Génie Logiciel et Gestion de Projets

Evolution

Roadmap

- Evolution: definitions
- Re-engineering
- Legacy systems
- Reverse engineering
- Software Visualisation
- Re-engineering Patterns

Evolution: Definitions

Software Evolution

- Software evolution is
  - all programming activity that is intended to generate a new software version from an earlier operational version [Lehman&Ramil 2000]
  - the life of the software after its initial development cycle (or after the first delivery of the software system)
    - Any subsequent change to the software, such as bug fixes, adding new functionality, even modifying existing functionality and major architectural changes, is considered to be software evolution

Software Maintenance

- Software maintenance is
  - The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [IEEE Std 610.12-1999]
  - The software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity [ISO Std 12207]

Software Aging

- We need to learn how to reverse the effects of aging

  "Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable." [Parnas, 1994]

- Reasons why software ages
  - maintenance
  - ignorant surgery and architectural erosion
  - inflexibility from the start
  - insufficient or inconsistent documentation
  - deadline pressure
  - duplicated functionality (code duplication)
  - lack of modularity
  - ...
- Possible solution: restructuring/refactoring

Observations

- Software change is
  - unavoidable (inevitable)
  - unpredictable
  - expensive (costly)
  - difficult
  - poorly supported by tools, techniques, formalisms
  - underestimated by managers

Change is *unavoidable*

- Software change is inevitable
  - New requirements emerge when the software is being used
    - Even when it is being developed !
  - The business environment changes
  - Errors must be repaired
  - New computers and equipment are added to the system
  - The performance or reliability of the system may have to be improved
  - New technology is being used (new standards, new OS, new software versions, ...)

Change is *unpredictable*

- The main difficulty in software engineering is that changes cannot be anticipated at design time
  "The fundamental problem, supported by 40 years of hard experience, is that many changes actually required are those that the original designers cannot even conceive of." [Bennett&Rajlich 2000]
- Software engineers are very bad in predicting which classes will change upon evolution

Change is *expensive*

- Software maintenance is costly
  - The majority of the software budget in large companies is devoted to adapting (i.e., evolving) existing software rather than developing new software
  - Maintenance costs exceed estimated costs with a factor between 2 and 100 (depending on the application)

Change is *difficult*

- Software maintenance is difficult
  - Maintenance incorporates aspects of all other phases of the software life-cycle
- Nevertheless, even today, many organizations assign maintenance to
  - Unsupervised beginners, and
  - Less competent programmers
- Tools used by the maintenance programmer to find faults?
  - The error report provided by the user
  - The source code
  - And often nothing else

Laws of Software Evolution

Laws of Software Evolution

- Program evolution dynamics is the study of the processes of software system change
- After major empirical studies, Lehman and Belady proposed a number of "laws" that seem to be applicable to all evolving software systems.
- “Laws”: Reflect the cooperative activity of many individuals and organisational behaviour
  - Influenced by thermodynamics
  - Reflect established observations and empirical evidence
  - “An emerging theory of software process and software evolution”, i.e., work in progress.

Laws of Software Evolution

- Lehman's laws of software evolution
  - Based on the evolution of IBM 360 mainframe OS over a period of 30 years
- References:
  - Lehman M.M. and Belady L.A. (Eds.), 1985 *Software Evolution – Processes of Software Change*, Academic Press, London (Free download from wiki.ercim.org/wg/SoftwareEvolution)
  - M. M. Lehman. *Laws of Software Evolution Revisited*. Lecture Notes in Computer Science 1149, pp. 108-124, Springer Verlag, 1997

Laws

- Law 1: Continuing change
- Law 2: Increasing complexity
- Law 3: Self regulation
- Law 4: Conservation of organisational stability
- Law 5: Conservation of familiarity
- Law 6: Continuing growth
- Law 7: Declining quality
- Law 8: Feedback system

Law 1: Continuing change

- "*An E-type program that is used in a real-world environment must be continually adapted, else it becomes progressively less satisfactory*"
- Reasons:
  - Evolution of the environment ("operational domain")
  - Hence, increasing mismatch between the system and its environment
- Remark
  - "E-type programs": E for Evolutionary
    - "Real-world problem": Stakeholders mainly humans
    - Requirements and environment continuously evolving
    - Continuous need for change

Law 2: Increasing complexity

- *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*
- Reasons:
  - Small changes are applied in a step-wise process
  - Each 'patch' makes sense locally, not globally
- Effort needed to reconcile accumulated changes with the overall performance goals
  - techniques like refactoring and performance optimisation are needed.

Applicability of Lehman's laws

- Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.
- It is not (yet) clear whether they are applicable to
  - Shrink-wrapped software products
  - Systems that incorporate a significant number of "off-the-shelf" components or external libraries
  - Small organisations
  - Medium sized systems
  - Open source software

References

- [Lehman&Ramil 2000] M. M. Lehman, J. F. Ramil. Effort Estimation from Change Records of Evolving Software, 2000
- [Parnas 1994] D. L. Parnas. Software Aging. Invited plenary talk. Proc. Int. Conf. on Software Engineering (ICSE '94), pp. 279-287, IEEE Computer Society Press, 1994
- [Bennett&Rajlich 2000] K. H. Bennett, V. T. Rajlich. Software Maintenance and Evolution: a Roadmap. The Future of Software Engineering, ACM Press, 2000

Legacy Systems

Legacy System

- "legacy":
  A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor.
  --- *Oxford English Dictionary*
- A legacy system is a piece of software that
  - you have inherited
  - is valuable to you

Problems with Legacy Systems

- original developers are no longer available
- outdated development methods are used
- extensive patches and modifications have been made
- documentation is missing or outdated

  so that further evolution may be prohibitively expensive

- The difference between legacy systems and adaptable software is the *ability to change*.
- Solution: *migration* to a new system

Object-oriented Legacy Systems

- = successful OO systems whose architecture and design no longer responds to changing requirements
- Compared to traditional legacy systems
  - The symptoms and the source of the problems are the same
  - The technical details and solutions may differ
- OO techniques promise better
  - flexibility,
  - reusability,
  - maintainability,
  - ...

do not come for free....

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
- **process symptoms**
  - too long to turn things over to production
  - need for constant bug fixes
  - maintenance dependencies
  - difficulties separating products
- **code symptoms**
  - duplicated code
  - code smells

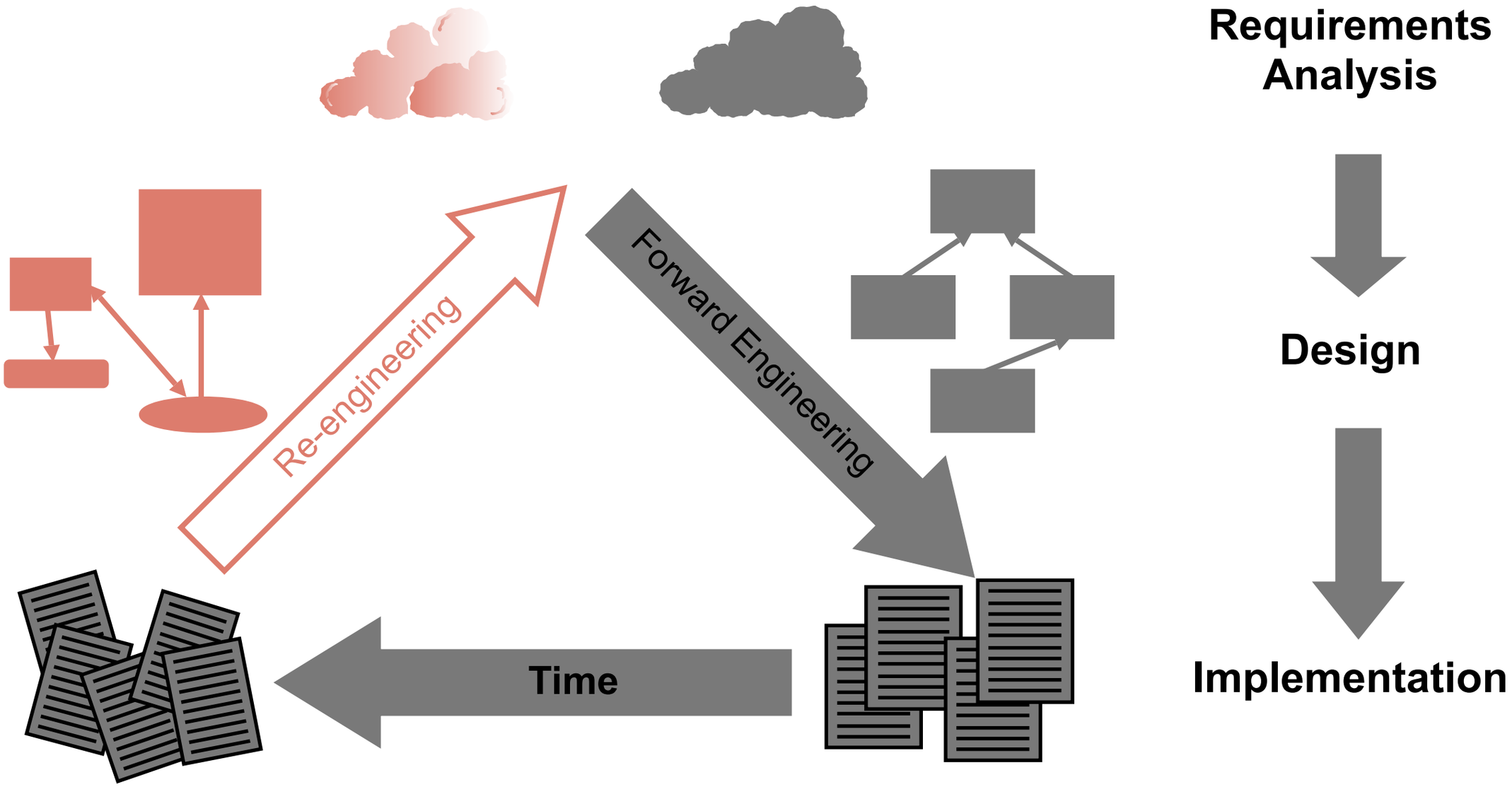
Common Problems

- **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 and 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

Re-engineering

Re-engineering

Software development is more than forward engineering alone...



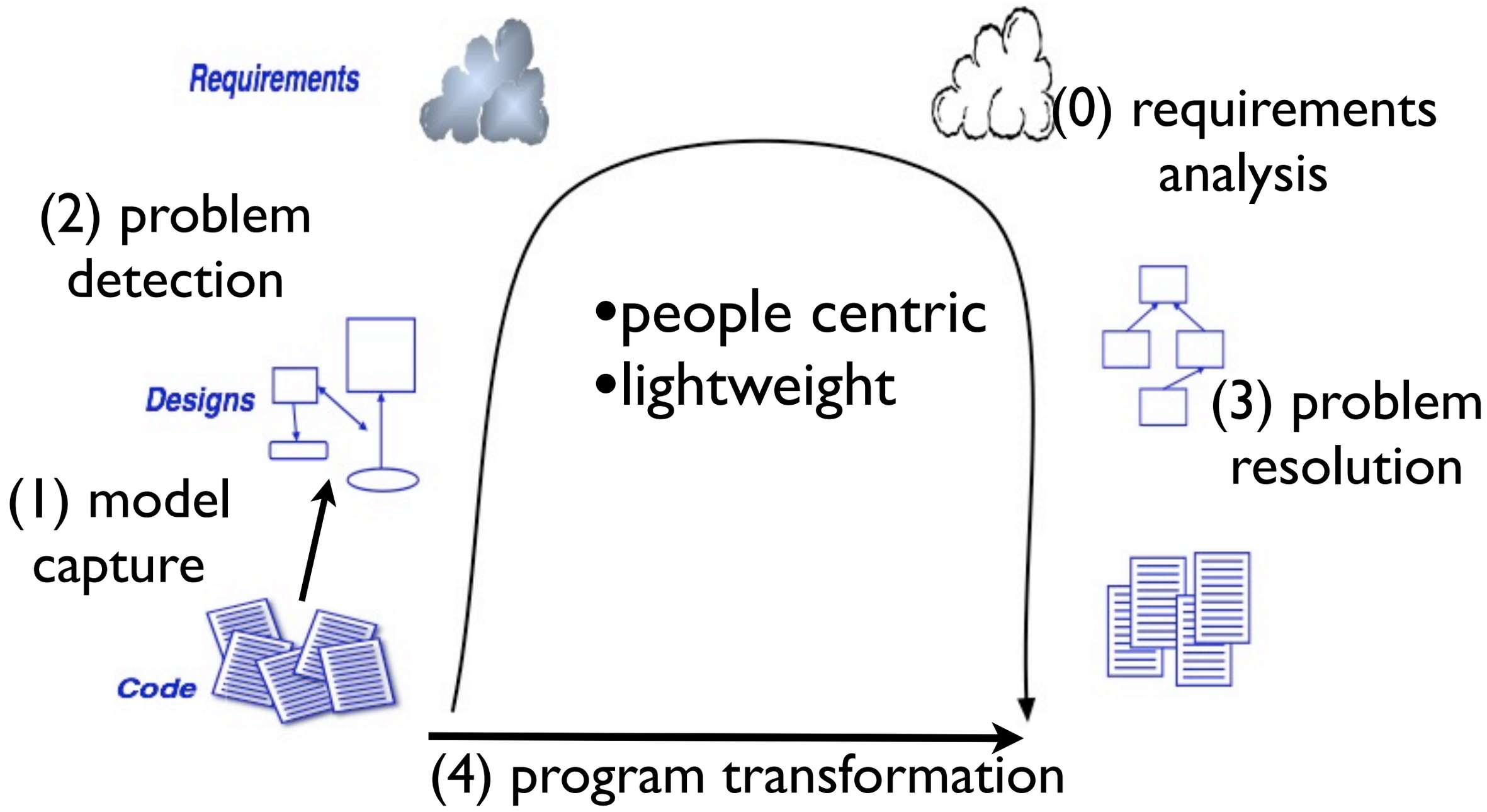
Reengineering

- "Reorganising and modifying existing software systems to make them more maintainable"
  - Re-structuring or re-writing part or all of a legacy system without changing its functionality
  - Applicable where some but not all sub-systems of a larger system require frequent maintenance
  - Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented

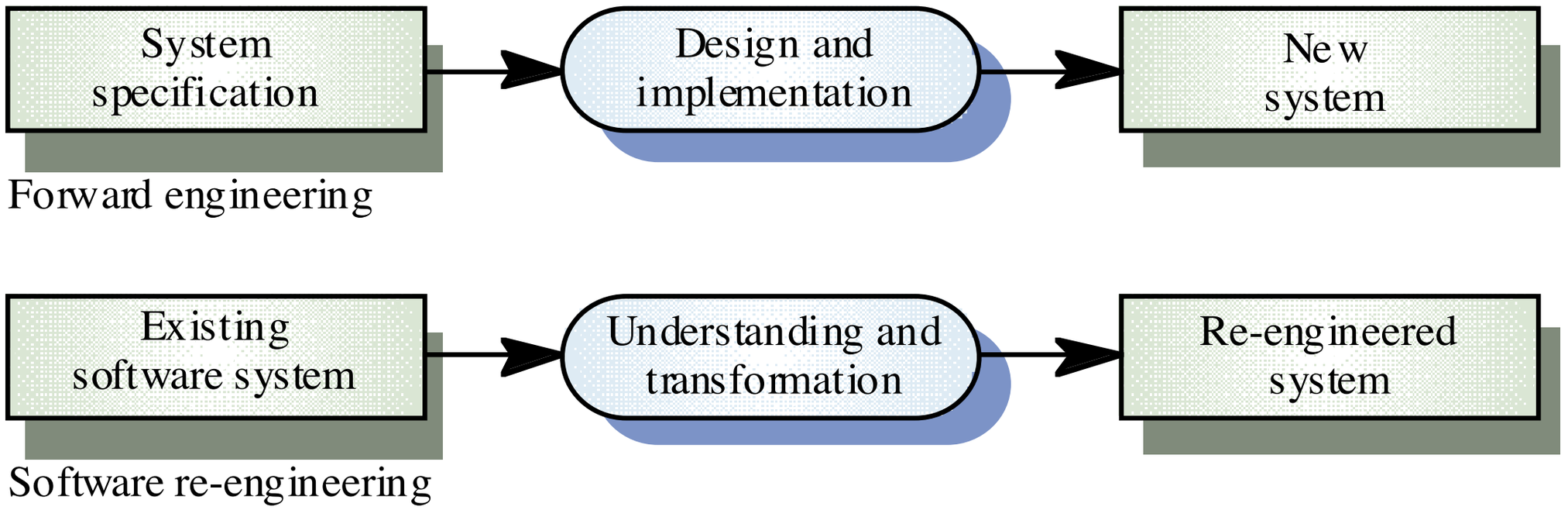
Advantages of Re-engineering

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

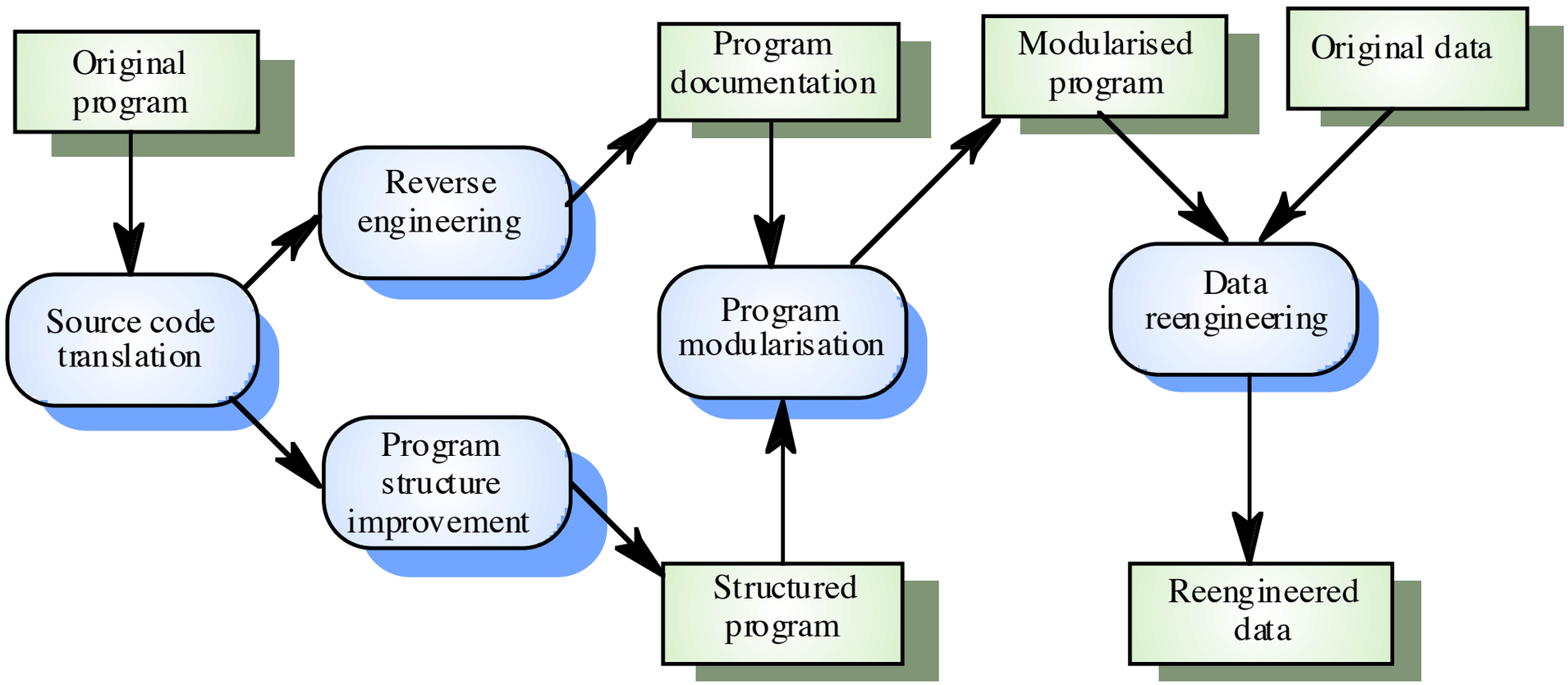
Re-engineering Life-cycle



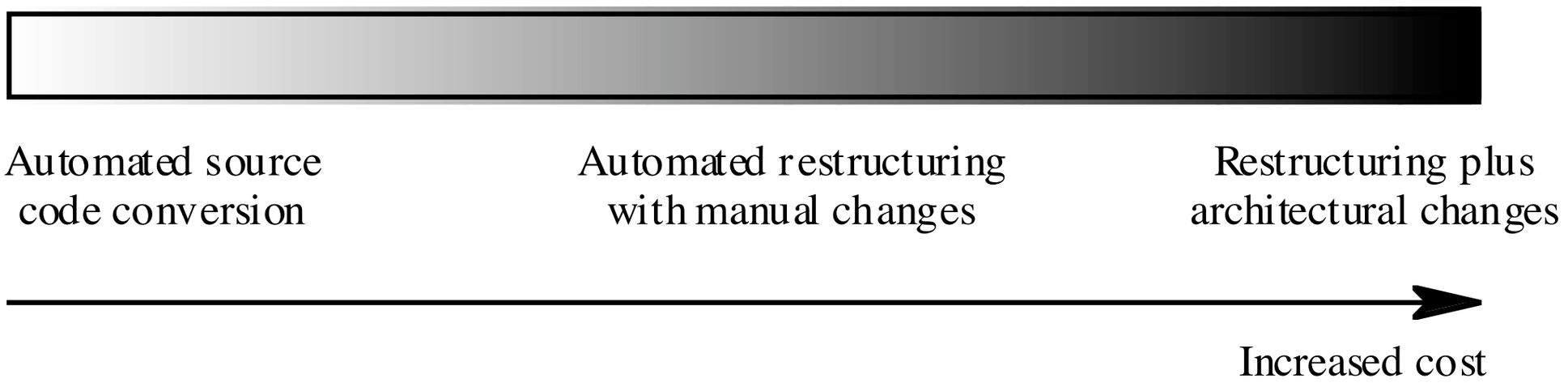
Forward and Re-engineering



Re-engineering Process



Re-engineering Approaches



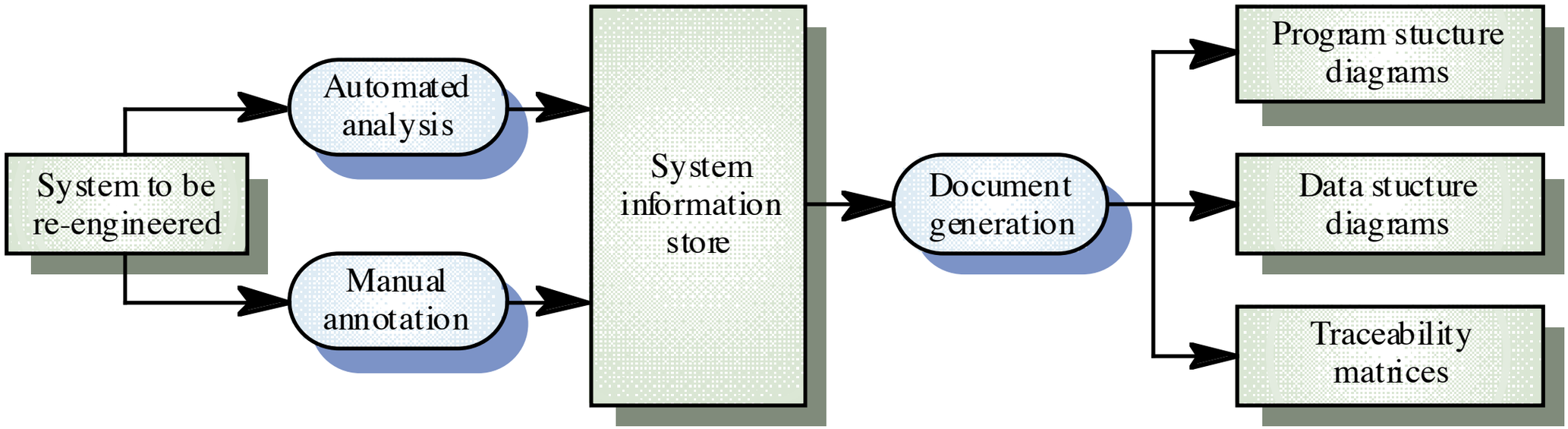
Reverse Engineering

- *Reverse engineering is the process of analyzing a subject system*
  - to identify the system's components and their interrelationships
  - create representations of the system in another form or at a higher level of abstraction
- Motivation
  - Understanding other people's code (cf. newcomers in the team, code reviewing, original developers left, ...)

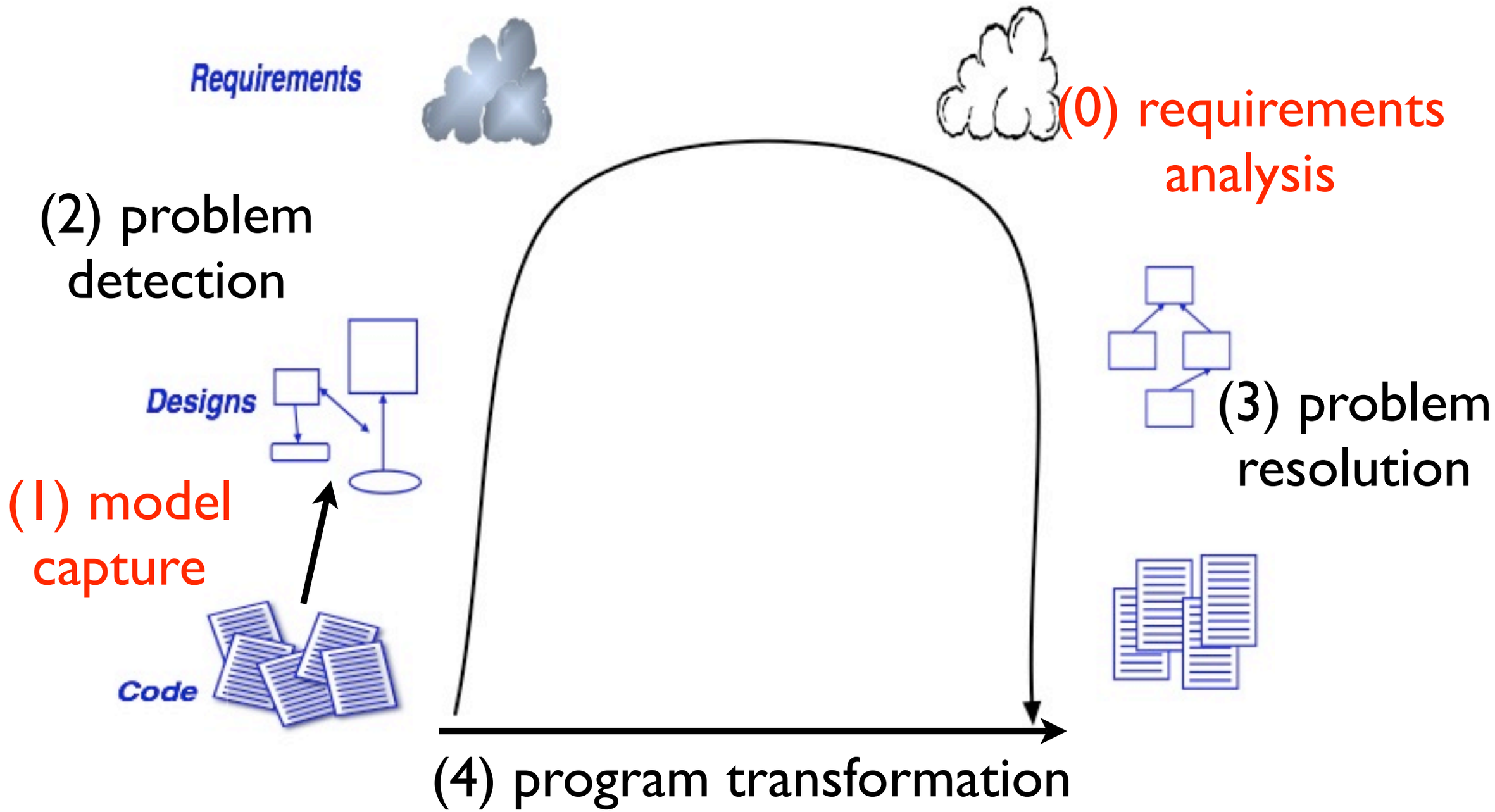
Reverse engineering

- May be part of a re-engineering process, but may also be used to respecify a system for reimplementation
  - Builds a program database and generates information from this
  - Program understanding tools (browsers, cross-reference generators, etc.) may be used in this process

Reverse Engineering Process



Re-engineering Life-cycle



Software Visualisation

- Motivation:
  - *Problem: Reverse Engineering*
    - We want to understand a (legacy) software system that is
      - Unknown to us
      - Very large and complex
      - Domain- and language-specific
      - Poorly documented
      - In bad shape
  - *Solution*
    - Construct a "mental model" of the system
    - Using visualisation techniques

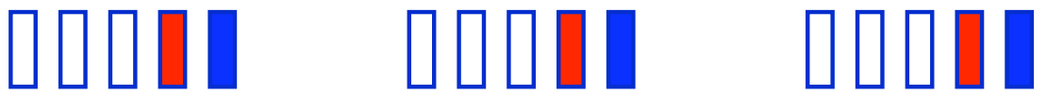
Software Visualisation Tools

- Metrics
  - Eclipse Metrics plug-in
    - includes a graphical dependency analyzer
- Lightweight approaches
  - CodeCrawler

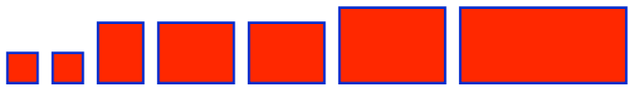
CodeCrawler

- Tool for visualising source code metrics in Smalltalk
- Developed by Michele Lanza, University of Bern
- Main challenge
  - The information needed to reverse engineer a legacy software system resides at various levels
  - We need to obtain and combine
    - Coarse-grained information about the whole system

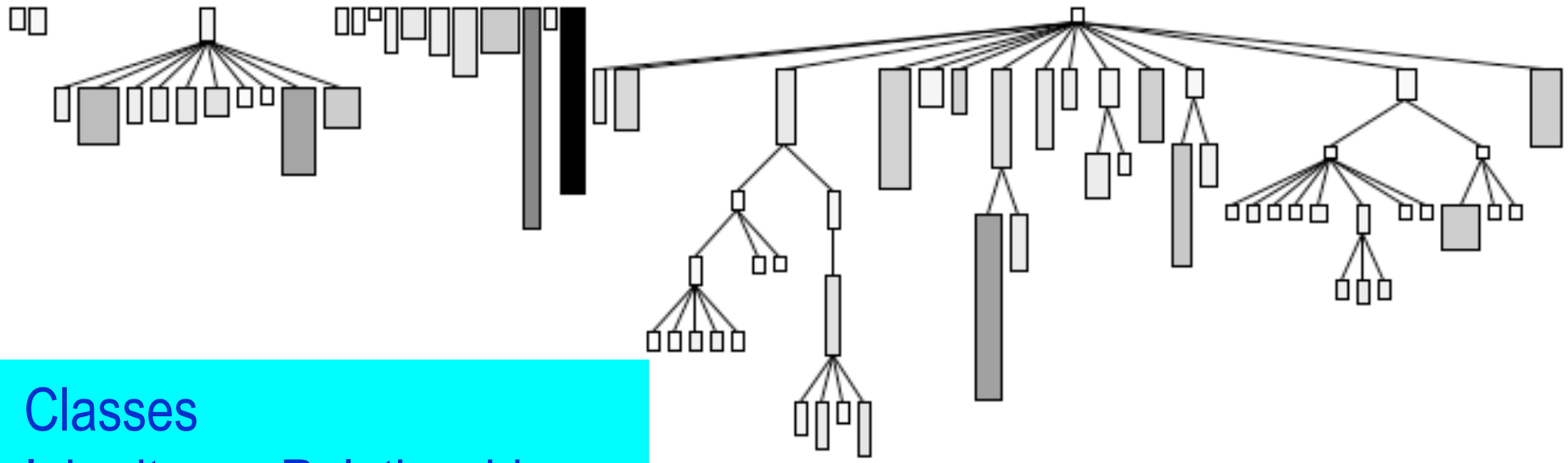
    - Fine-grained information about specific parts



    - Evolutionary information about the past of the system



CodeCrawler: Visualising Quality



Nodes = Classes
Edges = Inheritance Relationships

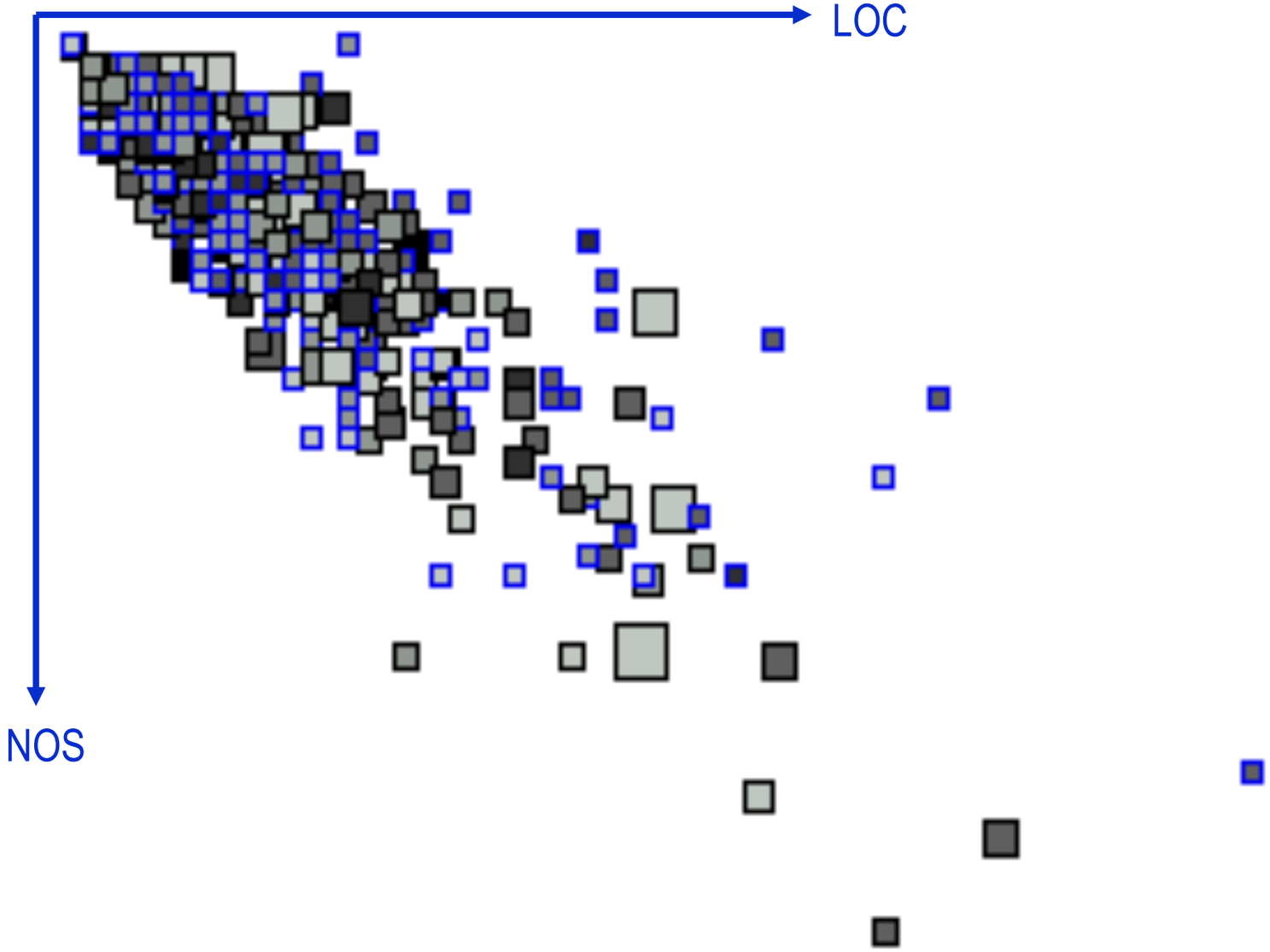
Width = Number of Attributes
Height = Number of Methods
Color = Number of Lines of Code

CodeCrawler: Coarse-grained

- Coarse-grained software visualisation
  - What is the size and the overall structure of the system?
- Coarse-grained reverse engineering goals
  - Gain an overview in terms of size, complexity, and structure
  - Asses the overall quality of the system
  - Locate and understand important (domain model) hierarchies
  - Identify large classes, exceptional methods, dead code, etc.
  - …
- Proposed visualisation
  - Method efficiency correlation view

CodeCrawler: Coarse-grained example

Method Efficiency Correlation View



Nodes:	Methods
Edges:	-
Size:	Number of method parameters
Position X:	Number of lines of code
Position Y:	Number of statements

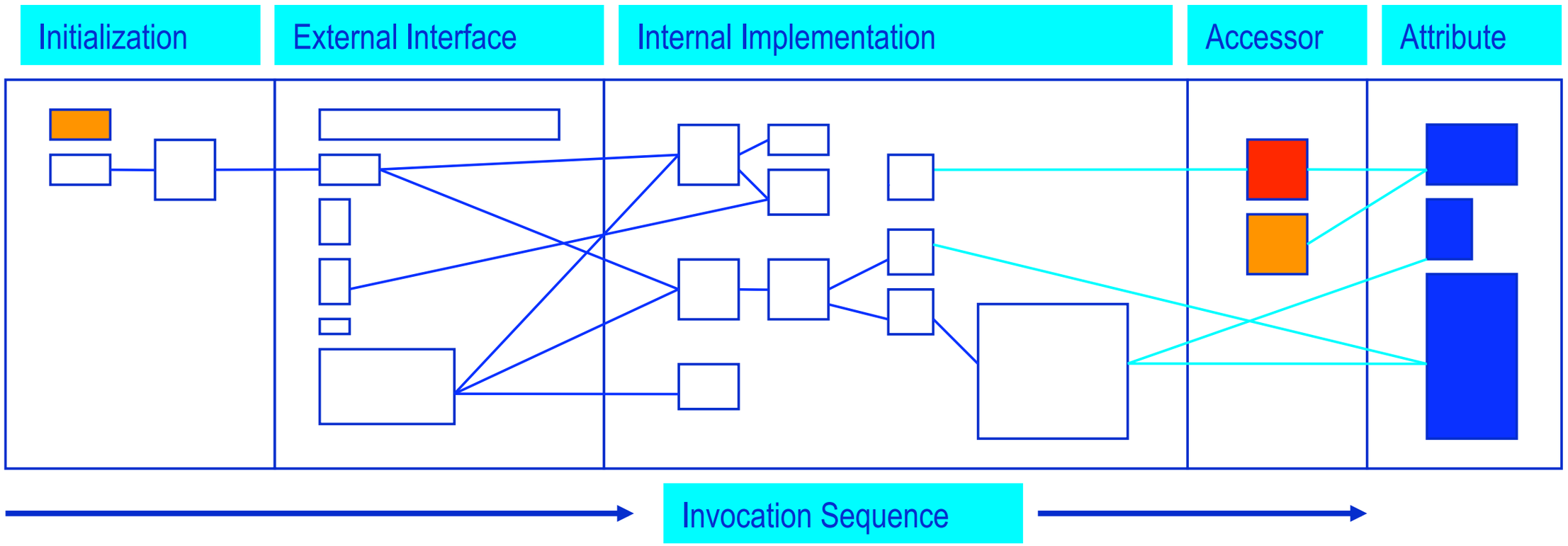
Goals:

- Detect overly long methods
- Detect “dead” code
- Detect badly formatted methods
- Get an impression of the system in terms of coding style
- Know the size of the system in # methods

CodeCrawler: Fine-grained

- Fine-grained software visualisation
  - What is the internal structure of the system and its elements?
- Fine-grained reverse engineering goals
  - Understand the internal implementation of classes and class hierarchies
  - Detect coding patterns and inconsistencies
  - Understand class/subclass roles
  - Identify key methods in a class
  - …
- Proposed visualisation
  - Class blueprint

CodeCrawler: Fine-grained Example

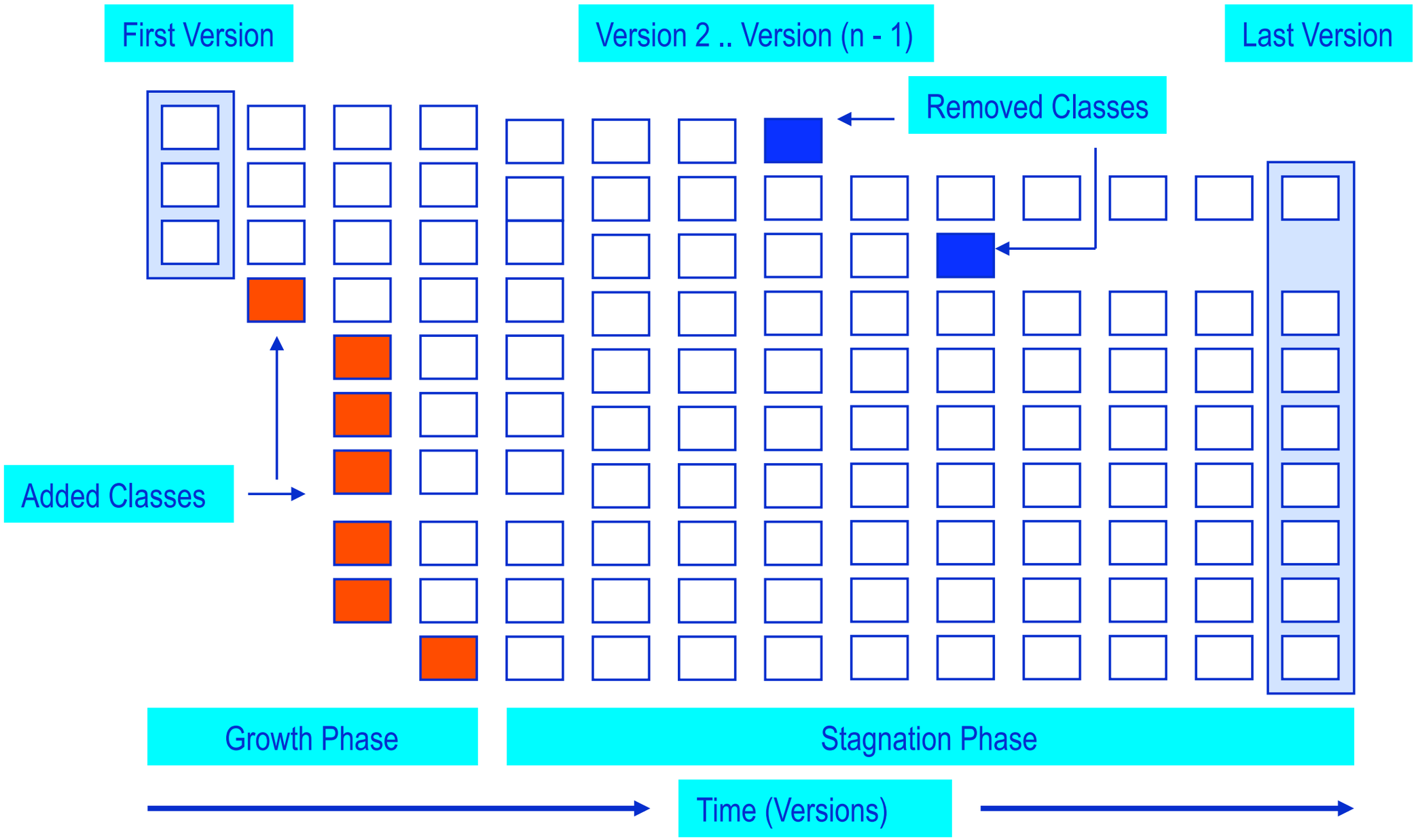


- The class is divided into 5 layers
  - Nodes
    - Methods, Attributes, Classes
  - Edges
    - Invocation, Access, Inheritance
- The method nodes are positioned according to
  - Layer
  - Invocation sequence

CodeCrawler: Evolutionary

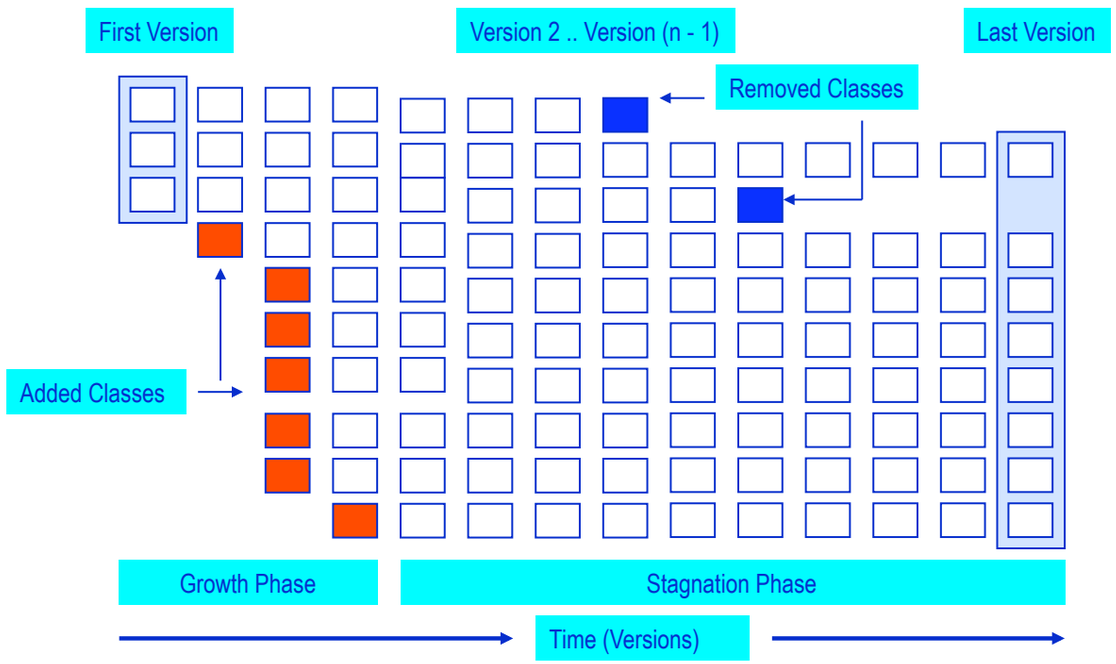
- Evolutionary Software Visualisation
  - How did the software system become like that?
- Evolutionary reverse engineering goals:
  - Understand the evolution of OO systems in terms of size and growth rate
  - Understand at which time an element, e.g., a class, has been added or removed from the system
  - Understand the evolution of single classes
  - Detect patterns in the evolution of classes
  - …
- Proposed visualisation
  - Evolution Matrix

CodeCrawler: Evolutionary Example

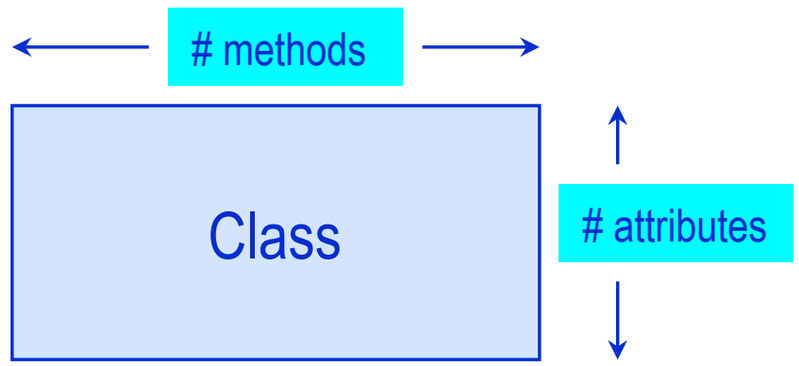


CodeCrawler: Evolution Matrix

- The Evolution Matrix reveals patterns
  - The evolution of the whole system (versions, growth and stagnation phases, growth rate, initial and final size)
  - The life-time of classes (addition, removal)



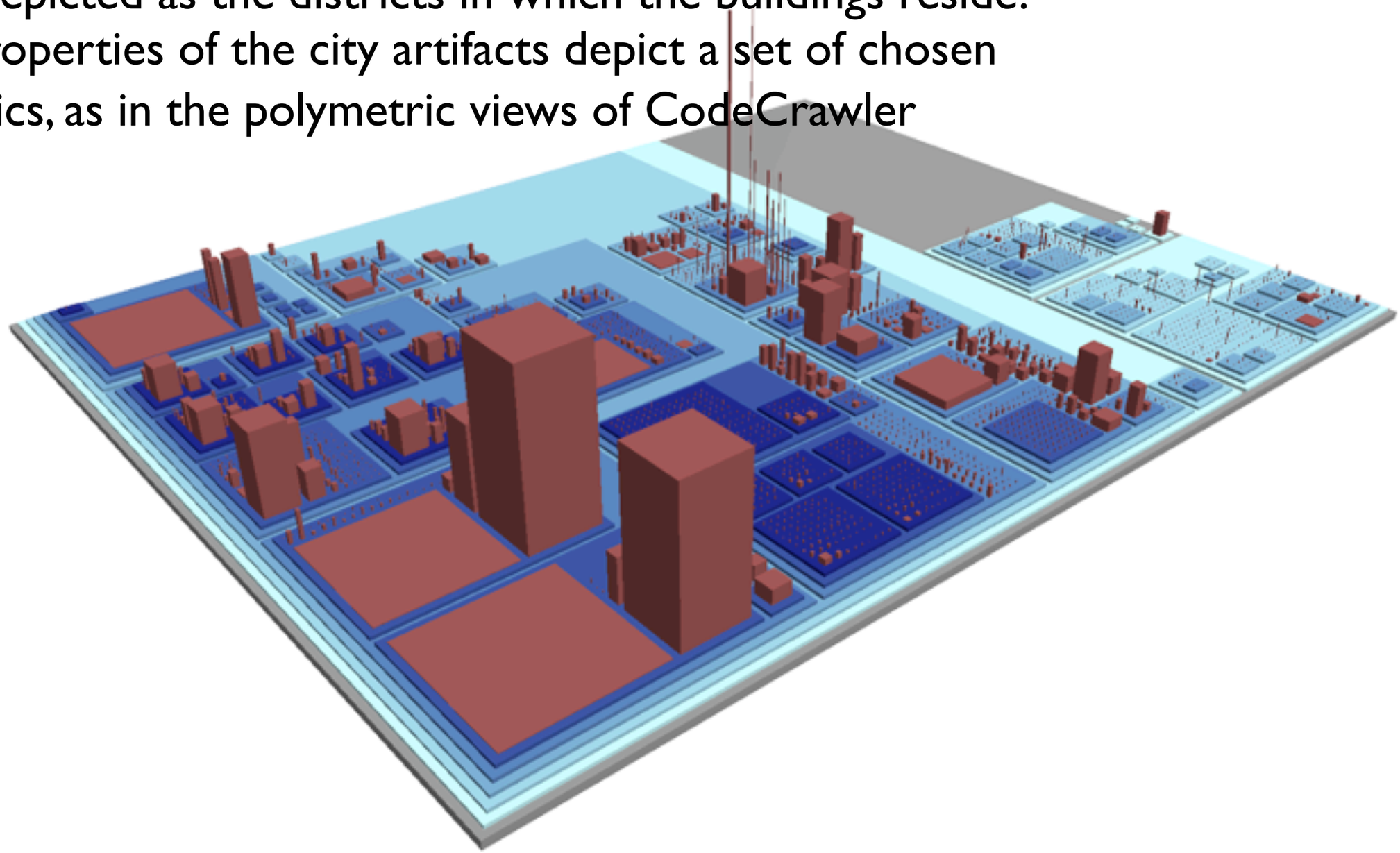
Moreover, we enrich the evolution matrix view with metric information



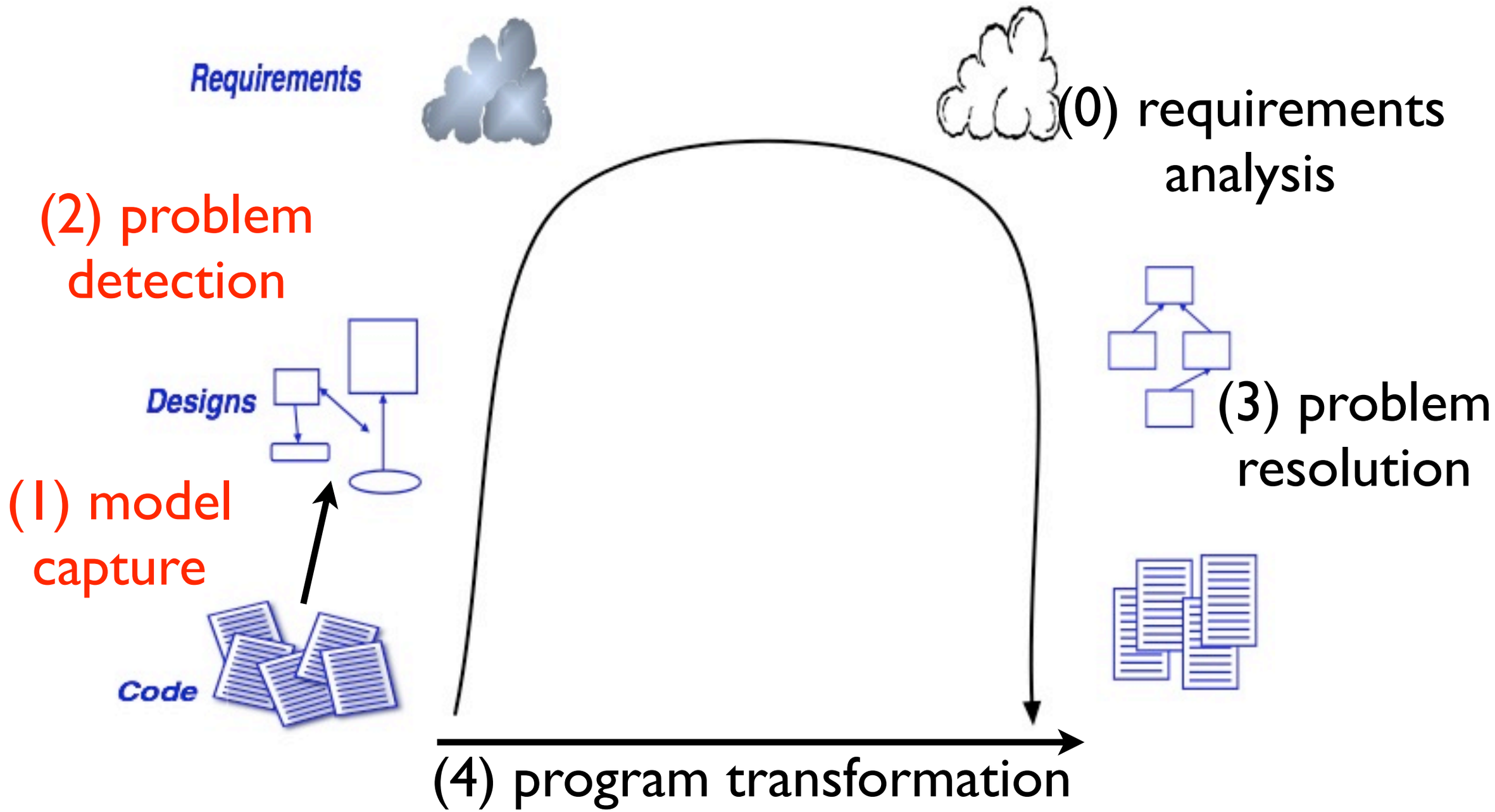
This allows us to see patterns in the evolution of classes

CodeCity: 3D Visualisation

- Environment for software analysis, in which software systems are visualized as interactive, navigable 3D cities.
- The classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside.
- The visible properties of the city artifacts depict a set of chosen software metrics, as in the polymetric views of CodeCrawler



Re-engineering Life-cycle



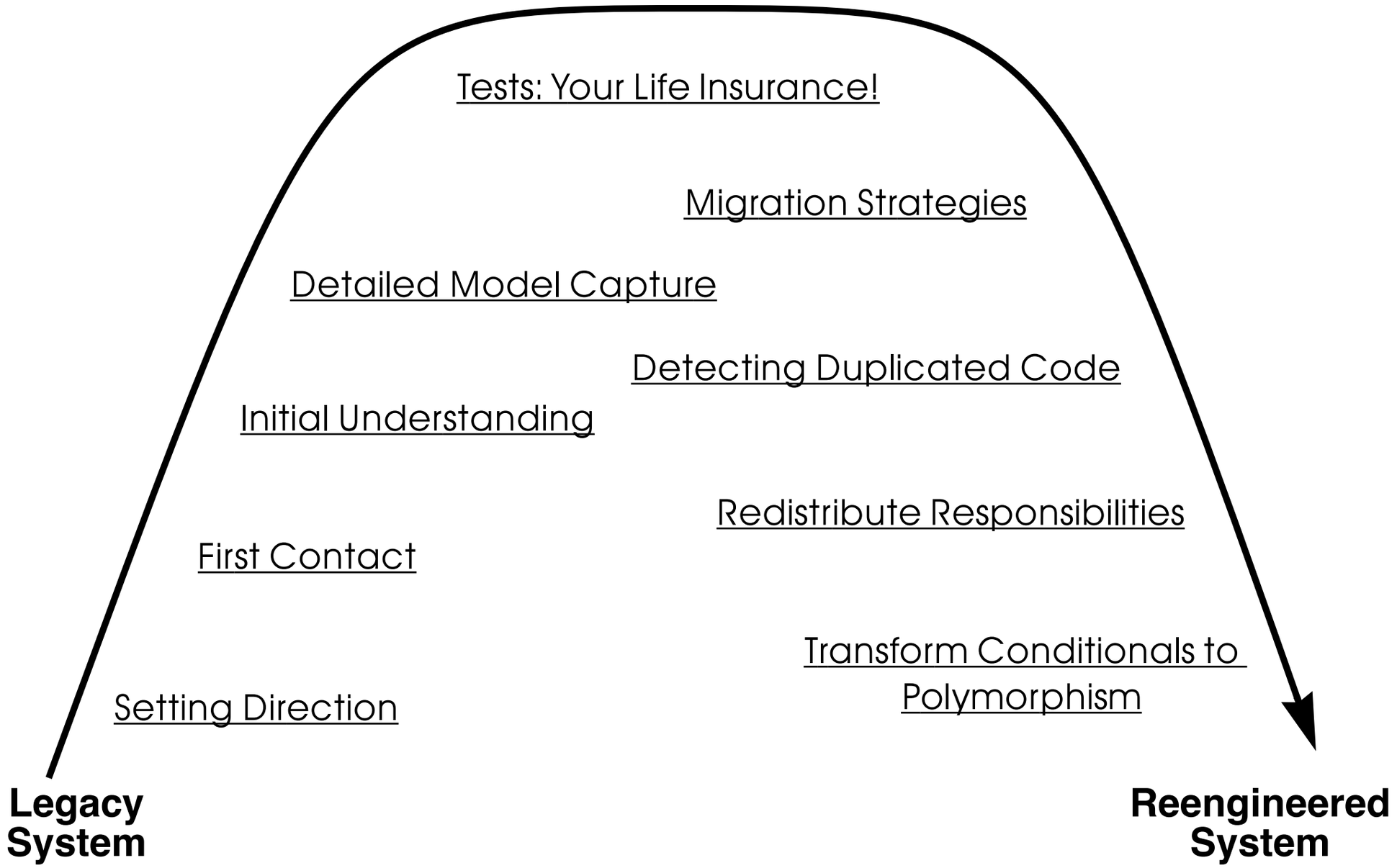
Reengineering Patterns

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2008/2009

Reengineering Patterns

- You might have noticed: Reengineering is difficult.
- Reengineering patterns capture
  - reengineering of *legacy object-oriented systems*
    - planning a reengineering project
    - reverse-engineering
    - problem detection
    - migration strategies
    - software redesign

Patterns in the lifecycle



Reengineering Pattern Structure

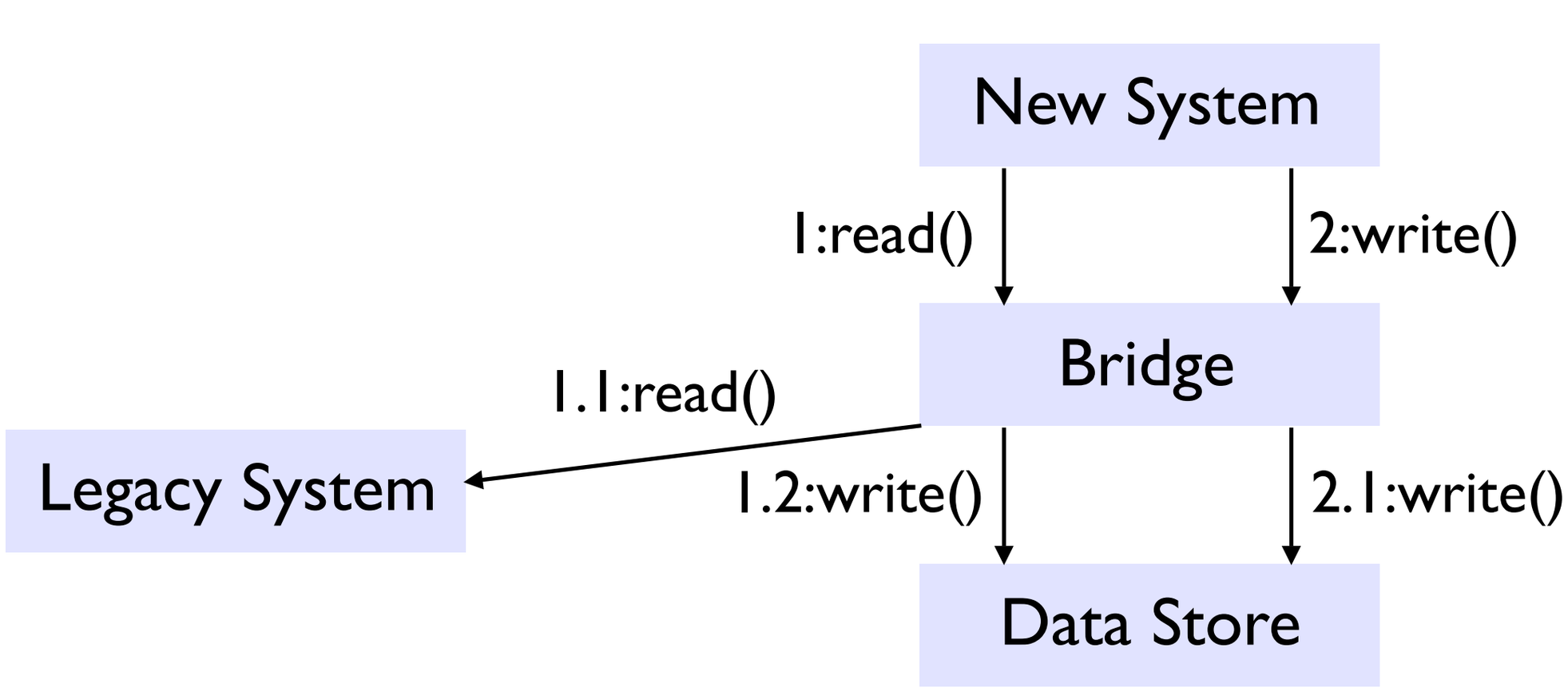
- Name
- Problem(s) it addresses
- Solution
- List of trade-offs
  - the pros
  - the cons
  - the difficulties
- Example
- Reasons for applying the pattern
- Related Patterns

Make a bridge to the new town

- Intent: Migrate data from a legacy system by running the new system in parallel, with a bridge in between them.
- Problem: how do you incrementally migrate data from a legacy system to its replacement while the two systems are running in tandem?

Make a bridge to the new town

Solution: Make a (data) bridge that will incrementally transfer data from the legacy system to the replacement system as new components are ready to take the data over from their legacy counterparts.



Make a bridge to the new town

- Pros
  - you can start using the new system without migrating all the legacy data
- Cons
  - can be tricky to implement
  - once some data is transferred it can be hard to go back
  - the data bridge will add performance overhead
- Known uses, rationale, related patterns

Summary

- Software "maintenance" is really continuous development.
- Object-oriented software also suffers from legacy symptoms
- Reverse 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. Common, lightweight techniques can be applied to keep software healthy.
- Despite these tools and techniques, people must do the job and represent the most valuable resource.

References

- Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz, Object-Oriented Reengineering Patterns, Morgan-Kaufmann, 2002
- CodeCity 3D engine http://www.inf.unisi.ch/phd/wettel/codecity.html
- Michele Lanza and Stéphane Ducasse. CodeCrawler--An Extensible and Language Independent 2D and 3D Software Visualization Tool. In Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series p. 74--94, Franco Angeli, Milano, 2005.
- Michele Lanza and Stéphane Ducasse and Harald Gall and Martin Pinzger. CodeCrawler --- An Information Visualization Tool for Program Comprehension. In Proceedings of ICSE 2005, p. 672--673, ACM Press, 2005.