Software Design and Evolution

1. Introduction

Oscar Nierstrasz

UNIVERSITÄT BERN

Roadmap

> Overview
> Laws of Software Evolution
> Reflection and Metaprogramming
> Smalltalk
> Reverse and Reengineering

SDE

Lecturers	Oscar Nierstrasz, Mircea Lungu
Assistants	Jorge Ressia
Lectures	IWI 001, Wednesdays @ 10h15-12h00
Exercises	IWI 001, Wednesdays @ 12h00-13h00
WWW	scg.unibe.ch/teaching/sde/

Roadmap

> **Overview**
> Laws of Software Evolution
> Reflection and Metaprogramming
> Smalltalk
> Reverse and Reengineering

Goals of this course

Understanding:

> how and why software evolves
> reflection and metaprogramming
> how to analyze evolving software
> how to enable graceful software evolution

Course Schedule (tentative)

Week	*Date*	*Lesson*
1	21-Sep-11	**Introduction to Software Design and Evolution**
2	28-Sep-11	**Smalltalk: A Reflective Language and System**
3	5-Oct-11	**Understanding Classes and Metaclasses**
4	12-Oct-11	**Reflection and Metaprogramming**
5	19-Oct-11	**Model-driven Development / Magritte *(Lukas Renggli)***
6	26-Oct-11	**Software Assessment *(Tudor Girba)***
7	2-Nov-11	***LAB: analyzing systems with Moose***
8	9-Nov-11	**Reverse Engineering and Architectural Extraction**
9	16-Nov-11	**Metrics and Problem Detection**
10	23-Nov-11	**Dynamic Analysis**
11	30-Nov-11	**Mining Software Repositories**
12	7-Dec-11	**Software Visualization**
13	14-Dec-11	**Software Ecosystems**
14	21-Dec-11	**Final exam**

Roadmap

- > Overview
- > **Laws of Software Evolution**
- > Reflection and Metaprogramming
- > Smalltalk
- > Reverse and Reengineering

What is a 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*, and
- is *valuable* to you

Typical **problems** with legacy systems:

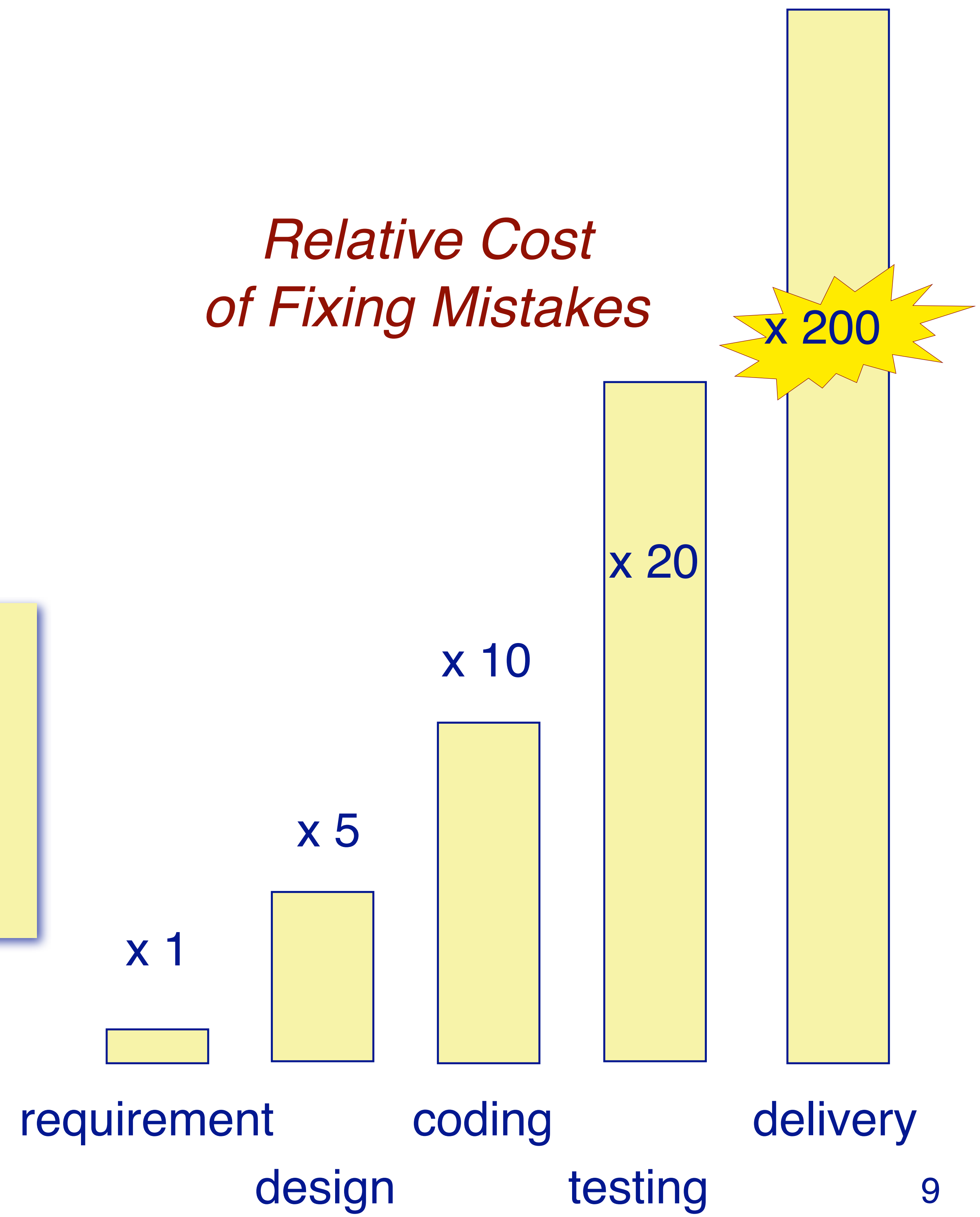
- original developers *not available*
- *outdated* development methods used
- extensive patches and *modifications* have been made
- *missing* or outdated documentation

⇒ *so, further evolution and development may be prohibitively expensive*

Software Maintenance - Cost

Relative Maintenance Effort
Between 50% and 75% of global effort is spent on "maintenance" !

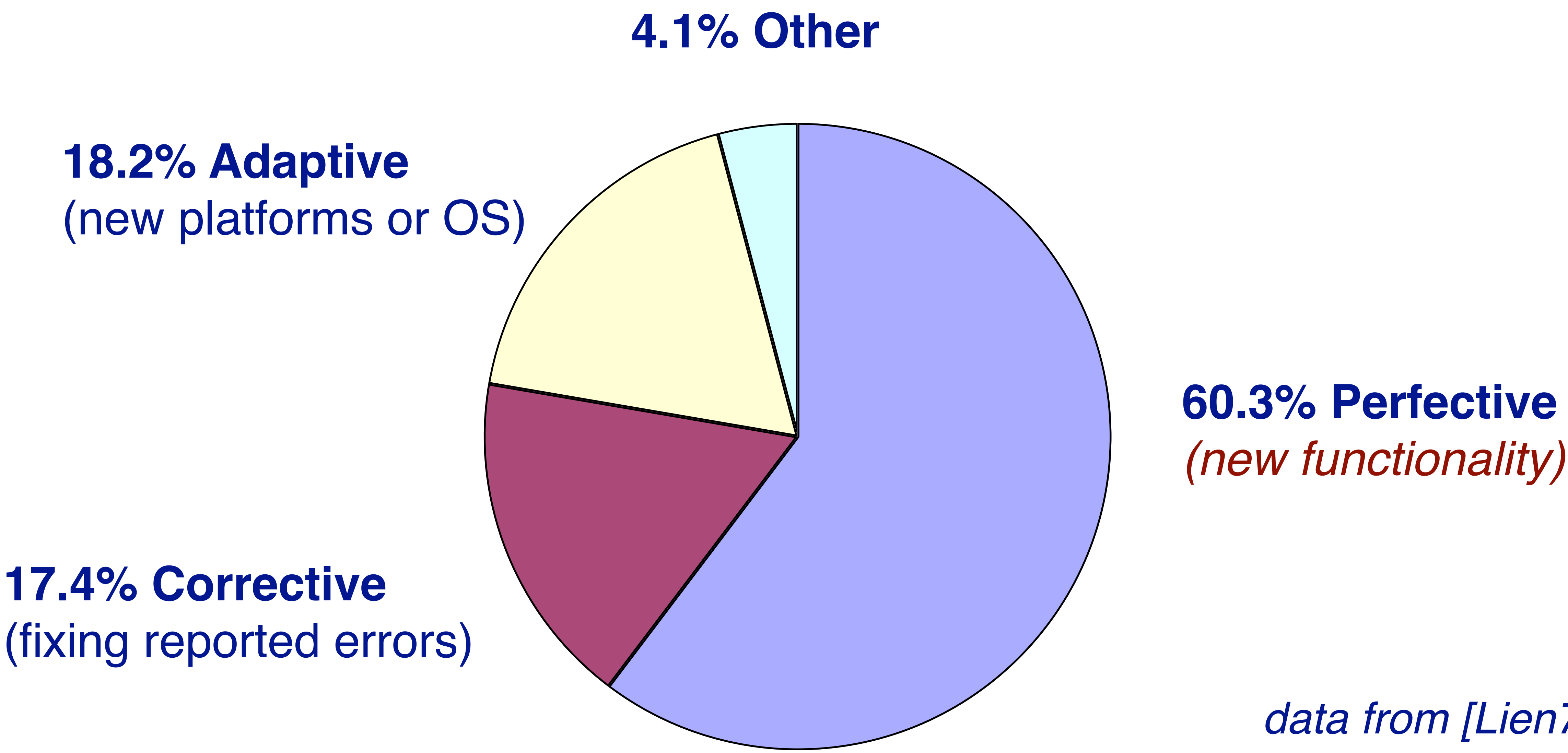
Relative Cost of Fixing Mistakes



Solution ?

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

Continuous Development



data from [Lien78a]

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…

What about Objects ?

Object-oriented legacy systems

- = successful OO systems whose architecture and design no longer responds to changing requirements

Compared to traditional legacy systems

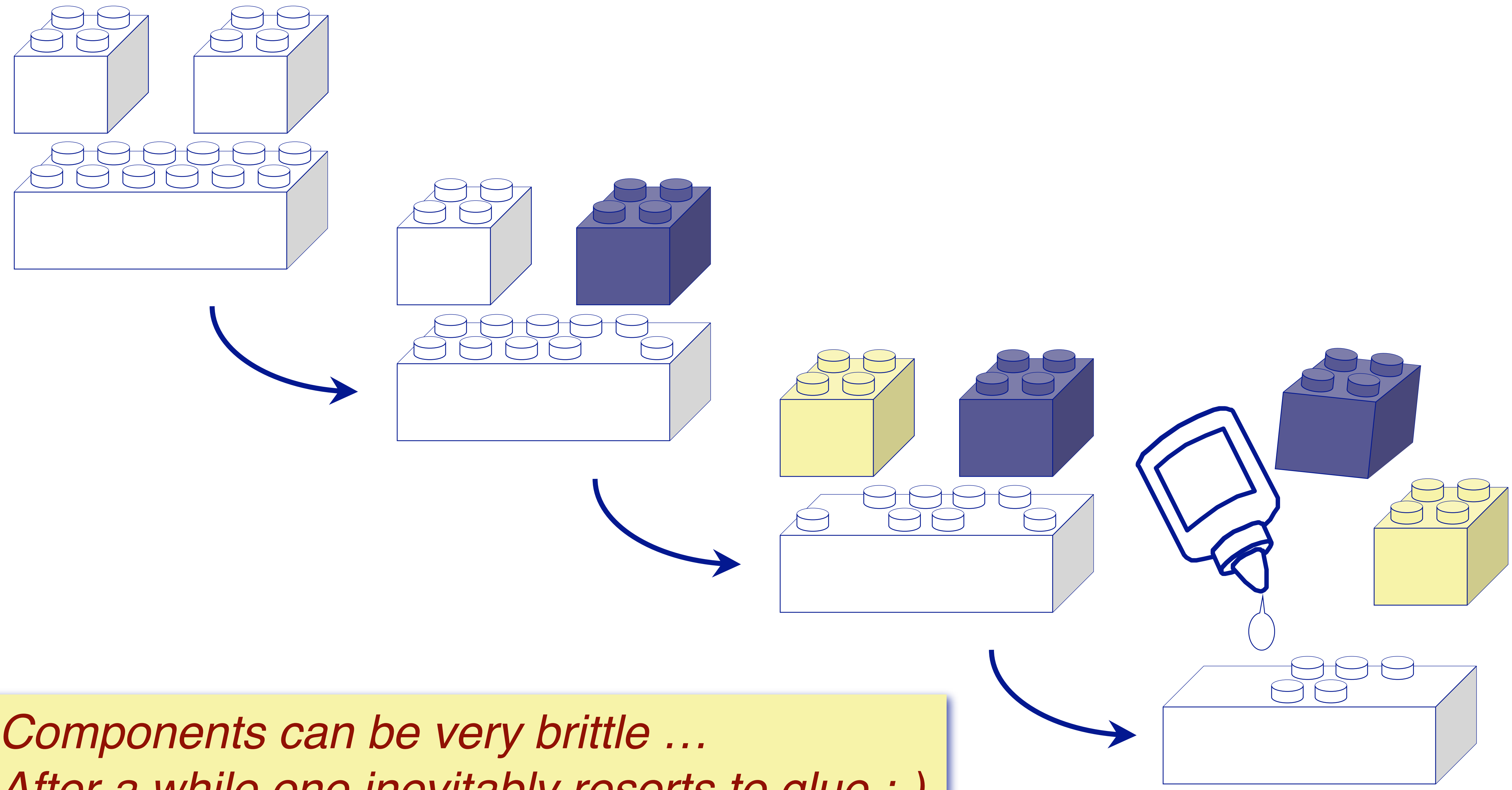
- The *symptoms* and the source of the problems are the *same*
- The *technical details* and solutions may *differ*

OO techniques promise better

- flexibility,
- reusability,
- maintainability
- …

⇒ *they do not come for free*

What about Components ?



Components can be very brittle …
After a while one inevitably resorts to glue :-)

Modern Methods & Tools ?

[Glas98a] quoting empirical study from Sasa Dekleva (1992)

> Modern methods(*) lead to *more reliable software*
> Modern methods lead to *less frequent software repair*
> and ...
> Modern methods lead to *more total maintenance time*

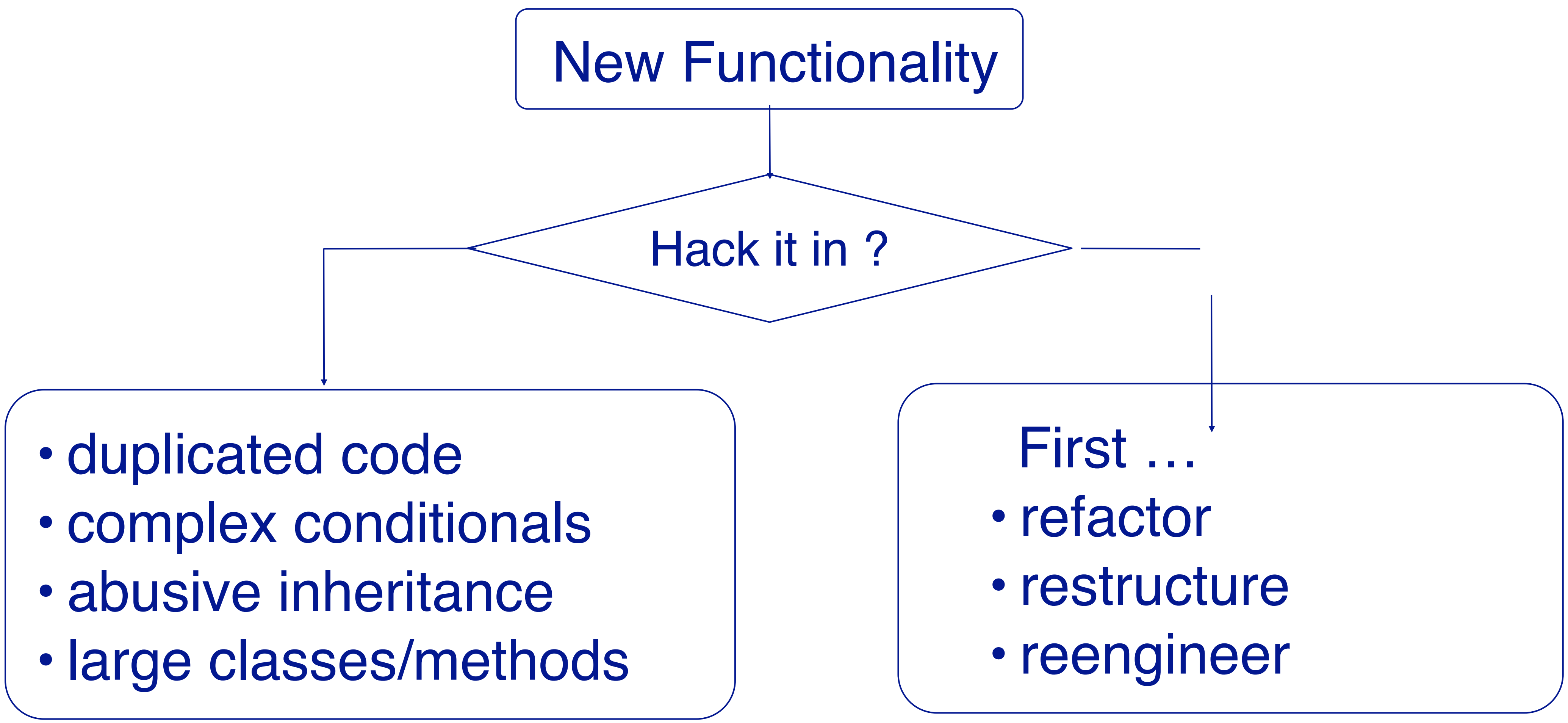
Contradiction ? *No!*

- modern methods make it easier to change
  ... this capacity is used to enhance functionality!

(*) process-oriented structured methods, information engineering, data-oriented methods, prototyping, CASE-tools – not OO !

How to deal with Legacy ?

New or changing requirements will gradually degrade original design
… unless extra development effort is spent to adapt the structure



Take a *loan* on your software
⇒ pay back via reengineering

Investment for the future
⇒ paid back during maintenance

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

How to cope with evolution?

- Need to *assess* evolution
- Need to *analyze* software and running systems
- Need to *adapt* evolving software systems
- Need to *enable* evolution, also at runtime

Roadmap

> Overview
> Laws of Software Evolution
> **Reflection and Metaprogramming**
> Smalltalk
> Reverse and Reengineering

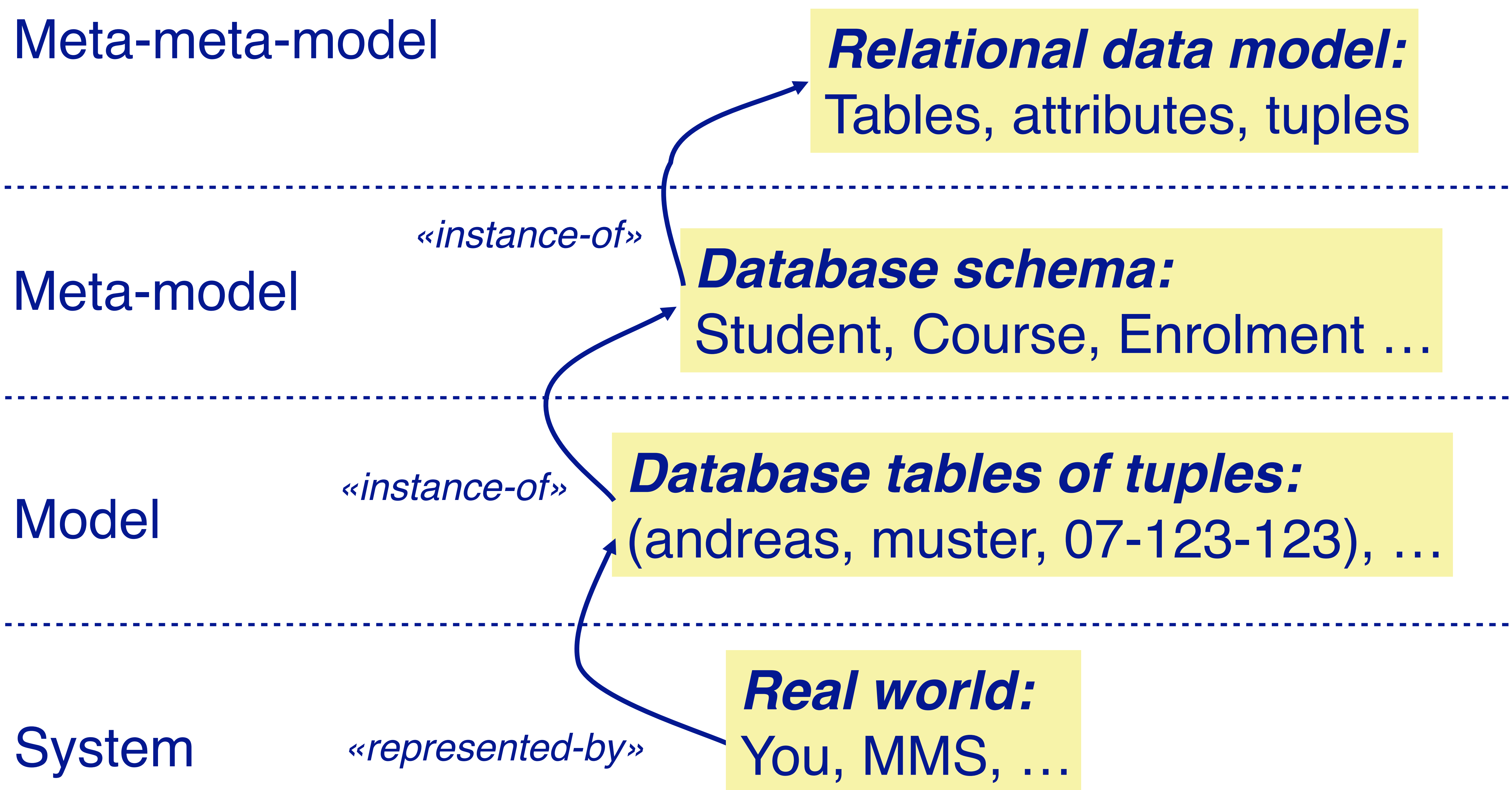
What is a model?

This slide intentionally left blank

What is a meta-model?

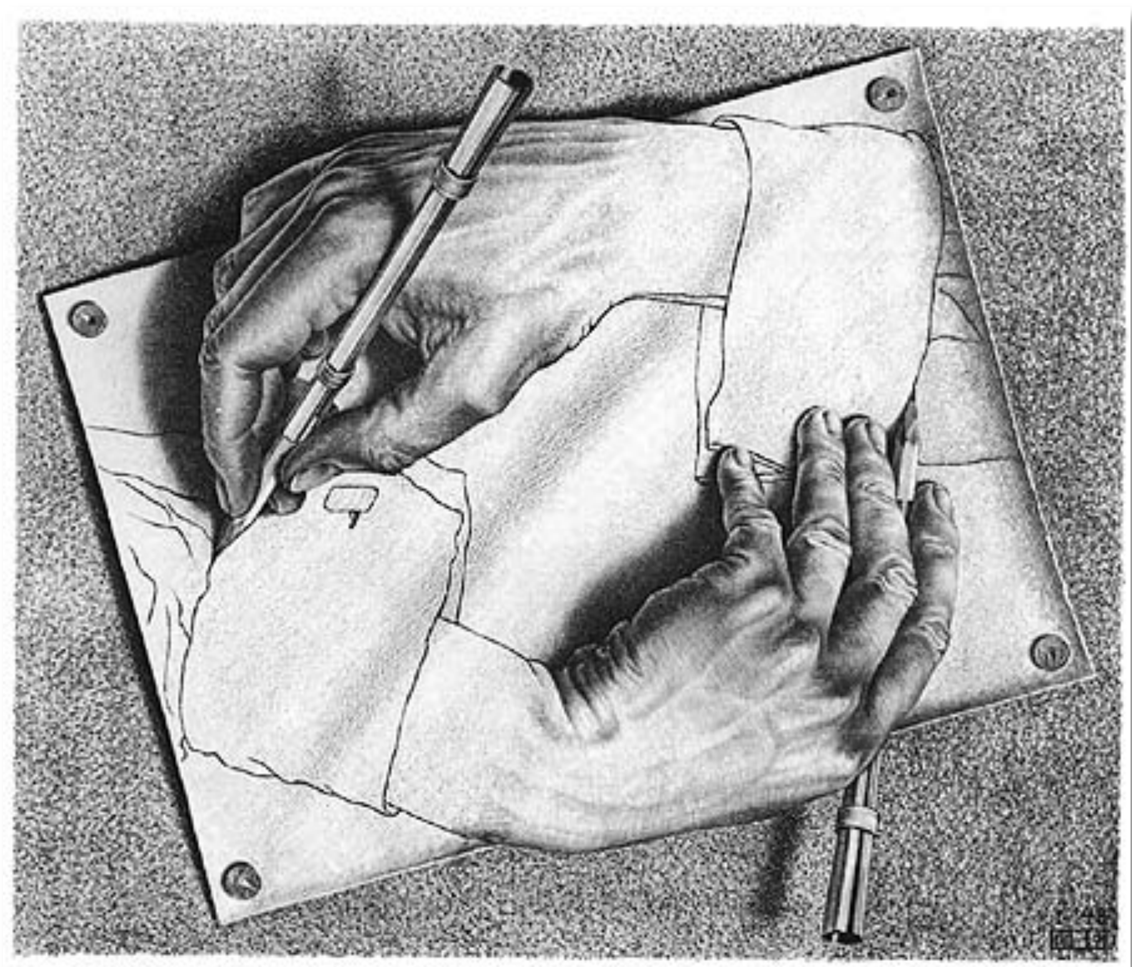
This slide intentionally left blank

Example from databases



Metaprogramming

A <u>metaprogram</u> is a program that manipulates a program *(possibly itself)*



Reflection

> "*Reflection* is the ability of a program to *manipulate as data* something representing the *state of the program* during its own execution.

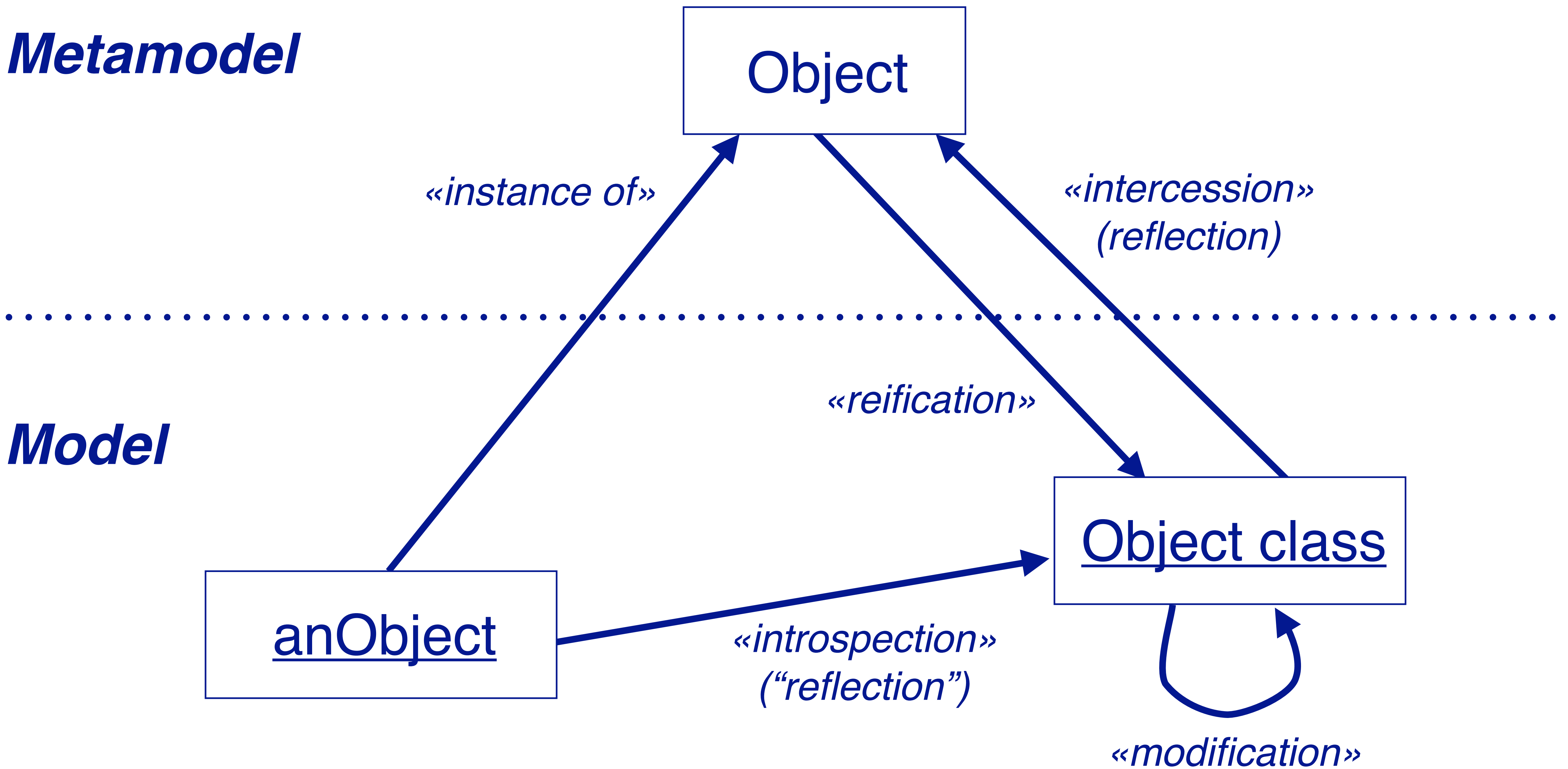
> *Introspection* is the ability for a program to *observe* and therefore *reason* about its own state.

> *Intercession* is the ability for a program to *modify* its own execution state or *alter its own interpretation* or meaning.

Both aspects require a mechanism for encoding execution state as data: this is called *reification*."

— *Bobrow, Gabriel & White, "CLOS in Context", 1993*

Reflection and Reification



Causal connection

> “A system having itself as application domain and that is *causally connected* with this domain can be qualified as a reflective system”

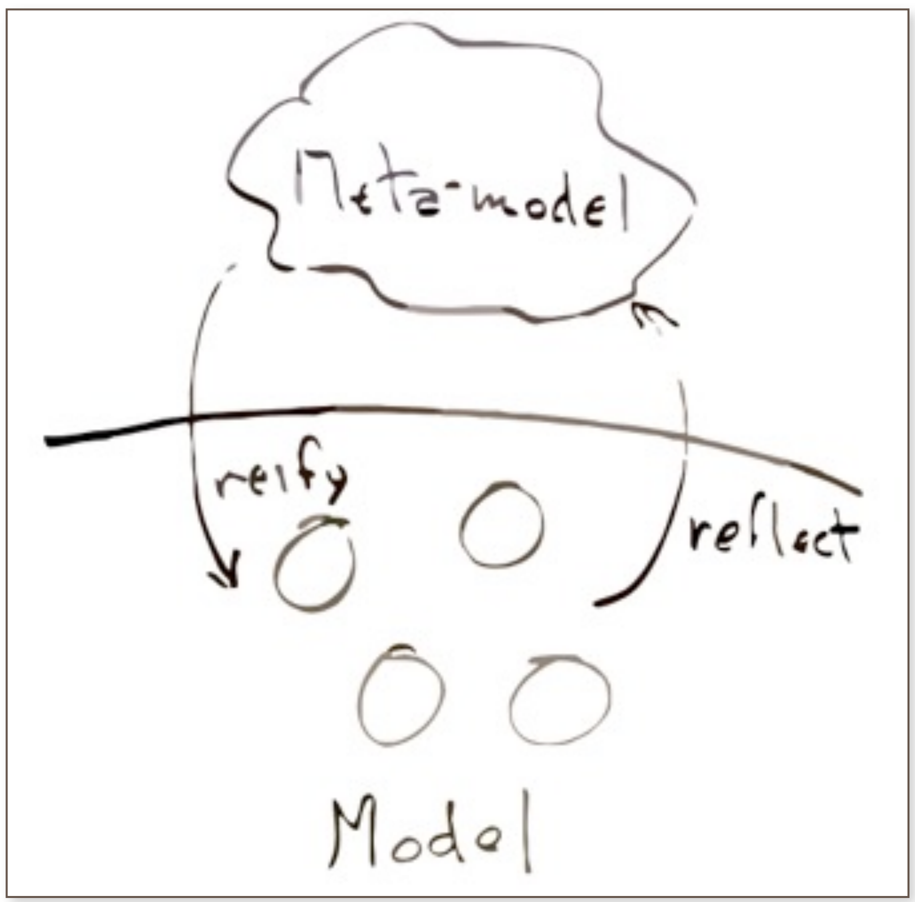
— Maes, OOPSLA 1987

- A reflective system has an *internal representation of itself*.
- A reflective system is able to *act on itself* with the ensurance that its representation will be causally connected (up to date).
- A reflective system has some static capacity of *self-representation* and dynamic *self-modification* in constant synchronization

Roadmap

> Overview
> Laws of Software Evolution
> Reflection and Metaprogramming
> **Smalltalk**
> Reverse and Reengineering

Birds-eye view



Smalltalk is still today one of the few fully reflective, fully dynamic, object-oriented development environments.

We will see how a simple, uniform object model enables live, dynamic, interactive software development.

What is Smalltalk?

> ***Pure OO language***
- —Single inheritance
- —Dynamically typed

> ***Language and <u>environment</u>***
- —Guiding principle: *"Everything is an Object"*
- —Class browser, debugger, inspector, …
- —Mature class library and tools

> ***Virtual machine***
- —Objects exist in a persistent *image* [+ *changes*]
- —Incremental compilation

What is interesting about Smalltalk?

> Everything is an object
> Everything happens by sending messages
> All the source code is there all the time
> You can't lose code
> You can change everything
> You can change things without restarting the system
> The Debugger is your Friend

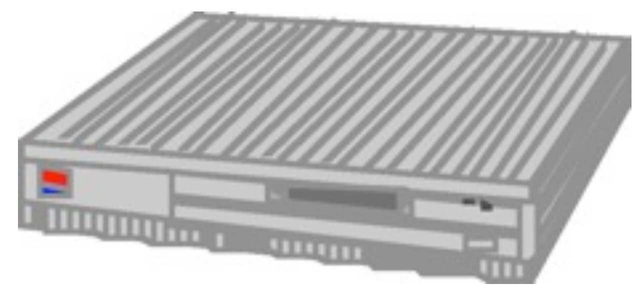
How does Smalltalk work?

Image

+

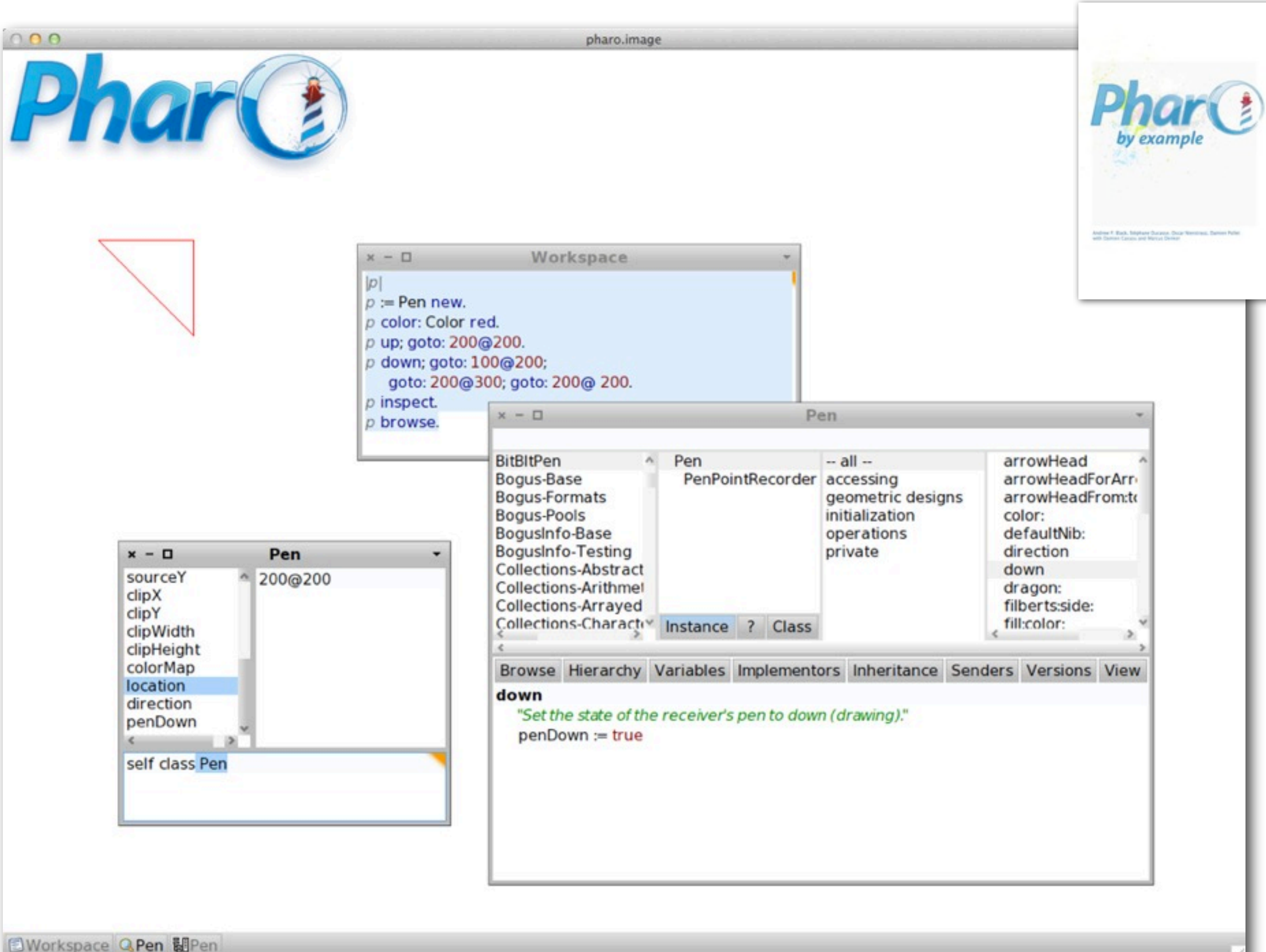
Changes

Virtual machine

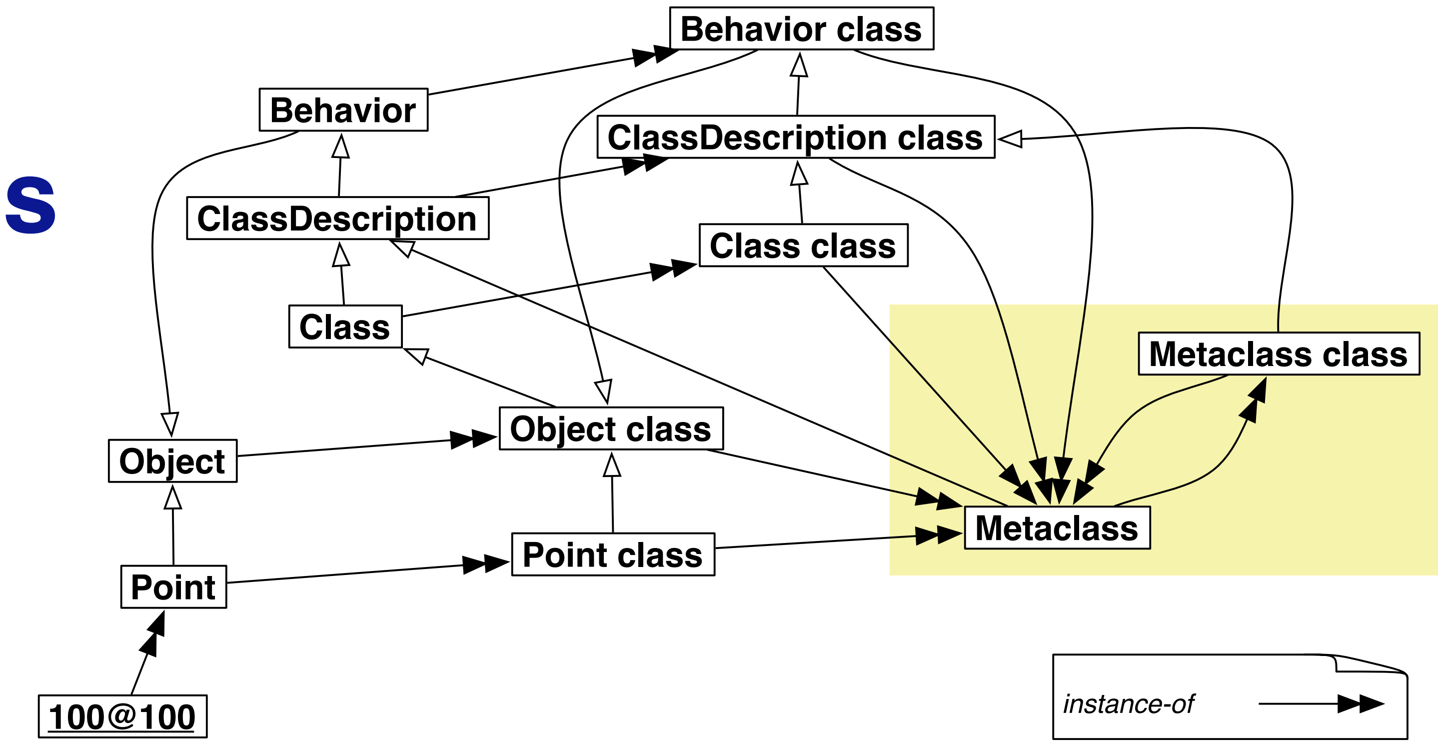


+

Sources


pharo.image
Pharo
Pharo by example
Workspace
|p|
p := Pen new.
p color: Color red.
p up; goto: 200@200.
p down; goto: 100@200;
goto: 200@300; goto: 200@ 200.
p inspect.
p browse.
Pen
BitBltPen
Bogus-Base
Bogus-Formats
Bogus-Pools
BogusInfo-Base
BogusInfo-Testing
Collections-Abstract
Pen
PenPointRecorder
-- all --
accessing
geometric designs
initialization
operations
private
arrowHead
color:
defaultNib:
direction
down
dragon:
filberts:side:
Instance
?
Class
Browse
Hierarchy
Variables
Implementors
Inheritance
Senders
Versions
View
down
"Set the state of the receiver's pen to down (drawing)."
penDown := true
sourceY
clipX
clipY
clipWidth
clipHeight
colorMap
location
direction
penDown
200@200
self class Pen
Workspace
Pen
Pen

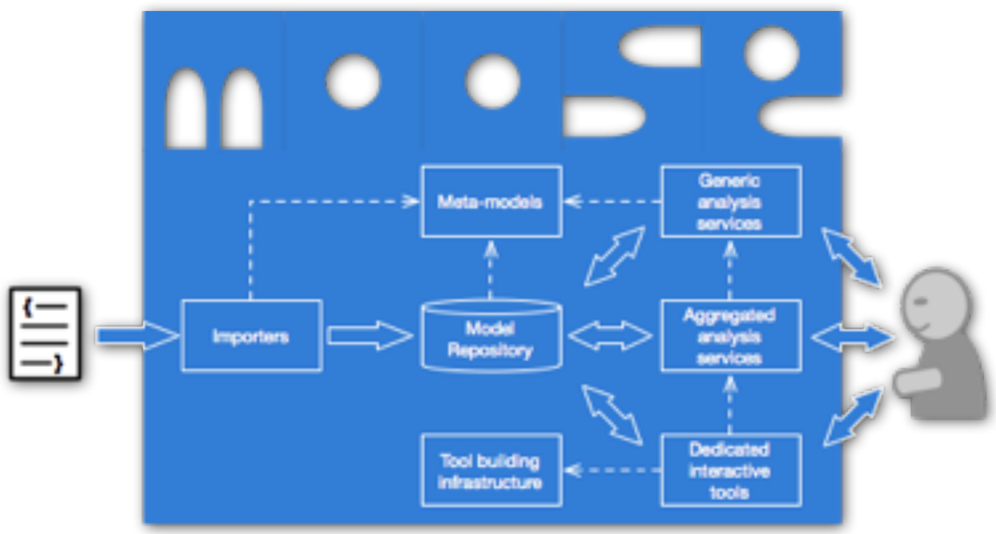
Metaclasses in 7 points



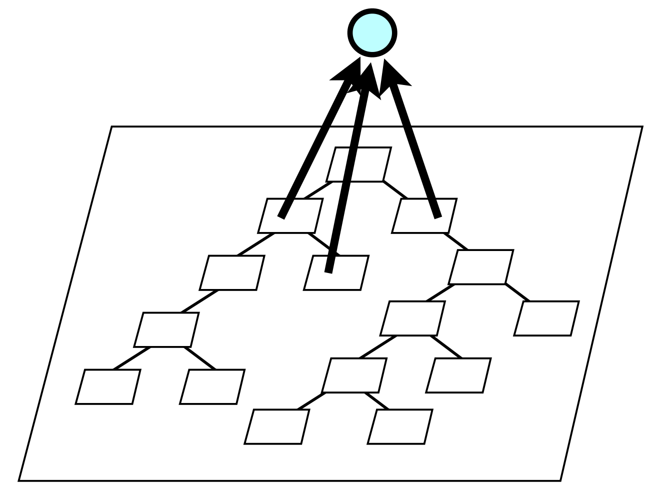
1. Every object is an instance of a class
2. Every class inherits from Object
3. Every class is an instance of a metaclass
4. The metaclass hierarchy parallels the class hierarchy
5. Every metaclass inherits from Class and Behavior
6. Every metaclass is an instance of Metaclass
7. The metaclass of Metaclass is an instance of Metaclass

Adapted from Goldberg & Robson, *Smalltalk-80 — The Language*

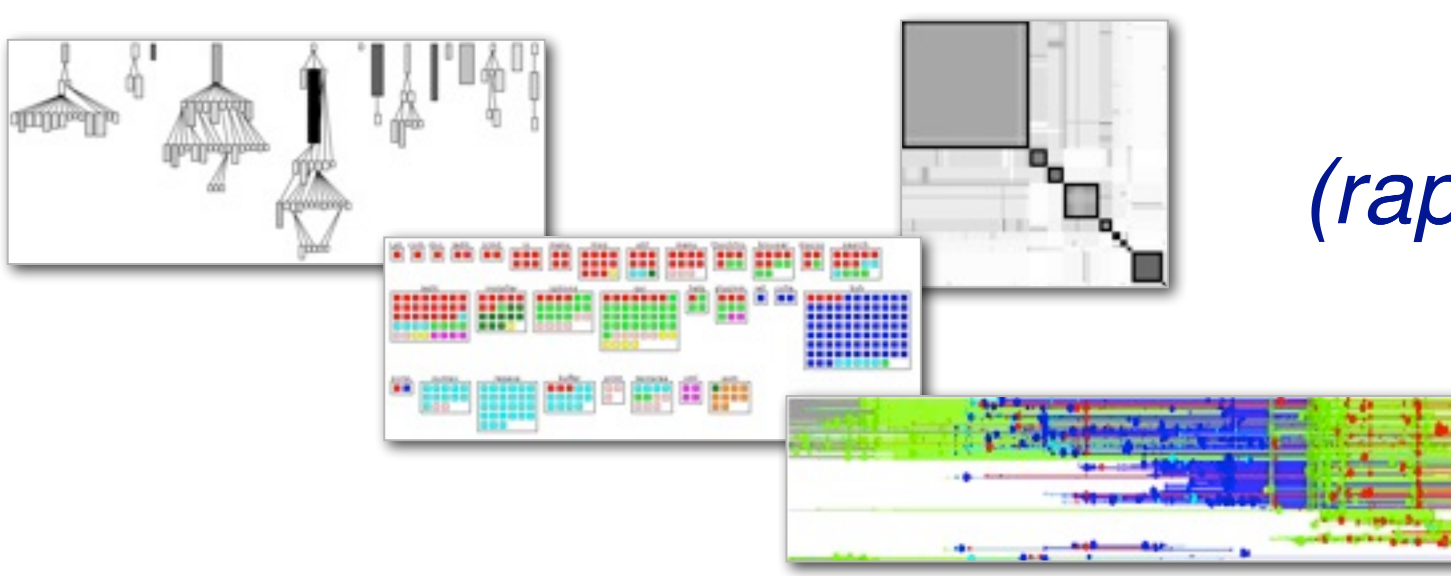
Why is Smalltalk interesting for Software Evolution?



Modeling
(fully OO)



Instrumentation
(dynamic adaptation)



Analysis
(rapid prototyping)

Roadmap

> Overview
> Laws of Software Evolution
> Reflection and Metaprogramming
> Smalltalk
> **Reverse and Reengineering**

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

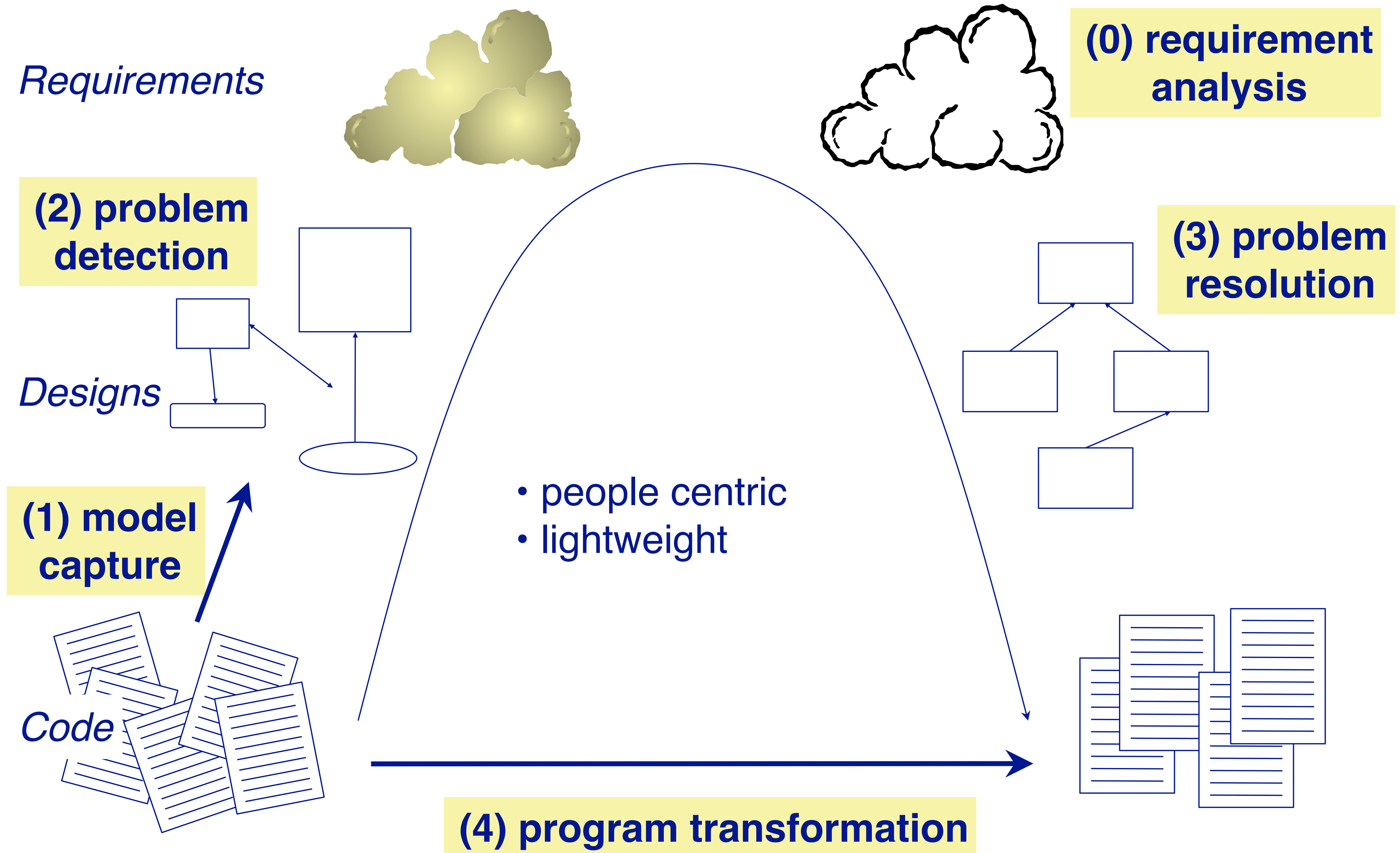
— Chikofsky and Cross

> *Data reengineering*
  - integrating and centralizing multiple databases
  - unifying multiple, inconsistent representations
  - upgrading data models

— Sommerville, ch 32

> *Refactoring*
  - renaming/moving methods/classes etc.

The Reengineering Life-Cycle



Reverse engineering Patterns

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.*

Example: **Interview During Demo**

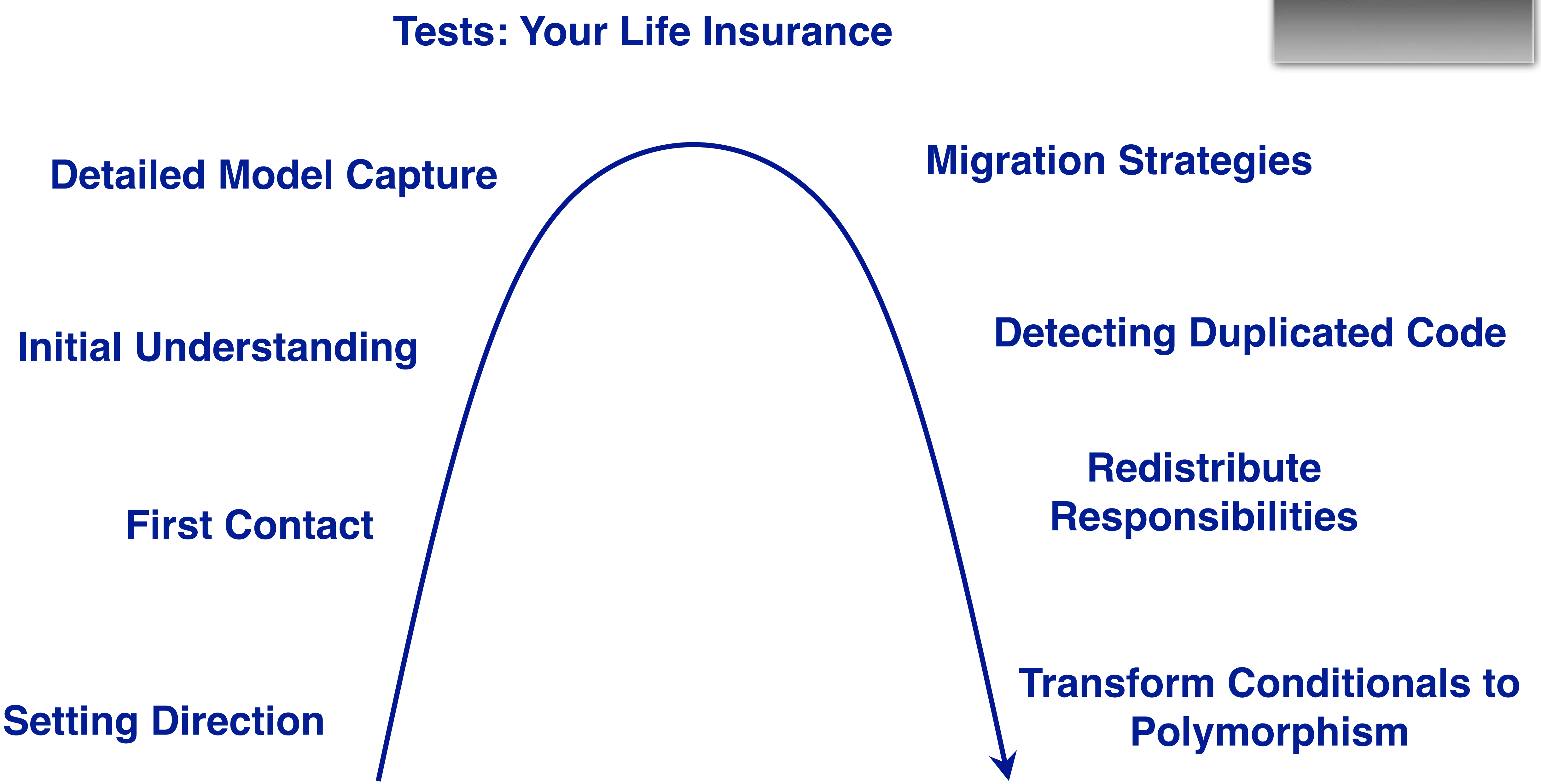
Reengineering Patterns

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*

Example: **Move Behaviour Close to Data**

A Map of Reengineering Patterns



What you should know!

- Software “maintenance” is really *continuous development*
- Real-world programs *must change* or become less useful over time
- What is the relationship between a model and its meta-model?
- What is the difference between reflection and reification? Between introspection and intercession?
- How does Smalltalk differ from Java?
- How does reverse-engineering differ from reengineering?

Can you answer these questions?

> Why do successful software systems always require more maintenance?

> What is a model? A meta-model?

> What kind of “reflection” does Java support?

> In Smalltalk, how can you reflect on the VM?

> How do static and dynamic analysis of software systems differ?