Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 13

Reengineering Patterns

Part 1

Reengineering

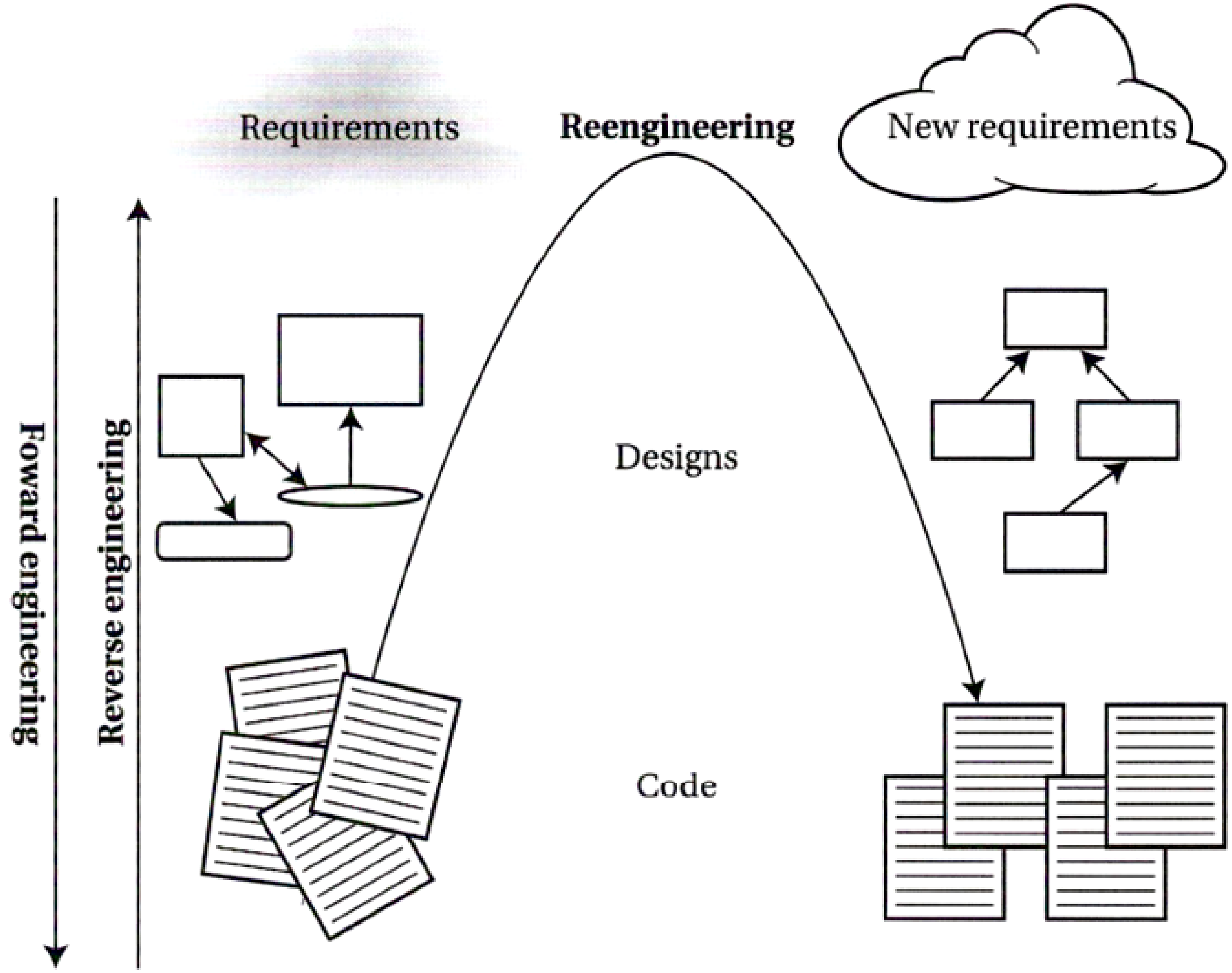
- **Goal of Reengineering**
  - □ Reducing the complexity of a *legacy system* sufficiently so that it can continue to be used and adapted at an acceptable cost.

- **Reasons for Reengineering**
  - □ *Unbundling a monolithic system* so that the individual parts can be more easily marketed separately or combined in different ways.
  - □ *Improving performance.*
  - □ *Porting the system to a new platform.*
  - □ *Extracting the design* as a first step to a new implementation.
  - □ *Exploiting new technology* as a step toward cutting maintenance costs.
  - □ *Reducing human dependencies* by documenting knowledge about the system and making it easier to maintain.

Symptoms of the Need for Reengineering

- *Obsolete or no documentation.*
- *Missing tests.*
- *Departure of the original developers or users.*
- *Disappearance of inside knowledge about the system:* The documentation is out of sync with the existing code base.
- *Limited understanding of the entire system.*
- *Too long to turn things over to production.*
- *Too much time to make simple changes.*
- *Need for constant bug fixes.*
- *Maintenance dependencies.*
- *Difficulties separating products.*
- *Duplicated code.*
- *Code smells.*

Reengineering Lifecycle



Reengineering Problems: Architectural

- *Insufficient documentation:* Documentation either does not exist or is inconsistent with reality.
- *Improper layering:* Missing or improper layering hampers portability and adaptability.
- *Lack of modularity:* Strong coupling between modules hampers evolution.
- *Duplicated code:* "Copy, paste, and edit" is quick and easy, but leads to maintenance nightmares.
- *Duplicated functionality:* Similar functionality is reimplemented by separate teams, leading to code bloat.

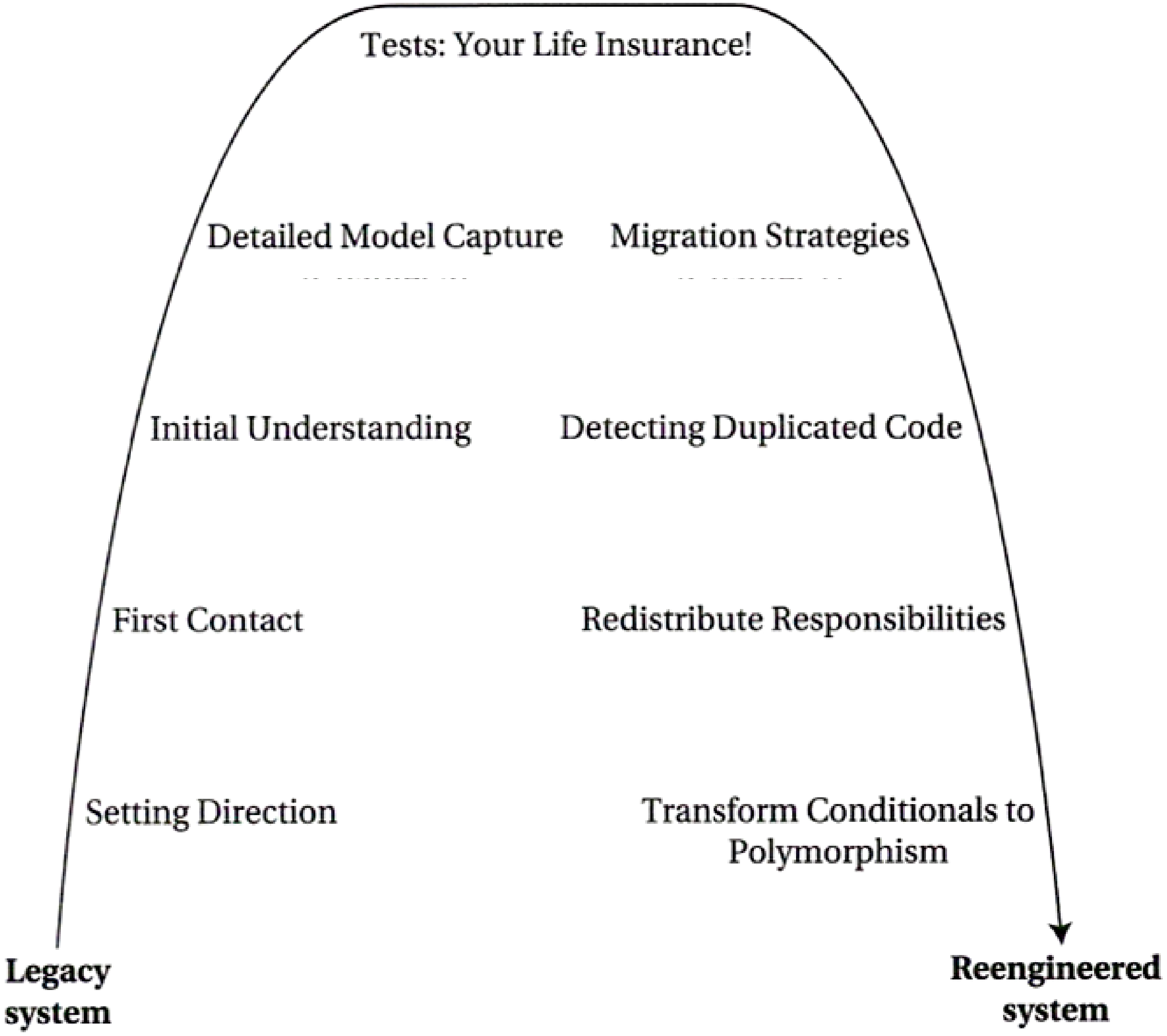
Reengineering Problems: Design

- *Misuse of inheritance:* For composition and code reuse rather than polymorphism
- *Missing inheritance:* Duplicated code and case statements to select behavior
- *Misplaced operations:* Excessive coupling
- *Violation of encapsulation*
- *Class abuse:* Lack of cohesion

Reengineering Patterns

- *Reengineering patterns* codify and record knowledge about modifying legacy software.
- They are stable units of expertise that can be consulted in any reengineering effort:
  - □ they help in diagnosing problems and identifying weaknesses that may hinder further development of the system, and
  - □ they aid in finding solutions that are more appropriate to the new requirements.

Reengineering Patterns: Categories



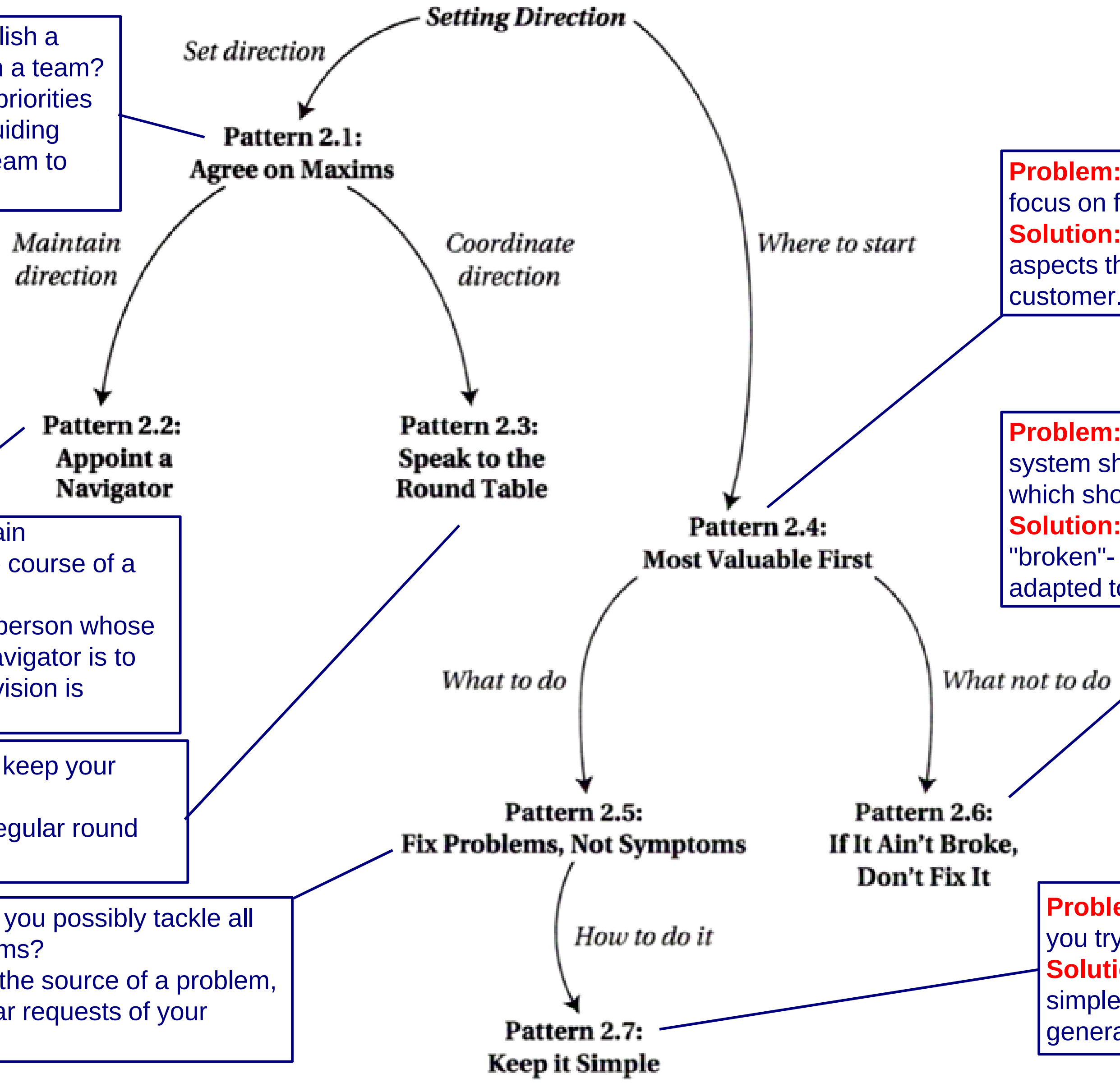
Reengineering Patterns: Categories (1)

1. *Setting Direction:* help determine where to focus reengineering efforts and make sure that they stay on track.
2. *First Contact:* useful when a legacy system is encountered for the first time.
3. *Initial Understanding:* help develop a first simple model of a legacy system, mainly in the form of class diagrams.
4. *Detailed Model Capture:* help develop a more detailed model of a particular component of the system.
5. *Tests:* use of testing not only to help understand a legacy system, but also to prepare it for a reengineering effort.

Reengineering Patterns: Categories (2)

6. *Migration Strategies:* help keep a system running while it is being reengineered and increase the chances that the new system will be accepted by its users.
7. *Detecting Duplicated Code:* help identify locations where code may have been copied and pasted, or merged from different versions of the software.
8. *Redistribute Responsibilities:* help discover and reengineer classes with too many responsibilities.
9. *Transform Conditionals to Polymorphism:* help redistribute responsibilities when an object-oriented design has been compromised over time.

Reengineering Patterns: *Setting Direction*



Setting Direction

Set direction

Pattern 2.1:
Agree on Maxims

Maintain direction

Coordinate direction

Where to start

Pattern 2.2:
Appoint a Navigator

Pattern 2.3:
Speak to the Round Table

Pattern 2.4:
Most Valuable First

What to do

What not to do

Pattern 2.5:
Fix Problems, Not Symptoms

Pattern 2.6:
If It Ain't Broke, Don't Fix It

How to do it

Pattern 2.7:
Keep it Simple

Problem: How do you establish a common sense of purpose in a team?
Solution: Establish the key priorities for the project and identify guiding principles that will help the team to stay on track.

Problem: Which problems should you focus on first?
Solution: Start working on the aspects that are most valuable to your customer.

Problem: Which parts of a legacy system should you reengineer and which should you leave as they are?
Solution: Only fix the parts that are "broken"- those that can no longer be adapted to planned changes.

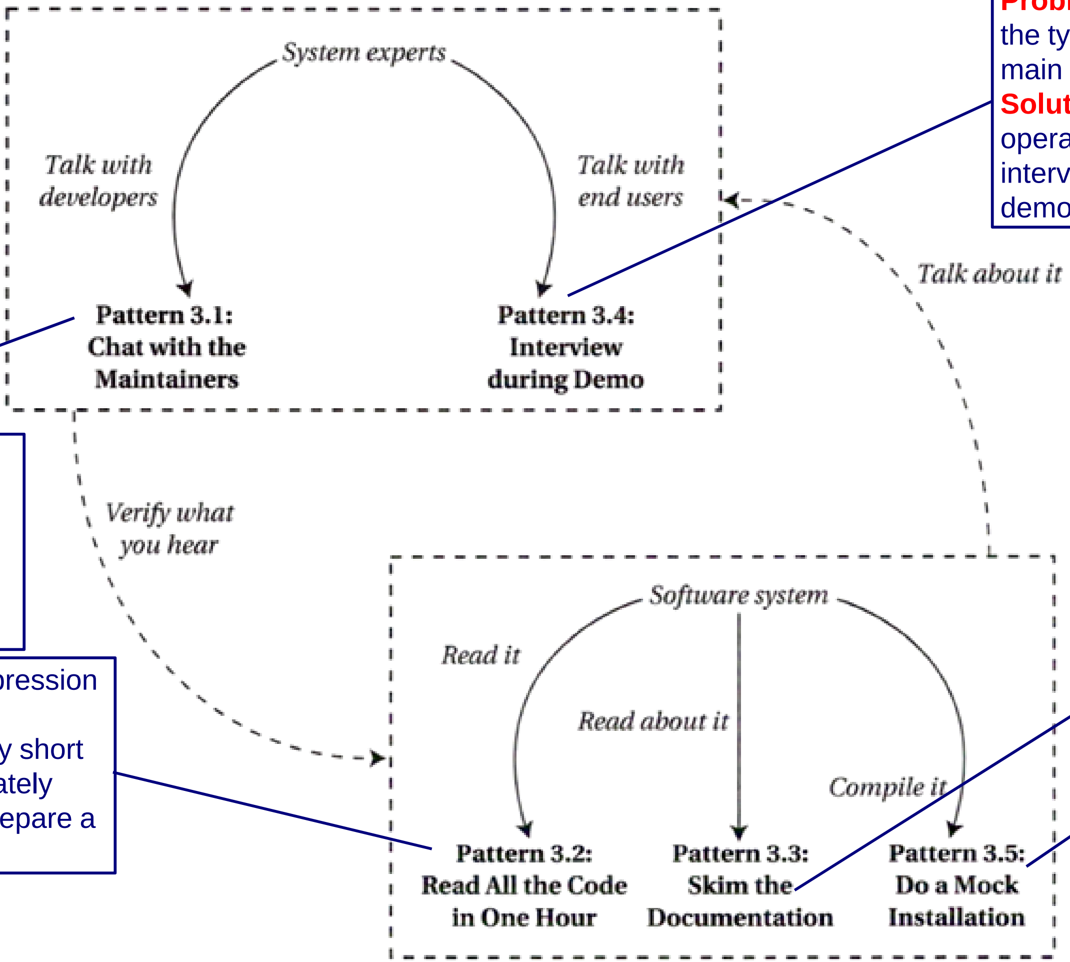
Problem: How do you maintain architectural vision during the course of a complex project?
Solution: Appoint a specific person whose responsibility in the role of Navigator is to ensure that the architectural vision is maintained.

Problem: How do you keep your team synchronized?
Solution: Hold brief, regular round table meetings.

Problem: How can you possibly tackle all the reported problems?
Solution: Address the source of a problem, rather than particular requests of your Stakeholders.

Problem: How much flexibility should you try to build into the new system?
Solution: Prefer an adequate but simple solution to a potentially more general but complex solution.

Reengineering Patterns: *First Contact*


System experts
Talk with developers
Talk with end users
Pattern 3.1: Chat with the Maintainers
Pattern 3.4: Interview during Demo
Talk about it
Verify what you hear
Software system
Read it
Read about it
Compile it
Pattern 3.2: Read All the Code in One Hour
Pattern 3.3: Skim the Documentation
Pattern 3.5: Do a Mock Installation

Problem: How do you get a good perspective on the historical and political context of the legacy system you are reengineering?
Solution: Discuss the problem with the system maintainers.

Problem: How can you get a first impression of the quality of the source code?
Solution: Grant yourself a reasonably short amount of study time (e.g., approximately one hour) to read the source code; prepare a report of your findings.

Problem: How can you get an idea of the typical usage scenarios and the main features of a software system?
Solution: Observe the system in operation by seeing a demo and interviewing the person who is demonstrating.

Problem: How do you identify those parts of the documentation that might be of help?
Solution: Prepare a list summarizing interesting aspects of the system, match this list against the documentation, and make a crude assessment of how up to date the documentation seems.

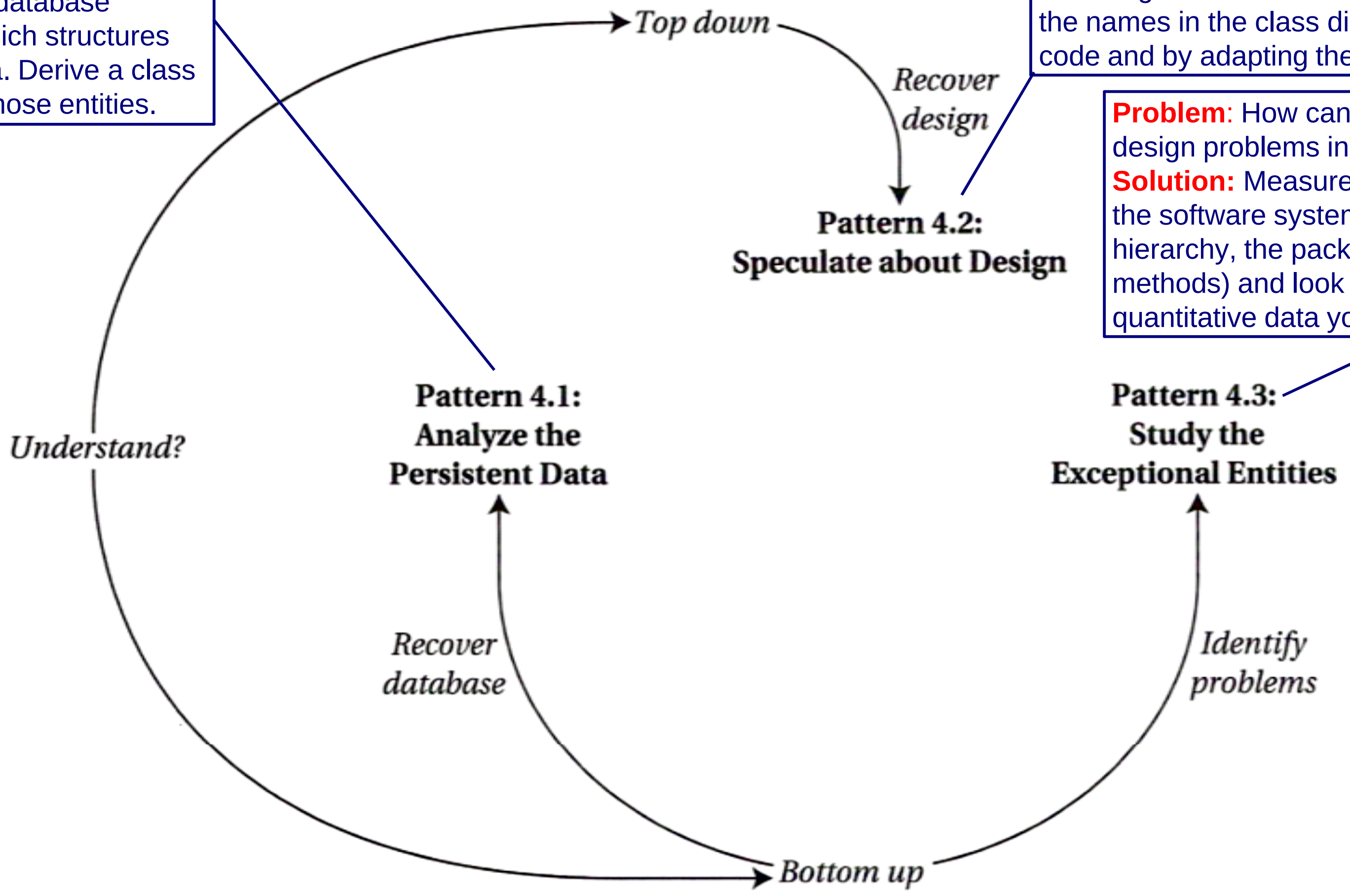
Problem: How can you be sure that you will be able to (re)build the system?
Solution: Try to install and build the system in a clean environment during a limited amount of time (at most one day).

Reengineering Patterns: *Initial Understanding*

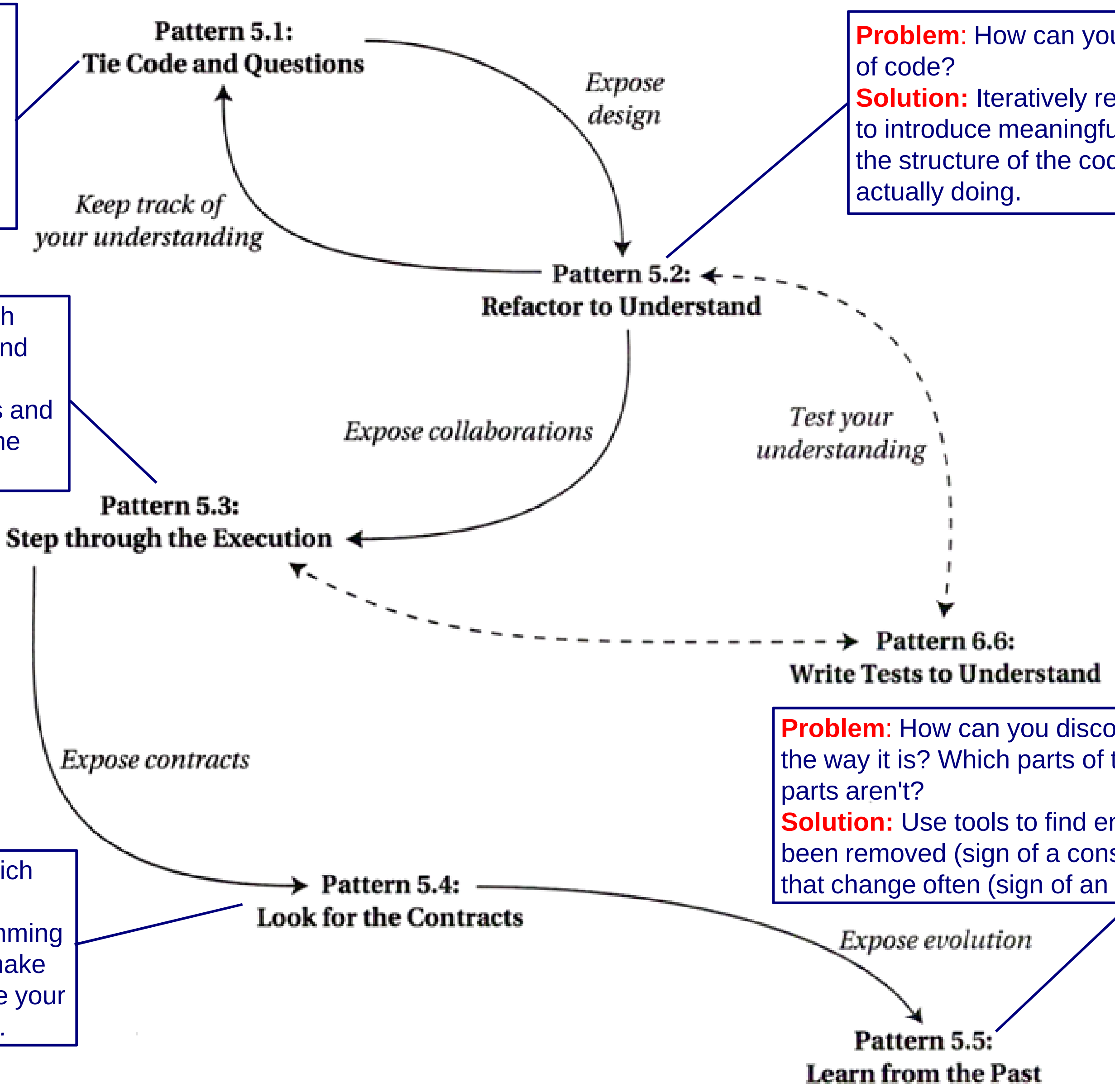
Problem: Which object structures represent the valuable data?
Solution: Analyze the database schema and assess which structures represent valuable data. Derive a class diagram representing those entities.

Problem: How do you recover the way design concepts are represented in the source code?
Solution: Use your development expertise to conceive a hypothetical class diagram representing the design. Refine that model by verifying whether the names in the class diagram occur in the source code and by adapting the model accordingly.

Problem: How can you quickly identify potential design problems in large software systems?
Solution: Measure the structural entities forming the software system (i.e., the inheritance hierarchy, the packages, the classes, and the methods) and look for exceptions in the quantitative data you collected.


Top down
Recover design
Pattern 4.2: Speculate about Design
Understand?
Pattern 4.1: Analyze the Persistent Data
Pattern 4.3: Study the Exceptional Entities
Recover database
Identify problems
Bottom up

Reengineering Patterns: *Detailed Model Capture*



Pattern 5.1: Tie Code and Questions

Expose design

Keep track of your understanding

Pattern 5.2: Refactor to Understand

Expose collaborations

Test your understanding

Pattern 5.3: Step through the Execution

Pattern 6.6: Write Tests to Understand

Expose contracts

Pattern 5.4: Look for the Contracts

Expose evolution

Pattern 5.5: Learn from the Past

Problem: How do you keep track of, synchronize and share your understanding about a piece of code and the questions that you have?
Solution: While you are working on the code, annotate it directly with the questions you are facing.

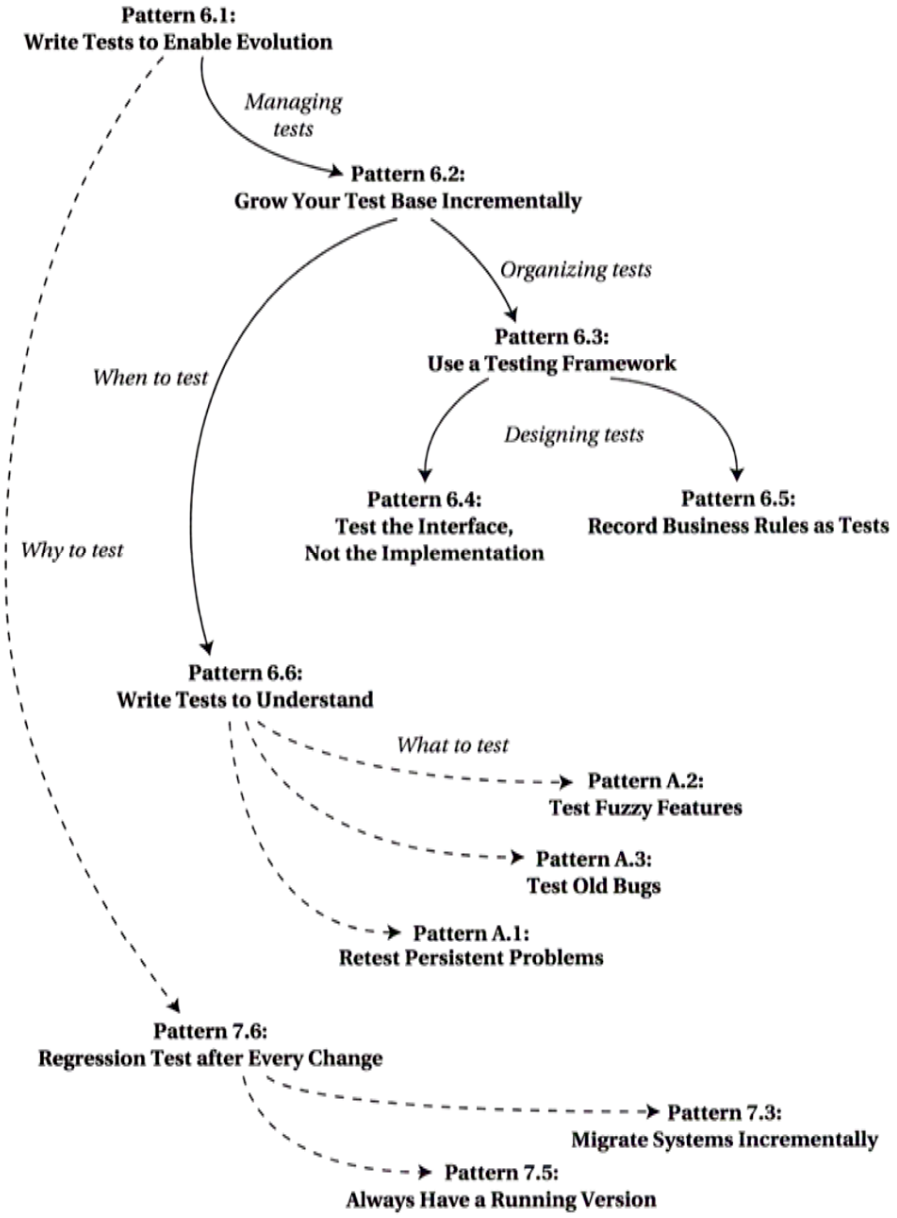
Problem: How can you understand a cryptic piece of code?
Solution: Iteratively rename and refactor the code to introduce meaningful names and to make sure the structure of the code reflects what the system is actually doing.

Problem: How do you discover which objects are instantiated at run time and how they collaborate?
Solution: Run each of the scenarios and use your debugger to step through the code.

Problem: How can you discover why the system is designed the way it is? Which parts of the system are stable and which parts aren't?
Solution: Use tools to find entities where functionality has been removed (sign of a consolidating design), and entities that change often (sign of an unstable part of the design).

Problem: How do you determine which *contracts* a class supports?
Solution: Look for common programming idioms that expose the way clients make use of the class interface. Generalize your observations in the form of *contracts.*

Reengineering Patterns: *Tests*



Pattern 6.1:
Write Tests to Enable Evolution

Managing tests

Pattern 6.2:
Grow Your Test Base Incrementally

Organizing tests

Pattern 6.3:
Use a Testing Framework

When to test

Designing tests

Pattern 6.4:
Test the Interface,
Not the Implementation

Pattern 6.5:
Record Business Rules as Tests

Why to test

Pattern 6.6:
Write Tests to Understand

What to test

Pattern A.2:
Test Fuzzy Features

Pattern A.3:
Test Old Bugs

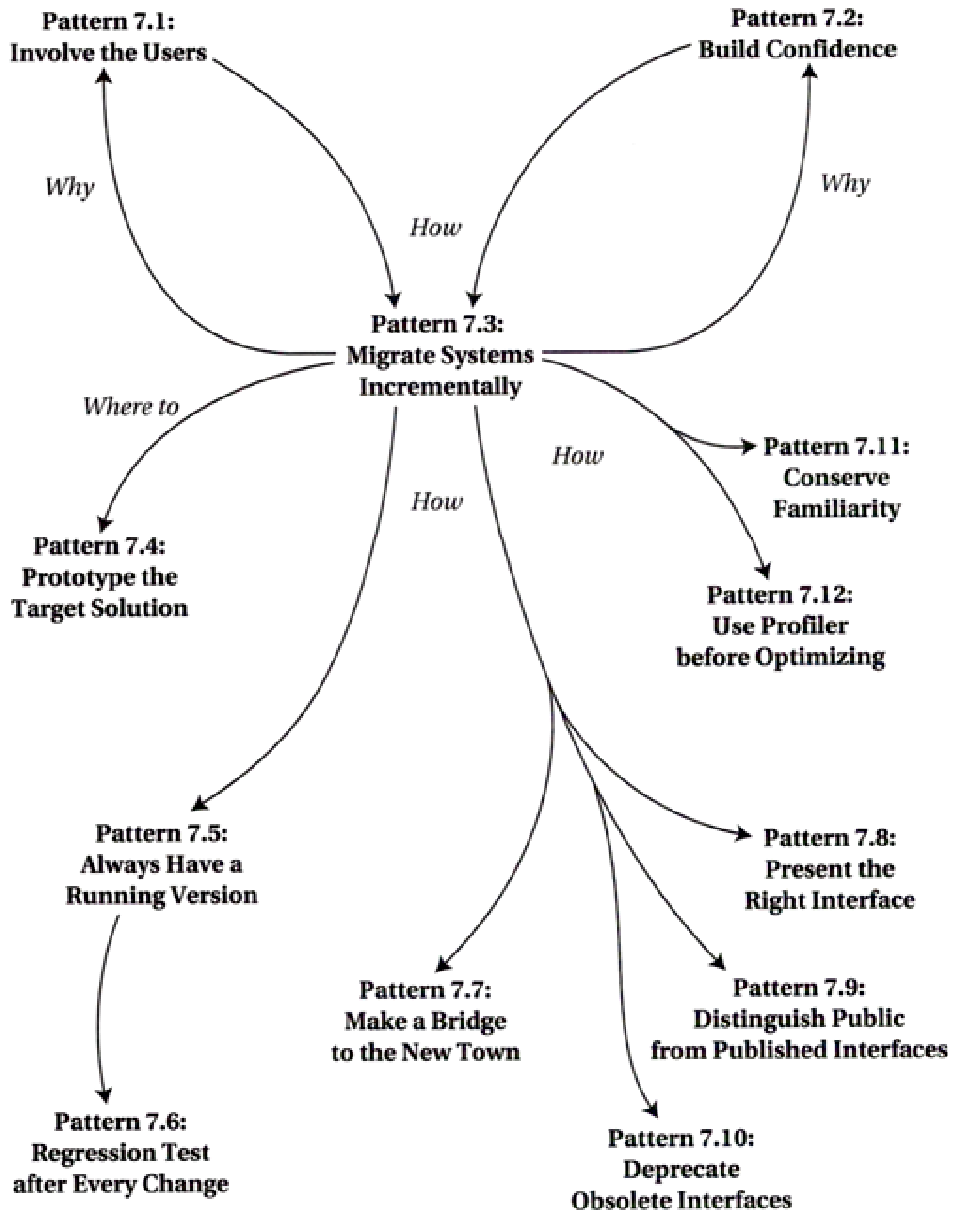
Pattern A.1:
Retest Persistent Problems

Pattern 7.6:
Regression Test after Every Change

Pattern 7.3:
Migrate Systems Incrementally

Pattern 7.5:
Always Have a Running Version

Reengineering Patterns: Migration Strategies



Reference

- Demeyer, S., Ducasse, S., and Nierstrasz, O., *Object-Oriented Reengineering Patterns,* Elsevier Science, 2003.