cs304
Software Engineering

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

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

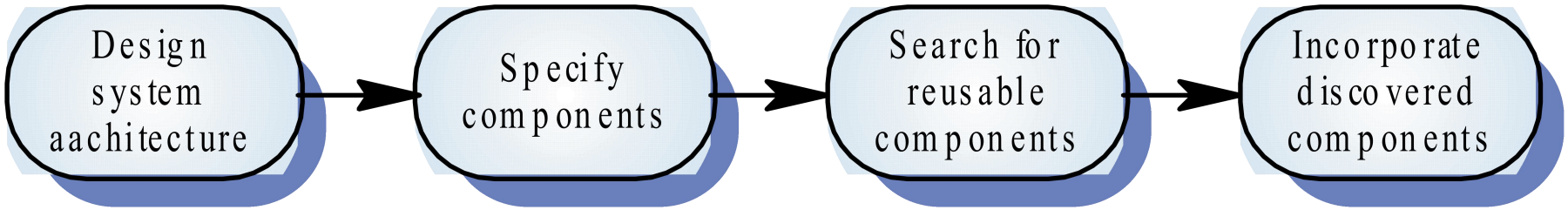
Administrative Info

- **Project Progress Report Uploaded:**
  - **due on 26 April 2021, 12.01am**
  - You need to know how to run and read the results for the **3 static analysis tools thought in the lab last week**
    - The leader needs to go to GitHub discussion to put the selected time slot
    - 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

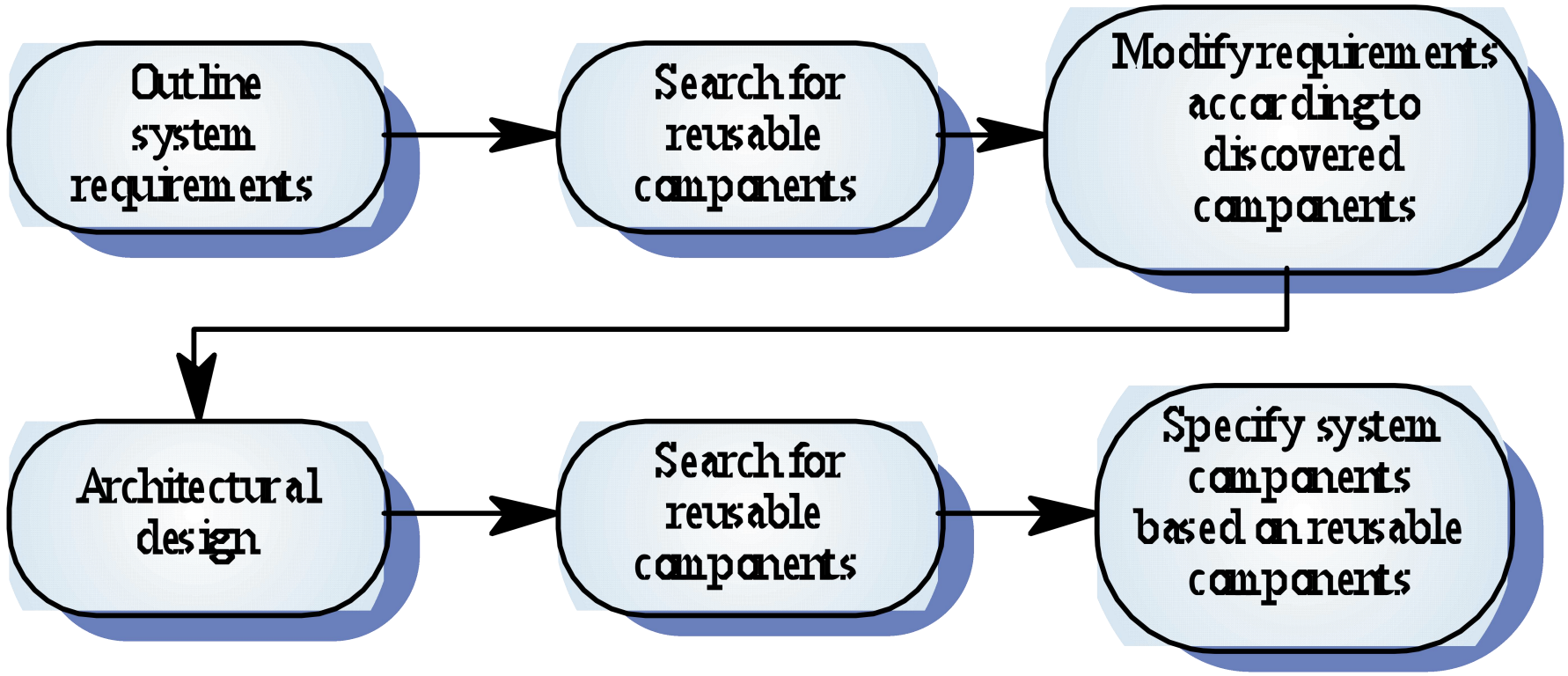
Software Reuse and Component-Based Software Engineering

Adapted from
http://groups.umd.umich.edu/cis/course.des/cis376/ppt/lec22.ppt

Opportunistic Reuse



Development Reuse as a Goal



Benefits of Reuse

- Increased Reliability
  - components already exercised in working systems
- Reduced Process Risk
  - less uncertainty in development costs
- Effective Use of Specialists
  - reuse components instead of people
- Standards Compliance
  - embed standards in reusable components
- Accelerated Development
  - avoid custom development and speed up delivery

Requirements for Design with Reuse

- You need to be able to find appropriate reusable components
- You must be confident that that component you plan to reuse is reliable and will behave as expected
- The components to be reused must be documented to allow them to be understood and modified (if necessary)

Reuse Problems

- Increased maintenance costs
- Lack of tool support
- Pervasiveness of the “not invented here” syndrome
- Need to create and maintain a component library
- Finding and adapting reusable components

Economics of Reuse - part 1

- Quality
  - with each reuse additional component defects are identified and removed which improves quality.
- Productivity
  - since less time is spent on creating plans, models, documents, code, and data the same level of functionality can be delivered with less effort so productivity improves.

Economics of Reuse - part 2

- Cost
  - savings projected by estimating the cost of building the system from scratch and subtracting the costs associated with reuse and the actual cost of the software as delivered.
- Cost analysis using structure points
  - can be computed based on historical data regarding the costs of maintaining, qualification, adaptation, and integrating each structure point.

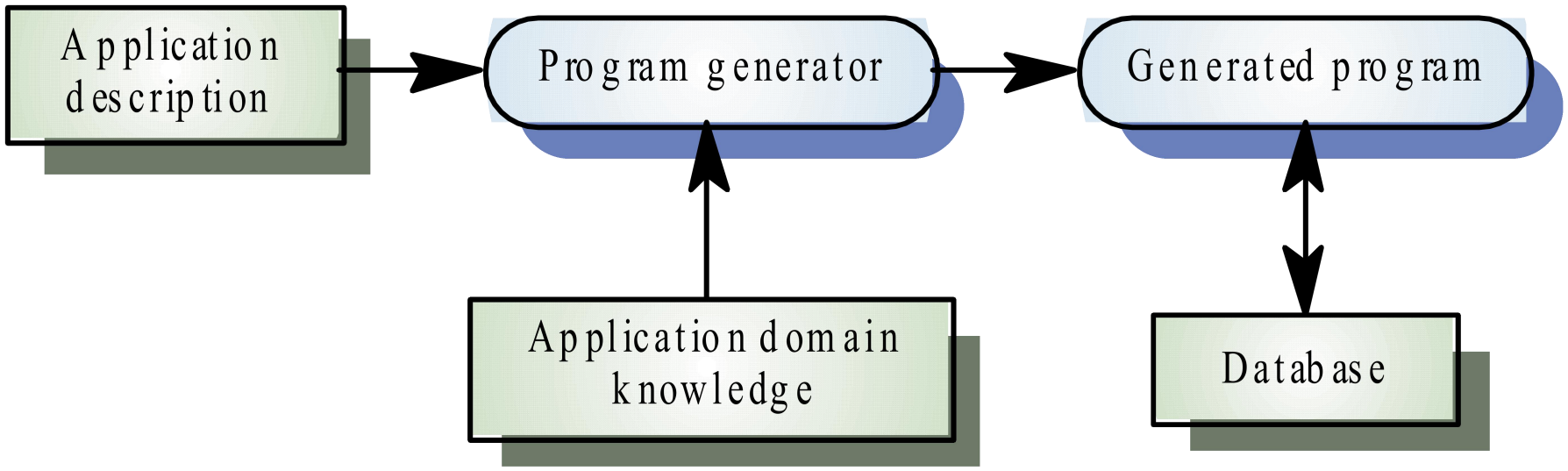
Generator-Based Reuse

- Program generators reuse standard patterns and algorithms
- Programs are automatically generated to conform to user defined parameters
- Possible when it is possible to identify the domain abstractions and their mappings to executable code
- Domain specific language is required to compose and control these abstractions

Types of Program Generators

- Applications generators for business data processing
- Parser and lexical analyzers generators for language processing
- Code generators in CASE tools
- User interface design tools

Program Generation

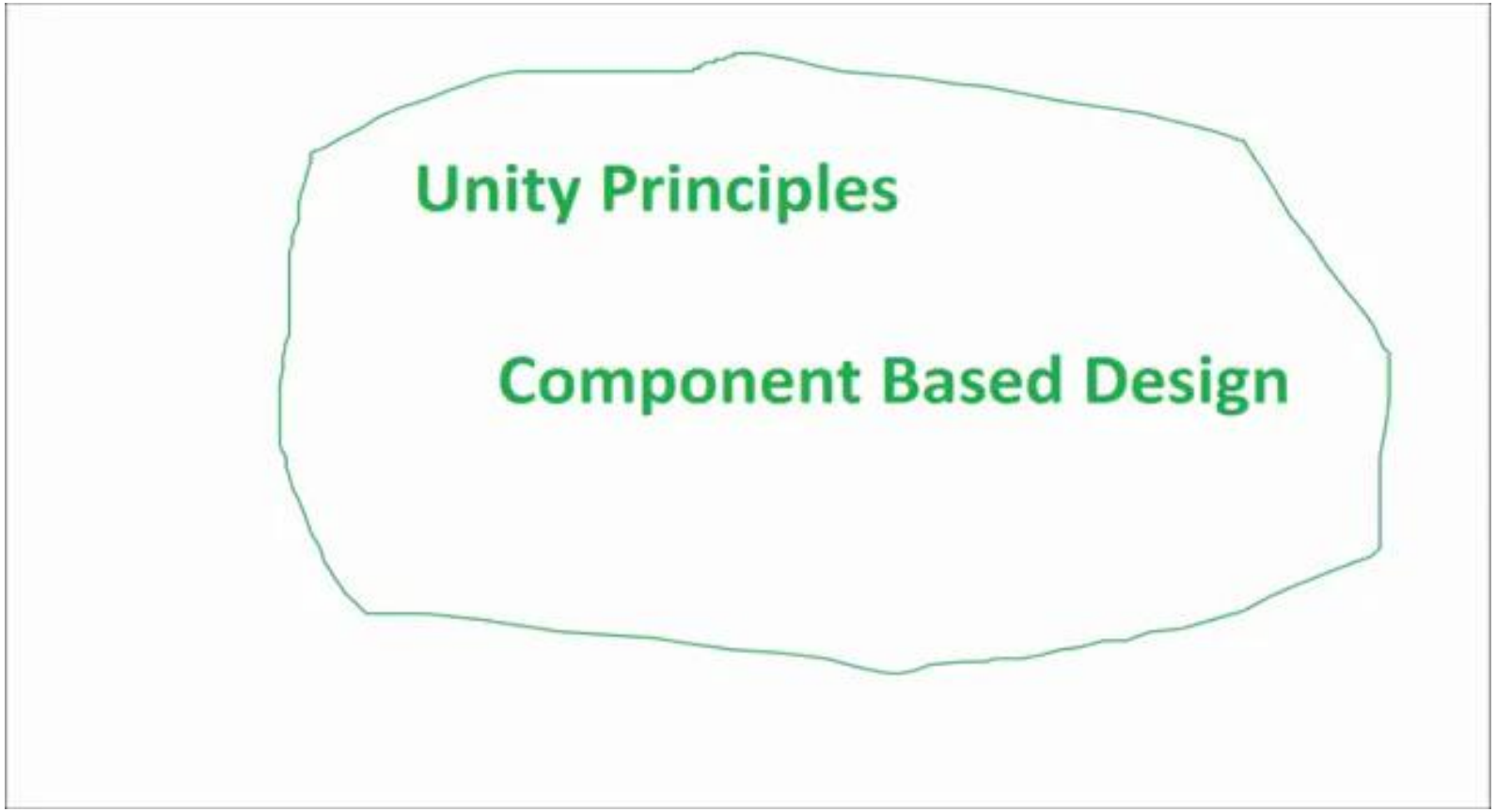


Assessing Program Generator Reuse

- Advantages
  - Generator reuse is cost effective
  - It is easier for end-users to develop programs using generators than other CBSE techniques
- Disadvantages
  - The applicability of generator reuse is limited to a small number of application domains

Component-Based Engineering

Component Based vs Object-oriented Programming



From: https://www.youtube.com/watch?v=1YGVP6wsxj0

Component-Based Software Engineering

- CBSE is an approach to software development that relies on reuse
- CBSE emerged from the failure of object-oriented development to support reuse effectively
- Objects (classes) are too specific and too detailed to support design for reuse work
- Components are more abstract than classes and can be considered to be stand-alone service providers

Component Abstractions

- Functional Abstractions
  - component implements a single function (e.g. *ln*)
- Casual Groupings
  - component is part of a loosely related entities like declarations and functions
- Data Abstractions
  - abstract data types or objects
- Cluster Abstractions
  - component from group of cooperating objects
- System Abstraction
  - component is a self-contained system

Engineering of Component-Based Systems - part 1

- Software team elicits system requirements
- Architectural design is established
- Team determines requirements are amenable to composition rather than construction
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within in the proposed system architecture?
- Team attempts to remove or modify requirements that cannot be implemented with COTS or in-house components

Engineering of Component-Based Systems - part 2

- For those requirements that can be addressed with available components the following activities take place:
  - component qualification
  - component adaptation
  - component composition
  - component update
- Detailed design activities commence for remainder of the system

Definition of Terms

- Component Qualification
  - candidate components are identified based on services provided and means by which consumers access them
- Component Adaptation
  - candidate components are modified to meet the needs of the architecture or discarded
- Component Composition
  - architecture dictates the composition of the end product from the nature of the connections and coordination mechanisms
- Component Update
  - updating systems that include COTS is made more complicated by the fact that a COTS developer must be involved

Commercial Off-the-Shelf Software (COTS)

- COTS systems are usually complete applications library the off an applications programming interface (API)
- Building large systems by integrating COTS components is a viable development strategy for some types of systems (e.g. E-commerce or video games)

COTS Integration Problems

- Lack of developer control over functionality and performance
- Problems with component interoperability as COTS vendors make different user assumptions
- COTS vendors may not offer users any control over the evolutions of its components
- Vendors may not offer support over the lifetime of a product built with COTS components

Developing Components for Reuse

- Components may constructed with the explicit goal to allow them to be generalized and reused
- Component reusability should strive to
  - reflect stable domain abstractions
  - hide state representations
  - be independent (low coupling)
  - propagate exceptions via the component interface

Reusable Components

- Tradeoff between reusability and usability
  - generic components can be highly reusable
  - reusable components are likely to be more complex and harder to use
- Development costs are higher for reusable components than application specific components
- Generic components may be less space-efficient and have longer execution times than their application specific analogs

Domain Engineering - part 1

- Domain analysis
  - define application domain to be investigated
  - categorize items extracted from domain
  - collect representative applications from the domain
  - analyze each application from sample
  - develop an analysis model for objects
- Domain model
- Software architecture development

Domain Engineering - part 2

- Structural model
  - consists of small number of structural elements manifesting clear patterns of interaction
  - architectural style that can be reused across applications in the domain
  - structure points are distinct constructs within the structural model (e.g. interface, control mechanism, response mechanism)
- Reusable component development
- Repository of reusable components is created

Structure Point Characteristics

- Abstractions with limited number of instances within an application and recurs in applications in the domain
- Rules governing the use of a structure point should be easily understood and structure point interface should be simple
- Structure points should implement information hiding by isolating all complexity contained within the structure point itself

Component-Based Development

- Analysis
- Architectural design
  - component qualification
  - component adaptation
  - component decomposition
- Component engineering
- Testing
- Iterative component update

Component Adaptation Techniques

- White-box Wrapping
  - integration conflicts removed by making code-level modifications to the code
- Grey-box Wrapping
  - used when component library provides a component extension language or API that allows conflicts to be removed or masked
- Black-box Wrapping
  - requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts

Engineering of Component-Based Systems - part 1

- Software team elicits system requirements
- Architectural design is established
- Team determines requirements are amenable to composition rather than construction
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within in the proposed system architecture?
- Team attempts to remove or modify requirements that cannot be implemented with COTS or in-house components

Engineering of Component-Based Systems - part 2

- For those requirements that can be addressed with available components the following activities take place:
  - component qualification
  - component adaptation
  - component composition
  - component update
- Detailed design activities commence for remainder of the system

Definition of Terms

- Component Qualification
  - candidate components are identified based on services provided and means by which consumers access them
- Component Adaptation
  - candidate components are modified to meet the needs of the architecture or discarded
- Component Composition
  - architecture dictates the composition of the end product from the nature of the connections and coordination mechanisms
- Component Update
  - updating systems that include COTS is made more complicated by the fact that a COTS developer must be involved

Commercial Off-the-Shelf Software (COTS)

- COTS systems are usually complete applications library the off an applications programming interface (API)
- Building large systems by integrating COTS components is a viable development strategy for some types of systems (e.g. E-commerce or video games)

COTS Integration Problems

- Lack of developer control over functionality and performance
- Problems with component interoperability as COTS vendors make different user assumptions
- COTS vendors may not offer users any control over the evolutions of its components
- Vendors may not offer support over the lifetime of a product built with COTS components

Developing Components for Reuse

- Components may constructed with the explicit goal to allow them to be generalized and reused
- Component reusability should strive to
  - reflect stable domain abstractions
  - hide state representations
  - be independent (low coupling)
  - propagate exceptions via the component interface

Reusable Components

- Tradeoff between reusability and usability
  - generic components can be highly reusable
  - reusable components are likely to be more complex and harder to use
- Development costs are higher for reusable components than application specific components
- Generic components may be less space-efficient and have longer execution times than their application specific analogs

Domain Engineering - part 1

- Domain analysis
  - define application domain to be investigated
  - categorize items extracted from domain
  - collect representative applications from the domain
  - analyze each application from sample
  - develop an analysis model for objects
- Domain model
- Software architecture development

Domain Engineering - part 2

- Structural model
  - consists of small number of structural elements manifesting clear patterns of interaction
  - architectural style that can be reused across applications in the domain
  - structure points are distinct constructs within the structural model (e.g. interface, control mechanism, response mechanism)
- Reusable component development
- Repository of reusable components is created

Structure Point Characteristics

- Abstractions with limited number of instances within an application and recurs in applications in the domain
- Rules governing the use of a structure point should be easily understood and structure point interface should be simple
- Structure points should implement information hiding by isolating all complexity contained within the structure point itself

Component-Based Development

- Analysis
- Architectural design
  - component qualification
  - component adaptation
  - component decomposition
- Component engineering
- Testing
- Iterative component update

Component Adaptation Techniques

- White-box Wrapping
  - integration conflicts removed by making code-level modifications to the code
- Grey-box Wrapping
  - used when component library provides a component extension language or API that allows conflicts to be removed or masked
- Black-box Wrapping
  - requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts

Abstraction

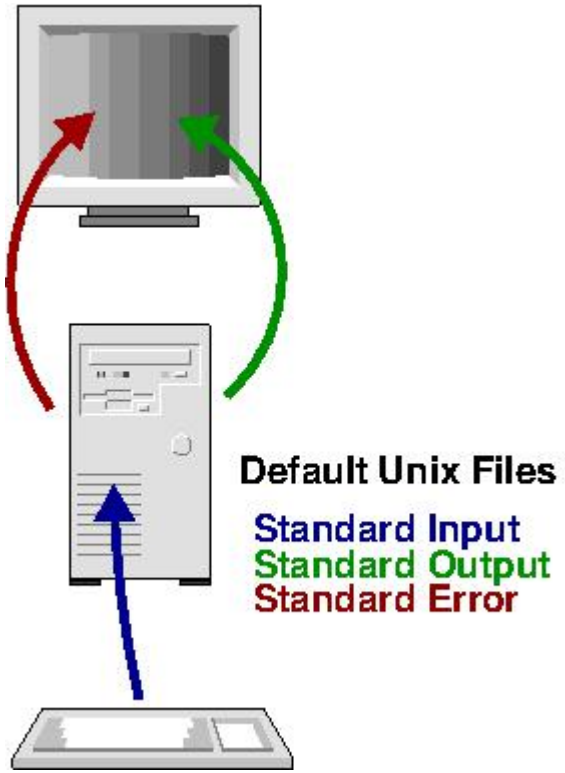
- Abstract
  1: disassociated from any specific instance
  2: difficult to understand

- Abstraction: ignoring unimportant details and focusing on key features

Examples are in “Is abstraction the key to computing” by Jeff Kramer
https://www.ics.uci.edu/~andre/informatics223s2007/kramer.pdf

Example of abstraction

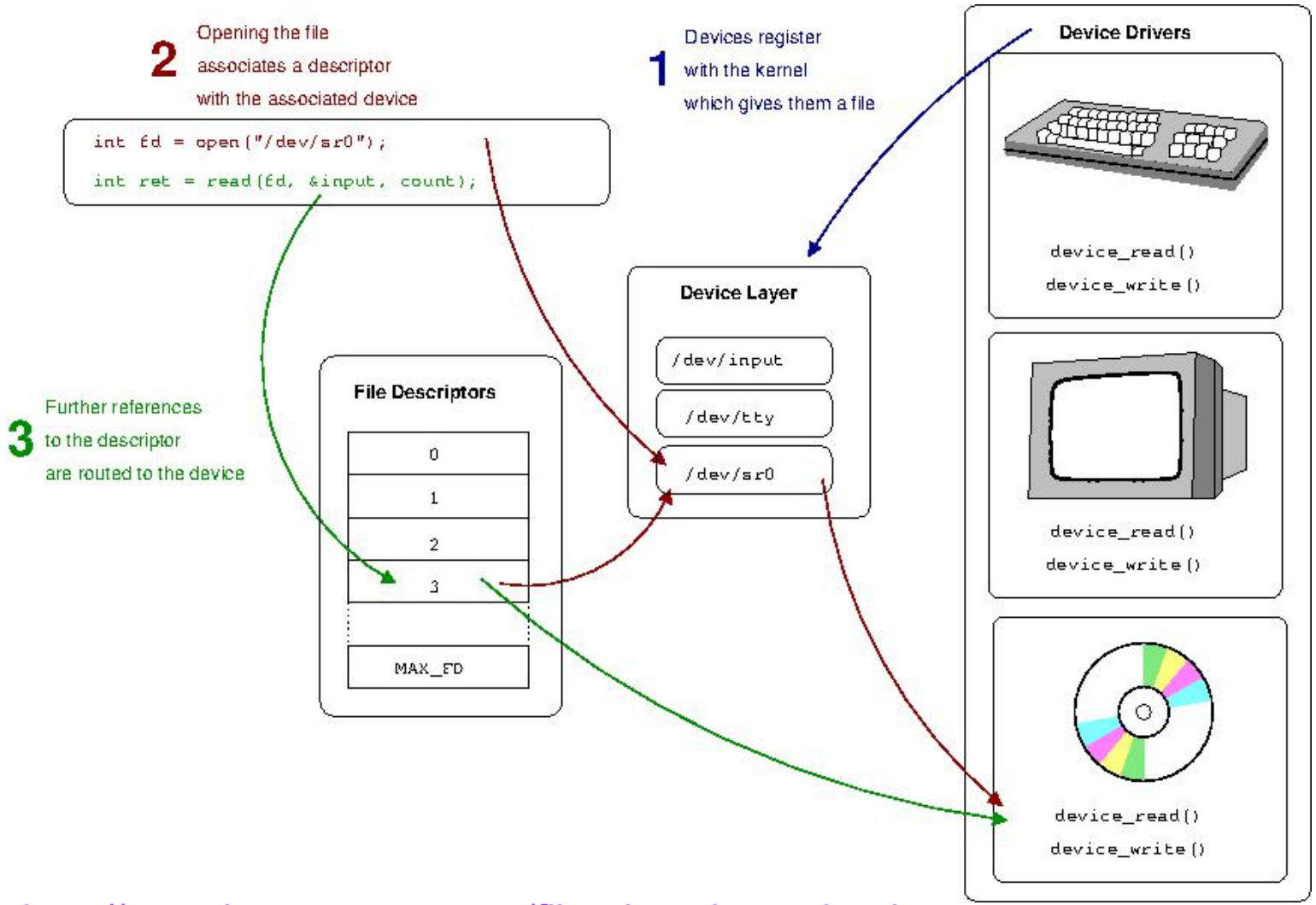
- Unix file descriptor
- Can read, write, and (maybe) seek
- File, IO device, pipe



Descriptive Name	File Number	Description
Standard In	0	Input from the keyboard
Standard Out	1	Output to the console
Standard Error	2	Error output to the console

http://www.bottomupcs.com/file_descriptors.html

Example of abstraction



http://www.bottomupcs.com/file_descriptors.html

Kinds of abstraction in S.E.

- Procedural abstraction
  - Naming a sequence of instructions
  - Parameterizing a procedure
- Data abstraction
  - Naming a collection of data
  - Data type defined by a set of procedures
- Control abstraction
  - W/o specifying *all* register/binary-level steps
- Performance abstraction O(N)

Abstract data types

- Complex numbers: +, -, *, real, imaginary

- Queue: add, remove, size

Facts about abstraction

- Abstractions are powerful
- People think about concrete details better than abstractions
- People learn abstractions from examples
- It takes many examples to invent a good abstraction

How to get good abstractions

- Get them from someone else
  - Read lots of books
  - Look at lots of code
- Generalize from examples
  - Try them out
  - Gradually improve them
- Look for duplication in your program and eliminate it

Abstractions can fool you

- Suppose collection has operation getItemNumbered(int index)
- How do you iterate?

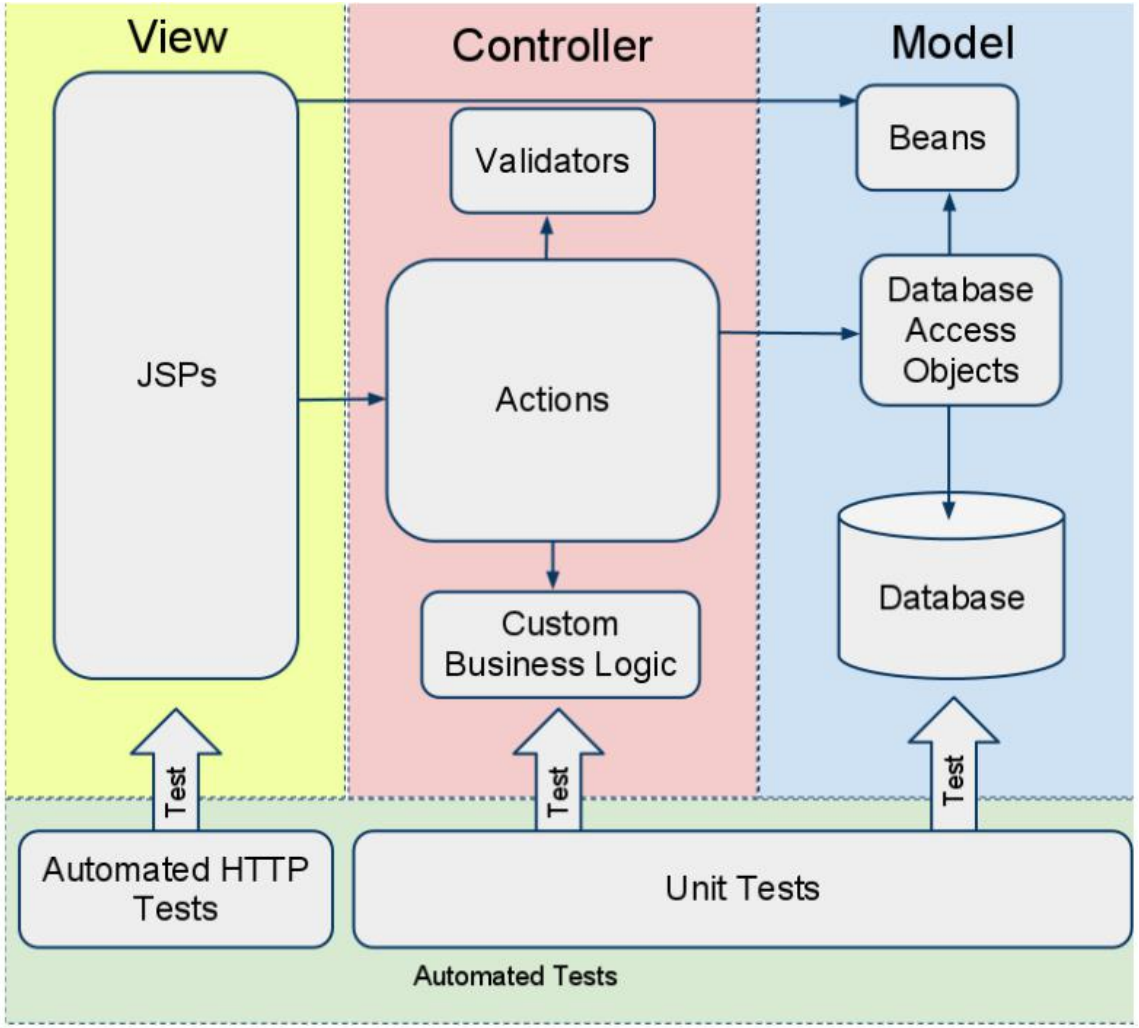
  for i := 1 to length { getItemNumbered(i) }

- But what if collection is a linked list?

High-level view: Architecture

- Big picture
- Structure(s) that support the system
- Early design decisions
  - Expensive to change
  - Key to meeting non-functional requirements
- Divide the system into modules
  - Divide the developers into (sub)teams
  - Subcontract work to other groups
  - Find packages to reuse

iTrust Architecture

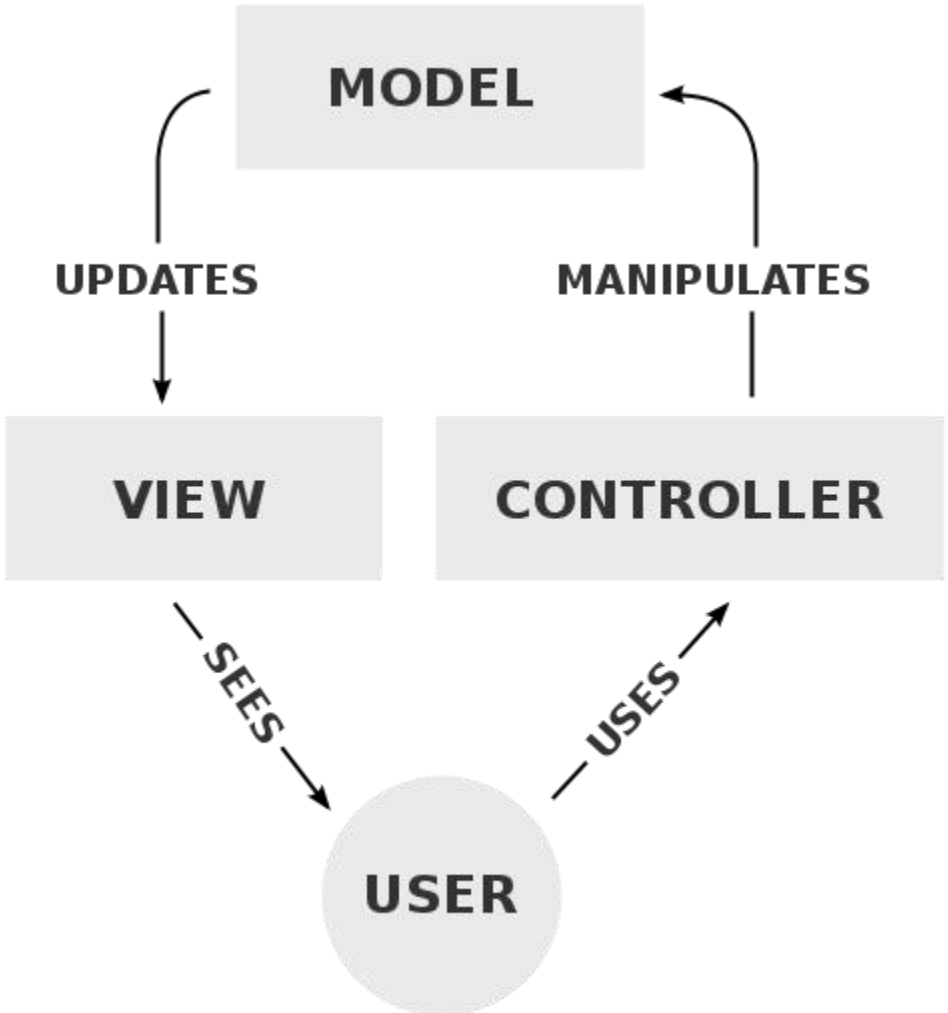


http://bensmith.s3.amazonaws.com/website/papers/sst2011.pdf

Picking an architecture

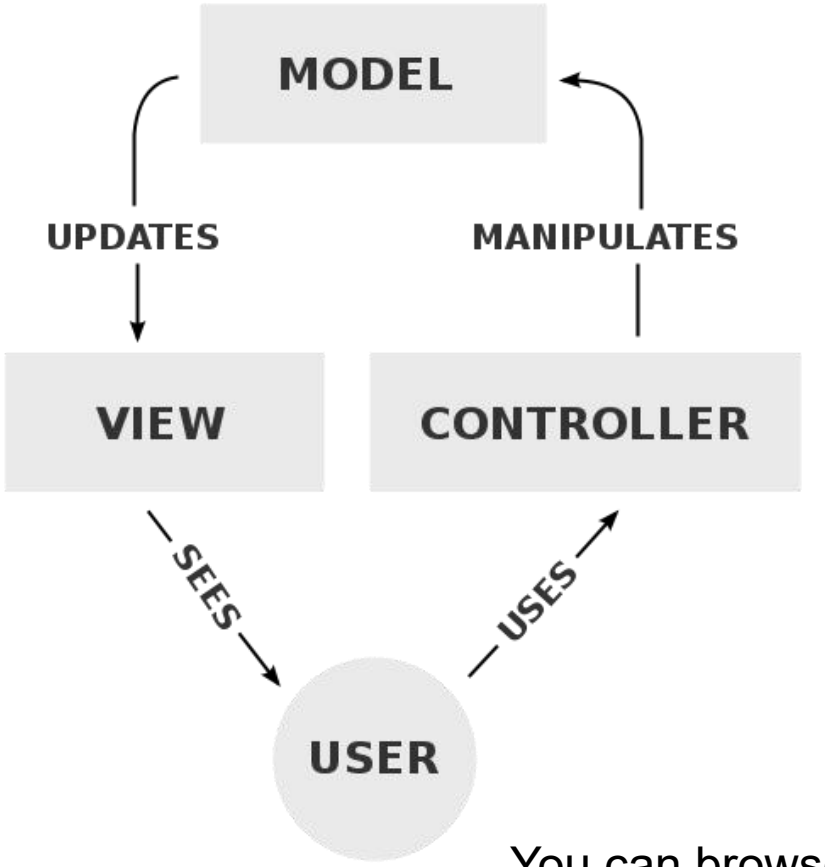
- Many standard architectures
  - Each has strengths and weaknesses
  - Each solves some problems & creates others
- How do you pick an architectural style?
- What is important?
  - Flexibility/ease of change, efficiency, reliability
- More in reading on Wiki (note architectural patterns, including MVC)

MVC Pattern



http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

MVC-View: iTrust View



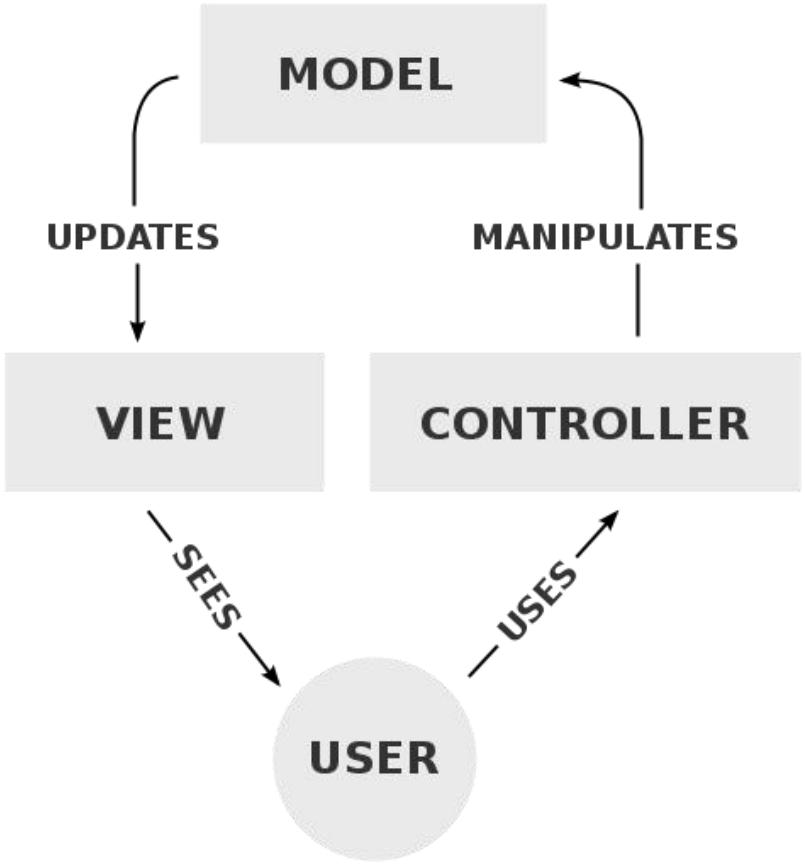
- Little logic; just display info
- JSPs

- JSP one-to-one mapping to action class (from controller)
- A JSP instantiates the mapped Action class

You can browse the iTrust code base at
http://agile.csc.ncsu.edu/iTrust/wiki/doku.php#source_codev210

http://bensmith.s3.amazonaws.com/website/papers/sst2011.pdf

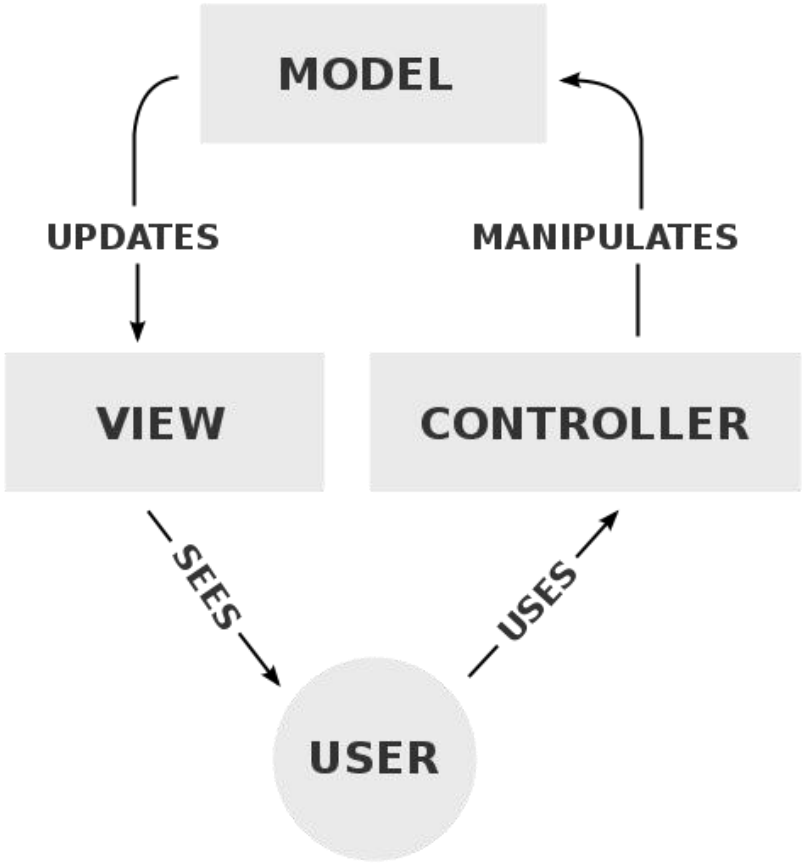
MVC-View: iTrust Model



- Logic related to persistent storage
- DB system (MySQL)
  - each DB entity maps to a single DAO and a single Bean
- Beans: placeholders for data related to an iTrust entity (e.g., Patient)
  - minimal functionality (only store data)
  - Other supporting classes
    - load beans from database result sets
    - validate beans based on input
    - any other custom logic needed.
- DAOs (DB Access Objects): Java objects that interact with the DB
  - reflect contents in the DB
  - offer to interact with the DB
  - query and update the DB (e.g., from Action classes)
  - should not have many branches (assuming incoming data is valid and any exception is handled by the Action classes)

http://bensmith.s3.amazonaws.com/website/papers/sst2011.pdf

MVC-View: iTrust Controller



- Handle all logic
  - validate data
  - process DB query results
- Everything between Action classes and DAO classes
  - Action classes
    - exception handling
    - a method <= 15 lines
  - Validators
  - Custom business logic
- Action classes delegate responsibilities to other classes
  - Delegate any input validation to a Validator
  - Log transactions for auditability
  - Delegate any custom business logic, such as risk factor calculations
  - Delegate database interaction
  - Handle exceptions in a secure manner

http://bensmith.s3.amazonaws.com/website/papers/sst2011.pdf

Modularity

- “Modularity is the single attribute of software that allows a program to be intellectually manageable.” (Myers)

- Split a larger program into smaller modules
  - A module can be procedure, class, file, directory, package, service…

Functional independence

- What are two terms related to modules?
- Cohesion - Each module should do one thing - high cohesion
- Coupling - Each module should have a simple interface - low coupling

Coupling

- Measure of interconnection among modules
- The degree to which one module depends on others

- Minimize coupling

Cohesion

- Measure of interconnection within a module
- The degree to which one part of a module depends on another

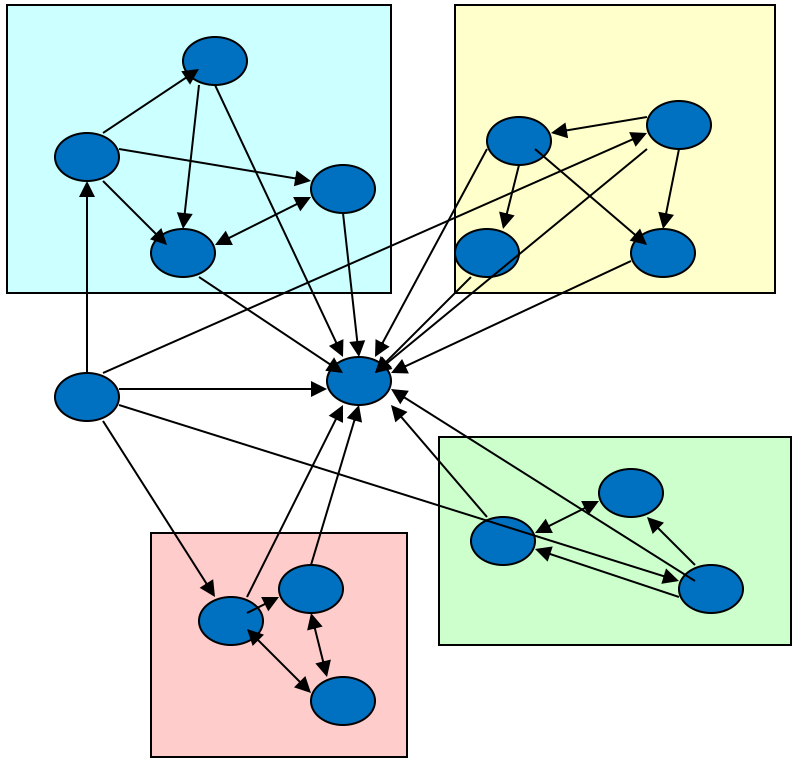
- Maximize cohesion

Cohesion

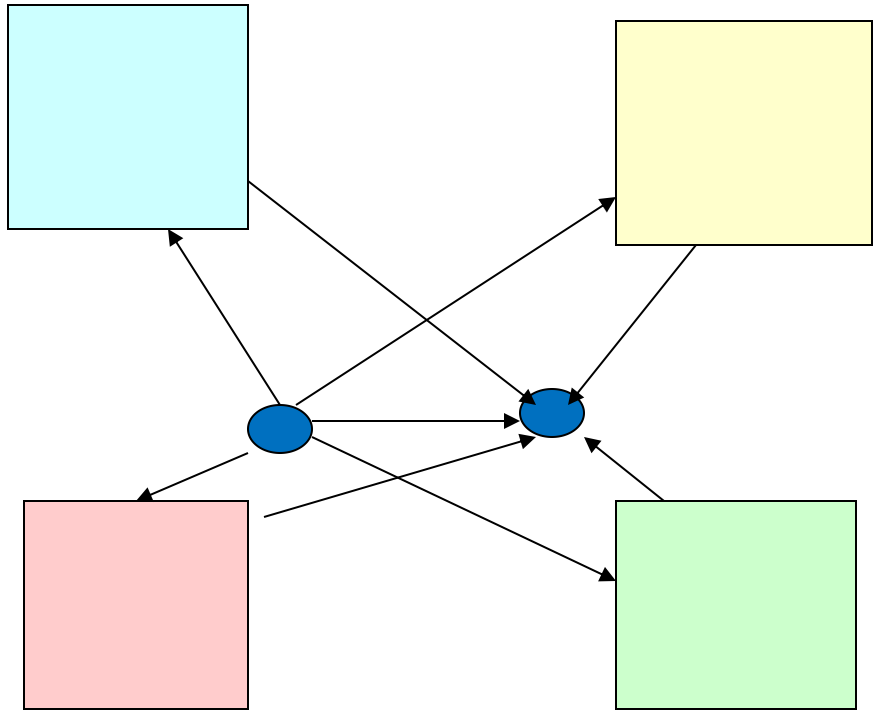
- Coincidental - grouped by chance
- Logical - same idea
- Temporal - same time
  - e.g., placing together all the code used during system start-up or initialization
- Procedural - one calls the other
- Communicational - shared data

- Change together

Turn spaghetti code…



…into a few modules



Information hiding

- Each module should hide a design decision from others
- Ideally, one design decision per module, but usually design decisions are closely related
- D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems Into Modules," *CACM, 5(12), December 1972*

Design decisions

- Representation of data
- Use of a particular software package
- Use of a particular printer
- Use of a particular operating system
- Use of a particular algorithm

Other reasons for modularity

- Let developers work on system in parallel
  - Conway's Law: The architecture of a system is the same as the structure of the group that developed it.
- Security - compartmentalization
- Reliability - localization of failure
- Parallelism – load balance processes
- Distributed programming - design modules to reduce communication

Trade-offs

- Suppose moving functionality from one module to another will
  - Decrease coupling between modules
  - Increase cohesion in one module
  - Decrease cohesion in another

- Should you do it?

XP Simplicity rules

- http://c2.com/xp/XpSimplicityRules.html

- Runs all the tests
- Minimizes coupling
- Maximizes cohesion
- Says everything “Once and only once”
  - E.g., avoid “clones”

How to get modularity

- Reuse a design with good modularity
- Think about design decisions – hide them
- Reduce coupling and increase cohesion
  - E.g., refactor (how many teams refactored?)
- Eliminate duplication
- Reduce impact of changes
  - If adding a feature requires changing large part of the system, refactor so change is easy

Modularity summary

- Modularity is key to managing complexity
- Modularity is about managing dependencies
- Modularity should be based on hiding design decisions
- No perfect answer
  - Trade-offs
  - Iteration

Enhancing Reliability

- Name generalization
  - names modified to use domain independent language
- Operation generalization
  - operations added to provide extra functionality
  - domain specific operations may be removed
- Exception generalization
  - application specific exceptions removed
  - exception management added to increase robustness
- Component certification
  - component warranted correct and reliable for reuse

Component Composition Infrastructure Elements - part 1

- Data exchange model
  - similar to drag and drop type mechanisms should be defined for all reusable components
  - allow human-to-software and component-to-component transfer
- Automation
  - tools, macros, scripts should be implemented to facilitate interaction between reusable components

Component Composition Infrastructure Elements - part 2

- Structured storage
  - heterogeneous data should be organized and contained in a single data structure rather several separate files
- Underlying object model
  - ensures that components developed in different languages are interoperable across computing platforms

Application Frameworks

- Frameworks are sub-system design containing a collection of abstract and concrete classes along with interfaces between each class
- A sub-system is implemented by adding components to fill in missing design elements and by instantiating the abstract classes
- Frameworks are reusable entities

Framework Classes

- System infrastructure frameworks
  - support development of systems infrastructure elements like user interfaces or compilers
- Middleware integration frameworks
  - standards and classes that support component communication and information exchange
- Enterprise application frameworks
  - support development of particular applications like telecommunications or financial systems

Extending Frameworks

- Generic frameworks need to be extended to create specific applications or sub-systems
- Frameworks can be extend by
  - defining concrete classes that inherit operations from abstract class ancestors
  - adding methods that will be called in response to events recognized by the framework
- Frameworks are extremely complex and it takes time to learn to use them (e.g. DirectX or MFC)

UI Design

User Interface Design

- Important
- Hard
- Isn’t covered well by most software development processes

From: https://www.youtube.com/watch?v=RFv53AxxQAo

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

Reading

- Joel Spolsky on Software
- Read nine “chapters” of the “book” on UI design for programmers at the link https://www.joelonsoftware.com/category/uibook/

UI Hall of Fame or Shame?

Desktop vs. Mobile

Slides from http://stellar.mit.edu/S/course/6/sp11/6.831/materials.html ©Rob Miller

Small Screen

Slides from http://stellar.mit.edu/S/course/6/sp11/6.831/materials.html ©Rob Miller

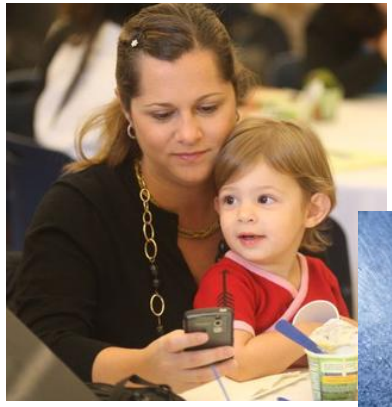
“Fat Finger”

Lorem Ipsum is simply dummy text of the pr
industry. Lorem Ipsum has been the industry's
ever since the 1500s, when an unknown printe
and scrambled it to make a type specimen boo

Text Input

Slides from http://stellar.mit.edu/S/course/6/sp11/6.831/materials.html ©Rob Miller

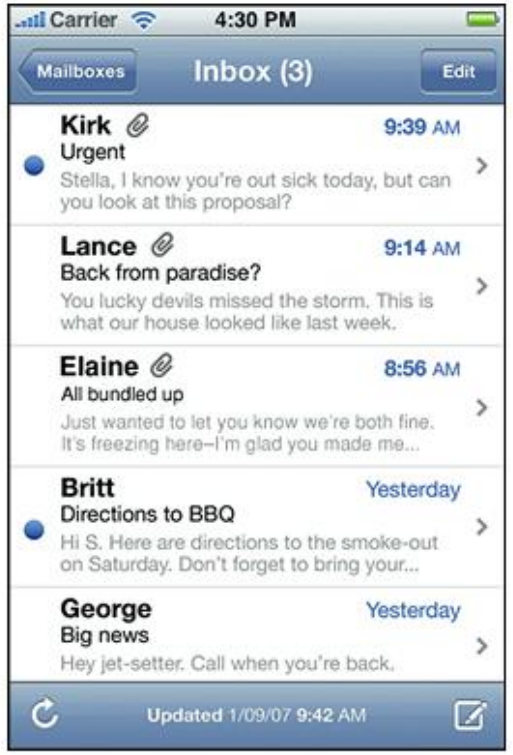
Context

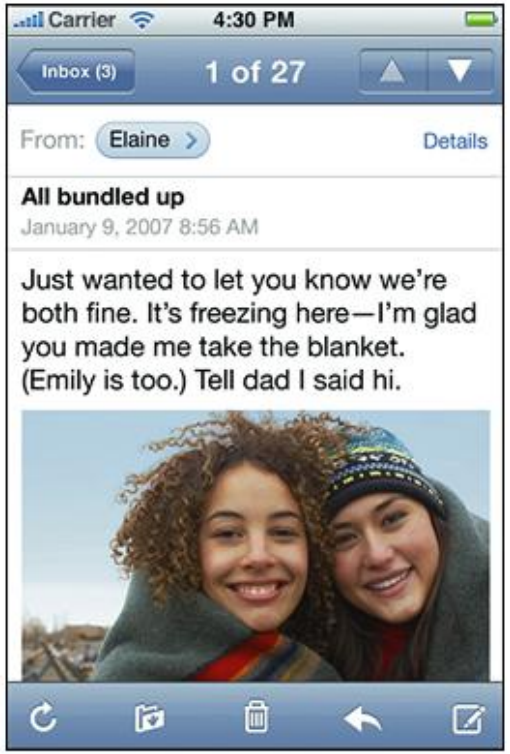


Other Issues in Mobile

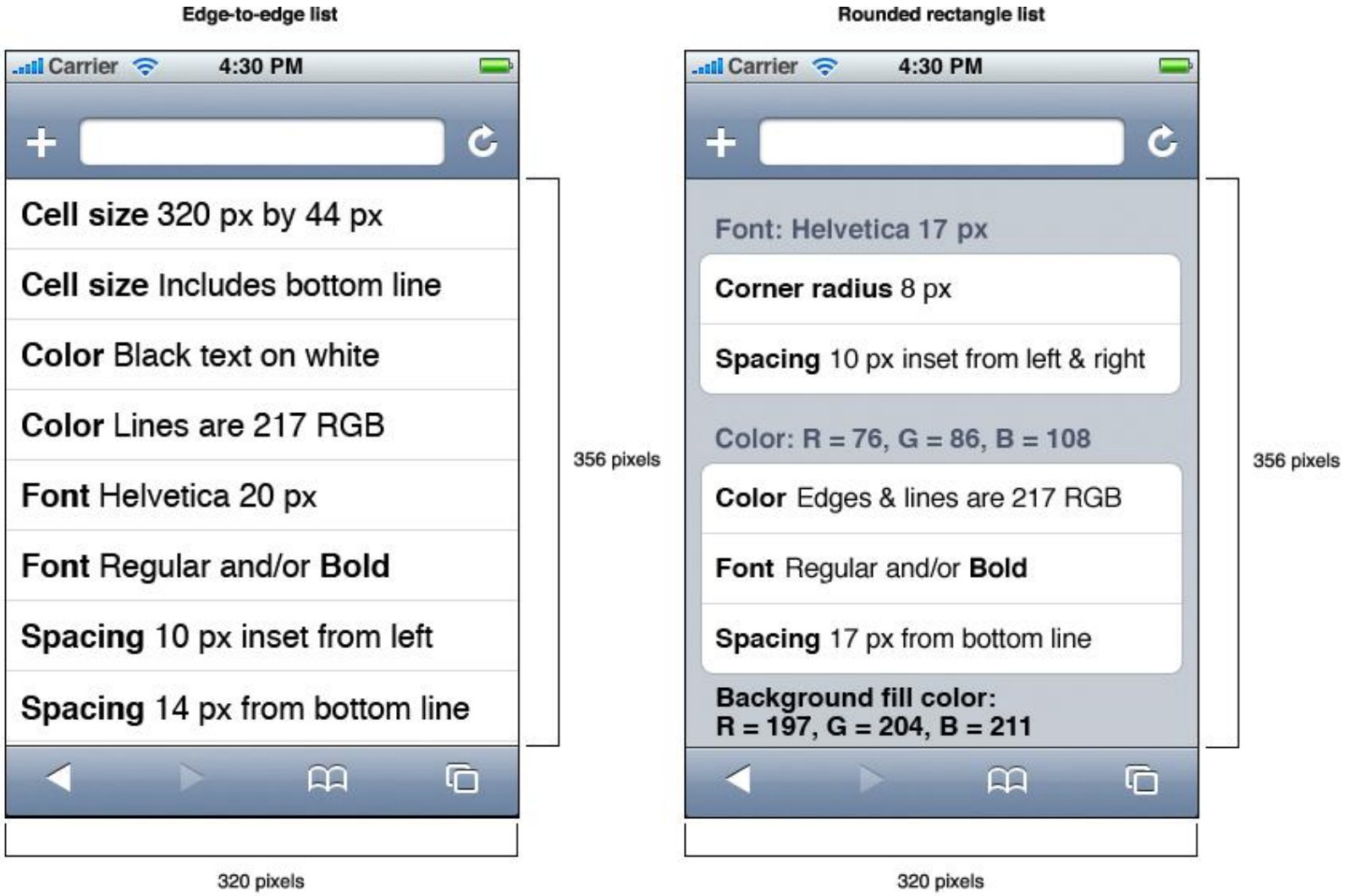
- Power & battery life
- Network latency, bandwidth, inconsistency
- CPU speed

Distinct Screens



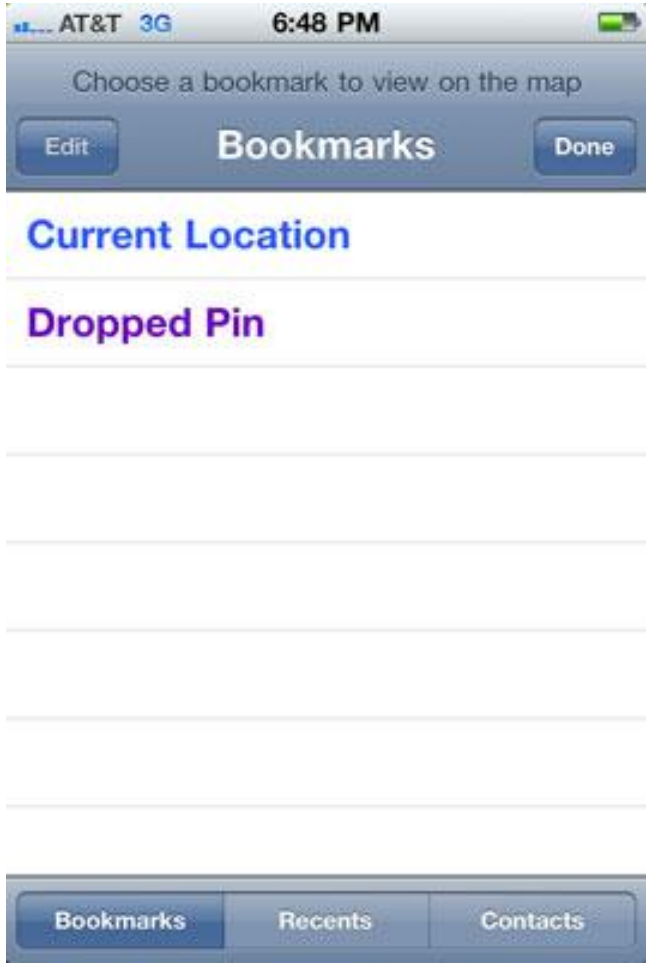


Scrolling Lists



Finger-Sized Targets

Minimize Text Input

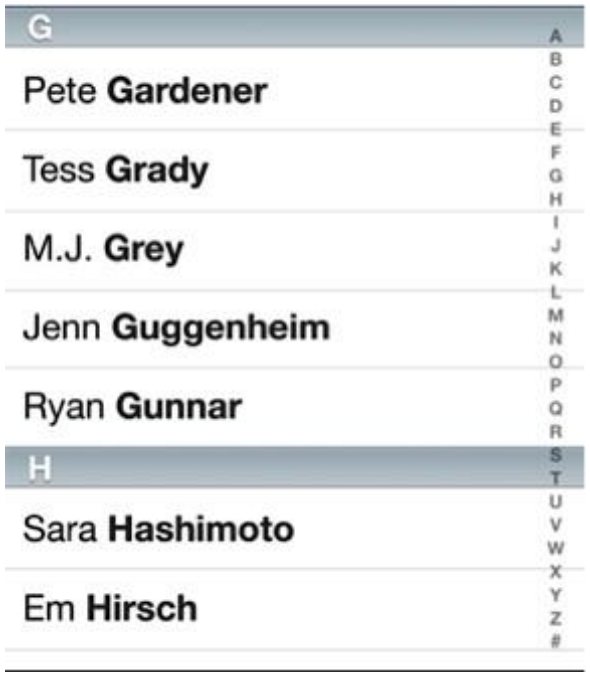


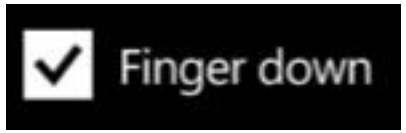
Slides from http://stellar.mit.edu/S/course/6/sp11/6.831/materials.html ©Rob Miller

Simplify, Simplify, Simplify!

Slides from http://stellar.mit.edu/S/course/6/sp11/6.831/materials.html ©Rob Miller

Mobile Widgets



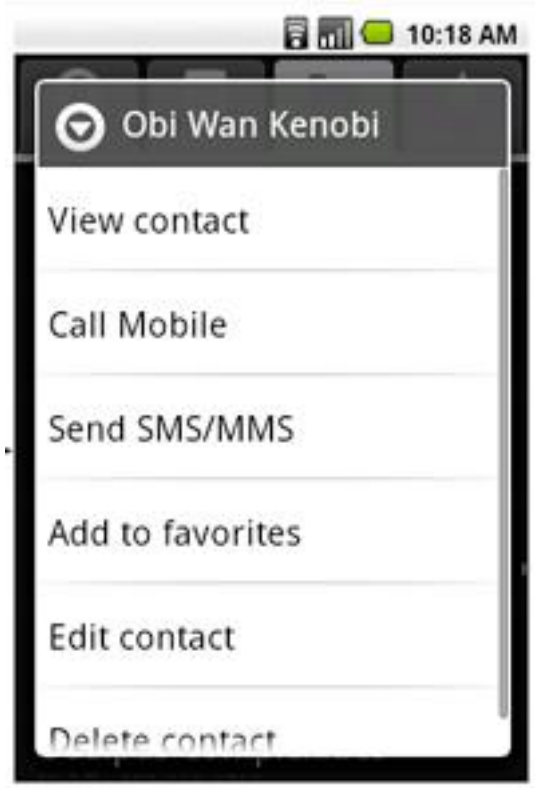


Many Kinds of Menus

Options icon menu

Touch

Options expanded menu



Context Menu

Touch Gestures

Slides from http://stellar.mit.edu/S/course/6/sp11/6.831/materials.html ©Rob Miller

Summary of Mobile UI Design

- Mobile UI design faces new challenges
  - Small screens
  - Fat fingers
  - Poor text entry
- Simplify
  - Follow design patterns
  - Use touch gestures where possible

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

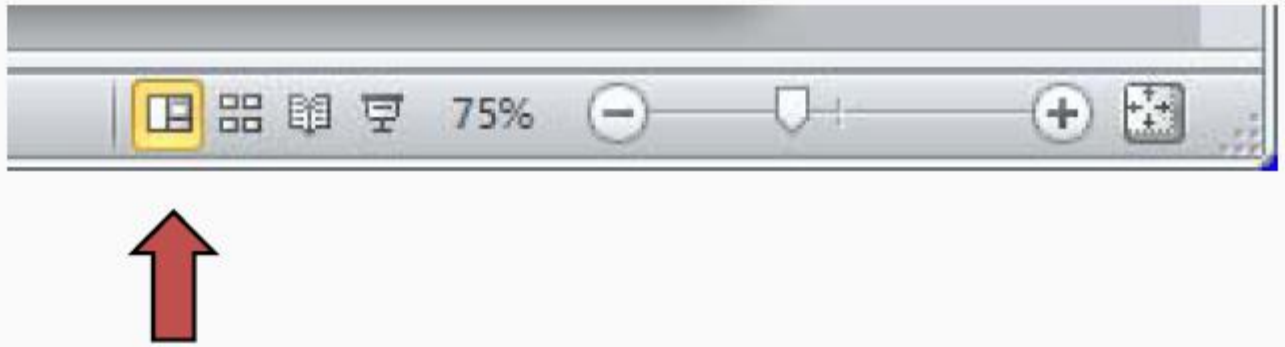
Let the User be in Control (1)

- Undo
- Macros
- Direct manipulation

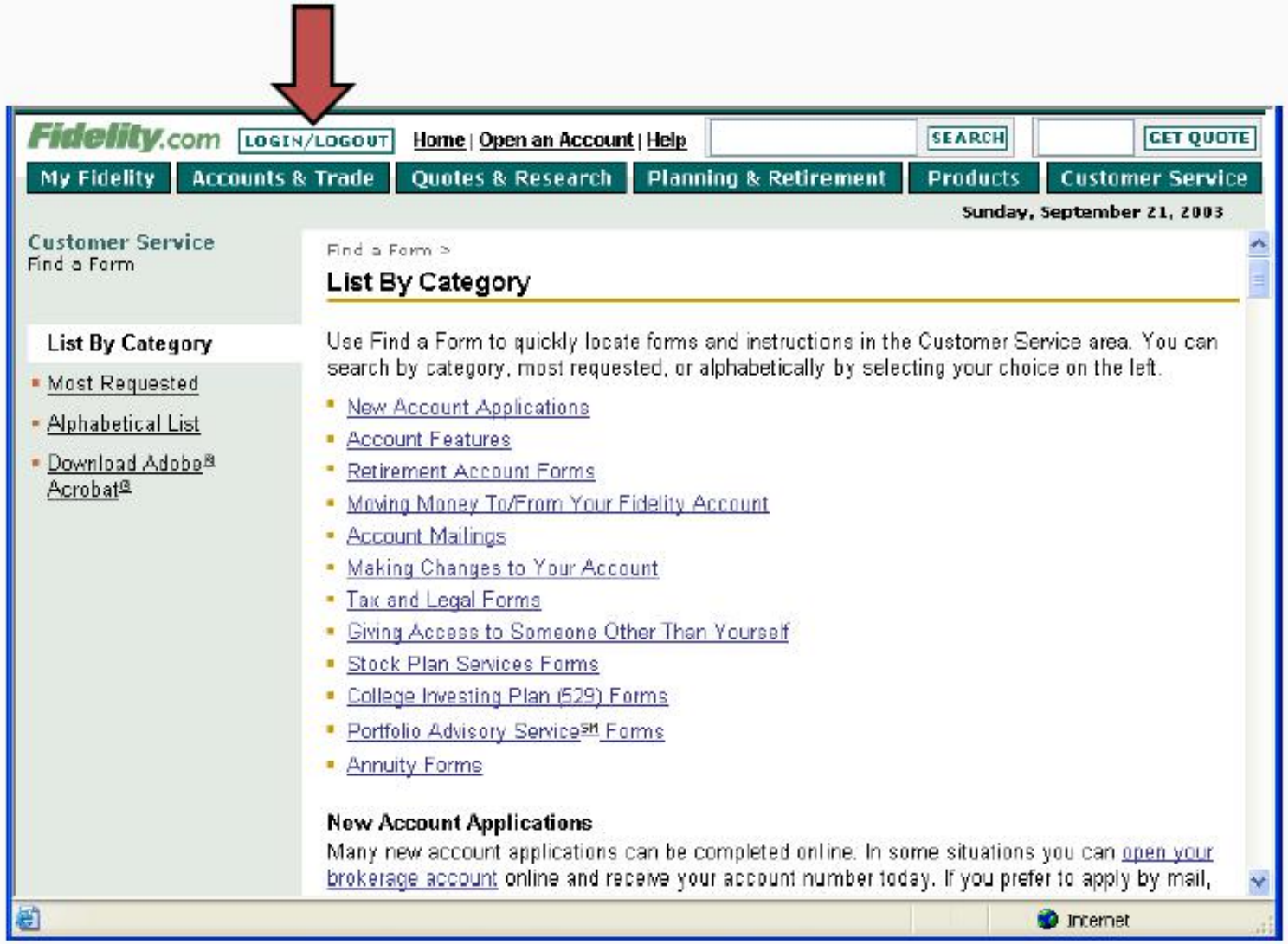
More info: http://www.cs.wm.edu/~kemper/cs435/slides/l18.pdf

Let the User be in Control (2)

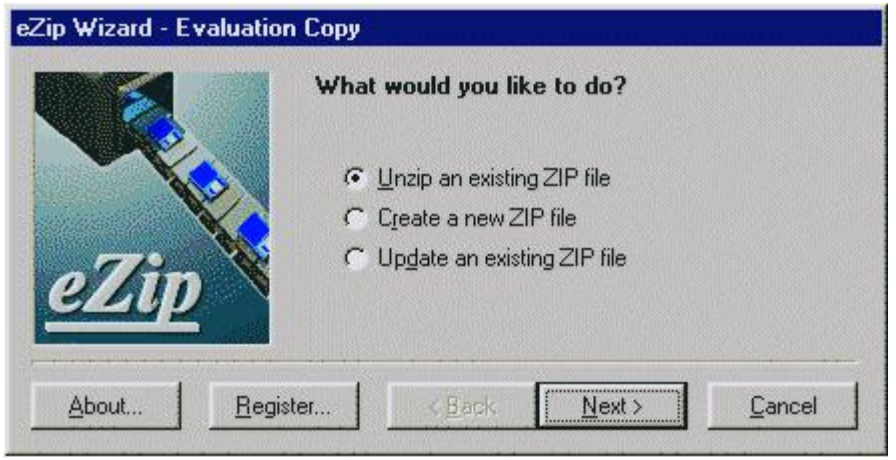
- Modes
  - Use a new window instead of a new mode
  - Make modes visible (signed in/logged out)



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)