cs304
Software Engineering

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

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

Arrangement for Week 8

- Week 10 is deadline free week so we have extended the deadline for Progress Report to 25 April 2022.
- Below is the arrangement due to the holiday on Week 8:
  - Lecture at usual time (10.20am-12.10pm) today (2 April)
  - Monday Lab at usual time today (2 April)
  - Tuesday Lab will be on April 24
  - Wednesday Lab at usual time (6 April)

RECAP: FORMAT OF A PATTERN

If It Ain't Broke, Don't Fix It

The name is usually an action phrase.

Intent: Save your reengineering effort for the parts of the system that will make a difference.

The intent should capture the essence of the pattern

Problem

Which parts of a legacy system should you reengineer?

The problem is phrased as a simple question. Sometimes the context is explicitly described.

This problem is difficult because:

- Legacy software systems can be large and complex.
- Rewriting everything is expensive and risky.

Yet, solving this problem is feasible because:

- Reengineering is always driven by some concrete goals.

Next we discuss the forces! They tell us why the problem is difficult and interesting. We also pinpoint the key to solving the problem.

Solution

Only fix the parts that are "broken" — that can no longer be adapted to planned changes.

The solution sometimes includes a recipe of steps to apply the pattern.

Tradeoffs

Pros You don't waste your time fixing things that are not only your critical path.

Each pattern entails some positive and negative tradeoffs.

Cons Delaying repairs that do not seem critical may cost you more in the long run.

Difficulties It can be hard to determine what is "broken".

There may follow a realistic example of applying the pattern.

Rationale

There may well be parts of the legacy system that are ugly, but work well and do not pose any significant maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.

We explain why the solution makes sense.

Known Uses

Alan M. Davis discusses this in his book, *201 Principles of Software Development* .

We list some well documented instances of the pattern.

Related Patterns

Be sure to Fix Problems, Not Symptoms.

Related patterns may suggest alternative actions. Other patterns may suggest logical followup action.

What Next

Consider starting with the Most Valuable First.

MAKING CHANGES TO LEARN

- Pick simple problems that are easy to fix
  - Fix bugs
  - Add simple features
  - Restructure
- Purpose is learning, not fixing
- Pick problems in various parts of system

EXISTING DOCUMENTATION

- Requirements
- Architectural overview
- Design of components
- Interface descriptions
- User manuals
- Code comments

WRITE NEW DOCUMENTATION

- Describe what exists
- Describe what you had to learn
- Simplify and abstract
- Describe
  - The problem
  - The overall design
  - The pieces
  - Examples

TESTS

- Tests can be documentation
- Executing tests helps you see how system works
- Write tests to learn
- Write tests to document what you've learned
- Systems with bad structure are hard to test

DON'T GET STUCK!

- Try many techniques! Be proactive!
- Talk to people
- Get help, help others
- Stick with it
- Take notes; otherwise you forget

SUMMARY

- Reverse engineering is analyzing an existing system
- Instead of inventing design ideas, discover them
- Tools useful, but people are more important and hard work is essential
- Come to reverse engineering lab next week!

REVERSE ENGINEERING DEMO

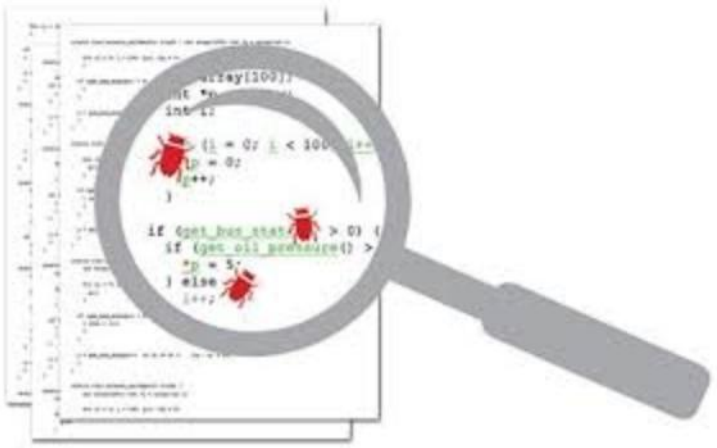
From: https://www.youtube.com/watch?v=zxr_j5XkTT8

WHAT COULD YOU LEARN FROM THE DEMO?

- Given an application "A",
  - Run A to see how it works
    - In the video: Enter two inputs and show "..invalid..."
  - Use the knowledge gained from the run
    - ➢ A has part that process a string "...invalid..."
  - Decompile the application and search for the string
    - Find the location where the string is at
  - Change/ Mutate the condition guarding the string
    - Change "eq" to "neq"
  - Successfully modified a Java application!
  - ✓ Could apply the same knowledge when you want to understand a large software project

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.

Tools for Static Analysis

- Checkstyle
- PMD
- FindBugs

CheckStyle tool

What is Checkstyle?

Checkstyle is an open-source development tool to help programmers write Java code that adheres to a coding standard

It automates the process of checking Java code to verify code following standard or not. This makes it ideal for projects that want to enforce a coding standard.

Official web site for releases and usage guide-line document http://checkstyle.sourceforge.net/

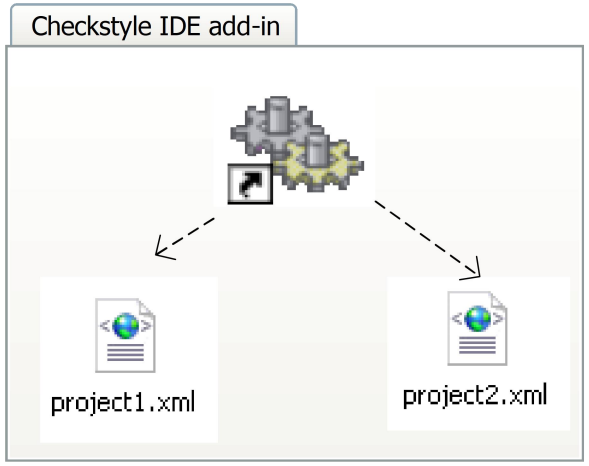
From: https://hienlth.info/java/Ebooks/ThamKhao/BTJB_Lesson05.21_Java%20Code%20Review%20with%20CheckStyle.ppt

Checkstyle

- Focus on Java coding style and standards.
  - whitespace and indentation
  - variable names
  - Javadoc commenting
  - code complexity
    - number of statements per method
    - levels of nested ifs/loops
    - lines, methods, fields, etc. per class
  - proper usage
    - import statements
    - regular expressions
    - exceptions
    - I/O
    - thread usage, ...

From: https://courses.cs.washington.edu/courses/cse403/13sp/lectures/11-staticanalysis.ppt

How Checkstyle works



As in the above figure:

- Checkstyle is add-ins component to IDE/Build tool
- Coding standard is user pre-defined, and embedded in each XML file
- Checkstyle will depend on XML file to parse code, then generate checking result to

PMD

PMD scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/StringBuffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

PMD RuleSets

- Android Rules: These rules deal with the Android SDK.
- Basic JSF rules: Rules concerning basic JSF guidelines.
- Basic JSP rules: Rules concerning basic JSP guidelines.
- Basic Rules: The Basic Ruleset contains a collection of good practices which everyone should follow.
- Braces Rules: The Braces Ruleset contains a collection of braces rules.
- Clone Implementation Rules: The Clone Implementation ruleset contains a collection of rules that find questionable usages of the clone() method.
- Code Size Rules: The Code Size Ruleset contains a collection of rules that find code size related problems.
- Controversial Rules: The Controversial Ruleset contains rules that, for whatever reason, are considered controversial.
- Coupling Rules: These are rules which find instances of high or inappropriate coupling between objects and packages.
- Design Rules: The Design Ruleset contains a collection of rules that find questionable designs.
- Import Statement Rules: These rules deal with different problems that can occur with a class' import statements.
- J2EE Rules: These are rules for J2EE
- JavaBean Rules: The JavaBeans Ruleset catches instances of bean rules not being followed.

PMD RuleSets: Continue

- JUnit Rules: These rules deal with different problems that can occur with JUnit tests.
- Jakarta Commons Logging Rules: Logging ruleset contains a collection of rules that find questionable usages.
- Java Logging Rules: The Java Logging ruleset contains a collection of rules that find questionable usages of the logger.
- Migration Rules: Contains rules about migrating from one JDK version to another.
- Migration15: Contains rules for migrating to JDK 1.5
- Naming Rules: The Naming Ruleset contains a collection of rules about names - too long, too short, and so forth.
- Optimization Rules: These rules deal with different optimizations that generally apply to performance best practices.
- Strict Exception Rules: These rules provide some strict guidelines about throwing and catching exceptions.
- String and StringBuffer Rules: Problems that can occur with manipulation of the class String or StringBuffer.
- Security Code Guidelines: These rules check the security guidelines from Sun.
- Type Resolution Rules: These are rules which resolve java Class files for comparisson, as opposed to a String
- Unused Code Rules: The Unused Code Ruleset contains a collection of rules that find unused code.

PMD Rule Example

PMD Basic Rules

- **EmptyCatchBlock:** Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.
- **EmptyIfStmt:** Empty If Statement finds instances where a condition is checked but nothing is done about it.
- **EmptyWhileStmt:** Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use Thread.sleep() for it; if it's a while loop that does a lot in the exit expression, rewrite it to make it clearer.
- **EmptyTryBlock:** Avoid empty try blocks - what's the point?
- **EmptyFinallyBlock:** Avoid empty finally blocks - these can be deleted.
- **EmptySwitchStatements:** Avoid empty switch statements.
- **JumbledIncrementer:** Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended.
- **ForLoopShouldBeWhileLoop:** Some for loops can be simplified to while loops - this makes them more concise.

Findbugs

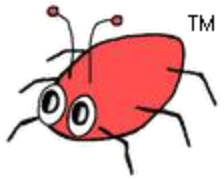
Adapted from swtv.kaist.ac.kr/courses/cs492-spring-13/lec9-findbugs.ppt

Tin Bui-Huy

September, 2009

Content

- What is Findbugs?
- How to use Findbugs?



What is FindBugs?

- Result of a research project at the University of Maryland
- Based on the concept of *bug patterns*. A bug pattern is a code idiom that is often an error.
  - Difficult language features
  - Misunderstood API methods
  - Misunderstood invariants when code is modified during maintenance
  - Garden variety mistakes: typos, use of the wrong boolean operator
- FindBugs uses *static analysis* to inspect Java bytecode for occurrences of bug patterns.
- Static analysis means that FindBugs can find bugs by simply inspecting a program's code: executing the program is not necessary.
- FindBugs works by analyzing Java bytecode (compiled class files), so you don't even need the program's source code to use it.
- FindBugs can report ***false warnings***, not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.

What is Findbugs?

- Not concerned by formatting or coding standards
- Concentrates on detecting potential bugs and performance issues
- Can detect many types of common, hard-to-find bugs

How it works?

- Use “bug patterns” to detect potential bugs
- Examples

NullPointerException

```
Address address = client.getAddress();
if ((address != null) || (address.getPostCode() != null)) {
 ...
}
```

Uninitialized field

```
public class ShoppingCart {
    private List items;
    public addItem(Item item) {
        items.add(item);
    }
}
```

What Findbugs can do?

- FindBugs comes with over 200 rules divided into different categories:
  - *Correctness*

    E.g. infinite recursive loop, reads a field that is never written
  - *Bad practice*

    E.g. code that drops exceptions or fails to close file
  - *Performance*
  - *Multithreaded correctness*
  - *Dodgy*
    - E.g. unused local variables or unchecked casts

How to use Findbugs?

- Standalone Swing application
- Eclipse plug-in
- Integrated into the build process (Ant or Maven)

FindBugs’ warnings

1. AT: Sequence of calls to concurrent abstraction may not be atomic
2. DC: Possible double check of field
3. DL: Synchronization on Boolean
4. DL: Synchronization on boxed primitive
5. DL: Synchronization on interned String
6. DL: Synchronization on boxed primitive values
7. Dm: Monitor wait() called on Condition
8. Dm: A thread was created using the default empty run method
9. ESync: Empty synchronized block
10. IS: Inconsistent synchronization
11. IS: Field not guarded against concurrent access
12. JLM: Synchronization performed on Lock
13. JLM: Synchronization performed on util.concurrent instance
14. JLM: Using monitor style wait methods on util.concurrent abstraction
15. LI: Incorrect lazy initialization of static field
16. LI: Incorrect lazy initialization and update of static field
17. ML: Synchronization on field in futile attempt to guard that field
18. ML: Method synchronizes on an updated field
19. MSF: Mutable servlet field
20. MWN: Mismatched notify()
21. MWN: Mismatched wait()
22. NN: Naked notify
23. NP: Synchronize and null check on the same field.

24. No: Using notify() rather than notifyAll()
25. RS: Class's readObject() method is synchronized
26. RV: Return value of putIfAbsent ignored, value passed to putIfAbsent reused
27. Ru: Invokes run on a thread (did you mean to start it instead?)
28. SC: Constructor invokes Thread.start()
29. SP: Method spins on field
30. STCAL: Call to static Calendar
31. STCAL: Call to static DateFormat
32. STCAL: Static Calendar field
33. STCAL: Static DateFormat
34. SWL: Method calls Thread.sleep() with a lock held
35. TLW: Wait with two locks held
36. UG: Unsynchronized get method, synchronized set method
37. UL: Method does not release lock on all paths
38. UL: Method does not release lock on all exception paths
39. UW: Unconditional wait
40. VO: An increment to a volatile field isn't atomic
41. VO: A volatile reference to an array doesn't treat the array elements as volatile
42. WL: Synchronization on getClass rather than class literal
43. WS: Class's writeObject() method is synchronized but nothing else is
44. Wa: Condition.await() not in loop
45. Wa: Wait not in loop

Defensive Programming

Making sure that no warning produced by Static Analysis is a form of defensive programming

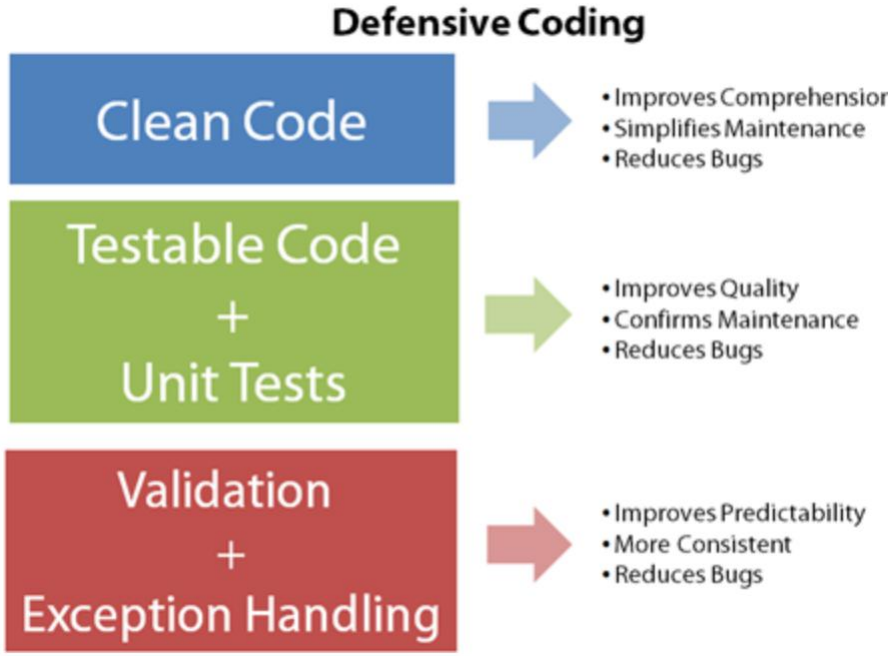
What if you receive the following project requirement?

- should allow manual and autonomous car driving
- it has to gather data from many attached sensors
- it has to send the gathered data 54M km away
- it cannot be tested in the production environment
- it's hard to apply hotfixes and patches after the release
- if something goes wrong with the software we will lose $2B and waste many years of work
- the entire OS and software should run on 10Mb of RAM
- you do not have physical access to the hardware after release
- the avg communication lag with the software is 15 minutes

From: https://coder.today/tech/2017-11-09_nasa-coding-standards-defensive-programming-and-reliability-a-postmortem-static-analysis./

What is Defensive Programming?

- **Defensive programming** is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances.
- Often used when **high availability, safety or security** is needed.



Pic From: https://blogs.msmvps.com/deborahk/what-is-defensive-coding/

Defensive Programming

Murphy's Law:

If anything can go wrong, it will.

Defensive Programming:

- Redundant code(多余代码) is incorporated to check system state after modifications.
- Implicit assumptions (隐式假设) are tested explicitly.
- Risky programming constructs are avoided.

Defensive Programming: Error Avoidance

Risky programming constructs

- Pointers
- Dynamic memory allocation
- Floating-point numbers
- Parallelism
- Recursion
- Interrupts

All are valuable in certain circumstances, but should be used with discretion

Defensive Programming Examples

- Use *boolean* variable not *integer*
- Test $i <= n$ not $i == n$ (greater range)
- Assertion checking (e.g., validate parameters)
- Build debugging code into program with a switch to display values at interfaces
- Error checking codes in data (e.g., checksum or hash)

Maintenance

Most production programs are maintained by people other than the programmers who originally wrote them.
(a) What factors make a program easy for somebody else to maintain?
(b) What factors make a program hard for somebody else to maintain?

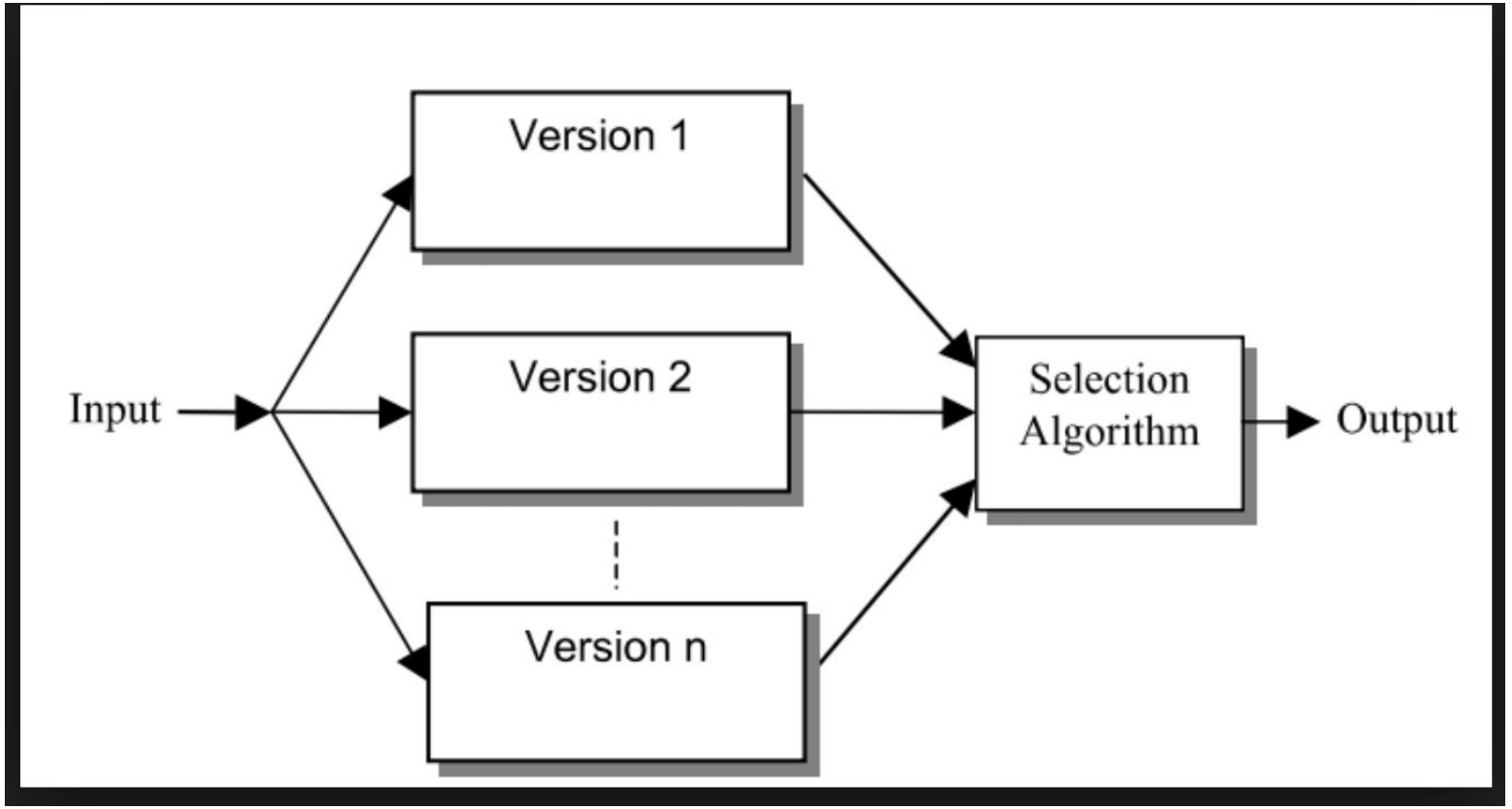
Fault Tolerance

General Approach:

- Failure detection
- Damage assessment
- Fault recovery
- Fault repair

N-version programming -- Execute independent implementation in parallel, compare results, accept the most probable.

N-version Programming



From: https://www.researchgate.net/figure/N-Version-programming-model-Torres-pomales-and-Langley-2000_fig11_324721688

Fault Tolerance

Basic Techniques:

- After error continue with next transaction (e.g., drop packet)
- Timers and timeout in networked systems
- User break options (e.g., force quit, cancel)
- Error correcting codes in data
- Bad block tables on disk drives
- Forward and backward pointers in databases

Report all errors for quality control

Fault Tolerance

Backward Recovery:

- Record system state at specific events (**checkpoints**). After failure, recreate state at last checkpoint.
- **Backup** of files
- Combine checkpoints with system **log** (**audit trail** of transactions) that allows transactions from last checkpoint to be repeated automatically.
- ***Test the restore software!***

Software Engineering for Real Time

The special characteristics of real time computing require extra attention to good software engineering principles:

- Requirements analysis and specification
- Special techniques (e.g., locks on data, semaphores, etc.)
- Development of tools
- Modular design
- Exhaustive testing

Software Engineering for Real Time

Testing and debugging need special tools and environments

- Debuggers, etc., cannot be used to test real time performance
- Simulation of environment may be needed to test interfaces -- e.g., adjustable clock speed
- General purpose tools may not be available

Some Notable Bugs

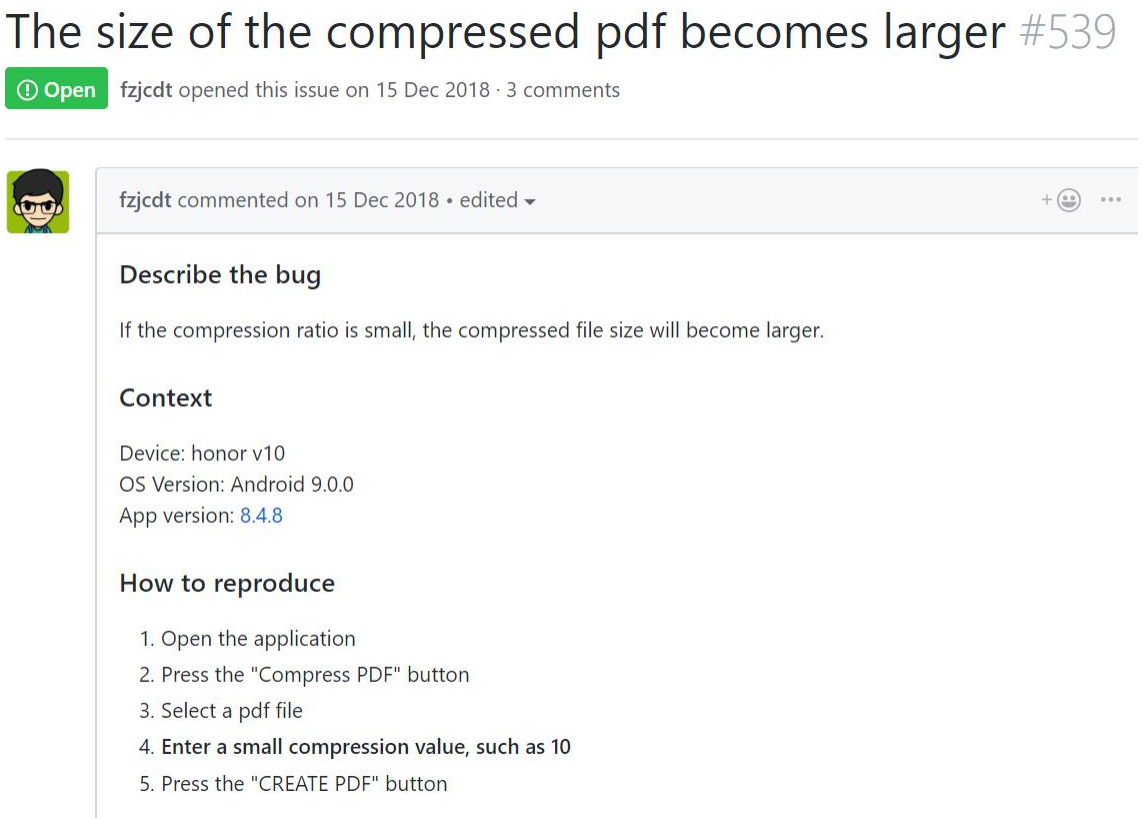
Even commercial systems may have horrific bugs

- Built-in function in Fortran compiler ($e^0 = 0$)
- Japanese microcode for Honeywell DPS virtual memory
- The microfilm plotter with the missing byte (1:1023)
- The Sun 3 page fault that IBM paid to fix
- Left handed rotation in the graphics package

Good people work around problems.

The best people track them down and fix them!

EXAMPLE OF BUGS FOUND BY STUDENTS



The size of the compressed pdf becomes larger #539

Open fzjcdt opened this issue on 15 Dec 2018 · 3 comments

fzjcdt commented on 15 Dec 2018 • edited

Describe the bug

If the compression ratio is small, the compressed file size will become larger.

Context

Device: honor v10
OS Version: Android 9.0.0
App version: 8.4.8

How to reproduce

1. Open the application
2. Press the "Compress PDF" button
3. Select a pdf file
4. **Enter a small compression value, such as 10**
5. Press the "CREATE PDF" button

https://github.com/Swati4star/Images-to-PDF/issues/539

Security in the Software Development Process

The security goal

The security goal is to make sure that the agents (people or external systems) who interact with a computer system, its data, and its resources, are those that the owner of the system would wish to have such interactions.

Security considerations need to be part of the entire software development process. They may have a major impact on the architecture chosen.

Example. Integration of Internet Explorer into Windows

Agents and Components

A large system will have many agents and components:

- each is potentially unreliable and insecure
- components acquired from third parties may have unknown security problems (COTS problem)

The software development challenge:

- develop secure and reliable components
- protect whole system from security problems in parts of it

Techniques: Barriers

Place barriers that separate parts of a complex system:

- Isolate components, e.g., do not connect a computer to a network
- Firewalls
- Require authentication to access certain systems or parts of systems

Every barrier imposes restrictions on permitted uses of the system

Barriers are most effective when the system can be divided into subsystems with simple boundaries

Techniques: Authentication & Authorization

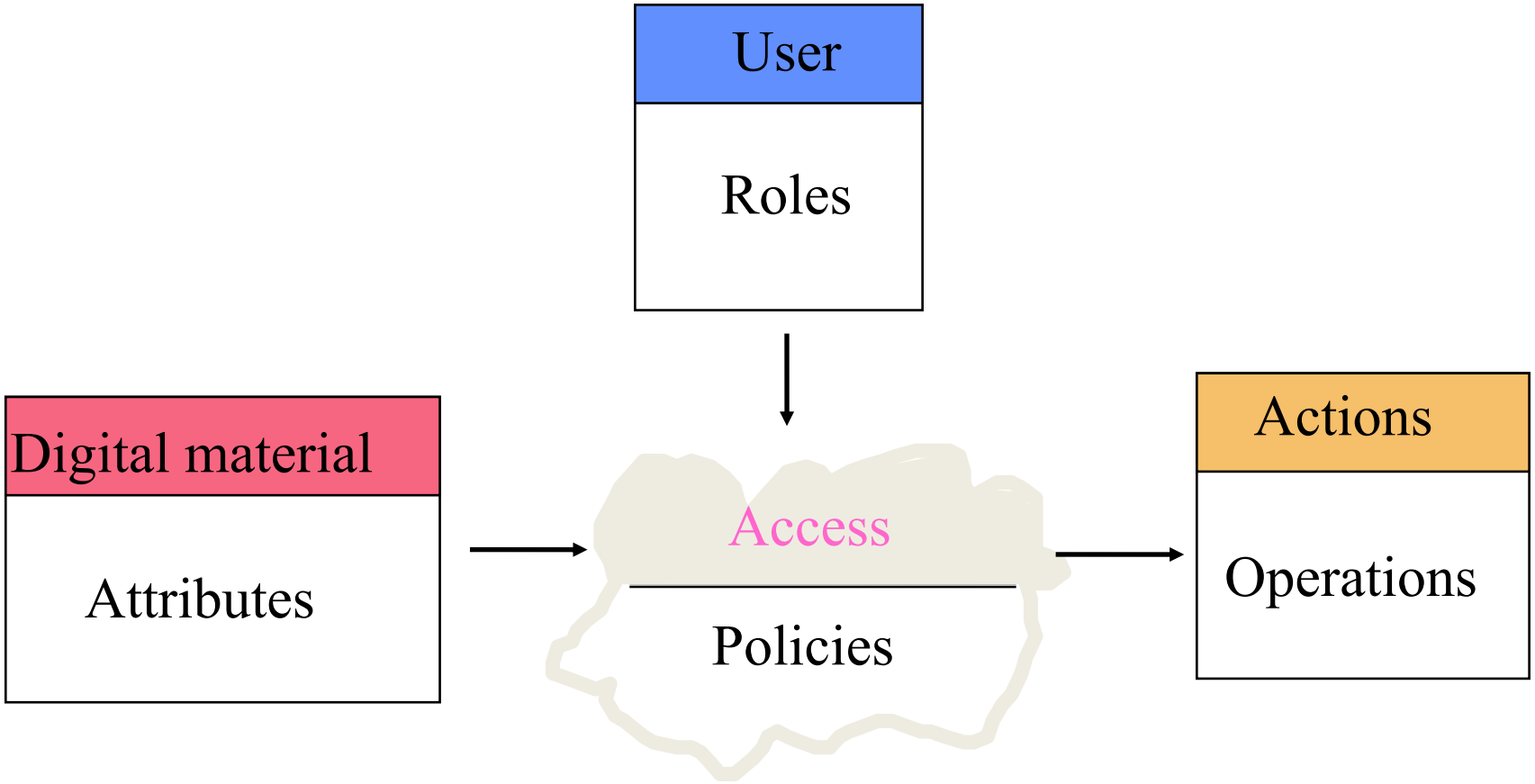
Authentication establishes the identity of an agent:

- What the agent knows (e.g., password)
- What the agent possess (e.g., smart card)
- Where does the agent have access to (e.g., controller)
- What are the physical properties of the agent (e.g., fingerprint)

Authorization establishes what an authenticated agent may do:

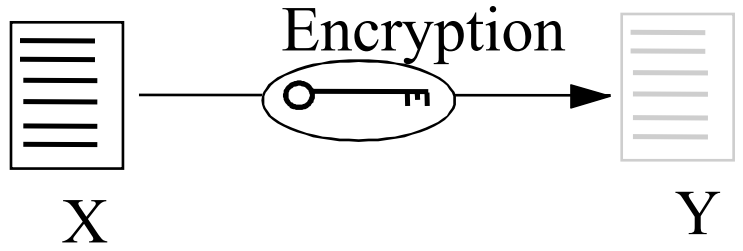
- Access control lists
- Group membership

Example: An Access Model for Digital Content

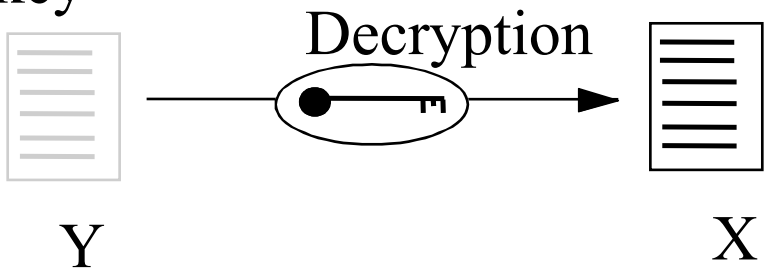


Techniques: Encryption

Allows data to be stored and transmitted securely, even when the bits are viewed by unauthorized agents



- Private key and public key
- Digital signatures



Security and People

People are intrinsically insecure:

- Careless (e.g, leave computers logged on, use simple passwords, leave passwords where others can read them)
- Dishonest (e.g., stealing from financial systems)
- Malicious (e.g., denial of service attack)

Many security problems come from inside the organization:

- In a large organization, there will be some disgruntled and dishonest employees
- Security relies on trusted individuals. What if they are dishonest?

Design for Security: People

- Make it easy for responsible people to use the system
- Make it hard for dishonest or careless people (e.g., password management)
- Train people in responsible behavior
- Test the security of the system
- Do not hide violations

Suggested Reading

Trust in Cyberspace, Committee on Information Systems Trustworthiness, National Research Council (1999)
http://www.nap.edu/readingroom/books/trust/
Fred Schneider, Cornell Computer Science, was the chair of this study.

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

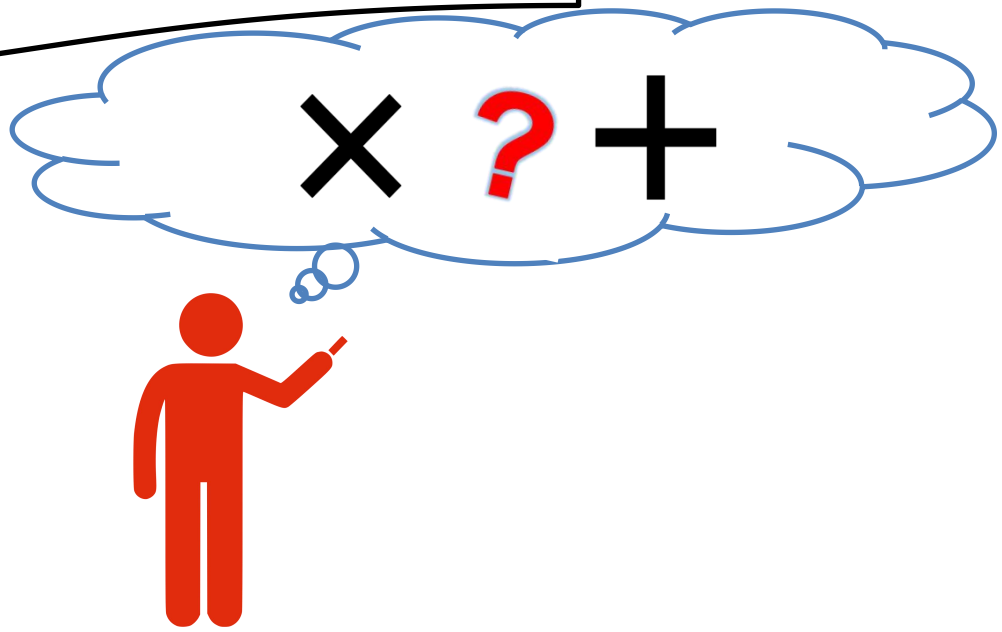
- Document known problems!
  - There are three “standard” flags that you can put into any comment
    - TODO -- describes a feature that should be added
    - FIXME -- describes a bug in the method
    - XXX -- this needs to be thought about some more
  - Eclipse and NetBeans both recognize these flags 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)
```

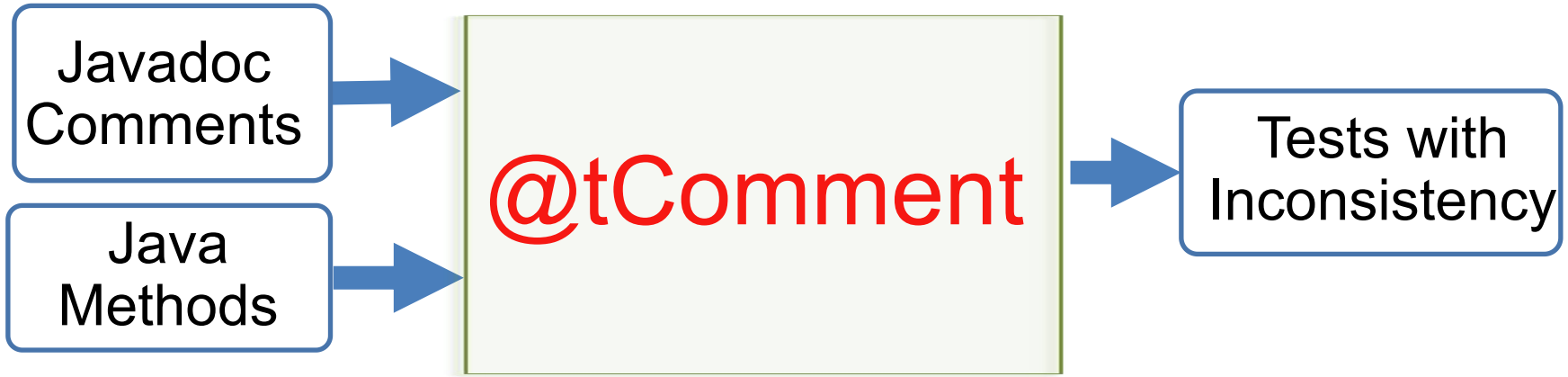
Expected behavior for `synchronizedMap(null)`:

- 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)
```

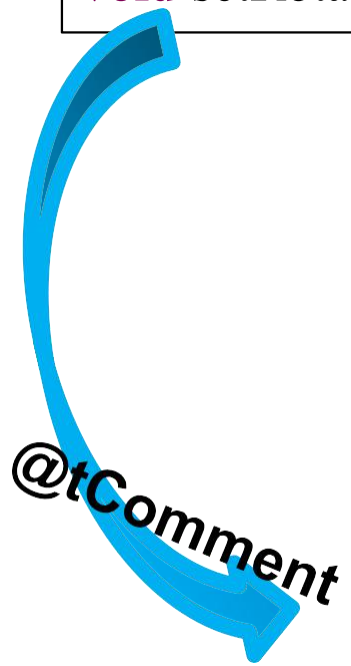
Example Inconsistency of Type 1: Correct Code, Incorrect Comment

```
/*…
* @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

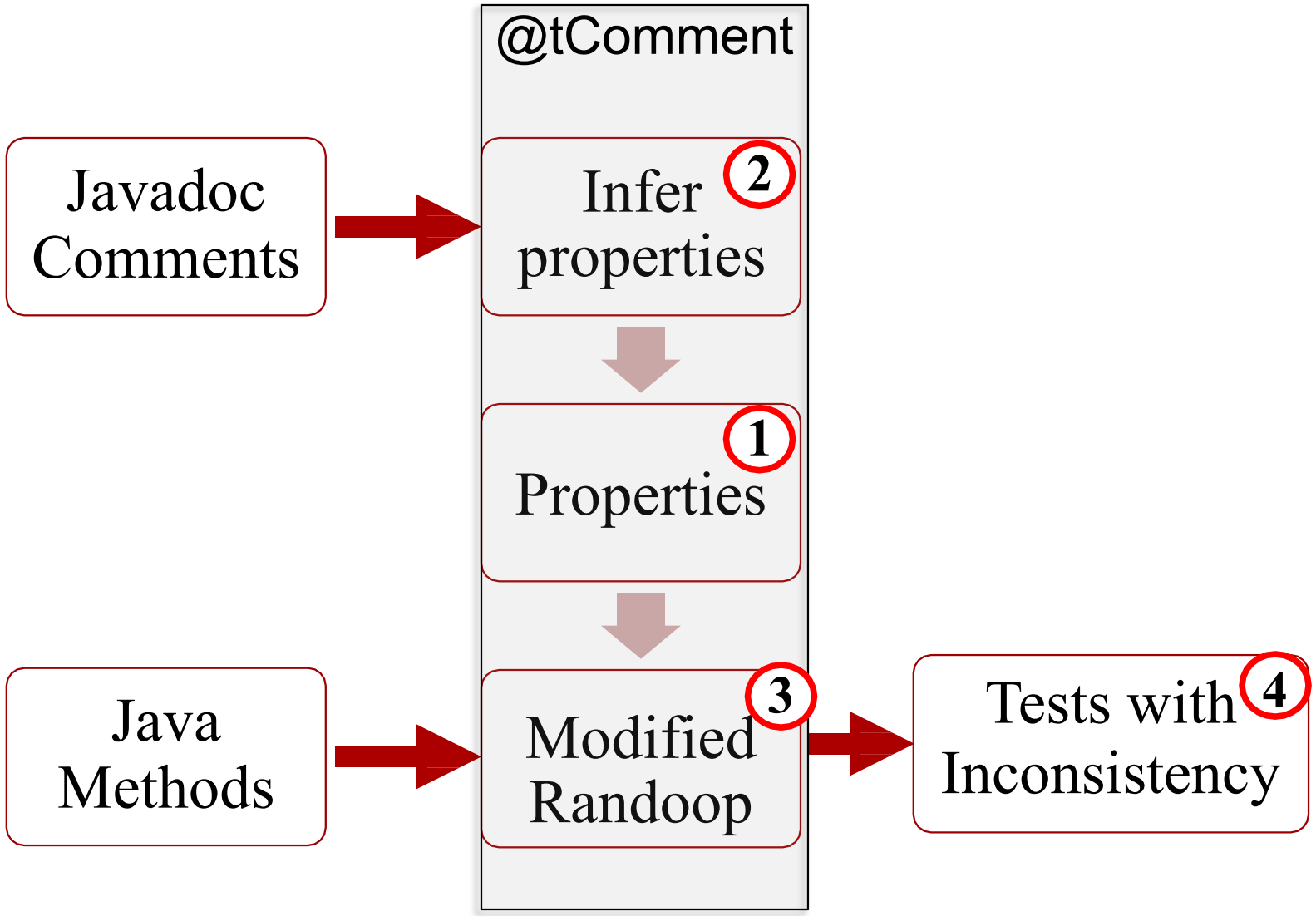
```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```

✓ 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

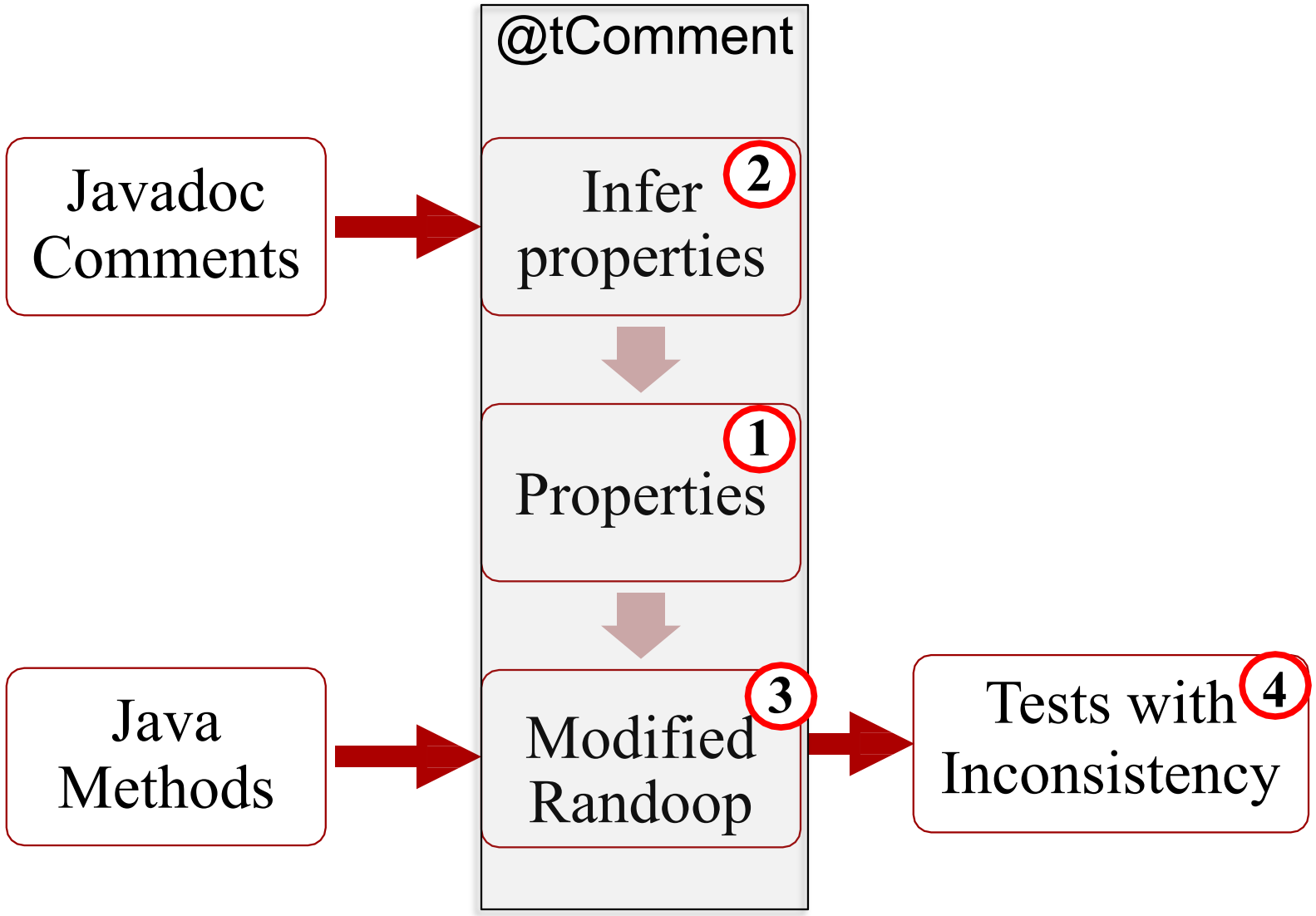
```
/*…
* @param anchor the anchor (<code>null</code> not permitted).
*/
void setRotationAnchor(TextAnchor anchor)
```

✓ 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

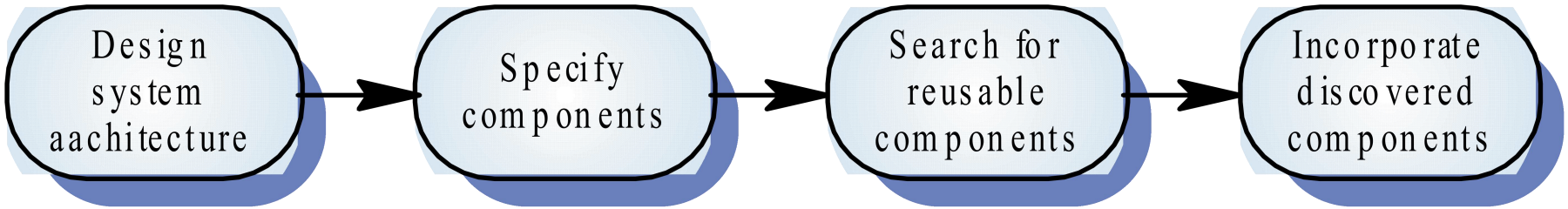
Software Reuse

- 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

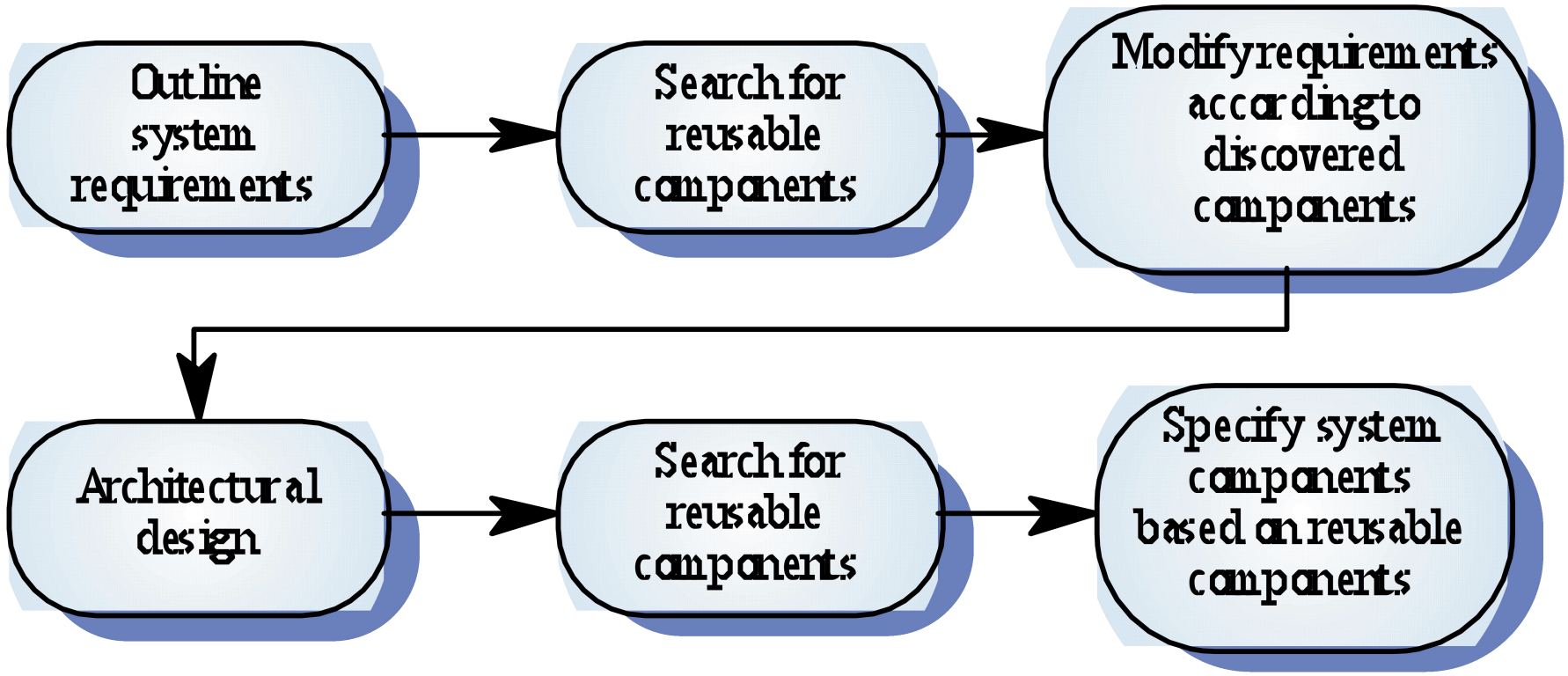
Types of Software Reuse

- 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.

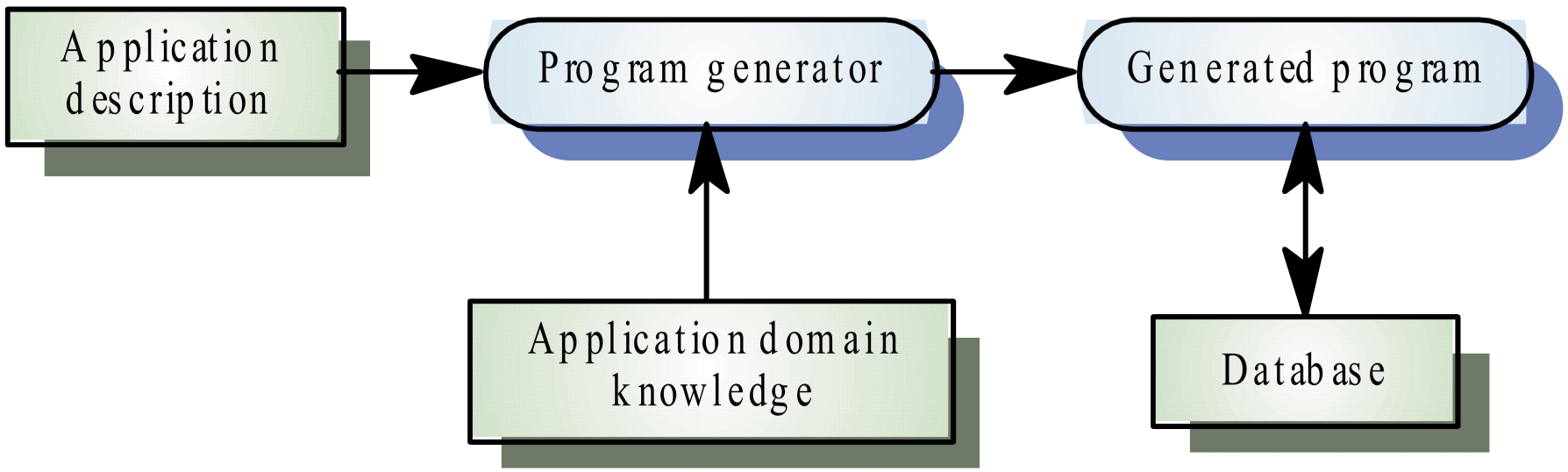
Generator-Based Reuse

- 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

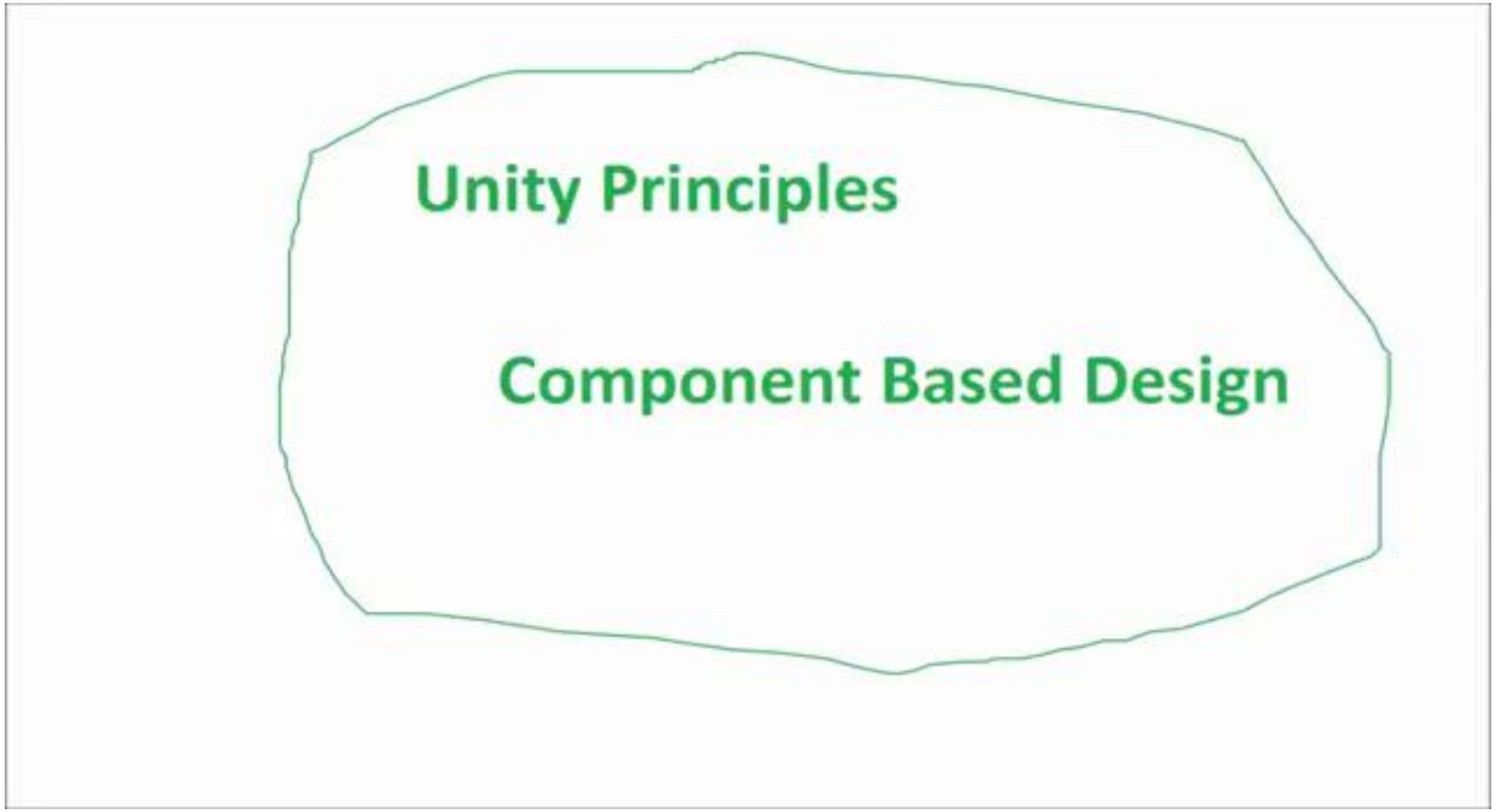


Assessing Program Generator Reuse

- 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