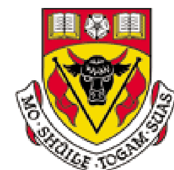
Digital Tabletop Concurrent Input Support Framework

Diplomarbeit
Fachbereich Informatik
an der Hochschule für Technik und Gestaltung
Mannheim

erstellt durch:	Andreas Herbig
Matrikelnummer:	0322496
wohnhaft:	Schütt 18 D-67433 Neustadt/Weinstraße
Email:	andreas.herbig@matabu.de
Betreuer:	Prof. Dr. Astrid Schmücker-Schend Hochschule für Technik und Gestaltung Mannheim
	Prof. Dr. Frank Maurer E-Business Engineering Group (EBE), University of Calgary, Canada

Neustadt/Weinstraße, den 15.02.2007



____
Andreas Herbig

Abstract

People in collaborative environment often need to work together on a shared task. In the physical world this is easily accomplishable. But in the world of computer this is not easily accomplished. Computer interfaces are designed for single input and all main operating systems do not support concurrent input. This means that one user can interact with one computer at a time. In a time in that large wall displays or electronic tabletops getting more and more common, the topic of concurrent input becomes more important. People working together on such large display devices and have the demand to work simultaneously.

In this diploma thesis I present the Concurrent Input Support Framework (CIF). This framework allows developer to design software written in Java using SWT that supports multiple mouse inputs. Furthermore, it automatically manages multiple cursors so that it is not necessary to add extra code to generate and process cursors – one for each mouse. The cursors themselves can be configured and rotated for an easy usage on digital tabletops. Furthermore, the CIF provides an easy to use interface to extend the framework with other components, which are using basic input events as well (e.g. mouse gestures, etc.).

Acknowledgments

This thesis would not have been possible without the support of many people:

Sincere thanks to my parents and to my brother. I am deeply grateful for the continuous support during my studies and especially during the time I have been at the University of Calgary.

Many thanks to Benjamin Deege. He helped me during this time not only with my studies.

I would like to thank Prof. Dr. Astrid Schmücker-Schend as well as Prof. Dr. Frank Maurer who gave me the chance to stay in Canada at the University of Calgary.

Furthermore, I thank all the members of the EBE group. You are all great. It is not easy to come to a foreign country and you accepted me as a friend as well as a full member of the team. It was really great to work with you all together. Special thanks to Jagoda Walny, Robert Morgan and Patrick Wilson. Thank you three for reviewing this thesis and supporting me during my work at the University of Calgary.

Contents

Abstract iii
Acknowledgments iv
Contents v
Figures viii
Listings ix
List of Abbreviations x
Chapter 1. Introduction 1
1.1 Motivation 1
1.2 Thesis Problems 2
1.3 Thesis Goals 3
1.4 Organizational Overview 4
Chapter 2. Related work 5
2.1 jUSB - The Java USB project 5
2.2 ManyMouse 6
2.3 Multiple Input Devices (MID) 6
2.4 SDGToolkit 7
2.5 CPNMouse - Multiple Mice in Windows 7
Chapter 3. Fundamentals 8
3.1 Standard Widget Toolkit (SWT) 8
3.2 Threads in SWT 11
3.3 Java Native Interface (JNI) 12
3.4 Windows messages and message queue 14
3.5 Receiving messages from the OS in SWT 16
3.6 The project ManyMouse 17
3.7 Event-Handling in Java 17
3.8 The Agile Planner 18
3.9 The Graphical Editing Framework (GEF) 19
3.10 Useful applications 19
3.10.1 Spy++ 20
3.10.2 ControlSpy 2.0 20
3.11 Digital Table 20
Chapter 4. The Implementation 22

4.1 Architecture of the CIF 23
4.1.1 Overview 23
4.1.2 The CIF event process 25
4.1.3 The "*MultipleMice*" class 26
4.2 Getting the mouse data from *ManyMouse* 28
4.3 Handle mouse events 29
4.3.1 Splitting events and process them separately 29
4.3.2 The thread responsibilities 29
4.3.3 Influence of the SWT thread concept in the implementation 30
4.4 Operating System Messages 31
4.4.1 Generate and send messages 31
4.4.2 Problems during the implementation 33
4.5 The mouse cursors 35
4.5.1 Detect the mouse cursors 35
4.5.2 Draw and update the mouse cursors 35
4.5.3 Handle the system cursor 37
4.5.4 Filter events of the system cursor 38
4.6 Provide an interface to the CIF 39
4.7 Using CIF in an application 41
4.7.1 The JAR-Files 41
4.7.2 Integrate the CIF into a standalone Java application 42
4.7.3 Integration into Agile Planner (Plugin) 43
Chapter 5. Problems 45
5.1 Open problems in the CIF 46
5.1.1 The Title bar 46
5.1.2 The Menu bar 46
5.1.3 Offset of the Title bar 47
5.1.4 The Scroll bar 47
5.1.5 The Combo box 48
5.1.6 The Textbox 48
5.1.7 Double click 49
5.1.8 The Tab folder problem 49
5.1.9 Side effects caused by the system cursor 50
5.2 Open problems in Agile Planner integration 50
5.2.1 The Menu bar 50

5.2.2 Upper button bar 51
5.2.3 System mouse cursor..... 51
5.2.4 Concurrent input..... 52
Chapter 6. Final Remarks..... 54
6.1 Future Work 54
6.1.1 Solving open problems..... 55
6.1.2 Changing the event handling design 55
6.1.3 Combining the CIF with lg3d-swt project 57
6.1.4 Integration of the tabletop data 59
6.2 Conclusion..... 60
References..... 62
Appendix A. Picture of Agile Planner 65
Appendix B. ControlSpy 2.0..... 66
Appendix C. Event Process..... 67
Erklärung..... 68

Figures

Figure 1.1. Agile Planner used on the digital table 2

Figure 3.1. The SWT architecture 9

Figure 3.2. The role of JNI 13

Figure 3.3. Java Event-Handling 18

Figure 3.4. The Digital Tabletop 21

Figure 3.5. Camera of the digital table 21

Figure 4.1. Architecture overview of the CIF 23

Figure 4.2. The CIF event process (Overview) 25

Figure 4.3. Overview of the "*MultipleMice*" class 26

Figure 4.4. Reading and handling mouse events in the CIF 29

Figure 4.5. Local widget coordinates has to be sent to the widget 32

Figure 4.6. Add different colors and text to a cursor 36

Figure 4.7. Rotating the mouse cursor – causes an offset to fire events 36

Figure 6.1. Event handling system 56

Figure 6.2. Combining CIF with *lg3-swt* 57

Figure 6.3. Adding table events to the CIF 60

Listings

Listing 3.1. Running the UI thread...11

Listing 3.2. Creating a continuously running thread...12

Listing 3.3. The "*readAndDispatch()*" method from SWT...16

Listing 4.1. Printout of the *ManyMouse* events 28

Listing 4.2. Local widget coordinates has to be sent to the widget 31

Listing 4.3. Calculating the right coordinates 32

Listing 4.4. Creating the "*lParam*" value..33

Listing 4.5. Filtering mouse events in the "*readAndDispatch()*" method...................38

Listing 4.6. Changes in the "*Display*" class 39

Listing 4.7. Register a listener in the "*EventListenerList*"..40

Listing 4.8. How to integrate the CIF 42

List of Abbreviations

API………………………………...	**A**pplication **P**rogramming **I**nterface
AWT………………………………	**A**bstract **W**indow **T**oolkit
CIF………………………………...	**C**oncurrent **I**nput Support **F**ramework
DLL………………………………..	**D**ynamic **L**ink **L**ibrary
DViT………………………………	**D**igital **Vi**sion **T**ouch
GEF………………………………..	The **G**raphical **E**diting **F**ramework
GIF………………………………...	**G**raphic **I**nterchange **F**ormat
GUI………………………………..	**G**raphical **U**ser **I**nterface
ID………………………………….	Unique **Id**entifier
JAR...	**J**ava **Ar**chive
JNI………………………………….	**J**ava **N**ative **I**nterface
JVM………………………………..	**J**ava **V**irtual **M**achine
OS...	**O**perating **S**ystem
SDG………………………………..	**S**ingle **D**isplay **G**roupware
SWT……………………………….	The **S**tandard **W**idget **T**oolkit
UI………………………………….	**U**ser **I**nterface
USB………………………………..	**U**niversal **S**erial **B**us

Chapter 1. Introduction

1.1 Motivation

Collaborating on the same computer is getting more and more important in the world. People working together on the same task trying to share one computer to support each other with their knowledge. Operating systems like Microsoft Windows XP™ do not support multiple concurrent inputs. For this problem informal research in the topic Single Display Groupware (SDG) has been done in the past. There are several projects dealing with toolkits to create applications that can handle multiple mouse and keyboard input. But there is no solution supporting concurrent input in Java using standard SWT widgets. The Concurrent Input Support Framework is an extension of the Standard Widget Toolkit (SWT) of Eclipse.

In the e-Business engineering (EBE) group at the University of Calgary we are dealing with the development of tools that assist distributed teams in developing software. One of the tools that have been developed is the Agile Planner. This is a tool which simulates a whiteboard that is used for planning meetings. We use this application on our digital table to create story cards for the next iteration. But in case of supporting only one input it is not possible to interact concurrently in such planning meetings like we do when we use paper story cards.

To demonstrate how to use the Concurrent Input Support Framework and to show the benefits using multiple input in a planning tool for the agile software development I want to integrate the framework into the Agile Planner.

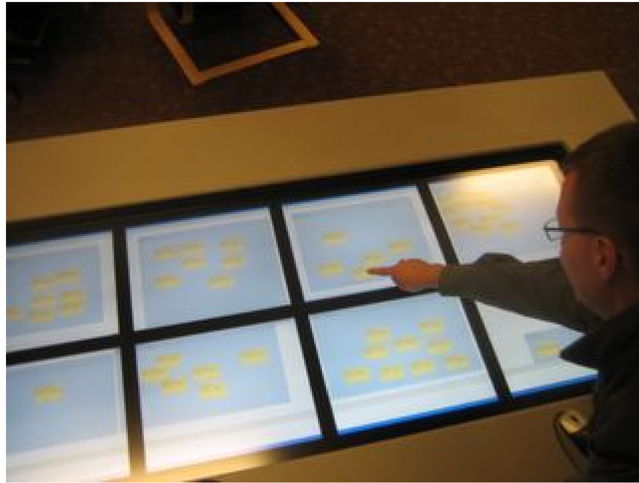


Figure 1.1. Agile Planner used on the digital table

1.2 Thesis Problems

1. ***Multiple input and identification of the mouse cursors***

   The first and most important problem is to get the data from the mice. Since Microsoft Windows XP^{TM} supports only one mouse a low level API is necessary to get the basic events which are caused from each hardware mouse (distinguishable by ID).

2. ***Cursor drawing***

   Only drawing individual mouse cursor for each mouse, giving them different colors and adding text to them is not enough. The framework should support applications especially build for digital tables. So it should be possible to rotate the mouse cursors depending on the position where a user is located around the table. Drawing manually a cursor results very often in a sluggish application performance if the implementation is not done in an efficient manner.

3. ***Handling the system cursor***

   Using individual manually build mouse cursors means to look into handling the system cursor because an active system cursor competes with the CIF cursors. If it is not possible to deactivate the system cursor then it is still visible on the application surface, moves and triggers continuously events.

This has to be handled in a way that the system cursor is not visible at all. Furthermore, the events it triggers have to be filtered to avoid side effects.

4. ***Interface to the Framework***

   In terms of extending the framework in the future there should be an interface with which it is possible to register to the CIF with other components like support of mouse gestures. These components should receive every triggered basic mouse event automatically. In other words an event handling system has to be introduced that automatically calls the components on receiving events from the hardware devices.

1.3 Thesis Goals

The goal for this thesis is to extend the SWT framework from Eclipse to handle multiple mouse input in such a way that it is easy to integrate this framework in existing SWT applications without significant effort from the average programmer who has no deep knowledge about this framework. To better define my goal I have broken it down into five subgoals:

1. ***Developing the Concurrent Input Support Framework:***
   I will develop a framework which easily can be used and integrated in already existing usual SWT applications. The framework should work with the standard SWT widgets to guarantee the compatibility to standard SWT applications. The supported operating system should be Microsoft Windows XP™.

2. ***Mouse cursors support***
   The mouse cursors should be automatically generated and it should be possible to customize them with the framework so programmers do not need to add extra code to draw cursors into their applications.

3. ***Interface to connect to the CIF***
   There should be an interface that can be used to register different applications in the CIF. This component should send all basic mouse events to the registered applications automatically.

4. ***Integrate the CIF into Agile Planner***
   To show how to use the framework and to demonstrate the benefits of having multiple mice input in an existing application I want to integrate the CIF into Agile Planner.

5. ***More description***
   Because of having the time limitation of four month the most important goal in this thesis is to provide enough information about this project. I want to impart the knowledge of how the framework works and how it has to be used. Also I want to show the difficulties and open problems which have to be solved in the future and how to combine this project with Sascha Hömigs 3D framework, called *lg3d-swt*, which deals with rotating SWT application windows. For more details about this project look into [Hoe07].

1.4 Organizational Overview

This thesis is divided into 6 chapters:

In chapter 2, to show how other projects deal with the issue of concurrent input, I give a little overview of other frameworks. I explain how they are working and for what they can used to.

In chapter 3, I introduce the fundamentals of this project. I want to give a short overview about the technologies that are used during the work in my thesis. These technologies are necessary to know to understand the implementation of the Concurrent Input Support Framework and if you want to continue with developing the framework.

In chapter 4, I present the design and the implementation of the CIF. The first step in this chapter is to introduce the structure/design of the project and then giving an introduction into the implementation. After that I explain how to use the CIF and integrate this framework into a SWT application. Furthermore, I talk about problems and bugs I experienced during the implementation and integration of the CIF.

In chapter 5, I explain possible routes to solve open problems.

In chapter 6, I finalize this thesis with purposing the next steps for the future and a conclusion.

Chapter 2. Related work

In this chapter I survey similar work related to the research of Single Display Groupware (SDG). I talk about frameworks dealing with reading raw input from hardware devices and provide this data in a simplified way. I sum up how the single solutions are working, discuss their purpose and show what kind of technologies they are using and which environment they need.

As we will see these projects are just focusing on the raw data part and the communication with the operating system. They can be used as components in a SDG framework but do not represent a complete toolkit. Most of them are special solutions of their developers' research topic. And one project that deals with multiple device inputs in Java is already out of date. It works just only with Microsoft Windows 98™ but not anymore with Microsoft Windows XP™.

2.1 jUSB - The Java USB project

The Java USB project provides an Open Source Java API for USB. It supports applications using Java to drive USB devices. Access to USB devices currently requires that they be connected to a GNU/Linux host system (it works on most recent distributions) [Bro+03]. Since the original jUSB project does not support Windows there is a related project, which extends the jUSB framework and runs with Microsoft Windows XP™. But it does not support all functionality from the jUSB core API. At the moment enumeration and monitoring of the USB is complete. Interrupt transfer[1]

[1] ***Interrupt transfers*** - devices that need guaranteed quick responses, e.g. pointing devices and keyboards. [Wik07]

and control transfer[2] are partly implemented. Bulk transfer[3] is still subject of future work [Sta03].

This project does not deal especially with input devices like mice our keyboards but it can be used to read USB devices out of Java. To use this framework you have to install a driver on the host system that enables the access to the USB devices. Furthermore, you have to access a Dynamic Link Library (DLL) using the Java Native Interface to get the data you need.

Since the interrupt transfer is only partly implemented I assume it causes some more effort to access the mouse data using the jUSB project.

2.2 ManyMouse

ManyMouse is a library that abstracts the handling of multiple mice input into a tiny, cross-platform API. It works with Linux, Mac OS and Microsoft Windows XP™ and is written in C. It is possible to include this into Java by using Java Native Interface (JNI). It is meant to be used with games and non-traditional applications with unique input needs. It does not only support USB. In many cases it can make serial mice and built-in laptop track pads available [Gor05]. For more details just look into Chapter 3.6. Because of its tiny structure and producing basic events which can easily be processed I decided to include *ManyMouse* as a component of the CIF to connect to the operating system and get the mice events of every single hardware device.

2.3 Multiple Input Devices (MID)

The Multiple Input Devices (MID) toolkit is a Java package that supports input from multiple devices on a single computer. It provides developers using Java the ability to write SDG applications. MID consists of a cross-platform Java layer, and a native platform-specific layer. It only supports USB mice on Microsoft Windows 98™. It

[2] ***Control transfers*** - typically used for short, simple commands to the device, and a status response. [Wik07]

[3] ***Bulk transfers*** - large sporadic transfers using all remaining available bandwidth (but with no guarantees on bandwidth or latency), e.g. file transfers. [Wik07]

does not work with other versions of Windows. In order for applications to use MID, a developer must replace all uses of the standard Java events with the extended MID events. Applications that use MID will always work. If an application that uses MID runs on a non-MID-ready system (i.e., the native DLL file is not installed and/or USB mice are not present and/or the operating system is not Microsoft Windows 98™), MID reverts to using Java mouse events and the application is usable like a simple Java application with just one input device [HB99].

2.4 SDGToolkit

SDGToolkit is a framework to develop SDG applications and is written in C/C# for the .NET framework. It provides the basic primitives required to obtain mouse and keyboard input from all standard device types, including both serial and USB. SDGToolkit automatically captures and manages multiple mice and keyboards, and presents them to the programmer as uniquely identified input events relative to either the whole screen or a particular window. It provides multiple cursors, one for each mouse. The only problem is how widgets (buttons, menus etc) are handled. SDGToolkit supports only limited interactions with standard widgets. However, it provides a widget class layer that support developers creating novel graphical components that recognize and respond multiple inputs [Tse04] and [TG02].

2.5 CPNMouse - Multiple Mice in Windows

CPNMouse was developed as a side-project for the scientific application CPNTools[4]. It is implemented in C as a mouse driver that has to be installed on the host system. The authors of this mouse driver say that CPNMouse allows using more than one pointing device in Microsoft Windows 2000™ or Microsoft Windows XP™ applications. However, features provided by the project are limited to features needed for the CPN tools project. Furthermore, it does not provide sufficient support for applications using mice concurrently. In other words, the implementation of the CPNMouse is still in the early phases of development [Wes02].

---

[4] ***CPN Tools*** is a tool for editing, simulating and analysing Coloured Petri Nets

Chapter 3. Fundamentals

In this chapter I introduce the basics and the components used during my project. These fundamentals are necessary to understand the surface around the project and to understand with which components the Concurrent Input Support Framework work with. First, I explain how SWT is designed, how it works and how it communicates with the operating system. In addition I present the advantage of SWT toolkit compared to another GUI toolkit (Swing). Then I will explain how to use or include the tools which are needed to communicate with the operating system. We will have a short look into JNI, Windows messages and message queues. Furthermore, I talk a little bit about event handling in Java which is used to provide an interface to the CIF and about the Agile Planner project and a plugin, called GEF, that is used in the Agile Planner and which is build atop of SWT. Finally, I introduce some useful tools I used during my project that are essential for the message (operating system) based development.

3.1 Standard Widget Toolkit (SWT)

The Standard Widget Toolkit (SWT) is a framework for user interface development in Java. It is an alternative to other GUI toolkits in Java like AWT and Swing. SWT itself is written in Java but it uses the native GUI libraries of the operating system by using the Java Native Interface (JNI). This means that platform independency is lost. But applications written in SWT are still portable. Each platform (operating system) has its own version of the SWT toolkit. At least it uses a different native object that is for example a "**.dll*" on Windows and a "**.so*" on Linux.

If there is no native code available for an operating system, then SWT cannot be used. The good news is that the native components of SWT have been written for the majority of operating systems [Rit02].

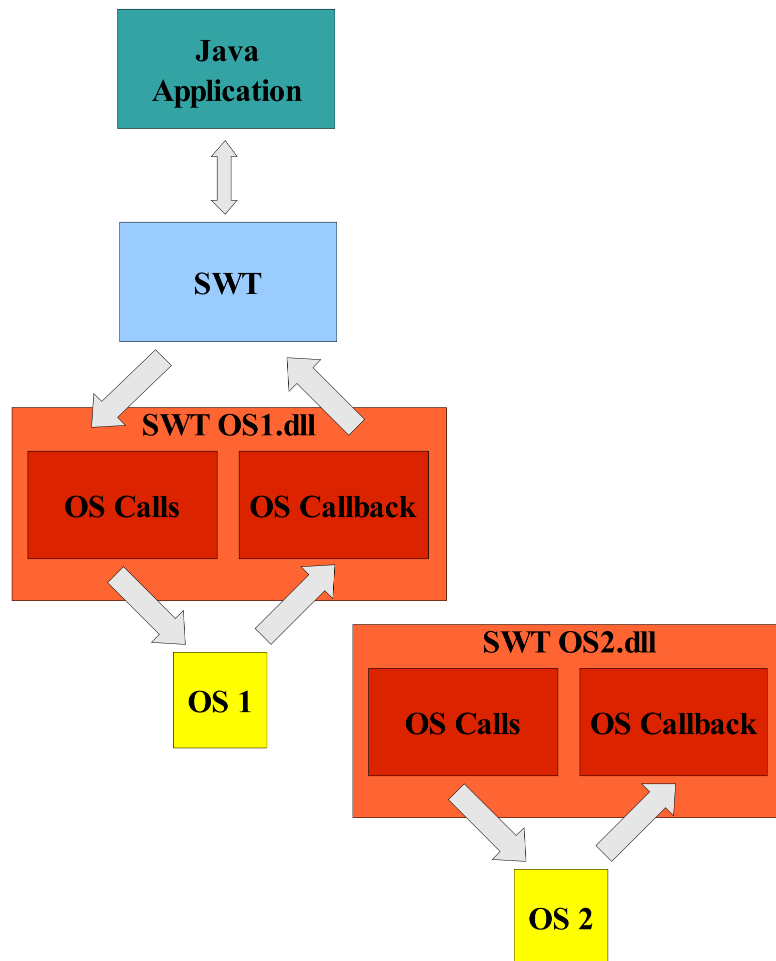


Figure 3.1. The SWT architecture

The main advantage of SWT is that it is fast. In a comparison between applications using different toolkits the graphical operations using SWT are faster then for example applications written in Swing.

To see this difference in performance between SWT and Swing, create a breakpoint in the method call of the "*readAndDispatch()*" in your application. Switching the window causes a paint event. Following this event after a few steps you will recognize that you are already in the operating system, i.e. not much executed SWT code. If you follow a fired AWT-event on its way to the operating system in Swing you will observe that there is a significant amount of Swing code that needs to be executed [Ass05].

Furthermore, SWT uses the operating system's user interface components which result in another advantage. It is not only fast, it presents a native GUI. So for

skinable operating systems like Microsoft Windows XP™ SWT presents the components in the same design.

Compared to other user interface toolkits the garbage collection of SWT is managed totally different. There is no automated garbage collection. The programmer is responsible to dispose every widget instance that is created. To make disposal easier, calling the dispose method on a widget automatically calls the dispose method for all children of the parent widget.

The two most important classes in the SWT toolkit in general are the "*Display*" and the "*OS*" class. The "*Display*" class is responsible for managing the connection between SWT and the underlying operating system. The "*OS*" class provides various methods for accessing information about the operating system. Furthermore, it has overall control over the operating system resources which SWT allocates [IBM⁺05b]. To provide these functionalities the "*OS*" class is used to enable the connection to the operating system. The "*OS*" class is nothing else more than a wrapper which calls the native methods of the operating system using Java Native Interface (see Chapter 3.3) and provides all needed static values that are necessary for the communication with the operating system.

How to create an application using the SWT toolkit?

In order to create an application that uses SWT widgets, you have to use the "*org.eclipse.swt*" and the "*org.eclipse.swt.widgets*" packages[5]. The organization of an SWT application is always the same. You have a "*Display*" and a "*Shell*" to hold the SWT widgets. The display is an object that contains all GUI components but is not visible, only the components added to the display are visible. A shell is a window in this application. It can be added to a display, then it is called top level shell. If it is added to another shell it is called child shell.

The properties of the SWT widgets are typically initialized. However, they can be defined or modified like those in Swing. In both frameworks there are several equal functions provided. In SWT for example to define the size of a shell we use the "*shell.setSize()*" function which is exactly the same call in Swing.

---

[5] Packages of the Eclipse™ Standard Widget Toolkit

Then after initializing the shell in SWT you need to open it with "*shell.open()*" and run an event loop that starts and manages the whole application (Chapter 3.2). The definition of all other widgets has to be done in front of this loop. When halting the application, the display must be disposed to trigger the garbage collection.
Placing widgets on the shell can be done by defining the location and size absolute. But there are also some layout managers available that can be used [Ram03]. A layout manager is an object that implements the LayoutManager interface. It determines the size and position of the components within a container. Although components can provide size and alignment hints, a container's layout manager has the final say on the size and position of the components within the container [Sun06b].

3.2 Threads in SWT

In any GUI program, Swing or SWT, threading is a problem. In SWT the entire application runs in a single UI thread. If you look into the code of an SWT application you will find following code:

```
while (!shell.isDisposed()) {
      if (!display.readAndDispatch())
            display.sleep();
}
```

Listing 3.1. Running the UI thread

These lines of code above are looking more or less like unnecessary code. But it is the most important part in a SWT application. This code is responsible that all user interface processes run in the main thread of the application [Ass05]. This means a SWT application runs in one thread and it is enforced from SWT that widgets are only allowed to be used from this thread. This prevents a non-UI thread from updating any UI component. If an update to such a component is done a "*SWTException*" will be thrown. This is likely to improve the structure and separates the view classes from the thread classes. It forces a developer to put long running code in threads and do UI updates in the correct place [Rit02].
Nevertheless, it is possible to run separate threads in an application. The "*Display*" class provides the "*syncExec()*" and "*asyncExec()*" functions as a mechanism to

resolve the issue. With these methods it is possible to start a new thread and to pass the display instance to access the SWT widgets out of other threads.

```
private int rate = 15;
...
display.timerExec(rate, new SwtExampleThread());
...
private class SwtExampleThread implements Runnable {
	public void run() {
		// do something here
		display.timerExec(rate, this);
	}
}
```

Listing 3.2. Creating a continuously running thread

One problem with these two methods is that the "*run()*" function of the threads is called just once. After finishing this method the thread stops. This works maybe fine firing SQL statements to a database or for including simple code that has to run just once, but if you need a thread that is running all the time, e.g. to keep something updated in the GUI, you need something else. For this issue the "*Display*" class provides the "*timerExec(int milliseconds, Runnable runnable)*" function. With this function it is possible to create a continuously running thread (Listing 3.2) with a recursive call of the "*timerExec()*" function. It causes the "*run()*" method of the runnable to be invoked by the UI thread. The first parameter is the delay in milliseconds before running the runnable. The second parameter is the runnable code that has to run on the UI thread. To stop this kind of thread you have to implement a constraint. But if the UI thread is stopped, these 'child'-threads will be stopped as well.

3.3 Java Native Interface (JNI)

The Java Native Interface (JNI) is a native programming interface of the Java platform. Applications that use the JNI can incorporate native code written in programming languages such as C and C++, as well as code written in the Java programming language. While you can write applications entirely in Java, there are situations where Java alone does not meet the needs of your application. Programmers

use the JNI to write Java native methods to handle those situations when an application cannot be written entirely in Java [Sun06a].

The following examples illustrate when you need to use Java native methods:

- The standard Java class library does not support the platform-dependent features needed by the application.
- You already have a library written in another language, and wish to make it accessible to Java code through the JNI.
- You want to implement a small portion of time-critical code in a lower-level language such as assembly.

The JNI is a powerful feature that allows you to take advantage of the Java platform, while still utilizing code written in other languages. As a part of the Java virtual machine implementation, the JNI is a *two-way* interface that allows Java applications to invoke native code and vice versa. Figure 3.2 illustrates the role of the JNI [Lia99].

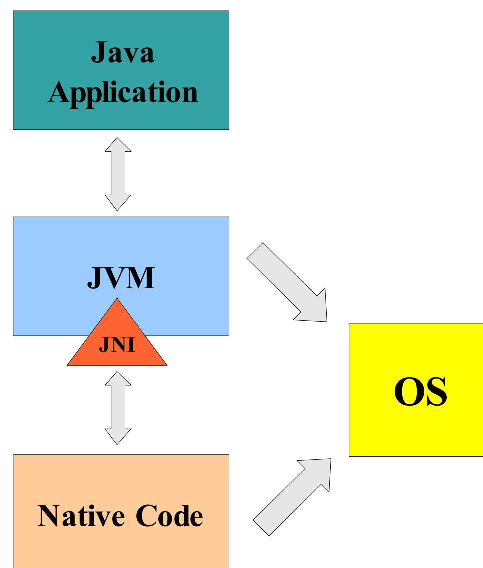


Figure 3.2. The role of JNI

As a two-way interface, the JNI can support two types of native code: native libraries and native applications:

- You can use the JNI to write native methods that allow Java applications to call functions implemented in native libraries.

- The JNI supports an invocation interface that allows you to embed a Java virtual machine implementation into native applications. For example, a web browser written in C can execute downloaded applets in an embedded Java virtual machine implementation.

In case of this thesis Java Native Interface is used to access the *ManyMouse* project. The *ManyMouse* project is written in C to access the mice data through the windows message queue, this is not possible from within Java. Also, the SWT framework communicates with the operating system through JNI to use the system specific widgets and sending and receiving events.

Using JNI causes problems because you are loosing two benefits of the Java platform. First, the application cannot be used on every platform anymore. The native part has to be recompiled for the specific operating system. Second, while the Java programming language is type-safe and secure, native languages such as C or C++ are not. As a result, you must use extra care when writing applications using the JNI [Lia99].

3.4 Windows messages and message queue

Windows-based applications do not make explicit function calls because they are event-driven. These applications wait for the system to pass input to them. Each window has a function (window procedure) which is called when the system has input for this window. The window procedure processes the input and returns the control to the operating system when it is finished.

The input is passed as a message to a window procedure. These messages can be generated by the operating system and an application. An example for a message from the operating system is an input event - when the user types, moves the mouse, or clicks a control (e.g. a button). An application can generate messages to its own windows or to communicate with windows in other applications.

To send a message to a window there are four parameters which have to be defined: a ***window handle***, a message ***identifier***, and *two values* called ***message parameters***.

- The ***window handle*** identifies the window to which the message should be sent.

- A ***message identifier*** is a named constant that identifies the purpose of a message. Through this identifier the system knows how to process the message.
- ***Message parameters*** specify data or the location of data used by a window procedure when processing a message. The meaning and value of the message parameters depend on the message.

To route mouse and keyboard inputs to the appropriate window, the system uses message queues. Whenever the user moves the mouse, clicks the mouse buttons, or types on the keyboard, the device driver for the mouse or keyboard converts the input into messages and adds them in the system message queue. The system removes the messages, one at a time, from the system message queue, examines them to determine the destination window, and posts them to the message queue of the thread that created the destination window. A thread's message queue receives all mouse and keyboard messages for the windows created by the thread. The thread removes messages from its queue and directs the system to send them to the appropriate window procedure for processing [MS07a].

Sending messages to the OS message queue

To create and send messages to a window procedure there are mainly two different approaches.

First, there is the "*PostMessage*" function which posts a message in a message queue and the "*PostThreadMessage*", which is similar to the "*PostMessage*" except of the first parameter. This is used as a thread identifier and not as a window handle.

Second, there is the "*SendMessage*" function that that is used in the CIF and has also been used by SWT itself. "*SendMessage*" calls a window procedure and waits until the messages has been processed and returns a result.

A message can be sent to any window in the system; all that is required is a window handle. The system uses the handle to determine which window procedure should receive the message [MS07a].

3.5 Receiving messages from the OS in SWT

In order to receive operating system messages in an application a message loop must be provided. This message loop retrieves messages from the thread's message queue and dispatches them to the appropriate window procedures. In fact this is what happens in the case of the "*readAndDispatch()*" method in SWT (Listing 3.3).

```
public boolean readAndDispatch() {
        ...
        if (OS.PeekMessage (msg, 0, 0, 0, OS.PM_REMOVE)) {
                if (!filterMessage (msg)) {
                        OS.TranslateMessage (msg);
                        OS.DispatchMessage (msg);
                }
                runDeferredEvents();
                return true;
        }
        ...
}
```

Listing 3.3. The "*readAndDispatch()*" method from SWT

As you see in the code snippet above there are several operating system methods called in the "*readAndDispatch()*" method:

- The "***PeekMessage***" function checks the message queue for a message that matches the filter criteria and copies this message to a MSG structure. In this case it looks for messages for the SWT framework. The "*PeekMessage*" is similar to the "*GetMessage*" function which is also provided from Microsoft Windows XP™. The main difference between these two functions is that "*GetMessage*" does not return until a message matches, whereas "*PeekMessage*" returns immediately regardless of which message is in the queue.
- The "***TranslateMessage***" function translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the "*GetMessage*" or "*PeekMessage*" function.
- The "***DispatchMessage***" function dispatches a message to a window procedure. In other words this method sends the events to the specific widget.

3.6 The project ManyMouse

ManyMouse is a library that abstracts the handling of multiple mice input into a tiny, cross-platform API. It is meant to be used with games and non-traditional applications with unique input needs. On most platforms, it can at least make all USB mice available, but in many cases it can make serial mice and built-in laptop track pads available too [Gor05].

As the Concurrent Input Support Framework should work with Microsoft Windows XP™ and Java it is necessary to prepare the project before starting to code. First, the Dynamic Link Library "*ManyMouseJava.dll*" has to be compiled and then copied into the root folder of the SWT framework. To use the framework with a different operating system the sources of *ManyMouse* has to be recompiled for the specific host system. For details see [Gor05].

How does ManyMouse work?

In case of Linux and Mac OS X *ManyMouse* reads the data from the input devices directly with the system-specific APIs. This is different for Windows. The Windows mouse events do not supply the needed information. For this *ManyMouse* reads the data from the Windows message queue and creates the *ManyMouse* events.

In the case of this project *ManyMouse* is used under Microsoft Windows XP™ and as the mice data is required in Java the *ManyMouse* project has been include through the Java Native Interface (JNI). Through this interface *ManyMouse* returns the basic events which are caused from the mice – *MouseUp*, *MouseDown* and *MouseMove* (relative change). It supports also the scroll wheel what is not handled in the CIF yet.

3.7 Event-Handling in Java

Events are caused whenever the user moves, clicks the mouse buttons or types something on the keyboard. Usually every UI element (in SWT every widget) provides an "*EventListenerList*" where an "*EventListener*" can be registered.

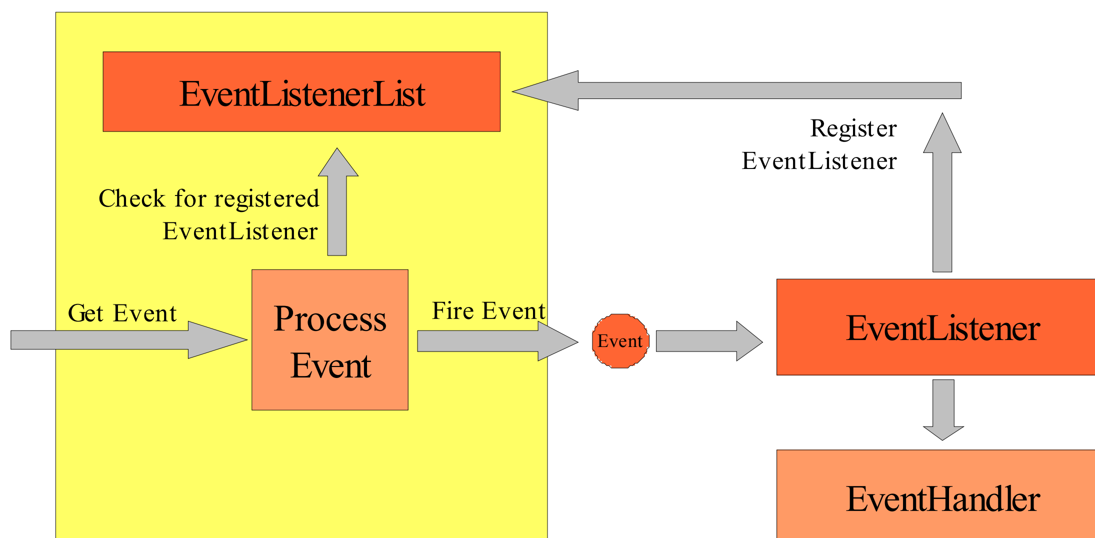


Figure 3.3. Java Event-Handling

Once an event is caused and sent to this UI element it checks its "*EventListenerList*" and notifies every "*EventListener*" that is registered for exactly this event.

This is the event handling concept how it is used for user interface development in Java in principal. But this concept can also be used for non UI elements. In case of this project I added an "*EventListenerList*" to the CIF that fires mouse events to every registered listener (Chapter 4.6).

3.8 The Agile Planner

The Agile Planner is designed to simulate one of the most common agile software planning processes – story cards on a whiteboard [Ral06]. This application is developed to improve the planning process by providing teams with a digital environment that supports information management in addition to natural interactions. Regarding support for natural interaction, Agile Planner allows for planning artefacts to be created, edited, and organized in a similar manner to paper based planning meetings. Agile Planner supports planning meetings by providing a common workspace similar to a whiteboard for story cards to be created and organized on [MM06]. As the application runs on a digital table and several people are present during a planning meeting there is obviously a need for concurrent input. If you compare this to a paper based planning meeting where everybody creates story cards and adds them to the current iteration at the same time, it would be an advantage if

multiple people could use the Agile Planner on a digital table creating story cards in the same way. To see a picture of the Agile Planner look into Appendix A.

3.9 The Graphical Editing Framework (GEF)

With the Graphical Editing Framework[6] (GEF) from Eclipse it is easy to develop a rich graphical editor from an existing application model. All graphical visualization is done by the Draw2D framework that is a standard drawing framework based on SWT. With it, it is possible to build nearly every model [MDG+04]. GEF is completely application neutral. It provides the groundwork to build almost any application, including but not limited to: activity diagrams, GUI builders, class diagram editors, state machines, etc. [Ecl07]. Also it is possible to change data in this model that can be handled in a graphical editor just using common functions like drag and drop, copy and paste and actions that are invoked from menus or toolbars [MDG+04].
Draw 2D is packed as a separate plugin in Eclipse and provides the graphical system that GEF depends on. It is hosted in a SWT canvas control and manages the painting and mouse events that occur in this canvas by delegating them to the Draw2D figures. Figures are like windows. They can be nonrectangular shapes and can be nested to compose complex scenes or custom controls. Furthermore, these figures can have focus and selection. They receive mouse events, have their own coordinate system, and can have a cursor [Mee06].

3.10 Useful applications

During application development there may be some cases where a traditional debugger is not very useful. This can be the case if the bug you are searching involves focus or other UI aspects. This is challenging because the debugger modifies the focus of the window whenever you hit a breakpoint. Another problem is that sometimes there are parts you want to investigate which are on the side of the operating system. Such as messages send to controls via the windows message queue. For such difficulties there are useful application which enable you to investigate what kind of

---

[6] http://www.eclipse.org/gef/

messages are received from such a control or with which you can analyse what kind of messages have to be generated to trigger a specified widget action.

3.10.1 Spy++

Spy++ is a Win32 based utility that gives a graphical view of the system processes, thread, windows and window messages. You can view the parent-child window relationships, as well as the flag settings and window positions [WB00]. This tool is useful to investigate widgets. Primarily to understand what kinds of messages are arriving on the widget when you just cause a button click. The second advantage was in comparing and controlling the messages by parameters that were sent from the CIF to the operating system.

3.10.2 ControlSpy 2.0

ControlSpy is a tool to help developers understand common controls and their response to messages and notifications. With *ControlSpy*, you can see instantly how different styles affect the behaviour and appearance of each control, and also how you can change the state of each control by sending messages [MS07c].

ControlSpy shows a selected common control in the centre of its application window. It is possible to send messages or notifications to this control. On the right side of the application all messages will be listed as they are received by the control. It is an ideal application to see what events/messages are necessary to be created. To see a picture of the Agile Planner look into Appendix B.

3.11 Digital Table

The digital table is build out of eight LCD monitors having an overall maximum resolution of 6400 by 2400 pixels (each display 1600 by 1200). To support 3D acceleration the computer that is used with the table comes with two MatroxQID Pro[7] graphics cards. Each card is responsible for four monitors.

[7] http://www.matrox.com/graphics/en/dispatch/products/qid/qidpro.php

The operating system is Microsoft Windows XP™. The table supports two different kind of inputs. First, is the normal mouse and keyboard input. Second, to provide a touch screen overall eight LCD monitors the table uses the Digital Vision Touch (DViT) technology from SMART Technologies, Inc.[8].

Figure 3.4. The Digital Tabletop

The DViT Board has a camera on each corner of the table that observe an array of infrared sensors on each side of the display. The board can track the 2D position and size of up to two touches and it supports finger as well as pen inputs. When a pen or finger is moved over surface of the table, these changes are observed by the cameras. Using these images the DViT Hardware calculates the position of the point and makes this information available through the low level SMART Board SDK.
A limitation of the SMART DViT Board is that it cannot uniquely identify a finger or pen. This means the ID that is provided of the SMART Board SDK may change if a person lifts the finger or pen from the display surface.

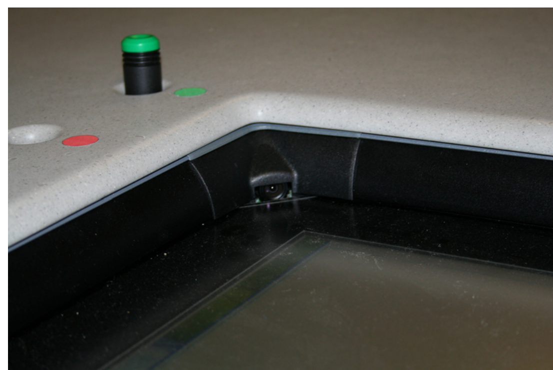


Figure 3.5. Camera of the digital table

[8] http://smarttech.com/

Chapter 4. The Implementation

In this chapter I want to give an overview about the implementation and the design of the Concurrent Input Support Framework. This chapter continues from the fundamentals in Chapter 3 and describes how these basics are used in the development process of this project. As we will see after an overview about the design of the CIF, the framework implementation is divided into five main parts. First, I explain how to get the data from the *ManyMouse* component via JNI. Second, I show how the data from *ManyMouse* gets handled and processed on the CIF side and I talk about the threads that have been created for this process. Third, I introduce how to generate event/messages for the operating system and how to send these to the Windows message queue. Fourth, I describe the part of the CIF that is responsible for drawing mouse cursors and explain what functionalities are available in the CIF. Furthermore, I explain in this part how I had to handle the system cursor and why I had to modify the "*Display*" class in SWT. Fifth, I describe the interface to the CIF that can be used to attach other projects which need the same mouse input.

The last part of this chapter ends with a description of using the Concurrent Input Support Framework including what is necessary to start and how to include the CIF into a standalone SWT application. I will finish with a discussion of how CIF was integrated into Agile Planner.

I ran into a lot of problems due to a lack of experience in SWT development and the Windows messaging. For this reason I will talk a little bit about some of the problems at the end of each component to show a developer who takes over this project how to begin and how to avoid the same problems, which I had to expend much effort to.

4.1 Architecture of the CIF

4.1.1 Overview

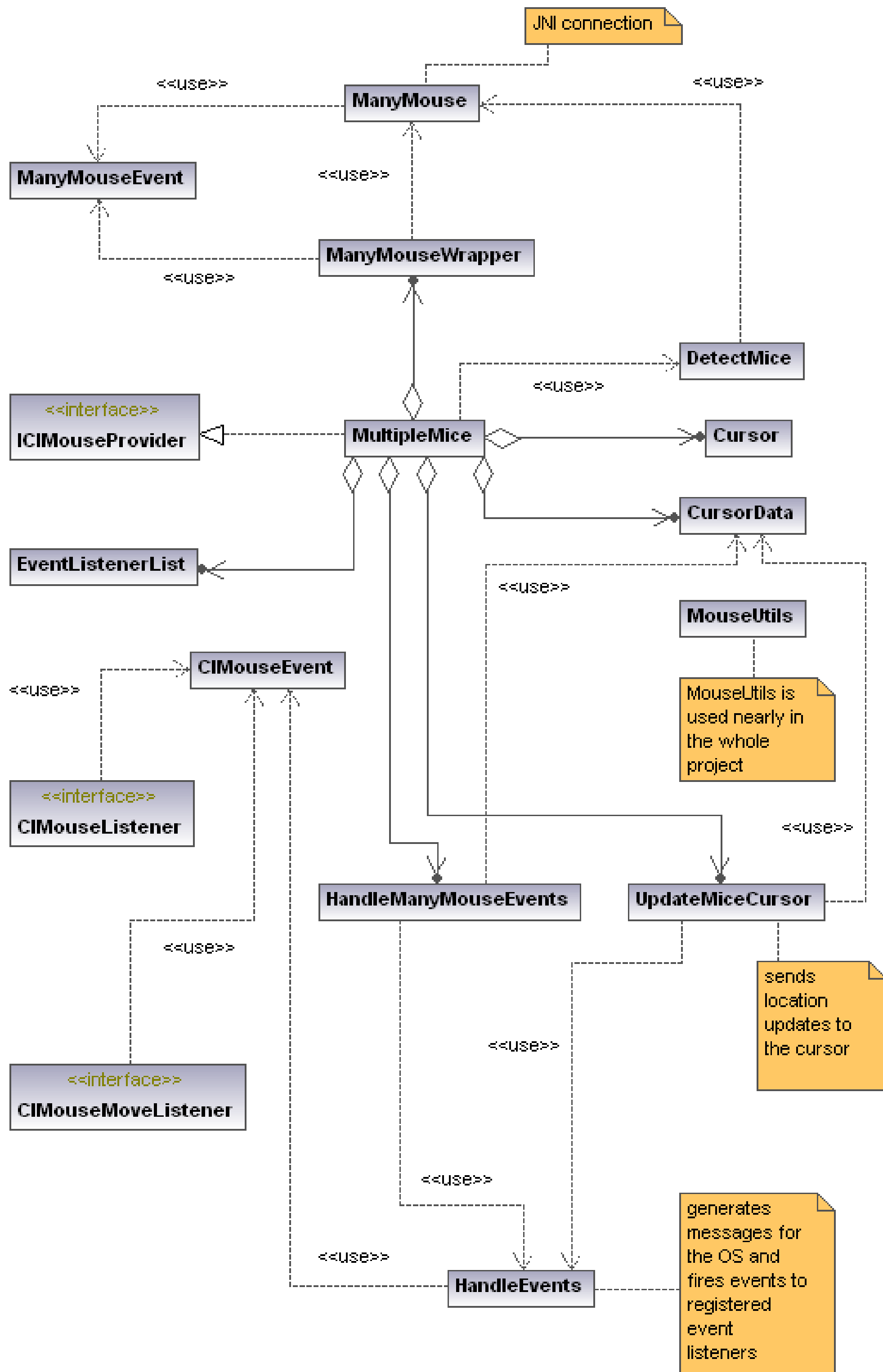


Figure 4.1. Architecture overview of the CIF

The above class diagram (Figure 4.1) does not show all the elements and dependencies of the CIF, but it can be seen as an overview of the most important classes and the structure/design of the framework. Figure 4.1 attemps to demonstrate how the classes interact with another. As you can see, the fundamental cornerstone is the "*MultipleMice*" class (see Chapter 4.1.3) which can be found in the middle of the picture. Also important to notice are the five previously mentioned parts in this project.

The ***first part*** that handles the mouse input from the operating system is on the top of this figure. The responsible classes are the "*ManyMouse"* class that enables the JNI connection, the "*ManyMouseWrapper*" class that polls as a thread the "*ManyMouse"* component and the "*ManyMouseEvent*" class that defines the event, which is send between the JNI part and the Java part of the project.

For the ***second part*** (process events) the responsible classes are the "*MultipleMice*" class itself, the "*UpdateMiceCursor*" class and the "*HandleManyMouseEvents*" class. They manage the whole interaction in the CIF by using the "*CursorData*" class and the "*HandleEvents*" class that is already the ***third part*** of the project and is responsible for generating messages for the operating system, as well calling all registered event listeners, which are registered in the "*EventListenerList*" that is located in the "*MultipleMice*" class.

The ***fourth part*** (cursor part) is located in the right part of the picture. Responsible classes are the "*Cursor*" class for the draw functionalities, the "*DetectMice*" class for initializing the right number of cursors and the "*CursorData*" class to store all needed values and the event data.

The ***fifth part*** (interface to the CIF) can be found at the bottom left corner of the figure. The two interfaces define the two possible listeners and the "*EventListenerList*" together with the "*CIMouseProvider*" interface define the connection to the CIF.

Furthermore, there is the "*MouseUtils*" class that is used in many parts of the project. It comprises static value definitions (e.g. colors, mouse buttons, etc.) and static methods, e.g. to find a widget by global (x,y) coordinates or to get the resolution of the monitor(s).

4.1.2 The CIF event process

An important part of the CIF to understand is how events are getting processed in the framework. Look at Figure 4.2 to get a basic overview of this flow of events. This picture does not represent the implemented structure in the framework. Instead of this it tries to show the event process in a more abstract way to make it easier to understand. A detailed sequence diagram can be found in Appendix C.

As you can see on the bottom of the picture, the framework has to communicate with the operating system.

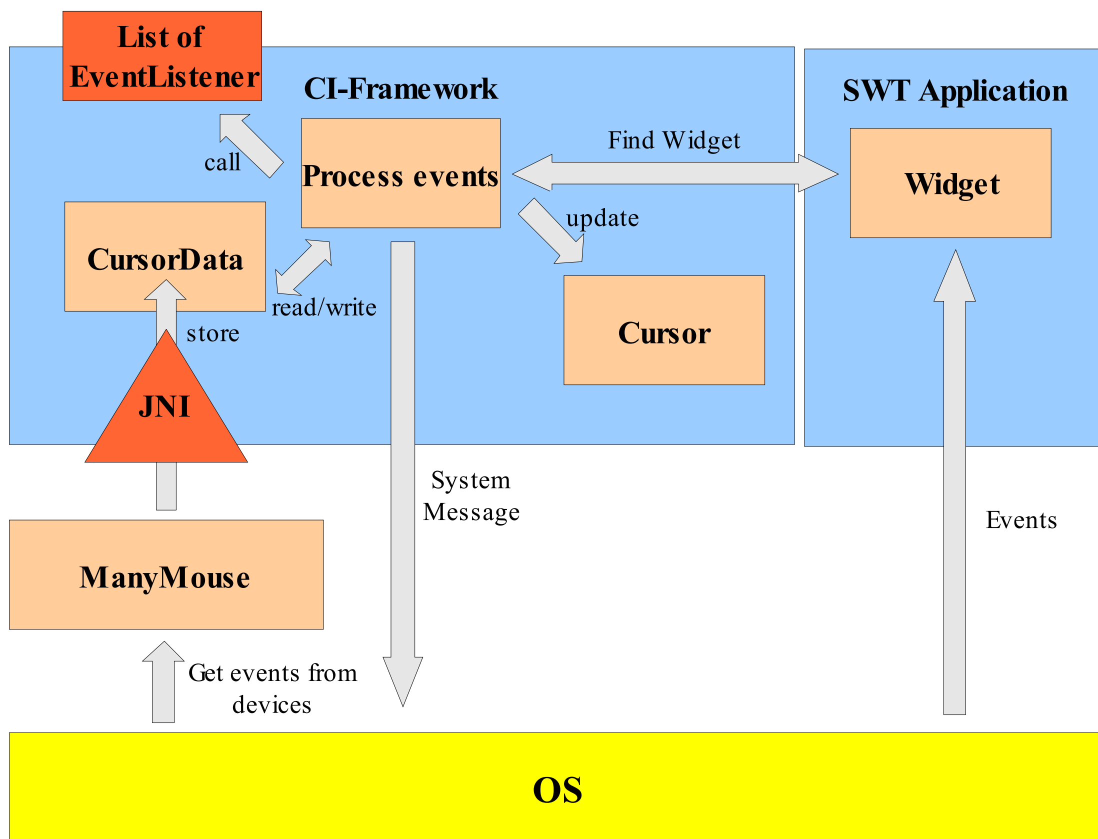


Figure 4.2. The CIF event process (Overview)

Everything starts with the native component *ManyMouse*. It is responsible for reading the Windows message queue and translating the mouse events into "*ManyMouseEvents*", which then get processed and stored in the "*CursorData*" object. The framework processes the 'collected' events and has to handle three different parts at the same time. In the upper left corner of the picture we see the list of event listeners. They have to be called if an event was triggered. In case of a

MouseMove event the cursor has to be updated in its location. The third and most complex part is the creation of a system message. For this we need the widget to get the window handle and the information regarding its location. Once all parameters are allocated the message gets sent to the operating system. The OS receives this event and creates all needed events, which it sends to the specified widget identified by the window handle.

4.1.3 The "*MultipleMice*" class

As we already saw the most important class in the CIF is the "*MultipleMice*" class. To get a little overview look into Figure 4.3. It does not show everything that is implemented in this class, but gives a whole picture of the most important parts where everything comes together. To use the framework a developer has to get the instance of this class. With this object all necessary values can be set up.

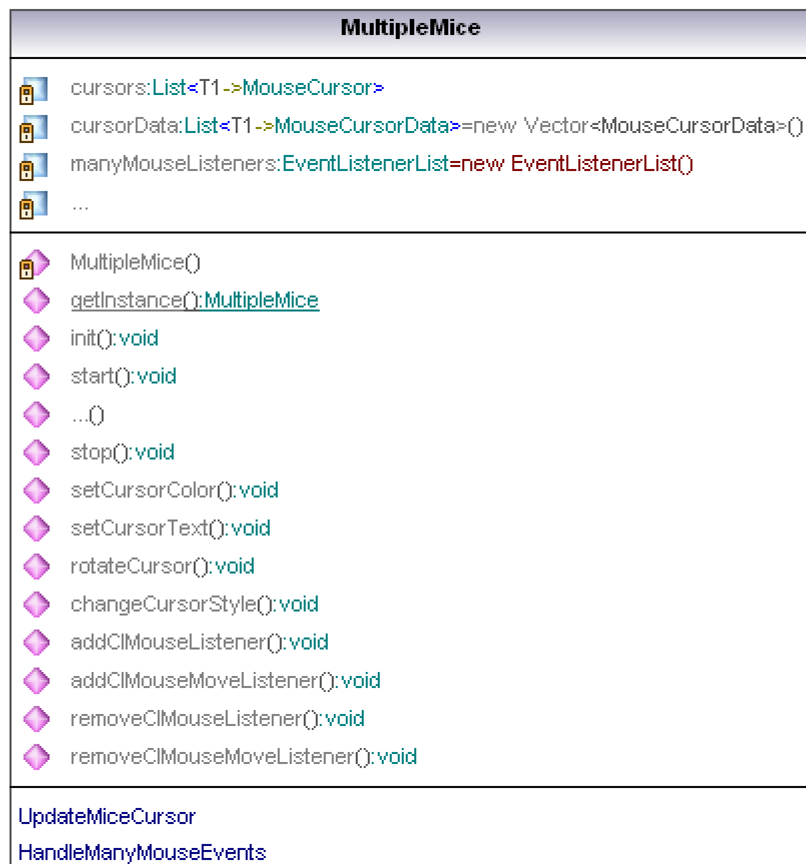


Figure 4.3. Overview of the "*MultipleMice*" class

"*MultipleMice*" is implemented as a singelton[9] to ensure that only one instance of this class is used at the same time. For an example on how this is implemented just look into the code snippets of [Jav05]. Once the "*MultipleMice*" instance is successfully initialized with the "*init()*" method it can be started. This is necessary because the framework needs to know the *display* and the current *shell* to work probably. To start the framework there are several "*start()*" functions provided. Depending on the initial configuration for the design of the mouse cursors (Figure 4.6 "*Add different colors and text to a cursor*") the appropriate method should be called. To deal with the issue of garbage collection in the Concurrent Input Support Framework the "*stop()*" method is provided. It should be called just before the SWT application gets terminated.

To configure the mouse cursors during the runtime of the framework, the following methods are available:

- ***"setCursorColor()"*** changes the color of the cursors. Colors are defined as static values in the "*MouseUtils*" class.
- ***"setCursorText()"*** adds a wanted text to a cursor.
- ***"rotateCursor()"*** rotates the cursor. The rotate directions are specified as static values in the "*MouseUtils*" class.
- ***"setCursorStyle()"*** changes all three things at the same time.

To change a cursor with one of these four functions, the cursor has to be specified by the cursor ID.

The "*EventListenerList*" has been implemented in this class as well, for providing an interface to the CIF. The "*add-*" and "*remove listener*" methods should be used to register and deregister an event listener.

The two inner classes are implementing the "*Runnable*" class of Java and are responsible for processing events. These resources are managed by the framework by itself. Furthermore, there are the two values "*cursors*" and "*cursorData*". The "c*ursors*" value is a list of the mouse cursors. It comprises the mouse cursors as

---

[9] The singleton design pattern is one of the best known patterns in software engineering. It ensures that a class has just one instance and provides a global point of access to it [Gea03].

shells. In other word in this list are the mouse cursors you see in the GUI. Since "*cursors*" are GUI elements the data gets stored separate in the "*cursorData*" list. According to the number of cursors the "*cursorData*" list has the same number of entries that are saving all data of the mouse cursors.

4.2 Getting the mouse data from *ManyMouse*

To receive mouse data from the *ManyMouse* component it was necessary first to generate the needed DLL file for the JNI connection. Once the DLL was copied into the project it was then possible to access it via the Java Native Interface. It is then relatively easy to poll the *ManyMouse* events by using the "*PollEvent(ev)*" method in a loop.

```
ManyMouse.Init() reported 2.
Mouse #0: HID-compliant mouse
Mouse #1: HID-compliant mouse
...
Mouse #1 relative motion Y axis -1
Mouse #1 mouse button 0 down
Mouse #1 mouse button 0 up
Mouse #1 relative motion X axis 1
...
Mouse #0 relative motion Y axis -1
Mouse #0 mouse button 0 down
Mouse #0 relative motion X axis -1
Mouse #0 mouse button 0 up
Mouse #0 relative motion X axis -1
```

Listing 4.1. Printout of the *ManyMouse* events

As you can see in Listing 4.1 the *ManyMouse* component returns all necessary basic data which is needed to process. Each mouse has its own ID, it returns relative coordinate changes separately for the X and Y axis. Furthermore, *ManyMouse* returns *MouseUp* and *MouseDown* events for each button.

To integrate this into the CIF a wrapper-thread had to be written. This thread calles the "*ManyMouseWrapper*", which polls the *ManyMouse* component to get the events and process them.

The way this thread processes the data depends on the event. In principal these events are sent to a list of the "*MouseCursorData"* data object, which saves the button events in an event queue along with the position of the cursor on the screen.

4.3 Handle mouse events

4.3.1 Splitting events and process them separately

After saving the mouse events in the *"MouseCursorData"* object I had to create two different SWT threads, which are responsible to process the events by using the *"HandleMiceEvents"* class. This class is the single point in the CIF where all events are getting handled. That means every event goes through the methods of this class. It is the connection between the CIF and the operating system as well as the connection between CIF and every registered listener.

To avoid a sluggish performance in redrawing the mouse cursors it was necessary to have two different threads that handle the mouse events on the SWT side. Having a separate thread for moving the mouse cursors minimizes the number of operations in this thread (see Figure 4.4).

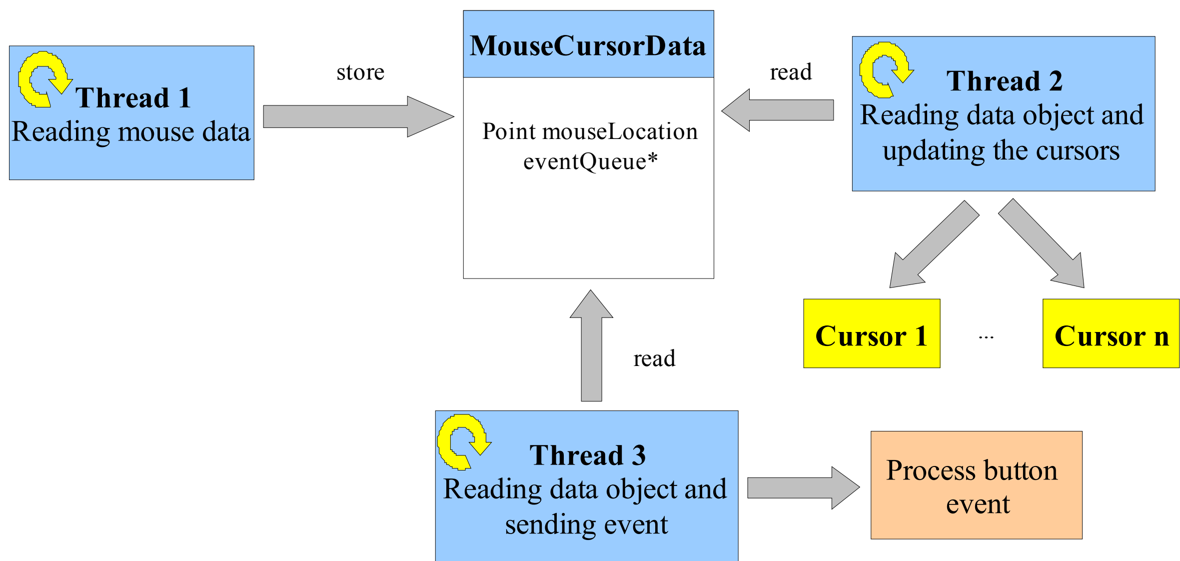


Figure 4.4. Reading and handling mouse events in the CIF

4.3.2 The thread responsibilities

As mentioned in Chapter 4.2 the first thread called "*ManyMouseWrapper*". This thread is just responsible to poll the *ManyMouse* events and store them into the "*MouseCursorData*" object for the specified mouse cursor.

The second thread called "*UpdateMiceCursor*" is not only responsible for updating the location of the manually drawn mouse cursors. The thread also updates the position of system cursor which has to be hidden at a defined, offscreen position (Chapter 4.5.3) and it has to create the *MouseMove* events for the operating system (Chapter 4.4). Furthermore, "*UpdateMiceCursor*" fires "*CIMouseMove*" events to all in the CIF registered "*CIMouseMoveListener*" (see Chapter 4.6).

Finally the third thread "*HandleManyMouseEvents*" creates the *MouseUp* and *MouseDown* events for the operating system (Chapter 4.4) and calls all listener in the CIF registered as "*CIMouseListener*".

4.3.3 Influence of the SWT thread concept in the implementation

One major point to know is that in SWT the threads are managed in a different way. Like mentioned in Chapter 3.2 SWT applications just run in one thread and it is enforced from SWT that widgets are only allowed to be used from this thread. To avoid this problem it is good to look into the thread concept of SWT before starting to implement a project.

This point influenced also the structure in the Concurrent Input Support Framework. Like showed in Figure 4.4 the wrapper thread that is polling the *ManyMouse* component had to be independent from the thread, which updates and access the user interface part of the framework (mouse cursor). Originally I planned to have just one thread that directly processes the event which it has polled from *ManyMouse*. But there are two reasons against this solution. Firstly, like I mentioned before, the thread structure of SWT. And secondly, events can be fired so fast that it becomes impossible to process them just in time. For this reason the "*MouseCursorData*" object, that persists the data between the wrapper and the UI threads, has been implemented. This data object provides beside other things the actual (x,y)-position of the mouse cursor and an event queue in that the events are getting buffered.

Knowing what I do today, I would redesign threading in the CIF by using an event listener list where event listener can be registered. But for details on this task look into part 6.1.2. "*Changing the event handling design*" in the "*Future Work*" part of Chapter 6.

4.4 Operating System Messages

4.4.1 Generate and send messages

Because of the structure and the implementation of SWT it is necessary to generate events, which are sent through the "*OS*" class to the operating system. To send the specified event the "*SendMessage (int hWnd, int Msg, int wParam, int lParam)*" has to be used.

The "*SendMessage*" function sends the message to a window. It calls the window procedure for the specified window and does not return until the window procedure has processed the message [MS07b].

But before sending the event we have to first generate the appropriate parameters. To know to which widget the event has to be sent we need the handle of the widget. The handle is something like the internal ID of this widget. Windows handles every widget as a separate window. For this reason, Windows assigns a unique window handle to every widget so the operating system knows exactly to which widget the event must be send to.

```
// get x,y coordinates
mouse_x = tempCursorData.getMouseLocation().x;
mouse_y = tempCursorData.getMouseLocation().y;

//get the widget which has to receive events
tempWidget = MouseUtils.findWidget(display, mouse_x, mouse_y);
```

Listing 4.2. Local widget coordinates has to be sent to the widget

To find out this handle on the SWT side we have to use the "*findWidget(Display display, int x, int y)*" method in the "*MouseUtils*" class with passing the current display and the global mouse coordinates. This returns the widget that 'knows' its own handle. After checking if the widget is a real SWT control and retrieving the handle for it, we can proceed with creating the (x,y)-coordinates that have to be sent as the "*lParam*" value.

Before doing that, we must know what kind of coordinate needs to be sent to the according widget. By sending the correct coordinates to the operating system, Windows will manage the event automatically. In other words Windows creates all

needed events just by sending a single message from the SWT application to operating system.

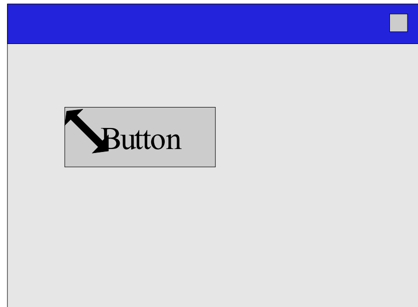


Figure 4.5. Local widget coordinates has to be sent to the widget

To generate the correct system event and send it with the "*SendMessage()*" method to the operating system it is important to find out the coordinates relative to the local widget location. Every widget knows its location relative to its parent. That means for example in Figure 4.5 to get the right coordinate, which has to be sent, has to be calculated with this equation (simplified for this example):

Searched coordinate = global cursor coordinate - shell location - button location

But a problem with this simplified example is that it could be possible for a widget to not have only a shell as its immediate parent. To solve a loop had to be implemented, which follows up the whole hierarchy to the active shell and calculates in this way the searched coordinate (Listing 4.3).

```
if (tempWidget instanceof Control) {
        tempControl = (Control)tempWidget;
        deltaX = tempControl.getLocation().x;
        deltaY = tempControl.getLocation().y;

        // calculate delta for getting the local coordinates which has
           to be passed to the widget
        tempParent = tempControl.getParent();
        while(tempParent!=null){
              deltaX += tempParent.getLocation().x;
              deltaY += tempParent.getLocation().y;
              tempParent = tempParent.getParent();
        }

        // create coords to fire the event
        event_x = mouse_x - deltaX - shellOffset.x;
        event_y = mouse_y - deltaY - shellOffset.y;
```

Listing 4.3. Calculating the right coordinates

Other values that have to be considered are the offset of the mouse rotation and the size of the title bar and the menu that are not included in the location of the shell.

Since the "*SendMessage*" requires the coordinates as the "*lParam*" parameter, which is just a single integer value, we need to know how to combine x and y.

Integer values consist of 32 bits. In case of the "*lParam*" value the upper 16 bits are reserved for the y location and the lower 16 bits are for the x location. For this reason we have to allocate "*lParam*" with the y location then shift this value sixteen bits to the left and add the x location to this result (Listing 4.4).

```
//create lParam for the OS.sendMessage - the x and y coordinates
lParam = event_y;
lParam = lParam << 16;
lParam = lParam + event_x;
```

Listing 4.4. Creating the "*lParam*" value

The next value to compile is the message value. This value is designed to specify the type event we want to send to the operating system. For the CIF, there are only three important events implemented (*MouseMove*, *MouseUp* and *MouseDown*) which are specified in the "*OS*" class as static values.

The last value to specify is the "*wParam*" value, which is only needed for *MouseMove* events to specify the button that is pressed during the movement. In case of mouse button events this value is zero.

4.4.2 Problems during the implementation

The biggest hurtle to overcome was learning how to generate messages that trigger visual feedback from the widgets. Computer users of today subconsciously expect visual feedback from computer controls. If I click on a button, the button appears to be pressed, if it does not, I assume the button is not working. To overcome this problem it was important to know something about the functionality of the "*Display*" and "*OS*" class (Chapter 3.1). To generate messages and send them to the operating system I tried several different methods. The first try was using the "*post()*" method from the "*Display*" class provided. The Javadoc of SWT says that this method can be used to generate low level system events for keyboard and mouse events. The intent

to provide such a method in SWT was to allow automated UI testing by simulating input. I assumed if I created a instance of the "*Event*" class[10], call the "*post()*" function passing this instance the event would be fired to the widget. But this did not work. It appears the post method uses the coordinates of the system mouse and not the coordinates I tried to pass with this call. That means the event gets fired at the position of the system cursor and not at the position of the CIF cursor. But by looking closer into the code it quickly became apparent that SWT uses operating system calls in this method by using the "*SendInput()*" function.

After this failed attempt, the next step was to look into this system method which is provided through the "*OS*" class. The "*SendInput()*" function synthesizes keystrokes, mouse motions, and button clicks. I thought with getting closer to the operating system sending mouse input would work. But while experimenting with this method I got the same problem with the "*post()*" method.

After reading the MSDN Library I found the "*SendMessage()*" function. This method sends messages to the operating system message queue by defining the needed parameters (see Chapter 3.4). First I tried this in a separate demo application to concentrate just on this problem. The demo application simply contained a shell with a button and *MouseDown* listener which printed out some text if the button received an event. The other application just created the system messages with hard coded values and sent these messages using the "*SendMessage()*" function to the operating system. I had success with this little prototype but I was still not fully satisfied. The button I created received the *MouseDown* events but failed to provide any feedback to the user as it did not move.

Later I found that this event is already triggered by sending a message with the correct handle. The main problem I encountered with the coordinates I just sent the global mice coordinates relatively to the upper left corner of the monitors. First I did not recognize that these coordinates were incorrect because the widget appeared to have received the events so I assumed that I had to generate the rest of the events by myself. So I analyzed these events by using *Spy++* and *SpyControl 2.0* and generated

---

[10] Instances of this class provide a description of a particular event which occurred within SWT. The SWT untyped listener API uses these instances for all event dispatching [IBM+05b].

the missing events. With these events the button moved but I was not happy with this solution. After a longer period of time and doing some other developing work I looked into this problem again and tried to change the messages in changing the coordinates in the "*lParam*" value. I had luck just using the local coordinates relative to the widget. And as a result, all the events that were needed to be sent to the widget were now being generated from the operating system. So the final solution of this problem is to generate the system events like described in the first part of this chapter.

4.5 The mouse cursors

4.5.1 Detect the mouse cursors

The framework recognizes by itself the number of mice that are connected to the computer and generates the corresponding number of mouse cursors. This happens by calling the "*start()*" method from main component of the CIF: the "*MultipleMice*" instance. This start method is responsible for calling the "*detectMice()*" function that gets the mice by calling the "*DetectMice*" component that initializes/starts *ManyMouse* through JNI by calling the provided "*init()*" function. This function returns the number of the connected hardware mice. As a result, the according number of cursors can be created as a list of "*MiceCursor*" and be returned to the main component. If the initial layouts of the mouse cursors are not the default (color, text and/or rotation of the cursor is defined initially) these values are set directly in the cursor objects.

4.5.2 Draw and update the mouse cursors

The implementation of the "*MouseCursor*" class includes everything that is necessary to draw and modify a mouse cursor in the Concurrent Input Support Framework. In principal a mouse cursor is nothing else then a GIF image drawn and shaped in a SWT shell. To avoid a sluggish performance the data is separated from the "*MouseCursor*" class. The reason for this is to be prevented from too many object accesses that read the event queue and the location to handle the events and process other parts of the CIF. The second reason for this design is that it is not possible to modify a SWT user interface element out of an external thread. For that a SWT thread has to be created which changes the location of the self drawn mouse cursor by

reading the actual location in the "*MouseCursorData*" data object and moving the appropriate mouse cursor shell to the correct location on the screen.

To be able to differentiate several mouse cursors from each other it was necessary to add some functionality to highlight every cursor with a different colour and/or a specified text like the name of the user who is using the corresponding mouse (Figure 4.6).

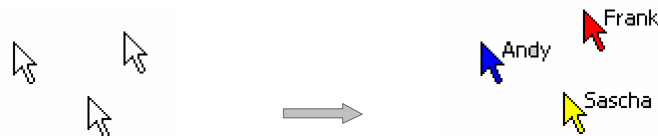


Figure 4.6. Add different colors and text to a cursor

To support conditions expected to be presented on a digital table it became necessary to be able to rotate mouse cursors depending on the position where the user stands. But rotating a cursor causes some problems with the coordinates. It is necessary to save an offset in every mouse cursor because the point where all events are going to be fired rotates with the cursor (Figure 4.7).

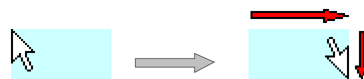


Figure 4.7. Rotating the mouse cursor – causes an offset to fire events

The picture above is the original mouse cursor picture that is used in the CIF. Usually the spot where the events are going to be fired is in the upper left corner of the picture. Which is the (x,y)-position of the shaped shell. But once the cursor has been rotated this spot moves depending on the rotation. In the figure above we have a 180 degree rotation and the spot moves to the bottom right corner. This means we get an offset for the x and y coordinates because the position of the shell, which simulates the mouse cursor, is still represented as the coordinates of the upper left corner of the shell.

4.5.3 Handle the system cursor

Intercepting mouse movements and button presses to control the CIF cursors works pretty well. But at the same time the operating system responds to these mouse activities. Movements of the mice are getting merged and the system cursor gets moved on the screen. Furthermore, the button events get passed to widgets such as buttons, window controls and menus.

The problem is that the active system cursor competes with the CIF cursors. Moving and clicking with a CIF cursor will also affect the system cursor. For example the system cursor moves to the application close button. This can cause unexpected results when the mouse button is clicked: namely the application closes.

To solve this problem I checked different possibilities and decided to focus just on one combined with filtering system mouse events (Chapter 4.5.4) in the SWT application by the framework. Important to note is that the system cursor reacts on the interaction with all mice.

- ***Hiding the system cursor behind a CIF cursor:*** Setting the position of the system mouse to the same position like the CIF cursor causes some kind of jumping of the system cursor, which does not look very good and still causes unwanted side effects.
- ***Use system mouse as first mouse and just create n-1 fake cursors:*** Because it is not possible to configure the system cursor in such a way that it just reacts on the events of one hardware mouse. The cursor jumps, which makes this configuration unusable.
- ***Replacing the image of the system cursor with an invisible GIF:*** Just changing the visibility of the system cursor does not mean that the cursor no longer sends events. It is still active and causes side effects. Furthermore, this works only inside an SWT application. The cursor becomes visible when it leaves the application.

- ***Enforce system mouse outside the monitor resolution:*** Setting the (x,y) of the system cursor to the maxima (which is the bottom right corner of the screen) does not help at all. The cursor still jumps and becomes visible and even fires events to the windows task bar.
- ***Set system cursor invisible and hide it in an unused part of the application:*** This works pretty well. But the developer of a CIF application has to ensure that the cursor has enough room to wander where it cannot cause events (because of the jumping) – huge distance to the widgets in the SWT application is necessary.

After looking in these different possibilities I decided to focus on the last point. But I was not fully satisfied and I decided to look into the "*readAndDispatch()*" method in the "*Display*" class of the SWT framework trying to filter the events fired from the system cursor in a SWT application.

4.5.4 Filter events of the system cursor

The "*readAndDispatch()*" method, provided from the "*Display*" class, reads an event from the operating system's event queue and dispatches it appropriately [IBM+05b]. As you can see in the code snippet in Listing 4.5, SWT polls the event queue with the "*OS.PeekMessage()*" method and process the message from the operating system.

```
public boolean readAndDispatchCIF(){
      ...
      if (OS.PeekMessage (msg, 0, 0, 0, OS.PM_REMOVE)) {
            if (!filterMessage (msg)) {
                  if((msg.message != OS.WM_LBUTTONDOWN)
                              ... // filter other events too
                              &&(msg.message != OS.WM_MOUSEMOVE)){
                        OS.TranslateMessage (msg);
                        OS.DispatchMessage (msg);
                  } else {
                        // do nothing
                  }
            }
            ...
      }
      ...
}
```

Listing 4.5. Filtering mouse events in the "*readAndDispatch()*" method

To filter the system cursor events it was enough to add an if-statement checking for every individual mouse event and just do nothing if the event is one of the unwanted mouse events caused from the system cursor.

But as I said in the first chapter of this thesis I wanted to guarantee the compatibility to standard SWT applications. For this reason it was not enough just to extend the "*readAndDispatch()*" method. To solve this problem the "*Display*" class had to be extended with a boolean value, called "*useCIF*", which is *true* if a CIF application has the focus on the monitor and the "*readAndDispatch()*" method had to be replaced with a tiny method, which just checks the "*useCIF*" value and calls the corresponding method. If the value is *true* the replaced "*readAndDispatch()*" method calls the modified "*readAndDispatchCIF()*" method and if the value is *false* it calls the just renamed but not modified "*readAndDispatchSWT()*" method (Listing 4.6).

```
public boolean useCIF = false;
...

public boolean readAndDispatch(){
	if(useCIF){
		return readAndDispatchCIF();
	} else {
		return readAndDispatchSWT();
	}
}
public boolean readAndDispatchCIF(){ ... }
public boolean readAndDispatchSWT(){ ... }
```

Listing 4.6. Changes in the "*Display*" class

Unfortunately performing these changes in the case of plugin development (like Agile Planner) means that the "*Display*" class has to be replaced in the SWT plugin to use the Concurrent Input Support Framework. But once this class is replaced both kind of applications, CIF and normal SWT applications, are able to run.

4.6 Provide an interface to the CIF

To guarantee the expandability of the Concurrent Input Support Framework there has to be something like an interface with that it is possible to receive mouse interaction

as events. Because of already having the device action in form of events the idea is to implement the standard event handling concept of Java. Having an "*EventListenerList*", "*EventListener*", a specified "*Event*" and a method to fire this event to the registered listener, is more or less everything that is required to provide such an interface.

In the CIF there are two different events – move events and button events. Both events are containing the same values. To differentiate them we have two different event listeners; the "*CIMouseMoveListener*" and the "*CIMouseListener*" interface.

To connect to the CIF at least one of these listeners has to be registered in the "*EventListenerList*" that is provided from the singleton instance, "*MultipleMice*", which also implements the "*ICIMouseProvider*" interface. This interface contains the "*add*" and "*remove*" functions that are used to register and deregister the event listener.

```
// initialize the CIF
MultipleMice cif = MultipleMice.getInstance(display, sShell, false);

cif.addCIMouseListener(new TestCIMouseListener());
cif.addCIMouseMoveListener(new TestCIMouseMoveListener());
...
```

Listing 4.7. Register a listener in the "*EventListenerList*"

In Listing 4.7 we see how to register a listener in the "*EventListenerList*". Everything that has to be done by a developer using this interface is to implement the interfaces of the event listener and specify in this implementation how to process the send events.

Internal structure of the event handling in the CIF:

The "*EventListenerList*" is implemented in the "*MulipleMice*" class and passed to the "*HandleEvents*" class where everything comes together. This class contains the method calls for polling the *ManyMouse* events. Once one of these events is triggered the "*fireCIMouseEvent()*" or "*fireCIMouseMoveEvent()*" function is called (depending on the received *ManyMouse* event). These both methods do nothing else

more than to create the "*CIMouseEvent*" and call the corresponding method in the event listener (depending on the event: *MouseUp*, *MouseDown* or *MouseMove*).

4.7 Using CIF in an application

To use the CIF in a SWT application there are several steps necessary. The first important step is to know which JAR file is needed for what kind of application. The second step is to know how to integrate the CIF into an application.

To create the specified JAR files there is an *ANT*[11] build available. This build script creates a last build folder, copies all needed DLLs into it and generates two different versions of the CIF JAR.

4.7.1 The JAR-Files

There are two different possibilities to develop applications with the SWT toolkit. One way is to develop a standalone Java application and the other is to develop an application as an Eclipse plugin. Usually there is a little difference between the two cases. However, developing an Eclipse plugin means that the SWT toolkit is already included through Eclipse as a plugin. In other words the CIF can be used without including the whole SWT toolkit, which is usually a part of the Concurrent Input Support Framework. For this reason it is possible to generate two different JAR files. One of these files contains the SWT and the whole CIF extension and the other is without the SWT toolkit just providing the CIF classes. Using the CIF JAR including the SWT means the framework is completely standalone. All that has to be done is to copy the DLLs, which are needed from SWT and the CIF into the project folder. Then the JAR file has to be add to the build path in Eclipse.

If you want to use just the CIF classes without SWT one step more is necessary to do. Because performing changes during the implementation of CIF in the "*Display*" class of the SWT toolkit (Chapter 4.5.4), it must be replaced in the JAR file of the SWT

---

[11] Apache Ant is a software tool for an automated software build for Java projects. Ant uses the Extensible Markup Language (XML) to describe the build process. By default the XML file is named build.xml.

plugin. In addition the “*ManyMouseJava.dll*” has to be copied into the project folder and the CIF JAR without the SWT toolkit has to be included to the build path.

4.7.2 Integrate the CIF into a standalone Java application

To integrate the Concurrent Input Support Framework into a SWT application there is just a little piece of code necessary. Everything else is managed by the framework. But before doing the coding, the environment has to be set up. First run the “*build.xml*” in the root folder of the CIF project. Copy the “*ManyMouseJava.dll*” and the SWT DLL files into the root directory of the new created project. Then after copying the “*cif-swt.jar*” (usually into a ‘library’ folder) and including it to the Java build path the code to include the CIF can be written. How to use the CIF you can see in Listing 4.8.

The first step is to get the instance of the CIF and pass the current display and the active shell. The third value which is passed specifies whether the system cursor should be shown or not. This value is more for developing purposes. Since the CIF does not work completely and is still under development, I have not deleted it.

```
// initialize the CIF
MultipleMice cif = MultipleMice.getInstance(display, shell, false);

// Creating a texts and colours for the different mice
String[] cursorTexts = {"Sascha", "Frank"};
RGB[] cursorColors = {MouseUtils.CC_YELLOW, MouseUtils.CC_BLUE};
int[] rotateDirections = {SWT.LEFT, SWT.DOWN, 0, SWT.RIGHT};

// Start the CIF; Hide system cursor in application on point
(application_X, application_Y)+(100,100)
cif.start(100, 100, cursorTexts, cursorColors, rotateDirections);

...

while (!MultipleMouseDemo.shell.isDisposed()) {
      if (!display.readAndDispatch())
            display.sleep();
}

// garbage collect CIF
cif.stop();
```

Listing 4.8. How to integrate the CIF

In the next part of the code we specify the mouse cursors. In this example two of them are configured. I chose arrays for this purpose because of the overview in the code demanded it. As a result of using arrays, and specifying them in the shown way, the developer can directly see the data he passes in. In the internal structure of the CIF this data is converted in lists which are easier to handle.

The only thing which still needs to be done is to call the "*start()*" and "*stop()*" methods. To start the CIF we have to pass some values by calling the "*start()*" function. The first two values are the x and y coordinates to define the hiding location of the system mouse cursor. This location is a combination of the shell location and the specified values. The other three values are just the definition of the mouse cursors designs.

Because of SWT handling the garbage collection manually, the "*stop()*" method has to be called at the end of the SWT application.

4.7.3 Integration into Agile Planner (Plugin)

The integration into Agile Planner is very similar but as mentioned in the introduction of this chapter there is a little difference in the integration into an application that is developed as a plugin.

The difference originates from the needed DLL files that must be copied into the project folder. Since SWT is included as a plugin and the SWT DLLs are already provided from this plugin only the "*ManyMouseJava.dll*" has to be copied into the root directory of the Agile Planner project. After this the CIF JAR file (without the SWT) has to be added to the build path. To do this, the "*Build Configuration*" in the "*plugin.xml*" (which can be found in the project root of the Agile Planner) has to be changed. The CIF JAR has to be selected for the "*Binary Build*" and has to be included as in the "*Extra Classpath Entries*".

Now the "*Display*" class in the JAR file of the SWT plugin has to be replaced with the modified CIF version of this class. This JAR can be found in the plugin folder of Eclipse. All that must be done is: a.) open this JAR file with a usual packer (e.g. WinRar) and b.) replace the "*Display*" class with the version of the Concurrent Input

Support Framework. This solution is just a workaround and cannot be the final solution for this problem.

After doing all these steps we can integrate the CIF in the way it was described in the previous chapter. The only problem in doing this is finding the right place for the code that calls the CIF. In the case of the Agile Planner, it was not easy to find a point where the current shell is available. The call of the CIF happens in the "*ApplicationWorkbenchWindowAdvisor*" class, which is responsible for initializing the whole workbench. The call of the CIF happens through a method added in this class. The method call happens directly after the editor of the application is created. Furthermore, there is a check for a static value called "*useCIFSetup*". This value is just a configuration flag which is set to *true* if the CIF should be activated for the Agile Planner.

Since the "*ApplicationWorkbenchWindowAdvisor*" class has a de-constructor for garbage collection it was easy to include the "*stop()*" call for the CIF into this function.

Chapter 5. Problems

The allotted time for developing this framework was just two months, so the whole project does not constitute a fully completed framework. Rather, it is more like a prototype and there are still some open problems left. My efforts were concentrated on solving the main problems: to have a framework that is usable and with which it can be demonstrated that it is possible to have concurrent input in SWT applications.

Because I was unable to solve all of the open problems, I describe them in this chapter first. Wherever I have a proposal for a solution I try to give advice on how to solve this problem directly after its description. Surely I cannot guarantee that this proposal is the right solution for this problem; it is more like a suggestion or an idea in which direction I would spend more time researching the right solution. Even if it is wrong I think this advice is useful to have as a starting point.

The entire project developed for this thesis can be split into two different parts. One part is the framework (CIF) itself and the other part was the integration of the framework into Agile Planner. For this reason I distinguish in this chapter between these two parts. There are also parts that overlap. In such cases, I reference between these two parts but I include it in that part of the development phase in which I recognized the issue.

Some of the proposals deal with the integration of CIF in the *lg3d-swt* project. In the future is planned to combine the CIF with the *lg3d-swt* project. To get more details about this project look into Chapter 6.1.3 "*Combining the CIF with lg3d-swt project*" or read [Hoe07].

5.1 Open problems in the CIF

5.1.1 The Title bar

The functionalities that are usually available with the title bar on the top of the window do not work. It is not possible to move the window or to use either the buttons in the upper right corner (close application, maximize the window and minimize the window). The reason for this is that this part of the window is not included in the shell[12] itself. It is the frame around the shell that is created by Microsoft Windows XP™. That means it is not a part of SWT. I was not able to find out how to send the events out of the framework to this 'part' of the window.

A workaround for this problem would be to compare the size of the title bar and organize a virtual surface on top of it. Clicking this bar with the CIF cursors still returns the handle of the shell. Converting the global mouse coordinates into coordinates relative to the window would return negative y-values. Comparing these values with the afore-mentioned virtual surface would enable recognition of the position of the mouse cursor. Knowing this, it would be possible to generate the usual window actions in this bar by using SWT shell methods. There is a "*shell.setLocation()*" for moving the window, "*shell.setMaximized()*" and "*shell.setMinimized()*" to maximize/minimize the window and "*System.exit()*" to close the application.

5.1.2 The Menu bar

The menu bar of the window does not work. I suggest the reason for this is that this widget is not a control. The function that searches and returns the widget using the global coordinates relative to the upper left corner of the monitor can only handle controls. When I generate a menu and click on it the CIF recognizes the shell underneath that menu and returns the wrong handle. That means all generated events are send to the wrong widget.

---

[12] The shell in SWT is the window of the application

My proposed solution is to have a look into the "*findWidget()*" method in the "*MouseUtils*" class and extend it. Once it can handle more than just controls, the correct handle can be added to generate the system message. But be careful if this works, the next problem would likely be the drop down menu, which causes another problem that is similar to the problem with the combo box widget (Chapter 5.1.5). If this does not work, another solution would be to implement it in the same way as for the title bar. Once it can be recognized that the cursor is in the area of the menu bar, it would be possible to get the handle by using the "*shell.getMenuBar()*" function that returns the "*Menu*" instance.

5.1.3 Offset of the Title bar

As I said in Chapter 5.1.1, the title bar is not included in the shell of the application. If you want to calculate the coordinates for generating system events for a widget, you have to do this relative to its own coordinate system. For example, for a shell this means (0,0) is the location of the shell relative to the upper left corner of the monitor. That means that you have to create an offset for this title bar. Since Windows supports different window skins (Windows classic and Windows XP style) this offset can be different across styles. This is just a matter of pixels but it can cause the user discomfort during use if not properly adjusted for.

I tried to solve this problem by finding a Windows function that returns at least the operating system skin that is currently in use. But I have not found anything like this. Furthermore, I think such a method would help in case of the "*The Title bar*" (Chapter 5.1.1) issue.

5.1.4 The Scroll bar

The scroll bars that are generated of some widgets (e.g. textboxes) do not react on events sent from the CIF cursor. I think this problem is similar to the menu bar issue. It has something to do with the handle. The scroll bar widget is not extending the control and for this reason it is not returned from the "*findWidget()*" functions. I assume once if this problem is solved and the right widget is returned the scrollbars will work. So my suggestion in this case is to look into the "*findWidget()*" method and extend it.

5.1.5 The Combo box

In general, there are no problems with the combo box itself. It reacts on clicking and opens the selection menu. However, the CIF mouse cursor ends up behind this little menu window. So it is not possible to choose anything from this menu and it is not possible to send events to this window. Upon closer inspection, you may recognize that the window has its own handle on system side. But in SWT there is just one handle available. In other words, this little window is generated by the operating system and lies on top because it is the active 'application' at the moment. Since the CIF mouse cursors are nothing else than a part of the SWT application the cursors are in the background if a dialog box or even this selection menu is on top.

This issue can only be solved by handling it very closely to the operating system. That means that a way has to be found to generate a mouse cursor like a usual system mouse cursor. But in the case of combining the CIF with the *lg3d-swt* project, this has to be managed differently. I know that there is a problem with this combo box, too. But if we would generate mouse cursors in this framework as 3D objects, they would end up on top of this selection menu. That does not solve the whole problem. The issue with the different handle would be still there and sending system messages to this window would not work. But it would be a step into the right direction.

5.1.6 The Textbox

Clicking in a textbox with the right mouse button usually opens up a selection menu that can be used to, e.g. copy and paste. This window is generated by the operating system and causes the same problem as does the window of the combo box. That means that the CIF cursor cannot be used to select any of its entries. To solve this issue I believe there is only one solution. The CIF cursors have to be created as normal cursors. I am not sure if there is something like a "*createCursor()*" method in the Windows API, but this would be the first thing I would look for. I am not sure how this is solved in the *lg3d-swt* project. However, I think that they simply do not provide this functionality by using the right click of a mouse. Another issue is the focus of the textboxes. If a textbox is active, ready to write something and someone clicks with another CIF cursor, the textbox loses the focus.

I have no idea how to solve this problem yet, but I think that in the case of dealing with this problem it would be easy to look into an extension of the CIF framework. I think it would be relatively simple to include multiple keyboard input into the framework on the same time. To get a starting point for this issue, it would be the best to look into the SDGToolkit (Chapter 2.4) and talk with the developer of this framework, if at all possible. As I know they already implemented multiple keyboard support.

5.1.7 Double click

The double click is still not implemented in the framework. It is planned and parts are already prepared so that it can easily be added. But since *ManyMouse* returns only *MouseUp* and *MouseDown* events and not a complete *DoubleClick* event it is not supported yet.

To implement the double click you have to look into the "*getDoubleClickTime()*" function of the "*Display*" class. It returns the longest duration (in milliseconds) between two mouse button clicks that will be considered as a double click. With this information it would be possible to watch the events and replace them with a double click event.

In the case of combining CIF with *lg3d-swt*, this may not be necessary if it is possible to use the event system of the Looking Glass project, which already recognizes double clicks. If it is not possible you have to go back to the afore-mentioned solution.

5.1.8 The Tab folder problem

Using a tab folder widget in SWT means that there are widgets added to each tab. The problem is that these widgets are technically on top of each other. That causes a problem with the "*findWidget()*" method. The function always returns only the first added widget. That means that by clicking one of the other widgets on a different tab, the generated events are always sent to the wrong widget.

To solve this problem, the method has to be extended through widget overlay functionality. The method must recognize that there is more than one widget on the

same coordinate and has to check what kind of widget is on the top. Once the right widget gets returned, everything else should be managed from the CIF.

5.1.9 Side effects caused by the system cursor

Pressing a mouse button generates side effects caused by the system cursor, which I am not able to explain. Usually all events sent from the system cursor are filtered in the "*readAndDispatch()*" method. Especially marking a text in a textbox does work, but only in a sluggish way. You have to move the mouse cursor very slow. Then it is possible to mark some text. I analyzed this problem by using *Spy++*[13] to find out what kind of events are sent to the widget. I realized that if a mouse button is pressed, the system cursor generates *MouseMove* events that influence the whole system. These events are not passing the "*readAndDispatch()*" so it is not possible to filter them out.

To get rid of this problem I suggest looking for a method to deactivate the system cursor in the MSDN Library. In the case of combining CIF with *lg3d-swt*, this issue would not be a problem at all. I had a look into that and it seems to me that they replace the system cursor with the 3D cursor object. I think they deactivate the system cursor somehow. So, if these two frameworks will be combined this problem will be fixed.

5.2 Open problems in Agile Planner integration

5.2.1 The Menu bar

The menu bar does not work. This is the same problem already described in Chapter 5.1.2. Once the "*findWidget()*" method in the CIF framework is extended and works also with other widgets, not only with controls, that problem should be solved directly in Agile Planner as well.

[13] see Chapter 3.10.1

5.2.2 Upper button bar

The button toolbar directly under the menu bar does not work at all. This is very surprising because the buttons on the left side worked well. So I ruled out that this has something to do with the GEF framework that is used in Agile Planner. I looked deeper into the problem and found out that these buttons are not GEF or SWT components; they are created by using JFace[14] and it looks like these components do not react on system events like the pure SWT widgets or the widgets of the GEF. To solve this issue, there are two possibilities. The first would be changing these buttons into real SWT buttons. This would be the easiest way. However, usually JFace is used in many SWT applications, so I think it would be better to look into the JFace component and find out how these widgets work and receive events.

5.2.3 System mouse cursor

After the integration into Agile Planner, the "*hideSystemCursor()*" function does not work anymore. This method generates a transparent image and replaces the system cursor with this picture as long as the cursor is inside the shell. The other part is to force the system cursor onto a point inside of the shell (Chapter 4.5.3).

It seems it has something to do with the way the Agile Planner is implemented. I tried to add the "*hideSystemCursor()*" method directly into Agile Planner and tried to replace the passed shell with the editor that sits on top of the shell, with no success. To solve this problem I assume it would be best to follow this point as maybe I did something wrong or I overlooked something. But a second way could be spending some time on researching how to deactivate a system cursor using an operating system method call.

---

[14] JFace is a UI toolkit that provides helper classes for developing UI features. It is a layer that sits on top of the raw widget system, and provides classes for handling common UI programming tasks. JFace is completely dependent on SWT. Furthermore, the Eclipse Workbench is built on both JFace and SWT; in some instances, it bypasses JFace and accesses SWT directly [Wik07].

5.2.4 Concurrent input

The standard widgets of SWT and GEF do not support real concurrent input. The events are fired as system events that do not support an identification of the mice. In the case of a simple widget like a button, this might not matter. But if you look into this problem deeper you will notice already in simple scenarios that there could be problems with a multiple mouse input. Usually, using widgets with just one mouse cursor happens very intuitively. But if several users start to use this application with concurrent input at the same time and the widgets are not really supporting such concurrent input, the whole system reacts in a different way than the user would assume.

For example, if a textbox has the focus and a user tries to write something in it, it would lose this focus if another user clicks a button or a different textbox in this application with a second mouse cursor. The reason for this is that Windows does not support concurrent input at all and the widgets, which are provided, are also implemented just for a single input system.

In Agile Planner this problem becomes very obvious: when a user moves a story card with the first mouse cursor and a second user just moves another mouse cursor over the editor the card jumps between these two mouse cursors. It just reacts on the sent *MouseMove* events and it cannot differentiate that the second input is from another mouse cursor.

A solution for this problem is not so easy to implement. Most likely, all standard widgets have to be replaced with extended widgets that support concurrent input. I do not think that this is possible with the usual Windows system messages. I believe the event handling in SWT has to be replaced with an event system that works similarly to the event handling of the MID project (Chapter 2.3). But this would cause a lot of work in rewriting the widgets so that they work with such an event system. Furthermore, SWT applications that should use concurrent input support have to be rewritten too because of the modified widgets. That means it would be more effort to integrate the CIF into an already existing application.

However, especially for the editor widget of the GEF, I can think of a second solution. The widget has to be rewritten, too. But this solution deals with a dispatcher. This

dispatcher receives the usual system messages and identifies the mouse cursor by using the coordinates of the event it has received, and manages the interaction of the widget itself by using an internal event system that creates mouse specified events, which the widget is able to interpret. Furthermore, the widget has to manage the access to the single graphic objects. For example, in Agile Planner the editor has to know exactly what kind of element (e.g. story card) gets the events of a specified mouse cursor. As long as this element receives events from this cursor other mouse cursors should not be able to interact with this card. In other words this means that parts are usable just with one mouse cursor at the same time but the whole widget interacts with concurrent input.

Chapter 6. Final Remarks

This chapter concludes the thesis. First, I give an overview of the next steps that have to be done. These steps deal with solving the open problems exemplified in Chapter 5. I also give a little introduction in how to combine the CIF with the *lg3d-swt* project.

Then I compare the goals I had in the beginning with the actual completed work. Especially in the face of the future work I broke one goal. I suggested modifying the standard widgets to be able to handle real concurrent input (see Chapter 1.3 the first goal). This does not match with the original goal. But I do not think we can get a final and working solution of the framework without touching the standard widgets of SWT.

6.1 Future Work

Since there was not a lot of time to implement this framework, I focused my work on the most important parts:

- Generating mouse cursors
- Getting events of each device
- Handling these events and generating system messages for the operating system.

That means the result is not the final solution that works perfectly in detail. It is more like a prototype and there are a lot of open issues left. Furthermore, the framework should be extended and combined with other components. Later the framework should work together with the 'window rotation' project *lg3d-swt*, and it should have support for mouse gestures. I think the following steps are the most important things to do in future.

6.1.1 Solving open problems

As you can see in Chapter 5, there are a lot of open problems that should be solved first before extending the framework with other components. I think they have different priorities and the most important point is to look into the "*findWidget()*" function in the "*MouseUtils*" class and extend it. Once it works also with other widgets then only controls and it can manage overlaying widgets (see Chapter 5.1.8 "*The Tab folder* problem") that would solve and simplify several issues. The second step could be the offset issue (see Chapter 5.1.3 "*Offset of the Title bar*"). If there is a system method provided that identifies the skin that is used in Windows, some code in the CIF could be refactored. After that the event handling design should be refactored as well. It is a huge performance issue for the whole system that the "*MouseCursorData*" object gets polled. For details, see Chapter 6.1.2. Once this is done, combining CIF with the *lg3d-swt* project should be next. As we saw in Chapter 5, some things (especially the cursor) are handled differently in the 3D framework and some of the described issues would be solved or would look different. In addition, since one of the future project goals is to have a combined framework, this step should be done as one of the next. For details, see Chapter 6.1.3. In the course of combining the two frameworks I suggest integrating the table data component as well.

Once the two frameworks are combined and the table events are handled correctly, the hardest issue will be the solution for the real concurrent input. This can be a whole project in itself: rewriting widgets and implementing a dispatcher for managing the concurrent input with a self-specified event system for the communication between the 3D and the 2D surface.

6.1.2 Changing the event handling design

As we know from Chapter 4.3.1 and 4.3.3, the "*MouseCursorData*" object gets polled to process the event in the message queue and to update the location of the mouse cursors. This cost a lot in performance time and it would be better to refactor this part of the project and implement an event handling system by using an "*EventListenerList*" and "*EventLister*" that is called if a new event occurs.

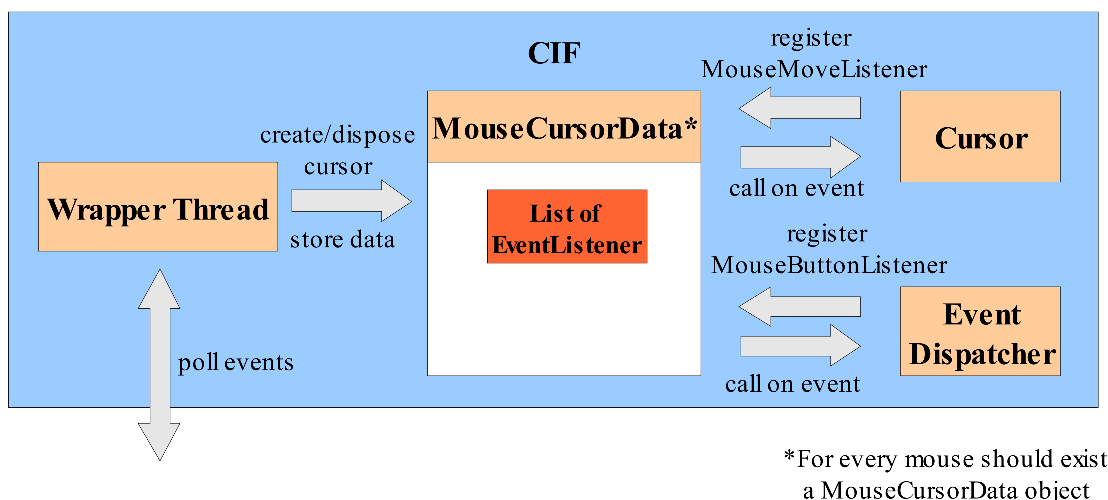


Figure 6.1. Event handling system

The best solution would be to add an "*EventListenerList*" and the needed functions to fire events and to add/remove "*EventListener*" in the "*MouseCursorData*" object. Surely, it could be implemented directly in the "*ManyMouseWrapper*" class but in case of adding table data it would be good to have a single point where all the data comes together. It has to be ensured that connecting and disconnecting mice is managed at least from the wrapper thread. Especially in the case of adding the events of the DViT from the table this will be an issue that should be handled (see Chapter 6.1.4). If a new mouse gets connected, a new cursor and a new data object have to be created and in the case of disconnecting the mouse, this object and especially the cursor that is a GUI component have to be disposed.

The cursors have to be registered in the "*EventListenerList*" with their IDs to identify and call the corresponding listener to send events to the specified mouse cursor. In other words, only the event listener of the mouse cursor that is to be updated should be called, to avoid overhead. Even in this design for handling events it would be good to differentiate between *MouseMove* and *MouseButton* events. So there should be a separate event listener for button events. With this listener the component that is used to create system events and later the event dispatcher for the button events can be registered in the same "*EventListenerList*".

6.1.3 Combining the CIF with lg3d-swt project

Combining the two frameworks should be relatively easy to do. The Concurrent Input Support Framework definitely cannot be integrated in as easy a way as into a normal SWT application.

The way the *lg3d-swt* project works is that it captures the GUI of a SWT application as an image. This picture is used as a texture to be mapped onto a 3D shape in the Looking Glass[15] project. This texture gets replaced whenever the image of the 2D application changes. To receive events in the SWT application, the events have to be mapped correctly from 3D to 2D [Hoe07].

How to combine both frameworks?

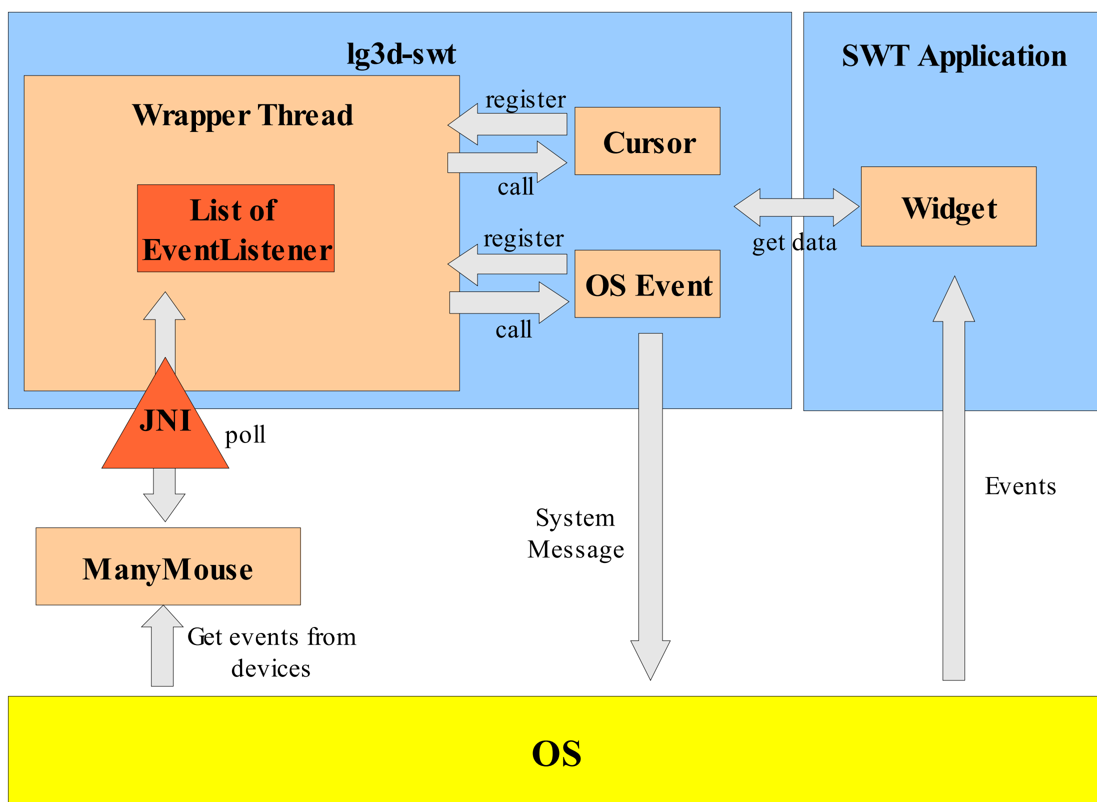


Figure 6.2. Combining CIF with *lg3-swt*

---

[15] Project Looking Glass is developed by Sun Microsystems. It is an open platform to explore an innovative 3D user interface [Sun07]. The *lg3d-swt* framework uses this project as its base and Sascha Hömig extended this framework with SWT support.

For combining both projects, the mouse cursor part has to be designed as new. In the 3D framework a mouse cursor is a 3D object, so it is not possible to take the mouse cursor part of the CIF and integrate it into the Looking Glass project. To get this working, this part has to be implemented from scratch. The first step would be embedding the *ManyMouse* project in the 3D surface. This part would be similar to the integration of *ManyMouse* in the CIF. First, a wrapper thread should be implemented. This thread should have an "*EventListenerList*" and methods that call all registered "*EventListener*" whenever an event gets polled by the wrapper. The second step is implementing the mouse cursors as 3D objects. For this I suggest looking into the already existing implementation of the Looking Glass project. Furthermore, to support the functionalities like changing the color, adding text or rotating the cursor in the 3D surface, have to be implemented again.

The event handling part of the CIF should work directly. As far as I know, the *lg3d-swt* sends mouse and keyboard events as system messages from the 3D platform to the operating system by using the "*SendMessage()*" function. In other words, this part is implemented in the same way as in the Concurrent Input Support Framework. For processing the multiple events, this means all "*ManyMouseEvents*" have to be translated into events that are used by the *lg3d-swt* platform.

Once these parts are implemented, the actual state of the project would be covered, and the multiple mice would be managed completely on the side of the Looking Glass platform. This implies that there is no CIF code anymore that has to be instantiated in the 2D SWT application.

But if there should be any real concurrent support the widgets have to be touched and modified. Since the system messages do not provide mouse cursor identification, the interface between the 3D and 2D has to be replaced too. On the side of the 2D application there should be something like a receiver component that handles events defined especially for this purpose. These events should have a value for identification of the mouse cursor. This receiver hands the events over to a component that processes these events and communicates with the modified SWT widgets. That should be something like the solution proposal for the "*Concurrent input*" issue in Chapter 5.2.4.

6.1.4 Integration of the tabletop data

Later to support more than just the hardware mice that are connected to the computer the data of the table, the Digital Vision Touch (DViT), has to be integrated into the CIF too. This DViT surface enables the user to use their fingers as mouse cursors. The events that are fired are mouse events including an ID for different 'devices'. In other words the surface provides multiple mouse input. The company SMART Technologies Inc., which developed the digital table, provides different interfaces (for different development technologies) to connect to their surface with the SMART Board SDK. Such an interface is necessary because the DViT device does not use the Windows message queue to send its events. The reason for this is that the message system in Windows does not provide any values to differentiate several mouse cursors. So they also handle their input separately in their table applications. As far as I know, the DViT device uses for this the Component Object Model (COM) that enables a client/server communication between the surface and a developed application. This works fine with the .NET framework. They also provide a JAR file that can be included in a Java application, but the Java support is not very good, especially as there is no support for SWT. For this reason, there is a project dealing with this issue in the EBE group. The goal is to adapt the COM object and provide all of the basic functionalities for Java.

Once the connection to Java works it should not be too hard to integrate the data that is returned by the DViT surface. All that has to be done is to translate the events into the CIF event format. Then the CIF can access and process this data for each mouse separately. In Figure 6.3 we see an overview of how to integrate and manage the tabletop events in the Concurrent Input Support Framework.

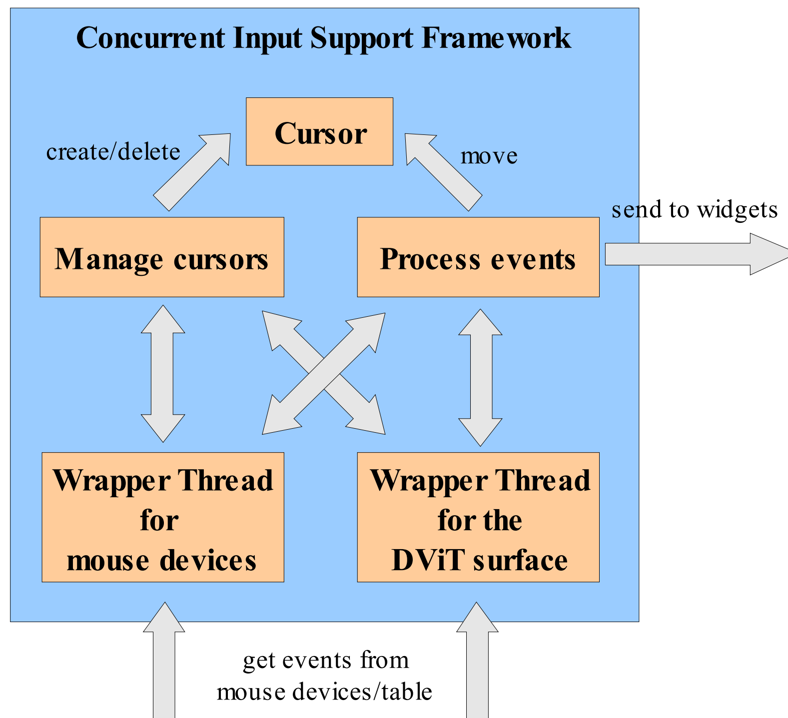


Figure 6.3. Adding table events to the CIF

But how should the framework handle the mouse cursors?

This will be an upcoming issue. The problem in the case of using fingers as a 'mouse device' is that there is no hardware device that can be clearly identified. Furthermore, if the finger is gone from the surface there is no mouse anymore, so the shown cursor should be disposed of. In other words, there should be another component in the CIF that manages this connect and disconnect issue of the mice. It should create a mouse cursor if a finger touches the table surface and it should dispose of the cursor if it is not needed anymore.

6.2 Conclusion

This diploma thesis has presented the Digital Tabletop Concurrent Input Support Framework. The implemented framework supports several hardware mice to use in SWT applications with multiple mouse cursors, one for each mouse. The state of the project should be described as still under development. There are some open issues that have to be resolved in future. This thesis gives an overview of the framework and describes the technologies that have been used as well as open problems.

Furthermore, it gives advice for continuing with the development and adding other components to it.

This thesis has presented some first steps on a long road of developing a good working framework. However, there are more steps to do and some of these will change the goals from the beginning of this thesis.

I have created a framework that support multiple mice inputs and can be used easily. The mouse cursors are automatically managed from the framework. There are methods to change the layout and the orientation of the mouse cursors and an interface to connect to the CIF for getting basic mouse cursor events is available as well. A developer has nothing more to do then including/calling the framework with some simple lines of code. At the moment it works with SWT standard widgets. No widget has to be changed. But as I said before, this goal cannot be kept in future. I do not think that it is possible to support a real concurrent input without touching the standard widgets because they do not work concurrently by default. We saw this problem appearing especially during the integration of CIF in Agile Planner (jumping story cards between the cursors) but I think with some more time effort these problems can be solved in future.

References

[Ass05] Ramin Assisi (2005), ***Eclipse 3 Java Entwicklung mit der Open-Source-Plattform***, Hanser, ISBN: 3-446-22865-9

[Bro+03] David Brownell and others (2003), ***jUSB: Java USB***, http://jusb.sourceforge.net/?selected=about
Viewed: 13th January 2007

[Dau05] Berthold Daum (2005), ***Java-Entwicklung mit Eclipse 3. Anwendungen, Plugins und Rich Clients***, D-Punkt, ISBN: 3-89864-281-X

[Ecl07] The Eclipse Foundation (2007), ***What is GEF?***, http://www.eclipse.org/gef/overview.html
Viewed: 1st February 2007

[Gea03] David Geary (2003), ***Simply Singleton - Navigate the deceptively simple Singleton pattern***, JavaWorld.com, http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html
Viewed: 13th January 2007

[Gor05] Ryan C. Gordon (2005), ***ManyMouse***, http://icculus.org/manymouse/
Viewed: 2nd January 2007

[HB99] Juan Pablo Hourcade, Benjamin B. Bederson (1999), ***Architecture and Implementation of a Java Package for Multiple Input Devices (MID)***, HCIL Technical Report No. 99-08, http://hcil.cs.umd.edu/trs/99-08/99-08.pdf
Viewed: 13th January 2007

[Hoe07] Sascha Hömig (2007), ***Feasibility Study and Prototypical Implementation of a Window Framework for Digital Tables***, Diploma Thesis at the Department of Computer Science, University of Applied Science Mannheim, Germany, in cooperation with the University of Calgary, Canada

[IBM+05a] IBM Corp. and others (2005), ***Event (Eclipse Platform API Specification)***, http://help.eclipse.org/help31/nftopic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/Display.html
Viewed: 10th January 2007

[IBM+05b] IBM Corp. and others (2005), ***Display (Eclipse Platform API Specification)***, http://help.eclipse.org/help31/nftopic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/widgets/Display.html
Viewed: 4th January 2007

[Jav05] Javangelist (2005), ***Singleton Pattern***, http://www.javangelist.de/space/Singleton+Pattern
Viewed: 13th January 2007

[Lia99] Sheng Liang (1999), ***The Java™ Native Interface Programmer's Guide and Specification***, Addison-Wesley, ISBN 0-201-32577-2

[MDG+04] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden (2004), ***Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework***, IBM Redbooks, ISBN: 0-738-45316-1
Download: http://www.redbooks.ibm.com/abstracts/sg246302.html?Open

[Mee06] Marco van Meegen (2006), ***Diagramme selbst gemalt Draw2d: das Diagramming-Framework von GEF***, Eclipse Magazin Volume 4

[MM06] R. Morgan, F. Maurer (2006), ***MasePlanner: A Card-Based Distributed Planning Tool for Agile Teams***, Proceedings International Conference on Global Software Engineering (ICGSE 2006), IEEE Computer, 2006.

[MS07a] Microsoft Corporation (2007), ***MSDN Library: Messages and Message Queues***, http://msdn2.microsoft.com/en-us/library/ms632590.aspx
Viewed: 4th January 2007

[MS07b] Microsoft Corporation (2007), ***MSDN Library: SendMessage Function***, http://msdn2.microsoft.com/en-us/library/ms644950.aspx
Viewed: 4th January 2007

[MS07c] Microsoft Corporation (2007), ***MSDN Library: ControlSpy v2.0***, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/commctls/controlspy20.asp
Viewed: 6th January 2007

[Ral06] Rally Software Development Corp.(2006), ***The Agile Planner***, http://www.rallydev.com/agile_planner.jsp
Viewed: 2nd January 2007

[Ram03] Shantha Ramachandran (2003), ***Basic SWT Widgets***, Department of Computer Science, University of Manitoba, Canada
http://www.cs.umanitoba.ca/~eclipse/2-Basic.pdf
Viewed: 15th January 2007

[Rit02] Simon Ritchie (2002), ***SWT - The Standard Widget Toolkit***, http://www.tucson-jug.org/presentations/SWT.pdf
Viewed: 5th January 2007

[Sta03] Michael Stahl (2003), ***Java USB API for Windows***, Diploma Thesis at the Institute for Information Systems, ETH Zürich, http://www.steelbrothers.ch/jusb/api/usb/windows/related-docs/JavaUSBforWindowsWeb.pdf
Viewed: 13th January 2007

[Sun06a] Sun Microsystems, Inc. (2006), ***Java Native Interface Specification***, http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html
Viewed: 2nd January 2007

[Sun06b] Sun Microsystems, Inc. (2006), ***The JavaTM Tutorials: Laying Out Components Within a Container***, http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html
Viewed: 31st January 2007

[Sun07] Sun Microsystems, Inc. (2007), ***Project Looking Glass***, http://www.sun.com/software/looking_glass/
Viewed: 19th January 2007

[TG02] Edward Hiatt Tse, Saul Greenberg (2002) ***SDGToolkit: A Toolkit for Rapidly Prototyping Single Display Groupware***, Poster in ACM CSCW '2002 Conference on Computer Supported Cooperative Work, http://grouplab.cpsc.ucalgary.ca/papers/2002/02-sdgToolkit.poster.CSCW02/sdgToolkit-Poster.CSCW02.pdf
Viewed: 14th January 2007

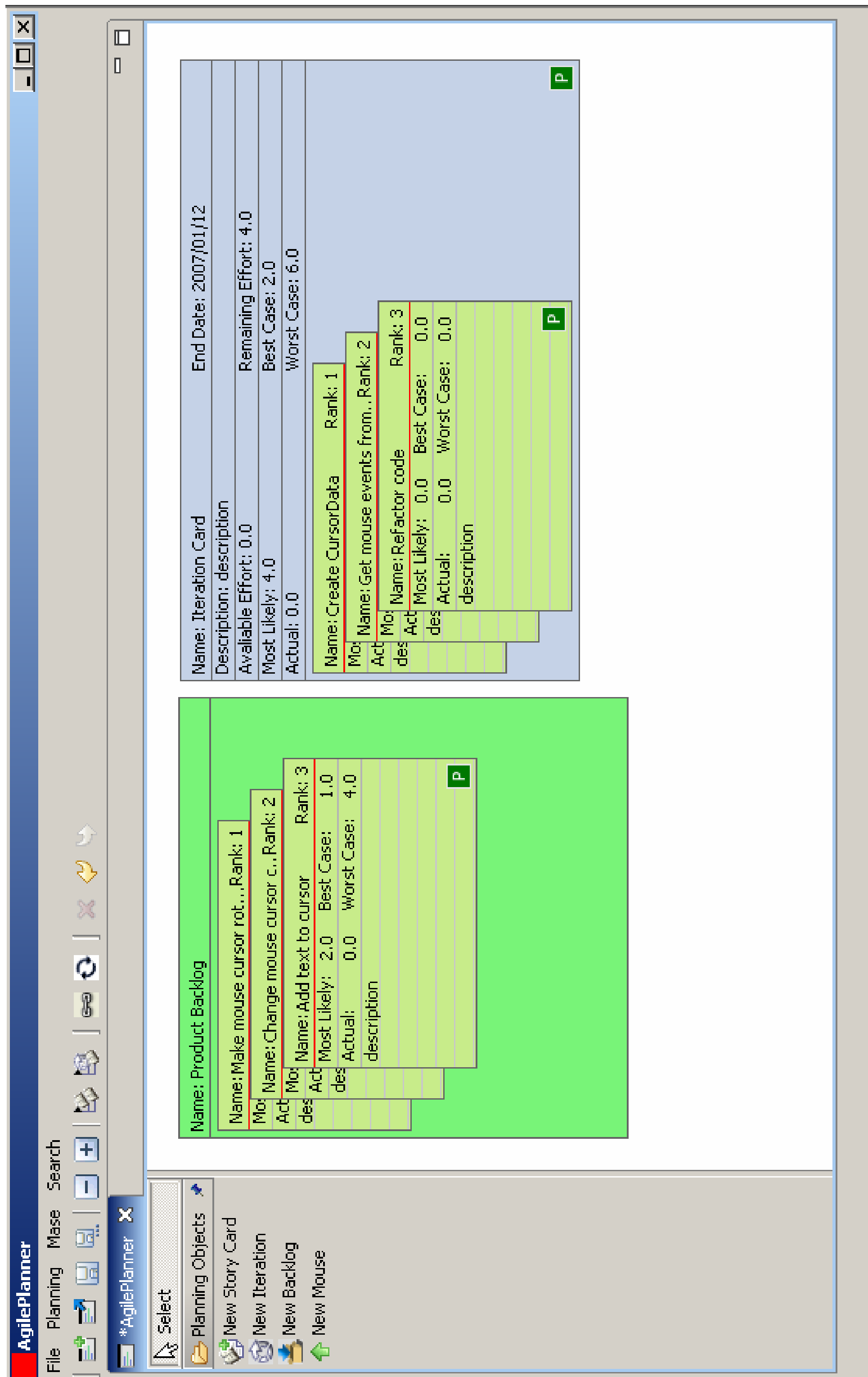
[Tse04] Edward Hiatt Tse (2004), ***The Single Display Groupware Toolkit***, Master Thesis at the Department of Computer Science, University of Calgary, Canada

[WB00] Mickey Williams, David Bennett (2000), ***Visual C++ 6 Unleashed. From Knowledge to Mastery***, Sams, ISBN: 0-672-31241-7

[Wes02] M. Westergaard (2002), ***Supporting Multiple Pointing Devices in Microsoft Windows***, Proceedings of Microsoft Summer Workshop for Faculty and PhDs. Cambridge, England

[Wik07] Wikipedia, (2007), The Free Enzyclopedia, http://www.wikipedia.org
Viewed: 17th January 2007

Appendix A. Picture of Agile Planner



Appendix B. ControlSpy 2.0

Appendix C. Event Process

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

Neustadt an der Weinstraße, 15. Februar 2007