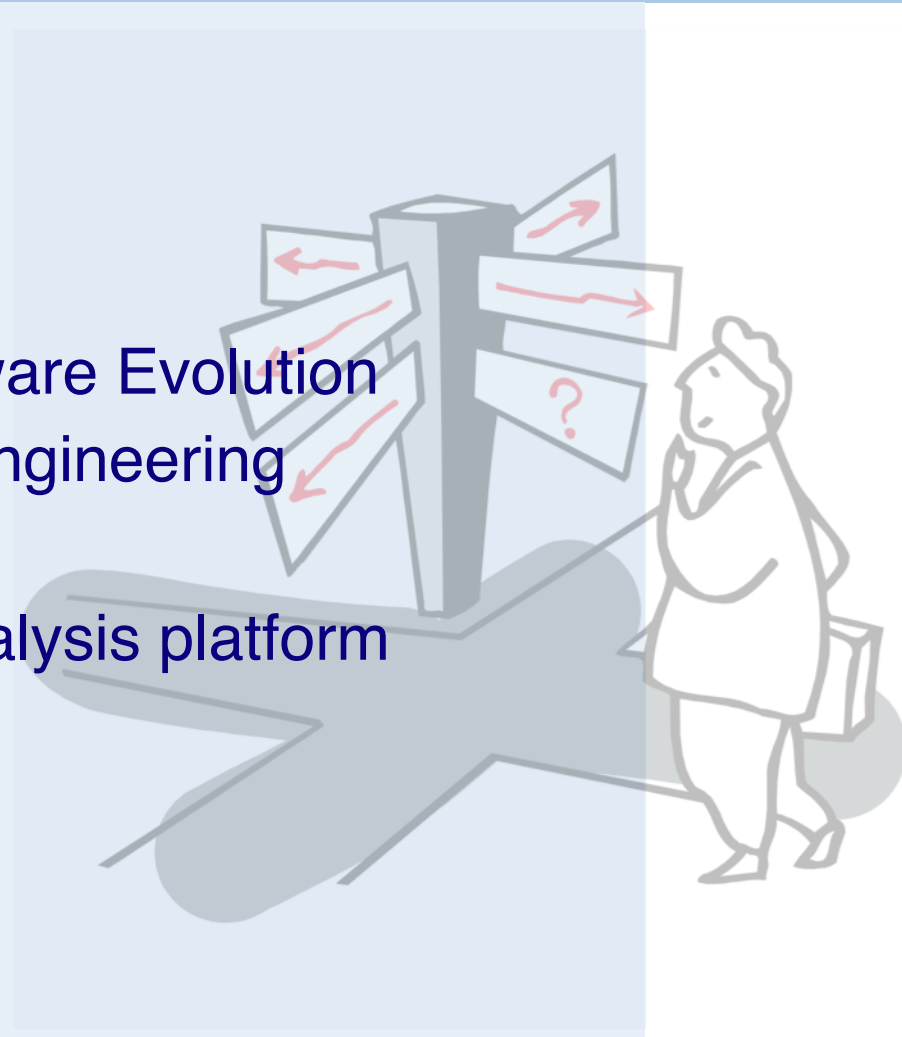


# Introduction to Software Engineering

## 13. Software Evolution

# Roadmap

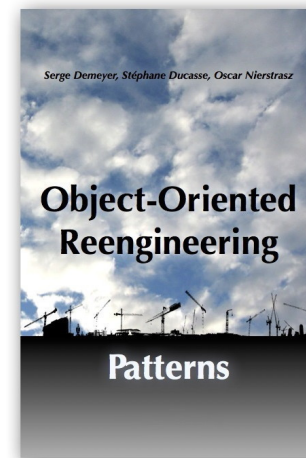
- > Lehman's Laws of Software Evolution
- > Forward and Reverse Engineering
- > Reengineering Patterns
- > The Moose software analysis platform



# Literature

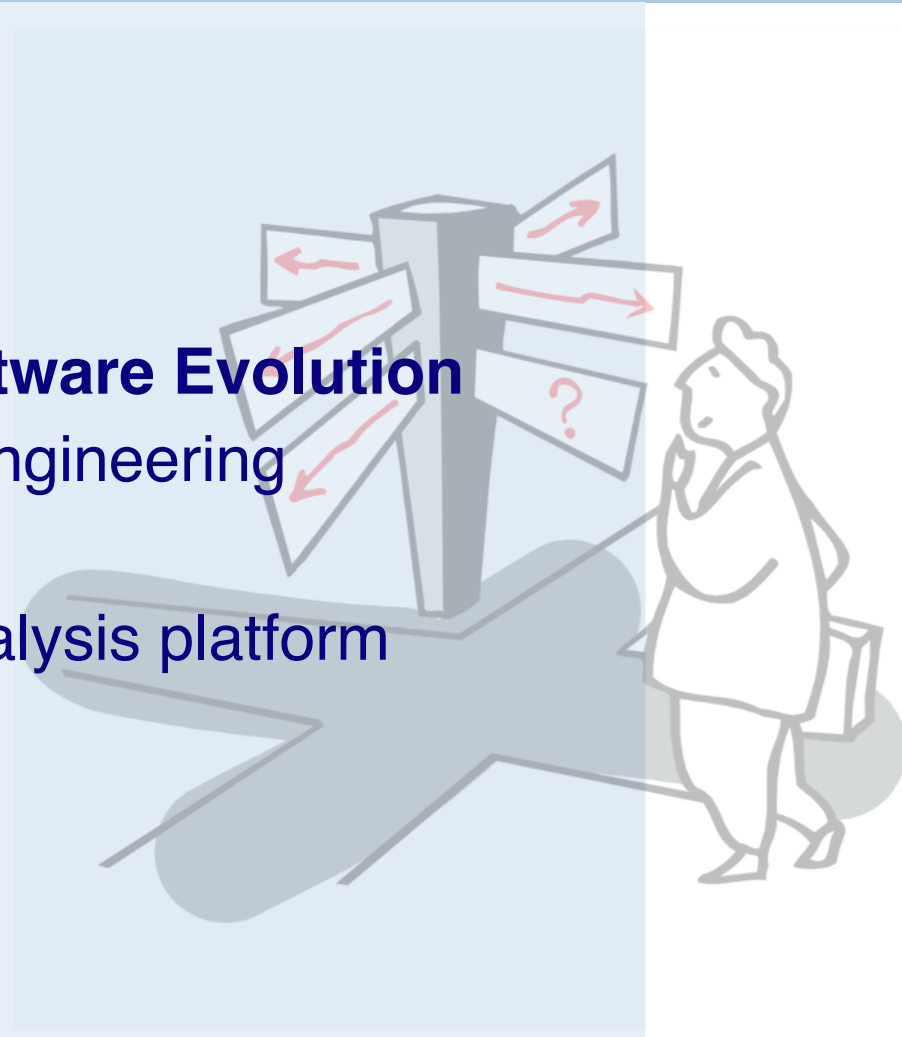
- > Demeyer, Ducasse, and Nierstrasz. *Object-Oriented Reengineering Patterns*, Square Bracket Associates, 2008.

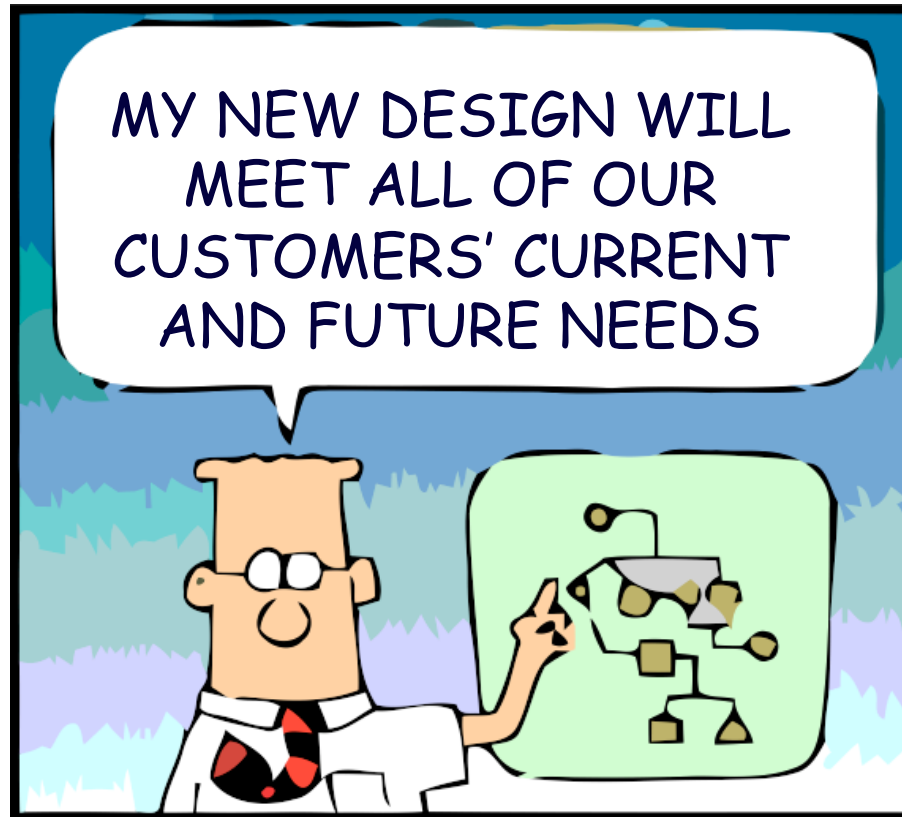
<http://scg.unibe.ch/download/oorp/>



# Roadmap

- > **Lehman's Laws of Software Evolution**
- > Forward and Reverse Engineering
- > Reengineering Patterns
- > The Moose software analysis platform





*What is wrong with this picture?*

# Lehman's Laws

## ***Continuing change***

- A program that is used in a real-world environment ***must change***, or become progressively less useful in that environment.

## ***Increasing complexity***

- As a program evolves, it becomes ***more complex***, and extra resources are needed to preserve and simplify its structure.

Lehman, Belady. *Program Evolution: Processes of Software Change*, London Academic Press, London, 1985

# The Dilemma of Legacy Software

A legacy system is a piece of software that:

- you have *inherited*, and
- is *valuable* to you.

***You can't afford to throw it out, but it is too expensive to change***

## Symptoms

- Loss of *knowledge*
- Architecture & design *drift*
- Hard to make *changes*
- ...



# OO Legacy

- > Object-oriented legacy systems are *successful* OO systems whose architecture and design *no longer respond to changing requirements*



# Common Symptoms

## Lack of Knowledge

- > *obsolete* or no documentation
- > *departure* of the original developers or users
- > *disappearance of inside knowledge* about the system
- > *limited understanding* of entire system  
⇒ *missing tests*

## Process symptoms

- > *too long* to turn things over to production
- > need for *constant bug fixes*
- > *maintenance dependencies*
- > *difficulties separating products*  
⇒ *simple changes take too long*

## Code symptoms

- *duplicated code*
- *code smells*  
⇒ *big build times*

# Common Problems

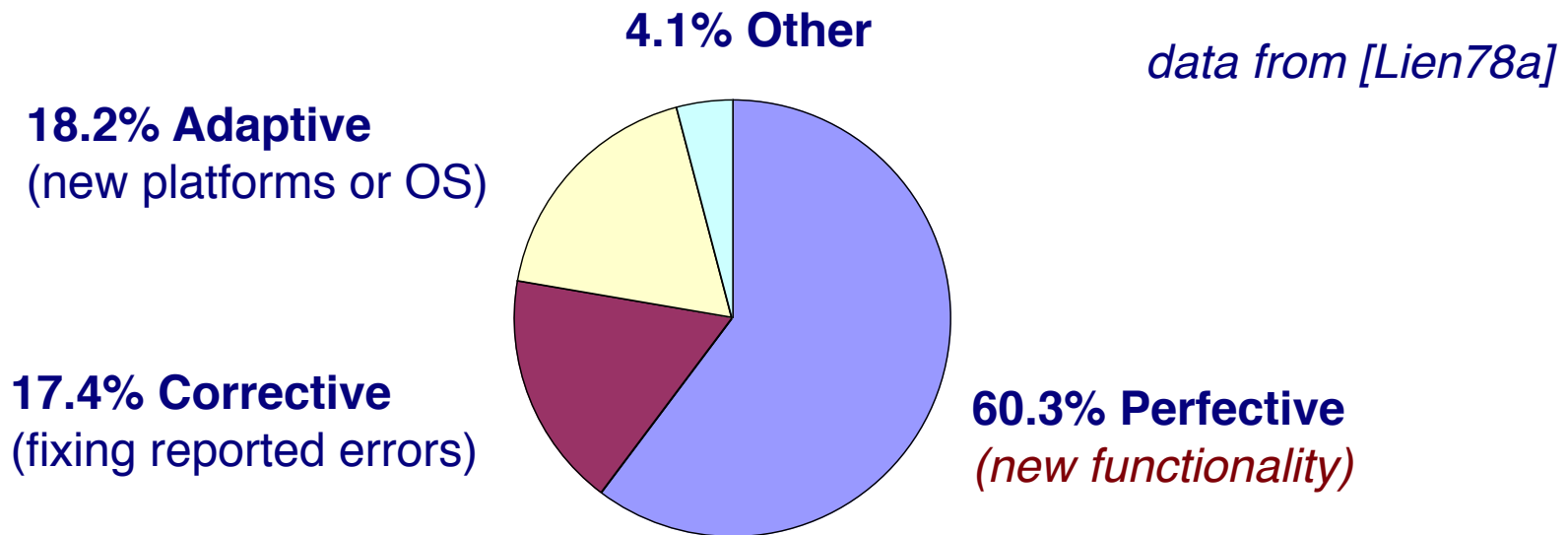
## Architectural Problems

- > insufficient *documentation*  
= non-existent or out-of-date
- > improper *layering*  
= too few or too many layers
- > lack of *modularity*  
= strong coupling
- > *duplicated code*  
= copy, paste & edit code
- > duplicated *functionality*  
= similar functionality  
by separate teams

## Refactoring opportunities

- > *misuse* of inheritance  
= code reuse vs polymorphism
- > *missing* inheritance  
= duplication, case-statements
- > *misplaced* operations  
= operations outside classes
- > *violation* of encapsulation  
= type-casting; C++ "friends"
- > *class abuse*  
= classes as namespaces

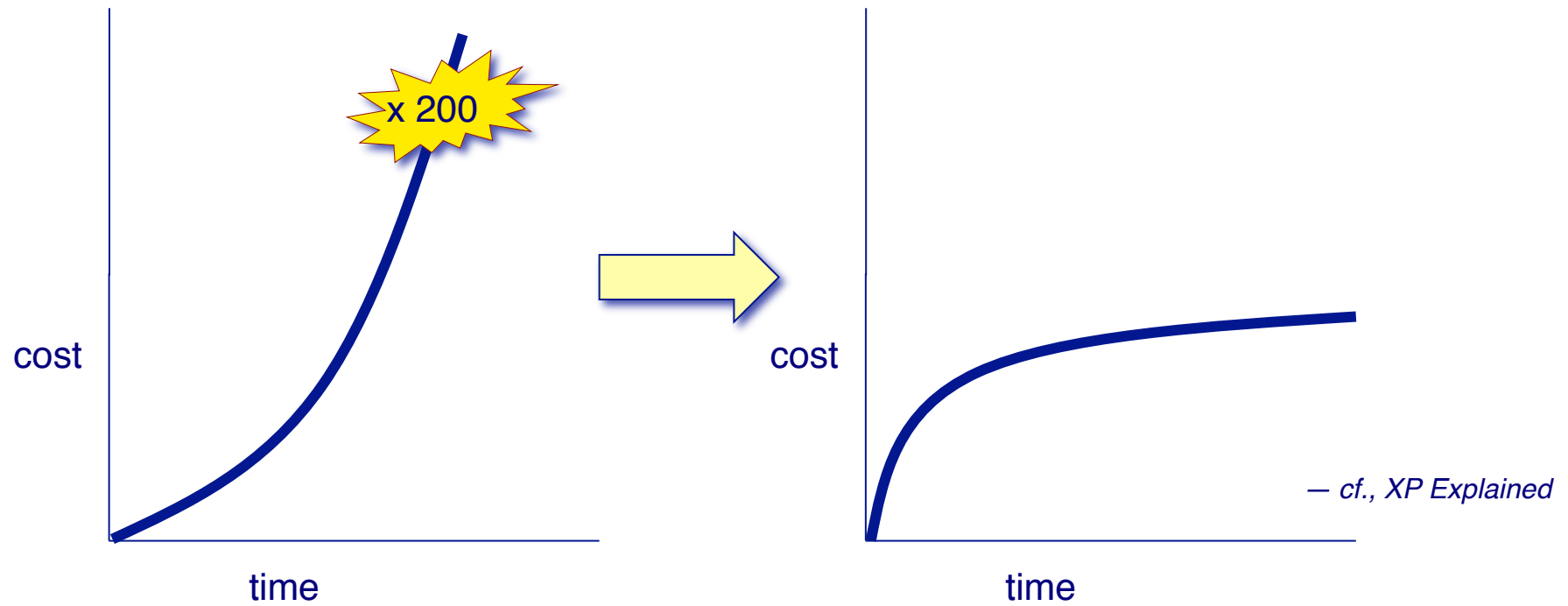
# Continuous Development



The bulk of the maintenance cost is due to *new functionality*  
⇒ even with better requirements, it is hard to predict new functions

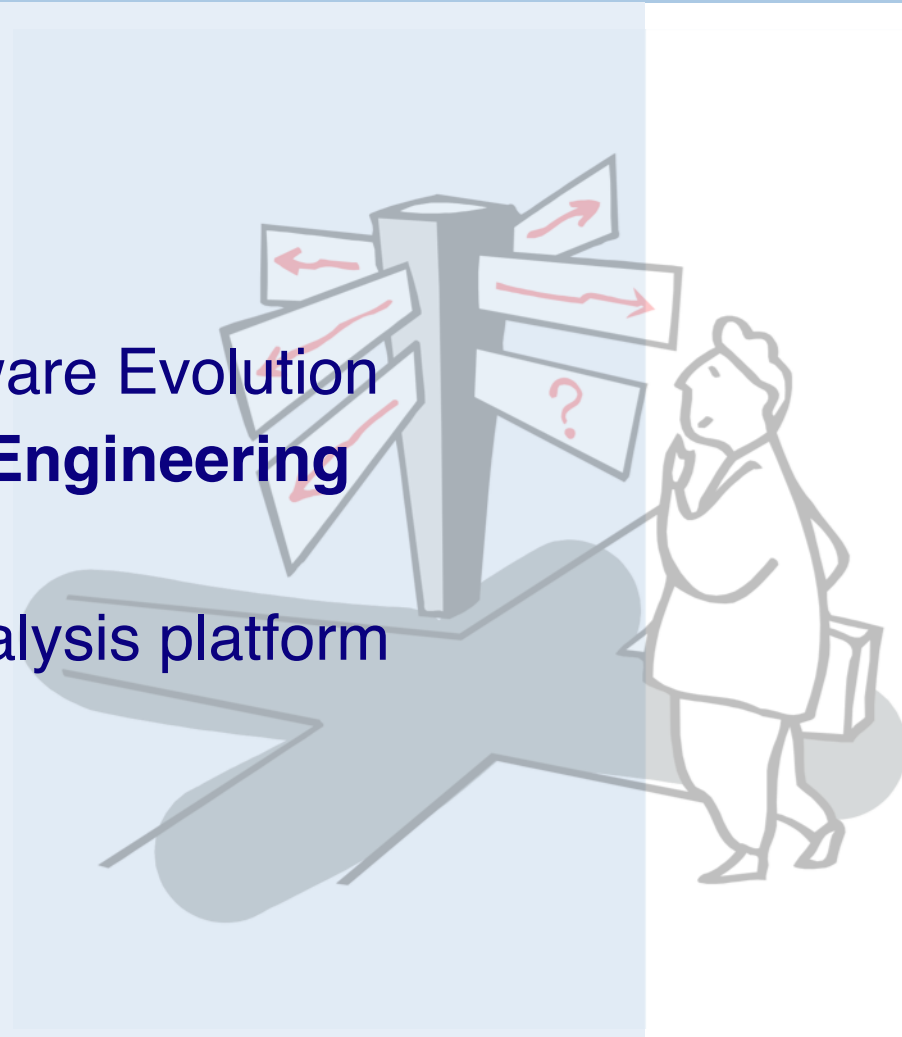
# The cost of change

*We need to reduce the cost of change over time ...*



# Roadmap

- > Lehman's Laws of Software Evolution
- > **Forward and Reverse Engineering**
- > Reengineering Patterns
- > The Moose software analysis platform



## Some Terminology

“**Forward Engineering** is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.”

“**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.”

“**Reengineering** ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.”

— *Chikofsky and Cross [in Arnold, 1993]*

# Goals of Reverse Engineering

- > Cope with *complexity*
  - need techniques to understand large, complex systems
- > Generate *alternative views*
  - automatically generate different ways to view systems
- > Recover *lost information*
  - extract what changes have been made and why
- > Detect *side effects*
  - help understand ramifications of changes
- > Synthesize *higher abstractions*
  - identify latent abstractions in software
- > Facilitate *reuse*
  - detect candidate reusable artifacts and components

— Chikofsky and Cross [in Arnold, 1993]

# Reverse Engineering Techniques

- > *Redocumentation*
  - pretty printers
  - diagram generators
  - cross-reference listing generators
  
- > *Design recovery*
  - software metrics
  - browsers, visualization tools
  - static analyzers
  - dynamic (trace) analyzers



# Goals of Reengineering

- > *Unbundling*
  - split a monolithic system into parts that can be separately marketed
- > *Performance*
  - “first do it, then do it right, then do it fast” — experience shows this is the right sequence!
- > *Port to other Platform*
  - the architecture must distinguish the platform dependent modules
- > *Design extraction*
  - to improve maintainability, portability, etc.
- > *Exploitation of New Technology*
  - i.e., new language features, standards, libraries, etc.

# Reengineering Techniques

> *Restructuring*

- automatic conversion from unstructured to structured code
- source code translation

— *Chikofsky and Cross*

> *Data reengineering*

- integrating and centralizing multiple databases
- unifying multiple, inconsistent representations
- upgrading data models

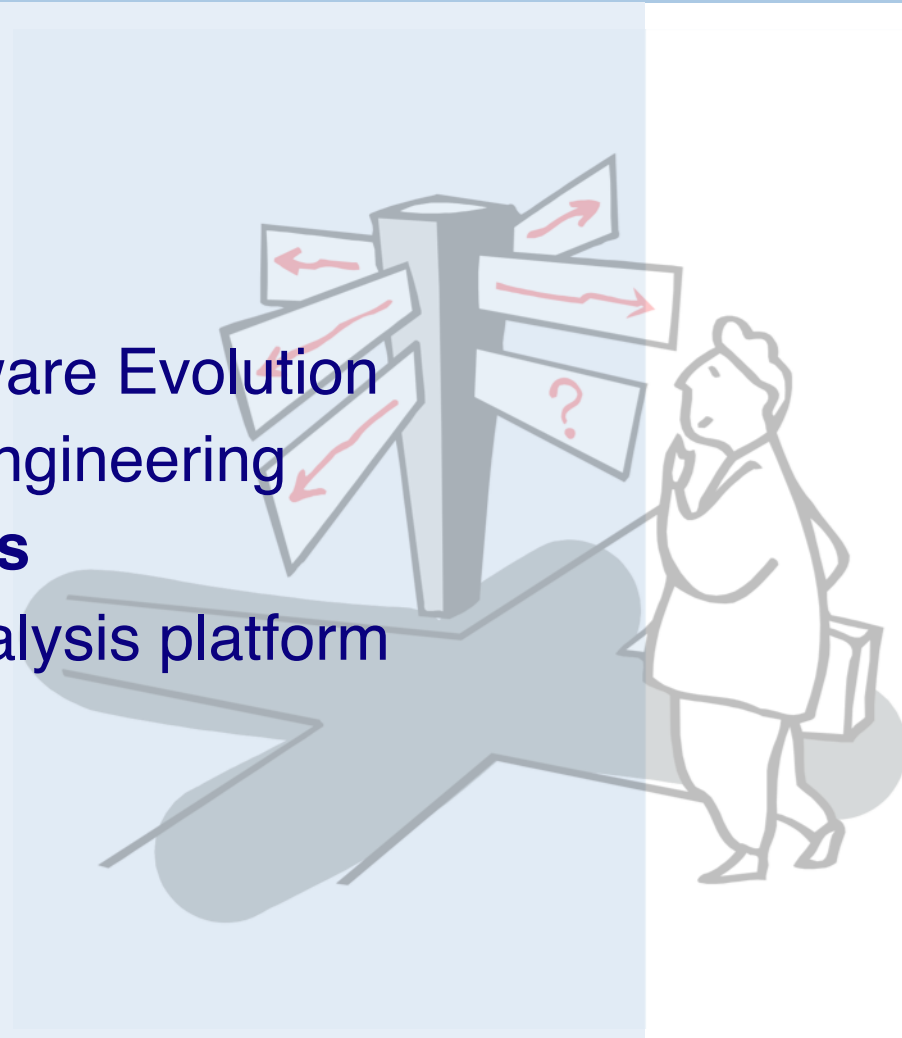
— *Sommerville, ch 32*

> *Refactoring*

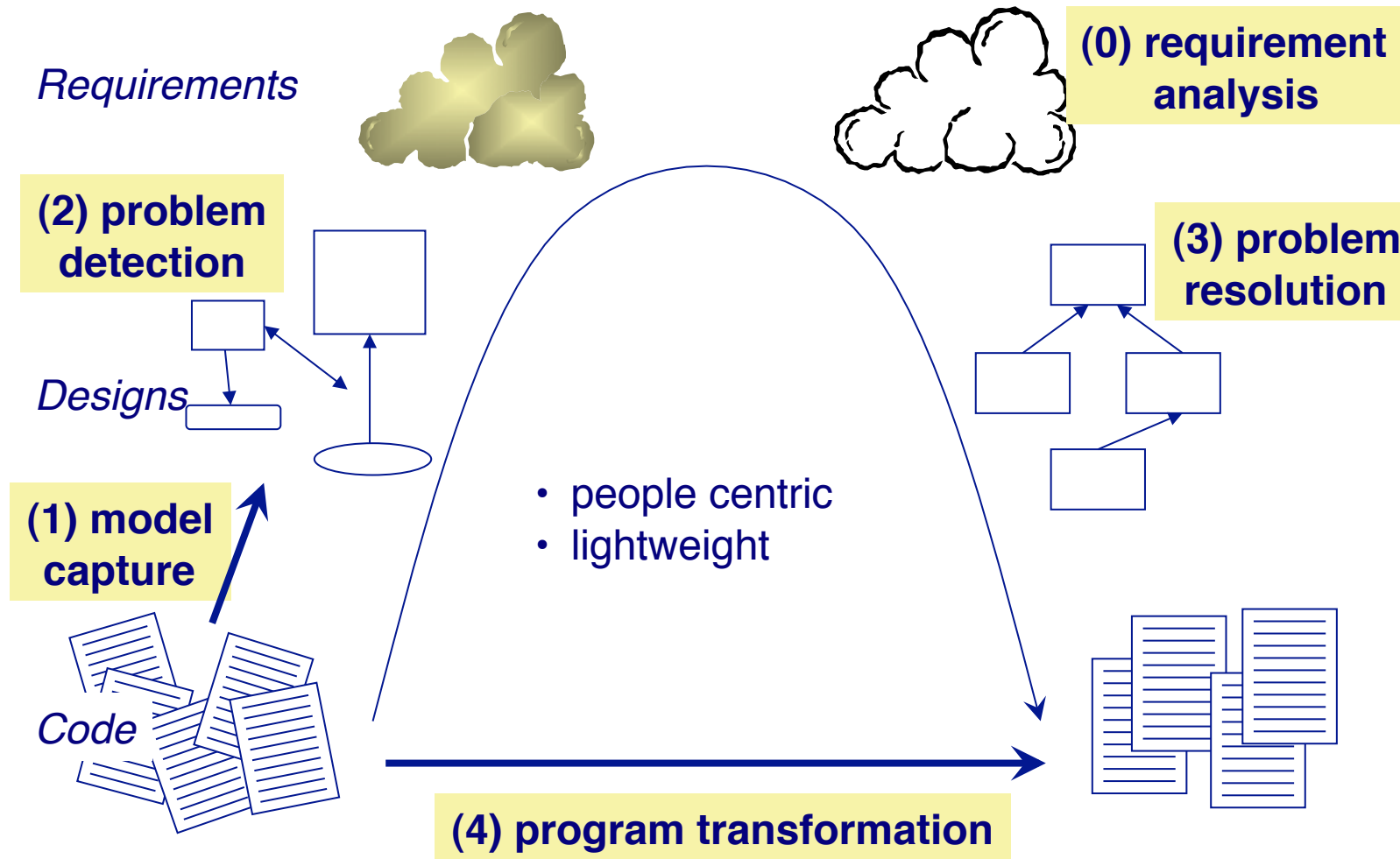
- renaming/moving methods/classes etc.

# Roadmap

- > Lehman's Laws of Software Evolution
- > Forward and Reverse Engineering
- > **Reengineering Patterns**
- > The Moose software analysis platform



# The Reengineering Life-Cycle



# Reengineering Patterns

In software engineering, a **design pattern** is a general solution to a common problem in software design. *A design pattern isn't a finished design that can be transformed directly into code; it is a description or template for how to solve a problem that can be used in many different situations.*



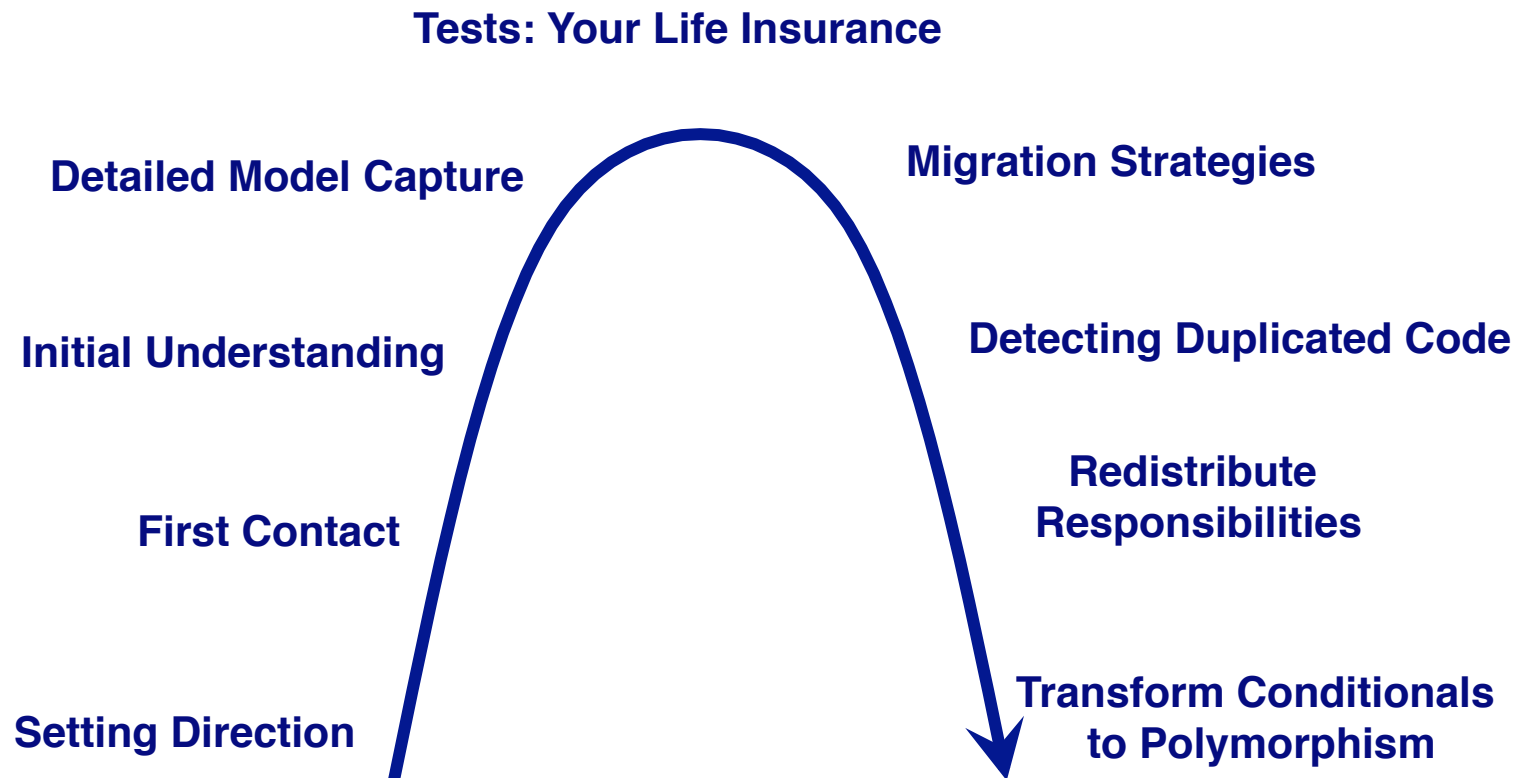
Reverse engineering patterns *encode expertise and trade-offs* in *extracting design* from source code, running systems and people.

*—Even if design documents exist, they are typically out of sync with reality.*

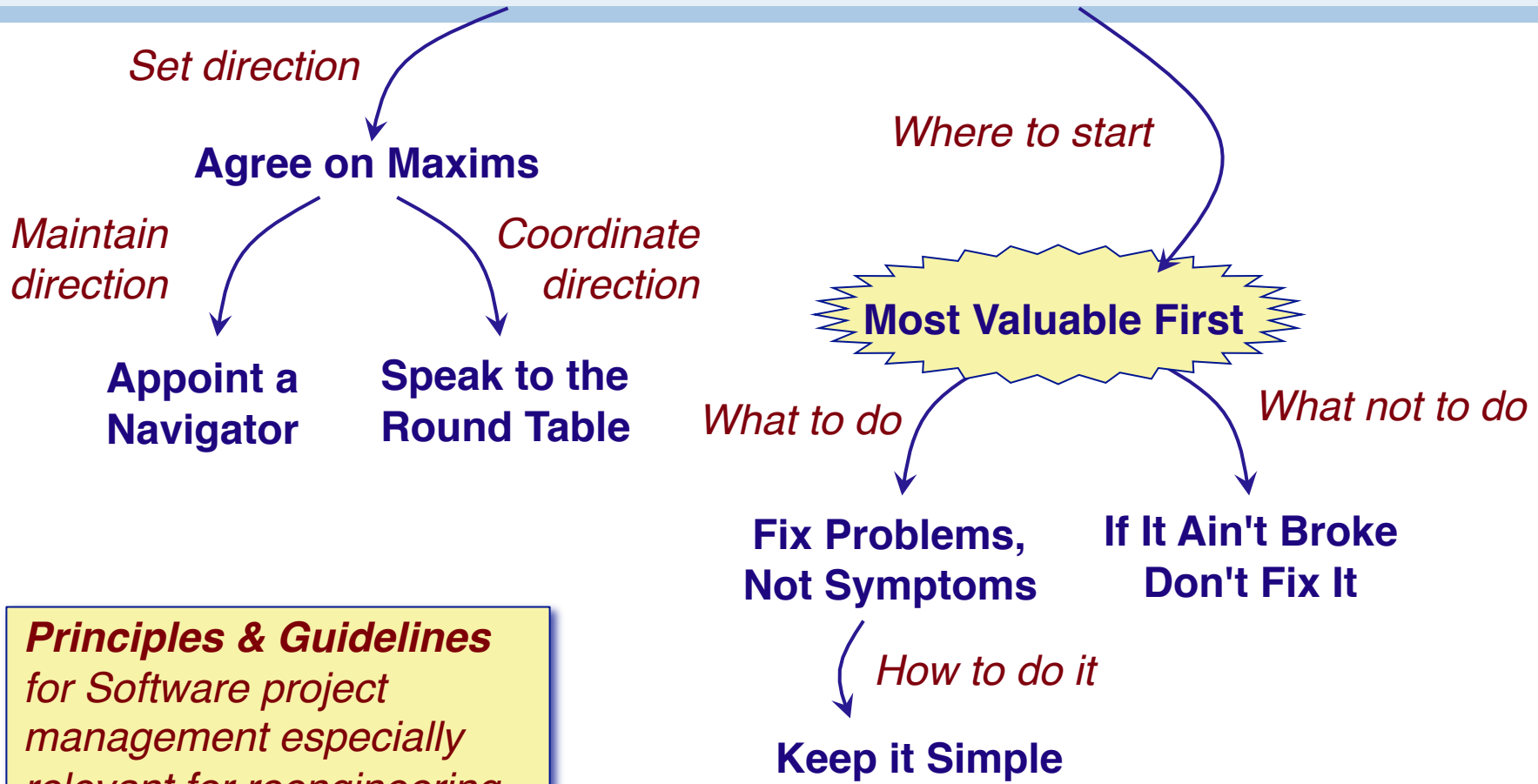
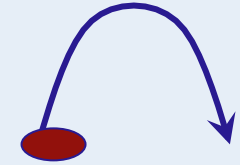
Reengineering patterns encode expertise and trade-offs in *transforming legacy code* to resolve problems that have emerged.

*—These problems are typically not apparent in original design but are due to architectural drift as requirements evolve*

# A Map of Reengineering Patterns



# Setting Direction



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

## Most Valuable First

**Problem:** *Which problems should you focus on first?*

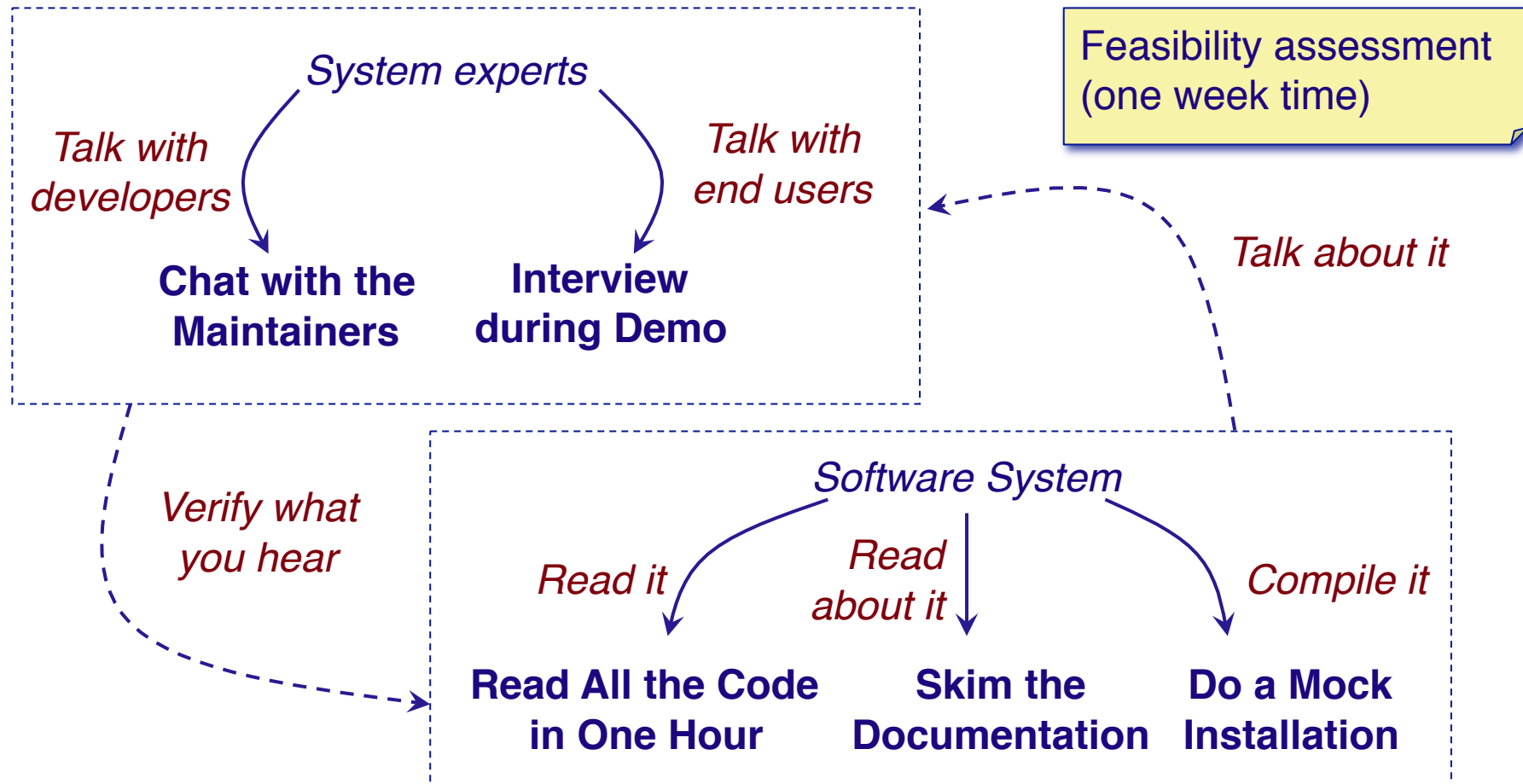
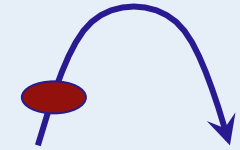
**Solution:** Work on aspects that are *most valuable* to your customer

- > Maximize commitment, early results; build confidence
- > Difficulties and hints:
  - Which *stakeholder* do you listen to?
  - What *measurable goal* to aim for?
  - Consult *change logs* for high activity
  - Play the *Planning Game*
  - Wrap, refactor or rewrite? — *Fix Problems, not Symptoms*

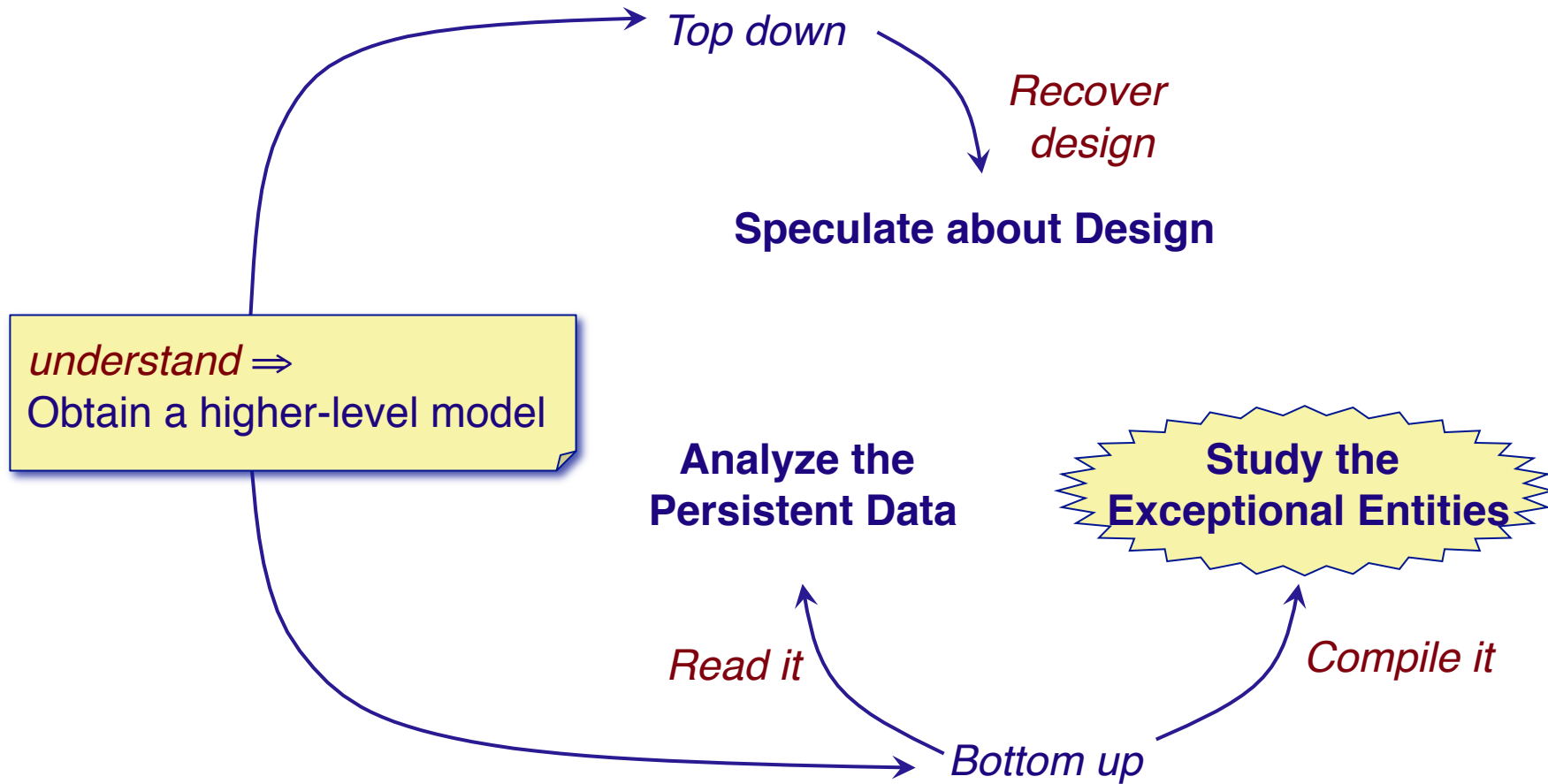
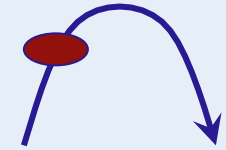
*Vs. Fix the  
Buggiest First?*



# First Contact



# Initial Understanding



# Pattern: Study the Exceptional Entities

## ***Problem***

- How can you quickly gain insight into complex software?

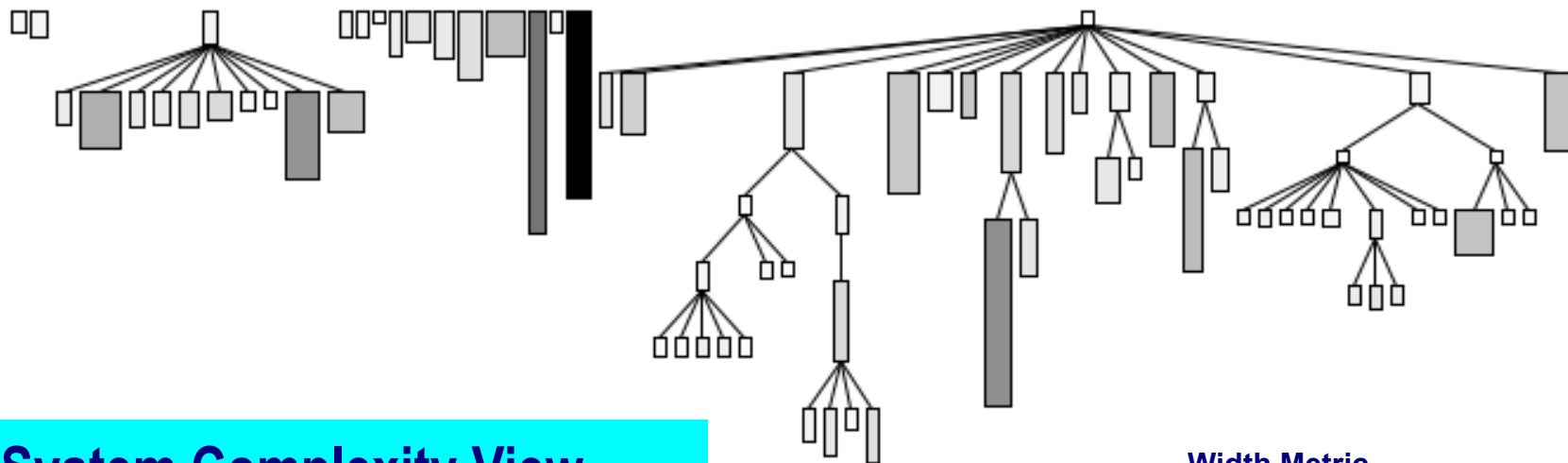
## ***Solution***

- *Measure* software entities and *study the anomalous ones*

## ***Steps***

- Use simple metrics
- Visualize metrics to get an overview
- Browse the code to get insight into the anomalies

# System Complexity View



## System Complexity View

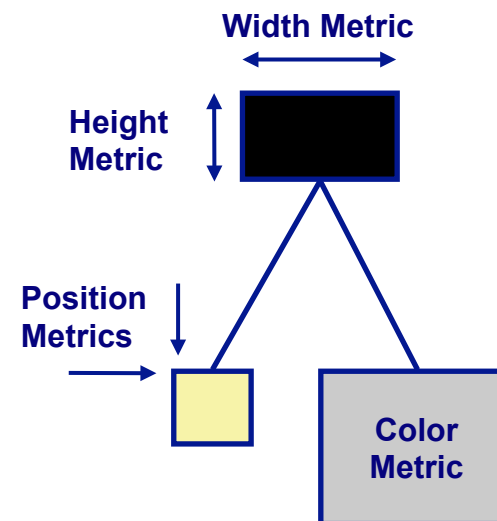
Nodes = Classes

Edges = Inheritance Relationships

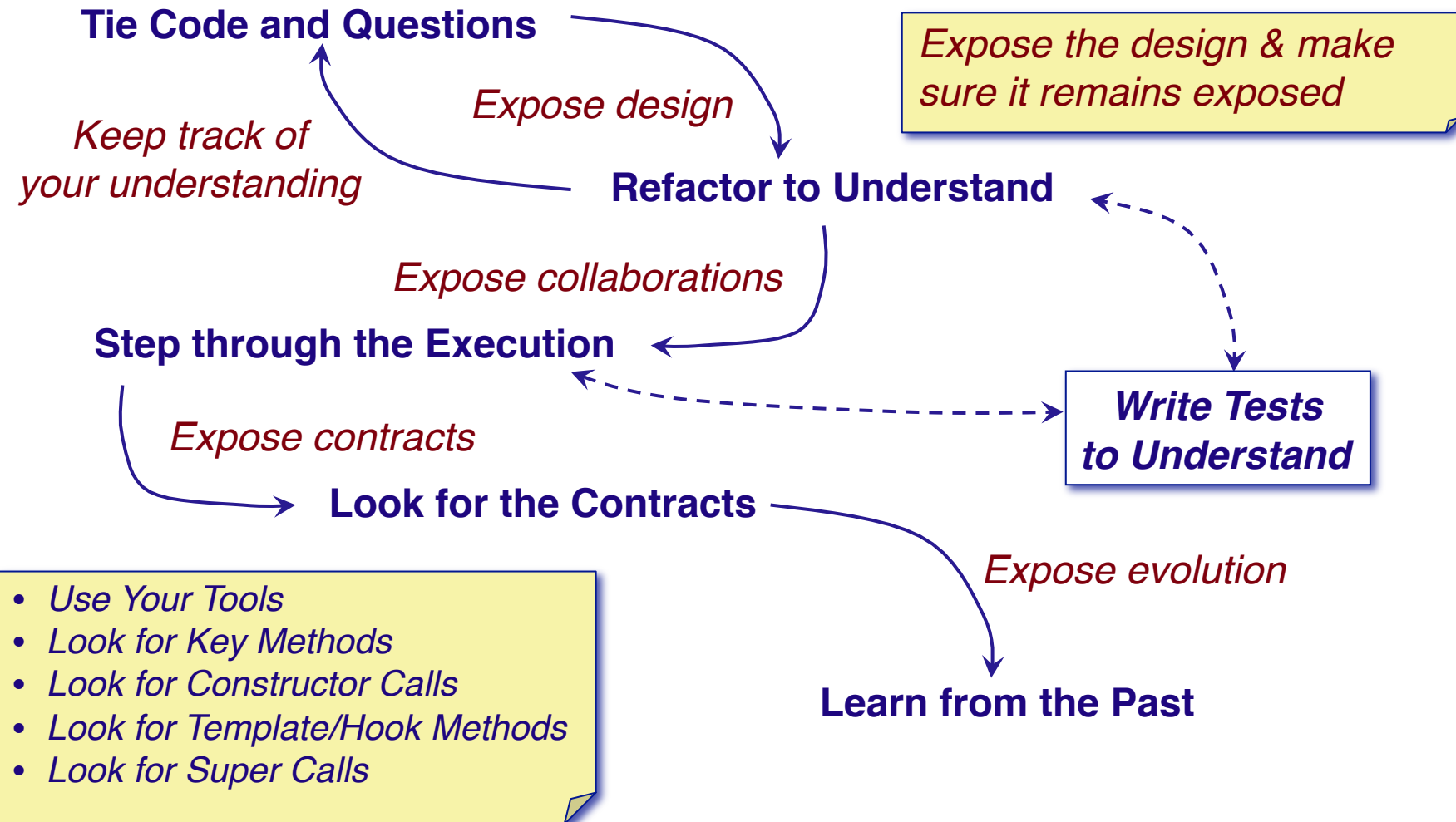
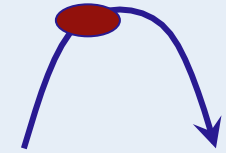
Width = Number of Attributes

Height = Number of Methods

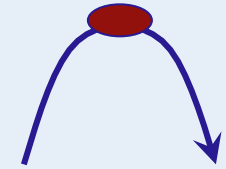
Color = Number of Lines of Code



# Detailed Model Capture



# Tests: Your Life Insurance



## Write Tests to Enable Evolution

*Managing tests*

**Grow Your Test Base Incrementally**

**Write Tests to Understand**

**Regression Test after Every Change**

## Use a Testing Framework

*Designing tests*

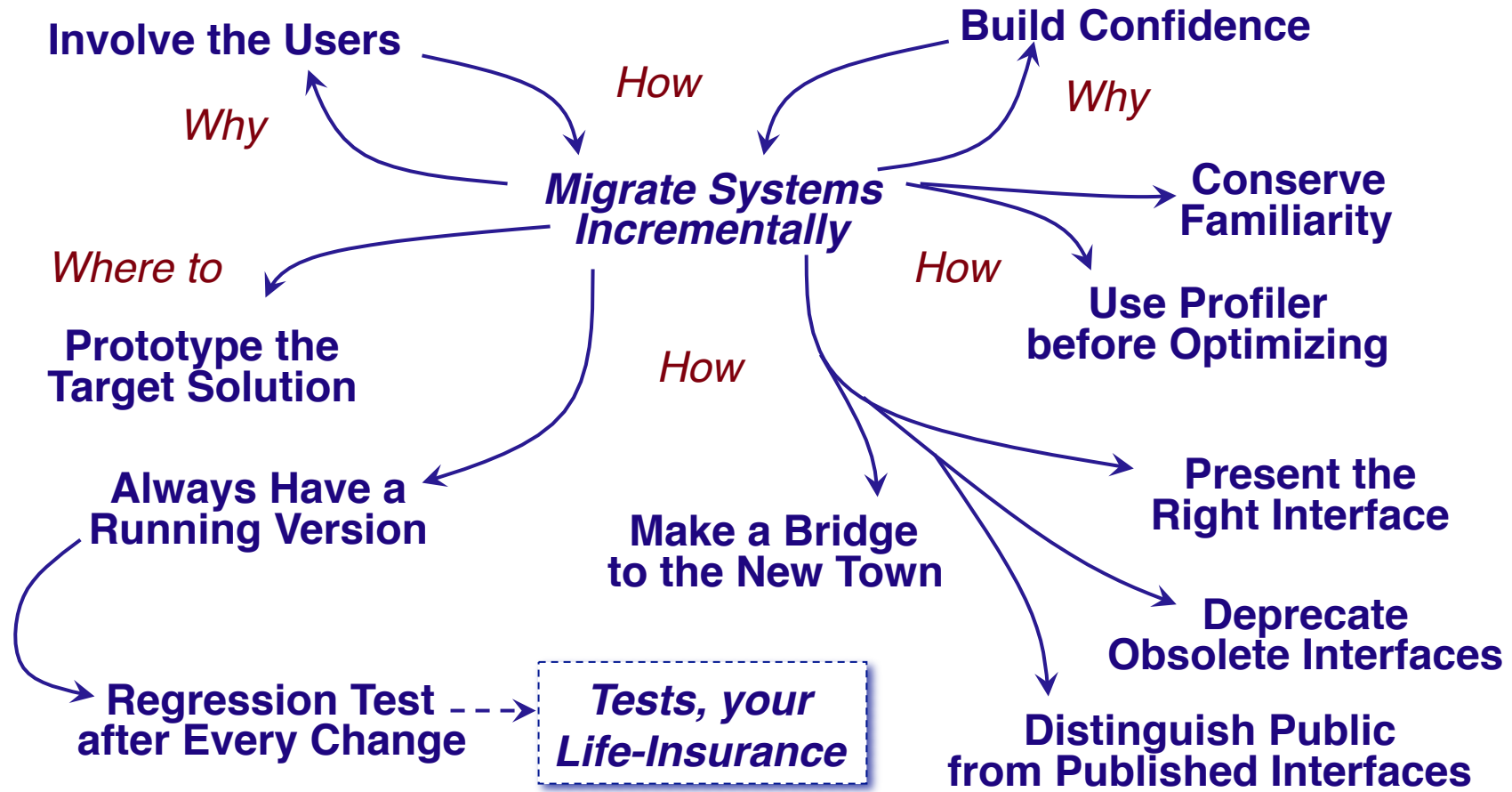
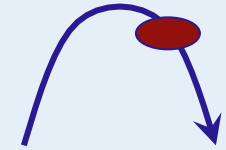
**Test the Interface, Not the Implementation**

**Record Business Rules as Tests**

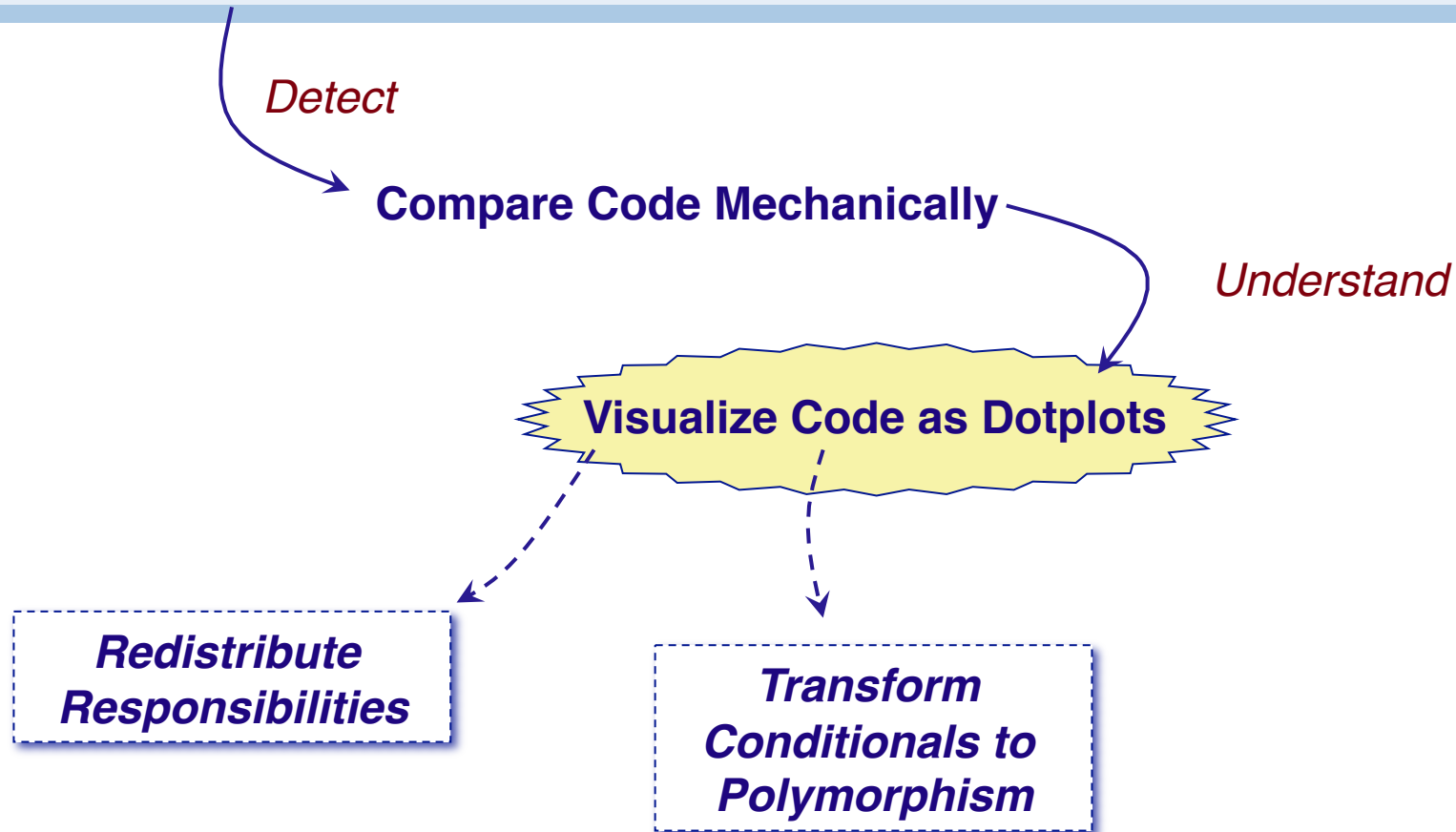
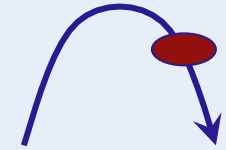
- *Test Fuzzy features*
- *Test Old Bugs*
- *Retest Persistent Problems*

**Migration Strategies**

# Migration



# Detecting Duplicated Code





# Pattern: Visualize Code as Dotplots

## ***Problem***

- How can you effectively identify significant duplication in a complex software system?

## ***Solution***

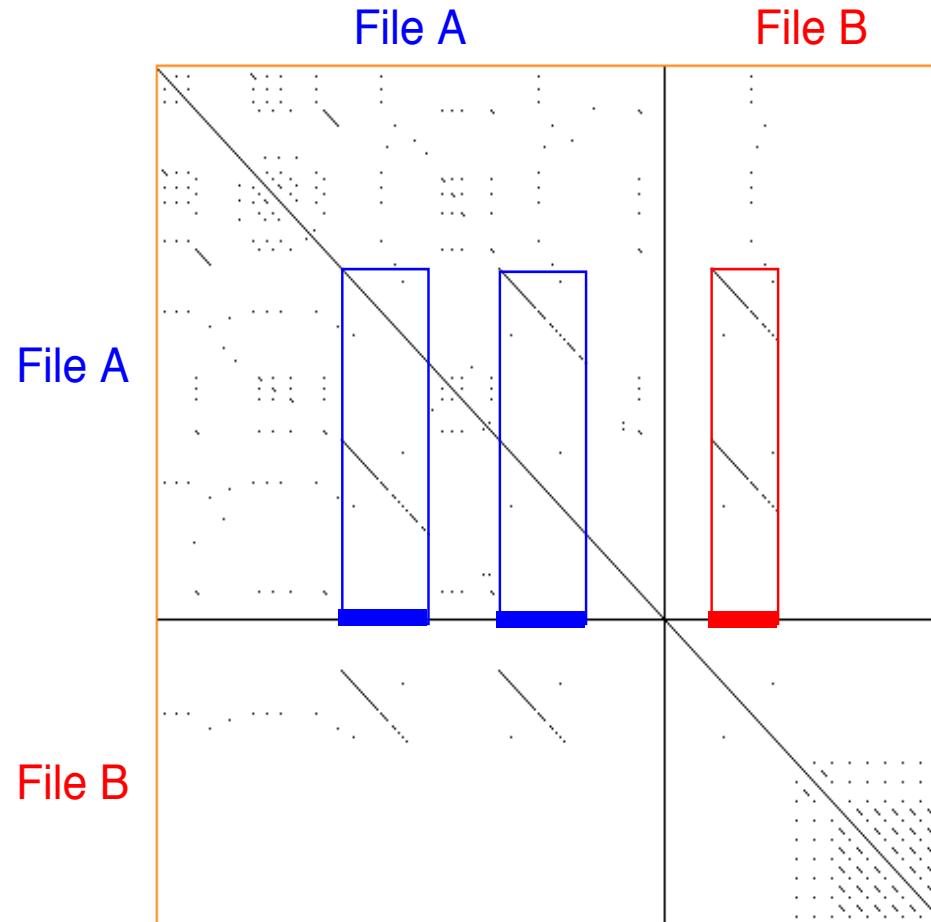
- Visualize the code as a *dotplot*, where dots represent duplication.

## ***Steps***

- Normalize the source files
- Compare files line-by-line
- Visualize and interpret the dotplots

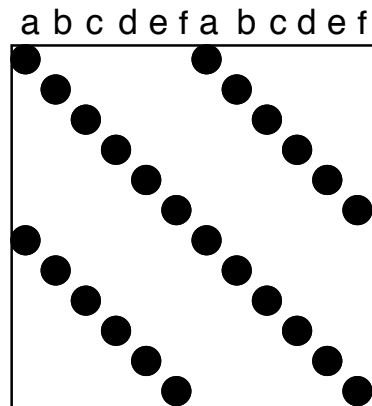
# Clone detection by string-matching

*Solid diagonals* indicate significant duplication between or within source files.

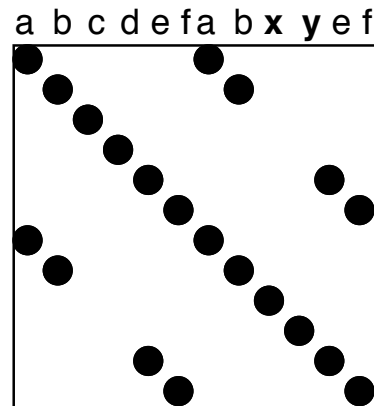


# Dotplot Visualization

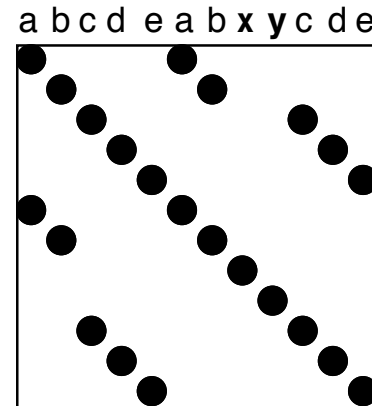
## Sample Dot Configurations:



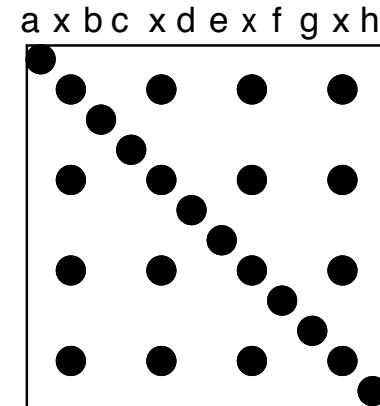
Exact Copies



Copies with Variations



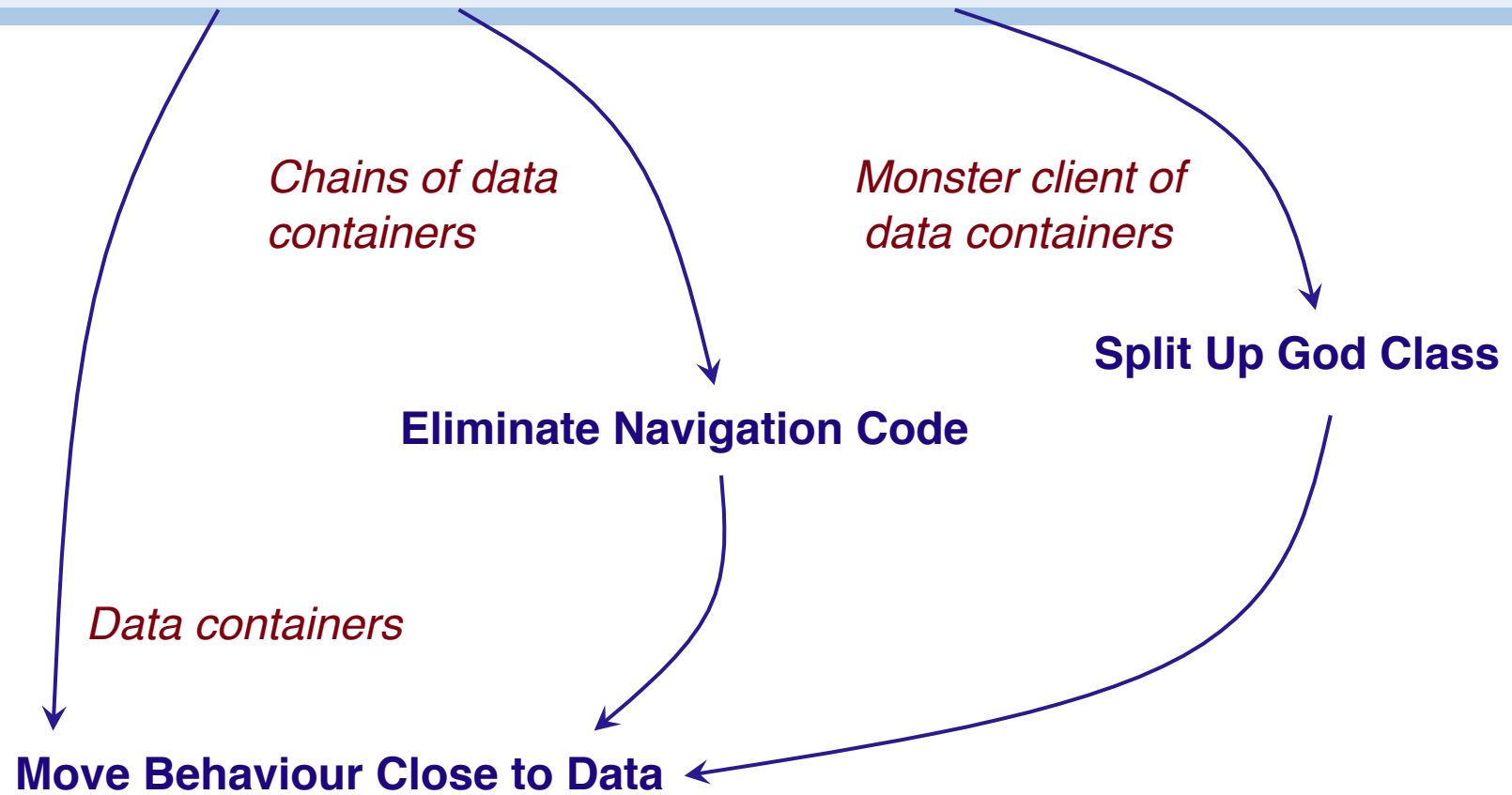
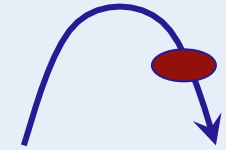
Inserts/Deletes



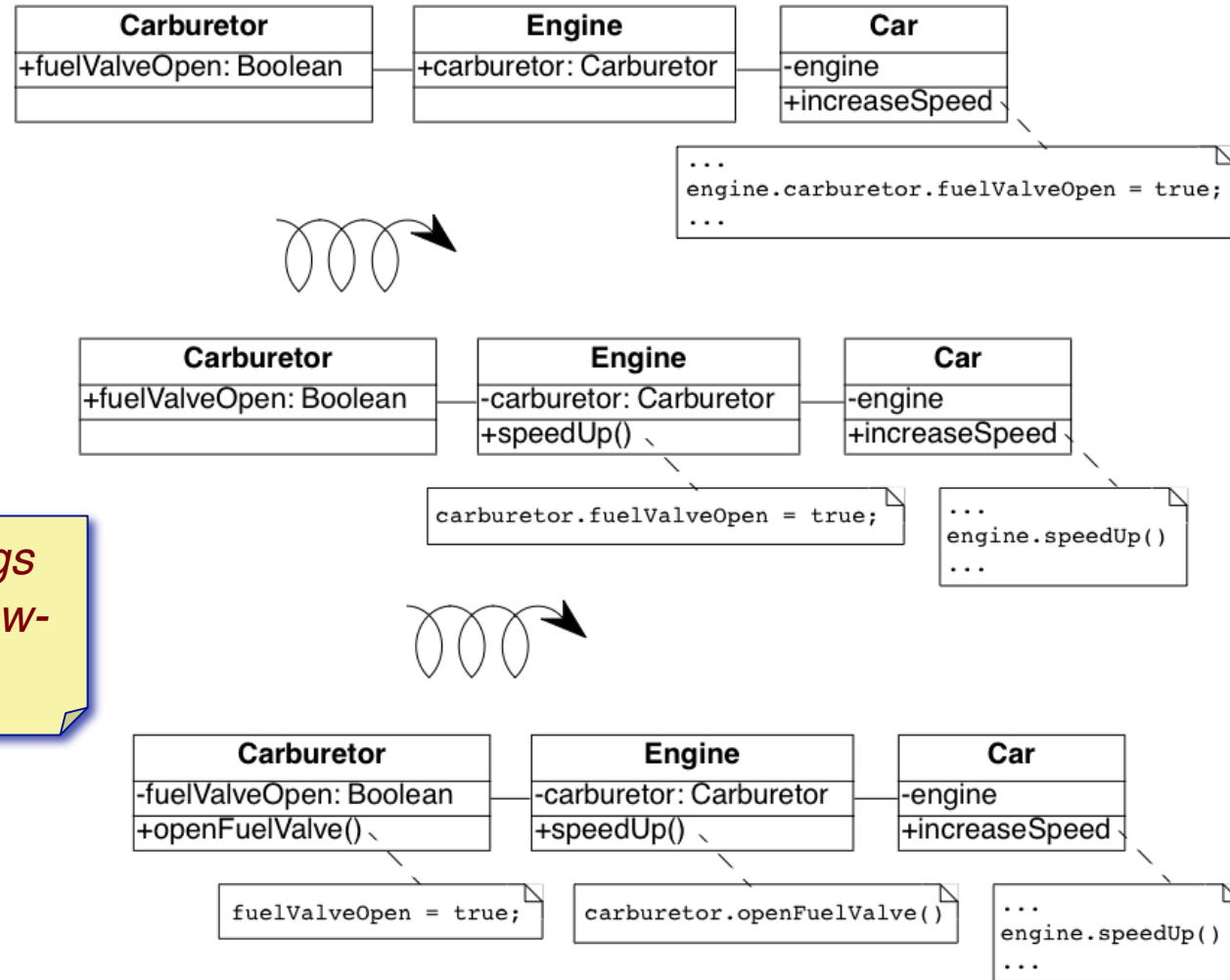
Repetitive Code Elements

(Helfman, 1995)

# Redistribute Responsibilities

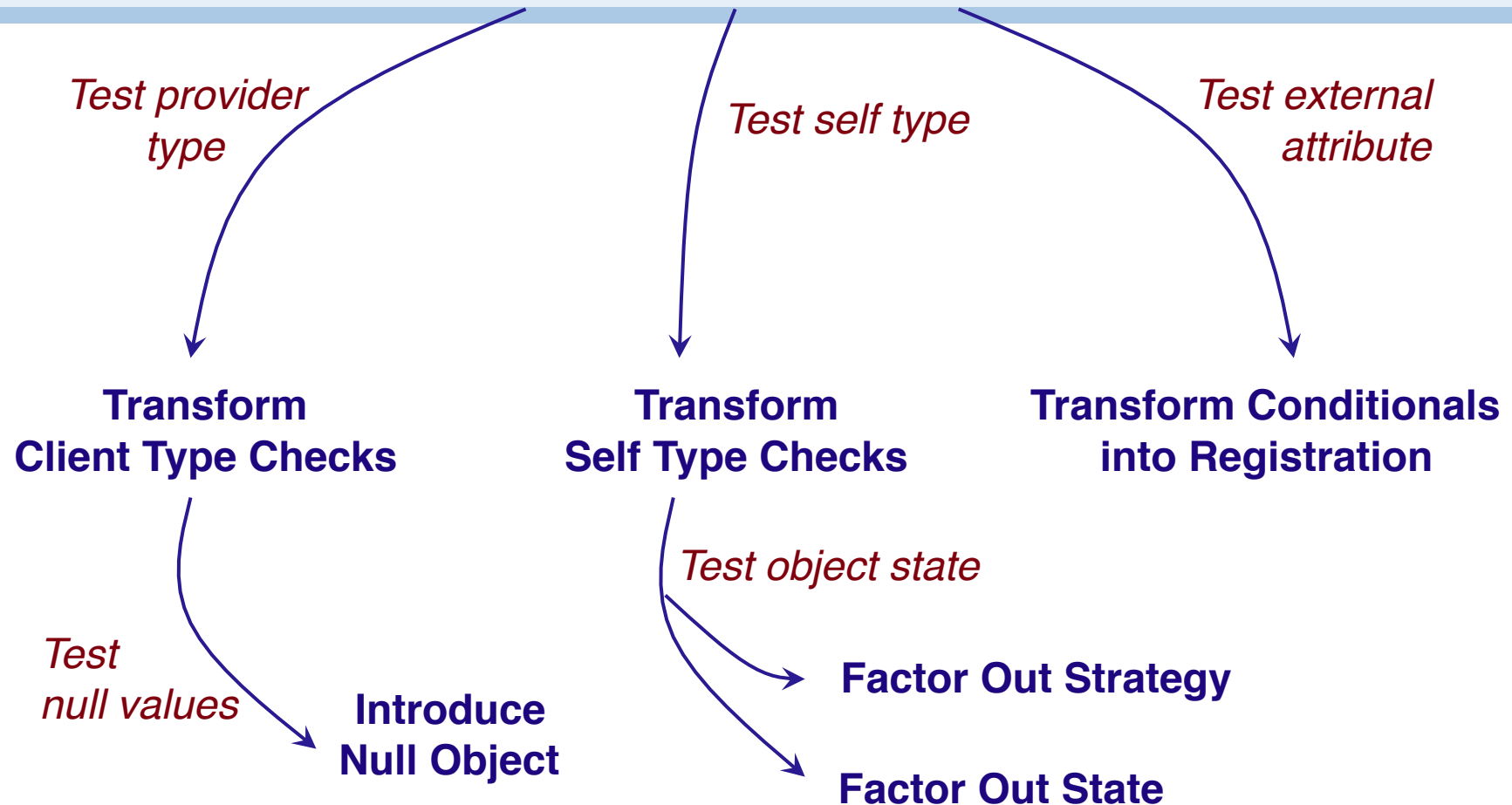
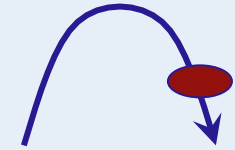


# High-level refactorings



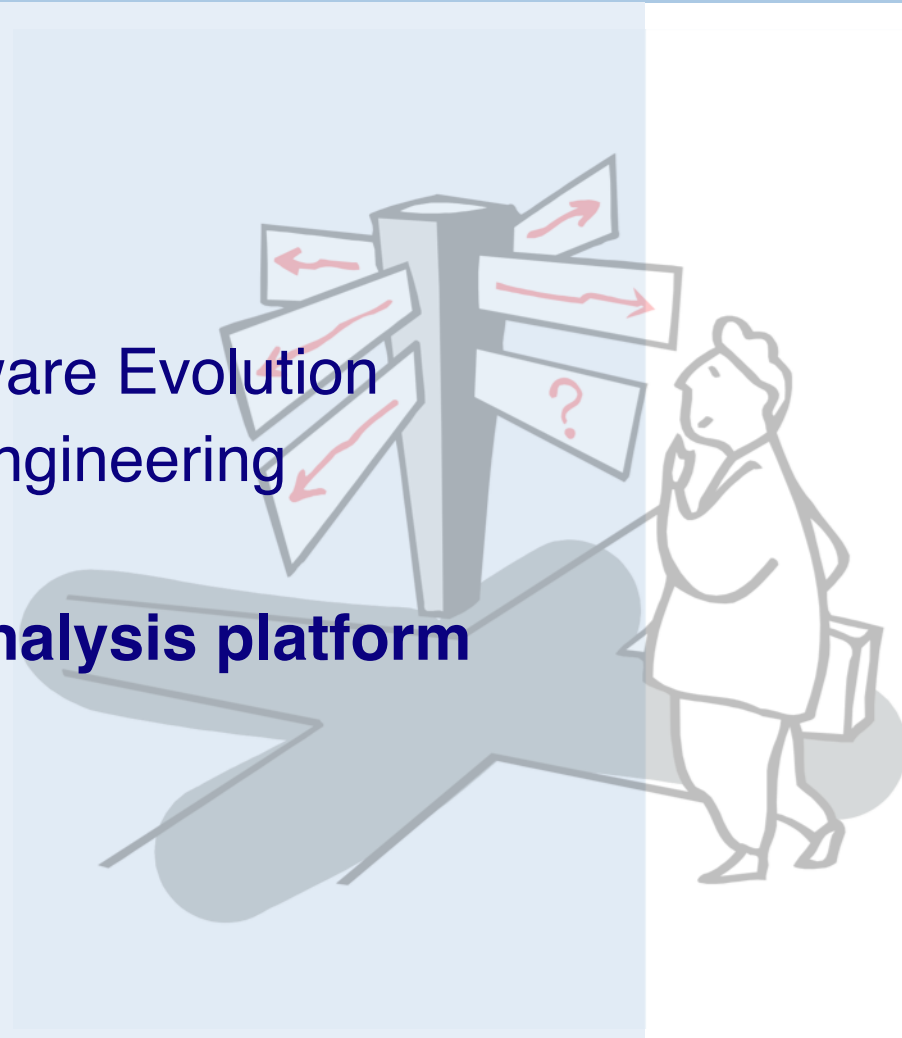
*High-level refactorings make use of many low-level refactorings*

# Transform Conditionals to Polymorphism

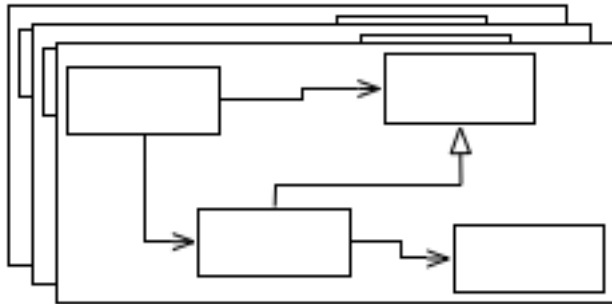


# Roadmap

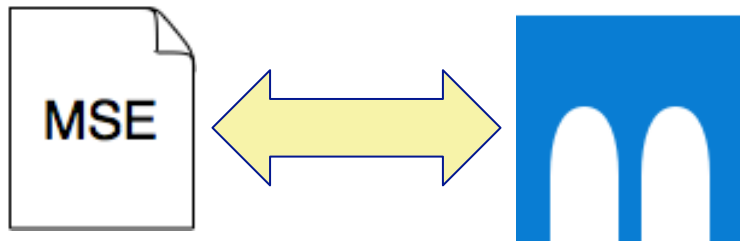
- > Lehman's Laws of Software Evolution
- > Forward and Reverse Engineering
- > Reengineering Patterns
- > **The Moose software analysis platform**



# Moose — an extensible platform for software and data analysis



FAMIX meta-models family



Import/export format for models

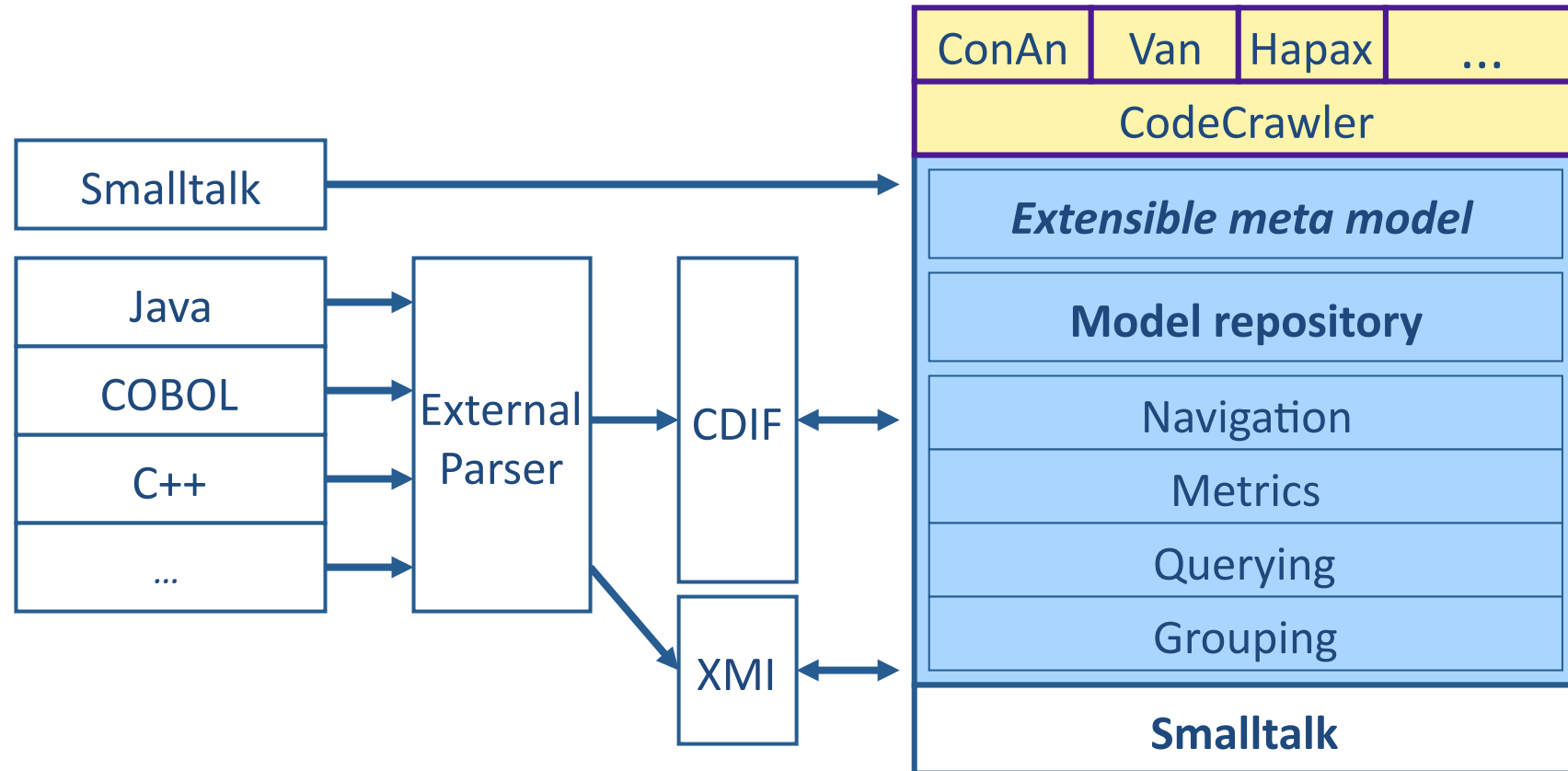


Data parsing support

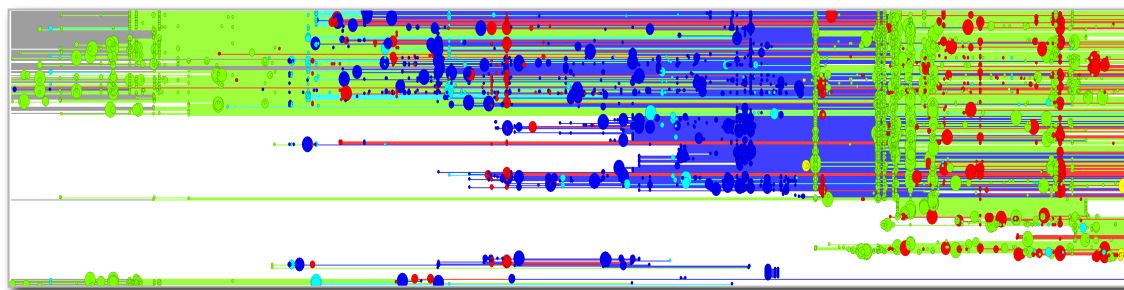
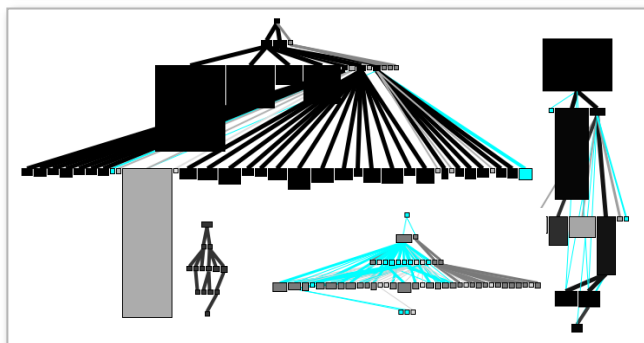
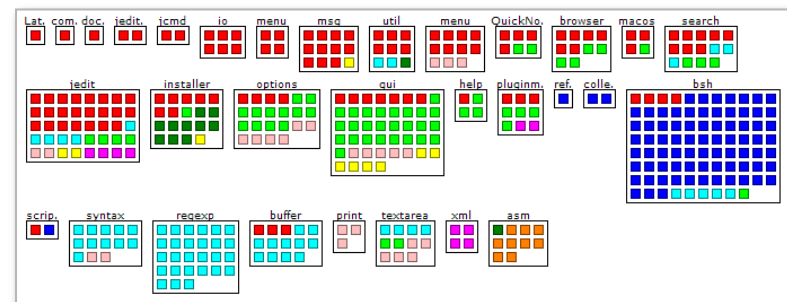
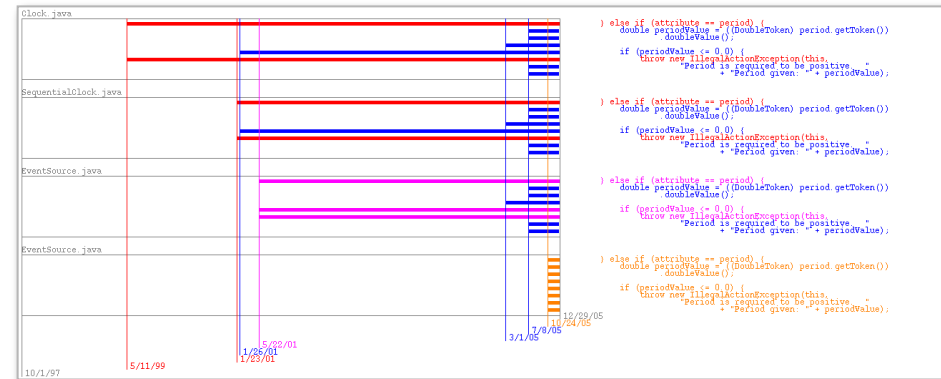
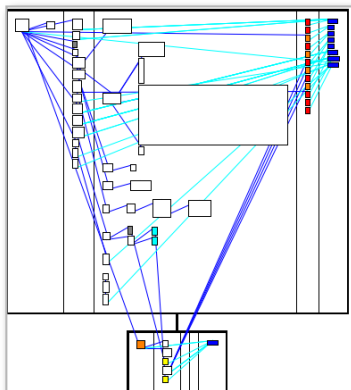
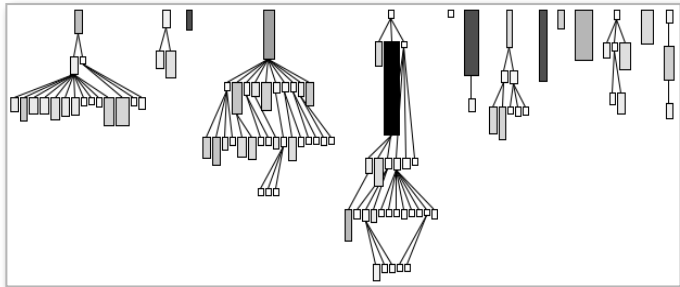
...more



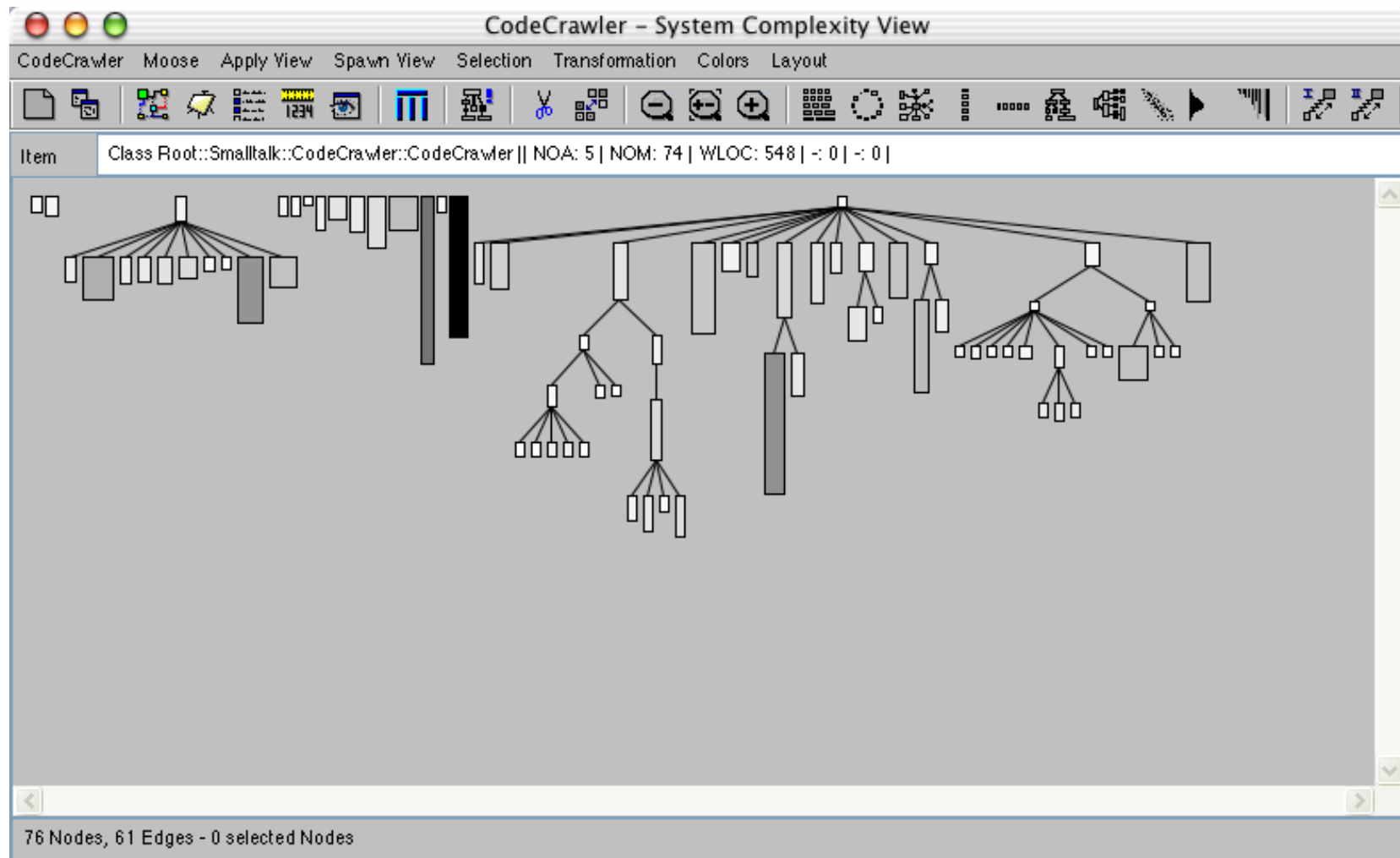
# Explicit metamodels enable change



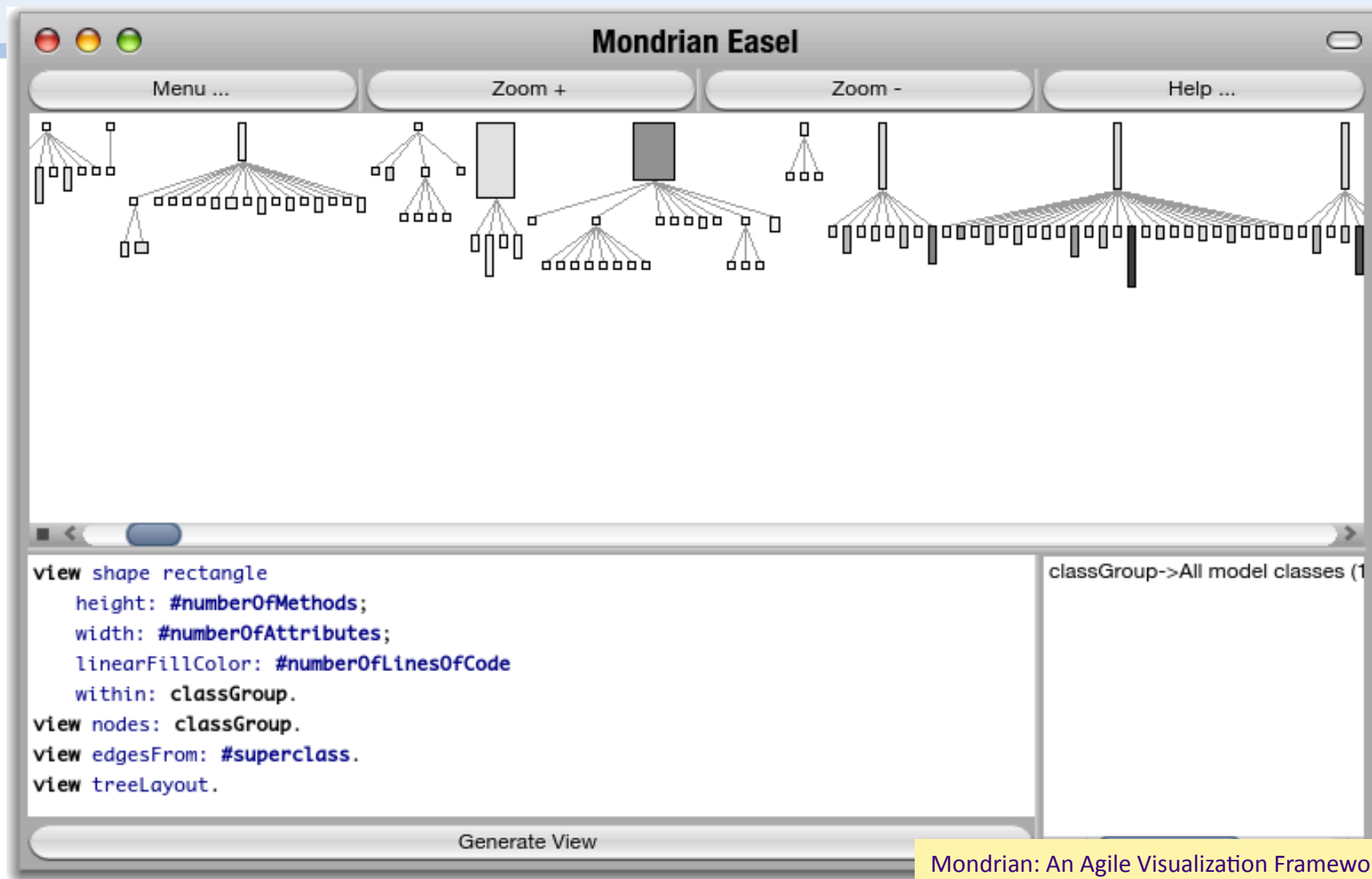
# Moose visualizations



# Programming visualizations with CodeCrawler



# Scripting visualizations with Mondrian



Mondrian: An Agile Visualization Framework.  
Meyer, Gîrba, Lungu. SoftVis 2006

# Data navigation through generic or dedicated browsers

The screenshot shows the Moose Finder application interface. The title bar reads "Moose Finder". The main window is titled "esv-projects-dec2009 (MooseModel)".

On the left side, there is a list of model classes with their counts:

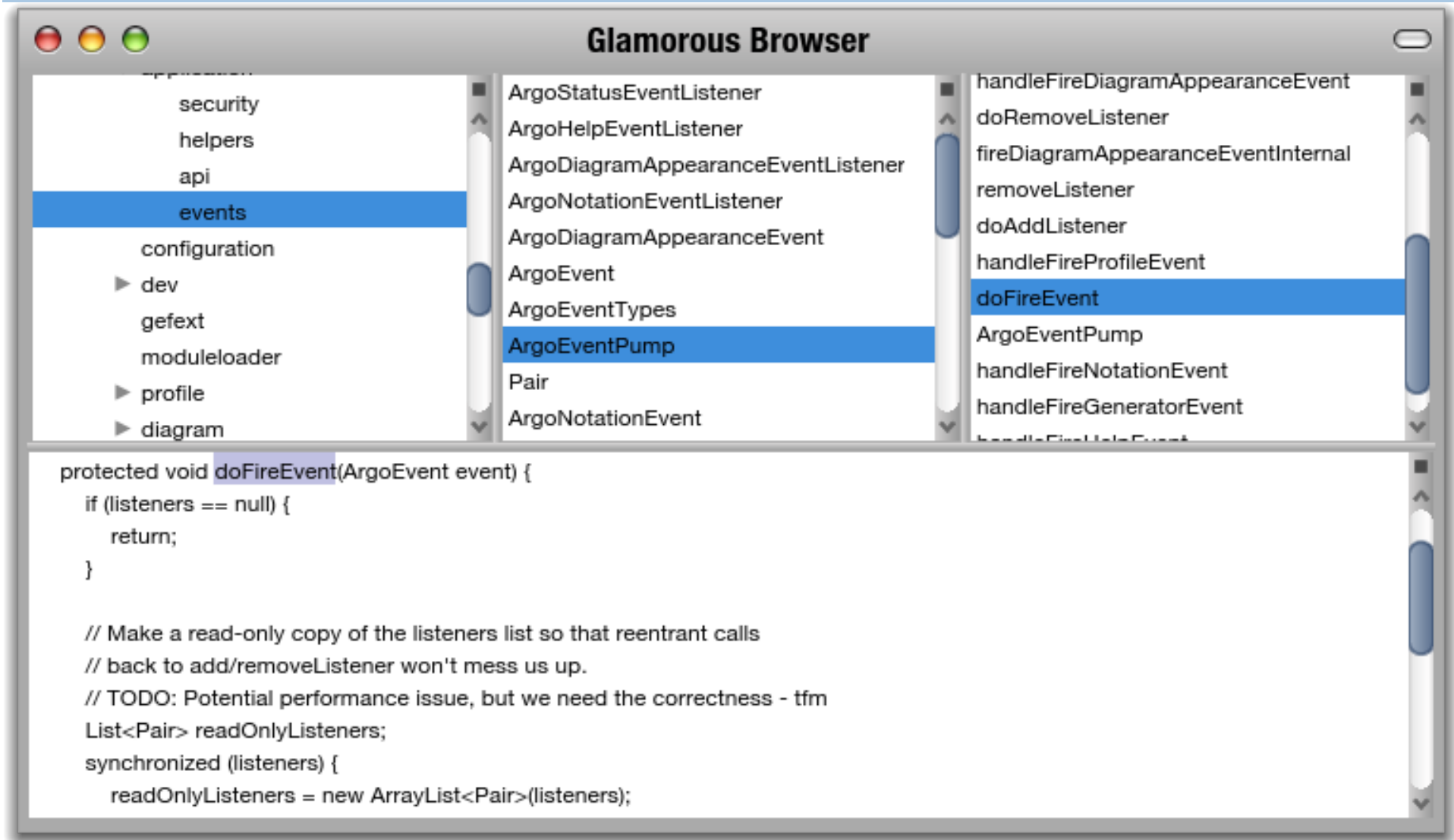
- All Data Base Tables (108)
- All Entity Beans (94)
- All Maps (311)
- All Message Driven Beans (3)
- All Session Beans (90)
- All famixaccesses (33796)
- All famixannotationinstances (5766)
- All famixannotationtypes (90)
- All famixattributes (7257)
- All famixcaughtexceptions (1107)
- All famixclasses (3615)
- All famixclasses (3615)
- All famixdeclaredexceptions (1803)
- All famixfunctions (1)
- All famixinheritances (4043)
- All famixinvocations (40631)
- All famixlocalvariables (13521)
- All famixmethods (19026)
- All famixnamespaces (560)
- All famixparameters (11094)
- All famixprimitivetypes (9)
- All famixthrownexceptions (818)
- All model classes (2202)
- All model namespaces (457)
- All model types (2531)

The right side of the window displays a hierarchical tree view titled "All model classes (2202) (FAMIXClassGroup)". The tree shows a complex structure of nodes and edges, representing the relationships between the model classes. A vertical scrollbar is visible on the right side of the tree view.

# Scripting browsers with Glamour

```
| browser |  
  
browser := GLMTabulator new.  
  
browser  
  row: [ :r | r column: #namespaces; column: #classes; column: #methods ];  
  row: #details.  
  
browser transmit to: #namespaces; andShow: [ :a |  
  a tree  
    display: [ :model | model allNamespaces select: [ :each | each isRoot ] ];  
    children: [ :namespace | namespace childScopes ];  
    format: [ :namespace | namespace stubFormattedName ] ].  
  
browser transmit from: #namespaces; to: #classes; andShow: [ :a |  
  a list  
    display: [ :namespace | namespace classes ];  
    format: [ :class | class stubFormattedName ] ].  
  
browser transmit from: #classes; to: #methods; andShow: [ :a |  
  a list  
    display: [ :class | class methods ];  
    format: [ :method | method stubFormattedName ] ].  
  
browser transmit from: #methods; to: #details; andShow: [ :a |  
  a text  
    display: [ :method | method sourceText ] ].  
  
browser openOn: MooseModel root allModels anyOne|
```

# Scripting browsers with Glamour

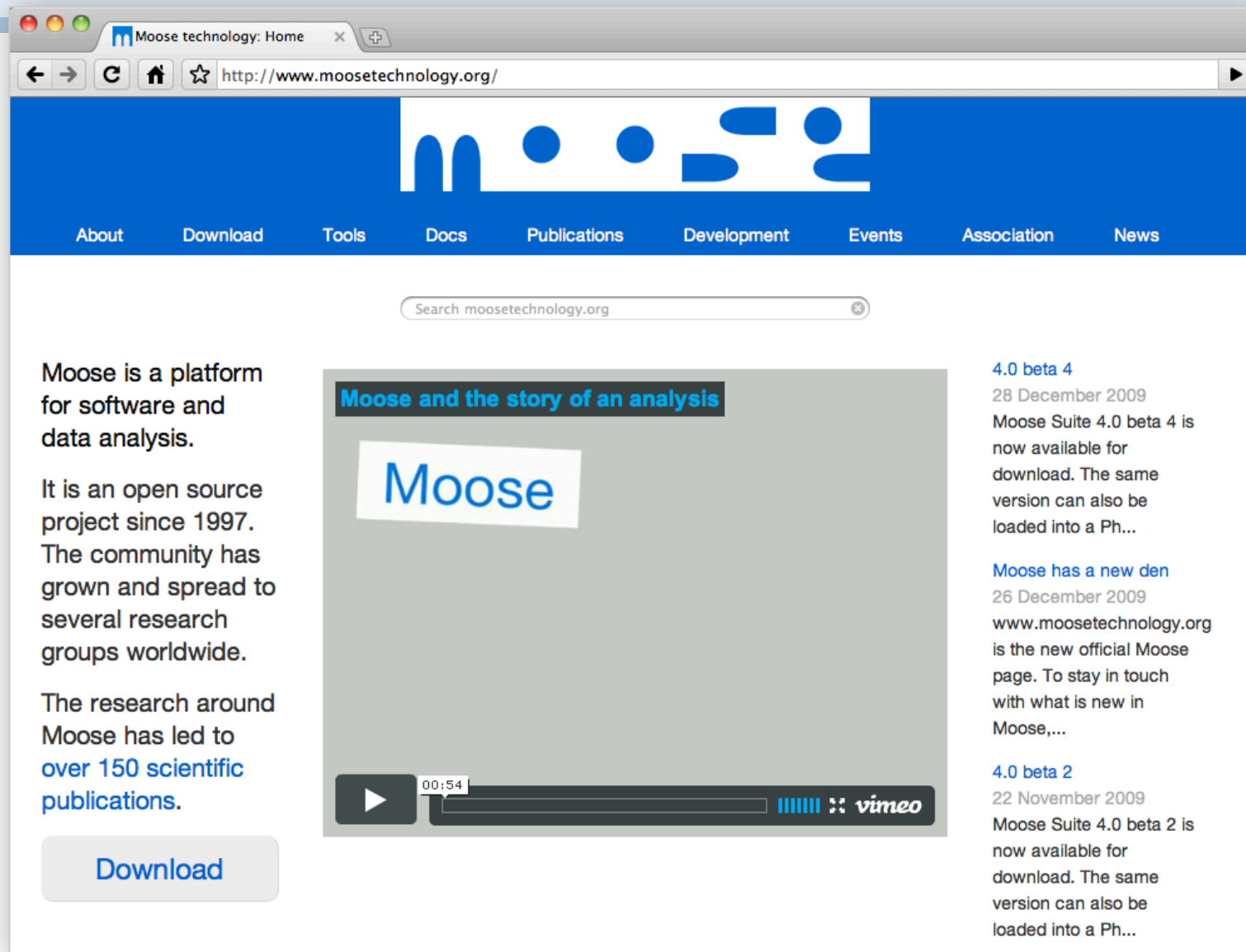


The screenshot shows the Glamorous Browser interface. The window title is "Glamorous Browser". The interface is divided into three main sections:

- Left Panel (Navigation):** A tree view showing a hierarchy of folders. The "events" folder is selected and highlighted in blue. Other folders include "security", "helpers", "api", "configuration", "dev", "gefext", "moduleloader", "profile", and "diagram".
- Middle Panel (Class List):** A list of classes and interfaces. The "ArgoEventPump" class is selected and highlighted in blue. Other items include "ArgoStatusEventListener", "ArgoHelpEventListener", "ArgoDiagramAppearanceEventListener", "ArgoNotationEventListener", "ArgoDiagramAppearanceEvent", "ArgoEvent", "ArgoEventTypes", "Pair", and "ArgoNotationEvent".
- Right Panel (Method List):** A list of methods. The "doFireEvent" method is selected and highlighted in blue. Other methods include "handleFireDiagramAppearanceEvent", "doRemoveListener", "fireDiagramAppearanceEventInternal", "removeListener", "doAddListener", "handleFireProfileEvent", "ArgoEventPump", "handleFireNotationEvent", and "handleFireGeneratorEvent".
- Bottom Panel (Code Editor):** A text editor showing the implementation of the "doFireEvent" method. The code is as follows:

```
protected void doFireEvent(ArgoEvent event) {  
    if (listeners == null) {  
        return;  
    }  
  
    // Make a read-only copy of the listeners list so that reentrant calls  
    // back to add/removeListener won't mess us up.  
    // TODO: Potential performance issue, but we need the correctness - tfm  
    List<Pair> readOnlyListeners;  
    synchronized (listeners) {  
        readOnlyListeners = new ArrayList<Pair>(listeners);  
    }  
}
```

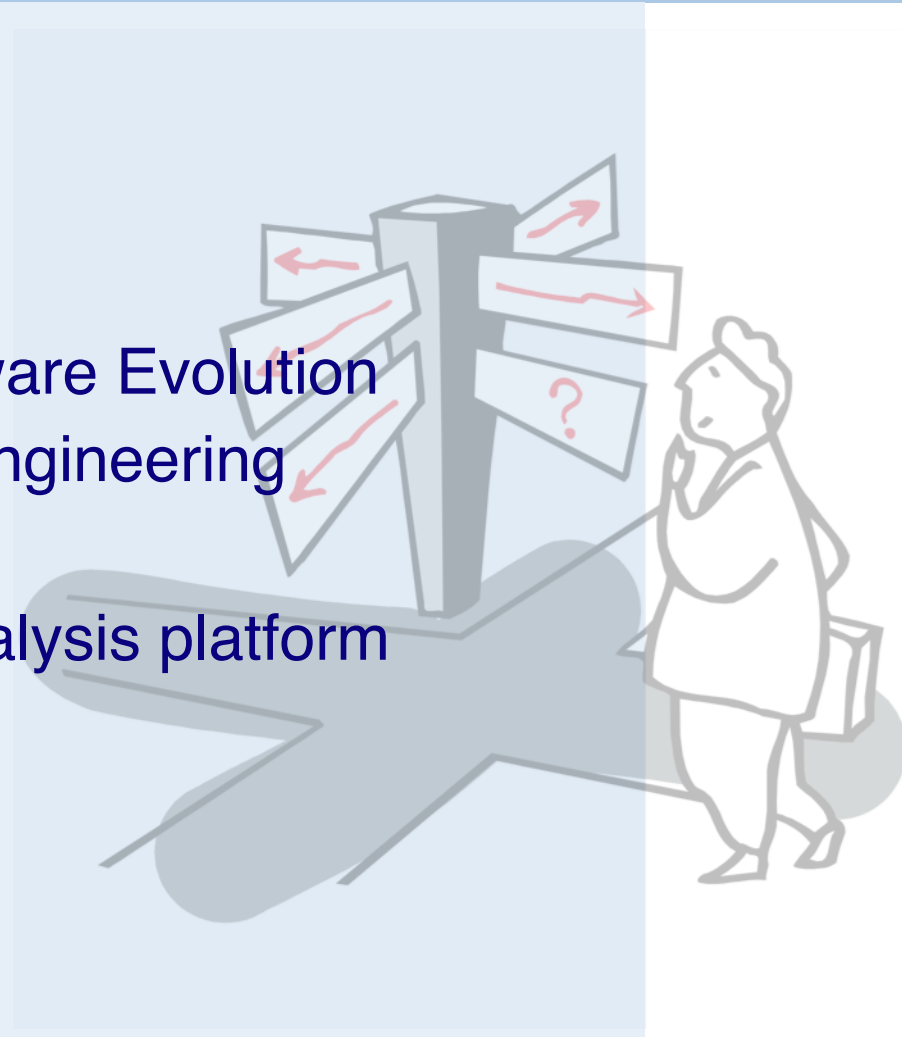
# Moose — a platform for collaborative research





# Roadmap

- > Lehman's Laws of Software Evolution
- > Forward and Reverse Engineering
- > Reengineering Patterns
- > The Moose software analysis platform



# Conclusion

- > Valuable software inevitably changes
- > Reverse and reengineering are necessary activities throughout the lifecycle of a software system
- > Simple techniques go a long way

# What you should know!

- > What is Lehman's Laws of Continuing Change?
- > Why do software systems become more complex over time?
- > Why is duplicated code considered to be a bad code smell?
- > How can you reduce the cost of software maintenance?
- > What is meant by "reverse engineering"?
- > How can studying exceptional entities help you to understand a software system?

## Can you answer the following questions?

- > Is a legacy software system a good thing or a bad thing to have?
- > How can you ensure that documentation stays in sync with implementation?
- > When should you start a reengineering project?
- > What are the dangers of trying to fix the buggiest code first?

# License



## Attribution-ShareAlike 3.0 Unported

### *You are free:*

- to Share** — to copy, distribute and transmit the work
- to Remix** — to adapt the work

### *Under the following conditions:*

**Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.

<http://creativecommons.org/licenses/by-sa/3.0/>