level graphics to high level applications. This means that it is much easier to use composition when creating a program with a graphical user interface.

A solution to how threading should work follows from the assumption that low level graphics windows are distributed objects.

Stored messages (or method pointers) are shown to be vital for a GUI framework. The flowback, a special kind of stored message, is introduced to solve how to flow data as a result of a callback. The typical use for a flowback is in drag-and-drop protocols, but it can be used for any kind of filtering on object types.

The object structures, the distribution and the stored messages are put together in a model which makes graphical user interfaces more flexible and easier to program.

The structures and the operations used to create them are described in the formal language B.

En formaliserad objektstruktur för ramverk för grafiska användargränssnitt

Sammanfattning

Denna rapport identifierar tre viktiga strukturer i objektorienterade program som använder sig av grafiska användargränssnitt; kompositions-, interaktions- och fönsterstrukturerna. Dessa strukturer har tidigare inte separerats vilket har medfört flexibilitetsproblem både i kod och i användargränssnittsdesign.

Genom att separera dessa strukturer är det möjligt att skapa en likformig hierarki av modeller och deras användargränssnitt från lågnivågrafik till högnivåprogram. Det betyder att det är mycket enklare att använda komposition av objekt för att skapa program med grafiska användargränssnitt.

En lösning till hur trådar ska användas följer av antagandet att lågnivågrafikfönster är distribuerade objekt.

Lagrade meddelanden (eller metodpekare) visar sig vara nödvändiga för grafiska användargränssnitt. Objektet flowback, ett speciell sorts lagrat meddelande, introduceras för att lösa hur data kan flöda som ett resultat av ett skickat meddelande. En vanlig användning för en flowback är i dra-och-släpp-protokoll, men den kan användas för alla typer av filtreringar på ett objekts typ.

Objektstrukturerna, distribueringen och de lagrade meddelandena sätts samman till en programmeringsmodell som gör grafiska användargränssnitt flexiblare och lättare att programmera.

Strukturerna och operationerna för att skapa dom, beskrivs i det formella språket B.

Contents

1 Introduction

Graphical user interface frameworks have been studied intensively for many years [Palay88, Weinand89, Myers97, Fresco]. However it still takes a substantial amount of effort to build applications with graphical user interfaces. The graphical interfaces created are also very inflexible. The sheer number of different frameworks (about a hundred) listed at the "GUI Toolkit Framework Page" [Tai98] gives a hint that these problems are not satisfactorily solved yet.

Many of the current frameworks use the name application framework instead of a GUI framework. The reason for this is that their framework design affects all of a program that uses it. Unfortunately many of the frameworks assume that every program is a graphical program and as such force every program to contain many GUI primitives. However many programs that do not have a GUI can still benefit from features that have come from GUI research, like resource management (e.g. fonts, colors) and constraints (e.g. how graphical objects are entangled), but do not need the actual GUI.

In this thesis I will discuss the unclear responsibilities faced by the programmer, the interface designer and the user (section 1.2); what I consider important for truly flexible frameworks (section 1.3); problems in current frameworks that are not commonly discussed (section 2); my goals for a new framework and the rationale behind the new object structures (section 3).

In section 4, I will describe an object model that can be used as a design specification for an application framework. This model solves the problems described in previous sections of the thesis. A program that is structured according to this model, will be easier to program and more flexible for the user. The actual design of a good GUI will be a difficult task for a foreseeable future [Myers93]. The object model is formally defined in section 5.

Some of the principles described in this thesis are general and applicable to other areas of object orientation too. The object model is partly a collection of classes, but more important is the strategy behind the creation of new classes and how they should be connected.

1.1 Aspects of GUI frameworks

There is much work being done on GUI frameworks. The aspects that attract most attention are:

- *User interface design*
  Interface design encompasses among many things: ease of use, appearance and layout, flexibility. Unfortunately these properties are often reduced to the latest set of features, like window tabs and balloon help. Computers today are so powerful that almost any 2D GUI could be implemented. Our biggest obstacle is our imagination. People are already looking at 3D user interfaces, but we have still not exhausted the possibilities of the 2D GUI.
- *Interface implementation*
  Ease of programming is a goal that almost every framework strives to attain. Portability is also common. Though creating a portable framework is an interesting goal since the graphical environment varies enormously between different computer systems. It is like building a train that can run on any track width and on ordinary roads too!
- *Problem specific aspects*
  Specific problems can require special support. CAD programs require 3D graphics like OpenGL. Desktop publishing requires support for paper simulation display like Display Postscript, scientific visualization requires support for voxels and graphs. None of these is really necessary for a GUI.

This thesis focuses on the interface implementation. As result of the improved implementation it is possible to create better user interface designs.

1.2 Relation between the programmer, the user interface designer and the user

One of the reasons that it is difficult to program a GUI is that the responsibilities of the programmer are not made clear. Today, two tasks are usually performed by the programmer; implementing the program that performs the actual task and the design of the user interface.

A program is used to solve a certain problem, like edit a text, calculate a value, ask a question. A programmer who is focused on this task only wants objects that gives easy access to the user. The actual design and layout of the windows is not needed. The programmer needs input and output objects. These should be grouped together according to function — not style. When this is done we can let somebody else worry about how these objects should appear on screen. Not everything can be left to the designer though. The foundation for a good user interface is the functional grouping of interaction objects and this must be laid out when the actual programming is done.

The programmer writes the code which will incorporate different sorts of user interaction components. When the program is run for the first time, the computer tries to do a layout of these components. This will most likely look ugly, therefore the designer can change the appearance of the layout while running the program. These adjustments are stored in the resources to be supplied as a default design when the user uses the program for the first time. There is absolutely no reason to lock the design when the designer is finished, this means that the user can be a designer as well. If the designer knows the problem well, it is also possible to let the designer create both the graphical design and the interaction structure. Code can then be generated from the interaction structure and be handed to a programmer.

1.3 Flexible GUI frameworks

The following examples show what truly flexible GUI frameworks should support. These are difficult or impossible to implement in current frameworks.

- *Choice of output design*
  The user should be able to change the appearance of the application. Details like colors, fonts should be changeable as well as the layout. Many of the current frameworks support setting colors, fonts and images, but very few support major layout changes like figures 1.1 and 1.2 show. With a latin alphabet it is natural to read from top left to bottom right. The OK button is placed in the bottom left corner so the eye can easily stop its movement over the most common choice. However if we read from the top right to the bottom left as with kanji, then the OK button should end up in the top left corner.

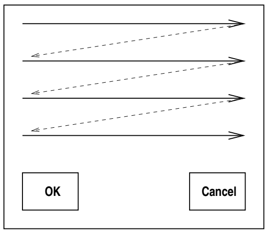

Figure 1.1: Latin layout.

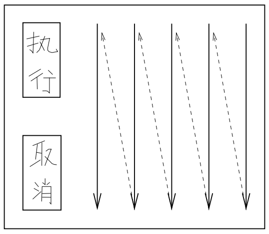

Figure 1.2: Kanji layout.

The user interface could also be changed by splitting one window into several windows or merging windows into fewer windows. Figures 1.3 and 1.4 show two possible appearances of a selection box.

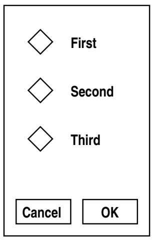

Figure 1.3: A standard layout.

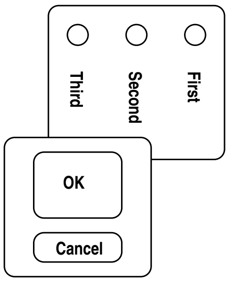

Figure 1.4: An unusual layout.

- *Choice of input design*

  As well as the user can decide how applications should present themselves, the user should also be able to decide how to control these applications. One user might have a four button mouse with a wheel, another user might have a mouse with only one button. Still the single button mouse can do the same work, together with keyboard modifier keys, as the multi button mouse. The user can bind physical actions to meaning based actions globally so input is consistent over all applications. One user might want to double click the left mouse button to activate a GUI button. While another user might want to use a single click with the right mouse button to do the same thing.

  In applications where the user can choose between many different commands, like CAD programs and word processors, the trend is to allow the user to compose the menus or tool boxes. The menu structure is a way for a user to easily find a command. The power user might want to create a special menu structure or a tool box that contains the most used commands as buttons. There is a clear trend in application design to allow the user to do this kind of changes to the application interface. Button bars can be moved or iconized within the application window. Drop down menus can be torn off and positioned anywhere on screen. To take this trend all the way, we make all of the menus and buttons user configurable. The application can present a list of all possible actions. The user drags one of the actions to a newly added button or menu element as in figure 1.5.

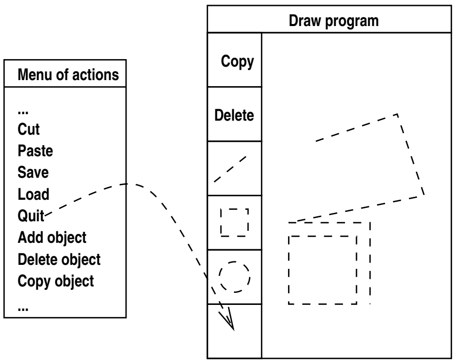

Figure 1.5: Adding the *quit* action to a new button.

In many applications the same buttons appear over and over again. Print, cut, copy and paste buttons are duplicated and take up valuable screen estate. These buttons could be grouped together and used by all applications, see figure 1.6.

One could also have small applications, called tools, work on other applications. Tools can work on objects with standardized interfaces. Such a tool could be used to change the layout of a program. Instead of telling the program to switch into a layout edit mode we can use a specific tool that can modify the appearance of a program while it is running. This way we do not risk to accidentally change the appearance.

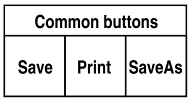

Figure 1.6: Buttons common for almost all applications.

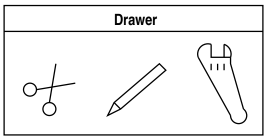

Figure 1.7: Tools usable in many applications.

- *Choice of functionality*

  To offer choice of functionality we must support replacement of objects. The layout and style of a particular component can be changed but the same mechanism cannot be used for change of functionality. If a user dislikes the file selection box where the file name must be typed in (figure 1.8) then it should be possible to replace the file selection box with another implementation. A box with an improved functionality is shown in figure 1.9.

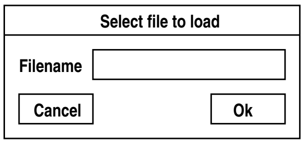

Figure 1.8: A simple file selection box.

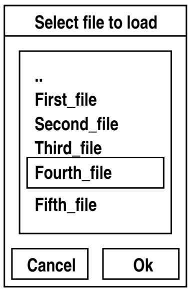

Figure 1.9: A more advanced file selection box.

2 Problems with current frameworks

2.1 Problems with model-view-controller frameworks

The model-view-controller (MVC) framework enforces a semantic division of the code into a model part, a visualization part, and a controller part. The MVC is popular in frameworks which are document centered since it is easy to map these parts onto a document and its viewer. The MVC model originally appeared in Smalltalk.

The MVC concept has recognized a common problem when dealing with GUI programming; the model [1] is often heavily affected by the GUI. A programmer can only use object oriented programming fully if there are no restrictions on how the objects are designed and used. If it is not possible to implement the object model exactly as designed (because of GUI restrictions) the program will suffer in flexibility, performance and readability.

By separating the model from the view, the MVC tries to rid the model of influences from the GUI. Unfortunately this is only useful when the model is such that it can be totally abstracted from the view. This is not the case in most of GUI programming, often the model is tightly coupled to the view. Also a view is a model in itself which has a view and so on. A file selection box has a model (which file is selected) and a view consisting of several smaller models (with their respective views) for buttons and text. The Smalltalk MVC supports sub canvases to create views within a view, but it does not extend this notion to create a complete structure to handle hierarchical models and views.

The separation of code into a model, a view and a controller object is often not needed for normal user interface programming. In most cases there is only one view and one controller for each model. This redundancy has been noted and in some systems the controller has been eliminated altogether to create the document and view framework like in OpenDoc.

2.2 Mistake in evolution for widget based frameworks

When the object oriented methods were introduced people started to convert old frameworks into object oriented ones. Unfortunately since this transition was stretched out over time, and people did not a have a clear perception of how object orientation should work, the transformation of existing frameworks into object oriented frameworks was sidetracked.

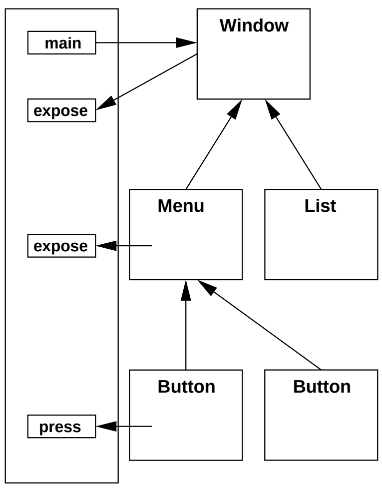

Figure 2.1: Non object oriented design.

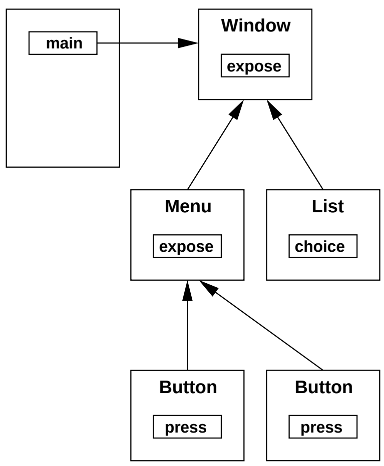

Figure 2.2: Erroneous object oriented design.

[1] In this thesis the words model, composition and refinement are used to describe object oriented concepts, not formal methods ones.

Figure 2.1 shows a typical non object oriented widget based program. There is one object, the whole program, that controls a tree of widgets. The widgets have callbacks implemented as function pointers to global functions. This is how Motif and others have implemented the user interface framework.

When old widget based frameworks were modeled with objects it was assumed that the callback functions should be moved into the widgets. This can be seen in figure 2.2. The widgets are subclassed from classes like window and button. To catch an event you override a member function like *expose* or *buttonPress*. This is exactly how it is done in Java 1.0 and in many other frameworks. However this raises several problems.

- *Inheritance where it should be composition.*

  You have to subclass a button every time you need a new behavior. If this was true in real life, the hardware store would be full of different kinds of light switches, at least one kind for every room ever built! Fortunately this is not the case, we have one kind of switch that can be used for different purposes and this is how it should be in the framework. Although the appearance of the switches might differ, flashy design for the dining room and rough design for the garage, the switches are all interchangeable.

- *The event handler location is fixed.*

  As most people realized, it was impossible to create a new kind of button for each use. A partial solution was to let the events flow up through the graphical tree and be taken care of at higher levels. This meant that instead of creating one new button for each use, only one new window was needed. The window is subclassed and your new class overrides the *handleEvent* member function. This means that the window takes care of all button events with a switch statement, but if you have a switch statement in an object oriented program, something is probably wrong with the object model. Event flow in a graphical tree is only a patch to the previous problem and it also creates the next problem. Note that sometimes it is correct to let the events flow upwards in the graphical tree. This is when a contained window is not interested in the information. E.g. in X-windows we can send a kill message to a window. Usually only the largest containing window owned by the application is interested, so it makes sense to pass this on upwards until it reaches a window that has registered an interest. When a mouse button is pressed the user is sending a message to the window in which the cursor is. Since the message is for the window it should not be passed up but instead handled directly by someone listening to that specific window.

- *No freedom in layout of graphics tree.*

  Since the event handler location is fixed inside a window, the tree topology is more or less permanent. It is difficult to split a window into four since then it would need three new event handlers.

2.3 GUIs are not documents

The problem to lay out buttons and windows on the screen is similar to lay out text and images in a document. That is the reason why some frameworks have taken text layout algorithms as the foundation for the GUI. Fresco [Fresco] does this. Everything on screen are glyphs that are organized with a typesetting algorithm similar to TEX. This makes it extremely easy to write a document viewer since the framework already does much of the work.

Unfortunately this limits how much the interface can be rebuilt. It is possible by changing the layout engine to achieve different results like the ones in figure 1.1 and 1.2. But a standard document layout mechanism is not enough for the complex layout changes in figure 1.3 and 1.4, where the window structure is different.

Similarly some GUIs like NextStep use a display language that is focused on paper output. However, the screen does not have the same properties as paper. Paper is static, the computer display is dynamic. Paper output needs high resolution fonts, GUI output needs animation and 3D graphics. Of course all needs can be incorporated in the same base architecture, but that would bloat the base enormously. Instead paper display and 3D graphics should be handled by separate modules. Interaction components in the base architecture can use these modules, but should not be locked to any of them.

2.4 The multiple event handler problem

Almost every framework that responds to external events requires that the main thread is handed over to the framework to execute an event handler. If there was only one framework in the universe this would not be a problem, but as soon as we want to use two frameworks at the same time we get into trouble. A common example today is to use distributed objects with a graphical user interface. Workarounds are usually quickly created but it would be superior if this problem had been addressed from the beginning so new event handlers can be added with no modification of the system.

One solution is to use threads. This works in systems designed with threads from the beginning, like Java. In other systems that has not been designed for threads, using threads to run separate event handlers can be a serious hassle.

3 Goals and rationale for new design

Existence makes something useful
but nonexistence makes it work
— Lao Tzu 300 B.C.

The house is useful because it has walls, but the windows and the doors make it work. An object oriented framework is useful because it defines an architecture for your program, but it is the empty space that can be filled with new components that makes it work.

3.1 The framework goals

I consider the following properties to be significant for the implementation to make a framework flexible and easy to program:

- *Programming by composition*

  In normal object oriented programming the standard behavior is to create more complex objects by composition of simpler objects. This should apply to user interaction objects too. For most GUI frameworks today it is not possible to use composition in this way. Programming by composition also implies that there should be an easy way to bind actions and data flow between objects, especially from the components to the composites.

- *Code locality*

  Parts of code that work cooperatively for the same functionality should be located close together. If the code is spread out we will spend more time writing and maintaining it.

- *Resource management*

  There should exist a good way to store resource information. The naming standard should be logical and make it easy to locate resources.

- *Low level routines*

  The programmer should always be able to use low level routines. Usually this is difficult because it is considered a violation of the high level interface of the framework. However if the low level access is modeled with objects then the programmer can use these objects in the same way as the framework uses them. This also means that it will be much easier to write your own GUI objects since you do not have to learn a raw non-object oriented graphics interface.

- *Distributed graphical embedding*

  One of the currently most active areas of research and development is distributed objects. If distributed objects use graphics then we have no choice but to solve the problem of distributed graphics. This can be done in many different ways. One system that has done distributed graphics for a long time is the X-window system.

- *Thread support*

  Many programs can benefit from thread support, especially graphical user interfaces. The typical example is when the web browser freezes the display because it is waiting for a slow site. This can be solved by clever programming in a standard framework but it would be much better if the framework supported an easy and standardized way to solve it.

- *Reflecting model changes and state in the user interface*

  Reflecting model changes is a very broad concept. Essentially it is what a GUI is all about. A typical example is greying out buttons to let the user know that the alternative is not available. The connection between a model state and the view can be arbitrarily complex. A simple example is the typical "save" menu choice that is enabled whenever some change is made and disabled after the document is saved. A more complex example is a supportive stock trading system for unit trust brokers. The investors can state in what kind of stocks they want the trust to invest their money. This gives an enormous rule base which states how many percent of the mutual funds money are allowed to be invested in different stocks. Of course, this is impossible to handle manually. We want the trader's GUI to reflect the state of the rule base and signal by greying out the "buy" button that this stock is not allowed.

This connection between different objects and their states can be obtained with constraint programming. Constraint programming actually evolved as a solution to typical GUI problems like drawing and GUI component relations [Myers97].

3.2 Distributed objects and GUI frameworks

Object technology is one of the best ways to model and construct graphical user interfaces. Mostly because it usually is easy to map objects on the screen to objects in the code. Object technology promised the programmers that it should be possible to build new systems with prefabricated objects that snap into place.

The distributed object model is essentially the meeting point between two technologies, client-server and object orientation. There are several problems inherent with distributed objects. You cannot for example split any object oriented program into two with no side effects. Splitting a system can mean quick exhaustion of system resources. Recursive calls within the same address space use only the stack. The stack can usually cope with several thousand calls. However if the recursion takes place between two separated objects the limit is probably the number of waiting threads which is much smaller.

Why are distributed objects interesting in a GUI? Replaceability can be achieved without it, adaptability also. However if we add support for distributed objects in the GUI we get replaceability for free and we can run different parts of the program on different machines (or address spaces on the same machine). We can even run a single button on another machine. Fresco can do this [Fresco].

Is this really useful? Let us look at an example: Previously an interface for the results of a super computer application was a typical client-server solution. A special purpose protocol transferred the images from the server to the client. The next step is to use a standardized protocol like Corba to exchange this information or to let a protocol like X-windows display the whole result from the server. The final step would be to allow a distributed graphical component be exported from the supercomputer that can be used within the client program. For this, we need distributed graphical objects.

Also for documents within documents we need distributed graphical embedded objects. OpenDoc defines a way to do this. However OpenDoc and similar designs focus on high level objects. This thesis focuses on a low level design that can be used for distributed graphical embedding.

3.3 X-windows and the window structure

X windows was designed at MIT to be used for distributed graphics in a network. It is operating system independent but is commonly used with Unix-like operating systems. An X-windows program is called the X-client and connects to a graphics server, the X-server. The server might be located on the same machine but can be any server running on the Internet. The naming of the client-server can be confusing because most people consider the terminal as the client that connects to a mainframe that can display its output on the terminal. This is not the case with X-windows where the X-terminal runs the graphic server software and the application programs connect to the X-terminal server software. This makes the system much more flexible since every program can display on different displays. In fact a single program can display output on several different displays and several programs can display on the same display.

X uses uncompressed packages that can be binary loaded into structures and used with little or no translation. This means that it is fast over high bandwidth links. X does not use confirmations when actions like drawing in windows or creating new windows are performed. They are assumed to work. This means that there are very few round trip times to be accounted for and that a whole series of requests can be buffered. This is an extremely important design consideration since bandwidth usually can be increased either with improved hardware or improved compression but it is very difficult to improve round trip times. The round trip time essentially boils down to the distance between the two communicators.

The graphics model used by the X-window system is a little bit different from other display systems. It is a mix between retained mode and immediate mode graphics. The X-server maintains a tree with possibly overlapping areas that are used for input and output. Such areas are called windows and can have any shape. Windows are used by the client applications to display graphics. The client application requests windows to be created. The server then manages their position on the screen and which background color (or image the window has). To display more graphics, the client must actively draw raw graphics (like pixels, lines, image blocks) in its window. It can do that at any time, but it is usually done only when the server tells the application that the window needs a redraw. For animations the client will redraw the window whenever it seems fit.

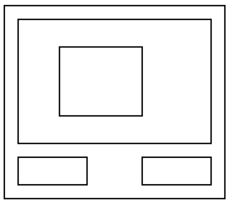

Figure 3.1: A window with subwindows.

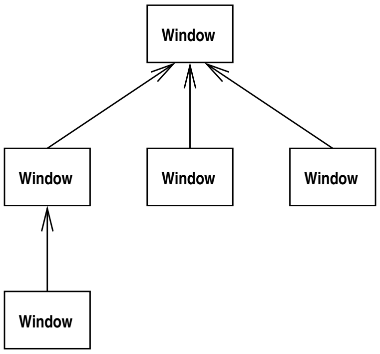

Figure 3.2: The corresponding window structure.

Since X does not enforce anything above absolutely raw distributed graphics access there are myriads of different incompatible frameworks running on top of X. This has given X-windows a bad reputation. People who believe X-windows is bad usually mistake the obvious confusion at the top level with the low level access protocol. The X-window protocol is sound and extensible. It is more than ten years old and new extensions have taken care of technology advances. Any new distributed graphics protocol is likely to reinvent X-windows.

What X-windows needs is a distributed object model that can merge X-windows with objects. This model could for example be used for drag and drop. The dropped object could be a standard distributed object. The model should make use of the fact that windows, pixmaps, and graphical contexts in X-windows are in fact distributed objects, instead of building yet another way of distributing graphics on top of X-windows.

3.4 Object composition and the compositional structure

One important structure of an object oriented program is the tree of objects within objects, built with composition. This structure is often ignored. It can be visualized as in figure 3.3. Composition diamonds from the unified modeling language (UML) are used to show the composition relationship. It is important to note that this diagram, and all of the following, visualize instances of classes and how they are related during runtime, not static class relationships.

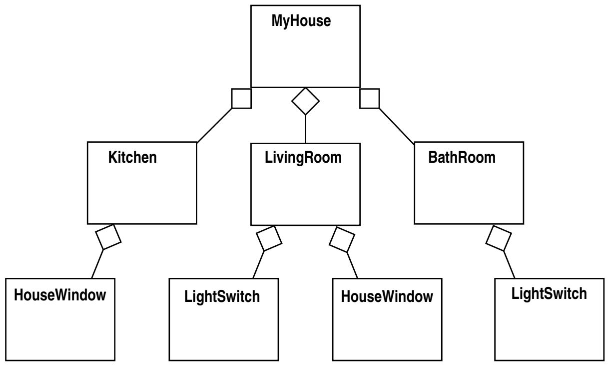

Figure 3.3: The composition of instances in a running program.

The compositional structure is highly dependent on the problem and how the programmer wants to solve it. It is essentially the model of the problem. This means that to solve any problem with an object oriented design we need to be able to design the structure exactly as we want to. Unfortunately almost all application and GUI frameworks prevents this by forcing their own structure on the compositional structure.

The solution that has been used when the problem is too complex to be fitted into the compositional structure of the GUI is to divide the system into a model and a view. This has led people to believe that GUI structure is incompatible with normal object oriented programs and therefore we have to separate them as described in section 2.1. This leads to severe problems when we create new GUI objects since a GUI object is a small program in itself and as such also has a compositional structure. There should be no difference between designing a new program and designing a new GUI object. This also makes sense since we are more and more dealing with distributed graphical embedding where the programs actually are objects which can be used as interaction components.

It is difficult to give an exact specification of the compositional structure. It is not static since elements can be added to composites during runtime. It is more how the programmer visualizes the problem in code. A good rule of thumb is that if an object cannot exist (or have no meaning) without its composite object then it is probably a parent/child relationship in the compositional structure. This rules out graphics to be part of the compositional structure since a graphical window can have a meaning without its parent window and reparenting of windows is common in the X-window system.

3.5 Interaction components and the interactional group structure

An interaction component is an object that can be used for interaction with the user. The purpose of an interaction component is to abstract the physical devices that are used by the user into high level descriptions of interaction. E.g. a button (which is an interaction component) could send a message that it is activated, not that a physical mouse button has been pressed. A simple acknowledge component reports if the user acknowledged or refuted something, not that the user pressed button one or two. An interaction component should not have methods in its interface that can be used to control the appearance of the component. Of course if the purpose of the component is to do a graphical decision, like picking a color, then the corresponding methods must exist, but otherwise the programmer should be oblivious to how exactly the component will appear. We want to base the interface used by the programmer on meaning, not physical action or graphical appearance.

The programmer needs to group interaction components together. An interaction group contains those interaction components that make sense when shown together on a certain scale. E.g. a small group contains a label and an input field as in figure 3.4. These could be placed side by side by the layout manager. Figure 3.5 shows a slightly larger group which contains several groups with labels and fields. In figure 3.6 we finally have a group with this group and a group of buttons.

Figure 3.4: A small group.

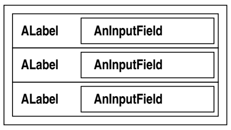

Figure 3.5: A group of groups.

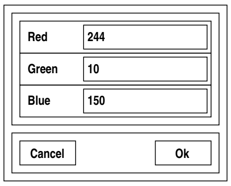

Figure 3.6: The previous group with a button group.

The tree of groups is shown in figure 3.7. Figure 3.6 shows just one way to do the layout of the interaction components. If the user is located in China, and with the appropriate font, the layout shown in figure 3.8 might be better.

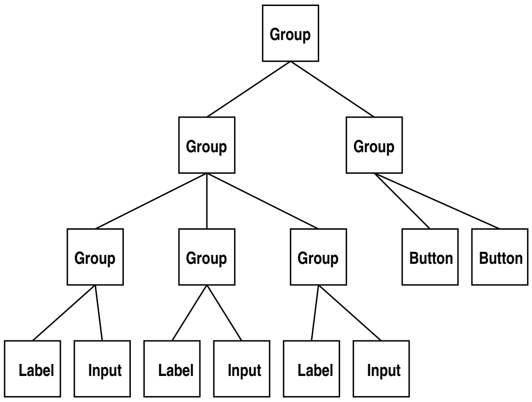

Figure 3.7: The tree structure of the groups.

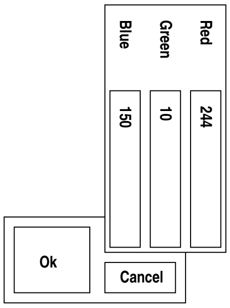

Figure 3.8: A top to bottom and right to left layout.

The actual tree of windows created for figure 3.6 looks like figure 3.9. In this case there is a one to one mapping from the group tree to the window tree. In the case of the layout in figure 3.8 we can see that the window structure in figure 3.10 is not particularly similar to the interactional structure.

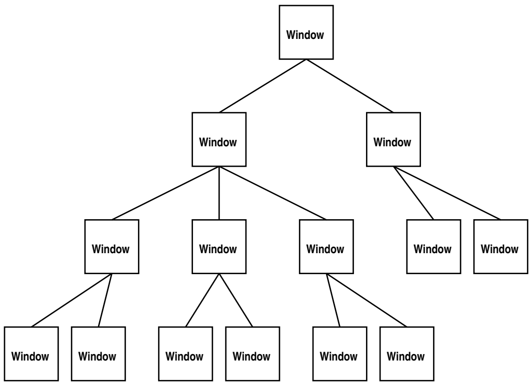

Figure 3.9: Realization of first design with windows.

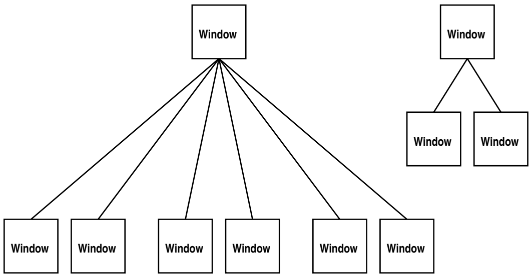

Figure 3.10: Realization of second design with windows.

This means that the interactional structure is not equivalent to the window structure. The interactional structure is not equivalent to the compositional structure either. Composition is used to implement an object with the help of simpler objects that solve subproblems. Assuming that the compositional structure is equal to the interactional structure is like assuming that every subproblem has an interface that should be integrated with the interface of the main problem. This is clearly not the case of popup dialogues and many other interaction components. Figure 3.11 shows a component that is used to display and change a color. The chosen color is displayed in the middle area. When the edit button is pressed the dialogue window in figure 3.6 will pop up. The compositional structure of this component is shown in figure 3.12.

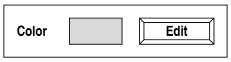

Figure 3.11: A component to display and edit a color.

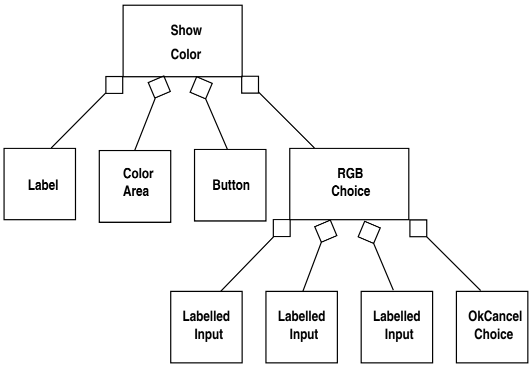

Figure 3.12: Parts of the compositional structure of the color component.

There will exist two separate interactional group trees inside the *Show Color* component. One group contains the three components *Label*, *Color Area*, and *Button* and is shown in figure 3.13. The other tree is the group tree used inside the *RGB Choice* component shown in figure 3.6.

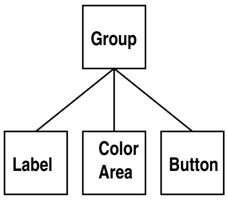

Figure 3.13: The group tree of the *Show Color* component.

We use components inside the new composite *Show Color* to solve subproblems. In this case one of the subproblems is to choose the values describing the color. If the interactional structure was the same as the compositional structure we would not be able to encapsulate how the subproblem is solved graphically. The *RGB Choice* user interface would interfere with the *Show Color* user interface.

It is now clear that there are three important structures in a program with a GUI; the compositional structure, the interactional structure, and the window structure. The question that needs to be answered is how should a program be organized so these structures do not interfere with each other? The next section discusses the solution to this problem.

4 Design of the CIW model

The CIW model defines three major structures in a program with a GUI: compositional, interactional and window structure. In short the CIW model. The whole model consists of the following parts:

- *Binding of objects*
  How to bind actions and data flow between objects.
- *Structuring of objects*
  How the program should be structured internally.
- *Scheduling of objects*
  Which threads run in which objects.
- *Production of objects*
  How to produce the right objects.

4.1 Binding of objects: Flowbacks and Callbacks

Normally control of information flow between objects is handled by calling member functions directly. If a library object wants to call your own code the library object has to know what member functions your object supports. In a totally dynamic system like Smalltalk this is verified at run-time. In more or less typed systems this has to be declared beforehand. This is done by inheritance of interfaces. If an object inherits an interface, either a special object as in Java or just a class with no implementation as in C++, it is a promise that your object implements the methods specified in the interface.

Interfaces are for example used if you want your object to be integrated in a finished framework. Since your new object inherits the framework object interface it can be used by the framework as a built-in object. This is the core of object oriented programming. Composition is then used to create more complex objects by using simpler objects together inside a new object.

However normal method calls and inheritance are not sufficient for a GUI framework. Assume that we have a window with several buttons. With the interface callback solution the buttons can only call the interface method *buttonPress* on the model object when they are pressed. Since the buttons are likely to modify the same model, they will all call the same method on the same object. If the *buttonPress* method does not take any arguments then we cannot distinguish the buttons! This is often worked around by supplying the identity of the button as an argument to the *buttonPress* method. Unfortunately this results in a switch statement in the code. Often the switch statement only calls different functions depending on which button has been pressed. This code is completely unnecessary for the programmer to write since it could just as well be handled by the framework if only it would be possible to hand the framework a pointer to a method instead of a pointer to an object implementing a certain interface.

Switch statements are often bad news for other reasons in object oriented programs. If the code switches on the type of object, then it probably is a bad object model since this should be solved with interface and letting the language call the right method from the start. Another good indication that something is wrong in an object model is if the methods cannot be named after their purpose but only after the action that triggers them. Java 1.0 abstract widget toolkit is the typical example, e.g. we have to name our methods *buttonPressed* instead of *quitApplication*. It is much better to name methods after their purpose instead of after how the are called. Also in many frameworks it is possible to reject an event. The reason for this is that in some frameworks the events flow in the graphical tree. It is described in section 2.2 why this is not a good idea. It is much better to make sure the right method is called from the beginning.

So we need to directly call the right method from the GUI framework. What we need to do is to store the message that is going to be sent to the object. In Smalltalk and Objective-C any message can be stored for future use. In more typed languages like Java and C++ we are not actually sending messages but calling methods. Storing a reference to a method for later use can be done with a method (function) pointer. If you hand a method pointer to a button it can call the right method directly when pressed. This is how callbacks from GUI objects are done in almost all non object oriented widget based GUI frameworks.

Why are not method pointers used in typed object oriented frameworks? Mostly because of the lack of support. In an object oriented language we want to call class methods, not global functions. This means that the method pointer must track which object to call as well as which method. C++ method pointers are very difficult to use. Java only supports arbitrary messaging as part of the reflect package and it is not intended to be used for everyday programming needs. Smalltalk does not need method pointers since it can send arbitrary messages to any object, which is the same as a method call. Objective-C which was inspired by Smalltalk also avoids this problem by using true messaging. The NextStep designers have noted that if Objective-C could not send arbitrary messages [ObjectiveC] then they would have the problems just described.

Unfortunately pointers to methods or stored messages have other problems connected with them. A method call signals that an event has happened, but events also have information like how hard a button was pressed or how fast the mouse was moved. This information is easiest to supply as an argument to the method, but if we break up all data into several arguments to the same method call, we get a very inflexible system. If more information must be supplied with the event method call, then we need to change the program in every place where the event has been used before. This can be solved by supplying a single object as argument which contains all data about the event, then this object can be extended without affecting old code (apart from that it might need to be recompiled).

In a distributed environment pointers cannot be distributed between address spaces so we need to wrap them inside an object. The command pattern [Gamma95] describes an object that knows how to perform an action when its *invoke* method is called. We can use a command object that is both a storage space for the event information and the wrapper around the method pointer. The method called needs only to take the command object as an argument. The command object has been used in MacApp, ET++ [Weinand89] and in Fresco [Fresco] to bind actions from GUI components to code. There are other ways to store a message in strongly typed languages. One way is to use signals and slots which have been used in Qt [Qt]. A message is described with very strictly defined signal and slot methods. A message is stored by binding one or several slots to a signal. When the signal is activated, all the slots will be activated too. This solution works but is not as flexible as arbitrary messaging.

The command object is perfect for action events where something happens and we want to transfer information about how it happened. Sometimes however, the action results in a transfer of data that is totally disconnected from the action itself. Drag and drop is a good example. The actual action of dropping an object can be described within a command object, like where and how the drop was carried out, but how should we supply the object that is actually being dropped?

We can view the action of storing a message for future use as writing a letter. Assume that you want to be notified by another person when something happens. Get an envelope with a stamp. Write your address on it and put a form inside. The form can have any fields and you can fill in some of them. Then hand the envelope with the form to the other person with instructions on when to notify you. When it is time to notify you, the person will fill in the rest of the form and post the envelope to you. If the notification action also requires an extra object to be handed to you, a separate note can be put in the envelope that tells you where to pick up the object.

This analogy is transformed into a program construct with the help of the command pattern. The command pattern is used to create a new kind of stored message object, the flowback. The flowback is the envelope with the address and the form to be filled in. A method that is going to be called through a flowback must take two arguments. One of the arguments is the same as described above, the object with all of the information about the event. The other is the possible object reference which might be sent as a result of the event.

The participants in a flowback interaction are:

- *Recipient*

  The recipient is the object whose method will be called by the flowback. In the case of a composite that requests notification from its components, the recipient is also the creator. The method in the recipient that is going to be called takes two arguments. An example signature of such a method is: `void gotString(String*, DropFlowback*);`.

- *Creator*

  The creator creates the flowback and hands it to the caller. Typical creators are constructors in composite objects that create components and bind the components together or to the composite itself. The command `flow(this,gotString)` will create a flowback addressed to `this` object and to the `gotString` method. Which type of flowback (that is which form to put in the envelope) and which type that is allowed to flow, is implicitly defined from the types accepted by the method in the recipient.

- *Caller*

  The caller fills in data in the flowback and invokes it. Typical callers are all interaction components that can notify other components, with buttons as the most common example. The flowback is filled in with `flowback->setDropPosition(10,20);` and it is invoked with `flowback->invoke(aString);`.

Creating a new flowback object for each event can induce a substantial overhead. So a flowback is only created once, when the binding is established. The invoker then prepares the flowback by filling in the correct data like drop position and time. When that is complete, the invoker will call the *invoke* method. For simple flowbacks with no event information, the call will be as fast as a couple of normal method calls. No objects need to be created. However this imposes some restrictions on how to use flowbacks with threads, which will be discussed later.

4.1.1 Flowback definition

A flowback object is used to bind action between objects, not to perform any action itself. The flowback has the following properties:

- It contains information about the action that resulted in the activation of the flowback.
- It knows what type of data it can send and when invoked, data of this type must be supplied to the flowback.
- It knows which method and object to send itself and the data to.

4.1.2 Example of use

Assume that we want to accept every string object dropped onto a window. In the component code we give the window a newly created flowback. The template function *flow* creates an appropriate flowback that is filled in with the relevant information about which object (*this*) and method (*gotString*) is going to be called. Also it extracts which type of class this flowback should be interested in by looking at the first argument of the method and which kind of flowback to create from the second argument. The flowback is stored in the window variable *on_drop* by calling the method *onDrop*. The window, or any other client that can extract the *on_drop* variable from the window, can then activate the flowback by executing `on_drop->invoke(aString)`. The figure 4.1 shows how the objects and methods are referenced.

```
...
window->onDrop (flow (this, gotString));
...
void gotString (String *s, DropFlowback *dfb)
{
  cout << "Got string " << *s << endl;
  cout << "Dropped at " << dfb->drop_location << endl;
}
```

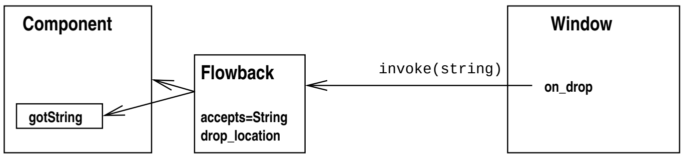

Figure 4.1: A flowback example.

The callback is just a simpler case of a flowback. A callback does not have any data flowing as the result of the action, the only information that is sent is the data about the event itself. This means that the argument to *invoke* is ignored. A convenience *invoke* method is supplied in the callback class that takes no argument. The figure 4.2 shows the callback situation.

```
...
window->onButtonPress (call (this, sayHi));
...
void sayHi (XCallback *cb)
{
  cout << "Hello World!" << endl;
}
```

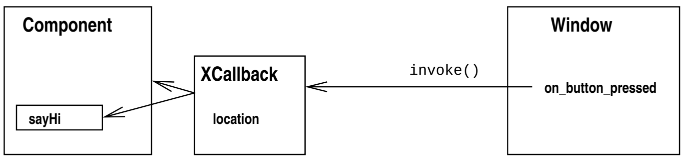

Figure 4.2: A callback example.

Several drop flowbacks can be supplied to a window. In this case the flowback that produces the best match according to the inheritance hierarchy is called. For example if a string or color object in figure 4.3 is dropped then that specific flowback is used. Any other object dropped will call the last flowback if *Object* is the parent to all objects.

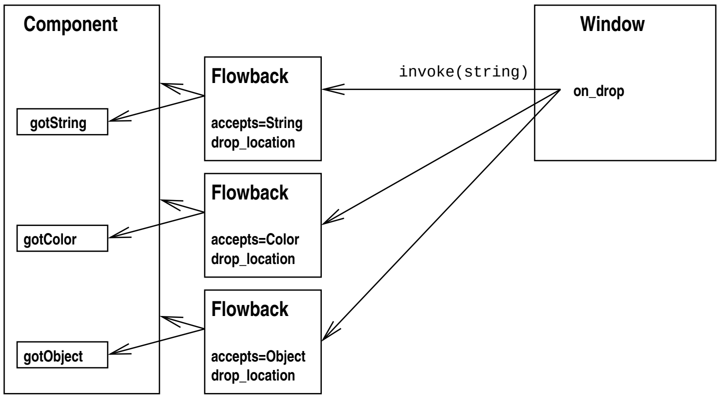

Figure 4.3: Accepting several types of data.

Distributed objects also give us a very simple way to actually transfer the data dropped between applications, i.e. we don't! The object handed to the flowback is the same object created by the dropper code. If the dropper code runs in a different address space, then the object received by the flowback is a proxy to the real object. Also since flowbacks are distributable objects, objects running in different address spaces can also use flowbacks to bind actions and data flow between themselves. As a matter of fact, the windows in an CIW modeled program are distributed objects, so in the CIW model we are always dealing with distributed flowbacks.

Distributed callbacks can also be used to bind actions to new buttons. Simply create a new button in a tool bar. Then find the action in a list. Drag the action and drop it on the button. Now when the button is pressed the action is performed. What happens is that the button receives a flowback with a callback as flowing data! When the button later is pressed it will invoke the callback.

4.1.3 New flowbacks are derived by inheritance

We can derive new flowbacks with more specialized purposes from the flowback superclass. For example, the drop flowback has methods to set the drop location and time, the button press callback can set which button was pressed and where, see figure 4.4. If the client object needs more information stored in the flowback, it can subclass for example the drop flowback into a *MyDropFlowback* with client specific methods and data. When the client object creates the new type of flowback it can set these client specific values and then hand it over to the caller. The caller is satisfied as long as the flowback fulfills the correct interface.

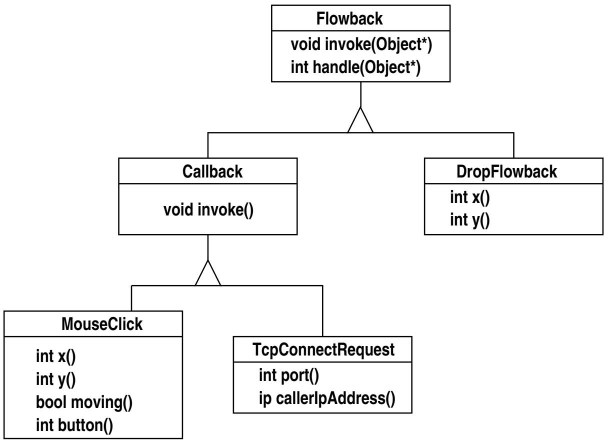

Figure 4.4: Inheritance tree of typical flowbacks.

At higher levels in a GUI it is very convenient to have more complex command objects which have a revoke method. These should not be inherited from the flowback since they actually perform work apart from just binding objects together. We can however use flowbacks to invoke these command objects or revoke them.

4.1.4 Flowbacks used for constraints

A GUI framework inevitably introduces the problems of real time update of values. Also a GUI must reflect changes in objects that change asynchronously. The problem with these systems is that we cannot create a single loop at the end of which, we can verify all values and update the interface accordingly. We need a tool to gather changes from separate objects, analyze these changes and present them to a final object. Such a tool is needed for any multithreaded system that has non-local properties or invariants that must be managed.

Constraints are a solution to this problem. A constraint is used to bind different objects together so that values can be analyzed, recalculated and propagated. Constraints offer an alternative way of programming that is otherwise difficult to achieve in a real time, multithreaded environment (read GUI framework).

Since a constraint binds objects together, a flowback seems like a good candidate to use. Actually simple constraints that only enforce an equality between values are easily implemented with callbacks.

A simple example is figure 4.5 where a gauge and digits are connected with a one-directional constraint. When the gauge is moved, the digits change. The gauge has an *onChange* method to which we supply a callback that the gauge invokes when moved. The callback is a *ValueChangedCallback* that contains the new value. Instead of creating the callback ourselves, which could be done with a command like `call(digits,valueChanged)`, we can let the digits object handle the creation of an appropriate callback. In this way we avoid as much as possible of the connection process [Myers91]. The *changeCB* method on the digits object produces a callback of type *ValueChange-Callback*, which means that the callback can be directly handed to the *onChange* method. The code looks like this: `gauge.onChange(digits.changeCB());`

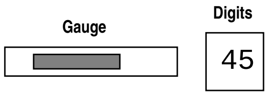

Figure 4.5: A gauge and digits that are connected.

To create more complex constraints that analyze behavior we cannot inherit from flowbacks since by definition (see section 4.1.1) a flowback should not do any work and analyzing data certainly is work. We can create analyzer objects independent from the flowback tree that uses flowbacks to send and receive data. The analyzer object performs different tasks depending on the incoming flowbacks. One analyzer could be the threshold analyzer. It can create a *ValueChangeCallback* callback that when invoked will trigger execution of either an *above* callback or a *below* callback if the value has changed to over or under the threshold. The code to create the situation in figure 4.6 looks like this:

```
gauge.onChange(digits.changeCB());
gauge.onChange(analyzer.changeCB());
analyzer.onAbove (call (button, enable));
analyzer.onBelow (call (button, disable));
```

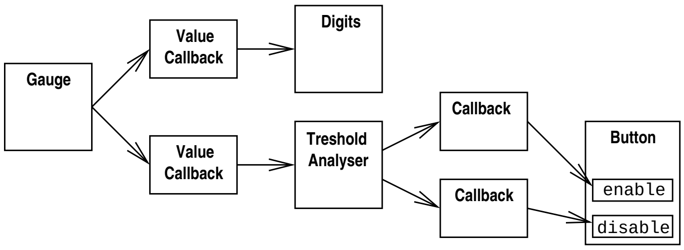

Figure 4.6: A threshold constraint enabling button only when above a certain value.

Note that we create the callbacks to the *enable* and *disable* methods. The button object could have had *enableCB* and *disableCB* methods that create appropriate callbacks. Where to use methods that create callbacks and where to create the callbacks yourself is a design consideration that depends on how the object is used. For objects where we never will call the method directly, only a create callback method is needed. For objects where we both bind callbacks to methods and call the methods directly, we might want both a create callback method and a normal method. In some cases it is easier to always create the callbacks ourselves, as in the button example.

A very common use of constraints is for dynamic layout of GUI components. Xt and Motif use constraints for layout, but the solve mechanism is incredibly inefficient. Amulet [Myers97] and SubArctic [SubArctic] use more efficient solving mechanisms. In the CIW model framework the code is written first, then the layout is created dynamically. Since constraints are extremely useful for handling resizable designs, we need to be able to add such constraints dynamically during the design phase. This will probably be done in a way that is not too different from adding callbacks to buttons with drag and drop as discussed in section 4.1.2.

4.1.5 Refining flowbacks

A callback connected from an X window object is directly related to the physical action that resulted in the callback, like a pressed button or a pressed key. However a programmer is not interested in how an action was triggered, only that it was an action with a certain meaning. We need to decouple the actual pressing of a mouse button from the action itself. We have at least three types of actions a programmer is interested in:

- *Primary action* is when an object is put to use, like pressing a button or spraying a dot in a paint program.
- *Secondary action* could be using an ubiquitous eraser in the drawing program.
- *Meta action* pops up a property menu where the user can set the properties of the object.

Instead of directly handing the callback to the window we hand it to an adapter object that will analyze the information coming from the window to see if it is time to invoke the client callback.

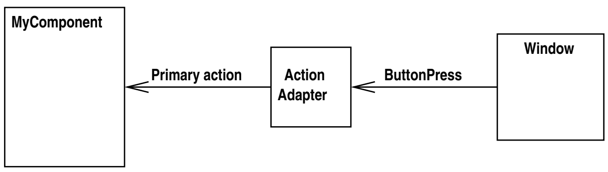

Figure 4.7: Refining flowbacks with adapters.

With the adapter as a separate object the user can decide if the primary action should be a single left button press or a right double click. There are several more possible action adapters:

pan action, focus action and drag (and drop) action. A pan action adapter can make use of a single button mouse together with the control key to let the mouse movement pan data in two directions. If we have a multi button mouse we can allocate a button directly on the mouse to let the mouse do panning of data. With action adapters, input style will be consistent over all applications. Not necessarily to a specific standard, but to the users preferences. Since the adapter is connected directly to a window, even programs that do not use GUI objects can benefit from having standardized input styles.

The adapter is similar to a control object in the MVC design. It binds actions from the view to the model. They differ because the control object is not specifically used to refine physical actions into higher level meaning based actions. The SubArctic [SubArctic] "agents" are very similar but are primarily used for translation between low level events to high level method calls. Not necessarily from physical action to meaning based action. Amulet [Myers97] has "interactors" that both bind to a window and do work. Such an interactor object can be composed by adapter objects and flowbacks.

4.2 Structuring of objects: The compositional structure versus the local structures

Sections 3.3, 3.4, and 3.5 showed that there are three structures in a program with a GUI. The compositional, the interactional and the window structure. How should these structures be aligned to work together without interfering with each other? Previous frameworks have often merged all these trees into one single tree which results in the flexibility problems that were discussed in sections 2.2 and 2.3. Since merging the compositional tree with the other trees causes severe problems for complex models, people have created separate compositional structures for the model. Unfortunately this is only useful when the model is such that it can be totally abstracted from the view. This is not the case in most of GUI programming as discussed in section 2.1.

It is not difficult to have three separate structures in an implementation. What is difficult is to define how these structures interact and where they overlap. Figure 4.7 visualizes how these structures should be organized. The solution is to orient the compositional structure orthogonally to the other structures and to assume that the structures are trees though they are in fact forests (directed acyclic graphs).

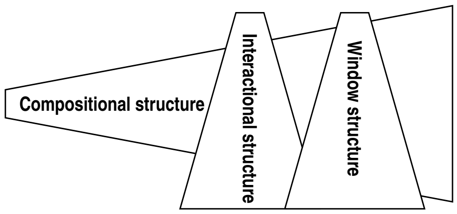

Figure 4.7: The compositional tree connects the other trees.

The object diagram in figure 4.8 shows how these structures are implemented. All objects are part of the compositional structure. A drawable is a non-viewable and non-hierarchical area where the client can draw. The drawable can be copied to a window to be displayed. Since the window is a drawable the client can draw directly into the window. A button or a scroll bar is an interaction component. These need to be grouped together with a group which usually is supplied when the interaction component is created. Groups can also be grouped since they are interaction components.

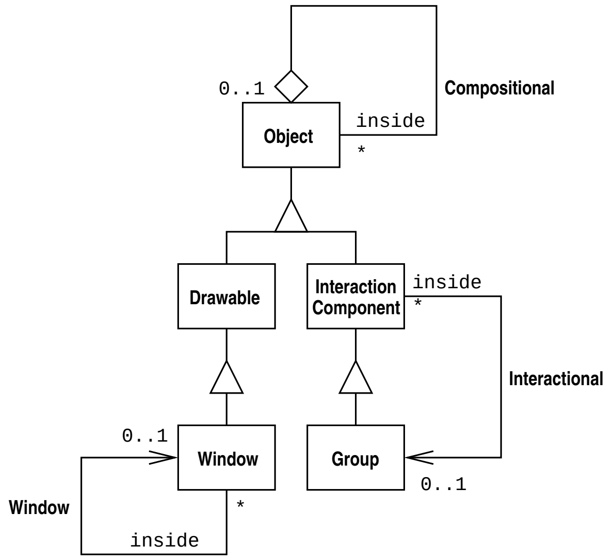

Figure 4.8: The three structures shown in an object diagram.

4.2.1 Structure definition

The structure of a program with a GUI has the following properties:

- The compositional structure is global and orthogonal to the local structures.
- The local structures consist of the interactional and the window structures.
- All structures are directed acyclic graphs.
- A component exists within a composite. A component cannot exist without its composite or has no meaning without it.

4.2.2 Example of use

A simple interaction component like a button creates the necessary window which it will use to display itself. Since the interaction component is supplied with a `group_to_reside_in` when created, the button can request a window from the group. This window is where the group wants the button to create a new window for display. The following code shows the constructor of the button. First the window is created, then a callback is created which will be called by the window whenever it is exposed.

```
Button::Button(Group *group_to_reside_in)
{
    ...
    window=x->newWindow (group_to_reside_in->getWindow(),
                         width,
                         height);
    window->onExpose (call (this, redrawButton));
    ...
}
```

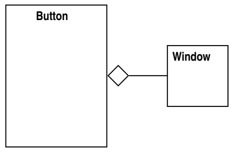

Figure 4.10: The button interaction component.

The button is then used inside the more complex *OkCancel* interaction component. The buttons are bound to methods in the *OkCancel* component with refined callbacks (see section 4.1.5). The *do* method returns the correct adapter. This component needs to create its own group since it consists of two separate components that are used together. The *invokeOK* and *invokeCancel* methods will invoke respective callbacks if they have been supplied by other components.

```
OkCancel::OkCancel(Group *group_to_reside_in)
{
    ...
    group=gui->newGroup (group_to_reside_in);
    ok_button=gui->newButton (group);
    cancel_button=gui->newButton (group);

    ok_button->do()->
          onAction(call (this, invokeOk));
    cancel_button->do()->
          onAction(call (this, invokeCancel));
    ...
}
```

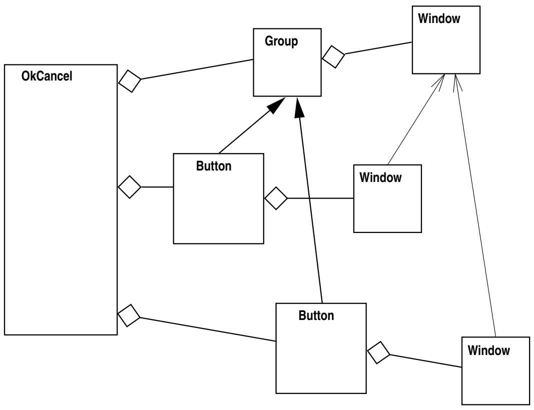

Figure 4.11: The *OkCancel* interaction component.

A complete component to set a value (or cancel the process of setting it) is built with a *LabelInput* component and an *OkCancel* component. The result is shown in figure 4.12.

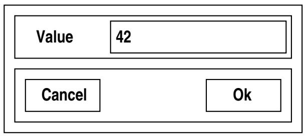

Figure 4.12: A value setting interaction component.

```
SetValue::SetValue(Group *group_to_reside_in)
{
    ...
    group=gui->newGroup (group_to_reside_in);
    labelinput=gui->newLabelInput (group);
    okcancel=gui->newOkCancel (group);

    okcancel->onCancel (call (this, cancelled));
    okcancel->onOk (call (this, okeyd));
    ...
}
```

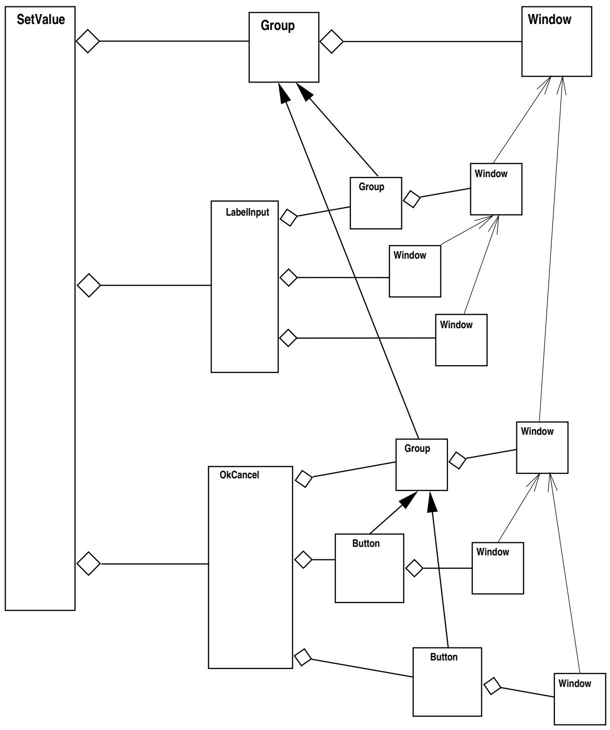

Figure 4.13: The value setting interaction component.

In figure 4.13 the three different structures can be seen. Note that some interaction components are not part of the interactional structure though they really are. But since these interaction components create their own groups, they have delegated the interactional structure responsibility to their group components.

This way to organize structures solves several of the framework properties described in section 3.1.

- Programming by composition and code locality are supported because we construct each object with composition of other objects and the methods called by the flowbacks are located within the new composite object.
- The resource management is simplified by the compositional tree structure. The resource:

  `SetValue.LabelInput.default_input: 42`

  would set the default input value of the *LabelInput* component to be 42. X windows have had this kind of hierarchical resources from the beginning. But it has never been very popular because it only followed the window structure (see figure 4.13). Because of this it was never logical where to place problem specific resources, so every program had at least two setup files, one for the graphics and one for the program itself. Since both graphics, GUI and problem specific components are reached by the compositional structure, only one resource manager should be necessary.

- We can also use low level graphics together with GUI components. This is possible since we use the same way of binding components together on both high and low level graphics. In fact using low level windows in your code is no different from how the *Button* component is implemented. Figure 4.14 shows what the structure of a new interaction component looks like.

```
MyComponent::MyComponent(Group *group_to_reside_in)
{
    ...
    group=gui->newGroup (group_to_reside_in);
    button=gui->newButton (group);
    button->do()->onAction (call (this, done));
    window=x->newWindow (group->getWindow(), width, height);
    window->onExpose(call (this, draw));
    ...
}
```

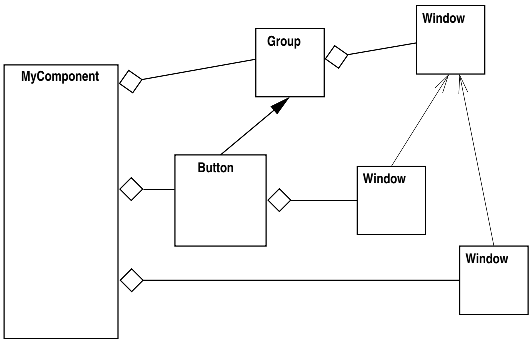

Figure 4.14: High and low level graphics used concurrently.

This is possible since the group is notified whenever a new window is created inside one of the group windows. The group has requested this information by the calling `window->onWindowCreated(call (this,handleNewWindow);`. Window managers under X-windows use this feature to handle the top level windows on the computer display. When the window manager detects that a new window is created it adds a frame with buttons to enable the user to move, resize or close the window. What we have in the CIW model is a generalization of the window manager concept. Every window is managed by a different window manager (an interaction group)! This makes it possible for users to change the layout of an application window in the same way as a normal window manager allows the user the resize and move the whole application window. The resize handles are not normally visible for a button component, but a resize tool can visualize them. Since windows can be shared between applications, tools can visualize such handles and constraints in the same window as the application they affect.

4.2.3 Flowbacks are components

A flowback has no proper meaning when the object it will call when invoked, does not exist any more. Therefore a flowback is by definition a component in the compositional tree.

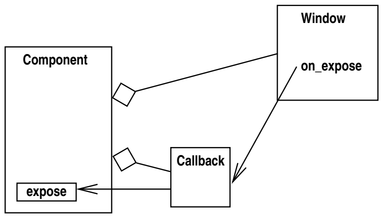

Figure 4.15: A callback is a component in the compositional tree.

In the example in figure 4.15 the window will be deleted together with the callback when the component ceases to exist. In other cases where the flowback binds to an object that can cease to exist separately, the pointer to the callback can be rendered useless. To simplify implementation in this case, the callback is not actually deleted until both the component and the objects pointing to it are gone. However when the component disappears, the callback is made inactive, which means that when invoked nothing happens.

4.2.4 Documents and views

A normal CIW component is a model that knows how to present itself given a group. If the component is running locally or remotely does not really matter. A component can by definition only exist at one place in the compositional structure but it is often very convenient to have objects that can be used simultaneously in several places. A standard example is the spreadsheet. The spreadsheet data can be stored in a multidimensional array. This data can then be viewed at the same time in several different places by viewing components. The data can be visualized as a bar graph, a simple list, a two dimensional layout or in any other conceivable way.

In figure 4.16 a data model is incorporated in the program structure. In figure 4.17, the data model is shared between spreadsheet components. Changes to the shared model are reported back through callbacks. In this case it would also be possible to use standard interfaces for callbacks.

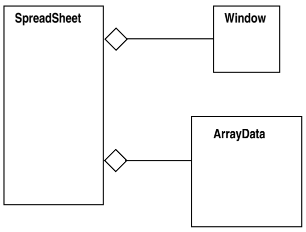

Figure 4.16: A spreadsheet with a builtin model.

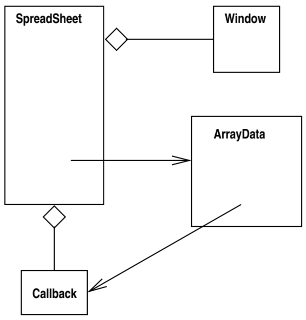

Figure 4.17: A spreadsheet with a shared distributed model.

The document view design has been studied earlier in OpenDoc, the Andrew Toolkit [Palay88] and ET++[Weinand89]. Another way to solve content embedding is to supply a group to a shared object. The shared object will then create the appropriate interaction components and/or windows in the group and display the data directly. Also for low level graphics and where complete control over the window is necessary, a single window can be handed to another object.

4.3 Scheduling of objects: Several threads that initiate callback chains

Most frameworks have only concluded that threading is necessary due to the asynchronous nature of the GUI. Within a GUI multiple things can happen in parallel. Events like a click in a window, a line drawn and a pressed key can all happen simultaneously. Unfortunately there are not many solutions to how this complex situation should be solved.

We have two directions of threading. The first is when several processes or threads are drawing at the same time. It is easy to make the GUI framework thread safe in this case, the usual solution is to use a global lock on the display. The performance benefit from a multithreaded graphics drawing system is very small because the display is essentially a single user device. Especially in the case of the graphics accelerating hardware.

The second direction is how the flowbacks are called, in which order and from which threads. Should all flowbacks be sequential or should they all be completely asynchronous? If all flowbacks are sequential, that is; they are essentially called from the same thread, then a method that does not return can block the whole system. On the other hand if every call is called from a separate thread then two button presses in the same window might arrive in the wrong order which could be bad news for a paint program.

It is important to view graphical objects as normal distributed objects since we want to solve the distributed graphical embedding problem. If each window is a separate distributed object then a call to the *drawLine* method in the window will behave exactly as if it was a call through a proxy to any other distributed object. That means that a message is sent from the caller and a thread at the called server will pick up this message and execute the corresponding code in the window see figure 4.18.

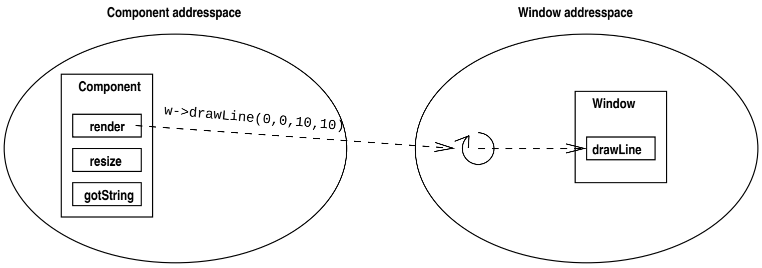

Figure 4.18: The method *render* draws a line in the window.

With the immediate calling mechanism set we can reason about how the flowbacks should be called. We want every component to behave the same whether it is used from a library in the same address space or as a distributed component. A component should not be allowed to affect other components and vice versa. If we have one single thread that handle all callbacks then a single component can lock the whole system if the called method never returns. On the other hand if every call has a separate thread then we get timing problems. Button presses can arrive simultaneously or even in the wrong order. A solution is to assume every window as an active object that monitors its state and will call any callbacks to inform others about state changes. This means that each window has its own thread. This thread will sequentially call callbacks that result from state changes like if the window was exposed or resized, or if a new window was created inside the window.

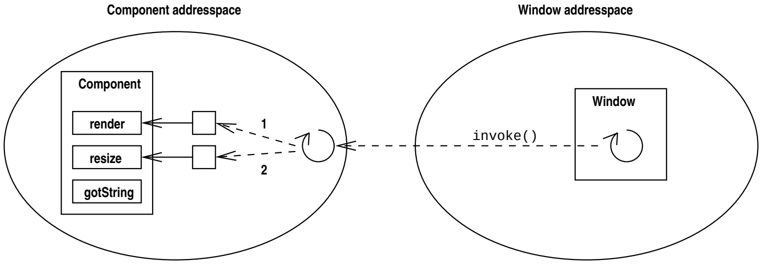

Figure 4.19: The two callbacks for the expose and resize events are handled sequentially.

A separate window will produce callbacks completely independently from the first window.

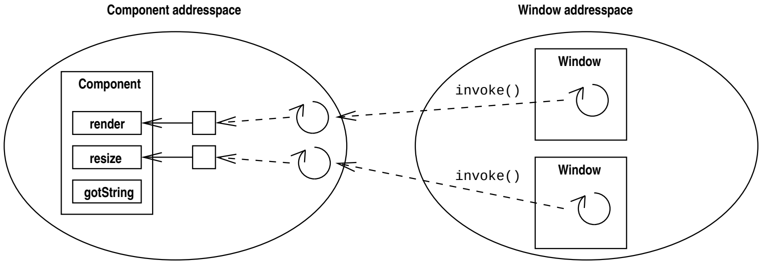

Figure 4.20: The same events are here handled by separate threads because they originate from different windows.

The action of pressing a mouse button when the mouse cursor is inside a window can be considered a state change of the window, but it does not need to be. Assume that we have several kinds of tools. One is used for pointing and pressing in windows. A tool that enters a window requests the appropriate callback from the window. If available, the callback can then be invoked by the tool when the user presses the mouse button. Drag and drop is performed in the same way. A drop flowback is stored in the window. When a client wants to drop on the window it will request a drop flowback that handles the specific type. The dropping client will then invoke this flowback directly as in figure 4.21.

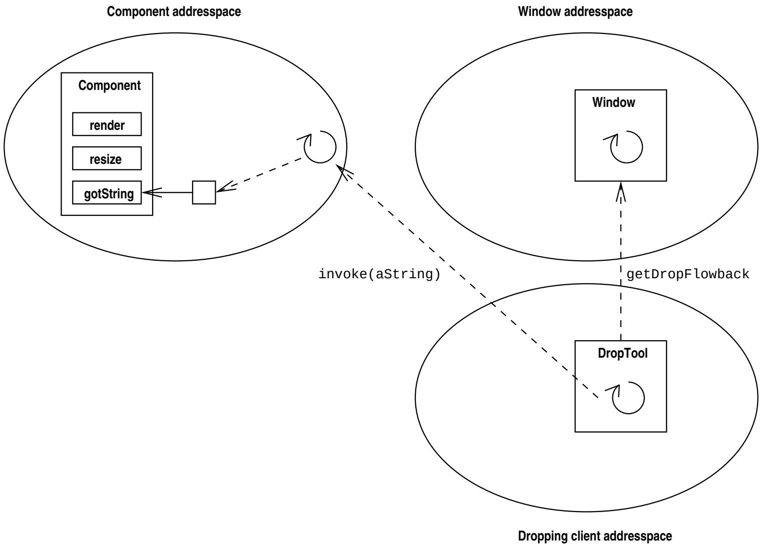

Figure 4.21: The drop flowback is called by the tool performing the drop.

The approach to bypass the window for user actions makes the system more flexible. The actions performed by a user can be recorded to create a macro. Since the flowbacks stored do not involve the window objects, the macro could be used without the graphical interface. Tools and adapters (described in section 4.1.5) perform similar tasks, which is to standardize interaction. If both are used then the adapter can be used to acquire interactions both from state changes in a window and from tools used on the same window. The adapter can decide which to use depending on system design and user preferences.

X-windows does not support threads in this way but it can be simulated on the component side by dispatching the incoming events to a queue in each window proxy. This queue is then dispatched by a separate thread for each window.

Since a flowback is a stored message it can only be used by one thread at a time. Otherwise the data in the flowback can be corrupted when two threads fill in new data concurrently.

4.3.1 Scheduling definition

Threads used in a framework based on the CIW model have the following properties:

- A thread object is part of the compositional tree.
- A callback is used to direct the thread to the correct method when started.
- Each window has its own thread.
- The window thread will handle events that happen to the window sequentially.
- Tools invoke flowbacks stored in the window.
- A flowback can only be used by one thread at a time.

4.3.2 Example of use

A thread is managed by a thread object and since it is an object it is also a part of the compositional tree. When a thread is started we need a way to tell which method the thread should run. Since an object can have several threads running inside it, it is convenient to use a callback to tell the thread where to start. This is what it looks like:

```
...
thread->start(call (this,drawLoop));
...
void drawLoop ()
{
   while (thread->running())
   {
      window->drawLine (rand1, rand2, rand3, rand4);
      x->sync ();
      sleep (1);
   }
}
```

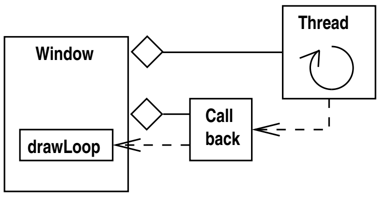

Figure 4.22: A thread is an object which uses a callback to start the appropriate method.

4.4 Production of objects: Factories

An abstract factory [Gamma95] is an object that is used to produce other objects. The factory is not an essential part of the object model. It is a convenience object that helps the programmer to create the right object, which could be a proxy to a remote object or one of several possible implementations.

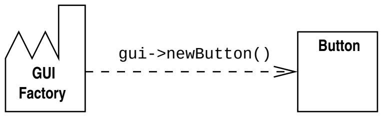

Figure 4.23: A factory and its product.

The factory can also be used to support the compositional tree. Very few object oriented languages have support for the compositional tree. (Java has internal classes that are aware of their container object. But since they can only reside within the one class where they were declared, they are not useful for the compositional tree.) To make a new object aware of its container in the compositional tree, we can supply the container as an argument at creation. However it gets a little bit tedious to always have a *this* as an argument in every creation call, so we let the factory supply it. For this we need a separate instance of the factory in each object. This instance does not contain much more than the current container object so it can be supplied when the objects are created.

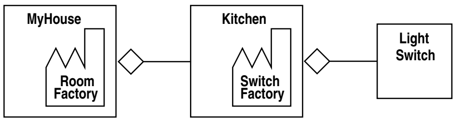

Figure 4.24: The compositional tree with a factory instance in each object.

A factory uses the resources to see if it should produce the requested object locally or remotely. The programmer can also explicitly tell the factory to use another factory. The local factory becomes a proxy and everything requested from it will actually be produced at the remote factory. The used factory must have allowed requests by issuing an *openFactory* call.

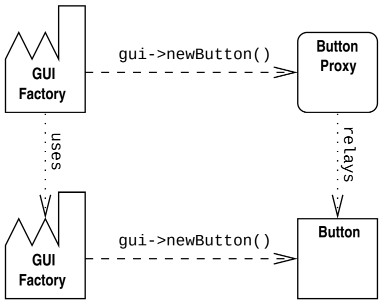

Figure 4.25: A factory can request the object from another factory.

4.4.1 Factory definition

Factories can be used in a CIW based framework. They should have the following properties:

- The factory is used to create objects. It decides which object to create and where.
- The factory keeps track of where in the compositional tree the new object will be positioned.
- A factory can use another factory with *useFactory*. It can also allow other factories to use it by calling *openFactory*.

4.4.2 Example of use

A factory can be used to produce GUI objects (like buttons and scrollbars), language objects (like sentences and words) and much more. Since sentences are objects and part of the compositional tree it is easy to use the resource system to override the actual message.

```
...
Lang::Factory lang (this);
...
sentence = lang->newSentence ("A greeting to the world.","Hello World!");
...
```

Connecting to the X server is equivalent to asking the X factory to use the another X factory. Since X is always distributed, you must always use another factory before creating any objects.

```
...
X::Factory x (this);
x->useFactory (":0.0");
...
w = x->newWindow ("hello_window", x->getScreen()->getRootWindow(), width, height);
...
```

5 Formal specification

I have described informally how a program using a graphical user interface can be structured, intra connected and threaded. My sample implementation can be used as an indication that my ideas indeed are correct, but an implementation is always full of low level design considerations and is ever changing.

Is it possible to create a description that can be used as a reference of the specification without involving actual code? Yes, and this is accomplished with formal methods. Formal methods are used to describe systems in such a way that it is difficult to make design errors. The use of formal methods makes it more likely that a program implemented from the specification will be correct. A programmer can use formal methods to achieve a deeper understanding of the process that is going to be implemented. In some cases the formal method can also be used to validate the actual solution. I will call my informal description the CIW model and the formal description the CIW specification.

There are several formal languages to chose from; VDM, Z, ObjectiveZ, B are some. I have used the B-method which makes use of the formal language B. The B-method is taking on all aspects of program development, from specification to implementation, within the same language, the B language. B was created by J-R Abrial who also created the formal language Z. B is interesting because it is used together with a toolkit that helps to develop, prove and implement the design. Since proving a formal design always is an arduous task, the support of the toolkit is very useful, even if the goal is not to produce actual code. The toolkit automatically verifies the syntax of the description and generates the necessary proof obligations. The B-toolkit can even automatically prove some proof obligations. Unfortunately for most obligations human interaction is still needed, but it makes it reasonable to actually prove the whole of the formal description.

Currently there are two toolkits, the B-toolkit developed by B-core in England and AtelierB developed by Digilog in France. B describes models with axiomatic typed set theory. A B design is made up of one or more abstract machines where the operations on the machine must be shown not to invalidate the invariants. A full explanation of B is impossible to give here but more information can be found in the B-Book [Abrial96] by J-R Abrial.

Formal methods are often used to create programs used in a safety critical environment. The actual GUI framework might not count as safety critical but it is indeed possible to create safety critical programs that depend on it. However I have not used B for the purpose of making a safety critical framework. Nor have I used B to create the actual framework code since the programs created by the B-method are mainly sequential programs that operate on booleans and integers.

I have used B to describe the CIW structure during its creation. The resulting specification can be used when implementing the CIW design. The *new* operations are equivalent to the constructors of the different objects. The influences I got from the formal design process were also very important. When I thought I had solved a problem in the model, I tried to describe it with B. Several times it was very difficult to make the formal description work out and this made me turn back to the design which indeed was flawed. The orthogonal trees are a result of this. I could not make the specification fit together when the trees were partly mixed with each other. By splitting the trees, the formal description became easier and the framework better. Also the factory pattern made a brief appearance in the formal specification before I realized that it should not be modeled. Objects can be created in many different ways and the factory is just one way. In my implementation it is convenient to use factories since they support the compositional structure.

B is not object oriented, even though abstract machines offer an object-like way to modularize the program. This does not really pose a problem since I want to describe the structure of objects and the structure itself is neither an object nor object oriented.

5.1 Specification goals

The goal of the formal specification is to describe the structure of an object oriented program that has a graphical user interface. The structure is how the objects reference each other. There can be any kind of reference between objects but I am only interested in those that are part of the CIW model. These are the references in the compositional structure, the interactional structure and the window structure.

Since these references only can exist between objects of the correct type it is interesting to model the type of each object. The different types of interest are shown in figure 5.1.

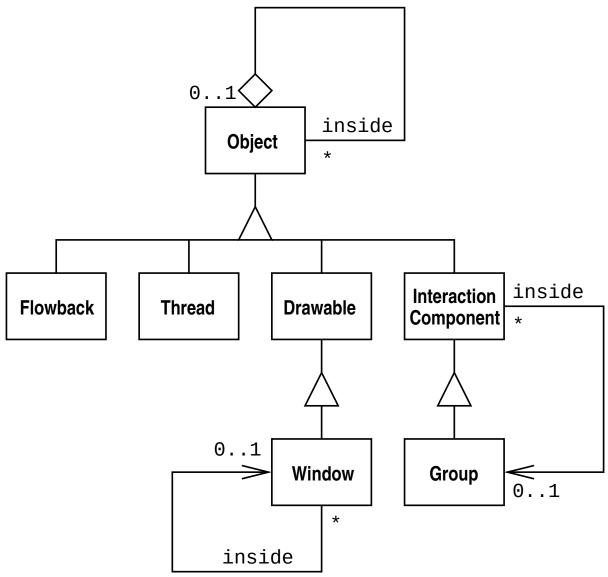

Figure 5.1: The objects that are of interest in modelling.

Any other object is either a subclass of *Object* (like the *Application* class) or *Interaction Component* (like a *Button* or *Label*) or *Drawable* (like a *Pixmap*).

I am not modeling the actual data stored in the objects or the methods available in each object. This means that the binding from a flowback to the method is ignored. Since the flowback is part of the compositional structure and it must be inside the object that it will call, the binding to the recipient is implicitly modeled.

I am also interested in the threads that traverse the system. Each object can be traversed by several threads at the same time. The exception is the flowback which is single threaded.

I want to describe the operations that are needed to construct a correct CIW structure. These operations define which preconditions must hold for the operation to be allowed. The operations must be proved to leave the system in a consistent state (the invariants hold) after they have been executed. I also need operations for traversing object with threads.

In short the specification should describe the following:

- Instances of objects and their type.
- The compositional, interactional and the window structures which are directed acyclic graphs.
- Which threads are traversing which objects.
- The operations that are used to create and traverse the structure.

5.2 Specification

MACHINE

CIW (Compositional, Interactional and Window structures)
A structure specification for programs with graphical user interfaces.

CIW

SETS

All possible object instances are in the set *OBJECTS*.

OBJECTS

VARIABLES

objects contains all instances.
flowbacks, *threads*, *interaction_components*, *drawables*, *groups*, *windows* contains the instances of respective type.
composition_struct puts objects into containers.
interaction_struct groups interaction components into groups.
window_struct puts windows inside windows.
traversing shows which threads are running in which objects.
thread_stack remembers in which order the threads called the objects.

objects ,
flowbacks ,
threads ,
interaction_components ,
drawables ,
groups ,
windows ,
composition_struct ,
interaction_struct ,
window_struct ,
traversing ,
thread_stack

INVARIANT

The different classes of objects are related in sets and subsets.

$objects \subseteq OBJECTS \land$
$flowbacks \subseteq objects \land$
$threads \subseteq objects \land$
$interaction\_components \subseteq objects \land$
$drawables \subseteq objects \land$
$groups \subseteq interaction\_components \land$
$windows \subseteq drawables \land$

An instance can not be member of mutually exclusive types but can be member of inherited types.

$flowbacks \cap threads = \{\} \land$
$flowbacks \cap interaction\_components = \{\} \land$
$flowbacks \cap drawables = \{\} \land$
$threads \cap interaction\_components = \{\} \land$
$threads \cap drawables = \{\} \land$
$interaction\_components \cap drawables = \{\} \land$

All object instances can be part of the compositional structure.
Interaction components are grouped together in groups.
Windows are inside windows.
A thread can traverse several objects and an object can be traversed by several threads.
The traversing order is remembered when the calls are rewinded.

$composition\_struct \in objects \nrightarrow objects \land$
$interaction\_struct \in interaction\_components \nrightarrow groups \land$
$window\_struct \in windows \nrightarrow windows \land$
$traversing \in threads \leftrightarrow objects \land$
$thread\_stack \in threads \rightarrow \mathsf{seq}\ (\ objects\ ) \land$

The composition, interaction and window structures are directed acyclic graphs. This means that there is no subset of the relation that maps onto itself which makes a cycle.

$\forall\ ss\ .\ (\ ss \subseteq composition\_struct \land ss \neq \{\} \Rightarrow \mathsf{dom}\ (\ ss\ ) \neq \mathsf{ran}\ (\ ss\ )\ ) \land$
$\forall\ ss\ .\ (\ ss \subseteq interaction\_struct \land ss \neq \{\} \Rightarrow \mathsf{dom}\ (\ ss\ ) \neq \mathsf{ran}\ (\ ss\ )\ ) \land$
$\forall\ ss\ .\ (\ ss \subseteq window\_struct \land ss \neq \{\} \Rightarrow \mathsf{dom}\ (\ ss\ ) \neq \mathsf{ran}\ (\ ss\ )\ ) \land$

Flowbacks can only be traversed by one thread at a time. Note that a thread object can be traversed by another thread, for example to stop or start it, so we do not restrict access to them.

$\forall\ fb\ .\ (\ fb \in flowbacks \Rightarrow \mathsf{card}\ (\ traversing^{-1}\ [\ \{\ fb\ \}\ ]\ ) \leq 1\ )$

INITIALISATION

$objects$, $flowbacks$, $threads$, $interaction\_components$, $drawables$, $groups$, $windows$,
$composition\_struct$, $interaction\_struct$, $window\_struct$, $traversing$, $thread\_stack$
$:= \{\} , \{\} , \{\} , \{\} , \{\} , \{\} , \{\} , \{\} , \{\} , \{\} , \{\} , \{\}$

OPERATIONS

Method: *newObject*
Purpose: Creates a new object and links it into a compositional tree. This is how any object that is not a flowback, thread, interaction component or drawable is created.

$oo \longleftarrow newObject \ (\ composite \ ) \quad \widehat{=}$
  PRE
    $objects \subset OBJECTS \ \wedge$
    $composite \in objects$
  THEN
    ANY $object$
    WHERE $object \in OBJECTS - objects$
    THEN
      $objects := objects \cup \{ \ object \ \} \ \ ||$
      $composition\_struct :=$
      $composition\_struct \cup \{ \ object \mapsto composite \ \} \ \ ||$
      $oo := object$
    END
  END ;

Method: *newRootObject*
Purpose: Creates a new object which is not linked into the compositional tree.

$oo \longleftarrow newRootObject \quad \widehat{=}$
  PRE
    $objects \subset OBJECTS$
  THEN
    ANY $object$ **WHERE** $object \in OBJECTS - objects$
    THEN
      $objects := objects \cup \{ \ object \ \} \ \ ||$
      $oo := object$
    END
  END ;

Method: *newFlowback*
Purpose: Creates a flowback to an object instance. The flowback will call the composite object

which the flowback is a component of. Exactly which method it will call is not specified since I do not model data or methods.

$oo \longleftarrow newFlowback\ (\ flowback\_to\ )\quad \widehat{=}$

PRE

$objects \subset OBJECTS\ \wedge$

$flowback\_to \in objects$

THEN

ANY $object$

WHERE $object \in OBJECTS - objects$

THEN

$objects := objects \cup \{\ object\ \}\ \ ||$

$flowbacks := flowbacks \cup \{\ object\ \}\ \ ||$

$composition\_struct :=$

$composition\_struct \cup \{\ object \mapsto flowback\_to\ \}\ \ ||$

$oo := object$

END

END ;

Method: *newThread*

Purpose: Creates a thread object.

$oo \longleftarrow newThread\ (\ composite\ )\quad \widehat{=}$

PRE

$objects \subset OBJECTS\ \wedge$

$composite \in objects$

THEN

ANY $object$

WHERE $object \in OBJECTS - objects$

THEN

$objects := objects \cup \{\ object\ \}\ \ ||$

$threads := threads \cup \{\ object\ \}\ \ ||$

$thread\_stack := thread\_stack \cup \{\ object \mapsto [\,]\ \}\ \ ||$

$composition\_struct :=$

$composition\_struct \cup \{\ object \mapsto composite\ \}\ \ ||$

$oo := object$

END

END ;

Method: *newInteractionComponent*

Purpose: Creates a new interaction component. Every interaction component like a button or file selection box (except groups) are created with this operation.

$oo \longleftarrow newInteractionComponent\ (\ composite\ ,\ group\ )\quad \widehat{=}$

PRE

$objects \subset OBJECTS \land$
$composite \in objects \land$
$group \in groups$

THEN

ANY $object$
WHERE $object \in OBJECTS - objects$
THEN

$objects := objects \cup \{ object \} \parallel$
$interaction\_components := interaction\_components \cup \{ object \} \parallel$
$composition\_struct := composition\_struct \cup \{ object \mapsto composite \} \parallel$
$interaction\_struct := interaction\_struct \cup \{ object \mapsto group \} \parallel$
$oo := object$

END

END ;

Method: *newGroup*
Purpose: Creates a new group inside an group.

$oo \longleftarrow newGroup \ (\ composite \ , \ group \ ) \quad \widehat{=}$

PRE

$objects \subset OBJECTS \land$
$composite \in objects \land$
$group \in groups$

THEN

ANY $object$
WHERE $object \in OBJECTS - objects$
THEN

$objects := objects \cup \{ object \} \parallel$
$interaction\_components := interaction\_components \cup \{ object \} \parallel$
$groups := groups \cup \{ object \} \parallel$
$composition\_struct := composition\_struct \cup \{ object \mapsto composite \} \parallel$
$interaction\_struct := interaction\_struct \cup \{ object \mapsto group \} \parallel$
$oo := object$

END

END ;

Method: *newRootGroup*
Purpose: Creates a new group inside no other group.

$oo \longleftarrow newRootGroup \ (\ composite \ ) \quad \widehat{=}$

PRE

$objects \subset OBJECTS \land$

$composite \in objects$

THEN

ANY $object$

WHERE $object \in OBJECTS - objects$

THEN

$objects := objects \cup \{ object \} \parallel$

$interaction\_components := interaction\_components \cup \{ object \} \parallel$

$groups := groups \cup \{ object \} \parallel$

$composition\_struct := composition\_struct \cup \{ object \mapsto composite \} \parallel$

$oo := object$

END

END ;

Method: *newWindow*

Purpose: Creates a new window inside a window.

$oo \longleftarrow newWindow\ (\ composite\ ,\ window\ )\quad \widehat{=}$

PRE

$objects \subset OBJECTS\ \wedge$

$composite \in objects\ \wedge$

$window \in windows$

THEN

ANY $object$

WHERE $object \in OBJECTS - objects$

THEN

$objects := objects \cup \{ object \} \parallel$

$drawables := drawables \cup \{ object \} \parallel$

$windows := windows \cup \{ object \} \parallel$

$composition\_struct := composition\_struct \cup \{ object \mapsto composite \} \parallel$

$window\_struct := window\_struct \cup \{ object \mapsto window \} \parallel$

$oo := object$

END

END ;

Method: *newRootWindow*

Purpose: Creates a new window inside no other window.

$oo \longleftarrow newRootWindow\ (\ composite\ )\quad \widehat{=}$

PRE

$objects \subset OBJECTS\ \wedge$

$composite \in objects$

THEN

ANY $object$

WHERE $\quad object \in OBJECTS - objects$
THEN
$objects := objects \cup \{ object \} \;\|$
$drawables := drawables \cup \{ object \} \;\|$
$windows := windows \cup \{ object \} \;\|$
$composition\_struct := composition\_struct \cup \{ object \mapsto composite \} \;\|$
$oo := object$
END
END ;

Method: *traverse*
Purpose: Traverse an object with a thread.

$traverse\ (\ thread\ ,\ object\ ) \quad \widehat{=}$
PRE
$thread \in threads \wedge$
$object \in objects \wedge$
$thread \mapsto object \notin traversing \wedge$
$(\ object \in flowbacks \Rightarrow object \notin \mathsf{ran}\ (\ traversing\ )\ )$
THEN
$traversing := traversing \cup \{ thread \mapsto object \} \;\|$
$thread\_stack\ (\ thread\ ) := thread\_stack\ (\ thread\ ) \leftarrow object$
END ;

Method: *deTraverse*
Purpose: Rewind a thread call from an object. The object must be on top of the *thread_stack* for the thread.

$deTraverse\ (\ thread\ ,\ object\ ) \quad \widehat{=}$
PRE
$thread \in threads \wedge$
$object \in objects \wedge$
$thread \mapsto object \in traversing \wedge$
$thread\_stack\ (\ thread\ ) \neq [\ ] \wedge$
$\mathsf{first}\ (\ thread\_stack\ (\ thread\ )\ ) = object$
THEN
$traversing := traversing - \{ thread \mapsto object \} \;\|$
$thread\_stack\ (\ thread\ ) := \mathsf{tail}\ (\ thread\_stack\ (\ thread\ )\ )$
END

END

5.3 Example of use

By executing the following operations the structure in figure 5.2 will be created.

program ⟵ *newRootObject* ();
group ⟵ *newRootGroup* (*program*);
top_window ⟵*newRootWindow*(*group*);
button ⟵ *newInteractionComponent*(*program*,*group*);
window ⟵*newWindow*(*button*,*top_window*);
thread ⟵*newThread*(*button*);
window_flowback ⟵ *newFlowback*(*window*);
button_flowback ⟵ *newFlowback*(*button*);
program_flowback ⟵ *newFlowback*(*program*);

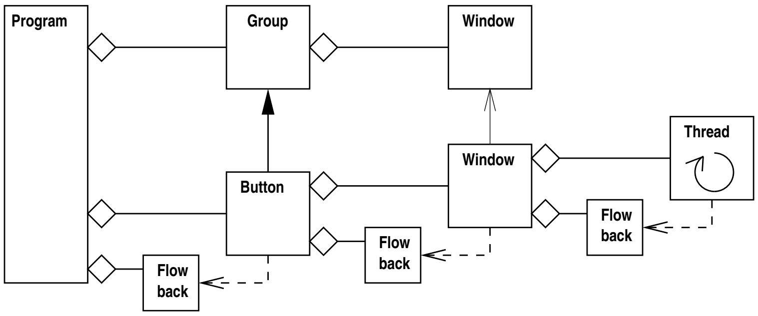

Figure 5.2: An example structure.

A button press event is then dispatched from the *thread* and traverses the chain of objects and flowbacks back and forth.

traverse (*thread*, *window_flowback*);
traverse (*thread*, *window*);
traverse (*thread*, *button_flowback*);
traverse (*thread*, *button*);
traverse (*thread*, *program_flowback*);
traverse (*thread*, *program*);
deTraverse (*thread*, *program*);
deTraverse (*thread*, *program_flowback*);
deTraverse (*thread*, *button*);
deTraverse (*thread*, *button_flowback*);
deTraverse (*thread*, *window*);
deTraverse (*thread*, *window_flowback*);

6 Implementation

I have implemented the basis of this system to show its viability. For this I have used C++ with the Gnu compiler. All code is available at [Öhrström98] where the B specification with proofs can also be found.

This is a short demonstration program that displays "Hello World!" and a button that when pressed prints "The button was pressed!" on the terminal. The program terminates after ten seconds.

```
#include"Gui.H"

struct Foo : Application
{
    Gui::Factory      gui;
    Lang::Factory     lang;
    ptr<Gui::Group>   group;
    ptr<Gui::Label>   label;
    ptr<Gui::Button>  button;

    Foo (Environment e) : Application ("Foo", e), gui (this), lang (this)
    {
        group = gui->newGroup ("FooGroup");
        label  = gui->newLabel (group, "label",
                                lang->newSentence ("user_greeting",
                                "Hello World!"));
        button = gui->newButton (group, "button",
                                 lang->newSentence ("press_command",
                                 "Press here."));
        button->on ()->action (call (this, &pressed));
        group->map ();
    }

    void pressed (Gui::PointerCallback *cb)
    {
        cout << "The button was pressed!" << endl;
    }
};

int main (int argc, char **argv, char **env)
{
    Environment e (argc, argv, env);

    Foo foo (e);

    sleep (10);
}
```

Figure 6.1: How the first example program looks on screen.

The next example is a simple low level program that creates a window which displays the text "Hello World!". It draws a random line once every second and the whole program terminates after ten seconds. Also when the user clicks inside the window, the text "Hello!" is printed.

```
#include"X.H"

struct Foo : public Application
{
    X::Factory     x;
    ptr<X::Screen> screen_;
    X::Orientation font_orientation_;
    ptr<X::Font>   font_;
    X::Text        text_;

    ptr<X::GC>     gc_;
    ptr<X::Window> root_window_;
    ptr<X::Window> window_;

    Thread         thread_;

    Foo (Environment e) : Application ("foo",e), x (this),
                          thread_ (this, "FooThread")
    {
        x->useFactory ("");
        screen_ = x->getScreen ();

        ptr<X::Color> c;
        gc_ = x.newGC (screen_);
        c = screen_->getBlackPixel ();
        gc_->setForeground (c->pixel());
        c = screen_->getWhitePixel ();
        gc_->setBackground (c->pixel());

        font_orientation_.scale (1,1);
        font_ = x->getFont ("fixedunicode", font_orientation_);
        root_window_ = screen_->getRootWindow ();

        text_.addItem (string ("Hello World!"), font_);

        window_ = x->newWindow ("HelloWorld",root_window_,0,0,200,100);
        window_->onExpose (call (this, &draw));
        window_->onButtonPress (call (this, &pressed));
        window_->map ();
        x->sync ();
        thread_.start (call (this, &run));
    }

    void pressed (X::Callback *cb)
    {
        cout << "Hello!" << endl;
    }
```

```
    void draw (X::Callback *cb)
    {
        window_->drawText (gc_,&text_,10,10+text_.height()-1,
                          font_orientation_);
        x->sync ();
    }

    void run ()
    {
        while (thread_.running())
        {
            window_->drawLine (gc_, random ()%200, random ()%100,
                              random ()%200, random()%100);
            x->sync ();
            sleep (1);
        }
    }
};

int main (int argc, char **argv, char **env)
{
    Environment e (argc, argv, env);

    Foo foo (e);

    sleep (10);
}
```

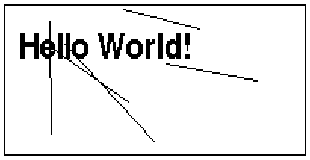

Figure 6.2: How the second example program looks on screen.

7 Discussion

With the help of the CIW model it is possible to create better user interface frameworks. The actual implementation of such a framework has to solve several conventional albeit difficult problems. For example how dynamically created and bound objects like layout constraints or callbacks bound to separate buttons should be preserved and stored in resource files. Also there are many higher level interfaces that need to be defined. The higher level interfaces are needed for tasks like content embedding in documents (how to display an image inside a text document), standard ways of offering actions that can operate on a model (which operations can be used in a drawing program), interfaces between shared tools and the objects they operate on. All these interfaces require more research and probably much trial and error, though things are better than they seem. High level interfaces have been studied more than the low level structure of graphical interfaces. OpenDoc is for example a high level interface for content embedding that has been thoroughly researched and designed.

There are also some implementation concerns on how the CIW model performs compared to current frameworks. Structuring a program according to the CIW model does not make it much larger than current object oriented frameworks. It is merely a reorganization of objects that exist in current frameworks. Compared to non object oriented frameworks a CIW framework can be larger or smaller. A well designed object oriented program can be smaller than an equivalent non object oriented program. But since programmers and users probably will want to make use of the flexibility the resulting framework will probably be larger.

A CIW based program does not run noticeably slower than a program using raw X-window graphics. The layout of a CIW based program is created dynamically from information in the resource files. There are improved layout mechanisms today that are much faster than the old ones in Xt. One reason why a CIW based framework can be slower in layout is that it might require a round-trip to the X-server for new windows created. An interaction group works as a small window manager that observes the windows created by other interaction components inside its window (as described last in section 4.2.2). The group can do the layout after all windows have been reported to the group by substructure notify events. However, waiting for the server to report new windows and then querying these windows for sizing and layout requests takes time since every query require a round trip. We cannot get rid of these queries since this is what makes it possible for several components in different address spaces to be organized together in the same window. We can however speed things up by reporting directly to the group whenever we create a new window so the group can do a layout without querying the windows again.

As for the input a CIW based program has several layers of message translation and refinement. But since the flowbacks are implemented as method calls, the number of translation levels are probably less than ten, and each translation is simple, the resulting slowdown is a small constant factor and not noticeable.

If we replace objects by remote objects, we could for example have a single button run on another computer, then we will definitely notice a slowdown. However if both the original object and the remote object are at the same network distance from the X-server then the slowdown is often neglible since every object connects directly to the X-server. If the same graphical data travel the same distance, the speed will be the same. The additional data sent if an object is distributed is the communication with the interaction component interface. But since this amount of data is much smaller than the graphical data (requests for redraw and the actual drawing commands use much more bandwidth than the message that the button was pressed), the slowdown will be small.

While the X-window system has been the inspiration for how distributed graphics should work. The CIW model is not by any means restricted to X-windows. Any graphical system that supports a retained tree of low level windows and allow distributed clients to draw in these, can be used.

A major complaint about these very tweakable systems is that no users will recognize themselves when they move to a new computer. However, if the system really is configurable then it should be possible for the user to move personal settings from one computer to another. The computer interface becomes truly personalized.

As for the flexibility, just because it is possible to create bad user interface designs does not mean that we need to do it. However if the framework designer decides what is good user interface design and forbids anything else, then there will surely be at least one person that disagrees. With a truly flexible framework almost everyone can be satisfied.

Future work will include improvement of the sample implementation to make it useful. There are also some interesting areas that need to be more thoroughly investigated. How exactly should a thread be ensured exclusive access to a flowback? Should the client lock the flowback before using it or should the flowback always be cloned in situations where several threads can use it? Another area of research is the tight coupling between the mouse and computer interaction. Can the tool concept be used to break this link? As was observed in section 4.3 a mouse button pressed inside a window is viewed as a state change in a window, not as the result of a pointing tool. This is fine as long as we assume every window to be touch sensitive, but it is a very simplistic way of interaction. In real life we use tools to do work, a pen to write, scissors to cut, the finger to point and press. This is currently simulated in each application by changing what a mouse button press will do by pressing a button in the interface. Shared tools evolved from shared buttons in section 1.3. It would be interesting to try to extend the part-editor concept in OpenDoc to a part-viewer-tool concept. This means that we are back to a model-view-controller design, but on a higher level than before, and the circle is closed.

8 Conclusions

The CIW model can be used to attain the goals described in section 3.1:

- *Programming by composition*
  The CIW model identifies three separate structures in object oriented programs with graphical user interfaces; the compositional, the interactional, and the window structures (CIW). The CIW model describes how to avoid interference between these structures. This makes it much easier to compose new complex user interaction components from simpler components.
- *Code locality*
  The CIW model describes why stored messages are a vital part of GUI frameworks and object oriented programming in general. It also shows how to make use of flowbacks (stored messages) to bind component events back to the composite without unnecessary interfaces or classes.
- *Resource management*
  The compositional structure in the CIW model is used for hierarchical naming of resources. Since the compositional structure essentially is the model of the program and it also reaches the user interaction components (since they also are part of the model), this naming scheme can be used both for the user interface resources and program specific resources.
- *Low level routines*
  The CIW model shows that the low level immediate mode graphics windows are an important part of a GUI framework. The inclusion of low level graphics in the CIW model creates a heterogeneous structure. In the CIW model there is no structural difference between a button component and an application.
- *Distributed graphical embedding*
  The CIW model demonstrates how to integrate distributed graphical embedding into a framework by treating every low level graphics window as a distributed object. This enables distribution of single interaction components and makes it easier to implement embedded content in documents.
- *Thread support*
  Threading is applied to graphical windows in the same way as if they were stand alone distributed objects. This gives the user interface a more logical threading behavior, which the programmer can rely on.
- *Reflecting model changes and state in the user interface*
  Constraints are useful to create dependencies that span over large parts of an object oriented program. Constraints in themselves can be very complicated or very simple but flowbacks can be used to implement them.

A GUI framework built according to the CIW model can support choice of output design, because it separates the meaning based parts of a program from the graphical parts. Such a framework can also support choice of input methods, because meaning based actions are decoupled from the physical actions. Also choice of input design is possible since bindings between buttons and actions can be created dynamically. Shared tools can be implemented by the same mechanism. The framework can also allow different implementations for a component. This makes choice of functionality possible.

This thesis has shown how to structure programs with graphical user interfaces to make them easier to program and more flexible for the user. The structuring makes it possible for GUI frameworks to be programmed according to the normal object oriented paradigm. GUI reusability is finally achieved by supporting programmatic composition of user interaction components.

A About this thesis

I wanted to do this work because I felt that there was more to do in GUI framework design. I had earlier during my studies realized that certain design principles made GUI programming easier. I was not sure exactly which these principles were and how they should be carried out. I knew that I wanted ease of programming, flexibility and distributed graphical embedding from a GUI framework. Unfortunately I could not start coding such a framework since I did not have the design clear and I did not want to end up where many other frameworks (commercial and research based) have ended up, as yet another nifty hack.

I had experience from GUI programming after working with the Motif framework at Ericsson and at the Karolinska Institute. I had some Smalltalk experience and I had my own experiments with GUI frameworks which I developed during the graphic interaction programming course. I guessed that I could find out the basic design principles of an improved GUI framework if I had time to study the subject over a longer period of time. Since it was time for me to do my masters thesis I felt it was a good idea to make this my thesis project.

I contacted Ken Robinson at the Department of Software Engineering of the University of New South Wales in Sydney and explained what I wanted to do in GUI research. Ken's first response after I had described my goals was "I thought you said five months not five years!". Apparently it took more than five months to complete this work. Effective full time probably nine months, total time a year and a half since I have been working after returning from Australia. The work I did after returning home, gave me extra insights in how Corba and distributed objects work and I could use that to define how graphical objects should be threaded. Most of the work on this thesis was done at the department of software engineering at the University of New South Wales. The report was written at the Royal Institute of Technology (KTH) back in Sweden. Ken introduced me to formal methods and suggested that I should use them to describe the model I hopefully would come up with.

I funded the stay in Australia with standard student loans except for the travel and insurance costs that were funded with a scholarship from KTH.

Ken Robinson was my supervisor in Australia and Jiaron Li was my supervisor in Sweden. I would like to thank Ken for making it possible for me to do this work and Martin deGroot for the many inspiring conversations we had.

References

[Abrial96] Abrial J-R *The B-Book* Cambridge University Press 1996

[Burbeck92] Burbeck S. *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)* http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html 1992

[Fresco] Fresco development team. *Fresco, a fresh approach to user interface systems* http://www.iuk.tu-harburg.de/Fresco/HomePage.html

[Gamma95] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software* Addison-Wesley 1995.

[Gtk] The GTK development team. *The Gimp Toolkit.* http://www.gtk.org/

[Li95] Li J. *Object-Oriented Constraint Programming for Interactive Applications.* Ph.D. thesis, Royal Institute of Technology, September 1995.

[Myers91] Myers B. *Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs.* Proceedings of the ACM Symposium on User Interface Software and Technology, Nov. 11-13, 1991.

[Myers93] Myers B. *Why are Human-Computer Interfaces Difficult to Design and Implement?* Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-93-183. July 1993.

A revised version appears as: *Challenges of HCI Design and Implementation,* ACM Interactions. Vol. 1, no. 1. January, pp. 73-83, 1994

[Myers97] Myers B., McDaniel R., Miller R., Ferrency A., Faulring A., Kyle B., Mickish A., Klimovitski A. and Doane P. *The Amulet Environment: New Models for Effective User Interface Software Development* IEEE Transactions on Software Engineering, Vol. 23, no. 6. June, pp. 347-365, 1997.

[Nichols96] Nichols B., Buttlar D. and Farrell J. *Pthreads programming* O'Reilly & Associates Inc 1996

[ObjectiveC] Apple *Object oriented programming and the Objective-C language.* http://developer.apple.com/techpubs/rhapsody/System/Documentation/Developer/YellowBox/TasksAndConcepts/ObjectiveC/

[Orfali96] Orfali R., Harkey D., Edwards J. *The Essential Distributed Objects Survival Guide* Wiley 1996

[Palay88] Palay, Hansen W., Kazar M., Sherman M., Wadlow M., Neueundorffer T., Stern Z., Bader M. and Peters T. *The Andrew Toolkit: an Overview* Proceedings of the USENIX Technical Conference, Dallas, February, 1988.

[Qt] TrollTech *The Qt toolkit.* http://www.troll.no/

[SubArctic] Georgia Institute of Technology. *SubArctic.* http://www.cc.gatech.edu/gvu/ui/sub_arctic/

[Tai98] Tai L-C. *The GUI Toolkit Framework page* http://www.theoffice.net/guitool/ 1998

[Weinand89] Weinand A, Gamma E and Marty R. *Design and implementation of ET++, a Seamless Object-Oriented Application Framework* Structured Programming, Vol.10, No.2, June, pp. 63-87, 1989.

[Öhrström98] Öhrström F. *C++ and B source code for a sample CIW implementation.* http://www.nada.kth.se/~d92-foh/ 1998