**REPUBLIC OF RWANDA**

**KIGALI INDEPENDENT UNIVERSITY ULK**

**P.O Box 2280, KIGALI**

Website: http://www.ulk.ac.rw

E-mail:ulk@rwandatel.com

# MODULE OF HUMAN COMPUTER INTERACTION

## BOTH UNITS

**By:**

**Dr. NDAYAMBAJE PIUS**

**E-mail: ndayambaje.pius@yahoo.com**

Tel. 0788302943

**Mr. NIYIKIZA Gaston**

Tel. 0788812265

1. **Module Code:** CSC 410 **Faculty of Science and Technology**
2. **Module Title: HUMAN COMPUTER INTERACTION**
3. **Year: 4 Credits: 10**
4. **Administering Faculty: Science and technology**
5. **Pre-requisite or co-requisite modules, excluded combinations**
6. **Allocation of study and teaching hours :**

| **Total student hours : 100** | Student Hours | Lecture Hours |
|---|---|---|
| **Lectures** | 40 | 60 |
| **Seminars/workshops** | | |
| **Practical classes/laboratory** | 15 | 15 |
| **Structured exercises** | 10 | 10 |
| **Set reading etc.** | 10 | |
| **Self-directed study** | 10 | |
| **Assignments – preparation and writing** | 5 | 5 |
| **Examination – revision and attendance** | 10 | 10 |
| **Other: ( Invigilation & Marking )** | | |

| | | |
|---|---|---|
| **TOTAL** | 100 | **100** |

**Objectives**

This course provides an introduction to and overview of the field of human-computer interaction (HCI). HCI is an interdisciplinary field that integrates theories and methodologies from computer science, cognitive psychology, design, and many other areas.

The course focuses on three key areas:

- Design: How to design good user interfaces, starting with human capabilities (including the human information processor model, perception, motor skills, color, attention, and errors) and using those capabilities to drive design techniques: task analysis, user-centered design, iterative design, usability guidelines, interaction styles, and graphic design principles.

- Implementation: Techniques for building user interfaces, including low-fidelity prototypes, and other prototyping tools; input models, output models, model-view-controller, layout, constraints, and toolkits.

- Evaluation: Techniques for evaluating and measuring interface usability, including heuristic evaluation, predictive evaluation, and user testing.

Students will design, implement and evaluate a user interface. The course also involves an introduction to software architectures used in modern graphical user interfaces, including the implementation of a few simple interfaces using the Java Swing toolkit and NetBeans GUI Builder.

Upon successful completion of this course, students should be able to:

- Design, implement and evaluate effective and usable graphical computer interfaces.
- Describe and apply core theories, models and methodologies from the field of HCI.
- Implement simple graphical user interfaces using the Java Swing toolkit and NetBeans IDE.

## Table of Contents

## I.Basic concepts

### Human–computer interaction

**Human–computer interaction** (**HCI**) is the study of interaction between people (users) and computers. Interaction between users and computers occurs at the user interface (or simply *interface*), which includes both software and hardware; for example, characters or objects displayed by software on a personal computer's monitor, input received from users via hardware peripherals such as keyboards and mice, and other user interactions with large-scale computerized systems such as aircraft and power plants. The Association for Computing Machinery defines human-computer interaction as "a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them. An important facet of HCI is the securing of user satisfaction.

Because human-computer interaction studies a human and a machine in conjunction, it draws from supporting knowledge on both the machine and the human side. On the machine side, techniques in computer graphics, operating systems, programming languages, and development environments are relevant. On the human side, communication theory, graphic and industrial design disciplines, linguistics, social sciences, cognitive psychology, and human factors are relevant. Engineering and design methods are also relevant. Due to the multidisciplinary nature of HCI, people with different backgrounds contribute to its success. Attention to human-machine interaction is important, because poorly designed human-machine interfaces can lead to many unexpected problems.

### Goals

A basic goal of HCI is to improve the interactions between users and computers by making computers more usable and receptive to the user's needs. Specifically, HCI is concerned with:

- methodologies and processes for designing interfaces (i.e., given a task and a class of users, design the best possible interface within given constraints, optimizing for a desired property such as learning ability or efficiency of use)
- methods for implementing interfaces (e.g. software toolkits and libraries; efficient algorithms)
- techniques for evaluating and comparing interfaces
- developing new interfaces and interaction techniques
- developing descriptive and predictive models and theories of interaction

A long term goal of HCI is to design systems that minimize the barrier between the human's cognitive model of what they want to accomplish and the computer's understanding of the user's task.

Professional practitioners in HCI are usually designers concerned with the practical application of design methodologies to real-world problems. Their work often revolves around designing graphical user interfaces and web interfaces.

Researchers in HCI are interested in developing new design methodologies, experimenting with new hardware devices, prototyping new software systems, exploring new paradigms for interaction, and developing models and theories of interaction.

## Human–computer interface

The human–computer interface can be described as the point of communication between the human user and the computer. The flow of information between the human and computer is defined as the loop of interaction. The loop of interaction has several aspects to it including:

- **Task Environment**: The conditions and goals set upon the user.
- **Machine Environment**: The environment that the computer is connected to, i.e a laptop in a college student's dorm room.
- **Areas of the Interface**: Non-overlapping areas involve processes of the human and computer not pertaining to their interaction. Meanwhile, the overlapping areas only concern themselves with the processes pertaining to their interaction.
- **Input Flow**: The flow of information that begins in the task environment, when the user has some task that requires using their computer.
- **Output**: The flow of information that originates in the machine environment.
- **Feedback**: Loops through the interface that evaluate, moderate, and confirm processes as they pass from the human through the interface to the computer and back.

## Components of a GUI

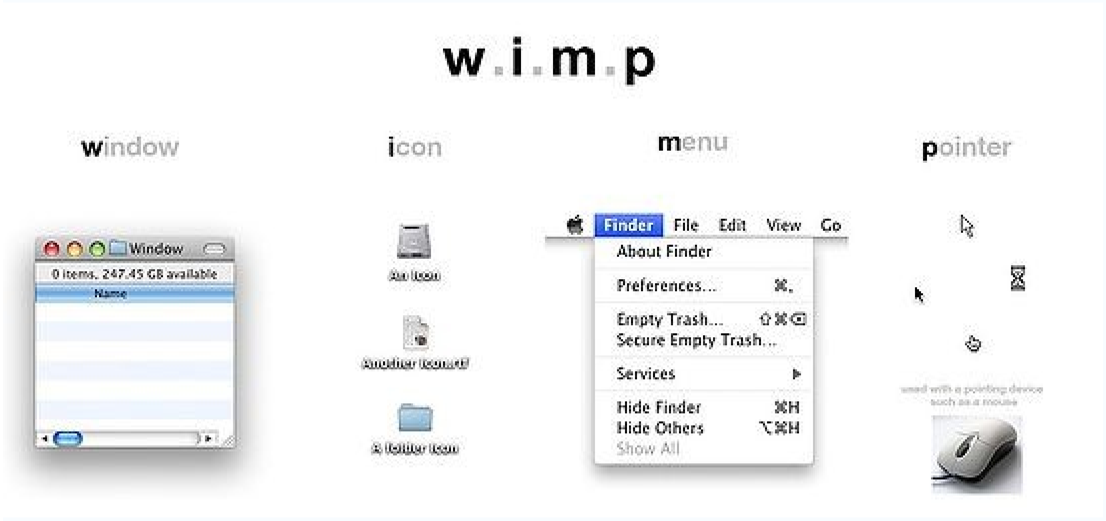
A GUI uses a combination of technologies and devices to provide a platform the user can interact with, for the tasks of gathering and producing information.

A series of elements conforming a visual language have evolved to represent information stored in computers. This makes it easier for people with few computer skills to work with and use computer software. The most common combination of such elements in GUIs is the WIMP ("window, icon, menu, pointing device") paradigm, especially in personal computers.

In human–computer interaction, **WIMP** stands for "window, icon, menu, pointing device", denoting a style of interaction using these elements. It was coined by Merzouga Wilberts in 1980. Although its usage has fallen out of favor, it is often used as an approximate synonym of "GUI". WIMP interaction was developed at Xerox PARC (see Xerox Alto, developed in 1973) and popularized with Apple's introduction of the Macintosh in 1984, where the concepts of the "menu bar" and extended window management were added.

This style of interaction uses a physical input device to control the position of a cursor and presents information organized in windows and represented with icons. Available commands are compiled together in menus and actioned through the pointing device. This is intended to reduce the cognitive load to remember the possibilities available, reducing learning times.

Other intended benefits of this style include its ease of use for non-technical people, both novice and power users. Also know-how can be ported from one application to the next, given the high consistency between interfaces.



The WIMP style of interaction uses a physical input device to control the position of a cursor and presents information organized in windows and represented with icons. Available commands are compiled together in menus, and actions are performed making gestures with the pointing device.

In personal computers all these elements are modeled through a desktop metaphor, to produce a simulation called a desktop environment in which the display represents a desktop, upon which documents and folders of documents can be placed.

Interface metaphors are designed to be similar to physical entities but also have their own properties (e.g., desktop metaphor and web portals).

They can be based on an activity, an object, or a combination of both. They work with users' familiar knowledge to help them understand ‘the unfamiliar.’ They conjure up the essence of the unfamiliar activity, but they put it in terms users are better able to understand

In the mid-twentieth century, computers were extremely rare and used only by specialists. They were equipped with complicated interfaces comprehensible only to these select few. In 1968, Douglas Englebart gave a demonstration which astonished executives at Xerox. They began work on what would eventually become the Xerox Alto. In 1973 Xerox completed work on the first personal computer, the Xerox Alto, which had a sophisticated graphical user interface (GUI) involving windows, icons, a mouse and a pointer. (WIMP)

Unfortunately, the Xerox Alto, and its successor the Xerox Star were far too expensive for the average consumer, and suffered from poor marketing. In 1984 Apple Computer launched the Apple Macintosh, which was the first affordable and commercially successful personal computer to include a graphical user interface. The Macintosh was the second Apple Computer to ship with a graphical user interface, with the Apple Lisa being the first.

In 1985, Microsoft released Microsoft Windows which bore a striking resemblance to both Macintosh, and to the Alto's interface. Windows eventually overtook Apple in the PC market to become the predominant GUI-based operating system.

Interface metaphors have come a long way since they were first used. Recently, it has been predicted that the latest metaphors will come from life sciences. Others may come from health care or other industries, as they are going to become information-dense environments. An interface for a next-generation technology might come from the gaming world, where quick visualization metaphors will be.

A downside to changing interface metaphors on a constant basis is that the owners of software with many users are reluctant to make big changes, and their interfaces tend to evolve incrementally and to keep their familiar look and familiarity.

Software designers attempt to make computer applications easier to use for both novice and expert users by creating concrete metaphors that resemble the users' real-world experiences. Continual technological improvement has made metaphors depict these real-world experiences more realistically to ultimately enhance interface performance.

Beginning users, however, could use a sort of help box, because the metaphor is not always going to be clear enough for them to understand, no matter how much effort its programmers devote to making it resemble something the users understand.

Experts, on the other hand, understand what is going on with the technical aspects of an interface metaphor. They know what they want to do and they know how to do it—-and so they design shortcuts to facilitate achieving their goals.

## Post-WIMP interfaces

Smaller mobile devices such as PDAs and smartphones typically use the WIMP elements with different unifying metaphors, due to constraints in space and available input devices. Applications for which WIMP is not well suited may use newer interaction techniques, collectively named as post-WIMP user interfaces.

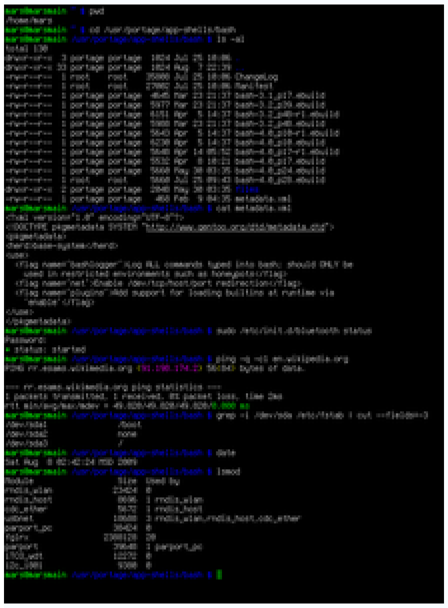
## User interface and interaction design

Designing the visual composition and temporal behavior of GUI is an important part of software application programming. Its goal is to enhance the efficiency and ease of use for the underlying logical design of a stored program, a design discipline known as usability. Techniques of user-centered design are used to ensure that the visual language introduced in the design is well tailored to the tasks it must perform.

Typically, the user interacts with information by manipulating visual widgets that allow for interactions appropriate to the kind of data they hold. The widgets of a well-designed interface are selected to support the actions necessary to achieve the goals of the user. A Model-view-controller allows for a flexible structure in which the interface is independent from and indirectly linked to application functionality, so the GUI can be easily customized. The visible graphical interface features of an application are sometimes referred to as "chrome".  Larger widgets, such as windows, usually provide a frame or container for the main presentation content such as a web page, email message or drawing. Smaller ones usually act as a user-input tool.

## Comparison GUI to other interfaces

### Command-line interfaces



```
mark@marsmain ~ $ pwd
/home/mark
mark@marsmain ~ $ cd /usr/portage/app-shells/bash
mark@marsmain /usr/portage/app-shells/bash $ ls -al
total 130
drwxr-xr-x  3 portage portage  1024 Jul 25 18:06 .
drwxr-xr-x 33 portage portage  1024 Aug  7 22:39 ..
-rw-r--r--  1 root    root    35000 Jul 25 18:06 ChangeLog
-rw-r--r--  1 root    root    27002 Jul 25 18:06 Manifest
-rw-r--r--  1 portage portage  4645 Mar 23 21:37 bash-3.1_p17.ebuild
-rw-r--r--  1 portage portage  5977 Mar 23 21:37 bash-3.2_p39.ebuild
-rw-r--r--  1 portage portage  6151 Apr  5 14:37 bash-3.2_p48-r1.ebuild
-rw-r--r--  1 portage portage  5988 Mar 23 21:37 bash-3.2_p48.ebuild
-rw-r--r--  1 portage portage  5643 Apr  5 14:37 bash-4.0_p10-r1.ebuild
-rw-r--r--  1 portage portage  6230 Apr  5 14:37 bash-4.0_p10.ebuild
-rw-r--r--  1 portage portage  5640 Apr 14 05:52 bash-4.0_p17-r1.ebuild
-rw-r--r--  1 portage portage  5532 Apr  8 18:21 bash-4.0_p17.ebuild
-rw-r--r--  1 portage portage  5668 May 30 03:35 bash-4.0_p24.ebuild
-rw-r--r--  1 root    root     5668 Jul 25 09:43 bash-4.0_p28.ebuild
drwxr-xr-x  2 portage portage  2048 May 30 03:35 files
-rw-r--r--  1 portage portage   468 Feb  9 04:35 metadata.xml
mark@marsmain /usr/portage/app-shells/bash $ cat metadata.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pkgmetadata SYSTEM "http://www.gentoo.org/dtd/metadata.dtd">
<pkgmetadata>
<herd>base-system</herd>
<use>
  <flag name='bashlogger'>Log ALL commands typed into bash; should ONLY be
    used in restricted environments such as honeypots</flag>
  <flag name='net'>Enable /dev/tcp/host/port redirection</flag>
  <flag name='plugins'>Add support for loading builtins at runtime via
    'enable'</flag>
</use>
</pkgmetadata>
mark@marsmain /usr/portage/app-shells/bash $ sudo /etc/init.d/bluetooth status
Password:
 * status: started
mark@marsmain /usr/portage/app-shells/bash $ ping -q -c1 en.wikipedia.org
PING rr.esams.wikimedia.org (91.198.174.2) 56(84) bytes of data.

--- rr.esams.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 49.820/49.820/49.820/0.000 ms
mark@marsmain /usr/portage/app-shells/bash $ grep -i /dev/sda /etc/fstab | cut --fields=-3
/dev/sda1               /boot
/dev/sda2               none
/dev/sda3               /
mark@marsmain /usr/portage/app-shells/bash $ date
Sat Aug  8 02:42:24 MSD 2009
mark@marsmain /usr/portage/app-shells/bash $ lsmod
Module                  Size  Used by
rndis_wlan             23424  0
rndis_host              8696  1 rndis_wlan
cdc_ether               5672  1 rndis_host
usbnet                 18688  3 rndis_wlan,rndis_host,cdc_ether
parport_pc             38424  0
fglrx                2388128  20
parport                39648  1 parport_pc
i2c_i801                9380  0
mark@marsmain /usr/portage/app-shells/bash $
```

Modern CLI

GUIs were introduced in reaction to the steep learning curve of command-line interfaces (CLI), which require commands to be typed on the keyboard. Since the commands available in command line interfaces can be numerous, complicated operations can be completed using a short sequence of words and symbols. This allows for greater efficiency and productivity once many commands are learnt, but reaching this level takes some time because the command words are not easily discoverable and not mnemonic. WIMPs ("window, icon, menu, pointing device"), on the other hand, present the user with numerous widgets that represent and can trigger some of the system's available commands.

Most modern operating systems provide both a GUI and some level of a CLI, although the GUIs usually receive more attention. The GUI is usually WIMP-based.

Applications may also provide both interfaces, and when they do the GUI is usually a WIMP wrapper around the command-line version.

## Three-dimensional user interfaces

For typical computer displays, *three-dimensional* is a misnomer—their displays are two-dimensional. Semantically, however, most graphical user interfaces use three dimensions - in addition to height and width, they offer a third dimension of layering or stacking screen elements over one another. This may be represented visually on screen through an illusionary transparent effect, which offers the advantage that information in background windows may still be read, if not interacted with. Or the environment may simply hide the background information, possibly making the distinction apparent by drawing a drop shadow effect over it.

Some environments use the techniques of 3D graphics to project virtual three dimensional user interface objects onto the screen. As the processing power of computer graphics hardware increases, this becomes less of an obstacle to a smooth user experience.

## Structural elements of GUI

User interfaces use visual conventions to represent the generic information shown. Some conventions are used to build the structure of the static elements on which the user can interact, and define the appearance of the interface.

### Window

A window is an area on the screen that displays information, with its contents being displayed independently from the rest of the screen. An example of a window is what appears on the screen when the "My Documents" icon is clicked in the Windows Operating System. It is easy for a user to manipulate a window: it can be opened and closed by clicking on an icon or application, and it can be moved to any area by dragging it (that is, by clicking in a certain area of the window – usually the title bar along the tops – and keeping the pointing device's button pressed, then moving the pointing device). A window can be placed in front or behind another window, its size can be adjusted, and scrollbars can be used to navigate the sections within it. Multiple windows can also be open at one time, in which case each window can display a different application or file – this is very useful when working in a multitasking environment. The system memory is the only limitation to the amount of windows that can be open at once. There are also many types of specialized windows.

- A **Container Window** a window that is opened while invoking the icon of a mass storage device, or directory or folder and which is presenting an ordered list of other icons that could be again some other directories, or data files or maybe even executable programs. All modern container windows could present their content on screen either acting as browser windows or text windows. Their behaviour can automatically change according to the choices of the single users and their preferred approach to the graphical user interface.

- A browser window allows the user to move forward and backwards through a sequence of documents or web pages. Web browsers are an example of these types of windows.

- Text terminal windows are designed for embedding interaction with text user interfaces within the overall graphical interface. MS-DOS and UNIX consoles are examples of these types of windows.

- A **child window** opens automatically or as a result of a user activity in a parent window. Pop-up windows on the Internet can be child windows.

- A **message window**, or dialog box, is a type of child window. These are usually small and basic windows that are opened by a program to display information to the user and/or get information from the user. They usually have a button that must be pushed before the program can be resumed.

**Menus**

Menus allow the user to execute commands by selecting from a list of choices. Options are selected with a mouse or other pointing device within a GUI. A keyboard may also be used. Menus are convenient because they show what commands are available within the software. This limits the amount of documentation the user reads to understand the software.

- A menu bar is displayed horizontally across the top of the screen and/or along the tops of some or all windows. A pull-down menu is commonly associated with this menu type. When a user clicks on a menu option the pull-down menu will appear.

- A menu has a visible title within the menu bar. Its contents are only revealed when the user selects it with a pointer. The user is then able to select the items within the pull-down menu. When the user clicks elsewhere the content of the menu will disappear.

- A context menu is invisible until the user performs a specific mouse action, like pressing the right mouse button. When the software-specific mouse action occurs the menu will appear under the cursor.[

- Menu extras are individual items within or at the side of a menu.

### Icons

An icon is a small picture that represents objects such as a file, program, web page, or command. They are a quick way to execute commands, open documents, and run programs. Icons are also very useful when searching for an object in a browser list, because in many operating systems all documents using the same extension will have the same icon.

### Controls (or Widgets)

Interface element that a computer user interacts with, and is also known as a **control** or Widget.

Window

A paper-like rectangle that represents a "window" into a document, form, or design area.

Pointer (or mouse cursor)

The spot where the mouse "cursor" is currently referencing.

Text box

A box in which to enter text or numbers.

Button

An equivalent to a push-button as found on mechanical or electronic instruments.

Hyperlink

Text with some kind of indicator (usually underlining and/or color) that indicates that clicking it will take one to another screen or page.

Drop-down list

A list of items from which to select. The list normally only displays items when a special button or indicator is clicked.

Check box

A box which indicates an "on" or "off" state via a check mark (✓) or a cross (✗).

Radio button

A button, similar to a check-box, except that only one item in a group can be selected. Its name comes from the mechanical push-button group on a car radio receiver. Selecting a new item from the group's buttons also deselects the previously selected button.

Datagrid

A spreadsheet-like grid that allows numbers or text to be entered in rows and columns.

**Tabs**

A tab is typically a rectangular small box which usually contains a text label or graphical icon associated with a view pane. When activated the view pane, or window, displays widgets associated with that tab; groups of tabs allow the user to switch quickly between different widgets. This is used in the web browsers Firefox, Internet Explorer. With these browsers, you can have multiple web pages open at once in one window, and quickly navigate between them by clicking on the tabs associated with the pages. Tabs are usually placed in groups at the top of a window, but may also be grouped on the side or bottom of a window.

**Interaction elements**

Some common idioms for interaction have evolved in the visual language used in GUIs. Interaction elements are interface objects that represent the state of an ongoing operation or transformation, either as visual remainders of the user intent (such as the pointer), or as affordances showing places where the user may interact.

**Cursor**

A cursor is an indicator used to show the position on a computer monitor or other display device that will respond to input from a text input or pointing device.

***Pointer***

One of the most common components of a GUI on the personal computer is a pointer: a graphical image on a screen that indicates the location of a pointing device, and can be used to select and move objects or commands on the screen. A pointer commonly appears as an angled arrow, but it can vary within different programs or operating systems. Example of this can be found within text-processing applications, which uses an I-beam pointer that is shaped like a capital I, or in web browsers which often indicate that the pointer is over a hyperlink by turning the pointer in the shape of a gloved hand with outstretched index finger.

The use of a pointer is employed when the input method, or pointing device, is a device that can move fluidly across a screen and select or highlight objects on the screen. Pointer trails can be used to enhance its visibility during movement. In GUIs where the input method relies on hard keys, such as the five-way key on many mobile phones, there is no pointer employed, and instead the GUI relies on a clear focus state.

**Selection**

A selection is a list of items on which user operations will take place. The user typically adds items to the list manually, although the computer may create a selection automatically.

**Adjustment handle**

A handle is an indicator of a starting point for a drag and drop operation. Usually the pointer shape changes when placed on the handle, showing an icon that represents the supported drag operation.

## Interface Design, implementation, and Evaluation

### The User Interface Is Important

• User interface strongly affects
perception of software
–Usable software sells better
–Unusable web sites are abandoned
• Perception is sometimes superficial
–Users blame themselves for UI failings
–People who make buying decisions are not
always end-users

### User Interfaces Are Hard to Design

• You are not the user
–Most software engineering is about
communicating with other programmers
–UI is about communicating with users
• The user is always right
–Consistent problems are the system's fault
• …but the user is not always right
–Users aren't designers

### User Interfaces are Hard to Build

• User interface takes a lot of software
development effort
• UI accounts for ~50% of:
–Design time
– Implementation time
–Maintenance time
–Code size

Usability strongly affects how software is perceived, because the user interface is the means by which the software presents itself to the world. "Ease of use" ratings appear in magazine

reviews, affect recommendations, and influence buying decisions. Usable software sells. Conversely, unusable software doesn't sell. If a web site is so unusable that shoppers can't find what they want, or can't make it through the checkout process, then they will go somewhere else.
Unfortunately, a user's perception of software usability is often superficial. An attractive
user interface may seem "user friendly" even if it's not really usable. Part of that is because users often blame themselves for errors they make, even if the errors could have been prevented by better interface design. ("Oops, I missed the File menu again! How stupid of me.") So usability is a little different from other important attributes of software, like reliability, performance, or security. If the program is slow, or crashes, or gets hacked, we know who to blame. If it's unusable, but not fatally so, the usability problems may go unreported.
Users don't obey Moore's Law. Their time doesn't get cheaper with every new generation, like processors do. In fact, user time is probably getting *more* expensive every year. Interfaces that waste user time repeatedly over a lifetime of use impose a hidden cost that companies are less and less inclined to pay. For some applications, like customer call centers, saving a few seconds per call may translate into millions of dollars saved per year. For many software companies, a single customer support call can wipe out all the profit on that sale.
Bad user interface design can also cost lives. In 1988, the USS Vincennes guided missile cruiser shot down an Iranian airliner over the Persian Gulf with almost 300 people aboard. There were two failures in this incident. The radar operator interpreted the airliner as an F-14, descending as if to attack, rather than (in reality) a civilian plane that was climbing after takeoff. Both failures seemed to be caused by user interface. The IFF system was reporting the signal from an F14 on the ground at an airport hundreds of miles away, not the signal from the airliner; and the plane's altitude readout showed only its current altitude, not the direction of change in altitude, leaving to the operator the mental comparison and calculation to determine whether the altitude was going up or down. (Peter Neumann,
"Aegis, Vincennes, and the Iranian Airbus", Risks v8 n74, May 1989).

Unfortunately, user interfaces are not easy to design. You (the developer) are not a typical user. You know far more about your application than any user will. This is how usability is different from everything else you learn about software engineering. Specifications, assertions, and object models are all about communicating with other *programmers*, who are probably a lot like us. Usability is about communicating with other *users*, who are probably not like us.
The user is always right. Don't blame the user for what goes wrong. If users consistently make mistakes with some part of your interface, take it as a sign that your *interface* is wrong, not that the users are dumb.
Unfortunately, the user is not always right. Users aren't oracles. They don't always know what they want, or
what would help them. In a study conducted in the 1950s, people were asked whether they would prefer lighter telephone handsets, and on average, they said they were happy with the handsets they had (which at the time were made rather heavy for durability). Yet an actual test of telephone handsets, identical except for weight, revealed that people preferred the handsets that were about half the weight that was normal at the time. (Klemmer, *Ergonomics*, Ablex, 1989, pp 197-201).
Users aren't designers, either, and shouldn't be forced to fill that role. It's easy to say, "Yeah, the interface is bad, but users can customize it however they want it." There are two problems with this statement: (1) most users don't, and (2) user customizations may be even worse! One study

of command abbreviations found that users made twice as many errors with their *own* command abbreviations than with a carefully-designed set (Grudin & Barnard, “When does an abbreviation become a word?”, CHI ’85). So customization isn’t the silver bullet.
The user interface also consumes a significant portion of software development resources.
One survey of 74 software projects found that user interface code accounted for about half
of the time put towards design, implementation, and maintenance, and constituted about
half the code (Myers & Rosson, “Survey on user interface programming”, CHI ’92).
So UI is an important part of software design.

## Usability Defined

• Usability: how well users can use the system’s functionality
• Dimensions of usability
– Learnability: is it easy to learn?
–Efficiency: once learned, is it fast to use?
–Memorability: is it easy to remember what
you learned?
–Errors: are errors few and recoverable?
–Satisfaction: is it enjoyable to use?
The property we’re concerned with here, **usability**, is more precise than just how “good”
the system is”. A system can be good or bad in many ways. If important requirements are unsatisfied by the system, that’s probably a deficiency in functionality, not in usability. If the system is very expensive or crashes frequently, those problems certainly detract from the user’s experience, but we don’t need user testing to tell us that.
More narrowly defined, usability measures how well users can use the system’s
functionality. Usability has several dimensions: learnability, efficiency, memorability, error rate/severity, and subjective satisfaction.
Notice that we can **quantify** all these measures of usability. Just as we can say algorithm X is faster than algorithm Y on some workload, we can say that interface X is more learnable, or more efficient, or more memorable than interface Y for some set of tasks and some class of users.

### Usability Dimensions Vary In Importance

• Depends on the user
–Novice users need learnability
– Infrequent users need memorability
–Experts need efficiency
• But no user is uniformly novice or expert
–Domain experience
–Application experience
– Feature experience
The usability dimensions are not uniformly important for all classes of users, or for all
applications. That’s one reason why it’s important to understand your users, so that you
know what you should optimize for. A web site used only once by millions of people

has such a strong need for ease of learning, in fact zero learning, that it far outweighs other concerns. A stock trading program used on a daily basis by expert traders, for whom lost seconds translate to lost dollars, must put efficiency above all else.
But users can't be simply classified as novices or experts, either. For some applications (like stock trading), your users may be *domain* experts, deeply knowledgeable about the stock market, and yet still be novices at your particular application. Even users with long experience using an application may be novices or infrequent users when it comes to some of its features.

## Usability Is Only One Attribute of a System

• Software designers have a lot to worry about:
– Functionality **– Usability**
– Performance – Size
– Cost – Reliability
– Security – Standards
• Many design decisions involve tradeoffs among different attributes
• We'll take an extreme position in this course
Usability doesn't exist in isolation, of course, and it may not even be the most important property of some systems. Astronauts may care more about reliability of their navigation computer than its usability; military systems would rather be secure than easy to log into. Ideally these should be false dichotomies: we'd rather have systems that are reliable, secure, *and* usable. But in the real world, development resources are finite, and tradeoffs must be made. In this course, we'll take an extreme position: usability will be our primary goal.

## **Usability Engineering** Is a Process

Design
Evaluate Implement
So how do we **engineer** usability into our systems? By means of a process, with careful attention to detail. One goal of this course is to give you experience with this usability engineering process.

## Design

• Task analysis
– "Know the user"
• Design guidelines
–Avoid mistakes
–May be vague or contradictory

In broad terms, we apply a **user-centered design (UCD)** which is a design philosophy and a process in which the needs, wants, and limitations of end users of an interface or document are given extensive attention at each stage of the design process. User-centered design can be characterized as a multi-stage problem solving process that not only requires designers to analyze and foresee how users are likely to use an interface, but also to test the validity of their assumptions with regards to user behaviour in real world tests with actual users. Such testing is necessary as it is often very difficult for the designers of an interface to understand intuitively what a first-time user of their design experiences, and what each user's learning curve may look like.

The chief difference from other interface design philosophies is that user-centered design tries to optimize the user interface around how people can, want, or need to work, rather than forcing the users to change how they work to accommodate the software developers approach.

The first step of usability engineering is knowing who your users are and understanding their needs. Who are they? What do they already know? What is their environment like? What are their goals? What information do they need, what steps are involved in achieving those goals?

UCD answers questions about users and their tasks and goals, then use the findings to make decisions about development and design. UCD of a web site, for instance, seeks to answer the following questions:

- Who are the users of the document?
- What are the users' tasks and goals?
- What are the users' experience levels with the document, and documents like it?
- What functions do the users need from the document?
- What information might the users need, and in what form do they need it?
- How do users think the document should work?

These are the ingredients of a **task analysis**. Often, a task analysis needs to be performed in the field – interviewing real users and watching them do real tasks.

**Design guidelines** are an important part of usability engineering. Design guidelines help you get started, and help avoid the most mistakes.
But guidelines are heuristics, not hard-and-fast rules. An interface cannot be made usable merely by slavish adherence to design guidelines, because guidelines may be vague or contradictory. *The user should be in control*, says one guideline, but at the same time we have to *prevent errors*. Another dictates that *the user should always know what's happening*, but don't forget to *keep the user mental workload*
*within acceptable limits*. Design guidelines help us discuss design alternatives sensibly, but they don't give all the answers.

As examples of UCD viewpoints, the essential elements of UCD of a web site are considerations of visibility, accessibility, legibility and language.

**Visibility**

Visibility helps the user construct a mental model of the document. Models help the user predict the effect(s) of their actions while using the document. Important elements (such as those that aid navigation) should be emphatic. Users should be able to tell from a glance what they can and cannot do with the document.

**Accessibility**

Users should be able to find information quickly and easily throughout the document, regardless of its length. Users should be offered various ways to find information (such navigational elements, search functions, table of contents, clearly labeled sections, page numbers, color coding, etc). Navigational elements should be consistent with the genre of the document. 'Chunking' is a useful strategy that involves breaking information into small pieces that can be organized into some type meaningful order or hierarchy. The ability to skim the document allows users to find their piece of information by scanning rather than reading. Bold and italic words are often used.

**Legibility**

Text should be easy to read: Through analysis of the rhetorical situation, the designer should be able to determine a useful font style. Ornamental fonts and text in all capital letters are hard to read, but italics and bolding can be helpful when used correctly. Large or small body text is also hard to read. (Screen size of 10-12 pixel sans serif and 12-16 pixel serif is recommended.) High figure-ground contrast between text and background increases legibility. Dark text against a light background is most legible.

**Language**

Depending on the rhetorical situation, certain types of language are needed. Short sentences are helpful, as well as short, well-written texts used in explanations and similar bulk-text situations. Unless the situation calls for it, do not use jargon or technical terms. Many writers will choose to use active voice, verbs (instead of noun strings or nominals), and simple sentence structure.

Implement
• Prototyping
–Cheap, throw-away implementations
– Low-fidelity: paper
–Medium-fidelity: HTML, Visual Basic
• GUI implementation techniques
– Input/output models
– Toolkits
–UI builders
Since we can't predict usability in advance, we build **prototypes** – the cheaper, the better.
We want to throw them away. We'll see that paper is a great prototyping tool!

Eventually, however, the prototypes must give way to an actual, interactive computer system. A range of techniques for structuring graphical user interfaces have been developed. We'll look at how they work, how to use them, and how UI toolkits themselves are implemented.

### Evaluate

• Evaluation puts prototypes to the test
• Expert evaluation
–Heuristics and walkthroughs
• Predictive evaluation
– Testing against an engineering model
(simulated user)
• Empirical evaluation
–Watching users do it

Then we **evaluate** our prototypes – sometimes using heuristics, but more often against real users. Putting an implementation to the test is the only real way to measure usability.

### Iterative Design

The most important element of usability engineering is **iterative design**. That means we don't go around the design-implement-evaluate loop just once. We admit to ourselves that we aren't going to get it right on the first try, and plan for it. Using the results of evaluation, we redesign the interface, build new prototypes, and do more evaluation. Eventually, hopefully, the process produces a sufficiently usable interface.

Many of the techniques we'll learn in this course are optimizations for the iterative design process: design guidelines reduce the number of iterations by helping us make better designs; cheap prototypes and discount evaluation techniques reduce the cost of each iteration. But even more important than these techniques is the basic realization that in general, **you won't get it right the first time**. If you learn nothing else about user interfaces from this class, I hope you learn this.

# II. Interface design

## II.1. UI Software Architecture

• Model-view-controller
• View hierarchy
• Observer

Let's take a high-level look at the software architecture of GUI software, focusing on the **design patterns** that have proven most useful. Three of the most important patterns are the **model-view-controller** abstraction, which has evolved somewhat since its original formulation in the early 80's; the **view hierarchy**, which is a central feature in the architecture of every popular GUI toolkit; and the **observer** pattern, which is essential to decoupling the model from the view and controller.

### Model-View-Controller Pattern

• Separates frontend concerns from backend
concerns

• Separates input from output
• Permits multiple views on the same
application data
• Permits views/controllers to be reused for
other models
• Example: text box
– Model: mutable string
– View: rectangle with text drawn in it
– Controller: keystroke handler
The **model-view-controller** pattern, originally articulated in the Smalltalk-80 user interface, has strongly influenced the design of UI software ever since. In fact, MVC may have single-handedly
inspired the software design pattern movement. MVC's primary goal is separation of concerns. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is supposed to handle output.
In principle, this separation has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and controllers to be reused for other models, in other applications. The MVC pattern enables the creation of user interface **toolkits**, which are libraries of reusable interface objects.
In practice, the MVC pattern doesn't quite work out the way we'd like.
A simple example of the MVC pattern is a text box widget. Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed by the user and inserts them in the string.

**Model-View-Controller (MVC)** is a classic design pattern often used by applications that need the ability to maintain multiple views of the same data. The MVC pattern hinges on a clean separation of objects into one of three categories — **models** for maintaining data, **views** for displaying all or a portion of the data, and **controllers** for handling events that affect the model or view(s).

Because of this separation, multiple views and controllers can interface with the same model. Even new types of views and controllers that never existed before can interface with a model without forcing a change in the model design.

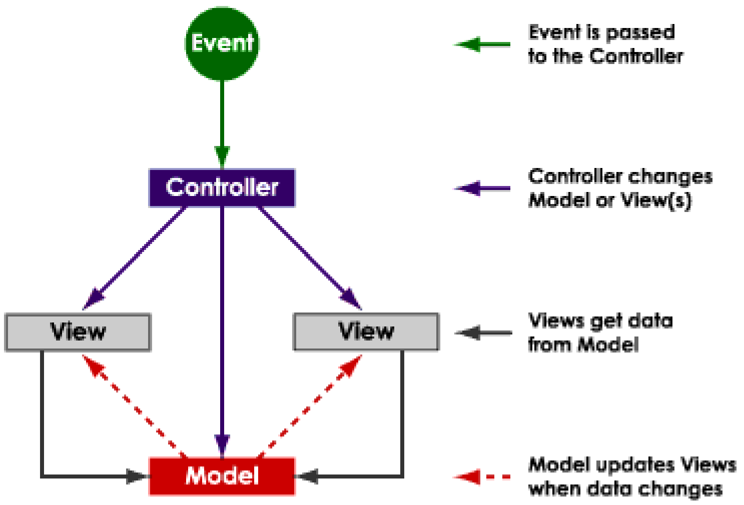
- **Model** - The model represents data and the rules that govern access to and updates of this data. In enterprise software, a model often serves as a software approximation of a real-world process.

- **View** - The view renders the contents of a model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed. This can be achieved by using a *push model*, in which the view registers itself with the model for change notifications, or a *pull model*, in which the view is responsible for calling the model when it needs to retrieve the most current data.

- **Controller** - The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in an enterprise web application, they appear as GET and POST HTTP requests. Depending on the context, a controller may also select a new view -- for example, a web page of results -- to present back to the user.

## How It Works

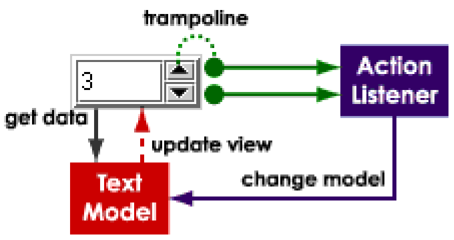
The MVC abstraction can be graphically represented as follows.


Event
Controller
View
View
Model
Event is passed to the Controller
Controller changes Model or View(s)
Views get data from Model
Model updates Views when data changes

Events typically cause a controller to change a model, or view, or both. Whenever a controller changes a model's data or properties, all dependent views are automatically updated. Similarly, whenever a controller changes a view, for example, by revealing areas that were previously hidden, the view gets data from the underlying model to refresh itself.

## A Concrete Example

We explain the MVC pattern with the help of a simple **spinner** component which consists of a text field and two arrow buttons that can be used to increment or decrement a numeric value shown in the text field. We currently do not have an element type that can directly represent a spinner component, but it easy is to synthesize a spinner using existing element types.



The spinner's data is held in a **model** that is *shared* with the text field. The text field provides a **view** of the spinner's current value. Each button in the spinner is an **event source**, that spawns an **action event** every time it is clicked. The buttons can be hooked up to trampolines that receive action events, and route them to an **action listener** that eventually handles that event. Recall that a trampoline is a predefined action listener that simply delegates action handling to another listener.

Depending on the source of the event, the ultimate action listener either increments or decrements the value held in the model — The action listener is an example of a **controller**.

The trampolines that initially receive the action events fired by the arrow buttons, are also controllers — However, instead of modifying the spinner's model directly, they delegate the task to a separate controller (action listener).

**Interaction Between MVC Components**

Once the model, view, and controller objects are instantiated, the following occurs:

1. The view registers as a listener on the model. Any changes to the underlying data of the model immediately result in a broadcast change notification, which the view receives. This is an example of the push model described earlier. Note that the model is not aware of the view or the controller -- it simply broadcasts change notifications to all interested listeners.

2. The controller is bound to the view. This typically means that any user actions that are performed on the view will invoke a registered listener method in the controller class.

3. The controller is given a reference to the underlying model.

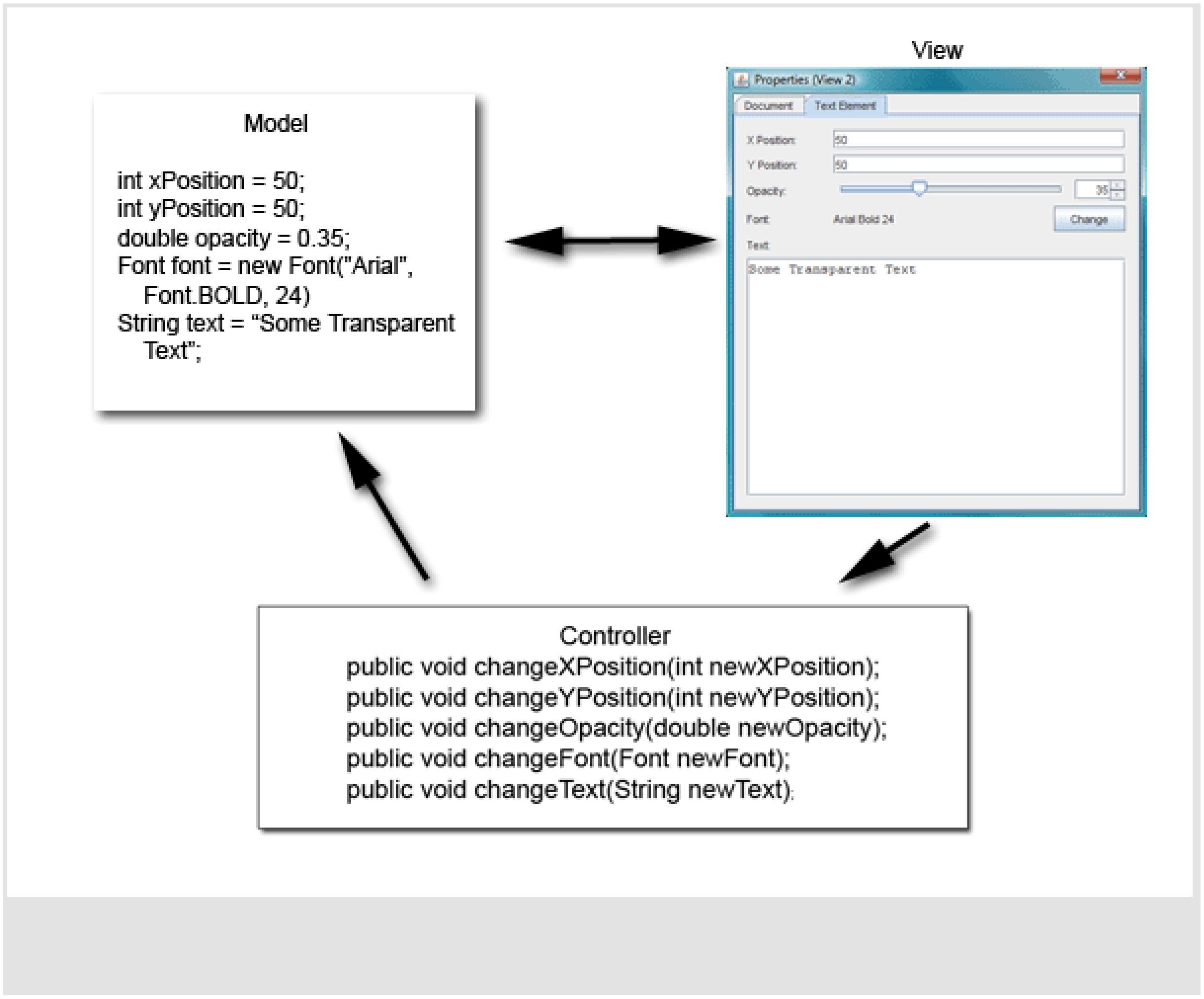
Once a user interacts with the view, the following actions occur:

1. The view recognizes that a GUI action -- for example, pushing a button or dragging a scroll bar -- has occurred, using a listener method that is registered to be called when such an action occurs.

2. The view calls the appropriate method on the controller.

3. The controller accesses the model, possibly updating it in a way appropriate to the user's action.

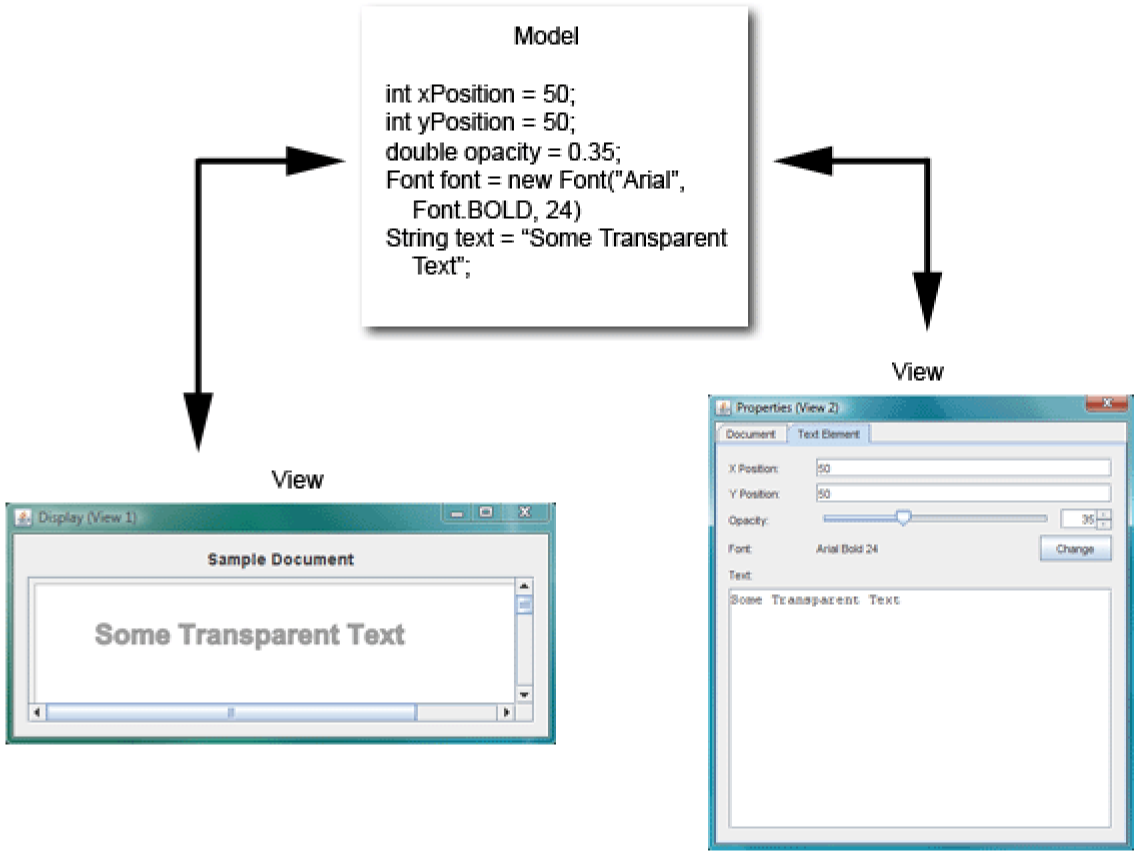
4. If the model has been altered, it notifies interested listeners, such as the view, of the change. In some architectures, the controller may also be responsible for updating the view. This is common in Java technology-based enterprise applications.



```
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
   Font.BOLD, 24)
String text = "Some Transparent
   Text";
```

```
public void changeXPosition(int newXPosition);
public void changeYPosition(int newYPosition);
public void changeOpacity(double newOpacity);
public void changeFont(Font newFont);
public void changeText(String newText);
```

The model does not carry a reference to the view but instead uses an event-notification model to notify interested parties of a change. One of the consequences of this powerful design is that the many views can have the same underlying model. When a change in the data model occurs, each view is notified by a property change event and can update itself accordingly. For example, Figure  below shows two views that use the same data model.



Model

```
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
    Font.BOLD, 24)
String text = "Some Transparent
    Text";
```

X Position: 50

Y Position: 50

Opacity: 35

Font: Arial Bold 24

Text: Some Transparent Text

*Multiple Views Using the Same Model*

Model

• Responsible for data
– Maintains application state (data fields)
– Implements state-changing behavior
– Notifies dependent views/controllers when changes occur (observer pattern)
• Design issues
– How fine-grained are the change descriptions?
• “The string has changed somehow” vs. “Insertion between offsets 3 and 5”
– How fine-grained are the observable parts?
• Entire string vs. only the part visible in a view

The model is responsible for maintaining application specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants.
All mutable objects do that. But a model must also notify its clients when there are changes to its data, so that dependent views can update their displays, and dependent controllers can respond appropriately. Models do this notification using the **observer pattern**, in which interested views and controllers register themselves as listeners for events generated by the model. Designing these notifications is not always trivial, because a model typically has many parts that might have changed. Even in our simple text box example, the string model has a number of characters. A list box has a list of items. When a model notifies its views about a change, how finely should the change be described? Should it simply say "something has changed", or should it say "these particular parts have changed"? Fine-grained notifications may save dependent views from unnecessarily querying state that hasn't changed, at the cost of more bookkeeping on the model's part.
Fine-grained notifications can be taken a step further by allowing views to make fine-grained registrations, registering interest only in certain parts of the model. Then a view displaying a small portion of a large model would only receive events for changes in the part it's interested in. Reducing the grain of notification or registration is crucial to achieving good interactive view performance on large models.

## View

• Responsible for output
–Occupies screen extent (position, size)
–Draws on the screen
– Listens for changes to the model
–Queries the model to draw it
• A view has only one model
–But a model can have many views
In MVC, view objects are responsible for output. A view occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

## Controller

• Responsible for input
– Listens for keyboard & mouse events
– Instructs the model or the view to change
accordingly
• e.g., character is inserted into the text string
• A controller has only one model and one view
Finally, the controller handles all the input. It receives keyboard and mouse events, and instructs the model to change accordingly. For example, the controller of a text box receives keystrokes and inserts them into the text string.
te of the user interface. It isn't clear where in the MVC pattern this kind of data should go.

## View Hierarchy

• Views are arranged into a hierarchy

• Containers
– Window, panel, rich text widget
• Components
– Canvas, button, label, textbox
– Containers are also components
• Every GUI system has a view hierarchy, and
the hierarchy is used in lots of ways
– Output
– Input
– Layout
The second important pattern we want to discuss  is the **view hierarchy**.
Views are arranged into a hierarchy of containment, which some views (called containers in the Java nomenclature) can contain other views (called components in Java). A crucial feature of this hierarchy is that containers are themselves components – i.e., Container is a subclass of Component.
Thus a container can include other containers, allowing a hierarchy of arbitrary depth.
Virtually every GUI system has some kind of view hierarchy. The view hierarchy is a powerful structuring idea, which is loaded with a variety of responsibilities in a typical GUI. We'll look at three ways the view hierarchy is used: for output, input, and layout.

## View Hierarchy: Output

• Drawing
– Draw requests are passed top-down through the hierarchy
• Clipping
– Parent container prevents its child components from drawing
outside its extent
• Z-order
– Children are (usually) drawn on top of parents
– Child order dictates drawing order between siblings
• Coordinate system
– Every container has its own coordinate system (origin
usually at the top left)
– Child positions are expressed in terms of parent coordinates
First, and probably primarily, the view hierarchy is used to organize output: drawing the views on the screen. **Draw requests** are passed down through the hierarchy. When a container is told to draw itself, it must make sure to pass the draw request down to its children as well.
The view hierarchy also enforces a spatial hierarchy by **clipping** – parent containers preventing their children from drawing anything outside their parent's boundaries.
The hierarchy also imposes an implicit layering of views, called **z-order**. When two components overlap in extent, their z-order determines which one will be drawn on top. The z-order corresponds to an in-order traversal of the hierarchy. In other words, children are drawn on top of their parents, and a child appearing later in the parent's children list is drawn on top of its earlier siblings.
Each component in the view hierarchy has its own coordinate system, with its origin (0,0) usually at the top left of its extent. The positions of a container's children are expressed in terms

of the container's coordinate system, rather than in terms of full-screen coordinates. This allows a complex container to move around the screen without changing any of the coordinates of its descendents.

## View Hierarchy: Input

• Event dispatch and propagation
–Raw input events (key presses, mouse movements, mouse clicks) are sent to lowest component
–Event propagates up the hierarchy until some component handles it
• Keyboard focus
–One component in the hierarchy has the focus (implicitly, its ancestors do too)
In most GUI systems, the view hierarchy also participates in input handling.
Raw mouse events – button presses, button releases, and movements – are sent to the smallest component (deepest in the view hierarchy) that encloses the mouse position. If this component chooses not to handle the event, it passes it up to its parent container. The event propagates upward through the view hierarchy until a component chooses to handle it, or until it drops off the top, ignored.
Keyboard events are treated similarly, except that the first component to receive the event is determined by the **keyboard focus**, which always points to some component in the view hierarchy.

## View Hierarchy: Layout

• Automatic layout: children are positioned and sized within parent
–Allows window resizing
–Smoothly deals with internationalization and platform differences (e.g. fonts or widget sizes)
– Lifts burden of maintaining sizes and positions from the programmer
• Although actually just raises the level of abstraction, because you still want to get the graphic design (alignment & spacing) right The view hierarchy is also used to direct the **layout** process, which determines the extents (positions and sizes) of the views in the hierarchy. Many GUI systems have supported automatic layout, including Java AWT and Swing.
Automatic layout is most useful because it allows a view hierarchy to adjust itself automatically when the user resizes its window, changing the amount of screen real estate allocated to it. Automatic layout also smoothly handles variation across platforms, such as differences in fonts, or differences in label lengths due to language translation.

## Observer Pattern

• Observer pattern is used to decouple model from views

### Basic Interaction

Here's the conventional interaction that occurs in the observer pattern. (We'll use the abstract representation of Model and Observer shown on the right. Real models and observers will have different, more specific names for the methods, and different method signatures. They'll also have multiple versions of each of these methods.)

1. An observer **registers** itself to receive notifications from the model.
2. When the model changes (usually due to some other object **modifying** it), the model broadcasts the change to all its registered views by calling **update** on them. The update call usually includes some information about what change occurred. One way is to have different update methods on the observer for each kind of change (e.g. treeStructureAdded() vs. treeStructureRemoved()). Another way is to package the change information into an **event** object. Regardless of how it's packaged, this change information that is volunteered by the model is usually called **pushed** data.
3. An observer reacts to the change in the model, often by **pulling** other data from the model using **get** calls.

## II.2. **Human Capabilities**

• Human information processing
– Perception
–Motor skills
–Memory
–Decision making
– Attention
– Vision

This course is about building effective human-computer interfaces. Just as it helps to understand the properties of the computer system you're programming for – its processor speed, memory size, hard disk, operating system, and the interaction between these components – it's going to be important for us to understand some of the properties of the human that we're designing for.

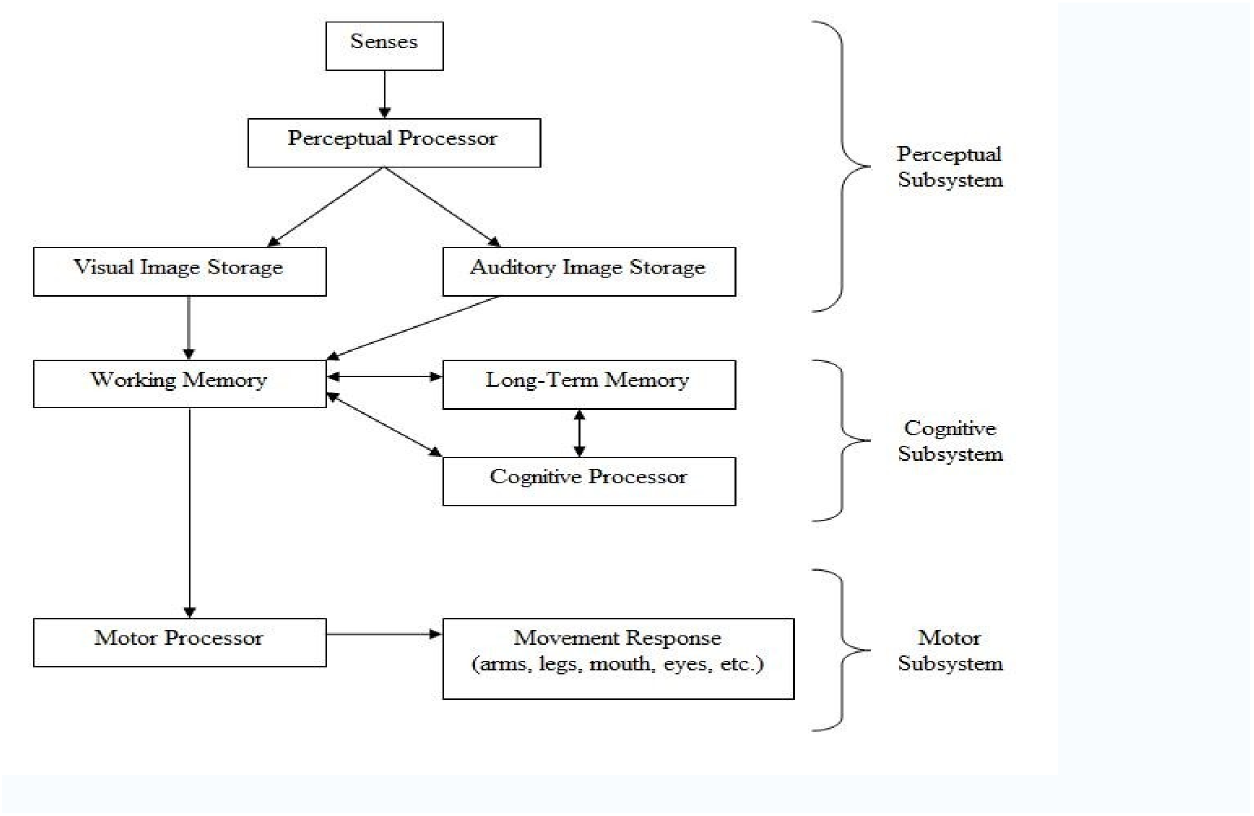
We talked about user analysis, which collects information about high-level properties of our target users, particularly ways in which the target users are different from ourselves.

Here, we're going to look at low-level details: the processors, memories, and properties of the human cognitive apparatus. And we will largely concentrate on properties that most of us have in common (with some important exceptions when we look at color).

### **Human Information Processing**

Here's a high-level look at the cognitive abilities of a human being -- really high level.

A model of the human processor is shown below.



This is a version of the Model Human Processor was developed by Card, Moran, and Newell as a way to summarize decades of psychology research in an **engineering model**. (Card, Moran, Newell, *The Psychology of Human- Computer Interaction*, Lawrence Erlbaum Associates, 1983)
This model is an abstraction, of course. But it's an abstraction that actually gives us *numerical parameters*
describing how we behave. Just as a computer has memory and processor, so does our model of a human.
Actually, the model has several different kinds of memory, and several different processors.
Input from the eyes and ears is first stored in the **short-term sensory store**. As a computer hardware analogy, this memory is like a frame buffer, storing a single frame of perception.
The **perceptual processor** takes the stored sensory input and attempts to recognize *symbols* in it: letters, words, phonemes, icons. It is aided in this recognition by the **long-term memory**, which stores the symbols you know how to recognize.
The **cognitive processor** takes the symbols recognized by the perceptual processor and makes comparisons and decisions. It might also store and fetch symbols in **working memory** (which you might think of as RAM, although it's pretty small). The cognitive processor does most of the work that we think of as "thinking".
The **motor processor** receives an action from the cognitive processor and instructs the muscles to execute it.
There's an implicit **feedback** loop here: the effect of the action (either on the position of your body or on the state of the world) can be observed by your senses, and used to correct the motion in a continuous process.

Finally, there is a component corresponding to your **attention**, which might be thought of like a thread of
control in a computer system.
Note that this model isn't meant to reflect the anatomy of your nervous system. There probably isn't a single area in your brain corresponding to the perceptual processor, for example. But it's a useful abstraction nevertheless.
We'll look at each of these parts in more detail.

## Memories

• Memory properties
– Encoding: type of things stored
– Size: number of things stored
–Decay time: how long memory lasts
Each component of our model has some properties. For example, memories are characterized by three properties: encoding, size, and decay time.

## Short-Term Sensory Store

• Visual information store
– encoded as physical image
– size ~ 17 [7-17] letters
– decay ~ 200 ms [70-1000 ms]
• Auditory information store
– encoded as physical sound
– size ~ 5 [4.4-6.2] letters
– decay ~ 1500 ms [900-3500 ms]
The **visual image store** is basically an image frame from the eyes. It isn't encoded as pixels, but as physical features of the image, such as curvature, length, edges. It retains physical features like intensity that may be discarded in higher-level memories (like the working memory). We measure its size in letters because psych studies have used letters as a convenient stimulus for measuring the properties of the VIS; this doesn't mean that letters are represented symbolically in the VIS. The VIS memory is fleeting, decaying in a few hundred milliseconds.
The **auditory image store** is a buffer for physical sound. Its size is much smaller than the VIS (in terms of letters), but lasts longer – seconds, rather than tenths of a second.
Both of these stores are **preattentional**; that is, they don't need the spotlight of attention to focus on them in order to be collected and stored. Attention can be focused on the visual or auditory stimulus after the fact. That accounts for phenomena like "What did you say? Oh yeah."

## Processors

• Processors have a cycle time
– Tp ~ 100ms [50-200 ms]
– Tc ~ 70ms [30-100 ms]
– Tm ~ 70ms [25-170 ms]
• Fastman may be 10x faster than

Slowman
Turning to the processors, the main property of a processor is its **cycle time**, which is analogous to the cycle time of a computer processor. It's the time needed to accept one input and produce one output.
Like all parameters, the cycle times shown above are derived from a survey of psychological studies. Each parameter is specified with a typical value and a range of reported values. For example, the typical cycle time for perceptual processor, Tp, is 100 milliseconds, but studies have reported between 50 and 200 milliseconds. The reason for the range is not only variance in individual humans; it is also varies with conditions. For example, the perceptual processor is faster (shorter cycle time) for more intense stimuli, and slower for weak stimuli. You can't read as fast in the dark. Similarly, your cognitive processor actually works faster under load! Consider how fast your mind works when you're driving or playing a video game, relative to sitting quietly and reading. The cognitive processor is also faster on practiced tasks.
It's reasonable, when we're making engineering decisions, to deal with this uncertainty by using all three numbers, not only the nominal value but also the range. Card, Moran, & Newell gave names to these three imaginary humans: Fastman, whose times are all as fast as possible; Slowman, whose times are all slow; and Middleman, whose times are all typical.

## Perceptual Fusion

• Two stimuli within the same PP cycle
(Tp ~ 100ms) appear **fused**
• Consequences
– 1/ Tp frames/sec is enough to perceive a
moving picture (10 fps OK, 20 fps smooth)
–Computer response < Tp feels
instantaneous
–Causality is strongly influenced by fusion
One interesting effect of the perceptual processor is **perceptual fusion**. Here's an intuition for how fusion works. Every cycle, the perceptual processor grabs a frame (snaps a picture). Two events occurring less than the cycle time apart are likely to appear in the same frame. If the events are similar – e.g., Mickey Mouse appearing in one position, and then a short time later in another position – then the events tend to *fuse* into a single perceived event – a single Mickey Mouse, in motion.
Perceptual fusion is responsible for the way we perceive a sequence of movie frames as a moving picture, so the parameters of the perceptual processor give us a lower bound on the frame rate for believable animation. 10 frames per second is good for Middleman, but 20 frames per second is better for Fastman (remember that Fastman's Tp = 50 ms represents not just the quickest humans, but also the most favorable conditions).
Perceptual fusion also gives an upper bound on good computer response time. If a computer responds to a user's action within Tp time, its response feels instantaneous with the action itself. Systems with that kind of response time tend to feel like extensions of the user's body. If you used a text editor that took longer than Tp response time to display each keystroke, you would notice.

Fusion also strongly affects our perception of causality. If one event is closely followed by another – e.g., pressing a key and seeing a change in the screen – and the interval separating the events is less than Tp, then we are more inclined to believe that the first event caused the second.

## Bottom-up vs. Top-Down Perception

• Bottom-up uses features of stimulus
• Top-down uses context
– temporal, spatial
– draws on long-term memory
Perception is not an isolated process. It uses both **bottom-up** processing, in which the features of a stimulus are combined to identify it, and **top-down** processing, where the context of the stimulus contributes to its recognition. In visual perception, the context is usually *spatial*, i.e., what's around the stimulus. In auditory perception, the context is *temporal* -- what you heard before or after the stimulus.

## Chunking

• "Chunk": unit of perception or memory
• Chunking depends on presentation and
what you already know
B M W R C A A O L I B M F B I
MWR CAA OLI BMF BIB
BMW RCA AOL IBM FBI
• 3-4 digit chunking is ideal for encoding
unrelated digits
The elements of perception and memory are called **chunks**. In one sense, chunks are defined symbols; in another sense, a chunk represents the activation of past experience.
Our ability to form chunks in working memory depends strongly on how the information is presented – a sequence of individual letters tend to be chunked as letters, but a sequence of three-letter groups tend to be chunked as groups. It also depends on what we already know. If the three letter groups are well-known TLAs (three-letter acronyms) with well-established chunks in long-term memory, we are better able to retain them in working memory.
Chunking is illustrated well by a famous study of chess players. Novices and chess masters were asked to study chess board configurations and recreate them from memory. The novices could only remember the positions of a few pieces. Masters, on the other hand, could remember entire boards, but only when the pieces were arranged in *legal* configurations. When the pieces were arranged randomly, masters were no better than novices. The ability of a master to remember board configurations derives from their ability to **chunk** the board, recognizing patterns from their past experience of playing and studying games.

## Attention and Perception

• Spotlight metaphor
– Spotlight moves serially from one input
channel to another
– **Visual dominance**: easier to attend to
visual channels than auditory channels
– All stimuli within spotlighted channel are

processed in parallel
•Whether you want to or not
Let’s look at how attention is involved with perception. The metaphor used by cognitive psychologists for how attention behaves in perception is the **spotlight**: you can focus your attention (and your perceptual processor) on only one input channel in your environment at a time. This input channel might be a location in your visual field, or it might be a location or voice in your auditory field. Humans are very visually-oriented, so it turns out to be easier to attend to visual channels than auditory channels.
Once you’ve focused your attention on a particular channel, all the stimuli within the area of the “spotlight” are then processed, whether you mean to or not. This can cause **interference**.

## Cognitive Processing

• Cognitive processor
– compares stimuli
– selects a response
• Types of decision making
– Skill-based
–Rule-based
– Knowledge-based
Let’s say a little about the cognitive processor, which is responsible for making comparisons and decisions.
Cognition is a rich, complex process. The best-understood aspect of it is **skill-based** decision making. A skill is a procedure that has been learned thoroughly from practice; walking, talking, pointing, reading, driving, typing are skills most of us have learned well.
Skill-based decisions are automatic responses that require little or no attention. Since skill based decisions are very mechanical, they are easiest to describe in a mechanical model like the one we’re discussing.
Two other kinds of decision making are **rule-based**, in which the human is consciously processing a set of rules of the form *if X, then do Y*; and **knowledge-based**, which involves much higher-level thinking and problem-solving. Rule-based decisions are typically made by novices at a task: when a student driver approaches an intersection, for example, he must think explicitly about what he needs to do in response to each possible condition.
Knowledge-based decision making is used to handle unfamiliar or unexpected problems, such as figuring out why your car won’t start.
We’ll focus on skill-based decision making for the purposes of this course, because it’s well understood.

## Hick-Hyman Law of Choice Reaction Time

• Reaction time depends on information
content of stimulus
Simple reaction time – responding to a single stimulus with a single response – takes just one cycle of the human information processor, i.e. Tp+Tc+Tm.
But if the user must make a **choice** – choosing a different response for each stimulus – then the cognitive processor may have to do more work. The Hick-Hyman Law of Reaction Time shows

that the number of cycles required by the cognitive processor is proportional to amount of **information** in the stimulus. For example, if there are N equally probable stimuli, each requiring a different response, then the cognitive processor needs log N cycles to decide which stimulus was actually seen and respond appropriately. So if you double the number of possible stimuli, a human's reaction time only increases by a constant.
Keep in mind that this law applies only to *skill-based* decision making; we assume that the user has practiced responding to the stimuli, and formed an internal model of the expected probability of the stimuli.

## Speed-Accuracy Tradeoff

• Accuracy varies with reaction time
–Can choose any point on curve
–Can move curve with practice
Another important phenomenon of the cognitive processor is the fact that we can tune its performance to various points on a **speed-accuracy** tradeoff curve. We can force ourselves to make decisions faster (shorter reaction time) at the cost of making some of those decisions wrong. Conversely, we can slow down, take a longer time for each decision and improve accuracy. It turns out that for skill-based decision making, reaction time varies linearly with the log of odds of correctness; i.e., a constant increase in reaction time can double the odds of a correct decision.
The speed-accuracy curve isn't fixed; it can be moved up by practicing the task. Also, people have different curves for different tasks; a pro tennis player will have a high curve for tennis but a low one for surgery.

## Divided Attention (Multitasking)

• Resource metaphor
– Attention is a resource that can be divided among different tasks simultaneously
• Multitasking performance depends on:
– Task structure
• Modality: visual vs. auditory
• Encoding: spatial vs. verbal
• Component: perceptual/cognitive vs. motor vs. WM
– Difficulty
• Easy or well-practiced tasks are easier to share
Earlier we saw the spotlight metaphor for attention. Now we'll refine it to account for our ability to handle multiple things at the same time. The **resource metaphor** regards attention as a limited resource that can be subdivided, under the human's control, among different tasks simultaneously.
Our ability to divide our attention among multiple tasks appears to depend on two things.
First is the **structure** of the tasks that are trying to share our attention. Tasks with different characteristics are easier to share; tasks with similar characteristics tend to interfere.
Important dimensions for task interference seem to be the **modality** of the task's input (visual or auditory), its **encoding** (e.g., spatial/graphical/sound encoding, vs. words), and the mental **components** required to perform it. For example, reading two things at the same time is

much harder than reading and listening, because reading and listening use two different modalities.
The second key influence on multitasking performance is the difficulty of the task.
Carrying on a conversation while driving a car is fairly effortless as long as the road is familiar and free of obstacles; when the driver must deal with traffic or navigation, conversation tends to slow down or even stop.

## Motor Processing

• Open-loop control
– Motor processor runs a program by itself
– cycle time is Tm ~ 70 ms
• Closed-loop control
– Muscle movements (or their effect on the world)
are perceived and compared with desired result
– cycle time is Tp + Tc + Tm ~ 240 ms
The motor processor can operate in two ways. It can run autonomously, repeatedly issuing the same instructions to the muscles. This is “open-loop” control; the motor processor receives no feedback from the perceptual system about whether its instructions are correct.
With open loop control, the maximum rate of operation is just Tm.
The other way is “closed-loop” control, which has a complete feedback loop. The perceptual system looks at what the motor processor did, and the cognitive system makes a decision about how to correct the movement, and then the motor system issues a new instruction. At best, the feedback loop needs one cycle of each processor to run, or Tp + Tc + Tm ~ 240 ms.

## Fitts’s Law

Fitts’s Law specifies how fast you can move your hand to a target of a certain size at a certain distance away (within arm’s length, of course). It’s a fundamental law of the human sensory-motor system, which has been replicated by numerous studies. Fitts’s Law applies equally well to using a mouse to point at a target on a screen.

## Explanation of Fitts’s Law

• Moving your hand to a target is closedloop
control
• Each cycle covers remaining distance D
with error εD
We can explain Fitts’s Law by appealing to the human information processing model. Fitt’s Law relies on closed-loop control. In each cycle, your motor system instructs your hand to move the entire remaining distance D. The accuracy of that motion is proportional to the distance moved, so your hand gets within some error εD of the target (possibly undershooting, possibly overshooting). Your perceptual and cognitive processors perceive where your hand arrived and compare it to the target, and then your motor system issues a correction to move the remaining distance εD – which it does, but again with proportional error, so your hand is now within ε2D. This process repeats, with the error decreasing geometrically, until *n* iterations have brought your hand within the target – i.e., εnD ≤ ½ S.

## Implications of Fitts’s Law

- Targets at screen edge are easy to hit
- Mac menubar beats Windows menubar
- Unclickable margins are foolish
- Hierarchical menus are hard to hit
- Linear popup menus vs. pie menus

Fitts's Law has some interesting implications:
•The edge of the screen stops the mouse pointer, so you don't need more than one correcting cycle to hit it. Essentially, the edge of the screen acts like a target with *infinite* size. So edgeof-screen real estate is precious. The Macintosh menu bar, positioned at the top of the screen, is faster to use than a Windows menu bar (which, even when a window is maximized, is displaced by the title bar). Similarly, if you put controls at the edges of the screen, they should be active all the way to the edge to take advantage of this effect. Don't put an unclickable margin beside them.
• Hierarchical submenus are hard to use, because of the correction cycles the user is forced to spend getting the mouse pointer carefully over into the submenu. Windows tries to solve this problem with a 500 ms timeout, and now we know another reason that this solution isn't ideal: it exceeds Tp (even for Slowman), so it destroys perceptual fusion and our sense of causality. Intentionally moving the mouse down to the next menu results in a noticeable delay.
•Fitts's Law also explains why pie menus are faster to use than linear popup menus. With a pie menu, every menu item is a slice of a pie centered on the mouse pointer. As a result, each menu item is the same distance D away from the mouse pointer, and its size S (in the radial direction) is comparable to D. Contrast that with a linear menu, where items further down the menu have larger D, and all items have a small S (height).

## Working Memory (WM)

• Small capacity: 7 ± 2 "chunks"
• Fast decay (7 [5-226] sec)
• **Maintenance rehearsal** fends off decay
• **Interference** causes faster decay
Working memory is where you do your conscious thinking. Working memory is where the cognitive processor gets its operands and drops its results. The currently favored model in cognitive science holds that working memory is not actually a separate place in the brain, but rather a pattern of **activation** of elements in the long-term memory.
A famous result, due to George Miller (unrelated), is that the capacity of working memory is roughly 7 ± 2 chunks.
Working memory decays in tens of seconds. **Maintenance rehearsal** – repeating the items to yourself – fends off this decay, much as DRAM must refresh itself. Maintenance rehearsal requires attentional resources. But distraction destroys working memory. A particularly strong kind of distraction is **interference** – stimuli that activate several conflicting chunks are much harder to retain.

## Long-term Memory (LTM)

• Huge capacity
• Little decay
• **Elaborative rehearsal** moves chunks

from WM to LTM by making
connections with other chunks
Long-term memory is probably the least understood part of human cognition. It contains the mass of our memories. Its capacity is huge, and it exhibits little decay. Long-term memories are apparently not intentionally erased; they just become inaccessible.
Maintenance rehearsal (repetition) appears to be useless for moving information into into long-term memory. Instead, the mechanism seems to be **elaborative rehearsal**, which seeks to make connections with existing chunks. Elaborative rehearsal lies behind the power of mnemonic techniques like associating things you need to remember with familiar places, like rooms in your childhood home. Elaborative rehearsal requires attention resources as well.

### Color Blindness

• Red-green color blindness (protonopia & deuteranopia)
– 8% of males
– 0.4% of females
• Blue-yellow color blindness (tritanopia)
– Far more rare
• Guideline: don’t depend solely on color distinctions
– use redundant signals: brightness, location, shape
Color deficiency (“color blindness”) affects a significant fraction of human beings. An overwhelming number of them are male.
There are three kinds of color deficiency:
•**Protonopia :**The consequence is reduced sensitivity to red-green differences and reds are perceived as darker than normal.
•**Deuteranopia :**Red-green difference sensitivity is reduced, but reds do not appear darker.
•**Tritanopia :**results in blue-yellow insensitivity.
Since color blindness affects so many people, it is essential to take it into account when you are deciding how to use color in a user interface. Don’t depend solely on color distinctions, particularly red-green distinctions, for conveying information. Microsoft Office applications fail in this respect: red wavy underlines indicate spelling errors, while identical green wavy underlines indicate grammar errors.
Traffic lights are another source of problems. How do red-green color-blind people know whether the light is green or red? Fortunately, there’s a spatial cue: red is always above (or to the right of) green. Protonopia sufferers (as opposed to deuteranopians) have an additional advantage: the red light looks darker than the green light.

## II.3. **Models & Metaphors**

• Conceptual models
• Interaction styles
• Direct manipulation
• Errors

### Models

• **Model** of a system = how it works

– its constituent parts and how they work
together to do what the system does
• Implementation models
– Pixel editing vs. structured graphics
– Text file as single string vs. list of lines
• Interface models

A **model** of a system is a way of describing how the system works. A model specifies what the parts of the system are, and how those parts interact to make the system do what it's supposed to do.

Example: most modern text editors model a text file as a single string, in which line endings are just like other characters. But it doesn't have to be this way. Some editors represent a text file as a list of lines instead. When this implementation model is exposed in the user interface, as in the *vi* text editor, line endings can't be deleted in the same way as other characters. *Vi* has a special join command for deleting line endings.

A model may concern only a small part of a system.

## Models in UI Design

• Three models are relevant to UI design:
System
model
Interface
model
User
model

There are actually several models you have to worry about in UI design:

•The **system model** (sometimes called implementation model) is how the system actually works.
•The **interface model** (or manifest model) is the model that the system presents to the user.
•The **user model** (or conceptual model) is how the user *thinks* the system works.

## Interface Model Hides System Model

• Interface model should be:
– Simple
– Appropriate: reflect user's model of the
task (learned from task analysis)
–Well-communicated

The interface model might be quite different from the system model. A text editor whose system model is a list of lines doesn't have to present it that way through its interface. The interface could allow deleting line endings as if they were characters, even though the actual effect on the system model is quite different.

Similarly, a cell phone presents the same simple interface model as a conventional wired phone, even though its system model is quite a bit more complex. A cell phone conversation may be handed off from one cell tower to another as the user moves around. This detail of the system model is hidden from the user.

As a software engineer, you should be quite familiar with this notion. A module interface offers a certain model of operation to clients of the module, but its implementation may be significantly different. In software engineering, this divergence between interface and implementation is valued as a way to manage complexity and plan for change. In user interface design, we value it primarily for other reasons: the interface model should be simpler and more closely reflect the user's model of the actual task, which we can learn from task analysis.

## Interaction Styles

• Command language
• Menus & forms
• Direct manipulation

Let's get a little more concrete now, and look at three major kinds of user interface styles. We'll tackle them in roughly chronological order as they were developed.

## Command Language

• User types in commands in an artificial language
• Examples
–Unix shell ("ls –l *.java")
– Search engine query language ("AND, OR, site:www.mit.edu")
–URLs ("http://www.ulk.ac.rw/admissions/")
• Command syntax is important

The earliest computer interfaces were command languages: job control languages for early computers, which later evolved into the Unix command line.
Although a command language is rarely the first choice of a user interface designer nowadays, they still have their place – often as an advanced feature embedded inside another interaction style. For example, Google's query operators form a command language. Even the URL in a web browser is a command language, with particular syntax and semantics.
When designing a command language, the key problem is designing the command syntax.
Task analysis drives the choice of commands, the names you give them, the parameters they have, and the syntax for fitting them together.

## Menus and Forms

• User is prompted to choose from menus and fill in forms
• Examples
– virtually all web sites
– dialog boxes
• Navigation structure is important
–Menu trees (Yahoo!)
–Wizard: linear sequence of forms

A menu/form interface presents a series of menus or forms to the user. Virtually all web sites behave this way. Dialog boxes are a form-style interface frequently found embedded inside another interaction style.
The navigation structure is the important design problem for menu/form interfaces. Task

analysis tells you what choices need to be available, where they should be placed in a menu tree, and what data types or possible responses need to be available in a form.

## Direct Manipulation

• User interacts with visual representation of data objects
– Continuous visual representation
– Physical actions or labeled button presses
– Rapid, incremental, reversible, immediately visible effects
• Examples
– Files and folders on a desktop
– Scrollbar
– Dragging to resize a rectangle
– Selecting text
• Visual representation and physical interaction are important

Finally, we have direct manipulation: the preeminent interface style for graphical user interfaces. Direct manipulation is defined by three principles [Shneiderman, *Designing the User Interface*, 2004]:

1. A **continuous visual representation** of the system's data objects. Examples of this visual representation include: icons representing files and folders on your desktop; graphical objects in a drawing editor; text in a word processor; email messages in your inbox. The representation may be verbal (words) or iconic (pictures), but it's continuously displayed, not displayed on demand.
2. The user interacts with the visual representation using **physical actions** or **labeled button presses**. Physical actions might include clicking on an object to select it, dragging it to move it, or dragging a selection handle to resize it. Physical actions are the *most* direct kind of actions in direct manipulation – you're interacting with the virtual objects in a way that feels like you're pushing them around directly. But not every interface function can be easily mapped to a physical action (e.g., converting text to boldface), so we also allow for "command" actions triggered by pressing a button – but the button should be visually rendered in the interface, so that pressing it is analogous to pressing a physical button.
3. The effects of actions should be **rapid** (within 100 ms), **incremental** (you can drag the scrollbar thumb a little or a lot, and you see each incremental change), **reversible** (you can undo your operation – with physical actions this is usually as easy as moving your hand back to the original place, but with labeled buttons you typically need an Undo command), and **immediately visible**.

Why is direct manipulation so powerful? It exploits perceptual and motor skills of the human machine – and depends less on linguistic skills than command or menu/form interfaces.

## Comparison of Interaction Styles

• Knowledge in the head vs. world
• Error messages
• Efficiency
• User experience
• Synchrony

• Programming difficulty
• Accessibility

**Error messages**: DM rarely needs them. No error message when you drag a scrollbar “too far”, for example.

**Knowledge in the head vs. the world.** Command languages require the user to put a lot of knowledge into their heads, by training, practice, etc. (Or else compensate by having manuals, reference cards, or online help close at hand while using the system.) Menus and forms put much more information into the world. Welldesigned DM also has information in the world, from the affordances and constraints of the visual metaphor.

**Efficiency**: experts can be very efficient with command languages (no need to scan system prompts; able to reuse commands in scripts and history). Efficient performance with menus and form interfaces demands good shortcuts (e.g. keyboard shortcuts, tabbing between form fields, typeahead). Efficient performance with DMs is possible when the DM is appropriate to the task (and depends on shortcuts and Fitts’s Law); but using a DM for a task it isn’t designed for may turn into manual labor.

**User experience:** command languages best for expert users – frequent, well-trained. Menus/forms and DMs better for novices and infrequent users.

**Synchrony:** Command languages are synchronous (first the user types a complete command, then the system does it). So are menu systems and forms; e.g. web model. DM is asynchronous: user can point mouse anywhere and do anything at any time.

**Programming difficulty:** Command languages are relatively easy: parsing text, rigid requirements. Menus and forms have substantial toolkit support; e.g., the web browser and HTML, Java Swing components. DM is hardest to program: have to draw, handle low-level input (keyboard, mouse), display feedback.

**Accessibility**: command and menu/form interfaces are more textual, so easier for vision-impaired users to read with screen readers. DM interfaces are much harder for these users.

## Direct Manipulation Cues

• Affordances
• Constraints
• Natural mapping
• Visibility
• Feedback

So what is the language by which a direct manipulation interface communicates its model to the user? Or, looking at it from the user’s perspective, what cues do users rely on in order to learn the model: the **parts** that make up the interface, and **how those parts work together**?

Don Norman, in his great book The Design of Everyday Things, identified a number of cues from our interaction with physical objects, like doors and scissors. Since a direct
manipulation interface is intended to be a visual metaphor for physical interaction, we’ll
look at some of these cues and how they apply to computer interfaces.

### Affordances

• Perceived and actual properties of a
thing that determine how the thing could
be used

–Chair is for sitting
– Knob is for turning
– Button is for pushing
– Listbox is for selection
– Scrollbar is for continuous scrolling or
panning
• Perceived vs. actual
According to Norman, affordance refers to “the perceived and actual properties of a thing”, primarily the properties that determine how the thing could be operated.
Note that **perceived** affordance is not the same as **actual** affordance. A facsimile of a chair made of papier-mache has a perceived affordance for sitting, but it doesn’t actually afford sitting: it collapses under your weight. Conversely, a fire hydrant has no perceived affordance for sitting, since it lacks a flat, human-width horizontal surface, but it actually does afford sitting, albeit uncomfortably.
The parts of a user interface should agree in perceived and actual affordances.

## Natural Mapping

• Physical arrangement of controls should
match arrangement of function
• Best mapping is direct, but natural
mappings don’t have to be direct
– Light switches
– Stove burners
– Turn signals
– Audio mixer
Logical constraints lead to another important principle of interface communication: **natural mapping** of functions to controls.
Consider the spatial arrangement of light switch panel. How does each switch correspond to the light it controls? If the switches are arranged in the same fashion as the lights, it is much easier to learn which switch controls which light.
Direct mappings are not always easy to achieve, since a control may be oriented differently from the function it controls. Light switches are mounted vertically, on a wall; the lights themselves are mounted horizontally, on a ceiling. So the switch arrangement may not correspond *directly* to a light arrangement.
Other good examples of mapping include:
•Stove burners. Many stoves have four burners arranged in a square, and four control knobs arranged in a row. Which knobs control which burners? Most stoves don’t make any attempt to provide a natural mapping.
•Car turn signals. The turn signal switch in most cars is a stalk that moves up and down, but the function it controls is a signal for left or right turn. So the mapping is not direct, but it is nevertheless natural.

## Visibility

• Relevant parts of system should be
visible
–Not usually a problem in the real world
– But takes extra effort in computer

interfaces
Visibility is an essential principle – probably the most important – in communicating a model to the user.
If the user can't *see* an important control, they would have to (1) guess that it exists, and (2) guess where it is.
Visibility is not usually a problem with physical objects, because you can usually tell its parts just by looking at it. Look at a bicycle, or a pair of scissors, and you can readily identify the pieces that make it work. Although parts of physical objects can be made hidden or invisible – for example, a door with no obvious latch or handle – in most cases it takes more design work to hide the parts than just to leave them visible.
The opposite is true in computer interfaces. A window can interpret mouse clicks anywhere in its boundaries in arbitrary ways. The input need not be related at all to what is being displayed. In fact, it takes more effort to make the parts of a computer interface visible than to leave them invisible. So you have to guard carefully against invisibility of parts in computer interfaces.

## Feedback

• Actions should have immediate, visible effects
– Push buttons
– Scrollbars
–Drag & drop
• Kinds of feedback
– Visual
– Audio
–Haptic
The final principle of interface communication is feedback: what the system does when you perform an action. When the user successfully makes a part work, it should appear to respond. Push buttons depress and release. Scrollbar thumbs move. Dragged objects follow the cursor.
Feedback doesn't always have to be visual. **Audio** feedback – like the clicks that a keyboard makes – is another form. So is **haptic** feedback, conveyed by the sense of touch. The mouse button gives you haptic feedback in your finger when you feel the vibration of the click. That's much better feedback then you get from a touchscreen, which doesn't give you any physical sense when you've pressed it hard enough to register.

## Modeling Human Error

• Description error
• Capture error
• Mode error
Let's say a bit about some of the kinds of errors that humans make.

## Description Error

• Intended action is replaced by another action with many features in common
– Pouring orange juice into your cereal
– Putting the wrong lid on a bowl
– Throwing shirt into toilet instead of hamper

• Avoid actions with very similar descriptions
– Long rows of identical switches
– Adjacent menu items that look similar
A description error occurs when two actions are very similar. The user intends to do one action, but accidentally substitutes the other. A classic example of a description error is reaching into the refrigerator for a carton of milk, but instead picking up a carton of orange juice and pouring it into your cereal. The actions for pouring milk in cereal and pouring juice in a glass are nearly identical – open fridge, pick up half-gallon carton, open it, pour–but the user's mental description of the action to execute has substituted the orange juice for the milk.
To limit description errors in computer interfaces, avoid actions with very similar descriptions, like long rows of identical switches.

## Capture Error

• A sequence of actions is replaced by another sequence that starts the same way
– Leave your house and find yourself walking to school instead of where you meant to go
– Vi :wq command
• Avoid habitual action sequences with common prefixes
A capture error occurs when a person starts executing one sequence of actions, but then veers off into another (often more familiar) sequence that happened to start the same way.
A good mental picture for this is that you've developed a mental groove from executing the same sequence of actions repeatedly, and this groove tends to capture other sequences that start the same way.
In a computer interface, you can deal with capture errors by avoiding habitual action sequences that have common prefixes.

## Mode Error

• Modes: states in which actions have different meanings
– Vi's insert mode vs. command mode
– Caps Lock
– Drawing palette
• Avoiding mode errors
– Eliminate modes
– Visibility of mode
– Spring-loaded or temporary modes
– Disjoint action sets in different modes
A third kind of error is a mode error. **Modes** are states in which the same action has different meanings. For example, when Caps Lock mode is enabled on a keyboard, the letter keys produce uppercase letters. The text editor vi is famous for its modes: in insert mode, letter keys are

inserted into your text file, while in command mode (the default), the letter keys invoke editing commands.
Mode errors occur when the user tries to invoke an action that doesn’t have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn’t notice that Caps Lock is enabled, then a mode error occurs.
There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always possible. When modes are necessary, it’s essential to make the mode visible. But visibility is a much harder problem for mode status than it is for affordances. When mode errors occur, the user isn’t actively looking for the mode, like they might actively look for a control. As a result, mode status indicators must be visible in the user’s locus of attention. That’s why the Caps Lock light, which displays the status of the Caps Lock mode on a keyboard, doesn’t really work. (Raskin, The Humane Interface, 2000 has a good discussion of locus of attention as it relates to mode visibility.)
Other solutions are spring-loaded or temporary modes. With a spring-loaded mode, the user has to do
something active to stay in the alternate mode, essentially eliminating the chance that they’ll forget what mode they’re in. The Shift key is a spring-loaded version of the uppercase mode. Drag-and-drop is another springloaded mode; you’re only dragging as long as you hold down the mouse button. Temporary modes are similarly short-term. For example, in many graphics programs, when you select a drawing object like a rectangle or line from the palette, that drawing mode is active only for one mouse gesture. Once you’ve drawn one rectangle, the mode automatically reverts to ordinary pointer selection.
Finally, you can also mitigate the effects of mode errors by designing action sets so that no two modes share any actions. Mode errors may still occur, when the user invokes an action in the wrong mode, but the action can simply be ignored rather than triggering any undesired effect.

## Metaphors

• Another way to address the model
problem
• Examples
–Desktop
– Trashcan

Let’s close by looking again at metaphors.
The advantage of metaphor is that you’re borrowing a conceptual model that the user
already has experience with. A metaphor can convey a lot of knowledge about the interface model all at once. It’s *a notebook*. It’s a *CD case*. It’s a *desktop*. It’s a *trashcan*. Each of these metaphors carries along with it a lot of knowledge about the parts, their purposes, and their interactions, which the user can draw on to make guesses about how the interface will work.
Some interface metaphors are famous and largely successful. The desktop metaphor –
documents and folders on a desk-like surface – is widely used and copied. The trashcan, a place for discarding things but for digging around and bringing them back, is another effective metaphor – so much so that Apple defended its trashcan with a lawsuit, and imitators are forced to use a different look (Recycle Bin, anyone?).

## Dangers of Metaphors

• Hard to find
• Deceptive

• Constraining
• Breaking the metaphor
• Use of a metaphor doesn't excuse bad
communication of the model:
–RealCD's bad affordances, visibility
The basic rule for metaphors is: use it if you have one, but don't stretch for one if you don't.
Appropriate metaphors can be very hard to find, particularly with real-world objects. The designers of RealCD stretched hard to use their CD-case metaphor (since in the real world, CD cases don't even play CDs), so it didn't work well.
Metaphors can also be deceptive, leading users to infer behavior that your interface doesn't provide. Sure, it looks like a book, but can I write in the margin? Can I rip out a page?
Metaphors can also be constraining. Strict adherence to the desktop metaphor wouldn't scale, because documents would always be full-size like they are in the real world.
The biggest problem with metaphorical design is that your interface is presumably more capable than the real-world object, so at some point you have to break the metaphor.
Nobody would use a word processor if *really it* behaved like a typewriter. Features like automatic word-wrapping break the typewriter metaphor, by creating a distinction between hard carriage returns and soft returns.
Most of all, use of a metaphor doesn't excuse an interface that does a bad job communicating its model to the user.

## II.4 Usability Guidelines ("Heuristics")

• Plenty to choose from
– Nielsen's 10 principles
• One version in his book
• A more recent version on his website
– Tognazzini's 16 principles
– Norman's rules from Design of Everyday Things
– Mac, Windows, Gnome, KDE guidelines
• Help designers choose design alternatives
• Help evaluators find problems in interfaces
("heuristic evaluation")
**Usability guidelines, or heuristics**, are rules that distill out the principles of effective user interfaces. There are plenty of sets of guidelines to choose from – sometimes it seems like every usability researcher has their own set of heuristics. Most of these guidelines overlap in important ways, however. The experts don't disagree about what constitutes good UI. They just disagree about how to organize what we know into a small set of operational rules.
For the basis of this course, we'll use Jakob Nielsen's 10 heuristics, which can be found on his web site. (An older version of the same heuristics, with different names but similar content, can be found in his *Usability Engineering* book, one of the recommended books for this course.) Another good list is **Tog's First Principles** (find it in Google), 16 principles from Bruce Tognazzini that include affordances and Fitts's Law.
We talked about some design guidelines proposed by Norman: visibility, affordances, constraints, feedback, and so on.
Platform-specific guidelines are also important and useful to follow. Platform guidelines tend to be very

specific, e.g. you should have a File menu, and there command called Exit on it (not Quit, not Leave, not Go Away). Following platform guidelines ensures consistency among different applications running on the same platform, which is valuable for novice and frequent users alike. However, platform guidelines are relatively limited in scope, offering solutions for only a few of the design decisions in a typical UI.
Heuristics can be used in two ways: during design, to choose among different alternatives; and during
evaluation, to find and justify problems in interfaces.

## Guidelines From Earlier chapters

• User-centered design
– Know your users
– Understand their tasks
• Fitts's Law
– Size and proximity of controls should relate to their importance
– Tiny controls are hard to hit
– Screen edges are precious
• Memory
– Use chunking to simplify information presentation
– Minimize working memory
• Color guidelines
– Don't depend solely on color distinctions (color blindness)
– Avoid red on blue text (chromatic aberration)
– Avoid small blue details
• Norman's principles of direct manipulation
– Affordances
– Natural mapping
– Visibility
– Feedback
Here are some guidelines we've already discussed in earlier chapters.

Let's look at each of Nielsen's 10 heuristics in detail.

## 1. Match the Real World

• Use common words,
not techie jargon
– But use domain-specific
terms where appropriate
• Don't put limits on userdefined
names
• Allow aliases/synonyms
in command languages
• Metaphors are useful
but may mislead
OK

Type mismatch
First, the system should match the real world of the user’s experience as much as possible. Nielsen’s original name for this heuristic was “Speak the user’s language”, which is a good slogan to remember. If the user speaks English, then the interface should also speak English, not Geekish. Technical jargon should be avoided. Use of jargon reflects aspects of the system model creeping up into the interface model,
unnecessarily.
Technical jargon should only be used when it is specific to the application domain and the expected users are domain experts. The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order. An interface designed for doctors shouldn’t dumb down medical terms.
If an interface allows users to name things, then users should be free to choose long, descriptive names.
Artificial limits on length or content should be avoided. DOS used to have a strong limit on filenames, an 8 character name and a 3 character extension. Echoes of these limits persist in Windows even today.
When designing an interface that requires the user to type in commands or search keywords, support as many aliases or synonyms as you can. Different users rarely agree on the same name for an object or command.
One study found that the probability that two users would mention the same name was only 7-18%. (Furnas et al, “The vocabulary problem in human-system communication,” *CACM* v30 n11, Nov. 1987).
Metaphors are one way you can bring the real world into your interface. A well-chosen, well-executed metaphor can be quite effective and appealing, but be aware that metaphors can also mislead. A computer interface must deviate from the metaphor at *some* point -- otherwise, why aren’t you just using the physical object instead? At those deviation points, the metaphor may do more harm than good. For example, it’s easy to say “a word processor is like a typewriter,” but you shouldn’t really *use* it like a typewriter. Pressing Enter every time the cursor gets close to the right margin, as a typewriter demands, would wreak havoc with the word processor’s automatic word-wrapping.

## 2. Consistency and Standards

• Principle of Least Surprise
– Similar things should look and act similar
– Different things should look different
• Other properties
– Size, location, color,
wording, ordering, …
• Command/argument order
– Prefix vs. postfix
• Follow platform standards
The second heuristic is Consistency. This rule is often given the name the Principle of Least Surprise, which basically means that you shouldn’t surprise the user with the way a command or interface

object works. Similar things should look, and act, in similar ways. Conversely, different things should be
visibly different.
A very important kind of consistency is in wording. Use the same terms throughout your user interface. If
your interface says “share price” in one place, “stock price” in another, and “stock quote” in a third, users will wonder whether these are three different things you’re talking about.
Incidentally, we’ve only looked at two heuristics, but already we have a contradiction! Matching the Real
World argued for synonyms and aliases, so a command language should include not only *delete* but *erase* and *remove* too. But Consistency argues for only one name for each command, or else users will wonder whether these are three different commands that do different things. One way around the impasse is to look at the context in which you’re applying the heuristic. When the *user* is talking, the interface should make a maximum effort to understand the user, allowing synonyms and aliases. When the *interface* is speaking, it should be consistent, always using the same name to describe the same command or object. What if the interface is smart enough to adapt to the user – should it then favor matching its output to the user’s vocabulary (and possibly the user’s inconsistency) rather than enforcing its own consistency? Perhaps, but adaptive interfaces are still an active area of research, and not much is known.
Command & argument ordering is another kind of consistency. In **noun-verb order**, the conventional order in graphical user interfaces, the user first selects the object of the command, and then invokes the command. In **verb-noun order**, the command is invoked first, and then the arguments are selected. A drawing program in which some commands were noun-verb and others were verb-noun would be very hard to learn and use.

## Kinds of Consistency

• Internal
• External
• Metaphorical

There are three kinds of consistency you need to worry about: **internal consistency** within your application (like the VB dialog boxes shown); **external consistency** with other applications on the same platform (how do other Windows apps lay out OK and Cancel?); and **metaphorical consistency** with your interface metaphor or similar real-world objects.

## 3. Help and Documentation

• Users don’t read manuals
– Prefer to spend time working toward their task
goals, not learning about your system
• But manuals and online help are vital
– Usually when user is frustrated or in crisis
• Help should be:
– Searchable
– Context-sensitive
– Task-oriented
– Concrete

– Short

The next heuristic is (good) Help and Documentation. The sad fact about documentation is that most users simply don’t read it, at least not before they try the interface. As a result, when they finally *do* want to look at the manual, it’s because they’ve gotten stuck. Good help should take this into account.

A good point was raised in class that exclusively task-oriented help (which has largely taken over in Microsoft Windows) makes it impossible to get a high-level overview of an interface from the manual. So it’s possible to go too far.

## 4. User Control and Freedom

• Provide undo

• Long operations should be cancelable

• All dialogs should have a cancel button

This heuristic used to be called “Clearly Marked Exits” in Nielsen’s old list. Users should not be trapped by

the interface. Every dialog box should have a cancel button, and long operations should be interruptible. Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo. Users should be able to explore the interface without fear of being trapped in a corner. Undo is a great way to support exploration.

## 5. Visibility of System Status

• Keep user informed of system state

– Cursor change

– Selection highlight

– Status bar

– Don’t overdo it…

• Response time

– < 0.1 s: seems instantaneous

– 0.1-1 s: user notices, but no feedback needed

– 1-5 s: display busy cursor

– > 1-5 s: display progress bar

This heuristic used to be called, simply, “Feedback.” Keep the user informed about what’s going on. The system should always keep users informed about what is going on, through appropriate feedback within reasonable time. We’ve developed lots of idioms for feedback in graphical user interfaces. Use them:

•Change the cursor to indicate possible actions (e.g. hand over a hyperlink), modes (e.g. drag/drop), and

activity (hourglass).

•Use highlights to show selected objects. Don’t leave selections implicit.

•Use the status bar for messages and progress indicators.

But don’t overdo it. This dialog box demands a click from the user. Why? Does the interface need a pat on the back for finishing the conversion? It would be better to just skip on and show the resulting documentation.

Depending on how long an operation takes, you may need different amounts of feedback. Even though we say “no feedback needed” if the operation takes less than a second, remember that something should change, visibly, within 100 ms, or perceptual fusion will be disrupted.

## 6. Flexibility and Efficiency

• Provide easily-learned shortcuts for frequent
operations
– Keyboard accelerators
– Command abbreviations
– Styles
– Bookmarks
– History

This heuristic used to be called “Shortcuts.” Frequent users need and want them.
Recently-used history is one very useful kind of shortcut, like this recently-used files menu.
Yes to All and No to All were good, but they don’t smoothly handle the case where the user wants to choose a mix of Yes and No. Eclipse’s list of checkboxes, with Select All and Deselect All, provides the right mix of flexibility and efficiency.

## 7. Error Prevention

• Selection is less error-prone than typing
– But don’t go overboard…
• Disable illegal commands

Now we get into heuristics about error handling. Since humans make errors if they’re given a chance (this is called Murphy’s Law: “if something can go wrong, it will”), the best solution is to prevent errors entirely.
One way to prevent errors is to allow users to select rather type. Misspellings then become impossible. This attitude can be taken to an extreme, however, as shown in this example.
If a command is illegal in the current state of the interface – e.g., Copy is impossible if nothing is selected – then the command should be disabled (“grayed out”) so that it simply can’t be selected in the first place.

## Description Error

• Intended action is replaced by another action
with many features in common
– Pouring orange juice into your cereal
– Putting the wrong lid on a bowl
– Throwing shirt into toilet instead of hamper
– Going to Kendall Square instead of Kenmore
Square
• Avoid actions with very similar descriptions
– Long rows of identical switches
– Adjacent menu items that look similar

A description error occurs when two actions are very similar. The user intends to do one action, but
accidentally substitutes the other. A classic example of a description error is reaching into the refrigerator for a carton of milk, but instead picking up a carton of orange juice and pouring it

into your cereal. The actions for pouring milk in cereal and pouring juice in a glass are nearly identical – open fridge, pick up half-gallon carton, open it, pour– but the user's mental description of the action to execute has substituted the orange juice for the milk.
Description errors can be fought off by applying the converse of the Consistency heuristic: different things
should look and act different, so that it will be harder to make description errors between them. Avoid actions with very similar descriptions, like long rows of identical switches.

## Capture Error

• A sequence of actions is replaced by
another sequence that starts the same
way
– Leave your house and find yourself walking
to school instead of where you meant to go
– Vi :wq command
• Avoid habitual action sequences with
common prefixes
A capture error occurs when a person starts executing one sequence of actions, but then veers off into another (often more familiar) sequence that happened to start the same way. A good mental picture for this is that you've developed a mental groove from executing the same sequence of actions repeatedly, and this groove tends to capture other sequences that start the same way.
In a computer interface, you can deal with capture errors by avoiding habitual action sequences that have
common prefixes.

## Mode Error

• Modes: states in which actions have different
meanings
– Vi's insert mode vs. command mode
– Caps Lock
– Drawing palette
• Avoiding mode errors
– Eliminate modes
– Visibility of mode
– Spring-loaded or temporary modes
– Disjoint action sets in different modes
A third kind of error is a mode error. **Modes** are states in which the same action has different meanings. For example,
when Caps Lock mode is enabled on a keyboard, the letter keys produce uppercase letters. The text editor vi is famous for its modes: in insert mode, letter keys are inserted into your text file, while in command mode (the default), the letter keys invoke editing commands. We talked about another mode error in Gimp: accidentally changing a menu shortcut because your mouse is hovering over it.
Mode errors occur when the user tries to invoke an action that doesn't have the desired effect in the current mode. For example, if the user means to type lowercase letters but doesn't notice that Caps Lock is enabled, then a mode error occurs.

There are many ways to avoid or mitigate mode errors. Eliminating the modes entirely is best, although not always
possible. When modes are necessary, it's essential to make the mode visible. But visibility is a much harder problem for mode status than it is for affordances. When mode errors occur, the user isn't actively looking for the mode, like they might actively look for a control. As a result, mode status indicators must be visible in the user's locus of attention. That's why the Caps Lock light, which displays the status of the Caps Lock mode on a keyboard, doesn't really work. (Raskin, The Humane Interface, 2000 has a good discussion of locus of attention as it relates to mode visibility.)
Other solutions are spring-loaded or temporary modes. With a spring-loaded mode, the user has to do something active to stay in the alternate mode, essentially eliminating the chance that they'll forget what mode they're in. The Shift key is a spring-loaded version of the uppercase mode. Drag-and-drop is another spring-loaded mode; you're only dragging as long as you hold down the mouse button. Temporary modes are similarly short-term. For example, in many graphics programs, when you select a drawing object like a rectangle or line from the palette, that drawing mode is active only for one mouse gesture. Once you've drawn one rectangle, the mode automatically reverts to ordinary pointer selection.
Finally, you can also mitigate the effects of mode errors by designing action sets so that no two modes share any actions.
Mode errors may still occur, when the user invokes an action in the wrong mode, but the action can simply be ignored
rather than triggering any undesired effect. (Although then, you might ask, why have two modes in the first place?)

## 8. Recognition, Not Recall

- Use menus, not command languages
- Use combo boxes, not textboxes
- Use generic commands where possible
(Open, Save, Copy Paste)
- All needed information should be visible

There's another reason why selection is better than typing – it reduces the user's memory load. "Minimize Memory Load" was the original name for this heuristic, and it drives much of modern user interface design.
Norman (in *The Design of Everyday Things*) makes a useful distinction between **knowledge in the head**, which is hard to get in there and still harder to recover, and **knowledge in the world**, which is far more accessible. Knowledge in the head is what we usually think of as knowledge and memory. Knowledge in the world, on the other hand, means not just documentation and button labels and signs, but also **nonverbal** features of a system that constrain our actions or remind us of what to do. Affordances, constraints, and feedback are all aspects of knowledge in the world. Command languages demand lots of knowledge in the head, while menus rely on knowledge in the world.
**Generic commands** are polymorphic, working the same way across a wide variety of data objects and applications.
Generic commands are powerful because only one command has to be learned and remembered.
Any information needed by a task should be visible or otherwise accessible in the interface for that task. The interface

shouldn’t depend on users to *remember* the email address they want to send mail to, or the product code for the product they want to buy.
This dialog box is a great example of overreliance on the user’s memory. It’s a modal dialog box, so the user can’t start following its instructions until after clicking OK. But then the instructions vanish from the screen, and the user is left to struggle to remember them. An obvious solution to this problem would be a button that simply executes the instructions directly! This message is clearly a last-minute patch for a usability problem.

## 9. Error Reporting, Diagnosis, Recovery

• Be precise; restate user’s input
– Not “Cannot open file”, but “Cannot open file named paper.doc”
• Give constructive help
– why error occurred and how to fix it
• Be polite and nonblaming
– Not “fatal error”, not “illegal”
• Hide technical details (stack trace) until requested
If you can’t prevent the error, give a good error message. A good error message should (1) be precise; (2) speak the user’s language, avoiding technical terms and details unless explicitly requested; (3) give
constructive help; and (4) be polite. The message should be worded to take as much blame as possible away from the user and heap the blame instead on the system. Save the user’s face; don’t worry about the computer’s. The computer doesn’t feel it, and in many cases it is the interface’s fault anyway for not finding a way to prevent the error in the first place.

## 10. Aesthetic and Minimalist Design

• “Less is More”
–Omit extraneous info, graphics, features
The final heuristic is a catch-all for a number of rules of good graphic design, which really boil down to one word: simplicity. Leave things out unless you have good reason to include them. Don’t put more help text on your main window than what’s really necessary. Leave out extraneous graphics. Most important, leave out unnecessary features. If a feature is never used, there’s no reason for it to complicate your interface.
Google and the Tivo remote offer great positive examples of the less-is-more philosophy.

## 10. Aesthetic and Minimalist Design

• Good graphic design
– Few, well-chosen colors and fonts
– Group with whitespace
– Align controls sensibly
• Use concise language
– Choose labels carefully
Use few, well-chosen colors. The toolbars at the top show the difference between cluttered and minimalist
color design. The first toolbar is full of many saturated colors. It’s not only gaudy and distracting, but

actually hard to scan. The second toolbar uses only a handful of colors – black, white, gray, blue, yellow. It's muted, calming, and the few colors are used to great effect to distinguish the icons. The whitespace separating icon groups helps a lot too.
The dialog box shows how cluttered and incomprehensible a layout can look when controls aren't aligned.

## Chunking the Heuristics Further

• **Meet expectations**
1. Match the real world
2. Consistency & standards
3. Help & documentation

• **User is the boss**
4. User control & freedom
5. Visibility of system status
6. Flexibility & efficiency

• **Handle errors**
7. Error prevention
8. Recognition, not recall
9. Error reporting, diagnosis, and recovery

• **Keep it simple**
10. Aesthetic & minimalist design

Since it's hard to learn 10 heuristics and hold them in your head when you're trying to design, it is useful to categorize Nielsen's heuristics still further.
**Meet expectations.** The first three heuristics concern how well the interface fits its environment, its task, and its users: speaking the user's language, keeping consistent with itself and other applications, and satisfying the expectation of help when it's needed.
**User is the boss.** The next three heuristics are related in that the interface should serve the user, rather than the other way around. Don't push the boss into the corner, keep the boss aware of things, and make the boss productive and efficient.
**Handle errors.** The next three heuristics largely concern errors, which are part and parcel of human computer interaction: prevent them as much as possible, don't rely on human memory, but when errors are unavoidable, report them properly.
Aesthetic & minimal design stays in its own category, as befits its overwhelming importance. **Keep it simple.**

## III. Implementation

### III.1 Prototyping

Protototyping is producing cheaper, less accurate renditions of your target interface. Prototyping is essential in the early iterations of a spiral design process, and it’s useful in later iterations too. We build prototypes for several reasons, all of which largely boil down to cost.

First, prototypes are much faster to build than finished implementations, so we can evaluate them sooner and get early feedback about the good and bad points of a design. Second, if we have a design decision that is hard to resolve, we can build multiple prototypes embodying the different alternatives of the decision. Third, if we discover problems in the design, a prototype can be changed more easily, fort he same reasons it could be built faster. Prototypes are more malleable. Most important, if the design flaws are serious, a prototype can be **thrown away**. It’s important not to commit strongly to design ideas in the early stages of design. Unfortunately, writing and debugging a lot of code creates a psychological sense of commitment which is hard to break. You don’t want to throw away something you’ve worked hard on, so you’re tempted to keep some of the code around, even if it really should be scrapped. The prototyping techniques we’ll see actually force you to throw the prototype away. For example, a paper mockup won’t form any part of a finished software implementation. This is a good mindset to have in early iterations, since it maximizes your creative freedom.

An essential property of a prototyping technique is its **fidelity**, which is simply how similar it is to the finished interface. Low-fidelity prototypes omit details, use cheaper materials, or use different interaction techniques. High-fidelity prototypes are very similar to the finished product. Fidelity is not just one-dimensional, however. Prototypes can be low-or high-fidelity in various different ways (Carolyn Snyder, *Paper Prototyping*, 2003). **Breadth** refers to the fraction of the feature set represented by the prototype. A prototype that is low-fidelity in breadth might be missing many features, having only enough to accomplish certain specific tasks. A word processor prototype might omit printing and spell-checking, for example.

**Depth** refers to how deeply each feature is actually implemented. Is there a backend behind the prototype that’s actually implementing the feature? Low-fidelity in depth may mean limited choices (e.g., you can’t print double-sided), canned responses (always prints the same text, not what you actually typed), or lack of robustness and error handling (crashes if the printer is offline).

A **horizontal prototype** is all breadth, and little depth; it’s basically a frontend with no backend. A **vertical prototype** is the converse: one area of the interface is implemented deeply. The question of whether to build a horizontal or vertical prototype depends on what risks you’re trying to mitigate. In user interface design, horizontal prototypes are more common, since they address usability risk. But if some aspect of the application is a risky implementation – you’re not sure if it can be implemented to meet the requirements – then you may want to build a vertical prototype to test that.

A special case lies at the intersection of a horizontal and a vertical prototype. A **scenario** shows how the frontend would look for a single concrete task. Scenarios are great for visualizing a design, but they’re hard to evaluate with users.

Two more crucial dimensions of a prototype's fidelity are, loosely, its look and its feel. **Look** is the appearance of the prototype. A hand-sketched prototype is low-fidelity in look, compared to a prototype that uses the same widget set as the finished implementation. **Feel** refers to the physical methods by which the user interacts with the prototype. A user interacts with a paper mockup by pointing at things to represent mouse clicks, and writing on the paper to represent keyboard input. This is a low-fidelity feel for a desktop application (but it may not be far off for a tablet PC application).

## III.2 Paper prototype

**Paper prototypes** are an excellent choice for early design iterations. A paper prototype is a physical mockup of the interface, mostly made of paper. It's usually hand-sketched on multiple pieces, with different pieces showing different menus, dialog boxes, or window elements.

The key difference between mere sketches and a paper prototype is **interactivity.** A paper prototype is brought to life by a design team member who simulates what the computer would do in response to the user's "clicks" and "keystrokes", by rearranging pieces, writing custom responses, and occasionally announcing some effects verbally that are too hard to show on paper. Because a paper prototype is actually interactive, you can actually user-test it: give users a task to do and watch how they do it.
A paper prototype is clearly low fidelity in both look and feel. But it can be arbitrarily high fidelity in breadth at very little cost (just sketching, which is part of design anyway). Best of all, paper prototypes can be **high-fidelity in depth** at little cost, since a human being is simulating the backend.

But why use paper? And why hand sketching rather than a clean drawing from a drawing

program? Hand-sketching on paper is faster. You can draw many sketches in the same time it would take to draw one user interface with code. For most people, hand-sketching is also faster than using a drawing program to create the sketch.
Paper is easy to change. You can even change it during user testing. If part of the prototype was a problem for one user, you can scratch it out or replace it before the next user arrives. Surprisingly, paper is more malleable than digital bits in many ways.
Hand-sketched prototypes in particular are valuable because they focus attention on the issues that matter in early design without distracting anybody with details. When you're sketching by hand, you aren't bothered with details like font, color, alignment, whitespace, etc. In a drawing program, you would be faced with all these decisions, and you might spend a lot of time on them – time that would clearly be wasted if you have to throw away this design. Hand sketching also improves the feedback you get from users. They're less likely to nitpick about details that aren't relevant at this stage. They won't complain about the color scheme if there isn't one. More important, however, a hand-sketch design seems less finished, less set in stone, and more open to suggestions and improvements. Architects have known about this phenomenon for many years. If they show clean CAD drawings to their clients in the early design discussions, the clients are less able to discuss needs and requirements that may require radical changes in the design. In fact,

many CAD tools have an option for rendering drawings with a “sketchy” look for precisely this reason.
A final advantage of paper prototyping: no special skills are required. So graphic designers, usability specialists, and even users can help create prototypes and operate them.

A paper prototype should be larger than life-size. Remember that fingers are bigger than a mouse pointer, and people usually write bigger than 12 point. So it’ll be easier to use your paper prototype if you scale it up a bit. It will also be easier to see from a distance, which is important because the prototype lies on the table, and because when you’re testing users, there may be several observers taking notes who need to see what’s going on. **Big is good.**

Don’t worry too much about color in your prototype. Use a single color. It’s simpler, and it won’t distract attention from the important issues. Needless to say, don’t use yellow. You don’t have to render every visual effect in paper. Some things are just easier to say aloud: “the basketball is spinning.” “A progress bar pops up: 20%, 50%, 75%, done.” If your design supports tooltips, you can tell your users just to point at something and ask “What’s this?”, and you’ll tell them what the tooltip would say. If you actually want to test the tooltip messages, however, you should prototype them on paper.
Figure out a good scheme for organizing the little pieces of your prototype. One approach is a three-ring binder, with different screens on different pages. Most interfaces are not sequential, however, so a linear organization may be too simple. Two-pocket folders are good for storing big pieces, and letter envelopes (with the flap open) are quite handy for keeping menus.

## How to test a paper prototype

Once you’ve built your prototype, you can put it in front of users and watch how they use it.

There are three roles for your design team to fill: The **computer** is the person responsible for making the prototype come alive. This person moves around the pieces, writes down responses, and generally does everything that a real
computer would do. In particular, the computer should *not* do anything that a real computer wouldn’t. Think mechanically, and respond mechanically. The **facilitator** is the human voice of the design team and the director of the testing session.
The facilitator explains the purpose and process of the user study, obtains the user’s informed consent, and presents the user study tasks one by one. While the user is working on a task, the facilitator tries to elicit verbal feedback from the user, particularly encouraging the user to “think aloud” by asking probing (but not leading) questions. The facilitator is responsible for keeping everybody disciplined and the user test on the right track.
Everybody else in the room (aside from the user) is an **observer**. The most important rule about being an observer is to keep your mouth shut and watch. Don’t offer help to the user, even if they’re missing something obvious. Bite your tongue, sit on your hands, and just watch. The observers are the primary note takers, since the computer and the facilitator are usually too busy with their duties.

Paper prototypes can reveal many usability problems that are important to find in early stages of design. Fixing some of these problems require large changes in design. If users don't understand the metaphor or conceptual model of the interface, for example, the entire interface may need to be scrapped.

But paper prototypes don't reveal every usability problem, because they are low-fidelity in several dimensions. Obviously, graphic design issues that depend on a high-fidelity **look** will not be discovered. Similarly, interaction issues that depend on a high-fidelity **feel** will also be missed. For example, Fitts's Law problems like buttons that are too small, too close together, or too far away will not be detected in a paper prototype. The human computer of a paper prototype rarely reflects the speed of an implemented backend, so issues of **response time –** whether feedback appears quickly enough, or whether an entire task can be completed within a certain time constraint --can't be tested either.

Paper prototypes don't help answer questions about whether subtle feedback will even be *noticed*. Will users notice that message down in the status bar, or the cursor change, or the highlight change? In the paper prototype, even the tiniest change is grossly visible, because a person's arm has to reach over the prototype and make the change. (If many changes happen at once, of course, then some of them may be overlooked even in a paper prototype, a clearly discernible. This is related to an interesting cognitive phenomenon called **change blindness**.)
There's an interesting qualitative distinction between the way users use paper prototypes and the way they use real interfaces. Experienced paper prototypers report that users are more deliberate with a paper prototype, apparently thinking more carefully about their actions. This may be partly due to the simulated computer's slow response; it may also be partly a social response, conscientiously trying to save the person doing the simulating from a lot of tedious and unnecessary paper shuffling. More deliberate users make fewer mistakes, which is bad, because you want to see the mistakes. Users are also less likely to randomly explore a paper prototype.
These drawbacks don't invalidate paper prototyping as a technique, but you should be aware of them. Several studies have shown that low-fidelity prototypes identify substantially the same usability problems as high-fidelity prototypes

## III.3. Some specific guidelines for graphic design

These guidelines are drawn from the excellent book *Designing Visual Interfaces* by Kevin Mullet and Darrell Sano (Prentice-Hall, 1995). Mullet & Sano's book predates the Web, but the principles it describes are timeless and relevant to any visual medium.

### Balance & Symmetry

• Choose an axis (usually vertical)
• Distribute elements equally around the axis
– Equalize both mass and extent

Balance and symmetry are valuable tools in a designer's toolkit. In graphic design, symmetry rarely means exact, mirror-image equivalence. Instead, what we mean by symmetry is more like balance: is there the same amount of stuff on each side of the axis of symmetry. We measure “stuff” by both mass (quantity of nonwhite pixels) and extent (area covered by those pixels); both mass and extent should be balanced.

## Symmetry Example

An easy way to achieve balance is to simply center the elements of your display. That automatically achieves balance around a vertical axis. If you look at Google’s home page, you’ll see this kind of approach in action. In fact, only one element of the Google home page breaks this symmetry: the stack of links for Advanced Search, Preferences, and Language Tools on the right. This slight irregularity actually helps emphasize these links slightly.
Symmetry is a kind of **simplicity**; asymmetry creates a **contrast**. Use that contrast wisely.

## Alignment

• Align labels on
left or right
• Align controls on
left *and* right
– Expand as needed
• Align text baselines
Finally, simplify your designs by aligning elements horizontally and vertically. Alignment contributes to the
simplicity of a design. Fewer alignment positions means a simpler design.
**Labels** (e.g. “Wait” and “Retry after”). There are two schools of thought about label alignment: one school
says that the left edges of labels should be aligned, and the other school says that their right edges (i.e., the colon following each label) should be aligned. Both approaches work, and experimental studies haven’t found any significant differences between them. Both approaches also fail when long labels and short labels are used in the same display. You’ll get best results if you can make all your labels about the same size, or else break long labels into multiple lines.
**Controls** (e.g., text fields, combo boxes, checkboxes). A column of controls should be aligned on both the left and the right. Sometimes this seems unreasonable -- should a short date field be expanded to the same length as a filename? It doesn’t hurt the date to be larger than necessary, except perhaps for reducing its perceived affordance for receiving a date. You can also solve these kinds of problems by rearranging the display, moving the date elsewhere, although be careful of disrupting your design’s functional grouping or the expectations of your user.
So far we’ve only discussed left-to-right alignment. Vertically, you should ensure that labels and controls on the same row share the same **text baseline**. Java Swing components are designed so that text baselines are aligned if the components are centered vertically with respect to each other, but not if the components’ tops or bottoms are aligned. Java AWT components are virtually impossible to align on their baselines.

## Grids Are Effective

A **grid** is one effective way to achieve both alignment and balance, nearly automatically. Notice the four-column grid used in this dialog box (excluding the labels on the left).

## Color Guidelines

• Remember limitations of human vision

– Color blindness, red-on-blue, small blue details
• Use few colors
• Avoid saturated colors
• Be consistent and match expectations
We’ve already talked a lot about the limits of human color vision. In general, colors should be used sparingly. An interface with many colors appears more complex, more cluttered, and more distracting. Use only a handful of colors.
Background colors should establish a good contrast with the foreground. White is a good choice, since it provides the most contrast; but it also produces bright displays, since our computer displays emit light rather than reflecting it. Pale (desaturated) yellow and very light gray are also good background colors.
In general, avoid strongly saturated colors. Saturated colors can cause visual fatigue because the eye must keep refocusing on different wavelengths. They also tend to saturate the viewer’s receptors (hence the name). One study found that air traffic controllers who viewed strongly saturated green text on their ATC interfaces for many hours had trouble seeing pink or red (the other end of the red/green color channel) for up to 15 minutes after their shift was over.
Use less saturated versions instead, pushing them towards gray.
To sharpen contrasts, you can use opponent colors: red/green, blue/yellow. But keep color blind users in mind; hue should not be the *only* way you establish the contrast. Both color-blind and color normal users will see the contrast better if you vary both hue *and* value.
Finally, match expectations. Red generally means stop, warning, error, or hot. Green conventionally means go, or OK. Yellow means caution, or slow.

_ _Perfection is achieved not when there is
nothing more to add, but when there is
nothing left to take away._ (Antoine de St-
Exupery)
_ _Simplicity does not mean the absence of any
decor_ It only means that the decor should
belong intimately to the design proper, and
that anything foreign to it should be taken
away._ (Paul Jacques Grillo)
_ _Keep it simple, stupid._ (KISS)
_ _Less is more._
_ _When in doubt, leave it out._
Okay, we’ll shout some slogans at you now. You’ve probably heard some of these before. What you should take from these slogans is that designing for simplicity is a process of *elimination*, not accretion. Simplicity is in constant tension with task analysis, information preconditions, and other design guidelines, which might otherwise encourage you to pile more and more elements into a design, “just in case.” Simplicity forces you to have a good reason for everything you add, and to take away anything that can’t survive hard scrutiny.

_ Remove inessential elements
Here are three ways to make a design simpler.
**Reduction** means that you eliminate whatever isn’t necessary. This technique has three

steps: (1) decide what essentially needs to be conveyed by the design; (2) critically examine every element (label, control, color, font, line weight) to decide whether it serves an essential purpose; (3) remove it if it isn't essential. Even if it seems essential, try removing it anyway, to see if the design falls apart.
**Icons** demonstrate the principle of reduction well. A photograph of a pair of scissors can't possibly work as a 32x32 pixel icon; instead, it has to be a carefully-drawn picture which includes the bare minimum of details that are essential to scissors: two lines for the blades, two loops for the handles.
We've already discussed the minimalism of Google and the Tivo remote . Here, the question is about **functionality**. Both Google and Tivo aggressively removed functions from their primary interfaces.

_ Use a regular pattern
_ Limit inessential variation among
elements
For the essential elements that remain, consider how you can minimize the unnecessary differences between them with **regularity**. Use the same font, color, line width, dimensions, orientation for multiple elements. Irregularities in your design will be magnified in the user's eyes and assigned meaning and significance. Conversely, if your design is mostly regular, the elements that you do want to highlight will stand out better.
PowerPoint's Text Layouts menu shows both reduction (minimalist icons representing each layout) and regularity. Titles and bullet lists are shown the same way.

_ Combine elements for leverage
_ Find a way for one element to play multiple
roles
title bar
scrollbar thumb
help prompt
Finally, you can **combine elements**, making them serve multiple roles in the design. The desktop interface has a number of good examples of this kind of design. For example, the "thumb" in a scroll bar actually serves three roles. It affords dragging, indicates the position of the scroll window relative to the entire document, and indicates the fraction of the document displayed in the scroll window. Similarly, a window's title bar plays several roles: label, dragging handle, window activation indicator, and location for window control buttons.

_ Contrast encodes information along
visual dimensions
_____ __ _____ _ ___ _____ _______ ____
**Contrast** refers to perceivable differences along a visual dimension, such as size or color. Contrast is the irregularity in a design that communicates information or makes elements stand out. Simplicity says we should eliminate **unimportant** differences. Once we've decided that a difference is important, however, we should choose the dimension and degree of contrast in such a way that the difference is salient, easily perceptible, and appropriate to the task.

**Title**
**Heading**
This is body text. It’s smaller than the heading, lighter in weight, and longer
in line length. We’ve also changed its shape to a serif font, because serifs
make small text easier to read. Redundant encoding produces an effective
contrast that makes it easy to scan the headings and distinguish headings from
body text.
Titles, headings, body text, figure captions, and footnotes show how contrast is used to
make articles easier to read. You can do this yourself when you’re writing papers and
documentation. Does this mean contrast should be maximized by using lots of different
fonts like Gothic and Bookman? No, for two reasons – contrast must be balanced
against simplicity, and text shape variations aren’t the best way to establish contrast.

_ Use white space for grouping, instead of
lines
_ Use margins to draw eye around design
_ Integrate figure and ground
_Object should be scaled proportionally to
its background
_ Don_t crowd controls together
_Crowding creates spatial tension and
inhibits scanning
White space plays an essential role in composition. Screen real estate is at a premium in
many graphical user interfaces, so it’s a constant struggle to balance the need for white
space against a desire to pack information and controls into a display. But insufficient
white space can have serious side-effects, making a display more painful to look at and
much slower to scan.
Put **margins** around all your content. Labels and controls that pack tightly against the
edge of a window are much slower to scan. When an object is surrounded by white
space, keep a sense of proportion between the object (the **figure**) and its surroundings
(**ground**). Don’t crowd controls together, even if you’re grouping the controls.
Crowding inhibits scanning, and produces distracting effects when two lines (such as the
edges of text fields) are too close. Many UI toolkits unfortunately encourage this
crowding by packing controls tightly together by default, but Java Swing (at least) lets
you add empty margins to your controls that give them a chance to breathe.

## Crowded Dialog
Here’s an example of an overcrowded dialog. The dialog has no **margins** around the edges;
the controls are **tightly packed** together; and **lines are used for grouping** where white
space would be more appropriate. Screen real estate isn’t terribly precious in a transient
dialog box.
The crowding leads to some bad perceptual effects. Lines appearing too close together – such as
the bottom of the Spacing text field and the group line that surround it – blend
together into a thicker, darker line, making a wart in the design. A few pixels of white

space between the lines would completely eliminate this problem.

## Using White Space to Set Off Labels

Images found in:
K. Mullet and D. Sano, "Designing visual interfaces: Communication-oriented techniques,“ Sunsoft Press, Prentice Hall (1995), p. 96.
A particularly effective use of white space is to put labels in the left margin, where the white space sets off and highlights them.
For the same reason, you should put labels to the left of controls, rather than above.

_ Gestalt principles explain how eye creates a whole
(*gestalt*) from parts
______ ________ ______
______ ____ _ ____
The power of white space for grouping derives from the Gestalt principle of proximity. These principles, discovered in the 1920’s by the Gestalt school of psychologists, describe how early visual processing groups elements in the visual field into larger wholes. Here are the six principles identified by the Gestalt psychologists:
**Proximity**. Elements that are closer to each other are more likely to be grouped together. You see four vertical columns of circles, because the circles are closer vertically than they are horizontally.
**Similarity.** Elements with similar attributes are more likely to be grouped. You see four *rows* of circles in the Similarity example, because the circles are more alike horizontally than they are vertically.
**Continuity.** The eye expects to see a contour as a continuous object. You primarily perceive the Continuity example above as two crossing lines, rather than as four lines meeting at a point, or two right angles sharing a vertex.
**Closure.** The eye tends to perceive complete, closed figures, even when lines are missing. We see a triangle in the center of the Closure example, even though its edges aren’t complete.
**Area.** When two elements overlap, the smaller one will be interpreted as a figure in front of the larger ground. So we tend to perceive the Area example as a small square in front of a large square, rather than a large square with a hole cut in it.
**Symmetry.** The eye prefers explanations with greater symmetry. So the Symmetry example is perceived as two overlapping squares, rather than three separate polygons.

1

## III.4. **Computer Prototype**

• Interactive software simulation
• High-fidelity in look & feel
• Low-fidelity in depth
–Paper prototype had a human simulating
the backend; computer prototype doesn’t
–Computer prototype is typically **horizontal**:
covers most features, but no backend

## Paper Prototyping is Not Enough

• Low fidelity in:
– Look
– Feel
–Dynamics
–Response time
–Context
• Users can't try it without a human to
simulate computer

We now turn to prototyping. Paper prototyping is neat, but it's not enough. We discussed some of these
drawbacks in the paper prototyping.

First, paper prototypes are low-fi in **look**. It's sometimes hard for users to recognize widgets that you've
hand-drawn, or labels that you've hastily scribbled. A paper prototype won't tell you what will actually fit on the screen, since your handwritten font and larger-than-life posterboard aren't realistic simulations. A paper prototype can't easily simulate some important characteristics of your interface's look – for example, does the selection highlighting have enough **contrast**, using visual variables that are **selective**, does it float above the rest of the design in its own layer? You have to make decisions about graphic design at some point, and you want to iterate those decisions just like other aspects of usability. So we need another prototyping method.

Paper prototypes are also low-fi in **feel**. Finger & pen doesn't behave like mouse & keyboard (e.g., the drag & drop issue mentioned earlier). Even pen-based computers don't really feel like pen & paper. You can't test Fitts's Law issues, like whether a button is big enough to click on.

Paper is low-fi in **dynamic feedback**. You don't get any animation, like spinning globes, moving progress
bars, etc. You can't test mouse-over feedback, as we mentioned earlier.

It's also low-fi in **response time**, because the human computer is far slower than the real computer. So you can't measure how long it takes users to do a task with your interface.

Finally, paper is low-fi with respect to **context of use**. A paper prototype can only really be used on a
tabletop, in office-like environment. Users can't take it to grocery store, subway, aircraft carrier deck, or
wherever the target interface will actually be used. If it's a handheld application, the ergonomics are all wrong; you aren't holding it in your hand, and it's not the right weight.

So at some point we have to depart from paper and move our prototypes into software. A typical computer prototype is a **horizontal** prototype. It's high-fi in look and feel, but low-fi in depth – there's no backend behind it. Where a human being simulating a paper prototype can generate new content on the fly in response to unexpected user actions, a computer prototype cannot.

## What You Can Learn From Computer Prototypes

• Everything you learn from a paper prototype, plus:
• Screen layout
– Is it clear, overwhelming, distracting, complicated?

– Can users find important elements?
• Colors, fonts, icons, other elements
– Well-chosen?
• Interactive feedback
– Do users notice & respond to status bar messages, cursor
changes, other feedback
• Fitts’s Law issues
– Controls big enough? Too close together? Scrolling list is too
long?
Computer prototypes help us get a handle on the graphic design and dynamic feedback of the interface.

## Why Use Prototyping Tools?

• Faster than coding
• No debugging
• Easier to change or throw away
• Don’t let Java do your graphic design
One way to build a computer prototype is just to program it directly in an implementation language, like Java or C++, using a user interface toolkit, like Swing or MFC. If you don’t hook in a backend, or use stubs instead of your real backend, then you’ve got a horizontal prototype.
But it’s often better to use a **prototyping tool** instead. Building an interface with a tool is usually faster than direct coding, and there’s no code to debug. It’s easier to change it, or even throw it away if your design turns out to be wrong. Recall Cooper’s concerns about prototyping: your computer prototype may become so elaborate and precious that it *becomes* your final implementation, even though (from a software engineering point of view) it might be sloppily designed and unmaintainable.
Also, when you go directly from paper prototype to code, there’s a tendency to let your UI toolkit
handle all the graphic design for you. That’s a mistake. For example, Java has layout managers that automatically arrange the components of an interface. Layout managers are powerful tools, but they produce horrible interfaces when casually or lazily used. A prototyping tool will help you envision your interface and get its graphic design right first, so that later when you move to code, you know what you’re trying to persuade the layout manager to produce.
Even with a prototyping tool, computer prototypes can still be a tremendous amount of work. When drag & drop was being considered for Microsoft Excel, a couple of Microsoft summer interns were assigned to develop a prototype of the feature using Visual Basic. They found that they had to implement a substantial amount of basic spreadsheet functionality just to test drag & drop. It took two interns their entire summer to build the prototype that proved that drag & drop was useful. Actually adding the feature to Excel took a staff programmer only a week. This isn’t a fair comparison, of course – maybe six intern-months was a cost worth paying to mitigate the risk of one fulltimer-week, and the interns certainly learned a lot. But building a computer prototype can be a slippery slope, so don’t let it suck you in too deeply. Focus on what you want to test, i.e., the design risk you need to mitigate, and only prototype that.

## Prototyping Techniques

• Storyboard

–Sequence of painted screenshots
connected by hyperlinks (“hotspots”)
• Form builder
–Real windows assembled from a palette of
widgets (buttons, text fields, labels, etc.)
There are two major techniques for building a computer prototype.
A **storyboard** is a sequence (a graph, really) of fixed screens. Each screen has one or more **hotspots** that you can click on to jump to another screen. Sometimes the transitions between screens also involve some animation in order to show a dynamic effect, like mouse-over feedback or drag-drop feedback.
A **form builder** is a tool for drawing real, working interfaces by dragging widgets from a palette and positioning them on a window.

## Storyboarding Tools

• HTML
– image maps
• Flash/Director
– animation + actions
• PowerPoint
– images + links + animation
• All these tools have scripting languages, too
– Help orchestrate the transitions
• For high fidelity look, take screenshots of
widgets from a form builder
Here are some tools commonly used for storyboarding.
A **PowerPoint** presentation is just a slide show. Each slide shows a fixed screenshot, which you can draw in a paint program and import, or which you can draw directly in PowerPoint. A PowerPoint storyboard doesn’t have to be linear slide show. You can create hyperlinks that jump to any slide in the presentation.
Macromedia **Flash** (formerly Director) is a tool for constructing multimedia interfaces. It’s particularly useful for prototyping interfaces with rich animated feedback.
**HTML** is also useful for storyboarding. Each screen is an imagemap. Macromedia Dreamweaver makes it easy to build HTML imagemaps.
All these tools have scripting languages – PowerPoint has Basic, Flash/Director has a language called Lingo, and HTML has Javascript – so you can write some code to orchestrate transitions, if need be.
If your storyboards need standard widgets like buttons or text boxes, you can create some widgets in a form builder and take static screenshots of them for your storyboard.

## Pros & Cons of Storyboarding

• Pros
–You can draw anything
• Cons
–No text entry
–Widgets aren’t active
– “Hunt for the hotspot”

The big advantage of storyboarding is similar to the advantage of paper: you can draw anything on a storyboard. That frees your creativity in ways that a form builder can't, with its fixed palette of widgets.
The disadvantages come from the storyboard's static nature. All you can do is click, not enter text.
You can still have text boxes, but clicking on a text box might make its content magically appear, without the user needing to type anything. Similarly, scrollbars, list boxes, and buttons are just pictures, not active widgets. Watching a real user in front of a storyboard often devolves into a game of **"hunt for the hotspot",** like children's software where the only point is to find things on the screen to click on and see what they do. The hunt-for-the-hotspot effect means that storyboards are largely useless for user testing, unlike paper prototypes. In general, horizontal computer prototypes are better evaluated with other techniques, like heuristic evaluation.

## Form Builders

• HTML pages and forms
– Natural if you're building a web application
– May have low-fidelity look otherwise
• Java GUI builders
– Sun NetBeans
– Eclipse Visual Editor
– Borland JBuilder
• Other GUI builders
– Visual Basic, .NET Windows Forms
– Mac Interface Builder
– Qt Designer
• Tips
– Use absolute positioning for now
Here are some form builder tools.
**HTML** is a natural tool to use if you're building a web application. You can compose static HTML
pages simulating the dynamic responses of your web interface. Although the responses are canned, your prototype is still better than a storyboard, because its screens are more active than mere screenshots: the user can actually type into form fields, scroll through long displays, and see mouseover feedback. Even if you're building a desktop or handheld application, HTML may still be useful. For example, you can mix static screenshots of some parts of your UI with HTML form widgets (buttons, list boxes, etc) representing the widget parts. It may be hard to persuade HTML to render a desktop interface in a high-fidelity way, however.
**Visual Basic** is the classic form builder. Many custom commercial applications are built entirely with Visual Basic.
There are several form builders for Java. Sun NetBeans and Borland JBuilder (Personal Edition) can be downloaded free.
Be careful when you're using a form builder for prototyping to avoid layout managers when you're doing your initial graphic designs. Instead, use **absolute positioning**, so you can put each component where you want it to go. Java GUI builders may need to be told not to use a layout manager.

## Pros & Cons of Form Builders

• Pros
–Actual controls, not just pictures of them
–Can hook in some backend if you need it
• But then you won't want to throw it away
• Cons
– Limits thinking to standard widgets
–Useless for rich graphical interfaces

Unlike storyboards, form builders use actual working widgets, not just static pictures. So the widgets look the same as they will in the final implementation (assuming you're using a compatible form builder – a prototype in Visual Basic may not look like a final implementation in Java).

Also, since form builders usually have an implementation language underneath them – which may even be the same implementation language that you'll eventually use for your final interface – you can also hook in as much or as little backend as you want.

On the down side, form builders give you a fixed palette of standard widgets, which limits your creativity as a designer, and which makes form builders largely useless for prototyping rich graphical interfaces, e.g., a circuit-drawing editor. Form builders are great for the menus and widgets that surround a graphical interface, but can't simulate the "insides" of the application window.

## Toolkits

• User interface toolkit consists of:
– Components (view hierarchy)
– Stroke drawing
– Pixel model
– Input handling
– Widgets

By now, we've looked at all the basic pieces of a user interface toolkit: widgets, view hierarchy, stroke drawing, and input handling. 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*).

## Widgets

• Reusable user interface components
– Also called controls, interactors, gizmos, gadgets
• Examples
– Buttons, checkboxes, radio buttons
– List boxes, combo boxes, drop-downs
– Menus, toolbars
– Scrollbars, splitters, zoomers
– One-line text, multiline text, rich text
– Trees, tables
– Simple dialogs

**Widgets** are the last part of user interface toolkits we'll look at. 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.

## Widget Pros and Cons

• Advantages
–Reuse of development effort
• Coding, testing, debugging, maintenance
• Iteration and evaluation
–External consistency
• 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, actually. 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 behavior 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 a clunkier, 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 aren't suited.

## Cross-Platform Widgets: AWT Approach

• AWT, HTML
–Use native widgets, but only those
common to all platforms
• Tree widget available on MS Win but not X, so
AWT doesn't provide it
–Very consistent with other platform apps,
because it uses the same code
java.awt.List MSWin List
peer

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, text boxes.

### Cross-Platform Widgets: Swing approach

• Swing, Amulet
–Reimplement all widgets
–Not constrained by least common
denominator
–Consistent behavior for application across
platforms
One reason NOT to reuse the native widgets is so 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 reimplementing the
widget set using its default ("Metal") look and feel. This essentially creates a Java "platform", independent of and distinct from the native platform.

## IV. Evaluation

### IV.1 Heuristic evaluation

One application of these 10 heuristics is a usability inspection process called **heuristic evaluation**. Heuristic evaluation was originally invented by Jakob Nielsen, and you can learn more about it on his web site. Nielsen has done a number of studies to evaluate the effectiveness of heuristic evaluation. Those studies have shown that heuristic evaluation's cost-benefit ratio is quite favorable; the cost per problem of finding usability problems in an interface is generally cheaper than alternative methods.

Heuristic evaluation is an inspection method. It is performed by a usability expert – someone who knows and
understands the heuristics we've just discussed, and has used and thought about lots of interfaces. The basic steps are simple: the evaluator inspects the user interface thoroughly, judges

the interface on the basis of the heuristics we've just discussed, and makes a list of the usability problems found – the ways in which individual elements of the interface deviate from the usability heuristics.
Let's look at heuristic evaluation from the evaluator's perspective. That's the role you'll be adopting in the next
homework, when you'll serve as heuristic evaluators for each others' computer prototypes. Here are some tips for doing a good heuristic evaluation. First, your evaluation should be grounded in known usability guidelines. You should justify each problem you list by appealing to a heuristic, and explaining how the heuristic is violated. This practice helps remove most of the (inevitable) subjectivity involved in inspections: You can't just say "that's an ugly yellow color." (If it's really yucky, you *should* pass that subjective opinion back to the design team, but you'll be forced to identify it as subjective if you can't find a heuristic to justify it.)
List every problem you find. If a button has several problems with it – inconsistent placement, bad color combination, confusing label – then each of those problems should be listed separately. Some of the problems may be more severe than others, and some may be easier to fix than others. It's best to get all the problems on the table in order to make these tradeoffs.
Inspect the interface at least twice. The first time you'll get an overview and a feel for the system. The second time, you
should focus carefully on individual elements of the interface, one at a time. Finally, although you have to justify every problem with a guideline, you don't have to limit yourself to the Nielsen 10.We've seen a number of specific usability principles that can serve equally well: affordances, visibility, Fitts's Law, perceptual fusion, color guidelines, graphic design rules are a few. The Nielsen 10 are helpful in that they're a short list that covers a wide spectrum of usability problems. For each element of the interface, you can quickly look down the Nielsen list to guide your thinking.

Heuristic evaluation is only one way to evaluate a user interface. User testing --watching users interact with the interface – is another. User testing is really the gold standard for usability evaluation. An interface has usability problems only if real users have real problems with it, and the only sure way to know is to watch and see.

A key reason why heuristic evaluation is different is that an evaluator is not a typical user either! They may be closer to a typical user, however, in the sense that they don't know the system model to the same degree that its designers do. And a good heuristic evaluator tries to think like a typical user. But an evaluator knows too much about user interfaces, and too much about usability, to respond like a typical user.
So, heuristic evaluation is not the same as user testing. A useful analogy from software engineering is the
difference between code inspection and testing. Heuristic evaluation may find problems that user testing would miss (unless the user testing was extremely expensive and comprehensive). For example, heuristic evaluators can easily detect problems like inconsistent font styles, e.g. a sans-serif font in one part of the interface, and a serif font in another. Adapting to the inconsistency slows down users slightly, but only extensive user testing would reveal it. Similarly, a heuristic evaluation might notice that buttons along the edge of the screen are not taking proper advantage

of the Fitts’s Law benefits of the screen boundaries, but this problem might be hard to detect in user testing.

Now let’s look at heuristic evaluation from the designer’s perspective. Assuming I’ve decided to use this

technique to evaluate my interface, how do I get the most mileage out of it? First, use more than one evaluator. Studies of heuristic evaluation have shown that no single evaluator can find all the usability problems, and some of the hardest usability problems are found by evaluators who find few problems overall (Nielsen, “Finding usability problems through heuristic evaluation”, CHI ’92). The more evaluators the better, but with diminishing returns: each additional evaluator finds fewer new problems. The sweet spot for cost-benefit, recommended by Nielsen based on his studies, is 3-5 evaluators.

One way to get the most out of heuristic evaluation is to alternate it with user testing in subsequent trips around the iterative design cycle. Each method finds different problems in an interface, and heuristic evaluation is almost always cheaper than user testing. Heuristic evaluation is particularly useful in the tight inner loops of the iterative design cycle, when prototypes are raw and low-fidelity, and cheap, fast iteration is a must.

In heuristic evaluation, it’s OK to help the evaluator when they get stuck in a confusing interface. As long as the usability problems that led to the confusion have already been noted, an observer can help the evaluator get unstuck and proceed with evaluating the rest of the interface, saving valuable time. In user testing, this kind of personal help is totally inappropriate, because you want to see how a user would really behave if confronted with the interface in the real world, without the designer of the system present to guide them. In a user test, when the user gets stuck and can’t figure out how to complete a task, you usually have to abandon the task and move on to another one.

Here’s a formal process for performing heuristic evaluation.

The training meeting brings together the design team with all the evaluators, and brings the evaluators up to speed on what they need to know about the application, its domain, its target users, and scenarios of use. The evaluators then go off and evaluate the interface separately. They may work alone, writing down their

own observations, or they may be observed by a member of the design team, who records their observations (and helps them through difficult parts of the interface, as we discussed earlier). In this stage, the evaluators focus just on generating problems, not on how important they are or how to solve them.

Next, all the problems found by all the evaluators are compiled into a single list, and the evaluators rate the severity of each problem. We’ll see one possible severity scale in the next slide. Evaluators can assign severity ratings either independently or in a meeting together. Since studies have found that severity ratings from independent evaluators tend to have a large variance, it’s best to collect severity ratings from several evaluators and take the mean to get a better estimate.

Finally, the design team and the evaluators meet again to discuss the results. This meeting offers a forum for brainstorming possible solutions, focusing on the most severe (highest priority) usability problems. When you do heuristic evaluations in this class, I suggest you follow this ordering as well: first focus on generating as many usability problems as you can, then rank their severity, and then think about solutions.

Here's one scale you can use to judge the severity of usability problems found by heuristic evaluation. It helps to think about the factors that contribute to the severity of a problem: its **frequency** of occurrence (common or rare); its **impact** on users (easy or hard to overcome), and its **persistence** (does it need to be overcome once or repeatedly). A problem that scores highly on several contributing factors should be rated more severe than another problem that isn't so common, hard to overcome, or persistent.

A final advantage of heuristic evaluation that's worth noting: heuristic evaluation can be applied to interfaces in varying states of readiness, including unstable prototypes, paper prototypes, and even just sketches. When you're evaluating an incomplete interface, however, you should be aware of one pitfall. When you're just inspecting a sketch, you're less likely to notice missing elements, like buttons or features essential to proceeding in a task. If you were actually *interacting* with an active prototype, essential missing pieces rear up as obstacles that prevent you from proceeding. With sketches, nothing prevents you from going on: you just turn the page. So you have to look harder for missing elements when you're heuristically evaluating static sketches or screenshots.

Here are some tips on writing good heuristic evaluations. First, remember your audience: you're trying to communicate to developers. Don't expect them to be experts on usability, and keep in mind that they have some ego investment in the user interface. Don't be unnecessarily harsh.

Although the primary purpose of heuristic evaluation is to identify problems, positive comments can be valuable too. If some part of the design is *good* for usability reasons, you want to make sure that aspect doesn't disappear in future iterations.

## IV.2 User testing

We'll talk about **user testing**: putting an interface in front of real users. There are several kinds of user testing, but all of them by definition involve human beings, who are thinking, breathing individuals with rights and feelings. When we enlist the assistance of real people in interface testing, we take on some special responsibilities. So first we'll talk about the **ethics** of user testing, which apply regardless of what kind of user test you're doing.

We will focus on one particular kind of user test: **formative evaluation**, which is a user test performed during iterative design with the goal of finding usability problems to fix on the next design iteration.
User testing is more expensive and time-consuming than heuristic evaluation, but produces better results.

The purpose of formative evaluation is finding usability problems in order to fix them in the next design iteration. Formative evaluation doesn't need a full working implementation, but can be done on a variety of prototypes. This kind of user test is usually done in an environment that's under your control, like an office or a usability lab. You also choose the tasks given to users, which are generally realistic (drawn from task analysis, which is based on

observation) but nevertheless fake. The results of formative evaluation are largely **qualitative observations**, usually a list of usability problems. Note that Prototype Testing Day was not the best way to do formative evaluation: first, because your
classmates are probably not representative of your target user population; and second, because we had artificial time constraints that raised the pressure on users and experimenters, prevented using substantial tasks, and didn't allow for much debriefing or discussion after the test. Better user tests would be use appropriate users and be more relaxed, which we'll see later in the lecture.
A key problem with formative evaluation is that you have to control too much. Running a test in a lab environment on tasks of your invention may not tell you enough about how well your interface will work in a real context on real tasks. A **field study** can answer these questions, by actually deploying a working implementation to real users, and then going out to the users' real environment and observing how they use it. We won't say much about field studies in this class.
A third kind of user test is a **controlled experiment**, whose goal is to test a quantifiable hypothesis about one or more interfaces. Controlled experiments happen under carefully controlled conditions using carefully-designed tasks – often more carefully chosen than formative evaluation tasks. Hypotheses can only be tested by quantitative measurements of usability, like time elapsed, number of errors, or subjective satisfaction.

Ethics of User Testing

Let's start by talking about some issues that are relevant to all kinds of user testing: ethics. Human subjects have been horribly abused in the name of science over the past century.

Experiments involving medical treatments or electric shocks are one thing. But what's so dangerous

about a computer interface? Hopefully, nothing – most user testing has minimal physical or psychological risk to the user. But user testing does put psychological pressure on the user. The user sits in the spotlight, asked to perform unfamiliar tasks on an unfamiliar (and possibly bad!) interface, in front of an audience of strangers (at least one experimenter, possibly a roomful of observers, and possibly a video camera). It's natural to feel some performance anxiety, or stage fright. "Am I doing it right? Do these people think I'm dumb for not getting it?" A user may regard the test as a psychology test, or more to the point, an IQ test. They may be worried about getting a bad score. Their self-esteem may suffer, particularly if they blame problems they have on themselves, rather than on the user interface.
A programmer with an ironclad ego may scoff at such concerns, but these pressures are real. Jared Spool, an influential usability consultant, tells a story about the time he saw a user cry

during a user test. It came about from an accumulation of mistakes on the part of the experimenters:

1. the originally-scheduled user didn't show up, so they just pulled an employee out of the hallway todo the test;

2. it happened to be her first day on the job;

3. they didn't tell her what the session was about;

4. she not only knew nothing about the interface to be tested (which is fine and good), but also nothing about the domain – she wasn't in the target user population at all;

5. the observers in the room hadn't been told how to behave (i.e., shut up);

6. one of those observers was her boss;

7. the tasks hadn't been pilot tested, and the first one was actually impossible. When she started struggling with the first task, everybody in the room realized how stupid the task

was, and burst out laughing – at their own stupidity, not hers. But she thought they were laughing at her, and she burst into tears. (story from Carolyn Snyder, Paper Prototyping)

Treat the User With Respect

The basic rule for user testing ethics is **respect** for the user as a intelligent person with free will and feelings. We can show respect for the user in 5 ways:

1. Respecting their **time** by not wasting it. Prepare as much as you can in advance, and don't make the user jump through hoops that you aren't actually testing. Don't make them install the software or load the test files, for example, unless your test is supposed to measure the usability of the installation process or file-loading process.

2. Do everything you can to make the user **comfortable**, in order to offset the psychological pressures of a user test.

3. Give the user as much **information** about the test as they need or want to know, as long as the information doesn't bias the test. Don't hide things from them unnecessarily.

4. Preserve the user's **privacy** to the maximum degree. Don't report their performance on the user test in a way that allows the user to be personally identified.

5. The user is always in **control**, not in the sense that they're running the user test and deciding what to do next, but in the sense that the final decision of whether or not to participate remains theirs, throughout the experiment. Just because they've signed a consent form, or sat down in the

room with you, doesn't mean that they've committed to the entire test. A user has the right to give up the test and leave at any time, no matter how inconvenient it may be for you.

Before a Test

Let's look at what you should do before, during, and after a user test to ensure that you're treating

users with respect. Long before your first user shows up, you should **pilot-test** your entire test: all questionnaires, briefings, tutorials, and tasks. Pilot testing means you get a few people (usually your colleagues) to act as users in a full-dress rehearsal of the user test. Pilot testing is essential for simplifying and working the bugs out of your test materials and procedures. It gives you a chance to eliminate wasted time, streamline parts of the test, fix confusing briefings or training materials, and discover impossible or pointless tasks. It also gives you a chance to practice your role as an experimenter. Pilot testing is essential for every user test.

When a user shows up, you should brief them first, introducing the purpose of the application and the purpose of the test. To make the user comfortable, you should also say the following things (in some form):

- "Keep in mind that we're testing the computer system. We're not testing you." (comfort)
- "The system is likely to have problems in it that make it hard to use. We need your help to find those problems." (comfort)
- "Your test results will be completely confidential." (privacy)
- "You can stop the test and leave at any time." (control)

You should also inform the user if the test will be audiotaped, videotaped, or watched by hidden observers. Any observers actually present in the room should be introduced to the user. At the end of the briefing, you should ask "Do you have any questions I can answer before we begin?" Try to answer any questions the user has. Sometimes a user will ask a question that may bias the experiment: for example, "what does that button do?" You should explain why you can't answer that question, and promise to answer it after the test is over.

During the Test

During the test, arrange the testing environment to make the user comfortable. Keep the atmosphere calm, relaxed, and free of distractions. (We failed on all three counts at Prototype Testing Day!) If the testing session is long, give the user bathroom, water, or coffee breaks, or just a chance to standup and stretch.

Don't act disappointed when the user runs into difficulty, because the user will feel it as disappointment in their performance, not in the user interface. Don't overwhelm the user with work. Give them only one task at a time. Ideally, the first task should be an easy warm up task, to give the user an early success experience. That will bolster their courage (and yours) to get them through the harder tasks that will discover more usability problems.

Answer the user's questions as long as they don't bias the test. Keep the user in control. If they get tired of a task, let them give up on it and go on to another. If they want to quit the test, pay them and let them go.

## After the Test

After the test is over, thank the user for their help and tell them how they've helped. It's easy to be

open with information at this point, so do so. Later, if you disseminate data from the user test, don't publish it in a way that allows users to be individually identified. Certainly, avoid using their names.
If you collected video or audio records of the user test, don't show them outside your development group without explicit written permission from the user.

## Formative Evaluation

We've seen some ethical rules that apply to running any kind of user test. Now let's look in

particular at how to do **formative evaluation**. You've already done one formative evaluation test already, using your paper prototypes. So you know the basic steps already: (1) find some representative users; (2) give each user some representative tasks; and (3) watch the user do the tasks.

## Roles in Formative Evaluation

There are three roles. The user's primary role is to perform the tasks using the interface. While the user is actually doing this, however, they should also be trying to **think aloud**: verbalizing what they're thinking as they use the interface. Encourage the user to say things like "OK, now I'm looking for the place to set the font size, usually it's on the toolbar, nope, hmm, maybe the Format menu…" Thinking aloud gives you (the observer) a window into their thought processes, so you can understand what they're trying to do and what they expect.

Unfortunately, thinking aloud feels strange for most people. It can alter the user's behavior, making the user more deliberate and careful, and sometimes disrupting their concentration. Conversely, when a task gets hard and the user gets absorbed in it, they may go mute, forgetting to think aloud. One of the facilitator's roles is to prod the user into thinking aloud.
One solution to the problems of think-aloud is **constructive interaction**, in which two users work on the tasks together (using a single computer). Two users are more likely to converse naturally with each other, explaining how they think it works and what they're thinking about trying. Constructive interaction requires twice as many users, however, and may be adversely affected by social dynamics (e.g., a pushy user who hogs the keyboard). But it's nearly as commonly used in industry as single-user testing.

## Facilitator's Roles

The facilitator (also called the experimenter) is the leader of the user test. The facilitator does the briefing, gives tasks to the user, and generally serves as the voice of the development team throughout the test. (Other developers may be observing the test, but should generally keep their mouths shut.)

One of the facilitator's key jobs is to coax the user to think aloud, usually by asking general questions. The facilitator may also move the session along. If the user is totally stuck on a task, the facilitator may progressively provide more help, e.g. "Do you see anything that might help you?", and then "What do you think that button does?" Only do this if you've already recorded the usability problem, and it seems unlikely that the user will get out of the tar pit themselves, and they need to get unstuck in order to get on to another part of the task that you want to test. Keep in mind that once you explain something, you lose the chance to find out what the user would have done by themselves.

Observer's Roles

While the user is thinking aloud, and the facilitator is coaching the think-aloud, any observers in the room should be doing the opposite: **keeping quiet**. Don't offer any help, don't attempt to explain the interface. Just sit on your hands, bite your tongue, and watch. You're trying to get a glimpse of how a typical user will interact with the interface. Since a typical user won't have the system's designer sitting next to them, you have to minimize your effect on the situation. It may be very hard for you to sit and watch someone struggle with a task, when the solution seems so *obvious* to you, but that's how you learn the usability problems in your interface.

Keep yourself busy by taking a lot of notes. What should you take notes about? As much as you can, but focus particularly on **critical incidents**, which are moments that strongly affect usability, either in task performance (efficiency or error rate) or in the user's satisfaction. Most critical incidents are negative. Pressing the wrong button is a critical incident. So is repeatedly trying the same feature to accomplish a task. Users may draw attention to the critical incidents with their think-aloud, with comments like "why did it do that?" or "@%!@#$!" Critical incidents can also be positive, of course. You should note down these pleasant surprises too.
Critical incidents give you a list of potential usability problems that you should focus on in the next round of iterative design.

Recording Observations

Here are various ways you can record observations from a user test. Paper notes are usually best,

although it may be hard to keep up. Having multiple observers taking notes helps. Audio and video recording are good for capturing the user's think-aloud, facial expressions, and body language. Video is also helpful when you want to put observers in a separate room, watching on a closed-circuit TV. Putting the observers in a separate room has some advantages: the user feels fewer eyes on them (although the video camera is another eye that can make users more self-conscious, since it's making a permanent record), the observers can't misbehave, and a big TV screen means more observers can watch. On the other hand, when the observers are in a separate room, they may not pay close attention to the test. It's happened that as soon as the user finds a usability problem, the observers start talking about how to fix that problem – and ignore the rest of the test. Having observers in the same room as the test forces them to keep quiet and pay attention.

Video is also useful for **retrospective testing** – using the videotape to debrief the user immediately after a test. It's easy to fast forward through the tape, stop at critical incidents, and ask the user what they were thinking, to make up for gaps in think-aloud.

The problem with audio and video tape is that it generates too much data to review afterwards. A few pages of notes are much easier to scan and derive usability problems. Screen capture software offers a cheap and easy way to record a user test, producing a digital movie. It's less obtrusive and easier to set up than a video camera, and some packages can also record an audio stream to capture the user's think-aloud.

How M any Users?

How many users do you need for formative evaluation? A simple model developed by Landauer and Nielsen ("A Mathematical Model of the Finding of Usability Problems", INTERCHI '93) postulates that every usability problem has a probability L of being found by a random user. So a single user finds a fraction L of the usability problems. If user tests are independent (a reasonable assumption if the users don't watch or talk to each other), then n users will find a fraction $1-(1-L)^n$ of the usability problems.

Based on user tests and heuristic evaluations of 11 different interfaces, Landauer and Nielsen estimated that L is typically 31% (the actual range was 12% to 60%). With L=31%, 5 users will find about 85% of the problems.

For formative evaluation, more users is not always better. Rather than running 15 users to find almost all likely usability problems with one design iteration, it's wiser to run fewer users in each iteration, in order to squeeze in more iterations.

5 users is the magic number often seen in the usability literature. But L may be much smaller than 31%. A study of a website found L=8%, which means that 5 users would have found only a third of the problems (Spool & Schroeder, "Testing web sites: five users is nowhere near enough", CHI 2001). Interfaces with high diversity – different ways of doing a task – may tend to have low L values.

The probability L of finding a problem may also vary from problem to problem (and user to user). And there's no way to compute L in advance. All published values for L have been computed *after* the fact. There's no model for determining L for a particular interface, task, or user.

## V. Designing a Swing GUI in NetBeans IDE

This tutorial guides you through the process of creating the graphical user interface (GUI) for an application called ContactEditor using the NetBeans IDE GUI Builder. In the process you will layout a GUI front-end that enables you to view and edit contact information of individuals included in an employee database.

In this tutorial you will learn how to:

- Use the GUI Builder Interface
- Create a GUI Container
- Add Components
- Resize Components
- Align Components
- Adjust Component Anchoring
- Set Component Auto-Resizing Behavior
- Edit Component Properties

### Getting Started

The IDE's GUI Builder makes it possible to build professional-looking GUIs without an intimate understanding of layout managers. You can lay out your forms by simply placing components where you want them.

### Creating a Project

Because all Java development in the IDE takes place within projects, we first need to create a new ContactEditor project within which to store sources and other project files. An IDE project is a group of Java source files plus its associated meta data, including project-specific properties files, an Ant build script that controls the build and run settings, and a project.xml file that maps Ant targets to IDE commands. While Java applications often consist of several IDE projects, for the purposes of this tutorial, we will build a simple application which is stored entirely in a single project.

To create a new ContactEditor application project:

1. Choose File > New Project. Alternately, you can click the New Project icon in the IDE toolbar.

2. In the Categories pane, select the Java node and in the Projects pane, choose Java Application. Click Next.

3. Enter ContactEditor in the Project Name field and specify the project location.

4. Leave the Use Dedicated Folder for Storing Libraries checkbox unselected.

5. Ensure that the Set as Main Project checkbox is selected and clear the Create Main Class field.

6. Click Finish.

   The IDE creates the ContactEditor folder on your system in the designated location. This folder contains all of the project's associated files, including its Ant script, folders for storing sources and tests, and a folder for project-specific metadata. To view the project structure, use the IDE's Files window.

**Creating a JFrame Container**

After creating the new application, you may have noticed that the Source Packages folder in the Projects window contains an empty <default package> node. To proceed with building our interface, we need to create a Java container within which we will place the other required GUI components. In this step we'll create a container using the JFrame component and place the container in a new package.

To add a JFrame container:

1. In the Projects window, right-click the ContactEditor node and choose New > JFrame Form.
   Alternatively, you can find a JFrame form by choosing New > Other > Swing GUI Forms > JFrame Form.

2. Enter ContactEditorUI as the Class Name.

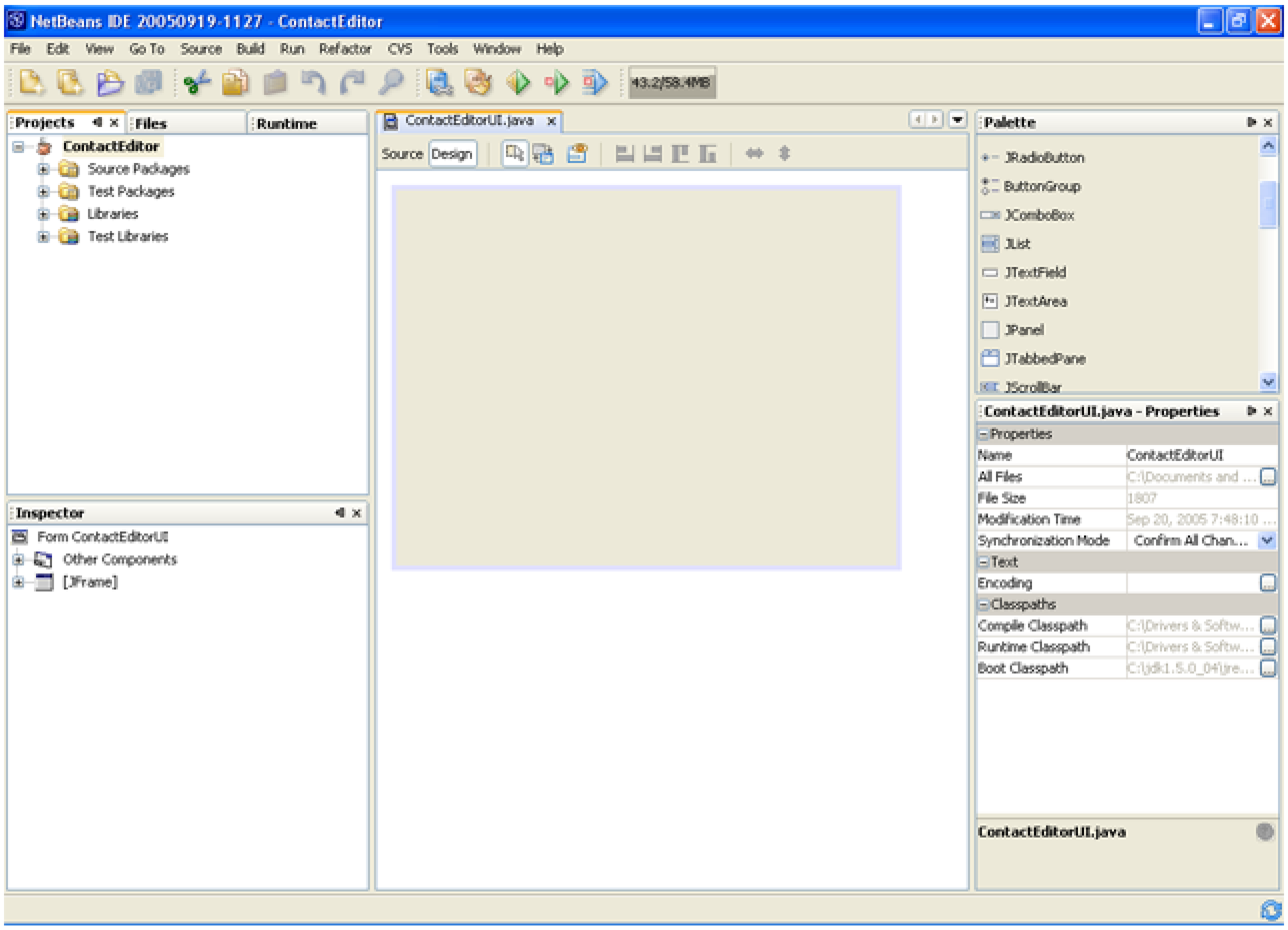
3. Enter my.contacteditor as the package.

4. Click Finish.

The IDE creates the ContactEditorUI form and the ContactEditorUI class within the ContactEditorUI.java application and opens the ContactEditorUI form in the GUI Builder. Notice that the my.contacteditor package replaces the default package.

## Getting Familiar with the GUI Builder

Now that we've set up a new project for our application, let's take a minute to familiarize ourselves with the GUI Builder's interface.



| | |
|---|---|
| Properties | |
| Name | ContactEditorUI |
| All Files | C:\Documents and ... |
| File Size | 1807 |
| Modification Time | Sep 20, 2005 7:48:10 ... |
| Synchronization Mode | Confirm All Chan... |
| Text | |
| Encoding | |
| Classpaths | |
| Compile Classpath | C:\Drivers & Softw... |
| Runtime Classpath | C:\Drivers & Softw... |
| Boot Classpath | C:\jdk1.5.0_04\jre... |

When we added the JFrame container, the IDE opened the newly-created ContactEditorUI form in an Editor tab with a toolbar containing several buttons, as shown in the preceding illustration. The ContactEditor form opened in the GUI Builder's Design view and three additional windows appeared automatically along the IDE's edges, enabling you to navigate, organize, and edit GUI forms as you build them.

The GUI Builder's various windows include:

- **Design Area.** The GUI Builder's primary window for creating and editing Java GUI forms. The toolbar's Source and Design toggle buttons enable you to view a class's source code or a graphical view of its GUI components. The additional toolbar buttons provide convenient access to common commands, such as choosing between Selection and Connection modes, aligning components, setting component auto-resizing behavior, and previewing forms.

- **Inspector.** Provides a representation of all the components, both visual and non-visual, in your application as a tree hierarchy. The Inspector also provides visual feedback about what component in the tree is currently being edited in the GUI Builder as well as allows you to organize components in the available panels.

- **Palette.** A customizable list of available components containing tabs for JFC/Swing, AWT, and JavaBeans components, as well as layout managers. In addition, you can create, remove, and rearrange the categories displayed in the Palette using the customizer.

- **Properties Window.** Displays the properties of the component currently selected in the GUI Builder, Inspector window, Projects window, or Files window.

If you click the Source button, the IDE displays the application's Java source code in the Editor with sections of code that are automatically generated by the GUI Builder indicated by blue areas, called Guarded Blocks. Guarded blocks are protected areas that are not editable in Source view. You can only edit code appearing in the white areas of the Editor when in Source view. If you need to make changes to the code within a Guarded Block, clicking the Design button returns the IDE's Editor to the GUI Builder where you can make the necessary adjustments to the form. When you save your changes, the IDE updates the file's sources.

**Note:** For advanced developers, the Palette Manager is available that enables you to add custom components from JARs, libraries, or other projects to the Palette. To add custom components through the Palette Manager, choose Tools > Palette > Swing/AWT Components.

**Key Concepts**

The IDE's GUI Builder solves the core problem of Java GUI creation by streamlining the workflow of creating graphical interfaces, freeing developers from the complexities of Swing layout managers. It does this by extending the current NetBeans IDE GUI Builder to support a straightforward "Free Design" paradigm with simple layout rules that are easy to understand and use. As you lay out your form, the GUI Builder provides visual guidelines suggesting optimal spacing and alignment of components. In the background, the GUI Builder translates your design decisions into a functional UI that is implemented using the new GroupLayout layout manager and other Swing constructs. Because it uses a dynamic layout model, GUI's built with the GUI Builder behave as you would expect at runtime, adjusting to accommodate any changes you make without altering the defined relationships between components. Whenever you resize the form, switch locales, or specify a different look and feel, your GUI automatically adjusts to respect the target look and feel's insets and offsets.

### Free Design

In the IDE's GUI Builder, you can build your forms by simply putting components where you want them as though you were using absolute positioning. The GUI Builder figures out which layout attributes are required and then generates the code for you automatically. You need not concern yourself with insets, anchors, fills, and so forth.

### Automatic Component Positioning (Snapping)

As you add components to a form, the GUI Builder provides visual feedback that assists in positioning components based on your operating system's look and feel. The GUI Builder provides helpful inline hints and other visual feedback regarding where components should be placed on your form, automatically snapping components into position along guidelines. It makes these suggestions based on the positions of the components that have already been placed in the form, while allowing the padding to remain flexible such that different target look and feels render properly at runtime.

### Visual Feedback

The GUI Builder also provides visual feedback regarding component anchoring and chaining relationships. These indicators enable you to quickly identify the various positioning relationships and component pinning behavior that affect the way your GUI will both appear and behave at runtime. This speeds the GUI design process, enabling you to quickly create professional-looking visual interfaces that work.

### First Things First

Now that you have familiarized yourself with the GUI builder's interface, it's time to begin developing the UI of our ContactEditor application. In this section we'll take a look at using the IDE's Palette to add the various GUI components that we need to our form.

Thanks to the IDE's Free Design paradigm, you no longer have to struggle with layout managers to control the size and position of the components within your containers. All you need to do is drag and drop (or pick and plop) the components you need to your GUI form as shown in the illustrations that follow.

### Adding Components: The Basics

Though the IDE's GUI Builder simplifies the process of creating Java GUIs, it is often helpful to sketch out the way you want your interface to look before beginning to lay it out. Many interface designers consider this a "best practice" technique, however, for the purposes of this tutorial you can simply peek at how our completed form should look by jumping ahead to the Previewing your GUI section.

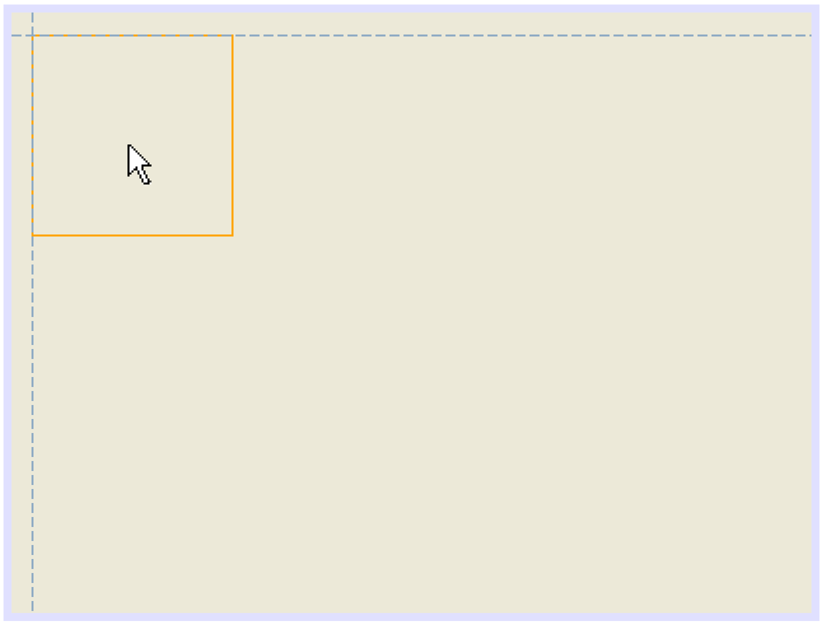
Since we've already added a JFrame as our form's top-level container, the next step is to add a couple of JPanels which will enable us to cluster the components of our UI using titled borders. Refer to the following illustrations and notice the IDE's "pick and plop" behavior when accomplishing this.

To add a JPanel:

1. In the Palette window, select the Panel component from the Swing category by clicking and releasing the mouse button.

2. Move the cursor to the upper left corner of the form in the GUI Builder. When the component is located near the container's top and left edges, horizontal and vertical alignment guidelines appear indicating the preferred margins. Click in the form to place the JPanel in this location.

   The JPanel component appears in the ContactEditorUI form with orange highlighting signifying that it is selected, as shown in the following illustration. After releasing the mouse button, small indicators appear to show the component's anchoring relationships and a corresponding JPanel node is displayed in the Inspector window.



Next, we need to resize the JPanel to make room for the components we'll place within it a little later, but let's take a minute to point out another of the GUI Builder's visualization features first. In order to do this we need to deselect the JPanel we just added. Because we haven't added a title border yet, the panel disappears. Notice, however, that when you pass the cursor over the JPanel, its edges change to light gray so that its position can be clearly seen. You need only to click anywhere within the component to reselect it and cause the resize handles and anchoring indicators to reappear.

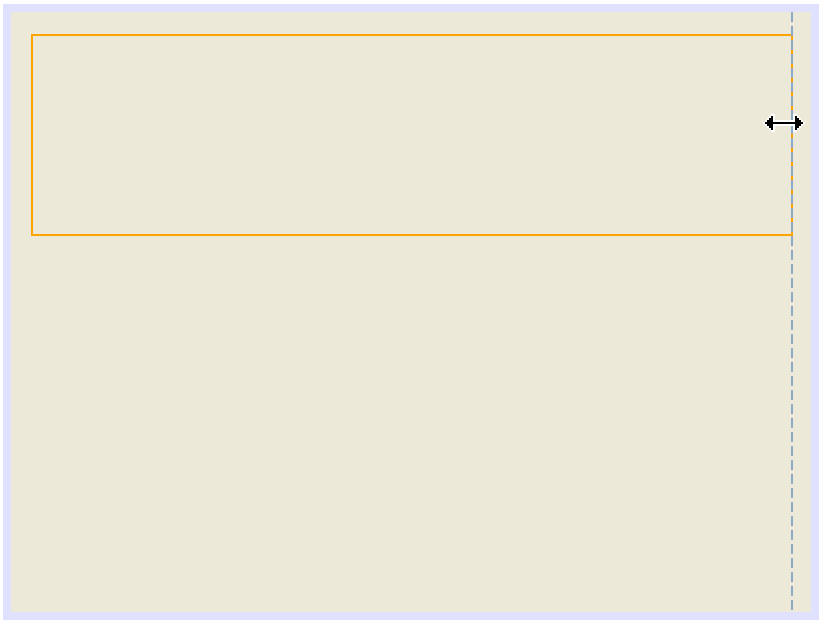
To resize the JPanel:

1. Select the JPanel you just added. The small square resize handles reappear around the component's perimeter.

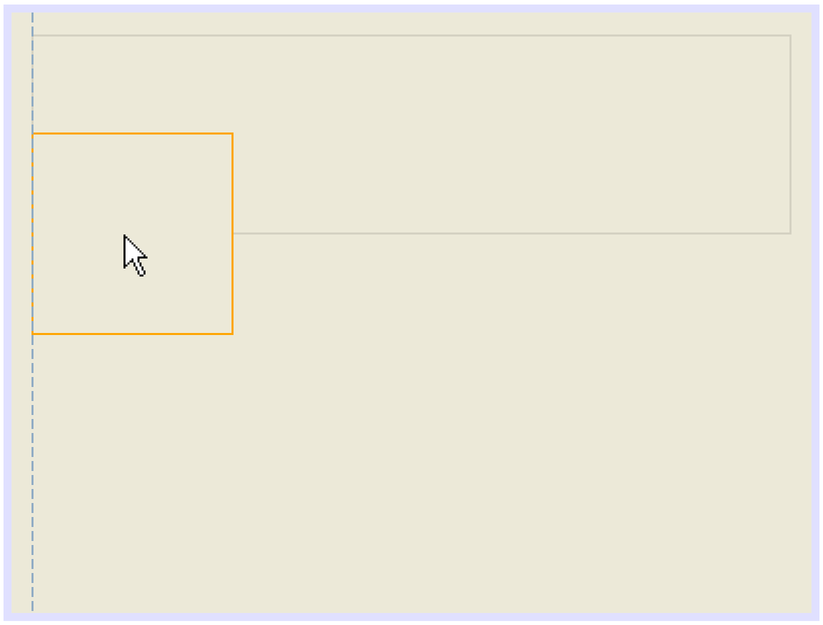
2. Click and hold the resize handle on the right edge of the JPanel and drag until the dotted alignment guideline appears near the form's edge.

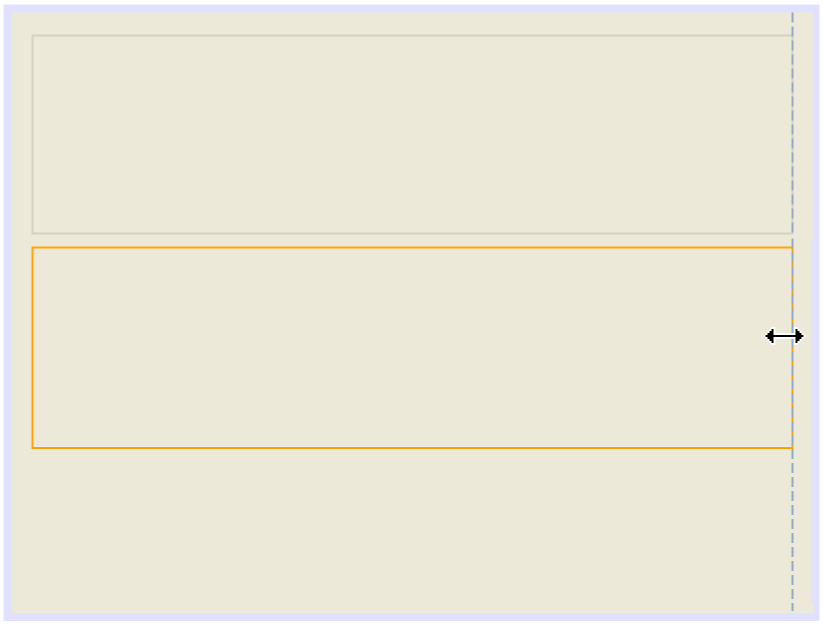
3. Release the resize handle to resize the component.

   The JPanel component is extended to span between the container's left and right margins in accordance with the recommended offset, as shown in the following illustration.



Now that we've added a panel to contain our UI's Name information, we need to repeat the process to add another directly below the first for the E-mail information. Referring to the following illustrations, repeat the previous two tasks, paying attention to the GUI Builder's suggested positioning. Notice that the suggested vertical spacing between the two JPanels is much narrower than that at the edges. Once you have added the second JPanel, resize it such that it fills the form's remaining vertical space.





Because we want to visually distinguish the functions in the upper and lower sections of our GUI, we need to add a border and title to each JPanel. First we'll accomplish this using the Properties window and then we'll try it using the pop-up menu.

To add title borders to the JPanels:

1. Select the top JPanel in the GUI Builder.

2. In the Properties window, click the ellipsis button (...) next to the Border property.

3. In the JPanel Border editor that appears, select the TitledBorder node in the Available Borders pane.

4. In the Properties pane below, enter Name for the Title property.

5. Click the ellipsis (...) next to the Font property, select Bold for the Font Style, and enter 12 for the Size. Click OK to exit the dialogs.

6. Select the bottom JPanel and repeat steps 2 through 5, but this time right-click the JPanel and access the Properties window using the pop-up menu. Enter E-mail for the Title property.

   Titled borders are added to both JPanel components.

**Adding Individual Components to the Form**

Now we need to start adding the components that will present the actual contact information in our contact list. In this task we'll add four JTextFields that will display the contact information and the JLabels that will describe them. While accomplishing this, notice the horizontal and vertical guidelines that the GUI Builder displays, suggesting the preferred component spacing as defined by your operating system's look and feel. This ensures that your GUI is automatically rendered respecting the target operating system's look and feel at runtime.

To add a JLabel to the form:

1. In the Palette window, select the Label component from the Swing category.

2. Move the cursor over the Name JPanel we added earlier. When the guidelines appear indicating that the JLabel is positioned in the top left corner of the JPanel with a small margin at the top and left edges, click to place the label.

   The JLabel is added to the form and a corresponding node representing the component is added to the Inspector window.

Before going further, we need to edit the display text of the JLabel we just added. Though you can edit component display text at any point, the easiest way is to do this as you add them.

To edit the display text of a JLabel:

1. Double-click the JLabel to select its display text.
2. Type First Name: and press Enter.

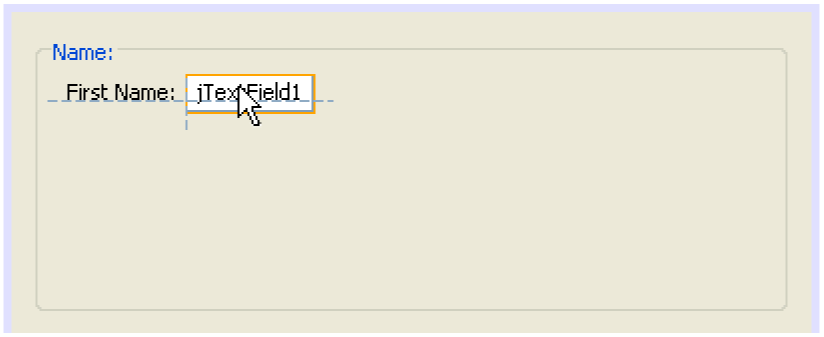
   The JLabel's new name is displayed and the component's width adjusts as a result of the edit.

Now we'll add a JTextField so we can get a glimpse of the GUI Builder's baseline alignment feature.

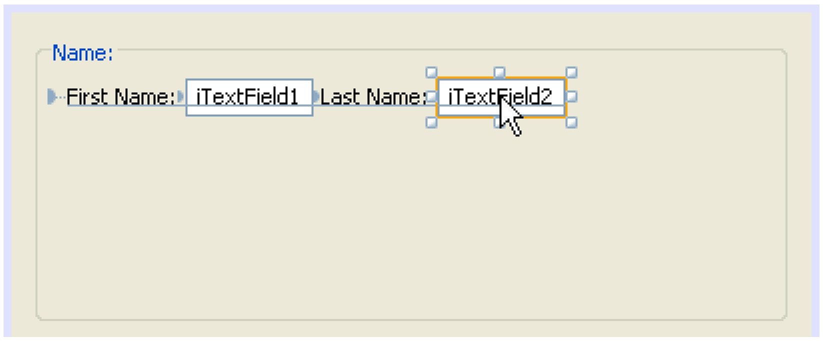
To add a JTextField to the form:

1. In the Palette window, select the Text Field component from the Swing category.
2. Move the cursor immediately to the right of the First Name: JLabel we just added. When the horizontal guideline appears indicating that the JTextField's baseline is aligned with that of the JLabel and the spacing between the two components is suggested with a vertical guideline, click to position the JTextField.

   The JTextField snaps into position in the form aligned with the JLabel's baseline, as shown in the following illustration. Notice that the JLabel shifted downward slightly in order to align with the taller text field's baseline. As usual, a node representing the component is added to the Inspector window.



Before proceeding further, we need to add an additional JLabel and JTextField immediately to the right of those we just added, as shown in the following illustration. This time enter Last Name: as the JLabel's display text and leave the JTextFields' placeholder text as it is for now.



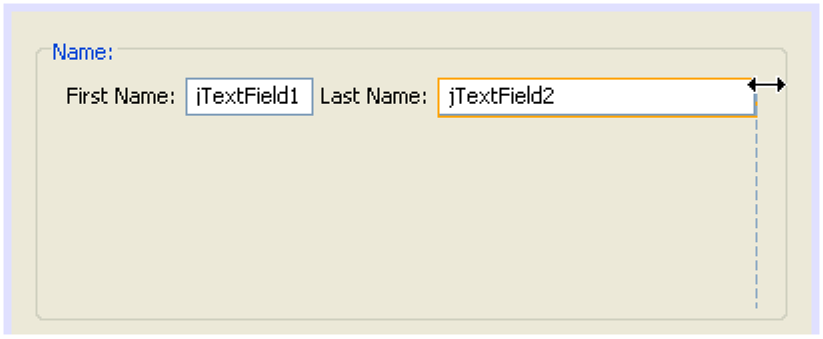
To resize a JTextField:

1. Select the JTextField we just added to the right of the Last Name: JLabel.

2. Drag the JTextField's right edge resize handle toward the right edge of the enclosing JPanel.

3. When the vertical alignment guidelines appear suggesting the margin between the text field and right edge of the JPanel, release the mouse button to resize the JTextField.

   The JTextField's right edge snaps into alignment with the JPanel's recommended edge margin, as shown in the following illustration.



**Note:** To view a demonstration of the previous procedure, open the quickstart-gui_swfs/003_add_individual_components.html file in the zip folder that you've downloaded.

### Adding Multiple Components to the Form

Now we'll add the Title: and Nickname: JLabels that describe two JTextFields that we're going to add in a minute. We'll pick and plop the components while pressing the Shift key, to quickly add them to the form. While accomplishing this, again notice that the GUI Builder displays horizontal and vertical guidelines suggesting the preferred component spacing.

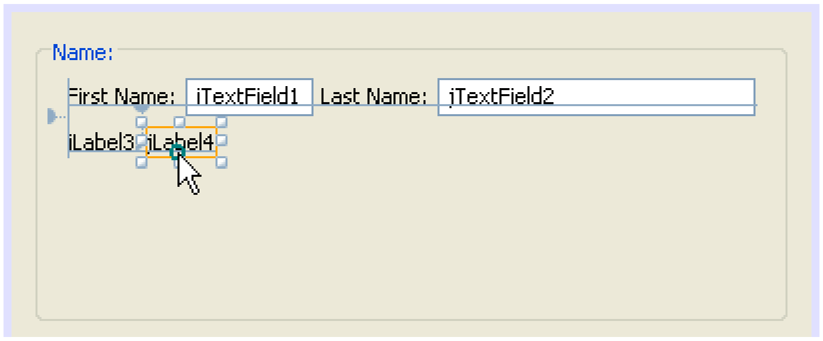
To add multiple JLabels to the form:

1. In the Palette window, select the Label component from the Swing category by clicking and releasing the mouse button.

2. Move the cursor over the form directly below the First Name: JLabel we added earlier. When the guidelines appear indicating that the new JLabel's left edge is aligned with that of the JLabel above and a small margin exists between them, shift-click to place the first JLabel.

3. While still pressing the Shift key, click to place another JLabel immediately to the right of the first. Make certain to release the Shift key prior to positioning the second JLabel. If you forget to release the Shift key prior to positioning the last JLabel, simply press the Escape key.

   The JLabels are added to the form creating a second row, as shown in the following illustration. Nodes representing each component are added to the Inspector window.



Before moving on, we need to edit the JLabels' name so that we'll be able to see the effect of the alignments we'll set later.

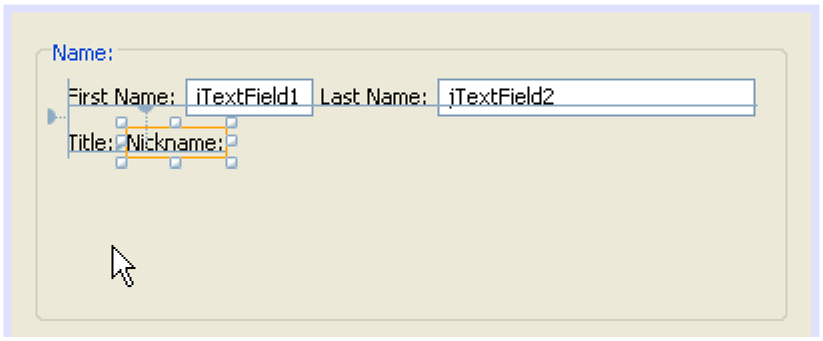
To edit the display text of JLabels:

1. Double-click the first JLabel to select its display text.

2. Type Title: and press Enter.

3. Repeat steps 1 and 2, entering Nickname: for the second JLabel's name property.

   The JLabels' new names are displayed in the form and are shifted as a result of their edited widths, as shown in the following illustration.



**Inserting Components**

Often it is necessary to add a component between components that are already placed in a form. Whenever you add a component between two existing components, the GUI Builder automatically shifts them to make room for the new component. To demonstrate this, we'll insert a JTextField between the JLabels we added previously, as shown in the following two illustrations.

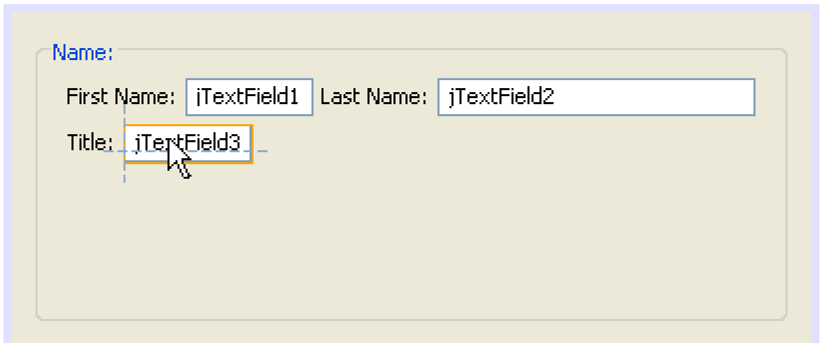
To insert a JTextField between two JLabels:

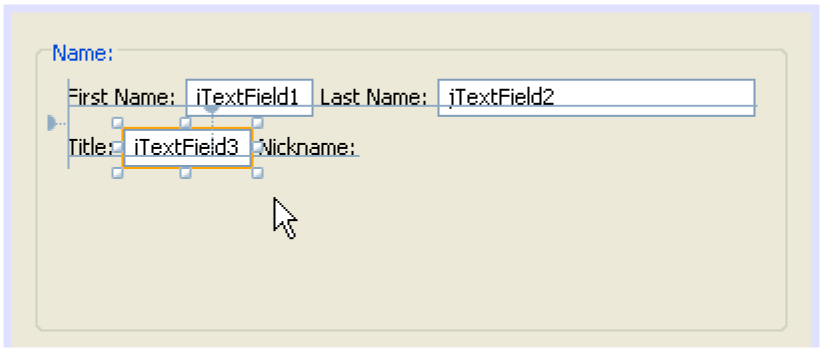
1. In the Palette window, select the Text Field component from the Swing category.

2. Move the cursor over the Title: and Nickname: JLabels on the second row such that the JTextField overlaps both and is aligned to their baselines. If you encounter difficulty positioning the new text field, you can snap it to the left guideline of the Nickname JLabel as shown in the first image below.

3. Click to place the JTextField between the Title: and Nickname: JLabels.

   The JTextField snaps into position between the two JLabels. The rightmost JLabel shifts toward the right of the JTextField to accommodate the suggested horizontal offset.





We still need to add one additional JTextField to the form that will display each contact's nickname on the right side of the form.

To add a JTextField:

1. In the Palette window, select the Text Field component from the Swing category.

2. Move the cursor to the right of the Nickname label and click to place the text field.

   The JTextField snaps into position next to the JLabel on its left.

To resize a JTextField:

1. Drag the resize handles of the Nickname: label's JTextField you added in the previous task toward the right of the enclosing JPanel.

2. When the vertical alignment guidelines appear suggesting the margin between the text field and JPanel edges, release the mouse button to resize the JTextField.

   The JTextField's right edge snaps into alignment with the JPanel's recommended edge margin and the GUI Builder infers the appropriate resizing behavior.

**Moving Forward**

Alignment is one of the most fundamental aspects of creating professional-looking GUIs. In the previous section we got a glimpse of the IDE's alignment features while adding the JLabel and JTextField components to our ContactEditorUI form. Next, we'll take a more in depth look at the GUI Builder's alignment features as we work with the various other components we need for our application.

## Component Alignment

Every time you add a component to a form, the GUI Builder effectively aligns them, as evidenced by the alignment guidelines that appear. It is sometimes necessary, however, to specify different relationships between groups of components as well. Earlier we added four JLabels that we need for our ContactEditor GUI, but we didn't align them. Now we'll align the two columns of JLabels so that their right edges line up.

To align components:

1. Hold down the Ctrl key and click to select the First Name: and Title: JLabels on the left side of the form.

2. Click the Align Right in Column button ( ) in the toolbar. Alternately, you can right-click either one and choose Align > Right in Column from the pop-up menu.

3. Repeat this for the Last Name: and Nickname: JLabels as well.

   The JLabels' positions shift such that the right edges of their display text are aligned. The anchoring relationships are updated, indicating that the components have been grouped.

Before we're finished with the JTextFields we added earlier, we need to make sure that the two JTextFields we inserted between the JLabels are set to resize correctly. Unlike the two JTextFields that we stretched to the right edge of our form, inserted components' resizeability behavior isn't automatically set.

To set component resizeability behavior:

1. Control-click the two inserted JTextField components to select them in the GUI Builder.

2. With both JTextFields selected, right-click either one of them and choose Auto Resizing > Horizontal from the pop-up menu.

   The JTextFields are set to resize horizontally at runtime. The alignment guidelines and anchoring indicators are updated, providing visual feedback of the component relationships.

To set components to be the same size:

1. Control-click all four of the JTextFields in the form to select them.

2. With the JTextFields selected, right-click any one of them and choose Set Default Size from the pop-up menu.

   The JTextFields are all set to the same width and indicators are added to the top edge of each, providing visual feedback of the component relationships.

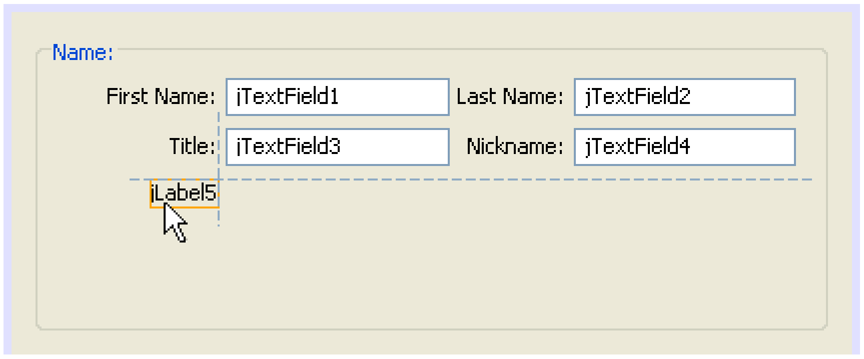
Now we need to add another JLabel describing the JComboBox that will enable users to select the format of the information our ContactEditor application will display.

To align a JLabel to a component group:

1. In the Palette window, select the Label component from the Swing category.

2. Move the cursor below the First Name and Title JLabels on the left side of the JPanel. When the guideline appears indicating that the new JLabel's right edge is aligned with the right edges of the component group above (the two JLabels), click to position the component.

   The JLabel snaps into a right-aligned position with the column of JLabels above, as shown in the following illustration. The GUI Builder updates the alignment status lines indicating the component's spacing and anchoring relationships.



As in the previous examples, double-click the JLabel to select its display text and then enter Display Format: for the display name. Notice that when the JLabel snaps into position, the other components shift to accommodate the longer display text.

**Baseline Alignment**

Whenever you add or move components that include text (JLabels, JTextFields, and so forth), the IDE suggests alignments which are based on the baselines of the text in the components. When we inserted the JTextField earlier, for example, its baseline was automatically aligned to the adjacent JLabels.

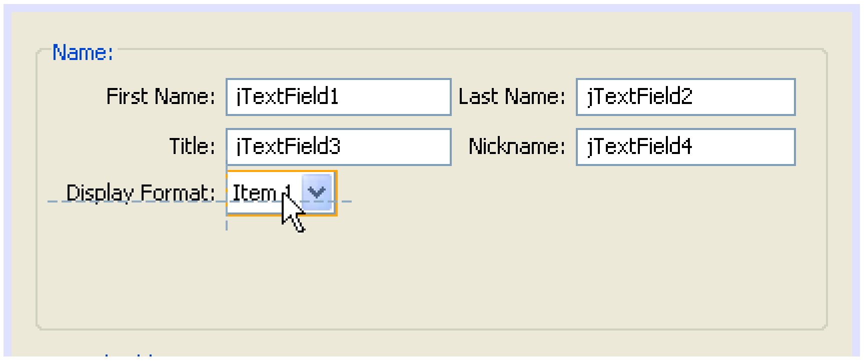
Now we'll add the combo box that will enable users to select the format of the information that our ContactEditor application will display. As we add the JComboBox, we'll align its baseline to that of the JLabel's text. Notice once again the baseline alignment guidelines that appear to assist us with the positioning.

To align the baselines of components:

1. In the Palette window, select the Combo Box component from the Swing category.

2. Move the cursor immediately to the right of the JLabel we just added. When the horizontal guideline appears indicating that the JComboBox's baseline is aligned with the baseline of the text in the JLabel and the spacing between the two components is suggested with a vertical guideline, click to position the combo box.

   The component snaps into a position aligned with the baseline of the text in the JLabel to its left, as shown in the following illustration. The GUI Builder displays status lines indicating the component's spacing and anchoring relationships.

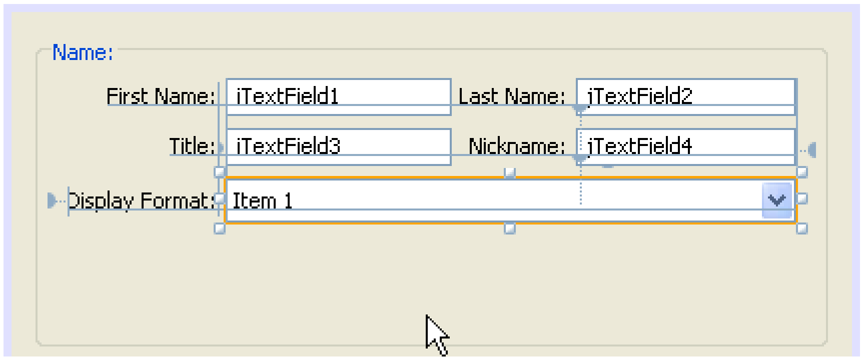


To resize the JComboBox:

1. Select the ComboBox in the GUI Builder.

2. Drag the resize handle on the JComboBox's right edge toward the right until the alignment guidelines appear suggesting the preferred offset between the JComboBox and JPanel edges.

As shown in the following illustration, the JComboBox's right edge snaps into alignment with the JPanel's recommended edge margin and the component's width is automatically set to resize with the form.



Editing component models is beyond the scope of this tutorial, so for the time being we'll leave the JComboBox's placeholder item list as it is.

top

**Reviewing What We've Learned**

We've gotten off to a good start building our ContactEditor GUI, but let's take a minute to recap what we've learned while we add a few more of the components our interface requires.

Until now we've concentrated on adding components to our ContactEditor GUI using the IDE's alignment guidelines to help us with positioning. It is important to understand, however, that another integral part of component placement is anchoring. Though we haven't discussed it yet, you've already taken advantage of this feature without realizing it. As mentioned previously, whenever you add a component to a form, the IDE suggests the target look and feel's preferred positioning with guidelines. Once placed, new components are also anchored to the nearest container edge or component to ensure that component relationships are maintained at runtime. In this section, we'll concentrate on accomplishing the tasks in a more streamlined fashion while pointing out the work the GUI builder is doing behind the scenes.

**Adding, Aligning, and Anchoring**

The GUI Builder enables you to lay out your forms quickly and easily by streamlining typical workflow gestures. Whenever you add a component to a form, the GUI Builder automatically snaps them into the preferred positions and sets the necessary chaining relationships so you can concentrate on designing your forms rather than struggling with complicated implementation details.

To add, align, and edit the display text of a JLabel:

1. In the Palette window, select the Label component from the Swing category.

2. Move the cursor over the form immediately below the bottom JPanel's E-mail title. When the guidelines appear indicating that it's positioned in the top left corner of the JPanel with a small margin at the top and left edges, click to place the JLabel.

3. Double-click the JLabel to select its display text. Then type E-mail Address: and press Enter.

   The JLabel snaps into the preferred position in the form, anchored to the top and left edges of the enclosing JPanel. Just as before, a corresponding node representing the component is added to the Inspector window.

To add a JTextField:

1. In the Palette window, select the Text Field component from the Swing category.

2. Move the cursor immediately to the right of the E-mail Address label we just added. When the guidelines appear indicating that the JTextField's baseline is aligned with the baseline of the text in the JLabel and the margin between the two components is suggested with a vertical guideline, click to position the text field.

   The JTextField snaps into position on the right of the E-mail Address: JLabel and is chained to the JLabel. Its corresponding node is also added to the Inspector window.

3. Drag the resize handle of the JTextField toward the right of the enclosing JPanel until the alignment guidelines appear suggesting the offset between the JTextField and JPanel edges.

   The JTextField's right edge snaps to the alignment guideline indicating the preferred margins.

Now we need to add the JList that will display our ContactEditor's entire contact list.

To add and resize a JList:

1. In the Palette window, select the List component from the Swing category.

2. Move the cursor immediately below the E-mail Address JLabel we added earlier. When the guidelines appear indicating that the JList's top and left edges are aligned with the preferred margins along the JPanel's left edge and the JLabel above, click to position the JList.

3. Drag the JList's right resize handle toward the right of the enclosing JPanel until the alignment guidelines appear indicating that it is the same width as the JTextField above.

   The JList snaps into the position designated by the alignment guidelines and its corresponding node is displayed in the Inspector window. Notice also that the form expands to accommodate the newly added JList.

Since JLists are used to display long lists of data, they typically require the addition of a JScrollPane. Whenever you add a component which requires a JScrollPane, the GUI Builder automatically adds it for you. Because JScrollPanes are non-visual components, you have to use the Inspector window in order to view or edit any JScrollPanes that the GUI Builder created.

**Component Sizing**

It is often beneficial to set several related components, such as buttons in modal dialogues, to be the same size for visual consistency. To demonstrate this we'll add four JButtons to our ContactEditor form that will allow us to add, edit, and remove individual entries from our contact list, as shown in the following illustrations. Afterwards, we'll set the four buttons to be the same size so they can be easily recognized as offering related functionality.

To add, align, and edit the display text of multiple buttons:

1. In the Palette window, select the Button component.

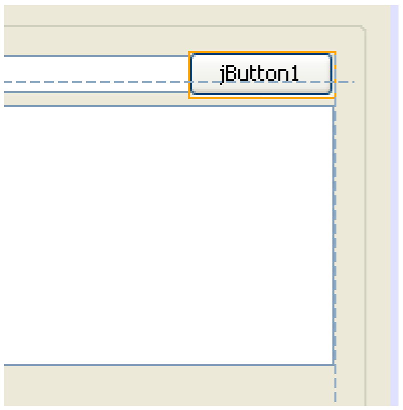
2. Move the JButton over the right edge of the E-mail Address JTextField in the lower JPanel. When the guidelines appear indicating that the JButton's baseline and right edge are aligned with that of the JTextField, shift-click to place the first button along the JFrame's right edge. The JTextField's width shrinks to accommodate the JButton when you release the mouse button.

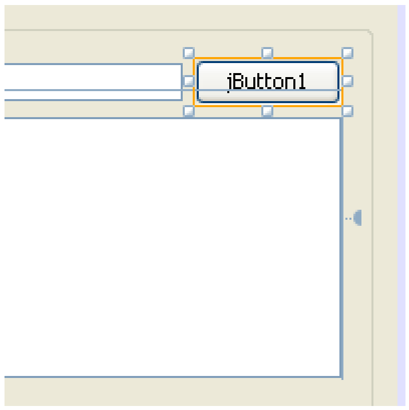
3. Move the cursor over the top right corner of the JList in the lower JPanel. When the guidelines appear indicating that the JButton's top and right edges are aligned with that of the JList, shift-click to place the second button along the JFrame's right edge.

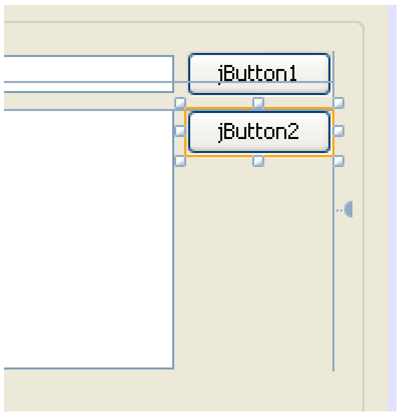
4. Add two additional JButtons below the two we already added to create a column. Make certain to position the JButtons such that the suggested spacing is respected and consistent. If you forget to release the Shift key prior to positioning the last JButton, simply press the Escape key.

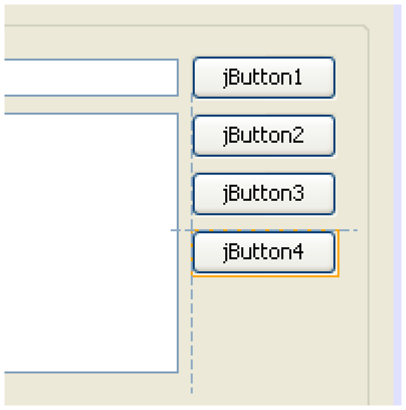
5. Set the display text for each JButton. (You can edit a button's text by right-clicking the button and choosing Edit Text. Or you can click the button, pause, and then click again.) Enter Add for the top button, Edit for the second, Remove for the third, and As Default for the fourth.

   The JButton components snap into the positions designated by the alignment guidelines. The width of the buttons changes to accommodate the new names.









Now that the buttons are positioned where we want them, we'll set the four buttons to be the same size for visual consistency as well as to clarify that they are related functionally.

To set components to the same size:

1. Select all four JButtons by pressing the Control key while making your selection.

2. Right-click one of them and choose Same Size > Same Width from the pop-up menu.

   The JButtons are set to the same size as the button with the longest name.

## Indentation

Often it is necessary to cluster multiple components under another component such that it is clear they belong to a group of related functions. One typical case, for example, is placing several related checkboxes below a common label. The GUI Builder enables you to accomplish indenting easily by providing special guidelines suggesting the preferred offset for your operating system's look and feel.

In this section we'll add a few JRadioButtons below a JLabel that will allow users to customize the way the application displays data. Refer to the following illustrations while accomplishing this or click the View Demo link following the procedure to view an interactive demonstration.

To indent JRadioButtons below a JLabel:

1. Add a JLabel named Mail Format to the form below the JList. Make certain the label is left aligned with the JList above.

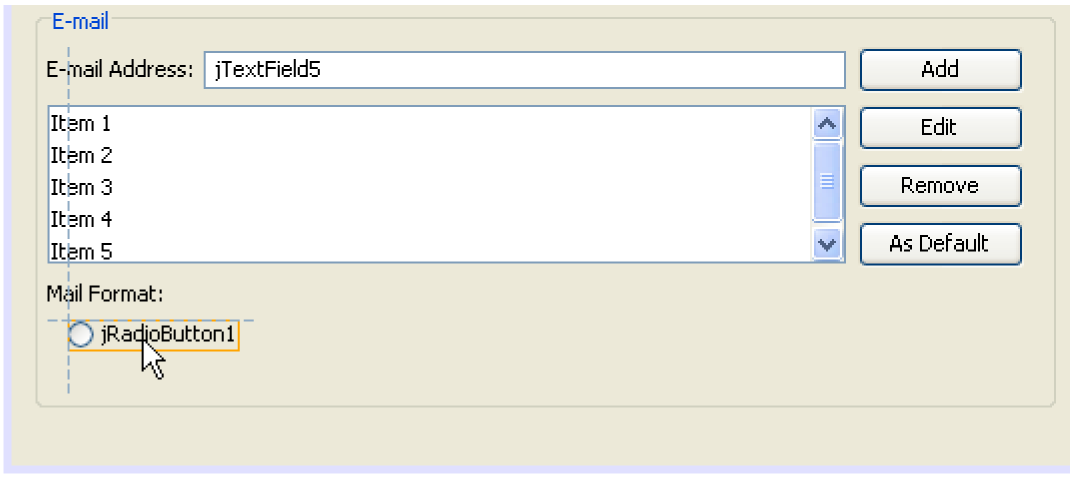
2. In the Palette window, select the Radio Button component from the Swing category.

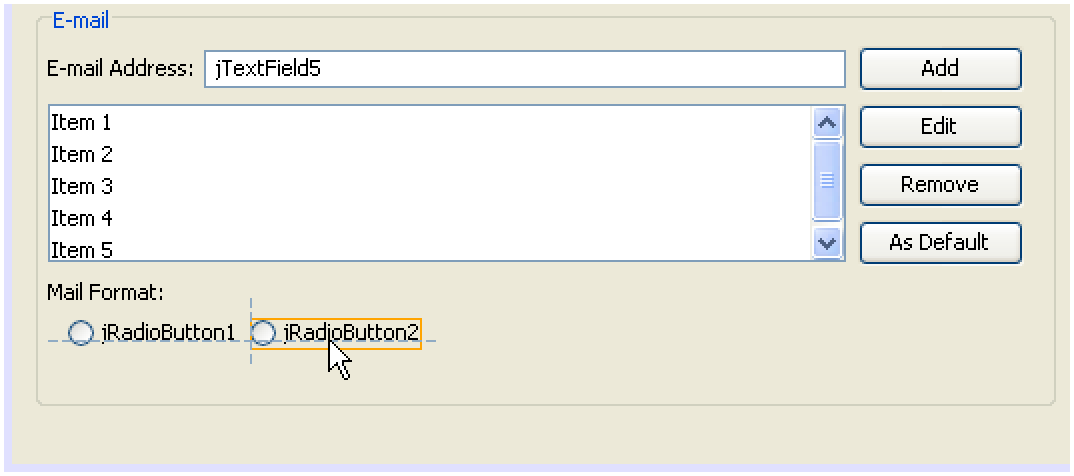
3. Move the cursor below the JLabel that we just added. When the guidelines appear indicating that the JRadioButton's left edge is aligned with that of the JLabel, move the JRadioButton slightly to the right until secondary indentation guidelines appear. Shift-click to place the first radio button.

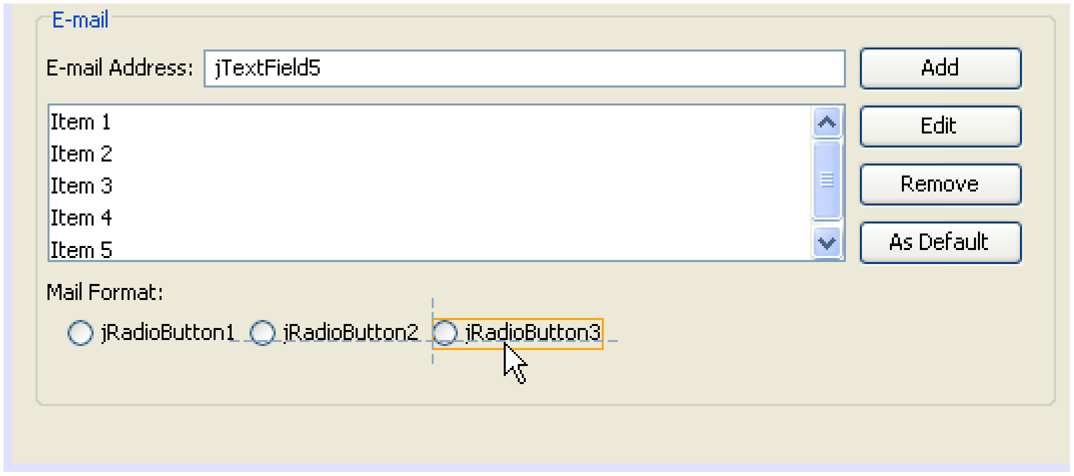
4. Move the cursor to the right of the first JRadioButton. Shift-click to place the second and third JRadioButtons, being careful to respect the suggested component spacing. Make certain to release the Shift key prior to positioning the last JRadioButton.

5. Set the display text for each JRadioButton. (You can edit a button's text by right-clicking the button and choosing Edit Text. Or you can click the button, pause, and then click again.) Enter HTML for the left radio button, Plain Text for the second, and Custom for the third.

Three JRadioButtons are added to the form and indented below the Mail Format JLabel.







Now we need to add the three JRadioButtons to a ButtonGroup to enable the expected toggle behavior in which only one radio button can be selected at a time. This will, in turn, ensure that our ContactEditor application's contact information will be displayed in the mail format of our choosing.

To add JRadioButtons to a ButtonGroup:

1. In the Palette window, select the Button Group component from the Swing category.

2. Click anywhere in the GUI Builder design area to add the ButtonGroup component to the form. Notice that the ButtonGroup does not appear in the form itself, however, it is visible in the Inspector's Other Components area.

3. Select all three of the JRadioButtons in the form.

4. In the Properties window, choose buttonGroup1 from the buttonGroup property combo box.

   Three JRadioButtons are added to the button group.

**Making the Final Adjustments**

We've managed to rough out our ContactEditor application's GUI, but there are still a few things remaining to do. In this section, we'll take a look at a couple of other typical layout tasks that the GUI Builder streamlines.

**Finishing Up**

Now we need to add the buttons that will enable users to confirm the information they enter for an individual contact and add it to the contact list or cancel, leaving the database unchanged. In this step, we'll add the two required buttons and then edit them so that they appear the same size in our form even though their display text are different lengths.

To add and edit the display text of buttons:

1. If the lower JPanel is extended to the bottom edge of the JFrame form, drag the bottom edge of the JFrame down. This gives you space between the edge of the JFrame and the edge of the JPanel for your OK and Cancel buttons.

2. In the Palette window, select the Button component from the Swing category.

3. Move the cursor over the form below the E-mail JPanel. When the guidelines appear indicating that the JButton's right edge is aligned with the lower right

corner of the JFrame, click to place the button.

4. Add another JButton to the left of the first, making certain to place it using the suggested spacing along the JFrame's bottom edge.

5. Set the display text for each JButton. Enter OK for the left button and Cancel for right one. Notice that the width of the buttons changes to accommodate the new names.

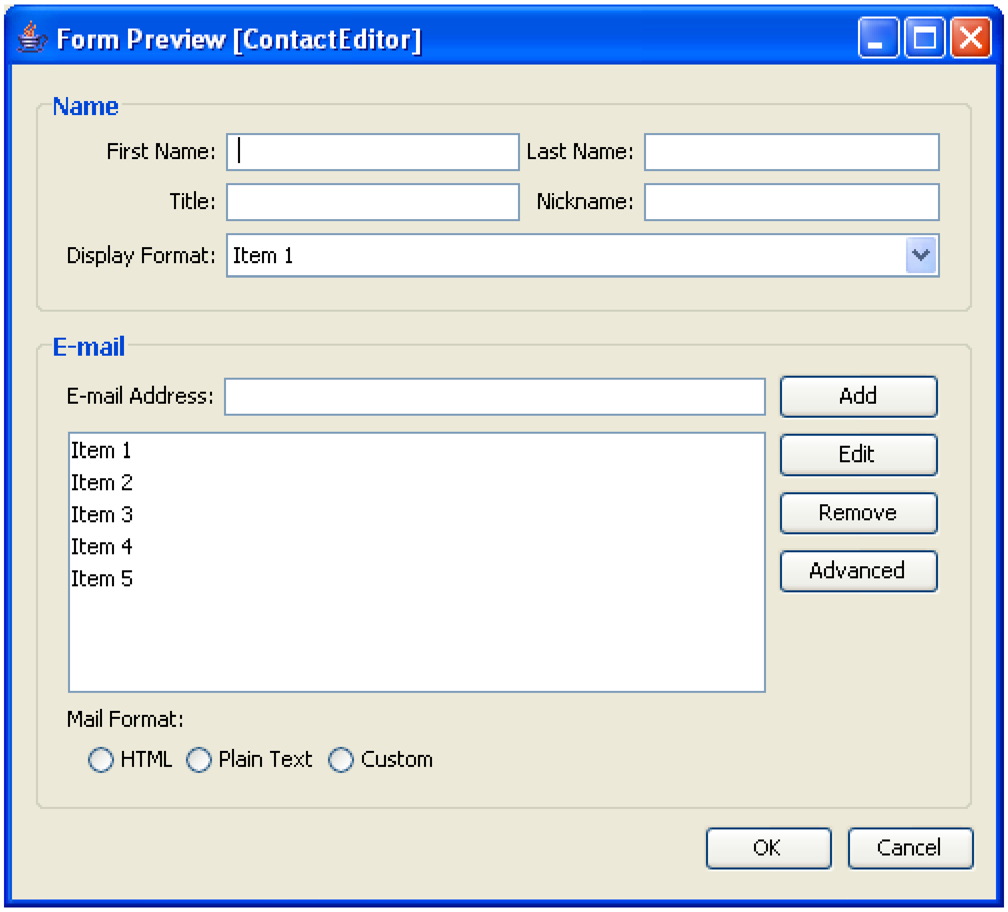
6. Set the two JButtons to be the same size by selecting both, right-clicking either, and choosing Same Size > Same Width from the pop-up menu.

   The JButton components appear in the form and their corresponding nodes are displayed in the Inspector window. The JButton components' code is also added to the form's source file which is visible in the Editor's Source view. Each of the JButtons are set to the same size as the button with the longest name.

The last thing we need to do is delete the placeholder text in the various components. Note that while removing placeholder text after roughing out a form can be a helpful technique in avoiding problems with component alignments and anchoring relationships, most developers typically remove this text in the process of laying out their forms. As you go through the form, select and delete the placeholder text for each of the JTextFields. We'll leave the placeholder items in both the JComboBox and JList for a later tutorial.

## Previewing Your GUI

Now that you have successfully built the ContactEditor GUI, you can try your interface to see the results. You can preview your form as you work by clicking the Preview Form button () in the GUI Builder's toolbar. The form opens in its own window, allowing you to test it prior to building and running.


Form Preview [ContactEditor]

Name

First Name:

Last Name:

Title:

Nickname:

Display Format: Item 1

E-mail

E-mail Address:

Item 1
Item 2
Item 3
Item 4
Item 5

Mail Format:

HTML Plain Text Custom

top

**Deploying GUI Applications**

In order for the interfaces you create with the GUI Builder to work outside of the IDE, the application must be compiled against classes for the GroupLayout layout manager and also have those classes available at runtime. These classes are included in Java SE 6, but not in Java SE 5. If you develop the application to run on Java SE 5, your application needs to use the Swing Layout Extensions library.

If you are running the IDE on JDK 5, the IDE automatically generates your application code to use the Swing Layout Extensions library. When you deploy the application, you need to include the Swing Layout Extensions library with the application. When you build the application (Build > Build Main Project), the IDE automatically provides a copy of the library's JAR file in the application's dist/lib folder. The IDE also adds each of the JAR files that are in the dist/lib folder to the Class-Path element in the application JAR file's manifest.mf file.

If you are running the IDE on JDK 6, the IDE generates your application code to use the GroupLayout classes that are in Java SE 6. This means that you can deploy the application to run on systems with Java SE 6 installed and you do not need to package your application with the Swing Layout Extensions library.

**Note:** If you create your application using JDK 6 but you need the application to also run on Java SE 5, you can have the IDE generate its code to use the Swing Layout Extensions library instead of the classes in Java SE 6. Open the ContactEditorUI class in the GUI Editor. In the Inspector, expand the ContactEditorUI node and choose Form ContactEditorUI. In the Properties window, change the value of the Layout Generation Style property to Swing Layout Extensions Library.

**Distributing and Running Standalone GUI Applications**

To prepare your GUI application for distribution outside of the IDE:

- Zip the project's dist folder into a ZIP archive. (The dist folder might also contain a lib folder, which you would also need to include.)

To run a standalone GUI application from the command line:

1. Navigate to the project's dist folder.
2. Type the following:

   ```
   java -jar <jar_name>.jar
   ```

**Note:** If you encounter the following error:

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/jdesktop/layout/GroupLayout$Group
```

Ensure that the manifest.mf file references the currently installed version of the Swing Layout Extensions Library.

## References

- **Carroll, J. (Ed.). HUMAN-COMPUTER INTERACTION in the New Millenium, 2001, Addison-Wesley Pub Co, ISBN: 0201704471.**

- **Nielsen, J. USABILITY ENGINEERING, AP Professional, 1993, ISBN:0125184069**

- **Raskin, J. THE HUMANE INTERFACE: NEW DIRECTIONS FOR DESIGNING INTERACTIVE SYSTEMS, Addison-Wesley, 2000, ISBN:0201379376**