Chapter 6

Slides by: Ms. Shree Jani- Jaswal

Usability, Testing & Quality

Topics to be covered:

- Usability principles
- User interface evaluating user interfaces
- Testing & quality strategies
- Defects
- Test cases & test plan
- Inspections
- Quality assurance

Usability

User Centred Design

- Software development should focus on the needs of users
  - Understand your users
  - Design software based on an understanding of the users' tasks
  - Ensure users are involved in decision making processes
  - Design the user interface following guidelines for good usability
  - Have users work with and give their feedback about prototypes, on-line help and draft user manuals

The importance of focusing on users

- Reduced training and support costs
- Reduced time to learn the system
- Greater efficiency of use
- Reduced costs by only developing features that are needed
- Reduced costs associated with changing the system later
- Better prioritizing of work for iterative development
- Greater attractiveness of the system, so users will be more willing to buy and use it

Characteristics of Users

- Software engineers must develop an understanding of the users
  - Goals for using the system
  - Potential patterns of use
  - Demographics
  - Knowledge of the domain and of computers
  - Physical ability
  - Psychological traits and emotional feelings

Basics of User Interface Design

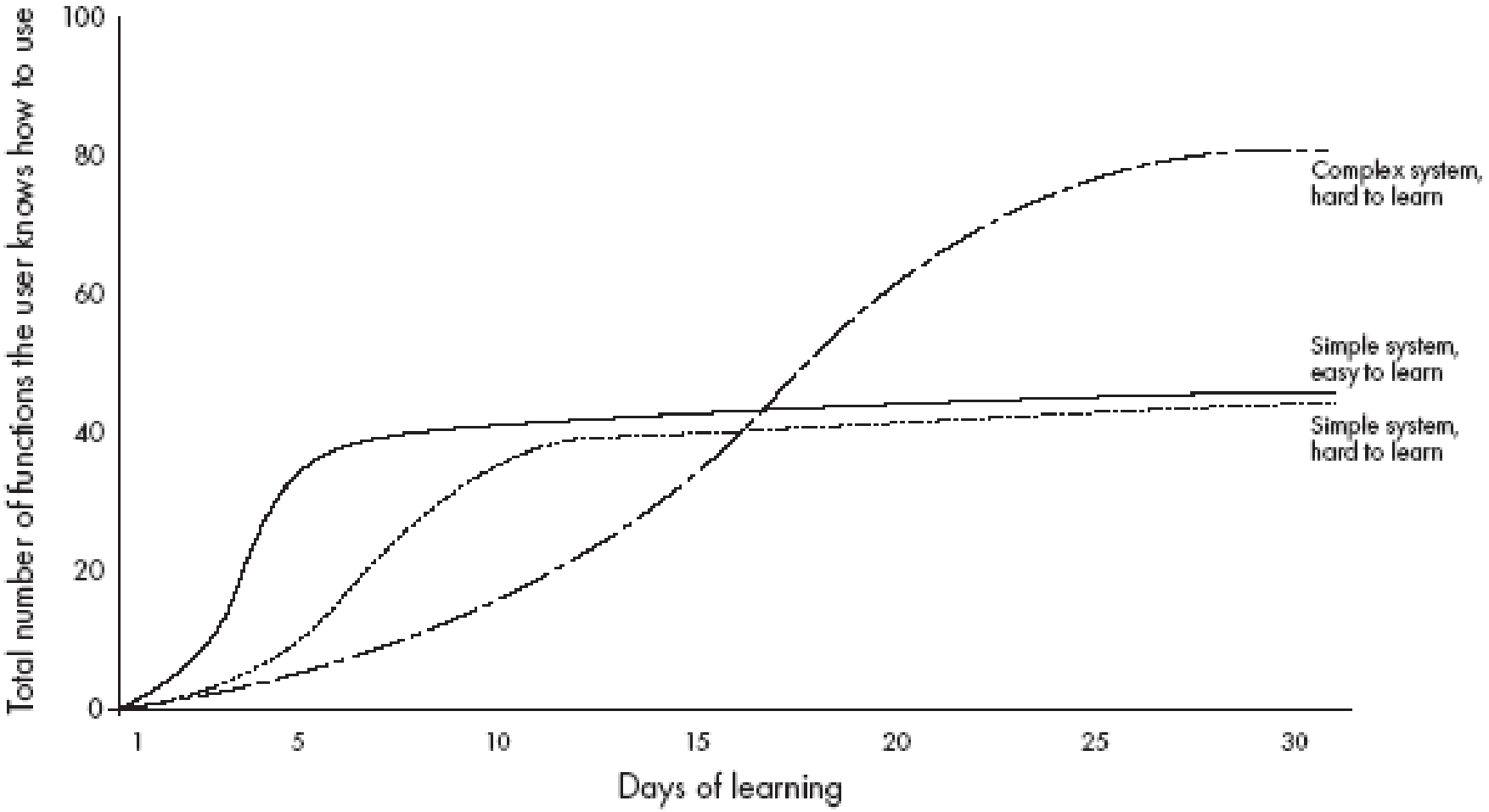
- User interface design should be done in conjunction with other software engineering activities.
- Do use case analysis to help define the tasks that the UI must help the user perform.
- Do *iterative* UI prototyping to address the use cases.
- Results of prototyping will enable you to finalize the requirements.

Usability vs. Utility

- Does the system provide the *raw capabilities* to allow the user to achieve their goal?
  - This is *utility*.
- Does the system allow the user to *learn and to use* the raw capabilities *easily*?
  - This is *usability*.
  - *Both utility and usability are essential*
  - They must be measured in the context of particular types of users.

Aspects of usability

- Usability can be divided into separate aspects:
  - Learnability
    - The speed with which a new user can become proficient with the system.
  - Efficiency of use
    - How fast an expert user can do their work.
  - Error handling
    - The extent to which it prevents the user from making errors, detects errors, and helps to correct errors.
  - Acceptability.
    - The extent to which users *like* the system.


Total number of functions the user knows how to use
100
80
60
40
20
0
1
5
10
15
20
25
30
Days of learning
Complex system, hard to learn
Simple system, easy to learn
Simple system, hard to learn

Some basic terminology of user interface design

- **Dialog**: A specific window with which a user can interact, but which is not the main UI window.
- **Control** or **Widget:** Specific components of a user interface.
- **Affordance**: The set of operations that the user can do at any given point in time.
- **State**: At any stage in the dialog, the system is displaying certain information in certain widgets, and has a certain affordance. Taken together these are system's user interface state
- **Mode**: A situation in which the UI restricts what the user can do.

Some basic terminology of user interface design

- **Modal dialog**: A dialog in which the system is in a very restrictive mode. The user cannot interact with any other window until he or she has dismissed the modal dialog.
- **Feedback**: The *response from the system* whenever the user does something, is called feedback.
- **Encoding techniques**. Ways of encoding information so as to communicate it to the user.

Usability Principles

1. Do not rely only on usability guidelines – *always test with users*.

- Usability guidelines have exceptions; you can only be confident that a UI is good if you test it successfully with users.

2: Base UI designs on users' *tasks*.

- Perform use case analysis to structure the UI.

3: Ensure that the sequences of actions to achieve a task are as *simple* as possible.

- Reduce the amount of reading and manipulation the user has to do.
- Ensure the user does not have to navigate anywhere to do subsequent steps of a task.

Usability Principles

4: Ensure that the user always knows what he or she can and should do next.

- Ensure that the user can see *what commands are available* and are not available.
- Make the *most important commands stand out*.

5: Provide good feedback including effective error messages.

- Inform users of the *progress* of operations and of their *location* as they navigate.
- When something goes wrong explain the situation in adequate detail and *help the user to resolve the problem*.

Usability Principles

6: Ensure that the user can always get out, go back or undo an action.

- Ensure that all operations can be *undone*.
- Ensure it is easy to *navigate back* to where the user came from.

7: Ensure that response time is adequate.

- Users are very sensitive to slow response time
  - They compare your system to others.
- Keep response time less than a second for most operations.
- Warn users of longer delays and inform them of progress.

Usability Principles

8: Use *understandable encoding techniques*.

- Choose encoding techniques with care.
- Use labels to ensure all encoding techniques are fully understood by users.

9: Ensure that the UI's appearance is *uncluttered*.

- Avoid displaying too much information.
- Organize the information effectively.

Usability Principles

10: Consider the needs of *different groups* of users.

- Accommodate people from different *locales* and people with *disabilities*.
- Ensure that the system is usable by both *beginners* and *experts*.

11: Provide all necessary *help*.

- Organize help well.
- Integrate help with the application.
- Ensure that the help is accurate.

Usability Principles

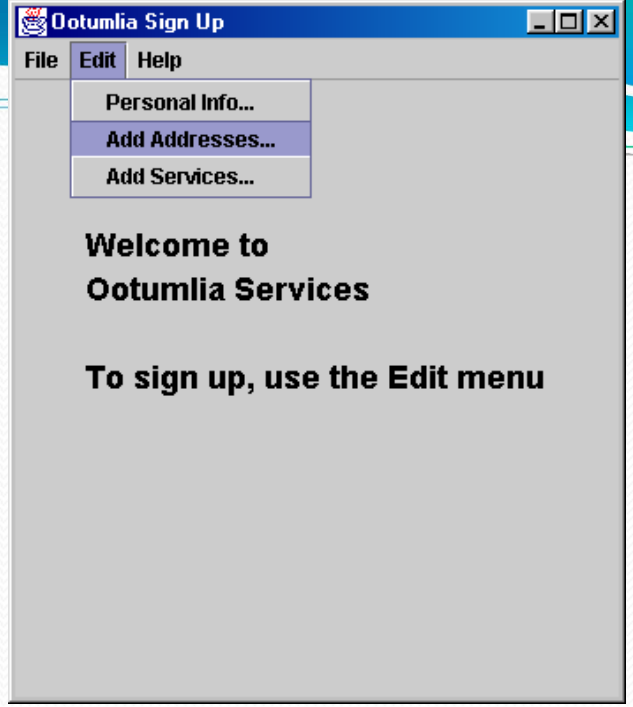
12. Be *consistent*.

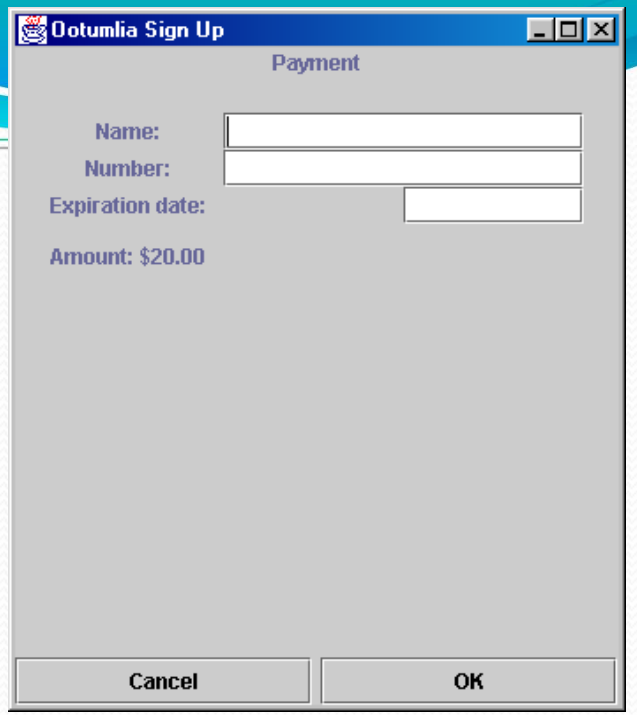
- Use similar layouts and graphic designs throughout your application.
- Follow look-and-feel standards.
- Consider mimicking other applications.

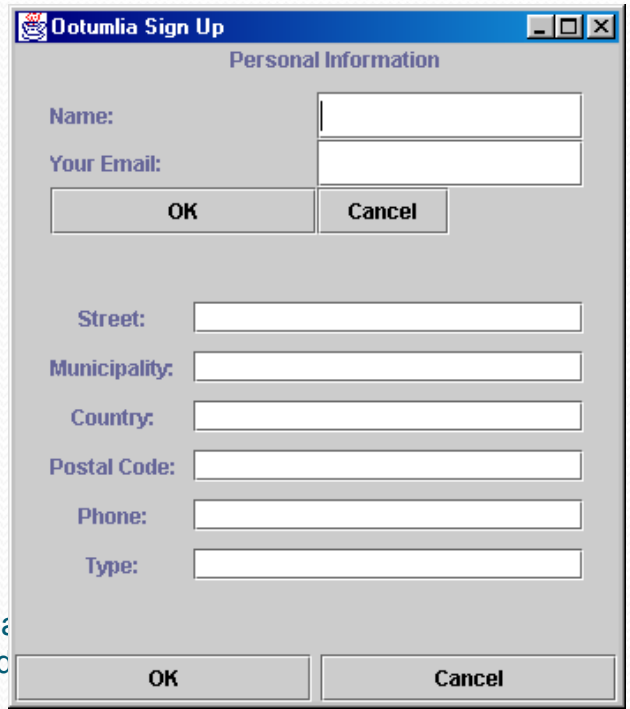
Some encoding techniques

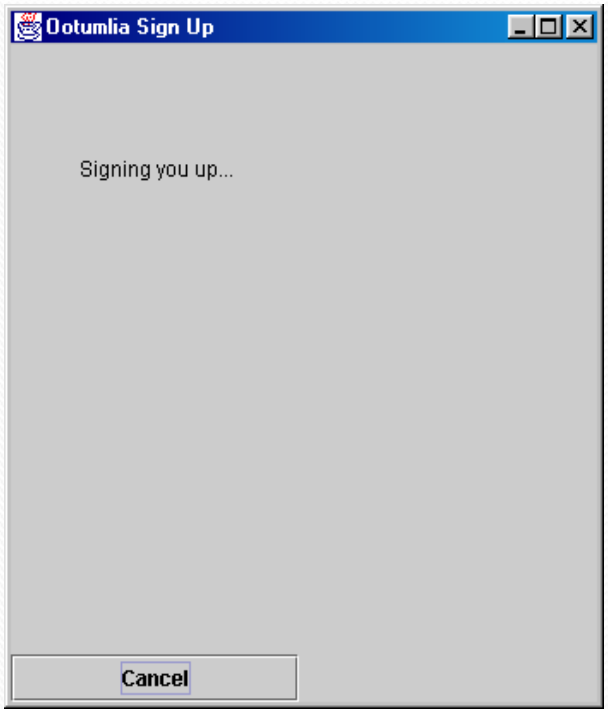
- **Text and fonts**
- **Icons**
- **Photographs**
- **Diagrams and abstract graphics**
- **Colors**
- **Grouping and bordering**
- **Spoken words**
- **Music**
- **Other sounds**
- **Animations and video**
- **Flashing**

Example (bad UI)



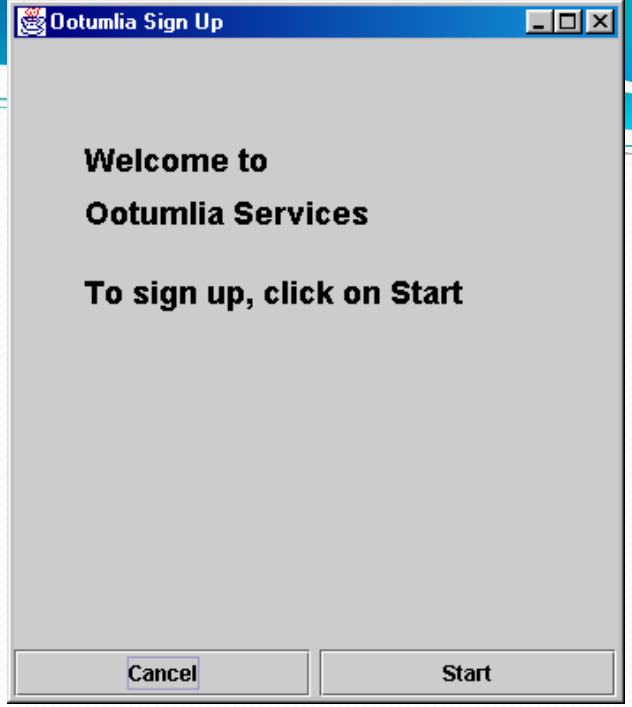


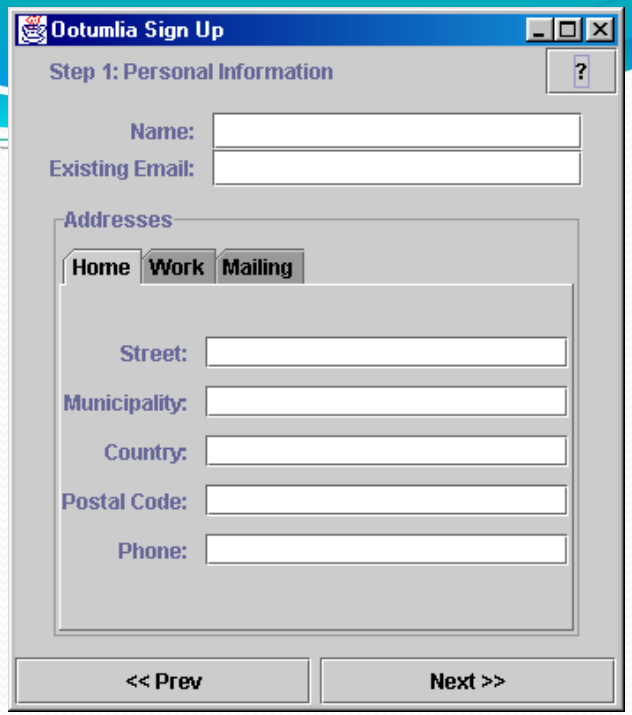


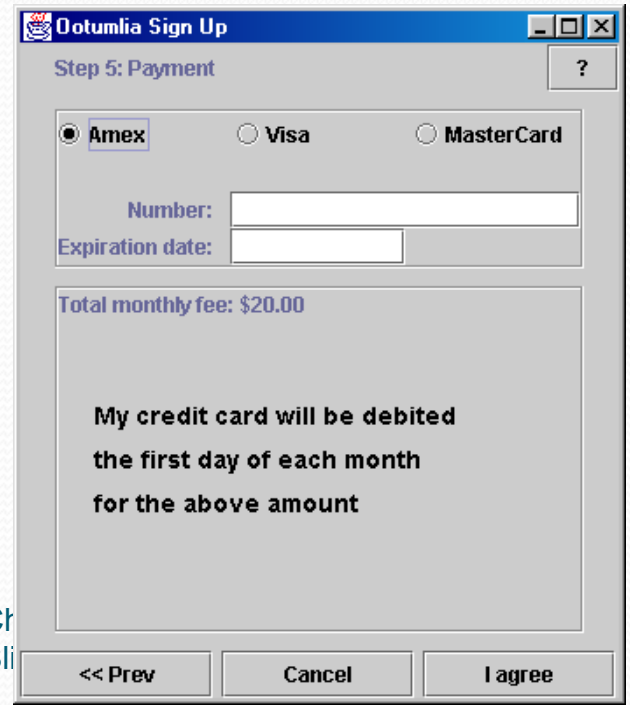


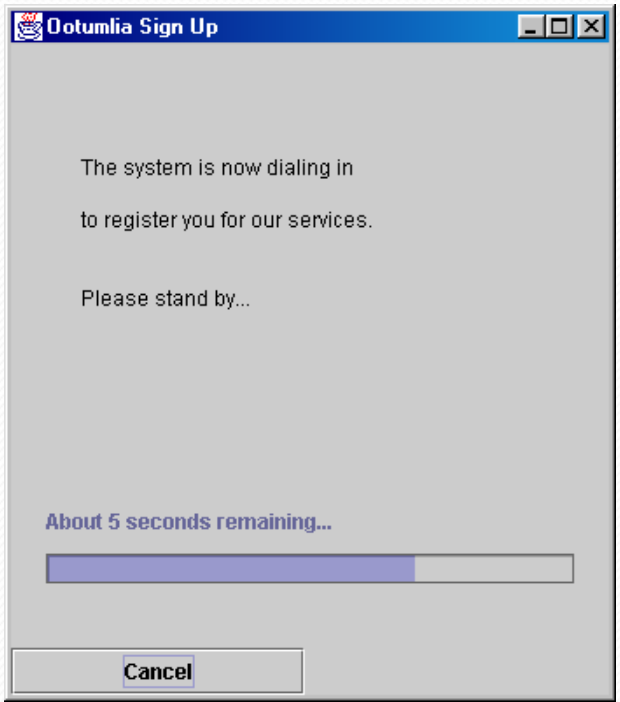
Cha
Slic

Example (better UI)









Evaluating User Interfaces

I) Heuristic evaluation

1. Pick some use cases to evaluate.
2. For each window, page or dialog that appears during the execution of the use case
   - Study it in detail to look for possible usability defects.
3. When you discover a usability defect write down the following information:
   - A short description of the defect.
   - Your ideas for how the defect might be fixed.

Evaluating User Interfaces

II) Evaluation by observation of users

- Select users corresponding to each of the most important actors
- Select the most important use cases
- Write sufficient instructions about each of the scenarios
- Arrange evaluation sessions with users
- Explain the purpose of the evaluation
- Preferably videotape each session

Evaluating User Interfaces

- Converse with the users as they are performing the tasks
- When the users finish all the tasks, de-brief them
- Take note of any difficulties experienced by the users
- Formulate recommended changes

Implementing a Simple GUI in Java

- The Abstract Window Toolkit (AWT)
  - **Component**: the basic building blocks of any graphical interface.
    - Button, TextField, List, Label, ScrollBar.
  - **Container**: contain the components constituting the GUI
    - Frame, Dialog and Panel
  - **LayoutManager**: define the way components are laid out in a container.
    - GridLayout, BorderLayout


```
public class ClientGUI extends Frame implements ChatIF
{
   private Button closeB = new Button("Close");
   private Button openB = new Button("Open");
   private Button sendB = new Button("Send");
   private Button quitB = new Button("Quit");
   private TextField portTxF = new TextField("12345");
   private TextField hostTxF = new TextField("localhost");
   private TextField message = new TextField();
   private Label portLB = new Label("Port: ", Label.RIGHT);
   private Label hostLB = new Label("Host: ", Label.RIGHT);
   private Label messageLB = new Label("Message: ",
Label.RIGHT);
   private List messageList = new List();
 ...
}
```

Example

```
public ClientGUI(String host, int port)
{
   super("Simple Chat");
   setSize(300,400);
   setVisible(true);
   setLayout(new BorderLayout(5,5));
   Panel bottom = new Panel();
   add("Center", messageList);
   add("South", bottom);
   bottom.setLayout(new GridLayout(5,2,5,5))
   bottom.add(hostLB);
   bottom.add(hostTxF);
   bottom.add(portLB);
   bottom.add(portTxF);
   bottom.add(messageLB);
   bottom.add(message);
   bottom.add(openB);
   bottom.add(sendB);
   bottom.add(closeB);
   bottom.add(quitB);
   ...
}
```

Example

```
sendB.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent e)
        {
                send();
        }
});
}

public void send()
{
   try
  {
      client.sendToServer(message.getText());
   }
   catch (Exception ex)
  {
      messageList.add(ex.toString());
      messageList.makeVisible(messageList.getItemCount()-1);
      messageList.setBackground(Color.yellow);
   }
}
```

Difficulties and Risks in UI Design

- **Users differ widely**

  Resolution:

  - *Account for differences among users when you design the system.*
  - *Design it for internationalization.*
  - *When you perform usability studies, try the system with many different types of users.*
- **User interface implementation technology changes rapidly**

  Resolution:

  - *Stick to simpler UI frameworks widely used by others.*
  - *Avoid fancy and unusual UI designs involving specialized controls that will be hard to change.*

Difficulties and Risk in UI design

- **User interface design and implementation can often take the majority of work in an application**:

  Resolution:

  - *Make UI design an integral part of the software engineering process.*
  - *Allocate time for many iterations of prototyping and evaluation.*

- **Developers often underestimate the weaknesses of a GUI**

  Resolution:

  - *Ensure all software engineers have training in UI development.*
  - *Always test with users.*
  - *Study the UIs of other software.*

Testing

Basic definitions

- A *failure* is an unacceptable behaviour exhibited by a system
  - The frequency of failures measures the *reliability*
  - An important design objective is to achieve a very low failure rate and hence high reliability.
  - A failure can result from a violation of an ex*plicit* or *implicit* requirement
- A *defect* is a flaw in any aspect of the system that contributes, or may potentially contribute, to the occurrence of one or more failures
  - It might take several defects to cause a particular failure
- An *error* is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect

Effective and Efficient Testing

- To test *effectively*, you must use a strategy that uncovers as many defects as possible.
- To test *efficiently*, you must find the largest possible number of defects using the fewest possible tests
  - Testing is like detective work:
    - The tester must try to understand how programmers and designers think, so as to better find defects.
    - The tester must not leave anything uncovered, and must be suspicious of everything.
    - It does not pay to take an excessive amount of time; tester has to be *efficient*.

Black-box testing

- Testers provide the system with inputs and observe the outputs
  - They can see none of:
    - The source code
    - The internal data
    - Any of the design documentation describing the system's internals

Glass-box testing

- Also called 'white-box' or 'structural' testing
- Testers have access to the system design
  - They can
    - Examine the design documents
    - View the code
    - Observe at run time the steps taken by algorithms and their internal data
  - Individual programmers often informally employ glass-box testing to verify their own code

Equivalence classes

- It is a black-box testing method
- It is inappropriate to test by *brute force*, using *every possible* input value
  - Takes a huge amount of time
  - Is impractical
  - Is pointless!
- You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
  - Such groups are called *equivalence classes*.
  - A tester needs only to run one test per equivalence class
  - The tester has to
    - understand the required input,
    - appreciate how the software may have been designed

Examples of equivalence classes

- Example: to test a Java method validMonth that takes an int argument which is supposed to correspond to a valid month.
  - The method returns true if the input is in range 1 to 12 inclusive & false otherwise
  - So Equivalence classes are: [-∞..0], [1..12], [13..∞]
  - Valid input is a month number (1-12)

Combinations of equivalence classes

- Combinatorial explosion means that you cannot realistically test every possible system-wide equivalence class.
  - If there are 4 inputs with 5 possible values there are 5^4 (i.e. 625) possible system-wide equivalence classes.
- You should first make sure that at least one test is run with every equivalence class of every individual input.
- You should also test all combinations where an input is likely to *affect the interpretation* of another.
- You should test a few other random combinations of equivalence classes.

Example equivalence class combinations

- Designing a system that is to contain info about all kinds of land vehicles, including passenger vehicles & racing vehicles
  - Such a system might require the user to enter specifications of a new type of vehicle
  - Your job is to divide this system into equivalence classes for testing

Example equivalence class combinations

- One valid input is either 'Metric' or 'US/Imperial'
  - Equivalence classes are:
    - Metric, US/Imperial, Other
- Another valid input is maximum speed: 1 to 750 km/h or 1 to 500 mph
  - Validity depends on whether metric or US/imperial
  - Equivalence classes are:
    - [-∞..0], [1..500], [501..750], [751.. ∞]
- Some test combinations
    - Metric, [1..500] valid
    - US/Imperial, [501..750] invalid
    - Metric, [501..750] valid
    - Metric, [501..750] valid

Testing at boundaries of equivalence classes

- More errors in software occur at the boundaries of equivalence classes
- The idea of equivalence class testing should be expanded to specifically test values at the extremes of each equivalence class
  - E.g. The number 0 often causes problems

- *E.g.*: If the valid input is a month number (1-12)
  - Test equivalence classes as before
  - Test 0, 1, 12 and 13 as well as very large positive and negative values

Detecting specific categories of defects

- A tester must try to uncover any defects the other software engineers might have introduced.
  - This means designing tests that explicitly try to catch a range of specific types of defects that commonly occur

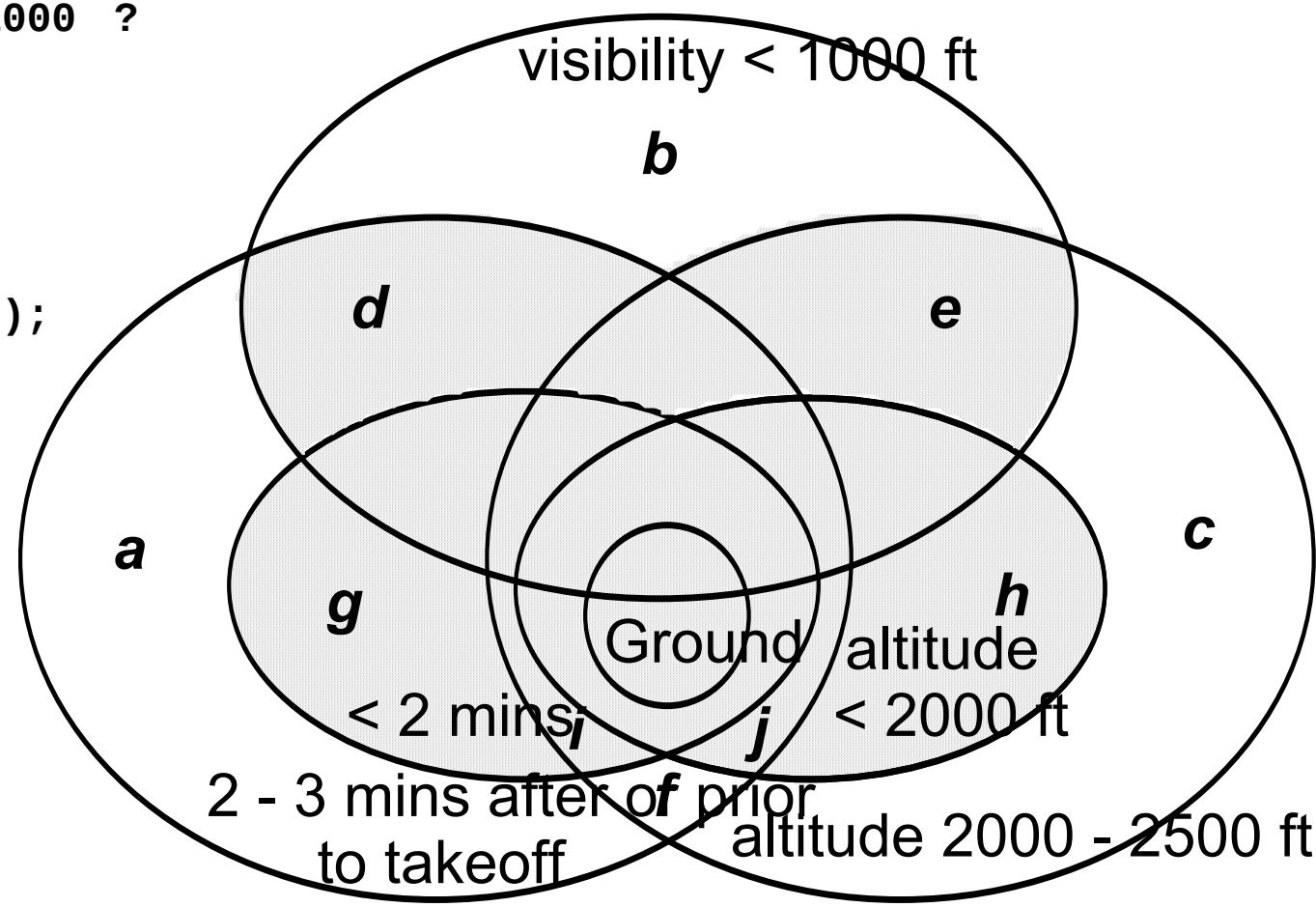
A: Defects in Ordinary Algorithms

1. Incorrect logical conditions
   - *Defect*:
     - The logical conditions that govern looping and if-then-else statements are wrongly formulated.
   - *Testing strategy*:
     - Use equivalence class and boundary testing.
     - Consider as an input each variable used in a rule or logical condition.

Example of incorrect logical conditions defect

- What is the hard-to-find defect in the following code?
  - An aircraft's alarm is supposed to sound if the landing gear is not deployed when the aircraft is close to the ground. The specifications might state this as follows:
    - The landing gear must be deployed whenever the plane is within 2 minutes from landing or takeoff, or within 2000 feet from the ground. If visibility is less than 1000 feet, then the landing gear must be deployed whenever the plane is within 3 minutes from landing or lower than 2500 feet

```
if(!landingGearDeployed &&
   (min(now-
takeoffTime,estLandTime-
now))<
     (visibility < 1000 ?
180 :120) ||
   relativeAltitude <
     (visibility < 1000 ?
2500 :2000)
  )
{
 throw
   new
LandingGearException();
}
```



Defects in Ordinary Algorithms

2. **Performing a calculation in the wrong part of a control construct**

- ***Defect***:
  - The program performs an action when it should not, or does not perform an action when it should.
  - Typically caused by inappropriately excluding or including the action from a loop or a if construct.
- ***Testing strategies***:
  - Design tests that execute each loop zero times, exactly once, and more than once.
  - Anything that could happen while looping is made to occur on the first, an intermediate, and the last iteration.
  - This kind of defect may be reliably caught by glass-box testing

Example of performing a calculation in the wrong part of a control construct

```
while(j<maximum)
{
  k=someOperation(j);
  j++;
}
if(k==-1) signalAnError();
```

Defects in Ordinary Algorithms

3. **Not terminating a loop or recursion**

- *Defect*:
  - A loop or a recursion does not always terminate, i.e. it is 'infinite'.
- *Testing strategies*:
  - Analyse what causes a repetitive action to be stopped.
  - Run test cases that you anticipate might not be handled correctly.
- Eg: in a program for counting total no of atoms in a complex organic molecule, it must be tested if the program goes into infinite loop due to circular molecular structures

Defects in Ordinary Algorithms

4. **Not setting up the correct preconditions for an algorithm**

- *Defect*:
  - *Preconditions:* state what must be true before the algorithm should be executed.
  - A defect would exist if a program proceeds to do its work, even when the preconditions are not satisfied.
- *Testing strategy*:
  - Run test cases in which each precondition is not satisfied.
  - Preferably its i/p values are just beyond what the algorithm can accept

Defects in Ordinary Algorithms

5. **Not handling null conditions**

- *Defect*:
  - A *null condition* is a situation where there normally are one or more data items to process, but sometimes there are none.
  - It is a defect when a program behaves abnormally when a null condition is encountered.

- *Testing strategy*:
  - Determine all possible null conditions & run test cases that would highlight any inappropriate behaviour.
- Eg: divide by zero error

Defects in Ordinary Algorithms

6. **Not handling singleton or non-singleton conditions**

- *Defect*:
  - A *singleton condition* occurs when there is normally *more than one* of something, but sometimes there is only one.
  - A *non-singleton condition* is the inverse.
  - Defects occur when the unusual case is not properly handled.
- *Testing strategy*:
  - Brainstorm to determine unusual conditions and run appropriate tests.
- Eg: a prog is designed to randomly assign members of a sports club into pairs who will play against each other. Does the prog do something intelligent with the left-over person when there are odd no. of members.

Defects in Ordinary Algorithms

7. Off-by-one errors

- *Defect*:
  - A program inappropriately adds or subtracts one.
  - Or loops one too many times or one too few times.
  - This is a particularly common type of defect.

- *Testing strategy*:
  - Develop boundary tests in which you verify that the program:
    - computes the correct numerical answer.
    - performs the correct number of iterations.

Example of off-by-one defect

```
Eg 1:
for (i=1; i<arrayname.length; i++)
{
  /* do something */
}
```

Use `Iterators` to help eliminate these defects

```
Eg 2:
while (iterator.hasNext())
{
  anOperation(++val);
}
```

Variable val is incremented too early so its initial value is not actually passed to anOperation

Defects in Ordinary Algorithms

8. **Operator precedence errors**

- *Defect*:
  - An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place.
  - Operator precedence errors are often extremely obvious...
    - but can occasionally lie hidden until special conditions arise.
  - E.g. If x*y+z should be x*(y+z) this would be hidden if z was normally zero.
- *Testing*:
  - In software that computes formulae, run tests that anticipate such defects.

Defects in Ordinary Algorithms

9. Use of inappropriate standard algorithms
   - *Defect*:
     - An inappropriate standard algorithm is one that is unnecessarily inefficient or has some other property that is widely recognized as being bad.
   - *Testing strategies*:
     - The tester has to know the properties of algorithms and design tests that will determine whether any undesirable algorithms have been implemented.

Example of inappropriate standard algorithms

- An inefficient sort algorithm
  - The most classical 'bad' choice of algorithm is sorting using a so-called 'bubble sort'
- An inefficient search algorithm
  - Ensure that the search time does not increase unacceptably as the list gets longer
  - Check that the position of the searched item does not have a noticeable impact on search time.
- A non-stable sort
- A search or sort that is case sensitive when it should not be, or vice versa

B: Defects in Numerical Algorithms

1. **Not using enough bits or digits**

- *Defect*:
  - A system does not use variables capable of representing the largest values that could be stored.
  - When the capacity is exceeded, an unexpected exception is thrown, or the data stored is incorrect.
- *Testing strategies*:
  - Test using very large numbers to ensure the system has a wide enough margin of error.

Defects in Numerical Algorithms

2. **Not using enough places after the decimal point or significant figures**

- *Defects*:
  - A floating point value might not have the capacity to store enough significant figures.
  - A fixed point value might not store enough places after the decimal point.
  - A typical manifestation is excessive rounding.
- *Testing strategies*:
  - Perform calculations that involve many significant figures, and large differences in magnitude.
  - Verify that the calculated results are correct.

Defects in Numerical Algorithms

3. **Ordering operations poorly so errors build up**

- *Defect*:
  - A large number does not store enough significant figures to be able to accurately represent the result.
  - This defect occurs when you do small operations on large floating point numbers & excessive rounding or truncation errors build up.
- *Testing strategies*:
  - Make sure the program works with inputs that have large positive and negative exponents.
  - Have the program work with numbers that vary a lot in magnitude.
  - Make sure computations are still accurately performed.

Defects in Numerical Algorithms

4. **Assuming a floating point value will be exactly equal to some other value**

- *Defect*:
  - If you perform an arithmetic calculation on a floating point value, then the result will very rarely be computed exactly.
  - To test equality, you should always test if it is within a small range around that value.
- *Testing strategies*:
  - Standard boundary testing should detect this type of defect.

Example of defect in testing floating value equality

Bad:

```
for (double d = 0.0; d != 10.0; d+=2.0) {...}
```

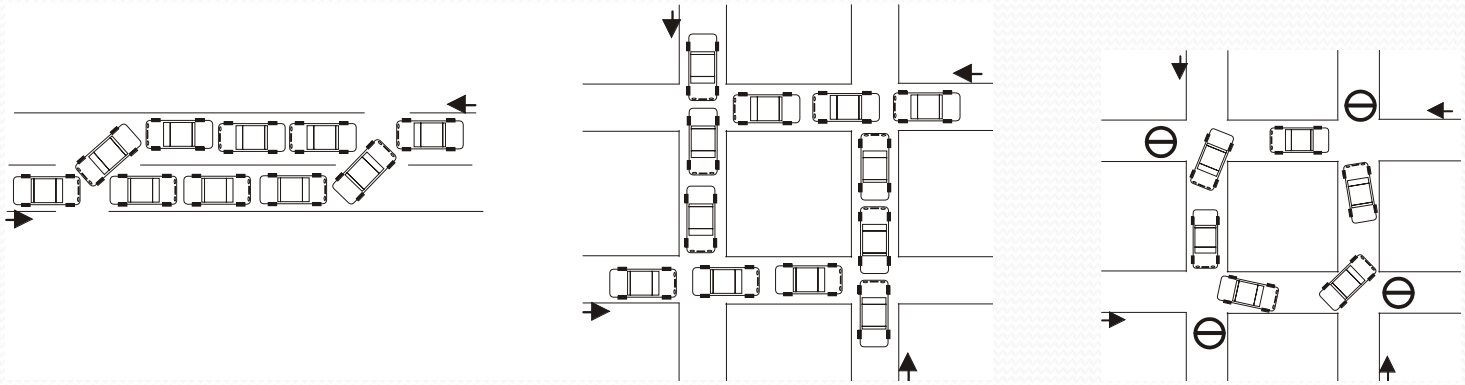
Better:

```
for (double d = 0.0; d < 10.0; d+=2.0) {...}
```

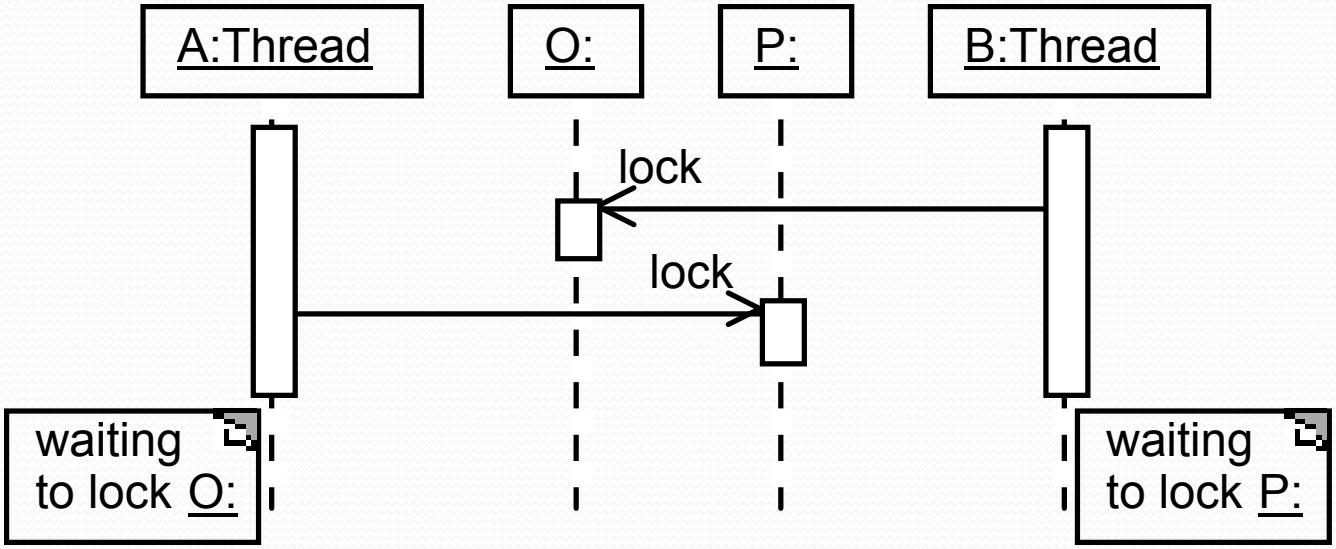
C: Defects in Timing and Co-ordination

1. **Deadlock and livelock**

- *Defects*:
  - A deadlock is a situation where two or more threads are stopped, waiting for each other to do something.
    - The system is hung
  - Livelock is similar, but now the system can do some computations, but can never get out of some states.



Example of deadlock



Defects in Timing and Co-ordination

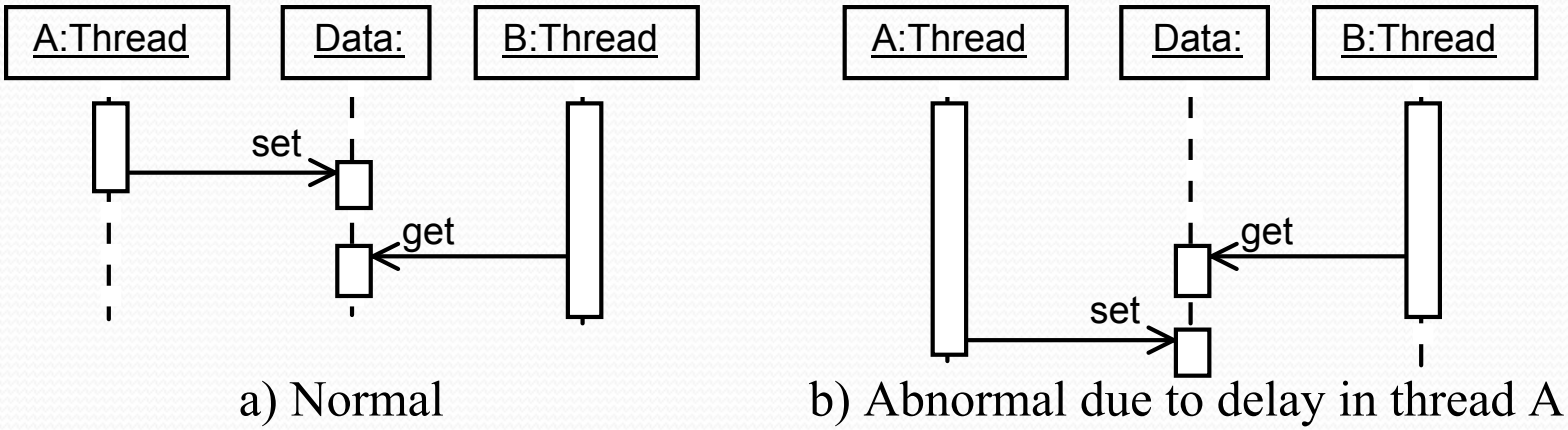
- Deadlock and livelock
  - ***Testing strategies***:
    - Deadlocks and livelocks occur due to unusual combinations of conditions that are hard to anticipate or reproduce.
    - It is often most effective to use *inspection* to detect such defects, rather than testing alone.
    - However, when doing black-box testing:
      - Vary the time consumption of different threads.
      - Run a large number of threads concurrently.
      - Deliberately deny resources to one or more threads.
    - If its cost is justifiable, glass-box testing is one of the best ways to uncover these defects

Defects in Timing and Co-ordination

2. **Critical races**

- *Defects*:
  - One thread experiences a failure because another thread interferes with the 'normal' sequence of events.
- *Testing strategies*:
  - It is particularly hard to test for critical races using black box testing alone.
  - One possible, although invasive, strategy is to deliberately slow down one of the threads.
  - Use inspection.

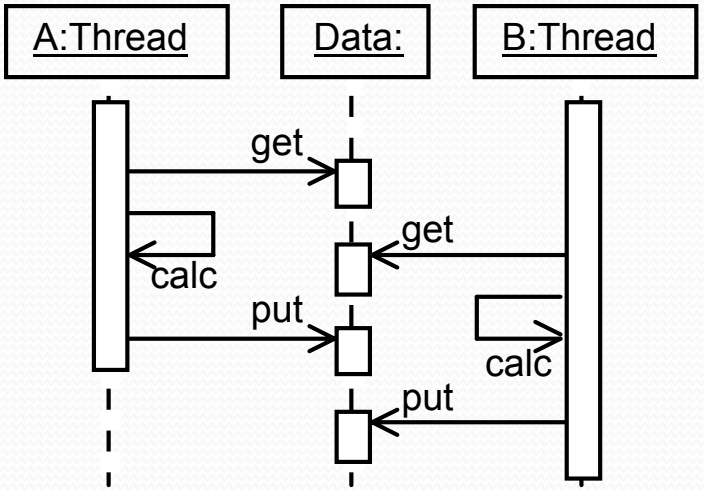
Example of critical race



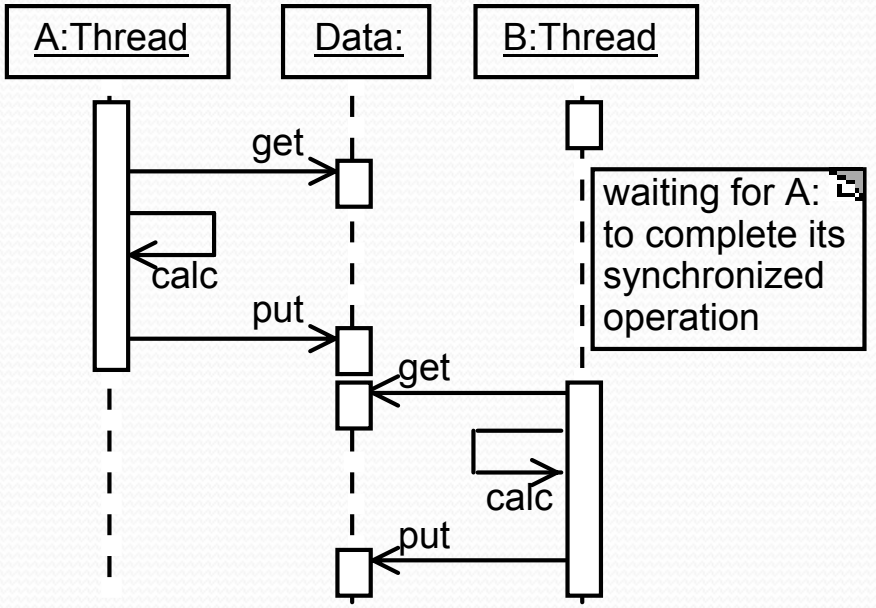
Semaphore and synchronization

- Critical races can be prevented by *locking* data so that they cannot be accessed by other threads when they are not ready
  - One widely used locking mechanism is called a *semaphore*.
  - In Java, the **`synchronized`** keyword can be used.
    - It ensures that no other thread can access an object until the synchronized method terminates.

Example of a synchronized method



a) Abnormal: The value put by thread A is immediately overwritten by the value put by thread B.



b) The problem has been solved by accessing the data using synchronized methods

D: Defects in Handling Stress and Unusual Situations

1. **Insufficient throughput or response time on minimal configurations**
   - *Defect*:
     - On a minimal configuration, the system's throughput or response time fail to meet requirements.
   - *Testing strategy*:
     - Perform testing using minimally configured platforms.

Defects in Handling Stress and Unusual Situations

2. **Incompatibility with specific configurations of hardware or software**

- *Defect*:
  - The system works properly with particular configurations of hardware, operating systems and external libraries but fails if it is run using other configurations
- *Testing strategy*:
  - Extensively execute the system with all possible configurations that might be encountered by users.
- Eg: a system might fail if a different graphics card is installed.

Defects in Handling Stress and Unusual Situations

3. **Defects in handling peak loads or missing resources**

- *Defects*:
  - The system does not gracefully handle resource shortage.
  - Resources that might be in short supply include:
    - memory, disk space or network bandwidth, permission.
  - The program being tested should report the problem in a way the user will understand.
- *Testing strategies*:
  - Devise a method of denying the resources.
  - Run a very large number of copies of the program being tested, all at the same time.

Defects in Handling Stress and Unusual Situations

4. **Inappropriate management of resources**
   - *Defect*:
     - A program uses certain resources but does not make them available when it no longer needs them.
     - A memory leak is a special case of inappropriate management of resources
   - *Testing strategy*:
     - Run the program intensively in such a way that it uses many resources, relinquishes them and then uses them again repeatedly.
   - Eg: a program might open many files, but not close them so as to enable other programs to open them.

Defects in Handling Stress and Unusual Situations

5. **Defects in the process of recovering from a crash**

- *Defects*:
  - Any system will undergo a sudden failure if its hardware fails, or if its power is turned off.
  - It is a defect if the system is left in an unstable state and hence is unable to fully recover.
  - It is also a defect if a system does not correctly deal with the crashes of related systems.
- *Testing strategies*:
  - Kill a program at various times during execution.
  - Try turning the power off, however operating systems themselves are often intolerant of doing that.

E: Documentation defects

- ***Defect***:
  - The software has a defect if the user manual, reference manual or on-line help:
    - gives incorrect information
    - fails to give information relevant to a problem.
- ***Testing strategy***:
  - Examine all the end-user documentation, making sure it is correct.
  - Work through the use cases, making sure that each of them is adequately explained to the user.

Writing Formal Test Cases and Test Plans

- A *test case* is an explicit set of instructions designed to detect a particular class of defect in a software system.
  - A test case can give rise to many tests.
  - Each test is a particular running of the test case on a particular version of the system.

Test plans

- A *test plan* is a document that contains a complete set of test cases for a system
  - Along with other information about the testing process.
- The test plan is one of the standard forms of documentation.
- If a project does not have a test plan:
  - Testing will inevitably be done in an ad-hoc manner.
  - Leading to poor quality software.
- The test plan should be written long before the testing starts.
- You can start to develop the test plan once you have developed the requirements.

Information to include in a formal test case

A. Identification and classification:

- Each test case should have a number, and may also be given a descriptive title.
- The system, subsystem or module being tested should also be clearly indicated.
- The importance of the test case should be indicated.

B. Instructions:

- Tell the tester exactly what to do.
- The tester should not normally have to refer to any documentation in order to execute the instructions.

Information to include in a formal test case

C. Expected result:

- Tells the tester what the system should do in response to the instructions.
- The tester reports a failure if the expected result is not encountered.

D. Cleanup (when needed):

- Tells the tester how to make the system go 'back to normal' or shut down after the test.

Levels of importance of test cases

- Level 1:
  - First pass critical test cases.
  - Designed to verify the system runs and is safe.
  - No further testing is possible.
- Level 2:
  - General test cases.
  - Verify that day-to-day functions correctly.
  - Still permit testing of other aspects of the system.
- Level 3:
  - Detailed test cases.
  - Test requirements that are of lesser importance.
  - The system functions most of the time but has not yet met quality objectives.

Determining test cases by enumerating attributes

- **It is important that the test cases test every aspect of the requirements.**
  - Each detail in the requirements is called an *attribute*.
    - An attribute can be thought of as something that is testable.
    - A good first step when creating a set of test cases is to *enumerate* the attributes.
    - A way to enumerate attributes is to circle all the important points in the requirements document.
  - However there are often many attributes that are *implicit*.

Strategies for Testing Large Systems

- Big bang testing versus integration testing
  - In *big bang* testing, you take the entire system and test it as a unit
  - A better strategy in most cases is *incremental testing*:
    - You test each individual subsystem in isolation
    - Continue testing as you add more and more subsystems to the final product
    - Incremental testing can be performed *horizontally* or *vertically*, depending on the architecture

Strategies for Testing Large Systems

- Horizontal testing can be used when the system is divided into separate sub-applications
- There are several strategies for vertical incremental testing:
  - Top-down
  - Bottom-up
  - sandwich

Top down testing

- Start by testing just the user interface.
- The underlying functionality are simulated by *stubs*.
  - Pieces of code that have the same interface as the lower level functionality.
  - Do not perform any real computations or manipulate any real data.
- Then you work downwards, integrating lower and lower layers.
- The big drawback to top down testing is the cost of writing the stubs.

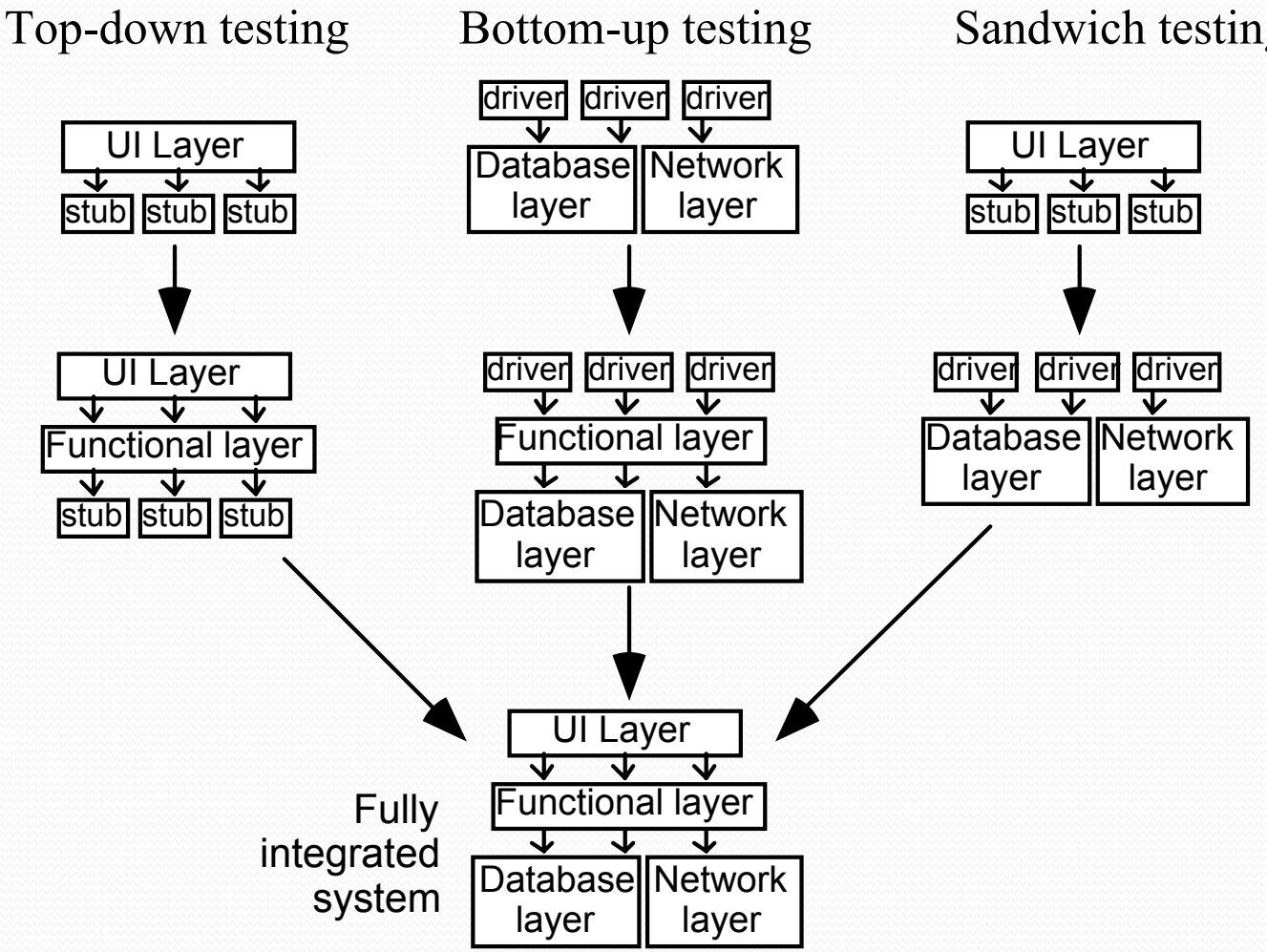
Bottom-up testing

- Start by testing the very lowest levels of the software.
- You needs *drivers* to test the lower layers of software.
  - Drivers are simple programs designed specifically for testing that make calls to the lower layers.
- Drivers in bottom-up testing have a similar role to stubs in top-down testing, and are time-consuming to write.

Sandwich testing

- Sandwich testing is a hybrid between bottom-up and top down testing.
- Test the user interface in isolation, using stubs.
- Test the very lowest level functions, using drivers.
- When the complete system is integrated, only the middle layer remains on which to perform the final set of tests.

Vertical strategies for incremental integration testing



The test-fix-test cycle

- **When a failure occurs during testing:**
  - Each failure report is entered into a failure tracking system.
  - It is then screened and assigned a priority.
  - Low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*.
  - Some failure reports might be merged if they appear to result from the same defects.
  - Somebody is assigned to investigate a failure.
  - That person tracks down the defect and fixes it.
  - Finally a new version of the system is created, ready to be tested again.

The ripple effect

- There is a high probability that the efforts to remove the defects may have actually added new defects
  - The maintainer tries to fix problems without fully understanding the ramifications of the changes
  - The maintainer makes ordinary human errors
  - The system *regresses* into a more and more failure-prone state

Regression testing

- It tends to be far too expensive to re-run every single test case every time a change is made to software.
- Hence only a subset of the previously-successful test cases is actually re-run.
- This process is called *regression testing*.
  - The tests that are re-run are called regression tests.
- Regression test cases are carefully selected to cover as much of the system as possible.

The "law of conservation of bugs":

- ***The number of bugs remaining in a large system is proportional to the number of bugs already fixed***

Deciding when to stop testing

- All of the level 1 test cases must have been successfully executed.
- Certain pre-defined percentages of level 2 and level 3 test cases must have been executed successfully.
- The targets must have been achieved and are maintained for at least two cycles of 'builds'.
  - A *build* involves compiling and integrating all the components.
  - Failure rates can fluctuate from build to build as:
    - Different sets of regression tests are run.
    - New defects are introduced.

The roles of people involved in testing

- The first pass of unit and integration testing is called *developer testing*.
  - Preliminary testing performed by the software developers who do the design.

- *Independent testing* is performed by a separate group.
  - They do not have a vested interest in seeing as many test cases pass as possible.
  - They develop specific expertise in how to do good testing, and how to use testing tools.

Testing performed by users and clients

- *Alpha testing*
  - Performed by the user or client, but under the supervision of the software development team.
- *Beta testing*
  - Performed by the user or client in a normal work environment.
  - Recruited from the potential user population.
  - An *open beta release* is the release of low-quality software to the general population.
- *Acceptance testing*
  - Performed by users and customers.
  - However, the customers do it on their own initiative.

Inspections

- An inspection is an activity in which one or more people systematically
  - Examine source code or documentation, looking for defects.
  - Normally, inspection involves a meeting…
    - Although participants can also inspect alone at their desks.

Roles on inspection teams

- The *author*
- The *moderator*.
  - Calls and runs the meeting.
  - Makes sure that the general principles of inspection are adhered to.
- The *secretary*.
  - Responsible for recording the defects when they are found.
  - Must have a thorough knowledge of software engineering.
- *Paraphrasers*.
  - Step through the document explaining it in their own words.

Principles of inspecting

- Inspect the most important documents of all types
  - code, design documents, test plans and requirements
- Choose an effective and efficient inspection team
  - between two and five people
  - Including experienced software engineers
- Require that participants prepare for inspections
  - They should study the documents prior to the meeting and come prepared with a list of defects
- Only inspect documents that are ready
  - Attempting to inspect a very poor document will result in defects being missed

Principles of inspecting

- Avoid discussing how to fix defects
  - Fixing defects can be left to the author
- Avoid discussing style issues
  - Issues like are important, but should be discussed separately
- Do not rush the inspection process
  - A good speed to inspect is
    - 200 lines of code per hour (including comments)
    - or ten pages of text per hour

Principles of inspecting

- Avoid making participants tired
  - It is best not to inspect for more than two hours at a time, or for more than four hours a day
- Keep and use logs of inspections
  - You can also use the logs to track the quality of the design process
- Re-inspect when changes are made
  - You should re-inspect any document or code that is changed more than 20%

A peer-review process

- Managers are normally not involved
  - This allows the participants to express their criticisms more openly, not fearing repercussions
  - The members of an inspection team should feel they are all working together to create a better document
  - Nobody should be blamed

Conducting an inspection meeting

1. **The moderator calls the meeting and distributes the documents.**
2. **The participants prepare for the meeting in advance.**
3. **At the start of the meeting, the moderator explains the procedures and verifies that everybody has prepared.**
4. **Paraphrasers take turns explaining the contents of the document or code, without reading it verbatim.**

   Requiring that the paraphraser not be the author ensures that the paraphraser say what he or she *sees*, not what the author *intended* to say.

5. **Everybody speaks up when they notice a defect.**

Inspecting compared to testing

- Both testing and inspection rely on different aspects of human intelligence.
- Testing can find defects whose consequences are obvious but which are buried in complex code.
- Inspecting can find defects that relate to maintainability or efficiency.
- The chances of mistakes are reduced if both activities are performed.

Testing or inspecting, which comes first?

- It is important to inspect software *before* extensively testing it.
- The reason for this is that inspecting allows you to quickly get rid of many defects.
- If you test first, and inspectors recommend that redesign is needed, the testing work has been wasted.
  - There is a growing consensus that it is most efficient to inspect software before *a*ny testing is done.
- Even before developer testing

Quality

Quality Assurance in General

- Root cause analysis
  - Determine whether problems are caused by such factors as
    - Lack of training
    - Schedules that are too tight
    - Building on poor designs or reusable technology

Measure quality and strive for continual improvement

- Things you can measure regarding the quality of a software product, and indirectly of the quality of the process
  - The number of failures encountered by users.
  - The number of failures found when testing a product.
  - The number of defects found when inspecting a product.
  - The percentage of code that is reused.
    - More is better, but don't count clones.
  - The number of questions posed by users to the help desk.
    - As a measure of usability and the quality of documentation.

Post-mortem analysis

- Looking back at a project after it is complete, or after a release,
  - You look at the design and the development process
  - Identify those aspects which, with benefit of hindsight, you could have done better
  - You make plans to do better next time

Process standards

- **The personal software process (PSP):**
  - Defines a disciplined approach that a developer can use to improve the quality and efficiency of his or her personal work.
  - Two of the key tenets is personally inspecting your own work & measuring the progress you make towards improving quality of your work.
- **The team software process (TSP):**
  - Describes how teams of software engineers can work together effectively.

Process standards

- **The software capability maturity model (CMM):**
  - Contains five levels, Organizations start in level 1, and as their processes become better they can move up towards level 5.
- **ISO 9000-3:**
  - An international standard that lists a large number of things an organization should do to improve their overall software process.

Difficulties and Risks in Quality Assurance

- It is very easy to forget to test some aspects of a software system:
  - *'running the code a few times' is not enough.*
  - *Forgetting certain types of tests diminishes the system's quality.*
- There is a conflict between achieving adequate quality levels, and 'getting the product out of the door'
  - *Create a separate department to oversee QA.*
  - *Publish statistics about quality.*
  - *Build adequate time for all activities.*

Difficulties and Risks in Quality Assurance

- People have different abilities and knowledge when it comes to quality
  - *Give people tasks that fit their natural personalities.*
  - *Train people in testing and inspecting techniques.*
  - *Give people feedback about their performance in terms of producing quality software.*
  - *Have developers and maintainers work for several months on a testing team.*

Thank You!