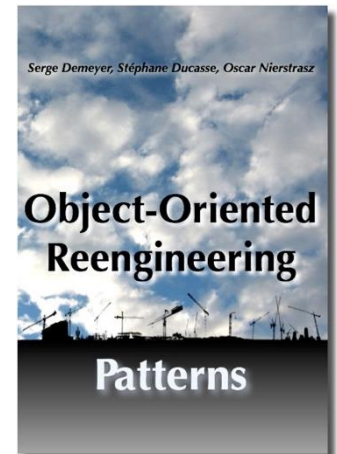
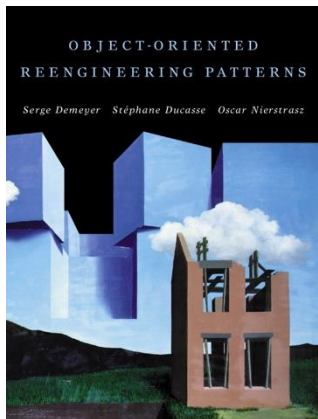


# Software Product Design and Development II

## CS 789

John Businge

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





# Overview of the Class

- <https://johnxu21.github.io/teaching/Software-Reengineering/>

# Schedule

## 1. Introduction

Software changes and that requires planning

## 2. Reverse Engineering

How to understand your code

## 3. Visualization

Scalable approach

## 4. Restructuring

How to Refactor Your Code

## 5. Dynamic Analysis (& Testing)

To be really certain

## 5. Code Integration

How to resolve conflicts

## 7. Mining Software Repositories

Learn from the past

## 8. Conclusion

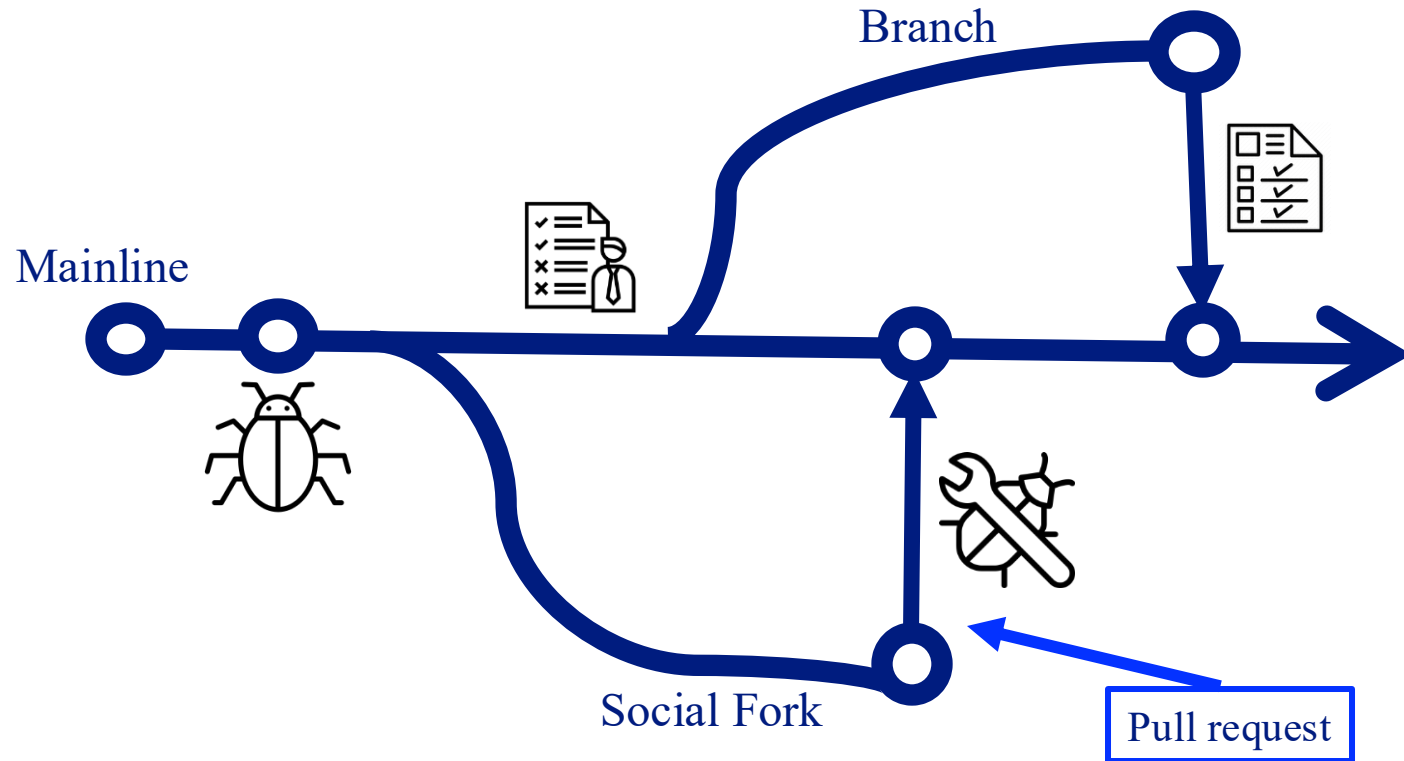


# Goals

***We will try to convince you:***

- Programs change!
- Reverse engineering, forward engineering and reengineering are *essential activities* in the lifecycle of any successful software system. (And especially OO ones!)
- There is a large set of *lightweight tools and techniques* to help you with reengineering.
- Despite these tools and techniques, *people must do job* and they represent the most valuable resource.

# Program Change



# Software Maintenance - Cost

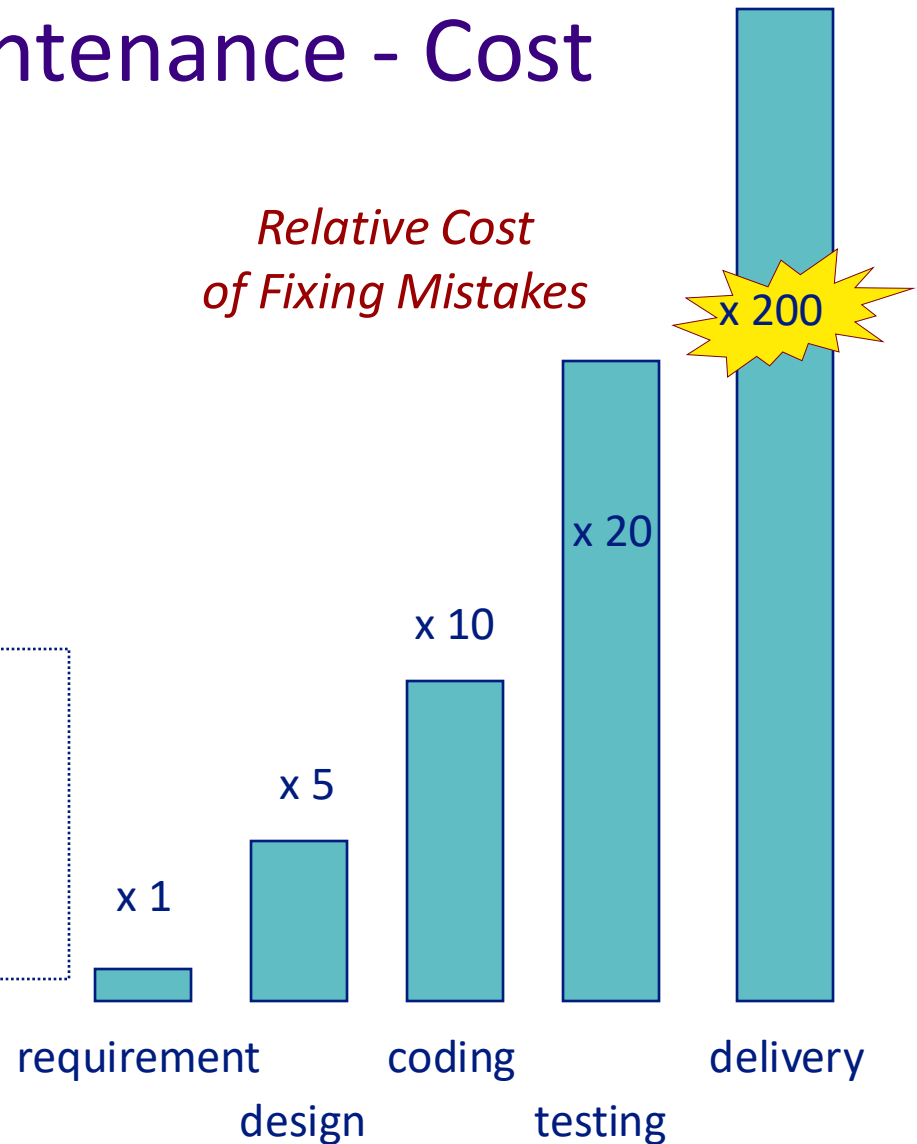
## *Relative Maintenance Effort*

Between 50% and 75% of global effort is spent on “maintenance” !

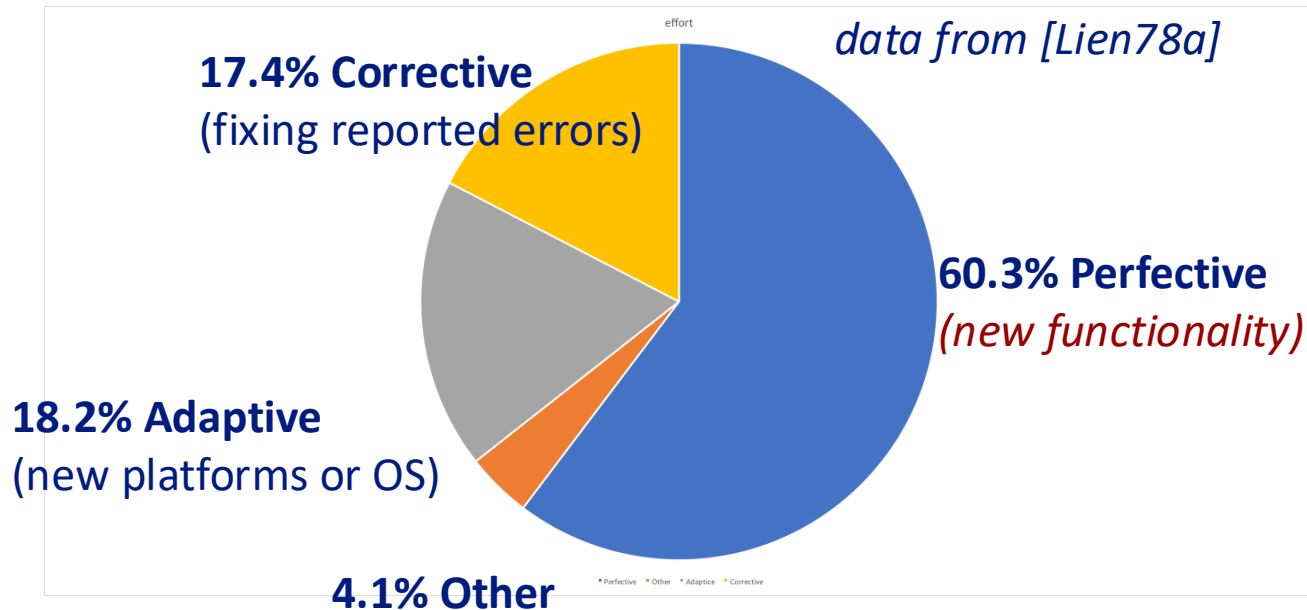
### *Solution ?*

- Better requirements engineering?
- Better software methods & tools (database schemas, CASE-tools, objects, components, ...)?

## *Relative Cost of Fixing Mistakes*



# Continuous Development



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



# Lehman's Laws

A classic study by Lehman and Belady [Lehm85a] identified several “laws” of system change.

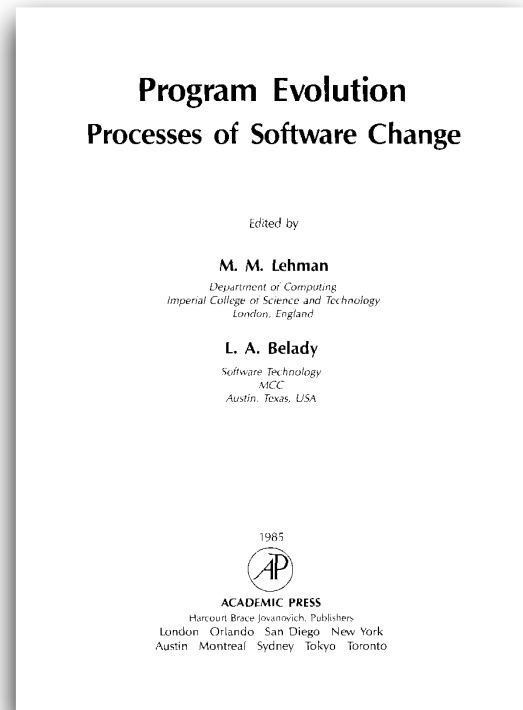
## *Continuing change*

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

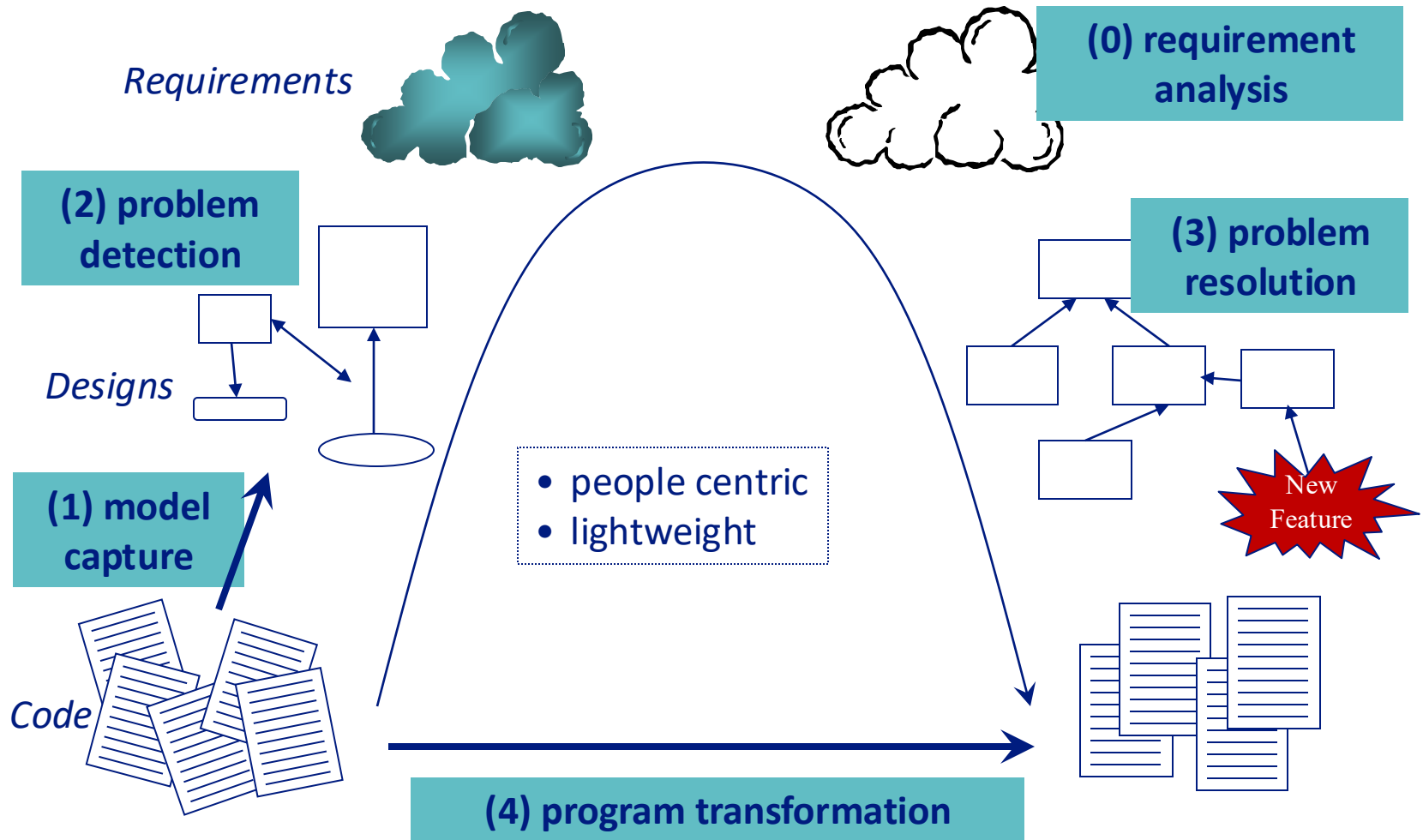
## *Increasing complexity*

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

Those laws are still applicable...

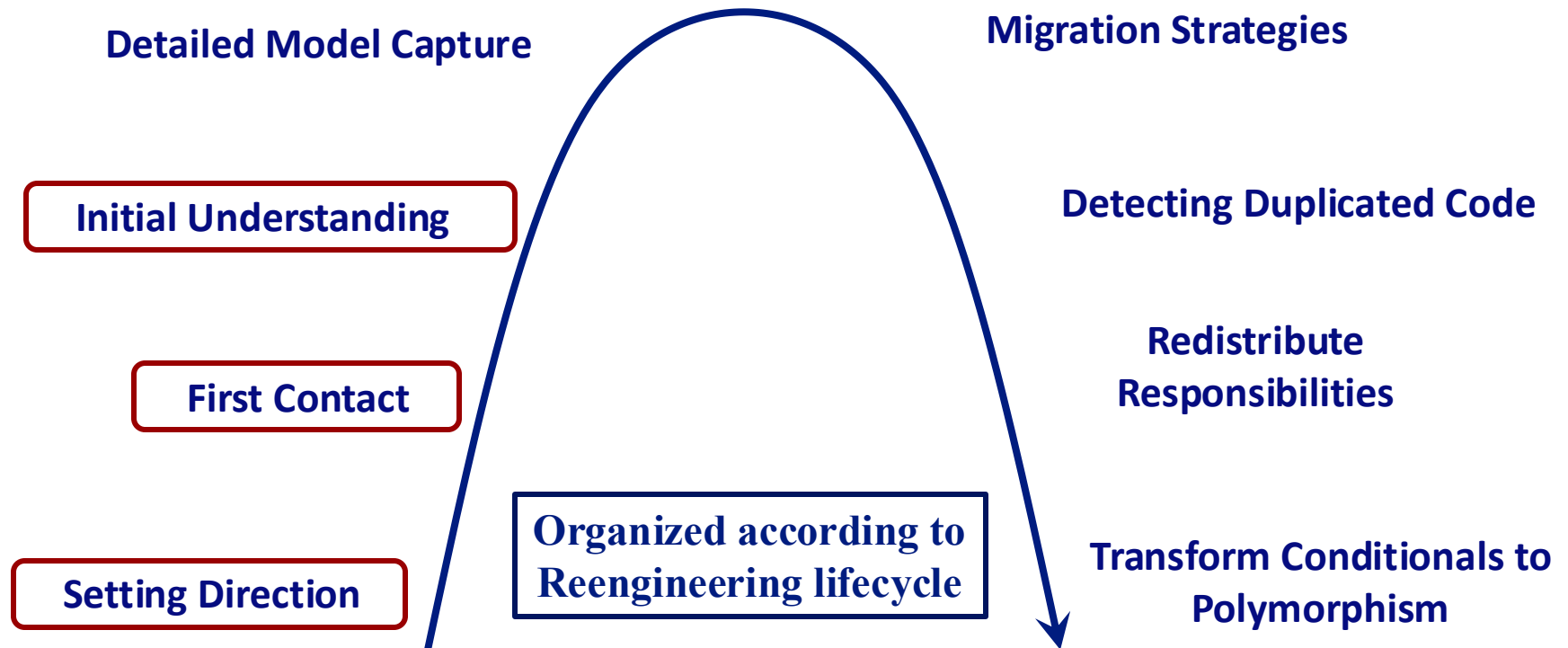


# The Reengineering Life-Cycle



# A Map of Reengineering Patterns

Tests: Your Life Insurance



## 2. Reverse Engineering

- What and Why
- First Contact
  - + Interview during Demo
- Initial Understanding



# What and Why ?

## **Definition**

Reverse Engineering is the *process of analysing* 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.
- Chikofsky & Cross, '90

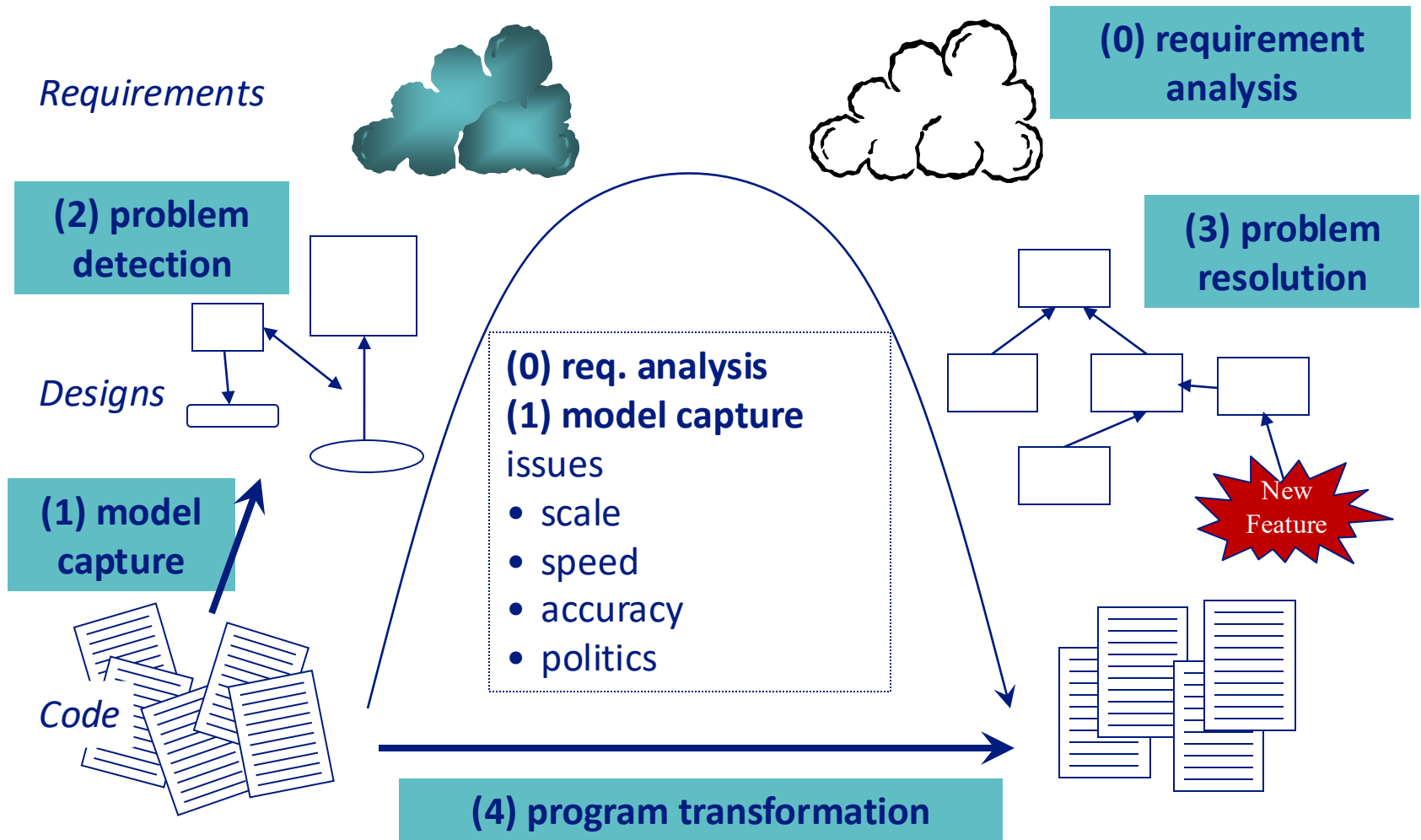
## **Motivation**

*Understanding* other people's code

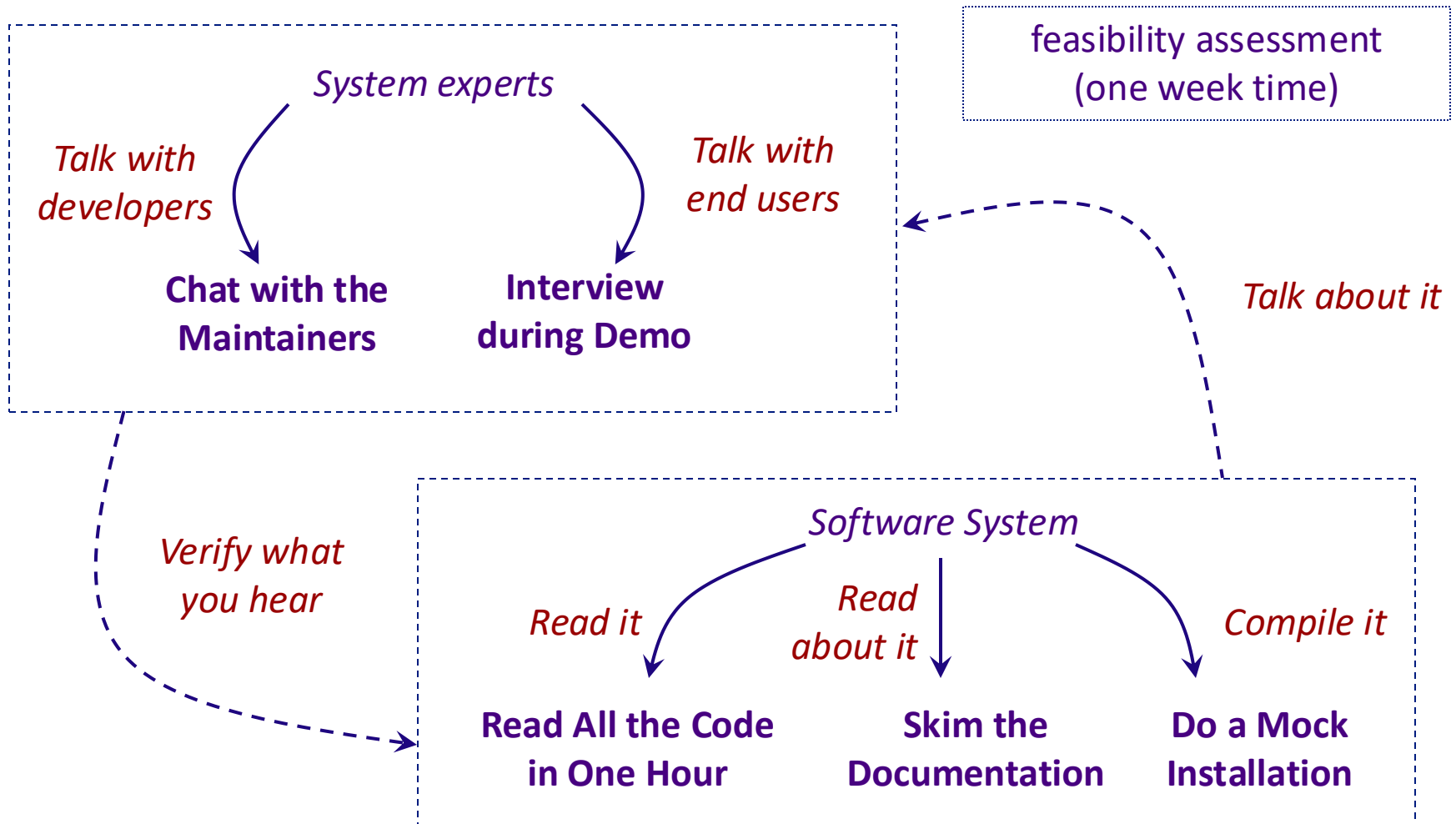
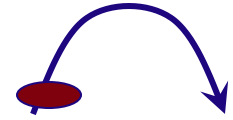
(cfr. newcomers in the team, code reviewing,  
original developers left, ...)

*Generating UML diagrams is NOT reverse engineering  
... but it is a valuable support tool*

# The Reengineering Life-Cycle



# First Contact



# Interview during Demo

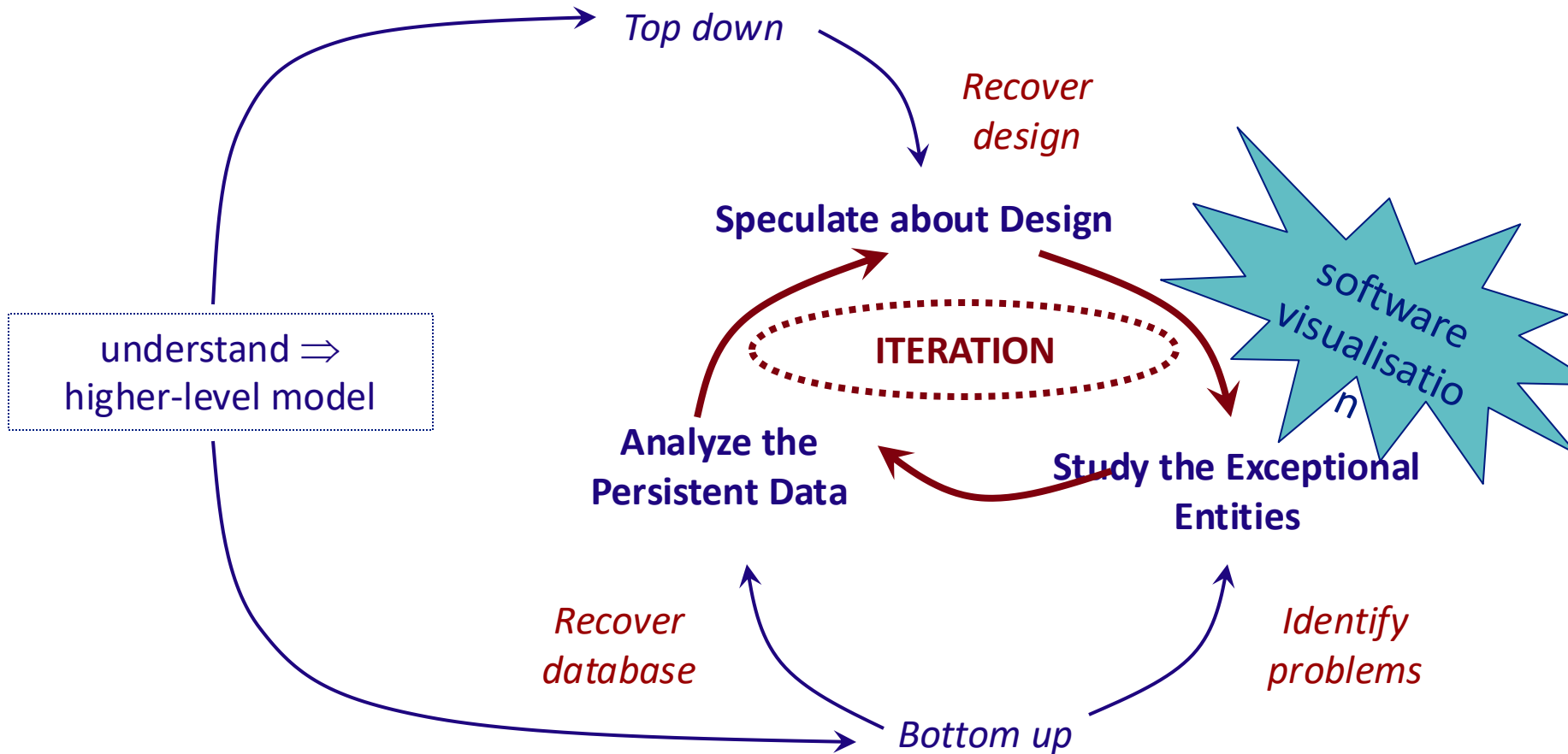
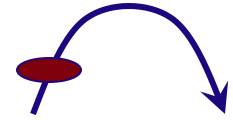
Problem: What are the typical usage scenarios?

Solution: Ask the user!

- Solution: interview during demo
  - select several users
  - demo puts a user in a positive mindset
  - demo steers the interview
- ... however
  - + Which user ?
  - + Users complain
  - + What should you ask ?



# Initial Understanding

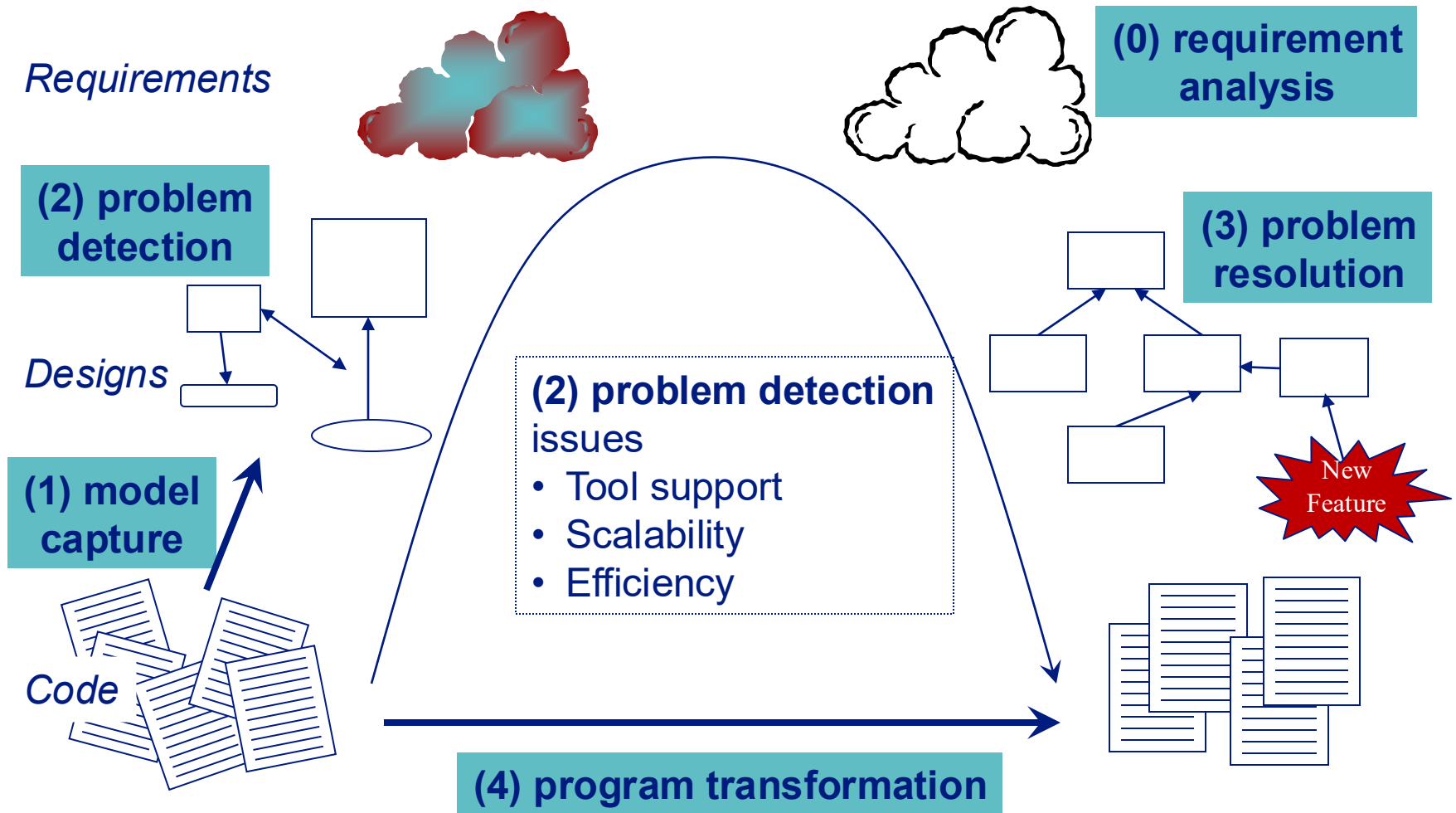


# 3. Software Visualization

- Introduction
  - + The Reengineering life-cycle
- Examples
- Lightweight Approaches
  - + tooling



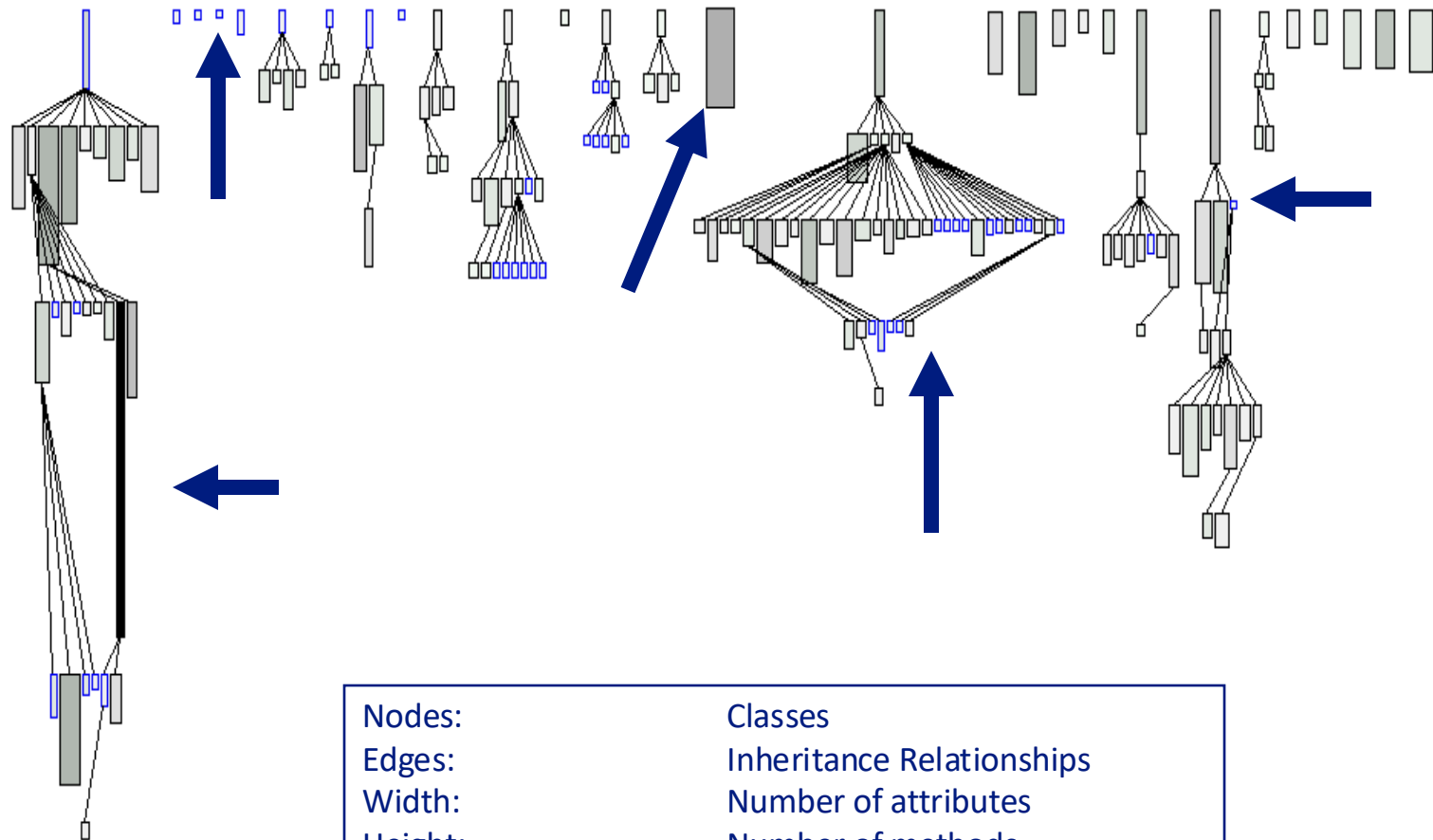
# The Reengineering Life-cycle



# UML Diagrams

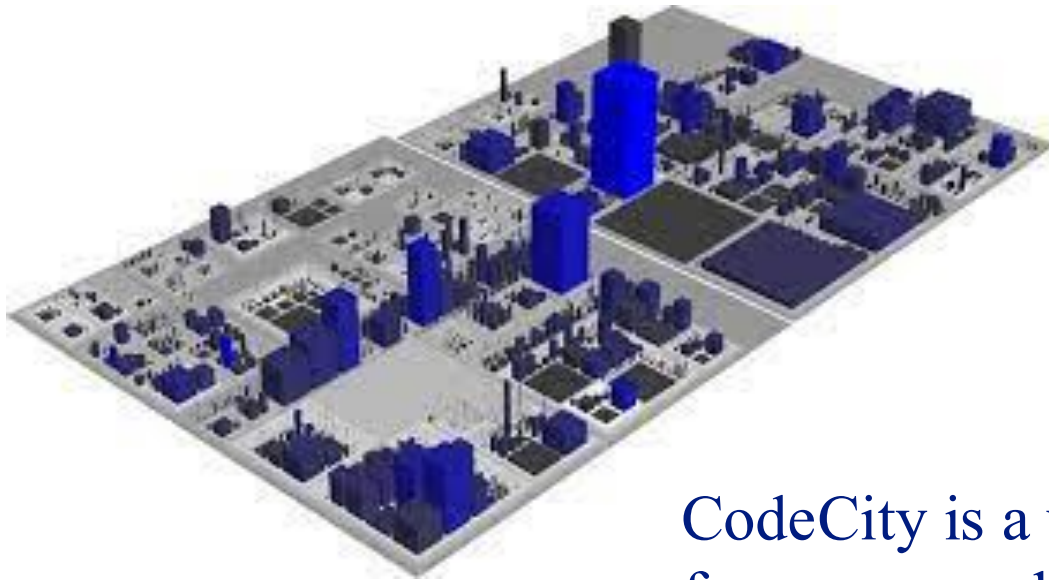
- (Mostly) Simple and Standard Way to present an abstract visualization of a system
- UML defines 14 diagrams
- Useful to plan and design the reengineering project
- You will be using UML diagrams to show the system before and after the change

# System Complexity View



Nodes:	Classes
Edges:	Inheritance Relationships
Width:	Number of attributes
Height:	Number of methods
Color:	Number of lines of code

# Code City



CodeCity is a visualization concept for source code.

The source code is shown as an interactive 3D city.

# Code City

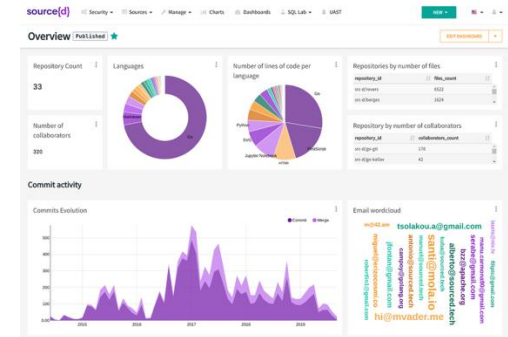
- Packages are “districts”, “neighborhoods,” or “city blocks”
- Each “building” represents a class \
- Width = Number of Attributes
- Height = Number of Methods
- Antennas => Classes with many methods and no attributes
- Parking lot => Classes with many attributes and no methods
- Skyscraper => Classes with a large number of methods and has many attributes

# State of the Art Tooling

## 1. source{d}

<https://sourced.tech>

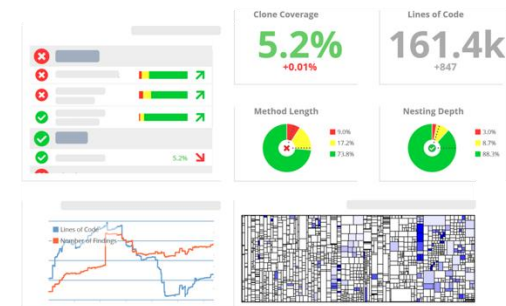
<https://github.com/src-d/engine>



## 2. teamscale

<https://www.cqse.eu/>

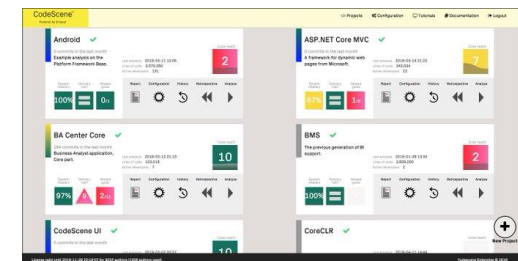
<https://github.com/cqse>



## 3. codescene

<https://codescene.io>

<https://github.com/empear-analytics>





# 4. Restructuring

## Identifying refactoring targets

### Redistribute Responsibilities

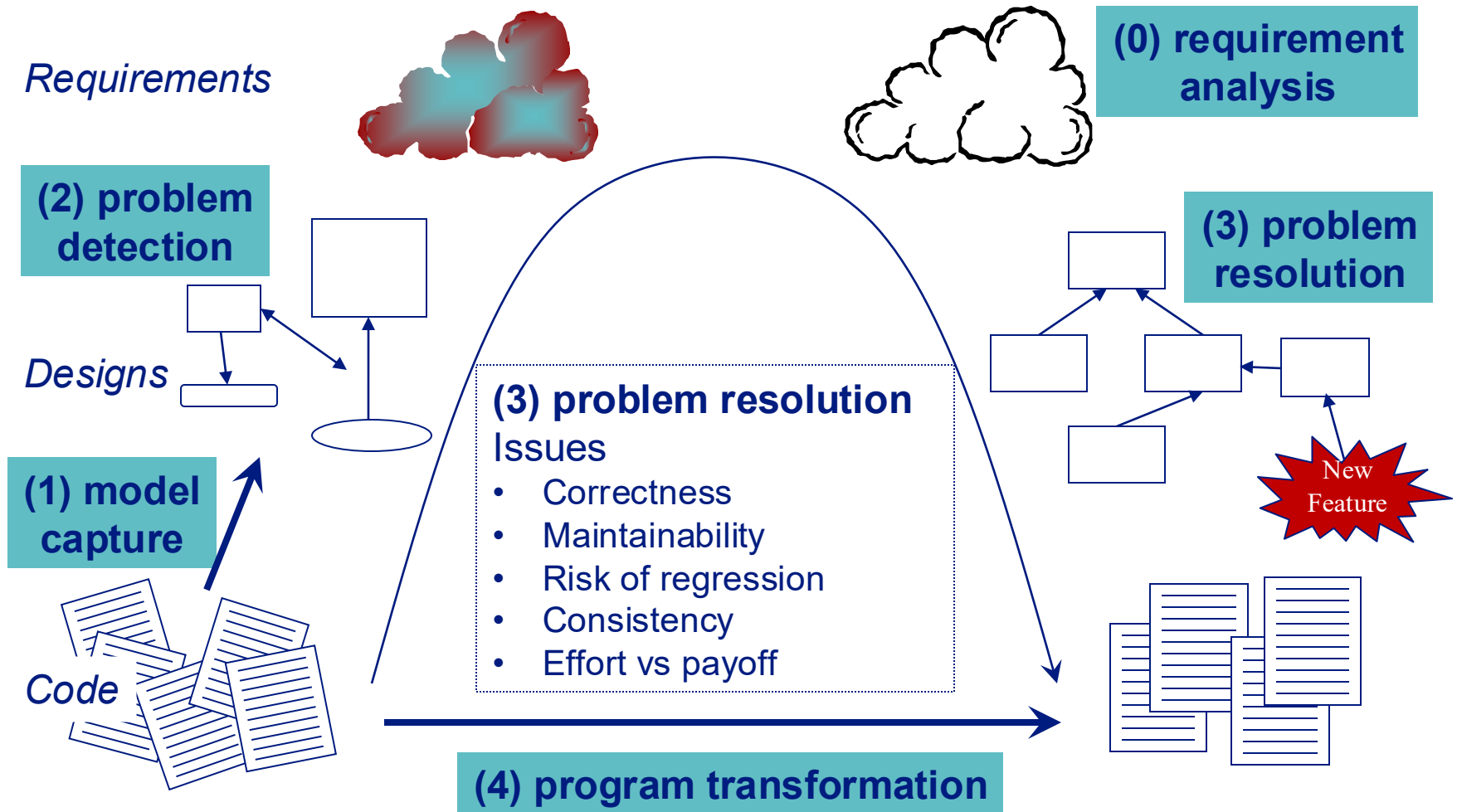
- + Move Behaviour Close to Data
- + Eliminate Navigation Code
- + Split up God Class
- + Empirical Validation



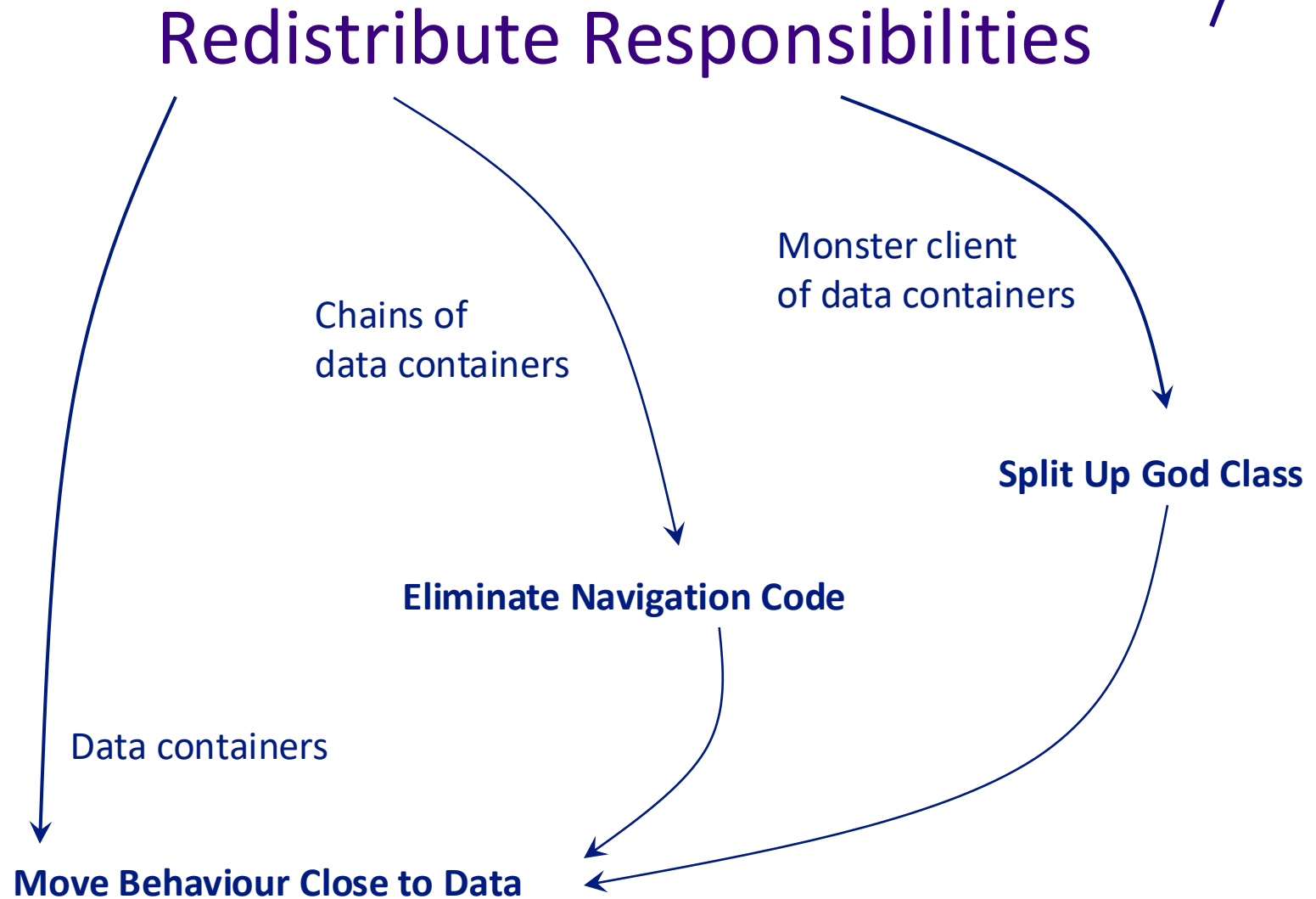
### Identifying refactorings in code

- ❖ Refactoring-aware techniques

# The Reengineering Life-cycle



# Identifying Refactoring Targets



# Split Up God Class

**Problem:** Break a class which monopolizes control?

**Solution:** Incrementally eliminate navigation code

- Detection:
  - + measuring size
  - + class names containing Manager, System, Root, Controller
  - + the class that all maintainers are avoiding
- How:
  - + move behaviour close to data + eliminate navigation code
  - + remove or deprecate façade
- However:
  - + If God Class is stable, then don't split  
⇒ shield client classes from the god class

# Split Up God Class

EmployeeManager
+hireEmployee(Employee employee) +terminateEmployee(int employeeId) +editEmployee(Employee employee) +addVacationTime(int employeeId, int days) +useVacationTime(int employeeId, int days) +addAddress(int employeeId, Address address) +removeAddress(int employeeId, int idAddress) +giveBonus(int employeeId, int bonus) +assignEquipment(int employeeId, Equipment equip) +giveRaise(int employeeId, int amount) +dockPay(int employeeId, int amount) +addSchedule(int employeeId, Schedule schedule) +addPhoneNumber(int employeeId, string phone)

EmployeeManager
+hireEmployee(Employee employee) +terminateEmployee(int employeeId) +editEmployee(Employee employee)

ScheduleManager
+addEmployeeSchedule(int employeeId, Schedule sch)

VacationManager
+addVacationTime(int employeeId, int days) +useVacationTime(int employeeId, int days)

PaymentManager
+giveBonus(int employeeId, int amount) +giveRaise(int employeeId, int amount) +dockPay(int employeeId, int amount)

EmployeeContactManager
+addAddress(int employeeId, Address address) +removeAddress(int employeeId, int addressId) +addPhoneNumber(int employeeId, string phone)

EquipmentManager
+assignEquipment(int employeeId, Equipment eq)

## 6. Dynamic Analysis (& Testing)

- Key Concept Identification
- Unit testing
- Test coverage
- Mutation testing



# Introduction

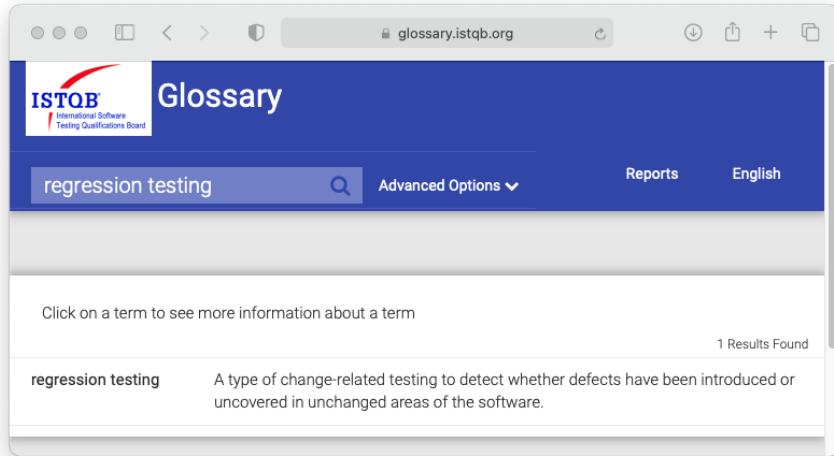
- Dynamic Analysis verifies properties of a system during execution
- Testing Analysis is one example of Dynamic Analysis
  - + Unit tests, integration tests, system tests, and acceptance tests use dynamic testing

# Testing

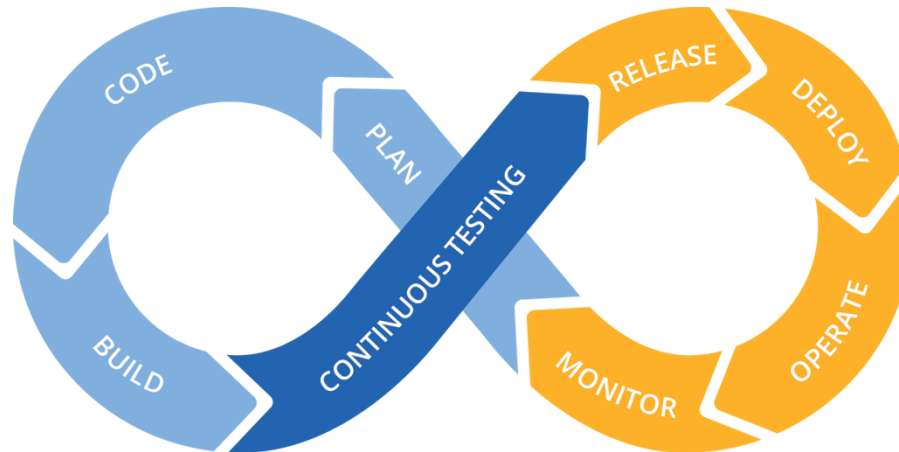
- Tests are your life insurance! (OORP, p. 149)
- Tests are essential to assure the quality of refactoring activities.
- Write Tests to Enable Evolution (OORP, p.153)
  - + Good tests can find bugs on your artifact
  - + Tests can also detect unwanted behavior
- You can also write tests to understand a part of a system (OORP, p.179)



# Regression Testing



A type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software.



# Coverage

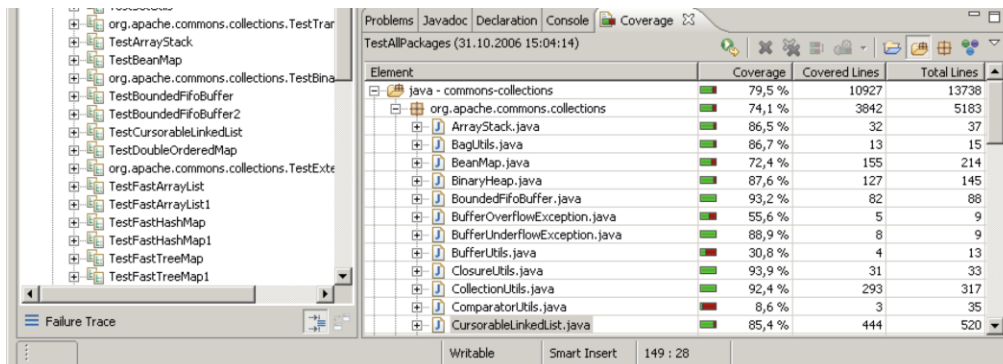
## LCOV - code coverage report

Current view: top level		
Test: libbash test coverage	Lines:	Hit 20640 Total 34749
Date: 2011-05-26	Functions:	1184 1287
	Branches:	15689 37086

Directory	Line Coverage	Functions
src/core	95.7 % 314 / 328	98.2 % 55 / 56
test	97.0 % 98 / 101	100.0 % 72 / 72
src/builtins/tests	98.6 % 144 / 146	100.0 % 203 / 203
src/builtins	98.6 % 214 / 217	100.0 % 45 / 45
src/core/tests	98.9 % 351 / 355	99.3 % 133 / 134
./src/builtins	100.0 % 9 / 9	93.3 % 14 / 15
src	100.0 % 35 / 35	91.7 % 11 / 12
./src/core	100.0 % 190 / 190	98.0 % 99 / 101

Generated by: [LCOV version 1.9](#)

Are the areas under change sufficiently covered by the current test suite?



Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

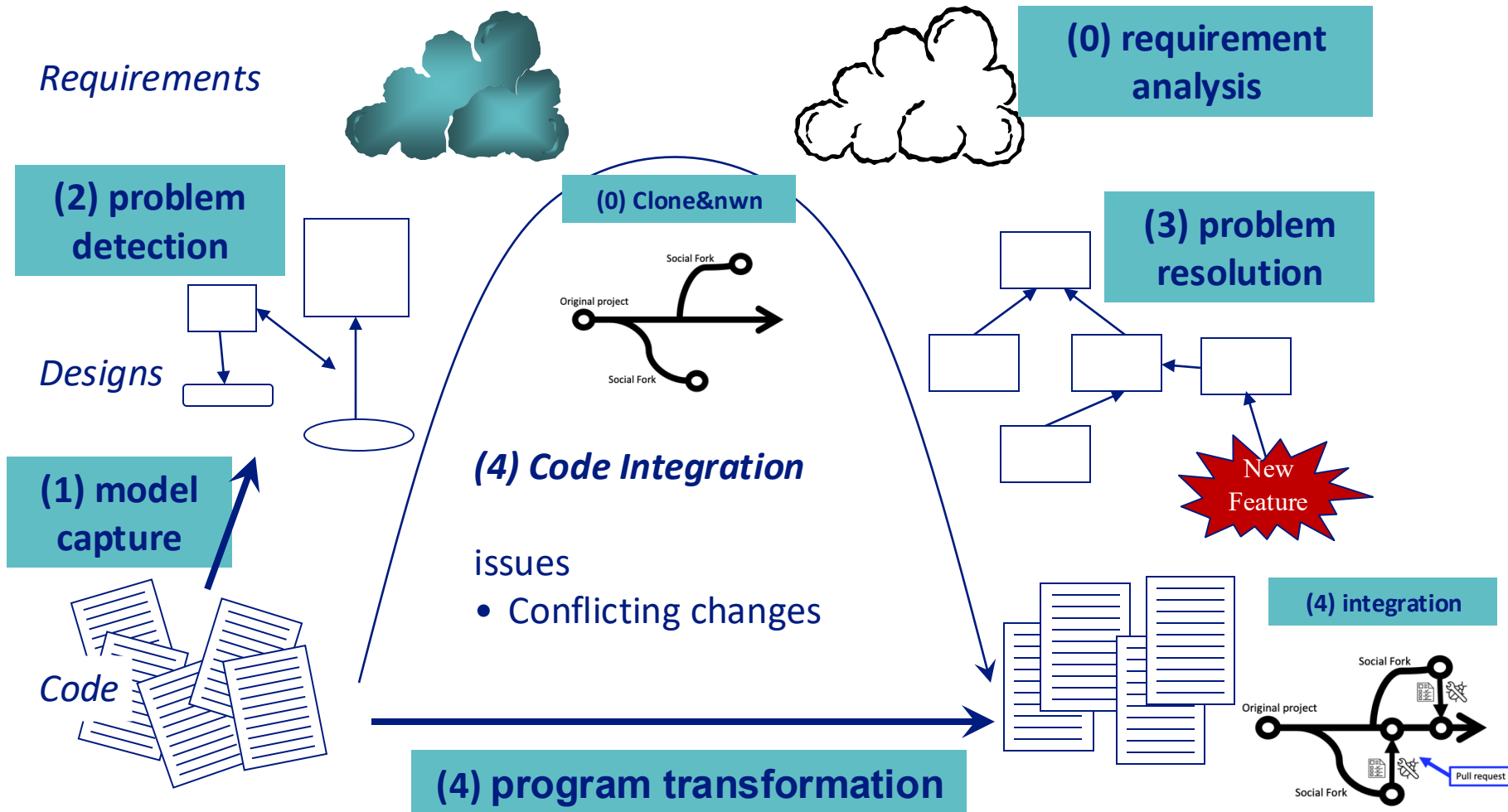
Compare coverage reports before and after refactoring!

# 5. Code integration

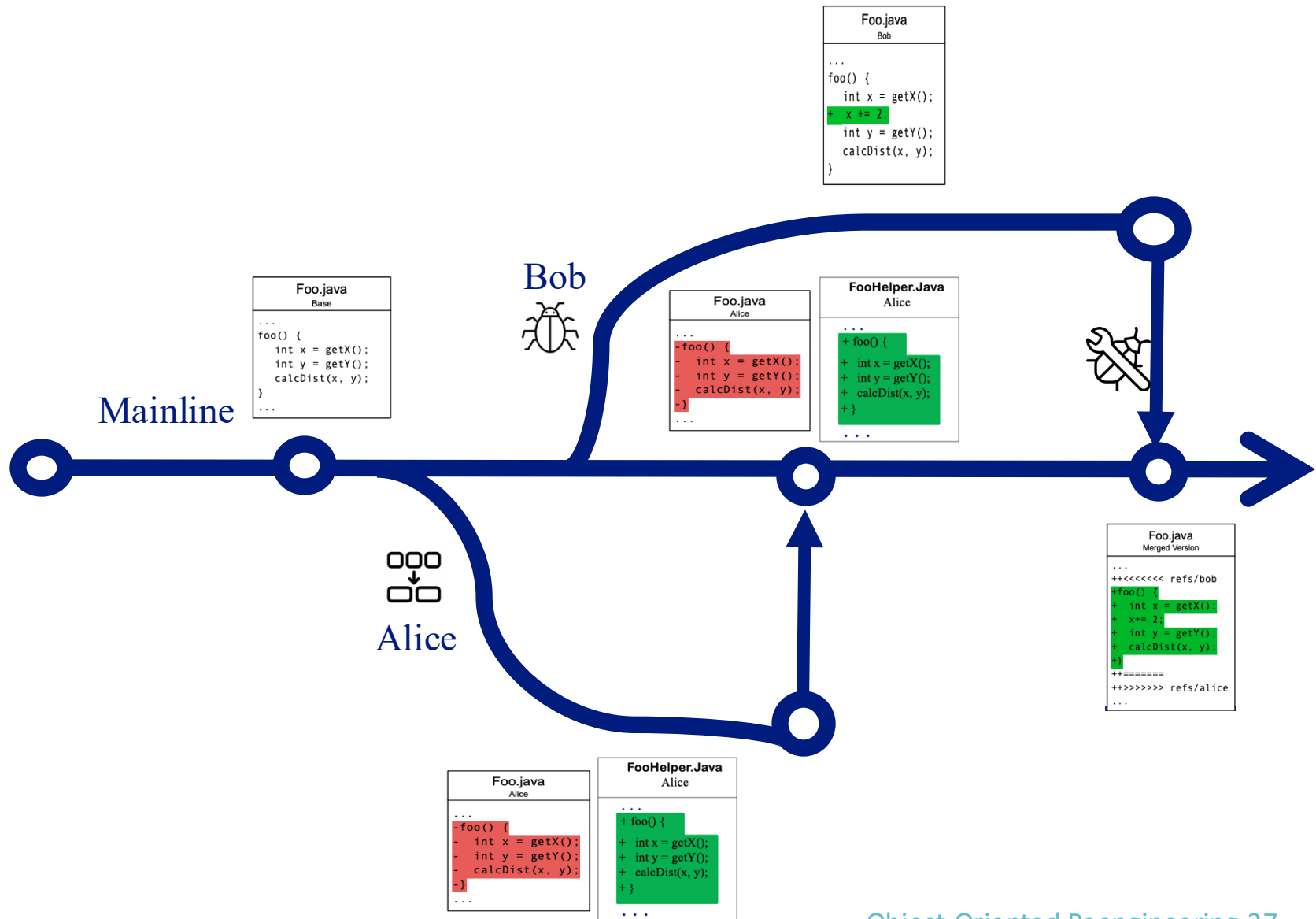
- Version Control Systems
- Branching
- Merging/integration
- Merge conflicts



# The Reengineering Life-Cycle



# Code Integration



# Refactoring-Aware Techniques

Many refactoring-aware techniques:

- IntelliMerge & Refmerge – merge branches
- APIDiff – adapt client software to library and framework updates
- RePatch - Ogenrwot and Businge (SCAM 2025)

All developed in the presence of refactoring operations.

# Code Integration

- Compare Code Mechanically (OORP, p.227)
  - Begins with mechanical comparison of patches across repositories
- Detecting Duplicated Code (OORP, p.223)
  - Identifying a reusable patch between branches is a duplicate detection problem
- Most Valuable First (OORP, p.29)
  - Not every patch can or should be integrated.
- Write Tests to Understand (OORP, p.179)
  - When the intent of an integrated patch is unclear, write a focused test to document and explore its behavior.
- Tests: Your Life Insurance (OORP, p.149)
  - After integration, run or extend tests to ensure functional behavior holds in the target.

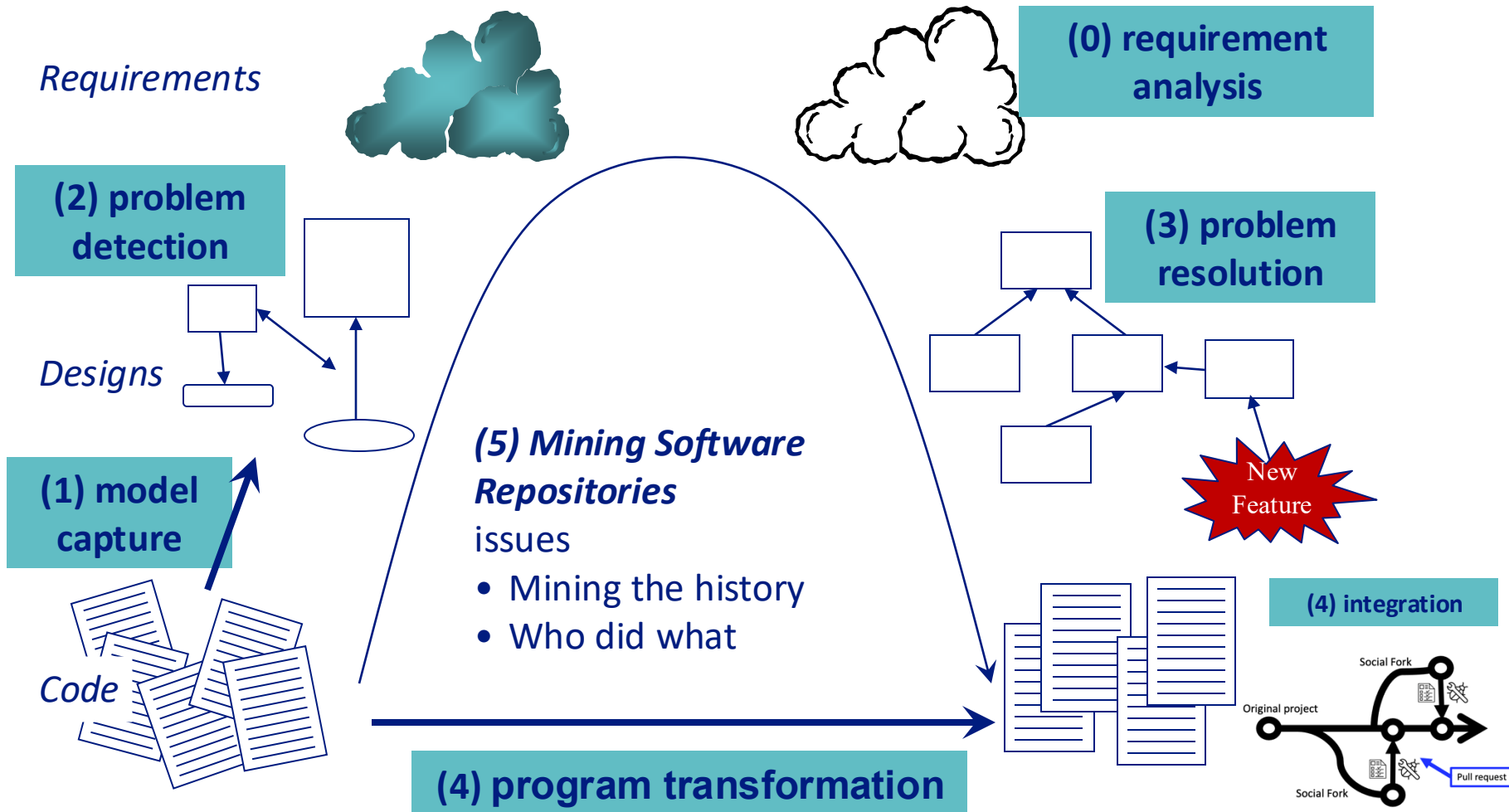
# 7. Mining Software Repositories (MSR)

- What are software repositories?
- Why should we mine Software repositories?
- What are some of the data sources of software engineering data?
- What are some of the existing tools we can use to mine software engineering data
- What can we learn from MSR





# The Reengineering Life-Cycle



# What is a Software Repository?

Artifacts produced and archived during software development

- Technical artifacts
- Social artifacts



# What is a Software Repository?

The screenshot shows the Apache Kafka GitHub repository page. Several elements are highlighted with blue boxes and connected by red arrows:

- Repository Name:** `apache / kafka` (Public)
- Watchers:** 1.1k
- Forks:** 11.3k
- Stars:** 21.5k
- Pull requests:** 953 Open, 11,016 Closed
- Latest Commit:** `ijuma KAFKA-13418` (Support key updates with TLS 1.3 (#11966)) 12 hours ago, 9,874 commits
- Contributors:** 884 contributors, including a highlighted version `1.7` and `+ 873 contributors`
- Languages:** Java 74.2%, Scala 22.7%, Python 2.7%, Shell 0.2%, Roff 0.1%, Batchfile 0.1%
- Files:** `config` (MINOR), `connect` (KAFKA), `core` (MINOR), `docs` (KAFKA), `examples`, `generator/src`, `gradle`, `jmh-benchmarks`, `licenses`
- Download Section:** **DOWNLOAD** 3.1.0 (Released January 24, 2022), 3.0.0 (Released September 21, 2021), 2.8.0 (Released April 19, 2021), 2.7.0 (Released Dec 21, 2020), 2.6.0 (Released Aug 3, 2020)

Red arrows indicate the flow of information from the repository name to the pull requests, from the pull requests to the latest commit, and from the contributors section to the download section.

Object-Oriented Reengineering.43

Apache Kafka is a distributed event store and stream-processing platform

# Why should we mine Software repositories?

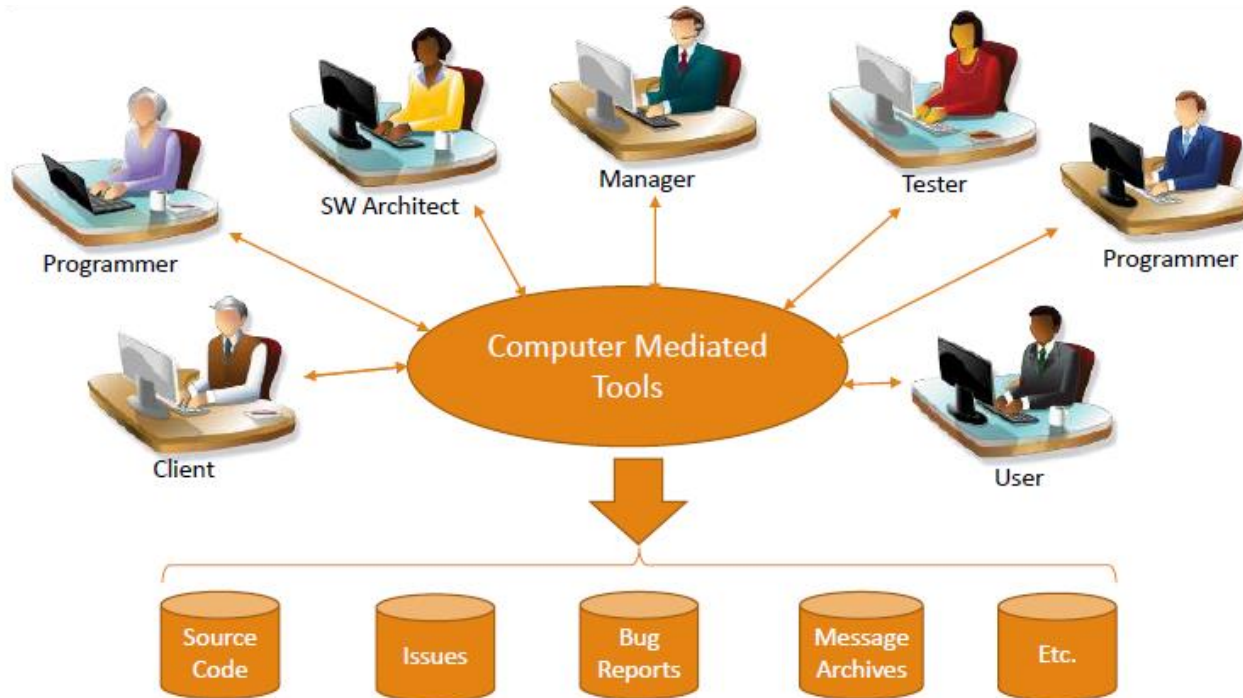
The goal ... is to improve software engineering practices by uncovering interesting and actionable information about software systems and projects using the vast amounts of software data

- + Understand software development process
- + Support and/or improve the maintenance of software systems
- + Exploit knowledge in planning the future development

- If the data analysis is not carefully designed and executed, it can lead to invalid conclusions



What are some of the data sources of software engineering data?



Current and historical artifacts and interactions are registered in software repositories

This list is not exhaustive.

**Qn. What are some of the additional software engineering data sources that can be maintained?**

What are some of the existing tools we can use to mine software engineering data?

### **PyDriller**

A Python framework that helps developers in analyzing Git repositories. With PyDriller you can easily extract information about **commits, developers, modified files, diffs, and source code**.

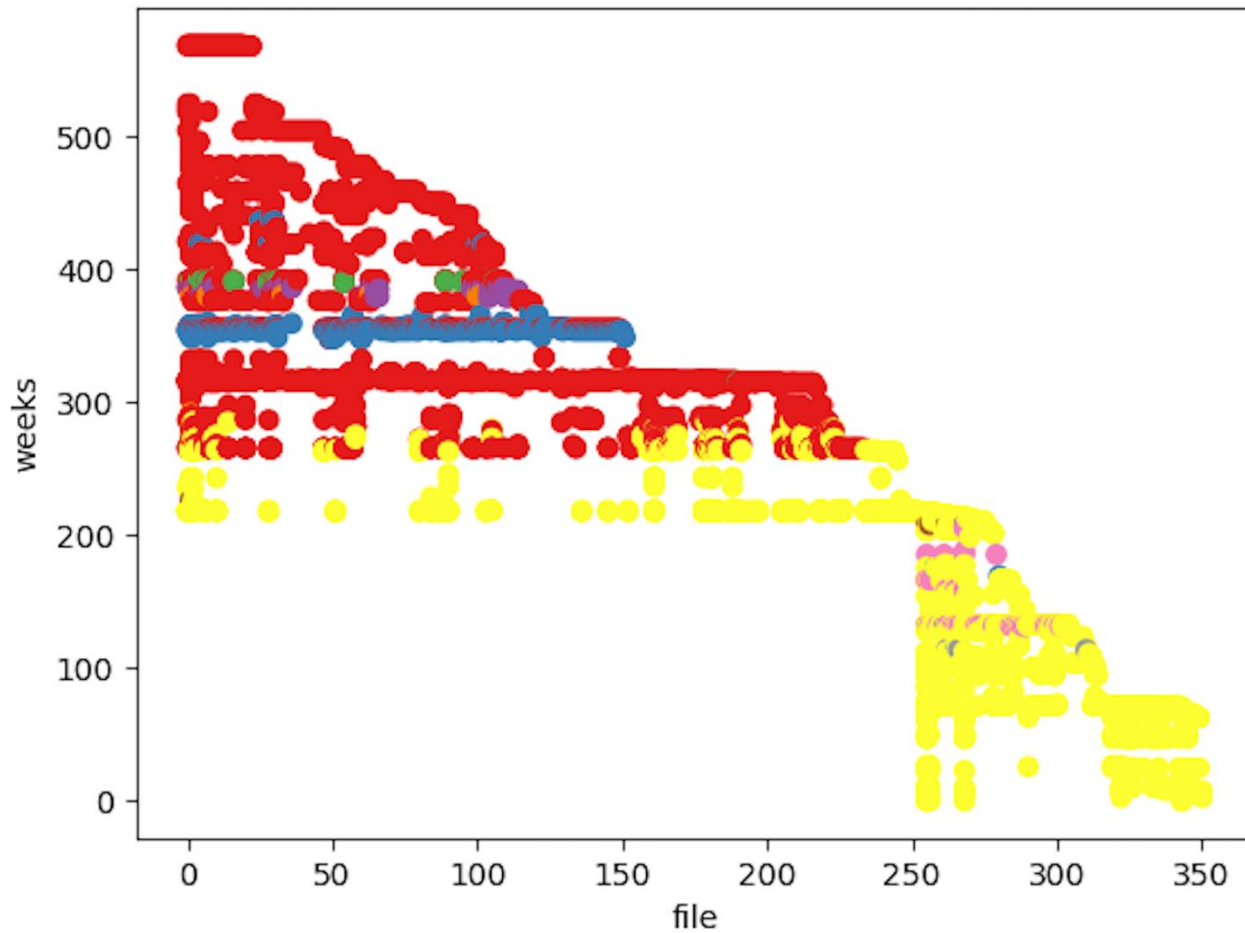
### **RepoDriller**

A Java framework that helps developers on mining software repositories. With it, you can easily extract information from any Git repository, such as commits, developers, modifications, diffs, and source codes, and quickly export CSV files.

### **Build your own tool/script**

Sometimes/ most of the times, you have to build your own tool or script to mine your own data

# Developers who touched files



# 7. Mining Software Repositories (MSR)

- **Learn from the Past (p.143)** – This pattern explicitly recommends reconstructing and analyzing past changes (using metrics, VCS history, etc.) to understand how the system evolved.
- **Compare Code Mechanically (p.227)** – Supports automated mining through diffs, metrics, and tooling before manual interpretation. Key for large-scale history analysis.
- **Detecting Duplicated Code (p.223)** – Identifies recurring fragments or reused fixes across repository history. Useful for spotting redundant effort or clone evolution.
- **Study the Exceptional Entities (p.107)** – Focuses on outliers (files with high churn, ownership concentration, or bug frequency). Repository mining surfaces such hotspots clearly.



# 9. Conclusion

## 1. Introduction

Software changes and that requires planning

## 2. Reverse Engineering

How to understand your code

## 3. Visualization

Scalable approach

## 4. Restructuring

How to Refactor Your Code

## 6. Dynamic Analysis (& Testing)

To be really certain

## 6 . Code Integration

How to resolve conflicts

## 7. Mining Software Repositories

Learn from the past

## 8. Conclusion



# Goals

*We will try to convince you:*

- Programs change!
- Reverse engineering forward engineering and reengineering are *essential activities* in the lifecycle of any successful software system. (And especially OO ones!)
- There is a large set of *lightweight tools and techniques* to help you with reengineering.
- Despite these tools and techniques, *people must do job* and they represent the most valuable resource.

