

# cs304

# Software Engineering

**TAN, Shin Hwei**

陈馨慧

Southern University of Science and Technology

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

## No class for Week 8

- No lecture will be held on Week 8 due to the 清明holiday.
- There will also be no lab for the labs on Monday (5 April)
- Other lab sessions (not on 5 April) will use the uploaded slides for revision. It contains reminder for previous lab assignments, and project.

# Administrative Info

- **The deadline for MP0 and Project Proposal has passed.** Late submission get 0 score!
- **MP1 part 1 and part 2 have two different deadlines**
  - Part 2 due on April 14
- **Progress Report has been posted due on April 23**
- **All assignments should be written in English**
  - One exception: The selected issues could be written by the developers in Chinese
- **All lab exercise should be submitted before next lab to avoid accumulating too much assignments**
- **Attend lab today for tools-supported fixing session! Important for MP1!**
  - What if you can run some tools to help you fix the bugs that you have selected?

# RECAP: CYCLOMATIC COMPLEXITY (1)

- A measure of logical complexity
- Tells how many tests are needed to execute every statement of program

=Number of branches (*if, while, for*) + 1

# DIFFERENT TOOLS TELLS YOU DIFFERENT VALUES OF CYCLOMATIC COMPLEXITY

- Original paper is not clear about how to derive the control flow graph
  - different implementations gives different values for the same code.
  - For example, the following code is reported with complexity 2 by the [Eclipse Metrics Plugin](#), with 4 by [GMetrics](#), and with complexity 5 by [SonarQube](#):

```
int foo (int a, int b) {  
    if (a > 17 && b < 42 && a+b < 55) {  
        return 1;  
    }  
    return 2;  
}
```

# COUPLING AND COHESION (1)

- Coupling - dependences among modules
- Cohesion - dependences within modules
- Dependences
  - Call methods, refer to class, share variable
- Coupling - bad
- Cohesion - good

# COUPLING AND COHESION (2)

- Number and complexity of shared variables
  - Functions in a module should share variables
  - Functions in different modules should not
- Number and complexity of parameters
- Number of functions/modules that are called
- Number of functions/modules that call me



# COUPLING AND COHESION IN OO LANGUAGES



# DHAMA'S COUPLING METRIC

Module coupling =  $1 / ($   
number of input parameters +  
number of output parameters +  
number of global variables used +  
number of modules called +  
number of modules calling  
 $)$

0.5 is low coupling, 0.001 is high coupling

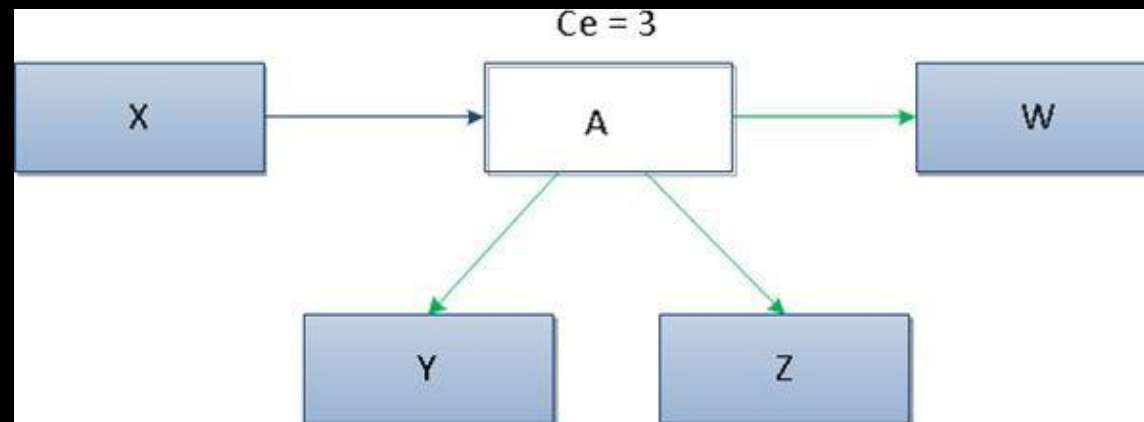
# MARTIN'S COUPLING METRIC

- **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

$$\text{Instability} = Ce / (Ca + Ce)$$

# EFFERENT COUPLING (CE)

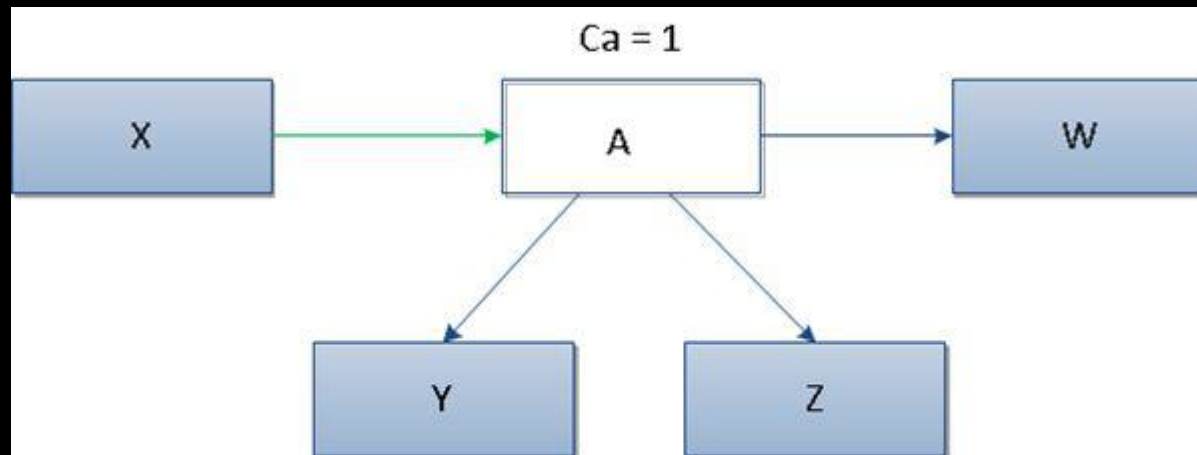
- **Definition:** A number of classes in a given package, which depends on the classes in other packages
- Measure interrelationships between classes.
- Enable us to measure the vulnerability of the package to changes in packages on which it depends.
- **High value** of the metric  $C_e > 20$  indicates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



Outgoing dependencies

# 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**.



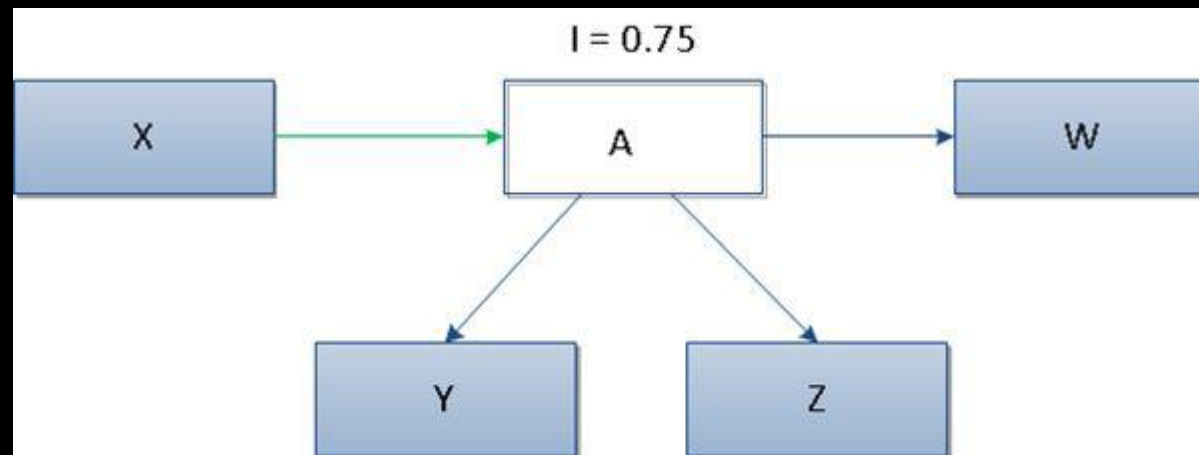
**Incoming dependencies**

$$I = \frac{Ce}{Ce + Ca}$$

# INSTABILITY

There are two types of components:

- Many outgoing dependencies and not many of incoming ones (value  $I$  is close to 1), are unstable due to the possibility of easy changes to these packages;
- Many incoming dependencies and not many of outgoing ones (value  $I$  is close to 0), therefore they are more difficult in modifying due to their greater responsibility.



# TECHNICAL OO METRICS

- Paper (not required for the exam, but you can read it if you are interested in this topic and the slides are not sufficient)
- A Metrics Suite for Object Oriented Design, Shyam R. Chidamber and Chris F. Kemerer IEEE Transactions on Software Engineering, June 1994, pp 476-493



# LIST OF METRICS

- Weighted Methods Per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling between Object Classes (CBO)
- Response for a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

# WEIGHTED METHODS PER CLASS

- WMC for a class is the sum of the complexities of the methods in the class
- Possible method complexities
  - 1 (number of methods)
  - Lines of code
  - Number of method calls
  - Cyclomatic complexity



# WEIGHTED METHODS PER CLASS

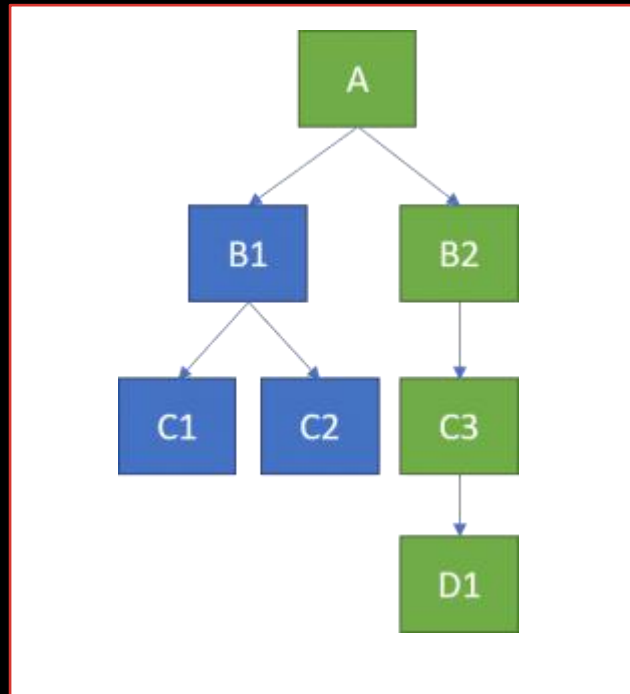
- The number of methods and the complexity of methods predict the time and effort required to develop and maintain a class
- The larger the number of methods in a class, the greater the potential impact on children
- Classes with large numbers of methods are more likely to be application specific and less reusable

# WEIGHTED METHODS PER CLASS

- WMC for a class is the sum of the complexities of the methods in the class
- Possible method complexities
  - 1 (number of methods)
  - Lines of code
  - Number of method calls
  - Cyclomatic complexity

# DEPTH OF INHERITANCE TREE

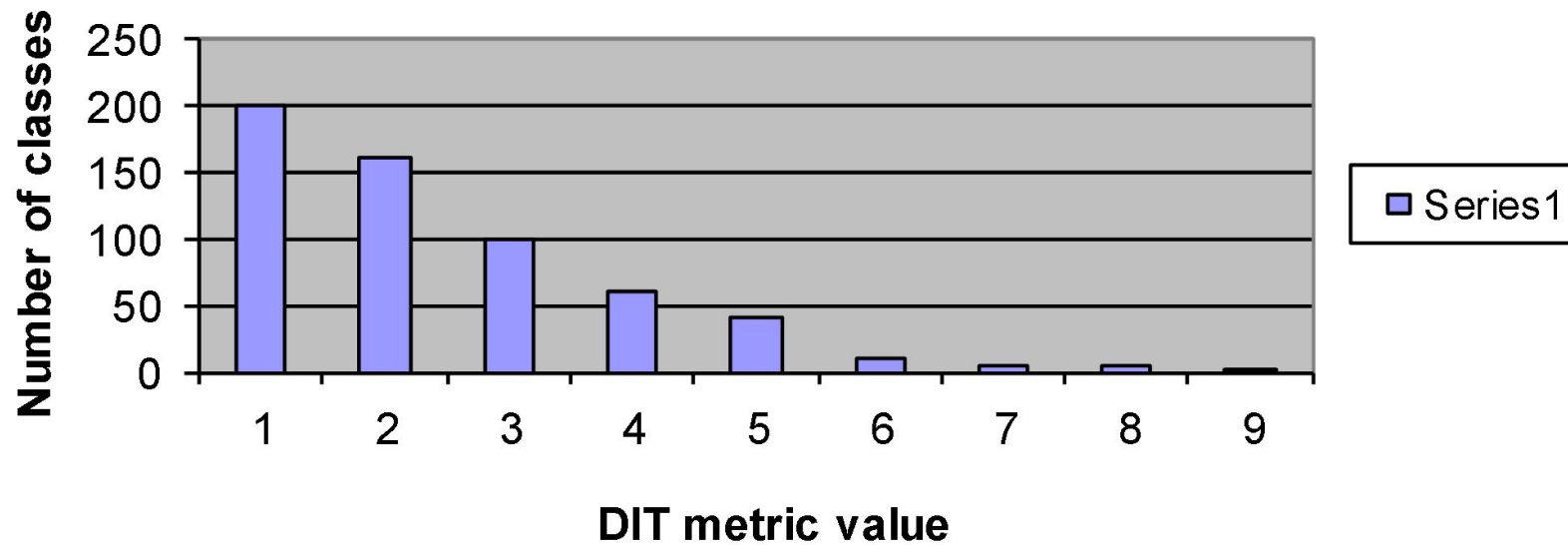
- Maximum length from a class to the root of the tree
- What is the depth of inheritance tree below?
  - 1
  - 2
  - 3
  - 4



# DEPTH OF INHERITANCE TREE

- The deeper a class is in the hierarchy, the more methods it inherits and so it is harder to predict its behavior
- The deeper a class is in the hierarchy, the more methods it reuses
- Deeper trees are more complex

# DEPTH OF INHERITANCE TREE





# NUMBER OF CHILDREN

- Number of immediate subclasses
- More children is more reuse
- A class might have a lot of children because of misuse of subclassing
- A class with a large number of children is probably very important and needs a lot of testing



# NUMBER OF CHILDREN

- Almost all classes have 0 children
- Only a handful of classes will have more than five children

# COUPLING BETWEEN OBJECT CLASSES

- Number of other classes to which a class is coupled
- Class A is coupled to class B if there is a method in A that invokes a method of B
- Want to be coupled only with abstract classes high in the inheritance hierarchy



# COUPLING BETWEEN OBJECT CLASSES

- Coupling makes designs hard to change
- Coupling makes classes hard to reuse
- Coupling is a measure of how hard a class is to test

# COUPLING BETWEEN OBJECT CLASSES

- C++ project: median 0, max 84
- Smalltalk project: median 9, max 234

# RESPONSE FOR A CLASS

- Number of methods in a class or called by a class
- The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class

# RESPONSE FOR A CLASS

- If a large number of methods can be invoked in response to a message, testing becomes more complicated
- The more methods that can be invoked from a class, the greater the complexity of the class

# RESPONSE FOR A CLASS

- C++: median 6, max 120
- Smalltalk: median 29, max 422

# LACK OF COHESION IN METHODS

- Number of pairs of methods that don't share instance variables minus number of pairs of methods that share instance variables

$$= Num_{don't\ shared-instance} - Num_{shared-instance}$$

- Cohesiveness of methods is a sign of encapsulation
- Lack of cohesion implies classes should be split

# LACK OF COHESION OF METHODS

- C++: median 0, max 200
- Smalltalk: median 2, max 17
- Smalltalk system had only a few zero

# ONE WAY TO USE METRICS

- Measure the amount of code produced each month by each programmer
- Give high producers big raise
- Is this good or bad?



# ANOTHER WAY TO USE METRICS

- Measure complexity of modules
- Pick the most complex and rewrite it
- Is this good or bad?

# YET ANOTHER WAY TO USE METRICS

- Use function points to determine how many lines of code a system will require
- When you write that many lines of code, stop and deliver the system
- Is this **good** or **bad**?

# YET^2 ANOTHER WAY TO USE METRICS

- Track progress
  - Manager review
  - Information radiator
- Code reviews
- Planning
- Is this good or bad?

# MANAGER REVIEW

- Manager periodically makes a report
- Growth in SLOC
- Bugs reported and fixed
- SLOC per function point (user story), SLOC per programmer (user stories per programmer)

# INFORMATION RADIATOR

- Way of letting entire group know the state of the project
  - **Green/red** test runner (green/red lava lamp or green/red traffic light)
  - Wall chart of predicted/actual stories
  - Wall chart of predicted/actual SLOC
  - Web page showing daily change in metrics (computed by daily build)

# CODE REVIEWS

- Look at metrics before a code review
  - Coverage - untested code usually has bugs
  - Large methods/classes
  - Code with many reported bugs

# PLANNING

- Need to predict amount of effort
  - Make sure you want to do it for your project
  - Hit delivery dates
  - Hire enough people
- Predict components and SLOC
- Predict stories & months (XP)
- Opinion of developers/tech lead/manager

# SUMMARY: METRICS (IN GENERAL)

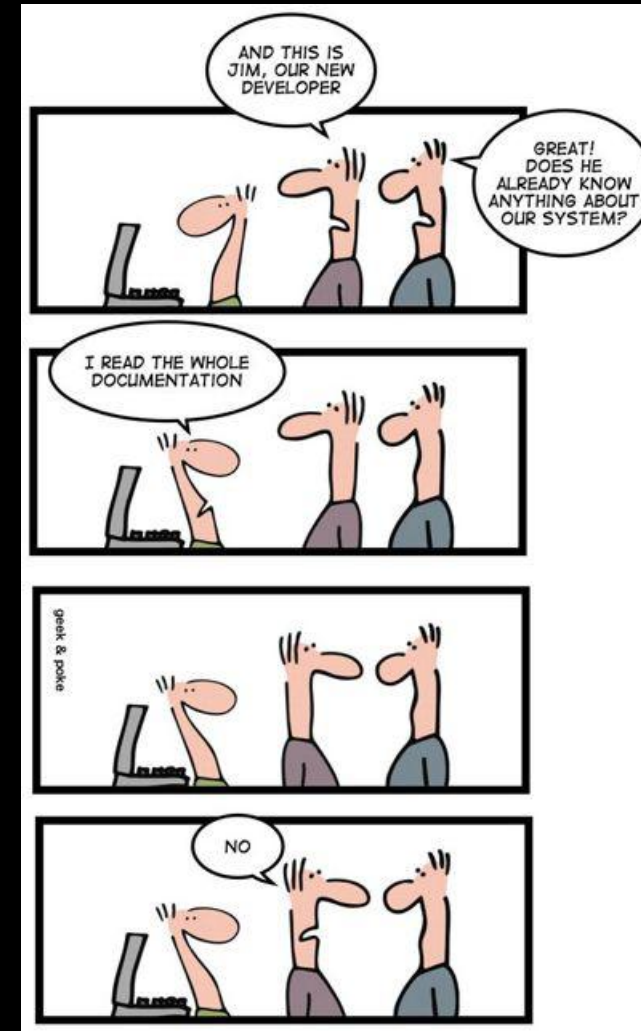
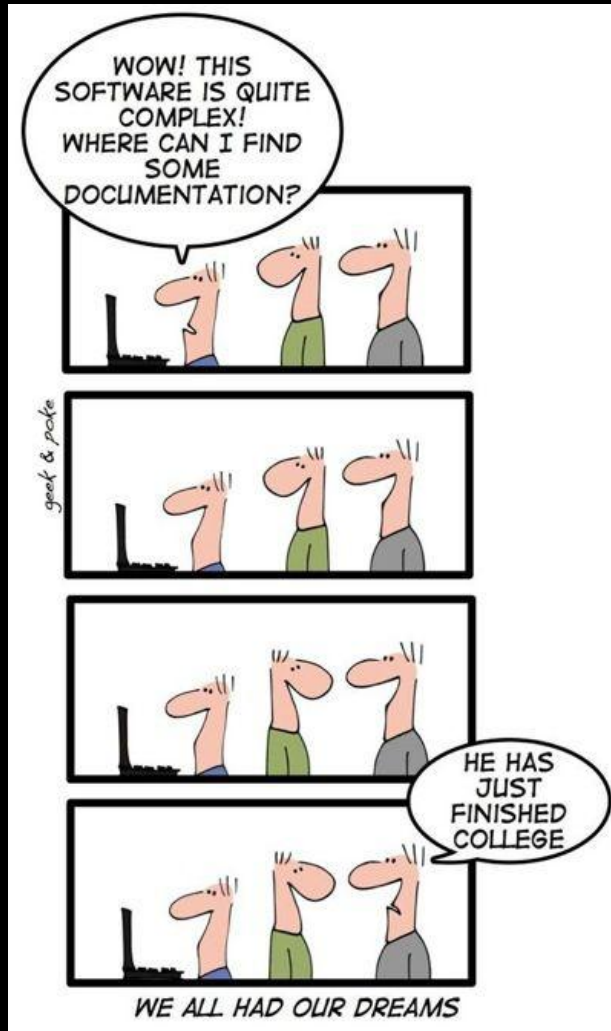
- Non-technical: about process
- Technical: about product
  - Size, complexity (cyclomatic, function points)
- How to use metrics
  - Prioritize work - compare modules in a system version, compare system versions over time
  - Measure programmer productivity
  - Make plans based on predicted effort



The background features a black field with dynamic, flowing waves of green and orange. The green waves are prominent on the left side, while the orange waves flow from the top right towards the bottom right. The waves have a soft, ethereal quality with some internal texture visible.

# REVERSE ENGINEERING

# IDEALLY, SOFTWARE SHOULD BE WELL-DOCUMENTED, BUT ...



# TODAY'S GOALS

- How to learn an existing software system that is **poorly documented**?
- What are some object-oriented reengineering patterns?
- Useful for your project, to navigate through thousands of Java LOC

# FROM THE READING

- Foreword by Martin Fowler:

“For a long time it’s puzzled me that most books on software development processes talk about what to do when you are starting from a blank sheet of editor screen. It’s puzzled me because that’s not the most common situation that people write code in. Most people have to make changes to an existing code base, even if it’s their own. In an ideal world this code base is well designed and well factored, but we all know how often the ideal world appears in our career.”

# FROM THE READING

- Reverse engineering definition by Chkofsky & Cross:

“Reverse Engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.”

# WHAT IS REVERSE ENGINEERING?

- Discovering design of an artifact
  - From lower level to higher level; for example:
    - Given binary, discover source code
    - Given code, discover specification & design rationale
- Layman: trying to understand how the system works
- When are you done?
  - Learn enough to:
    - Change it, or
    - Replace it, or
    - Write a book about it

# WHY TO REVERSE ENGINEER?

- Replace entire system
  - Re-engineer, or modify system
  - Use system
  - Write documentation for system
- 
- Know your purpose!

# WHEN TO REVERSE ENGINEER?

- In the large:
  - Big changes: Y2K (99 → 00 → 01)
  - Technology change: desktop → web → mobile
- In the small
  - Add a feature, fix a bug, improve performance
- Your project:
  - Understand and modify parts of some large codebase



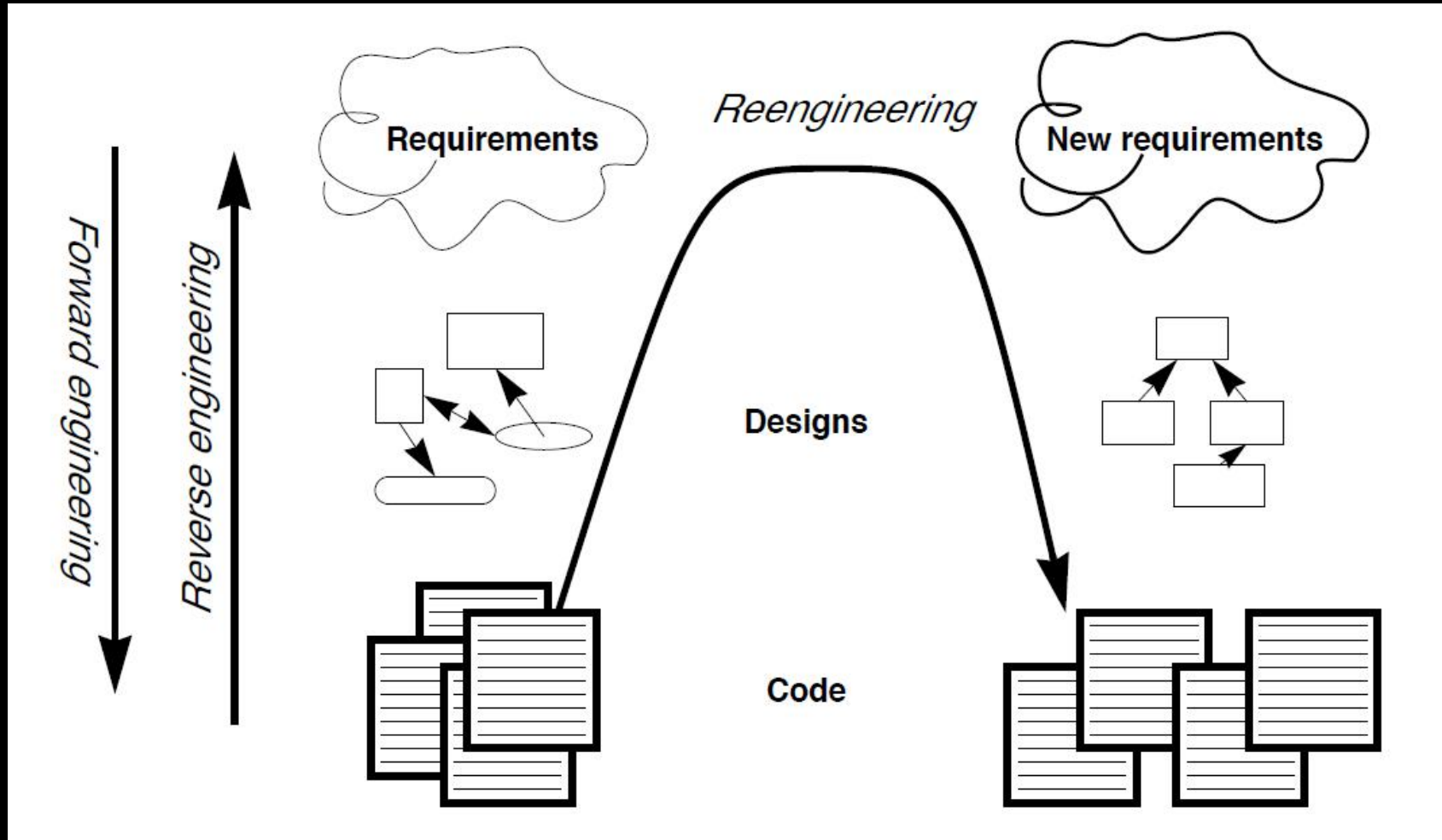
# HOW TO REVERSE ENGINEER?

- Make use of various/any sources, to incrementally build an explicit or a conceptual model of software. E.g.:
  - Read existing documentation
  - Read source code
  - Run the software
  - Interview users and developers
  - Execute existing or write new test cases
  - Generate and analyze traces
  - Use tools to generate high-level views of code / traces
  - Analyze the version history
  - Etc., basically you can use anything that helps

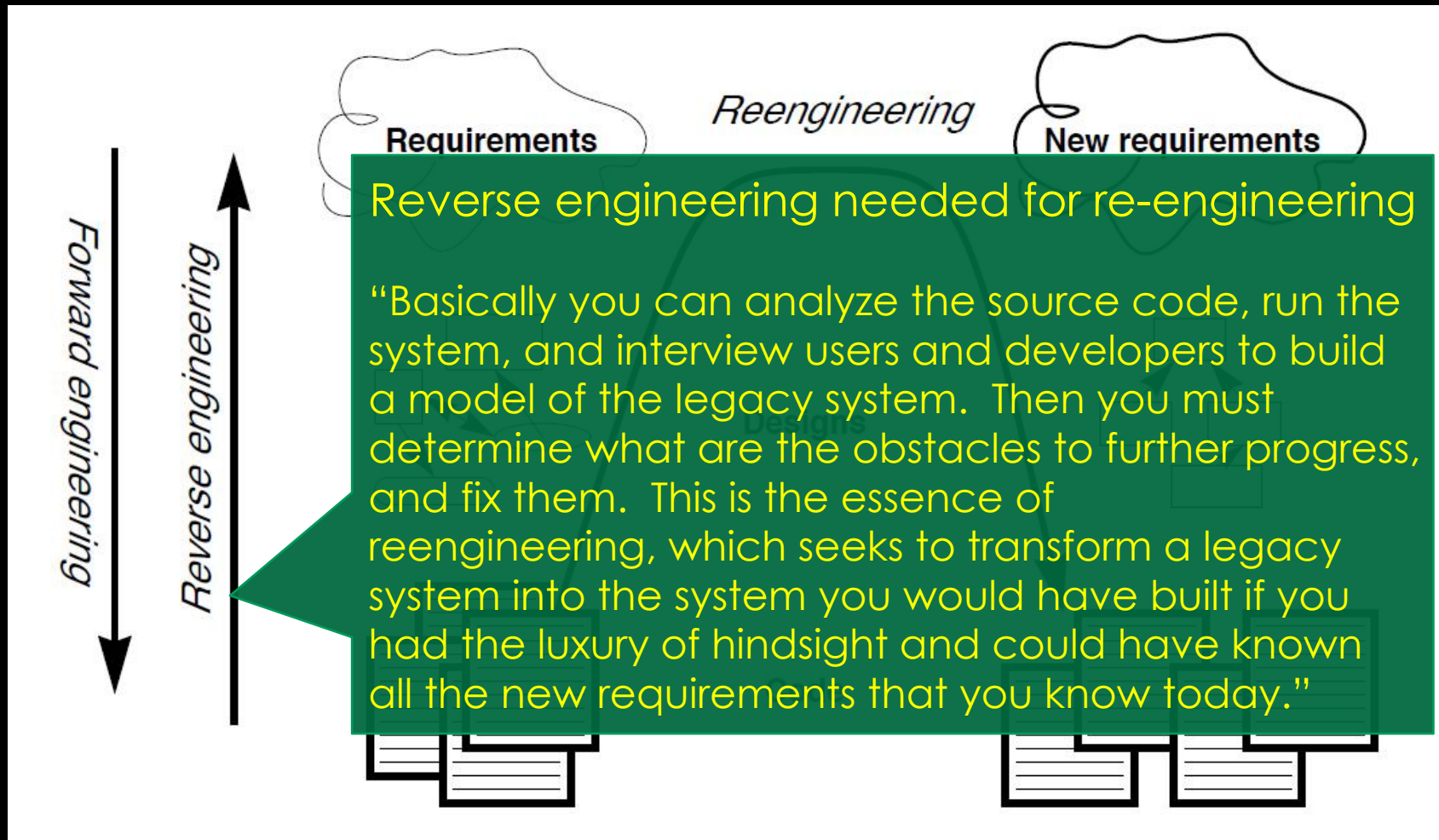
# SOME TERMINOLOGY

- Forward engineering
  - From requirements to design to code
- Reverse engineering
  - From code to design, maybe to requirements
- Reengineering
  - From old code to new code via some design

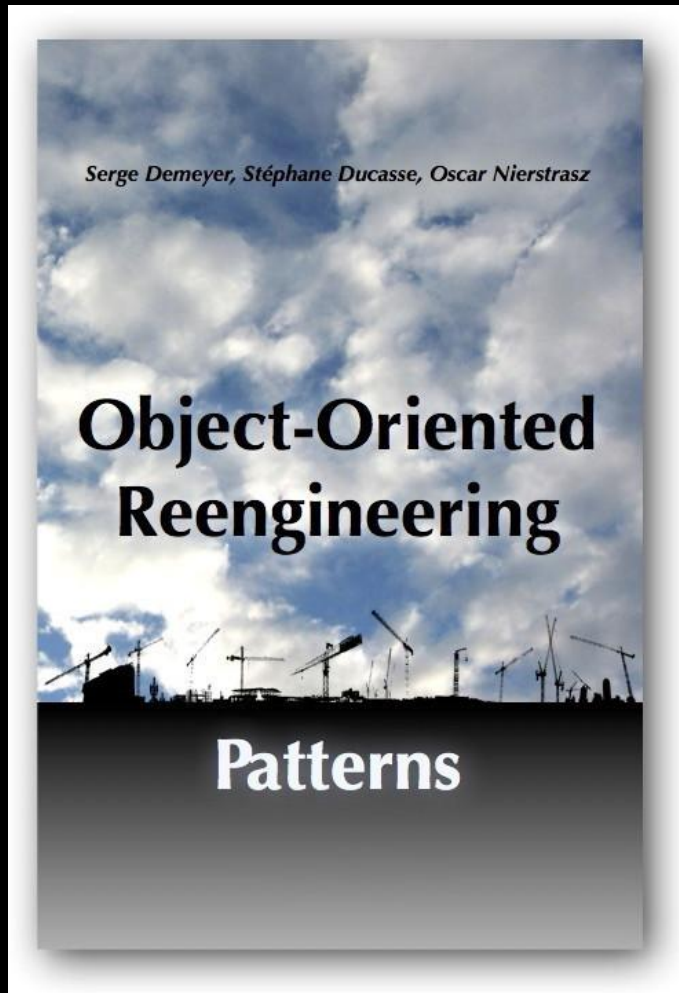
# FORWARD & REVERSE ENGINEERING, RE-ENGINEERING



# FORWARD & REVERSE ENGINEERING, RE-ENGINEERING



# OORP BOOK



- Patterns
  - Expert solutions to common problems
  - Document best practices
  - Should not apply blindly
  - Described using names, context, forces
- You may have heard of design patterns

# 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 on 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.



# FORMAT OF A PATTERN

<b><i>If It Ain't Broke, Don't Fix It</i></b>	<i>The name is usually an action phrase.</i>
<i>Intent: Save your reengineering effort for the parts of the system that will make a difference.</i>	<i>The intent should capture the essence of the pattern</i>
<b>Problem</b> Which parts of a legacy system should you reengineer? <i>This problem is difficult because:</i> <ul style="list-style-type: none"><li>• Legacy software systems can be large and complex.</li><li>• Rewriting everything is expensive and risky.</li></ul>	<i>The problem is phrased as a simple question. Sometimes the context is explicitly described.</i> <i>Next we discuss the forces! They tell us why the problem</i>
<b><i>If It Ain't Broke, Don't Fix It</i></b>	<i>The name is usually an action phrase.</i>
<i>Intent: Save your reengineering effort for the parts of the system that will make a difference.</i>	<i>The intent should capture the essence of the pattern</i>
<b>Cons</b> Delaying repairs that do not seem critical may cost you more in the long run.	
<b>Difficulties</b> It can be hard to determine what is "broken".	<i>There may follow a realistic example of applying the pattern.</i>
<b>Rationale</b> 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.	<i>We explain why the solution makes sense.</i>
<b>Known Uses</b> Alan M. Davis discusses this in his book, <i>201 Principles of Software Development</i> .	<i>We list some well documented instances of the pattern.</i>
<b>Related Patterns</b> Be sure to Fix Problems, Not Symptoms.	<i>Related patterns may suggest alternative actions. Other patterns may suggest logical followup action.</i>
<b>What Next</b> Consider starting with the Most Valuable First.	

# FORMAT OF A PATTERN

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

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

**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.

**Solution**

Only fix the parts that are "broken" that

**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.

**What Next**

Consider starting with the Most Valuable First.

*The name is usually an action phrase.*

*The intent should capture the essence of the pattern*

*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.*

*The solution sometimes includes a*

*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.*

*Suggest alternative actions. Other patterns may suggest logical followup action.*



# 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 this

**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.*

**Cons** Maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.

*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

<b><i>If It Ain't Broke, Don't Fix It</i></b>	<i>The name is usually an action phrase.</i>
<i>Intent: Save your reengineering effort for the parts of the system that will make a difference.</i>	<i>The intent should capture the essence of the pattern</i>
<b>Problem</b> Which parts of a legacy system should you reengineer? <i>This problem is difficult because:</i> <ul style="list-style-type: none"><li>• Legacy software systems can be large and complex.</li><li>• Rewriting everything is expensive and risky.</li></ul> <i>Yet, solving this problem is feasible because:</i> <ul style="list-style-type: none"><li>• Reengineering is always driven by some concrete goals.</li></ul>	<i>The problem is phrased as a simple question. Sometimes the context is explicitly described.</i>  <i>Next we discuss the forces! They tell us why the problem is difficult and interesting. We also pinpoint the key to solving the problem.</i>
<b>Solution</b> Only fix the things that are broken and can no longer be fixed.	
<b>Tradeoffs</b> <b>Pros</b> You don't waste your time fixing things that are not on your critical path. <b>Cons</b> Delaying repairs that do not seem critical may cost you more in the long run. <b>Difficulties</b> It can be hard to determine what is "broken".	<i>Each pattern entails some positive and negative tradeoffs.</i>
<b>Rationale</b> 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.	<i>There may follow a realistic example of applying the pattern.</i>
<b>Known Uses</b> Alan M. Davis, <i>201 Principles of Reengineering</i>	
<b>Related Patterns</b> Be sure to fix the problem before you reengineer it.	
<b>What Next?</b> Consider starting with a small, isolated part of the system.	<i>We explain why the solution makes sense.</i>

# 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.

*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**

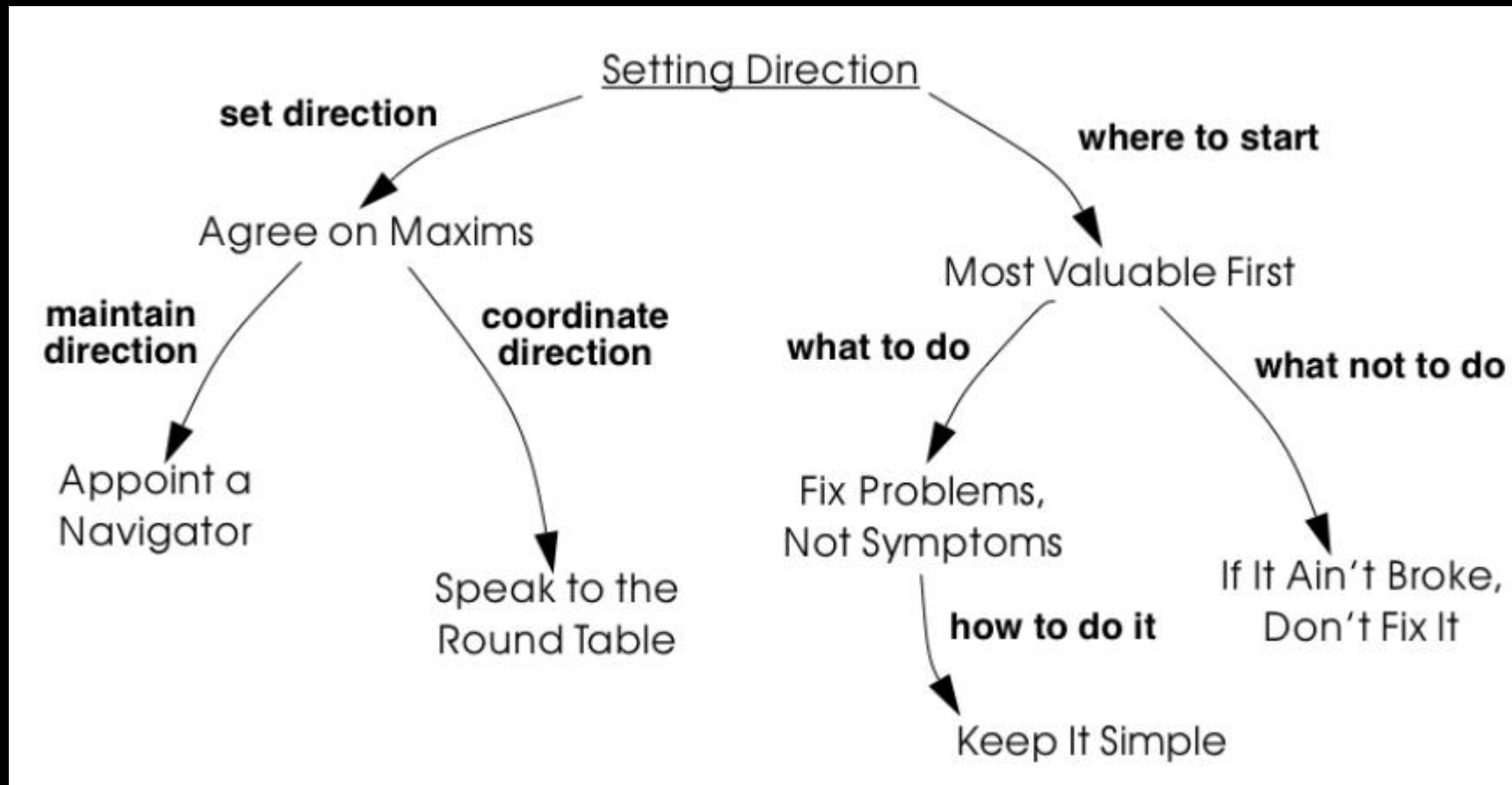
Consider starting with the Most Valuable First.

# SOME ASPECTS / FORCES

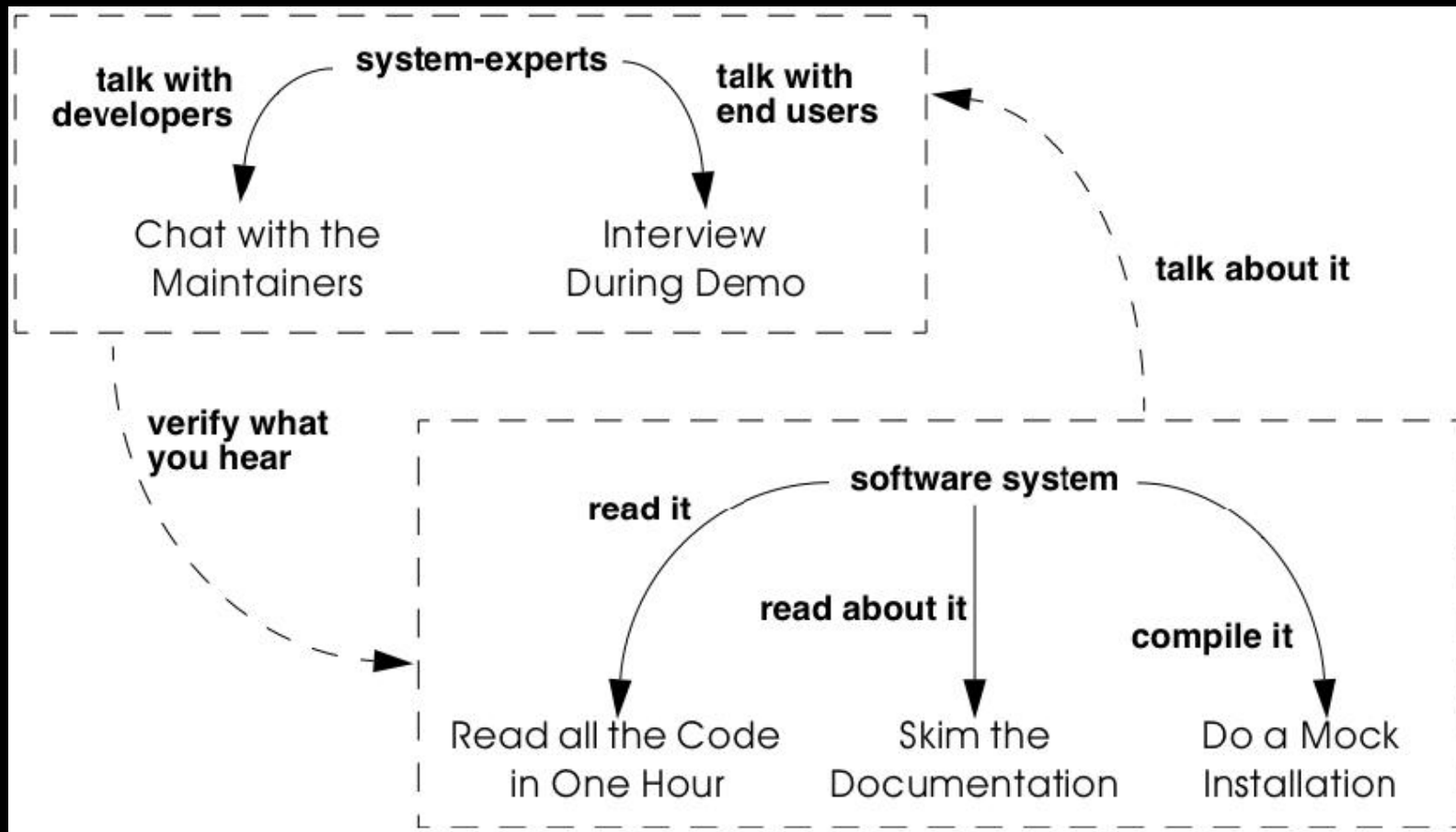
- Limited resources
- Techniques and tools
- Reliable information
- Accurate abstraction
- Skeptical colleagues



# SETTING DIRECTION PATTERNS



# FIRST CONTACT PATTERNS



# CLASS EXERCISE

How would you read all code in one hour?

4 Min