UNIVERSITÄT BERN

Compositional Programming

Oscar Nierstrasz

Software Composition Group
scg.unibe.ch

Dagstuhl Seminar 12511, Dec 2012

http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=12511

Roadmap

Early history

Objects

Components

Features

Metaprogramming

Conclusions

Roadmap

Early history

Objects

Components

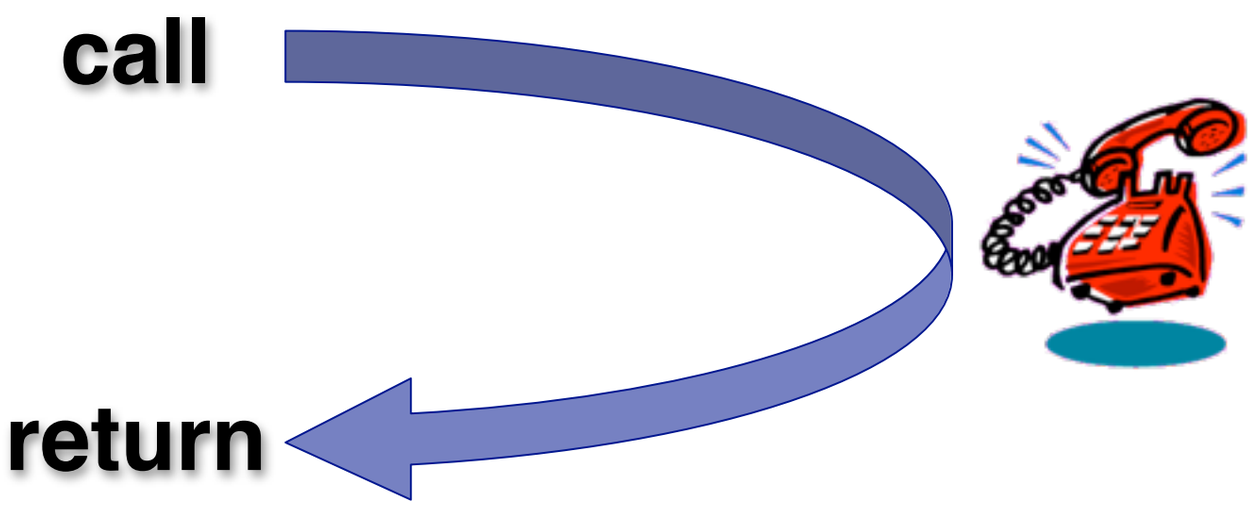
Features

Metaprogramming

Conclusions

Subroutines (1949)

Repeatedly invoke common code sequences



David Wheeler is credited with the invention of the “closed subroutine”. Dijkstra points to this as one of the most fundamental contributions to PL design.

Wheeler is often quoted as saying "Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem."

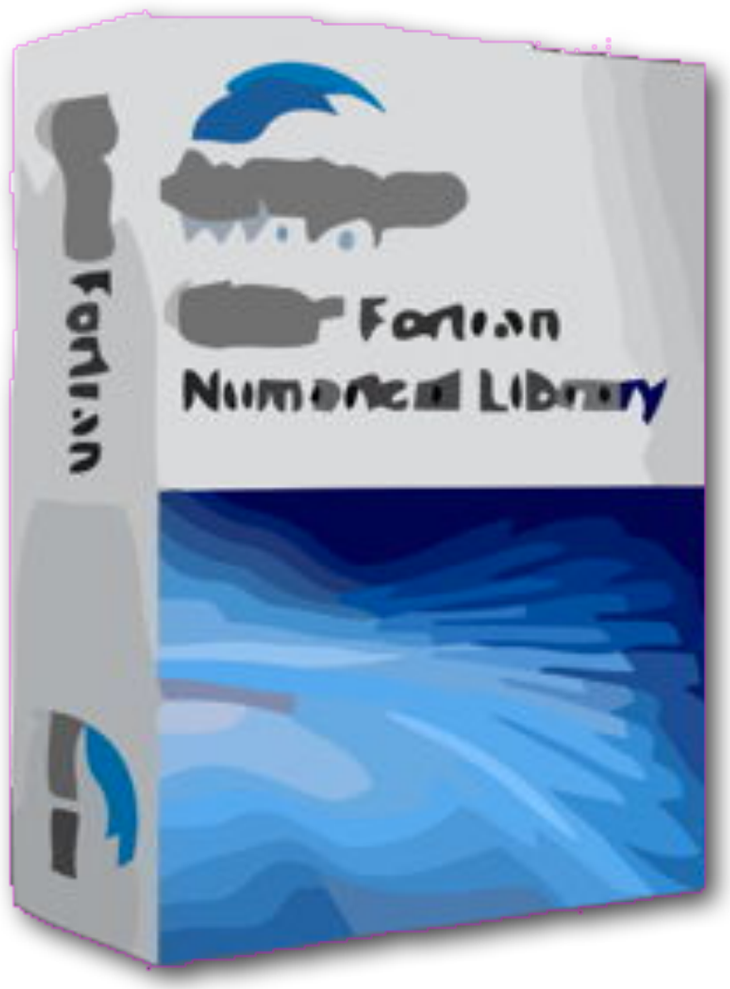
Another quotation attributed to him is "Compatibility means deliberately repeating other people's mistakes.”

http://en.wikipedia.org/wiki/Subroutine#cite_note-2

http://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)

EDSAC computer

Libraries — FORTRAN II (1958)



Large reusable libraries of scientific functions led to the long-term success of FORTRAN

User subroutines are introduced in FORTRAN II (1958).

Recursion — ALGOL (1958)

ALGOL brought recursion into the mainstream

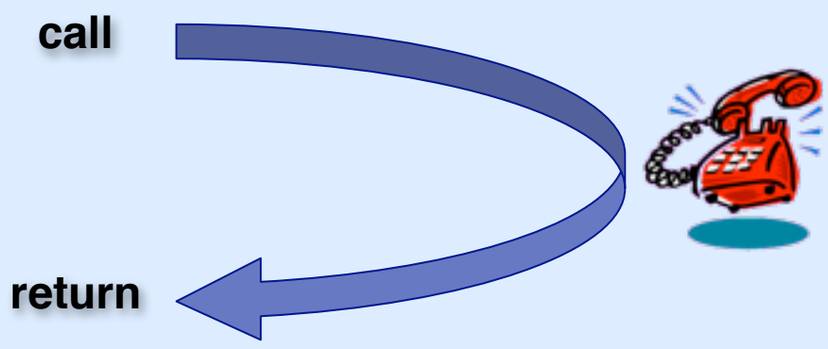
FORTRAN did not support reentrant procedures. ALGOL introduced a run-time stack to support recursive procedures, but the impact was only realized later.

Modules — COBOL (1959)

Modules enabled the stepwise decomposition of large software systems

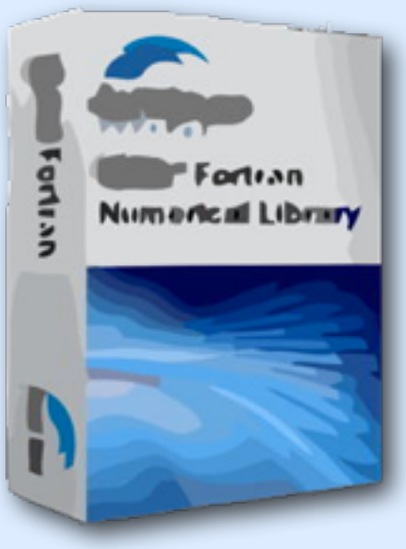
Cobol tried to be readable (for managers) but ended up just being verbose.
Still the most widely used PL today.
Main innovation was in supporting modular programming.

Summary



Paradigm: procedural composition

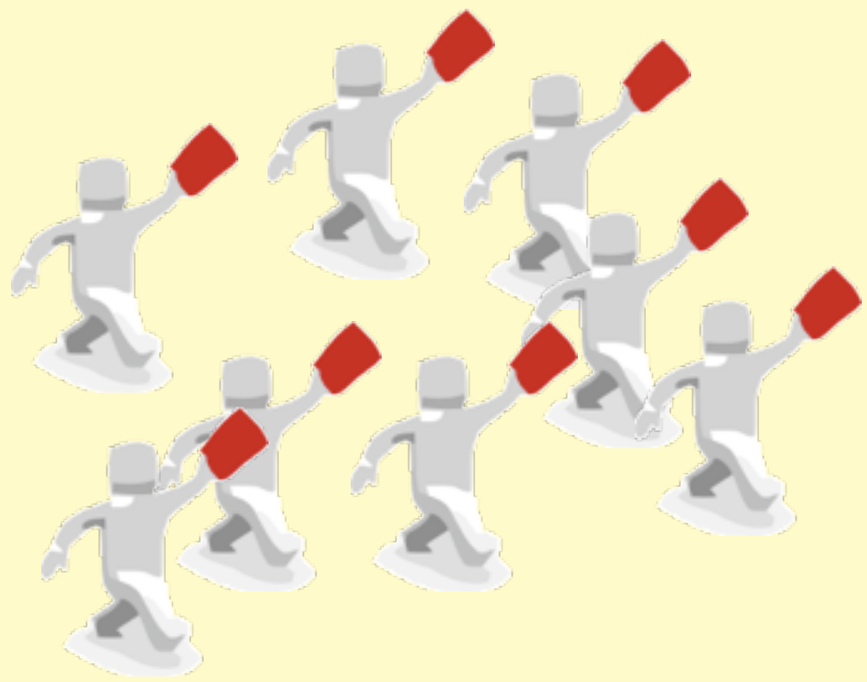
Motivation: code reuse, managing complexity



Roadmap

Early history

Objects



Components

Features

Metaprogramming

Conclusions

Data Abstraction

Abstraction = *elimination of inessential detail*

Information hiding = *providing only the information a client needs to know*

Encapsulation = *bundling operations to access related data as a data abstraction*

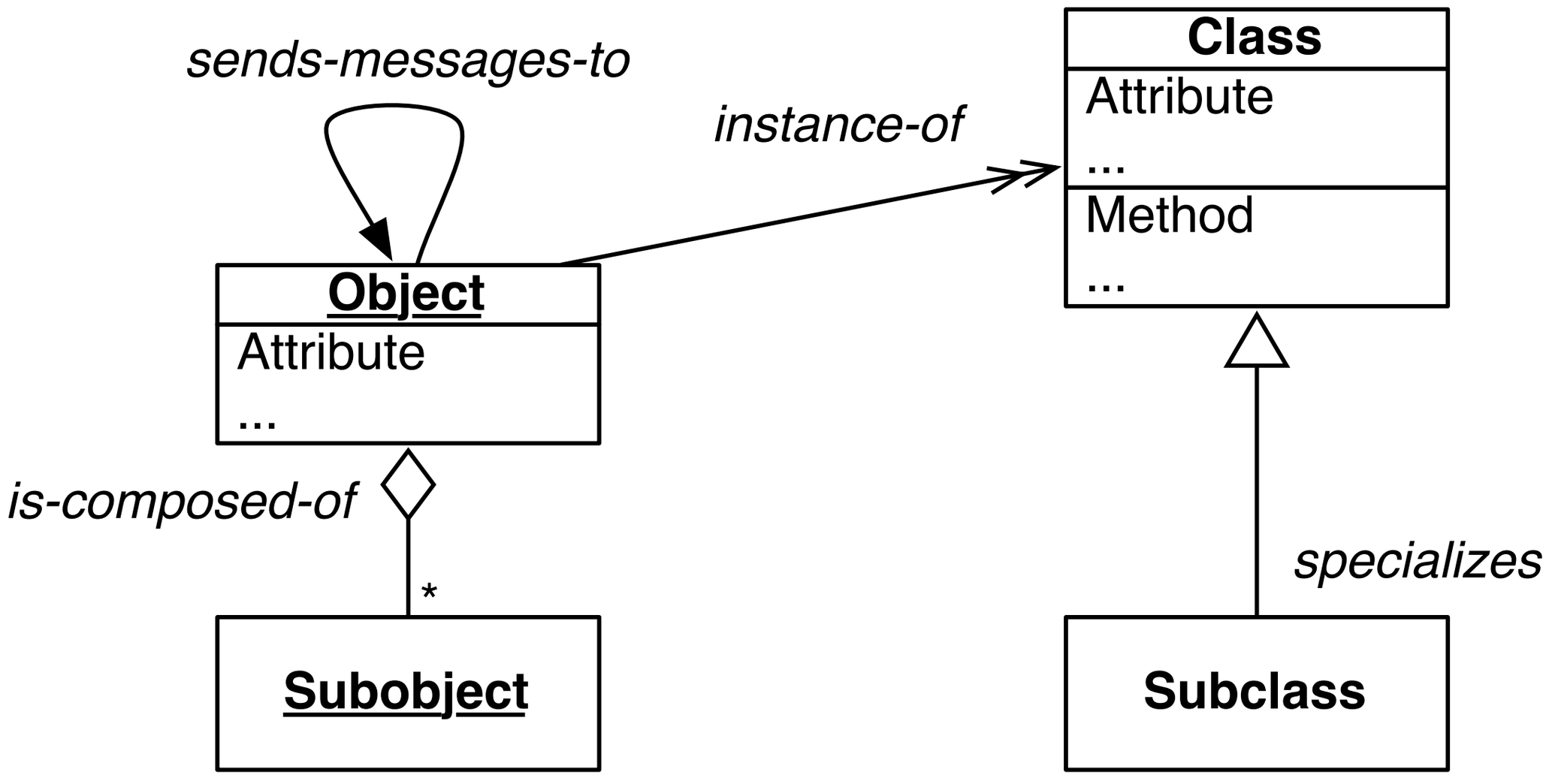
In object-oriented languages we can implement data abstractions as classes.

Edward V. Berard, “Abstraction, Encapsulation, and Information Hiding” in Essays On Object-Oriented Software Engineering, 1993.

These three concepts are often confused, but in fact any one may be present without the other two.

See also: William Cook, OOPSLA 2009 for a discussion on the distinction between data abstractions and abstract data types.

Object-Oriented Programming (1962)



OOP was introduced in Simula as an extension to ALGOL to model simulations.

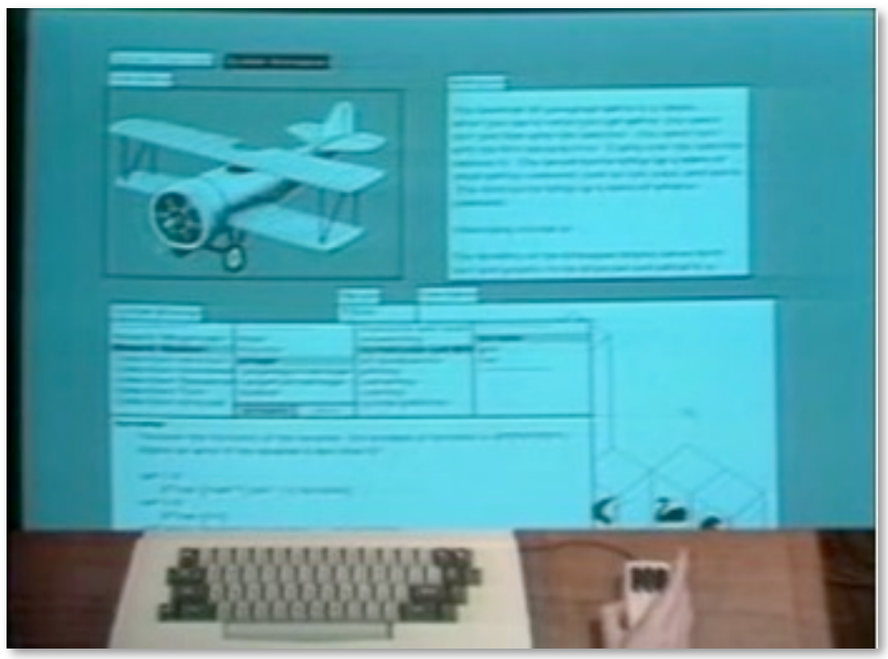
OOP introduces objects, class and inheritance.
Objects are composed of subobjects, and subclasses specializations of superclasses.

Smalltalk (1972)

In “pure” OOP, objects are used to model all aspects of design

Everything is an object

Everything happens by sending messages



```
5 factorial → 120
```

```
Integer»factorial
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'
```

Alan C. Kay. *The Early History of Smalltalk*. ACM SIGPLAN Notices, March 1993.
http://www.smalltalk.org/smalltalk/TheEarlyHistoryOfSmalltalk_Abstract.html

Smalltalk was the first language to use objects as the only basis for programming.
It was inspired by the need for a new language and run-time system needed for the next generation of interactive workstations.

The open / closed principle

Software entities should be *open for extension*, but *closed for modification*.

"In other words, we want to be able to change what the modules do, without changing the source code of the modules."

Bertrand Meyer, *Object-Oriented Software Construction*, 1988.
See also: http://www.objectmentor.com/resources/articles/ocp.pdf

Example: Class — instantiate as an encapsulated object; extend as a subclass
Component — fixed interface; hooks to plug in new behaviour (cf eclipse)

Design by Contract

Services should specify clear contracts

"If you promise to call S with the *precondition* satisfied, then I, in return, promise to deliver a final state in which the *post-condition* is satisfied."

If the precondition fails, it is the client's fault.

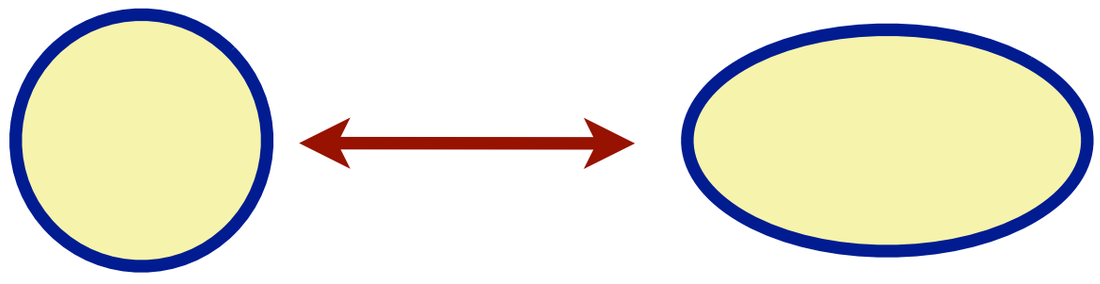
If the postcondition fails, it is the supplier's fault.
If the class invariant fails, it is the supplier's fault.

Bertrand Meyer, *Object-Oriented Software Construction*, 1988.
See also: http://www.objectmentor.com/resources/articles/ocp.pdf

DbC is one of the foundations of OO design. It simplifies design decisions, and leads to the development of more robust software by formalizing the expectations of clients and suppliers of services.

Principle of Substitutability

An instance of a subtype can always be used in any context in which an instance of a supertype was expected.



substitutable?

Peter Wegner, Stanley Zdonik. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*. ECOOP 1988. http://www.ifs.uni-linz.ac.at/~ecoop/cd/tocs/t0322.htm

Wegner and Zdonik made a first attempt to formulate the notion of “plug compatibility” between objects. The notion is still rather informal — Is a Circle an Ellipse? Depends on the contract clients expect! (Ditto for Square and Rectangle.)

Liskov substitution principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S, where S is a subtype of T.

Restated in terms of *contracts*, a derived class is substitutable for its base class if:

- *Its preconditions are no stronger than the base class method.*
- *Its postconditions are no weaker than the base class method.*

Barbara Liskov, Jeannette M. Wing. *A behavioral notion of subtyping*. ACM TOPLAS, 1994.
http://www.cse.ohio-state.edu/~neelam/courses/788/lwb.pdf

Note that Liskov and Wing actually refer to a much stronger notion of behavioral substitutability than Uncle Bob (or Wegner and Zdonik do), and is much stronger than what OO programs usually require. It all depends on how strong your type system is!

Polymorphism

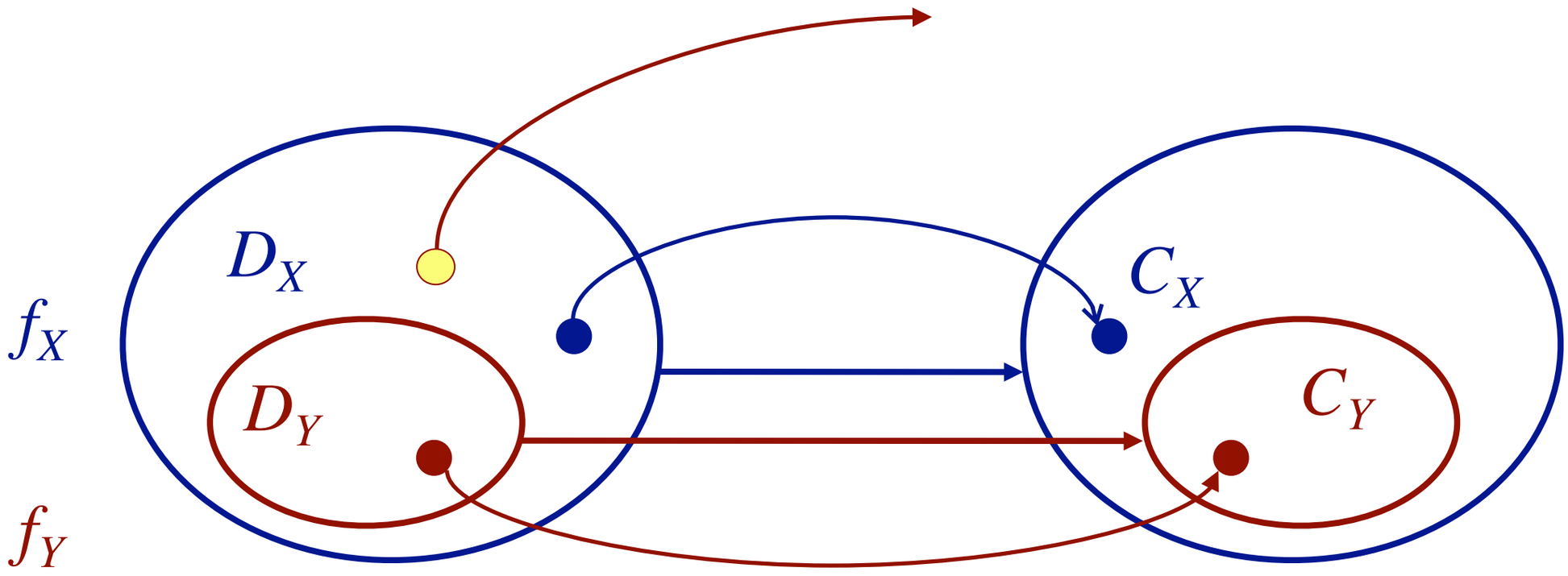
Polymorphic client code does not depend on the concrete type of the service provider

- ***Universal:***
  - —Parametric: map function in Haskell
  - —Inclusion: subtyping — graphic objects
- ***Ad Hoc:***
  - —Overloading: integer vs real addition
  - —Coercion: automatic conversion from ints to floats

Luca Cardelli and Peter Wegner. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys 17(4) p. 471—522, 1985

Polymorphism is often confused with “dynamic binding”.
It simply means that entities may have many types, whereas in monomorphic languages (like Pascal or C) entities have unique types.
Universal polymorphism means one function accepts many types. With ad hoc polymorphism, there are actually many functions with the same name.
Java and C++ support all four kinds of polymorphism.
Polymorphism is useful because it enables generic client code to be written that does not depend on the concrete type of the service provided.

Covariant subtyping

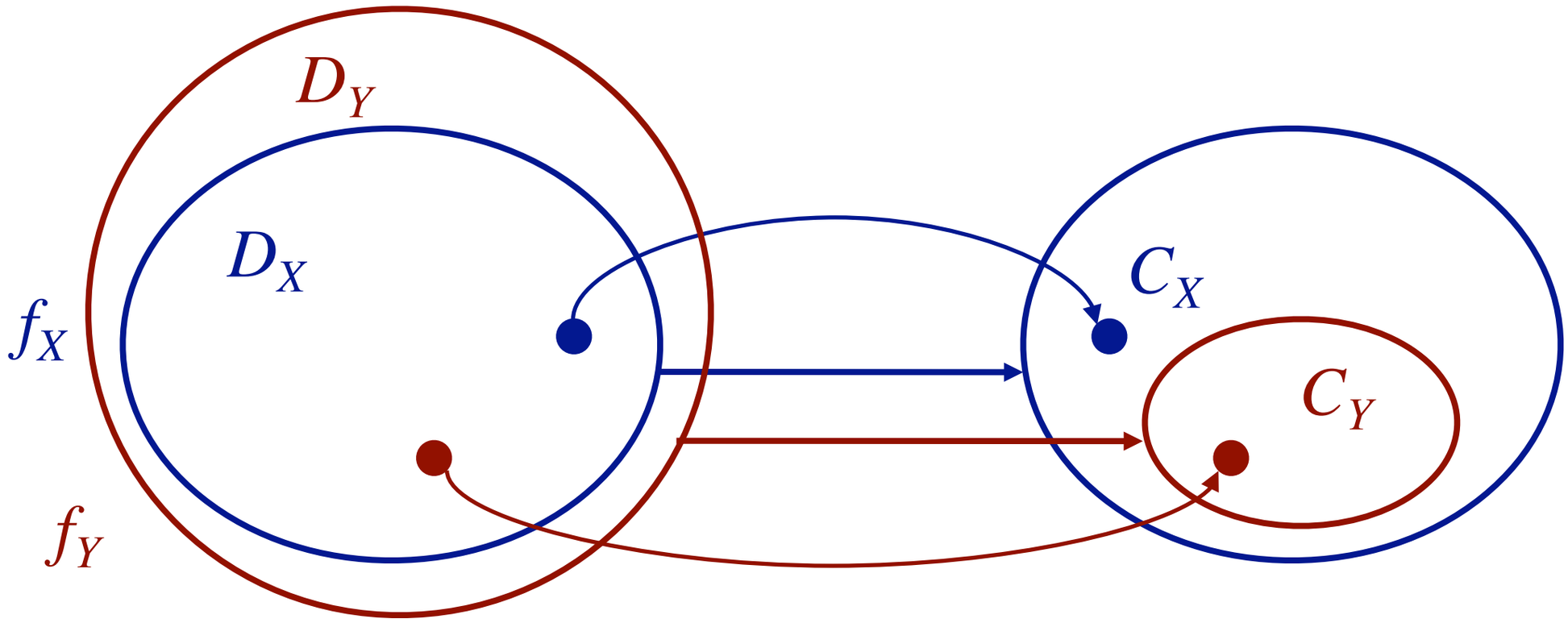


A client who expects the behaviour of f_X, and applies f_Y to a value in D_X might get a run-time type error.

Anthony J. H. Simons, *"The Theory of Classification"*, Parts 1-3, Journal of Object Technology, 2002-2003, www.jot.fm.

Covariance intuitively makes sense but is unsafe.
It is supported in Eiffel, but is caught at runtime ("CAT-calls" — covariant argument type).

Contravariant subtyping



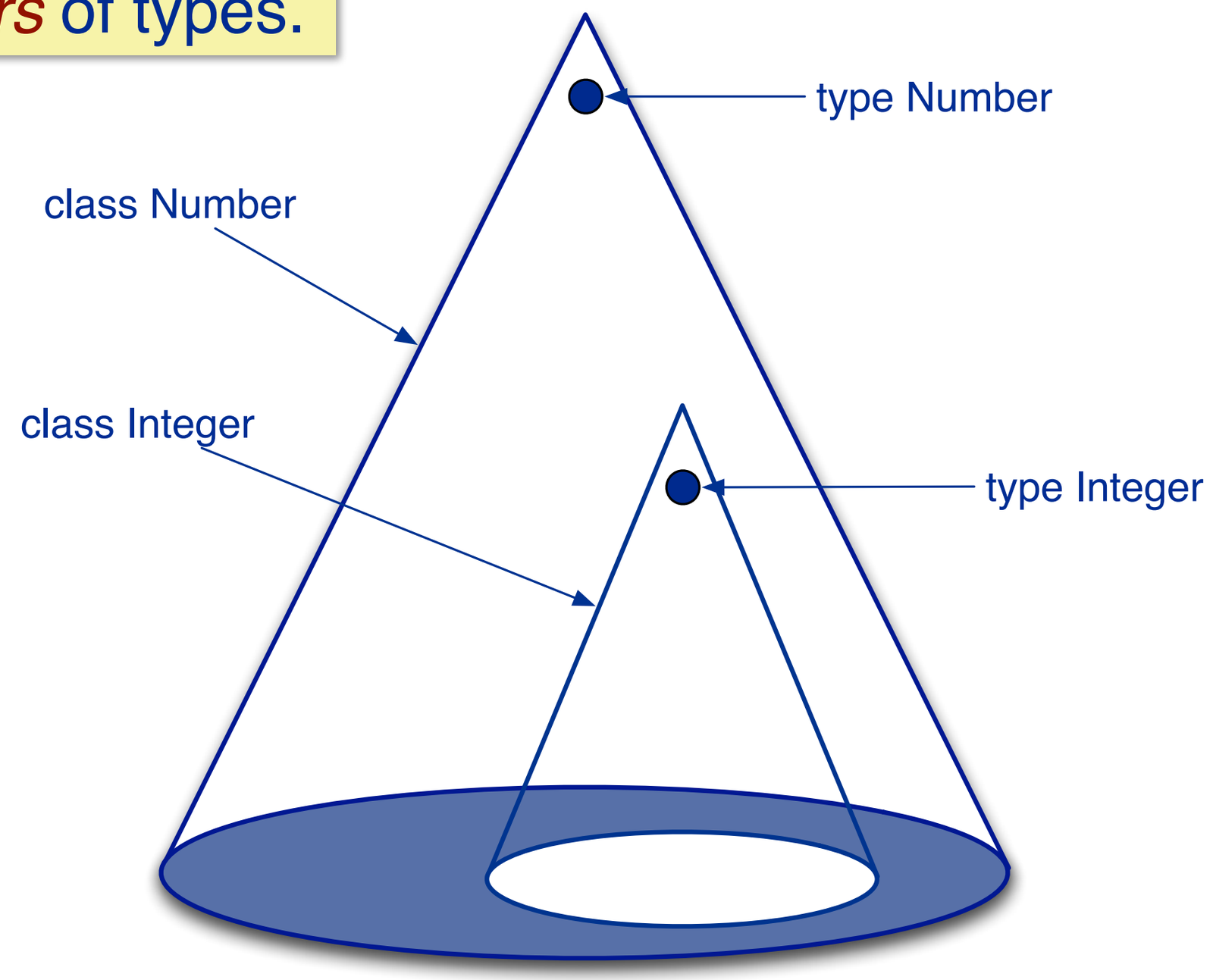
A contravariant result type guarantees that the client will receive no unexpected results.

Anthony J. H. Simons, *"The Theory of Classification"*, Parts 1-3, Journal of Object Technology, 2002-2003, www.jot.fm.

But contravariance runs counter to intuition, so is rarely used in real languages.
Java requires that overridden methods have the same signature. Methods with the same name but different signatures are actually overloaded, opening a whole other can of worms.

Types vs Classes

Classes are *generators* of types.



"Classes are nested volumes in the space of types. Types are points at the apex of each bounded volume."

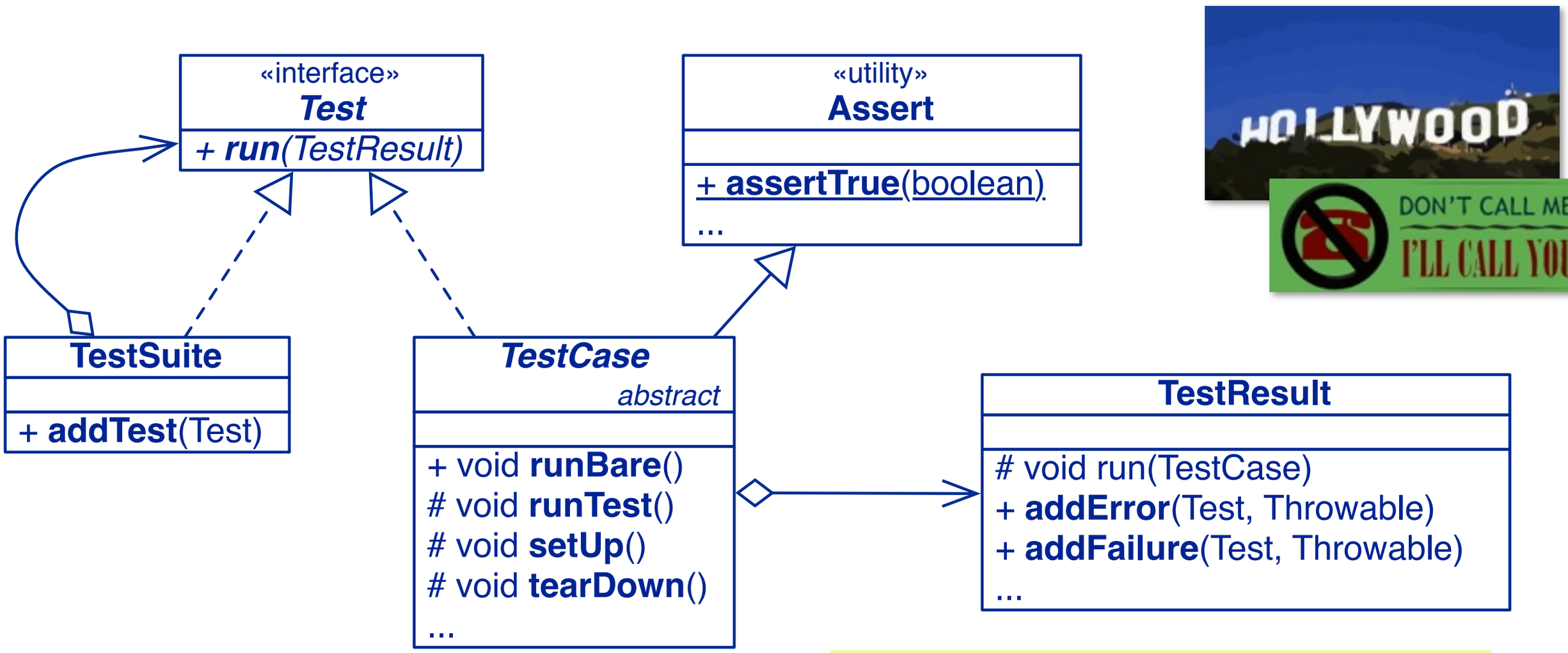
Anthony J. H. Simons, *"The Theory of Classification"*, Parts 4-8, Journal of Object Technology, 2002-2003, www.jot.fm.

Classes are generators of types, and represent families of types.

A specific object has a given, specific type. A class may instantiate a given object, but may also be used to generate subclasses, so it represents a whole family of types.

Frameworks

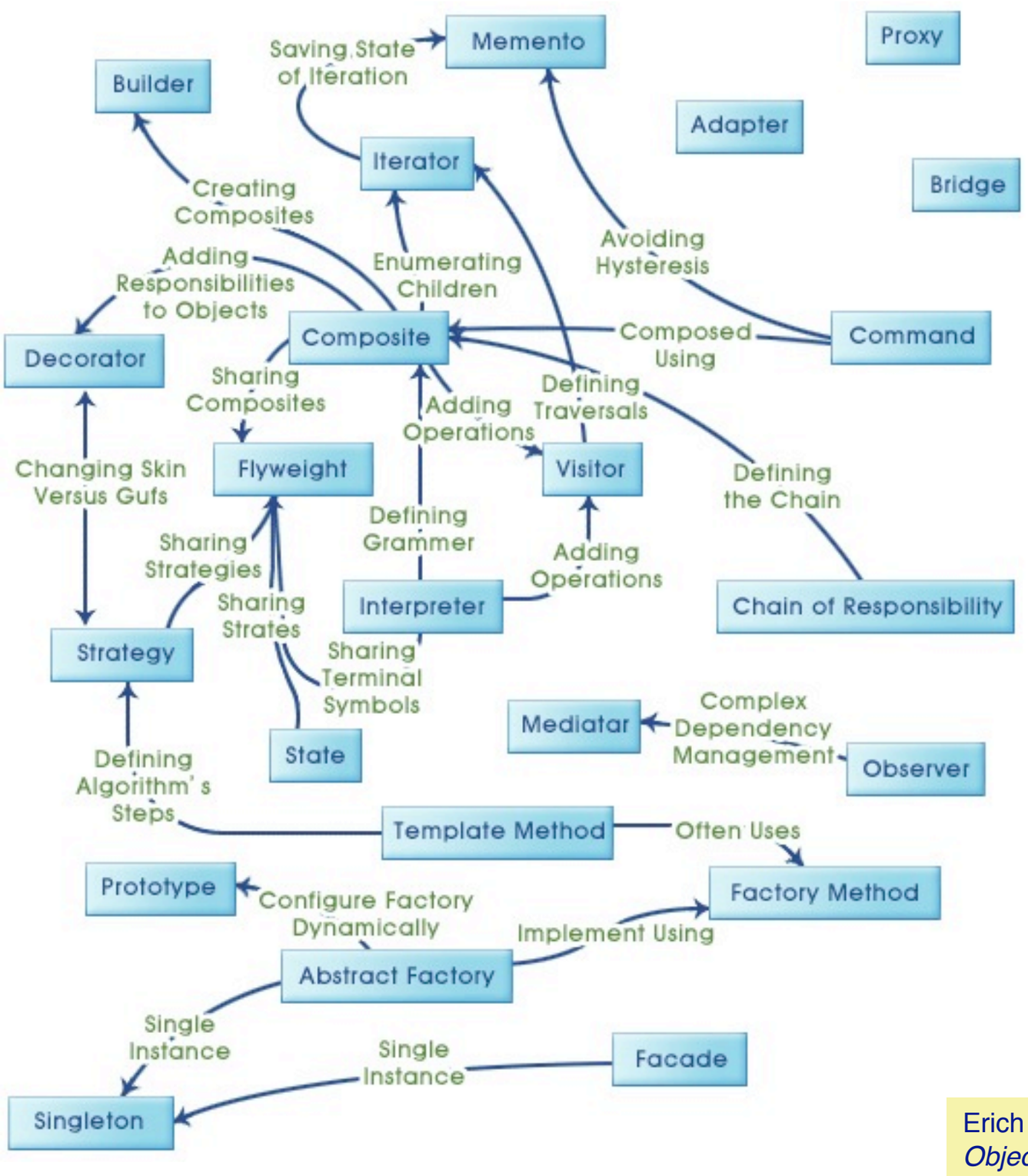
White box frameworks define a generic application that you instantiate by subclassing.



Ralph E. Johnson & Brian Foote. *Designing Reusable Classes.* Journal of Object-Oriented Programming, June/July 1988. http://www.laputan.org/drc.html

Frameworks reverse the usual flow of control: you don't call them; they call you!
The problem with white box frameworks is that the contracts for subclassing the Framework classes are implicit

Design Patterns



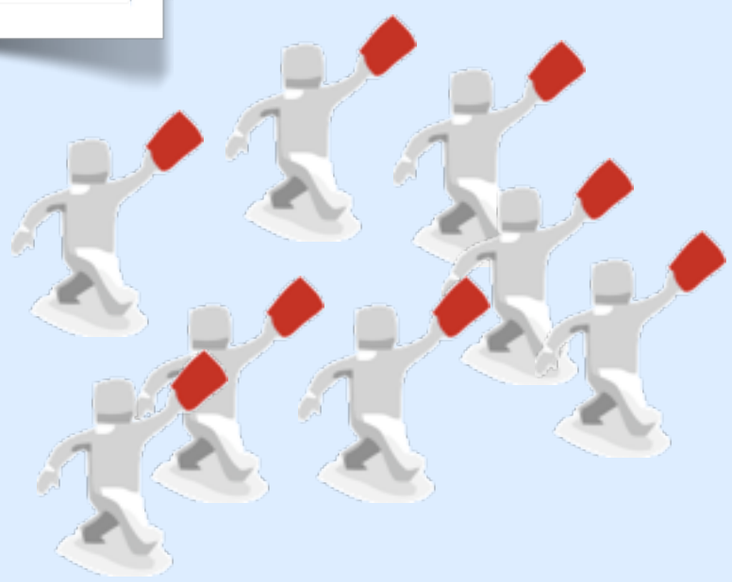
Patterns document common language-independent solutions to design problems.

Most of the GOF patterns achieve flexibility by adding a level of indirection.

Erich Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

Most patterns define a set of collaborating roles, which are to be played by objects of your concrete design.

Summary



Paradigm: object composition + incremental refinement

Motivation: domain modeling

Roadmap

Early history

Objects

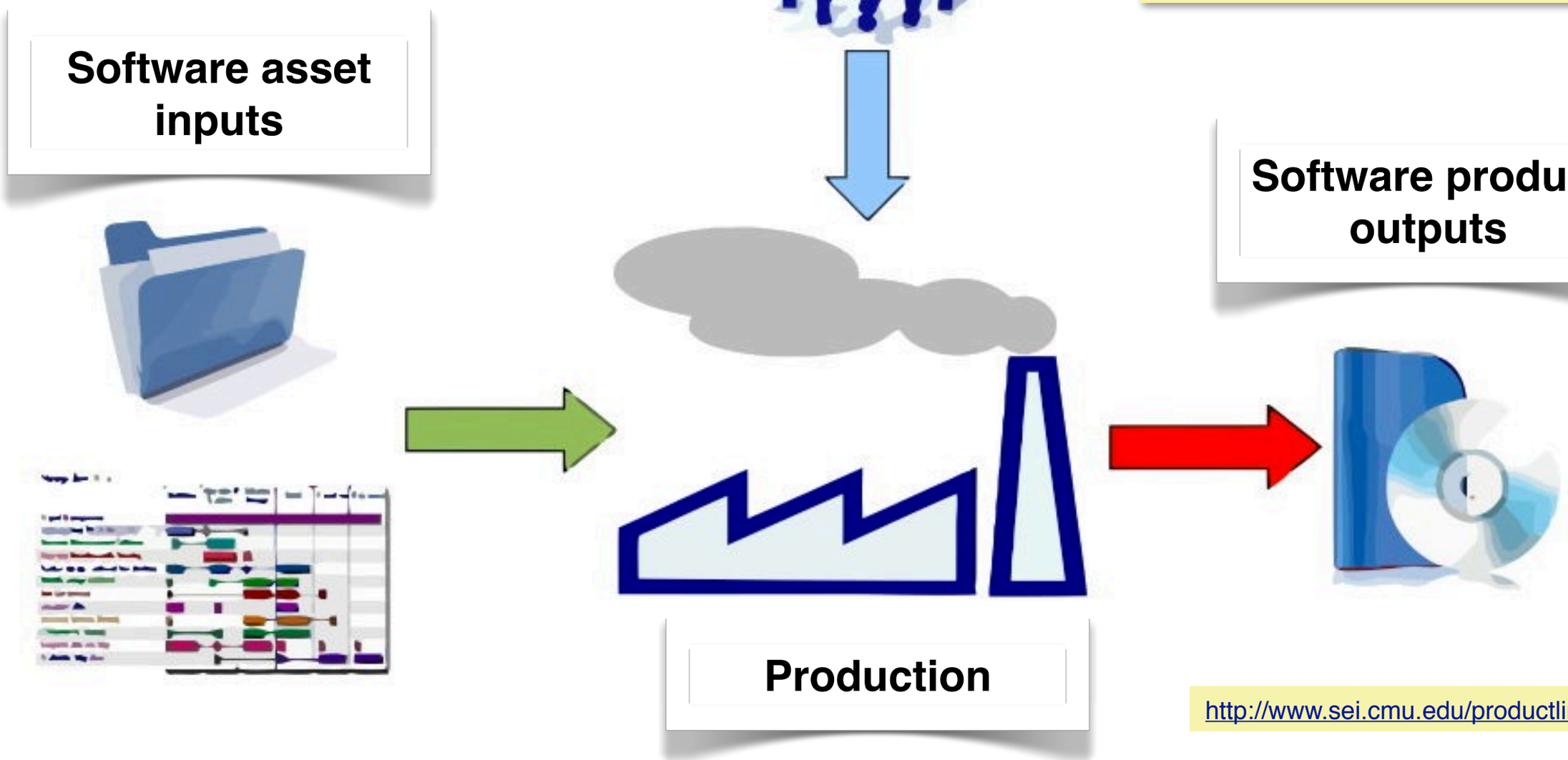
Components

Features

Metaprogramming

Conclusions

Software Product Lines



A SPL is a domain specific framework for producing a range of related applications.
Key concern: managing variation. Can use a range of techniques.

Software components

A software component is a *unit of composition* with contractually specified *interfaces* and *explicit context dependencies* only.

A software component can be *deployed independently* and is subject to composition by third parties.

Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed.. Boston, MA: Addison-Wesley, 2002.

NB: This definition emphasizes composition, not refinement.

But what is a software component?

an object
a class
a template
a method
a procedure
a mixin
a trait
a module
a package
a subsystem
a framework
a script
a service
a metaobject
a metaclass
a design pattern
...
?

it depends ...

Applications = Components + Scripts

Components
both *import* and
export services

Scripts *plug*
components
together

A scripting language is a dedicated language for for orchestrating a set of tasks (or components).

Jean-Guy Schneider, Oscar Nierstrasz. *Components, Scripts and Glue*.
In Software Architectures — Advances and Applications, Springer-Verlag, 1999.

This definition emphasizes the need to configure components.
Components should be plug-compatible, so they are plugged — not wired — together.
Sometimes “glue” is additionally needed to adapt components to fit together.

DSLs

```
SELECT *
    FROM Book
    WHERE price > 100.00
    ORDER BY title;
```

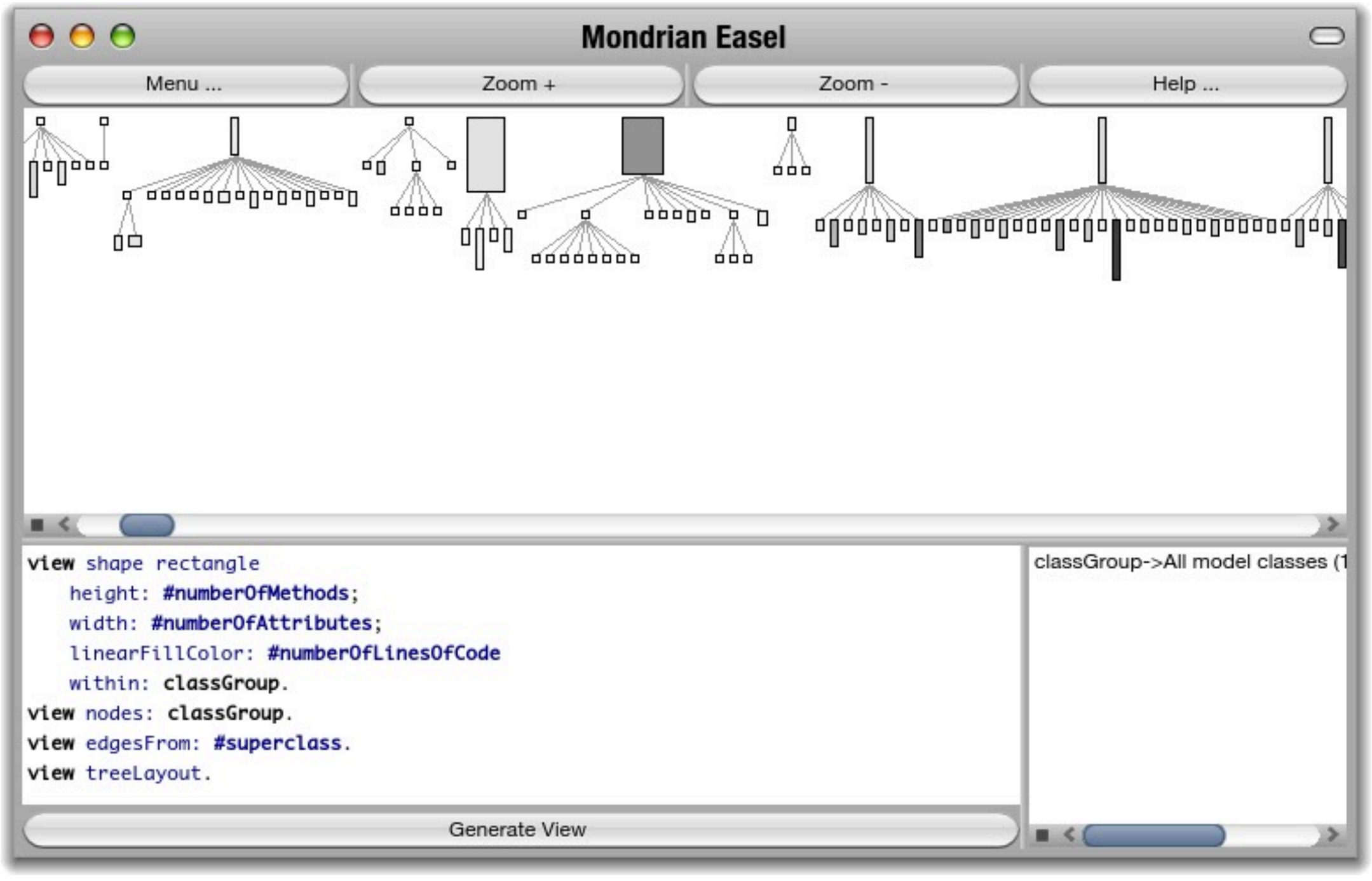
A DSL is a dedicated programming or specification language for a specific problem domain

```
cat Notes.txt
| tr -c '[:alpha:]' '\012'
| sed '/^$/d'
| sort
| uniq –c
| sort –rn
| head -5
```

```
14 programming
14 languages
 9 of
 7 for
 5 the
```

SQL is dedicated to the domain of manipulating tables.
The Bourne shell is dedicated to scripting Unix commands.
Neither language can be used without the components it is intended to script.

Internal DSLs



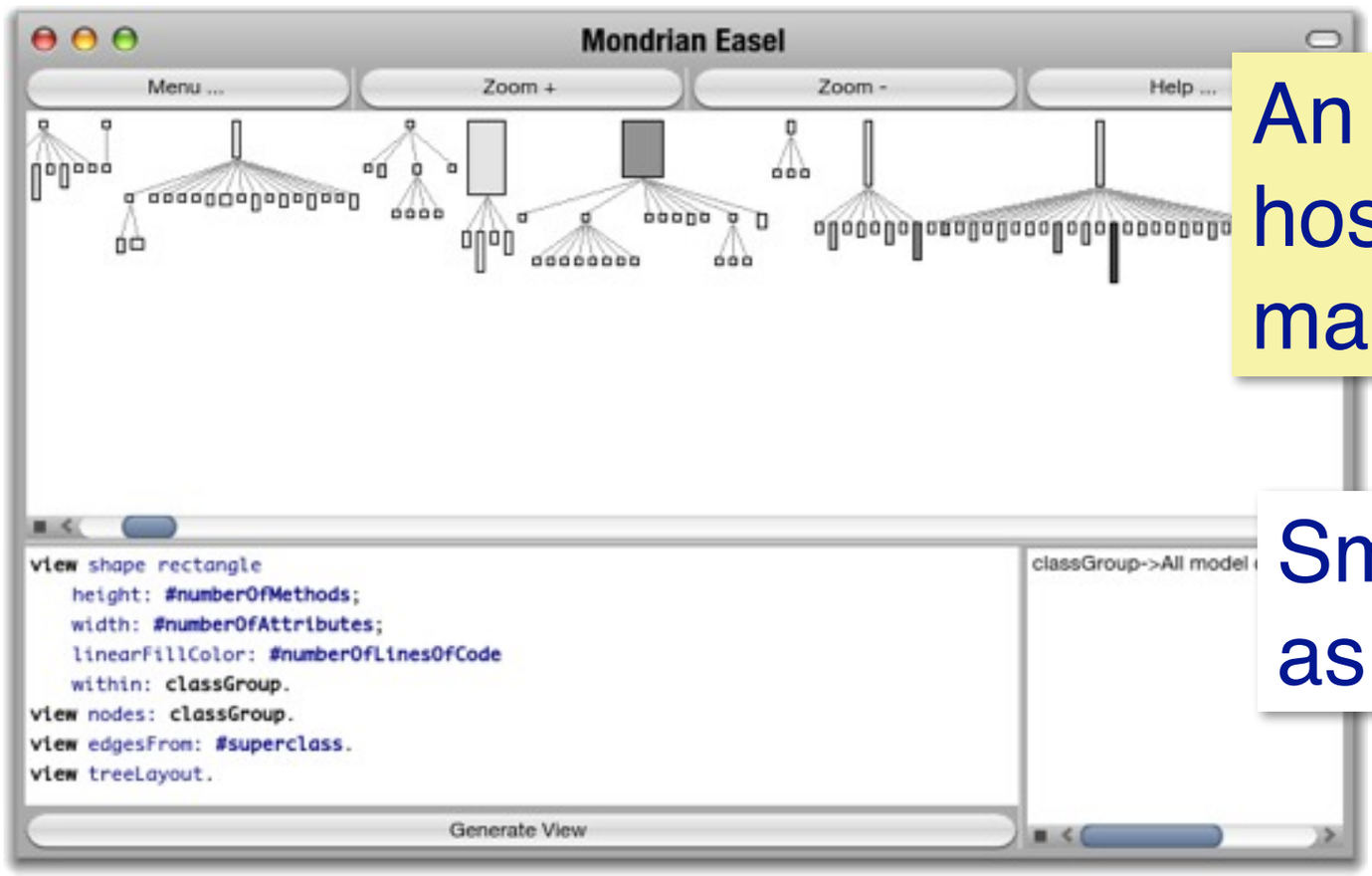
Michael Meyer et al. *Mondrian: An Agile Visualization Framework*. SoftVis 2006.

Mondrian is a tool for scripting visualizations of software models.
In fact it is a component framework, and scripts are simply Smalltalk code using the framework API, but have the flavor of a DSL. This is an “internal” or “embedded” DSL.

“Fluent interfaces”

```
mail()
   .from("build@example.com")
   .to("example@example.com")
   .subject("build notification")
   .message("some details about build status")
   .send();
```

Java used as a DSL



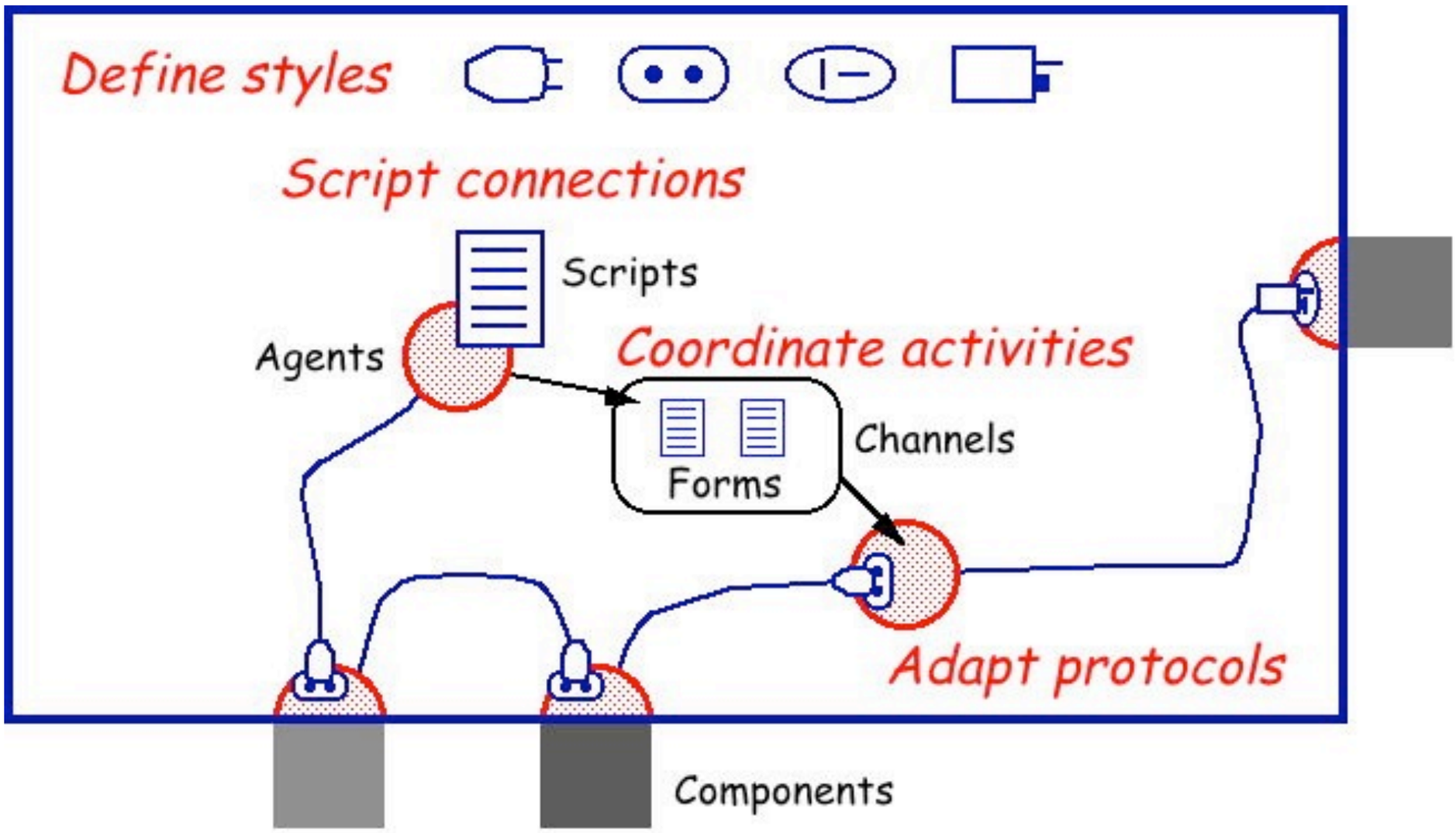
An internal DSL leverages host language syntax to make an API look like a DSL.

Smalltalk used as a DSL

Martin Fowler. *Domain-Specific Languages*, Addison-Wesley, 2010.

The API is designed as a “fluent interface” so code that uses it resembles a DSL.
The second example shows how this can be done in Java.

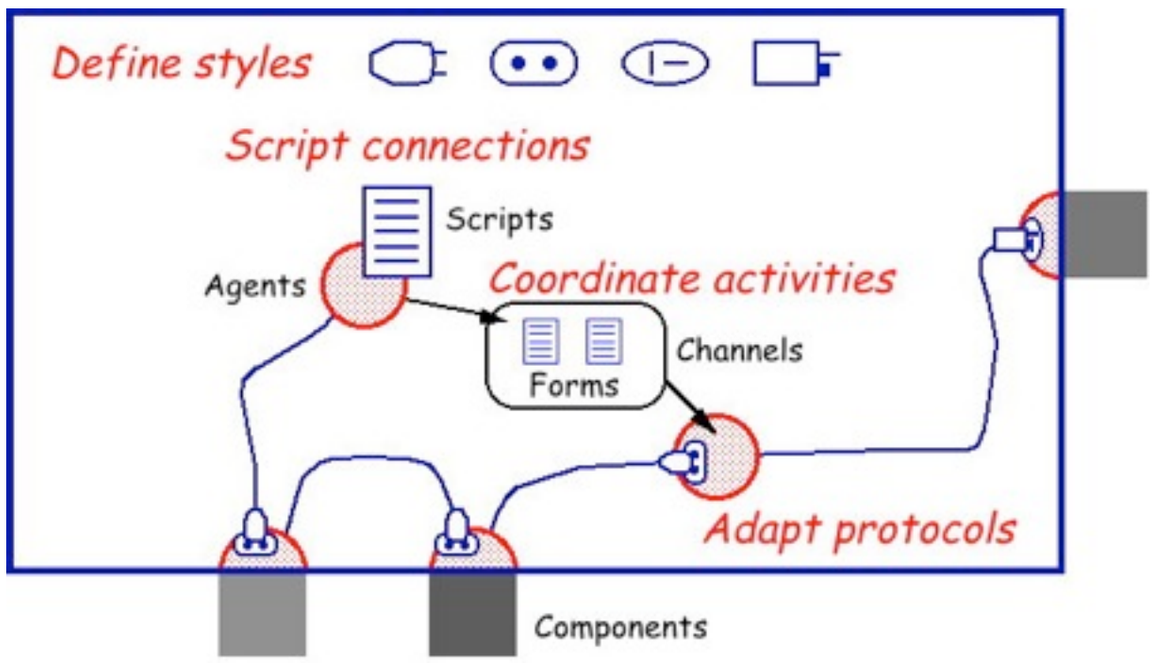
Piccola



Piccola is a minimal language for defining plugs, connectors and scripts

Piccola was designed as a “pure composition language” for defining “styles” (fluent interfaces), and scripts using those styles. The core paradigm is of communicating agents. Forms are first-class environments to control the scope of scripts.

Piccola

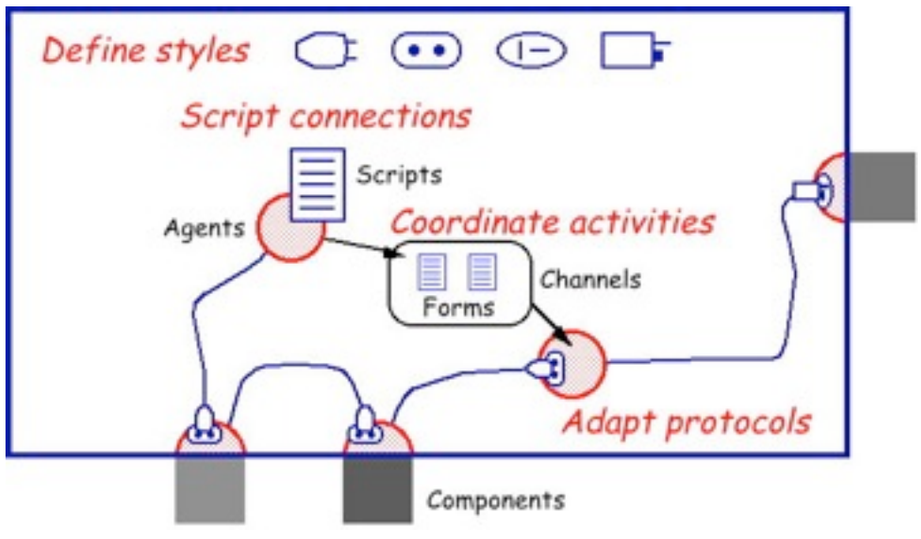


$$
\begin{array}{rcll}
A, B, C & ::= & \epsilon & \textit{empty form} \\
 & | & x\mapsto & \textit{bind} \\
 & | & x & \textit{variable} \\
 & | & A; B & \textit{sandbox} \\
 & | & hide_x & \textit{hide} \\
 & | & A \cdot B & \textit{extension} \\
 & | & \lambda x.A & \textit{abstraction} \\
 & | & AB & \textit{application} \\
 & | & \mathbf{R} & \textit{current root} \\
 & | & \mathbf{L} & \textit{inspect} \\
 & | & A \mid B & \textit{parallel} \\
 & | & \nu c.A & \textit{restriction} \\
 & | & c? & \textit{input} \\
 & | & c & \textit{output}
\end{array}
$$

built on a process calculus with explicit environments

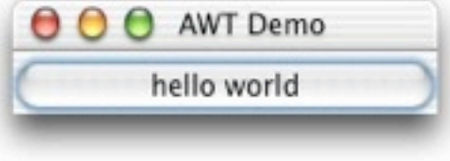
Franz Achermann and Oscar Nierstrasz. *A Calculus for Reasoning about Software Components*. Theoretical Computer Science 331(2), 2005.

Piccola is an extension of the pi calculus with first class environments.

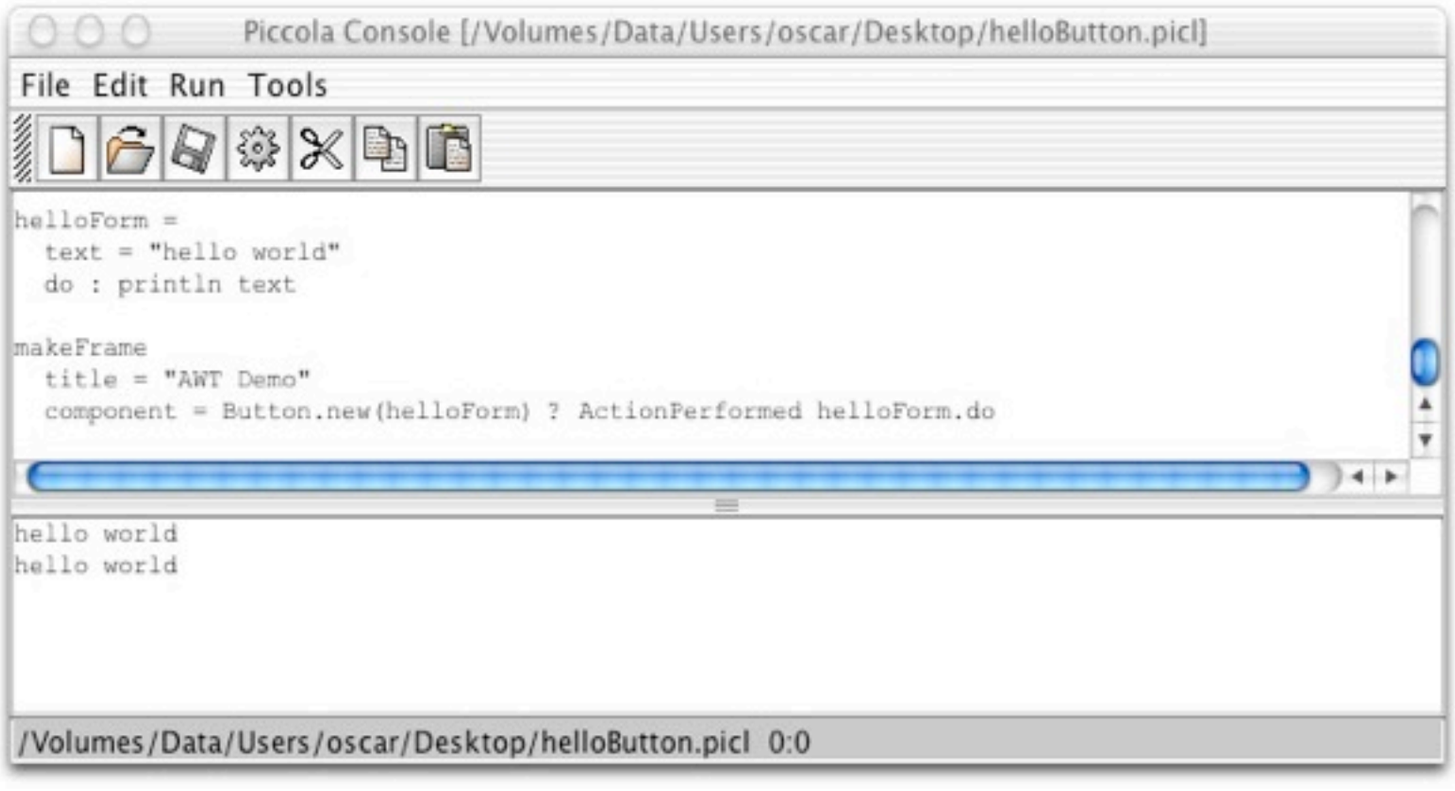


Piccola

$$
\begin{array}{rcll}
A, B, C & ::= & \epsilon & \textit{empty form} \\
 & | & x\mapsto & \textit{bind} \\
 & | & x & \textit{variable} \\
 & | & A; B & \textit{sandbox} \\
 & | & hide_x & \textit{hide} \\
 & | & A \cdot B & \textit{extension} \\
 & | & \lambda x.A & \textit{abstraction} \\
 & | & AB & \textit{application} \\
 & | & \mathbf{R} & \textit{current root} \\
 & | & \mathbf{L} & \textit{inspect} \\
 & | & A \mid B & \textit{parallel} \\
 & | & \nu c.A & \textit{restriction} \\
 & | & c? & \textit{input} \\
 & | & c & \textit{output}
\end{array}
$$



for scripting components written in Java

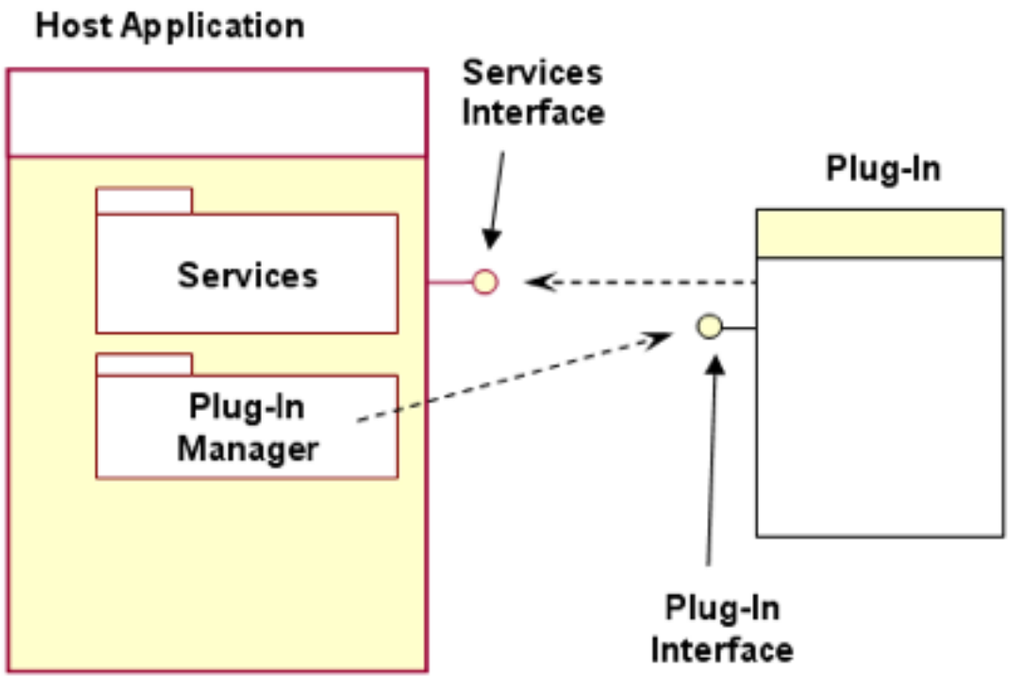


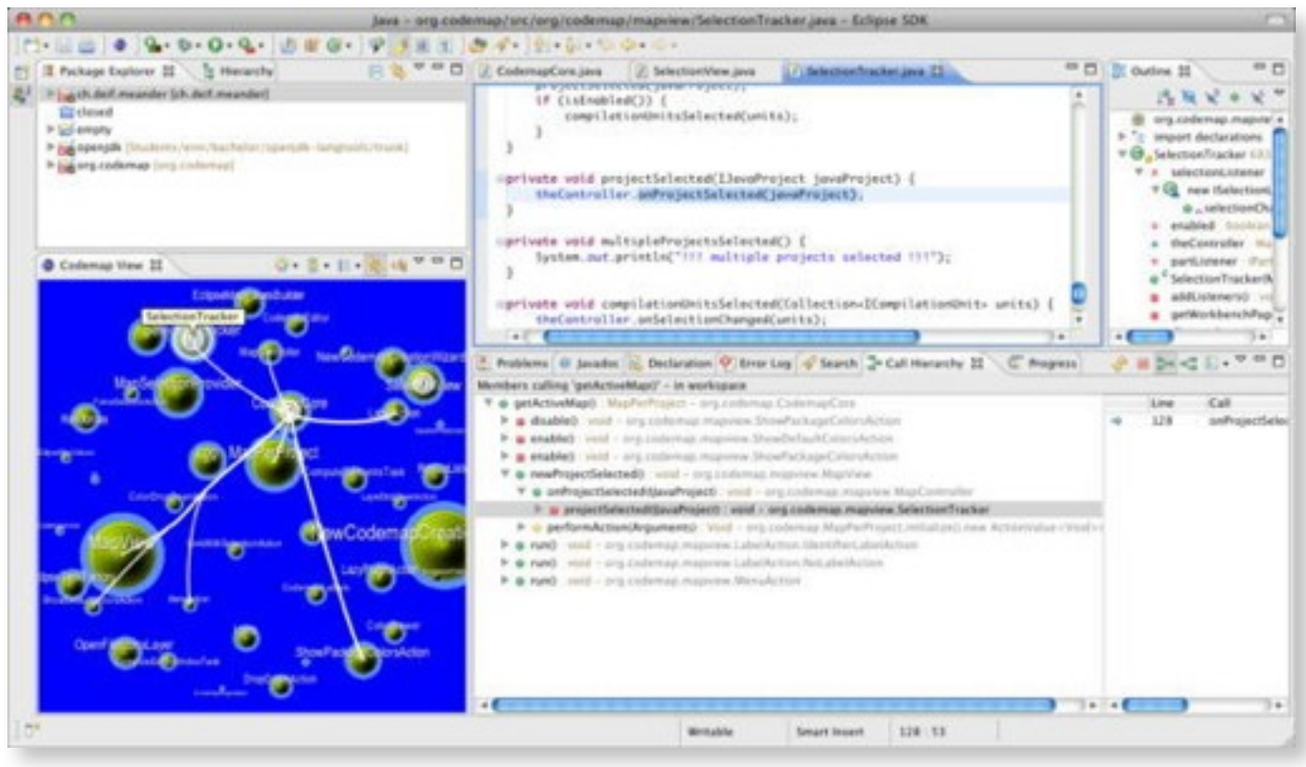
http://scg.unibe.ch/research/piccola

Piccola is implemented in Java, and can be used to script Java components (objects) adhering to a particular fluent interface.
In the “hello world” example, a button widget is bound to an action that prints “hello world”.

Plug-in Architectures

Plug-ins allow you to extend and (sometimes) configure the host application.

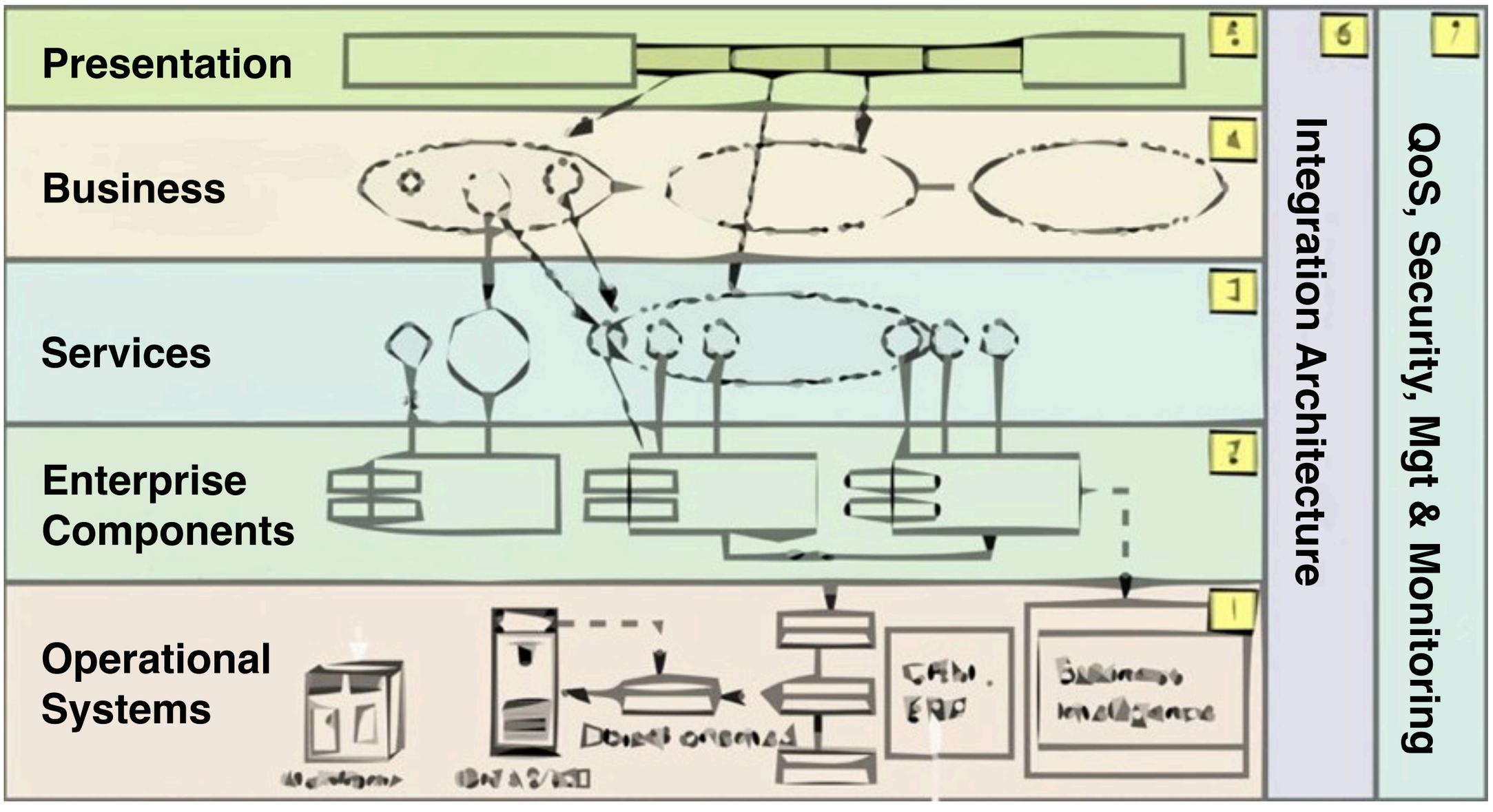




Best known examples: web browsers and IDEs.

Service-oriented architecture

SOA enables composition of distributed, heterogeneous web-based services.



SOA requires adherence to certain principles (like stateless services).
In many ways a throwback to libraries, but can be very effective.

Summary

Paradigm: configuration of interacting components

Motivation: manage variation

Roadmap

Early history

Objects

Components

Features

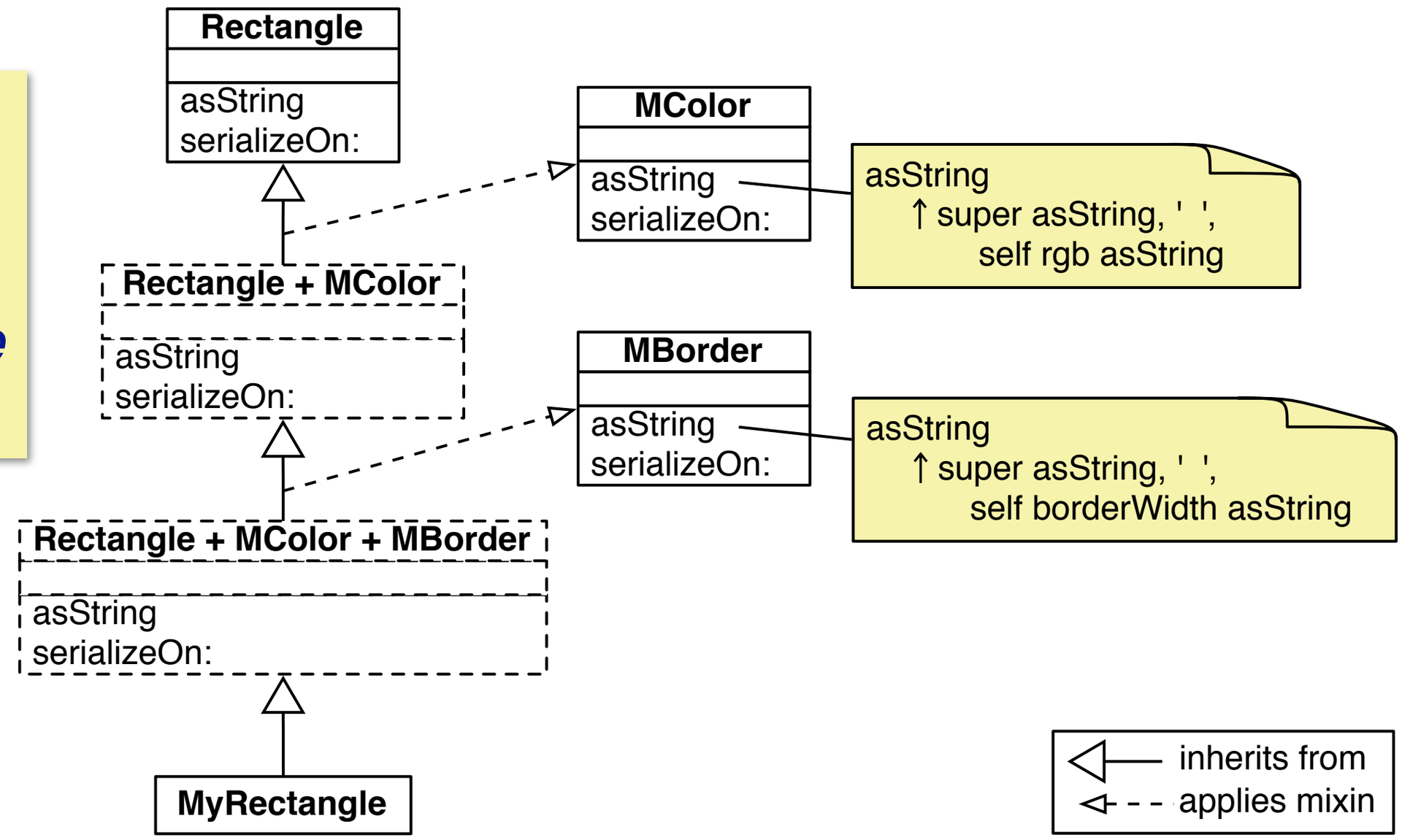
Metaprogramming

Conclusions

Mixins (1980)

Mixins are “abstract subclasses”

Mixins are sensitive to the order in which they are composed.



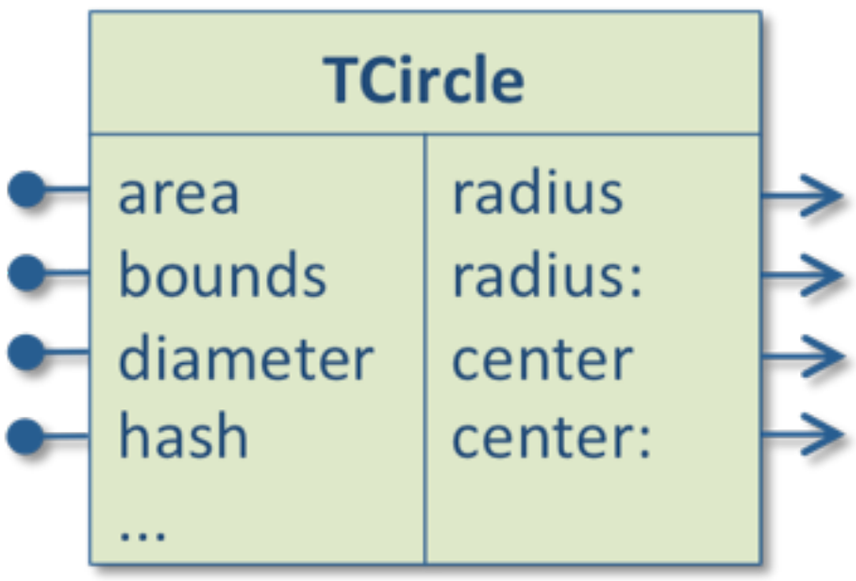
David A. Moon. *Object-Oriented Programming with Flavors*. OOPSLA 1986

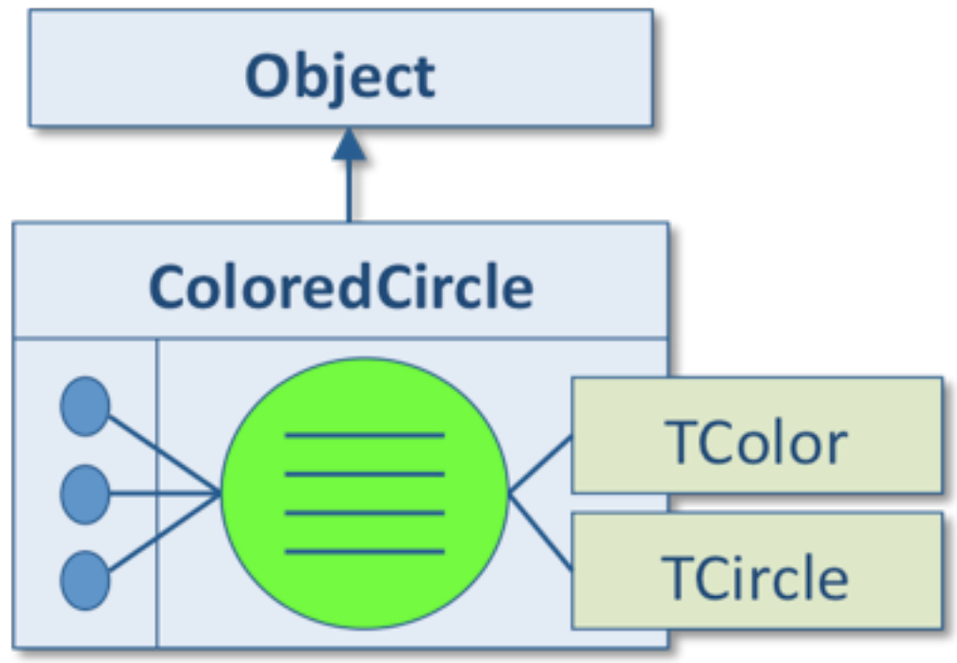
Mixins were introduced in “Flavors”, an early Lisp dialect.
A mixin is a fragment of a class that can be mixed in to an existing class to add new features.
Mixins avoid the need for multiple inheritance or code duplication in single-inheritance systems.
The key drawback is that systems that make heavy use of mixins cannot easily be modified since a change to a mixin can percolate to many classes.

Traits

Class = superclass + state + traits + glue

Traits provide and require methods





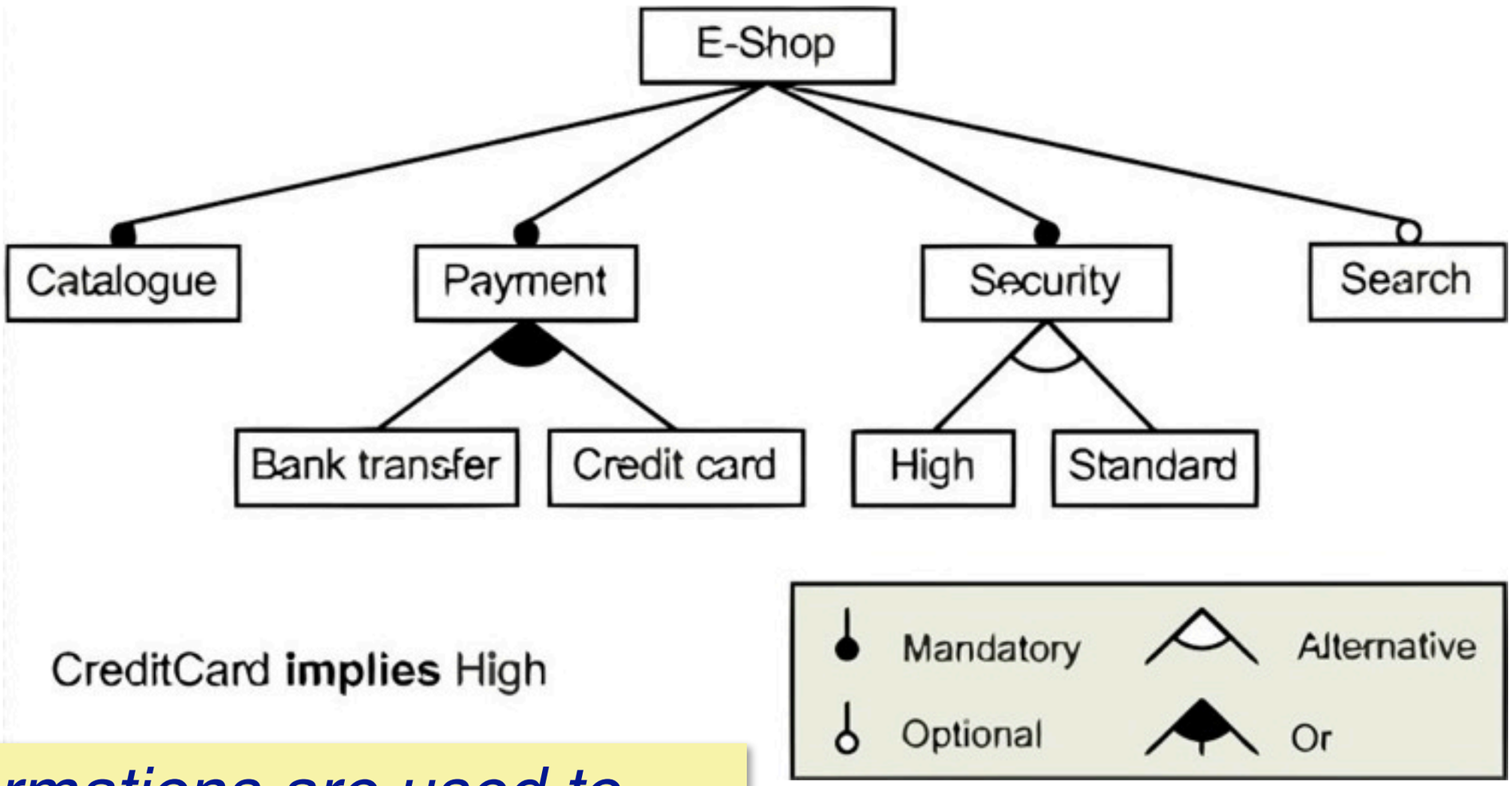
The composing class retains control

Stéphane Ducasse, et al. *Traits: A Mechanism for fine-grained Reuse*. ACM TOPLAS, 2006

Unlike mixins, traits are insensitive to the order of composition.
Glue takes the form of explicit aliasing and exclusion of features.
Traits are purely static and can be flattened away.
“Talents” are dynamic traits for objects.

Feature-Oriented Programming

FOP is a SPL paradigm for synthesizing programs from features



Transformations are used to add features to base programs.

Don Batory and Sean O'Malley. *The Design and Implementation of Hierarchical Software Systems With Reusable Components*. ACM TOSEM, 1992

The diagram is a “feature model”, or “feature diagram”.

Generative Programming

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0>
{
    enum { value = 1 };
};

// Factorial<4>::value == 24
// Factorial<0>::value == 1
void foo()
{
    int x = Factorial<4>::value; // == 24
    int y = Factorial<0>::value; // == 1
}
```

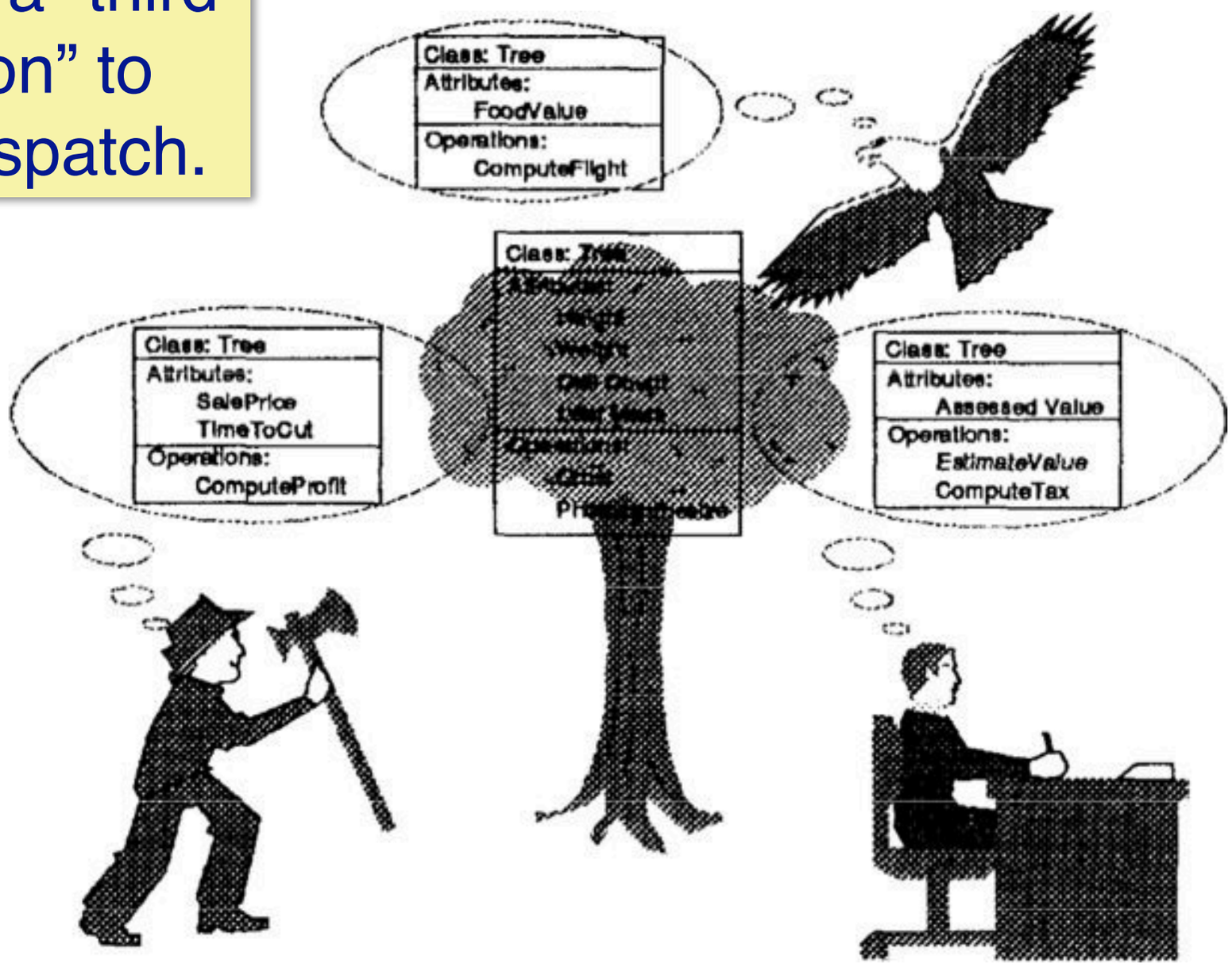
Template meta-programming is a form of compile-time code generation.

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*, ACM Press/Addison-Wesley Publishing, 2000.

Template meta-programming can be used to select features at compile-time.
C++ templates are actually Turing-complete.
In this example, a factorial is computed at compile-time by composing C++ templates.

Subject-Oriented Programming

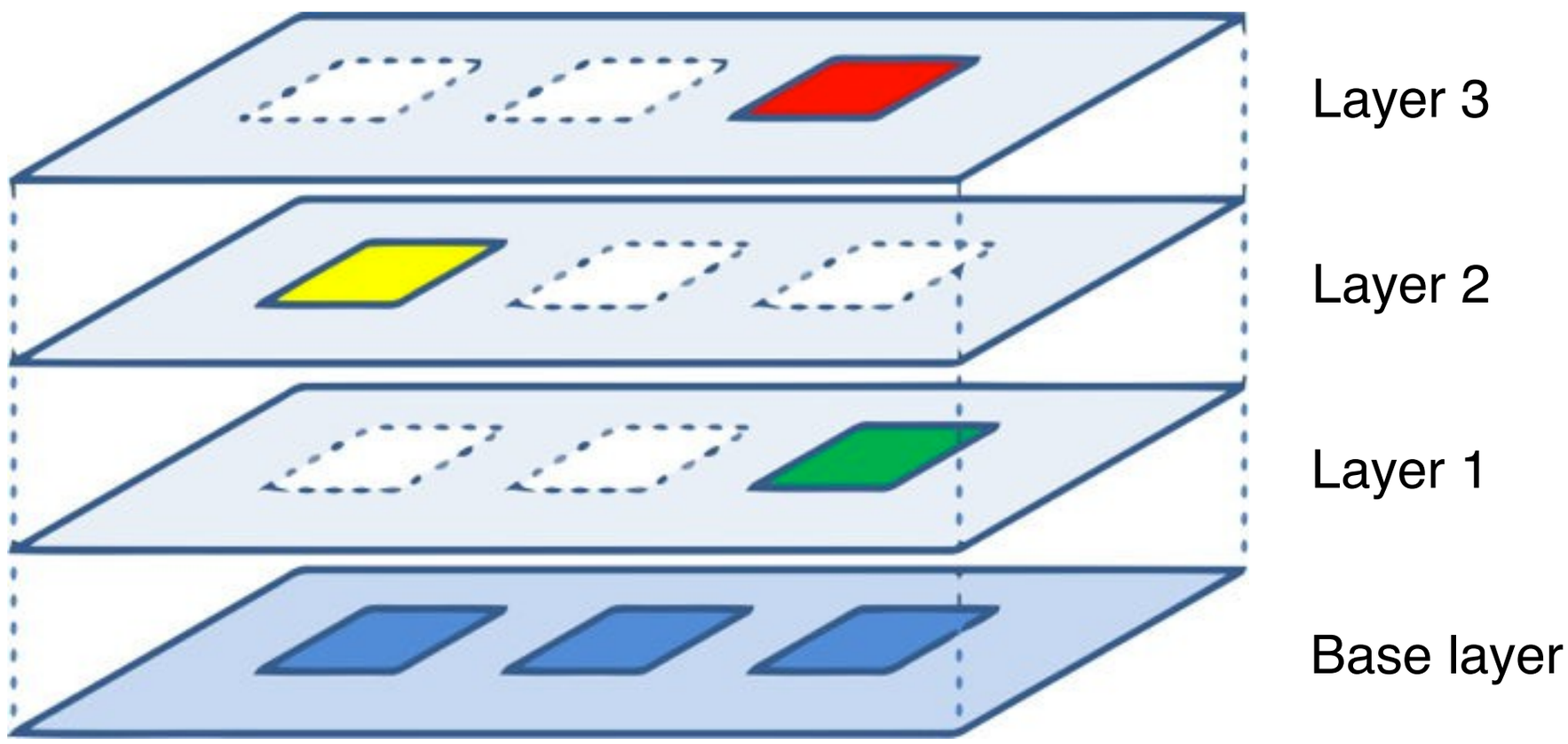
SOP adds a “third dimension” to method dispatch.



William Harrison and Harold Ossher. *Subject-Oriented Programming (A Critique of Pure Objects)*. OOPSLA 1993

Procedural invocation is single dispatch;
OOP is doubly-dispatched, since it takes the receiver of the message into account;
SOP is triply dispatched, by taking the sender into account.

Context-Oriented Programming



COP offers multi-dimensional dispatch, with multiple "layers" triggered by contextual information.

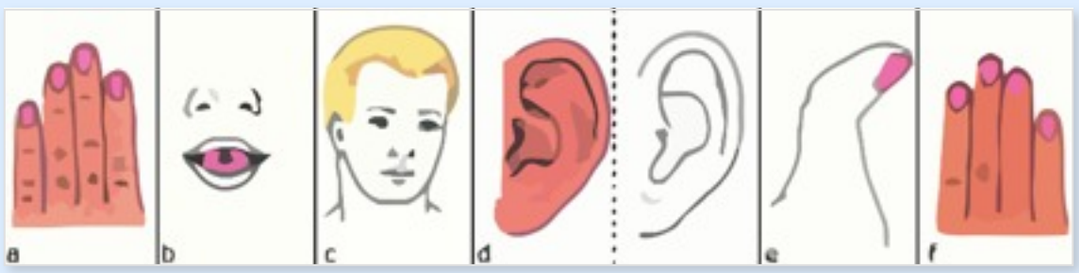
Robert Hirschfeld, et al. *Context-Oriented Programming*. Journal of Object Technology, March 2008. http://dx.doi.org/10.5381/jot.2008.7.3.a4

Each layer may define a number of class extensions (additions, modifications).

Summary

Paradigm: model and compose elementary features

Motivation: features represent domain concepts



Roadmap

Early history

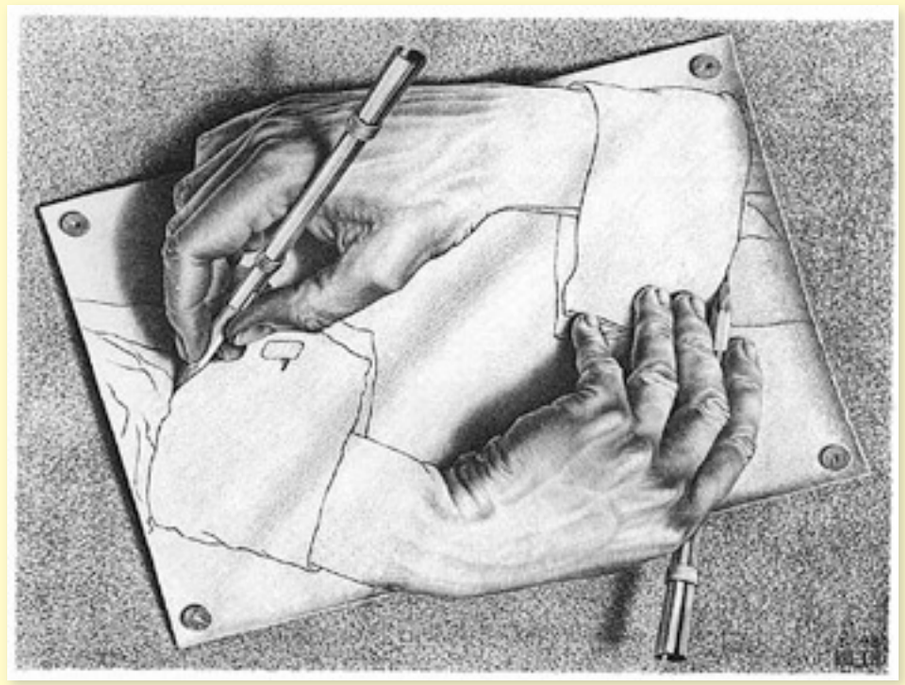
Objects

Components

Features

Metaprogramming

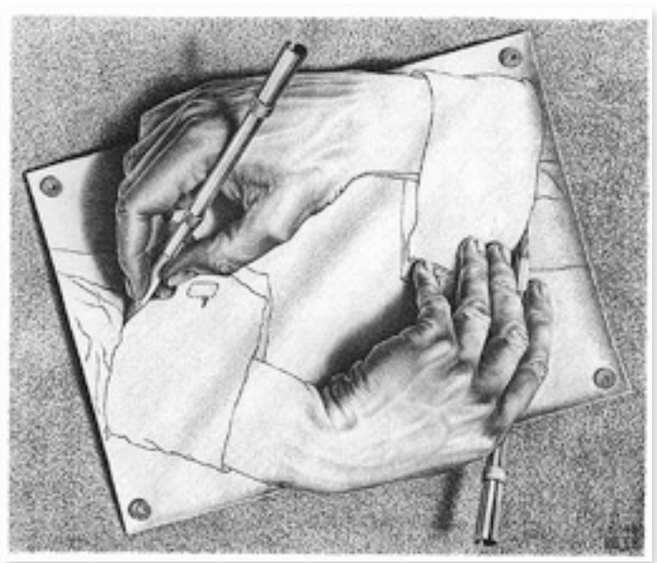
Conclusions



Reflection

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

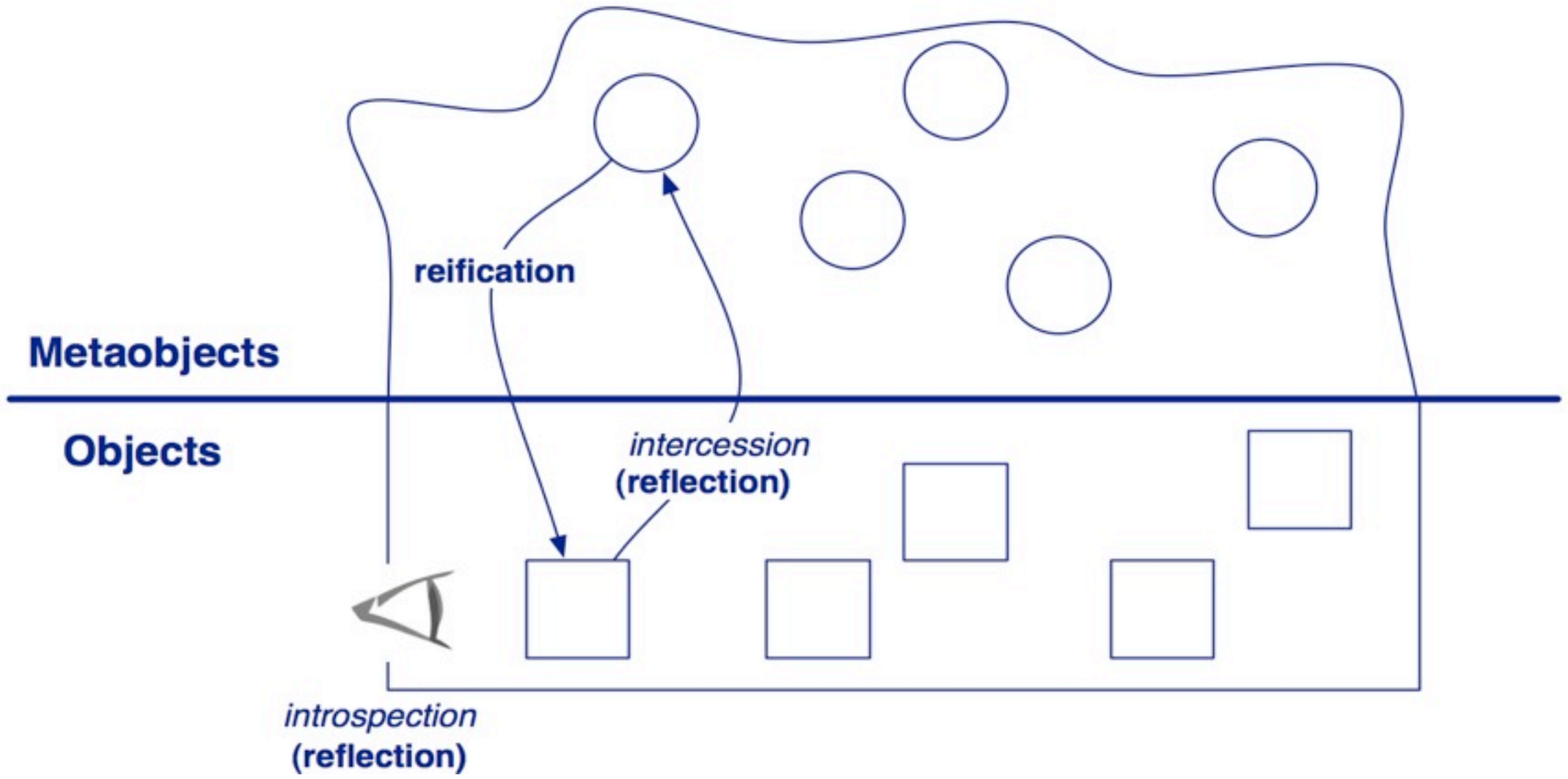
Daniel G. Bobrow, et al. *CLOS in Context — The Shape of the Design*. In “Object-Oriented Programming: the CLOS perspective”, MIT Press, 1993.



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

Reflection and reification

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

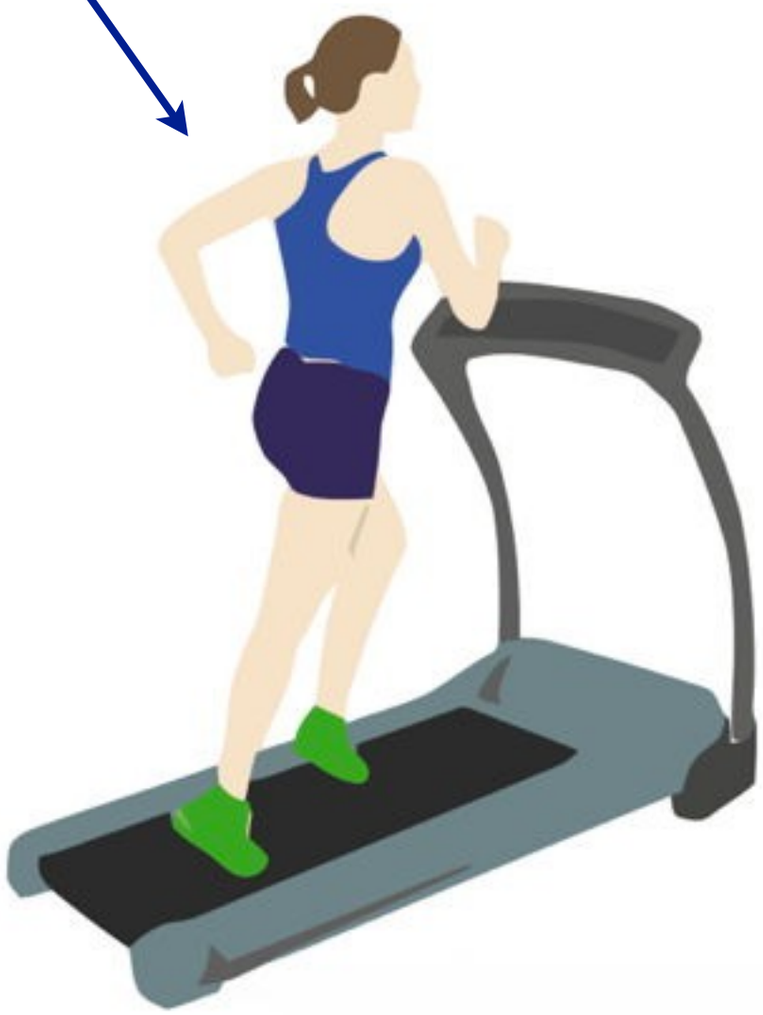


Introspection is the ability for a program to *observe* and therefore reason about its own state.

"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.
NB: Java "reflection" is actually just intercession.

Structural and behavioral reflection

Structural reflection lets you reflect on the *program* being executed



Behavioral reflection lets you reflect on the language *semantics* and *implementation*

Malenfant et al., *A Tutorial on Behavioral Reflection and its Implementation*, 1996

Behavioural reflection is especially interesting for realizing language extensions.

Applications of metaprogramming

IDE tools

— debugging
— profiling
— refactoring

Dynamic applications

— UI generation
— Modeling tools

Mostly compiler and development tools can benefit from metaprogramming, but some highly dynamic applications make use of it too.

Three approaches to reflection

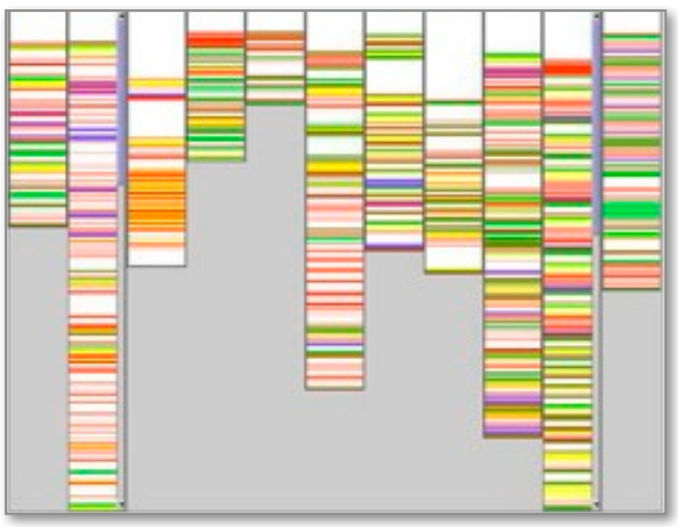
The tower of meta-circular interpreters

Reflective languages

Open Implementation (MOP)

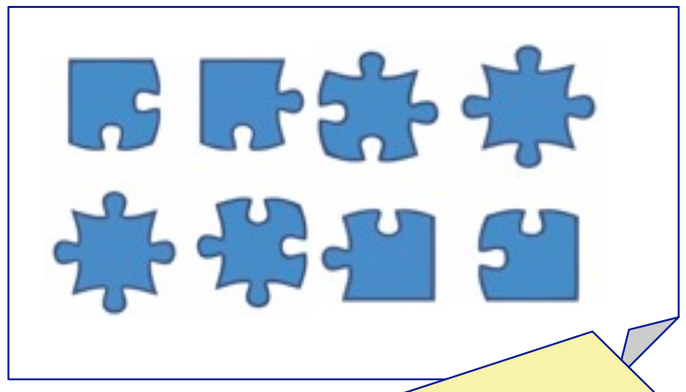
1. Towers of interpreters are reified on need in practice
2. Reflective languages like Smalltalk are often written in themselves
3. Open implementations like CLOS offer an API (MOP) to the implementation

Aspect-Oriented Programming



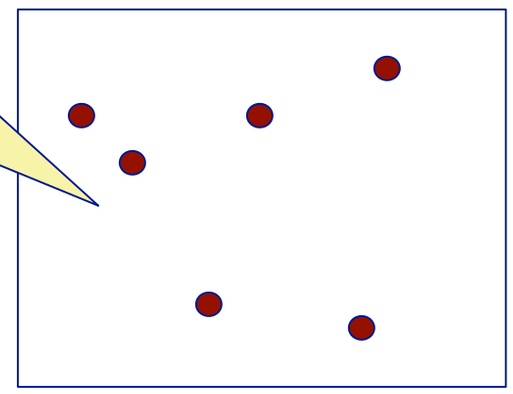
AOP improves modularity by supporting the separation of cross-cutting concerns.

An *aspect* packages cross-cutting concerns



A *pointcut* specifies a set of *join points* in the target system to be affected

"Weaving" is the process of applying the aspect to the target system



Gregor Kiczales, et al. *Aspect-Oriented Programming*. ECOOP'97

factor out cross-cutting concerns
pointcuts apply aspects to joinpoints in code
joinpoints may be static or dynamic

AspectJ

```
public class Demo {
 static Demo d;
 public static void main(String[] args){
  new Demo().go();
 }
 void go(){
  d = new Demo();
  d.foo(1,d);
  System.out.println(d.bar(new Integer(3)));
 }
 void foo(int i
  System.out.pri
 }
 String bar (In
  System.out.pri
  return "Demo.
 }
}
```

```
aspect GetInfo {
 pointcut goCut(): cflow(this(Demo) && execution(void go()));
 pointcut demoExecs(): within(
 Object around(): demoExecs() &
 {
  println("Intercepted message:
  thisJoinPointStaticPart.getSi
  ...
 } ...
}
```

```
Demo.foo(1, tjp.Demo@939b78e)
Demo.bar(3)
Demo.bar(3)
```

Intercept execution within control flow of `Demo.go()`

Identify all methods within `Demo`

Wrap all methods except `Demo.go()`

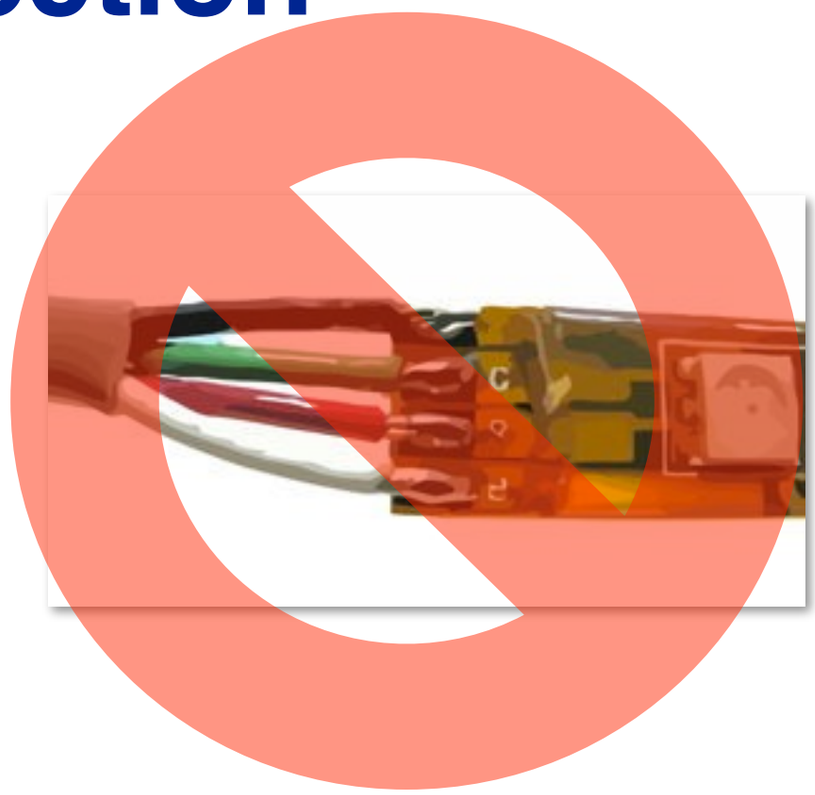
```
Intercepted message: foo
in class: tjp.Demo
Arguments:
  0. i : int = 1
  1. o : java.lang.Object = tjp.Demo@c0b76fa
Running original method:

Demo.foo(1, tjp.Demo@c0b76fa)
  result: null
Intercepted message: bar
in class: tjp.Demo
Arguments:
  0. j : java.lang.Integer = 3
Running original method:

Demo.bar(3)
  result: Demo.bar(3)
Demo.bar(3)
```

http://www.eclipse.org/aspectj/downloads.php

This (rather lame) foobar example shows how pointcuts can be used to specify not only static locations in the code, but also dynamic locations (ie within the execution of Demo.go()).

Dependency injection

Dependency injection externalizes dependencies to be configurable



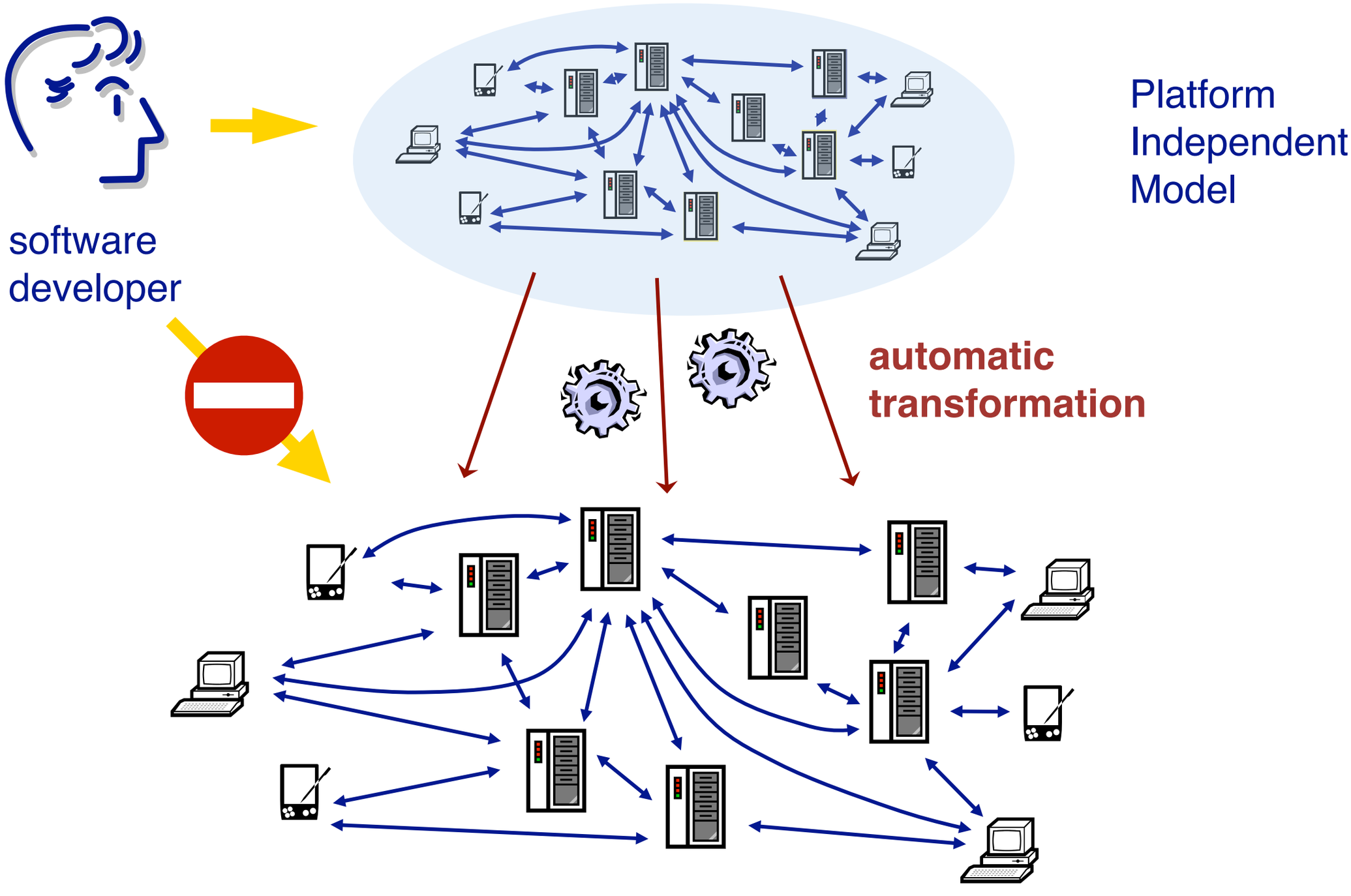
Typical application: injecting mocks

Key techniques:
— parameterization
— code generation
— reflection

Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. http://martinfowler.com/articles/injection.html

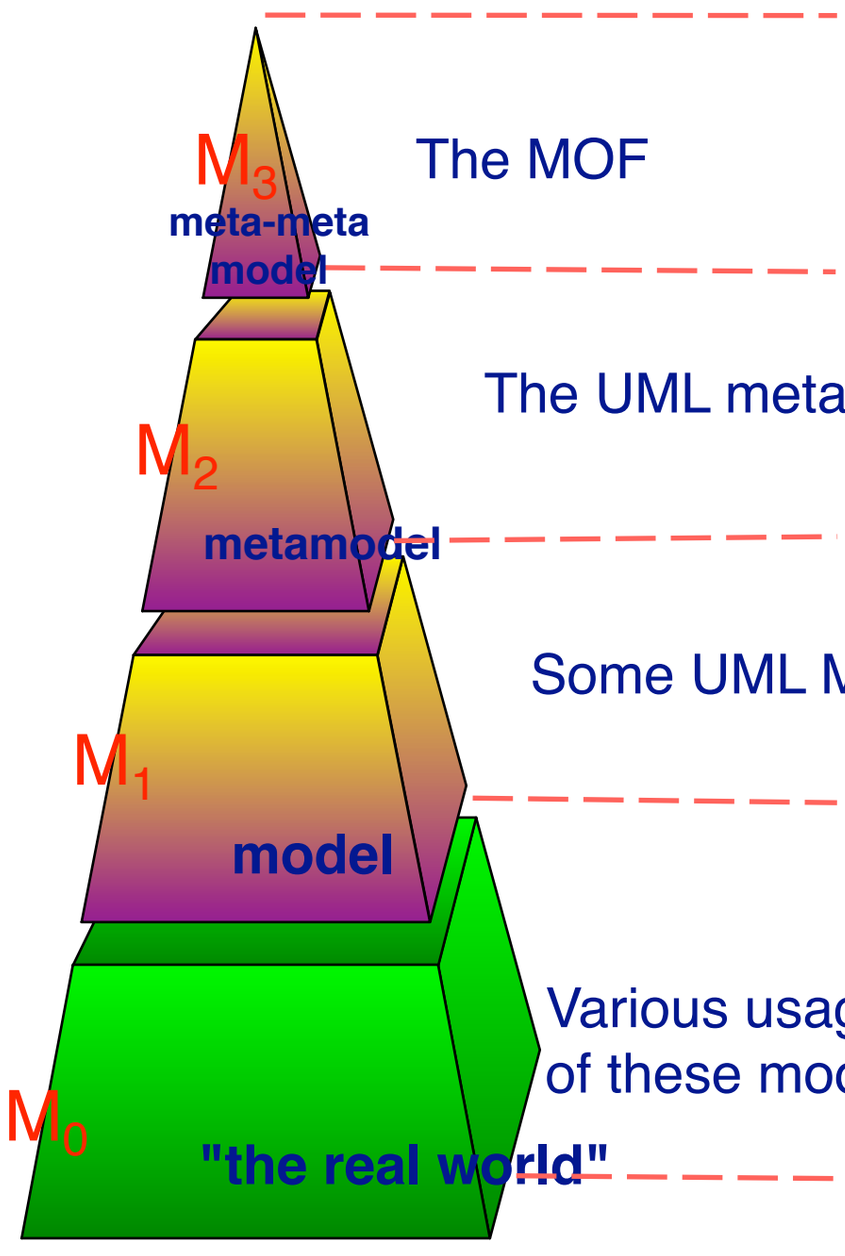
Dependency injection refers to techniques to make internal dependencies externally configurable. A good example is to inject mock objects for testing purposes.

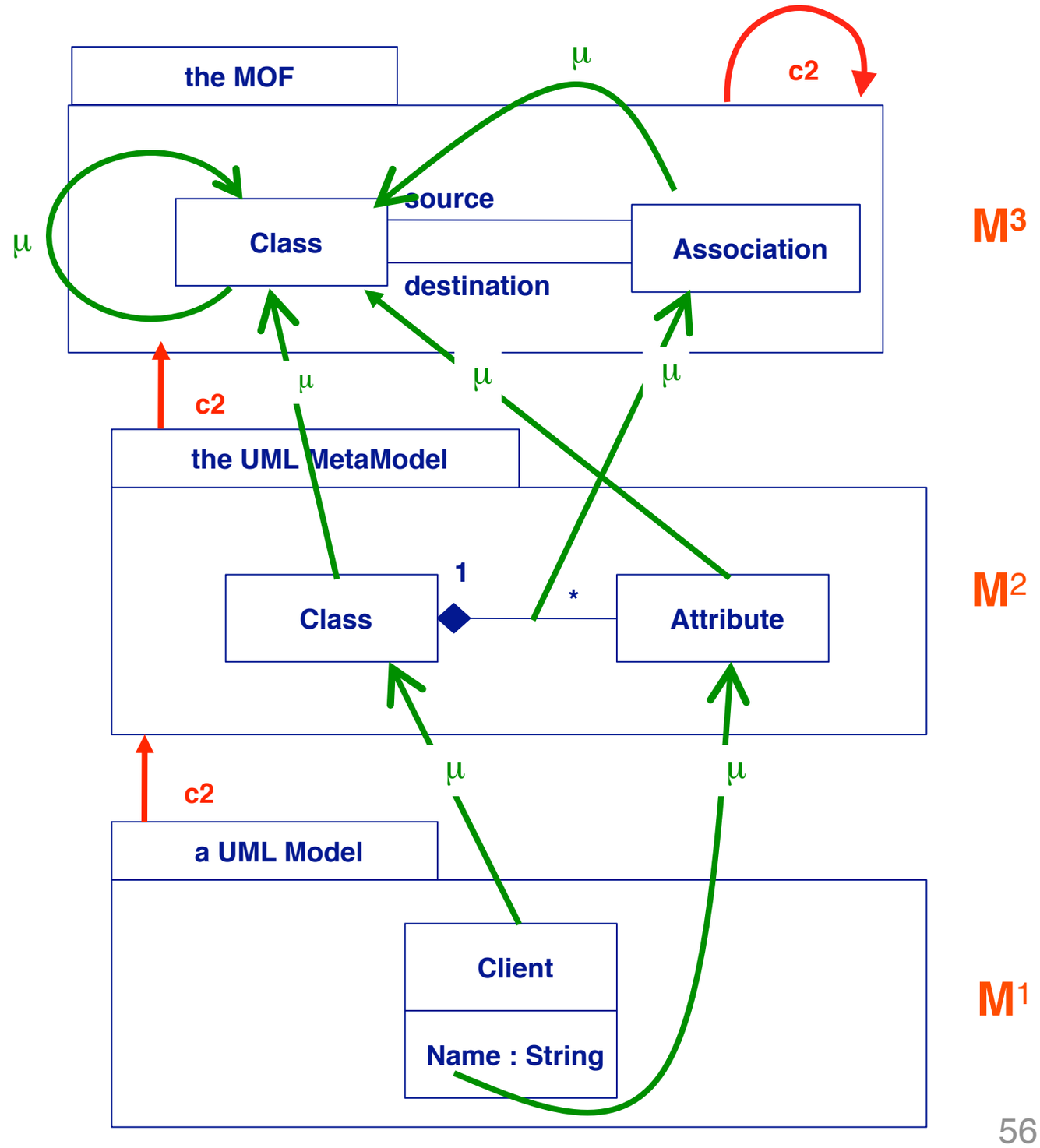
Model-Driven Engineering



MDE makes sense especially when you have the same application running on many platforms.
Slide courtesy Colin Atkinson, Universität Mannheim

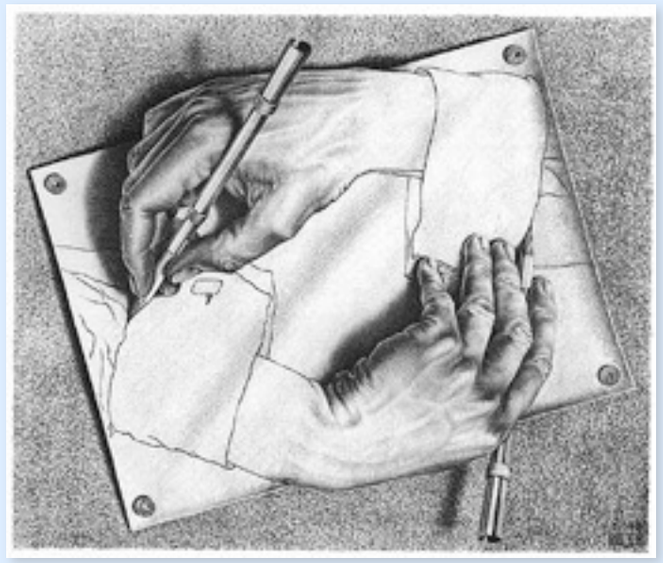
The OMG / MDA Stack





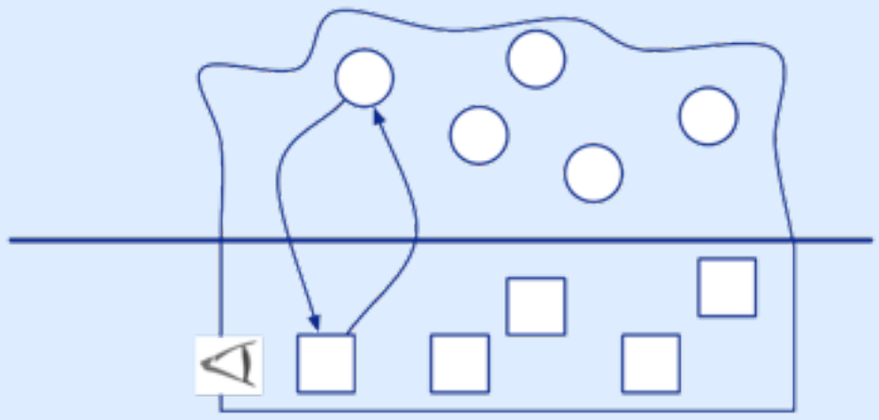
Slide courtesy Jean Bézivin, INRIA/U Nantes

Summary



Paradigm: composition as metaprogramming

Motivation: separation of base and meta-levels



Roadmap

Early history

Objects

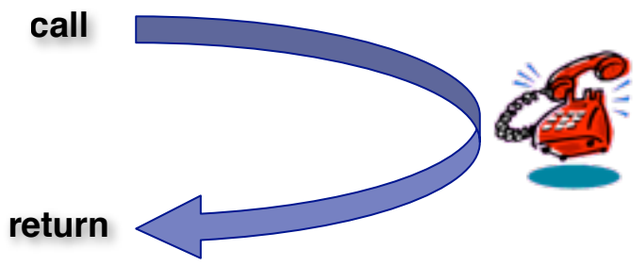
Components

Features

Metaprogramming

Conclusions

Mechanisms



invocation

messages

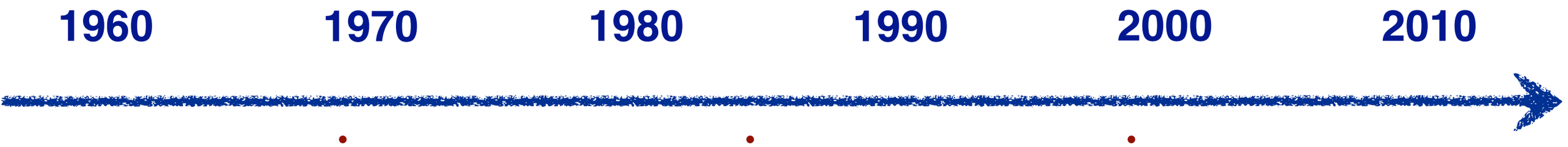
binding

code generation

Dimensions

The core composition paradigm is component-based, but it varies along several dimensions.

Conclusion: Trends



procedural technology	object technology	component technology	model technology
Procedures, Pascal, C, ...	Objects, Classes, Smalltalk, C++, ...	Components, Frameworks, Patterns, ...	Models, Transformations, UML, MOF, QVT ...
procedural refinement	object composition	component composition	model transformation