UNIVERSITÄT
BERN

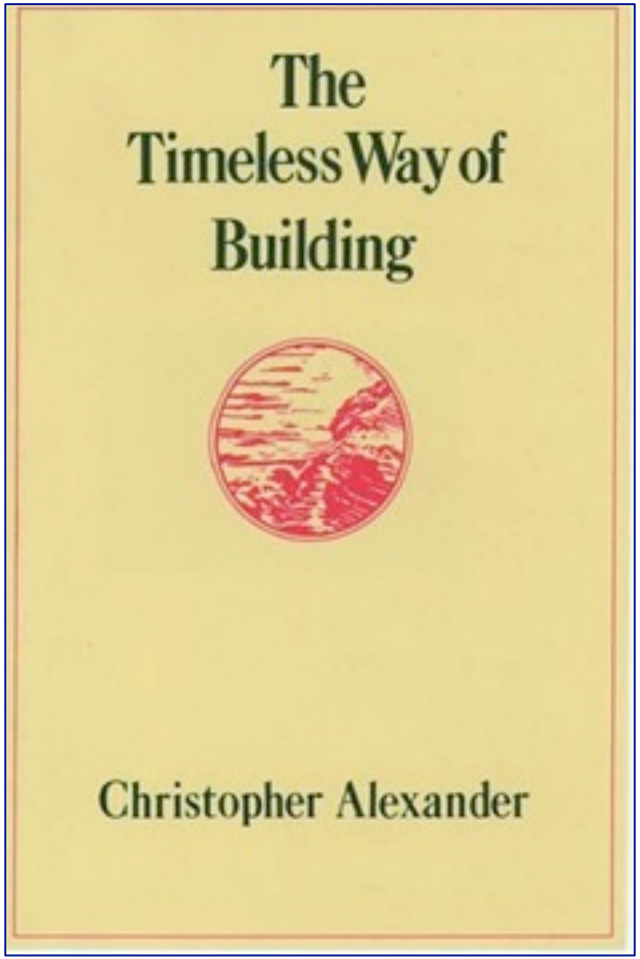
Introduction to Reverse Engineering

(based on the Object Oriented Reengineering Patterns)

Mircea Lungu

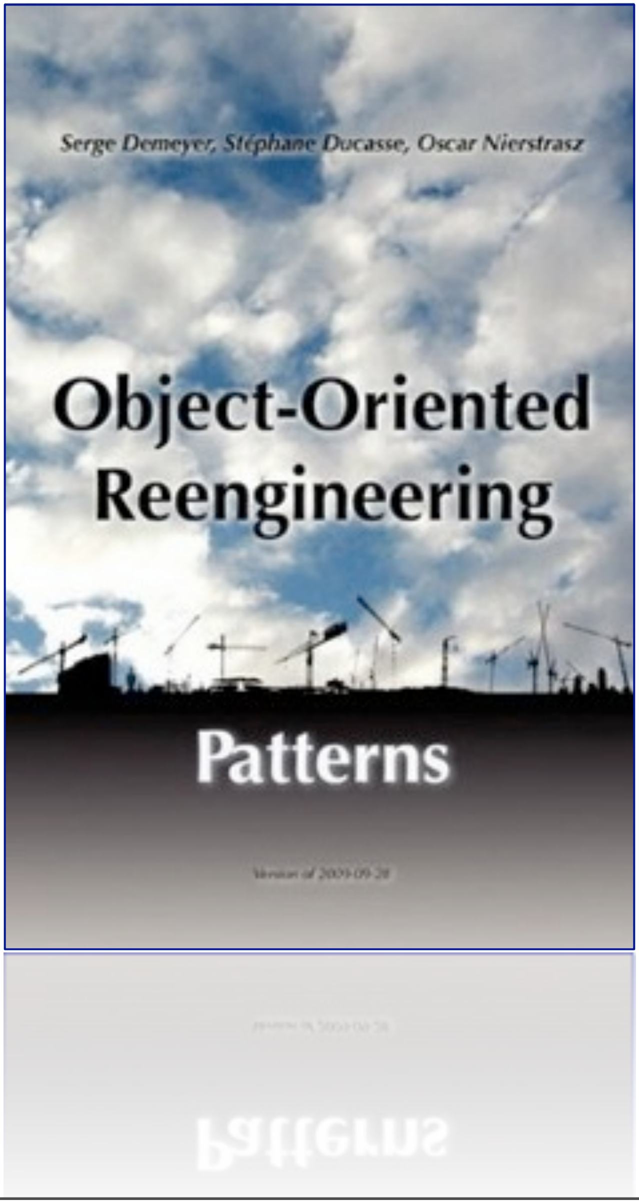
Selected material courtesy Oscar Nierstrasz

Date

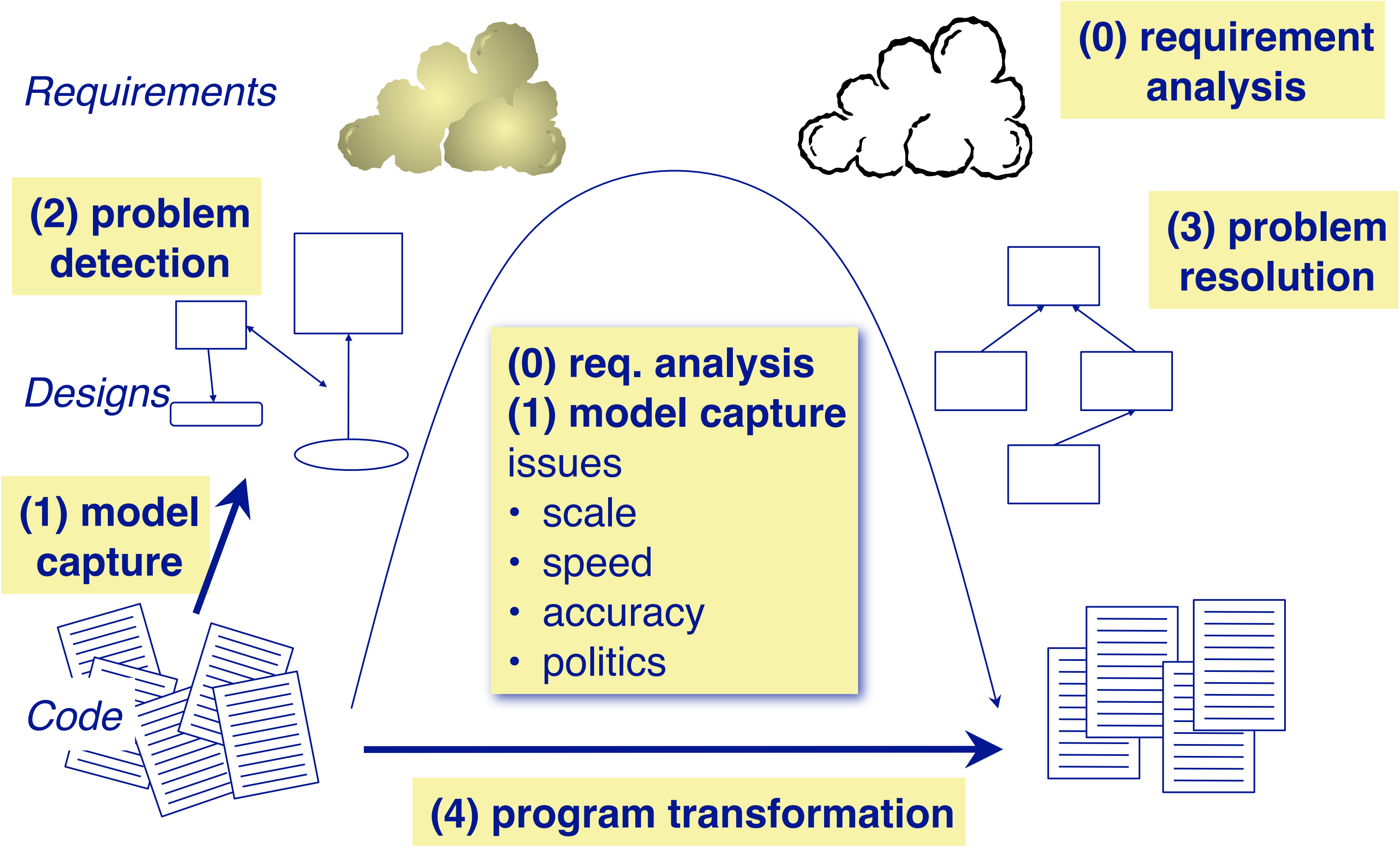


The
Timeless Way of
Building

Christopher Alexander


Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Object-Oriented Reengineering
Patterns
Version of 2009-09-28

The Reengineering Life-Cycle



Requirements

(0) requirement analysis

(2) problem detection

(3) problem resolution

Designs

(0) req. analysis
(1) model capture
issues

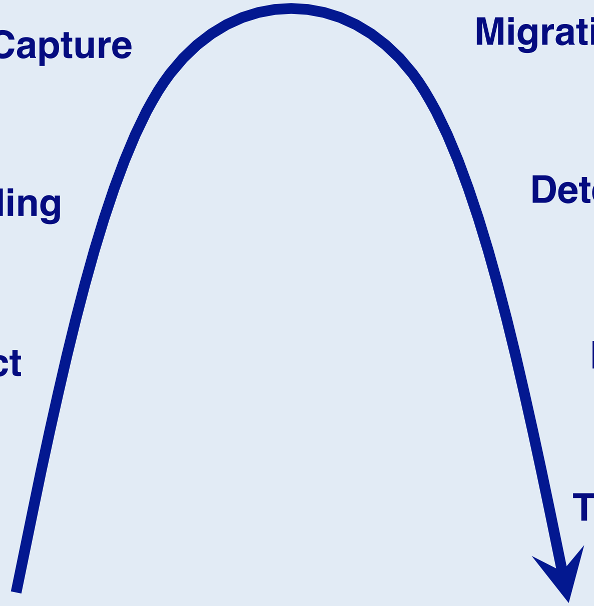
- scale
- speed
- accuracy
- politics

(1) model capture

Code

(4) program transformation

Roadmap



Setting Direction

First Contact

Initial Understanding

Detailed Model Capture

Tests: Your Life Insurance

Migration Strategies

Detecting Duplicated Code

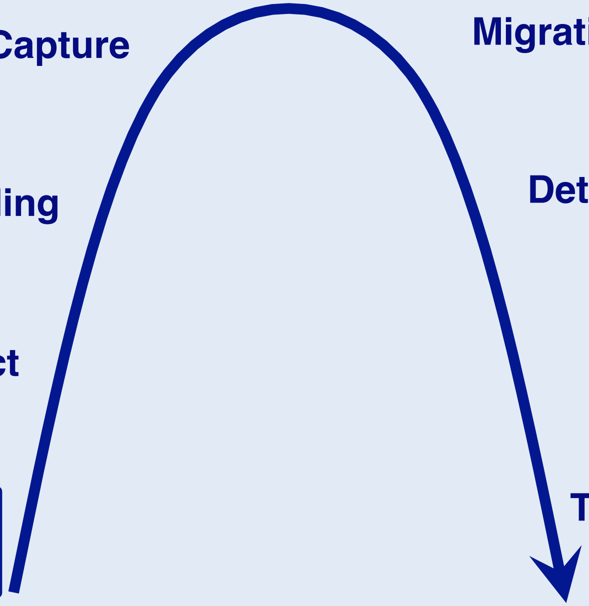
Redistribute
Responsibilities

Transform Conditionals
to Polymorphism

Reengineering

Roadmap

Tests: Your Life Insurance



Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

Reengineering

Setting Direction: Forces

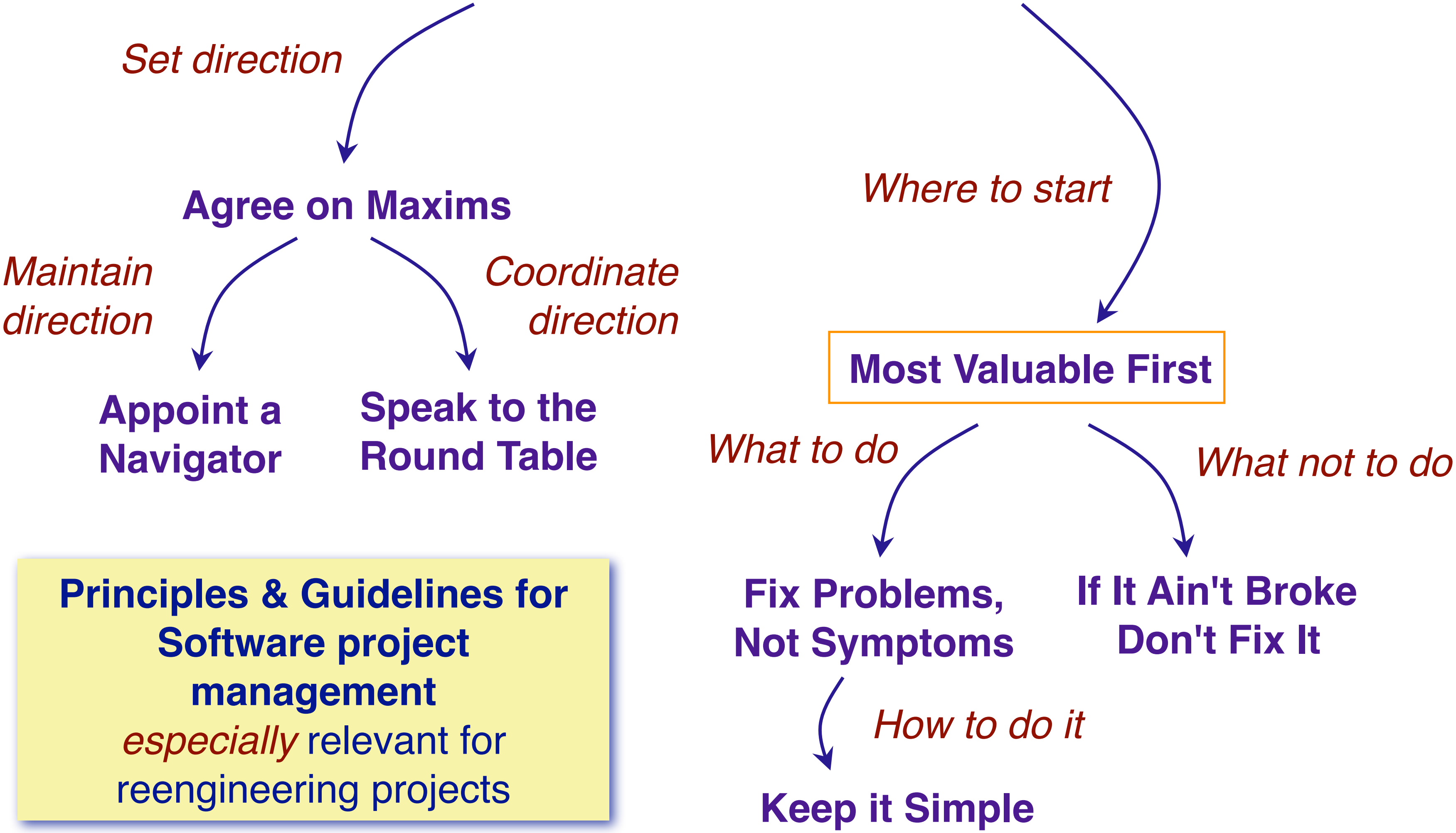
Conflicting interests (technical, economic, political)

Complication: presence or absence of *original developers*

Which problems to tackle?

- Interesting vs important problems?
- Wrap, refactor or rewrite?

Setting Direction: Patterns



Set direction

Agree on Maxims

Maintain direction

Appoint a Navigator

Coordinate direction

Speak to the Round Table

Where to start

Most Valuable First

What to do

Fix Problems, Not Symptoms

How to do it

Keep it Simple

What not to do

If It Ain't Broke Don't Fix It

Principles & Guidelines for Software project management
especially relevant for reengineering projects

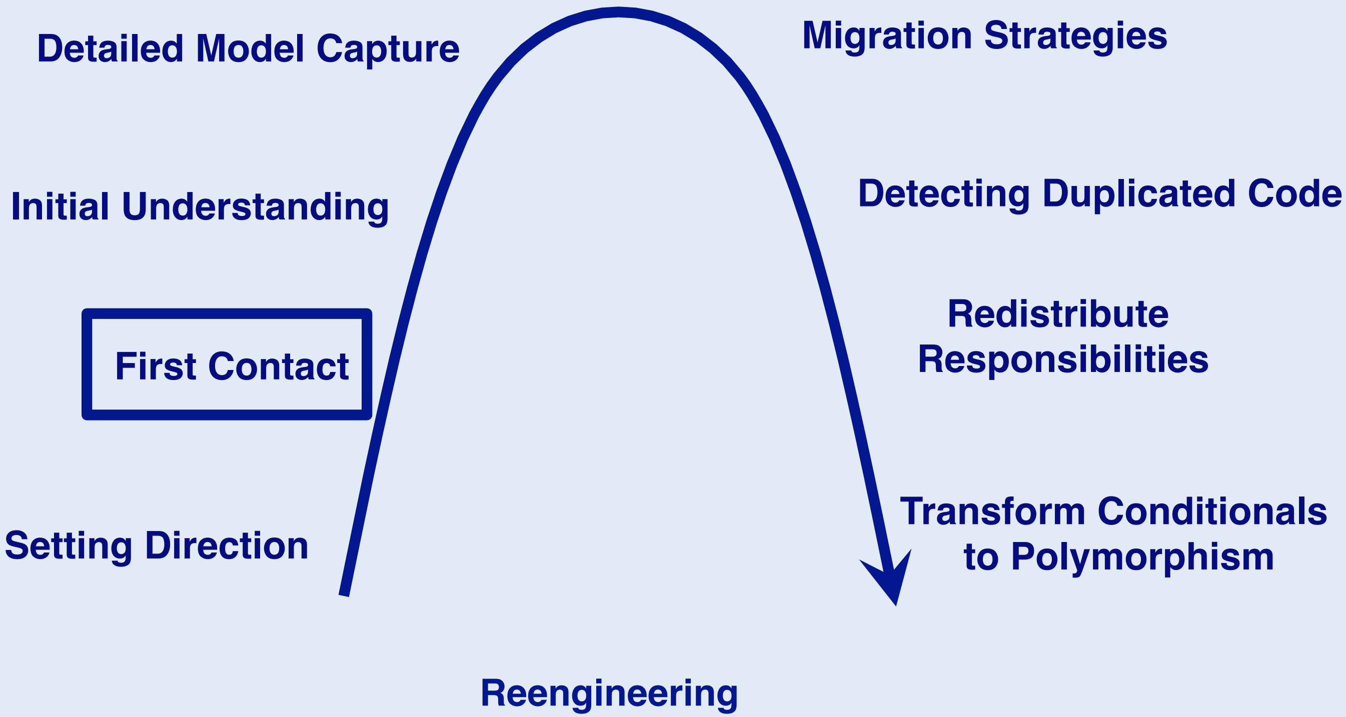
Most Valuable First

Problem: Which problems should you focus on first?

Solution: Work on aspects that are most *valuable* to your customer

- Aim for early results
- Difficulties and hints:
  - What *measurable goal* to aim for?
  - "Valuable" might be a rat's nest
  - Play the *planning game*

Roadmap



Tests: Your Life Insurance

Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

Reengineering

First Contact: Forces

Legacy systems are large and complex

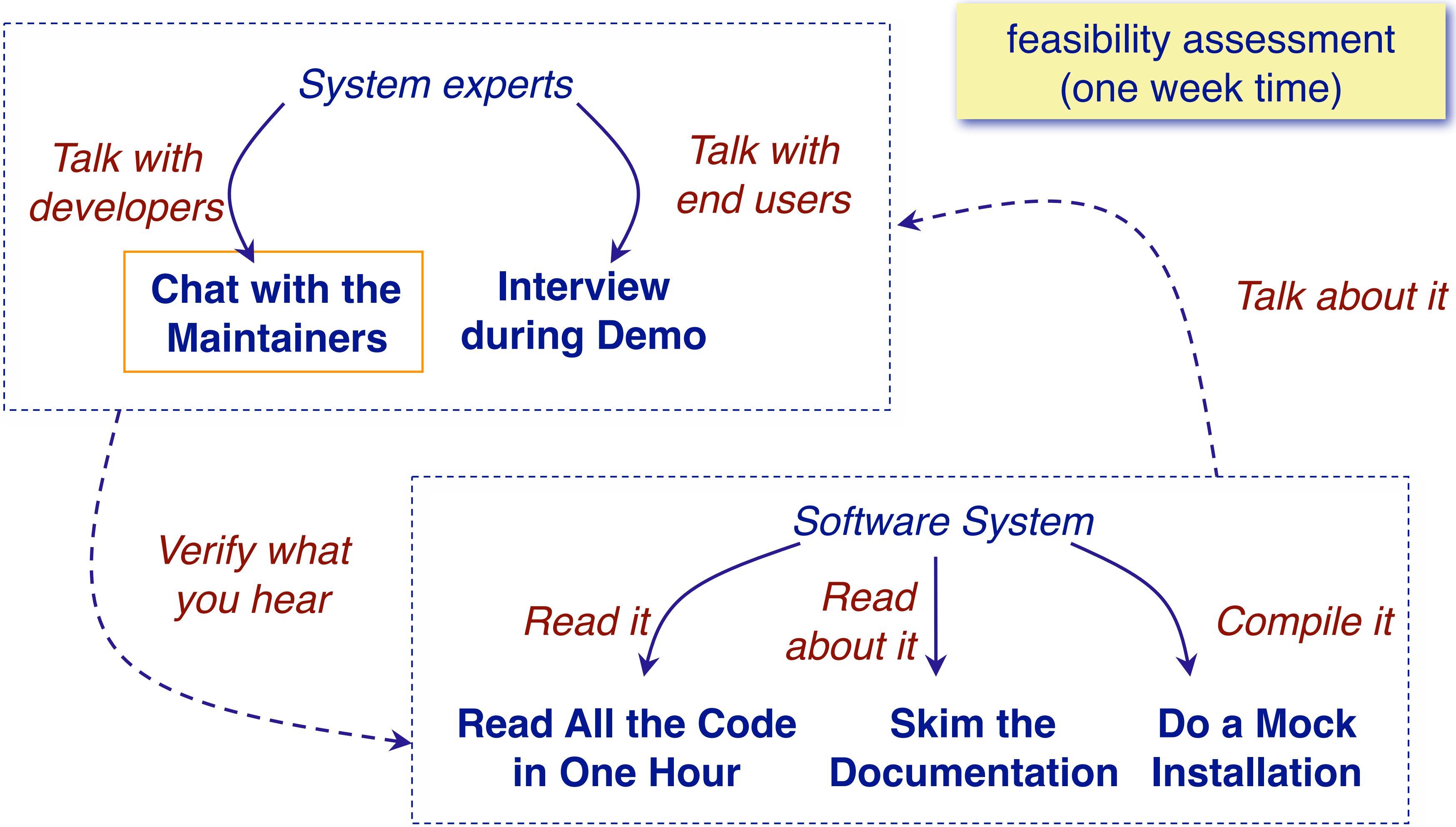
- Split the system into *manageable* pieces

Time is scarce

- Apply lightweight techniques to *assess feasibility and risks*

First impressions are dangerous

First Contact: Patterns



feasibility assessment
(one week time)

System experts

Talk with developers

Chat with the Maintainers

Talk with end users

Interview during Demo

Talk about it

Verify what you hear

Software System

Read it

Read All the Code in One Hour

Read about it

Skim the Documentation

Compile it

Do a Mock Installation

Chat with the Maintainers

Problem: *What are the history and politics of the legacy system?*
Solution: *Discuss the problems with the system maintainers.*

- Documentation will mislead you (various reasons)
- Stakeholders will mislead you (various reasons)
- The maintainers know both the technical and political history

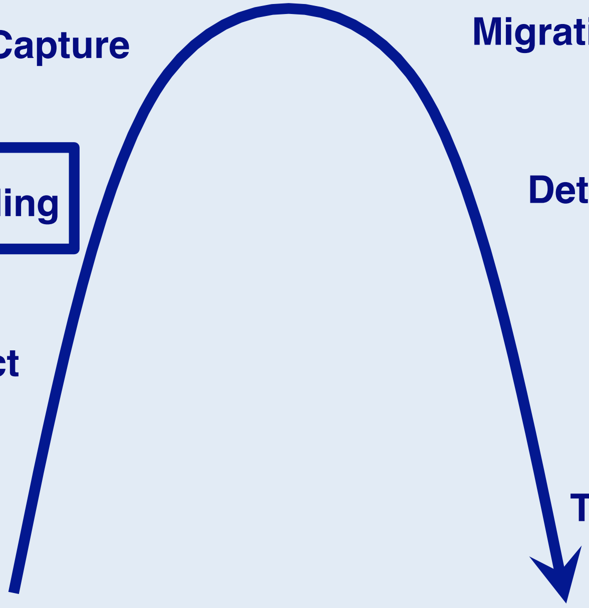
Chat with the Maintainers

Questions to ask:

- Easiest/hardest bug to fix in recent months?
- How are change requests made and evaluated?
- How did the development/maintenance team evolve during the project?
- How good is the code? The documentation?

> *The major problems of our work are no so much technological as sociological.*
> *— DeMarco and Lister, Peopleware '99*

Roadmap



Tests: Your Life Insurance

Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

Reengineering

Initial Understanding: Forces

Understanding entails iteration

- Plan *iteration* and feedback loops

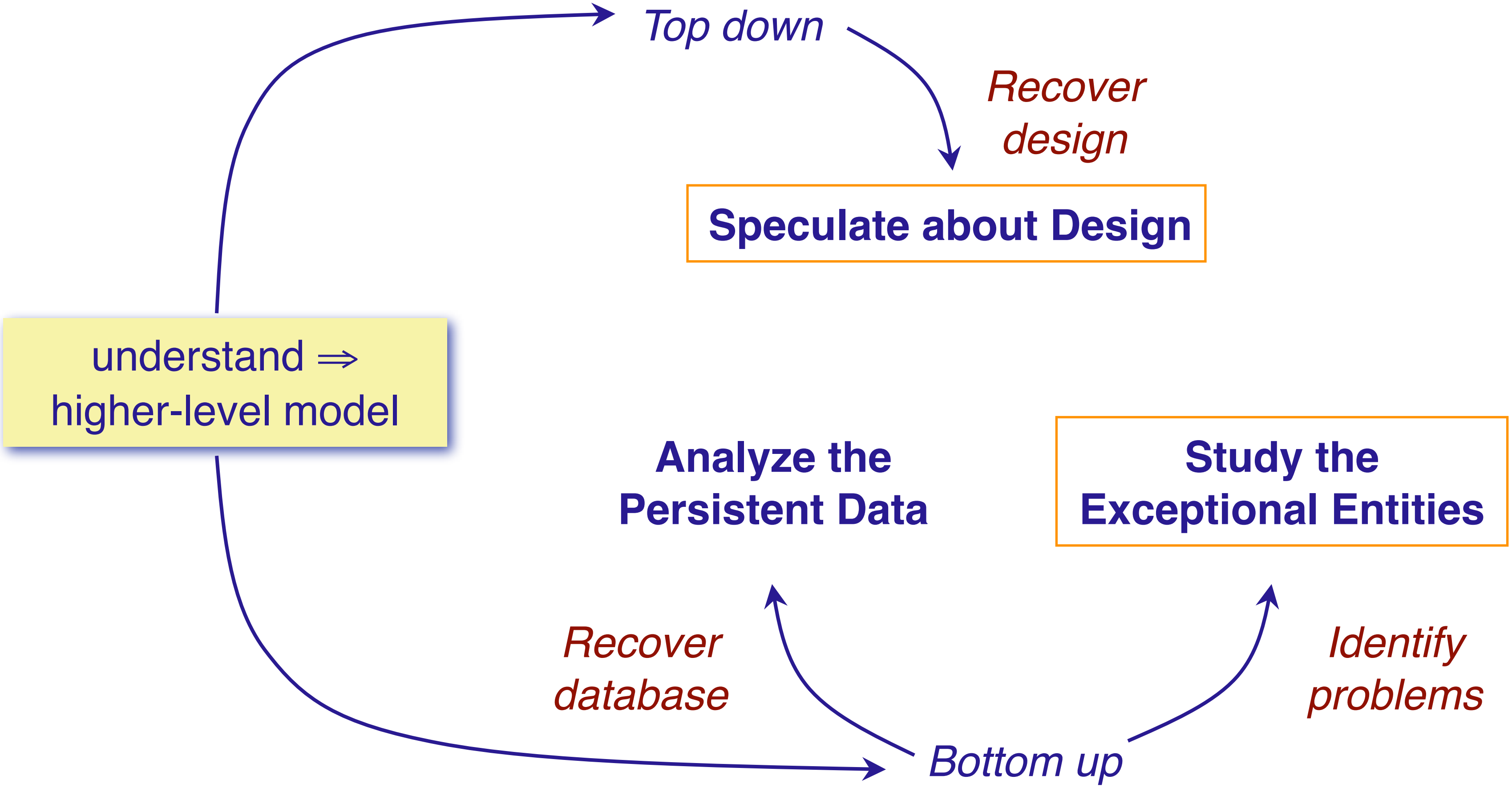
Knowledge must be shared

- “Put the map on the wall”

Teams need to communicate

- “Use their language”

Initial Understanding: Patterns



Top down

Recover design

Speculate about Design

understand ⇒ higher-level model

Analyze the Persistent Data

Study the Exceptional Entities

Recover database

Identify problems

Bottom up

Speculate about Design

Problem: How do you recover design from code?

Solution: Develop hypotheses and check them

- Develop a plausible class diagram and iteratively check and refine your design against the actual code.

Variants:

- Speculate about Design Patterns
- Speculate about Architecture

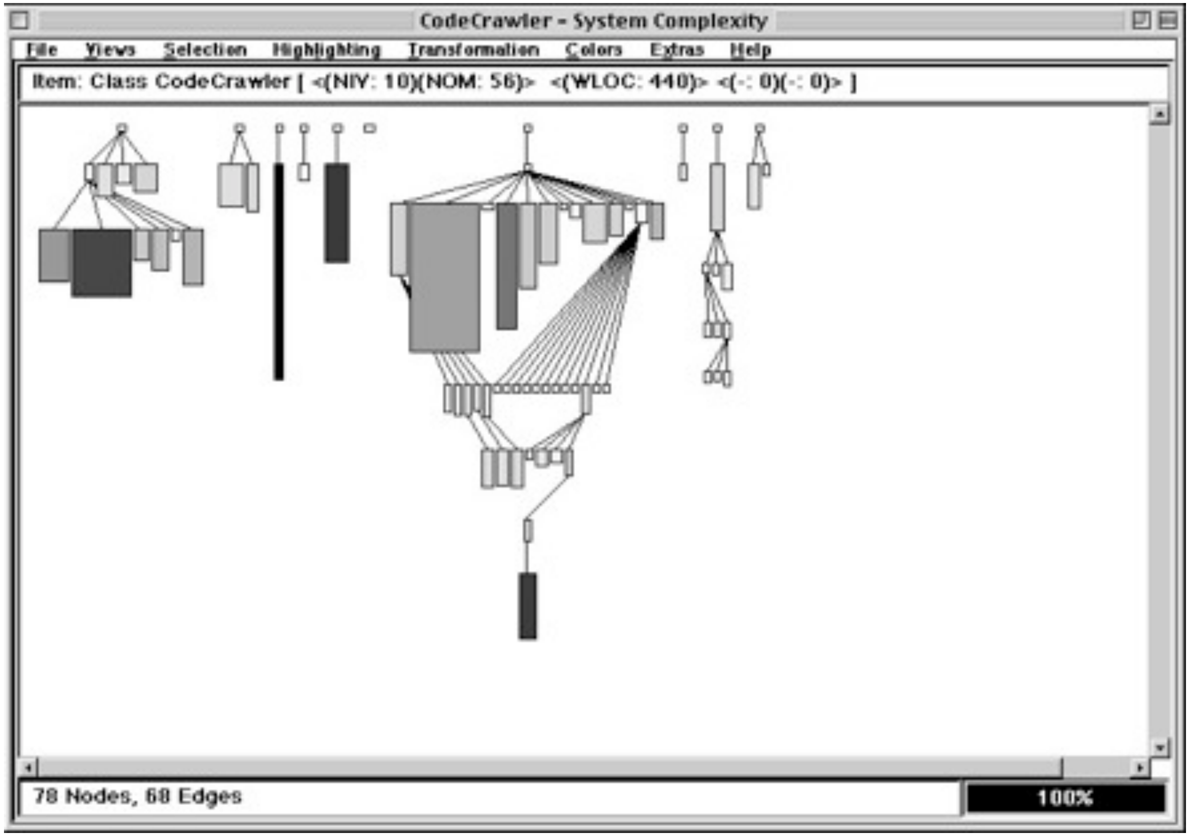
Study the Exceptional Entities

Problem: How can you quickly identify design problems?

Solution: Measure software entities and study the anomalous ones

- Combine metrics with structure to get an overview
- Browse the code to get insight into the anomalies

Visualizing Metrics

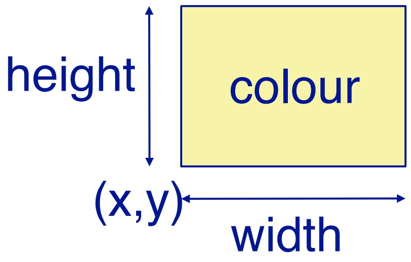

CodeCrawler - System Complexity
File Views Selection Highlighting Transformation Colors Extras Help

Item: Class CodeCrawler [<(NIV: 10)(NOM: 56)> <(WLOC: 440)> <(-: 0)(-: 0)>]

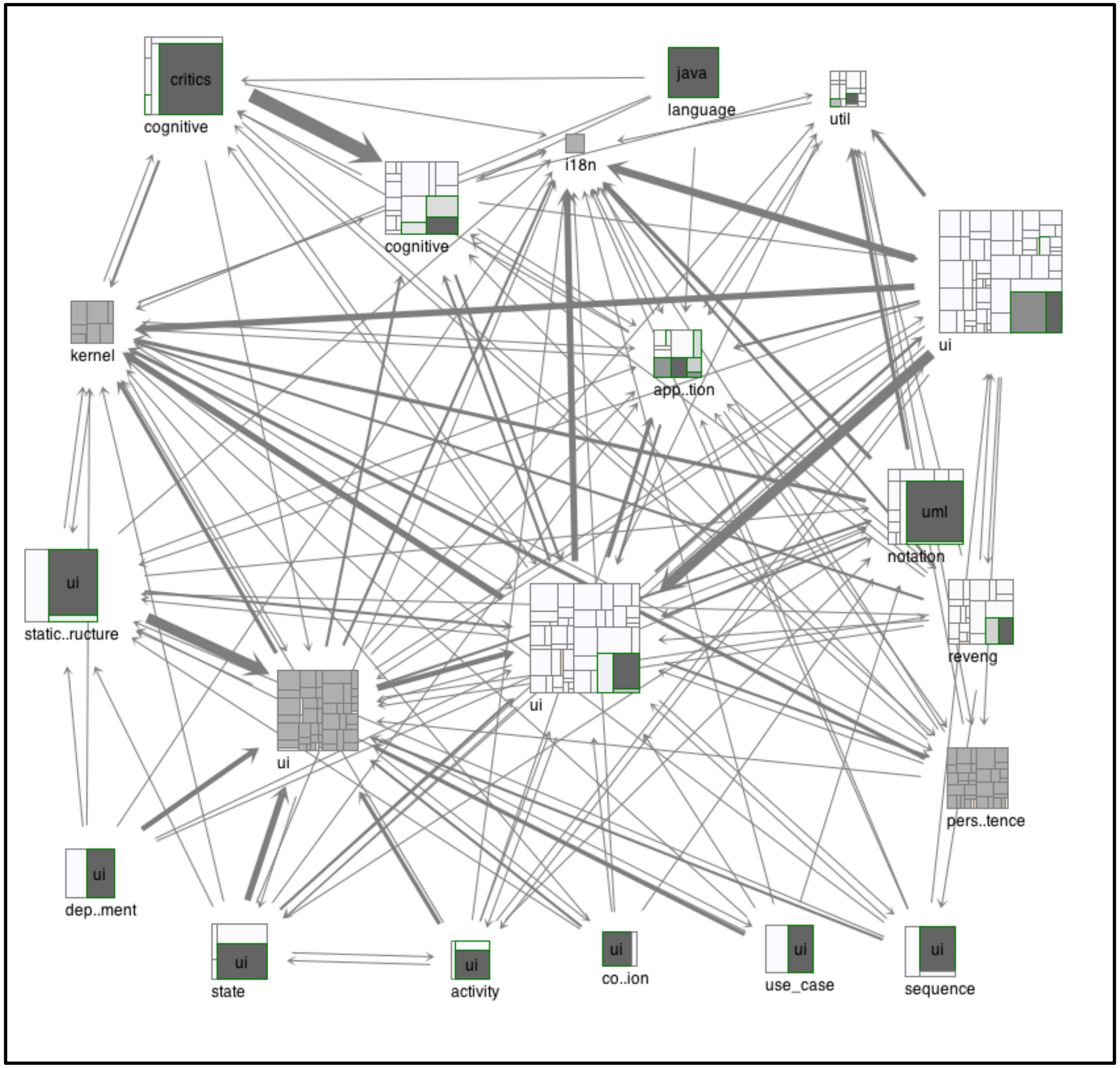
78 Nodes, 68 Edges

100%

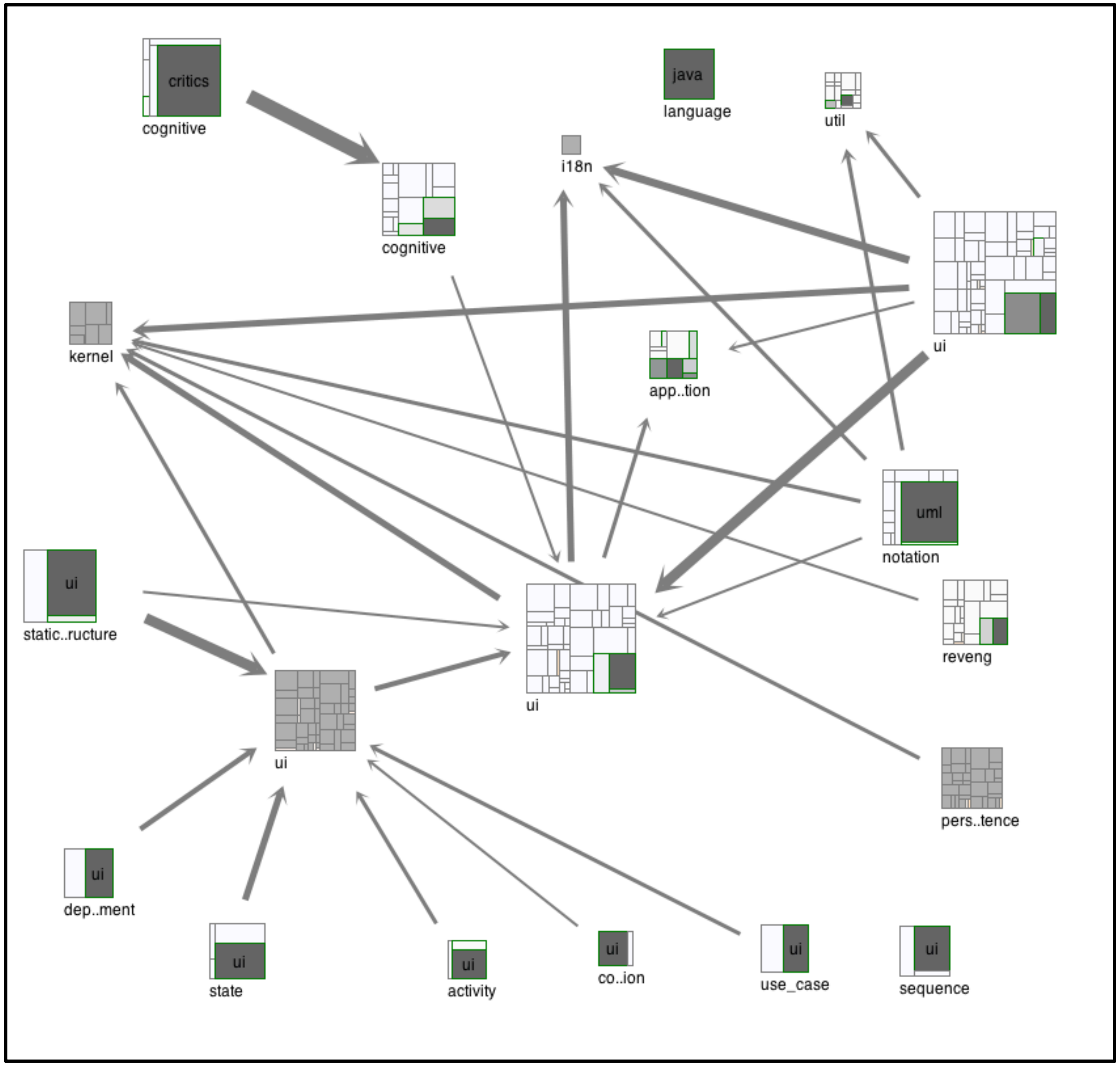
Use *simple* metrics and layout algorithms



Visualizes up to 5 metrics per node

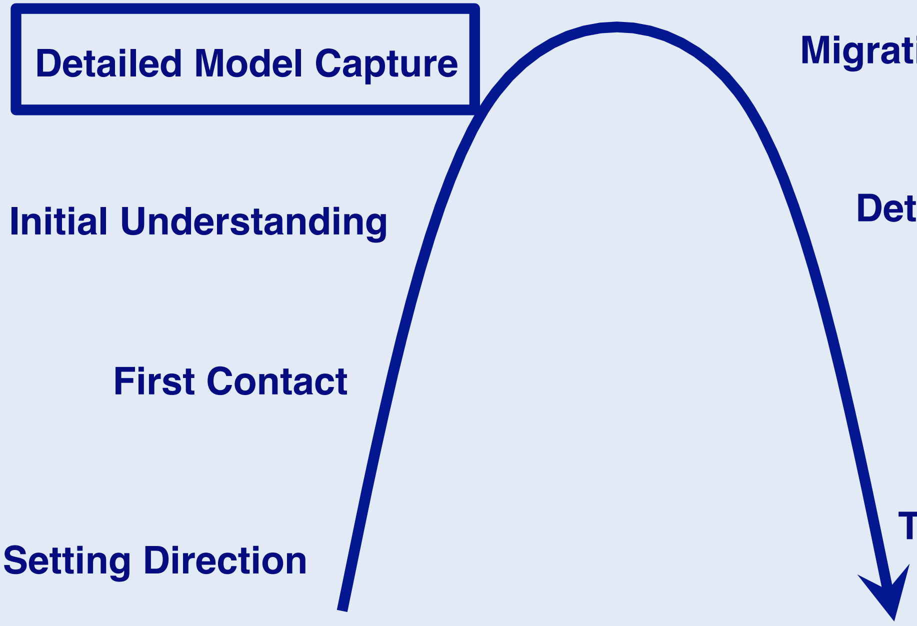

critics
cognitive
java
language
util
i18n
cognitive
ui
kernel
app..tion
uml
notation
ui
static..ructure
reveng
ui
ui
pers..tence
ui
dep..ment
ui
state
ui
activity
ui
co..ion
ui
use_case
ui
sequence

Visualizing Exceptional Relationships



Roadmap

Tests: Your Life Insurance



Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

Reengineering

Detailed Model Capture: Forces

Details matter

- Pay attention to the *details*!

There is usually a lot of data!

- How to filter what does not matter?

Design evolves

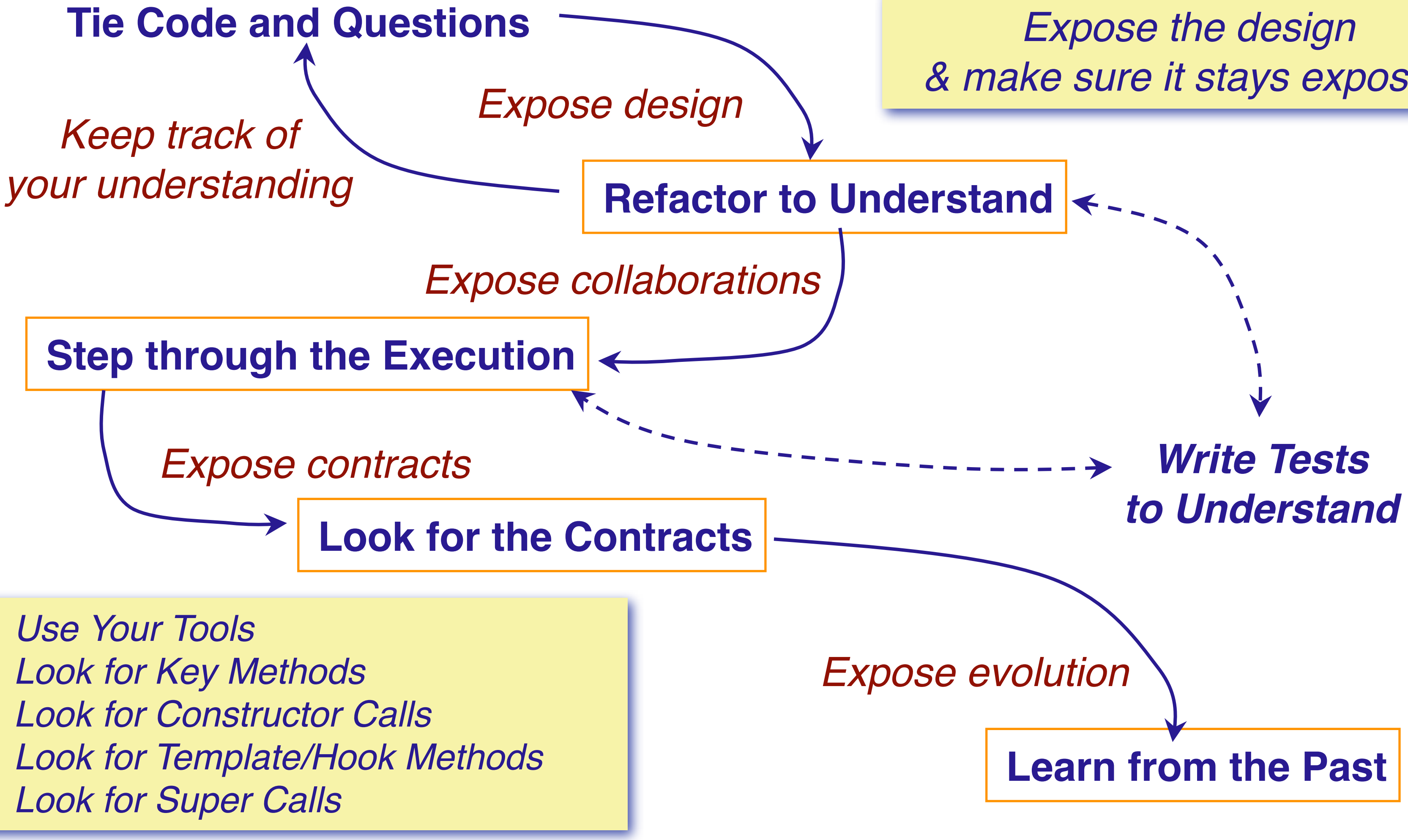
- Important issues are reflected in *changes* to the code!

Source code analysis has limitations

- Study *dynamic behaviour* to extract detailed design

Detailed Model Capture

Expose the design
& make sure it stays exposed



Tie Code and Questions

Keep track of your understanding

Expose design

Refactor to Understand

Expose collaborations

Step through the Execution

Expose contracts

Look for the Contracts

Write Tests to Understand

Expose evolution

Learn from the Past

- *Use Your Tools*
- *Look for Key Methods*
- *Look for Constructor Calls*
- *Look for Template/Hook Methods*
- *Look for Super Calls*

Refactor to Understand

Problem: How do you decipher cryptic code?

Solution: Refactor it till it makes sense

- Goal (for now) is to understand, not to reengineer
- Work with a copy of the code
- Refactoring requires an adequate test base
  - If this is missing, Write Tests to Understand

Hints:

- Rename attributes to convey roles
- Rename methods and classes to reveal intent
- Remove duplicated code
- Replace condition branches by methods

http://objectmentor.com/resources/articles/Naming.pdf

Look for the Contracts

Problem: How to understand a class?

Solution: Look for common programming idioms

- Look for "*key methods*"
  - Intention-revealing names
  - Key parameter types
  - Recurring parameter types represent temporary associations
- Look for *constructor* calls
- Look for *Template/Hook* methods
- Look for *super* calls
- *Use your tools!*

Learn from the Past

Problem: How did the system get the way it is?

Solution: Compare versions to discover where code was <u>removed</u>

- *Removed* functionality is a sign of design evolution
- Use or develop appropriate *tools*
- Look for signs of:
  - *Unstable design* — repeated growth and refactoring
  - *Mature design* — growth, refactoring and stability

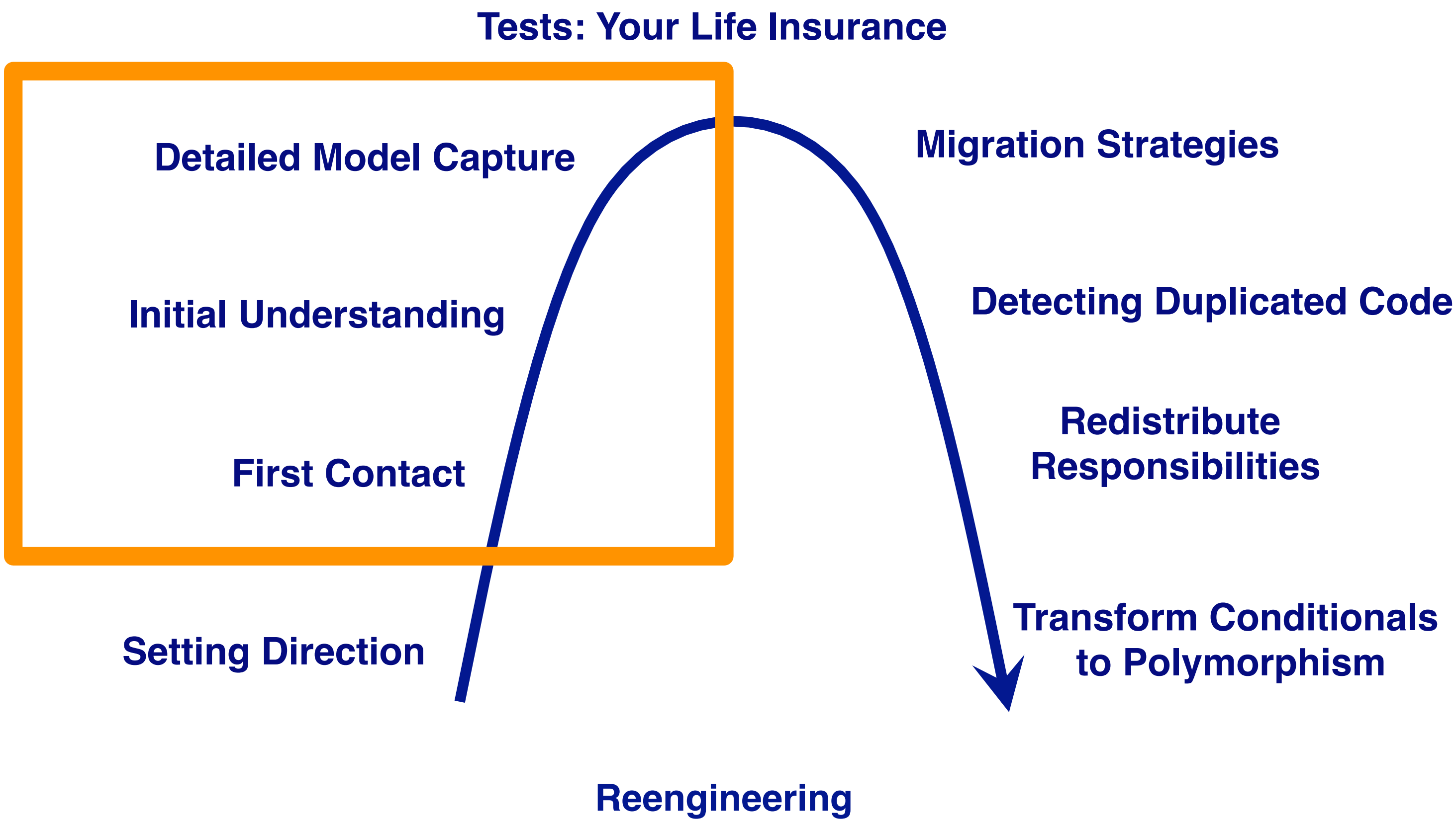
Step Through the Execution

Problem: How do you uncover the run-time architecture?

Solution: Execute scenarios of known use cases and step through the code with a debugger

- Tests can also be used as scenario generators
  - If tests are missing *Write Tests to Understand*
- Difficulties
  - OO source code exposes a *class hierarchy*, not the run-time *object collaborations*
  - Collaborations are spread throughout the code
  - Polymorphism may hide which classes are instantiated
- Focused use of a debugger can expose collaborations

Source Code is Data!



Tests: Your Life Insurance

Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

Redistribute
Responsibilities

Transform Conditionals
to Polymorphism

Reengineering

What you should know!

- What is the difference between reengineering, reverse engineering, and forward engineering.
- Be able to ennumerate and talk about several of the reengineering patterns.
- Source code is also data!