cs304
Software Engineering

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

Slides adapted from cs427 (UIUC) and cs409 (SUSTech)

Administrative Info

- **Final presentation Uploaded:**
  - **due on 27 May 2022, 11.59am**
    - Choose the time slot for prsentation
      - Choose the time based on the registered lab time
- All lab exercise should be submitted before next lab to avoid accumulating too much assignments
  - Reverse engineering lab not many students have submitted
- **Ask question on GitHub discussion instead of Wechat:**
  - Wechat group is for posting announcement

First SUSTech Forum on open source

会议议程

主持人开场

Tan Shin Hwei
14:00-14:10

《从Openjdk社区看基础软件开源》

杨晓峰
14:10-15:00

《第四人称单数——开源社区运作经验谈》

黄之鹏
15:00-15:50

《开源创新群智范式》

余跃
15:50-16:40

开放讨论

16:40-17:00

腾讯会议:468 994 122

开源高校行第三期参会报名统计

微信扫码或长按识别，填写内容

扫码报名

UI Design

Recap: 10 rules of Good UI Design

1. Make Everything the User Needs Readily Accessible
2. Be Consistent
3. Be Clear
4. Give Feedback
5. Use Recognition, Not Recall
6. Choose How People Will Interact First
7. Follow Design Standards
8. Elemental Hierarchy Matters
9. Keep Things Simple
10. Keep Your Users Free & In Control

https://www.elegantthemes.com/blog/resources/10-rules-of-good-ui-design-to-follow-on-every-web-design-project

Group Discussion

- Form a group with 2-3 students
- Each student identifies a mobile app that she/he feels to be the best in UI/usability
- Each student identifies a mobile app that she/he feels to be the worst in UI/usability
- Discuss in your group what UI designs make the mobile app best/worst
  - Share some commonalities in answers across students

Principle

- UI design is more like film-making than bridge-building
  - About communication
  - Requires understanding audience
  - Requires specialized skills
  - Requires iteration

Back to Joel: Golden Rules

- Let the user be in control
  - Ask the user whether he/she is sure about making this change
- Reduce the user's memory load
  - Ask for saving passwords, saving preferences, etc.
- Be consistent
  - If you are using a shortcut in one program, keep it consistent in the other programs

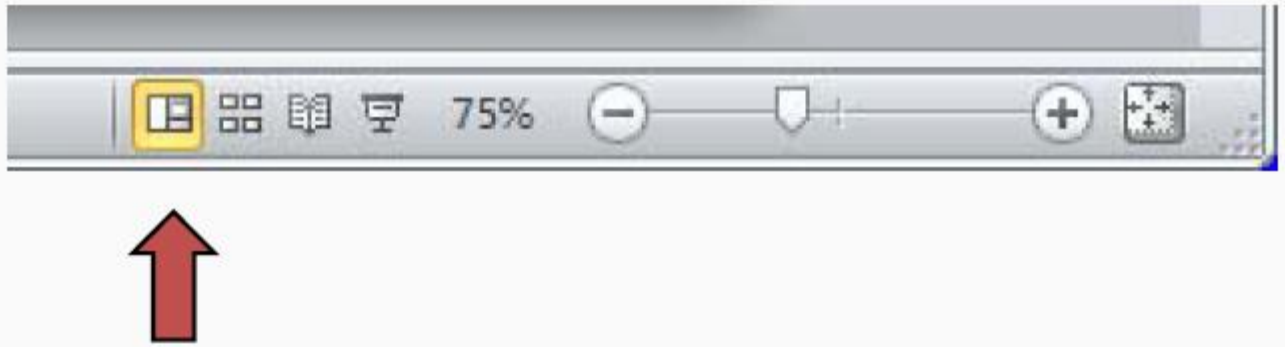
Let the User be in Control (1)

- Undo
- Macros
- Direct manipulation

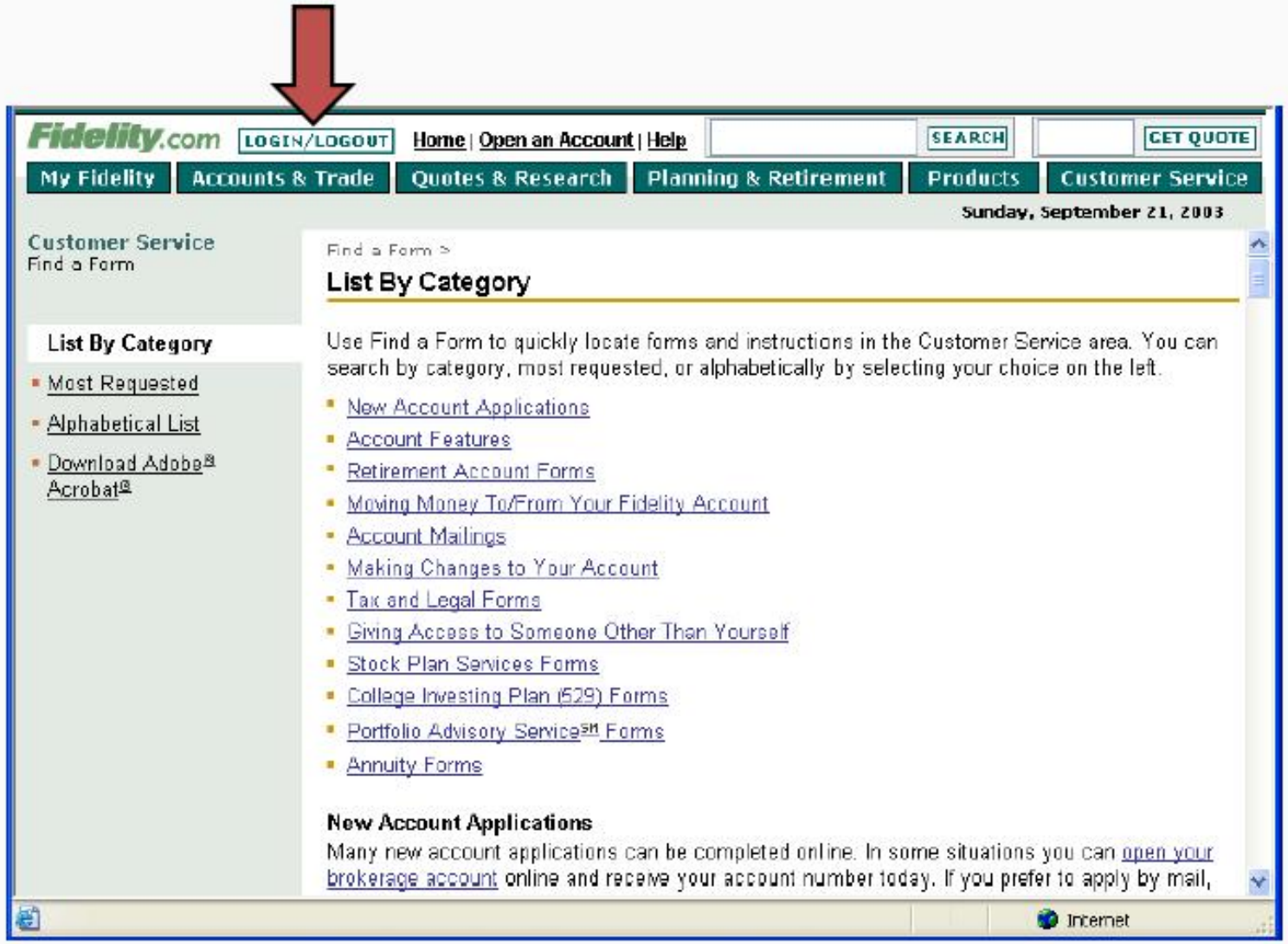
More info: http://www.cs.wm.edu/~kemper/cs435/slides/l18.pdf

Let the User be in Control (2)

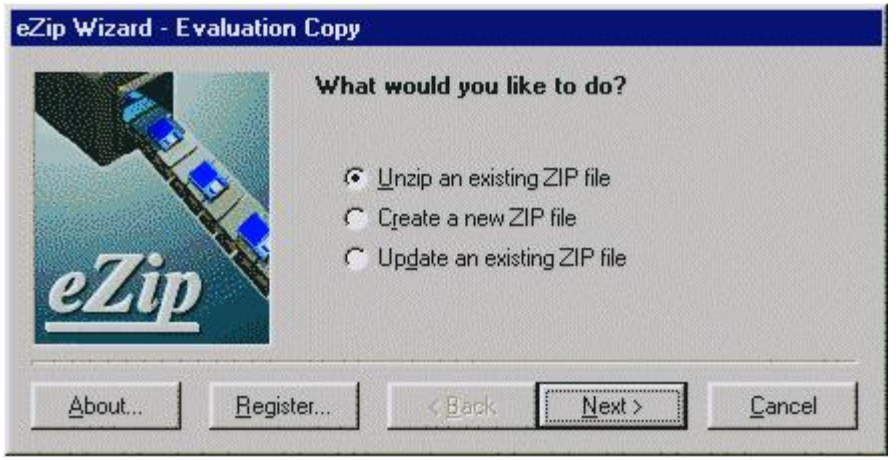
- Modes
  - Use a new window instead of a new mode
  - Make modes visible (signed in/logged out)



Let the User be in Control (3)



Wizards



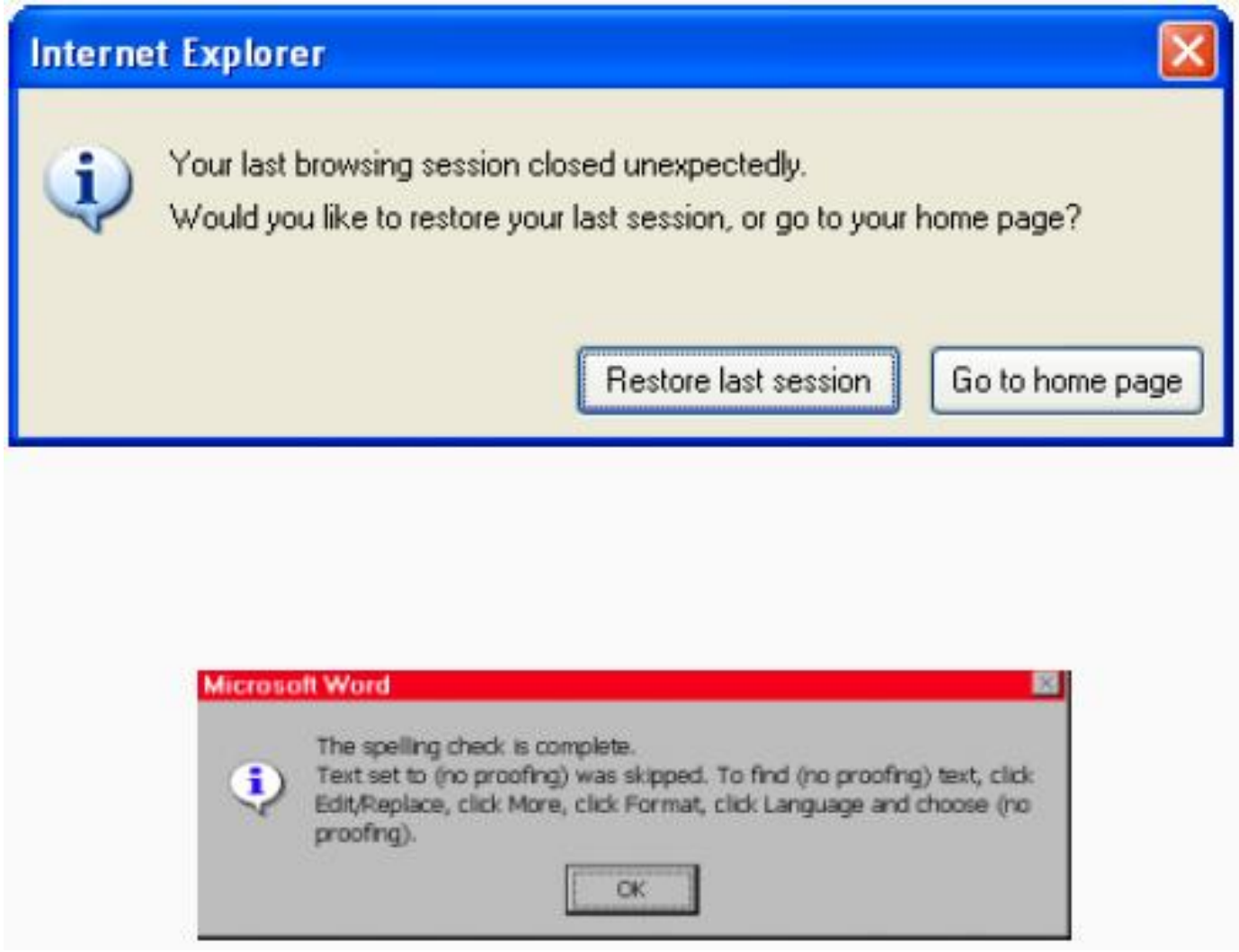
- No uncommon tasks for beginners
- Guide through steps with option to leave
- Different versions for different level of expertise

Reduce Memory Load (1)

- Reduce demand on short-term memory
- Establish meaningful defaults
- Define intuitive shortcuts
- Use real-world metaphors
- Speak user's language
- Let user recognize, not remember

TEENS REACT TO WINDOWS 95:
https://www.youtube.com/watch?v=8ucCxtgN6sc

Reduce Memory Load (2)



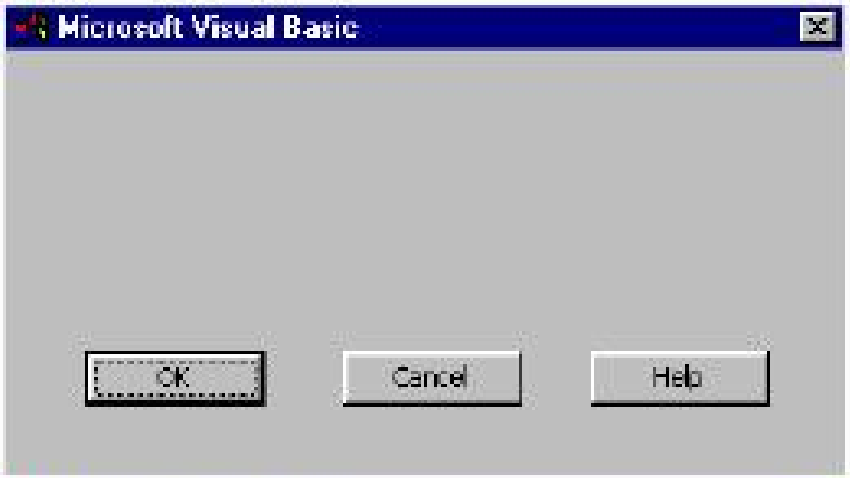
Common techniques

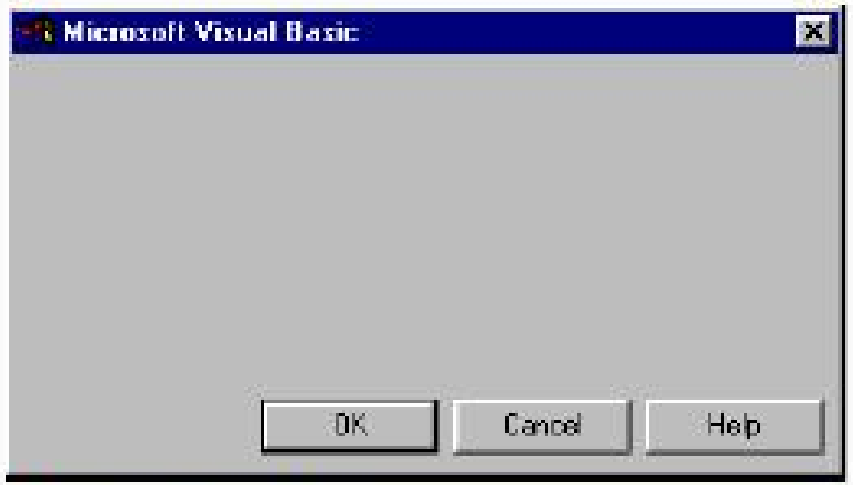
- Menus with keyboard shortcuts
- Dialog boxes
- Tabs
- Toolbar

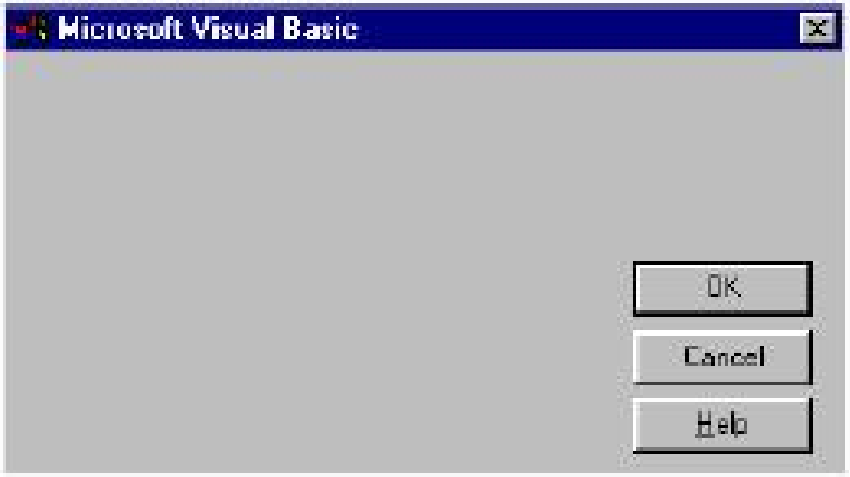
Be Consistent (1)

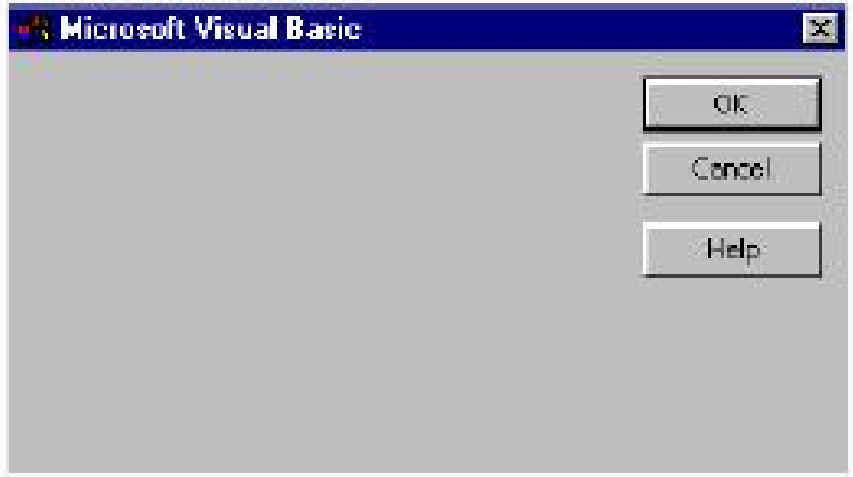
- Use visual interface standards
  - For operating system
  - For organization
  - For product or set of products
- Keep user from getting lost
- System will explain itself

Be Consistent (2)

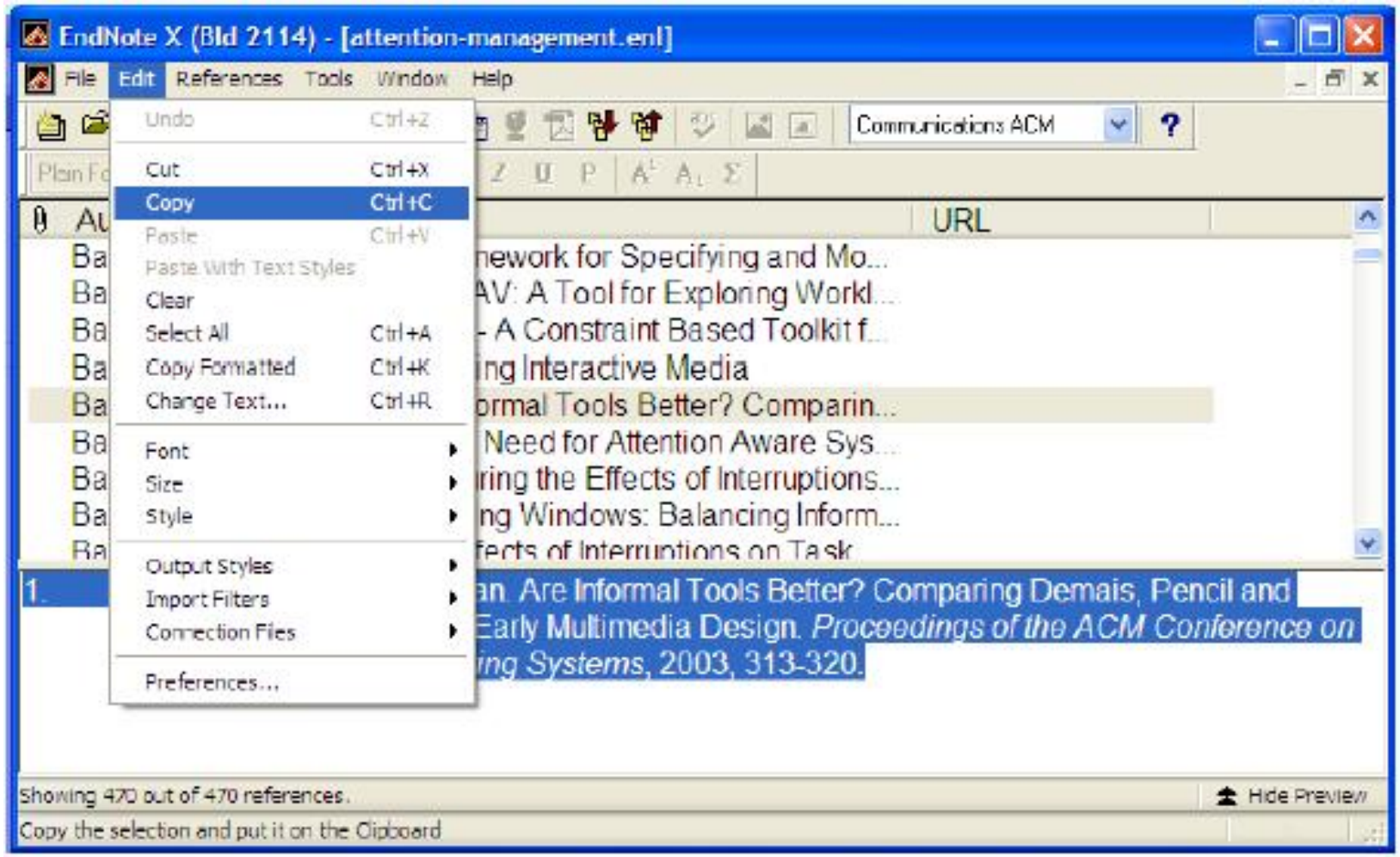




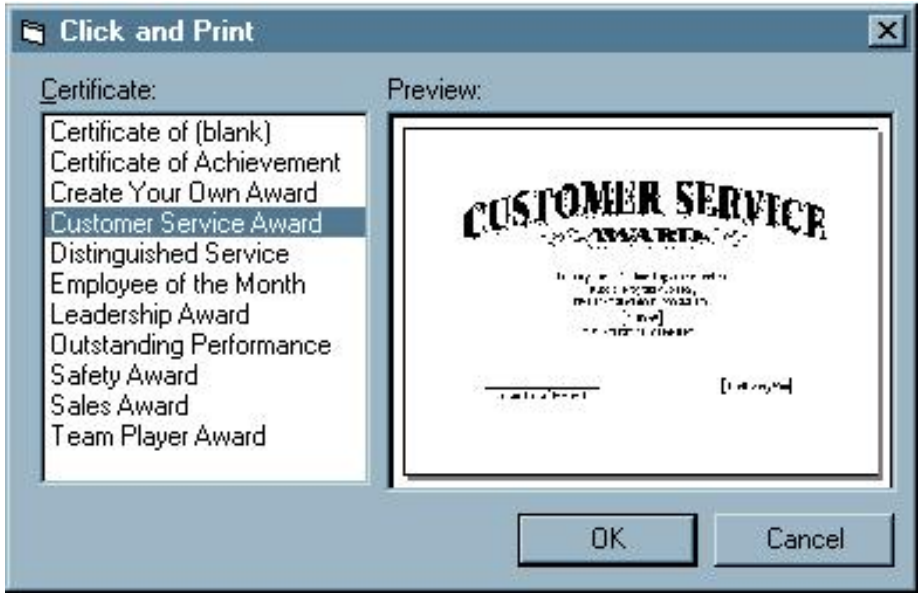




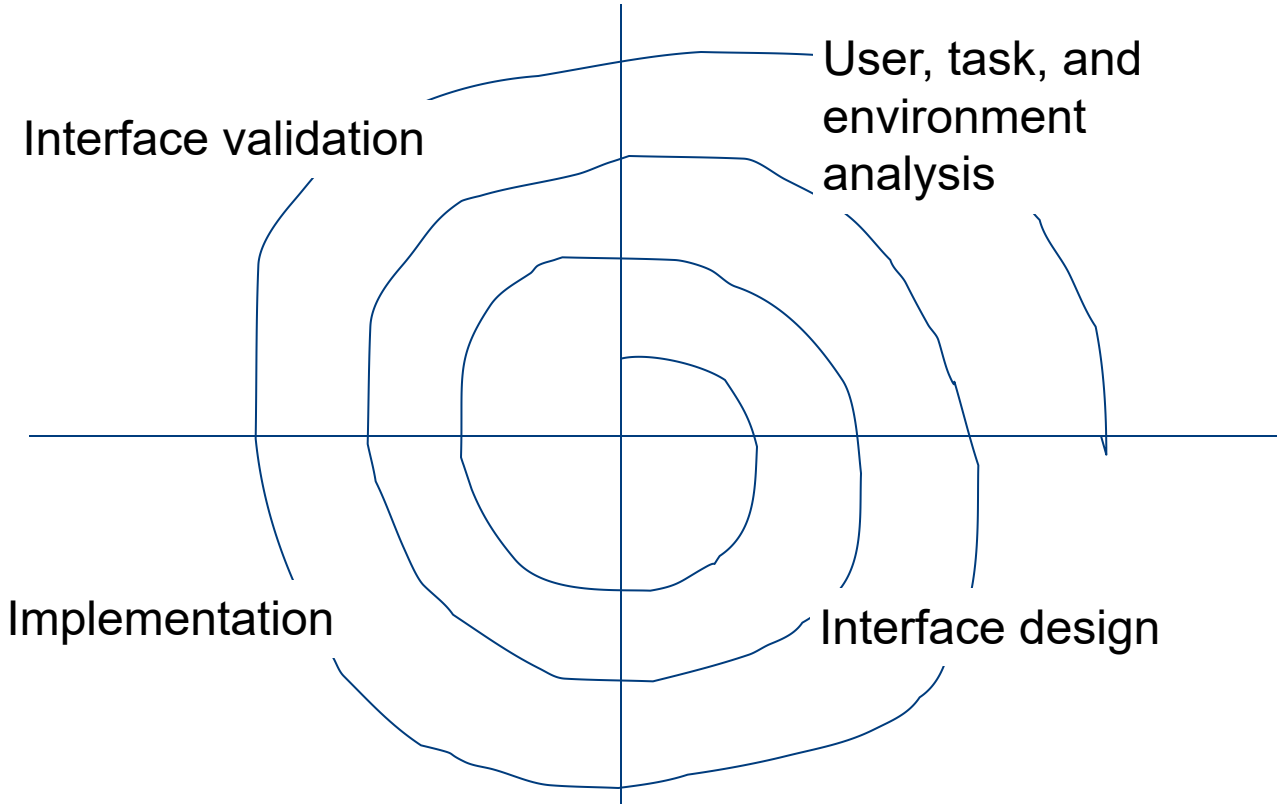
Be Consistent (3)



Easier to navigate



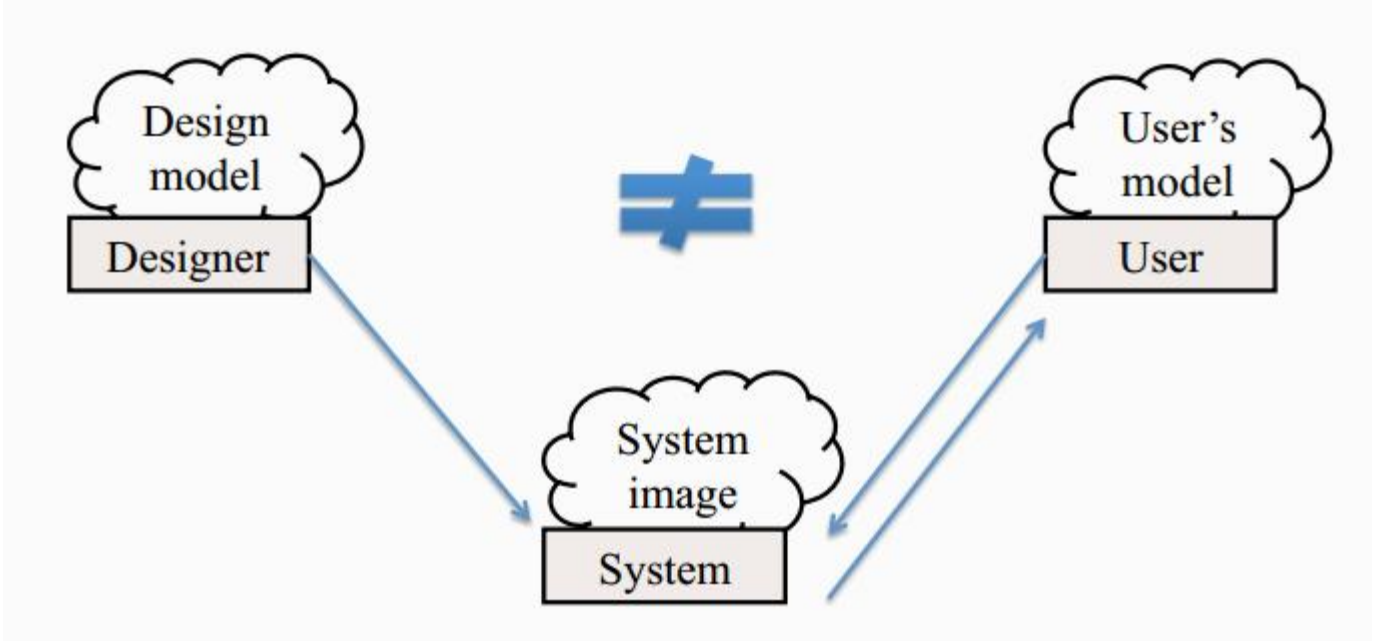
The UI Design Process



Models (1)

- Design model - what the designer thinks about the system
- User model - what the user thinks about the system
- System image - interface, manuals, training material, web site

Models (2)


Design
model
Designer
≠
User's
model
User
System
image
System

Design in nutshell

Pressman says:

> “The role of interface designer is to reconcile these differences and derive a consistent representation of the interface”

Early phases

- What are users like?
- What do they think the system should be like?
- What is a single, consistent model of the system that can satisfy all the users?

Later phases

Design

- What should system be like?
- How can we make the users understand it?
- For each aspect of the system, design the system image to match the desired user model

Validation

- Does user model match our goal?

Task Analysis and Modeling

- What tasks will a user of the system perform?

- High level - why people use the system
- Low level - tasks involved in using the system

Tasks and Use Cases

- Use cases are high-level tasks
  - Decompose high-level ones into low-level ones
  - Find ones that are missing
  - Simplify by generalizing
- UI design requires more detail than use case analysis usually provides

Tasks

- For each task:
  - Is it easy to start the task?
  - Is all the needed information easily accessible?
  - Is it easy to see what to do next?

User Interface Design

- UI communicates with the user
- Like any form of communication
  - Needs feedback and iteration
  - There are standard ways of making a UI
  - Great UIs are rare and require creativity

Stages of UX Design

From: https://www.youtube.com/watch?v=K44tP606seE

Low-level design

- Map task into actions that can be directly implemented by standard widgets
- Use consistent labels across tasks
- Use consistent widgets across tasks

E-mail

- Tasks
  - Read a message
  - Check to see if there is more mail
  - Reply to a new message
  - Send a message to a set of people
  - Stop half-way through writing a message and wait till tomorrow
  - Save a message

Design model

- Message
- Mailbox
- Incoming messages go to in-box
- Messages in out-box can be marked “ready to be sent”
- New messages created in out-box
- Can move messages from one mailbox to another

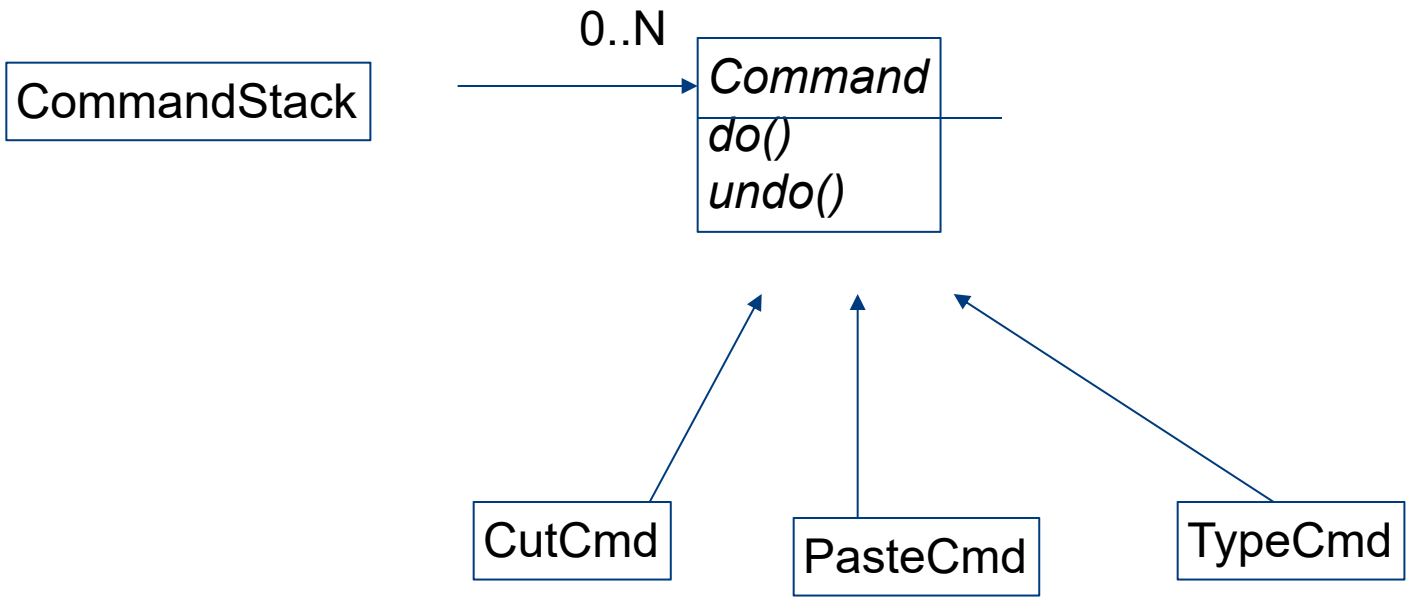
Object-oriented UI

- Objects represented by lists, icons
- Operations are on menus, buttons
- Perform operation on selected object
- Make a few operations that work on many kinds of objects
- Specify arguments by:
  - Dialog box
  - Multiple selection

Relate UI to design

- Window for domain object
  - Commands taken from operations on object
  - Direct manipulation interface
- Window for use case
  - Several windows on same domain object
  - Window might display state of use case
  - Window might display argument (dialog box)
  - Commands are steps in use case

Command design pattern



Example Code – Not Using Command Design Pattern

```
public void actionPerformed(ActionEvent e)
{
  Object o = e.getSource();
  if (o instanceof fileNewMenuItem)
       doFileNewAction();
  else if (o instanceof fileOpenMenuItem)
       doFileOpenAction();
  else if (o instanceof fileOpenRecentMenuItem)
       doFileOpenRecentAction();
  else if (o instanceof fileSaveMenuItem)
       doFileSaveAction();
  // and more ...
}
```

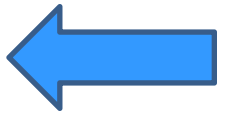
http://alvinalexander.com/java/java-command-design-pattern-in-java-examples

Example Code – Using Command Design Pattern

```
// the Command Pattern in Java
public interface Command
{
  public void execute();
}


public class FileOpenMenuItem extends JMenuItem implements Command
{
  public void execute()
  {
    // your business logic goes here
  }
}


public void actionPerformed(ActionEvent e)
{
  Command command = (Command)e.getSource();
  command.execute();
}
```



http://alvinalexander.com/java/java-command-design-pattern-in-java-examples

Joel on Software

- “In most UI decisions, before you design anything from scratch, you absolutely have to look at what other popular programs are doing and emulate that as closely as possible.”
- “Users don't have the manual, and if they did, they wouldn't read it.”
- “In fact, users can't read anything, and if they could, they wouldn't want to.”

Response times

From *Usability Engineering* by Jakob Nielsen

- 0.1 sec - limit for “instantaneous”
- 1 sec - limit for “doesn’t interrupt flow”
  - Consider progress indicator
- 10 sec - limit for “keeping attention focused on dialog”
  - Consider making it a background task

Evaluating UI

- Must evaluate UI
  - To see how to improve it
  - To see whether it is good enough to be released

UI metrics

- Size of written specification
- Number of user tasks
- Number of actions per task
- Number of system states
- Number of help messages

UI evaluation with users

- Once system has users …
  - Surveys
  - Focus groups
  - Mailing list for support
  - Analyze help desk logs
  - Analyze access logs for web apps

https://www.youtube.com/watch?v=Ml92QEqE-RQ

https://www.youtube.com/watch?v=ELYVpikRNEE

Early UI evaluation

- Have people use the system
  - Give them tasks
- Find out what is wrong with it
  - Surveys
  - Direct observation
    - Qualitative - did they seem to be having trouble?
    - Quantitative - measure time for tasks

Very early UI evaluation

- Evaluate paper prototypes
- Evaluation team
  - Person to talk to user
  - Person to record observations
  - Person to play computer
- UI made from paper, plastic (pop up menus), and colored ink

UI evaluation

- Be purposeful
  - Decide on purpose of evaluation
    - “Is this menu confusing?”
    - “Can someone start using the system without reading a manual?”
  - Choose tasks
  - Make goals and measure to see if goals are met

Size of evaluation

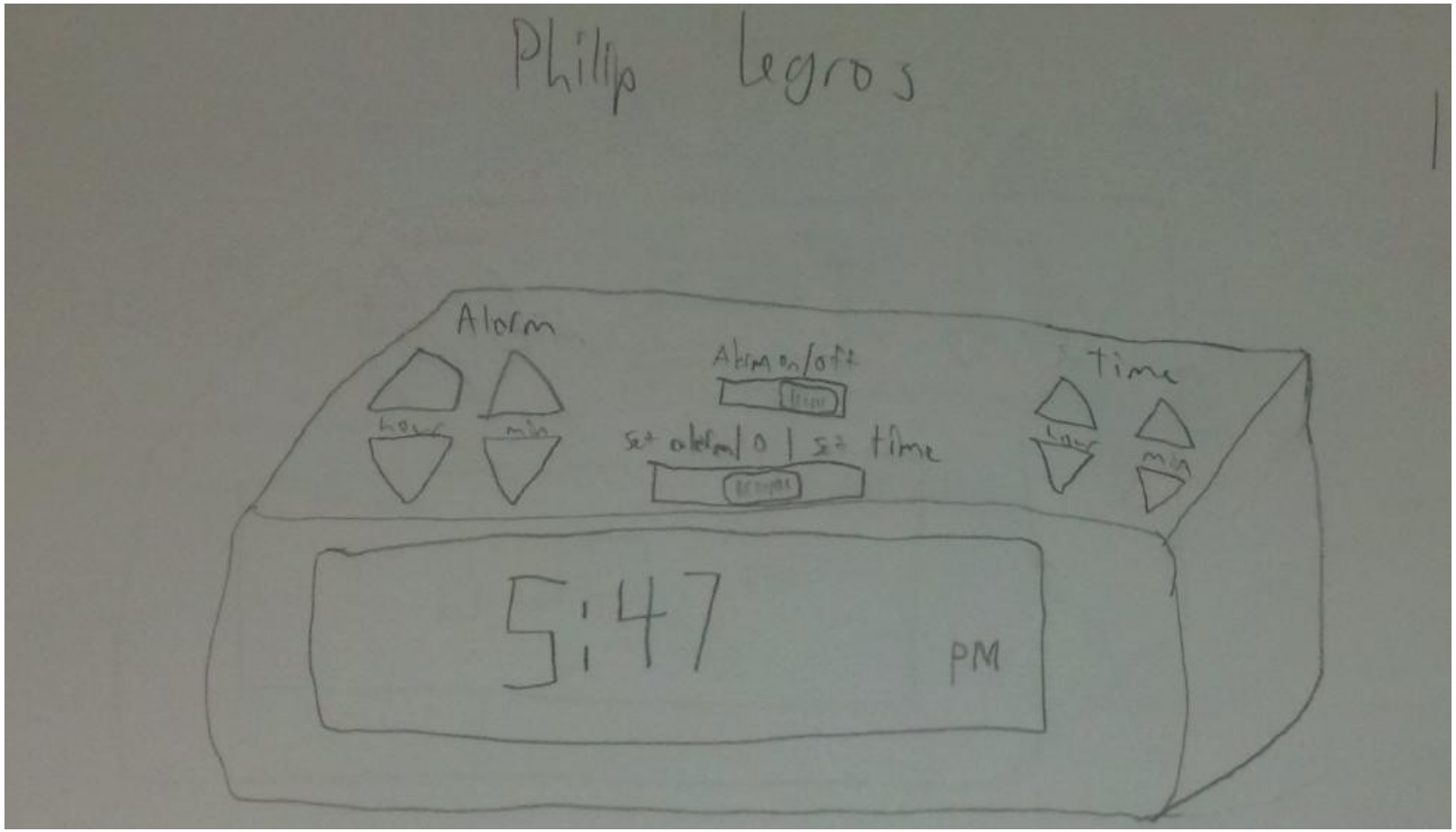
- Statistically valid sample maybe: 20-100
- Most common size: 5

- Purpose is to invent good UI
- Perform evaluations after every iteration

Alarm Clock Prototype - 1

1: Default Clock State

Alarm off, time set to 5:47PM.

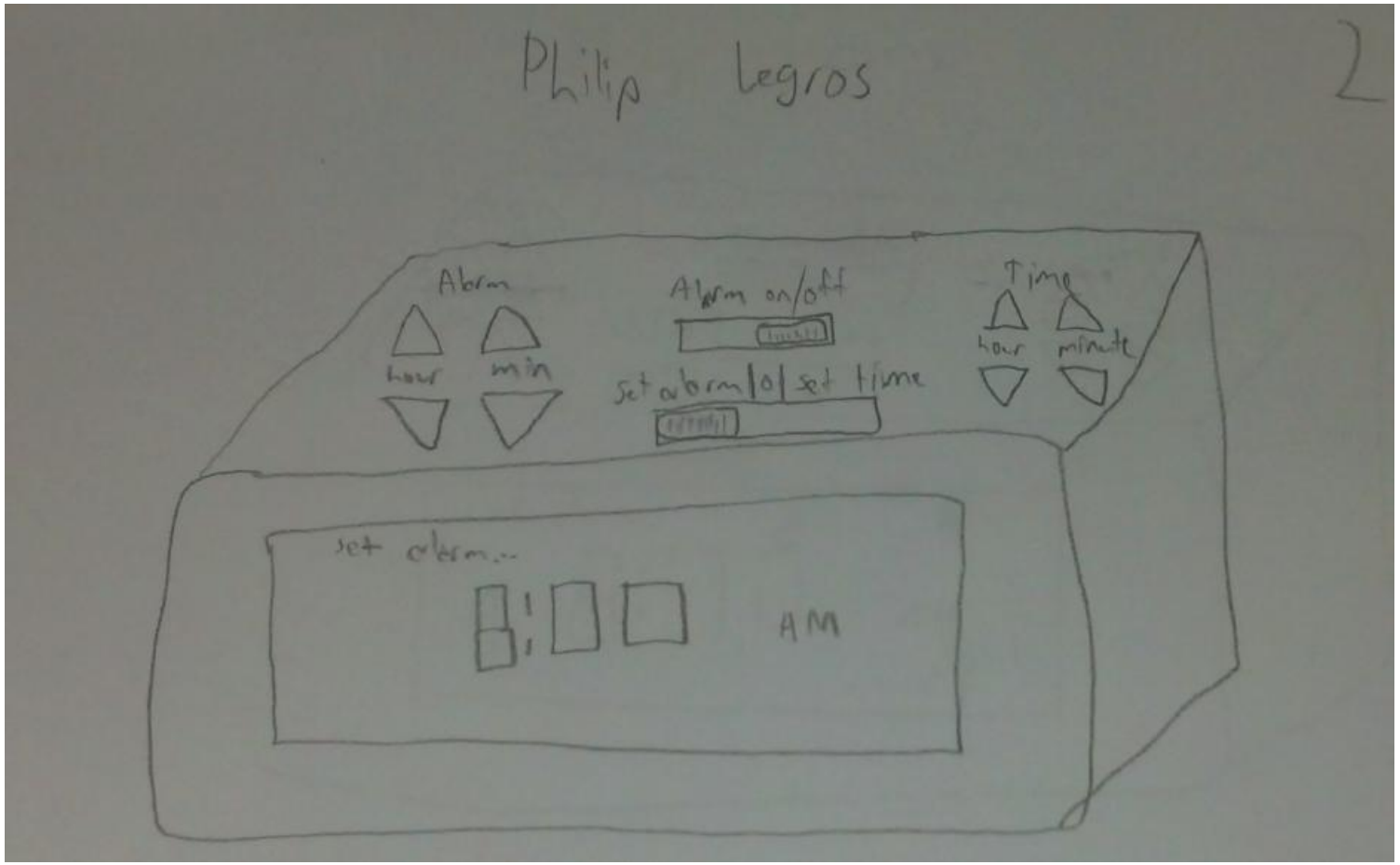


http://www.theforgeconceptual.com/alarm-clock-prototyping

Alarm Clock Prototype - 2

2: Set Alarm

User sets alarm to 8AM.

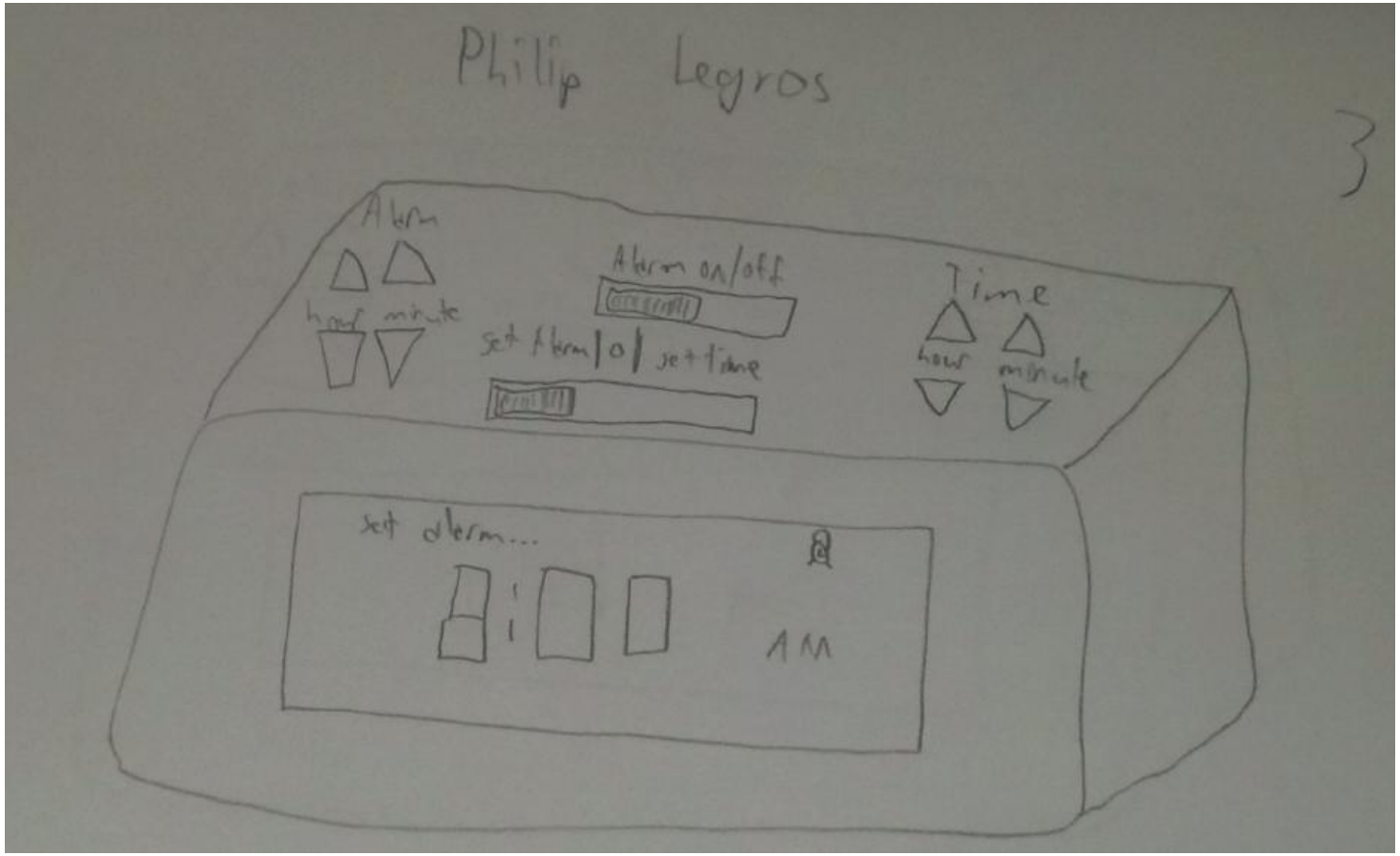


http://www.theforgeconceptual.com/alarm-clock-prototyping

Alarm Clock Prototype - 3

3: Activate Alarm

User turns on the alarm.

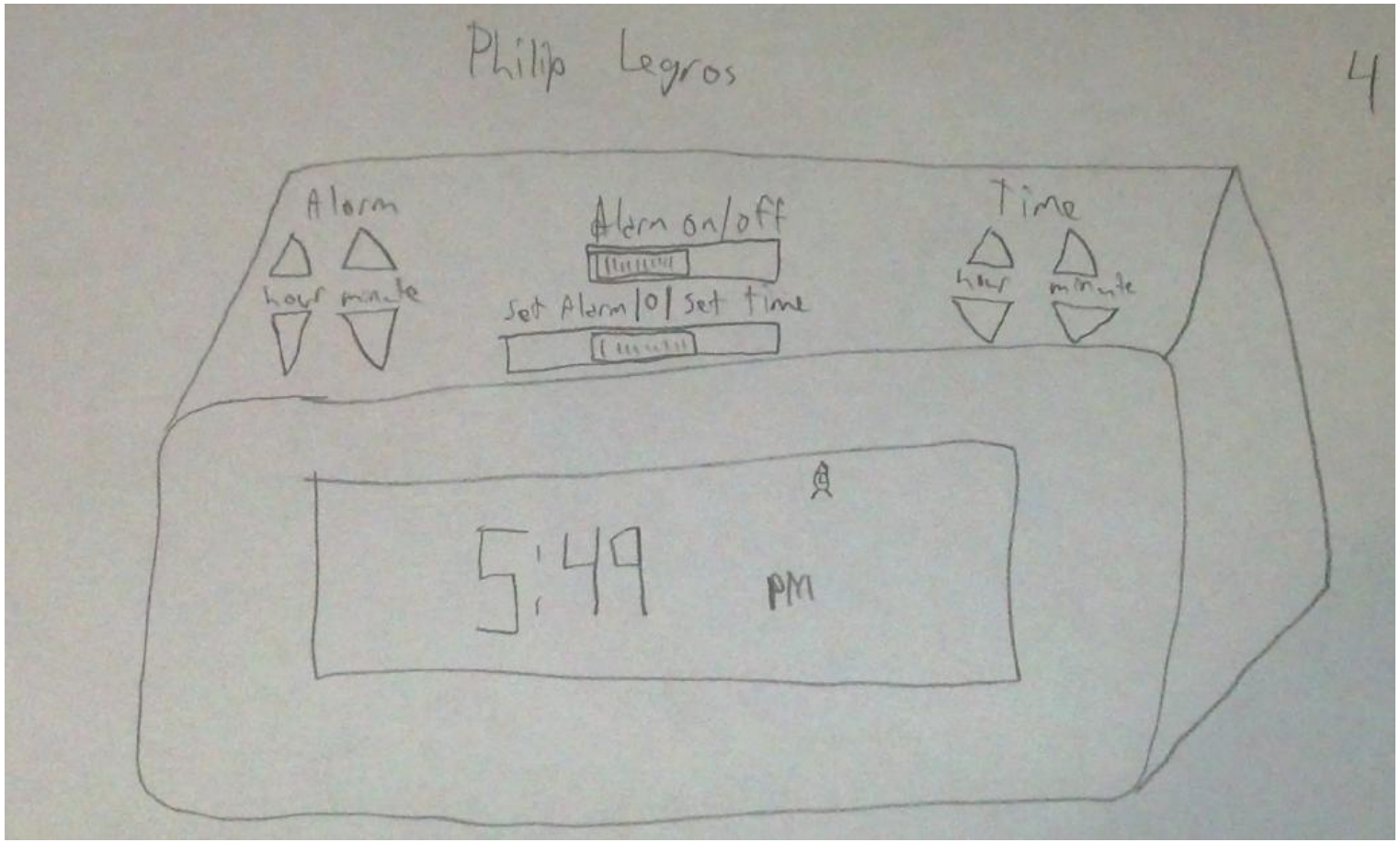


http://www.theforgeconceptual.com/alarm-clock-prototyping

Alarm Clock Prototype - 4

4: Return to Display

User returns to current time mode.



http://www.theforgeconceptual.com/alarm-clock-prototyping

Group Exercise: Paper Prototyping

- Get into group of 2 students
  - or 3 if you there are an odd number of students in the lecture
- **Make a low-fidelity UI prototype of an alarm clock smartphone app OR part of your own project (6 mins)**
  - Each of you should draw your own alarm clock at the same time; don't discuss it with your partner yet
- **Then simulate your prototype (4 mins)**, acting as the phone, while your partner acts as user. Use these tasks:
  - Is the alarm set to wake me up at 9am?
  - Suppose not; set the alarm to wake me up at 9am
  - Set the current time one hour backward for a daylight savings time switch
- Then switch roles, so that the other person acts as the phone simulating their own prototype on you

Microsoft Customer Experience Improvement Program (CEIP)

- Multiple channels for user feedback
  - Usability test, surveys, focus groups & other field studies
  - Limited customer base
- CEIP
  - Providing all customers with ability to contribute to the design and development of Microsoft products
  - Voluntary participation
  - Anonymous

CEIP data

- Usage
  - How software is used
  - Examples: general feature usage, commands on Ribbon, actions taken in wizards, etc.
- Reliability and performance
  - Whether software performs as expected
  - Examples: assertions for logical inconsistency, measuring execution speed, etc.
- Hardware/software configuration
  - Providing context for data interpretation
  - Example: long document loading time only on machines with low RAM or a particular processor speed?

Questions answered by usage data

Command usage

- How frequently is it used? *[Prominence on UI]*
- How many people use it? *[Impact]*
- What is the most frequent way of accessing it? *[Ease of access]*
- Does this command occur as part of a clear workflow? *[Better support]*

Feature usage

- How many files contain a Table? *[Impact]*
- How big is the average Table? *[Optimization choice]*
- What are the most frequently used Table styles? *[Design choices]*
- What other features are used in files containing Tables? *[interaction with other features]*

Design alternatives

- Novice users
  - Menus
  - Make it look like something else
  - Simple
- Expert users
  - Commands
  - Specialize to make users efficient
  - Powerful

Design alternatives

- Standard IO vs. new IO
- Existing metaphors/idioms vs. new metaphors/idioms
- Narrow market vs. broad market

Implementation concerns

- Simplicity
- Safety
- Use standard libraries/toolkits
- Separate UI from application
  - Model-View-Controller (MVC)
  - Three-tier: presentation, application, data

1. Simplicity

- Tradeoff between number of features and simplicity
- Don't compromise usability for function
- A well-designed interface fades into the background
- Basic functions should be obvious
- Advanced functions can be hidden

Make controls obvious & intuitive

- Is the trash-can obvious and intuitive?
- Are tabbed dialog boxes obvious and intuitive?
- Is a mouse obvious and intuitive?

2. Safety

- Make actions predictable and reversible
- Each action does one thing
- Effects are visible
  - User should be able to tell whether operation has been performed
- Undo

3. Use standard libraries

- Don’t build your own!
  - If necessary, add to it, but try to use standard parts instead of building your own
- Provide familiar controls
- Provide consistency
- Reduce cost of implementation
- Library designers probably better UI designers than you are

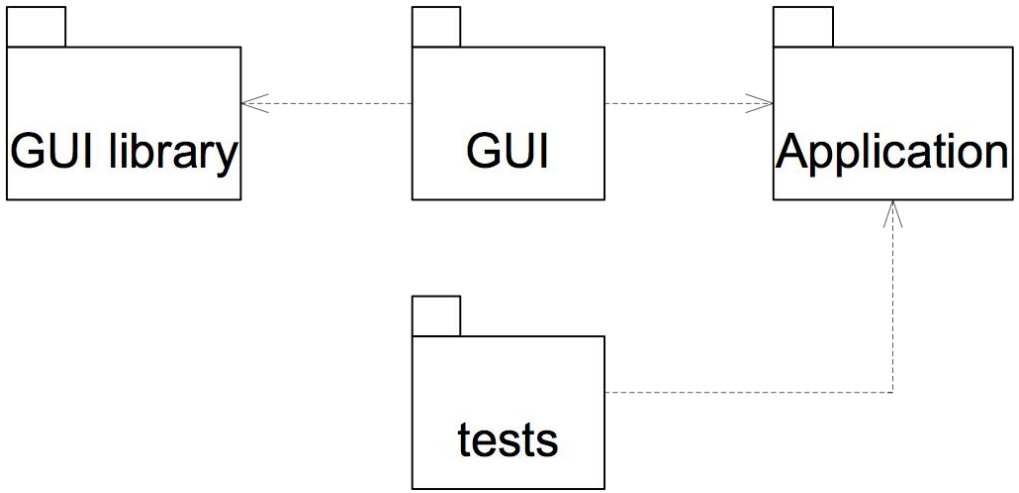
When to build your own

- You are a platform provider or
- You have special needs and a lot of money and

- You are not in a hurry and
- You know what you are doing

4. Separate UI and application

- UI and application change independently
- UI and application built by different people



UI in Web: ASP/JSP/Rails…

- Embed code in your HTML
  - VB code in ASP, Java code in JSP, Ruby code in Rails…
- Can call other code
- Need to decide on how much code goes in the web page, and how much goes outside

Separate UI from application

- HTML is UI
- Put as little code on web page as possible
- Web page has just enough code to call the actual application logic

Benefits

- Write automatic tests for application objects, not for UI
- People who write HTML don't need to know how to program well
- Programmers don't need to be good UI designers

Downside

- Application objects generate HTML
  - But you can make standard set of “adapters” and so don’t have to duplicate code
  - Lists, radio buttons, etc.
- Code tends to creep into web pages
  - Refactor
  - Review

Results

- Easier to test
  - Automatic tests for application objects
  - Test GUI manually or with automatic “smoke tests” or use something like Selenium
- Easier to change
  - Can change “business rule” independently of GUI
  - Can add web interface, speech interface, etc.

One issue: selecting colors

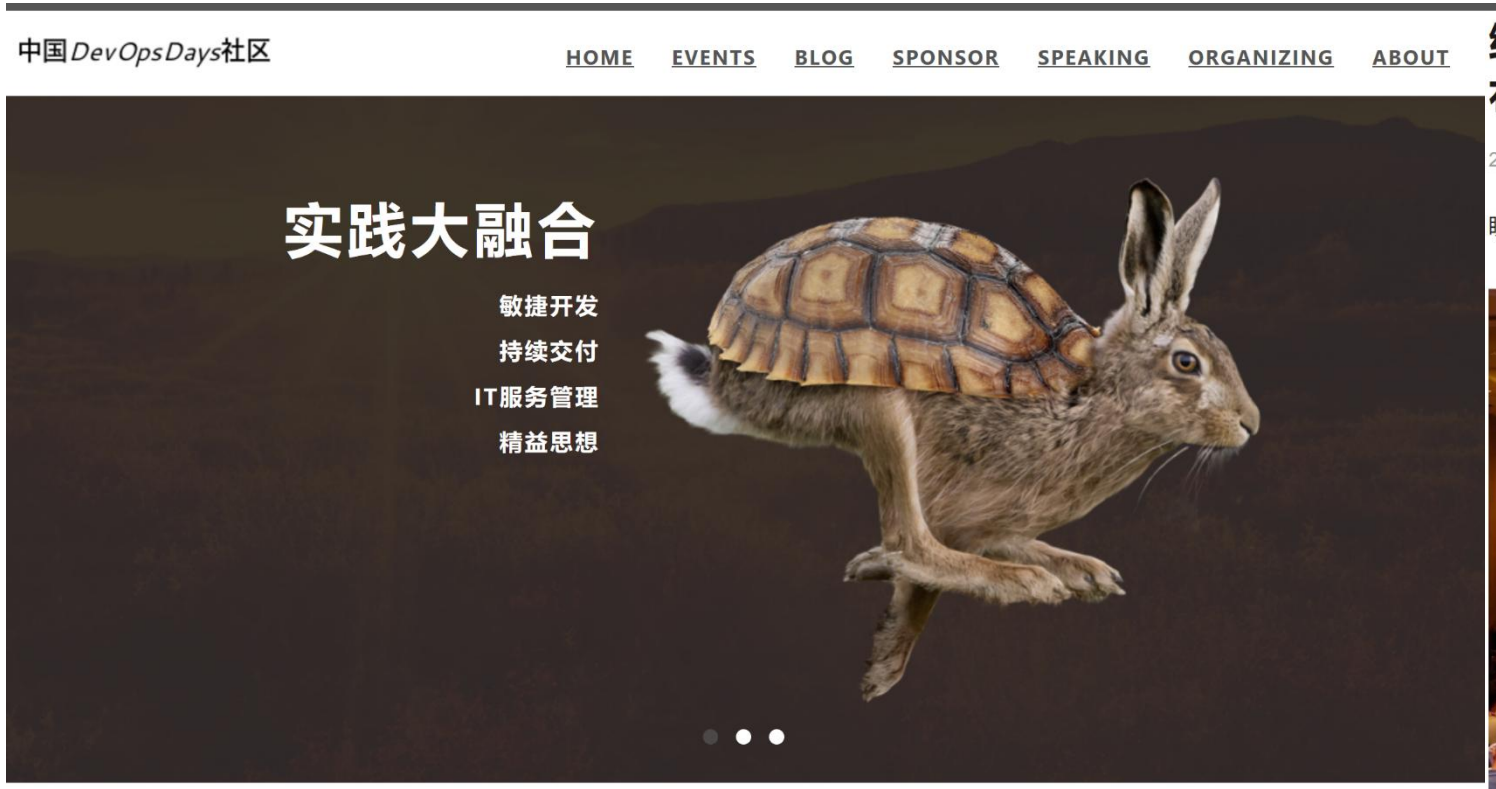
- Leave it to a graphic designer
- Use system colors (actually pull them from config)
- Use the company/university colors
- Use a color palette generator

Summary

- UI design is hard
  - Must understand users
  - Must understand problems
  - Must understand technology
  - Must understand how to evaluate
- UI design is important
  - UI is what the users see
  - UI can “make it or break it” for software

DevOps and Continuous Integration

Popularity of DevOps



From：https://chinadevopsdays.org/

终于等到你 | 国内外首个 DevOps 标准今日全量发布

2018-06-29 15:40

盼星星盼月亮，终于、首届 DevOps 国际峰会·北京站今日终于开幕了。

What is DevOps?

From: https://www.youtube.com/watch?v=_l94-tJlovg

What is DevOps?

A. Design + IT Operations
B. Design + Optimization
C. Development + IT Operations
D. Development + Optimization

What is Continuous Integration?

*"**Continuous Integration** is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."*

Benefits of Continuous Integration:

- Reduce integration problems
- Allows team to develop cohesive software more rapidly

Martin Fowler

Integration Problem

Line 2: x+y→ x + y

commit

Main Branch: Calculator.java

commit

Line 1-2: Rename x → num1,
Rename y → num2

```
1:public static int add(int x, int y){
2:      int result = x+y;
3:      return result;
4:}
```

Merge Conflict at line 2!

commit

Line 2-3: ~~int result = x+y;~~ → return x + y;
~~return result;~~

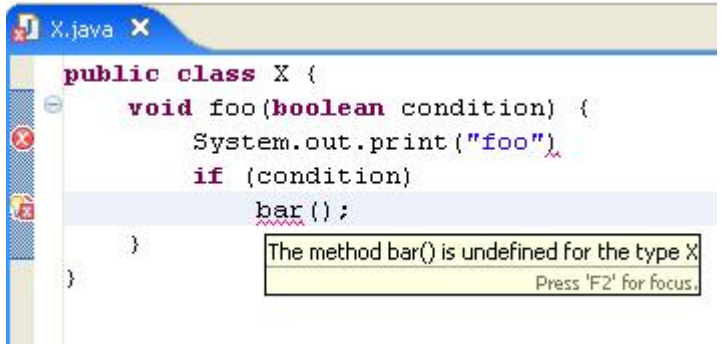
Integration Problems

Merge Conflict

- Modifying the same file concurrently

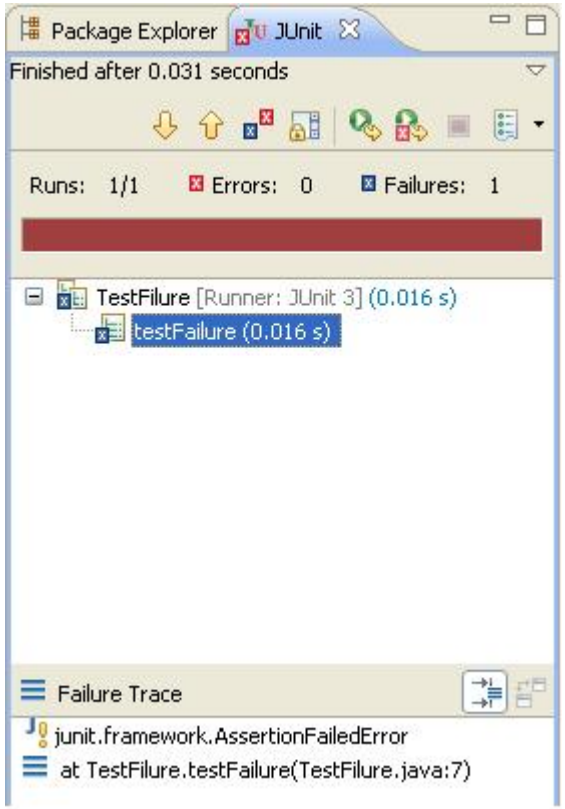
Compile Conflict

- Modified program no longer compile



Test Conflict

- Modified program compile but fails to pass the test

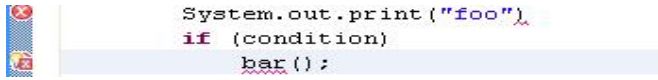


10 Principles of Continuous Integration

- Maintain a code repository – version control
- Automate the build
- Make your build self-testing
- Everyone commits to mainline every day
- Every commit should build mainline on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what's happening
- Automate deployment

Automate the build

Daily build: Compile working executable on a daily basis

- Develop scripts for compiling & running the system
  - Build Automation Tools:
    - Java: Apache Ant, Maven, Gradle
    - C/C++: make
- Benefits:
  - Allows you to test the quality of your integration
  - Visible Progress 

  - Quickly catches/exposes bug that breaks the build

Build from command line

- require `build.xml` for building
  - One common way of automated build is to allows project to be compiled from the command line.

```
<project name="simpleCompile" default="deploy" basedir=".">
   <target name="init">
      <property name="sourceDir" value="src"/ >
      <property name="outputDir" value="classes" />
      <property name="deployJSP" value="/web/deploy/jsp" />
      <property name="deployProperties" value="/web/deploy/conf" />
   </target>
   <target name="clean" depends="init">
      <deltree dir="${outputDir}" />
   </target>
   <target name="prepare" depends="clean">
      <mkdir dir="${outputDir}" />
   </target>
   <target name="compile" depends="prepare">
     <javac srcdir="${sourceDir}" destdir="${outputDir}" />
   </target>
   <target name="deploy" depends="compile,init">
     <copydir src="${jsp}" dest="${deployJSP}"/>
     <copyfile src="server.properties" dest="${deployProperties}"/>
   </target>
</project>
```

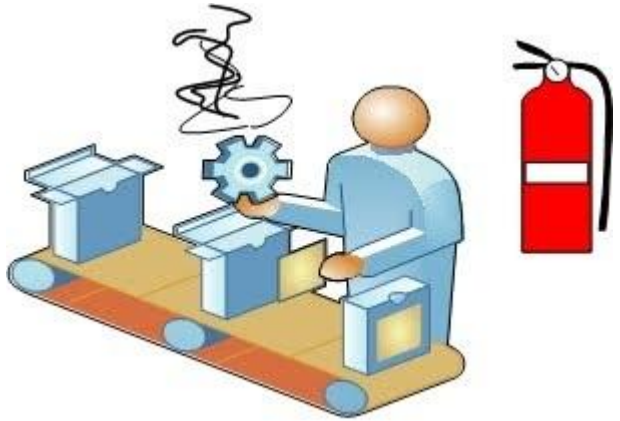
`Target:` Build Target
`depends:` Target dependency
`property:` Define a simple name value pair

Make your build self-testing

- **Automated tests:** Tests that can be run from the command line
- Examples:
  - Unit tests
  - Integration tests
  - Smoke test

Smoke Test

- A quick set of tests run on the daily build.
  - Cover most important functionalities of the software but NOT exhaustive
  - Check whether code catches fire or “smoke” (breaks)
  - Expose integration problems earlier



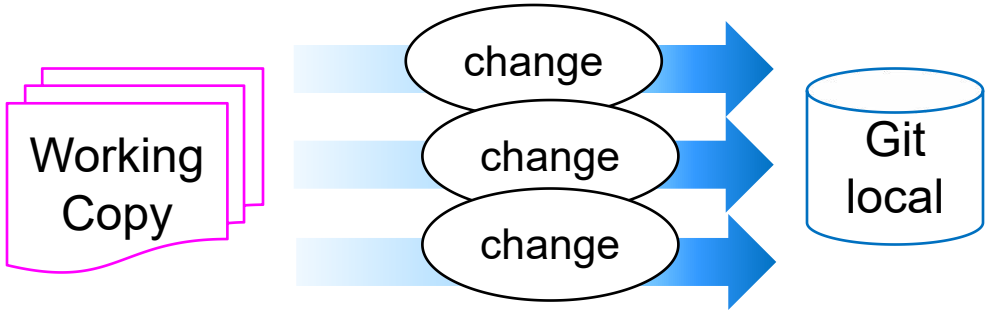
Question:

How do you perform smoke test for a new cup?

Fill it with water. If there is a leaking, then it is a bad cup

Daily Commits

- Submit work to main repo at end of each day.
  - Idea: Reduce merge conflicts
  - This is the key to "continuous integration" of new code.



  - *Caution:* Don't check in faulty code (does not compile, does not pass tests) just to maintain the daily commit practice.
  - If your code is not ready to submit at end of day, either submit a coherent subset or be flexible about commit schedule.

Question:

Does it mean that programmers need to constantly sit and wait for the build and run test? Is it possible to have someone this for me automatically?

Yes, use CI Server

Continuous Integration server

An external machine that automatically pulls your latest repo code and fully builds all resources.

- If anything fails, contacts your team (e.g. by email).
- Ensures that the build is never broken for long.

Examples of CI Server

First CI Server: *CruiseControl*

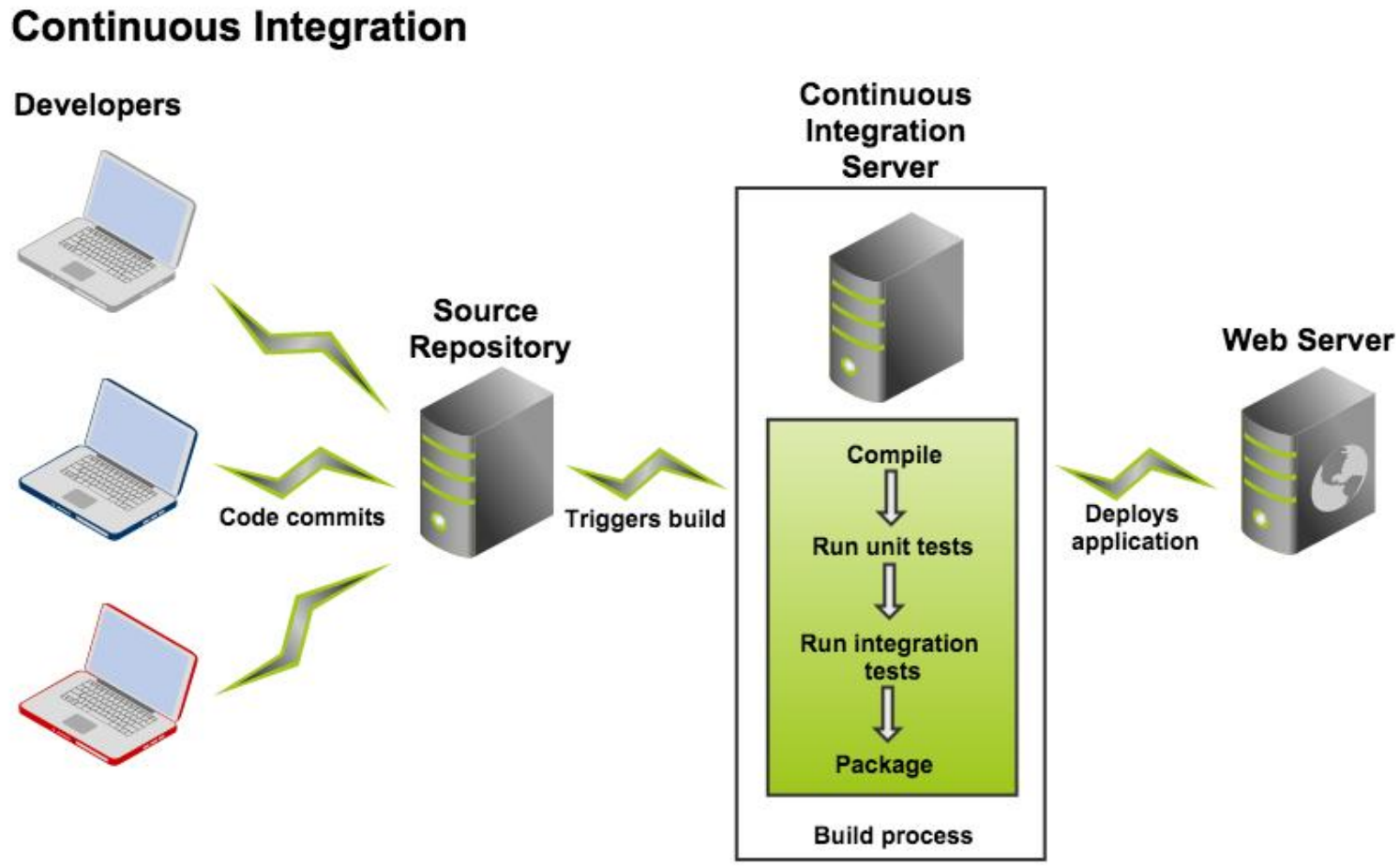
Jenkins

An extendable open source continuous integration server

Bamboo

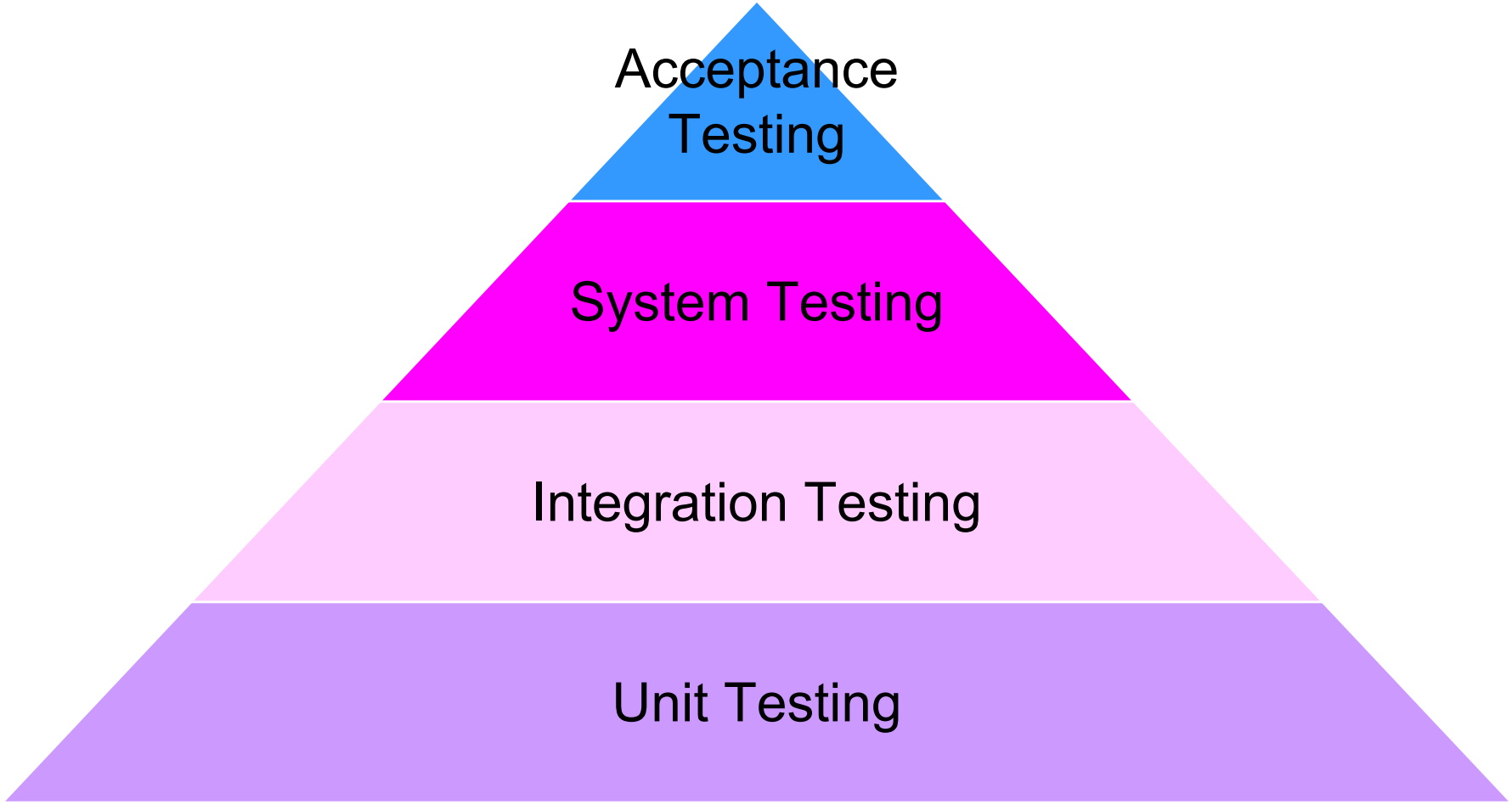
Travis CI

What happen in CI Server?



Pic from: https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html

Levels of Software Testing



Unit Testing

- Test individual units of a software
- A unit: smallest testable part of an application
  - E.g., method

```
public static double div(int x, int y){
        return x/y;
}
```

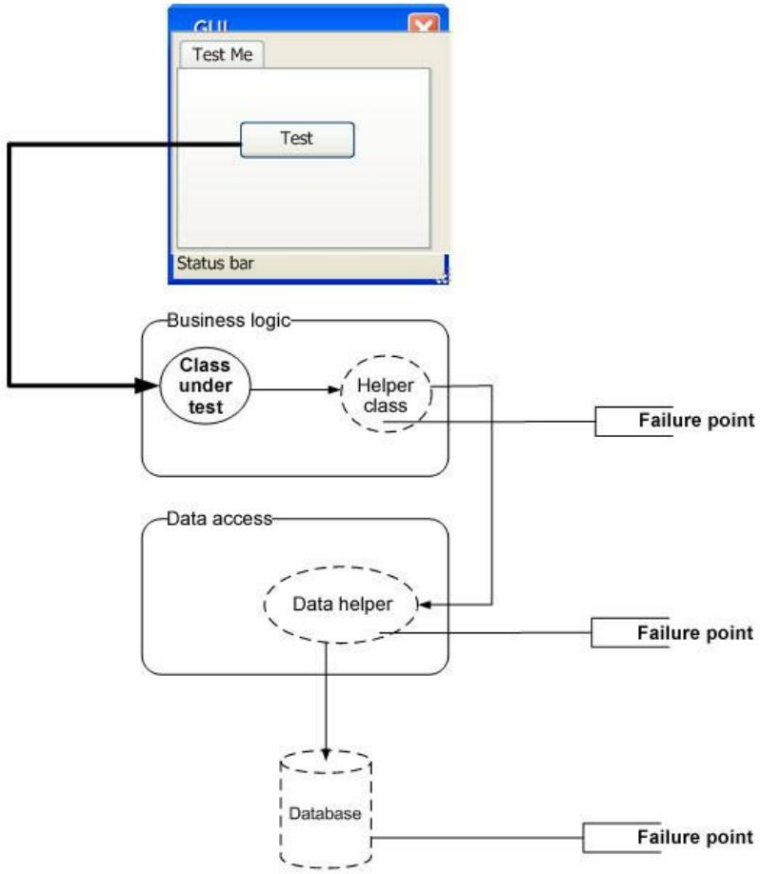
Input 1	Input 2	Output	Unit Test
1	2	0.5	`assertEquals(0.5, div(1, 2));`
1	1	1.0	`assertEquals(1.0, div(1, 1));`

1	0	ArithmeticException	`@Test(expected=java.lang.ArithmeticException.class)`
			`public void testDivideByZero() {`
			`  div(1,0)`
			`}`

```
assertEquals(expected, div(1, 2));
```

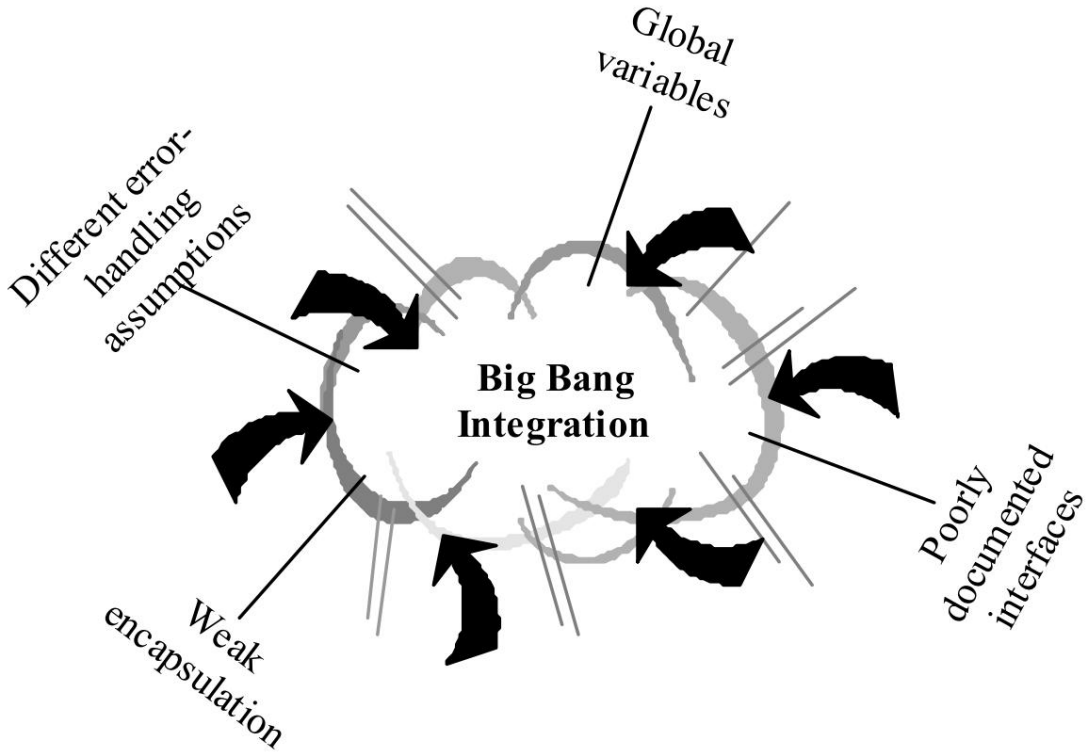
Integration testing

- **Integration testing**: Verify software quality by testing two or more dependent software modules as a group.
- Challenges:
  - Combined units can fail in more places and in more complicated ways.
  - How to test a partial system where not all parts exist?
  - How to properly simulate the behavior of unit A so as to produce a given behavior from unit B?



Big-bang Integration Testing

- *All* component are integrated together at *once*



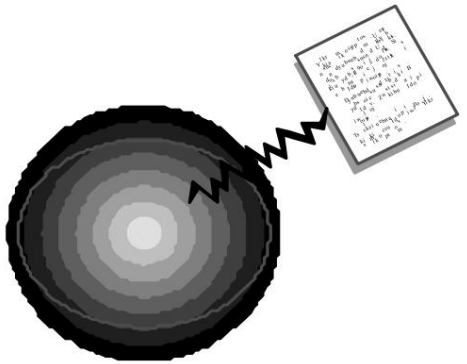
Big-bang Integration Testing

- Advantages:
  - Convenient for small systems.
- Disadvantages:
  - Finding bugs is difficult.
  - Due to large number of interfaces that need to be tested, some interfaces could be missed easily.
    - Testing team need to wait until everything is integrated so will have less time for testing.
    - High risk critical modules are not isolated and tested on priority.

Incremental Integration Testing

- **Incremental integration**:
  - Develop a functional "skeleton" system
  - Design, code, test, debug a small new piece
  - Integrate this piece with the skeleton
    - test/debug it before adding any other pieces

Big-bang Integration



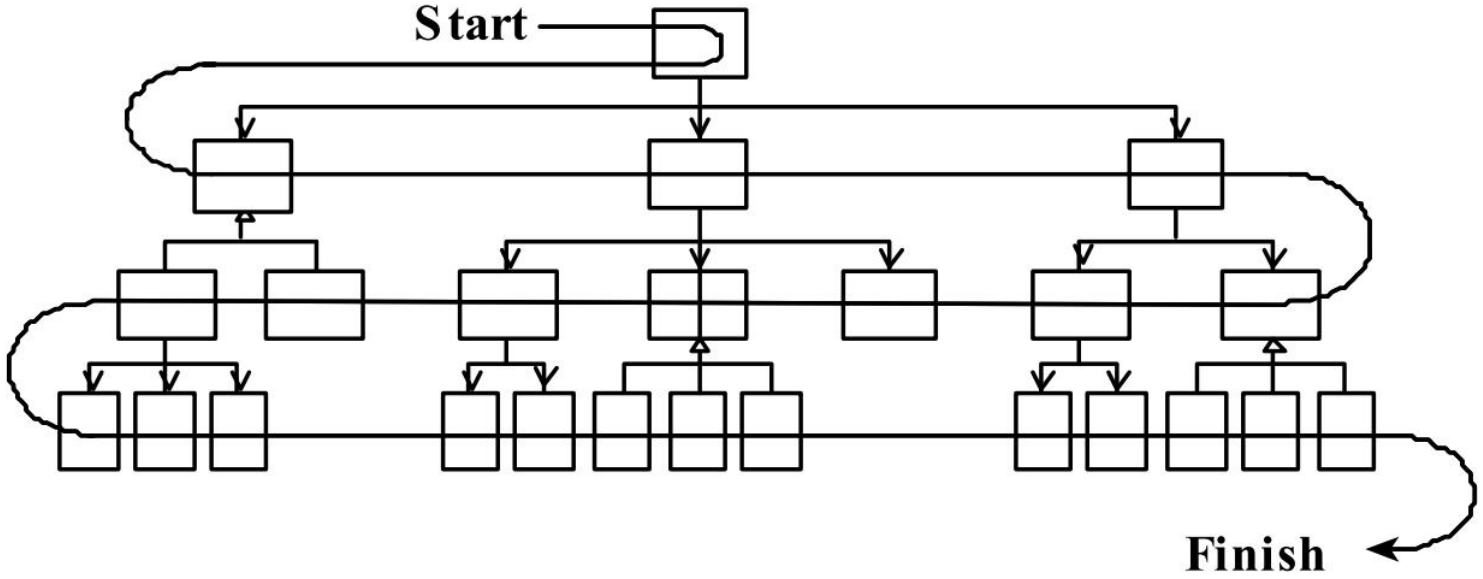
Incremental Integration

Incremental Integration Testing

- Advantages:
  - Errors easier to isolate, find, fix
    - Reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - Good for customer relations, developer morale
- Disadvantages:
  - May need to create "stub" versions of some features that have not yet been integrated

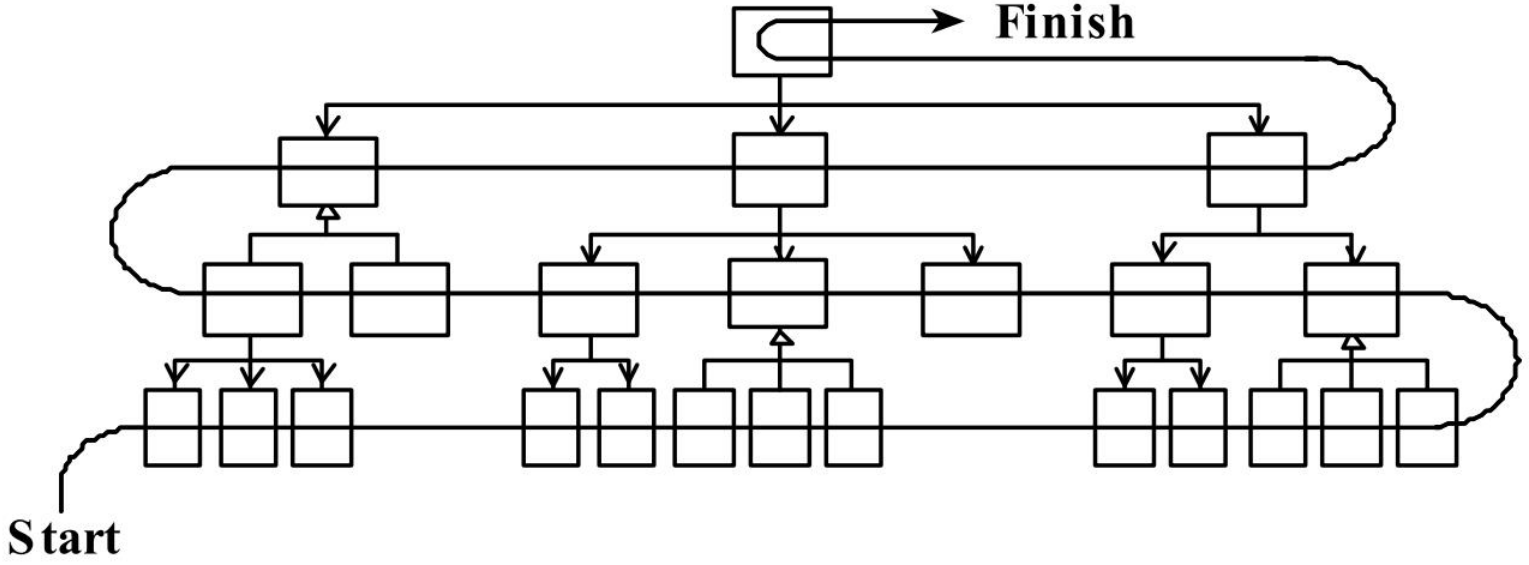
Top-down integration

- Start with outer UI layers and work inward
  - Must write (lots of) stub lower layers for UI to interact with
  - Allows postponing tough design/debugging decisions (bad?)



Bottom-up integration

- Start with low-level data/logic layers and work outward
  - Must write test drivers to run these layers
  - Won't discover high-level / UI design flaws until late



"Sandwich" integration

- Connect top-level UI with crucial bottom-level classes
  - Add middle layers later as needed
  - More practical than top-down or bottom-up?