Dynamic User Interface Environment

Sun-Woo Kim

This project dissertation was submitted as fulfillment of a requirement for the MSc in Data Engineering

Department of Computer Science,

Keele University,

Staffordshire ST5 5BG

September 1997

Acknowledgement

I would like to thank my parents, sister and brother for their love and support. I am deeply thankful to Dr. Geoff Hamilton for supervising the project. My thank also goes to Professor C.Y. Jin of Wonkwang University and Professor J.H. Lee of GSMIS for their guidance and encouragement. Finally, I am grateful to M. Hamad for useful discussions.

CONTENTS

List of Figures

List of Tables

Abstract

This project discusses the lack of dynamic design within current interface design tools; proposes a conceptual model for a tool which supports the generation of dynamic user interfaces; and describes the development of the Dynamic User Interface Design Tool (DUIDT) based on this conceptual model. The conceptual model uses a state transition diagram (STD) to represent the behaviour of a system, and links static interface designs to STDs to add dynamics. DUIDT allows the designer to simulate the dynamic behaviour of an application-specific user interface without having to perform any coding. This makes it possible to develop an interface prototype in a shorter time, so that the designer and end-user can evaluate and verify the dynamic interface of the system earlier in the development cycle.

PART I INTRODUCTION

1. Introduction

In the early days of computer use, only a few computer specialists and professionals interacted with computer systems. However, in the last few years, as more non-specialists use computer systems in their work, a greater emphasis has been placed on effective user interfaces. Thus, the concept of a graphical user interface (GUI) was developed. The interface of most modern applications is based on the GUI interface because of the great advantages over character-based interfaces.

Recent recognition of the importance and difficulty of GUI design in software development has prompted the development of many tools to help the interface designer. GUI design can be divided into two groups - dynamic and static. Static user interface tools represent the layout of windows and allow interface designers to create, display, and manipulate objects such as text, drawings, icons, and other graphic images. User Interface Design Builders such as Delphi, Visual Basic, etc. are examples of static user interface design tools. These builders allow developers to build up the representation of interfaces quickly and to generate the screen design code with ease. However, these builders have some limitations. Since most builders support only static screen design, difficulty arises when trying to define the dynamics of the interface.

Dynamic means system behaviour as being derivable from the events that must be dealt with. Dynamic user interface tools support developing the behaviour of the interface during runtime by linking layout of interface design (representational aspects) to the functions of the system (operational aspects). While static user interface design is supported today by a wide range of visual programming tools, there are few dynamic user interface tools available despite the importance of dynamic design.

The aim of this project is to propose a conceptual model for building a dynamic user interface tool and to develop a tool based on this model. The dynamic aspects of user interfaces can be modelled through the use of state transition diagrams (STDs), which are conventionally used to capture the behavioural viewpoint of a system. A STD can be produced for each class of the system to describe the dynamic behaviour of its instances. A STD is concerned with modelling the dynamic aspects of the system in terms of entities such as states, events and conditions. The state represents an externally observable mode; the condition is an event in the external environment that the system is capable of detecting; and the action is the activity that the system invokes when it changes state. In order to add dynamics to a static user interface, the manipulation of window properties and events is needed. This can be achieved by linking static window designs to STD conditions and actions.

The Dynamic User Interface Design Tool (DUIDT) has been developed to demonstrate the conceptual model suggested. DUIDT takes the output from a state transition diagram editor; gets user window designs from the user, links these to state transition diagram information; and, generates event handling code extending an existing static user interface design to add dynamics. The intended function of the generated event code is window navigation and content sharing.

DUIDT allows prototype user interfaces to be produced in a shorter time, and feedback to be obtained from the developer and the potential end-user earlier in the development cycle. In addition, this tool reduces the development time of application-specific systems by allowing dynamic interfaces to be created without the designer having to write code, thus freeing the interface designer from laborious and error-prone hand-coding.

The target potential user is a system developer, i.e., anyone who is responsible for GUI development. The implementation language is the JDK (Java Development Toolkit) 1.1 and the Java Delegation Event Model is adopted to handle events. The hardware used is a SUN Workstation. The structured systems analysis and design, and GUIDE (Graphical User Interface Design and Evaluation) methodologies are used for system and interface development, respectively. In addition, the client-server approach is used to store data in a database, i.e., Oracle 7 is used as a database server and JDBC (Java Database Connectivity) is used to connect the Java program to database.

The rest of the paper consists of three parts - background, tool development, and conclusion. Part II begins with background of user interfaces. The historical perspective of user interfaces and the concept of graphical user interface (GUI) are briefly reviewed, and the categories of user interface tools are classified. Chapter 3 gives an explanation of dynamic modelling and of three types of state transition diagrams, and outlines briefly Mayerl, an STD editor. Chapter 4 illustrates dynamic user interface tools and the lack of dynamic user interface tool problem. In addition, Chapter 4 describes the conceptual model suggested to solve the perceived problem. Chapter 5 introduces a Keele Ferries Booking System which is used as a running example. Chapters 6 and 7 describe the processes of the tool development and shows the products of each development process stage. Chapter 8 presents the results and gives an analytical evaluation of the project. Finally, Chapter 9 summarises the work done and the usefulness of the work, and suggests further work.

PART II BACKGROUND

Part II introduces key concepts from the study of user interfaces and of dynamic modelling, and briefly mentions the Mayerl STD editor for drawing state transition diagrams. The limitations of current GUI interface design tools will be discussed, and a conceptual model of a dynamic interface design tool is suggested.

2. User Interfaces

2.1 Historical Perspective of User Interfaces

The user interface is the part of a tool or system with which the user interacts directly. The user interface is also the user's only means of accessing the inner functionality of the system. Since the use of computers has shifted from calculations or information processing to a wider variety of applications, and the direct contact between the human and the computer has become more and more advanced, the perspective of user interfaces and interaction styles has been changed.

When a few computer specialists interacted with computer systems through punch cards, the user interface was of no concern because the end user did not interface directly with the computer system. Later, users were allowed to communicate directly with the computer system through a terminal. Command-driven interfaces and form-fill applications are examples of this primitive user interface. Now, the user can communicate directly with computer system, and get fast feedback. However, command-driven interfaces caused problems in comprehension. In a command-driven system, the user has to know what the allowable commands are and needs to have a clear idea of the functions to be performed. In the last few years, as computer systems are operated by more non-professional people as well as by professionals, the character-based interface was no longer usable and effective.

The purpose of designing new effective user interfaces was to make the computer system more usable for unskilled users. Thus, the graphical user interface (GUI) was developed. Whereas characters were the building blocks of command-driven interfaces, a component is a building block for GUIs. A component is a user interface object through which the user communicates, such as a label, text editor, push button and so on. Objects can be represented on the screen to appear in a physical form to be manipulated by the user directly (Shneiderman, 1985). Most modern interfaces are developed using GUIs because well-designed systems with a graphical user interface are more usable - they are easier to learn, more effective to use and more satisfying.

2.2 Graphical User Interfaces (GUIs)

The central components of most Graphic User Interface (GUI) systems are a set of windows and components.

2.2.1 Window

A window is a communication channel through which the user looks to see and interact with the objects in the system. Typically, a window has a frame (borders), which defines its boundary, and titles; and performs various functions (such as opening, closing, moving, resizing, minimising to an iconic form, and maximising to fill the current screen). Each window contains one or more components, through which the users communicates with the system, and a set of window actions. A window action is an action the user can perform on a component (e.g. click a button, edit text in a text edit control, dragging and dropping an icon) in order to achieve some effect (e.g. displaying information in the window, moving to another window). Most modern interfaces allow several windows to be displayed at one time.

2.2.2 Component[1]

A component is a user interface object through which the user communicates, and it can correspond in appearance to some objects which can be manipulated. Different environments have different component sets. However, a number of them contain similar kinds of components, so that one can think of a generic component set which is applicable to most interfaces. For example, menu, label, text edit, push button and checkbox are components that are available in most environments.

Components have three basic functions:

- to allow the user to perform actions
- to allow the user to see the state of the objects in the system
- to allow the user to enter data.

Simple components may be limited to just one of the above functions; more complex components allow the user to perform two or all three functions. For instance, a text edit component is an example of a control used to display and enter data; a push button component is an example of a control used by the user to perform an action.

2.2.3 Event-based interface

The GUI can be viewed as a collection of events and event handlers. From the user interface point of view, windowing systems are principally driven by asynchronous events generated by the user. An event is an unexpected, external happening that imposes itself on components. The component looks for certain events which can occur through the peripheral input devices (Most computer systems use two basic input devices: the keyboard and a mouse). Component events could be any of the following: typing any key, moving the mouse cursor into or out of a window, pressing the mouse button, releasing the mouse button, and clicking on a mouse button, and so on. User actions give rise to events which are then passed to those handlers that have indicated they will handle them. If an event occurs in a component which the component is not looking for, then that event will be ignored. Some components might not have many relevant events. For example, a label component usually does nothing except provide a label for another component.

2.3 User Interface Tools

Any computer program used to design, make, maintain, manage, or test a software product can be called a software tool. Effective tools provide the benefits of increased productivity, enhanced software quality, and easier and better management of software development. In terms of the user interfaces of today's software systems, the user interface itself is large and complex, and the manipulation of windows can remain as a distraction from the user's tasks. Therefore, user interface tools are needed as with most software tools. In addition, a description on paper has a number of limitations. It cannot show the full dynamic

[1] A component is also called a widget, a gadget, a control, an interactors, or a GUI element according to environments. The term component will be used throughout the paper.

properties of the user interface, and it may not be able to communicate the true nature of the application to people outside the design team. Furthermore, the cycle of enhancement and evaluation is likely to be slowed down by documenting all details of each stage on paper. For these and other reasons it is common to use prototyping tools to describe ideas for user interface designs. Once described in this way, the designs can be tested directly, without the need to wait while they are implemented.[2]

The benefits provided by interface design tools are:

- Better communication with the user : The user can obtain early hands-on experience of how the GUI will look, feel and perform. They can assess earlier how it will impact their business.
- Improved design through feedback and iteration : The iterative nature of the approach means that a number of different approaches can be evaluated with the user, and their feedback used to evolve and improve the design.
- Reduced risk : The risk of producing an unsuitable GUI is greatly reduced.

There are a number of tools for various purposes. Some of the most widely advertised type of GUI tools are shown in Table 1. Many of GUI tools have functionality that place them in more than one category. One example is a GUI builder that also provides support for dynamic data visualisation.

[2] William M. Newman & Michael G. Lamming, Interactive System Design, (Addison-Wesley, 1995), p. 287.

Table 1 Categories of User Interface tools[3]

Category	Description	Product
A GUI builder	A developer can typically create and manipulate user interface displays in a "what you see is what you get" environment. Interface builders allow the designer to place predefined items (from an interface toolkit) on the screen and to specify behavioural aspects of the system. The designer normally has to specify behavioural aspects manually through the use of low-level code.	Builder Xcessory X-Designer Visual Basic PowerBuilder
A graphical user environment (Data visualisation tool)	Provide dynamic data representation and visualisation. A developer can define and animate objects or scenes of objects. Using a drawing editor, the developer designs static displays with standard or custom objects. Dynamic behaviour can then be attached to the objects.	Data Views SL-GMS
A user interface management system (UIMS)	A UIMS is a comprehensive set of tools intended for the development and use of user interfaces. It provides the functionality of a GUI builder as well as the capability to define and execute the functionality of the displays from within the tool.	UIM/X Garnet Amulet
Specialised widgets	Provide functionality beyond the basic capabilities of a typical widget set.	X/Motif
GUI porting tools	Automatically port user interface code to a different platform.	Wind/U

2.4 The Limitation of Current User Interface Tools[4]

A number of interface construction tools have been developed over the last few years, and are now widely available. These tools, commonly known as interface builders, offer a partial solution to the problem of building graphical interfaces. They allow their users to instantiate and customise interactive objects, which are usually called components. The designers need to be able to lay out windows (variable sized regions of the screen) which are composed of display objects such as text, lines, shapes, and icons. Designers may also want to create other graphic objects such as buttons, menus, scroll bars, dialogue boxes, and so on. A user interface toolkit (also called palette) is a library of such interface objects and related information. The palette offers a choice of elements which can be dragged across to the screen design. For instance, commercial interface builders such as HP Interface Architect, TeleUSE, or XFaceMaker, are based on the ODF-Motif component set (OSF, 1993). Visual Basic for Windows uses the set of components provided by the CUA standard (IBM, 1991).

These tools provide graphical representations of the objects of interest (the components themselves), and allow the designer to manipulate these graphical representations in order to set the parameters of the objects. However, most of such interface builders do not have programming facilities, since setting parameters is not programming. The lack of programming capabilities can be related to another weakness of these tools: they can only help to build what we call static interfaces. Such interfaces are able to react to user's actions, but in fact, the only

[3] Laura A. Valaer, "Choosing a User Interface Development Tool," IEEE Software, July/August (1997), p. 29.

[4] O. Esteban, S. Chatty, P. Palanque, WHIZZ'ED: A Visual Environment for Building Highly, Human-Computer Interaction Interact '95, Chapman & Hall 1995, p121-126

difference from static screens lies in predefined behaviours, which can hardly be changed. Creating more dynamic and highly interactive interfaces, requires more than the instantiation and parameterization of predefined components. In fact, some of the interface builders offer facilities for reprogramming or enriching the behaviour of their components. But for this purpose, they provide specialised textual programming languages whose manipulation requires traditional programming skills.

3. Dynamic Modelling

3.1 The Dynamic View of a System

Software development usually involves modelling and describing its behaviour as well as its structure, and also the functions that it will perform. That is, the system view can be categorised into three viewpoints - functional, structural, and behavioural (dynamic). The dynamic viewpoint of the system is captured by dynamic modelling which is used to analyse and describe the dynamic behaviour. Dynamic modelling is useful where there is significant state-dependent behaviour, i.e., where the behaviour of an object changes through time, depending on the state that the object is in.

Since the dynamic behaviour is sometimes complex and difficult to understand, it is important to have an appropriate diagrammatic notation to help visualise the behaviour. Though there are several such representations of a behavioural view, a state transition diagram (also called finite state or state diagram) is the most widely used and well-known notation. The state transition diagram (STD) is concerned with modelling the dynamic attributes of the system, in terms of entities such as states, events, and conditions.

3.2 State Transition Diagrams

The state transition diagram describes the temporal evolution of an object of a given class in response to interactions with other objects either inside or outside the system. It provides a convenient means for modelling many systems which have an event-driven nature. There are a number of different extended methods but the main idea is the same. Three representative diagrams will be considered in a little more detail.

3.2.1 The State Transition Diagram (STD) - Ward and Mellor (1985)[5]

The components are the state, represented by a rectangle; the transition, represented by an arrow with a horizontal line; the transition condition, shown adjacent to the transition above the line; and the transition action, shown adjacent to the transition below the line. A state represents an externally observable mode of behaviour. There are two special types of state - initial and final states. The initial state is the state of the system before any transitions have occurred. A final state should be thought of as a ‘dead end’ in the behaviour of the system. A transition represents the movement from one state to another. Transitions can exist between any state and any other state, including the state from which the transition started. Multiple transitions to and from a given state are permissible. The transition condition identifies the condition that will cause the transition to occur - usually in terms of the event that caused the transition. The transition action is the action that arises as a result of the transition. Several independent actions may be taken on a single transition simultaneously, or in a specific sequence.

---

[5] Paul T. Ward & Stephen J. Mellor, Structured Development for Real-Time Systems, (Yourdon Press, Prentice-Hall, 1985).

The STD (also called flat state machine) is a useful modelling tool and has been widely used to model real-time systems. However, the principal disadvantage of the technique is that if many transitions are permitted between a small number of states, the diagram can become very tangled. Similarly, large and complex systems lead to large and complex diagrams because of the STD's lack of hierarchy.

Figure 1 Example of state transition diagram - (Ward & Mellor, 1985, p68)

3.2.2 The Statechart - David Harel (1987; 1988)[6]

David Harel's statechart is a major advance on the limitation of traditional flat state machines. Like the STD, the statechart is concerned with providing a behavioural description of a system. However, it provides a rather more abstract and modular form of description than the STD, by enabling hierarchy, orthogonality (i.e., concurrency), and encouraging 'zoom' capabilities for moving easily back and forth between different levels of abstraction. The ability to create a hierarchy of abstraction enables viewing the description at different levels of detail, and make even very large specifications manageable and comprehensible. Concurrency permits the designer to describe transitions that are orthogonal in that they are completely independent of each other. This makes it possible to easily describe synchronised behaviours.

A state is denoted by a box with rounded corners, labelled in the upper left corner. Hierarchy is represented by encapsulation, and directed arcs are used to denote a transformation between states. The arcs are also labelled with a description of the event and, optionally, with a parenthesised condition.

[6] David Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, Vol. 8, (Elsevier Science Publishers B.V., North-Holland, 1987), p. 231-274.

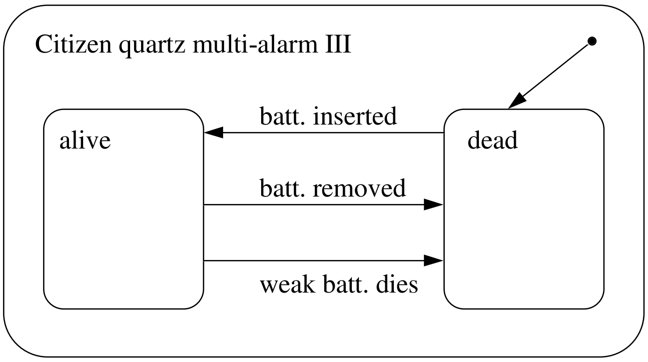


Figure 2 Example of statechart - (Harel, 1987, p246)

Two elements of the statechart -hierarchy and concurrency, allow statecharts to be compact and expressive, i.e., small diagrams can express complex behaviour. The statechart is thus increasingly accepted as the most powerful form of dynamic modelling for object-oriented analysis. The difference between the statechart and the STD is that the statechart has more refined mechanisms for describing abstraction, while the STD provides a more detailed description in terms of the actions that occur. In that sense, they are largely complementary rather than alternative. The statechart is perhaps more suited to modelling problems, and the STD better suited for modelling detailed solutions.

3.2.3 State Diagram - UML (Unified Modelling Language)[7]

The semantics and notation of state diagrams described in the UML Notation Guide are substantially those of David Harel's statechart with some modifications. A state diagram shows the sequence of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.

A state has state variables, an internal activity compartment, and a name. State variables are attributes of the owning class affected by or used by actions in the state diagram. The internal activity compartment holds a list of internal actions. The 'entry', 'exit', and 'do' special actions have the same form but represent reserved words that cannot be used for event names. The entry action is always performed when a state is entered, and the exit action is always performed when a state is exited. An internal do action is an ongoing process performed while the object is in the given state. A state might have concurrent substates (and-relationship) or mutually exclusive disjoint substates (or-relationship).

[7] Rational Software Corporation, UML Notation Guide, http://www.rational.com/uml/references/notation-guide.html/, (1997), Chapter 8.

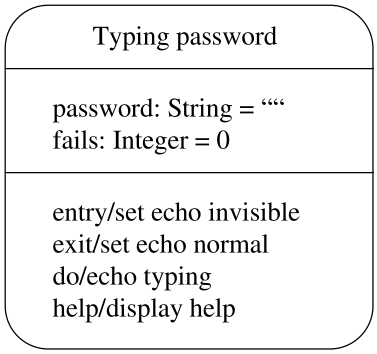


Figure 3 A state of State Diagram

As an alternative to showing actions on transitions, actions can be associated with entering or exiting a state. There is no difference in expressive power between the two notations, but frequently all transitions into a state perform the same action, in which case attaching the action to the state is more concise. An 'entry' action is shown inside the state box following the keyword entry and a "/" character. Whenever the state is entered, by any incoming transition, the entry action is performed. An 'entry' action is equivalent to attaching the action to every incoming transition. If an incoming transition already has an action, its action is performed first. An 'exit' action is shown inside the state box following the keyword 'exit' and a '/' character. Whenever the state is exited, by any outgoing transition, the 'exit' action is performed first. If multiple operations are specified on a state, they are performed in the following order: actions on the incoming transition, 'entry' actions, 'do' activities, 'exit' actions, actions on the outgoing transition. 'do' activities can be interrupted by events that cause transitions out of state, but entry actions and exit actions are completed regardless, since they are considered to be instantaneous actions. If a 'do' activity is interrupted, the 'exit' action is nevertheless performed. Moving into or out of a substate in a nested diagram can cause execution of several 'entry' or 'exit' actions, if the transition reaches across several levels of generalisation. The 'entry' actions are executed from the outside in and the exit actions from the inside out.

'Entry' and 'exit' actions are particularly useful in nested state diagrams because they permit a state to be expressed in terms of matched entry-exit actions without regard for what happens before or after the state is active.[8]

A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform a certain specified action when a specified event occurs and the specified condition is satisfied. This transition is shown as a solid arrow from one state to another state labelled with a transition string. The string has the following format:

event-signature [guard-condition] / action-expression ^ send-clause

[8] J. Rumbaugh & M. Blaha, etc., Object-Oriented Modelling and Design, (Prentice-Hall, 1991).

A complex transition may have multiple source states and target states. It is shown as a short heavy vertical bar. A self-transition is from a state back to the same state. The self-transition causes the exit and entry actions on the state to be executed and the initial state to be entered.

3.3 The STD Editor - Mayerl

The Dynamic User Interface Design Tool (DUIDT) does not include the STD diagram editor. Instead, it uses the Mayerl STD editor developed by James Willans in Keele university. The main function of the STD editor is to draw a state transition diagram of a class, and to save STD information extracted from the diagram, to a database. The Mayerl editor is implemented with the JDK 1.1, following UML notation (see Section 3.2.3), and uses JDBC and Oracle 7 server.

The main screen of the STD editor with a STD diagram example on it is shown in Figure 4. An example diagram is a STD of the 'Ferries Service' object in the Keele Ferries Booking System (see Figure 9 on page 29).

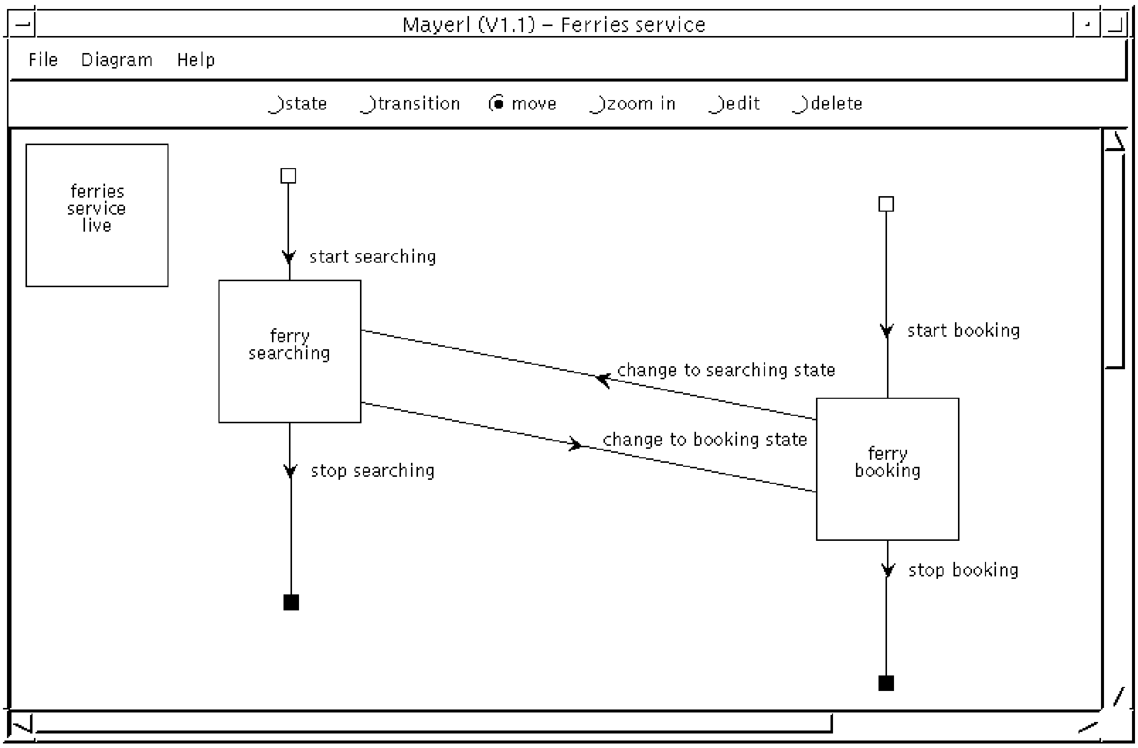


Figure 4 The main screen of Mayerl STD editor

4. Dynamic Interfaces

4.1 Issues Raised by Interface Building

Graphical interfaces have static and dynamic properties. The static properties are simply the graphical appearance of a system (representational aspects). The dynamic properties are the underlying functional behaviour of the user interface specification (static interface). Dynamic properties can be represented much better and tested more effectively with an executable prototype than is the case with other means of representation.

Problems with current design tools for the generation of user interfaces are poor facilities for specification of dynamic behaviour and poor integration between generation tools and GUI builders. The user interface layout specification specifies the static layout, but without considering the dynamic behaviour. GUI builders often have no appropriate specification language. Only direct code generation and internal layout representations are possible in most cases. However, within todays graphical user interfaces (GUIs), a large number of closely coupled dynamic effects exist. Actually, the most important aspect during the development of a prototype for exploring the user's requirements is not the layout of the user interface but its dynamic behaviour. The complexity of an interactive application actually lies in the behaviour of graphical objects, and in the many interrelations between them.(Myers et al., 1990). For instance, a real-time monitoring and control system typically requires graphical displays to improve understandability. The dynamic characteristics of these displays help users perform their required activities more efficiently. In addition, even when the static structure of a program is complete, the correctness of its dynamic behaviour still needs to be confirmed. This need to keep both static form and eventual dynamic behaviours continually in mind when developing interfaces of a system.

Another point is that the design of an interactive application interface usually involves human factors specialists, future users of the application, and occasionally graphics designers. These people usually do not have an education in computer science, and most of them have to rely on highly specialised programmers for building prototypes of the system they are designing. Furthermore, the design of efficient and usable interfaces requires many iterations of the design and evaluation of prototypes. As long as complex programming is necessary, the development of a system with a high quality and better interface will be very costly.

Despite the importance of dynamic design, much less support is available for the dynamic part of a GUI. The development of graphical user interfaces with application-specific appearance and behaviour is still a challenge not met by current tools. One possible reason for this is that it is quite difficult to generalise the functions of a dynamic tool because of its business-specific characteristics. Thus the challenge here consists of providing easy-to-use programming design facilities, so that interface designers can build highly interactive applications or prototypes rapidly.

4.2 The Conceptual Model

In design builders, the way dynamics are added into the interface design is that the interface designer manipulates component attributes and specifies reaction routines by hand. The process of developing dynamic user interfaces may be improved by combining formal specifications of the dynamic behaviour with interactive layout specifications of the static appearance of user interfaces (Lauridsen, 1995). Therefore, I present here a conceptual model which allows the designer to create application-specific dynamic user interfaces automatically.

The conceptual model is based on the 'object-oriented' ideas, i.e., the organisation and behaviour of user interface are object-oriented. The model considers static interfaces (the appearance of windows) as views of user classes allocated to the window components, and the behaviour of interface as the action of classes identified in the system. This approach provides a useful basis for the definition of windows, and promotes an object-action style of user interface.

The conceptual model consists of classes identified; state transition diagrams of the classes allowing the designer to describe the behaviour of the classes in a visual way; static window designs derived from the classes; and predefined interface behaviours. The classes should reflect the business world and the requirements of user tasks. The identified classes are the basis of an object-based GUI design and of drawing state transition diagrams. The use of graphical notations for describing behaviour is easier and can be used by a broad range of people with different programming experience. State transition diagrams allow the description of behaviours in a very natural way based on the transition paradigm in which one describes paths, attaches graphical objects to them, and then executes the resulting design allowing us to specify separately the behaviour of the computer system. Static window design is simply the layout of components in the window (frame) not having any events and actions. Two additional interface behaviours are also defined. Window navigation and content sharing are predefined elementary functions supported by the conceptual model. Window navigation is moving from one window to another based on user actions; and content sharing is saving the current state of the display of a window component's contents to object attributes, and setting it to other windows' components which are linked to the same object attributes.

These two behaviours can be obtained by connecting static window designs to state transition diagrams. Figure 5 explains the linking of STDs and static windows. STDs have states, conditions, and actions. A window has components, and a component has events and attributes. Events and consequent actions of static window designs can be manipulated based on the linking information, and thus dynamic user interfaces can be created.

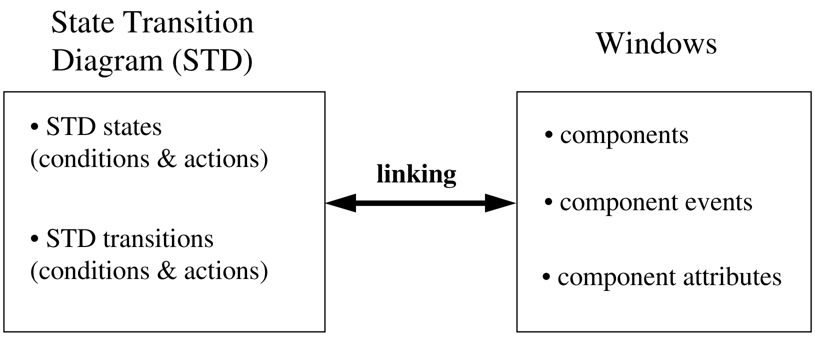


Figure 5 Linking of STDs and windows

Window-oriented user interfaces can also be represented by state transition diagrams; they are defined in terms of a set of states and possible state transitions. From the viewpoint of the user of a software system, these states are defined in terms of certain representations on the window such as text display. The state transitions are triggered by the user's events (such as clicking a button), and then proper reaction is performed by the event handling code, i.e., STD and window have a semantic homogeneity. It must be possible to achieve each user object action using window events, such as pressing a button, editing a component, selecting a menu, double-clicking. etc. This characteristic makes it natural to link window events to STD events (also called STD conditions); and link window actions to STD actions.

Figure 6 explains the general overview of the system - how the tool works. Firstly, the system gets a component event such as clicking a button component. Since this event is linked to the STD condition, we can find the corresponding STD action. This action is linked to the window which should be activated when that event happens. For example, clicking on the button might cause a window to appear. In addition, the action is linked to a component attribute. So, the event causes the change of the component attribute as well. For example, an event on a component may have an effect of changing the text it maintains; changing the object attribute linked to that component attribute; and changing other component attributes linked to this object attribute. In summary, the tool should provide facilities to link windows to the STD, and to generate event handling code which performs proper window actions (window navigation, window content sharing) throughout the run time of system.

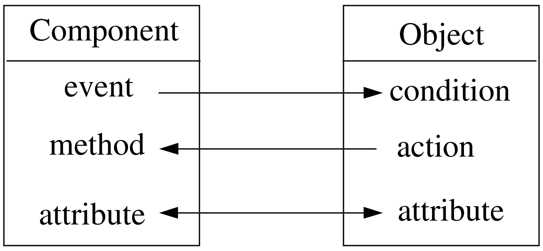


Figure 6 Overview of the conceptual model

4.3 Dynamic Interface Tools

User interface prototyping only makes sense if prototypes can be developed rapidly and incrementally, which is not possible without appropriate tools. In addition, since a number of the events and properties of window components need to be manipulated, writing event handling code is not trivial work and takes time. Especially, if developers are building a large system or if interface designers are not accustomed to the programming language, even writing very simple event code for a prototype will require lots of effort and time. Dynamic user interface tools can generate predefined event handling code to be activated when a component event occurs. Thus important parts of the dynamic behaviour of the system can be specified without writing a single line of code. Consequently, the dynamic interface tools allow the developers to make a rapid prototype which permits the simulation of the dynamic behaviour of a system. In this way, the correspondence of prototype functionality and the user's requirements can be verified by playing realistic examples without programming; and the developers can get useful feedback from the user earlier in the development cycle.

PART III TOOL DEVELOPMENT

Part III introduces a business system development used as an example of dynamic interface design. Dynamic User Interface Design Tool (DUIDT) has been developed using two methodologies. Chapter 6 shows the development process and products for system development based on the Structured System Analysis and Design Methodology. The functionality and data structures of the system, the structure of the objects that constitute the system, and the operational aspects of the system are covered here. Since DUIDT has a GUI, it is necessary to use an interface development methodology. That is, the user interface of the user interface design tool needs to be designed following an interface methodology. The Graphical User Interface Design and Evaluation (GUIDE)[9] methodology is used to develop the interface of the system. Chapter 7 describes the representational aspects of the system (the form and content of the windows) and documents the processes and products of each interface development stage.

[9] D. Redmond-Pyle & A. Moore, Graphical User Interface Design and Evaluation, (Prentice-Hall, 1995).

5. Example - Keele Ferries Booking System

This chapter introduces an example of a GUI system development, which is used as a running example through the tool development. This simple example is used to illustrate and test how the dynamic user interface design tool works. The example is a Keele ferries booking system based on a fictitious GUI design project.[10] The project is conducted following the GUIDE methodology (D. Redmond & A. Moore, 1995). The GUIDE methodology will be explained in detail in Section 7.1.

Since the main purpose of the example is to test and to demonstrate the DUIDT, the design processes will be simplified as much as possible. That is, the irrelevant system development procedures such as data structures, business processes, etc. and even some GUI design processes, which are not directly relevant, will not be discussed. Instead, the design work focuses on task scenarios, dynamic modelling, and user interface designs (window design).

5.1 System Requirements

A travel agent currently uses a manual system for booking ferries through the ferry company Keele Ferries. Using this system, the following form is completed by the travel agent, and sent away to Keele Ferries to make the booking.

<table>
<tr><td colspan="2">Outward Voyage</td><td>Inward Voyage</td><td colspan="5">Reserved Accommodation</td></tr>
<tr><td rowspan="4">First choice

Date
Sailing time</td><td>From</td><td>From</td><td>Type of cabin preferred</td><td colspan="2">Outward
Night/day</td><td colspan="2">Inward
Night/Day</td></tr>
<tr><td>To</td><td>To</td><td rowspan="5">If whole cabin is not required. No. of berths/couchettes*

*delete as applicable</td><td rowspan="5">Male</td><td rowspan="5">Female</td><td rowspan="5">Male</td><td rowspan="5">Female</td></tr>
<tr><td></td><td></td></tr>
<tr><td></td><td></td></tr>
<tr><td rowspan="4">Second choice

Date
Sailing time</td><td>From</td><td>From</td></tr>
<tr><td>To</td><td>To</td></tr>
<tr><td></td><td></td><td>No. of reclining seats</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>No. of club class seats</td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="3">Name and Address
(Block capital please)
Name</td><td colspan="5">Vehicle Details

Reg. No</td></tr>
<tr><td colspan="3" rowspan="2">Address (or Agent's stamp)</td><td colspan="5">Overall length (inc. tow-bar) m Height under 1.83m*/over 1.83m*
(inc. roof-top luggage)
* delete as applicable</td></tr>
<tr><td colspan="5">Caravan*/Trailer* details
*delete as applicable</td></tr>
<tr><td colspan="3">Post Code</td><td colspan="5">Motorcycle Reg. No.
Solo/combination*
* delete as applicable</td></tr>
<tr><td colspan="3">Telephone No.</td><td colspan="5">Passengers No. of children (over 4 and under 14)
No. of adults (inc. driver)</td></tr>
<tr><td colspan="3">Chalet/Caravan/Camping site</td><td colspan="5">Insurance</td></tr>
</table>

[10] This example is mainly extracted from the assignment of the Human-Computer Interaction module in Keele University.

please tick appropriate box	Holiday insurance ❐ Vehicle cover extension ❐ Caravan/trailer cover extension ❐	
Tent rental ❐	Car make	Car model
Chalet ❐ Caravan/camping site ❐	Date of return if not stated above	Age of vehicle if personalised number plate
	Please tick box if cover required for winter sports activities ❐	

Customers contact the sales staff by either visiting the travel agency or telephoning to make enquiries and/or bookings for ferry services. Enquiries for potential travellers concern routes, sailing and journey times, facilities and additional services. The additional services which are available are insurance, reserved accommodation, and a booking service for holiday sites. Bookings are for a specific outward and, optionally, a specific inward journey. Keele ferries need to know the total number of people travelling and the characteristics of any vehicle.

A new system has been proposed to support the sales staff in travel agents and to provide a fast and more efficient enquiry and booking service. The purpose of the proposed system is to provide an on-line system for enquiries and bookings at each sale position. This system should support detailed enquiries and validate bookings as they are entered.

5.2 Task Model

The following tasks can be identified

- answer customer enquiry
- book ferry crossing
- sell additional service

A hierarchical task model is in Figure 7. Subtasks requiring computer support are shown with a bold outline.

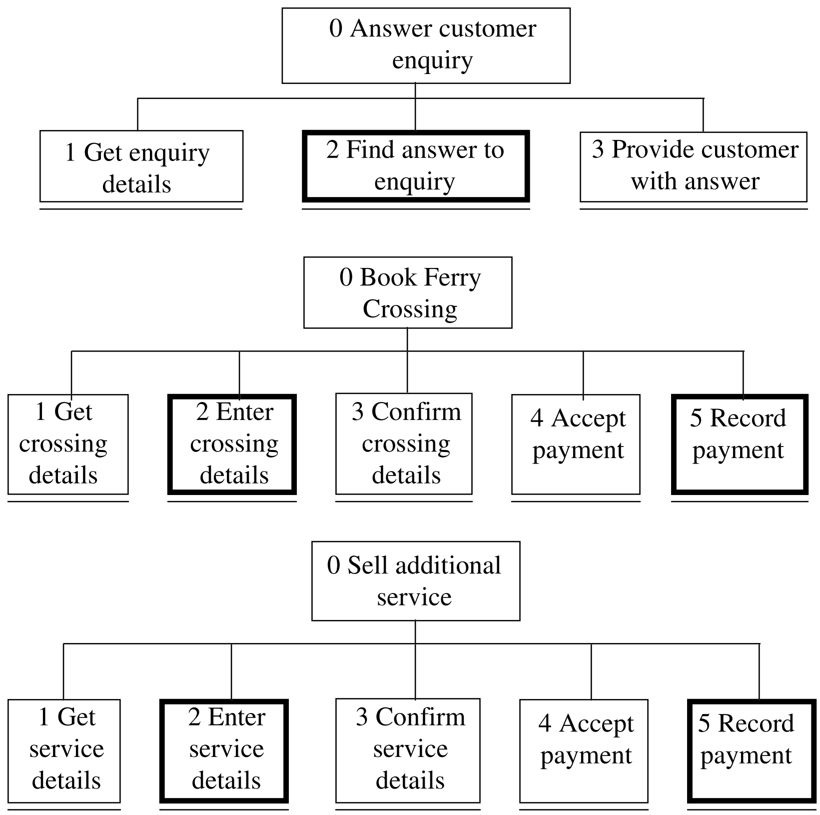


Figure 7 The task model of Keele ferries booking system

5.3 Task Scenarios

Some example scenarios are as follows:

Scenario 1: Answer Customer Enquiry

Customer: ‘I would like to go to France from the south east of England. Can you please tell me which routes I can take?’

Salesperson: ‘You can go from Dover to Calais or from Ramsgate to Dunkirk.’

Customer: ’I would like to travel from Dover to Calais on Saturday 14th December. Can you please tell me what the sailing times are on that date?’

Salesperson: ’The sailing times are 7: 00am, 9:30am, 12:00 noon, 2:30pm, 5:00pm, 7:30 pm and 10:00pm (Greenwich Mean Time).’

Customer: ’How long does each sailing take?’

Salesperson: ’45 minutes.’

Customer: ’I would like to return by the same route on Friday 20th December. Can you please tell me what the sailing times are on that date?’

Salesperson: ’The sailing times are 8:15am, 10:45am, 1:15pm, 3:45pm, 6:15pm, 8:45pm and 11:15pm (Greenwich Mean Time).’

Customer: ’If I were to take the 9:30 am sailing out and the 6:15pm sailing back, how much would it cost me as a foot passenger?’

Salesperson: ’Forty pounds.’

Customer: ‘If I were to take holiday insurance as well, how much would that cost me?’

Salesperson: ’An additional fifteen pounds.’

Scenario 2: Book Ferry Crossing

Customer: ’I would like to book a return ferry crossing from Dover to Calais as a foot passenger, leaving at 9:30am on Saturday 14th December, and returning at 6:15pm on Friday 20th December.’

Salesperson: ’Can you give me your name and address please?’

Customer: ’Joe Bloggs, 22 Acacia Avenue, London.’

Salesperson: ’Can you give me your post code please?’

Customer: ’SW1 1AA’

Salesperson: ’Can you give me your daytime telephone number please?’

Customer: ’0171 1234567’

Salesperson: (Enters details) ’Can I just confirm you will be sailing from Dover to Calais as a foot passenger at 9:30am on Saturday 14th December, and returning at 6:15 pm on Friday 20th December.’

Customer: ‘That is correct’

Salesperson: ’That will be forty pound please.’ (Accepts payment and records the fact that it has been made).

Scenario 3: Sell Additional Service

Salesperson: ’Would you like to buy holiday insurance?’

Customer: ’Yes please’

Salesperson: ’That will be fifteen pounds please.’ (Accepts payment and records the fact that it has been made).

5.4 User Object Model

The system has four objects - customer, ferries service, insurance, and holiday sites service. The attributes and actions of each object are as follows.

Customer

Attribute	Action
SSN (to identify a customer) name address post code phone number	create a customer instance change personal details

Ferries Service

Attribute	Action
ferry registration number port from (port name) port to date time sailing time list of fares (foot passenger, cabin, vehicle...) ferry details (capacity, ...) occupied details (no. of booked passengers, vehicles...)	ferry service created reserve a place release(cancel) a booked place service searched change availability status ferry service deleted

Insurance

Attribute	Action
name price description	create a service change service details (price, condition, ...) delete a service

Holiday sites Service

Attribute	Action
site identification number availability status site details (type, price, capacity, location,...) occupied details	holiday site service created reserve a site release a reserved site search for price, location, availability holiday site service deleted

5.5 Object Relationships

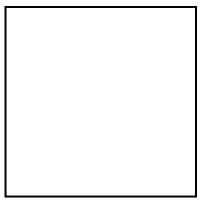


Figure 8 Object relationships

5.6 Dynamic Modelling

A STD is attached to an object’s class. STDs of all objects are shown in Figure 9, Figure 10, Figure 11, and Figure 12.

Figure 9 A STD of 'Ferry Service' object

Figure 10 A STD of 'Customer' object

Figure 11 A STD of 'Holiday sites Service' object

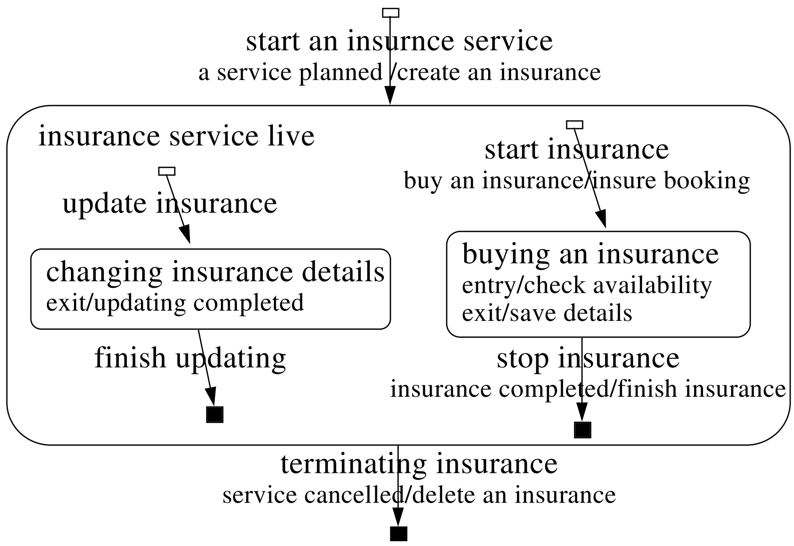


Figure 12 A STD of 'Insurance' object

5.7 Window Design & Prototype

The initial windows have been designed on paper by hand. After the evaluation of the initial design, some changes have been made to the window designs. The last version of paper sketches have been prototyped with Java AWT. Two window prototypes are shown in Figure 13 and Figure 14.

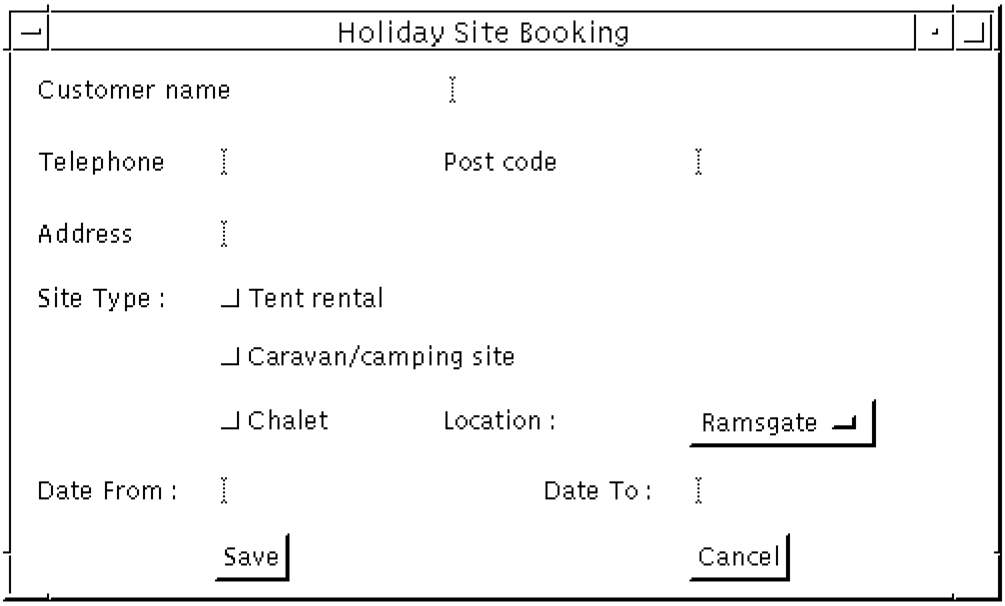


Figure 13 'Holiday sites Booking' window

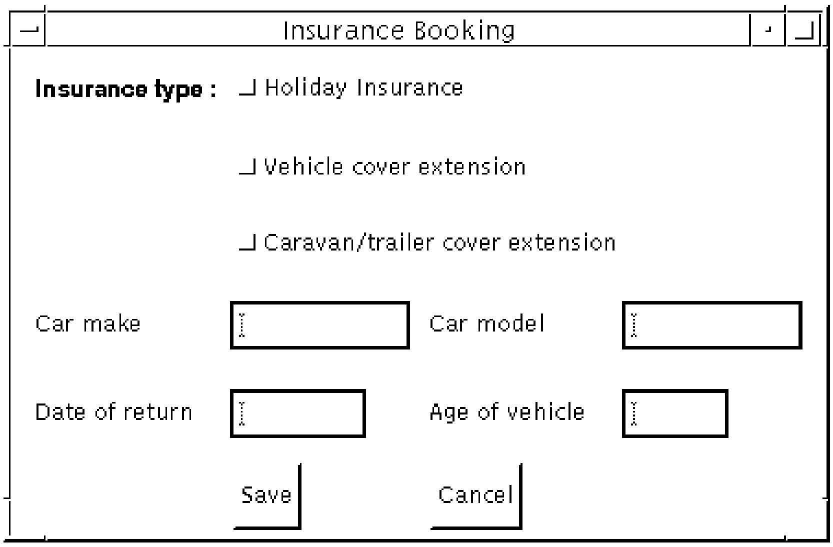


Figure 14 'Insurance' window

5.8 Window Navigation

In order to perform the task scenarios described in Section 5.3, seven windows have been designed. The navigation between windows is shown in Figure 15.

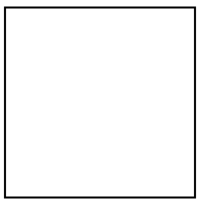


Figure 15 Window navigation diagram of Keele Ferries Booking System

6. System Development

6.1 Methodology

The methodology used mainly follows Structured System Analysis and Design Methodology (SSADM).[11] However, there are some differences between the two. Since the purpose of the project is to develop a software development tool which supports the dynamic user interface design of object-oriented systems, the object-oriented concept must be handled. In addition, the language of implementation was the Java, object-oriented programming language. Therefore, Object-Oriented Analysis and Design techniques have also been adopted as a development methodology. In addition, the evolutionary prototyping approach is used. That is, the first prototype of system with core functions is implemented quickly, and then, the prototype is developed to the final product through iterative testing, re-analysing, and re-designing processes.

The development processes and product of each stage are as follows:

Analysis

The goal of analysis is to describe the scope and functionality of a system.

- Statement of purpose : System requirements analysis.
- Context diagram : The scope of the system.
- Class model : Class identification, brief description of logical classes.
- Data Flow Diagram : Functional model.
- Entity-Relationship Diagram : Logical data structure model.

Design

This stage focuses on the question of how the user's requirements are to be implemented. The basic design rules for each object, such as rules on how to manage user interfaces and how to handle databases are discussed.

- Data structures : Relational structure of system data
- Detailed design of classes : internal structure, assumptions, and restrictions of a class.

Implementation

- Module structure chart : Structure chart of physical (implementation) class.
- Pseudo code : Brief algorithm of implementation class.
- Source code

Testing

[11] E. Yourdon, Modern Structured Analysis, Prentice-Hall,1989.

- Testing forms and data
- Testing results

6.2 Analysis

6.2.1 System Requirements

The statement of purpose represents a description of the goals of the system, the functions it must perform, and the constraints of the system. The statement of purpose is shown in Appendix A.

6.2.2 Context Diagram

The scope of the system is shown in the context diagram in Appendix B. The outside entities which interact with the system are a developer and a STD editor. The developer is anyone who working on the GUI system development. The group of developers will be discussed in more detail in Section 7.2. STD editor is the state transition diagram drawing tool as explained in Section 3.3.

6.2.3 Class Model

The classes identified in the system are :

- project
- user class
- user window

The attributes and behaviours of each class and their relationships will be explained in detail in Chapter 7.

Project

Project means developer's (tool user's) project - a system development. For instance, the Keele ferries booking system used as a running example in this dissertation is a project. A project has business-specific classes, objects (an instance of a class), window designs, state transition diagrams for classes, and linking between STD and user windows.

User Class

User classes represent things in the real world that the project system is concerned with. For example, 'customer' and 'ferries service' are user classes of the Keele ferries booking system. A user class has a state transition diagram, and a number of user objects which are instances of that user class.

User Window

A user window is a static user interface design for a project. For example, Figure 13 on page 31 is a user window of the Keele Ferries Booking System. A window consists of a number of components, component events, and component attributes. A user window has event handling code which is generated by the tool based on linking data. This should do the intended work - window navigation and component content sharing.

6.2.4 Data Flow Diagram

The function flows are shown in Appendix C.

6.2.5 Entity-Relationship Diagram

ERD represents the logical data model of the system as entities and their relationships. In the system, the ERD can be classified into three logical parts - STD, user window, and linking of user window and STD. The ERDs are shown in Appendix D.

6.3 Design

6.3.1 Data Structures

In order to store data files, a relational database will be used. Appendix E shows the data structures of the system. Some attributes represented as a primary key such as user class name, screen name, etc. should be unique throughout the system.

6.3.2 Detailed Design of Class

Before starting implementation, it is useful to specify a scope, assumptions, restrictions, and a description of the internal structure of each class module.

6.3.2.1 Project

A project is the basic unit of work to be done by the system. When the system starts, a user needs to open a project and then, all information belonging to that project such as STD information of classes and user window designs will be retrieved. In order to add dynamics to the static window designs, windows need to be linked to STDs by the user. This linking can be specified in four possible cases - link windows to STD actions, link component events to STD conditions, link component attributes to STD actions, and link component attributes to object attributes.

6.3.2.2 User Class and Class Attributes

The system should allow users to add or update class attributes of a user class. However, the class itself should not be deleted or updated in the system because it is directly related to a state transition diagram of that class, which is outside the scope of the system. For example, assume the ‘ferries service’ class of the Keele ferries booking system has a ‘ferries service STD’ drawn on the STD editor. If the class name is changed to ‘scheduled ferries’, the ferries service STD information will be lost. For that reason, a user class can be updated only in the STD editor.

6.3.2.2.1 User object and object attributes

Users can add or update an object and object attributes freely. The change of object and object attributes might cause the deletion of linking data because some component attributes could already be linked to these object attributes.

6.3.2.2.2 STDs

The form for representing class dynamics used in the system substantially follows UML's State Diagram. However, some part of UML's notation are omitted to make it simple. The form is classified into state and transition. A class state diagram has initial and final states. A state has a name and internal activities, and might have parent and child states. An internal activity has two reserved words - entry and exit. A transition has a transition name and transition activity which is represented by possibly more than one condition and action pairs. Since it might have several condition/action pairs, the transition name is necessary to identify a unique transition.

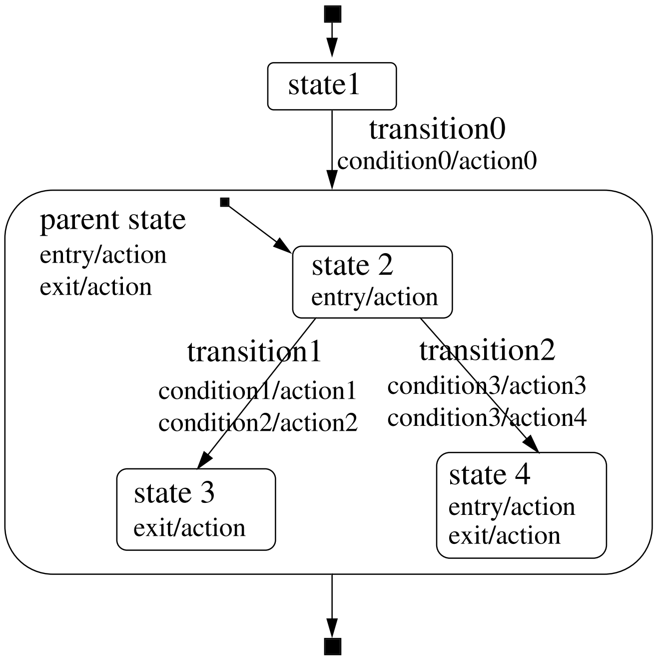


Figure 16 STD Notation used

The possible cases of transition activity are (a) one condition/one action, (b) one condition/several actions, and (c) several conditions/one action, and (d) many conditions/many actions.

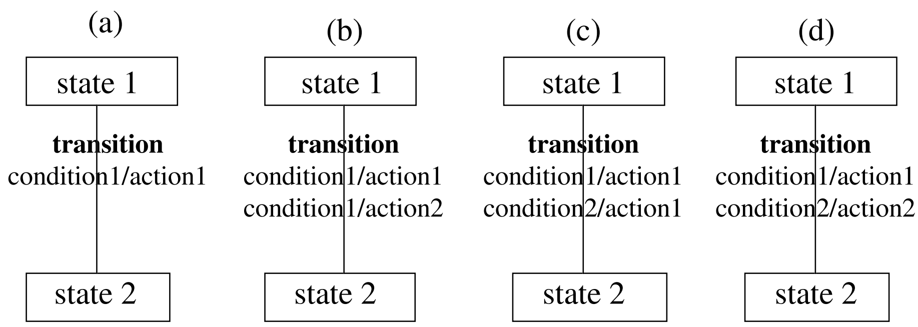


Figure 17 The cases of condition/action pairs

Transition and state relationships could be complex. That is, one state might be linked to more than one transition. The possible cases are (a) one "from state" and one "to state", (b) several "from states" and one "to state", (c) one "from state" and several "to states", and (d) many transitions linked to the same "from state" and the same "to state."

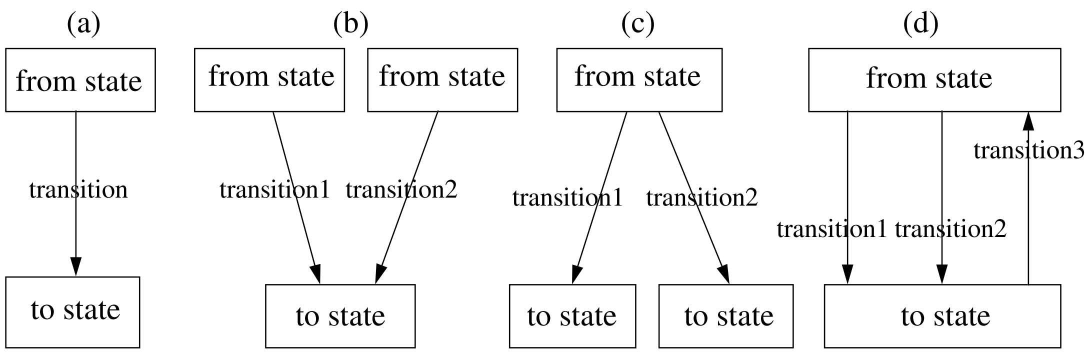


Figure 18 The cases of state transition relationships

6.3.2.3 User Windows

User windows are tool users' (system developers') business-specific window designs. The window designs should be made and added into the system by the system developers. Then, window design files are read and the components it holds are detected.

6.3.2.3.1 Components

Firstly, it is necessary to specify what kind of components should be detected. The list of Java standard components are Checkbox, Choice, List, Button, Canvas, Container (Panel, ScrollPane, Window), Label, Scrollbar, TextComponent (TextArea, TextField), and MenuComponent (MenuBar, MenuItem).

Since the purpose of the system is to add dynamics to the components, there is no reason to take care of all Java components. For example, the Scrollbar component is not likely to be used to navigate screens or to share object attributes. Therefore, the components which are suitable for the purpose are selected. They are:

Button, Choice, Label, Checkbox, Choice, List, TextArea, TextField

In order to add events for the components, these components should be detected from the user window. Thus, the next stage is to think of how to catch the components from the Java code. The method used is pattern matching. That is, the basic patterns of coding are set, and then the lines of screen code are compared to find a component type such as Button, Label, etc. and its name (variable). One matter to be considered is that the pattern matching should be general because different users have different coding styles. Actually, the user can adopt any way of coding as long as the Java compiler does not complain.

For example, if we regard the general coding style for making an instance of the Button component class as in the following:

```
Button btn = new Button("Ok");
```

Then, the pattern might be '= new Button('. If a line of screen code is matched with this pattern, the String right before '=' is the variable name ('btn') that we want to know.

However, the Button component could be coded in other ways as well.

```
a) Button btn =
        new Button("Ok");
b) Button btn = new Button();
   btn.setText("Ok");
c) add(new Button("Ok"));
```

These are all legal Java code. The code a) is written over two lines, the code b) uses the setText() method to give the Button label rather than using the constructor, and the code c) has no variable name and is added into a container directly. For example in b), the previous pattern might work fine because the thing to be detected is not a Button label but a variable name. However, for the other two cases, the pattern set in the above section is not valid any more.

The example a) can be solved simply. Though the code can be spread over several lines, the programmer must enter the end of line terminator (;) to indicate a new line. The solution is for the whole program code to be combined into one String, and chopped up with ';' delimiter to re-organise the line of the program. The code in c) is problematic. Since it has no variable name, it does not make sense to detect a component variable. This also affects the event handling process. In the Java 1.1 event model, a component should register with an event listener the variable name to be listened for. For example, if the above Button component has an action event to be handled, the code should include :

```
btn.addActionListener(this);
```

Another possible case is if one variable name is used for several different components.

```
d)  add(btn = new Button("Ok"));
e)  add(btn = new Button("Cancel"));
```

The above code has two different Button components, but uses only one variable name. This case is also ignored.

When you consider the spaces between code Strings, it becomes more complex.

```
f)  Button btn = new Button("Ok");
g)  Button btn = new Button  ("Ok");
h)  Button btn= new Button("Ok");
i)  Button btn =new Button("Ok");
j)  Button btn=new Button("Ok");
```

Since the pattern matching method recognises a space as a character, the similar above code fragments require different patterns to be detected.

In order to make the pattern matching generic and reliable, which can detect all required components from the different style of user programs, all these possible cases need to be considered.

6.3.2.3.2 Component Event

The Java AWT provides two conceptual type of events; low-level and semantic-level. A low-level event is one which represents a low-level input or window-system occurrence on a visual component on the screen. Semantic events are defined at a higher-level to encapsulate the semantics of a user interface component's model. For semantic events, the source is typically a higher-level interface representing the semantic model, that is the event type relevant to the Dynamic User Interface tool is the semantic event. For example, a low-level event "window size changed" is not likely to be used to get the next screen or to display the text in a component.

Table 2 Event types and Listeners

	Event	EventListener
low-level	ComponentEvent (component resized, moved, etc) FocusEvent (component got focus, lost focus) InputEvent - KeyEvent - MouseEvent ContainerEvent WindowEvent	ComponentListener FocusListener KeyListener MouseListener, MouseMotionListener ContainerListener WindowListener
semantic-level	ActionEvent (do a command) AdjustmentEvent (value was adjusted) ItemEvent (item state has changed) TextEvent (the value of the text object changed)	ActionListener AdjustmentListener ItemListener TextListener

Now, we need to consider components, their semantic-level event types, and event listeners. A component has more than one event type and it can register with the appropriate event listeners according to its event type. Event listeners receive notification only about the types of events that are registered. For example, a Button object will fire an ActionEvent when it is pressed; and a List object will fire an ItemEvent when it is clicked and an ActionEvent when an item is double-clicked. The ActionEvent is handled by an ActionListener.

Table 3 Components, event types, and event listeners

Component	Event type	Eventlistener
Button	onClick	ActionListener
Choice	onClick	ItemListener
Checkbox	onClick	ItemListener
List	onClick onDblClick	ItemListener ActionListener
TextField	onChange onEnter	TextListener ActionListener
TextArea	onChange	TextListener

6.3.2.3.3 Component Attribute

A component has a number of attributes (properties). For example, the Button component attributes are Height, Width, Label, Name, and so on. Since the goal of the component attribute object in the system is to share the same object attribute values and to reflect the object values on the linked components, the component types and attributes that are not relevant to getting or setting the text values such as the Height and Width attributes of Button component can be ignored. The component types, their attributes, and methods used in the system are in Table 4.

Table 4 Component types, attributes and methods

component	component attribute	get method	set method
Button	Label	getLabel()	setLabel()
Label	Text	getText()	setText()
Choice	AddItem SetItem	getSelectedItem() getSelectedItem()	addItem() select()
Checkbox	Label	getLabel()	setLabel()
List	Item	getSelectedItem()	addItem()
TextArea	Text AppendText	getText() getText()	setText() appendText()
TextField	Text	getText()	setText()

6.3.2.3.4 Event handing class

An event handling class should be generated for each window design. The functionalities of the event classes are:

- to dispose of existing windows
- to show new windows
- to get the content of a component attribute
- to save the component contents to an object attribute value
- to set the component contents of other component attributes which share the same object attributes

Then, how should the system create event handling code? The object-oriented concept makes things easier. The event code generation facility of the system can be employed in an object-oriented way by extending a class - inheritance. Inheritance is the ability to derive one class (subclass) from another (super class) and inherit state and behaviour from their super classes so that you do not have to write that code again. Inheritance allows you to reuse code in each subclass you create. The subclass can use just the items inherited from its super class as is, or the subclass can modify or override it. That is, an event handling class extends a static window design class, and adds event code to the components of the super class without re-coding those components.

However, declaring member variables as a private member needs to be restricted. The Java language supports four distinct access levels for member variables and methods: private, protected, public, and package. The following charts shows the access level permitted by each specifier.

Table 5 Access levels of Java specifier

Specifier	class	subclass	package	world
private	X			
protected	X	X	X	
public	X	X	X	X
package	X		X	

As in the Table 5, subclasses do not inherit the super class' private member variables. A private member is accessible only to the class in which it is defined. Since the event handling subclass uses the component names of its super class to generate event code, the window component variables must not be private in order to be accessed by the subclass.

6.4 Implementation

6.4.1 Structure Chart (Module Chart)

The system consists of 13 classes. Figure 19 shows the implementation classes of the system. A class is represented by a rectangular box with a solid line representing a class not having an interface window (frame); a bold line representing a class having an interface window; and a dotted line representing a class for a dialogbox.

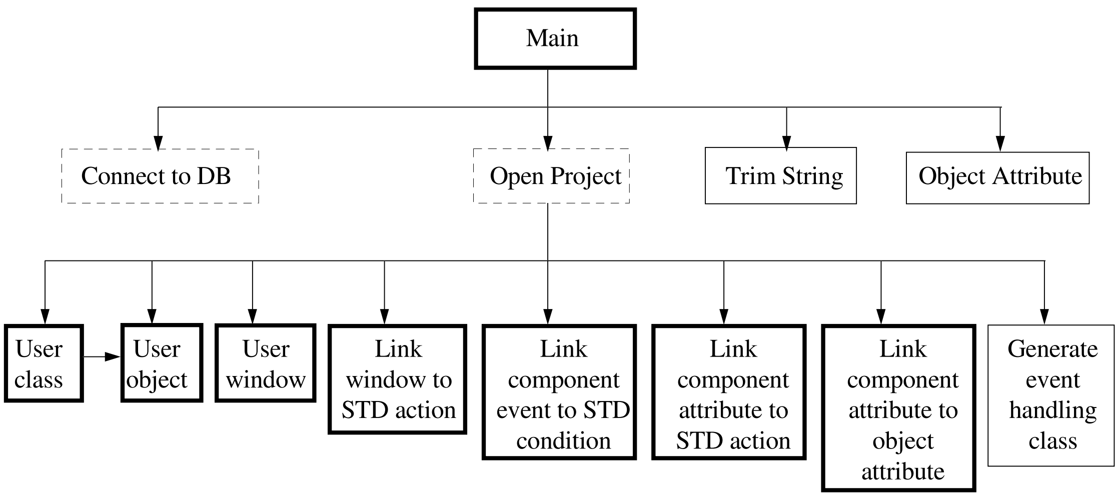


Figure 19 Structure chart of implementation classes

There are two special classes. One is 'TrimString' which is used to trim characters of saved data. The system uses the Oracle database server and JDBC (Java Database Connectivity) to keep information. One problem with JDBC is that String data saved in the database is retrieved not by its actual size but the size of the database column field it belongs to. That is, JDBC takes the field size which is defined when the table is created as a String size.

For example, suppose a Java program has a 'name' variable with 7 characters.

```
String name = "Smith";
```

Assume the 'name' column of a 'Customer' database table, which is supposed to hold the above Java variable, has been defined with a 30 character type when the table is created.

```
create table Customer(name char(30) not null,
                tel char(10), address char(50) );
```

When saving the ‘name’ variable of Java to the ‘Customer’ table, there is no problem; however, when retrieving the ‘name’ field of the ‘Customer’ table and setting it to a Java String variable, the result is:

```
String getName = "Smith                        ";
```

The result variable has 7 actual data characters and 23 extra spaces. So, comparing getName to name will always be considered different, which is not intended. So, a Java String variable needs to be trimmed of its spaces after being given the data. TrimString has a method to do this.

The other special class is ‘ObjectAttribute’ used to save the content of a component attribute to an object attribute of the database. This class is added to the generated event code by the event code generator.

6.4.2 Pseudo code

Pseudo code describes a brief algorithm used in a class. Since the ‘Event handler’ class has a significant algorithm, it will be shown in Appendix F.

6.5 Testing

Testing involves exercising the system to ensure that it produces the proper outputs and exhibits the proper behaviour for a wide range of inputs. Testing needs to be done in several ways. Three different types of testing have been conducted in the system - module, integration, and system testing. The module testing (unit testing) verifies that a module functions properly. Each program module is tested as a single program, usually isolated from the other modules in the system. Each module is tested before trying to integrate that module with others. When collections of modules have been unit tested, the next step is to ensure that the interfaces among the modules are defined and handled properly. Integration testing is the process of verifying that the components of a system work together. The entire system is viewed as a hierarchy of individual modules which are combined together for the testing. The next step is to make certain that the system works according to the user’s expectations. System testing involves verifying that a system meets its requirements.

Basically, the testing is conducted using a test sheet. A test sheet is a form consisting of items to be checked, of the methods describing how a corresponding item is checked, and of blank fields to allow the test result to be written on it. Different testing forms are used for different types of testing. For example, the test sheet of module testing consists of the list of functions and critical points requiring to be checked. The test sheet of the system testing uses real case examples - the Keele Ferries Booking System to form the sheet. After the test, the prototype was changed and redesigned several times. The test sheets and results made on the first prototype product are shown Appendix G. The results became valuable information for making the later prototype.

6.6 Developing the First Prototype to the Final Product

Applying the evolutionary prototyping approach leads to an iterative, and cyclical development process. The goal of the first prototype is to obtain an initial conception of the proposed system and to develop a prototype that makes it possible to test these conceptions on the basis of realistic examples. The result of the first prototype serves as a base system for the succeeding iterative process of later prototype development. Prototypes are successively elaborated toward the final product by solving problems exposed, by adding redefined or new functions, and by making it more complete and usable.

After completing the testing of first prototype, the following have been added to the second prototype (the final product).

Solve the problems of the first prototype exposed by the testing

- Null values: Some data must have values. The program has been changed so that null values are prevented.
- Redundant data: There is no restriction to entering the same data several times. A module which removes redundancy has been implemented.
- Update information: After the change of either user classes or user window designs, all linking data related to them must be deleted.

Complete functions

- Complete functions: Since the purpose of the first prototype is to make a prototype system quickly, some of the functions which are not urgent were skipped. For example, in the 'EventHandler' class of the first prototype, the event handling code for only the 'ActionListener' was implemented. Since all other listeners basically have the same algorithm, it could be delayed.

Add new functions

- File menu: Users can see a file in the text area, edit and save it.
- View menu: After linking information, it is impossible to memorise all the linked data. This function allows users to view and to delete the linked data.
- Compile and run: Users can compile and run the Java file.
- View screen : User can see the designed user windows inside the system.

7. Interface Development

7.1 Methodology

Developing usable interfaces is important for all kinds of system. GUI design is not a quick and dirty activity of prototyping a few screens. It requires a thoughtful, organised approach in order to make certain that the proper steps are carried out in the proper order. That is, well-defined methodology is required for interface development.

Designing the graphical user interface for the dynamic interface design tool follows the GUIDE process (Redmone-Pyle and Moore, 1995). Usability is the central design objective in the GUIDE process. The GUIDE approach is to first model the tasks that the user proposes to carry out in the application, and then model the business objects required to carry out those tasks; the user interface can be derived from that. The processes of every stage are integrated, i.e., the output of a previous stage becomes the input of the next stage. GUIDE is an iterative process - the initial design is followed by successive revisions and redesigns until an acceptance design is produced.

The schematic overview diagram of the GUIDE process is shown in Figure 20. Boxes represent processes, and lines represent how products are produced by one process and input to another.

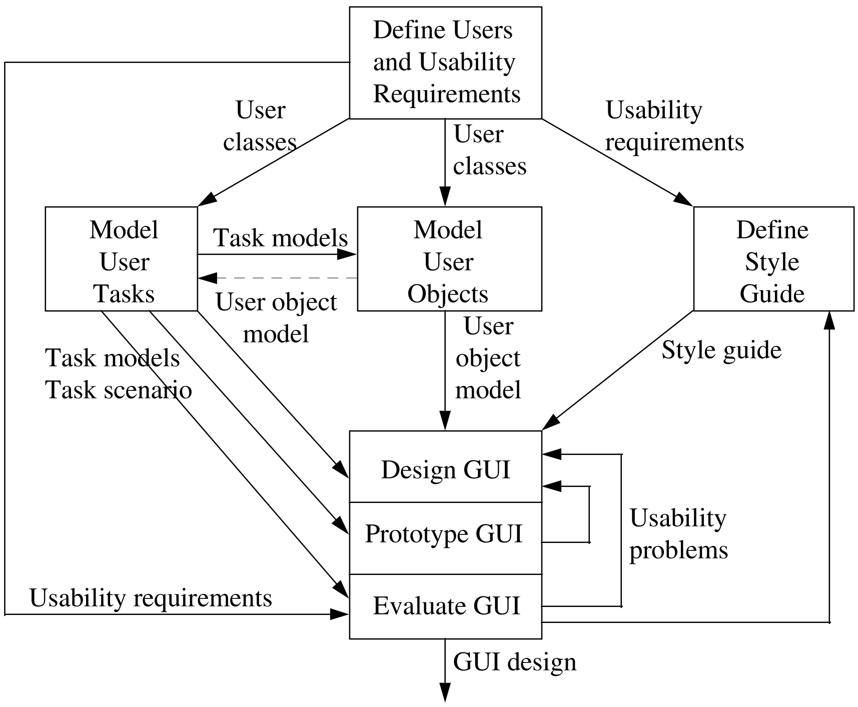


Figure 20 Overview of GUIDE process - extracted from GUIDE (1985)

Brief descriptions of the process and products of each stage are as follows:

Define users and usability requirements

The process of identifying the end-users and their special characteristics; specifying their usability requirements. The products are

- user class descriptions : describing the different types of end-user
- usability requirements : each with one or more evaluation criteria and the level which the GUI must achieve to be acceptable.

Model user tasks

The aim of this process is analysis and modelling of user tasks. The products are

- task models : what tasks a user will perform, hierarchical task analysis showing the generic hierarchical structure of each task
- task scenarios : example of individual instances of the task, which are suitable for prototyping.

Model user objects

The aim is identifying the objects and object relationships in the system, and actions that people perform on these objects. Important work to be done at this stage is dynamic modelling. If the objects has significant dynamic behaviour, the dynamic behaviour of the objects in the system should be modelled. The products are

- one or more user object model
- a glossary of user terms
- Statechart : dynamic model diagram

Define style guide

The aim is for the GUI to have a consistent style. The product is

- an application style guide defining GUI standards to be used for the system.

Design GUI

The aim is to design an initial GUI which supports user tasks and present the user's objects clearly. The products are

- window design, including specification of interactive behaviour
- window navigation design : the relationship between windows, shows how users can move from one window to another

Prototype GUI

The aim is to investigate the usability (and feasibility) of the proposed GUI design and the validity of the task model, user object model and style guide. The products are

- a working prototype GUI

- revisions to the GUI design

Evaluate GUI

The aim is to evaluate the GUI to check whether it satisfies all usability requirements and adequately supports all tasks. The products are

- an evaluation of the usability of the GUI design and prototype
- usability problems
- proposed changes to the GUI design

As described, the overall GUIDE process is a rich process. Performing activities takes time and costs money. So, the GUIDE processes are customised and the individual techniques of a process are also cut down to a simplified form. For example, defining the style guide, and the glossary of user terms will not be covered.

7.2 User and Usability Specification

7.2.1 User Classes

In some systems, developers know who will be the real user and can collect information through questionnaire, observation, and interviews with the users. However, the user of this project cannot be specified explicitly. So, only generic and heuristic identification of potential user classes and their usability requirements are possible.

The system is intended primarily for GUI system developers. This is a very broad definition and covers anyone who is responsible for the specification, development and implementation of GUI system development. For instance, system analysts, designers, and students working on interface design projects could be the users.

Table 6 User characteristics table

GUI system developer	Notes on characteristics	Requirements implied
Type of user	direct	
Mandatory/discretionary	discretionary	usable, efficient, and effective
existing computer experience and skills	experienced, professional expert	Flexibility and efficiency of use fast access, short cut, tool bar, abbreviations, reduce tedious verification
Education/intellectual abilities	good analytical and design skills, highly educated	
Motivation /goals	save development time developing a good system	
Training on system	experienced, learn quickly	
Task knowledge needed	Knowledge of system development, GUI design	
Other system used	various system experience	Familiarity with a large number of user interfaces, support conventional interface styles

7.2.2 Usability Requirements

Usability requirements are mainly extracted from Nielsen's check-list of ten principles.

Table 7 Usability requirements

Usability attribute	Description
Fast access	accelerators
Consistency and standards	same words, situations, or actions mean the same thing.
Visibility of system status	show appropriate feedback within reasonable time
Speak the users' language	
Critical performance area	reduce overhead
Minimise user memory load	support investigation of system status
Error prevention	
Error messages	expressed in plain language (no codes), suggest a solution

This usability requirements in Table 7 is used as a basis for designing, and evaluating the interface.

7.3 Task Analysis

Task analysis is a general purpose requirements analysis technique which helps you to identify required system functionality as well as to design the user interface.

7.3.1 Task Model

The main goal is to get a dynamic user interface which does window navigation and component content sharing. The following tasks can be identified.

- identify classes and class attributes
- identify objects and object attribute values
- dynamic modelling - identify dynamic behaviour of a class and represent it. (draw a STD for a class)
- design static window interface
- link static window data to STD
- generate event code

Figure 21 and Figure 22 give the hierarchical model of the tasks. Some tasks will be entirely performed by the person without any interaction with the computer system such as "identify class"; other tasks will be performed by the person, but with some interaction with the computer system such as "draw a STD"; and others will be performed entirely by the computer such as "generate event handling code". Subtasks requiring computer support are shown with a bold outline.

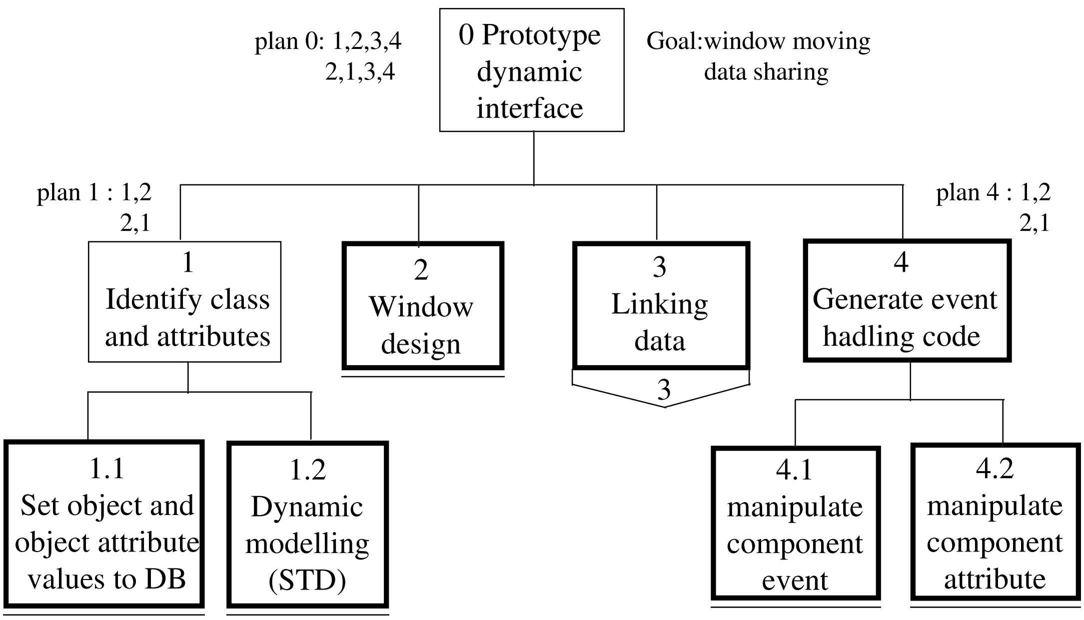


Figure 21 Task model - top level

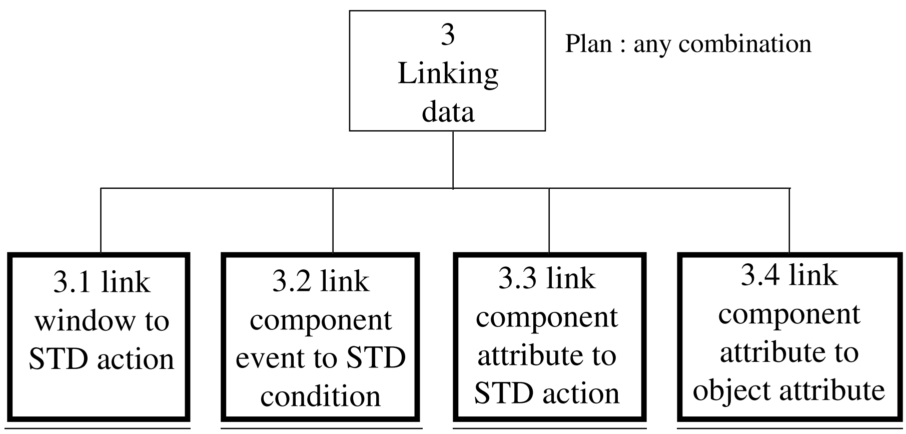


Figure 22 Task model - linking window to STD

7.3.2 Task scenarios

Scenario 1 - Developing dynamic user interface of Keele ferries booking system.

Background: A developer is developing a booking system for Keele Ferries. In order to make a more usable first prototype on system user interface design, the developer wants a prototype of the dynamic interface for scenario 1,2, and 3 (described in Chapter 5 - Keele Ferries Booking System).

Developer : Identify classes and class attributes in the system at the analysis stage, and make object model table (Chapter 5).

Developer : To capture the behaviour of the system, the developer does dynamic modelling. All events and actions for each class are analysed, and then represented by a state transition diagram. STDs have been drawn with the Mayerl STD editor (Chapter 3.3 - Mayerl STD editor).

Developer : Designs user interface (static window design) on paper first, and implements a prototype of the user interface using Java AWT (Abstract Window Tookit). Windows designed are added into the project.

Developer: Consider which window attributes (window name, component event, and component attributes) need to be linked to which STD attributes (state activity, transition activity) to do the intended work (window navigation, data sharing). One concrete example of intended work is that when the user clicks the ‘booking’ button on the ‘searching’ window, the ‘booking’ window should be popped up; the ‘searching’ window should be disposed; and the content of ‘port from’, ‘port to’, ‘date’ components of the ‘searching’ window should be copied to the ‘from’, ‘to’, and ‘date’ of ‘booking’ window, and saved to ‘from’, ‘to’, and ‘date’ object attributes of the ‘booking’ object. Link windows to STDs .

Developer : Generate event code, compile and run the event code and see whether it works properly.

7.4 Class Modelling

7.4.1 The Classes of Objects

The system consists of classes of objects which correspond to the concepts being dealt with and their inter-relationships. The identified classes of the DUIDT are as follows:

- project
- user classes
- user windows

In object oriented analysis, a class is described by its attributes and methods (behaviours). These will be explained in detail in the next section.

Project

The basic unit of work in the system is a project. A project has a number of classes and user windows needed to be handled together.

Class attributes	Class actions
project name list of classes list of screen designs	create a project open a project

User Classes

A user class is something thought of by the user as a class in a project system, commonly: a business system class, a computer system class or device, and a container class.

Class attributes	Class actions
class name list of objects list of class attributes (state) list of class actions (behaviour) State transition diagram (STD)	create class attributes retrieve a class and its class attributes draw a STD update a STD link class STD to window add to a project remove from a object

User Windows

Class attributes	Class actions
window name list of components list of component events list of component attributes event handling code	add components to a window remove a component from a window detect components in a window view window dispose a window link to STD add to a project remove from an object generate event code compile and run

7.4.2 Class Relationship

The user object model diagram is shown in Figure 23

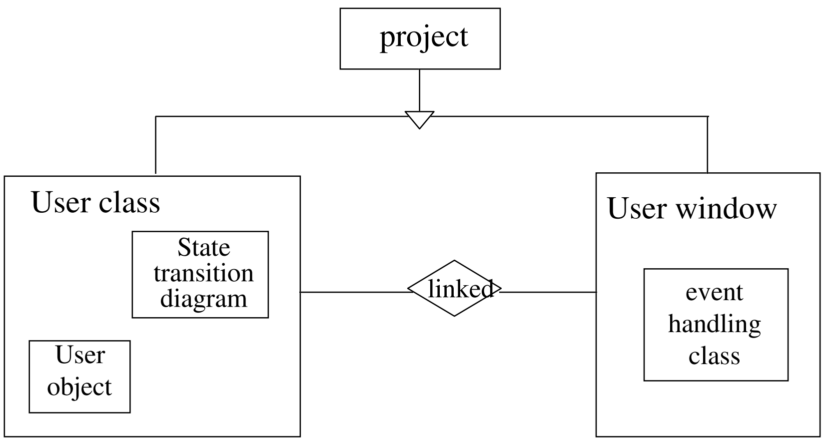


Figure 23 Object relationship

7.4.3 Action-Object Matrix

This matrix is to show how actions affect objects. The action-object matrix provides a useful way of checking the scope and complexity of actions. Most user object actions only affect one user object. However, where an action does affect more than one object, this is significant for GUI design. For example, the ‘linking data’ object gets information from several other objects, so the change of update of other objects will cause the change of ‘linking data’ object as well. The following two tables show an example of a matrix. D indicates ‘delete’; R is ‘read only’; U is ‘update’; and C means ‘create’.

User object

Action	User Class	User Object	User Screen	Linking data	Event class
Object.create	R	C			
Object.update		U		D	D
Object.retrieve		R			
Object.link		R	R	C	C

User screen object

action	User Class	User Object	User Screen	Linking Data	Event Class
Screen.create			C		
Screen.update			U	D	U
Screen.link		R	R	C	C

This should be referenced when implementing the tool. That is, the program must support a function module which updates all related other objects when an object is changed.

7.5 Initial GUI design - paper sketches

7.5.1 View Definition of Objects

Before starting an initial design of a window, the view of each object should be defined.

Project

A project class does not require any single view. It is just used to identify the unit of work.

User Window

All information about a ‘User Window’ object can be displayed in a single view. However, there is a special case which requires a different view. When it is used for linking to STD information, different parts of the ‘User Window’ object are viewed for different tasks. In addition, the generated event handling class also occupies a single view.

User Class

This object requires a single view for itself, and for each user object which is its attribute. A STD attribute does not occupy a whole window. Instead, a view of a part of the STD information is required for the relevant task. That is, different views of the same information are used for different tasks.

7.5.2 Initial interface design - paper sketches

GUIDE recommends that the initial design starts to take shape as paper sketches because of some advantages of pen and paper design. It is quick, it is recognisably only an unfinished draft, and there is little resistance to making changes and event redrawing completely. Figure 24 shows the sketch of 'UserClass' object.

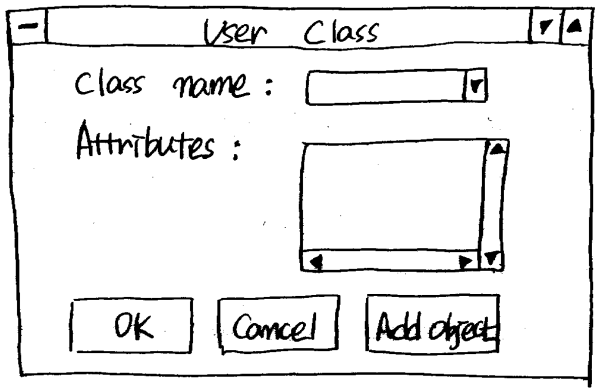


Figure 24 The initial design of 'UserClass' object

7.6 GUI prototyping

The interface development of a project has three increments, each involving GUI design, prototyping and evaluation.

7.6.1 Prototype 1 (Paper)

7.6.1.1 GUI design 1

An example of the initial design sketched by hand is shown in Figure 24.

7.6.1.2 Prototyping 1

The objectives of the first prototyping are to validate the main windows requirements, and their approximate layout; to validate the main navigation paths between windows; and to validate support for various scenarios of the central task.

The prototype is composed of a paper sketch of the windows, and connected by window navigation links drawn in pen. Investigation and evaluation was performed on both an individual window and linking. For example, whether a window includes all needed components, there are too many components, a different object is separated physically, and so on. Linking investigation is whether call and return relationships between windows are correct, all necessary linking has been made, there is unnecessary linking, and so on.

7.6.1.3 Evaluation 1

There are several different approaches to usability evaluation, and the first part which should be done is to decide what is feasible and most appropriate in the project circumstances. Main types of evaluation are as follows:

- review by human-computer interaction expert
- heuristic evaluation : end-user evaluation
- user testing (or observation of user) : Survey of user attitudes or perceptions.

The heuristic evaluation method identifies many problems in a GUI by giving reviewers a check-list. You ask the reviewers to examine each part of the interface referring to the check-list. This greatly reduces the cost, so, it is feasible in low-budget projects. For this reason, the heuristic evaluation is used as an evaluation method in parallel with user testing with scenarios.

Nielsen's ten principles with some modification are used as a check-list for heuristic evaluation. This is already defined in Table 7 on page 46 as a usability requirement. An evaluation sheet having these principles as a check-list item and a blank column for making notes about the check-list item is made for an evaluation. The evaluation sheet after the evaluation of the first prototype is in Table 8 Prototype 1 evaluation sheet. However, this sheet does not have all usability requirement items described in Table 7 because some attributes are not possible to be evaluated in this stage.

Table 8 Prototype 1 evaluation sheet

Usability attribute	Notes
Fast access	Toolbar supported, but no shortcut
Consistency and standards	many inconsistent cases e.g. both exit and cancel are used for same meaning.
Visibility of system status	most windows do not have any place for showing status of system
Speak the users' language	Ok

Another evaluation method used is user testing with Keele ferries booking system scenarios (see Section 5.3 for scenarios). A user attempts to perform three scenarios of the tasks of the Keele Ferries Booking System by navigating windows, so the user could see the current active window. Of course, this is not very realistic, but helps the designer to evaluate the linking of the interface.

As prototyping proceeded, the original window sketches were replaced with updated versions of windows; and linking has also been changed. Some missing links have been added and some unnecessary links have been deleted.

The problems found are:

- Lack of visibility : The system should keep the user informed about what is going on, through appropriate feedback within reasonable time.
- Shortcut missing : Accelerators may often speed up the interaction for the expert user so that the system can cater for both inexperienced and experienced users.

One significant area of discussion and revision concerned is the window design of the 'event code generator'. As in Figure 25, the purpose of this interface is to get a user window name to generate event code for it. However, during the user testing of the first prototype, it was realised that this interface is not needed because all event code for the windows in the project should be generated whenever the user wants to.

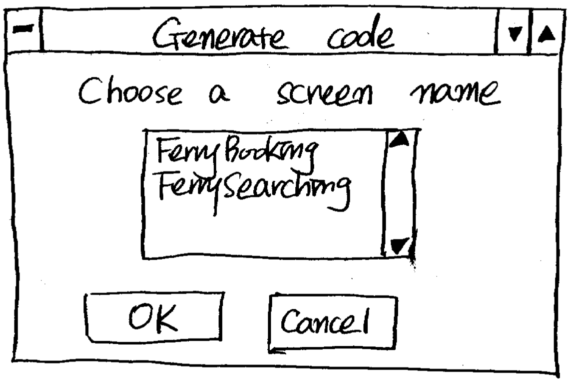


Figure 25 The initial window design of 'Event Code Generator'

7.6.2 Prototype 2 (Java AWT)

7.6.2.1 GUI Window design

The objectives of the second design are to validate the detailed window content and layout; to explore usability by allowing the user to interact with windows in the prototype; and to support evaluation by fairly realistic user testing. Sample test data is provided to look at and evaluate the interface.

Windows were redesigned according to the result of the first prototype evaluation. The following table shows problems found and their solutions.

Table 9 Problems and solutions of prototype 1

Problem	Solution
System status is not visible.	There are two possible solutions. One is using a status bar on the bottom of a window; another is making an individual error message dialogbox. The status bar has been chosen to reduce unproductive work time (clicking 'Ok' button of an error message dialogue box). The statusbar might be less noticeable than dialogbox, but could save time.
Consistency	Fix a terminology and use it.
Shortcut missing	Unfortunately, this is not settled. The first prototype has been implemented with Java 1.0, and it does not support shortcuts.

In addition, some new characteristics were added to the second prototype. For example, in the main menu some menu items should not be activated before some other menu items are selected. The greying of components is used for this. The components which are not available in the current state should be greyed out so that they are prevented from being selected. This is one way of preventing possible errors.

7.6.2.2 Prototyping - Java AWT 1.0

This time windows were designed with Java 1.0 AWT components. The window design of 'UserClass' object is in Figure 26.

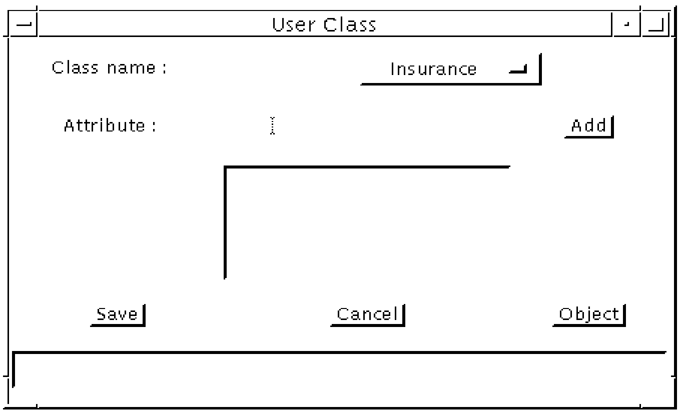


Figure 26 'UserClass' window - second prototype

7.6.2.3 Window Navigation Diagram

Window navigation is concerned with how the user opens new windows and transfers focus from one window to another. A window navigation diagram shows a number of windows and the navigation paths between them. The window navigation diagram of the second prototype is shown in Figure 27. A solid line represents a frame, and a dotted line means a dialogbox.

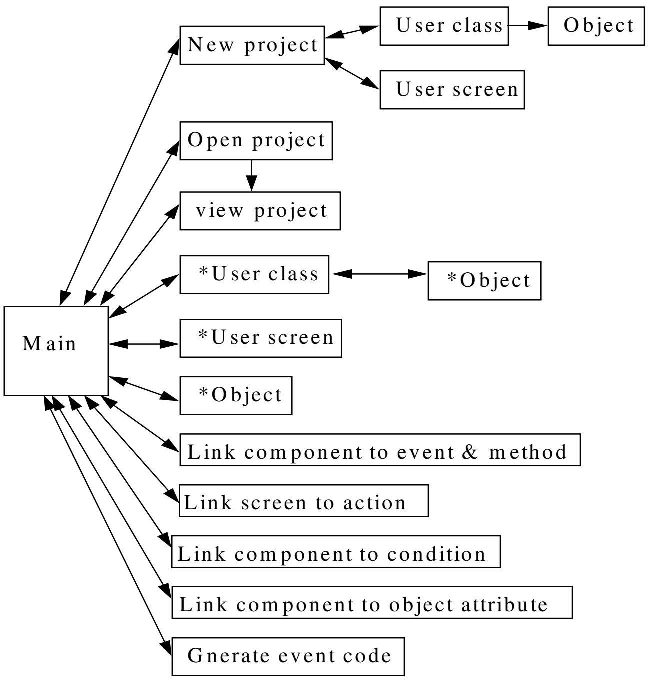


Figure 27 Window navigation diagram - Prototype 2

7.6.3 Evaluation 2

The evaluation methods used are the same as evaluation 1 - heuristic evaluation with an evaluation sheet, and user testing with scenarios.

Table 10 Prototype 2 evaluation sheet

Usability attribute	Note
Fast access	Toolbar supported but not shortcuts
Consistency and standards	Ok
Visibility of system status	Ok
Speak the users' language	Two questions should be answered. 1. Which one is more appropriate term either 'user screen' or 'user window'? 2. Some users might not know the meaning of STD.
Minimise user memory load	Problem. After linking data, users can not remember all linked data.
Error prevention	Greying might help error prevention.
Error messages	using system error messages, need to change it to plain language

Exercising user testing with the scenarios led to a number of small changes in window content and layout.

The problems found are:

- not enough error messages
- Some menus are in the wrong place : User Class, User object, and User Screen items are located on the File menu. This should be reorganised.
- Linking interfaces (e.g. linking window to STD action) are too big and complex and might cause confusion (see Figure 28). Currently the linking interface design shows both state and transition information of the STD. If the user wants to link windows to both state and transition at the one time, this interface might be convenient. However, if not, it will cause a mistake.

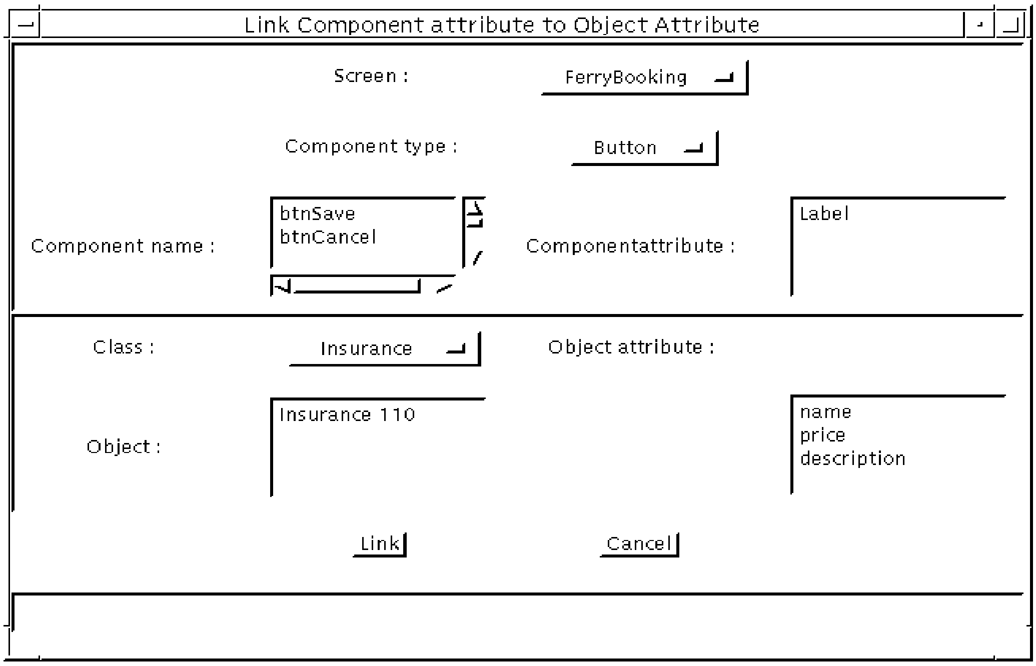


Figure 28 Linking window to STD action - Prototype 2

7.6.4 Final product

7.6.4.1 GUI design - Java 1.1

The objectives of this design stage are to confirm that modifications have successfully resolved usability problems identified in earlier prototypes; and to support informal exercising of all task scenarios.

The problems provoked in the second prototype evaluation are considered and redesign solutions have been made.

Table 11 Problems & solutions of Prototype 2

Problem	Solution
no way of investigating data in the system	add ‘View’ functions showing the linked data in the system
poor design of ‘linking data’ interface	Show only one set of information (either state or transition) at one item and allow the user to swap them. After data linked, all selections which have been made are deselected so that unintended links are prevented from being saved by mistake
shortcuts missing	Though adding shortcuts was suggested in the first prototype evaluation, it was not implemented because of the limitation of implementation language. Fortunately, Java 1.1 supports shortcuts so they were added to the system.

7.6.4.2 Windows

The user interface of the final product are as follows.

Figure 29 is the main screen of DUIDT. All functions are shown on the main menu and a tool bar is also supported. User can open and edit files in the text area of the screen.

Figure 29 DUIDT main window

Figure 30 is the interface of ‘window design’ object. Users can add (or remove) a window design into (or from) the list of project. The users can also view the appearance of window design.

Figure 30 ‘User Window’ interface

Figure 31 shows a 'linking window to STD action' interface.

Figure 31 'Link window to STD action' interface

7.6.4.3 Window Navigation Diagram - the Final Version

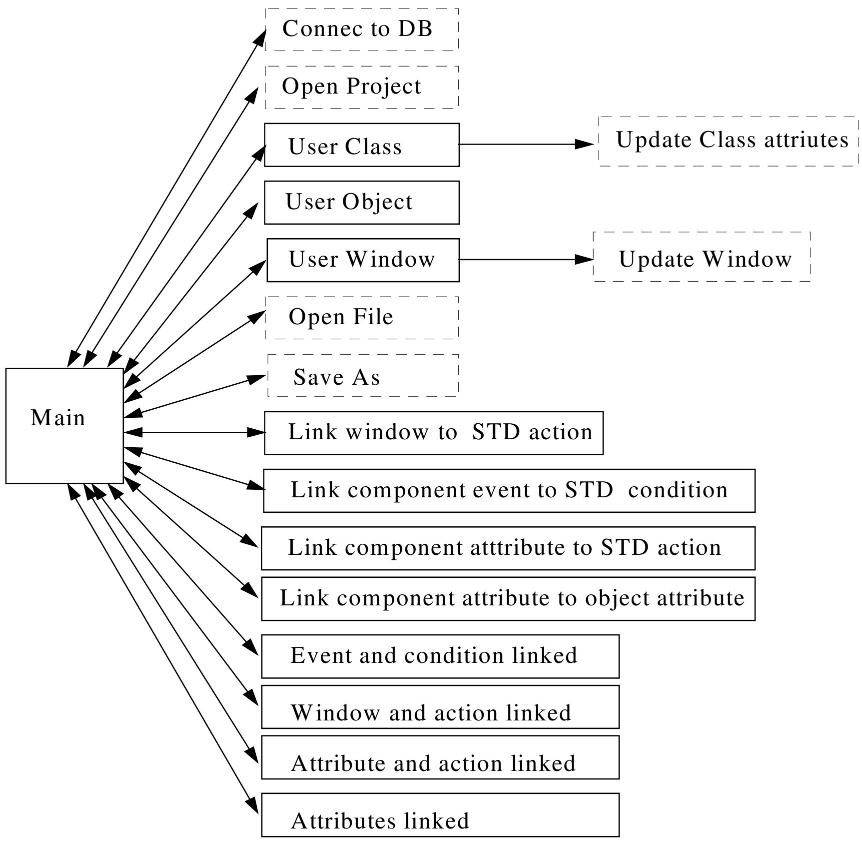


Figure 32 Window navigation diagram of DUIDT

PART IV CONCLUSION

8. Result & Critical Evaluation

The Dynamic User Interface Design Tool (DUIDT) has been successfully implemented based on the conceptual model suggested. In order to use the tool, STDs and windows need to be designed beforehand. The tool can read STD information from a database, which is saved by the STD editor, and recognise components from a user window as long as it is implemented using Java. Then, the users are allowed to link windows to STDs to add dynamics. A link can be added or deleted whenever a class STD or a user window is created or removed; or whenever class attributes are modified. Consequently, different event code will be generated because it is based on this linking data.

The DUIDT interface consists of three parts - menu, text area, and statusbar area. The main menu and toolbars show functions offered by the system so that a user selects what they want to do. The user can open a file in the text area, and edit and save it. The statusbar shows the status of system such as whether database is connected, the project name opened, or error messages.

The final product of the DUIDT is a number of event handling classes (a window interface having events and actions) which extends their user window classes (a window interface with a set of components but not their events and actions). The project is the basic unit of generating event classes. That is, an event handling class is generated for every window design class of the project whether or not they are linked to STDs. Each event class consists of a constructor, a number of event handling methods, and a main method. The constructor registers components to listen to an event; and main method is to override the super class's (static window) main method. If a window is not linked to any data, the event method will have only the event handling method shell without actual code. The intended work of the event classes is to show windows, to dispose of windows, to get contents from a component, to save it to a database, and to set it to other window's components having the same object attribute. In addition, one more class is also generated for a project to instantiate all event classes generated. This simply creates an instance of each event classes as a class variable so that a window can be shown or disposed only once because there is only one copy of the variable, associated with the class, rather than many copies of the variable associated with each instance of the class. The class variables can be used even if the class is never actually instantiated because class variables are initialised when the class is first loaded. The example event handling classes and an instantiation class for Keele Ferries Booking System are shown in Appendix H.

The generated event code has been compiled and run with the JDK virtual machine. After the system testing of the event code of Keele Ferries Booking System example (see Appendix G), the following results are noted.

- The window navigation during run time is exactly correspondent to the window navigation diagram.
- Window navigation and component content sharing are independent of each other. That is, an event action might cause only window moving or only content sharing or both at the same time. This characteristic also implies that component content can be shared not only between windows but also inside a single window.

- The content of a component can be shared by several other components of the same window, or by those of many other windows as long as they belong to the same object attribute. Therefore, an action on a component (such as change of content) affects all other related components.

It is a necessary and main part of the system to link windows to STDs to generate event handling code. However, this is time-consuming, difficult, and requires a lot of effort from the user. Therefore, how the overhead of linking window to STD data can be reduced as much as possible was the main area of concern during the project. In order to generate event code for window navigation and content sharing, four links need to be made. That is, the main performance of the system depends on the overhead of linking data. If a tool has a complicated interface for linking and is difficult to use, performing work is slow and can actually make the development process much less efficient. Developers may resist using the tool if it makes their work harder than it would have been to simply code the event handling method. So, the effort trying to improve the interface for the linking of data has been especially emphasised in interface development. The system design was also affected by this problem. Users often make an unwanted linking by mistake and will need to delete it. The system should allow the user to link and delete data freely. This function is added in the second prototype of the system.

The restriction of the system is that the component member variables of a user window design should not be private so that event handling class (subclass) can extend the user window class (super class). If a member variable of a super class is defined private, the generated event handling subclass cannot be compiled because it cannot access the variable of the super class.

9. Conclusion

Building a GUI design for a window-based system development is still a stimulating challenge. In addition to simply designing layout -the representational aspects- an interface designer must link the screens and text to the functions of the system, and sequence their screens in order to describe operational aspects of an interface. Therefore, the development of GUIs can be simplified and aided by using the appropriate tool. While static user interface design is supported today by a wide range of interface design tools, there are few dynamic user interface tools available. The purpose of this project was thus developing a dynamic user interface tool which supports representing the behaviour of interface from a high level specification.

In this project, I have suggested a conceptual model showing that how dynamics can be added into the static interface, and implemented the Dynamic User Interface Design Tool (DUIDT) based on the conceptual model. This conceptual model is based on object-oriented ideas. The organisation and behaviour of user interface is mainly object-based. Defining user objects tends to encourage the design of a separable user interface, where the appearance and behaviour of interface components is only loosely coupled to underlying system functionality. This yields benefits in system construction, maintenance and enhancement. The conceptual model uses a state transition diagram for representing dynamics, and addresses two functional behaviours - window navigation and component content sharing. DUIDT achieves these goals with two major features: linking static window designs to STD conditions and actions, and linking windows to object attributes. It allows an extension of the representation (static interface) so that dynamics can be added to it; or of functionality of an earlier prototype already having some behaviours, by generating object-oriented code for an application window design which can be extended by writing subclasses.

The tool has been demonstrated with a business system example - the Keele Ferries Booking System, for giving tips about how the tool could be used and for proving it works. User windows and STDs are designed based on the task analysis of a business domain. In order to create a dynamic interface, the required linking was made by the user. The generated dynamic interface simulates performing task scenarios during runtime. For instance, if a ‘booking’ button is clicked on the screen, the ‘booking’ window is popped up; or if customer details such as customer name, address, etc. are entered on the ‘booking’ window, this information is automatically copied to the customer information fields of ‘holiday sites booking’ window when the related action is triggered.

The programming facility of the DUIDT automatically generates event handling code or at least part of the event handling methods. This is helpful because generated code is typically more consistent, and coding standards such as naming conventions and standard headers and comments are maintained automatically. Also, generated event code contains fewer bugs, and can be used for further programming during application development. In comparison to conventional techniques - programming by hand, this not only saves development time but can provide a level of consistency across the different interfaces produced and allow nonprogrammers and human factors engineers to contribute their expertise.

In terms of the software development life cycle, the developers are able to run the kind of investigations which would normally be carried out much later in the life cycle of the project. There are cases where the resulting behaviour of a interface does not meet the designer’s expectations even though the static forms are satisfied. So, the correctness of its dynamic behaviour still needs to be confirmed. End users can evaluate each prototype and provide feedback, increasing their satisfaction with the final product.

The tool would be more useful when it can be used in combination with static design tools for building user interface in the system development. A static interface designer defines the internal structure of a system by declaring the graphical appearance such as its position, colour, height, layout, and so on. Then, dynamic behaviour can be added on this designed interface. For example, an interface designer might use Semantic Cafe to design the Java static interface, and DUIDT for generating the dynamic interface. This will save the designer's time and make his or her work easier by reducing time-consuming coding. Besides the static interface builders, DUIDT can be used in cooperation with other kinds of interface development tools such as the STD editor used in this project to draw a state transition diagram.

To conclude, the results of this project suggest that the introduction of such tools into a software development environment would improve software development processes but not at a cost of making the interface more annoying. However the tool is not a general purpose visual state diagram programming system. Rather, it offers a set of domain specific, high-level abstraction services for creating a dynamic interface. The design method for developing a dynamic tool proposed here tries to overcome some of the problems found in current practice of user interface design, i.e., the lack of behavioural design. This is done by making a stronger connection between the behavioural design and the user interface design, by reuse of static interface design (by using inheritance of object-oriented concept), and by automation of the translation between behavioural design specifications and user interface layout specifications.

Currently, the tool uses textual linking in order to link windows to STD information. Textual linking is less understandable than graphical linking. A graphical display has the advantage of letting you grasp aspects of parallel execution more immediately and more accurately than a textual interface. Developing a graphical tool which supports the linking of static user interface design and state transition diagram in a graphical manner would have an obvious advantage. This will make linking easier, faster, and more understandable than textual linking.

Another suggested future work is developing an integrated methodology for both system and interface development. There are many system development methodologies, and a few interface development methodologies. However, there are no applicable methodologies which support processes and guides for both system and interface development. Most current system development methodologies do not deal with the interface at all. The GUIDE methodology used in this project for the interface development suggests the linkage between system and interface development and some processes needs to be done, but it is not enough as a complete system development methodology. That is why the functions and interface of the system had to be developed independently with different methodologies. Since most modern applications use graphical user interfaces, it would be worthwhile developing an integrated development methodology.

Bibliography

Bischofberger, W. & Pomberger, G., Prototyping-Oriented Software Development:Concept and Tools, Springer-Verlag,1992.

Budgen, David, Software Design, Addison-Wesley, 1994.

Cornell, Gary & Horstmann, Cay S. Core Java, 2nd ed., Sun Softpress: A Prentice Hall, 1997.

Downton, Andy, Engineering the Human-Computer Interface, Dept of Electronic Systems Engineering, Imniv. of Essex, McFraw-Hill, 1991.

Eberts, R. E., User Interface Design, Prentice-Hall, 1994.

Esteban, O. & Chatty, S. & Palanque, P., “WHIZZ’ED: A Visual Environment for Building Highly”, Human-Computer Interaction Interact ‘95, Chapman & Hall 1995, p. 121-126.

Flanagan, D., Java in a Nutshell: A Desktop Quick Reference, 2nd. ed., O’REILLY, 1996

Harel, David, “Statecharts: A Visual Formalism for Complex Systems”, Science of Computer Programming, 8 (North-Holland, 1987), p 231-274.

Kimgour, A. & Earnshaw, R., Graphics Tools for Software Engineers, the Press Syndicate of the University of Cambridge, 1989

Lamb, David A., Software Engineering: Planning for Change, Prentice-Hall International Editions, 1998, p. 109.

Lauridsen, O., “Generation of User Interfaces Using Formal Specification”, Human-Computer Interaction Interact ‘95, Chapman & Hall 1995, p121-126

Newman, William M. & Lamming, Michael G. Interactive System Design, Addison-Wesley, 1995.

Pfleeger, Shari L., Software Engineering:The Production of Quality Software, 2nd ed. Maxwell Macmillan Internaltilnal editions, 1991, p. 291.

Popping, M. & Szwillus, G., “Constraint-Based Definition of Application-Specific Graphics”, Human-Computer Interaction Interact ‘95, Chapman & Hall, (1995), p. 85-90.

Preece, Jenny, A Guide to Usability:Human Factors in Computing, Addison-Wesley, 1993.

Redmond-Pyle, D. & Moore, A., Graphical User Interface Design and Evaluation: A Practical Process, Prentice Hall, 1995.

Rumbaugh, J.& Blaha, M., etc., Object-Oreiented Modelling and Design, Prentice-Hall, 1991.

Seats, A. & Lund, A.M., “Creating Effective User Interface”, IEEE Software, July/August (1997), p. 21-24.

Shneiderman, B. Desiging the User Interface: Strategies for Effective Human-Computer Interaction, 2nd ed., Addison-Wesley, 1992.

Valaer, L.A. & Babb, R., “Choosing a User Interface Development Tool”, IEEE Software, July/August, (1997).

Vanhelsuwe L. and etc. Mastering Java 1.1, 2nd ed., SYBES, 1997.

Ward Paul T. & Mellor Stephen J., Structured Development for Real-Time Systems, Yourdon Press Prentice Hall, 1985.

Yourdon E., Modern Structured Analysis, Prentice-Hall, 1989.

http://java.sun.com/, (Java SUN web page)

http://java.sun.com/products/jdk/1.1/docs/api/packages.html, (Java JDK 1.1.4 Core API)

http://www.rational.com/uml/references/notation-guide.html/, (UML notation guide, Rational software corporation)

http://www.java.sun.com/ docs/books/tutorial/, (Java Internet Tutorial)

Appendices

Appendix A Statement of Purpose

System : Dynamic User Interface Design Tool

Description :

This tool take the output from a state-transition diagram editor; gets user windows from the user and finds components which are linked to event types and methods; links conditions to window events and actions to windows; and generates a class which extends an existing static user interface design to add dynamics.

Input
User classes and attributes User objects and attributes The output from STD Editor User window design

Process
link window event to STD condition link window to STD action generate event handling classes

Output
Java event handling class

Responsibilities :

- get user class attributes, objects, and object attributes from the user.
- get user window design from the user and identify components.
- link the components of windows to the events of the state transition diagram.
- link the windows to the actions of the state transition diagram.
- link the component attributes of the window to. the actions of the state diagram.
- link component attributes to object attributes.
- generate event handling classes

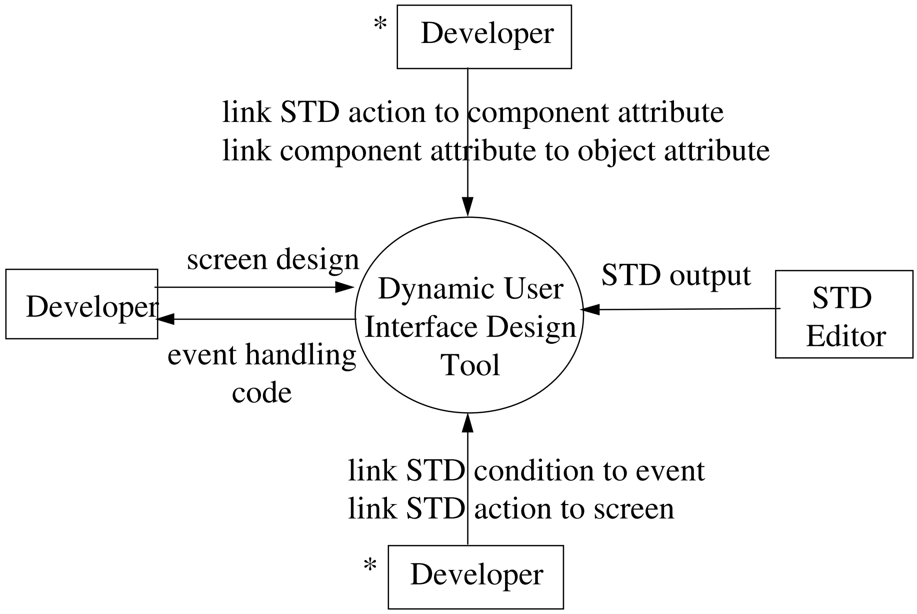
Exclusion :

- This tool is not responsible for static screen designs.
- This tool does not include STD (state transition diagram) Editor.
- This tool supports only Java language.

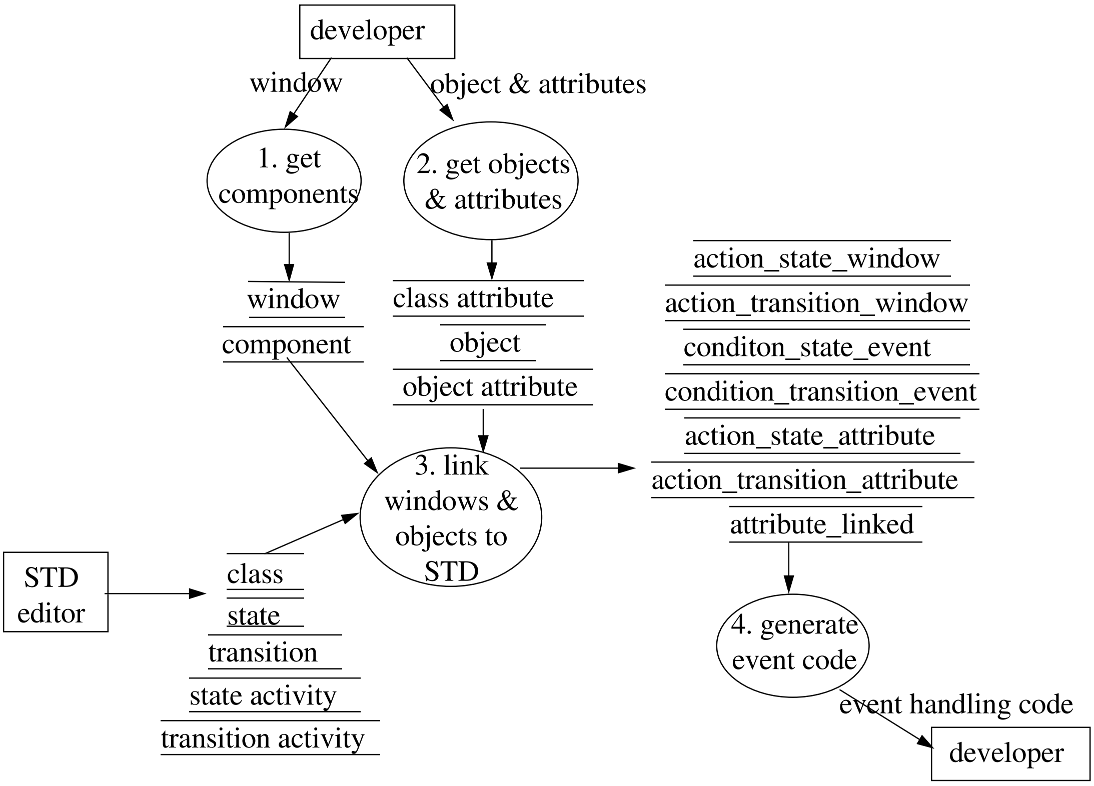
Assumption :

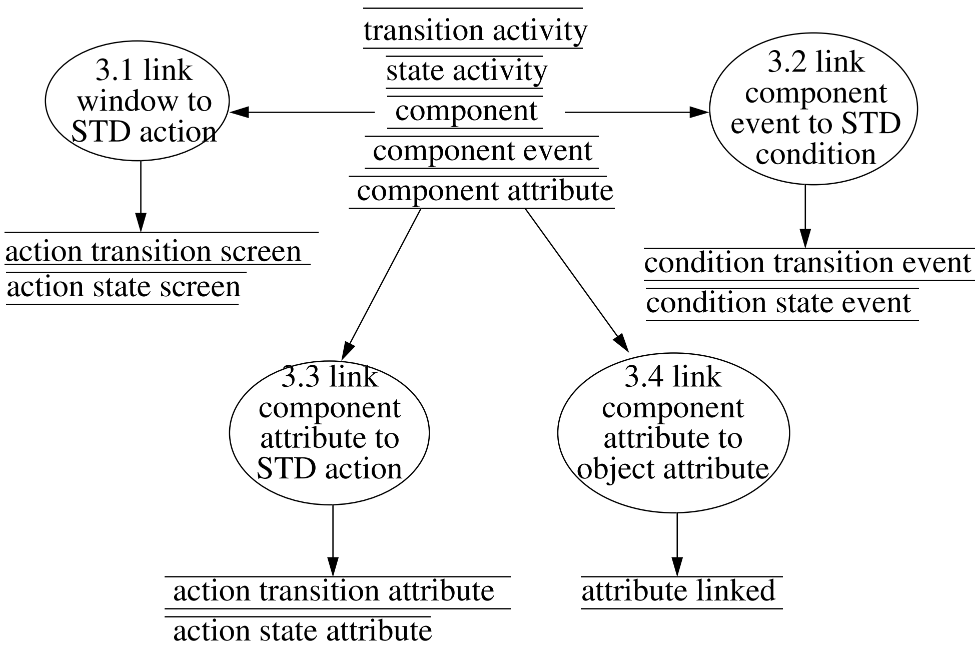
- User object and window name should be unique throughout the whole system.
- User should make state-transition diagram and user screen design, and link necessary screen information to STD information to generate event handling class.
- The user window design must be implemented in Java.

Appendix B Context Diagram



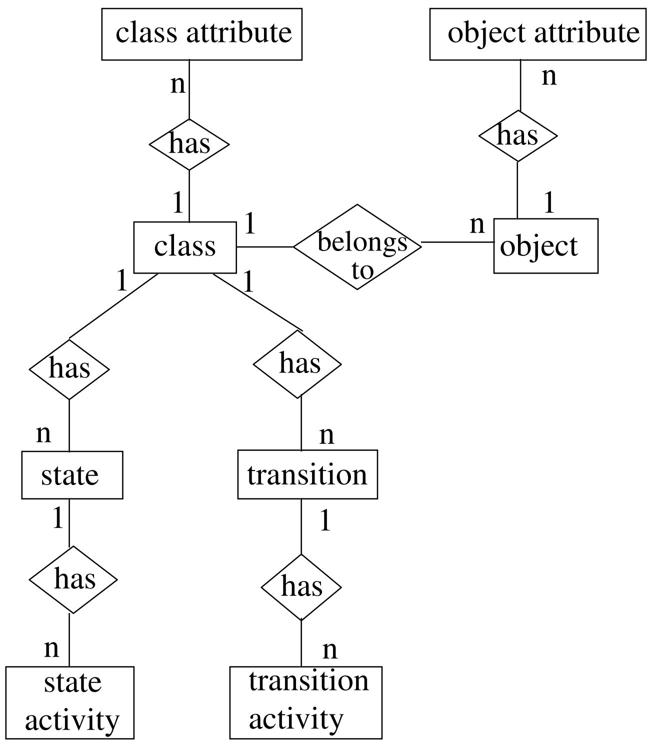
Appendix C Data Flow Diagram



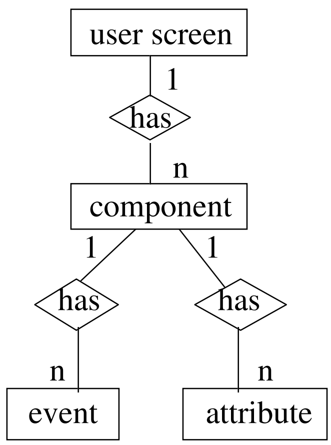


Appendix D Entity-Relationship Diagram

- **ERD of STD**



- **ERD of User Window**



- **ERD of Linking data of User window and STD**

Appendix E Data Structures

- Class

project_name	<u>class_name</u>

- Class_Attribute

class_name	class_attribute

f.k

- Object

object_name	class_name

f.k

- Object_Attribute

object_name	object_attribute	value

- State

class_name	state_name	parent	child	ssize	x	y

- Transition

class_name	transition_name	alignment	from_state	to_state

- Transition_activity

class_name	transition_name	condition	action

- State_activity

class_name	state_name	condition	action

- Screen

project_name	<u>screen_name</u>

- Component

component_name	component_type	screen_name

f.k

- Component_Event

component_type	event_type

- Component_Attribute

component_type	component_attribute	get_method	set_method

- Condition_Transition_Event

condition_name	transition_name	class _name	event_type	component_name	screen_name

- Condition_State_Event

condition_name	state_name	class _name	event_type	component_name	screen_name

- Action_Transition_Screen

action_name	transition_name	class_name	screen_name

- Action_State_Screen

action_name	state_name	class _name	screen_name

- Action_Transition_Attribute

action_name	transition_name	class_name	component_attribute	component_name	screen_name

- Action_State_Attribute

action_name	state_name	class _name	component_attribute	component_name	screen_name

- Attribute_Linked

object_name	object_attribute	component_attribute	component_name	screen_name

Appendix F Pseudo Code

Event handler class

1. event code part (event handling code)
read both transition event and state event, and get the unique event of screen
repeat
    handle transition event
        read the condition(s) of the transition event
        get the transition action(s) of the transition condition
        get the screen(s) linked to the transition action, and show it(them)
        get the screen(s) linked to the From State 'Exit' action, and dispose it(them)
        get the screen(s) linked to the To State 'Entry' action, show it(them)
        get the component attribute(s) and name(s) of the transition action,
            get the contents of transition component attribute,
            save the contents to the object attribute,
            set the contents to other component attributes
        get the component attribute(s) and name(s) of the From State Exit action,
            get the contents of the From State Exit component attribute,
            save the contents to the object attribute,
            set the contents to other component attributes
        get the component attribute(s) and name(s) of the To State Entry action,
            get the contents of the To State Entry component attribute,
            save the contents to the object attribute,
            set the contents to other component attributes

    handle state event
        read the condition(s) of the state event
        get the state action(s) of the state condition
        get the state action screen(s), and show it(them)
        get the component attribute(s) and name(s) of the state action,
            get the contents of the state component attribute,
            save the contents to the object attribute,
            set the contents to other component attributes
until no more component event

2. windows instantiation part
get all window names in the project
generate a class instantiating all window names in the project as class variables

Appendix G Testing Form and Result

Module testing

The test sheet of 'User Window' and 'Link Window to STD Action' class are following:

- User Window class

Item	Method	Result
All components in a window file are detected.	compare the detected components to component of file manually	Error: TextField components are detected as a Choice component.
The same user window design is popped up more than once.	Show the same window before disposing it	Ok
All user window design shown on screen are disposed when the object is disposed.	Press 'Cancel' button without disposing user window designs shown on the screen.	Windows still remains on the screen.
Does listbox show all user window design in the project?	Check database table.	ok
Does the program prevent the same screen is added to a project more than once?	Add the same user window design several times.	No, window are saved each time it is added. A lot of redundant data in the table.
The components are updated after a window design is changed.	Check the detected component after adding the redesigned window.	ok

- Test sheet of 'Linking Window to STD Action'

Item	Method	Result
The same data is not linked more than once.	Link same data more than once	Error: The same data linked several times
All components of a user window are shown.	Compare components of List to those of window file.	Ok
All STD information (state, transition, state activity, transition activity) are shown.	Check items shown on component using STD diagram printed.	ok
Entering null data is prevented.	Link without selecting some data	Error: null data inserting is not prevented.
When selecting a different class, the following state and transition are changed.	Select different items in the List component, and check the change.	ok

Integration testing

The following is the test sheet for the integration testing of 'Main' class and 'UserObject' class.

Item	Method	Result
Does proper project name passed from main to user class?	Print out project name on the screen whenever it is passed.	ok
While an instance of 'user object' class is shown, another instance should not be created.	Click 'User Object' menu item several times from the main menu.	The same window is shown as many as created.
After disposing the created instance, an instance ca be created again.	Click 'User Object' menu item after disposing the created window.	ok, but need to pay attention because of the problem of the second item.

System testing

In order to verify that the finished system meets its requirements, the real example data is used. The scenario described in Section 7.3.2 (the scenarios is about prototyping dynamic user interface of the Keele Ferries Booking System by performing its scenario 1, 2, 3 in Section 5.3) is used for system testing.

System testing is different from program testing because it does dynamic analysis. Static analysis is performed when the program is not actually executing; however, dynamic analysis is performed while the program is running. Since the goal of the system is developing dynamic user interface tool, the dynamics of Keele Ferries Booking System example must to be tested. Therefore, event code classes generated by the tool are tested while they are running.

Main Window

- Goal : window moving

Component type	Component name	Component event	Show window	Dispose window	Result
Button	Booking	onClick	FerryBooking		ok
Button	Searching	onClick	FerrySearching		ok
Button	Holiday sites	onClick	HolidayBooking		ok

Ferry searching

- Goal : window moving

Component type	Component name	Component event	Show window	Dispose window	Result
Button	Exit	onClick		FerrySearching	ok
Button	Booking	onClick	FerryBooking	FerrySearching	ok

- Goal : component content sharing

  Component action : Click 'Booking' Button

Component type	Component name	Affected components			Result
		window name	component type	component name	
Choice	Port from	FerryBooking FerryBooking	Choice Choice	from (outward) to (inward)	ok ok
Choice	Port to	FerryBooking FerryBooking	Choice Choice	to (outward) from (inward)	ok ok
TextField	Date	FerryBooking	TextField	Date (outward)	ok

Attribute linked (object attribute & component attribute)

Class	Object	Object attribute	Window name	Component type	component name	component attribute
Booking	Booking 1	from	FerrySearching FerryBooking FerryBooking	Choice Choice Choice	Port from from (outward) to (inward)	Item Item Item
		to	FerrySearching FerryBooking FerryBooking	Choice Choice Choice	Port to to (outward) from (inward)	Item Item Item
		date	FerryBooking	TextField	Date (outward)	Text

Appendix H The Example Code of Interface Design

Appendix H shows the code of a static window design, of the generated event code extending the static window design, and of an instantiation class which instantiates all window designs of a project as class variables.

'Holiday sites Booking' window of the Keele Ferries Booking System

```
import java.awt.*;

public class HolidayBooking extends Frame {

    public Checkbox tent, chalet, camping;
    public TextField textDateFrom, textDateTo, textLocation, tfName, tfTel, tfPost,
                tfAddress;
    public String eventName;
    public Button btnSave, btnCancel;
    public Label labelName, labelTel, labelPost, labelAddress, labelSiteType,
            labelDateFrom, labelDateTo,labelLocation;
    public Choice chLocation;

    GridBagLayout gbl;
    GridBagConstraints gbc;

    public HolidayBooking() {

        setTitle("Holiday Site Booking");
        setSize(500,300);
        gbl = new GridBagLayout();
        gbc = new GridBagConstraints();
        setLayout(gbl);
        getInsets();
        labelName = new Label("Customer name");
        tfName = new TextField(23);
        labelTel = new Label("Telephone");
        tfTel = new TextField(8);
        labelPost = new Label("Post code");
        tfPost = new TextField(8);
        labelAddress = new Label("Address");
        tfAddress = new TextField(45);
        labelSiteType = new Label("Site Type :");
        tent = new Checkbox("Tent rental");
        chalet = new Checkbox("Chalet");
        camping = new Checkbox("Caravan/camping site");
        labelDateFrom = new Label("Date From : ");
        textDateFrom = new TextField(8);
        labelDateTo = new Label("Date To : ");
        textDateTo = new TextField(8);
        labelLocation = new Label("Location :");
        chLocation = new Choice();
        chLocation.addItem("Ramsgate");
        chLocation.addItem("Dunkirk");
        chLocation.addItem("Calais");
        btnSave = new Button("Save");
        btnCancel = new Button("Cancel");

        gbc.weightx = 20;
        gbc.weighty = 100;
        gbc.anchor = GridBagConstraints.WEST;
        add(labelName, gbl,gbc,0,0,2,1);
        add(tfName, gbl,gbc,2,0,2,1);
        add(labelTel, gbl,gbc,0,1,1,1);
        add(tfTel, gbl,gbc,1,1,1,1);
        add(labelPost, gbl,gbc,2,1,1,1);
        add(tfPost, gbl,gbc,3,1,1,1);
        add(labelAddress, gbl,gbc,0,2,1,1);
        add(tfAddress, gbl,gbc,1,2,3,2);
        add(labelSiteType, gbl,gbc,0,4,1,1);
        add(tent, gbl,gbc,1,4,3,1);
        add(camping, gbl,gbc,1,5,3,1);
        add(chalet, gbl,gbc,1,6,3,1);
```

```
        add(labelLocation, gbl,gbc,2,6,1,1);
        add(chLocation, gbl,gbc,3,6,1,1);
        add(labelDateFrom, gbl,gbc,0,7,1,1);
        add(textDateFrom, gbl,gbc,1,7,1,1);
        gbc.anchor = GridBagConstraints.EAST;
        add(labelDateTo, gbl,gbc,2,7,1,1);
        gbc.anchor = GridBagConstraints.WEST;
        add(textDateTo, gbl,gbc,3,7,1,1);
        gbc.fill = GridBagConstraints.VERTICAL;
        add(btnSave, gbl,gbc,1,8,1,1);
        add(btnCancel, gbl,gbc,3,8,1,1);
    }

    public Insets getInsets() {
        return new Insets(25,15,5,15);
    }

    public void add(Component c, GridBagLayout gbl,
        GridBagConstraints gbc, int x, int y, int w, int h) {
        gbc.gridx = x;
        gbc.gridy = y;
        gbc.gridwidth = w;
        gbc.gridheight = h;
        gbl.setConstraints(c, gbc);
        add(c);
    }

    public static void main(String[] args) {

        HolidayBooking f = new HolidayBooking();
        f.setVisible(true);
    }
}
```

The event handling code of 'Holiday sites Booking' window, generated by DUIDT

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

public class HolidayBookingEvent extends HolidayBooking implements ActionListener,
                                            ItemListener, TextListener {

    HolidayBookingEvent() {

        //to stop running
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);

        btnSave.addActionListener(this);
        tfName.addActionListener(this);
    }
    public void processWindowEvent(WindowEvent e) {
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
            setVisible(false);
            dispose();
            System.exit(0);
        }
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();

        if (source == btnSave) {
            KeeleFerriesBooking.theHolidayBookingEvent.setVisible(false);
            KeeleFerriesBooking.theHolidayBookingEvent.dispose();
        }
        if (source == tfName) {
            KeeleFerriesBooking.theFerryBookingEvent.setVisible(true);
            String valuetfName = tfName.getText();
            String valuetfTel = tfTel.getText();
            String objectNametfName = "Ferry Customer 1";
            String objectAttributetfName = "name";
            KeeleFerriesBooking.oa.saveValue(valuetfName, objectNametfName,
                                                      objectAttributetfName);
            KeeleFerriesBooking.theFerryBookingEvent.textName.setText(valuetfName);
            String objectNametfTel = "Ferry Customer 1";
            String objectAttributetfTel = "phone number";
            KeeleFerriesBooking.oa.saveValue(valuetfTel, objectNametfTel,
                                                      objectAttributetfTel);
            KeeleFerriesBooking.theFerryBookingEvent.textPhone.setText(valuetfTel);
        }
    }

    public void itemStateChanged(ItemEvent e) {
        Object source = e.getSource();
    }

    public void textValueChanged(TextEvent e) {
        Object source = e.getSource();
    }

    public static void main(String[] args) {
        HolidayBookingEvent theHolidayBooking = new HolidayBookingEvent();
        theHolidayBooking.setVisible(true);
    }
}
```

Window instantiation class, generated by DUIDT

```
public class KeeleFerriesBooking {

    //to instantiate all window objects
    static FerryBookingEvent theFerryBookingEvent = new FerryBookingEvent();
    static FerrySearchingEvent theFerrySearchingEvent = new FerrySearchingEvent();
    static FerryMainEvent theFerryMainEvent = new FerryMainEvent();
    static InsuranceEvent theInsuranceEvent = new InsuranceEvent();
    static AccommodationEvent theAccommodationEvent = new AccommodationEvent();
    static VehicleEvent theVehicleEvent = new VehicleEvent();
    static HolidayBookingEvent theHolidayBookingEvent = new HolidayBookingEvent();
```

```
    //to save object attributes to database
    static ObjectAttribute oa = new ObjectAttribute("p6i21", "lila9610");
}
```

Appendix I Source Code

Number of classes

- classes with a frame - 11
- dialog classes - 6
- working modules without interface - 2

Total lines of code : 7,689 lines of Java code

Event Handler Class

'Event Handler' class reads the linked information of window designs and STDs, and generates an event handling class for each window desgin of a project. The source code of 'Event Handler' class is as follows.

```
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.io.*;
import java.util.*;

public class EventHandler {

    static final int ACTIONLISTENER = 1;
    static final int ITEMLISTENER = 2;
    static final int TEXTLISTENER = 3;

    private Statement stmt;
    protected Vector
            compNameVector, eventTypeVector,

            projectScreenVector,

            conditionVector, transitionVector, classVector,
            actionVector, actionTSVector, actionClassVector,
            showNextScreenVector,
            transitionCompAttributeVector, transitionCompNameVector,

            fromStateVector, toStateVector, fromStateClassVector, toStateClassVector,
            fromActionVector, fromActionStateVector, fromActionClassVector,
            toActionVector, toActionStateVector, toActionClassVector,
            fromStateScreenVector, toStateScreenVector,
            FTCompAttributeVector, FTCompNameVector,

            stateVector,
            showStateScreenVector,
            stateCompAttributeVector, stateCompNameVector,

            addListenerVector,
            actionListenerVector, itemListenerVector, textListenerVector,
            otherCompAttributeVector, otherCompNameVector, otherScreenVector,
            objectNameVector, objectAttributeVector;

    private String project;
    private String packedProject;
    public EventHandler(String prj) {

        project = prj;

        /* project name is used as a filename of the class which is generated to
            instantiate all window designs in the project. So, the space between
            project name characters should be removed */
        packedProject = Main.ts.trimPack(project);
```

```
projectScreenVector = new Vector(); //save all windows in a project

/* set component and event type */
compNameVector = new Vector();
eventTypeVector = new Vector();

/* handle transition event */
conditionVector = new Vector();
transitionVector = new Vector();
classVector = new Vector();

actionVector = new Vector();
actionTSVector = new Vector();
actionClassVector = new Vector();

// keep the next screen to generate event code
showNextScreenVector = new Vector();

// keep transition component attribute to indicate method
transitionCompAttributeVector = new Vector();
//keep transition component name to generate event code
transitionCompNameVector = new Vector();

/* handle from/to state event */
fromStateVector = new Vector();
toStateVector = new Vector();
fromStateClassVector = new Vector();
toStateClassVector = new Vector();

//researve from action
fromActionVector = new Vector();
fromActionStateVector= new Vector();
fromActionClassVector= new Vector();

//researve to action
toActionVector = new Vector();
toActionStateVector= new Vector();
toActionClassVector= new Vector();

// researve from state screen
fromStateScreenVector = new Vector();

// researve to state screen
toStateScreenVector = new Vector();

// researve from/to state component name and attribute
FTCompAttributeVector = new Vector();
FTCompNameVector = new Vector();

/* handle state event */

//researve state name
stateVector = new Vector();

//researve screen of state action
showStateScreenVector = new Vector();

//researve component name and attribute of state action
stateCompAttributeVector = new Vector();//keep to generate code
stateCompNameVector = new Vector();     //keep to generate code

/* handle event listener */

//to write code e.g. btn.addActionListener;
addListenerVector = new Vector();

actionListenerVector = new Vector();
itemListenerVector = new Vector();
textListenerVector = new Vector();

// researve object name and attribute
objectNameVector = new Vector();
objectAttributeVector = new Vector();

//researve other components linked to same object attribute
otherCompAttributeVector = new Vector();
otherCompNameVector = new Vector();
```

```
    otherScreenVector = new Vector();
}

/* read all screen in project and set the to the vector
 * generate an event code class for each screen */
public void screenEventCode() {

    projectScreenVector.removeAllElements();

    try {
        stmt = Main.con.createStatement();

        String query ="SELECT screen_name FROM screen WHERE project_name = '"
            + project + "'";
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String screens = rs.getString(1);
            String screen = Main.ts.trimUnpack(screens);
            projectScreenVector.addElement(screen);
        }
        rs.close();
    } catch(Exception ex) {
        Main.tfStatusOthers.setText("Error:1" + ex);
    }


    /* Generate an event handling code for a window which belongs to the project
     * Repeat until all windows' event codes are generated */

    for (int i=0; i< projectScreenVector.size(); i++ ) {
        String projectScreen = (String)projectScreenVector.elementAt(i);
        generateCode(projectScreen);

    }
}

/* This method is generating not event handling classes but a creating object
    class of all windows in the project. All windows is created as static
    variables (class variables). */

public void writeAllScreenFile() {
    try {

        File outputFile = new File(packedProject +".java");
        DataOutputStream dos = new DataOutputStream(new
                                FileOutputStream(outputFile));
        dos.writeBytes("public class "+ packedProject + " {\n\n\t//to instantiate
                        all window objects\n");

        /* write all screen class variables */
        for (int i=0; i<projectScreenVector.size(); i++ ) {
            String projectScreen = (String)projectScreenVector.elementAt(i);

            dos.writeBytes("\tstatic " + projectScreen + "Event the"
                + projectScreen + "Event = new "
                + projectScreen + "Event();\n");
        }

        dos.writeBytes("\n\t//to save object attributes to database\n\tstatic
                ObjectAttribute oa = new ObjectAttribute(\""
                + Main.userName + "\", \"" + Main.password+"\");\n}");
        dos.close();
    } catch (IOException ex) {
        Main.tfStatusOthers.setText("Error :2" + ex);
    }
}

/* generate an event code class for each screen and save to file. */

public void generateCode(String prjScreen) {

    addListenerVector.removeAllElements();
    actionListenerVector.removeAllElements();
    itemListenerVector.removeAllElements();
    textListenerVector.removeAllElements();

    String screen = prjScreen;
```

```
        //get a component event of a window to generate event codes
        setComponentEvent(screen);

        //generate event codes and save to Vector
        handleComponentEvent(screen);

        //write Vectors having event codes to file
            writeToFile(screen);
    }

    /* get unique component name and event type from both transition
        and state event table */

    public void setComponentEvent(String scr) {
        String screen = scr;

        compNameVector.removeAllElements();
        eventTypeVector.removeAllElements();

        /* get transition event */

        try {
            String query ="SELECT DISTINCT component_name, event_type "
                + "FROM condition_transition_event WHERE screen_name = '"
                + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String componentName = rs.getString(1);
                String eventType = rs.getString(2);
                String compName = Main.ts.trimUnpack(componentName);
                String evtType = Main.ts.trimUnpack(eventType);

                compNameVector.addElement(compName);
                eventTypeVector.addElement(evtType);

            }
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error:3" + ex);
        }

        //* get state event */
        try {
            String query ="SELECT DISTINCT component_name, event_type "
                + "FROM condition_state_event WHERE screen_name = '"
                + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String componentName = rs.getString(1);
                String eventType = rs.getString(2);
                String compName = Main.ts.trimUnpack(componentName);
                String evtType = Main.ts.trimUnpack(eventType);

                compNameVector.addElement(compName);
                eventTypeVector.addElement(evtType);
            }
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error:4" + ex);
        }

        /* A component having the same event type can be linke to both state
            and transtition event.
            remove redundant component name and event type, make it unique */

        for (int i=0; i < compNameVector.size()-1; i++) {

            String preCompName = (String)compNameVector.elementAt(i);
            String preEvtType = (String)eventTypeVector.elementAt(i);

            for (int j = i+1; j < compNameVector.size(); j++) {
                String compName = (String)compNameVector.elementAt(j);
                String evtType = (String)eventTypeVector.elementAt(j);

                if ((compName.equals(preCompName)) && (evtType.equals(preEvtType))) {
                    compNameVector.removeElementAt(j);
                    eventTypeVector.removeElementAt(j);
                }
```

```
            }
        }
    }

    /* generate event codes for a component event type */
    public void handleComponentEvent(String scr) {

        String screen = scr;

        for (int i=0; i< compNameVector.size(); i++) {
            String compName = (String)compNameVector.elementAt(i);
            String evtType = (String)eventTypeVector.elementAt(i);
            /* read transition condition and action,
             * set transition next screen to the show vector,
             * set transition component name and attribute,
             * object name and attribute to the vector */

            handleTransitionEvent(compName, evtType, screen);

            /* read condition_transition_event table,
             * and get unique transition (to remove redundancy for the
             * case that several conditions of one transition are linked
             * to the same event), get from/to state of the transition,
             * remove existing redundancy again (for the case one state has several
             * transitions), get from/to state action,
             * get the screen of from/to state action, read action_state_attribute,
             * and get component attribute and name of the from/to state action,
             * get object name and attribute of the component attribute and name */

            handleFromToStateEvent(compName, evtType, screen);

            /* get state event condition and action,
             * get state action screen (show),
             * get component name & attribute of state action,
             * object name & attribute */

            handleStateEvent(compName, evtType, screen);

            /* read all generate code vectors for event, and write them to the
                suitable listener or shell */
            handleEventListener(compName, evtType, screen);
        }
    }

    public void handleTransitionEvent(String cName, String eType, String scr) {

        String compName = cName;
        String evtType = eType;
        String screen = scr;

        /* vector initialisation */

        // researve condition of transition event
        conditionVector.removeAllElements();
        transitionVector.removeAllElements();
        classVector.removeAllElements();

        // researve transition action for the condition
        actionVector.removeAllElements();
        actionTSVector.removeAllElements(); //both transition and state are researved
        actionClassVector.removeAllElements();

        // researve next screen to generate code --> show screen
        showNextScreenVector.removeAllElements();

        /* researve transition component attribute and name to generate code
             --> get component text */
        transitionCompAttributeVector.removeAllElements();
        transitionCompNameVector.removeAllElements();

        /* read condition transition event table */

        try {
            String query ="SELECT condition_name, transition_name, class_name "
                + "FROM condition_transition_event WHERE component_name = '"
                + compName + "' AND event_type ='" + evtType + "' and screen_name = '"
                + screen + "'";
```

```
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String conditionName = rs.getString(1);
                String transitionName = rs.getString(2);
                String classNames = rs.getString(3);

                String condition = Main.ts.trimUnpack(conditionName);
                String transition = Main.ts.trimUnpack(transitionName);
                String className = Main.ts.trimUnpack(classNames);

                conditionVector.addElement(condition);
                transitionVector.addElement(transition);
                classVector.addElement(className);

            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error5: " + ex);
        }

        for (int i=0; i < conditionVector.size(); i++) {
            String condition = (String)conditionVector.elementAt(i);
            String transition = (String)transitionVector.elementAt(i);
            String className = (String)classVector.elementAt(i);

            // read action(s) for each condition
            getTransitionAction(condition, transition, className);

        }

        /* if more than one condition liked to the same event have the same action,
         * this cause redundant work such as showing the same screen twice or
         * get the same content of component attribute twice. So, need to remove
            redundancy */
        removeActionRedundancy();

        for (int i=0; i < actionVector.size(); i++) {
            String action = (String)actionVector.elementAt(i);
            String actionTransition = (String)actionTSVector.elementAt(i);
            String actionClass = (String)actionClassVector.elementAt(i);

            // get the next screen liked to the transition action,
            // this screen should be shown when event happens
            getTransitionActionScreen(action, actionTransition, actionClass);

            // action also causes the change of component Attribute
            getTransitionComponentAttribute(action, actionTransition, actionClass,
screen);

        }
    }

    /* read transition activity table and get transition action(s) for a condition */

    public void getTransitionAction(String cdn, String trnsn, String cls) {

        String condition = cdn;
        String transition = trnsn;
        String className  = cls;

        try {
            String query ="SELECT action FROM transition_activity where class_name =
                '" className + "' and transition_name = '" + transition
                + "' and condition = '" + condition + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String actionName = rs.getString(1);
                String action = Main.ts.trimUnpack(actionName);
                actionVector.addElement(action);

                // this vector reserves both transition and state to use one
                // removeActionRedundancy method
                actionTSVector.addElement(transition);
                actionClassVector.addElement(className);

            }
```

```
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error:6" + ex);
        }
    }

    public void removeActionRedundancy() {
        for (int i=0; i < actionVector.size()-1; i++) {

            String preAction = (String)actionVector.elementAt(i);
            String preActionTS = (String)actionTSVector.elementAt(i);
            String preActionClass= (String)actionClassVector.elementAt(i);

            for (int j = i+1; j < actionVector.size(); j++) {
                String curAction = (String)actionVector.elementAt(j);
                String curActionTS= (String)actionTSVector.elementAt(j);
                String curActionClass= (String)actionClassVector.elementAt(j);

                if ( (curAction.equals(preAction)) &&
(curActionTS.equals(preActionTS))
                    && (curActionClass.equals(preActionClass)) ) {

                    actionVector.removeElementAt(j);
                    actionTSVector.removeElementAt(j);
                    actionClassVector.removeElementAt(j);
                }
            }
        }
    }

    /* find transition action screen, and set it to the showNextScreenVector */

    public void getTransitionActionScreen(String act, String trnsn, String cls) {

        String action = act;
        String transition = trnsn;
        String className = cls;

        try {
            String query = "SELECT screen_name FROM action_transition_screen WHERE
action_name = '"
                + action + "' and transition_name = '" + transition + "' and
class_name = '"
                + className + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String nextScreenName = rs.getString(1);
                String nextScreen = Main.ts.trimUnpack(nextScreenName);
                showNextScreenVector.addElement(nextScreen);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error7:" + ex);
        }


    }

    public void getTransitionComponentAttribute(String act, String tran, String cls,
String scr) {

        String action = act;
        String transition = tran;
        String className = cls;
        String screen = scr;

        try {
            String query = "SELECT component_attribute, component_name "
                + "FROM action_transition_attribute WHERE action_name = '"
                + action + "' and transition_name = '" + transition
                + "' and class_name = '"+ className
                + "' and screen_name = '" + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String componentAttribute = rs.getString(1);
                String componentName = rs.getString(2);
```

```
                String compAttribute = Main.ts.trimUnpack(componentAttribute);
                String compName = Main.ts.trimUnpack(componentName);

                transitionCompAttributeVector.addElement(compAttribute);
                transitionCompNameVector.addElement(compName);

            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error8:" + ex);
        }

    }

    //to avoid same state is read more than one because of more than one conditon of
same transition

    public void handleFromToStateEvent(String cName, String eType, String scr) {

        String compName = cName;
        String evtType = eType;
        String screen = scr;

        //researve unique condition and class of the compoennt event, this vector used
for handleTransitionEvent as well
        transitionVector.removeAllElements();
        classVector.removeAllElements();

        //transition has only one from state and to state
        fromStateVector.removeAllElements();
        toStateVector.removeAllElements();

        //to find from/to state action and remove redundancy
        fromStateClassVector.removeAllElements();
        toStateClassVector.removeAllElements();

        // researve from state action to find from state action exit screen
        fromActionVector.removeAllElements();
        fromActionStateVector.removeAllElements();
        fromActionClassVector.removeAllElements();

        // researve to state action to find to state action entry screen
        toActionVector.removeAllElements();
        toActionStateVector.removeAllElements();
        toActionClassVector.removeAllElements();

        //researve exit screen ----> dispose
        fromStateScreenVector.removeAllElements();

        //researve entry screen --> show
        toStateScreenVector.removeAllElements();

        //researve component name and attribute of from/to state

        FTCompNameVector.removeAllElements();
        FTCompAttributeVector.removeAllElements();

        try {
            String query = "SELECT DISTINCT transition_name, class_name "
                + "FROM condition_transition_event WHERE event_type = '"
                + evtType + "' and component_name = '" + compName + "' and screen_name
='"
                + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String transitionName = rs.getString(1);
                String className = rs.getString(2);
                String transition = Main.ts.trimUnpack(transitionName);
                String clsName = Main.ts.trimUnpack(className);
                transitionVector.addElement(transition);
                classVector.addElement(clsName);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error9:" + ex);
        }
```

```
        for (int i=0; i< transitionVector.size(); i++ ) {
            String transition = (String)transitionVector.elementAt(i);
            String className = (String)classVector.elementAt(i);

            getFromToState(transition, className);
        }

        /* even though the transition has only one from and to state,
         * there is still redundant cases.
         * For example, two transitions may be linked to the same state,
         * or one state has several transitions.
         * So, states could be redundant again, need to remove this */

        removeFromStateRedundancy();
        removeToStateRedundancy();

        for(int i=0; i< fromStateVector.size(); i++) {
            String fromState = (String)fromStateVector.elementAt(i);
            String fromStateClass = (String)fromStateClassVector.elementAt(i);
            getFromStateAction(fromState, fromStateClass);
        }

        for(int i=0; i< toStateVector.size(); i++) {
            String toState = (String)toStateVector.elementAt(i);
            String toStateClass = (String)toStateClassVector.elementAt(i);
            getToStateAction(toState, toStateClass);
        }

        for (int i=0; i < fromActionVector.size(); i++) {

            String fromAction = (String)fromActionVector.elementAt(i);
            String fromActionState = (String)fromActionStateVector.elementAt(i);
            String fromActionClass = (String)fromActionClassVector.elementAt(i);

            // get from state action screen
            getFromStateScreen(fromAction, fromActionState, fromActionClass);

            //get the component attibute of from state action
            getFTStateComponentAttribute(fromAction, fromActionState, fromActionClass,
screen);
        }

        for (int i=0; i < toActionVector.size(); i++) {

            String toAction = (String)toActionVector.elementAt(i);
            String toActionState = (String)toActionStateVector.elementAt(i);
            String toActionClass = (String)toActionClassVector.elementAt(i);

            //get to state action screen
            getToStateScreen(toAction, toActionState, toActionClass);

            //get the component attribute of to state action
            getFTStateComponentAttribute(toAction, toActionState, toActionClass,
screen);
        }

    }

    public void getFromToState(String trnsn, String cls) {

        String transition = trnsn;
        String className = cls;

        try {
            String query = "SELECT from_state, to_state FROM transition WHERE
class_name = '"
                + className + "' and transition_name = '" + transition + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String fromStateName = rs.getString(1);
                String toStateName = rs.getString(2);

                String fromState = Main.ts.trimUnpack(fromStateName);
                String toState = Main.ts.trimUnpack(toStateName);

                fromStateVector.addElement(fromState);
                toStateVector.addElement(toState);
```

```
                //need to keep class name to find state action, and remove redundancy
                fromStateClassVector.addElement(className);
                toStateClassVector.addElement(className);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error10:" + ex);
        }
    }

    //remove the redundant data of fromState
    public void removeFromStateRedundancy() {
        for(int i=0; i< fromStateVector.size()-1; i++) {
            String preFromState = (String)fromStateVector.elementAt(i);
            String preFromStateClass = (String)fromStateClassVector.elementAt(i);

            for (int j = i+1; j < fromStateVector.size(); j++) {
                String curFromState = (String)fromStateVector.elementAt(j);
                String curFromStateClass = (String)fromStateClassVector.elementAt(j);

                if ( (curFromState.equals(preFromState)) &&
(curFromStateClass.equals(preFromStateClass)) ) {
                    fromStateVector.removeElementAt(j);
                    fromStateClassVector.removeElementAt(j);
                }
            }
        }
    }

    //remove the redundant data of toState
    public void removeToStateRedundancy() {
        for(int i=0; i< toStateVector.size()-1; i++) {
            String preToState = (String)toStateVector.elementAt(i);
            String preToStateClass = (String)toStateClassVector.elementAt(i);

            for (int j = i+1; j < toStateVector.size(); j++) {
                String curToState = (String)toStateVector.elementAt(j);
                String curToStateClass = (String)toStateClassVector.elementAt(j);

                if ( (curToState.equals(preToState)) &&
(curToStateClass.equals(preToStateClass)) ) {
                    toStateVector.removeElementAt(j);
                    toStateClassVector.removeElementAt(j);
                }
            }
        }
    }

    public void getFromStateAction(String fState, String cls) {
        String fromState = fState;
        String className = cls;

        try {
            String query = "SELECT action FROM state_activity WHERE condition = 'exit'
AND state_name = '"
                + fromState + "' AND class_name = '" + className + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {

                String stateAction = rs.getString(1);
                String fromStateAction = Main.ts.trimUnpack(stateAction);
                fromActionVector.addElement(fromStateAction);

                //researve from state and class name to find exit screen
                fromActionStateVector.addElement(fromState);
                fromActionClassVector.addElement(className);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error11:" + ex);
        }
    }

    public void getToStateAction(String tState, String cls) {
        String toState = tState;
        String className = cls;
```

```
        try {
            String query = "SELECT action FROM state_activity WHERE condition =
'entry' and state_name = '"
                + toState + "' AND class_name = '" + className + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {

                String stateAction = rs.getString(1);
                String toStateAction = Main.ts.trimUnpack(stateAction);
                toActionVector.addElement(toStateAction);

                //researve from state and class name to find exit screen
                toActionStateVector.addElement(toState);
                toActionClassVector.addElement(className);

            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error12:" + ex);
        }
    }

    public void getFromStateScreen(String fAction, String fState, String fCls) {

        String action = fAction;
        String state = fState;
        String className = fCls;

        try {
            String query = "SELECT screen_name FROM action_state_screen WHERE
action_name = '"
                + action + "' AND state_name = '" + state + "' and class_name = '"
                + className + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String screen = rs.getString(1);
                String fromStateScreen = Main.ts.trimUnpack(screen);
                fromStateScreenVector.addElement(fromStateScreen);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error113:" + ex);
        }
    }

    public void getToStateScreen(String tsAction, String tState, String cls) {
        String toStateAction = tsAction;
        String toState = tState;
        String className = cls;

        try {
            String query = "SELECT screen_name FROM action_state_screen WHERE
action_name = '"
                + toStateAction + "' AND state_name = '" + toState + "' and class_name
= '"
                + className + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String screen = rs.getString(1);
                String toStateScreen = Main.ts.trimUnpack(screen);
                toStateScreenVector.addElement(toStateScreen);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error14:" + ex);
        }
    }

    //this method is used to find component name and attribute for both from state and
to state
    public void getFTStateComponentAttribute(String act, String sta, String cls,
String scr) {

        String action = act;
        String state =sta;
        String className = cls;
```

```
        String screen = scr;

        try {
            String query = "SELECT component_attribute, component_name FROM
action_state_attribute WHERE action_name = '"
                + action + "' AND state_name = '" + state + "' AND class_name = '"+
className
                + "' AND screen_name = '" + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String componentAttribute = rs.getString(1);
                String componentName = rs.getString(2);

                String compAttribute = Main.ts.trimUnpack(componentAttribute);
                String compName = Main.ts.trimUnpack(componentName);

                FTCompAttributeVector.addElement(compAttribute);
                FTCompNameVector.addElement(compName);

            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error:15" + ex);
        }
    }

    public void handleStateEvent(String cName, String eType, String scr) {

        String compName = cName;
        String evtType = eType;
        String screen = scr;

        //researve state condition, reusing but cleaned condition and class vector
        conditionVector.removeAllElements();
        stateVector.removeAllElements();
        classVector.removeAllElements();

        //researve state action, reusing vectors, but cleaned
        actionVector.removeAllElements();
        actionTSVector.removeAllElements();
        actionClassVector.removeAllElements();

        //researve state action screen --> show
        showStateScreenVector.removeAllElements();

        //researve component name and attribute
        stateCompAttributeVector.removeAllElements();
        stateCompNameVector.removeAllElements();
        try {
            String query ="SELECT condition_name, state_name, class_name FROM
condition_state_event "
                + "WHERE component_name = '" + compName + "' and event_type ='" +
evtType
                + "' and screen_name = '" + screen + "' and condition_name != 'entry'
and condition_name != 'exit'";

            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String conditionName = rs.getString(1);
                String stateName = rs.getString(2);
                String classNames = rs.getString(3);

                String condition = Main.ts.trimUnpack(conditionName);
                String state = Main.ts.trimUnpack(stateName);
                String className = Main.ts.trimUnpack(classNames);

                conditionVector.addElement(condition);
                stateVector.addElement(state);
                classVector.addElement(className);

            }
            rs.close();

        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error16: " + ex);
        }
```

```
        for (int i=0; i< conditionVector.size(); i++) {
            String condition = (String)conditionVector.elementAt(i);
            String state = (String)stateVector.elementAt(i);
            String className = (String)classVector.elementAt(i);
            getStateAction(condition, state, className);
        }

        // if two state conditions of the same event has same action name, it is
redundant.
        removeActionRedundancy();

        for (int i=0; i < actionVector.size(); i++) {
            String action = (String)actionVector.elementAt(i);
            String actionState = (String)actionTSVector.elementAt(i);
            String actionClass = (String)actionClassVector.elementAt(i);
            getStateActionScreen(action, actionState, actionClass);

            //action causes the change of component Attribute
            getStateComponentAttribute(action, actionState, actionClass, screen);
        }
    }

    public void getStateAction(String cdn, String sta, String cls) {
        String condition = cdn;
        String state = sta;
        String className  = cls;
        try {
            String query ="SELECT action FROM state_activity WHERE class_name = '" +
className
                + "' AND state_name = '" + state + "' AND condition = '"
                + condition + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String actionName = rs.getString(1);
                String action = Main.ts.trimUnpack(actionName);

                //use same vector name to reuse removeRedundancy method
                actionVector.addElement(action);
                actionTSVector.addElement(state);
                actionClassVector.addElement(className);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error17:" + ex);
        }
    }

    public void getStateActionScreen(String act, String sta, String cls) {
        String action = act;
        String state = sta;
        String className = cls;

        try {
            String query = "SELECT screen_name FROM action_state_screen WHERE
action_name = '"
                + action + "' AND state_name = '" + state + "' AND class_name = '"
                + className + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String nextScreenName = rs.getString(1);
                String nextScreen = Main.ts.trimUnpack(nextScreenName);
                showStateScreenVector.addElement(nextScreen);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error:18" + ex);
        }
    }

    public void getStateComponentAttribute(String act, String sta, String cls, String
scr) {
        String action = act;
        String state =sta;
        String className = cls;
        String screen = scr;

        try {
```

```
            String query = "SELECT component_attribute, component_name FROM
action_state_attribute "
                + "WHERE action_name = '" + action + "' and state_name = '" + state
                + "' and class_name = '"+ className
                + "' and screen_name = '" + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String componentAttribute = rs.getString(1);
                String componentName = rs.getString(2);

                String compAttribute = Main.ts.trimUnpack(componentAttribute);
                String compName = Main.ts.trimUnpack(componentName);
                stateCompAttributeVector.addElement(compAttribute);
                stateCompNameVector.addElement(compName);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error19:" + ex);
        }
    }

    //get all vectors and write to the listener

    public void handleEventListener(String cName, String eType, String scr) {

        String compName = cName;
        String evtType = eType;
        String screen = scr;

        String compType = getComponentType(compName, screen);
        int eventListener = identifyEventListener(compType, evtType);

        setToAddListenerShell(compName, eventListener);
        setToEventListener(compName, eventListener, compType, screen);
    }

    public String getComponentType(String cName, String scr) {

        String compName = cName;
        String screen = scr;
        String compType = "";

        try {
            String query = "SELECT component_type FROM component WHERE component_name
= '" + compName
                + "' AND screen_name = '" + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String componentType = rs.getString(1);
                compType = Main.ts.trimUnpack(componentType);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error20: " + ex);
        }
        return compType;
    }

    public int identifyEventListener(String cType, String eType) {
        String compType = cType;
        String event = eType;

        int listenerType = 0;

        if (compType.equals("Button")) {
            if(event.equals("onClick")) {
                listenerType = ACTIONLISTENER;
            }
        } else if (compType.equals("Choice")) {
            if (event.equals("onClick")) {
                listenerType = ITEMLISTENER;
            }
        } else if (compType.equals("Checkbox")) {
            if (event.equals("onClick")) {
                listenerType = ITEMLISTENER;
            }
        } else if (compType.equals("TextField")) {
```

```
            if (event.equals("onEnter")) {
                listenerType = ACTIONLISTENER;

            } else if (event.equals("onChange")) {
                listenerType = TEXTLISTENER;

            }
        } else if (compType.equals("TextArea")) {
            if (event.equals("onChange")) {
                listenerType = TEXTLISTENER;

            }
        } else if (compType.equals("List")) {
            if (event.equals("onClick")) {
                listenerType = ITEMLISTENER;
            } else if (event.equals("onDblClick")) {
                listenerType = ACTIONLISTENER;
            }
        }
        return listenerType;
    }

    /* The shell is for saving add linstsner code such as
"btn.addActionListener(this)" */

    public void setToAddListenerShell(String cName, int evtListener) {

        String compName = cName;
        switch (evtListener) {
            case ACTIONLISTENER : addListenerVector.addElement("\t\t" + compName +
".addActionListener(this);\n");
                        break;
            case ITEMLISTENER : addListenerVector.addElement("\t\t" + compName +
".addItemListener(this);\n");
                        break;
            case TEXTLISTENER : addListenerVector.addElement("\t\t" + compName +
".addTextListener(this);\n");
                        break;
            default : Main.tfStatusOthers.setText("No event listener.");
        }
    }

    public void setToEventListener(String cName, int evtListener, String cType, String
scr) {

        String compType = cType;
        String compName = cName;
        String screen = scr;
        int eventListener = evtListener;

        String codeBegin = "\n\n\t\tif (source == " + compName + ") {\n";
        String codeEnd = "\n\t\t}";

        switch (eventListener) {
            case ACTIONLISTENER : actionListenerVector.addElement(codeBegin);

                //set transition action screen to ActionListsner  --> show
                String actionNextScreenCode = getShowNextScreenCode();
                actionListenerVector.addElement(actionNextScreenCode);

                //set fromState exit screen to ActionListsner --> dispose
                String actionDisposeScreenCode = getFromStateScreenCode();
                actionListenerVector.addElement(actionDisposeScreenCode);

                //set toState entry screen to ActionListsner  --> show
                String actionEntryScreenCode = getToStateScreenCode();
                actionListenerVector.addElement(actionEntryScreenCode);

                //set state screen to ActionListsner and allScreenShell  --> show
                String actionStateScreenCode = getStateScreenCode();
                actionListenerVector.addElement(actionStateScreenCode);


                //handle transition component name & attribute   --> get/save/set text
                String actionTransitionAttributeCode = getTransitionAttributeCode(
screen);
                actionListenerVector.addElement(actionTransitionAttributeCode);
```

```
                //handle from/to state component name & attribute   --> get/save/set
text
                String actionFromToStateAttributeCode =
getFromToStateAttributeCode(screen);
                actionListenerVector.addElement(actionFromToStateAttributeCode);

                //handle state component name & attribute   --> get/save/set text
                String actionStateAttributeCode = getStateAttributeCode(screen);
                actionListenerVector.addElement(actionStateAttributeCode);
                actionListenerVector.addElement(codeEnd);
                break;

            case ITEMLISTENER : itemListenerVector.addElement(codeBegin);
                //set transition action screen to ActionListsner  --> show
                String itemNextScreenCode = getShowNextScreenCode();
                itemListenerVector.addElement(itemNextScreenCode);

                //set fromState exit screen to ActionListsner --> dispose
                String itemDisposeScreenCode = getFromStateScreenCode();
                itemListenerVector.addElement(itemDisposeScreenCode);

                //set toState entry screen to ActionListsner  --> show
                String itemEntryScreenCode = getToStateScreenCode();
                itemListenerVector.addElement(itemEntryScreenCode);

                //set state screen to ActionListsner and allScreenShell  --> show
                String itemStateScreenCode = getStateScreenCode();
                itemListenerVector.addElement(itemStateScreenCode);


                //handle transition component name & attribute   --> get/save/set text
                String itemTransitionAttributeCode = getTransitionAttributeCode(
screen);
                itemListenerVector.addElement(itemTransitionAttributeCode);


                //handle from/to state component name & attribute   --> get/save/set
text
                String itemFromToStateAttributeCode = getFromToStateAttributeCode(
screen);
                itemListenerVector.addElement(itemFromToStateAttributeCode);


                //handle state component name & attribute   --> get/save/set text
                String itemStateAttributeCode = getStateAttributeCode( screen);
                itemListenerVector.addElement(itemStateAttributeCode);

                itemListenerVector.addElement(codeEnd);
                break;

            case TEXTLISTENER : textListenerVector.addElement(codeBegin);
                //set transition action screen to ActionListsner  --> show
                String textNextScreenCode = getShowNextScreenCode();
                textListenerVector.addElement(textNextScreenCode);

                //set fromState exit screen to ActionListsner --> dispose
                String textDisposeScreenCode = getFromStateScreenCode();
                textListenerVector.addElement(textDisposeScreenCode);

                //set toState entry screen to ActionListsner  --> show
                String textEntryScreenCode = getToStateScreenCode();
                textListenerVector.addElement(textEntryScreenCode);

                //set state screen to ActionListsner and allScreenShell  --> show
                String textStateScreenCode = getStateScreenCode();
                textListenerVector.addElement(textStateScreenCode);


                //handle transition component name & attribute   --> get/save/set text
                String textTransitionAttributeCode = getTransitionAttributeCode(
screen);
                textListenerVector.addElement(textTransitionAttributeCode);
```

```
                //handle from/to state component name & attribute   --> get/save/set
text
                String textFromToStateAttributeCode = getFromToStateAttributeCode(
screen);
                textListenerVector.addElement(textFromToStateAttributeCode);


                //handle state component name & attribute   --> get/save/set text
                String textStateAttributeCode = getStateAttributeCode(screen);
                textListenerVector.addElement(textStateAttributeCode);

                textListenerVector.addElement(codeEnd);
                break;
        }
    }

    public String getShowNextScreenCode() {
        String nextScreenCode = "";

        for (int i=0; i< showNextScreenVector.size(); i++) {
            String nextScreen = (String)showNextScreenVector.elementAt(i);
            String screenCode = "\t\t\t" + packedProject + ".the"+ nextScreen +
"Event.setVisible(true);\n\n";
            nextScreenCode += screenCode;
        }
        return nextScreenCode;
    }

    public String getFromStateScreenCode() {
        String disposeScreenCode = "";
        for (int i=0; i < fromStateScreenVector.size(); i++) {

            String disposeScreen = (String)fromStateScreenVector.elementAt(i);
            String disposeCode = "\t\t\t" + packedProject + ".the" + disposeScreen +
"Event.setVisible(false);\n\t\t\t"
                    + packedProject + ".the" + disposeScreen +
"Event.dispose();\n\n\n";
            disposeScreenCode += disposeCode;
        }
        return disposeScreenCode;
    }

    public String getToStateScreenCode() {
        String entryScreenCode ="";

        for (int i=0; i < toStateScreenVector.size(); i++) {
            String entryScreen = (String)toStateScreenVector.elementAt(i);
            String entryCode = "\t\t\t" + packedProject + ".the"+ entryScreen +
"Event.setVisible(true);\n\n";

            entryScreenCode += entryCode;

        }
        return entryScreenCode;
    }

    public String getStateScreenCode() {
        String stateScreenCode ="";

        for (int i=0; i < showStateScreenVector.size(); i++) {
            String stateScreen = (String)showStateScreenVector.elementAt(i);
            String stateCode = "\t\t\t" + packedProject + ".the"+ stateScreen +
"Event.setVisible(true);\n\n";

            stateScreenCode += stateCode;
        }
        return stateScreenCode;
    }

    public String getTransitionAttributeCode( String scr) {
        String screen = scr;

        String getCompAttributeCode ="";
        String saveCompAttributeCode ="";
        String allCompAttributeCode = "";

        for (int i=0 ; i < transitionCompNameVector.size(); i++) {
```

```
            String compName = (String)transitionCompNameVector.elementAt(i);
            String compAttribute = (String)transitionCompAttributeVector.elementAt(i);
            String compType = getComponentType(compName, screen);
            String getCode = writeGetTextCode(compName, compAttribute, compType);
            getCompAttributeCode += getCode;

            String saveCode = writeSaveTextCode(compName, compAttribute, screen);
            saveCompAttributeCode += saveCode;
        }
        allCompAttributeCode = getCompAttributeCode + saveCompAttributeCode;
        return allCompAttributeCode;
    }

    public String writeGetTextCode(String cName, String cAttribute, String cType) {
        String compName = cName;
        String compAttribute = cAttribute;
        String compType = cType;
        String getCode ="";

        if (compType.equals("TextField")) {
            if (compAttribute.equals("Text")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getText();\n\n";
            }
        } else if(compType.equals("TextArea")) {
            if (compAttribute.equals("Text")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getText();\n\n";
            } else if (compAttribute.equals("AppendText")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getText();\n\n";
            }
        } else if(compType.equals("List")) {
            if (compAttribute.equals("Item")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getSelectedItem();\n\n";
            }
        } else if(compType.equals("Checkbox")) {
            if (compAttribute.equals("Label")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getLabel();\n\n";
            }
        } else if(compType.equals("Choice")) {
            if (compAttribute.equals("SetItem")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getSelectedItem();\n\n";
            } else if (compAttribute.equals("AddItem")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getSelectedItem();\n\n";
            }
        } else if(compType.equals("Label")) {
            if (compAttribute.equals("Text")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getText();\n\n";
            }
        } else if(compType.equals("Button")) {
            if (compAttribute.equals("Label")) {
                getCode = "\t\t\tString value" + compName + " = " + compName +
".getLabel();\n\n";
            }
        }

        return getCode;
    }

    public String writeSaveTextCode(String cName, String cAttribute, String scr) {
        String compName = cName;
        String compAttribute = cAttribute;
        String screen = scr;
        objectNameVector.removeAllElements();
        objectAttributeVector.removeAllElements();

        String saveComponentCode = "";
        String setCompAttributeCode = "";
        String saveSetCode = "";

        try {
```

```
            String query = "SELECT object_name, object_Attribute FROM attribute_linked
WHERE component_attribute ='"
                + compAttribute + "' AND component_name = '" + compName + "' AND
screen_name = '"
                + screen + "'";
            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String oName = rs.getString(1);
                String oAttribute = rs.getString(2);
                String objectName = Main.ts.trimUnpack(oName);
                String objectAttribute = Main.ts.trimUnpack(oAttribute);

                objectNameVector.addElement(objectName);
                objectAttributeVector.addElement(objectAttribute);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error:here" + ex);
        }

        for (int i=0; i < objectAttributeVector.size(); i++) {
            String objectName = (String)objectNameVector.elementAt(i);
            String objectAttribute = (String)objectAttributeVector.elementAt(i);
            String saveCode = "\t\t\tString objectName" + compName + " = \"" +
objectName
                + "\";\n\t\t\tString objectAttribute"
                + compName + " = \"" + objectAttribute + "\";\n\t\t\t" + packedProject
                +".oa.saveValue(value" + compName + ", objectName" + compName
                + ", objectAttribute" + compName + ");\n\n";
            saveComponentCode += saveCode;
            String setCode = writeSetTextCode(compName, objectName,
objectAttribute,screen);
            setCompAttributeCode += setCode;
        }
        saveSetCode = saveComponentCode + setCompAttributeCode;
        return saveSetCode;
    }

    //write setCode and set relevant screen to allScreenShell
    public String writeSetTextCode( String cName, String oName, String oAttr, String
scr) {
        String compName = cName;
        String objectName = oName;
        String objectAttribute = oAttr;
        String screen = scr;

        otherCompAttributeVector.removeAllElements();
        otherCompNameVector.removeAllElements();
        otherScreenVector.removeAllElements();

        String setComponentCode ="";

        try {
            String query = "SELECT component_attribute, component_name, screen_name
FROM attribute_linked WHERE object_name ='"
                + objectName + "' and object_attribute = '" + objectAttribute + "'";

            ResultSet rs = stmt.executeQuery(query);
            while (rs.next()) {
                String ocAttribute = rs.getString(1);
                String ocName = rs.getString(2);
                String oScreen = rs.getString(3);
                String otherCompAttribute = Main.ts.trimUnpack(ocAttribute);
                String otherCompName = Main.ts.trimUnpack(ocName);
                String otherScreen = Main.ts.trimUnpack(oScreen);

                otherCompAttributeVector.addElement(otherCompAttribute);
                otherCompNameVector.addElement(otherCompName);
                otherScreenVector.addElement(otherScreen);
            }
            rs.close();
        } catch(Exception ex) {
            Main.tfStatusOthers.setText("Error:111" + ex);
        }
        removeAttributeRedundancy(screen);

        for (int i=0 ; i< otherCompNameVector.size(); i++) {
```

```
            String otherCompName = (String)otherCompNameVector.elementAt(i);
            String otherCompAttribute = (String)otherCompAttributeVector.elementAt(i);
            String otherScreen = (String)otherScreenVector.elementAt(i);
            String setCode ="";
            String otherCompType = getComponentType(otherCompName, otherScreen);

            if (otherCompType.equals("TextField")) {
                if (otherCompAttribute.equals("Text")) {
                    setCode = "\t\t\t" + packedProject +".the" + otherScreen +
"Event."
                    + otherCompName + ".setText(value" + compName + ");\n\n";
                }
            } else if(otherCompType.equals("TextArea")) {
                if (otherCompAttribute.equals("Text")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".setText(value" + compName + ");\n\n";
                } else if (otherCompAttribute.equals("AppendText")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".appendText(value" + compName + ");\n\n";
                }
            } else if(otherCompType.equals("List")) {
                if (otherCompAttribute.equals("Item")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".addItem(value" + compName + ");\n\n";
                }
            } else if(otherCompType.equals("Checkbox")) {
                if (otherCompAttribute.equals("Label")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".setLabel(value" + compName + ");\n\n";
                }
            } else if(otherCompType.equals("Choice")) {
                if (otherCompAttribute.equals("AddItem")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".addItem(value" + compName + ");\n\n";
                } else if (otherCompAttribute.equals("SetItem")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".select(value" + compName + ");\n\n";
                }
            } else if(otherCompType.equals("Label")) {
                if (otherCompAttribute.equals("Text")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".setText(value" + compName + ");\n\n";
                }
            } else if(otherCompType.equals("Button")) {
                if (otherCompAttribute.equals("Label")) {
                    setCode = "\t\t\t" + packedProject + ".the" + otherScreen +
"Event."
                    + otherCompName + ".setLabel(value" + compName + ");\n\n";
                }
            }
            setComponentCode += setCode;
        }
        return setComponentCode;
    }

    public void removeAttributeRedundancy(String scr) {

        String screen = scr;

        for (int i=0; i < otherCompNameVector.size(); i++) {
            String otherScreen = (String)otherScreenVector.elementAt(i);

            if (otherScreen.equals(screen)) {
                otherCompNameVector.removeElementAt(i);
                otherCompAttributeVector.removeElementAt(i);
                otherScreenVector.removeElementAt(i);
            }
        }
    }
```

```
    public String getFromToStateAttributeCode(String scr) {
        String screen = scr;

        String getCompAttributeCode = "";
        String saveCompAttributeCode = "";

        String allCompAttributeCode = "";

        for (int i=0 ; i < FTCompNameVector.size(); i++) {
            String compName = (String)FTCompNameVector.elementAt(i);
            String compAttribute = (String)FTCompAttributeVector.elementAt(i);
            String compType = getComponentType(compName, screen);
            String getCode = writeGetTextCode(compName, compAttribute, compType);
            getCompAttributeCode += getCode;

            String saveCode = writeSaveTextCode(compName, compAttribute, screen);
            saveCompAttributeCode += saveCode;
        }
        allCompAttributeCode = getCompAttributeCode + saveCompAttributeCode;
        return allCompAttributeCode;
    }

    public String getStateAttributeCode( String scr) {
        String screen = scr;
        String getCompAttributeCode = "";
        String saveCompAttributeCode = "";
        String allCompAttributeCode = "";

        for (int i=0 ; i < stateCompNameVector.size(); i++) {
            String compName = (String)stateCompNameVector.elementAt(i);
            String compAttribute = (String)stateCompAttributeVector.elementAt(i);
            String compType = getComponentType(compName, screen);
            String getCode = writeGetTextCode(compName, compAttribute, compType);
            getCompAttributeCode += getCode;
            String saveCode = writeSaveTextCode(compName, compAttribute, screen);
            saveCompAttributeCode += saveCode;
        }
        allCompAttributeCode = getCompAttributeCode + saveCompAttributeCode;
        return allCompAttributeCode;
    }
    public void writeToFile(String scr) {
        String screen = scr;
        try {
            File outputFile = new File(screen +"Event.java");
            DataOutputStream dos = new DataOutputStream(new
FileOutputStream(outputFile));
            dos.writeBytes("import java.awt.event.*;\nimport java.awt.*;\n\npublic
class "+ screen + "Event extends " + screen
+ " implements ActionListener, ItemListener, TextListener {\n\n");
            //write a constructor
            dos.writeBytes("\n\n\t" + screen + "Event() {\n\n");
            dos.writeBytes("\n\t\t//to stop
running\n\t\tenableEvents(AWTEvent.WINDOW_EVENT_MASK);\n\n");

            //write addListener
            for (int i=0; i < addListenerVector.size(); i++ ) {
            String add = (String)addListenerVector.elementAt(i);
            dos.writeBytes(add);
        }
        dos.writeBytes("\n\n\t}\n\n\tpublic void processWindowEvent(WindowEvent e)
{\n\t\t" + "if (e.getID() == WindowEvent.WINDOW_CLOSING)
{\n\t\t\tsetVisible(false);\n\t\t\t"
+ "dispose();\n\t\t\tSystem.exit(0);\n\t\t}\n\t}\n\n");
        dos.writeBytes("\n\tpublic void actionPerformed(ActionEvent e) {\n\t\tObject
source = e.getSource();\n");

            //write actionListener
            for (int i=0; i<actionListenerVector.size(); i++) {
                String code = (String)actionListenerVector.elementAt(i);
                dos.writeBytes(code);
            }
            dos.writeBytes("\n\n\t}\n\n\tpublic void itemStateChanged(ItemEvent e)
{\n\t\tObject source = e.getSource();\n");
            //write itemListener
            for (int i=0; i<itemListenerVector.size(); i++) {
                String code = (String)itemListenerVector.elementAt(i);
                dos.writeBytes(code);
```

```
            }
            dos.writeBytes("\n\n\t}\n\n\tpublic void textValueChanged(TextEvent e)
{\n\t\tObject source = e.getSource();\n");

            //write textListener
            for (int i=0; i<textListenerVector.size(); i++) {
                String code = (String)textListenerVector.elementAt(i);
                dos.writeBytes(code);
            }
            dos.writeBytes("\n\t}\n\n\tpublic static void main(String[] args) {\n\t\t"
+ screen + "Event the"
            + screen + " = new " + screen + "Event();\n\t\tthe" + screen +
".setVisible(true);\n\n\t}\n\n}");
            dos.close();
        } catch (IOException ex) {
            Main.tfStatusOthers.setText("Error :117" + ex);
        }
    }
}
```