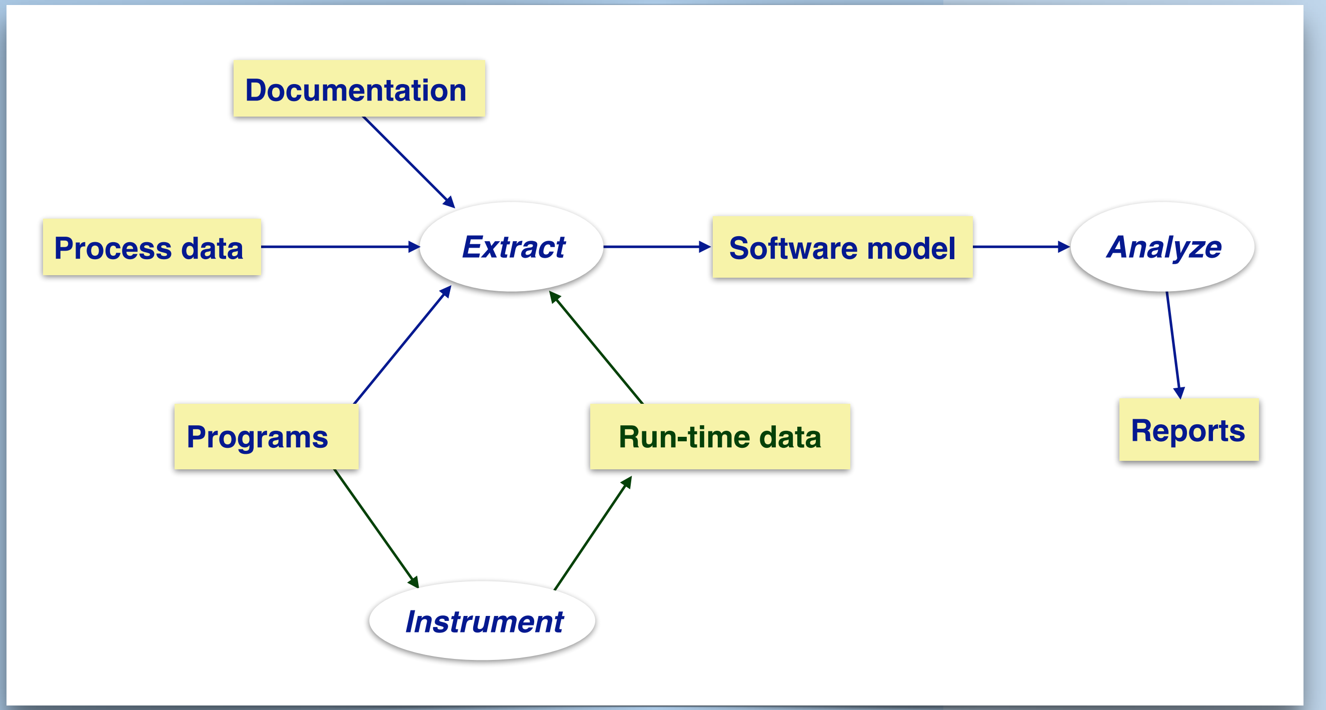
UNIVERSITÄT BERN

Software Modeling and Analysis

Introduction

Oscar Nierstrasz



Roadmap

> Overview
> Laws of Software Evolution
> Reverse and Reengineering
> Software Modeling
> Software Analysis

SMA

Lecturers	Oscar Nierstrasz
Assistants	Pascal Gadient, Pooja Rani
Lectures	IWI 001, Wednesdays @ 10h15-12h00
Exercises	IWI 001, Wednesdays @ 12h00-13h00
WWW	scg.unibe.ch/teaching/sma

Covid-19 Logistics

- Due to Covid-19, this class will largely be held remotely
  - Please *sign up on Piazza* so we can communicate with you
  - Details: scg.unibe.ch/teaching/sma
- Lectures will generally be pre-recorded (or live via Zoom)
  - If pre-recorded, please view the lecture *before* the regular class
  - An interactive Q&A zoom session will then take place at the scheduled lecture hour (Wednesdays at 10h15)
  - Zoom link and Google doc for questions will be communicated by Piazza

This is a note (a hidden slide). You will find some of these scattered around the PDF versions of the slides.

NB: some links to copyrighted materials are only accessible within the unibe.ch domain.

Roadmap

> **Overview**
> Laws of Software Evolution
> Reverse and Reengineering
> Software Modeling
> Software Analysis

Goals of this course

You will learn how to:

- recognize the problems of legacy software
- use reflection and metaprogramming techniques
- extract software models from source code and other artifacts
- apply software metrics to detect quality problems
- visualize software to support program comprehension
- apply basic static and dynamic analysis techniques

Real software systems continuously evolve over time. As they evolve, they become harder to understand and maintain. In this course we will explore techniques to model complex software systems and analyze them to support program comprehension and reengineering tasks.

Most lectures will combine theoretical background and practical application of tools and techniques. A portion of this course will make heavy use of Smalltalk, a live programming environment that supports advanced reflection metaprogramming techniques.

Some material is based on two open−source textbooks: Pharo by Example, and Object-Oriented Reengineering Patterns.

- http://pharobyexample.org
- http://scg.unibe.ch/download/oorp/

Course Schedule

Week	***Date***	***Lesson***
1	16-Sep-20	**Introduction to Software Modeling and Analysis**
2	23-Sep-20	**Smalltalk: A Reflective Language and System**
3	30-Sep-20	**Understanding Classes and Metaclasses**
4	7-Oct-20	**Reflection and Metaprogramming**
5	14-Oct-20	**Moldable Software Exploration (Tudor Girba)**
6	21-Oct-20	**Software visualization (Leonel Merino)**
7	28-Oct-20	**Software Metrics and Problem Detection; Moose (Andrei Chiş)**
8	4-Nov-20	**Code and Test Smells (Fabio Palomba)**
9	11-Nov-20	**Static Program Analysis / Soot (ON / Manuel Leuenberger)**
10	18-Nov-20	**Comment analysis (Pooja Rani)**
11	25-Nov-20	**Dynamic Analysis (Nataliia Stulova)**
12	2-Dec-20	**Fuzz Testing (Reza Hazhirpasand)**
13	9-Dec-20	**Socio-technical Aspects in Software Systems (Alberto Bacchelli)**
14	16-Dec-20	***Final exam (NB: Room E8-001)***

Roadmap

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…

Lehman and Belady studied numerous industrial software systems and identified seven “laws” of software evolution. The two listed here are especially relevant in a reengineering context.

Although the studies were carried out in the early 80s, the laws are still relevant today.

http://scgresources.unibe.ch/Literature/Books/Lehm85aProgramEvolution.pdf

Software Maintenance vs. Cost

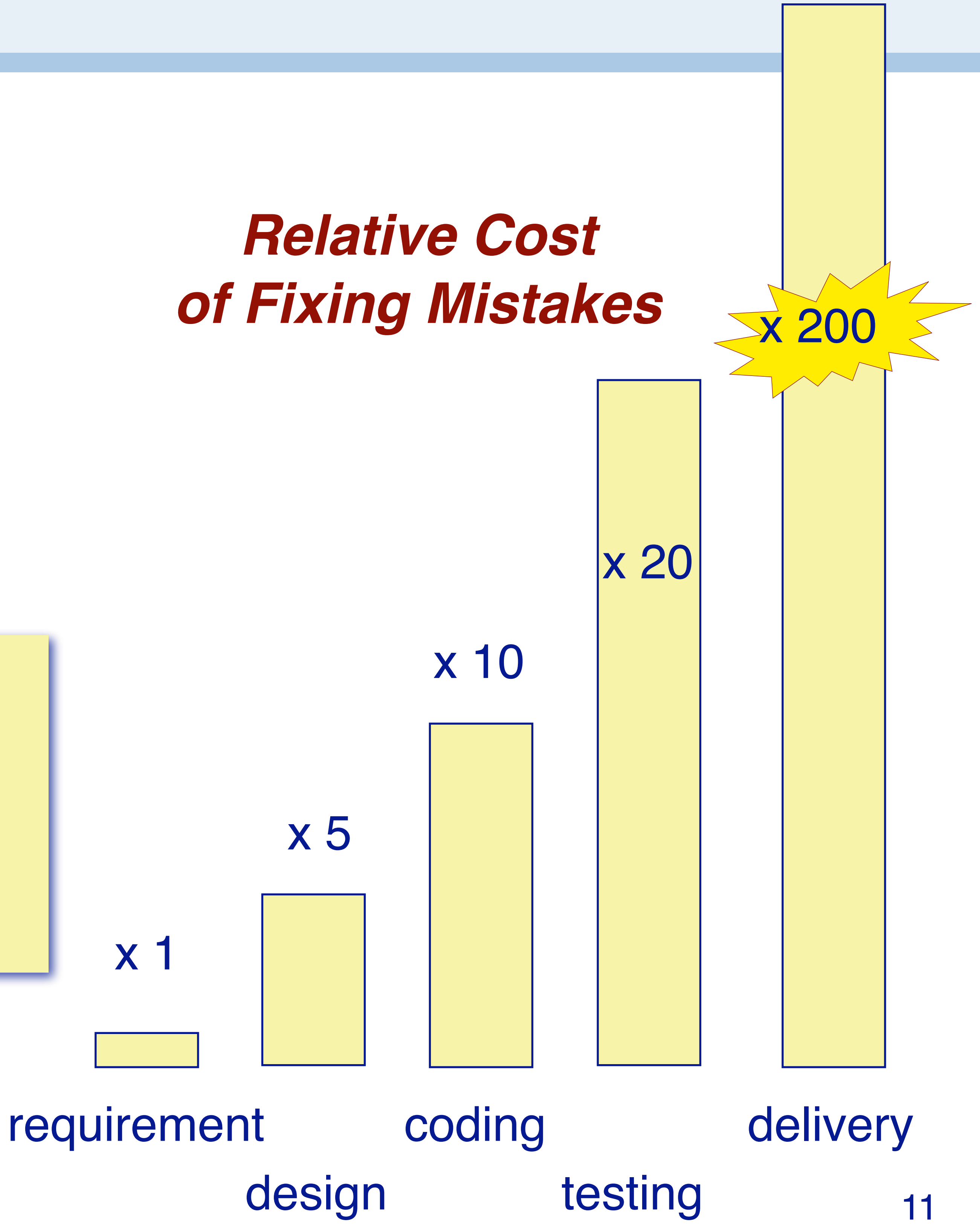
Relative Maintenance Effort

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, …)?

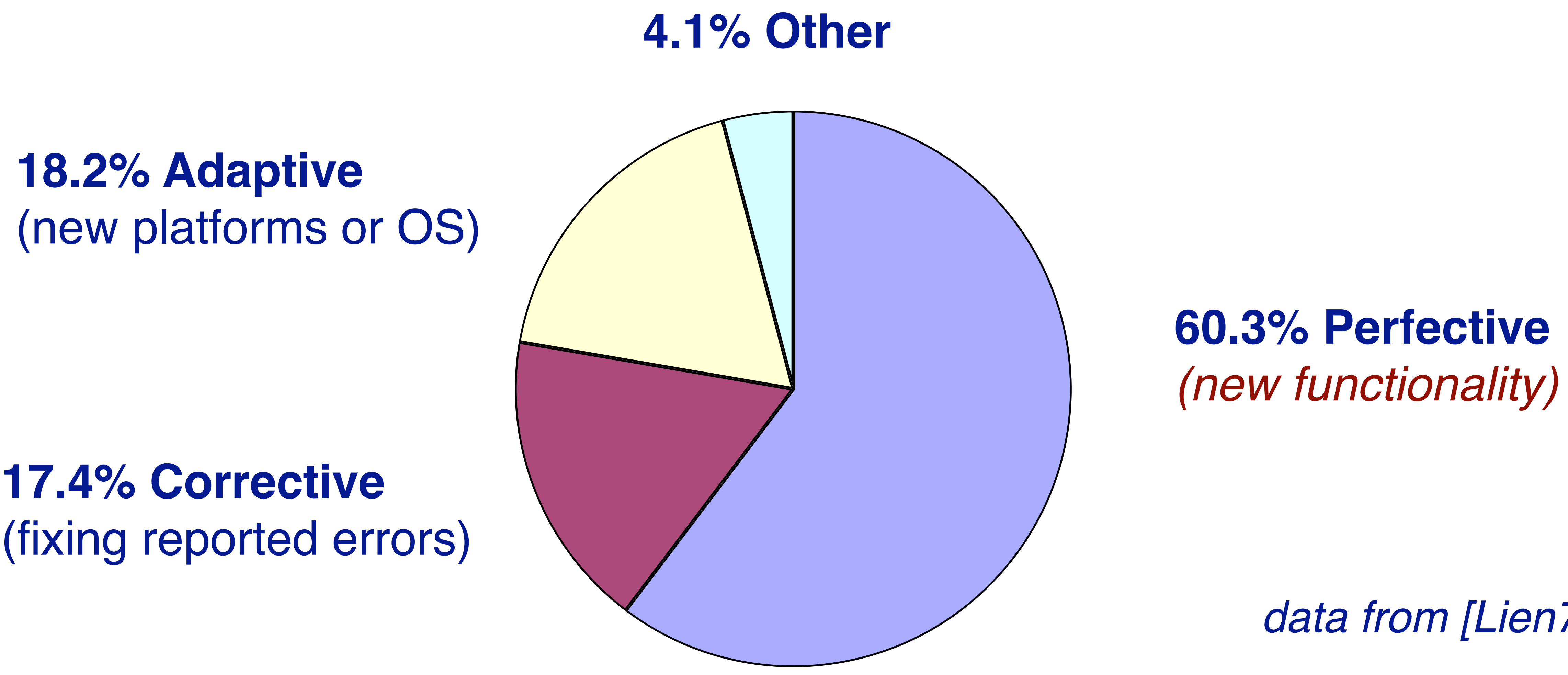
Relative Cost of Fixing Mistakes



Studies have shown that more than half of total software budgets are devoted to “software maintenance”. Furthermore, there are many claims that the cost of fixing errors in software projects goes up over time. This suggests that one should try to catch errors as early as possible, and perhaps one should invest more effort into “getting requirements right”. The latter concludion is a fallacy, however, if we recall that Lehman's laws tell us that requirements will always change.

The real conclusion should be that we should search for ways to reduce the cost of change in the long-term.

“Maintenance” = 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

If we take a closer look at what “maintenance” is, we see that it is mostly “continuous development”, not just bug fixing.

Modern development methods like “agile development” acknowledge this point and strive to reduce the cost of changing a deployed system.

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*

Very rarely do we have the luxury of developing a “greenfield” software project. We are almost always confronted with “legacy” software.

But what does this term mean? Although it is often equated with “outdated” and “bad” software, the reality is somewhat different.

The term “legacy” applied to software means that it is (1) inherited, and (2) has real business value. If it did not have value, we could just discard it. But because it has value, it needs to be maintained. Why does this pose a problem? ...

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

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

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

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
> **Reverse and Reengineering**
> Software Modeling
> Software Analysis

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 [IEEE Software 1990]

Elliot Chikofsky and James Cross II. *Reverse Engineering and Design Recovery: A Taxonomy*. In IEEE Software 7(1) p. 13—17, January 1990.

http://dx.doi.org/10.1109/52.43044

http://scgresources.unibe.ch/Literature/Reengineering/Chik90a.pdf

What does this have to do with me?

Our claim:

All Software Engineering entails:

- Forward engineering
- Reverse engineering
- Reengineering
- Software maintenance

Real software systems evolve and must be maintained. You cannot escape legacy software. Although some development entails forward engineering, you will need to apply reverse engineering techniques to understand the legacy, and you must apply reengineering techniques to keep down the costs of maintaining legacy software over time.

Goals of Reverse Engineering

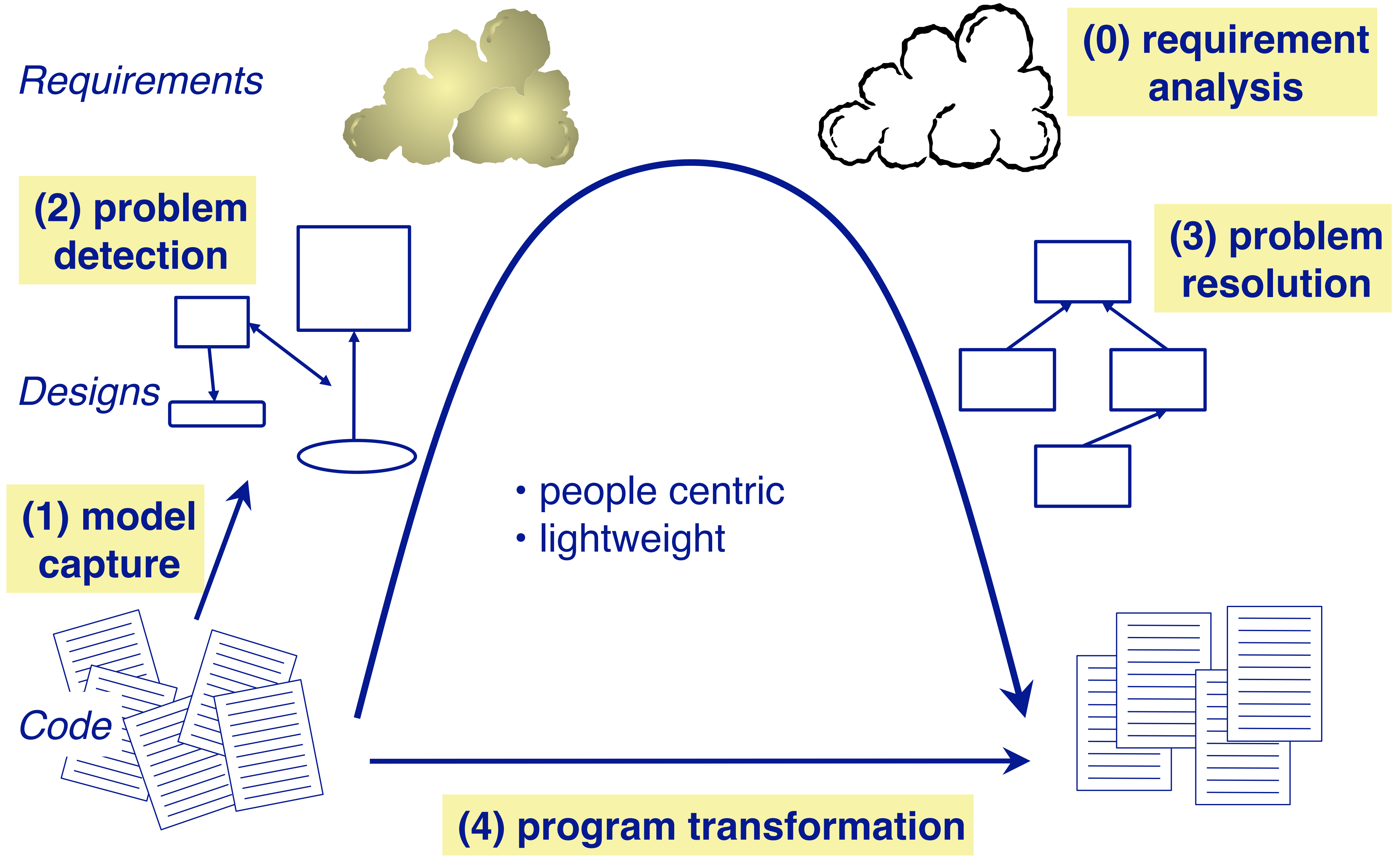
- > Cope with *complexity*
  - — need techniques to understand large, complex systems
- > Generate *alternative views*
  - — automatically generate different ways to view systems
- > Recover *lost information*
  - — extract what changes have been made and why
- > Detect *side effects*
  - — help understand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

— Chikofsky and Cross [IEEE Software 1990]

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.

The Reengineering Life-Cycle



- people centric
- lightweight

This picture offers an ideal view of the software reengineering lifecycle. It should not be read as a purely sequential process, but as a set of interleaving activities.

(0) Requirement analysis: analyse which parts of your requirements have changed

(1) Model capture: reverse engineer from the source-code to a more abstract form, i.e., a software model

(2) problem detection: identify design problems in that model

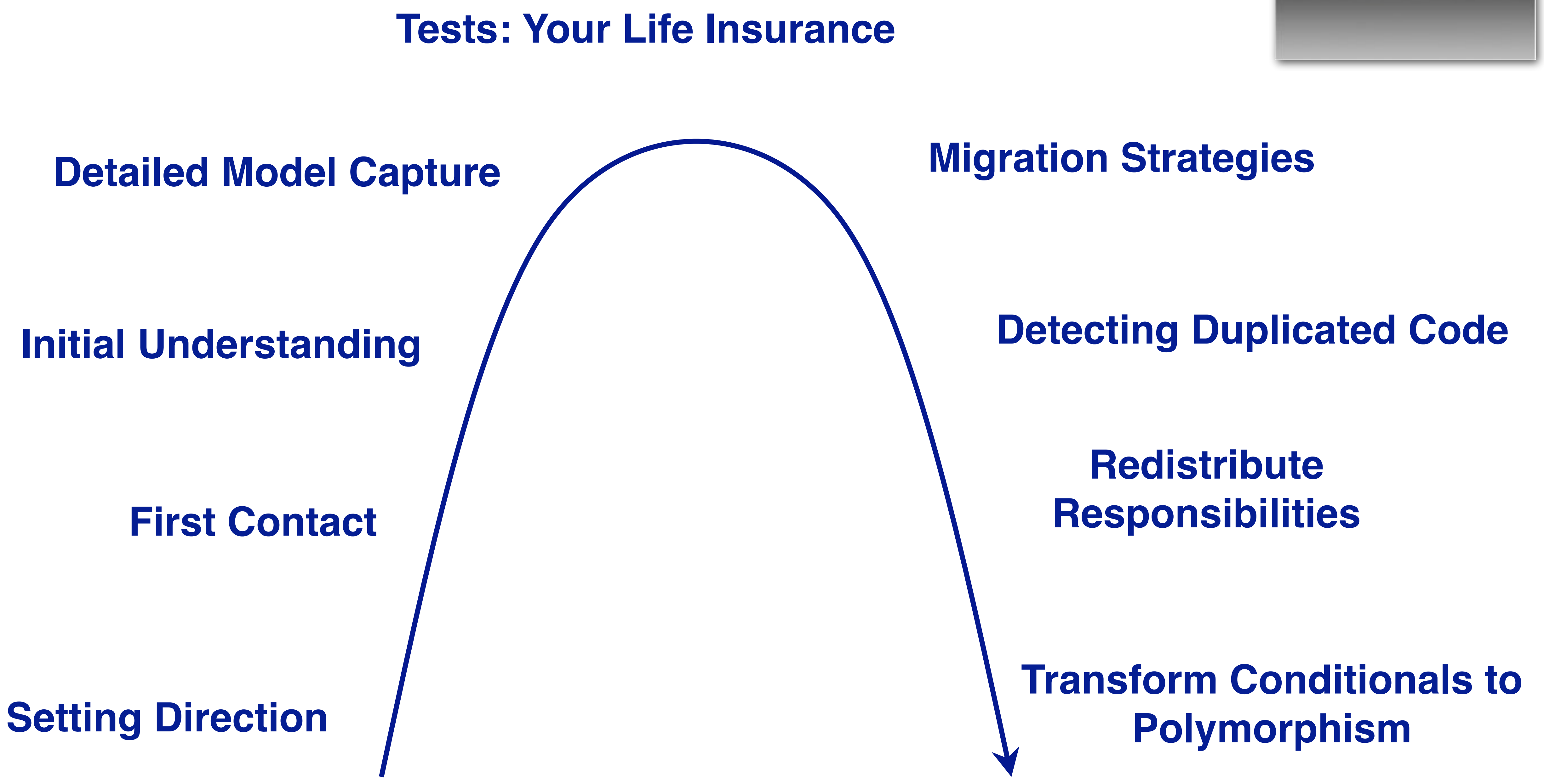
(3) problem resolution: propose an alternative design to fix the problems

(4) program transformations: make the necessary changes to the code

We will focus mainly on lightweight tools and techniques for modeling and analysis

A Map of Reengineering Patterns

Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Object-Oriented Reengineering
Patterns


Tests: Your Life Insurance
Detailed Model Capture
Initial Understanding
First Contact
Setting Direction
Migration Strategies
Detecting Duplicated Code
Redistribute Responsibilities
Transform Conditionals to Polymorphism

The book *Object-Oriented Reengineering Patterns* presents a series of process “patterns” to reverse and reengineer complex software systems. The patterns have been mined from experience reengineering a large number of legacy software systems.

The book is available as open source and will be referred to periodically in this course:

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

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**

Roadmap

What is a model?

This slide intentionally left blank

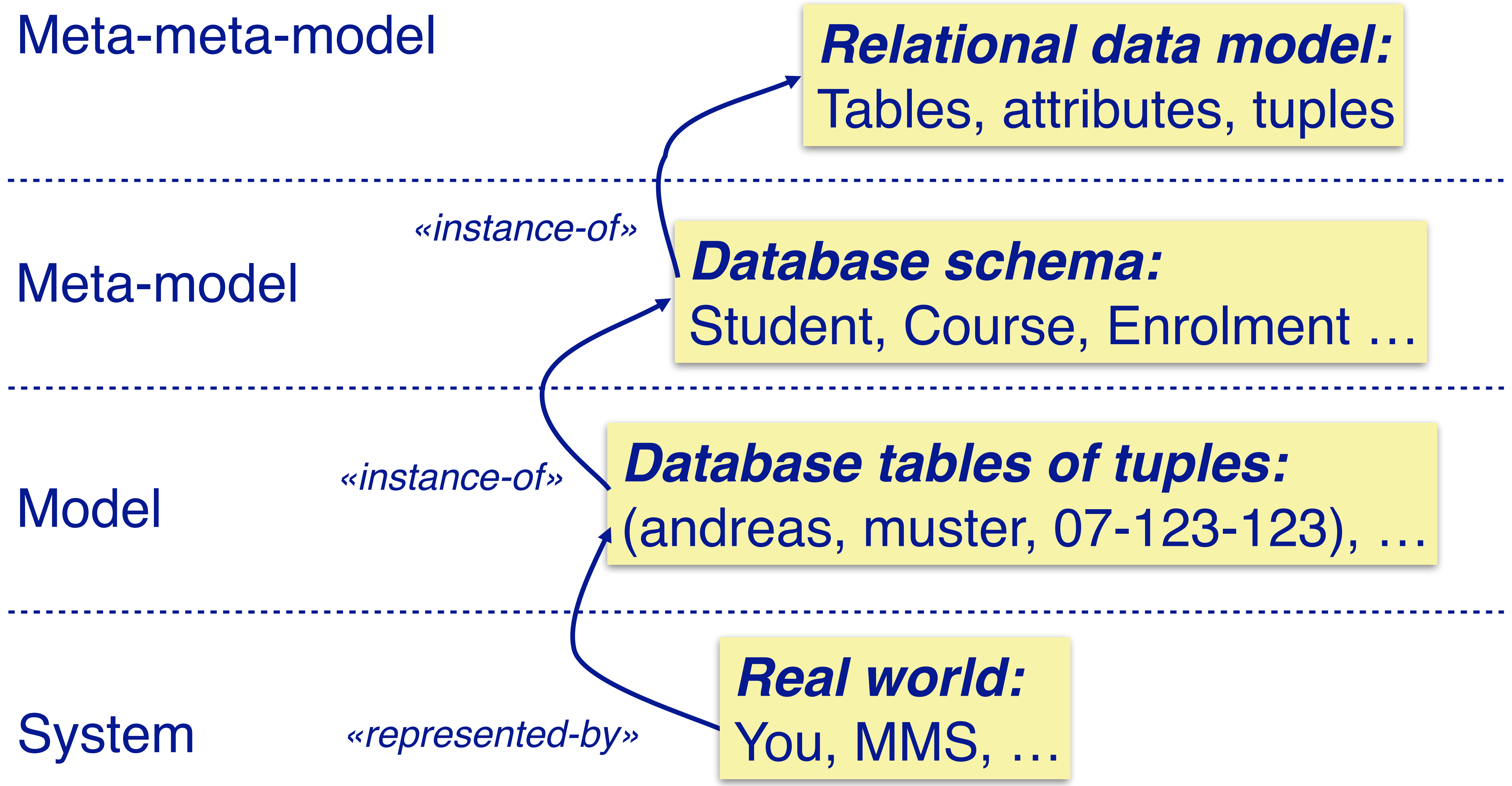
What are examples of models?
What about software models?
What does a model describe?
What are models used for?
How are models specified?

What is a meta-model?

This slide intentionally left blank

What are examples of meta-models?
What does a meta-model describe?
Is a meta-model a kind of model?
Do we need meta-meta-models?
If so, when does it all end?

Example from databases



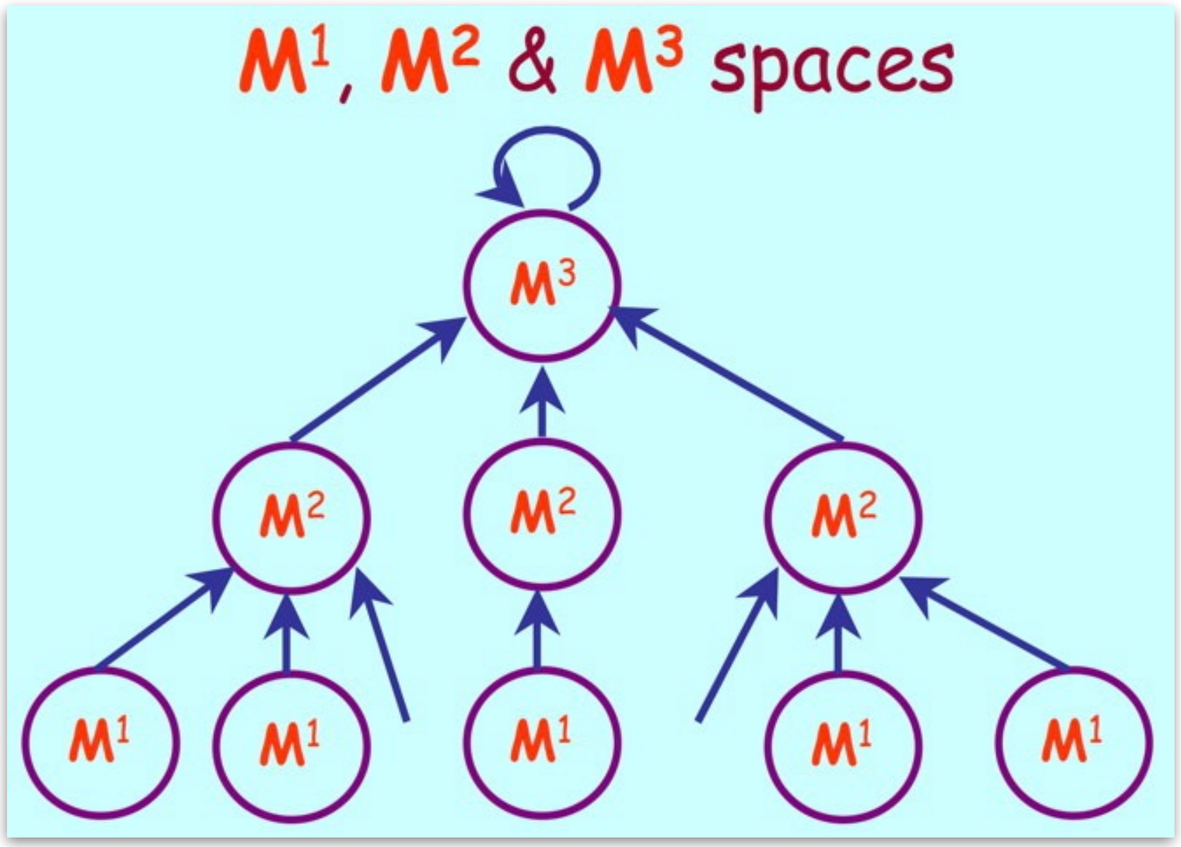
In the world of relational databases, a DB tuple represents some entity in the “real” world (NB: we can also model imagined entities, but that is beside the point).

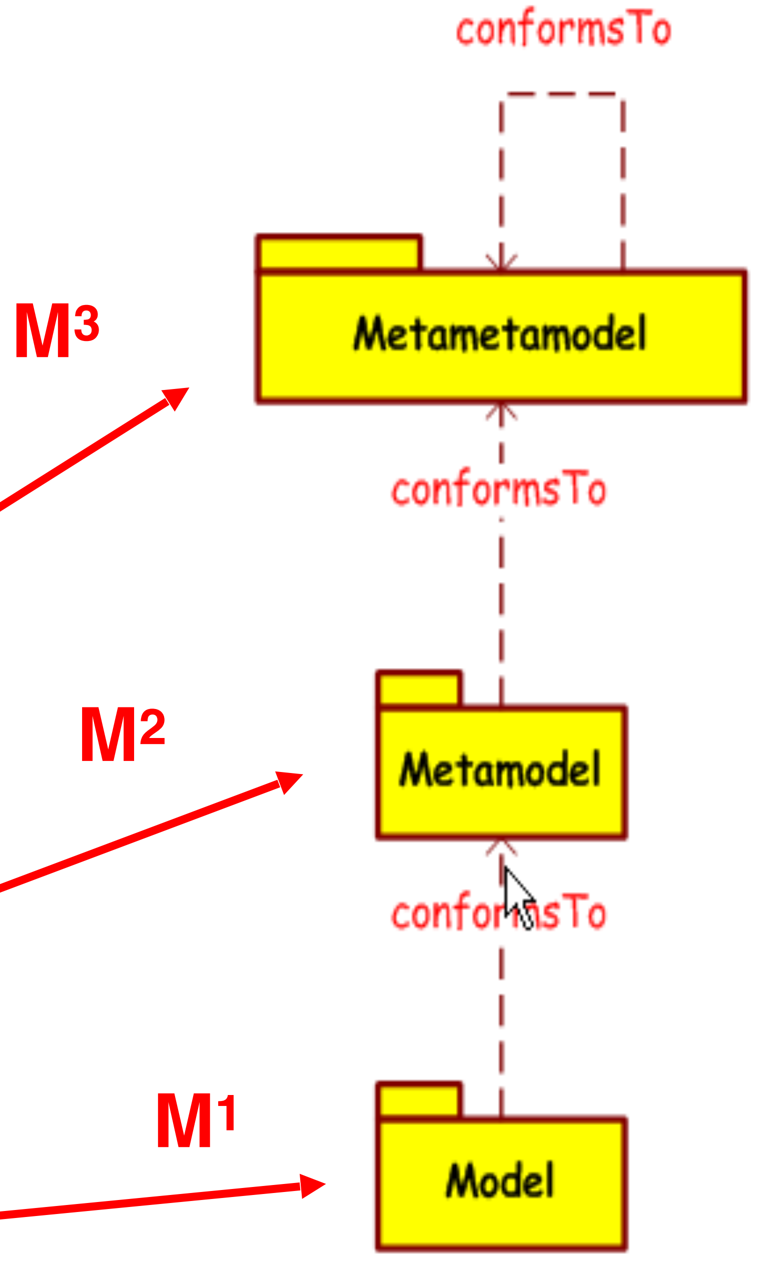
The model entities, on the other hand, are instances of some database schema (i.e., a metamodel) that dictates the structure of the model. We can have many such instances, each of which represents some other real world (modeled) entities.

The metamodel itself is an instance of a meta-meta-model, which dictates the structure of the metamodel.

In the UML world, we refer to the real world entities as living at the M0 level. Models are at M1, metamodels at M2, and meta-meta-models are at level M3.

The MOF





- One unique Metametamodel (the MOF)
- An important library of compatible Metamodels, each defining a DSL
- Each of the models is defined in the language of its unique metamodel

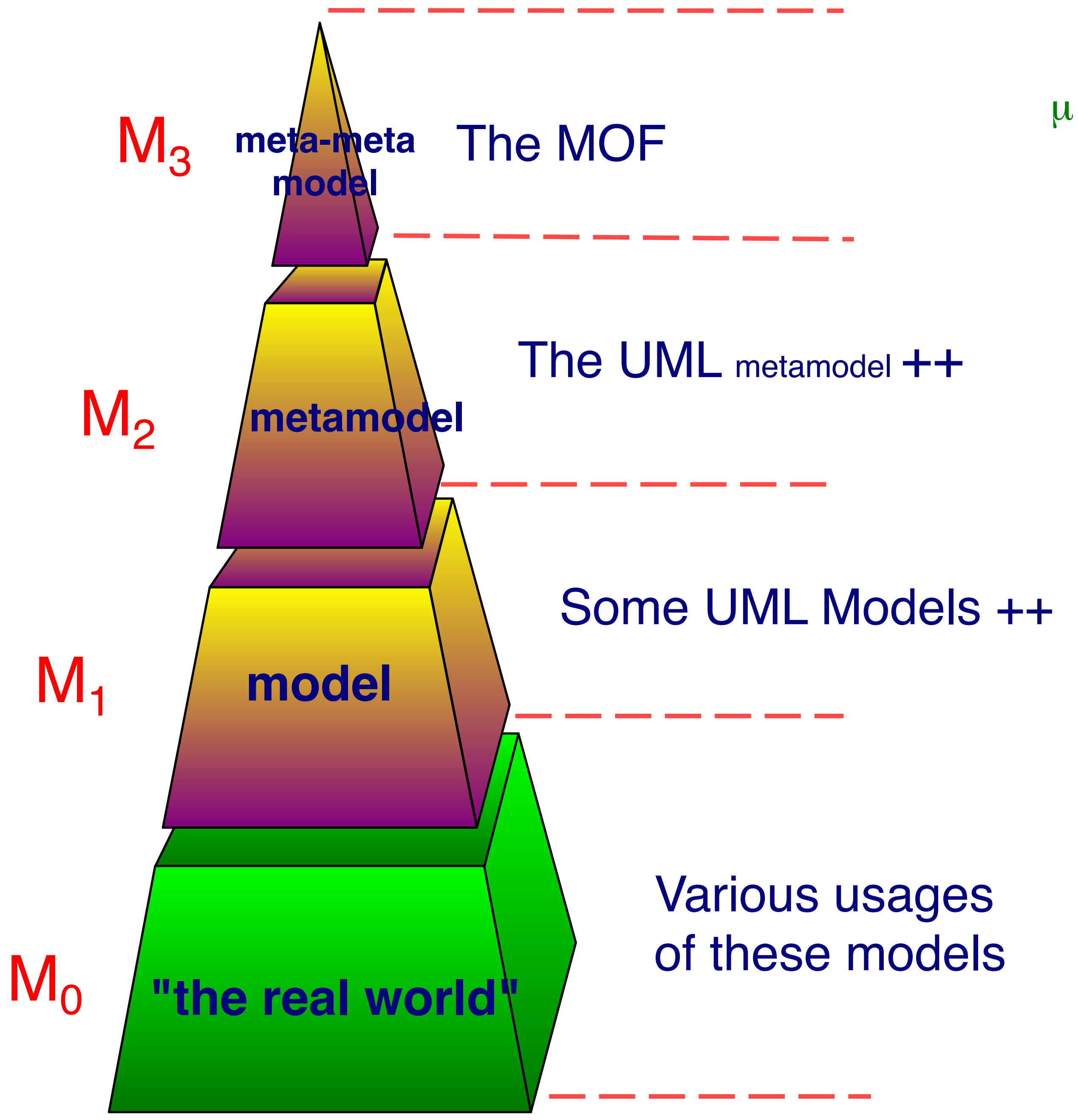
The Meta-Object Facility (MOF) is a standard of the Object Management Group (OMG) for Model-Driven Engineering (MDE), an approach in which code is generated from models.

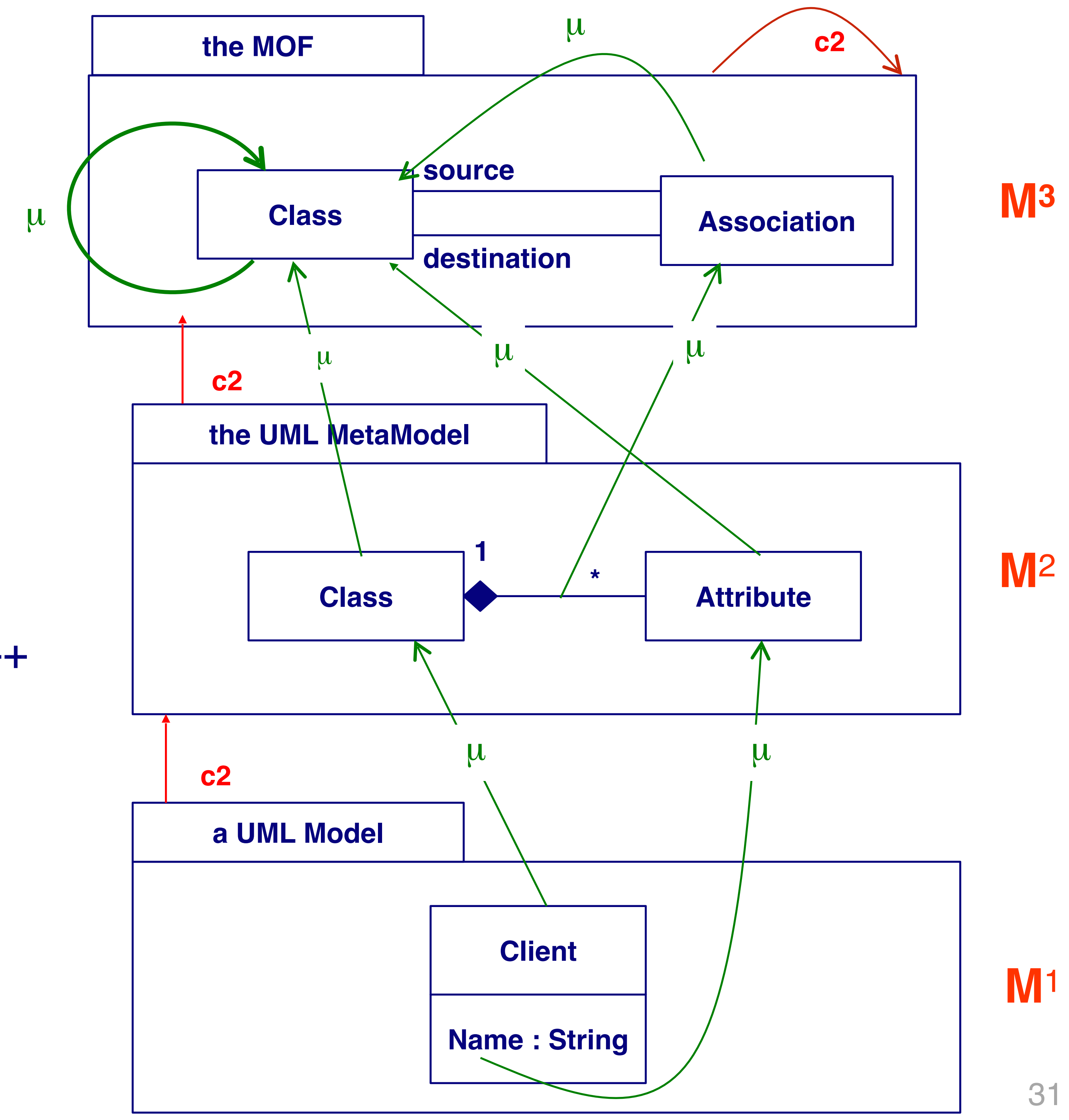
Models exist at levels M1, M2 and M3. A model at level M1 conforms to its metamodel at level M2, which conforms to its metametamodel at level M3.

In the MOF world, M3 conforms to itself, so there is no need for an infinite tower of metamodels.

https://en.wikipedia.org/wiki/Meta-Object_Facility

The OMG/MDA Stack





Level M0 refers to “the real world”. Note that M0 does not “conform to” M1, but rather M1 models M0.

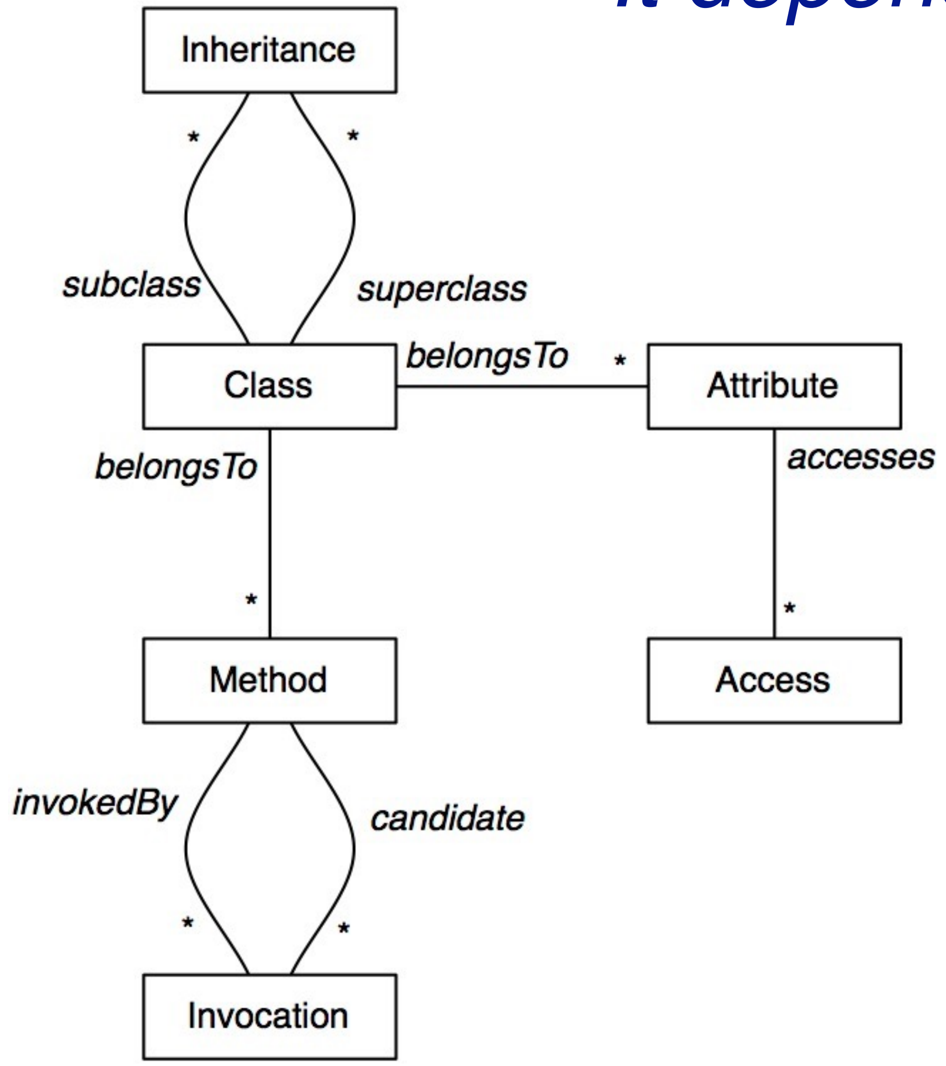
UML models at level M1 model various aspects of the real world in M0. Each UML model (a class diagram, a sequence diagram etc.) conforms to its particular metamodel at level M2. All UML metamodels conform to the MOF at level M3.

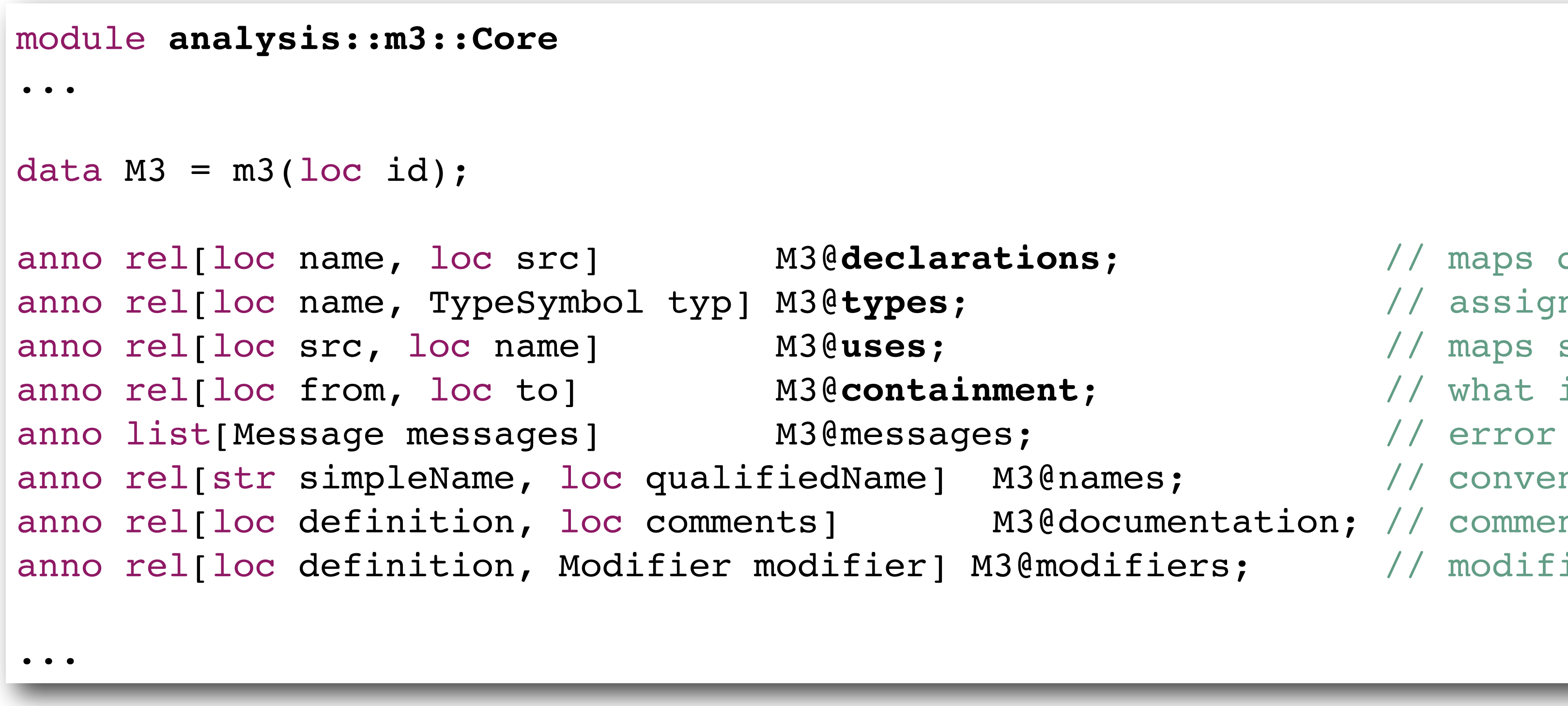
For example, a `Client` class has a `Name` attribute, conforming to the UML metamodel stating that classes may have attributes. The UML metamodel conforms to the MOF, which states that `Class` and `Attribute` (at M2) are classes (M3), which are related by an `Association` (belongs-to).

At level M3, `Class` and `Association` are both modeled as classes, and the relationships between them are modeled as associations.

What kinds of software models are useful?

It depends on the analysis task





```
module analysis::m3::Core
...

data M3 = m3(loc id);

anno rel[loc name, loc src]         M3@declarations;            // maps d
anno rel[loc name, TypeSymbol typ] M3@types;                   // assign
anno rel[loc src, loc name]         M3@uses;                    // maps s
anno rel[loc from, loc to]          M3@containment;             // what i
anno list[Message messages]         M3@messages;                // error
anno rel[str simpleName, loc qualifiedName]  M3@names;          // conver
anno rel[loc definition, loc comments]       M3@documentation;  // commen
anno rel[loc definition, Modifier modifier] M3@modifiers;       // modifi

...
```

FAMIX (Moose)

M3 (Rascal)

Software models may be very simple or complex, depending on the task you need to perform. FAMIX is a simple object-oriented metamodel for Moose, a data and software analysis system. It captures basic relations between software entities, but does not express program logic at the level of statements or expressions. M3 is a similar, relational metamodel for Rascal, a metaprogramming language integrated into Eclipse.

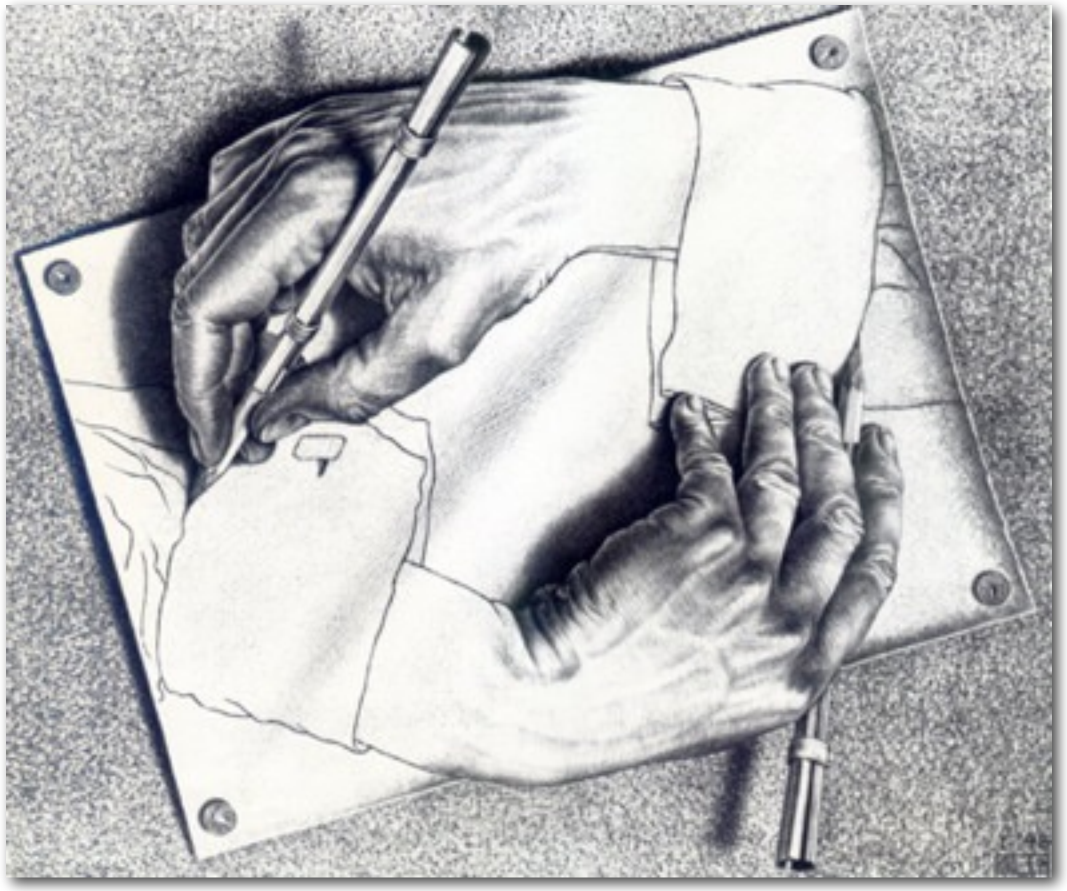
Both FAMIX and M3 can easily be extended to express different or more detailed kinds of information.

http://www.moosetechnology.org

http://www.rascal-mpl.org

Metaprogramming

A metaprogram is a program that manipulates a program *(possibly itself)*



The original metaprogramming system is Lisp, a language in which all data is represented as lists, including programs themselves. It is easy to write a Lisp interpreter in Lisp.

Metaprogramming is useful for generating programs, transforming progarms, analyzing them, and instrumenting them. All program development tools are essentially metaprograms.

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*."

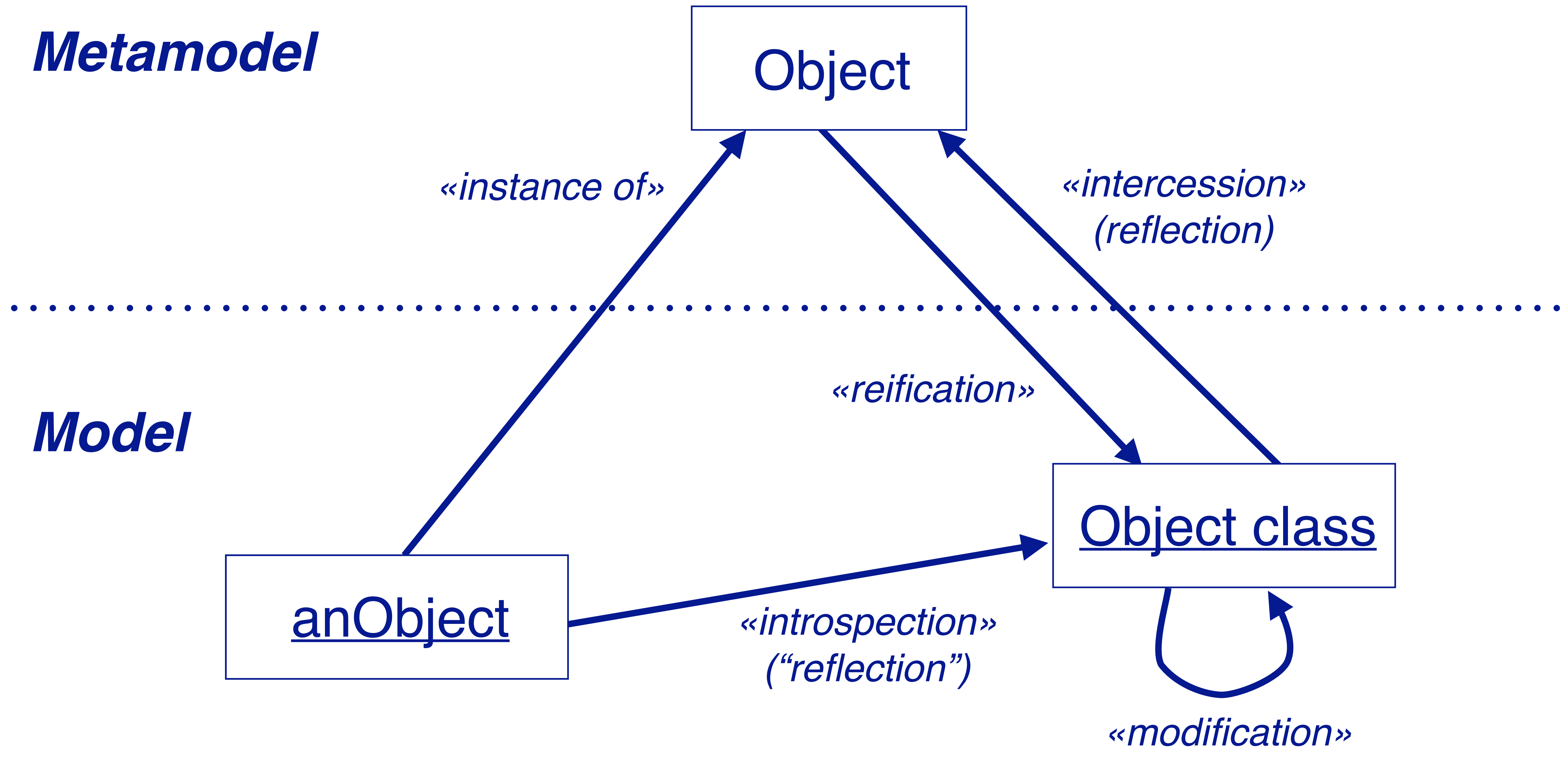
— *Bobrow, Gabriel & White, "CLOS in Context", 1993*

Many programming languages provide some form of reflective features, allowing programs to ask at run time questions about the running system.

We draw a strong distinction between "*introspection*", which only allows programs to query the running system, and "*intercession*", which allows a system to be modified while it is running.

In both cases we need to "*reify*" parts of the running system, i.e., *turn them into data (or objects)* that are accessible to the running program. With intercession, changes to these reified entities will actually be *reflected* back to the system itself.

Reflection and Reification



An *object* (`anObject`) at run time is a *model* (M1) entity that is an instance of a class `Object`, which is itself a *metamodel* (M2) entity.

We can query the class of an object by *reifying* the class, obtaining a model entity that represents the class (`Object class`). This entity lives in the M1 world, so we can interact with it and query it.

With *intercession*, if we change this model entity (for example, by adding or deleting some methods), then these changes will be *reflected* back to the metamodel (M2) level, *thus changing the behaviour of the system* and in particular of all instances of the class `Object`.

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

Pattie Maes provided a very influential description of reflection in object-oriented languages in this classic 1987 paper:

Pattie Maes. *Concepts and Experiments in Computational Reflection*. OOPSLA '87, pp. 147—155, December 1987.

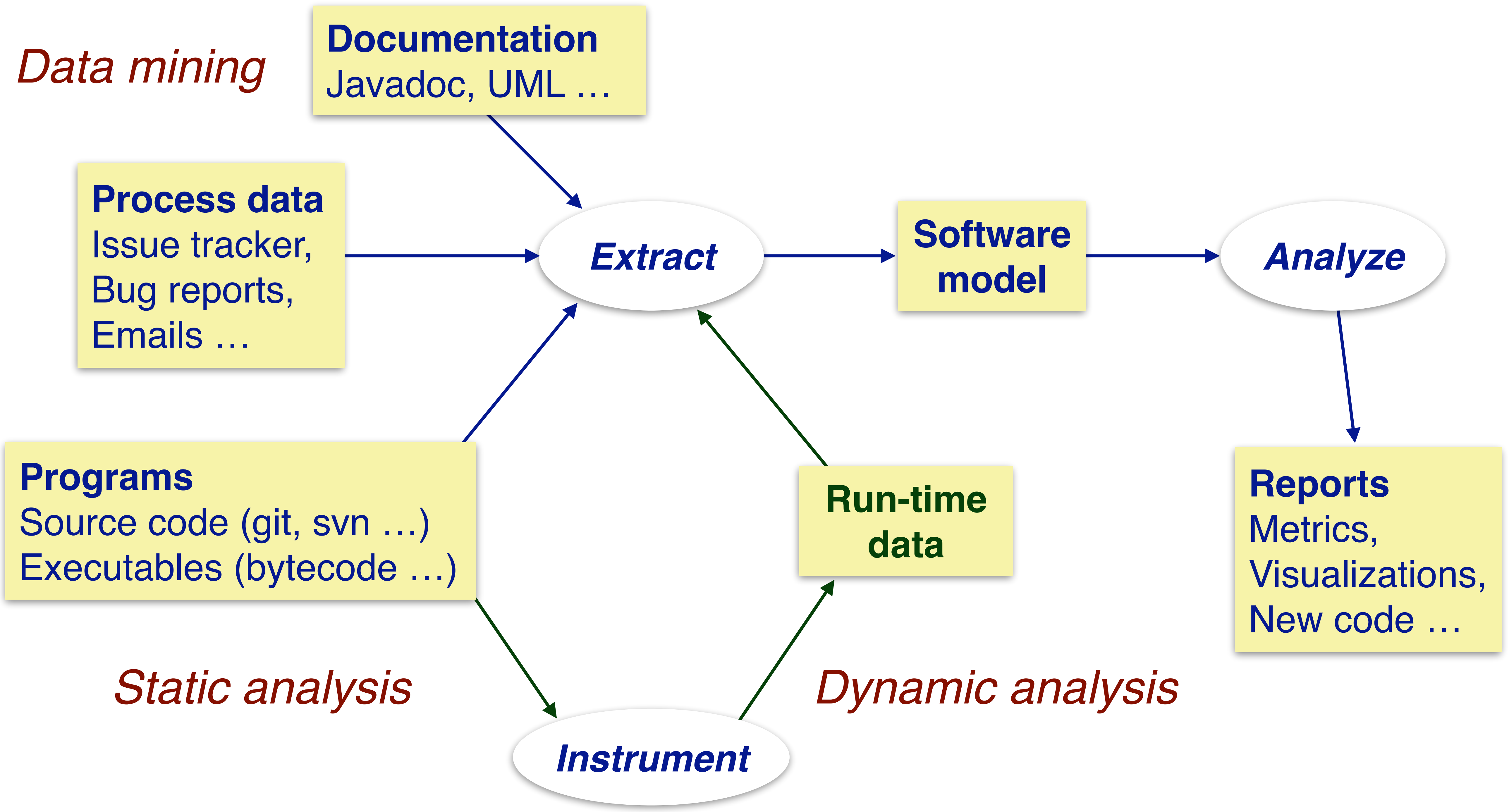
http://dx.doi.org/10.1145/38765.38821

http://scgresources.unibe.ch/Literature/OOPSLA/oopsla87/p147-maes.pdf

Roadmap

- Overview
- Laws of Software Evolution
- Reverse and Reengineering
- Software Modeling
- **Software Analysis**

What is Software Analysis?



Software analysis is concerned with various kinds of *software artifacts*, ranging from source code and executables, through to all other kinds of documents related to the software process (bug reports, documentation etc.).

To perform an analysis, one must first *extract some form of model* of the information of interest. The extraction process typically entails some kind of *parsing technology*. The extracted models may be very coarse (e.g., a list of file names and their sizes) or fine-grained (e.g., abstract syntax trees of the source code), depending on the task at hand.

The result of the analysis take any form, such as a textual or visual report, or even refactored or generated code.

We traditionally distinguish *static analysis* from *dynamic analysis*. The latter requires the code to be instrumented and executed, and a log of run-time data to be produced for further processing.

Whereas static and dynamic analysis focus on programs (source code or executables), *data mining* focuses on other kinds of software artifacts. All these techniques may be combined in software analysis.

What is static analysis?

Static software analysis refers to analysis performed based on source code (and other static data sources) alone, i.e., *without running the software.*

Sample tasks:
- dependency analysis
- method senders/receivers
- code clone detection

Static analysis tools:
- parsing technology
- introspection
- metrics
- type inference
- model checking
- data flow analysis
- symbolic execution

“Static analysis” simply refers to analysis that does not require the code to be executed. The term is commonly used in the context of compiler technology, to refer to analyses to improve the performance of generated code, or to reason about desirable properties (type safety, liveness, etc.).

In this course we focus on static analysis for *program comprehension* (reverse engineering) and *software maintenance* (reengineering) tasks. We will mostly focus on *lightweight static analyses* that do not require advanced techniques like model checking or data flow analysis.

What is dynamic analysis?

Dynamic software analysis refers to analysis based on data collected while running the studied software system.

Sample tasks:

- test coverage
- memory usage
- performance benchmarking

Dynamic analysis tools:

- metaprogramming technology
- bytecode instrumenters
- profilers

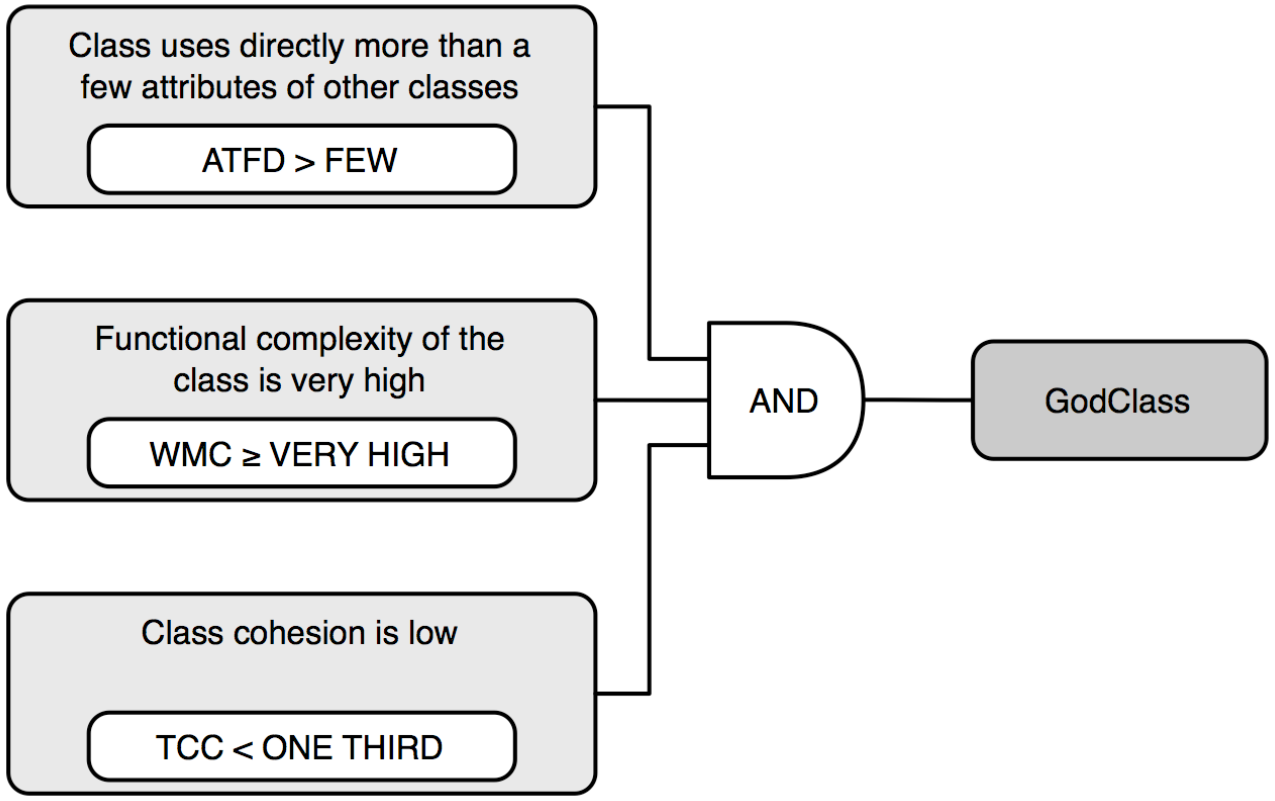
Dynamic analysis simply refers to any software anlysis that requires the code to be executed. This typically entails the code to be instrumented using reflection and metaprogramming techniques. Alternatively, in the case of bytecode machines, the virtual machine itself might provide the needed information without instrumentation of the code.

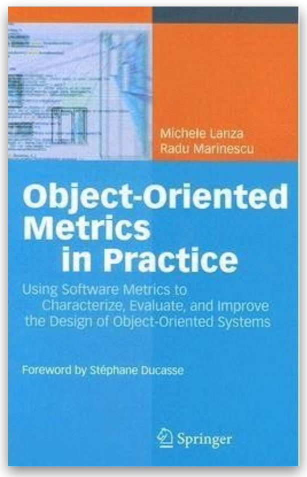
In any case, executing the code will produce a run-time log which can be further processed to produce the needed software model. Obviously dynamic analysis can also make use of statically obtained information.

Dynamic analysis is often used in virtual machines to improve run-time performance. In this course, as with static analysis, we focus instead on dynamic analyses that aid in program comprehension.

What are software metrics?

Software metrics *measure any attribute* of software or related entities.
A detection strategy is a *metrics-based predicate* to identify artifacts that conform to (or violate) a particular *design rule*


Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
GodClass


Michele Lanza
Radu Marinescu
Object-Oriented Metrics in Practice
Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems
Foreword by Stéphane Ducasse
Springer

Software metrics are used for many purposes in Software Engineering, for example in cost estimation (to estimate development effort) and process control (to track progress). We focus on software metrics to detect possible violations of design rules, or more generally to gain an understanding of the characteristics of a software system.

In this example, we use metrics with thresholds to detect possible "God classes" — classes that assume too much responsibility in the the system, and hence may pose difficulties for further development.

The acronyms refer to common metrics:

ATFD = access to foreign data

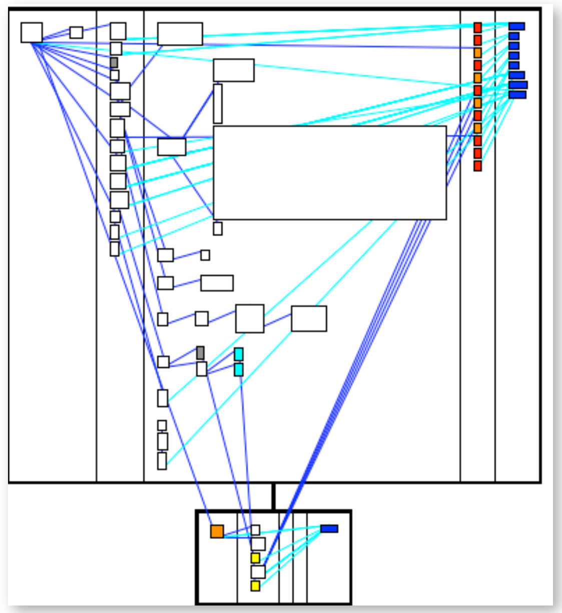
WMC = weighted method count

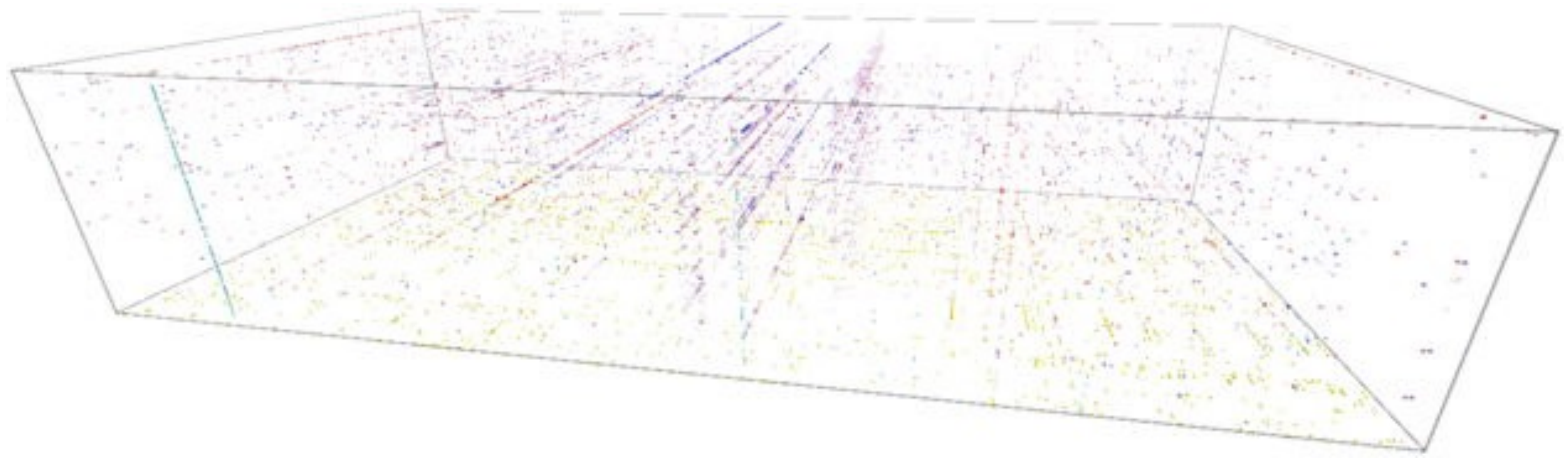
TCC = tight class cohesion

See: Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*, Springer-Verlag, 2006.

What is software visualization?

A software visualization maps selected attributes of software artifacts to a visual representation, in order to highlight some properties of that software.



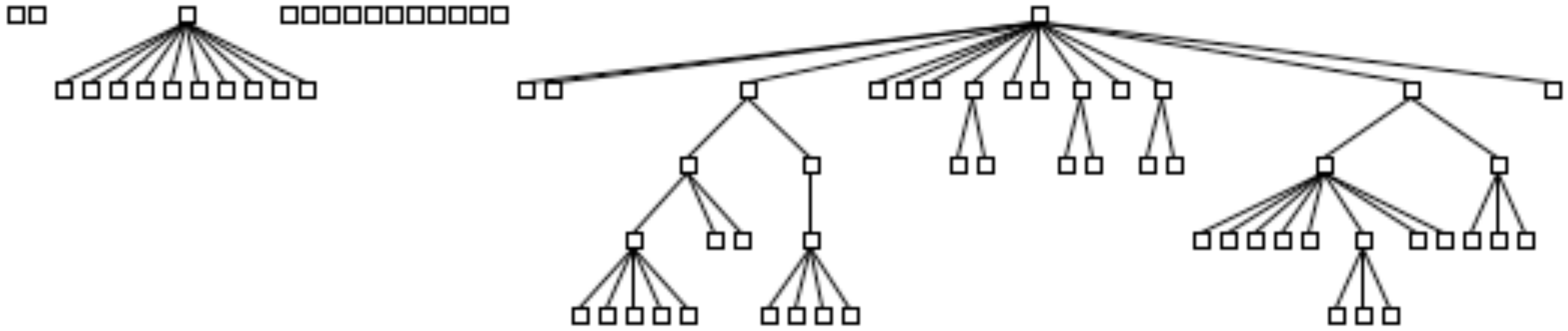


A software visualization may be *static* or *interactive*.
The visualization technique may be either *lightweight* (e.g., polymetric views) or *advanced* (e.g., spring layout).

The visualization on the left is a “class blueprint”, a so-called *polymetric view* that maps software metrics to a simple visualization. The large boxes represent classes and the small ones are methods and attributes. From left to right we see 1) creation methods, 2) public interface methods, 3) internal methods, 4) accessors, 5) attributes. Calls between methods are visualized. The height and width of the method boxes can represent various metrics (e.g., lines of code, number of invocations).

The second visualization shows the evolution of “code critiques” (metrics indicating possible code quality issues) over time. by navigating through the 3D space, one can get an idea where the quality of the system improved or degraded over time.

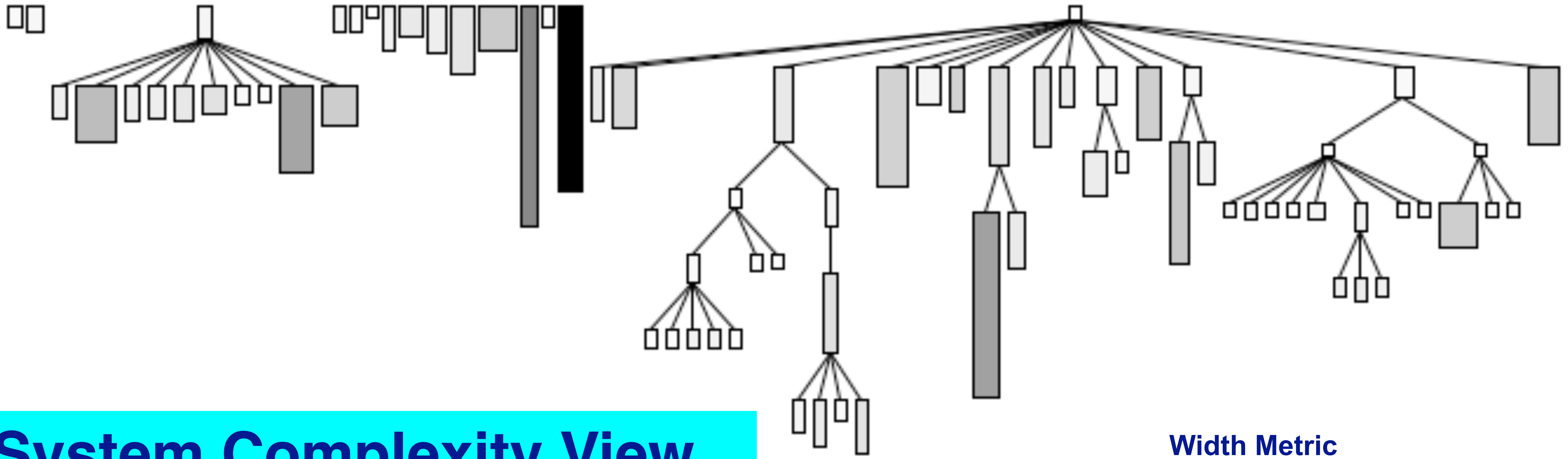
A simple class hierarchy view



Nodes = Classes
Edges = Inheritance Relationships

A simple class hierarchy view does not give much insight into a software system.

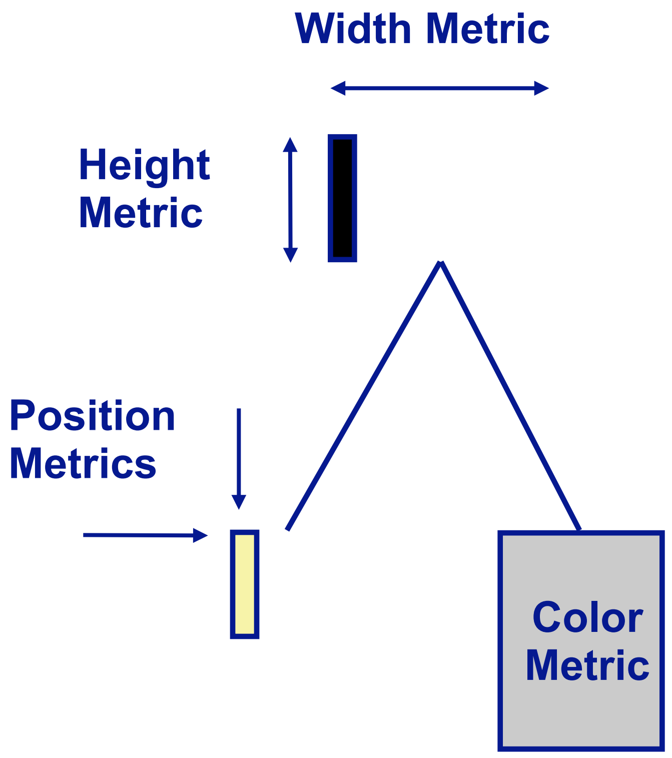
System Complexity View



System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

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



The System Complexity View is a basic *polymetric view* that maps software metrics to the height, width and color of the class nodes in the hierarchy.

Interestingly, outliers jump out in such a visualization, giving you some insight into the system.

For example: a tall, narrow, dark node has many methods, many lines of code per method, and few attributes. This suggests that the class represents an algorithm.

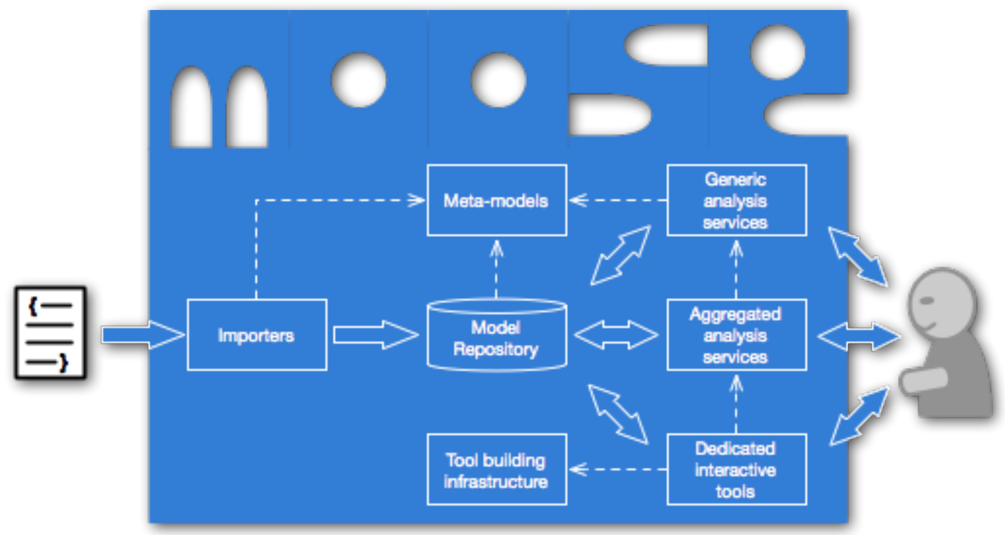
What else can you learn from this picture?

See: Michele Lanza and Stéphane Ducasse. *Polymetric Views—A Lightweight Visual Approach to Reverse Engineering*. IEEE TSE 29(9) p. 782—795, Sept. 2003.

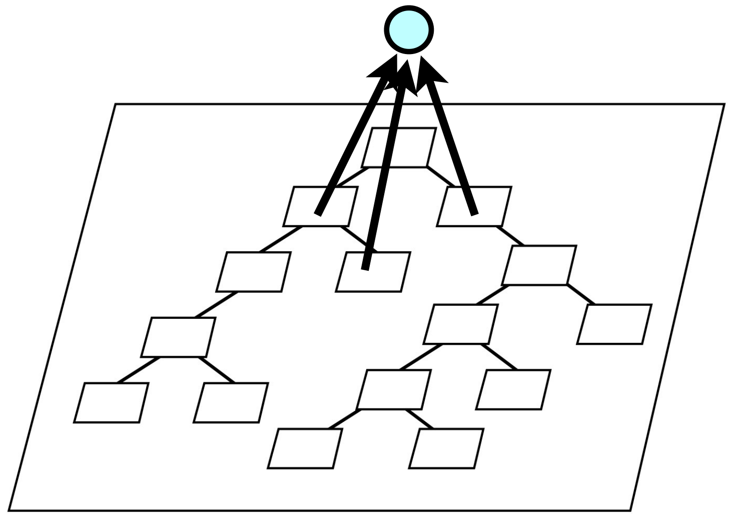
http://dx.doi.org/10.1109/TSE.2003.1232284

http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf

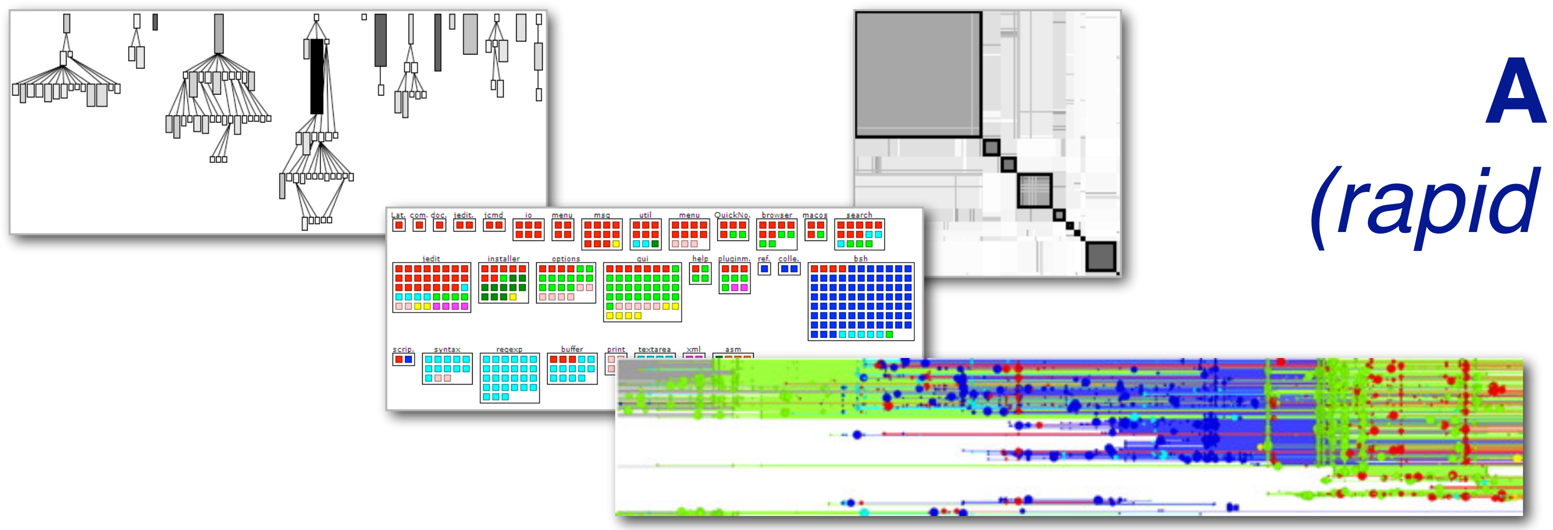
Why is Smalltalk interesting for Software Analysis?


Meta-models
Generic analysis services
Importers
Model Repository
Aggregated analysis services
Tool building infrastructure
Dedicated interactive tools

Modeling
(live, fully OO)



Instrumentation
(dynamic adaptation)



Analysis
(rapid prototyping)

In this course we will make heavy (thought not exclusive) use of Smalltalk. Smalltalk is especially interesting because it offers a live programming environment that supports metaprogramming with the help of a mature and rich metamodel. These features make Smalltalk an excellent prototyping environment, as well as a good platform for modeling, instrumentation and analysis.

We will use the Pharo dialect of Smalltalk, and the Moose data and software analysis platform.

- http://pharo.org
- http://files.pharo.org/books/
- http://www.moosetechnology.org

What you should know!

> What are "*legacy*" software systems? What problems do they suffer from?

> Why is "*software maintenance*" really continuous development?

> How does *reverse-engineering* differ from *reengineering*?

> What is the difference between *reflection* and *reification*? Between *introspection* and *intercession*?

> What is *metaprogramming*?

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

> What role do *software metrics* play in software analysis?

Can you answer these questions?

> Why must real-world programs *change* or become *less useful* over time?

> Why do *successful* software systems always require *more maintenance*?

> Why is *duplicated code* a problem in legacy software?

> What is the relationship between a *model* and its *meta-model*?

> Why do we seldom need to consider *meta-meta-meta-models* (level M4)?

> What kind of “*reflection*” does Java support?