cs304
Software Engineering

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

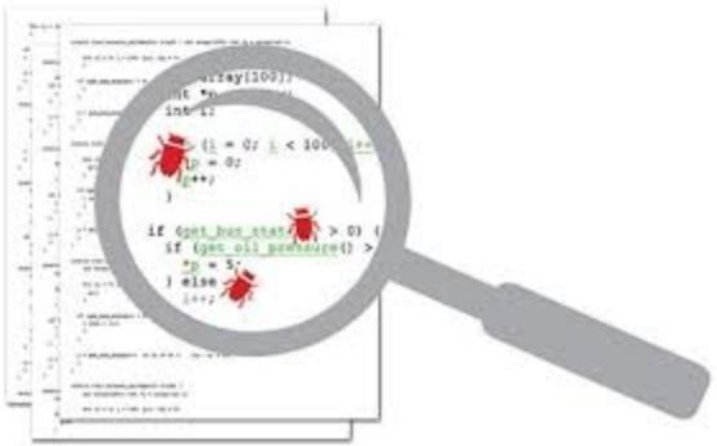
Slides adapted from cs427 (UIUC) and cs409 (SUSTech)

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
  - Make sure you have installed and use the 3 static analysis tools
- **Ask question on GitHub discussion instead of Wechat:**
  - Wechat group is for posting announcement

Recap: What is testing?

Dynamic Analysis
Static Analysis



What is Static Analysis?

Static analysis involves no dynamic execution of the software under test and can detect possible defects in an early stage, before running the program.

Static analysis is done after coding and before executing unit tests.

Static analysis can be done by a machine to automatically “walk through” the source code and detect noncomplying rules. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.

What are the static analysis tools that you learned during the lab in previous week?

Question: Which tools check for coding standard?

- PIT
- CheckStyle
- FindBugs
- PMD

Documentation

Technical Writers:
The person in charge of documentation

https://www.youtube.com/watch?v=qnnkAWP55Ww

What are *Javadoc* Comments?

```
/*
* Returns a synchronized map backed by the given map.
…
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

Overall Description

Block Tags

- @param - Parameter name, Description
- @return - Description
- @throws - Exception name, Condition under which the exception is thrown

javadoc format

- Use this format for all doc comments:

```
/**
  * This is where the text starts. The asterisk lines
  * up with the first asterisk above; there is a space
  * after each asterisk. The first sentence is the most
  * important: it becomes the "summary."
  *
  * @param x Describe the first parameter (don't say its type).
  * @param y Describe the first parameter (don't say its type).
  * @return Tell what value is being returned (don't say its type).
  */
public String myMethod(int x, int y) {  // p lines up with the / in /**
```

Javadoc placement

- javadoc comments begin with `/**` and end with `*/`
  - In a javadoc comment, a * at the beginning of the line is not part of the comment text
- javadoc comments must be *immediately before:*
  - a class (plain, inner, abstract, or enum)
  - an interface
  - a constructor
  - a method
  - a field (instance or static)
- Anywhere else, javadoc comments will be *ignored!*
  - Plus, they look silly

HTML in doc comments

- javadoc comments (but not other kinds of comments) are written in HTML
- In a doc comment, you *must* replace:
  < with < > with > & with &
  ...because these characters are special in HTML
- Other things you may use:
  - ... to make something italic
    - Example: This case should never occur!
  - ... to make something boldface
  - <p> to start a new paragraph
  - <code>...<code> to wrap the names of variables, methods, etc.
  - <pre>...</pre> to use a monospaced font and preserve whitespace
  -   to put in a nonbreaking space

More HTML

- You can create a bulleted list with:

```
<ul>
    <li>first list element</li>
    <li>second list element</li>
</ul>
```

- You can create a numbered list with:

```
<ol>
    <li>first list element</li>
    <li>second list element</li>
</ol>
```

- You can insert a hyperlink with:

```
<a href="url">visible text</a>
```

- There's more--most non-document oriented HTML can be used

Who the javadoc is for

- Javadoc comments *should* be written for **programmers who want to use your code**
  - Example: The way you use Sun's Java API
- Javadoc comments **should *not*** be written for:
  - Programmers who need to debug, maintain, or upgrade the code
    - Internal (`//` or `/*...*/`) comments should be used for this purpose
  - People who just want to use the program
- **Therefore:**
  - Javadoc comments **should** describe exactly *how to use* the class, method, constructor, etc.
  - Javadoc comments **should *not*** describe the internal workings of the class or method (unless it affects the user in some way)
  - In addition, javadoc method comments **should not** tell who uses the method (inappropriate, but also difficult to keep up to date)

Tags in doc comments

- Use the standard ordering for javadoc tags
  - In class and interface descriptions, use:

    @author *your name*
    @version *a version number or date*

    - ***Use the @author tag in your assignments for all top-level classes!!!***
    - These tags are only used for classes and interfaces
  - In method descriptions, use:

    @param ***p*** *A description of parameter p.*
    @return *A description of the value returned (not used if the method returns void).*
    @exception *e* *Describe any thrown exception.*

  Do *not* mention the type in @param and @return tags--javadoc will do this (and get it right)

Which comment is better?

A. Finds the first blank in the string.
B. Find the first blank in the string.
C. This method finds the first blank in the string.
D. Method findBlank(String s) finds the first blank in the string.

```
 int findBlank(String s){


…
}
```

Rules for writing summaries

- The *first sentence* should summarize the purpose of the element
- For methods, omit the subject and write in the third-person narrative form
  - Good: Finds the first blank in the string.
  - Not as good: Find the first blank in the string.
  - Bad: This method finds the first blank in the string.
  - Worse: Method findBlank(String s) finds the first blank in the string.
- Use the word this rather than "the" when referring to instances of the current class (for example, Multiplies this fraction…)
- Do not add parentheses to a method or constructor name unless you want to specify a particular signature
- Keep comments up to date!

Bugs and missing features

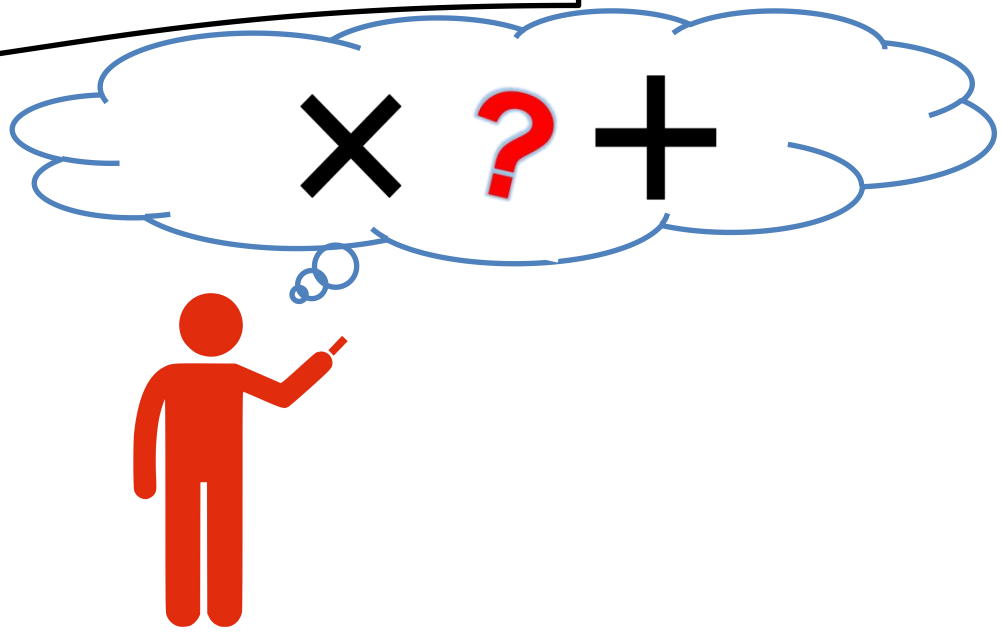
- Document known problems!
  - There are three “standard” flags that you can put into any comment
    - TODO -- describes a feature that should be added
    - FIXME -- describes a bug in the method
    - XXX -- this needs to be thought about some more
  - Eclipse and NetBeans both recognize these flags and can open a window that lists all occurrences of them
  - You can create additional tags of your own in your IDE

Problems of Javadoc

Research

Outdated Code Comments

```
/* This method adds two numbers*/
int multiply(int x, int y){
```



Javadoc Comments can be Inconsistent with Code

```
/*…
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

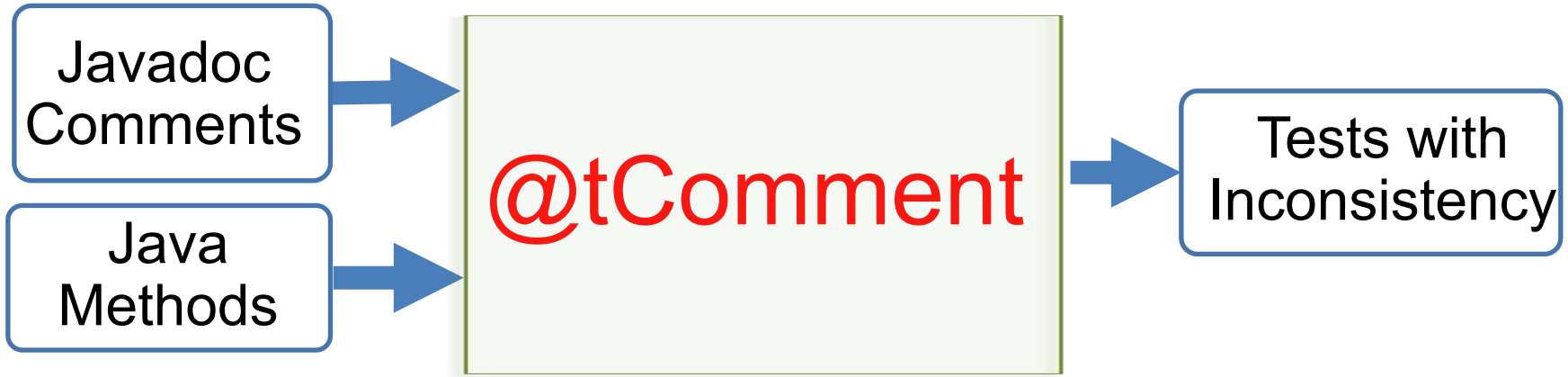
Expected behavior for `synchronizedMap(null)`:

- Throws `IllegalArgumentException`

Actual behavior:

- Throws `NullPointerException`

An approach for testing comment-code inconsistency



Published in ICST 2012

Example Inconsistency of Type 1: Correct Code, Incorrect Comment

```
/*…
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

Example Inconsistency of Type 1: Correct Code, Incorrect Comment

```
/*…
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

✓ Confirmed & fixed by Collections developers


```
public void test1() throws Throwable
    {  java.util.Map var0 = null;
    try {
            java.util.Map var1= ...synchronizedMap(var0);
     } catch (IllegalArgumentException expected) {return;}
      fail("Expected exception of type IllegalArgumentException
                    but got NullPointerException“);
}
```

Example Inconsistency of Type 2: Fault in Code, Correct Comment

```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```

Javadoc Comments can be Inconsistent with Code

```
/*…
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

Expected behavior for synchronizedMap (null):

- Throws *IllegalArgumentException*

Actual behavior:

- Throws *NullPointerException*

Example Inconsistency of Type 1: Correct Code, Incorrect Comment

```
/*…
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

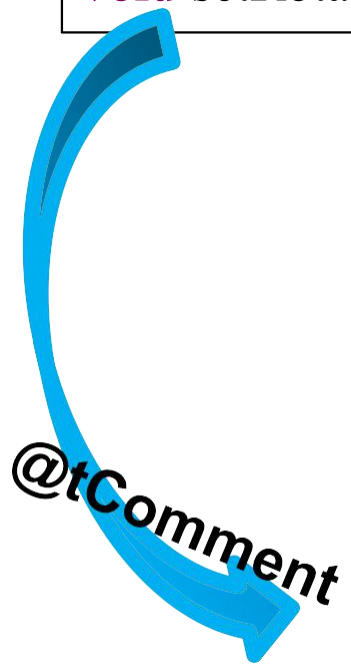
Example Inconsistency of Type 1: Correct Code, Incorrect Comment

```
/*…
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```


```
public void test1() throws Throwable
    {  java.util.Map var0 = null;
    try {
            java.util.Map var1= ...synchronizedMap(var0);
     } catch (IllegalArgumentException expected) {return;}
      fail("Expected exception of type IllegalArgumentException
                        but got NullPointerException“);
}
```

Example Inconsistency of Type 2: Fault in Code, Correct Comment

```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```



Example Inconsistency of Type 2: Fault in Code, Correct Comment

```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```

@tComment

```
public void test2() throws Throwable {
    ...CategoryPointerAnnotation var0 =
                    new …CategoryPointerAnnotation(  “$0.00”,
                    (java.lang.Comparable)'#', 10.0d, 10.0d);
    org.jfree.ui.TextAnchor var1 = null;
    try {
            var0.setRotationAnchor(var1);
            fail("Expected exception but got Normal Execution");
        } catch (Exception expected) {}
}
```

Example Inconsistency of Type 2: Fault in Code, Correct Comment

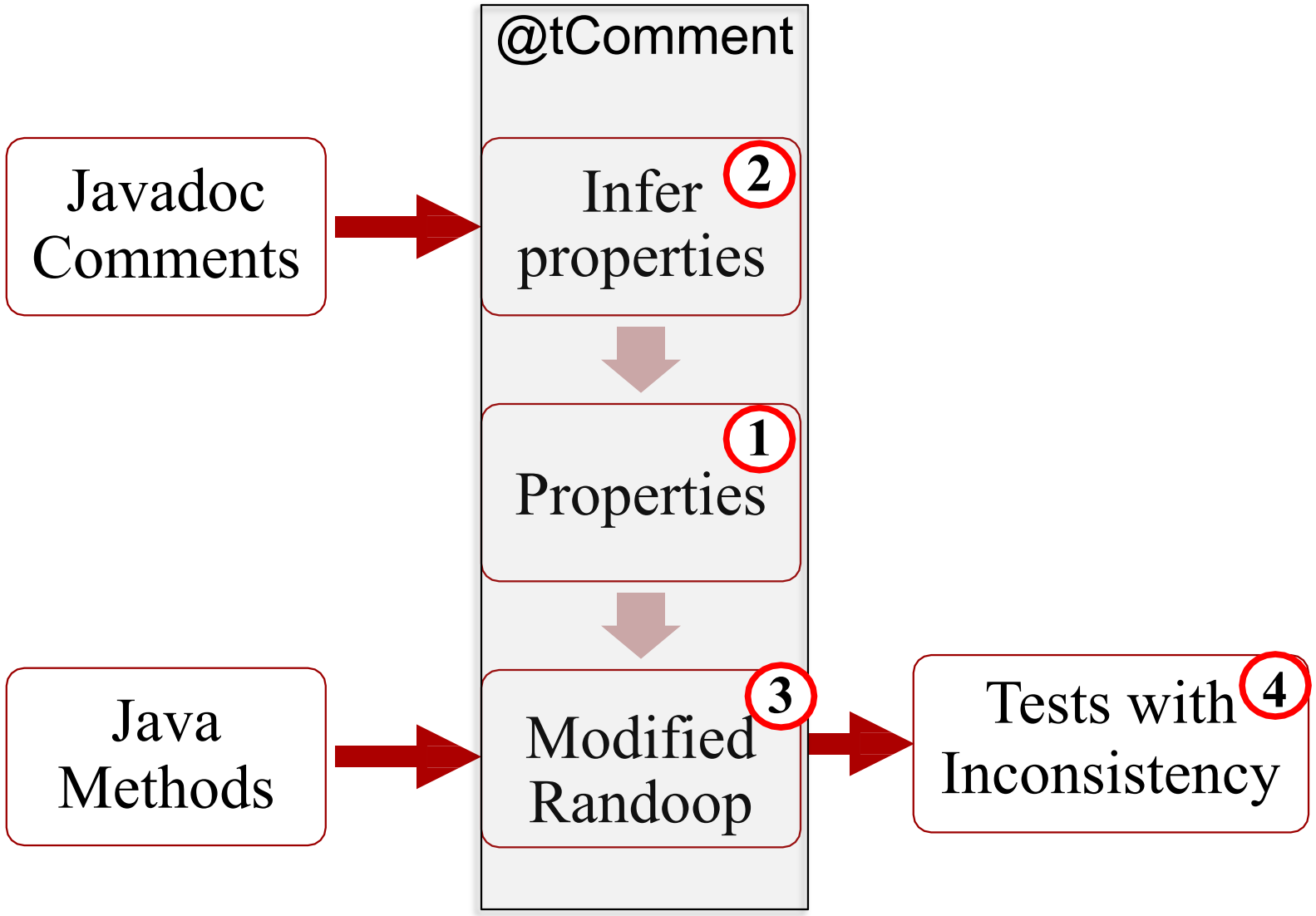
```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```

✓ Confirmed & fixed by JFreeChart developers

@tComment

```
public void test2() throws Throwable {
    ...CategoryPointerAnnotation var0 =
                    new …CategoryPointerAnnotation(  “$0.00”,
                    (java.lang.Comparable)'#', 10.0d, 10.0d);
    org.jfree.ui.TextAnchor var1 = null;
    try {
            var0.setRotationAnchor(var1);
            fail("Expected exception but got Normal Execution");
        } catch (Exception expected) {}
}
```

@tComment Design



Example Inconsistency of Type 2: Fault in Code, Correct Comment

```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```

@tComment

```
public void test2() throws Throwable {
    ...CategoryPointerAnnotation var0 =
                    new …CategoryPointerAnnotation(  “$0.00”,
                    (java.lang.Comparable)'#', 10.0d, 10.0d);
    org.jfree.ui.TextAnchor var1 = null;
    try {
            var0.setRotationAnchor(var1);
            fail("Expected exception but got Normal Execution");
        } catch (Exception expected) {}
}
```

Example Inconsistency of Type 2: Fault in Code, Correct Comment

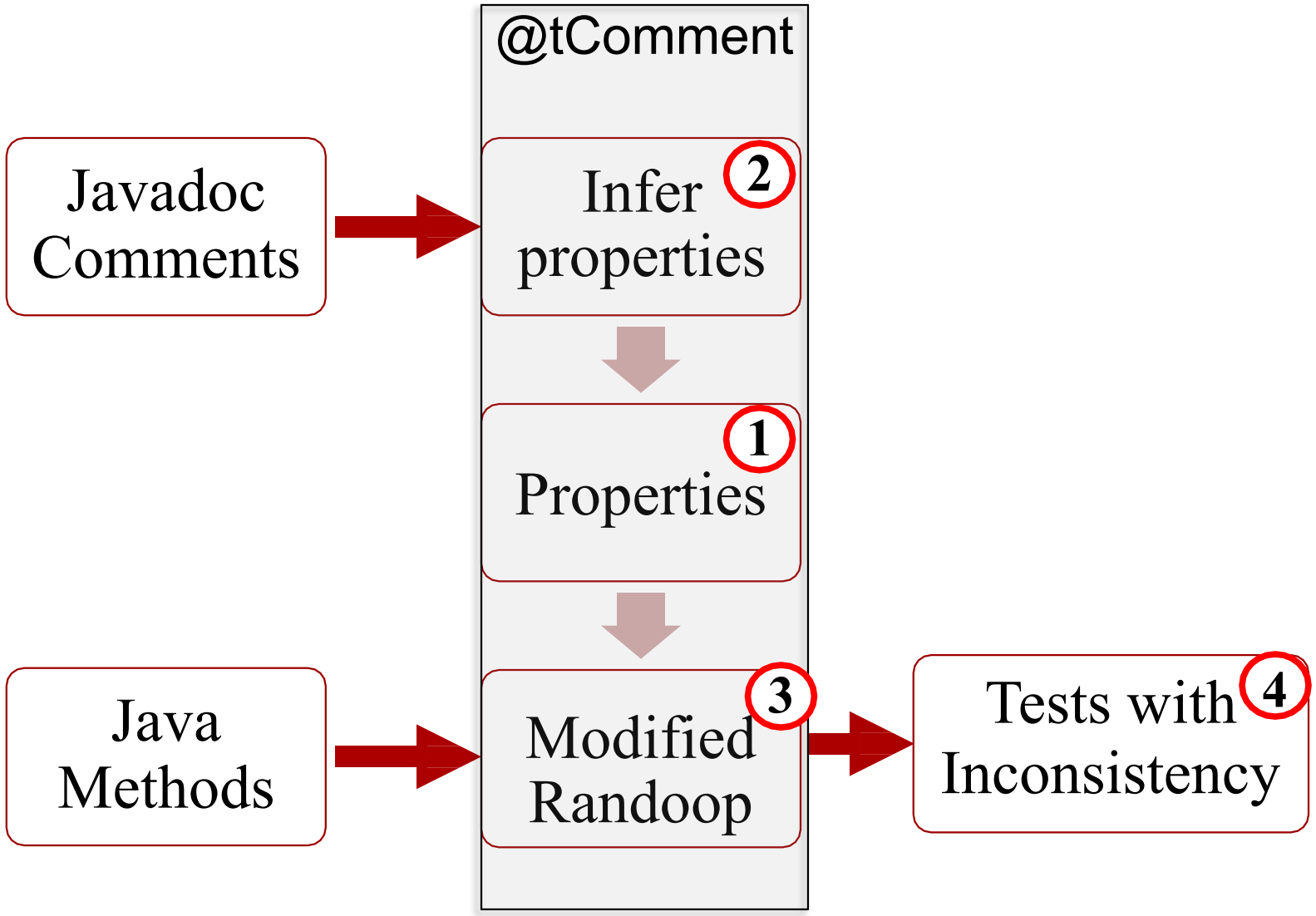
```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```

✓ Confirmed & fixed by JFreeChart developers

@tComment

```
public void test2() throws Throwable {
    ...CategoryPointerAnnotation var0 =
                    new …CategoryPointerAnnotation(  “$0.00”,
                    (java.lang.Comparable)'#', 10.0d, 10.0d);
    org.jfree.ui.TextAnchor var1 = null;
    try {
            var0.setRotationAnchor(var1);
            fail("Expected exception but got Normal Execution");
        } catch (Exception expected) {}
}
```

@tComment Design



Software Reuse and Component-Based Software Engineering

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

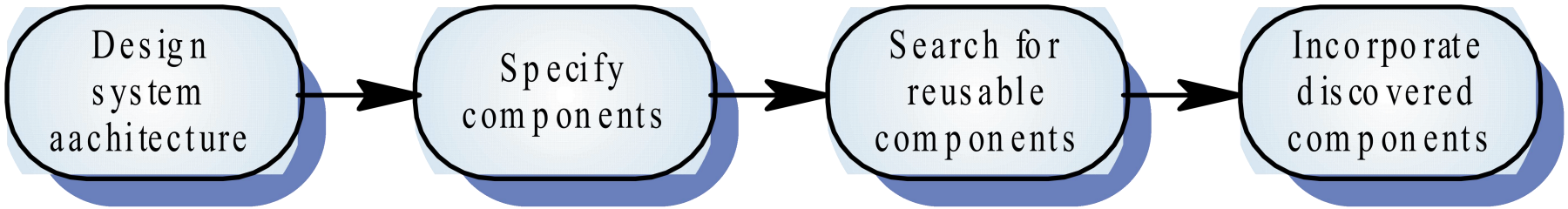
Software Reuse

- In most engineering disciplines, systems are designed by composition (building system out of components that have been used in other systems)
- Software engineering has focused on custom development of components
- To achieve better software quality, more quickly, at lower costs, software engineers are beginning to adopt *systematic reuse* as a design process

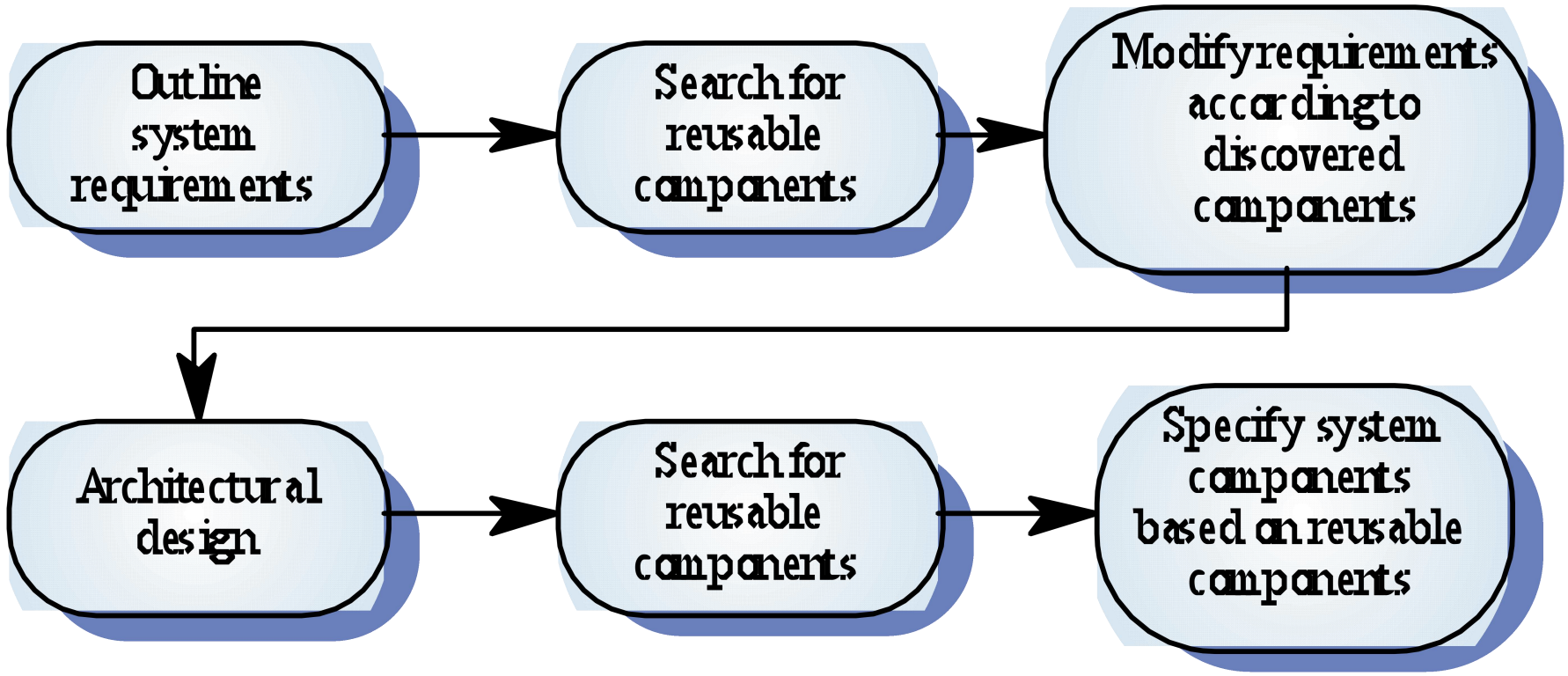
Types of Software Reuse

- Application System Reuse
  - reusing an entire application by incorporation of one application inside another (COTS reuse)
  - development of application families (e.g. MS Office)
- Component Reuse
  - components (e.g. subsystems or single objects) of one application reused in another application
- Function Reuse
  - reusing software components that implement a single well-defined function

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.

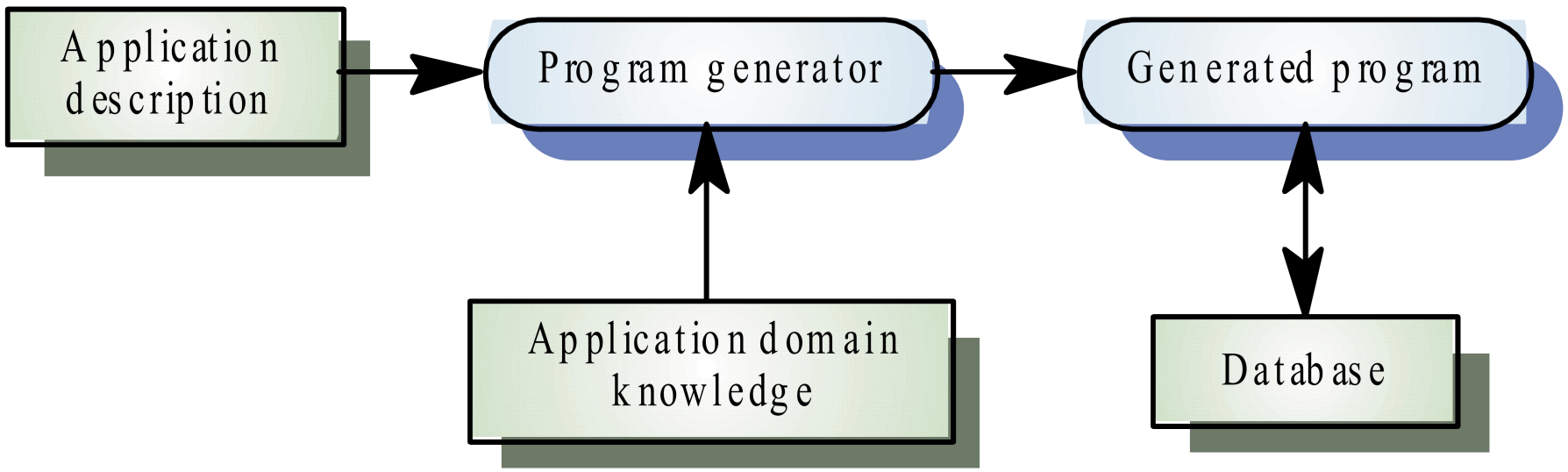
Generator-Based Reuse

- 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

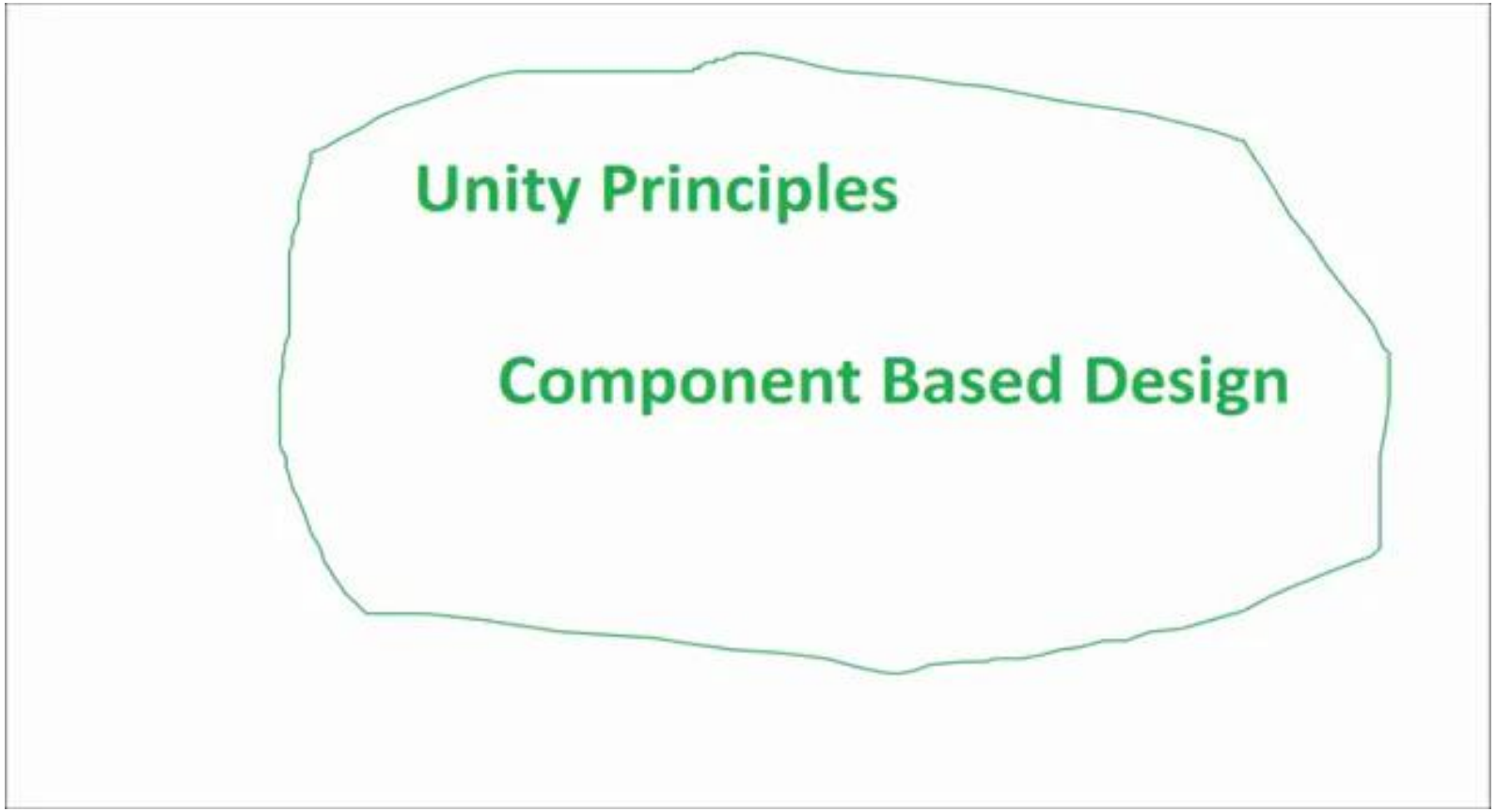


Assessing Program Generator Reuse

- 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

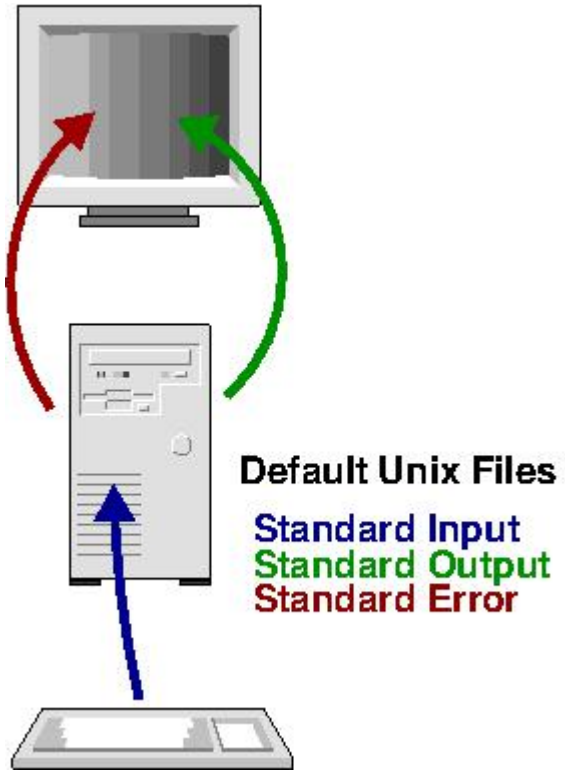
- Abstract
  1: disassociated from any specific instance
  2: difficult to understand

- 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

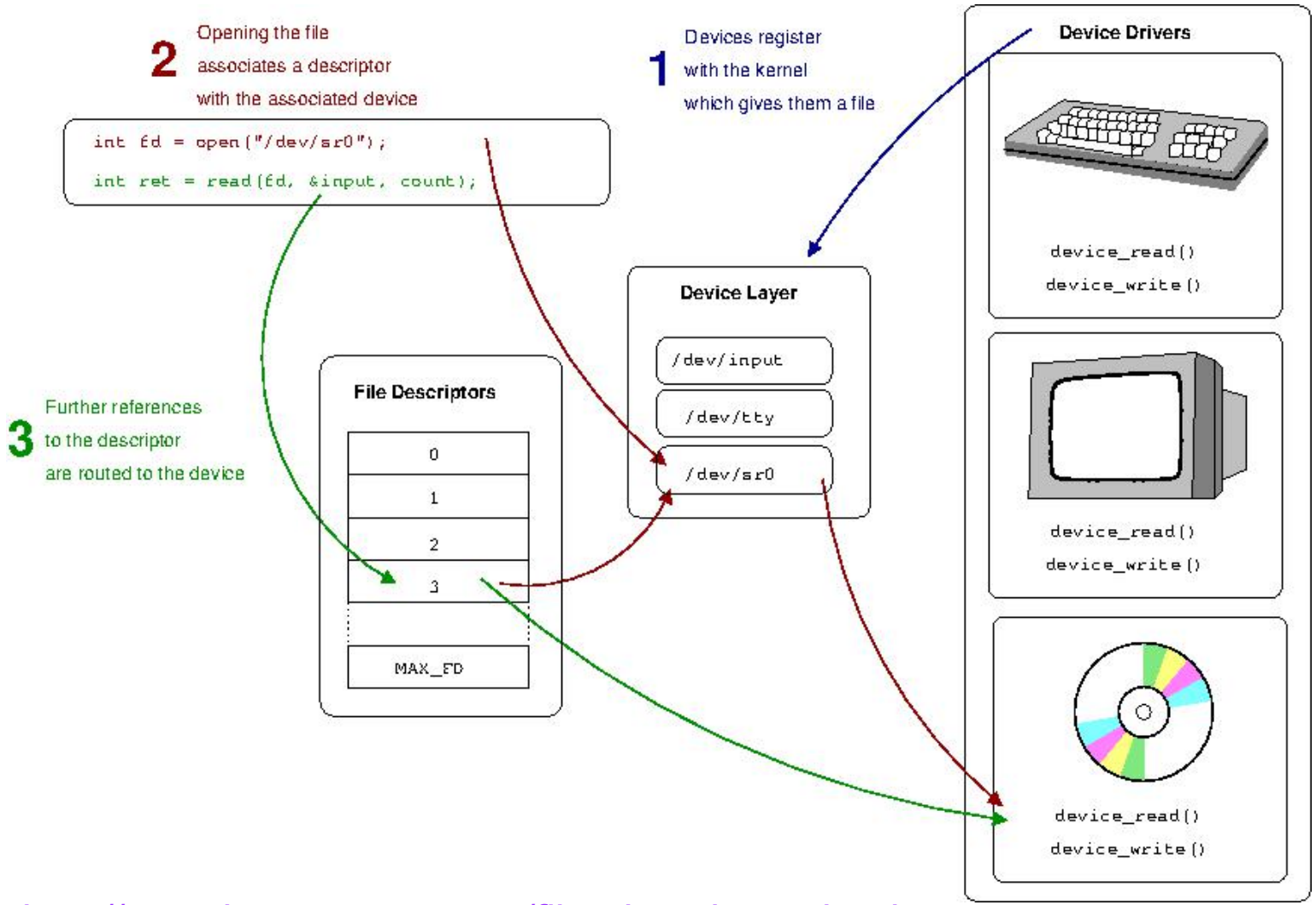
- Unix file descriptor
- Can read, write, and (maybe) seek
- File, IO device, pipe



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?

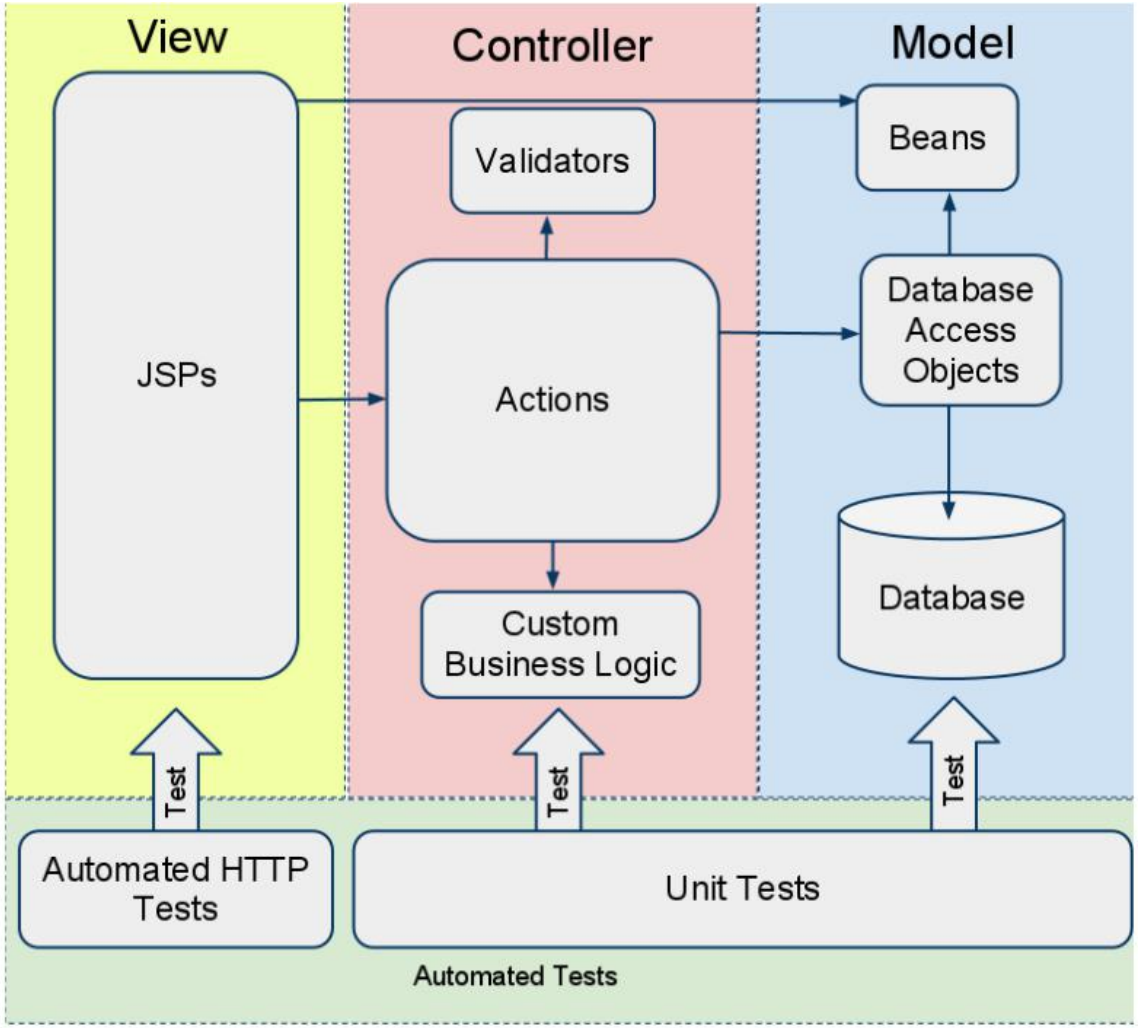
  for i := 1 to length { getItemNumbered(i) }

- 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

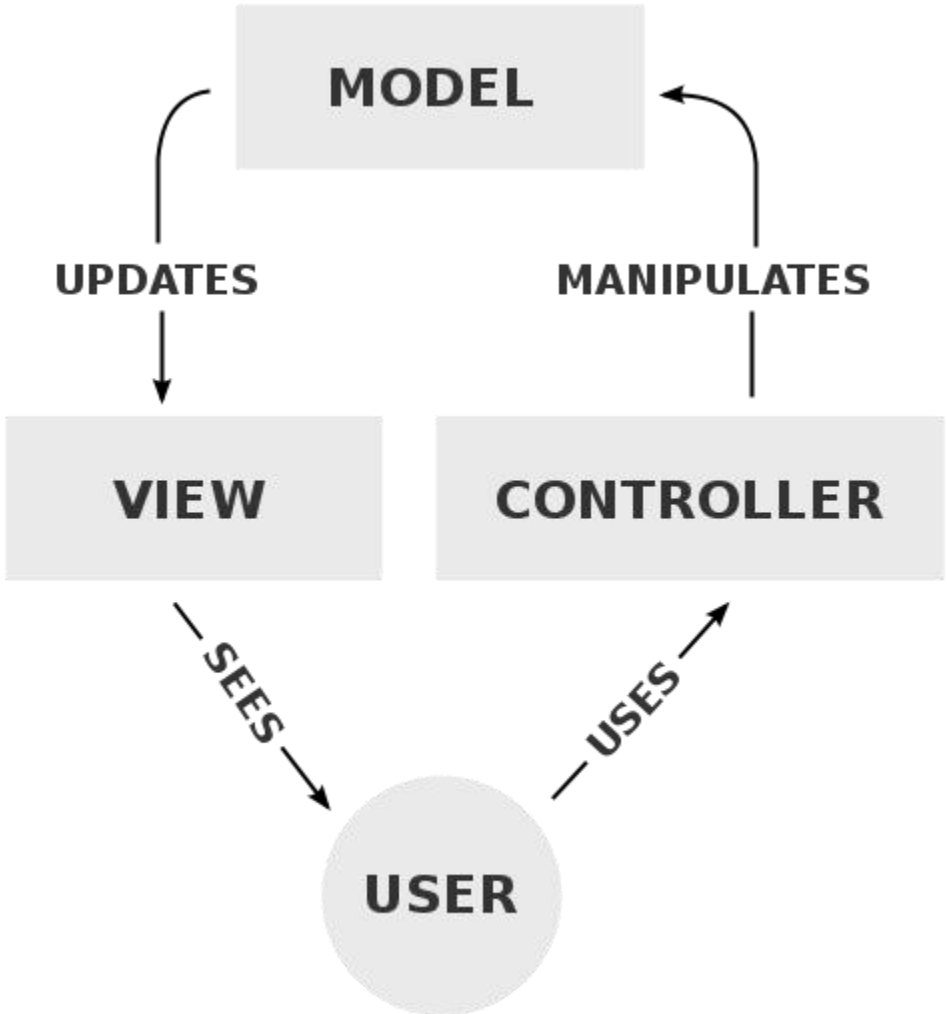


http://bensmith.s3.amazonaws.com/website/papers/sst2011.pdf

Picking an architecture

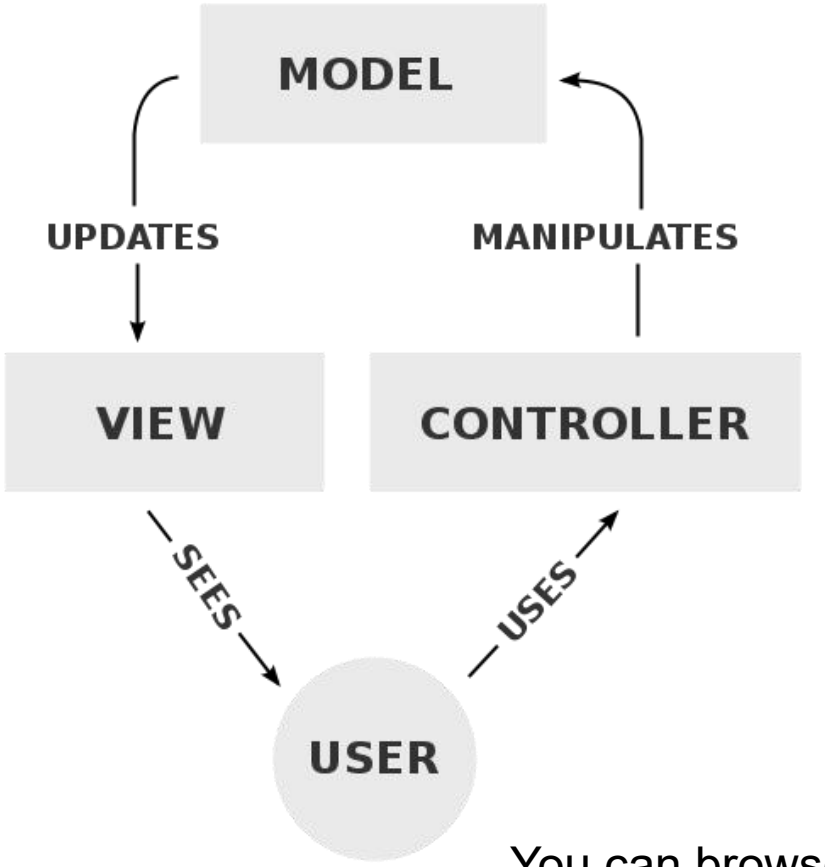
- Many standard architectures
  - Each has strengths and weaknesses
  - Each solves some problems & creates others
- How do you pick an architectural style?
- What is important?
  - Flexibility/ease of change, efficiency, reliability
- More in reading on Wiki (note architectural patterns, including MVC)

MVC Pattern



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

MVC-View: iTrust View



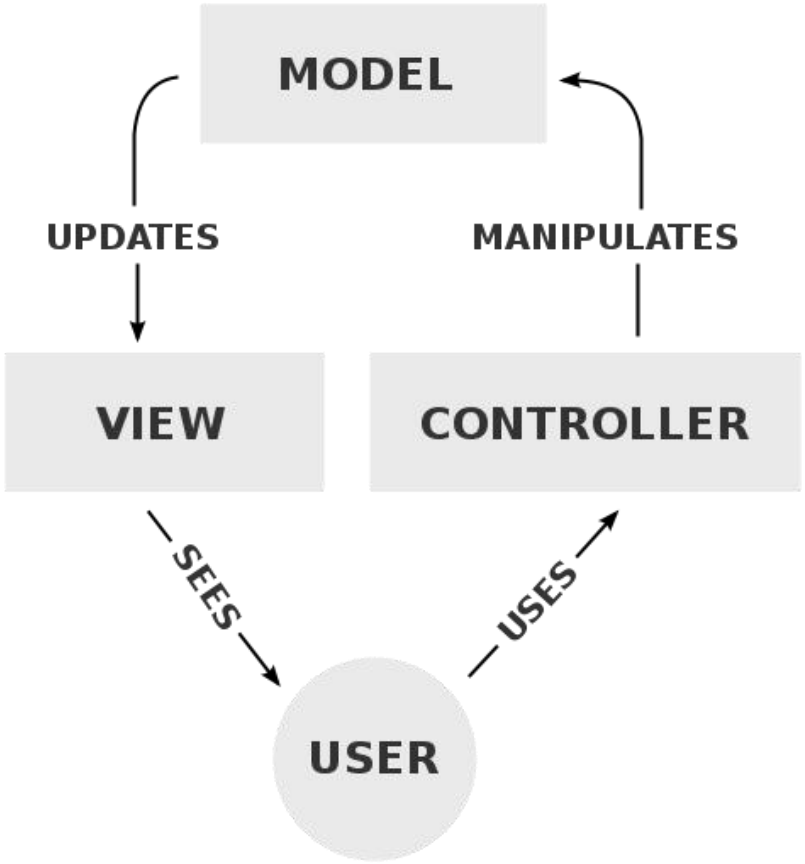
- Little logic; just display info
- JSPs

- 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

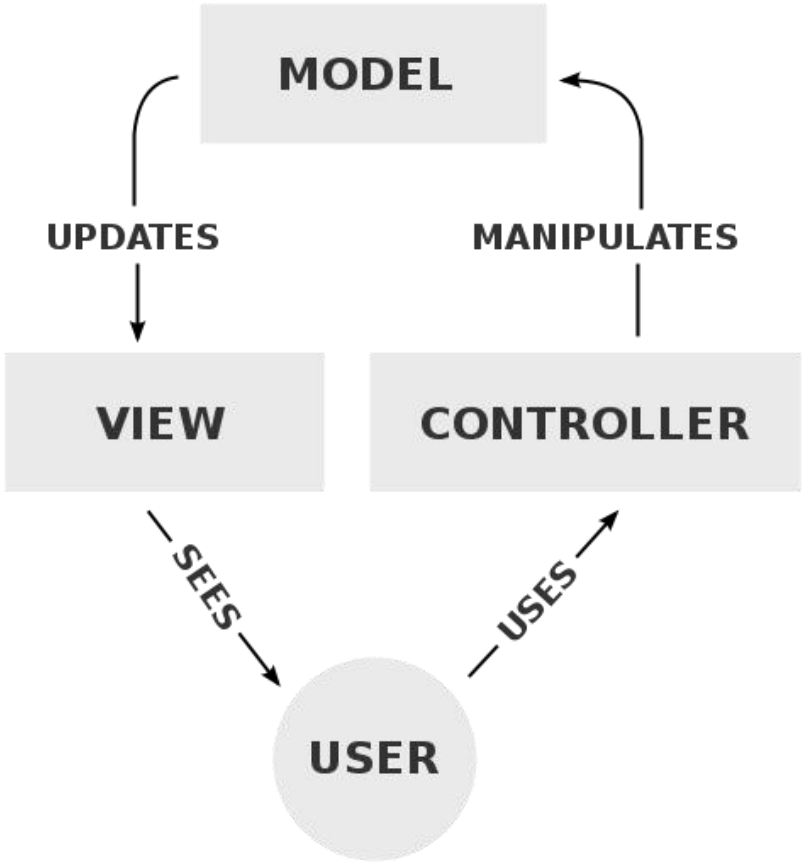
MVC-View: iTrust Model



- 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

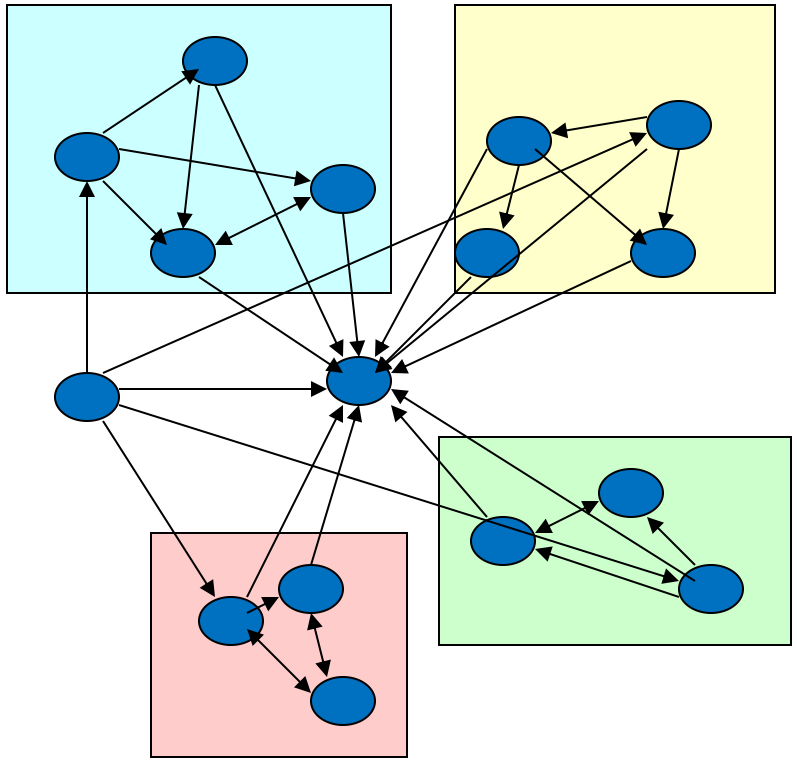
- Maximize cohesion

Cohesion

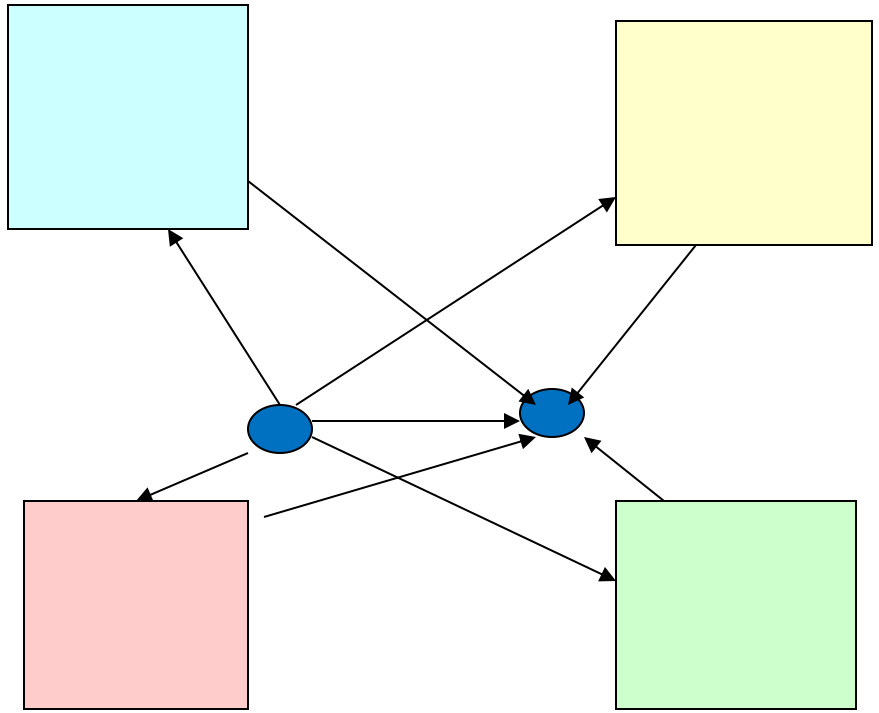
- Coincidental - grouped by chance
- Logical - same idea
- Temporal - same time
  - e.g., placing together all the code used during system start-up or initialization
- Procedural - one calls the other
- Communicational - shared data

- 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?