NATIONAL OPEN UNIVERSITY OF NIGERIA

CIT 811

User Interface Design and Erogonomics

Module 3

CIT 811 User Interface Design and Ergonomics Module 3

Course Developer/Writer
Dr. A. S. Sodiya, University of Agriculture, Abeokuta

Course Coordinator
A. A. Afolorunso, National Open University of Nigeria

Programme Leader
Prof. MoniOluwa Olaniyi

Credits of cover-photo: Henry Ude, National Open University of Nigeria

National Open University of Nigeria - University Village, Plot 91, Cadastral Zone, Nnamdi Azikiwe Expressway, Jabi, Abuja-Nigeria

www.nou.edu.ng centralinfo@nou.edu.ng
oer.nou.edu.ng oerunit@noun.edu.ng OER repository
Published in 2013, 2021 by the National Open University of Nigeria

Unit 1 Prototyping

1.0 Introduction

This unit will give you a general understanding of prototyping. User interface prototyping will be discussed in details as well as the various tips and techniques of prototyping.

2.0 Objectives

At the end of this unit, you should be able to:
- explain prototyping in detail
- describe the user interface prototyping process
- discuss various tips and techniques in prototyping
- describe interface-flow diagram.

3.0 Main Content

3.1 Prototyping

A prototype is a concrete, but partial implementation of a system design. Prototypes are used extensively in design and construction work; to demonstrate a concept (e.g. a prototype car), in early design, later design and as specification. It may be made of paper and cardboard or it may use a sophisticated software package.

Prototyping involves creating mock-ups representing the user interface of the final design. Prototypes serve as a common language with users, software engineers, and other stakeholders, offering a way for designers to explore design ideas and elicit feedback from users prior to committing to designs. Since prototyping helps flesh out requirements, prototypes may be used as a specification for developers. Prototyping is important in arriving at a well-designed user interface, and from many users' perspective the user interface is the software. Prototyping is very important for ensuring the most usable, accurate, and attractive design is found for the final product.

3.2 User Interface Prototyping Process

Prototyping is an iterative analysis technique in which users are actively involved in the mocking-up of screens and reports. The four major stages of prototyping are:

Determine the needs of your users
The requirements of your users drive the development of your prototype as they define the business objects that your system must support. You can gather these requirements in interviews, in CRC (Class Responsibility Collaborator) modelling sessions, in use-case modelling sessions, and in class diagramming sessions.

Build the prototype
Using a prototyping tool or high-level language you develop the screens and reports needed by your users. The best advice during this stage of the process is to not invest a lot of time

in making the code "good" because chances are high that you may just scrap your coding efforts anyway after evaluating the prototype.

Evaluate the prototype
After a version of the prototype is built it needs to be evaluated. The main goal is that you need to verify that the prototype meets the needs of your users. I've always found that you need to address three basic issues during evaluation: What's good about the prototype, what's bad about the prototype, and what's missing from the prototype. After evaluating the prototype you'll find that you'll need to scrap parts, modify parts, and even add brand-new parts.

Determine the end
You want to stop the prototyping process when you find the evaluation process is no longer generating any new requirements, or is generating a small number of not-so-important requirements.

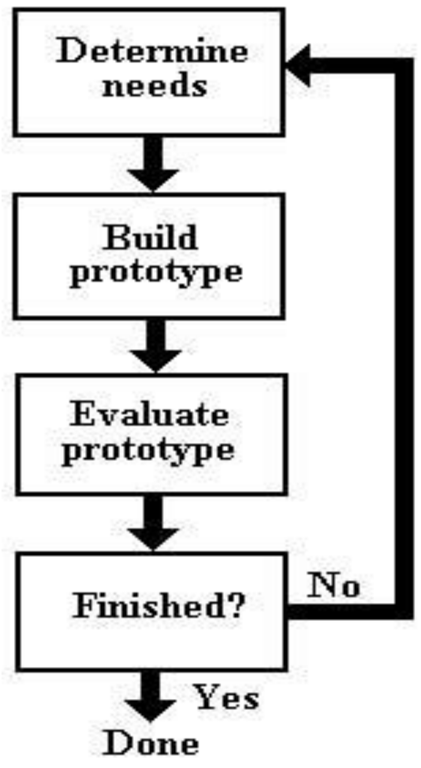


Fig. 1.1: The Iterative Steps of Prototyping

3.3 Prototyping Tips and Techniques

Let us now share with you several tips and techniques that you can use to create truly world-class prototypes.

Look for real-world objects
Good user interface allow users to work with the real-world objects they are used to. Therefore, you should start by looking for these kinds of objects and identify how people interact with them.

Work with the real users
The best people to get involved in prototyping are the ones who will actually use the application when it's done. These are the people who have the most to gain from a successful implementation, and these are the ones who know their own needs best.

Set a schedule and stick to it
Set a prototyping schedule that will guide you on how the prototyping will take place. The schedule will show when the prototype will be completed, who will evaluate it, expectations, etc.

Use a prototyping tool
The use of a prototyping tool will allow one to put screens together quickly. Prototyping tools can give one all possibilities fast and this will make designers to finally come with good design.

Get the users to work with the prototype
Just like you want to take a car for a test drive before you buy it, your users should be able to take an application for a test drive before it is developed. Furthermore, by working with the prototype hands-on, proposed users will quickly be able to determine whether or not the system will meet their needs. A good approach is to ask them to work through some use-case scenarios using the prototype as if it is the real system.

f. Understand the underlying business

You need to understand the underlying business before you can develop a prototype that will support it. Perform interviews with key users, read internal documentation of how the business runs, and read documentation about how some of your competitors operate. The more you know about the business, the more likely it is that you are able to build a prototype that supports it.

Know the different levels of prototype
The three different types of prototypes of a system are:

- a hand-drawn prototype that shows its basic/rough functionality
- an electronic prototype that shows the screens but not the data that will be displayed on them, and then finally
- the screens with data.

Start out simple in the beginning and avoid investing a lot of time in work that will most likely be thrown away. By successively increasing the complexity of the prototype as it gets closer to the final solution, many users get a better idea of how the application will actually work.

Do not spend a lot of time making the code good
At the beginning of the prototyping process, you will throw away a lot of your work as you learn more about the business. Therefore, it doesn't make sense to invest a lot of effort in code that you probably aren't going to keep anyway.

3.4 Interface-Flow Diagrams

To the users, the user interface is the system. In Figure 1.2, we see an example of an interface-flow diagram for an order-entry system. The boxes represent user interface objects (screens, reports, or forms) and the arrows represent the possible flow between screens. For example, when you are on the main menu screen you can go to either the customer search screen or to the order-entry screen. Once you are on the order-entry screen you can go to the product search screen or to the customer order list. Interface-flow diagrams allow you to easily gain a high-level overview of the interface for your application.

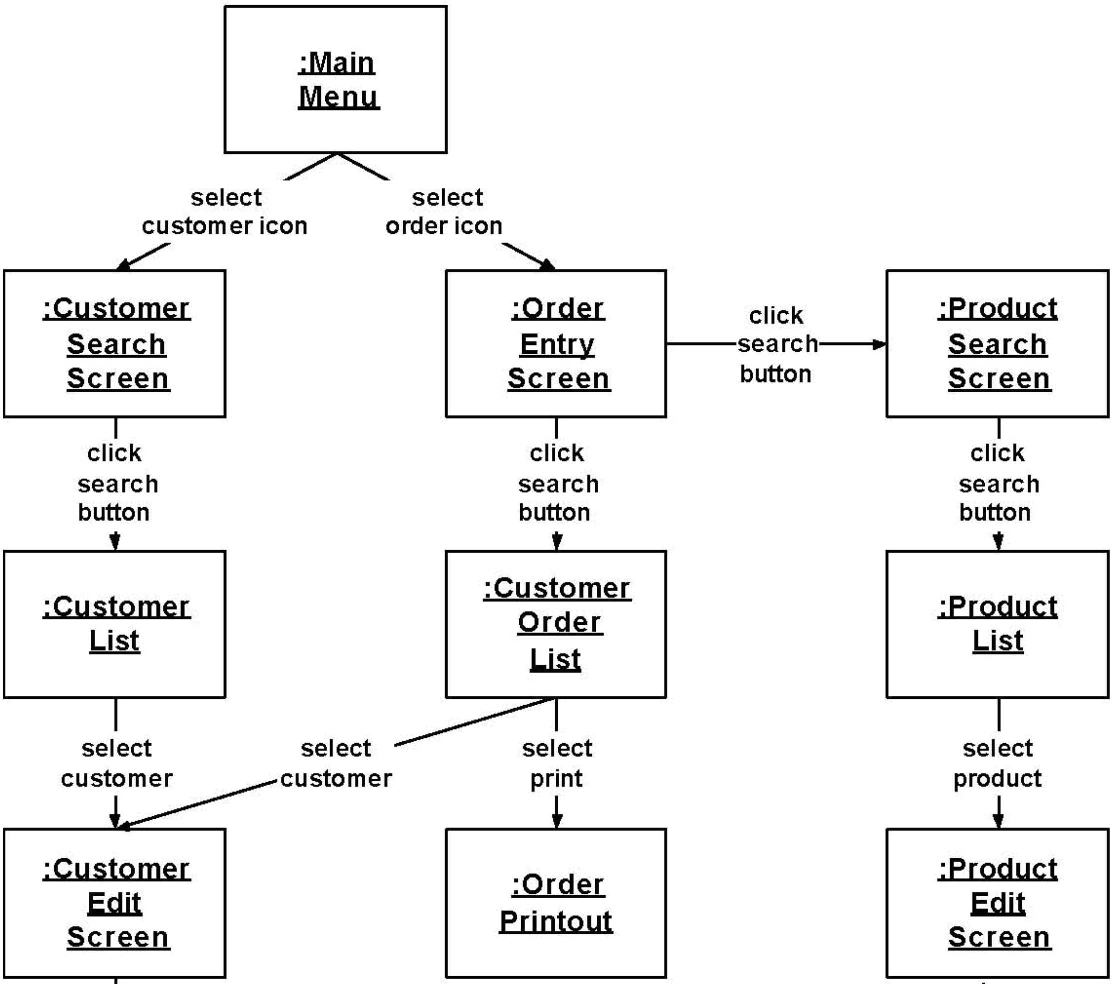


Fig. 1.2: An Interface-Flow Diagram for an Order-Entry System

Since interface-flow diagrams offer a high-level view of the interface of a system, one can quickly gain an understanding of how the system is expected to work. Also, interface-flow diagrams can be used to determine if the user interface has been designed consistently. For example, in Figure 1.2, you will see that to create the customer summary report and a printed order, you must select the print command. It appears from the diagram that the user interface is consistent, at least with respect to printing.

Self-Assessment Exercise

Practice more examples on how to draw interface-flow diagram. (The facilitator can assist in doing this).

4.0 Conclusion

In this unit, prototyping was discussed. Various tips and techniques of prototyping were highlighted. The concept of Interface-Flow Diagram was also introduced.

5.0 Summary

- In this unit, you have learnt the following:
- the requirements of your users drive the development of your prototype
- during evaluation ask: What's good about the prototype, what's bad about the prototype, and what's missing from the prototype
- stop the prototyping process when you find the evaluation process is generating few or no new requirements
- look for real-world objects and identify how users work with them
- work with the people who will use the application when it's done
- set a prototyping schedule and stick to it
- use a prototyping tool
- get the users to work with the prototype, to take it for a test drive
- understand the underlying business
- do not invest a lot of time in something that you'll probably throw away
- document interface objects once they have stabilised
- develop an interface-flow diagram for your prototype
- for each interface object that makes up a prototype, document
- ▪ its purpose and usage
- ▪ an indication of the other interface objects it interacts with
- ▪ the purpose and usage of each of its components.

6.0 Self-Assessment Exercise

1. List and explain the different prototyping techniques.
2. Design a payroll system and draw the interface-flow diagram.

7.0 References/Further Reading

Ambler, S.W. (2001). *The Object Primer* (2nd ed.). *The Application Developer's Guide to Object Orientation*. New York: Cambridge University Press.

Gilroy, CA: CMP Books. http://www.ambysoft.com/inceptionPhase.html.

Unit 2 Prototyping Methods and Tools

1.0 Introduction

You will be introduced to various prototyping methods and tools in this unit. Fidelity of prototypes will be discussed with emphasis on Low-Fidelity, Medium-Fidelity and High-Fidelity Prototypes. User Interface modes and modalities are also mentioned.

2.0 Objectives

At the end of this unit, you should be able to:

- explain various fidelities of prototypes
- describe the user interface modes and modalities
- explain the term widget.

3.0 Main Content

3.1 Fidelity of Prototypes

The term 'Fidelity' is used to determine how accurately a prototype resembles the final design in terms of visual appearance, interaction style, and level of detail.

The three main fidelity types are:

- Low-Fidelity prototypes
- Medium-Fidelity prototypes
- High-Fidelity prototype.

3.2 Low-Fidelity Prototypes

Low-fidelity prototypes, also known as Lo-fi prototypes, depict rough conceptual interface design ideas. Low-fidelity prototypes consist of little details of the actual interface. They are traditionally paper-based prototypes, making them quick, easy, and low-cost to create and modify. Low-fidelity prototypes are sketches of static screens, presented either separately or in a series to tell a specific story, which is called storyboarding. These prototypes convey the general look and feel of the user interface as well as basic functionality. Low-fidelity prototypes are particularly well suited for understanding screen layout issues but not for navigation and interaction issues. The purpose of low-fidelity prototypes is to try out alternative design ideas while seeking frequent feedback from users and other stakeholders. Low-fidelity prototypes are best used early in the design process when trying to understand basic user requirements and expectations.

3.2.1 Techniques Used in Low-Fidelity Prototypes

Sketching
Sketching is one of the most common techniques used in creating low-fidelity prototypes. It is a natural and low effort technique that allows for abstract ideas to be rapidly translated from a designer's conception onto a more permanent medium. Sketching is beneficial to the

design process because it encourages thinking and, ultimately, creativity. Sketches are also important to design because they are intentionally vague and informal, which allows for details to be later worked out without hindering the creative flow of the moment. This technique also encourages contributions from users and other stakeholders since it is in a visual form that everyone is familiar with.

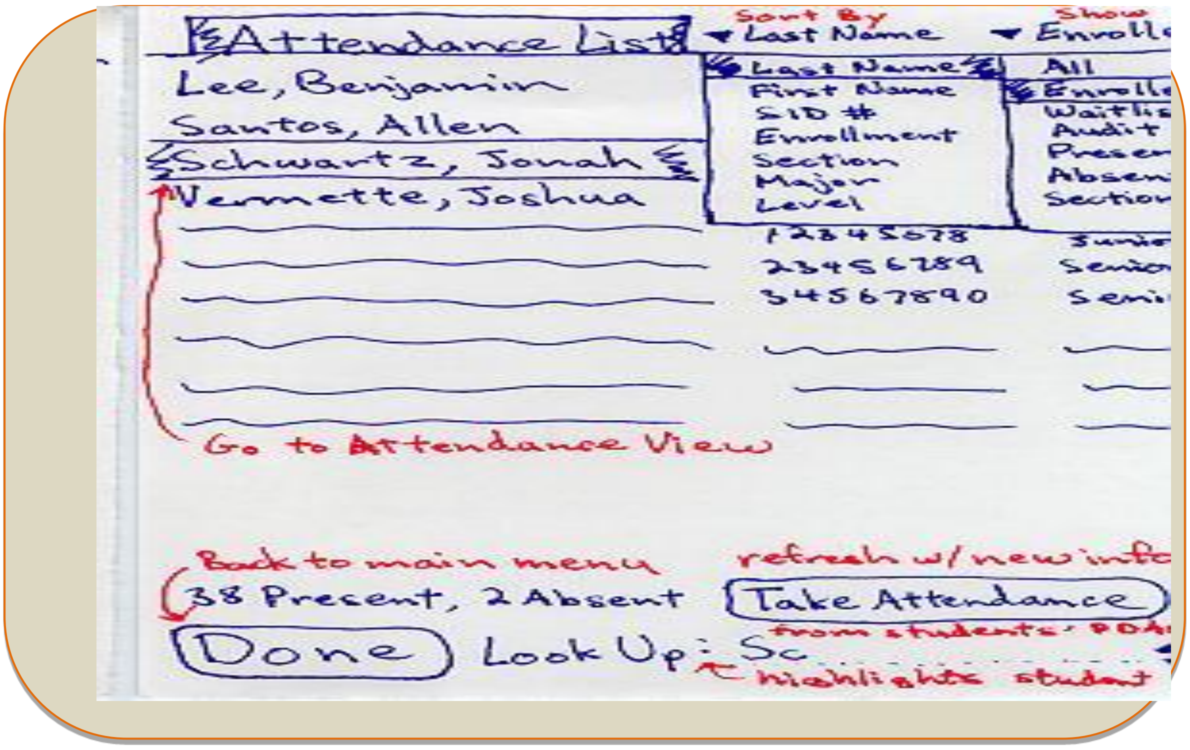


Fig. 2.1: Example of Sketching I

(Source: Petrie, J. (2006))

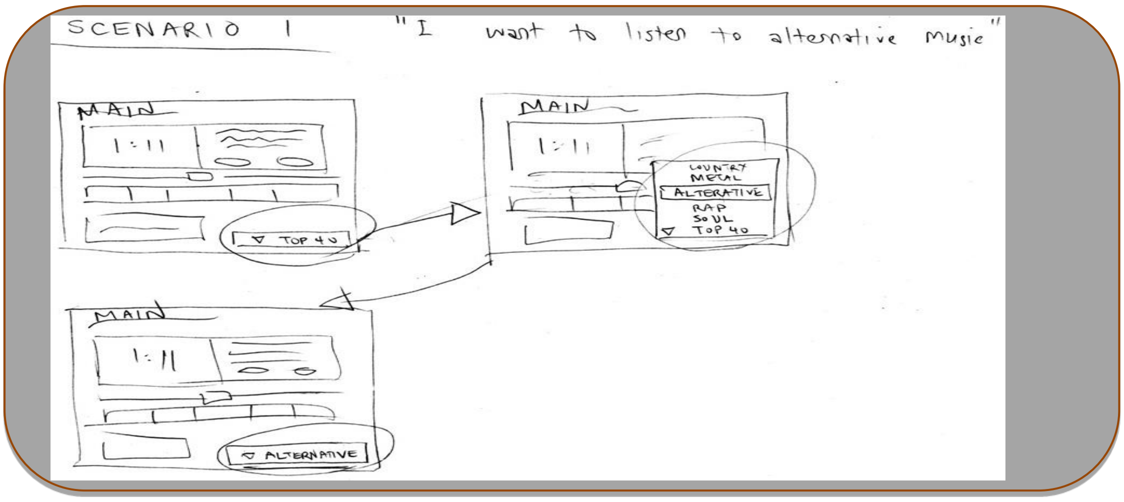
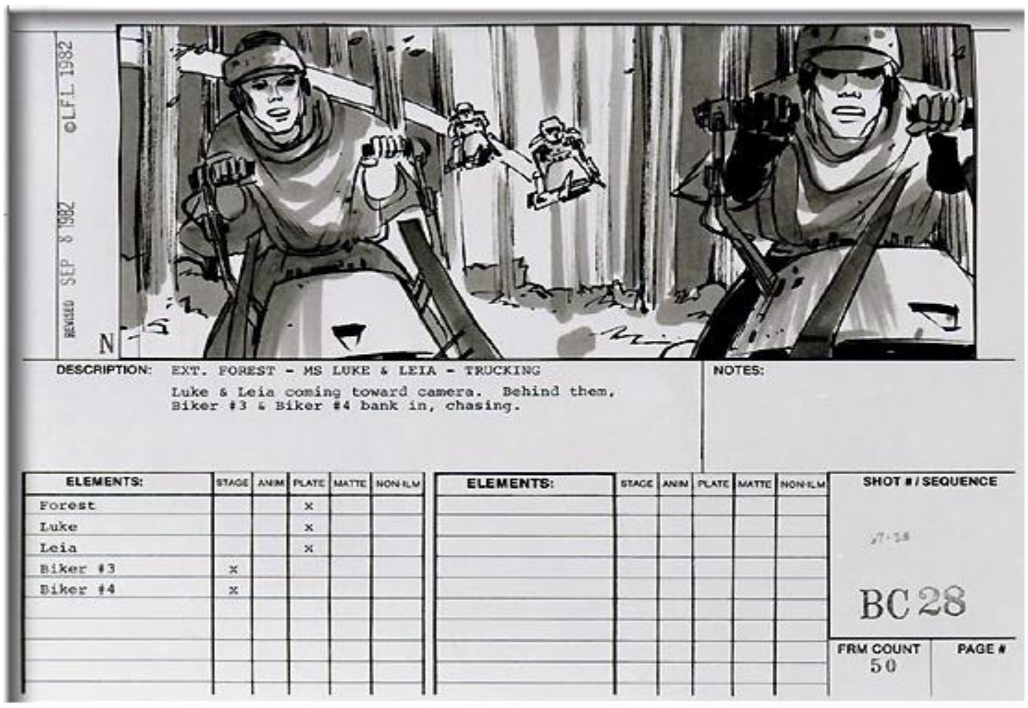


Fig. 2.2: Example of Sketching II

(Source: Petrie, J. (2006))

Storyboards

This Lo-fi technique is similar to that of sketching but differ in the sense that it is mainly obtained from films and animation. Storyboards give a script of important events by leaving out the details and concentrating on the important interactions.


©L.F.L 1982

REVISED SEP 8 1982

N

DESCRIPTION: EXT. FOREST - MS LUKE & LEIA - TRUCKING

Luke & Leia coming toward camera. Behind them, Biker #3 & Biker #4 bank in, chasing.

NOTES:

ELEMENTS:	STAGE	ANIM	PLATE	MATTE	NON-ILM
Forest			x		
Luke			x		
Leia			x		
Biker #3	x				
Biker #4	x				

SHOT # / SEQUENCE: BC 28

FRM COUNT: 50

PAGE #

Fig. 2.3: An Example of Storyboard I

(Source: Petrie, J. (2006))

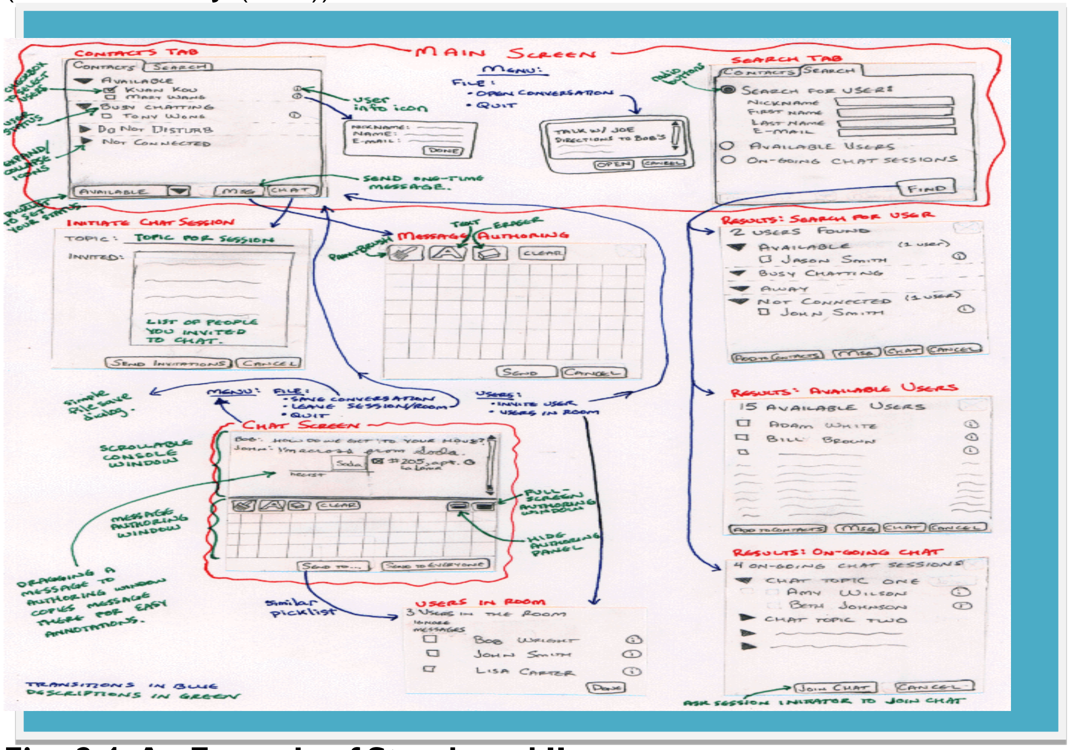


Fig. 2.4: An Example of Storyboard II

(Source: Petrie, J. (2006))

Pictive
The other commonly used low-fidelity prototyping technique is the PICTIVE technique. PICTIVE also called Plastic Interface for Collaborative Technology Initiatives through Video Exploration, is a technique for creating and modifying prototypes in real-time with users. The PICTIVE technique involves using standard office supplies such as sticky notes, labels, and plastic overlays as well as paper copies of pre-built interface components such as buttons, icons, and menus. Materials are transformed through cutting and labelling to represent desired interface elements. Multiple layers of these elements are attached to the paper prototype, as needed, to demonstrate interaction to the users. PICTIVE is a flexible technique that encourages active user participation. As the name suggests, PICTIVE prototyping sessions with the users may be videotaped and later analysed by designers to see how the prototypes evolved and how users responded to different designs.

3.2.2 Advantages of Low-Fidelity Prototypes

- takes only a few hours. No expensive equipment needed
- fast iterations. It can test multiple alternatives and the number of iterations is tied to final quality
- almost all iterations can be faked. Example of this is got from the film ”Wizard of OZ” and “The man behind the curtain”
- it can evolve into implementation
- much more important for features that are difficult to implement such as speech and handwriting recognition.

3.3 Medium-Fidelity Prototypes

Medium-fidelity prototypes lie on the continuum between low- and high-fidelity prototypes, thus sharing some of the advantages and disadvantages of the other two fidelities. Medium-fidelity prototypes are refined versions of the low-fidelity prototypes and are created on computer. They are best used after low-fidelity prototyping once only a small number of alternative designs remain under consideration and require further refinement. They resemble the end product more than low-fidelity prototypes and require less effort than high-fidelity prototypes. Medium-fidelity prototypes are commonly created using multimedia design tools, interface builders, or certain scripting languages.

3.4 High-Fidelity Prototypes

High-fidelity prototypes also known as Hi-fi prototypes allow users to interact with the prototypes as though they are the end product. High-fidelity prototypes strongly represent the final product in terms of visual appearance, interactivity, and navigation. As well, high-fidelity prototypes usually have some level of functionality implemented and may link to some sample data. High-fidelity prototypes are computer-based prototypes that are often developed using interface builders or scripting languages such as tcl/tk to speed up the process. High-fidelity prototypes are particularly useful for performing user evaluations as well as for serving as a specification for developers and as a tool for marketing and stakeholder buy-in. On the negative side, these prototypes are time-consuming and costly to develop. As such, high-fidelity prototypes are best used near the end of the design phase once the user interface requirements have been fully understood and a single design has been agreed upon.

3.5 User Interface Prototyping Tools

Today, the majority of applications are developed using some type of user interface prototyping tool such as an **interface builder** or **a multimedia design tool**. However, there are several limitations in using these types of tools that hinder the design process. In attempts to overcome some of these limitations, researchers in the academic community have been investigating informal prototyping tools such as **SILK, DENIM**, and **Freeform**. While useful, these tools are not without drawbacks.

Interface Builders

Interface builders are tools for creating and laying out user interfaces by allowing interface components to be dragged and placed into position on the desired window or dialog box. Some commercial examples include Microsoft Visual Basic, Java's NetBeansTM, Borland DelphiTM, and MetrowerksTM CodeWarriorTM. While interface builders are primarily intended for final product implementation, they are useful for medium- and high-fidelity prototyping.

Interface builders are commonly used for the following reasons: They

- visually represent visual concepts such as layout
- speed up implementation by auto-generating certain code
- are generally easy to use even for non-programmers.

On the other hand, interface builders are restrictive in terms of what designs designers can build and the order in which designers have to build it. Also, interface builders require significant time and effort to create a prototype. Thus they are not suitable for early stages of prototyping when many alternate and ill-defined design concepts need to be explored.

Multimedia Design Tools

Multimedia design tools are often used in designing user interfaces, not because they are particularly well suited for software interfaces, but rather because of the lack of better prototyping-specific tools. Multimedia tools are useful in creating and demonstrating storyboards in medium-fidelity prototyping. Specifically, multimedia tools allow for creation of images that can represent user interface screens and components. They also allow for playing out transitions from one screen to the next that can convey the general picture of a user navigating through the interface screens. On the negative side, the interactivity supported by multimedia design tools is very limited, usually to only basic mouse clicks, and so is support for creating functionality and tying in data. Examples of commonly used commercial multimedia design tools are Macromedia Director and Flash. Apple HyperCard is another commercial tool that has been widely used in the past. There are several multimedia tools of the user interface research community including DEMAIS and Anecdote.

Silk

SILK was one of the first tools to support informal sketching of user interfaces. The purpose of SILK is to preserve the benefits of paper-based sketching while taking advantage of computerised design tools' features. The main features of SILK include support for stylus-based freehand sketching, annotation layers that allow for notes to be created and associated with specific user interface elements, and a run mode to show screen transitions. Also, SILK attempts to provide support for transitioning to higher-fidelity prototyping through automatic interface component recognition and transformation to real components; however, this feature is not suitable for low-fidelity prototyping and restricts the designer to existing toolkit components and interactions.

Denim

DENIM, an extension of SILK, is a tool aimed at supporting the early stages of web design through informal sketching. While DENIM is intended for website design, many of the features and concepts are applicable to the design of most graphical user interfaces.

DENIM provides the following design features:

- informal pen-based sketching
- zooming to different levels of granularity (from sitemap to storyboards to individual pages)
- linking of pages to create storyboards
- a run mode to preview and interact with a prototype site
- supporting the creation of reusable components.

These features make DENIM appropriate for low-fidelity and medium-fidelity prototyping. On the other hand, DENIM provides little support for transitioning to high-fidelity prototyping as shown in the figure below.

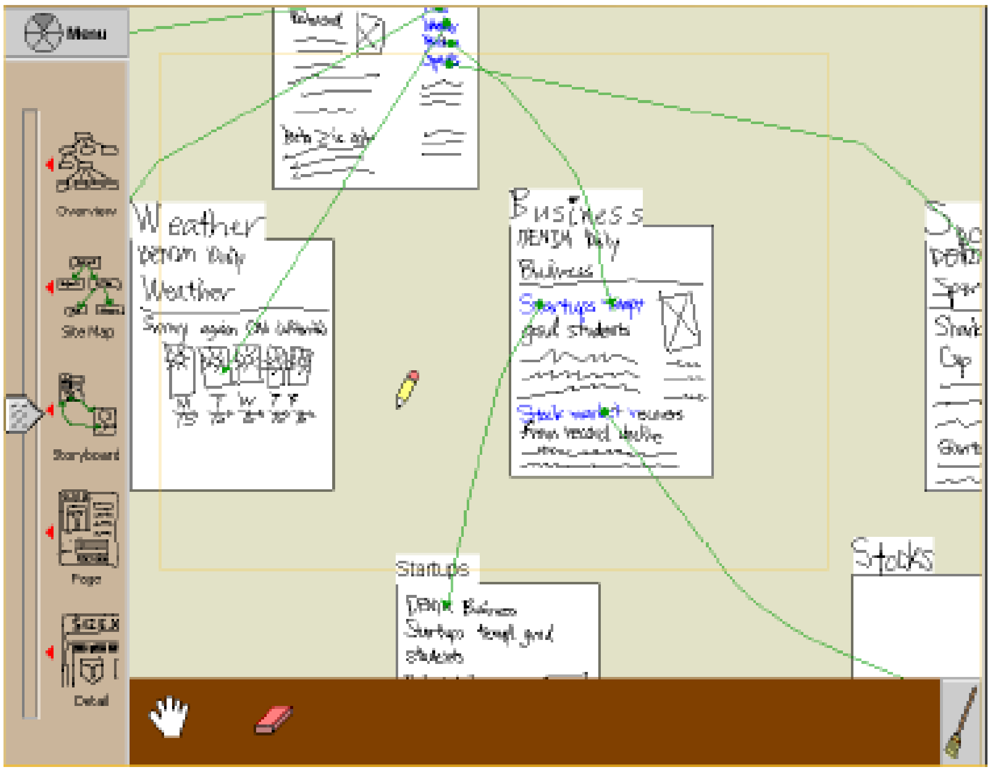


Fig. 2.5: Showing an Example of DENIM Prototyping Screen

(Source: Petrie, J. (2006))

Freeform

Freeform is another tool that supports sketching of user interfaces. It also aims at supporting high-fidelity prototyping. Specifically, Freeform is a Visual Basic add-in that translates the recognised sketched interface components and text into VB code. However, this feature restricts the interface designer to simple forms-based interfaces that use standard Visual Basic components. Also, the interfaces generated from sketches are not very visually appealing. Freeform is intended for use on large displays; however, there are no unique features in Freeform that make it better suited for use on large displays versus traditional desktop displays as shown in the figure below:

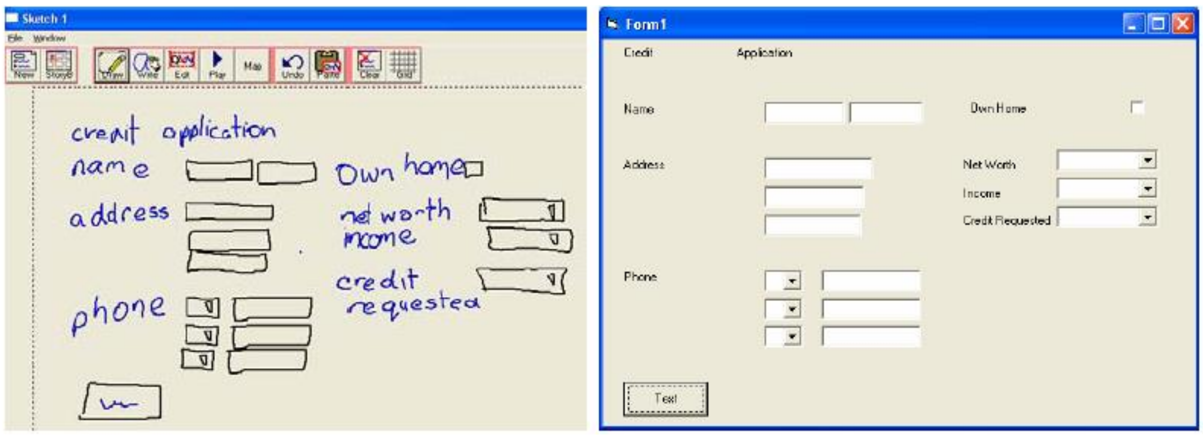


Fig. 2.6: Showing an Example of Freeform Prototyping Screen

3.5.1 Analysis of Prototyping Tools

Each type of user interface prototyping tool discussed above offers some unique features or advantages to designers. However, most tools are designed for very specific purposes and have very little or no support for transitioning between each of the different fidelities. None of the tools support mixing of all three fidelities within a single prototype. Also, none of the tools are specifically designed for collaborative use on a large display; however, simply using tools such as DENIM on a large display may offer some benefits over use on a traditional desktop display.

Self-Assessment Exercise

Get DENIM or Freeform tool and practice how to use them to create interface layout.

3.6 WIDGETS

Widgets are reusable user interface components. As stated in module 1, they are also called controls, interactors, gizmos, or gadgets. Some examples of widgets have been given in unit 5 of module 1. **Widgets** are user interface toolkits. Widgets are a success story for user interface software, and for object-oriented programming in general. Many GUI applications derive substantial reuse from widgets in a toolkit.

Advantages of Using Widgets:

- reuse of development efforts
- coding, testing, debugging, maintenance
- iteration and evaluation.

Disadvantages:

- constrain designer's thinking
- encourage menu & forms style, rather than richer direct manipulation style
- may be used inappropriately.

Widget reuse is beneficial in two ways: First are the conventional software engineering benefits of reusing code, like shorter development time and greater reliability. A widget encapsulates a lot of effort that somebody else has already put in. Second are usability benefits. Widget reuse increases consistency among the applications on a platform. It also (potentially) represents *usability* effort that its designers have put into it. A scrollbar's affordances and behaviour have been carefully designed, and hopefully evaluated. By reusing the scrollbar widget, you don't have to do that work yourself.

One problem with widgets is that they constrain your thinking. If you try to design an interface using a GUI builder – with a palette limited to standard widgets – you may produce more complex interface than you would if you sat down with paper and pencil and allowed yourself to think freely. A related problem is that most widget sets consist mostly of form-style widgets: text fields, labels, checkboxes – which leads a designer to think in terms of menu/form style interfaces. There are few widgets that support direct visual representations of application objects, because those representations are so application-dependent. So if you think too much in terms of widgets, you may miss the possibilities of direct manipulation.

Finally, widgets can be abused, applied to UI problems for which they are not suited. An example is when a scrollbar is used for selection, rather than scrolling.

Widgets generally combine a view and a controller into a single tightly-coupled object. For the widget's model, however, there are two common approaches. One is to fuse the model into the widget as well, making it a little MVC complex. With this embedded model approach, application data must be copied into the widget to initialise it. When the user interacts with the widget, the user's changes or selections must be copied back out.

The other alternative is to leave the model separate from the widget, with a well-defined interface that the application can implement. Embedded models are usually easier for the developer to understand and use for simple interfaces, but suffer from serious scaling problems. For example, suppose you want to use a table widget to show the contents of a database. If the table widget had an embedded model, you would have to fetch the entire database and load it into the table widget, which may be prohibitively slow and memory-intensive. Furthermore, most of this is wasted work, since the user can only see a few rows of the table at a time. With a well-designed linked model, the table widget will only request as much of the data as it needs to display.

The linked model idea is also called **data binding**.

Generally, every user interface consists of the following:
- components (view hierarchy)
- stroke drawing
- pixel model
- input handling
- widgets

Every modern GUI toolkit provides these pieces in some form. Microsoft Windows, for example, has widgets (e.g., buttons, menus, text boxes), a view hierarchy (consisting of *windows* and *child windows*), a stroke drawing package (GDI), pixel representations (called bitmaps), and input handling (messages sent to a *window procedure*).

User interface toolkits are often built on top of other toolkits, sometimes for portability or compatibility across platforms, and sometimes to add more powerful features, like a richer stroke drawing model or different widgets.

X Windows demonstrates this layering technique. The view hierarchy, stroke drawing, and input handling are provided by a low-level toolkit called XLib. But XLib does not provide widgets, so several toolkits are layered on top of XLib to add that functionality: Athena widgets and Motif, among others. More recent X-based toolkits (GTK+ and Qt) not only add widgets to XLib, but also hide XLib's view hierarchy, stroke drawing, and input handling with newer, more powerful models, although these models are implemented internally by calls to XLib.

Here is what the layering looks like for some common Java user interface toolkits.

AWT (Abstract Window Toolkit, usually pronounced like "ought") was the first Java toolkit. Although its widget set is rarely used today, AWT continues to provide drawing and input handling to more recent Java toolkits. Swing is the second-generation Java toolkit, which appeared in the Java API starting in Java 1.2.

Swing adds a new view hierarchy (JComponent) derived from AWT's view hierarchy (Component and Container). It also replaces AWT's widget set with new widgets that use the new view hierarchy. subArctic was a research toolkit developed at Georgia Tech. Like Swing, subArctic relies on AWT for drawing and input handling, but provides its own widgets and views.

Not shown in the picture is SWT, IBM's Standard Widget Toolkit. (Usually pronounced "swit").

Confusingly, the W in SWT means something different from the W in AWT). Like AWT, SWT is implemented directly on top of the native toolkits. It provides different interfaces for widgets, views, drawing, and input handling.

Cross-platform toolkits face a special issue: should the native widgets of each platform be reused by the toolkit? One reason to do so is to preserve consistency with other applications on the same platform, so that applications written for the cross-platform toolkit look and feel like native applications. This is what we've been calling external consistency.

Another problem is that native widgets may not exist for all the widgets the cross-platform toolkit wants to provide. AWT throws up its hands at this problem, providing only the widgets that occur on every platform AWT runs on: e.g., buttons, menus, list boxes and text boxes.

The reason the reuse of native widgets is NOT allowed is that the application looks and behaves consistently with itself across platforms – a variant of internal consistency, if you consider all the instantiations of an application on various platforms as being part of the same system. Cross-platform consistency makes it easier to deliver a well-designed, usable application on all platforms – easier to write documentation and training materials, for example. Java Swing provides this by re-implementing the widget set using its default ("Metal") look and feel. This essentially creates a Java "platform", independent of and distinct from the native platform.

3.6.1 Piccolo Toolkit

Piccolo is a novel UI toolkit developed at University of Maryland. Piccolo is specially designed for building **zoomable** interfaces, which use smooth animated panning and zooming around a large space. We can look at Piccolo in terms of the various aspects we have discussed in this unit.

Layering: First, Piccolo is a layered toolkit: it runs on top of Java Swing. It also runs on top of .NET, making it a cross-platform toolkit. Piccolo ignores the platform widgets entirely, making no attempt to re-implement or reuse them (An earlier version of Piccolo, called Jazz, could reuse Swing widgets.)

Components: Piccolo has a view hierarchy consisting of PNode objects. The hierarchy is not merely a tree, but in fact a graph: you can install camera objects in the hierarchy which act as viewports to other parts of the hierarchy, so a component may be seen in more than one place on the screen. Another distinction between Piccolo and other toolkits is that every component has an arbitrary transform relative to its parent's coordinate system – not just translation (which all toolkits provide), but also rotation and scaling. Furthermore, in Piccolo, parents do not clip their children by default. If you want this behaviour, you have to request it by inserting a special clipping object (a component) into the hierarchy. As a result, components in Piccolo have two bounding boxes – the bounding box of the node itself (getBounds()), and the bounding box of the node's entire subtree (getFullBounds()).

Strokes: Piccolo uses the Swing Graphics package, augmented with a little information such as the camera and transformations in use.

Pixels: Piccolo uses Swing images for direct pixel representations.

Input: Piccolo has the usual mouse and keyboard input (encapsulated in a single event-handling interface called BasicInput), plus generic controllers for common operations like dragging, panning, and zooming. By default, panning and zooming is attached to any camera you create: dragging with the left mouse button moves the camera view around, and dragging with the right mouse button zooms in and out.
Widgets: the widget set for Piccolo is fairly small by comparison with toolkits like Swing and .NET, probably because Piccolo is a research project with limited resources. It's worth noting, however, that Piccolo provides reusable components for shapes (e.g. lines, rectangles, ellipses, etc), which in other toolkits would require revering to the stroke model.

3.7 The Wizard of Oz Prototype

The Wizard of Oz is a 1939 American musical-fantasy film mainly directed by Victor Fleming and based on the 1900 children's novel *The Wonderful Wizard of Oz* by L. Frank Baum.
A **Wizard of Oz prototype** uses a human in the backend, but the frontend is an actual computer system instead of a paper mock-up. The term Wizard of Oz comes from the movie of the same name, in which the wizard was a man hiding behind a curtain, controlling a massive and impressive display.

In a Wizard of Oz prototype, the "wizard" is usually but not always hidden from the user. Wizard of Oz prototypes are often used to simulate future technology that is not available yet, particularly artificial intelligence. A famous example was the listening typewriter. This study sought to compare the effectiveness and acceptability of isolated-word speech recognition, which was the state of the art in the early 80's, with continuous speech recognition, which wasn't possible yet. The interface was a speech-operated text editor. Users looked at a screen and dictated into a microphone, which was connected to a typist (the wizard) in another room. Using a keyboard, the wizard operated the editor showing on the user's screen.

The wizard's skill was critical in this experiment. She could type 80 wpm, she practiced with the simulation for several weeks (with some iterative design on the simulator to improve her interface), and she was careful to type *exactly* what the user said, even exclamations and parenthetical comments or asides. The computer helped make her responses a more accurate simulation of computer speech recognition. It looked up every word she typed in a fixed dictionary, and any words that were not present were replaced with X's, to simulate misrecognition. Furthermore, in order to simulate the computer's ignorance of context, homophones were replaced with the most common spelling, so "done" replaced "dun", and "in" replaced "inn". The result was an extremely effective illusion. Most users were surprised when told (midway through the experiment) that a human was listening to them and doing the typing.

Thinking and acting mechanically is harder for a wizard than it is for a paper prototype simulator, because the tasks for which Wizard of Oz testing is used tend to be more "intelligent". It helps if the wizard is personally familiar with the capabilities of similar interfaces, so that a realistic simulation can be provided. It also helps if the wizard's interface can intentionally dumb down the responses, as was done in the Gould study.
A key challenge in designing a Wizard of Oz prototype is that you actually have two interfaces to worry about: the user's interface, which is presumably the one you're testing, and the wizard's.

4.0 Conclusion

In this unit, you have been introduced to various prototyping methods and tools. Fidelity of prototypes like Low-Fidelity, Medium-Fidelity and High-Fidelity Prototypes was also discussed in detail. User interface modes and modalities was also introduced along with a discussion on widgets.

5.0 Summary

In this unit, you have learnt:

- the fidelity of prototypes refers to how accurately the prototypes resemble the final design in terms of visual appearance, interaction style, and level of detail
- low-fidelity prototypes, also known as Lo-fi prototypes, depict rough conceptual interface design ideas. Low-fidelity prototypes consist of little details of the actual interface
- high-fidelity prototypes also known as Hi-fi prototypes allow users to interact with the prototypes as though they are the end product. High-fidelity prototypes strongly represent the final product in terms of visual appearance, interactivity, and navigation
- user interface prototyping tools include **interface builder** or **multimedia design tools**. However, there are several limitations in using these types of tools because they hinder the design process
- interface builders are tools for creating and laying out user interfaces by allowing interface components to be dragged and placed into position on the desired window or dialog box
- multimedia design tools are often used in designing user interfaces, not because they are particularly well suited for software interfaces, but rather because of the lack of better prototyping-specific tools

- widgets are reusable user interface components. They are also called controls, interactors, gizmos, or gadgets.

6.0 Self-Assessment Exercise

1. Explain mixed-fidelity prototyping.
2. What is the significance of user interface design?

7.0 References/Further Reading

Ambler, S.W. (2001). *The Object Primer* (2nd ed.). *The Application Developer's Guide to Object Orientation*. New York: Cambridge University Press.

Petrie, J. (2006). "Mixed-Fidelity Prototyping of User Interfaces." M.Sc. Thesis. Piccolo home page: http://www.cs.umd.edu/hcil/ piccolo/

Unit 3 Input and Output Models

1.0 Introduction

This unit looks at mechanics of implementing user interfaces, by looking at **input** and **output model** in detail. Different kinds of both input and output events will also be examined in greater detail.

2.0 Objectives

At the end of this unit, you should be able to:

- explain the input model
- describe the output model.

3.0 Main Content

3.1 Input Model

Virtually all GUI toolkits use event handling for input. Why? Recall, when you first learned to program, you probably wrote user interfaces that printed a prompt and then waited for the user to enter a response. After the user gave their answer, you produced another prompt and waited for another response. Command-line interfaces (e.g. the Unix shell) and menu-driven interfaces (e.g. Pine) have interfaces that behave this way. In this user interface style, the system has complete control over the dialogue – the order in which inputs and outputs will occur.

Interactive graphical user interfaces cannot be written this way at least, not if they care about giving the user control and freedom. One of the biggest advantages of GUIs is that a user can click anywhere on the window, invoking any command that's available at the moment, interacting with any view that's visible. In a GUI, the balance of power in the dialogue swings strongly over to the user's side.

As a result, GUI programs cannot be written in a synchronous, prompt-response style. A component can't simply take over the entire input channel to wait for the user to interact with it, because the user's next input may be directed to some other component on the screen instead. So GUI programs are designed to handle input asynchronously, receiving it as events.

3.1.1 Kinds of Input Events

There are two major categories of input events: raw and translated.
A raw event comes right from the device driver. Mouse movements, mouse button down and up, and keyboard key down and up are the raw events seen in almost every capable GUI system. A toolkit that does not provide separate events for down and up is poorly designed, and makes it difficult or impossible to implement input effects like drag-and-drop or video game controls. For many GUI components, the raw events are too low-level, and must be translated into higher-level events. For example, a mouse button press and release is translated into a mouse click event (assuming the mouse didn't move much between press

and release, if it did, these events would be translated into a drag rather than a click). Key down and up events are translated into character typed events, which take modifiers into account to produce an ASCII character rather than a keyboard key.

If you hold a key down, multiple character typed events may be generated by an auto-repeat mechanism. Mouse movements and clicks also translate into keyboard focus changes. When a mouse movement causes the mouse to enter or leave a component's bounding box, entry and exit events are generated, so that the component can give feedback e.g., visually highlighting a button, or changing the mouse cursor to a text I-bar or a pointing finger.

3.1.2 Characteristics of an Input Event

Input events have some or all of these properties. On most systems, all events include the modifier key state, since some mouse gestures are modified by Shift, Control, and Alt. Some systems include the mouse position and button state on all events; some put it only on mouse-related events.

The timestamp indicates when the input was received, so that the system can time features like autorepeat and double-clicking. It is essential that the timestamp be a property of the event, rather than just read from the clock when the event is handled. Events are stored in a queue, and an event may languish in the queue for an uncertain interval until the application actually handles it.

User input tends to be delayed at times – many seconds may go by while the user is thinking, followed by a flurry of events. The event queue provides a buffer between the user and the application, so that the application doesn't have to keep up with each event in a burst. Recall that perceptual fusion means that the system has 100 milliseconds in which to respond.

Edge events (button down and up events) are all kept in the queue unchanged. But multiple events that describe a continuing state – in particular, mouse movements – may be **coalesced** into a single event with the latest known state. Most of the time, this is the right thing to do. For example, if you're dragging a big object across the screen, and the application can't repaint the object fast enough to keep up with your mouse, you don't want the mouse movements to accumulate in the queue, because then the object will lag behind the mouse pointer, diligently (and foolishly) following the same path your mouse did.

Sometimes, however, coalescing hurts. If you're sketching a freehand stroke with the mouse, and some of the mouse movements are coalesced, then the stroke may have straight segments at places where there should be a smooth curve. If application delays are bursty, then coalescing may hurt even if your application can usually keep up with the mouse.

The event loop reads events from the queue and dispatches them to the appropriate components in the view hierarchy. On some systems (notably Microsoft Windows), the event loop also includes a call to a function that translates raw events into higher-level ones. On most systems, however, translation happens when the raw event is added to the queue, not when it is removed.

Every GUI program has an event loop in it somewhere. Some toolkits require the application programmer to write this loop (e.g., Win32); other toolkits have it built-in (e.g., Java Swing). Unfortunately, Java's event loop is written as essentially an infinite loop, so the event loop threads never cleanly exits. As a result, the normal clean way to end a Java program – waiting until all the threads are finished – doesn't work for GUI programs. The only way to end a Java GUI program is System.exit(). This despite the fact that Java best practices say *not* to use System.exit(), because it doesn't guarantee to garbage collect and run finalisers.

Swing lets you configure your application's main JFrame with EXIT_ON_CLOSE behaviour, but this is just a shortcut for calling System.exit().

Event dispatch chooses a component to receive the event. Key events are sent to the component with the keyboard focus, and mouse events are generally sent to the component under the mouse. An exception is **mouse capture**, which allows any component to grab all mouse events (essentially a mouse analogue for keyboard focus). Mouse capture is done automatically by Java when you hold down the mouse button to drag the mouse. Other UI toolkits give the programmer direct access to mouse capture – in the Windows API, for example, you'll find a SetMouseCapture function.

If the target component declines to handle the event, the event propagates up the view hierarchy until some component handles it. If an event bubbles up to the top without being handled, it is ignored.

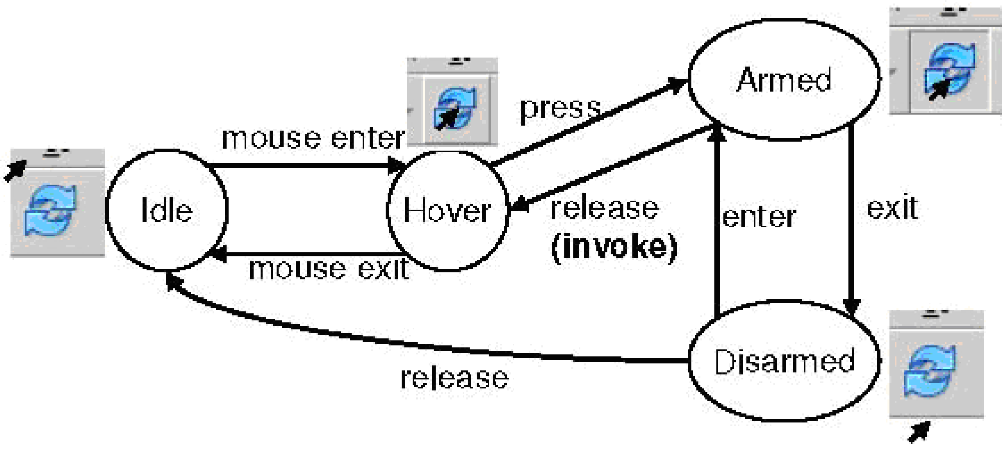


Fig. 3.1The Model for Designing a Controller (From MIT Open Courseware)

Now let's look at how components that handle input are typically structured. A controller in a direct manipulation interface is a **finite state machine**. Here's an example of the state machine for a push button's controller. **Idle** is the normal state of the button when the user isn't directing any input at it. The button enters the **Hover** state when the mouse enters it. It might display some feedback to reinforce that it affords clickability. If the mouse button is then pressed, the button enters the **Armed** state, to indicate that it is being pushed down. The user can cancel the button press by moving the mouse away from it, which goes into the **Disarmed** state. Or the user can release the mouse button while still inside the component, which invokes the button's action and returns to the **Hover** state.

Transitions between states occur when a certain input event arrives, or sometimes when a timer times out. Each state may need different feedback displayed by the view. Changes to the model or the view occur on transitions, not states: e.g., a push button is actually invoked by the release of the mouse button.

An alternative approach to handling low-level input events is the **interactor** model, introduced by the Garnet and Amulet research toolkits from CMU. Interactors are generic, reusable controllers, which encapsulate a finite state machine for a common task. They're mainly useful for the component model, in which the graphic output is represented by objects that the interactors can manipulate.

3.2 Output Model

Output model describes the ways in which output of GUIs are represented. The three basic ways are:

Components are parts of the display that are represented by view objects arranged in a spatial hierarchy, with automatic redraw propagating down the hierarchy. There have been many names for this idea over the years; the GUI community has not managed to settle on a single preferred term.

Strokes draws output by making calls to high-level drawing primitives, like drawLine, rawRectangle, drawArc, and drawText.

Pixels regard the screen as an array of pixels and deals with the pixels directly.

All three output models appear in virtually every modern GUI application. The component model always appears at the very top level, for windows, and often for components within the windows as well. At some point, we reach the leaves of the view hierarchy, and the leaf views draw themselves into stroke calls. A graphics package then converts those strokes into pixels displayed on the screen.

For performance reasons, a component may short-circuit the stroke package and draw pixels on the screen directly. On Windows, for example, video players do this using the DirectX interface to have direct control over a particular screen rectangle.
What model does each of the following representations use? HTML (component); Postscript laser printer (stroke input, pixel output); plotter (stroke input and output); PDF (stroke); LCD panel (pixel).

Since every application uses all three models, the design question becomes: at which points in your application do you want to step down into a lower-level output model? Here's an example. Suppose you want to build a view that displays a graph of nodes and edges.

One approach would represent each node and edge in the graph by a component. Each node in turn might have two components, a rectangle and a label. Eventually, you'll get down to primitive components available in your GUI toolkit. Most GUI toolkits provide a label component; most don't provide a primitive circle component. One notable exception is Amulet, which has component equivalents for all the common drawing primitives. This would be a **pure component model**, at least from your application's point of view – stroke output and pixel output would still happen, but inside primitive components that you took from the library.

Alternatively, the top-level window might have *no* subcomponents. Instead, it would draw the entire graph by a sequence of stroke calls: drawRectangle for the node outlines, drawText for the labels, drawLine for the edges. This would be a **pure stroke model**.

Finally, your graph view might bypass stroke drawing and set pixels in the window directly. The text labels might be assembled by copying character images to the screen. This **pure pixel model** is rarely used nowadays, because it's the most work for the programmer, but it used to be the only way to program graphics.

Hybrid models for the graph view are certainly possible, in which some parts of the output use one model, and others use another model. The graph view might use components for nodes, but draw the edges itself as strokes. It might draw all the lines itself, but use label components for the text.

3.2.1 Issues in Choosing Output Model

The major considerations in choosing an output model are:
Layout: Components remember where they were put, and draw themselves there. They also support automatic layout. With stroke or pixel models, you have to figure out (at drawing time) where each piece goes, and put it there.

Input: Components participate in event dispatch and propagation, and the system automatically does **hit-testing** (determining whether the mouse is over the component when an event occurs) for components, but not for strokes. If a graph node is a component, then it can receive its own click and drag events. If you stroked the node instead, then you have to write code to determine which node was clicked or dragged.

Redraw: An automatic redraw algorithm means that components redraw themselves automatically when they have to. Furthermore, the redraw algorithm is efficient: it only redraws components whose extents intersect the damaged region. The stroke or pixel model would have to do this test by hand. In practice, most stroked components don't bother, simply redrawing everything whenever some part of the view needs to be redrawn.

Drawing order: It's easy for a parent to draw before (underneath) or after (on top of) all of its children. But it's not easy to interleave parent drawing with child drawing. So if you're using a hybrid model, with some parts of your view represented as components and others as strokes, then the components and strokes generally fall in two separate layers, and you can't have any complicated z-ordering relationships between strokes and components.

Heavyweight objects: Every component must be an object (and even an object with no fields costs about 20 bytes in Java). As we've seen, the view hierarchy is overloaded not just with drawing functions but also with event dispatch, automatic redraw, and automatic layout, so that further bulks up the class. The flyweight pattern used by InterView's Glyphs can reduce this cost somewhat. But views derived from large amounts of data – say, a 100,000-node graph – generally can't use a component model.

Device dependence: The stroke model is largely device independent. In fact, it's useful not just for displaying to screens, but also to printers, which have dramatically different resolution. The pixel model, on the other hand, is extremely device dependent. A directly-mapped pixel image will not look the same on a screen with a different resolution.

Drawing is a top-down process: starting from the root of the component tree, each component draws itself, and then draws each of its children recursively. The process is optimised by passing a **clipping region** to each component, indicating the area of the screen that needs to be drawn. Children that do not intersect the clipping region are simply skipped, not drawn. In the example above, nodes B and C would not need to be drawn.

When a component partially intersects the clipping region, it must be drawn – but any strokes or pixels it draws when the clipping region is in effect will be masked against the clip region, so that only pixels falling inside the region actually make it onto the screen.

For the root component, the clipping region might be the entire screen. As drawing descends the component tree, however, the clipping region is intersected with each component's bounding box. So the clipping region for a component deep in the tree is the intersection of the bounding boxes of its ancestors.

For high performance, the clipping region is normally rectangular, using component **bounding boxes** rather than the components' actual shape. But it doesn't have to be that way. A clipping region can be an arbitrary shape on the screen. This can be very useful for visual effects: e.g., setting a string of text as your clipping region, and then painting an image through it like a stencil. Postscript was the first stroke model to allow this kind of nonrectangular clip region. Now many graphics toolkits support nonrectangular clip regions. For example, on Microsoft Windows and X Windows, you can create nonrectangular windows, which clip their children into a nonrectangular region.

When a component needs to change its appearance, it doesn't repaint itself directly. It *can't*, because the drawing process has to occur at top-down through the component hierarchy: the component's ancestors and older siblings need to have a chance to paint themselves underneath it. (So, in Java, even though a component can call its paint method directly, you shouldn't do it!)

Instead, the component asks the graphics system to repaint it at some time in the future. This request includes a **damaged region**, which is the part of the screen that needs to be repainted. Often, this is just the entire bounding box of the component; but complex components might figure out which part of the screen corresponds to the part of the model that changed, so that only that part is damaged.

The repaint request is then **queued** for later. Multiple pending repaint requests from different components are consolidated into a single damaged region, which is often represented just as a rectangle – the bounding box of all the damaged regions requested by individual components. That means that undamaged screen area is being considered damaged, but there is a trade-off between the complexity of the damaged region representation and the cost of repainting.

Eventually, after the system has handled all the input events (mouse and keyboard) waiting on the queue -the repaint request is finally satisfied, by setting the clipping region to the damaged region and redrawing the component tree from the root.

There is an unfortunate side-effect of the automatic damage/redraw algorithm. If we draw a component tree directly to the screen, then moving a component can make the screen appear to flash – objects flickering while they move, and nearby objects flickering as well.

When an object moves, it needs to be erased from its original position and drawn in its new position. The erasure is done by redrawing all the objects in the view hierarchy that intersect this damaged region. If the drawing is done directly on the screen, this means that all the objects in the damaged region temporarily *disappear*, before being redrawn. Depending on how screen refreshes are timed with respect to the drawing, and how long it takes to draw a complicated object or multiple layers of the hierarchy, these partial redraws may be briefly visible on the monitor, causing a perceptible flicker.

Double-buffering solves this flickering problem. An identical copy of the screen contents is kept in a memory buffer. (In practice, this may be only the part of the screen belonging to some sub-tree of the view hierarchy that cares about double-buffering.) This memory buffer is used as the drawing surface for the automatic damage/redraw algorithm. After drawing is complete, the damaged region is just copied to screen as a block of pixels. Double-buffering reduces flickering for two reasons: first, because the pixel copy is generally faster than redrawing the view hierarchy, so there's less chance that a screen refresh will catch it half-done; and second, because unmoving objects that happen to be caught, as innocent victims, in the damaged region are never erased from the screen, only from the memory buffer.

It's a waste for every individual view to double-buffer itself. If any of your ancestors is double-buffered, then you'll derive the benefit of it. So double-buffering is usually applied to top-level windows.

Why is it called double-buffering? It is so called because it used to be implemented by two interchangeable buffers in video memory. While one buffer was showing, you'd draw the next frame of animation into the other buffer. Then you'd just tell the video hardware to switch which buffer it was showing, a very fast operation that required no copying and was done during the CRT's vertical refresh interval so it produced no flicker at all.

Every stroke model has some notion of a **drawing surface**. The screen is only one place where drawing might go. Another common drawing surface is a memory buffer, which is an array of pixels just like the screen. Unlike the screen, however, a memory buffer can have arbitrary dimensions. The ability to draw to a memory buffer is essential for double-buffering. Another target is a printer driver, which forwards the drawing instructions on to a printer. Although most printers have a pixel model internally (when the ink actually hits the paper), the driver often uses a stroke model to communicate with the printer, for compact transmission. Postscript, for example, is a stroke model.

Most stroke models also include some kind of a **graphics context**, an object that bundles up drawing parameters like colour, line properties (width, end cap, join style), fill properties (pattern), and font. The stroke model may also provide a current **coordinate system**, which can be translated, scaled, and rotated around the drawing surface. We've already discussed the **clipping region**, which acts like a stencil for the drawing. Finally, a stroke model must provide a set of **drawing primitives**, function calls that actually produce graphical output.

Many systems combine all these responsibilities into a single object. Java's Graphics object is a good example of this approach. In other toolkits, the drawing surface and graphics context are independent objects that are passed along with drawing calls.

When state like graphics context, coordinate system, and clipping region are embedded in the drawing surface, the surface must provide some way to save and restore the context. A key reason for this is so that parent views can pass the drawing surface down to a child's draw method without fear that the child will change the graphics context. In Java, for example, the context can be saved by Graphics.create(), which makes a copy of the Graphics object. Notice that this only duplicates the graphics context; it doesn't duplicate the drawing surface, which is still the same.

3.2.2 Hints for Dubugging Output

Wrong place: what's the origin of the coordinate system? What's the scale? Where is the component located in its parent?
Wrong size: if a component has 0 width and 0 height, it will be completely invisible no matter what it tries to draw– everything will be clipped. 0 width and 0 height is the default for all components in Swing – you have to use automatic layout or manual setting to make it a more reasonable size. Check whether the components (and its ancestors) have nonzero sizes.
Wrong colour: is the drawing using the same colour as the background? Is it using 100% alpha?
Wrong z-order: is something else drawing on top?

4.0 Conclusion

In this unit you have been introduced to **input** and **output models** in detail. Different kinds of both input and output events have also be examined in greater detail.

5.0 Summary

In this unit, you have learnt the following:

- input models are GUI programs that are designed to handle input asynchronously, receiving it as events
- there are two major categories of input events: raw and translated. A raw event comes right from the device driver and is low-level while an event that is changed into a higher- level is called translated event
- output model is represented using components, strokes or pixels
- issues in choosing output model includes layout, input redraw e.t.c.

6.0 Self-Assessment Exercise

1. Describe the input and output model.
2. Highlight various issues considered in choosing an output model.

7.0 References/Further Reading

Boodhoo, Jean-Paul (2006). "Design Patterns: Model View Presenter". http://msdn.microsoft.com/en-us/magazine/cc188690.aspx. Retrieved on 2009-07-07.
World Wide Web Consortium (2008). "The Forms Working Group". http://www.w3.org/MarkUp/Forms/. Retrieved on 2009-07-07.

Unit 4 Model View-Controller (Mvc)

1.0 Introduction

The concept of Model-View-Controller (MVC) is introduced in this unit. We will also discuss the history of MVC, pattern description, MVC implementation framework. Techniques for implementing MVC as GUI frameworks are also explained.

2.0 Objectives

At the end of this unit, you should be able to:

- explain the term model- view-controller (MVC)
- discuss the history of MVC
- explain pattern description and MVC implementation framework
- describe how to implement MVC as GUI frameworks.

3.0 Main Content

3.1 Model -View-Controller

Model–view–controller (MVC) is an architectural pattern used in software engineering. The pattern isolates business logic from input and presentation, permitting independent development, testing and maintenance of each.

An MVC application is a collection of model/view/controller triplets (a central dispatcher is often used to delegate controller actions to a view-specific controller). Each model is associated with one or more views (projections) suitable for presentation (not necessarily visual presentation). When a model changes its state, it notifies its associated views so that they can refresh. The controller is responsible for initiating change requests and providing any necessary data inputs to the model.

MVC is frequently and needlessly convoluted by typing it directly to a graphical user interface. That a controller is often driven indirectly from a GUI is incidental. Likewise, rendering views graphically is an application of MVC, not part of the pattern definition. A business-to-business interface can leverage MVC architecture equally well.

3.2 Pattern Description

Model–view–controller is both an architectural pattern and a design pattern, depending on where it is used.

3.2.1 As an Architectural Pattern

It is common to split an application into separate layers that can be analysed, and sometimes implemented, separately.

MVC is often seen in web applications, where the view is the actual HTML or XHTML page, and the controller is the code that gathers dynamic data and generates the content within the HTML or XHTML. Finally, the model is represented by the actual content, which is often stored in a database or in XML nodes, and the business rules that transform that content based on user actions.

Though MVC comes in different flavours, control flow is generally as follows:

- The user interacts with the user interface in some way (for example, presses a mouse button).
- The controller handles the input event from the user interface, often via a registered handler or call-back.
- The controller notifies the model of the user action, possibly resulting in a change in the model's state. (For example, the controller updates the user's shopping cart.)
- A view uses the model indirectly to generate an appropriate user interface (for example, the view lists the shopping cart's contents). The view gets its own data from the model. The model and controller have no direct knowledge of the view.
- The user interface waits for further user interactions, which restarts the cycle.

Some implementations such as the W3C XForms also use the concept of a dependency graph to automate the updating of views when data in the model changes.
By decoupling models and views, MVC helps to reduce the complexity in architectural design and to increase flexibility and reuse of code.

3.2.2. As a Design Pattern

MVC encompasses more of the architecture of an application than is typical for a design pattern. When considered as a design pattern, MVC is semantically similar to the observer pattern.

Model
Is the domain-specific representation of the data on which the application operates. Domain logic adds meaning to raw data (for example, calculating whether today is the user's birthday, or the totals, taxes, and shipping charges for shopping cart items).

Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the data access layer because it is understood to be underneath or encapsulated by the model.

View
Converts the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes.

Controller
Processes and responds to events (typically user actions) and may indirectly invoke changes on the model.

3.3 MVC Implementation Framework

3.3.1 GUI Framework

Java: Java Swing
Java Swing is different from the other frameworks in that it supports two MVC patterns:

Model
Frame level model—like other frameworks, the design of the real model is usually left to the developer.
Control level model—Swing also supports models on the level of controls (elements of the graphical user interface). Unlike other frameworks, Swing exposes the internal storage of each control as a model.

View
The view is represented by a class that inherits from Component.

Controller
Java Swing doesn't use a single controller. Because its event model is based on interfaces, it is common to create an anonymous action class for each event. In fact, the real controller is in a separate thread, the event dispatching thread. It catches and propagates the events to the view and model.

Combined Frameworks
Java: Java Platform, Enterprise Edition (Java EE)
Simple version using only Java Servlets and JavaServer Pages from Java EE:

Model
The model is a collection of Java classes that forms a software application intended to store, and optionally moves, data. There is a single front end class that can communicate with any user interface (for example: a console, a graphical user interface, or a web application).

View
The view is represented by JavaServer Page, with data being transported to the page in the HttpServletRequest or HttpSession.

Controller
The controller servlet communicates with the front end of the model and loads the HttpServletRequest or HttpSession with appropriate data, before forwarding the HttpServletRequest and Response to the JSP using a RequestDispatcher.
The servlet is a Java class, and it communicates and interacts with the model but does not need to generate HTML or XHTML output; the JSPs do not have to communicate with the model because the servlet provides them with the information—they can concentrate on creating output.
Unlike the other frameworks, Java EE defines a pattern for model objects.

Model
The model is commonly represented by entity beans, although the model can be created by a servlet using a business object framework such as Spring.

View
The view in a Java EE application may be represented by a JavaServer Page, which may be currently implemented using JavaServer Faces Technology (JSF). Alternatively, the code to generate the view may be part of a servlet.

Controller
The controller in a Java EE application may be represented by a servlet, which may be currently implemented using JavaServer Faces (JSF).

XForms
XForms is an XML format for the specification of a data processing model for XML data and user interface(s) for the XML data, such as web forms.

Model
XForms stores the Model as XML elements in the browser. They are usually placed in the non-visible <head> elements of a web page.
View
The views are XForms controls for screen elements and can be placed directly in the visible section of web page. They are usually placed in the <body> elements of a web page.
The model and views are bound together using reference or binding statements. These binding statements are used by the XForms dependency graph to ensure that the correct views are updated when data in the model changes. This means that forms developers do not need to be able to understand either the push or pull models of event processing.

Controller
All mouse events are processed by XForms controls and XML events are dispatched.

3.5 Implementations of MVC as GUI Frameworks

Smalltalk's MVC implementation inspired many other GUI frameworks, such as the following:

- Cocoa framework and its GUI part AppKit, as a direct descendant of OpenStep, encourage the use of MVC. Interface Builder constructs views, and connects them to controllers via outlets and actions.
- GNUstep, also based on OpenStep, encourages MVC as well.
- GTK+.
- JFace.
- MFC (called Document/View architecture here).
- Microsoft Composite UI Application Block, part of the Microsoft Enterprise Library.
- Qt since Qt4 release.
- Java Swing.
- Adobe Flex.
- Wavemaker open source, browser-based development tool based on MVC.
- WPF uses a similar Model–view–viewmodel pattern.
- Visual FoxExpress is a Visual FoxPro MVC framework.

Self-Assessment Exercise

Search the internet and find out about the implementation strategies of some of the mentioned frameworks.

Some common programming languages and tools that support the implementation of MVC are:

- NET
- Actionscript
- ASP
- C++
- ColdFusion
- Flex
- Java
- Informix 4GL
- Lua
- Perl
- PHP
- Python
- Ruby
- Smalltalk
- XML

4.0 Conclusion

In this unit, you have been introduced to the model- view-controller (MVC). We also discussed in detail the history of MVC, pattern description, MVC implementation framework and how to implement MVC as GUI frameworks.

5.0 Summary

In this unit, you have learnt the following:

- The **Model–view–controller** (MVC) is an architectural pattern used in software engineering. The pattern isolates business logic from input and presentation, permitting independent development; testing and maintenance of each requirements of your users drive the development of your prototype.
- MVC was first described in 1979 by Trygve Reenskaug, then working on Smalltalk at Xerox PARC.
- **Model–view–controller** is both an architectural pattern and a design pattern, depending on where it is used.
- GUI framework includes Java Swing, Combined and XForms.
- Java Swing is different from the other frameworks in that it supports two MVC patterns: model and controller.
- Combined frameworks use only Java Servlets and JavaServer Pages from Java EE.
- XForms is an XML format for the specification of a data processing model for XML data and user interface(s) for the XML data, such as web forms.
- Smalltalk's MVC implementation inspired many other GUI frameworks, such as Cocoa framework, GNUstep, GTK+, e.t.c.

6.0 Self-Assessment Exercise

1. List and explain any two implementation of MVC as GUI framework
2. Write a short note on Xforms.

7.0 References/Further Reading

Boodhoo, Jean-Paul (2006). "Design Patterns: Model View Presenter". http://msdn.microsoft.com/en-us/magazine/cc188690.aspx. Retrieved on 2009-07-07.
World Wide Web Consortium (2008). "The Forms Working Group". http://www.w3.org/MarkUp/Forms/. Retrieved on 2009-07-07.
"JavascriptMVC Learning Center". http://www.javascriptmvc.com.

Unit 5 Layouts and Constraints

1.0 Introduction

Understanding layouts and constraints will be the main goal of this unit. Layout managers and hints for layouts are discussed. Various types of constraints are also explained.

2.0 Objectives

At the end of this unit, you should be able to:
- explain user interface
- state the significance of user interface and identify the various types of user interfaces
- discuss the history of user interfaces
- describe the modes and modalities of user interfaces.

3.0 Main Content

3.1 Layout

Layout is determining the positions and sizes of graphical objects. This can be done manually or automatically. Layout ranges in difficulty and constraints. Some layouts require simple one pass algorithm and some require dynamic programming and other advanced techniques. There is need to do layout automatically since there is need to change the states and conditions of windows, screens, fonts, widgets, etc. This must be planned for at designed level and must be implemented effectively to enhance the quality user interface.

3.2 Layout Managers

Layout managers are software tools for specifying and designing the appropriate layout for a job. They are also called geometry managers and are used to represent a bundle of constraint equations.

Layout Propagation Algorithm
- Layout (Container parent, Rectangle parentSize)
- For each child in parent,

Get child's size request
- Apply layout constraints to fit children into parentSize
- For each child,

Set child's size and position

3.3 Kinds of Layout Managers
- Packing
    - One dimensional
    - Java: BorderLayout, FlowLayout, BoxLayout
- Gridding

   - Two dimensional
   - Java: GridLayout, GridBagLayout, TableLayout
- General
   - Java: SpringLayout

3.4 Hints for Layout

- Use packing layouts when alignments are one dimensional (1D)
   - Borders for top-level
   - Nested boxes for internal
- Reserve gridding layouts for two dimensional (2D) alignment
   - Common when fields have captions
   - TableLayout is easier than GridBag

3.5 Constraints

Constraints are relationships expressed by the programmer and automatically maintained by the UI toolkit.

3.5.1 Uses

Layout: Constraints are used in layout to express the relationships between interface items. For example,   field.left = label.right + 10

Value propagation: They are used for an action to be invoked when a value is entered or reached. An example is

deleteAction.enabled = (selection != null)

Synchronisation of views to models: Constraints are used to express the relationships between models.

Interaction: They are also used to express interaction such as

rect.corner = mouse

3.6 Types of Constraints

One-Way Constraints

- Also called formulas, after spreadsheet
   - Y= f(x1,x2,x3, …)
   - Y depends on (points to) x1,x2,x3, …
- Algorithms
   - Data-driven
     Re-evaluate formulas when a value is changed
   - Demand-driven
     Re-evaluate formulas whenever a value is requested
   - Lazy
     When dependent value changes, invalidate all values that depend on it.
     When invalid value is requested, recalculate it

Variants

- Multi-output formulas
    - (y1,y2, …) = f(x1,x2,x3, …)
- Cyclic dependencies
    - Detect cycles and break them
- Constraint hierarchies
    - Some constraints stronger than others
- Side effects
    - If f has side effects, when do they happen?
        - Lazy evaluation makes side effects unpredictable
    - Amulet: eager evaluation

4.0 Conclusion

Detailed description of layouts and constraints were discussed. Layout managers and hints for layouts were also described.

5.0 Summary

In this unit, you have learnt the following:

- layout is determining the positions and sizes of graphical objects. This can be done manually or automatically
- layout managers are software tools for specifying and designing the appropriate layout for a job
- layout managers include packing, gridding and general kinds
- hints for layout include using packing layouts when alignments are 1D and reserving gridding layouts for 2D alignment
- constraints are relationships expressed by the programmer and automatically maintained by the UI toolkit
- Constraints are of two types: One-way constraints and variants.

6.0 Self-Assessment Exercise

1. Explain layouts and constraints.
2. Describe any two layout manager.

7.0 References/Further Reading

Ambler, S.W. (2001). *The Object Primer* (2nd ed.). *The Application Developer's Guide to Object Orientation*. New York: Cambridge University Press.

Gilroy, CA: CMP Books. http://www.ambysoft.com/inceptionPhase.html