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)

Project

How to start from 0 involving in open source project and resolve their issue?

如何从 0->1，我觉得有几个方面吧:
1. 熟悉 git 操作以及 github workflow
2. 使用某个开源组件(当然也可以从 issue 里面去发现)，然后发现不足
3. 参与进去解决问题

How to go from 0->1, I think there are several aspects of it:
1. Familiarize yourself with git operations and github workflow
2. Use an open source component (or find it in the issue, of course), and then discover the deficiency
3. Engaging in problem solving

WeChat Translate

- I have conducted an Interview with previous SE student helper (王泽淮)
- He has previously contributed to an open source project in GitHub (Kubernates) with an accepted pull request

✓ We have taught you about git and github workflow in one of the lab and you have been using git and Github Classroom for homework submission!

- Use the open source project/libraries that you have selected
  - You could find bugs in them and issue the pull request for the bug that you found! (Starting from finding bugs is actually starting from 0! You are currently starting from a GitHub issue so you already have some information about the bugs)

How to start from 0 involving in open source project and resolve their issue?

- Recorded video with previous student who took SE and ST (徐逸飞)
- He has previously contributed to an open source project in GitHub with an accepted pull request
  - In the software testing class in previous semester, he has gone through all the steps (starting from 0 in the software testing class)!
    1. Find bugs in open-source Android apps
    2. Prepare test case for the bug
    3. File a GitHub issue: https://github.com/SimpleMobileTools/Simple-Calculator/issues/139
    4. Fix the bug by creating pull request
    5. His pull request has been recently acccepted by the developer!

    Watch the video recorded by him at https://www.bilibili.com/video/BV1dc411h7VH

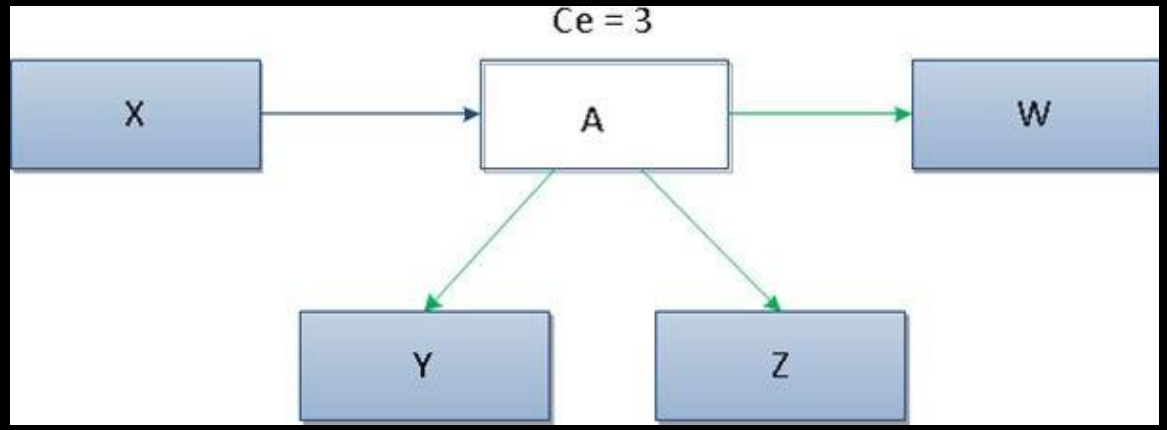
RECAP: MARTIN'S COUPLING METRIC

- Ca : Afferent coupling: the number of classes outside this module that depend on classes inside this module
- Ce : Efferent coupling: the number of classes inside this module that depend on classes outside this module

Instability = Ce / (Ca + Ce)

RECAP: EFFERENT COUPLING (CE)

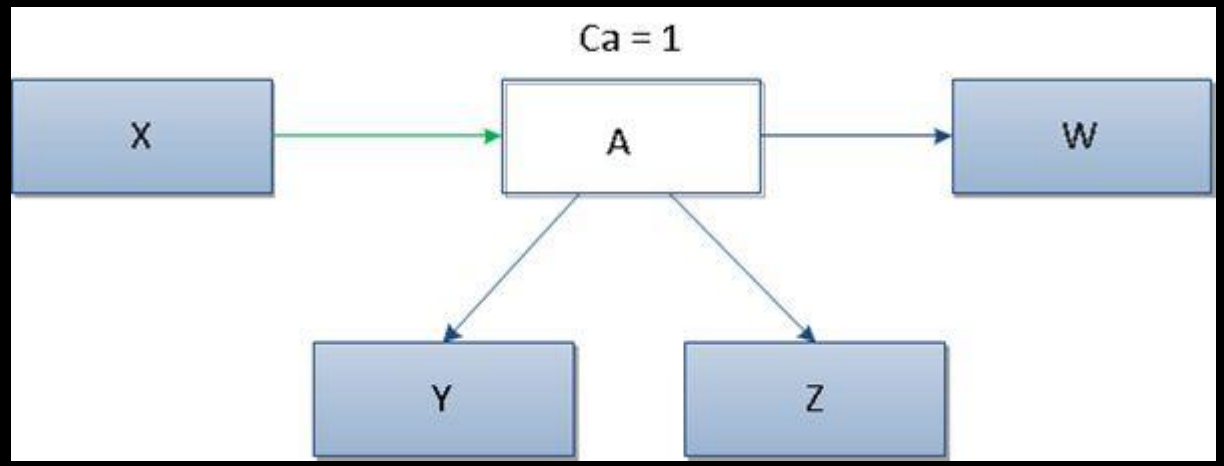
- Definition: A number of classes in a given package, which depends on the classes in other packages
- Measure interrelationships between classes.
- Enable us to measure the vulnerability of the package to changes in packages on which it depends.
- High value of the metric Ce> 20 indicates instability of a package,



Outgoing dependencies

RECAP:AFFERENT COUPLING (CA)

- Measure incoming dependencies.
- Enables us to measure the sensitivity of remaining packages to changes in the analysed package.
- High values of metric Ca usually suggest high component stability.



Incoming dependencies

RECAP: WHY TO REVERSE ENGINEER?

- Replace entire system
- Re-engineer, or modify system
- Use system
- Write documentation for system

- Know your purpose!

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

Each pattern entails some positive and negative tradeoffs.

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

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

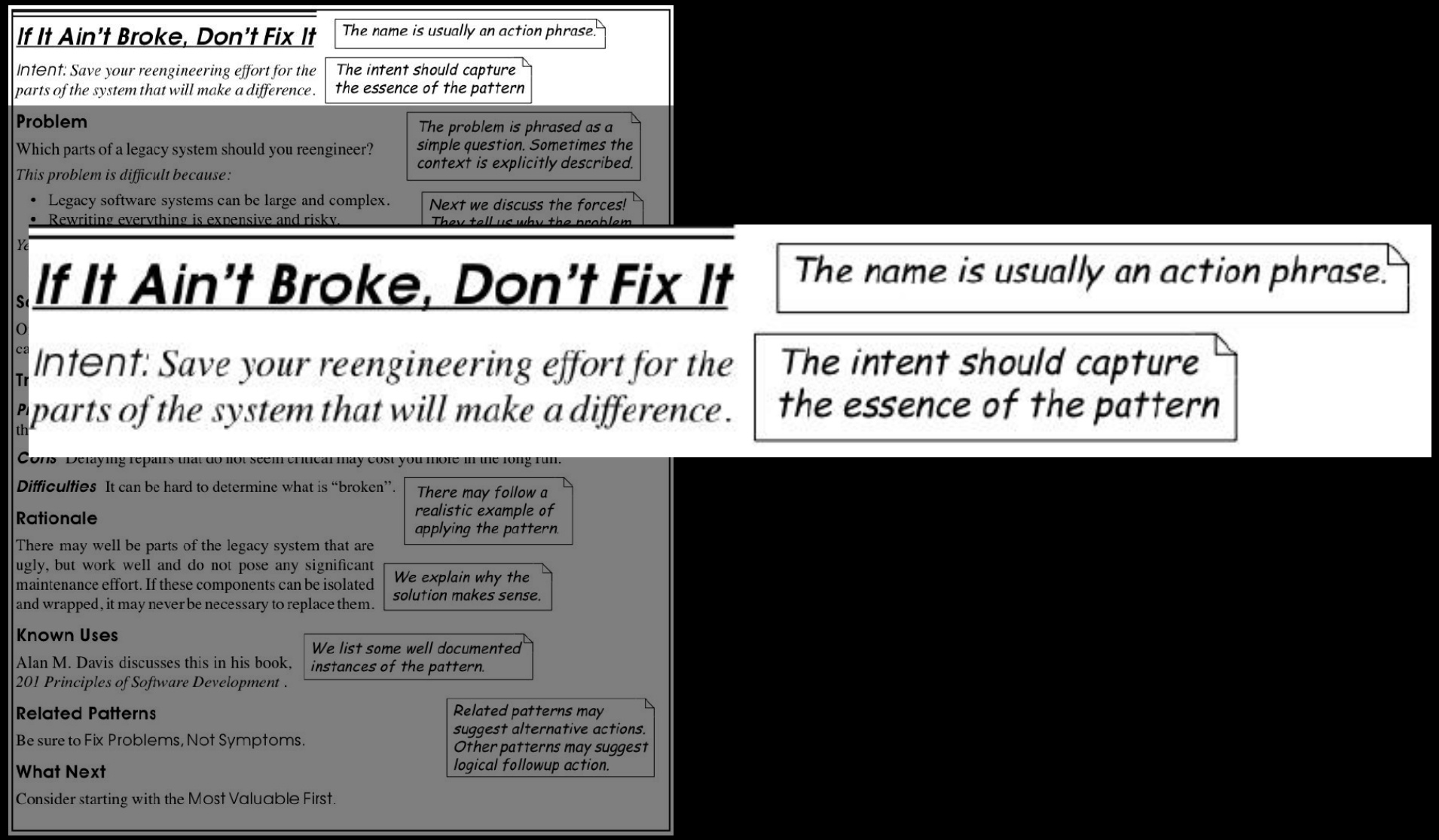
Be sure to Fix Problems, Not Symptoms.

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

What Next

Consider starting with the Most Valuable First.

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?

This problem is difficult because:

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

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

Next we discuss the forces! They tell us why the problem

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

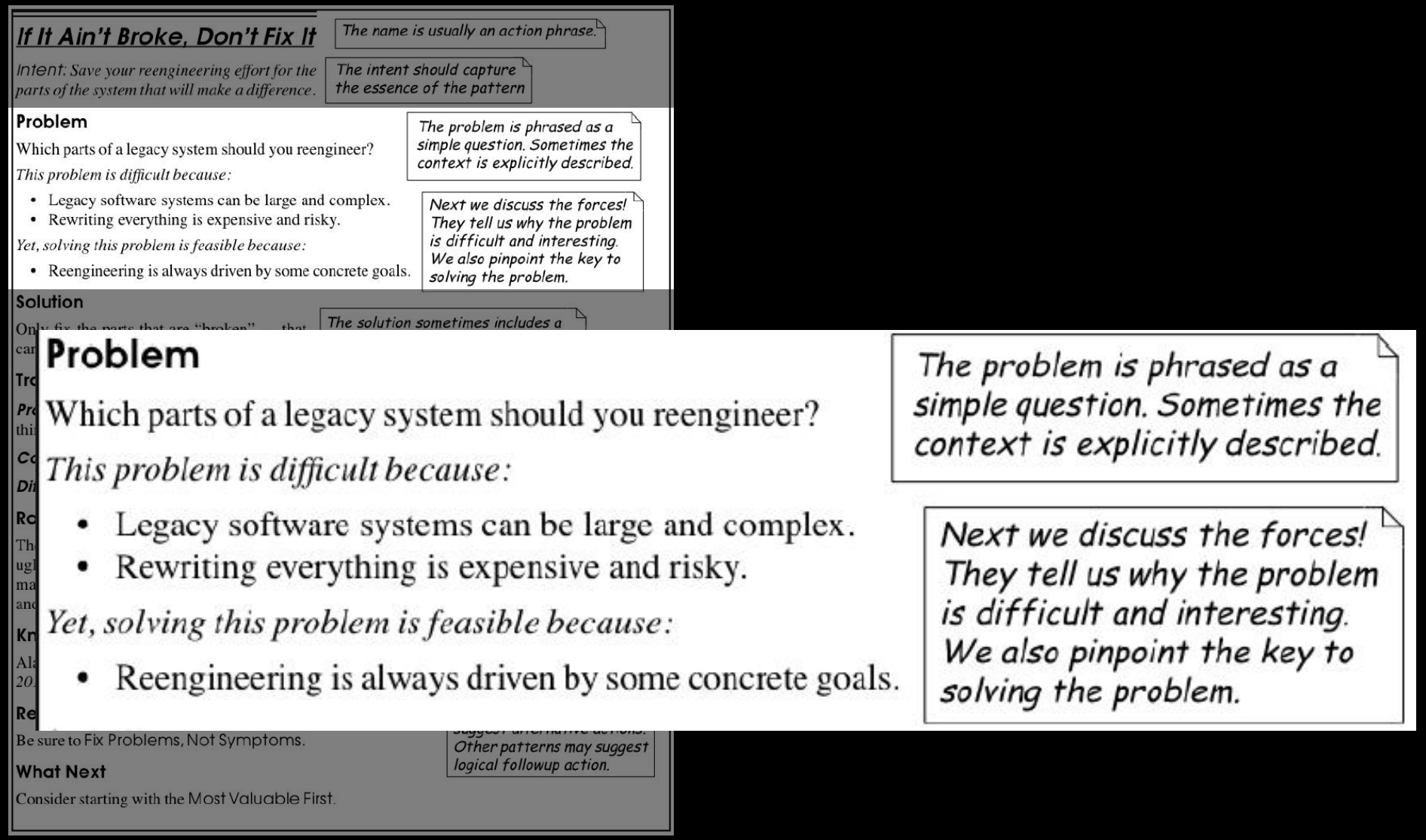
Be sure to Fix Problems, Not Symptoms.

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

What Next

Consider starting with the Most Valuable First.

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?

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.

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

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

The solution sometimes includes a

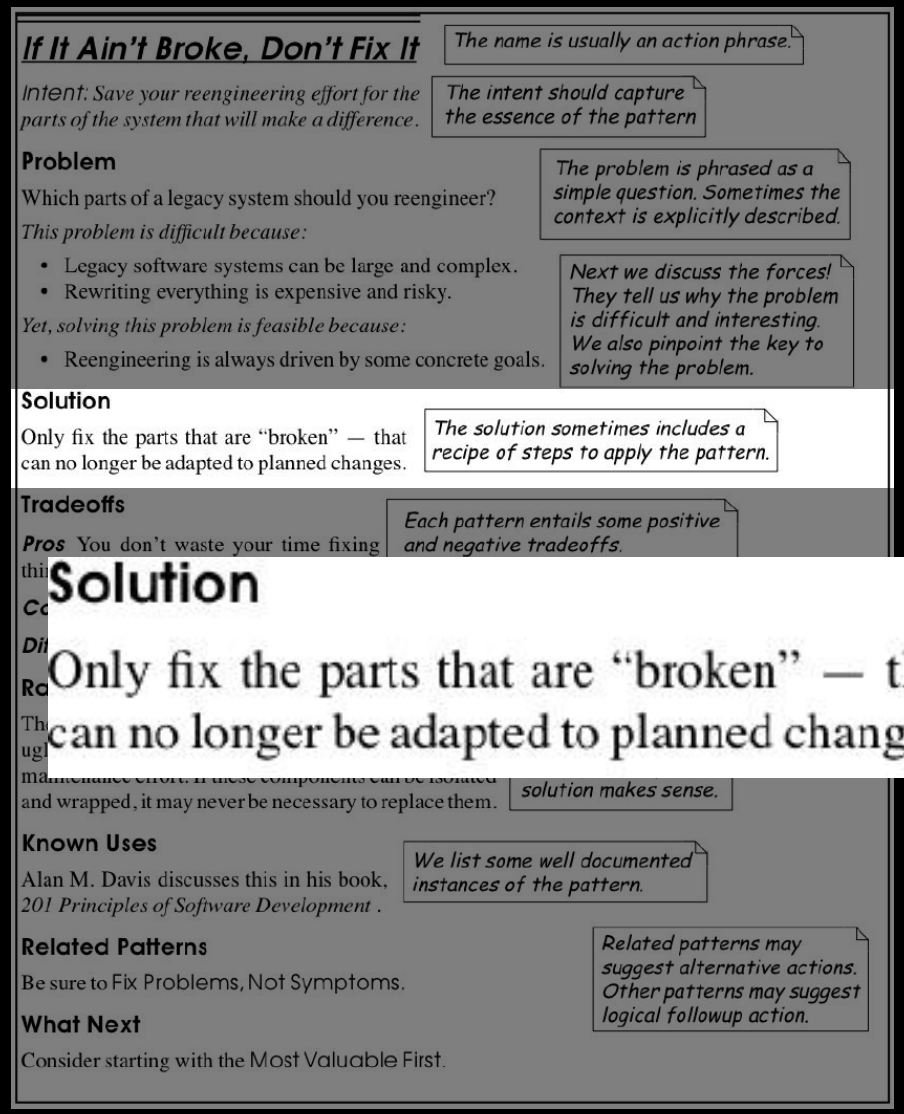
Be sure to Fix Problems, Not Symptoms.

Other patterns may suggest logical followup action.

What Next

Consider starting with the Most Valuable First.

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

Each pattern entails some positive and negative tradeoffs.

Pros You don't waste your time fixing thi

Co

Dit

Ro

Th
ug
ma
and wrapped, it may never be necessary to replace them.

solution makes sense.

Known Uses

We list some well documented instances of the pattern.

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

Related Patterns

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

Be sure to Fix Problems, Not Symptoms.

What Next

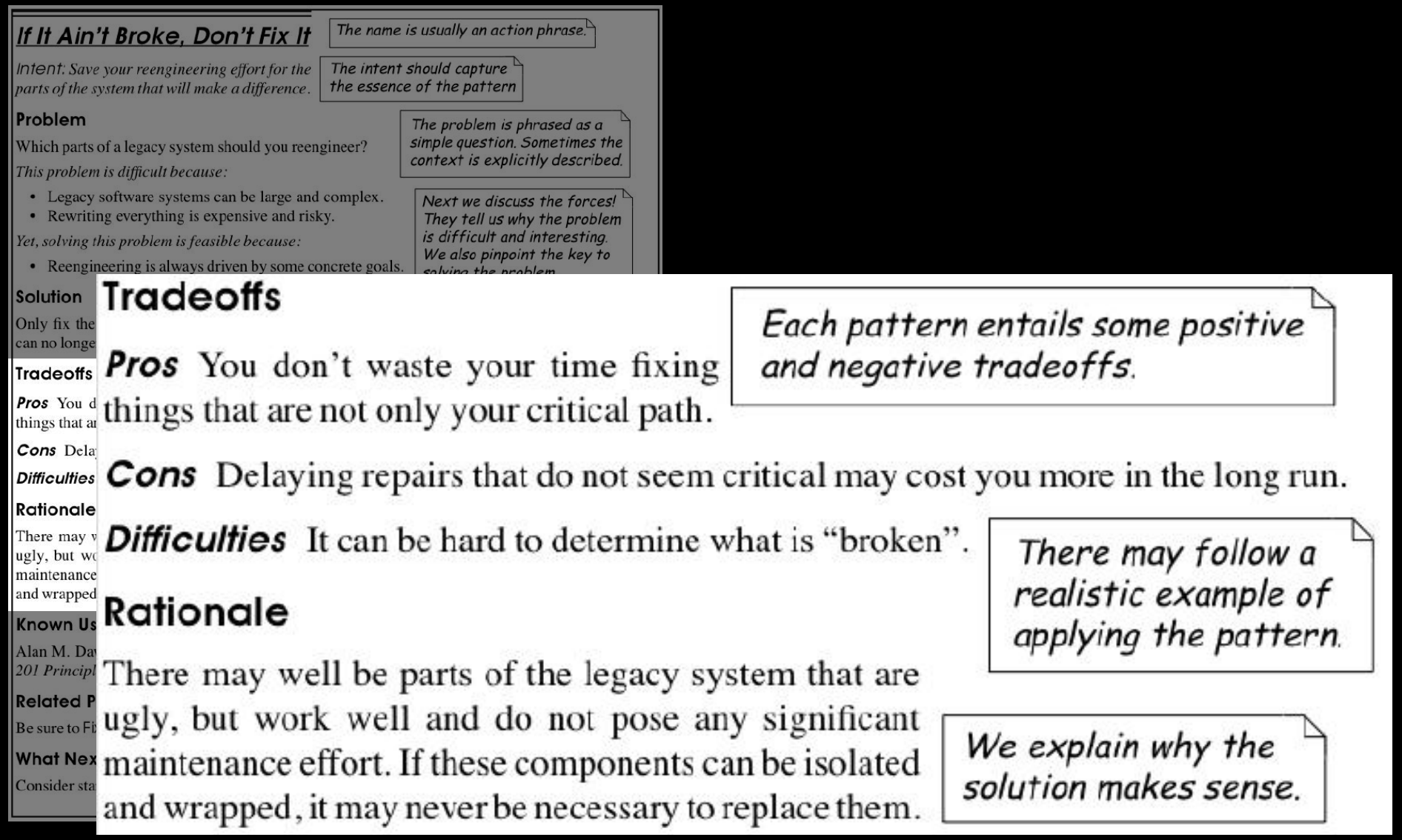
Consider starting with the Most Valuable First.

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.

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?

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.

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

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
can no longe

Tradeoffs

Pros You d
things that ar

Cons Dela

Difficulties

Rationale

There may v
ugly, but wo
maintenance
and wrapped

Known Us

Alan M. Da
201 Princip

Related P

Be sure to Fi

What Nex

Consider sta

Tradeoffs

Each pattern entails some positive and negative tradeoffs.

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

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".

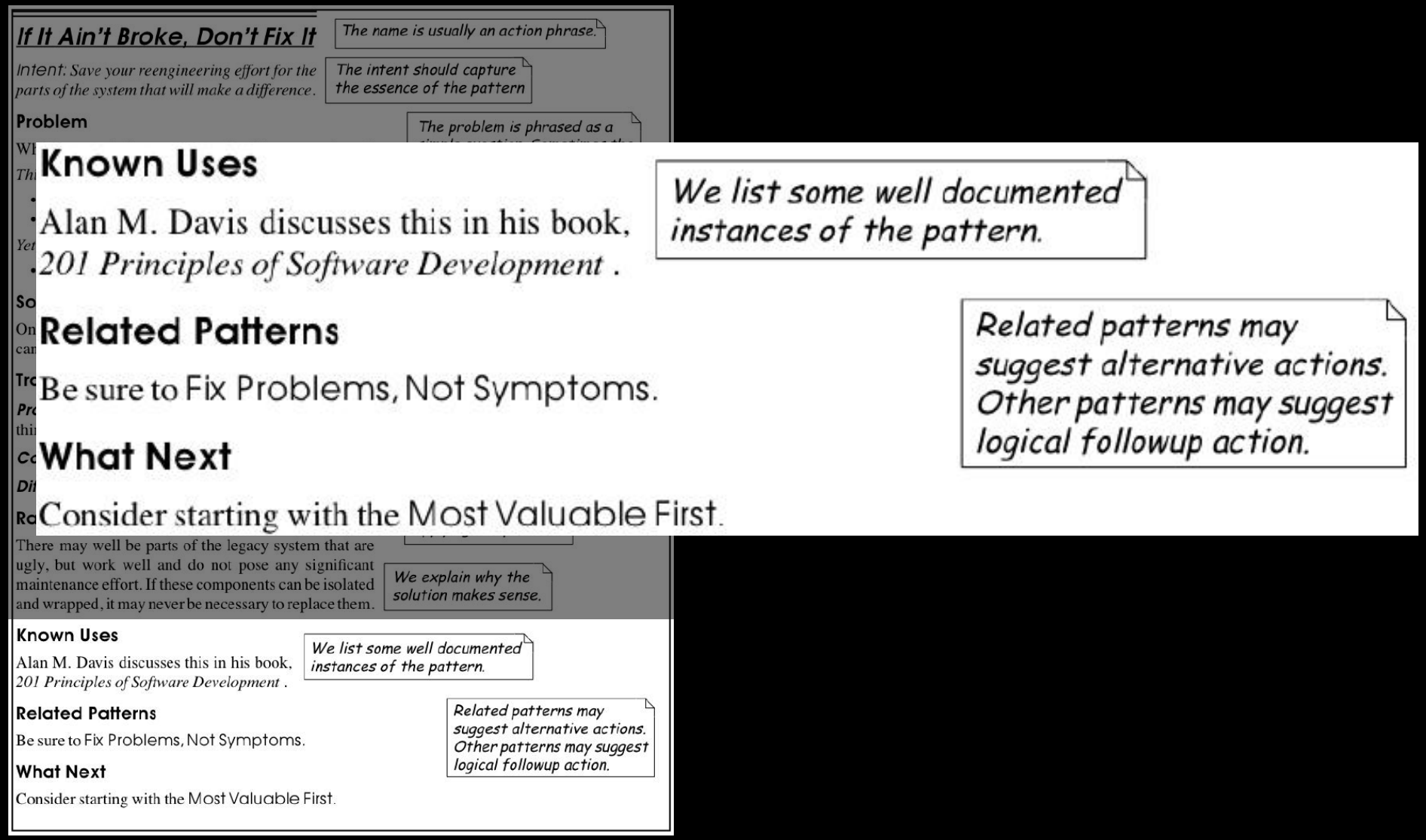
There may follow a realistic example of applying the pattern.

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.

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

The problem is phrased as a

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.

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.

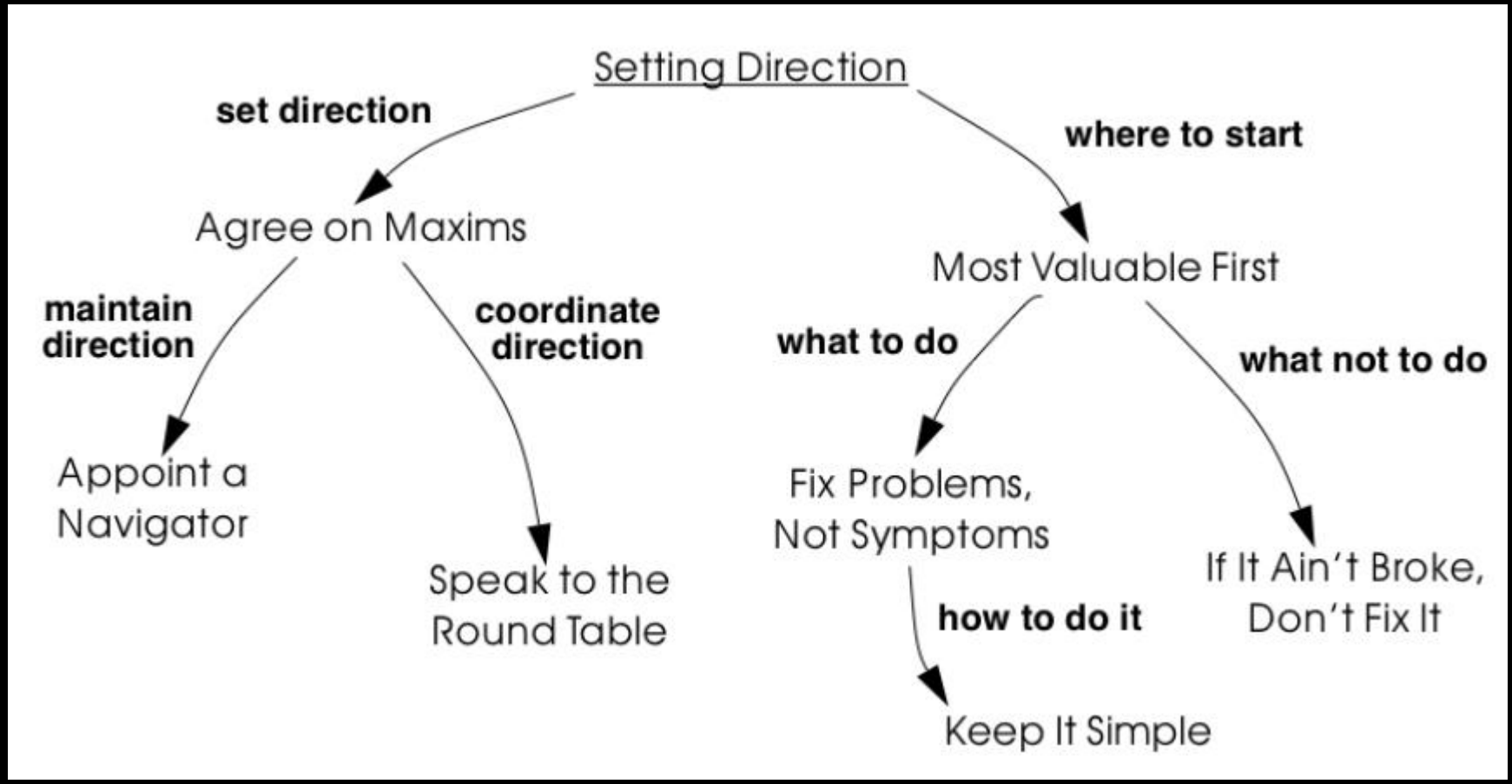
What Next

Consider starting with the Most Valuable First.

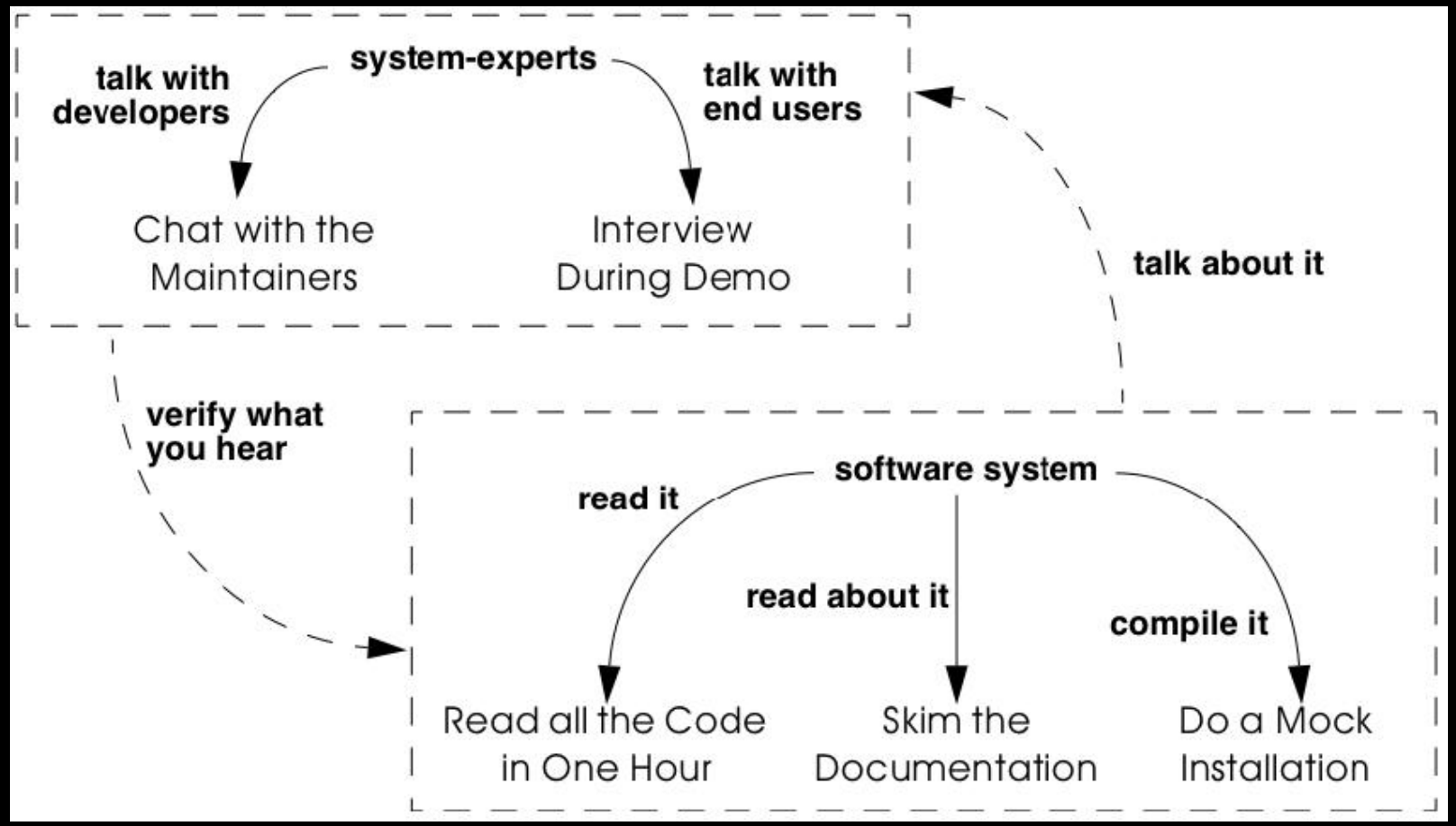
SOME ASPECTS / FORCES

- Limited resources
- Techniques and tools
- Reliable information
- Accurate abstraction
- Skeptical colleagues

SETTING DIRECTION PATTERNS



FIRST CONTACT PATTERNS



CLASS EXERCISE

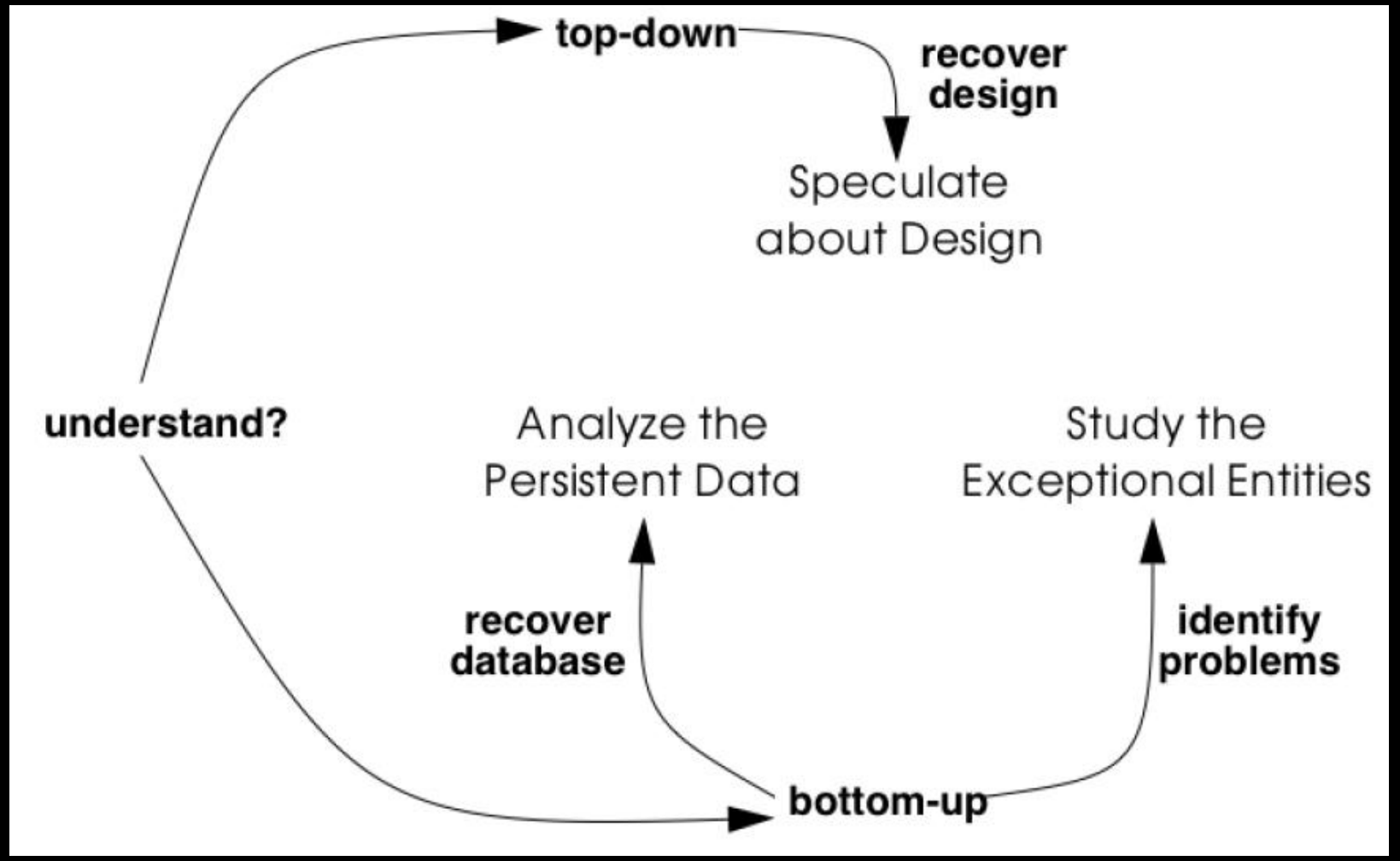
How would you read all code in one hour?

4 Min

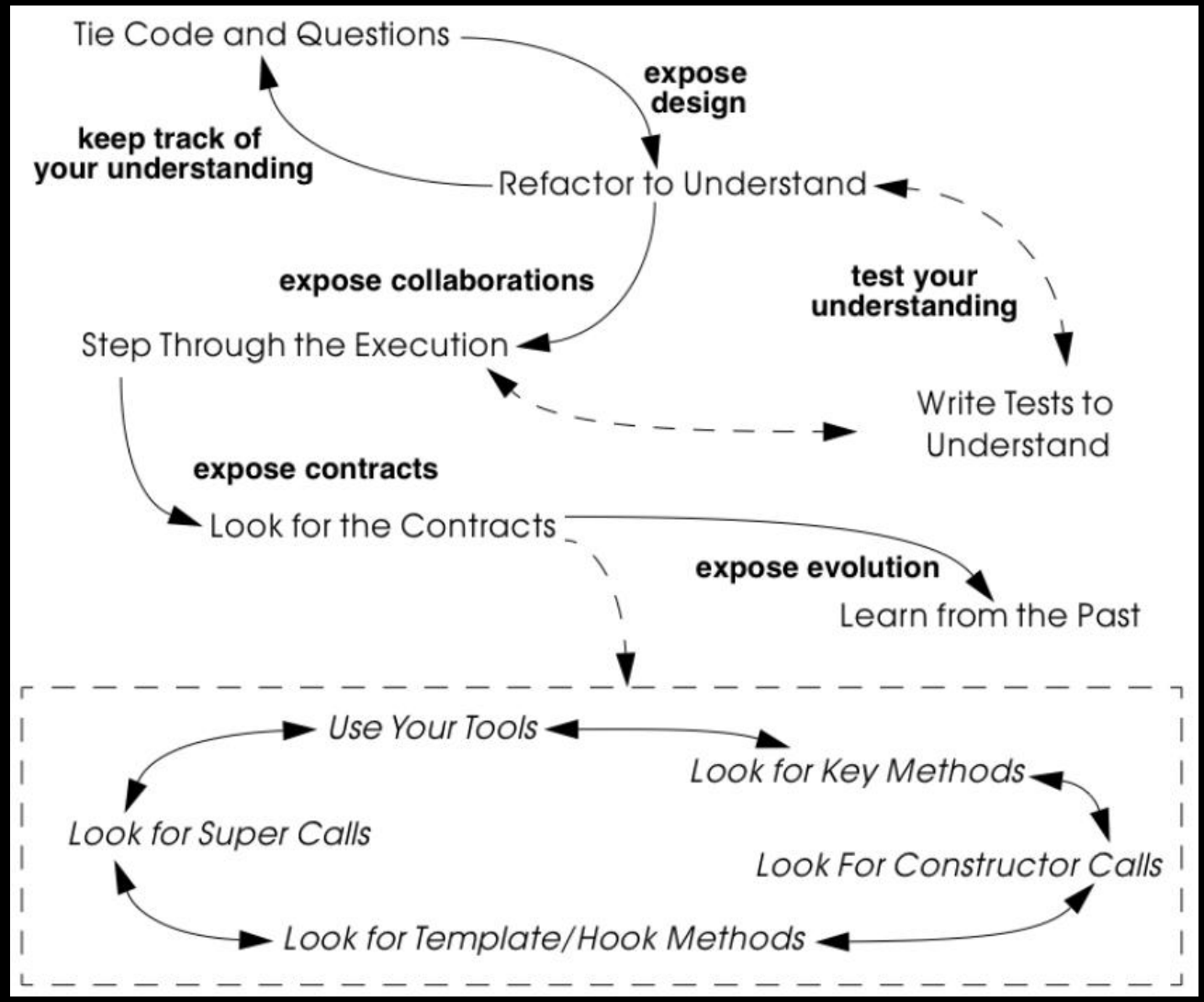
EXAMPLE: READ ALL CODE IN 1 HOUR

- Intent: assess a software system via a brief but intensive code review
- Problem: system is large/varied, unfamiliar
- Solution: read code & document findings (important entities, “code smells”, tests)
- Hints: what to look for?
  - Coding styles/idioms, tests (system and unit)
  - Abstract classes or classes high in hierarchy
  - Large entities, comments

PATTERNS FOR INITIAL UNDERSTANDING



PATTERNS FOR DETAILED MODEL CAPTURE



REVERSE ENGINEERING ACTIVITIES

- Purpose
  - Learn the system from the system
- Activities
  - Read documentation
  - Talk to people
  - Look at the code
  - Work with the system
  - Write documentation

SOME ACTIVITIES

- Read documents
  - Might not be accurate
  - Usually not enough
  - Often missing
- Talk to people
  - What does it do? Show me!
  - What does this mean?
  - Who uses it? How?
  - What is the input and output?

MORE ACTIVITIES

- Look at the code
  - Find major modules and draw a picture of them
  - Spend no more than a few hours
  - Run a test case
- Make a change
  - Fix a simple problem
  - Spend no more than a few hours

DRAW A PICTURE

- Need big picture
  - Class hierarchy - class diagram
  - Modules
- Tools
  - Diagramming tools
  - Code browsers
  - Debuggers

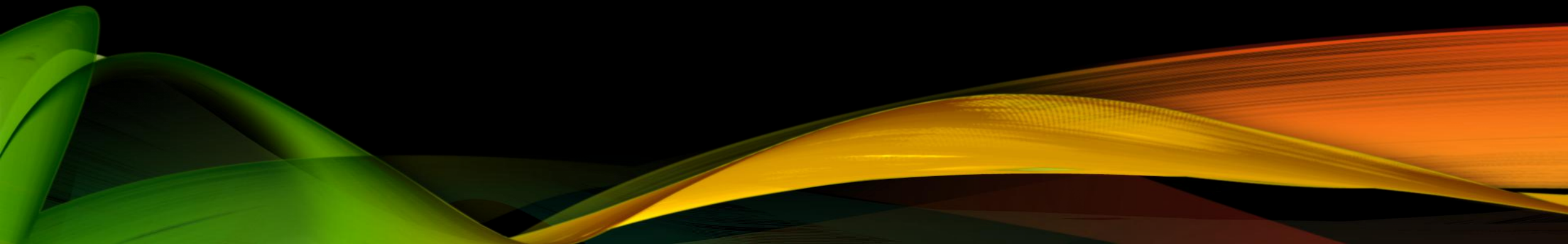
DIAGRAMMING TOOLS

- Java → UML class diagram
- C → call graph
- Smalltalk → collaboration diagram

- Language specific

LEARN FROM DESIGN OF OTHER APP

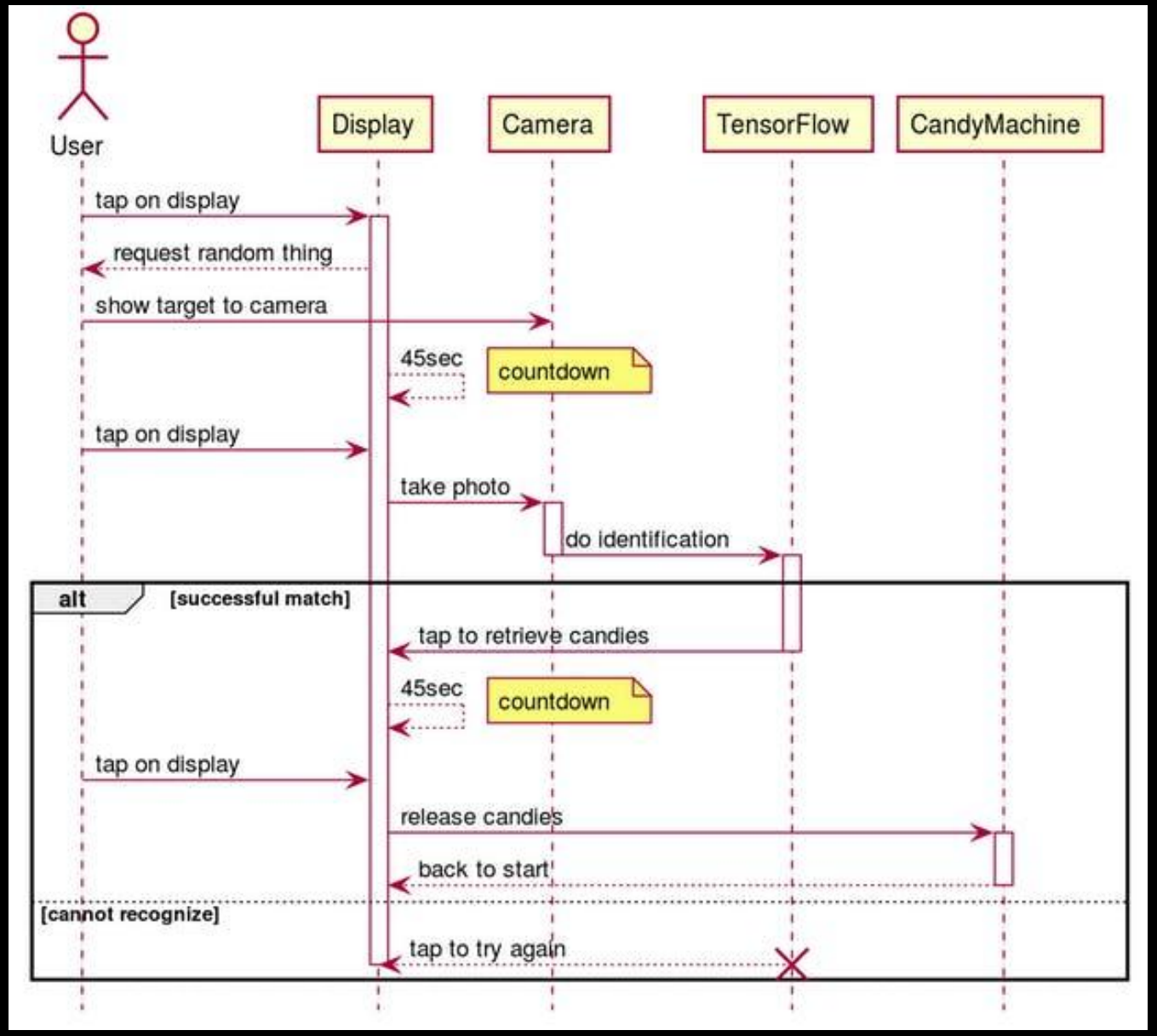
TO GET INSPIRATION FOR YOUR PROJECT



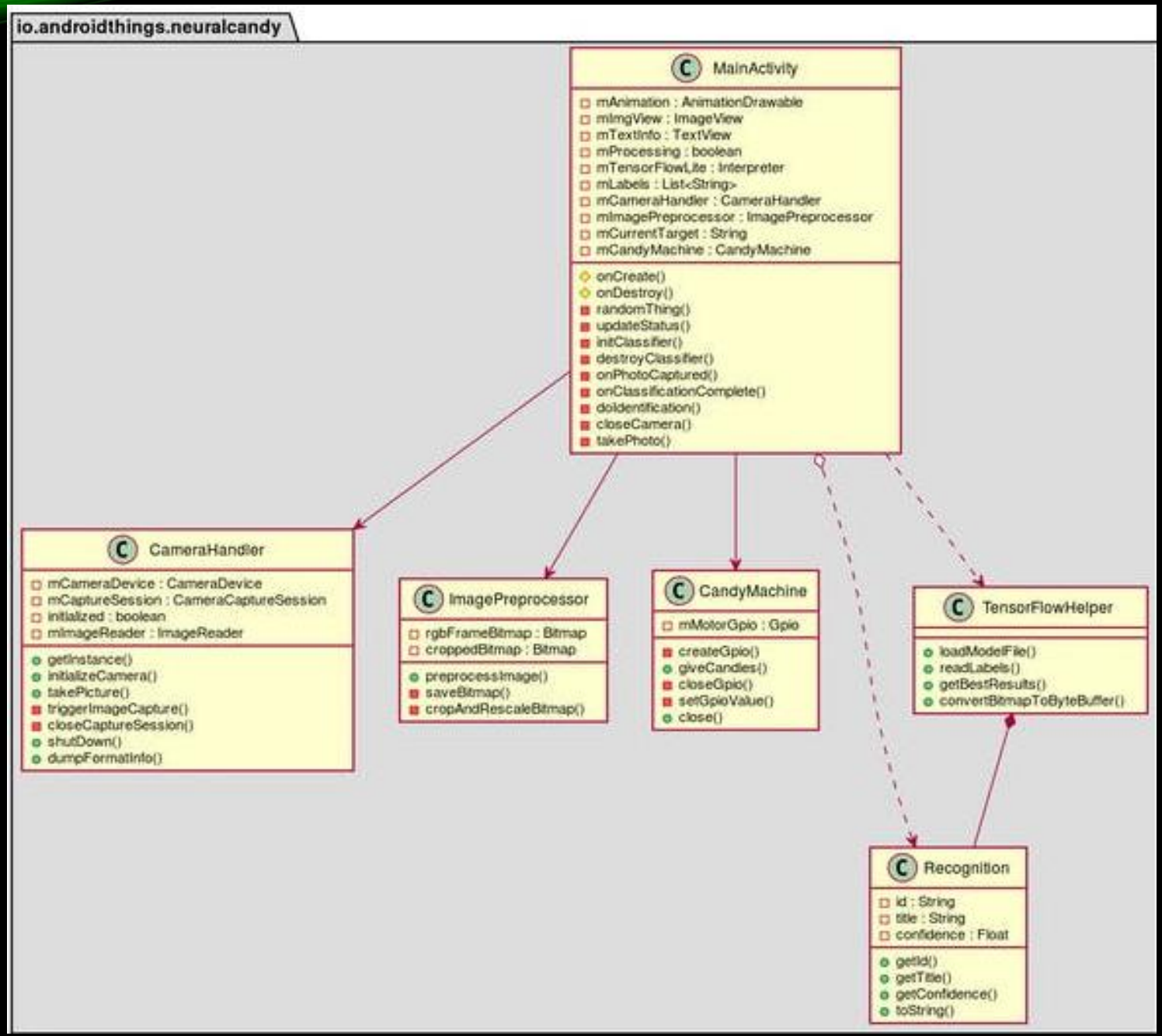
EXAMPLE：NEURAL CANDYAPP

From：https://www.hackster.io/abencomo/neuralcandy-67d2d5

SEQUENCE DIAGRAM

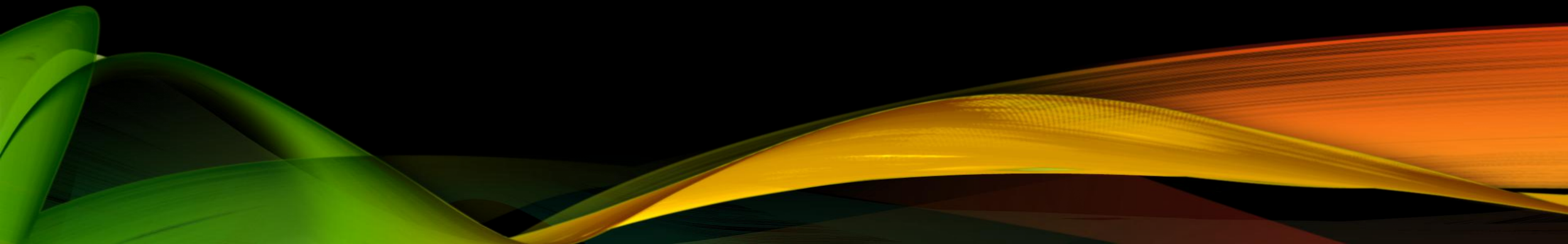


CLASS DIAGRAM



WHAT COULD YOU LEARN FROM RELATED APP?

- NEED TO DRAW SEQUENCE DIAGRAM & CLASS DIAGRAM FOR YOUR PROJECT!
- CAN USE RELATED APP TO GUIDE YOUR DESIGN!



WHAT IS REALLY IMPORTANT?

The big picture – the layers

The problem modules

- The ones that change
- The ones with bugs

In your project: the ones you work on

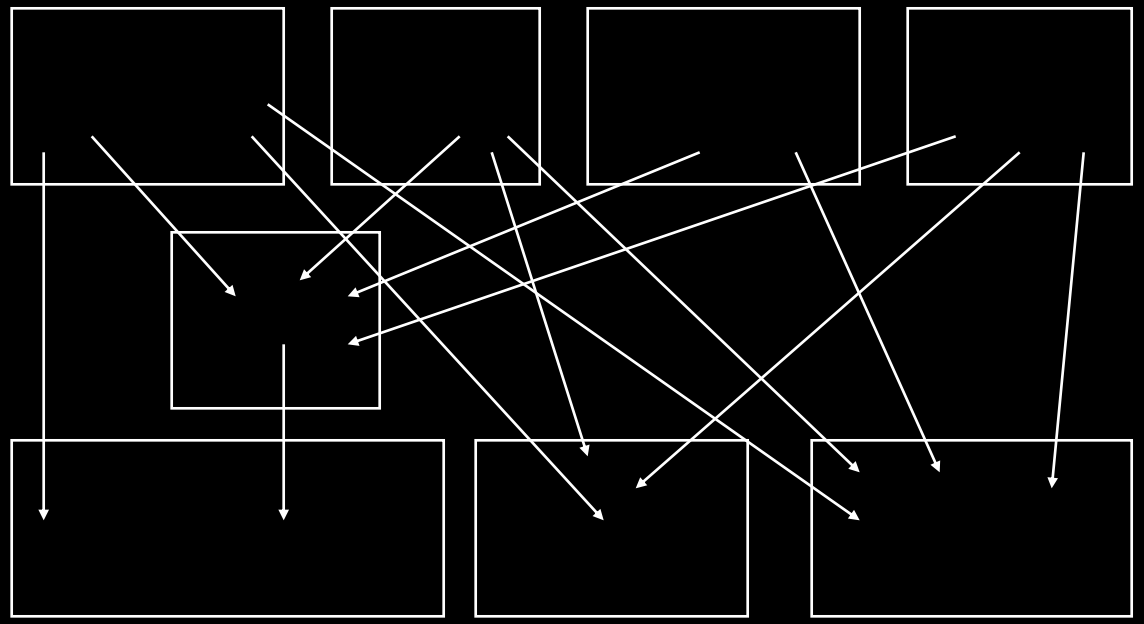
METRICS FOR REVERSE ENGINEERING

- Coupling
  - Top layers depend on lower layers
  - Lower layers do not depend on anything
- Frequent changes
  - % of git changes that affect a module (class)
- Bugs
  - Code coverage

COVERAGE METRICS FOR FINDING BUGS

- Which modules are executed by this test?
- Which modules are executed by this test and not by other tests?
- Given a set of tests, coverage tool reports the functions/lines/paths that are executed
- Infrastructure modules executed by all tests
- “Feature” modules executed by few tests

LAYERS AND COVERAGE



BROWSERS

- Tool for reading code
- Cross reference
  - Definition of variable, procedure, class
  - Uses of variable, procedure, class
- Interactive, multiple views

DEBUGGERS

- See how code executes
- Singlestep
- Breakpoints, watchpoints

- Language specific

DEALING WITH SIZE

- Get an overview
- Decompose
- Examine each part individually
- Experiment to test knowledge
- Correct overview

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

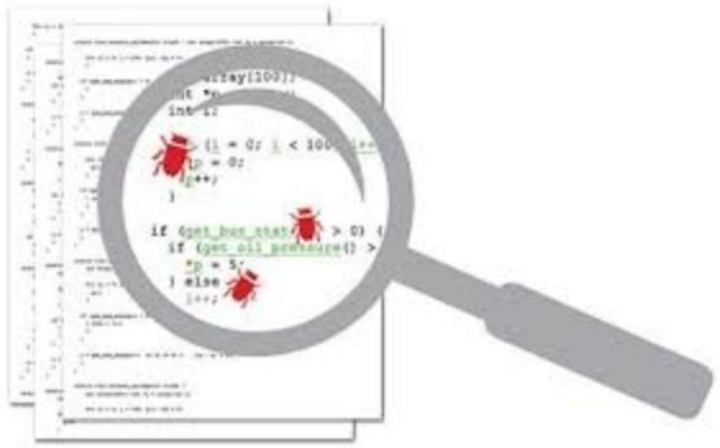
From: https://www.youtube.com/watch?v=zxr_j5XkTT8

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/

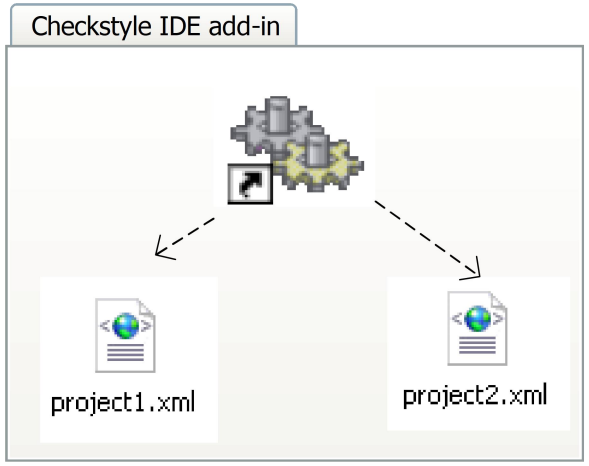
From: https://hienlth.info/java/Ebooks/ThamKhao/BTJB_Lesson05.21_Java%20Code%20Review%20with%20CheckStyle.ppt

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

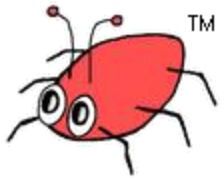
Adapted from swtv.kaist.ac.kr/courses/cs492-spring-13/lec9-findbugs.ppt

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./