33. Unifying Refactorings and Compositions as Software Operators

Software Operators in Code Algebras and Composition Systems as a Basis for a Unified View on Software Engineering

Prof. Dr. Uwe Aßmann
TU Dresden
Lehrstuhl Softwaretechnologie
12-1.0, 1/26/13

Obligatory Literature

- Class algebra:
- Gilad Bracha, William Cook. Mixin-based inhertiance. OOPSLA 1990. citeseer.nj.nec.com/bracha90mixinbased.html
- James O. Coplien, Liping Zhao. Symmetry Breaking in Software Patterns. Springer Lecture Notes in Computer Science, LNCS 2177, October 2001, ff. 37. http://users.rcn.com/jcoplien/Patterns/Symmetry/Springer/SpringerSymmetry.html

Objectives

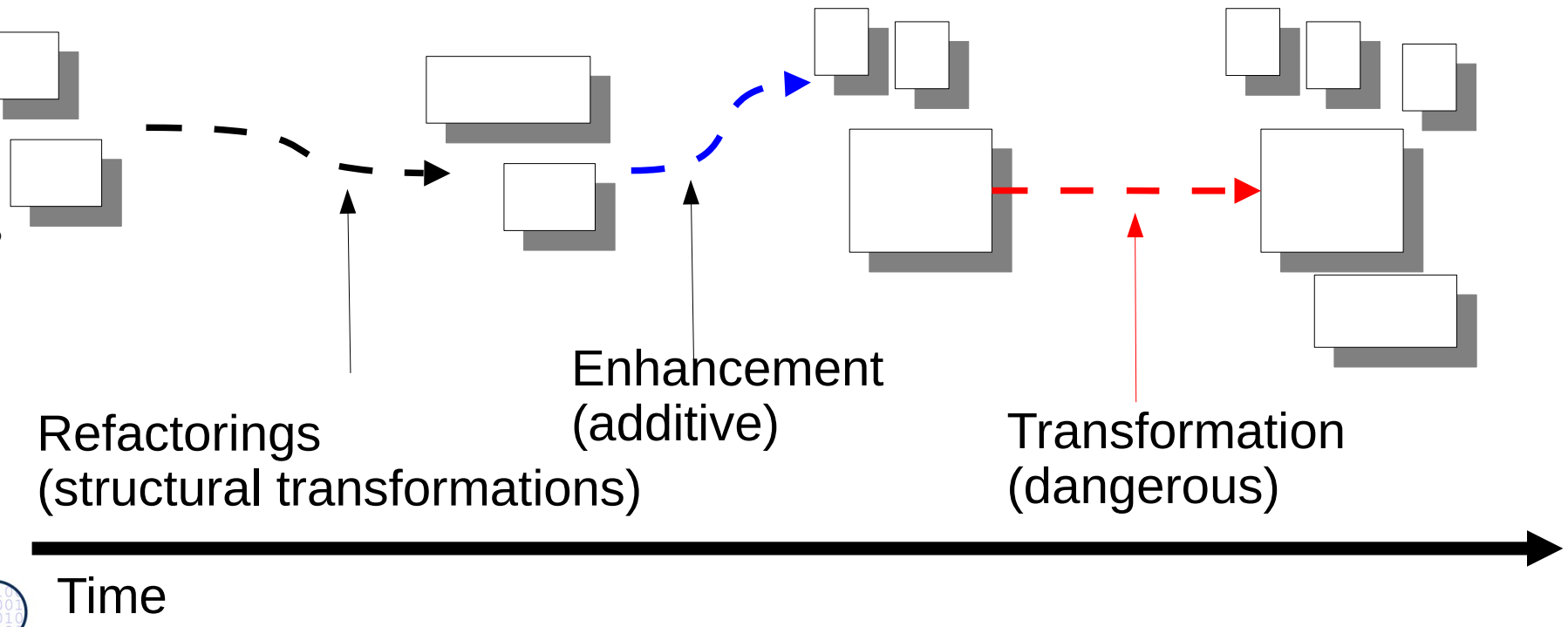
- There are, beyond class and role models, other composition systems
- Model algebras, class algebras, code algebras and composition systems are different
- The algebraic features of the composition operators make the difference
- Refactorings are symmetries, algebraic code operators retaining invariants

33.1 From Refactoring to Software Composition

Refactorings are Harmless Evolution Operations

- To arrive at a design pattern in the code, one has to refactor
- Idea: split of operations into ***harmless, enhancing (additive)***, and ***dangerous*** ones.

Evolution = Refactorings + Enhancements + Transformations



Harmless Operators

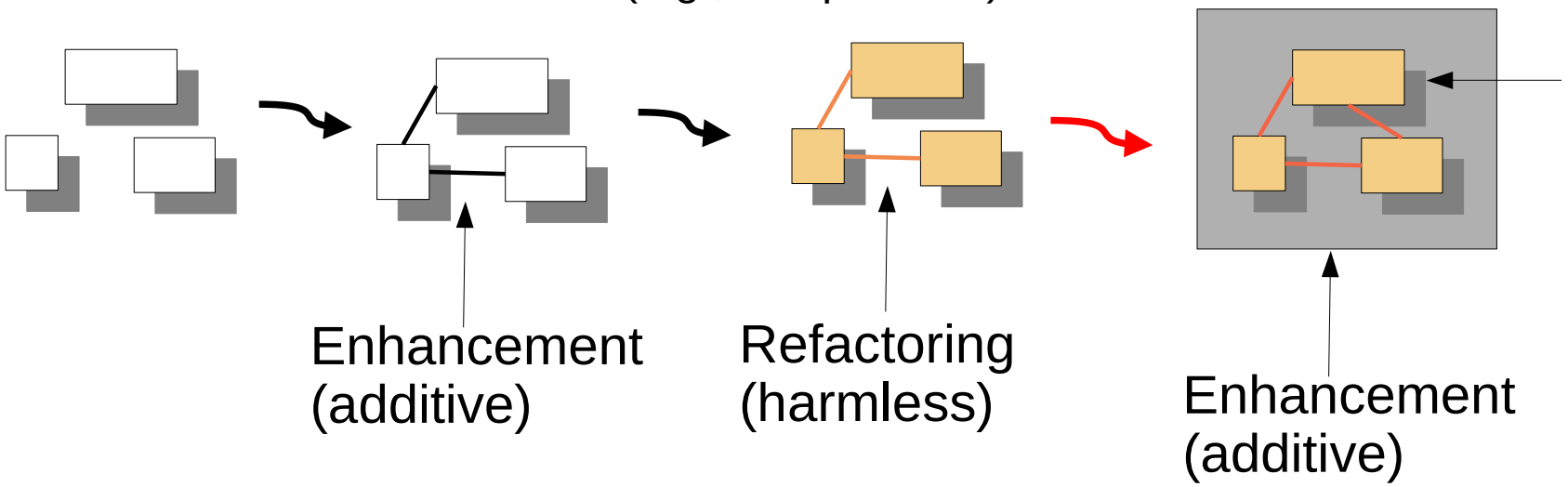
- **Harmless operators** do preserve the semantics of the program
  - **Lowerings** lower an expressive language construct to less expressive ones. Lowerings prepare optimizations on lower level
    - Transform inheritance to flat records
    - Transform recursion to loops
    - Unroll loops
  - **Refactorings** change the structure of the program
  - **Higherings** recognize a more expressive language construct from a set of less expressive one
    - Higherings are used in reengineering
    - Recognize a loop or recursions from gotos
    - Recognize a vector operation from a loop (vectorizer)
  - **Optimizers** replace program elements with more efficient ones
    - Peephole optimization
    - Strength reduction

Enhancement Operators

- There are other software operators in modern software engineering approaches
- **Enhancement operators** augment the semantics of a program with new features (see CBSE)
  - **Composition operators** compose components
    - **Connectors** connect components at ports (architecture languages)
  - **Inheritance** compose superclasses with mixins
    - [Braha&Cook 90 OOPSLA]
  - **Parameterizations** fill templates with values
    - Generic programming with BETA or C++ template metaprogramming
    - [GenVoca/Batory parameterization as composition]
  - **Role Model merge** composes roles into classes
- **Transformation operators (dangerous)**
  - Rewrite rule systems (graph rewrite rules, term rewrite rules)
  - Strategic rewriting (rewriting with higher order functions)

Enhancement in Software Build and Composition

- Enhancements also occur, when components are composed together to a syst (system build, system composition): linking, template expansion, connector composition, etc.
- Transformations also occur (e.g., compilations)



Can There Be A Uniform Operator-Based Software Technology?

- Scaling for all these approaches
- Supported by uniform tools
- Implemented in a library
- Embedded in the every-day software process (as refactorings)

Software Development as Operations of an Algebra

- Idea: the activities for build and evolution are represented as operators in a **model algebra** or **code algebra**
  - Implementation: library
- How do the elements of the algebra look:
  - Refactorings: change the abstract syntax graph (ASG) directly
  - Inheritance: Classes with feature list
  - Package merges: Packages with sets of classes
- Can there be a component model for all of them?
  - Solution: graybox components

33.2 Model and Code Algebras

Merging classes...

Model Algebra

- A **model algebra** contains a carrier set (models) and operations on these:
- union: Model x Model → Model
- merge: Model x Model → Model
- diff: Model x Model → Model
- join: Model x Model → Model
- patch: Model x Model → Model

Class Algebra

- A **class algebra** contains a carrier set (classes) and operations on these:
- union: Class x Class → Class
- merge: Class x Class → Class
- diff: Class x Class → Class
- join: Class x Class → Class
- patch: Class x Class → Class
- mixin: Class x Class → Class

Discussion

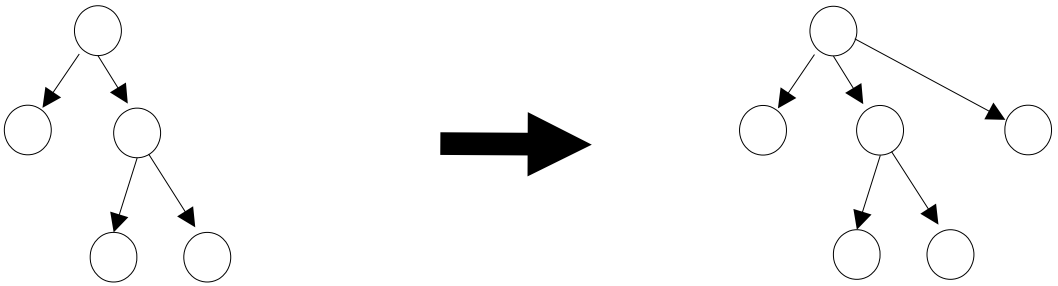
- Model and class algebrae have problems:
  - Coarse-grained composition: it is hard to adapt a class or a model during merge in a fine-grained way
  - From a merge, too many model element merges result
  - The larger the models, the more difficult it becomes

33.3 Software Operators Unify Refactorings and Composition Operators

Operations on Different Levels

- Refactoring works directly on the AST/ASG
- Attaching/removing/replacing fragments

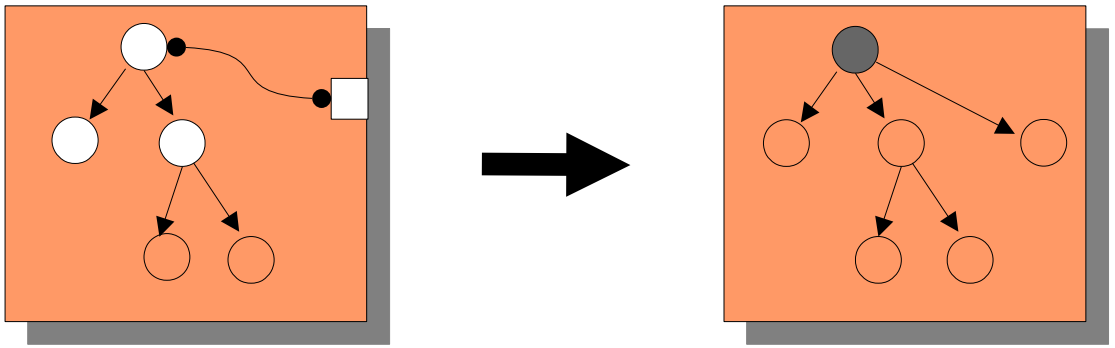
Refactorings
Transformations



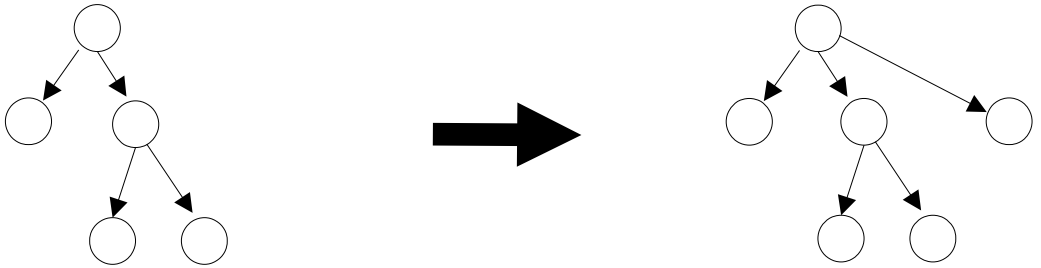
Operations on Different Levels

- Class composition, model composition, aspect weaving, view composition, GenVoca parameterization works on implicit hooks *(join points),* role model merge

Composition with implicit hooks



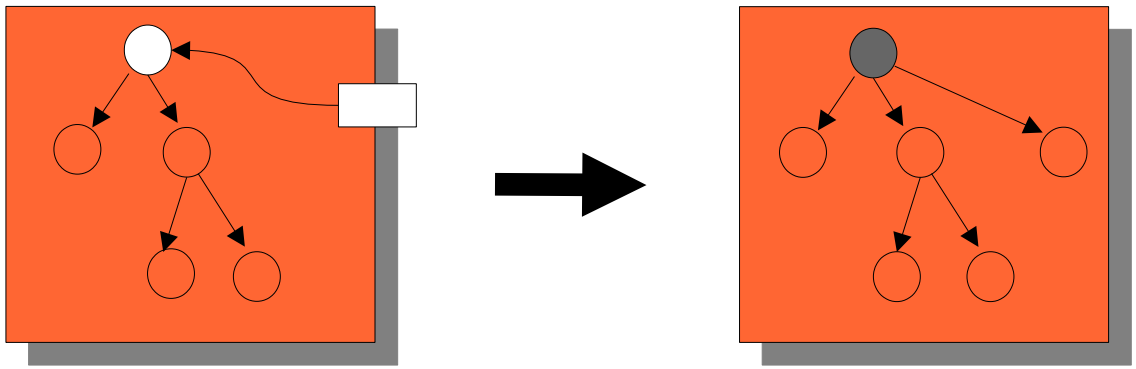
Refactorings Transformations



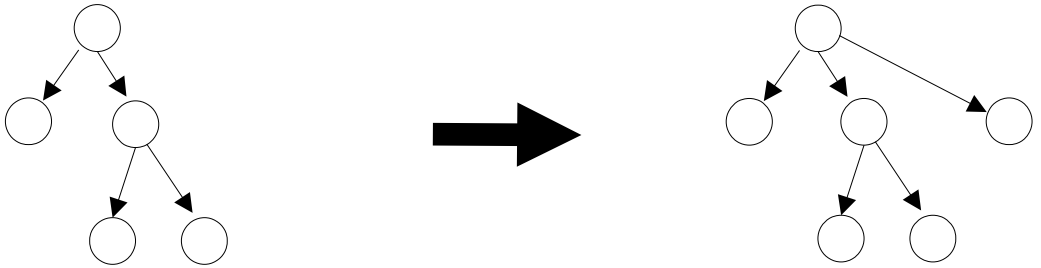
Operations on Different Levels

- Templates in generic programming, connectors work on declared *hooks*

Composition with declared hooks

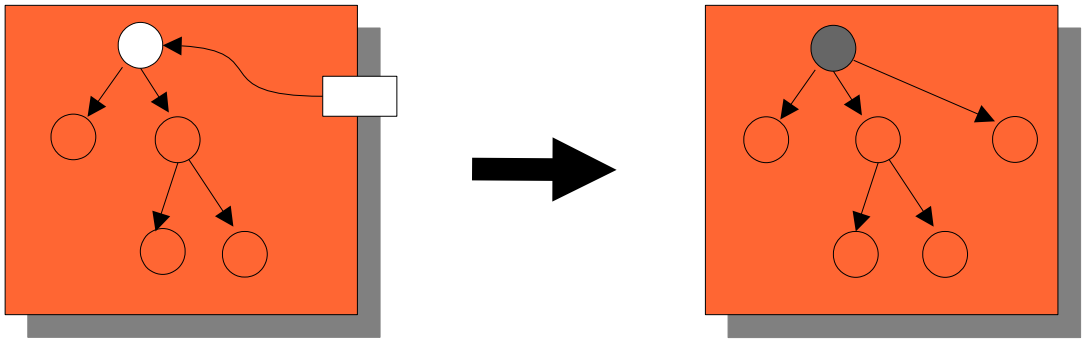


Refactorings Transformations

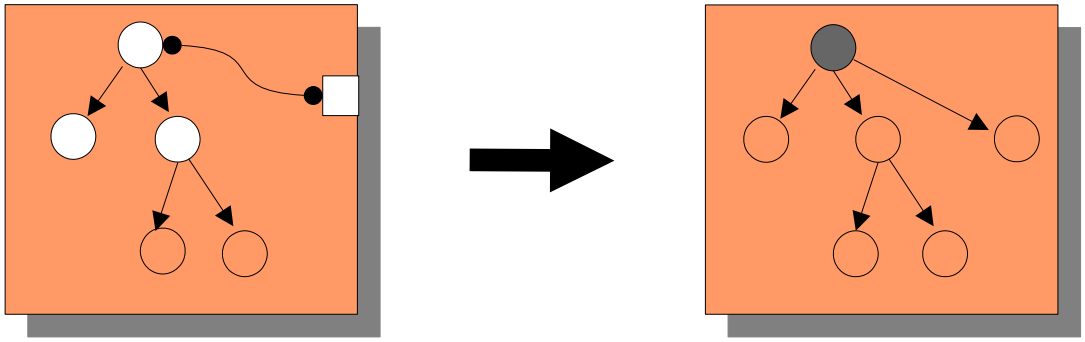


Systematization Towards Graybox Component Models

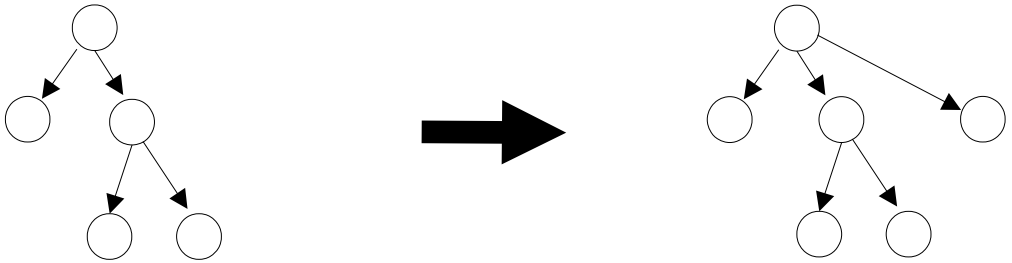
Composition with declared hooks



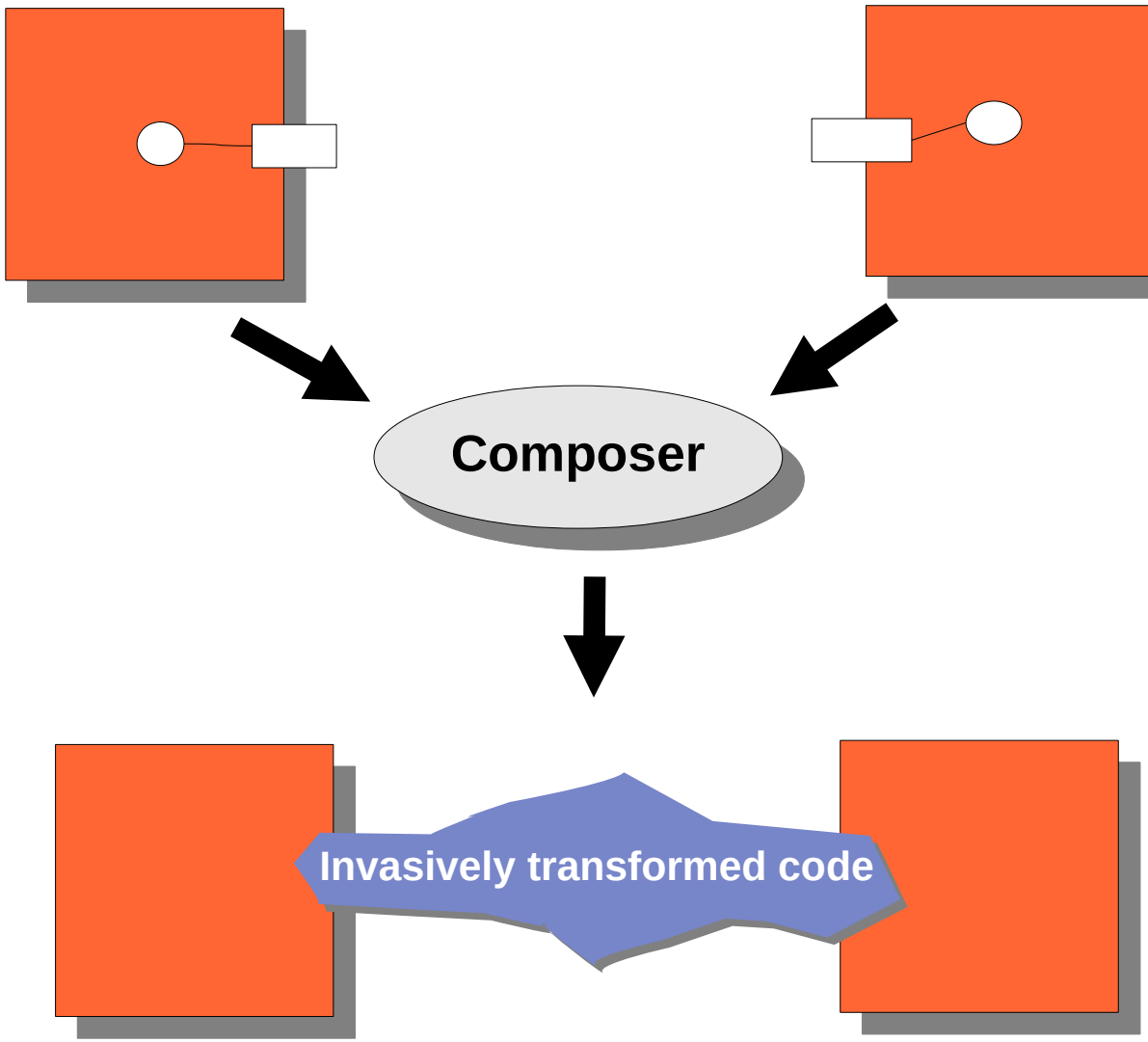
Composition with implicit hooks



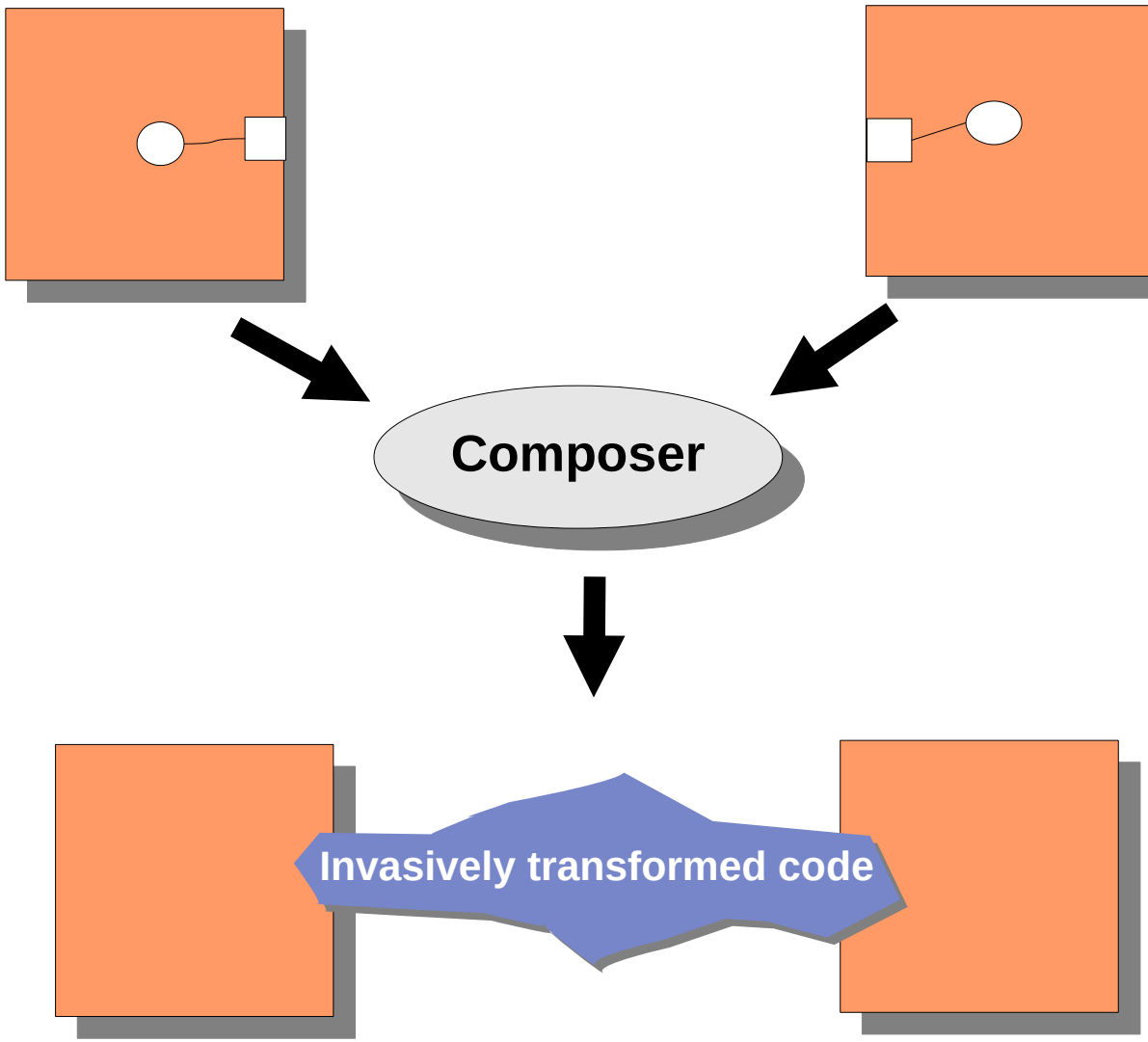
Refactorings Transformations



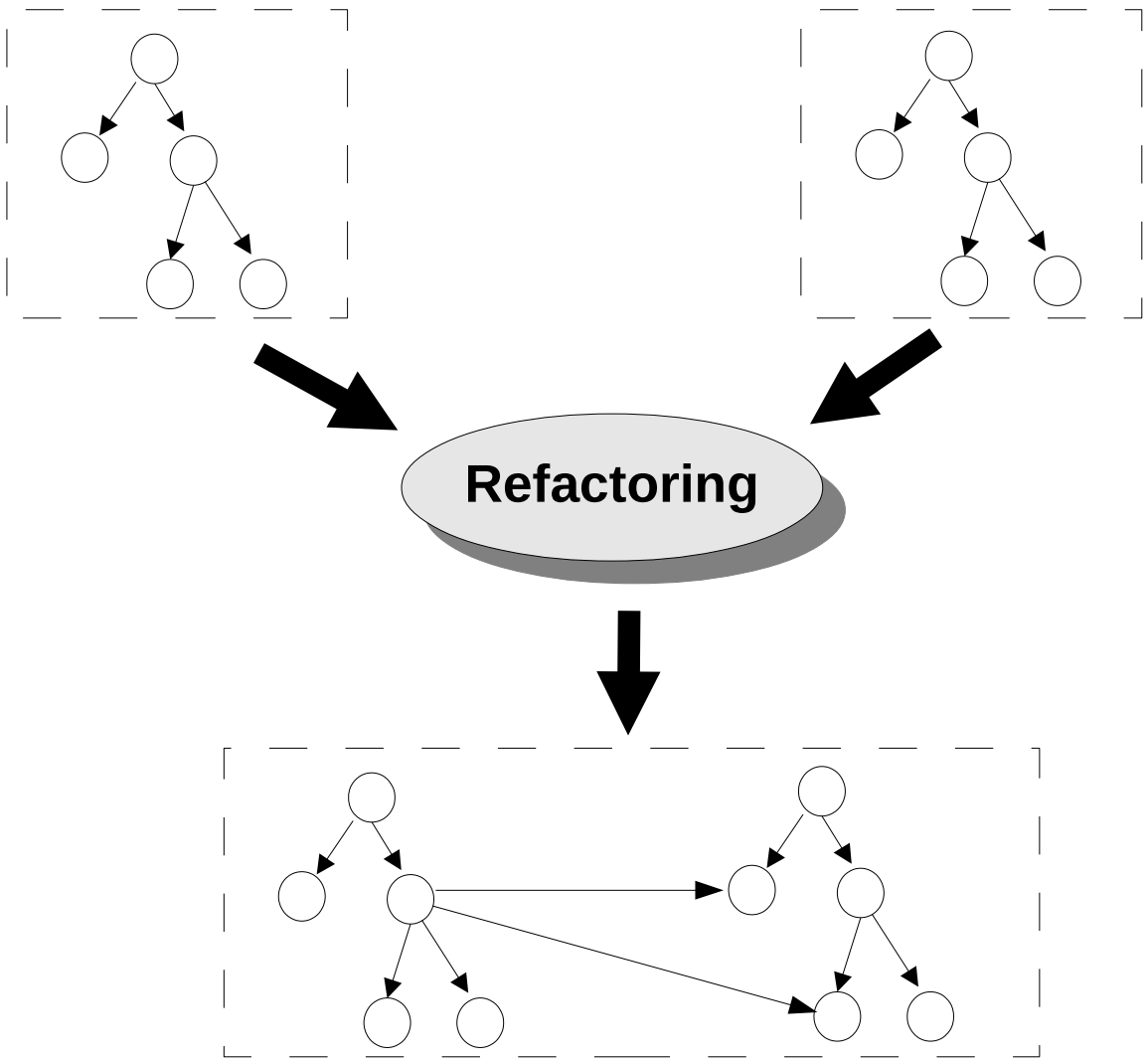
Invasive Composition Builds On Transformation on Declared Hooks



Invasive Composition Builds On Transformation Of Implicit Hooks



Refactoring Builds On Transformation Of Abstract Syntax

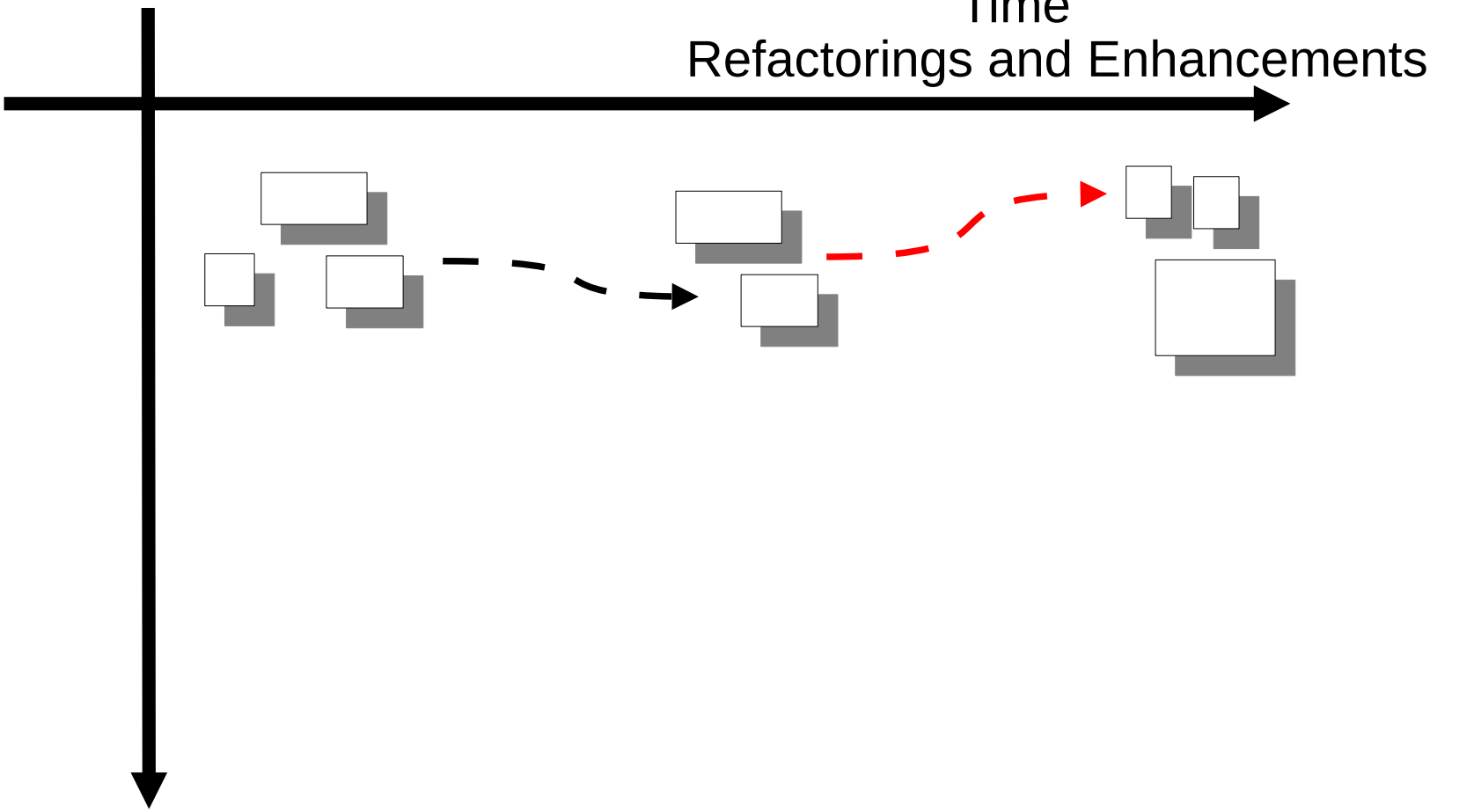


Unification of Approaches

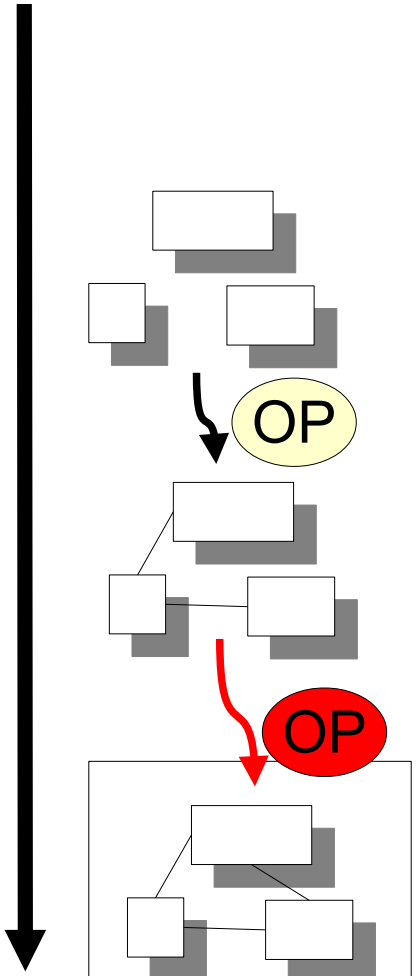
- Invasive composition, based on refactoring operations, can realize most of the current composition operations
  - inheritance
  - views, aspects, role-model merging
  - connectors
- But the component models differ slightly

33.4 Unifying Composition and Evolution

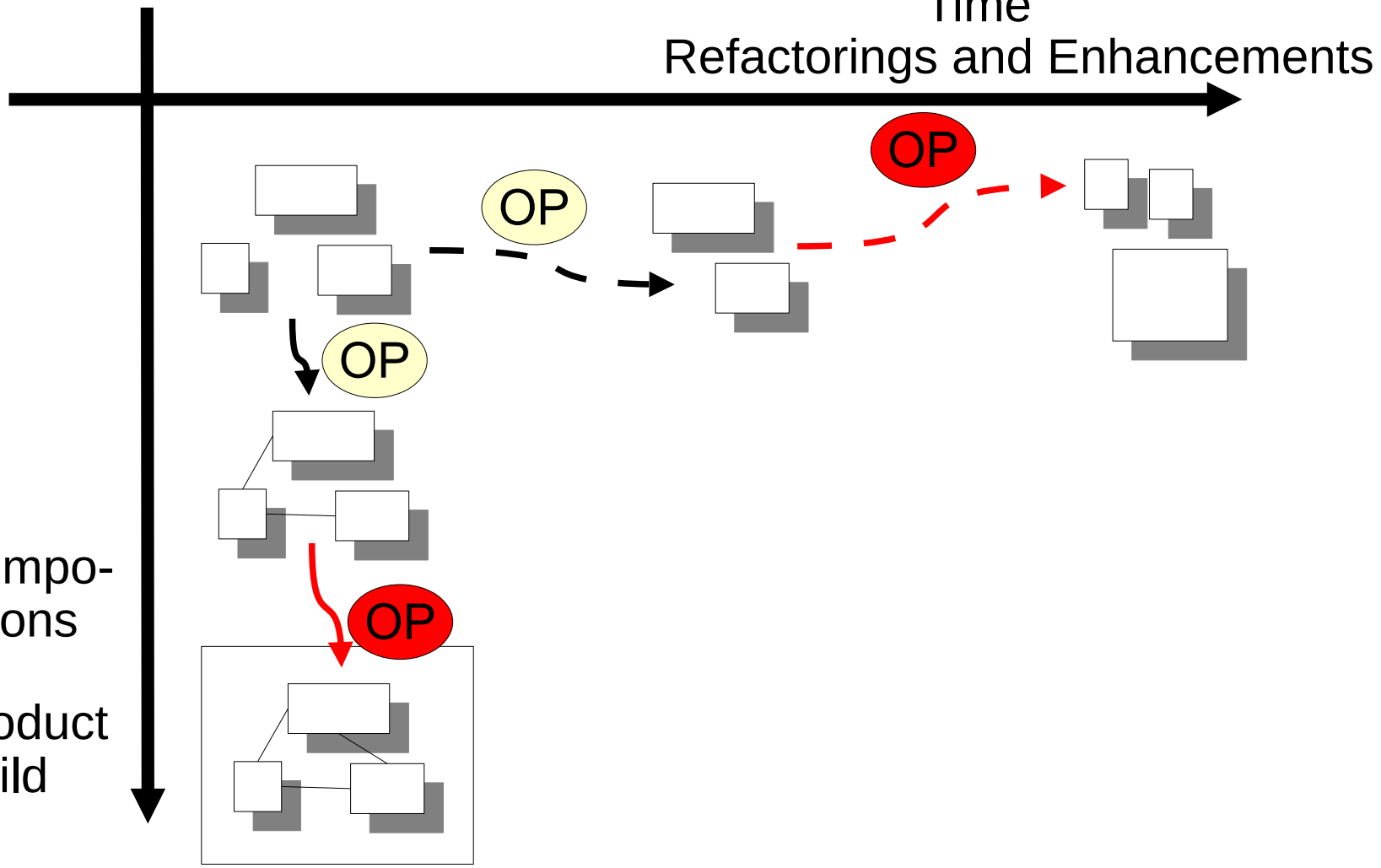
The Dimension of Refactoring



The Dimension of Build



A Uniform Operator-Based View on Two Dimensions of Software Engineering

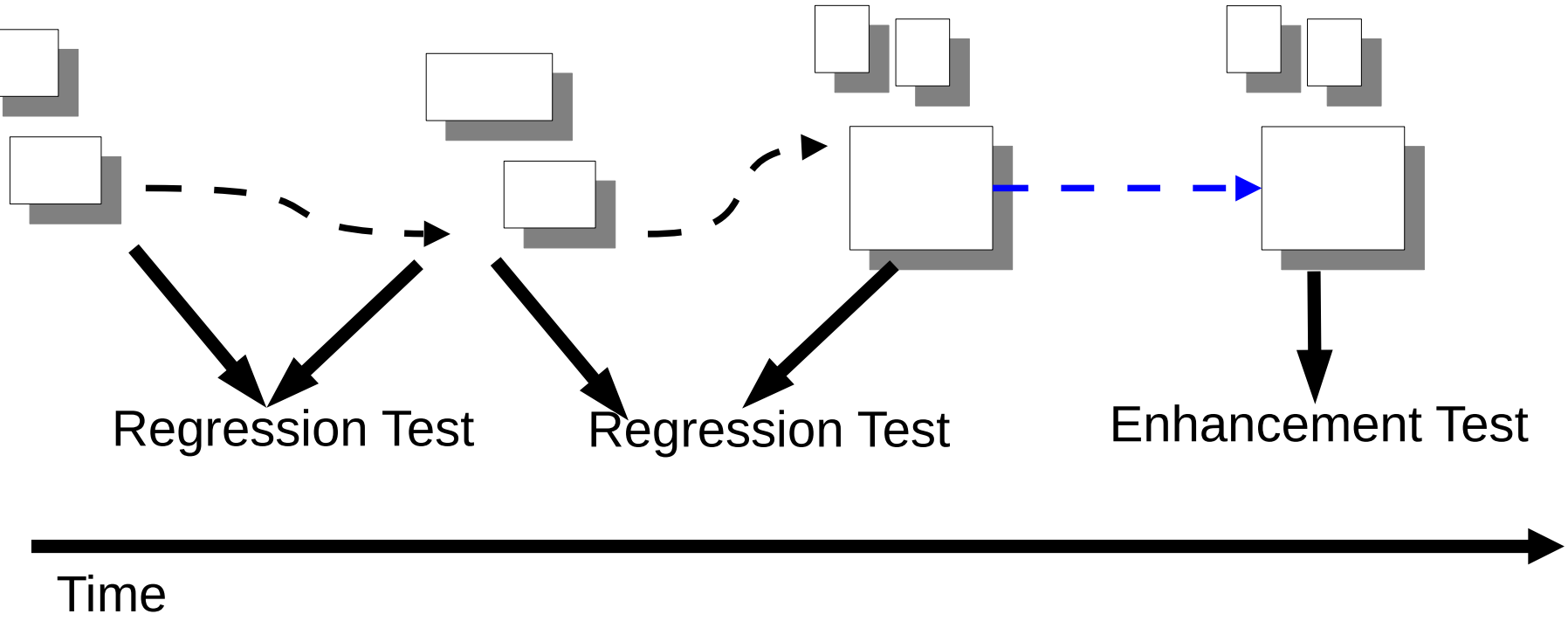


Algebraic Features of Refactoring Operators

- **Identity** (Semantics preserving)
  - Refactorings are identity operations concerning the semantics
  - Connector exchange is semantics preserving
- **Identity** (Syntactic)
  - Refactorings should be syntax-preserving
  - Y2K problem
    - Only syntax-preserving transformations were accepted by the developers and companies

Regression Tests as Composition Operation on Subsequent Versions

- Regression tests are operators that check semantic identity



Other Useful Algebraic Features

- **Idempotence** +; + == +
  - Syntactically, refactorings must be idempotent
    - RECODER is syntactically idempotent
- **Commutativity** a+b = b+a
  - If two operations are commutative, they can be interchanged to implement the more important requirement
  - Connections on different parts are commutative
    - Order of build becomes unimportant
- **Associativity** (a+b)+c = a+(b+c)
  - Order of build becomes unimportant
- **Monotonicity**: Refactorings that merely add stuff
  - Glueing operations (Adapters, Bridges): Do not modify, but produce glue
  - Enrichments (extensions)

Semantically Invariant Composers are Symmetries

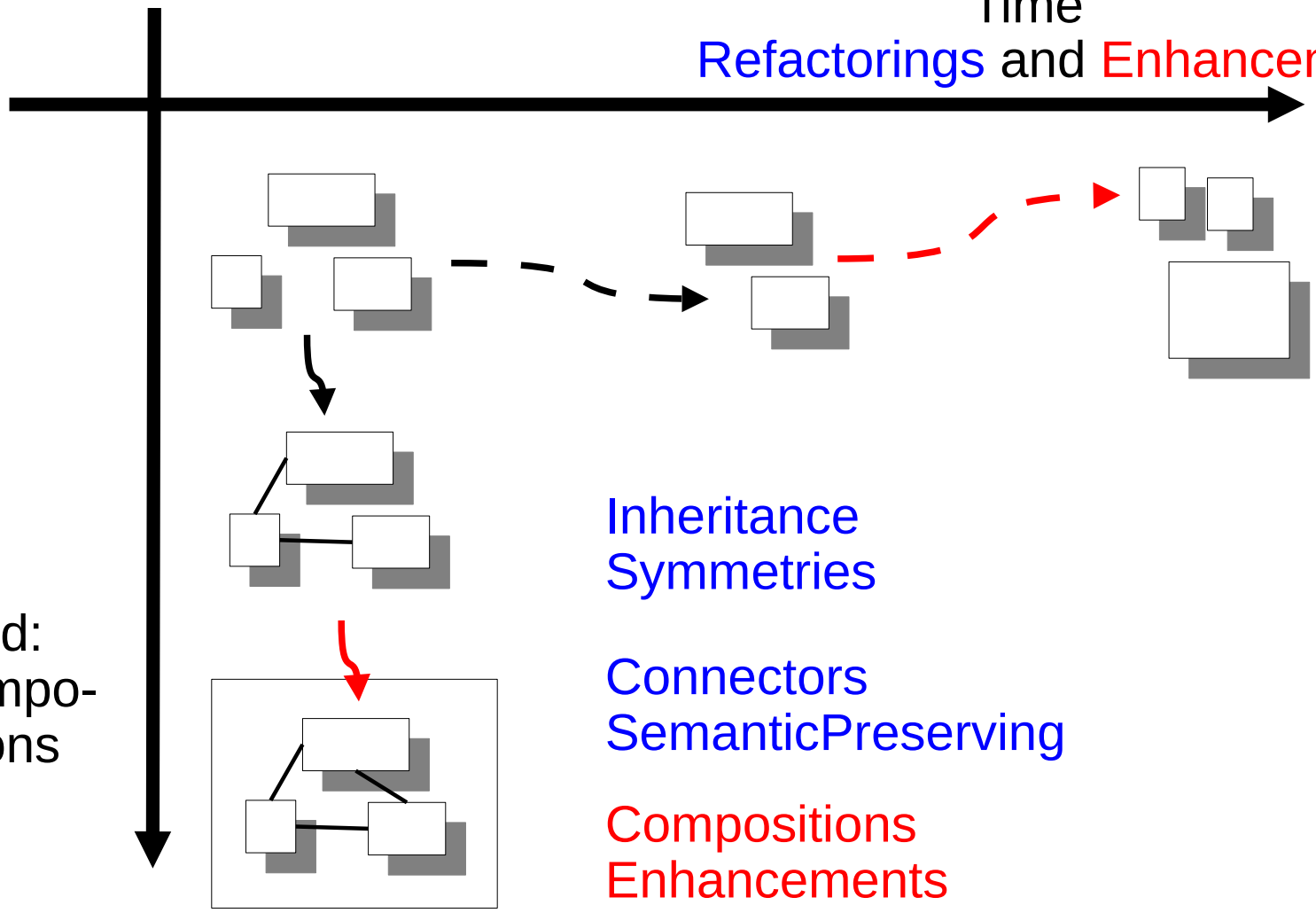
- **Symmetries** [Coplien]
  - Symmetric operations have an invariant which they preserve
    - Rotation preserves shape, but reorients a symmetric artifact
  - Symmetric operations form symmetry groups
- Examples:
  - Refactorings are symmetries
    - Because they preserve the semantics of the code, but only change the structure
  - Conformant inheritance is a symmetry
    - Conformance maintains the contracts of arguments of methods
  - Connectors are symmetries
    - Because they preserve communication semantics

Central Idea of Refactoring-Based Software Development

- **Harmless**
  - Semantics preserving (refactoring)
  - Contract preserving
  - Syntax preserving
- **Additive** (enhancements, but preserving)
  - Symmetries (invariant preserving)
- **Dangerous**
  - Non-preserving enhancements
  - Modifications

Split up development steps into applications of harmless, additive, and dangerous software operators

Use Harmless Steps in Two Dimensions



Beyond Refactoring

- What started as refactorings, is now ending up in a concept of harmless software evolution operators
  - Refactoring is strong, due to its *harmlessness*
  - We will split development into harmless, monotonous and difficult operations

- Software *build* and *evolution* get a common background
  - Both are based on transformation operators from an algebra
  - Design patterns are no isolated concept, but are related to component-based software engineering (graybox component systems)
    - Both forms of operators can be realized as static metaprograms with graybox component models
    - Can be supported by common tools (RECODER and COMPOST as examples, http://sf.recoder.net http://www.the-compost-system.org)

Software Engineering Beyond Refactoring

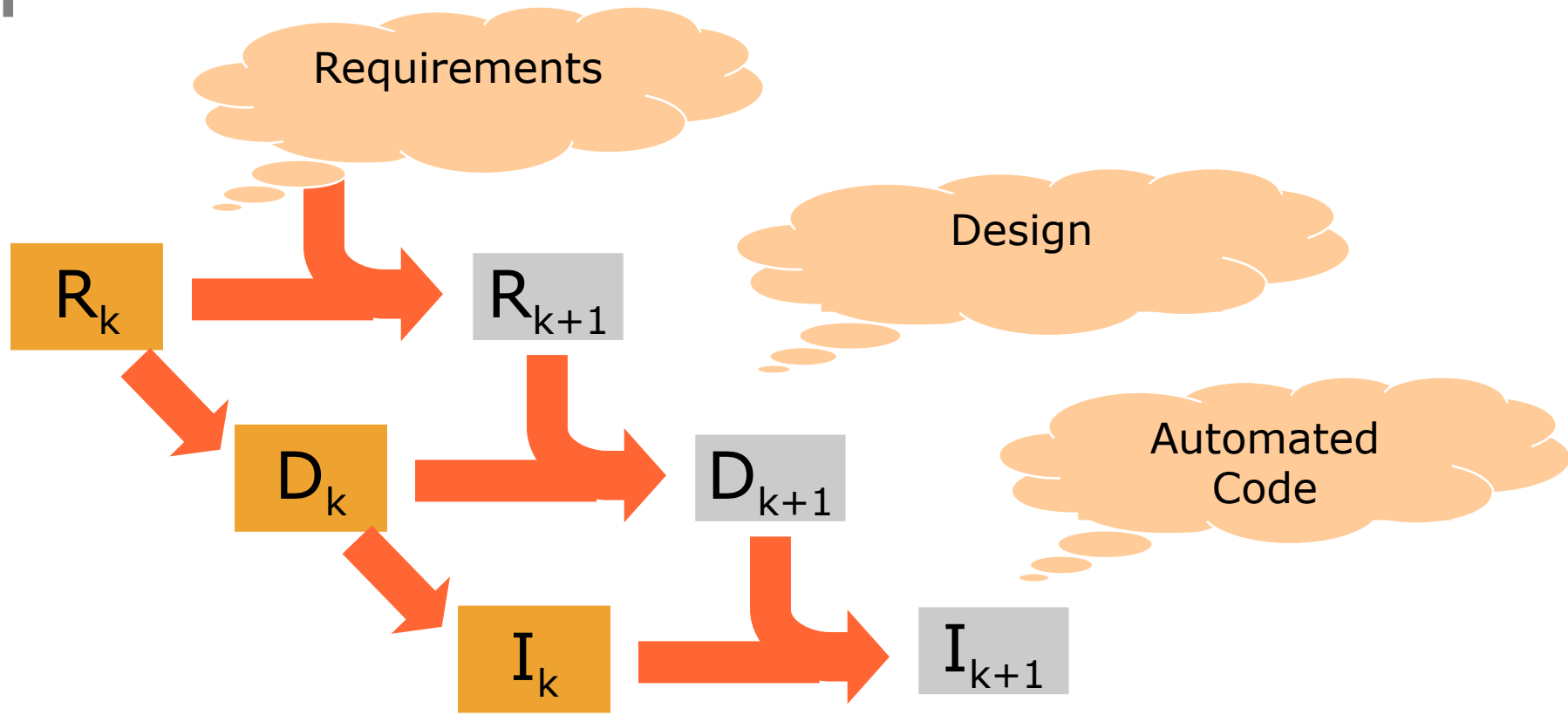
- Use harmless operations as long as possible
  - Semantics-preserving (refactorings)
  - Symmetries (conformant inheritance)
  - Syntax-preserving
  - Idempotents
- Validate algebraic features
  - Program analysis
  - Contract checker
  - Regression test
  - diff
- Compositions are software operators, too

- Software Engineering needs more harmless operations!!

Vision

- Replace old tools by refactoring operators and composition languages...
  - Build tools
    - Linker
  - Modelling
    - Inheritance
    - Architecture systems
  - Evolution
    - Refactorings

Vision: Automated Design, Build, And Evolution



Appendix

www.the-compost-system.org

recoder.sourceforge.net

Book “Invasive Software Composition”

Springer, Feb 2003

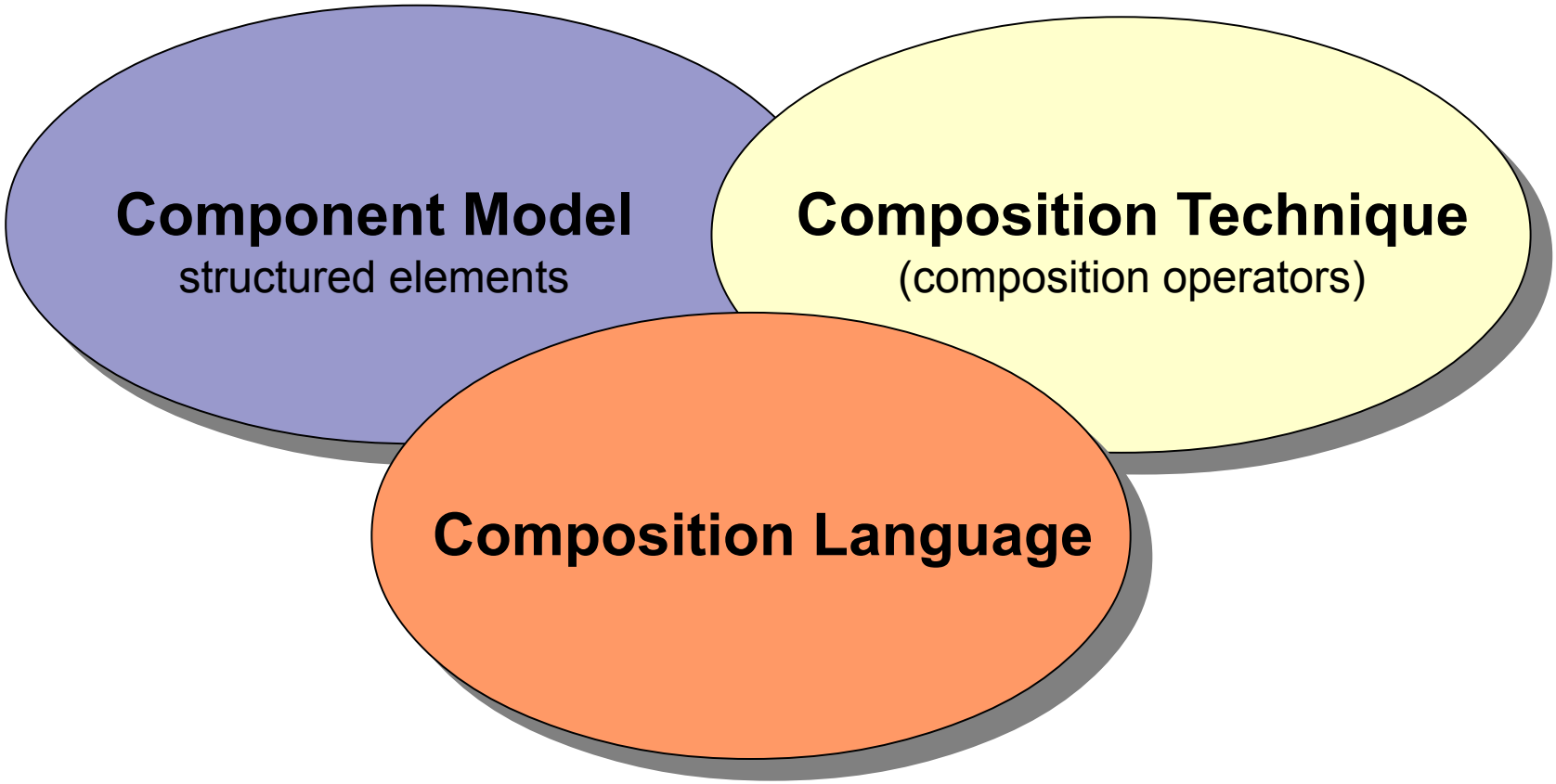
33.A Invasive Software Composition Operators

... preview onto the summer

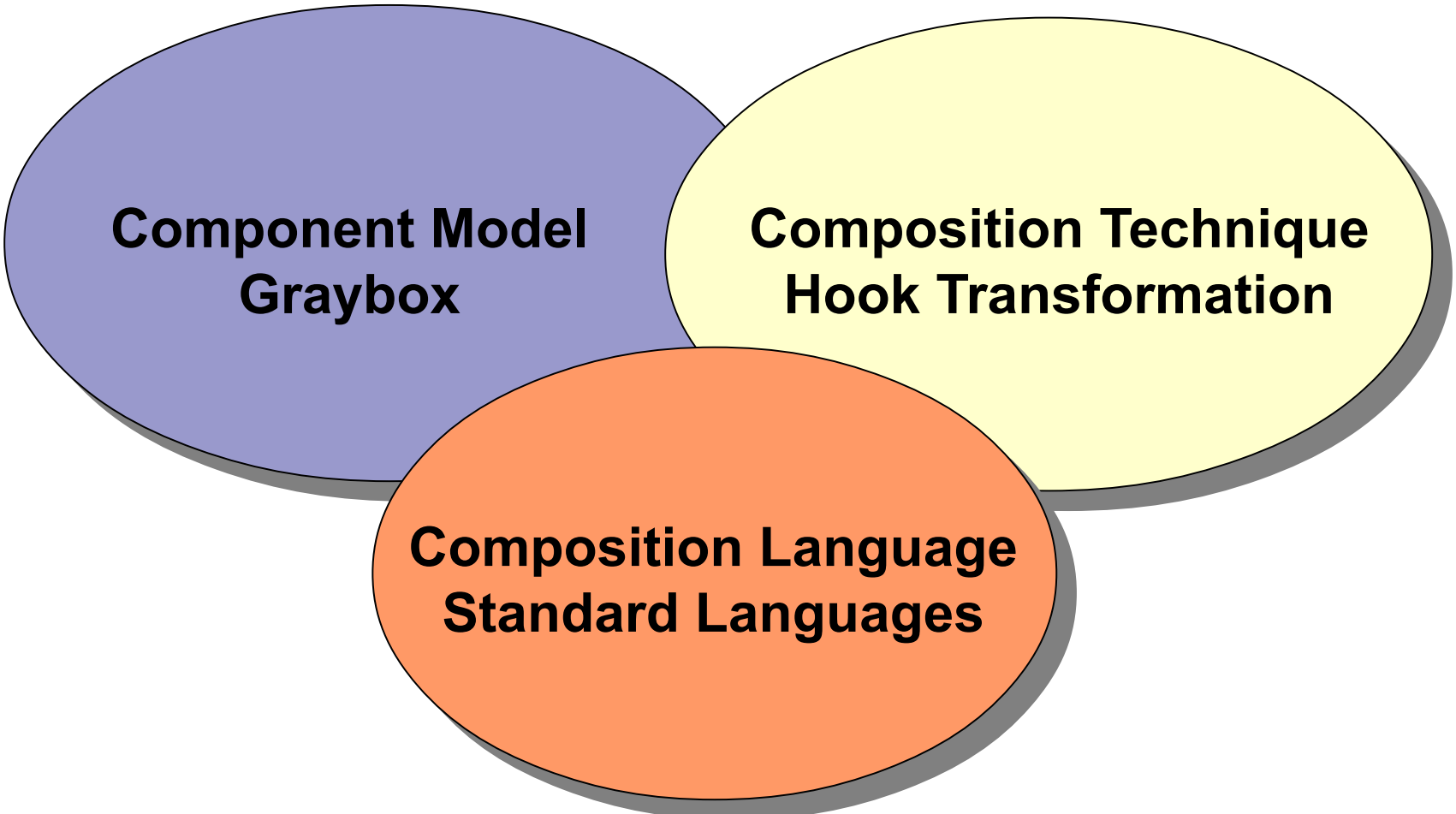
(CBSE course)

Composition Systems

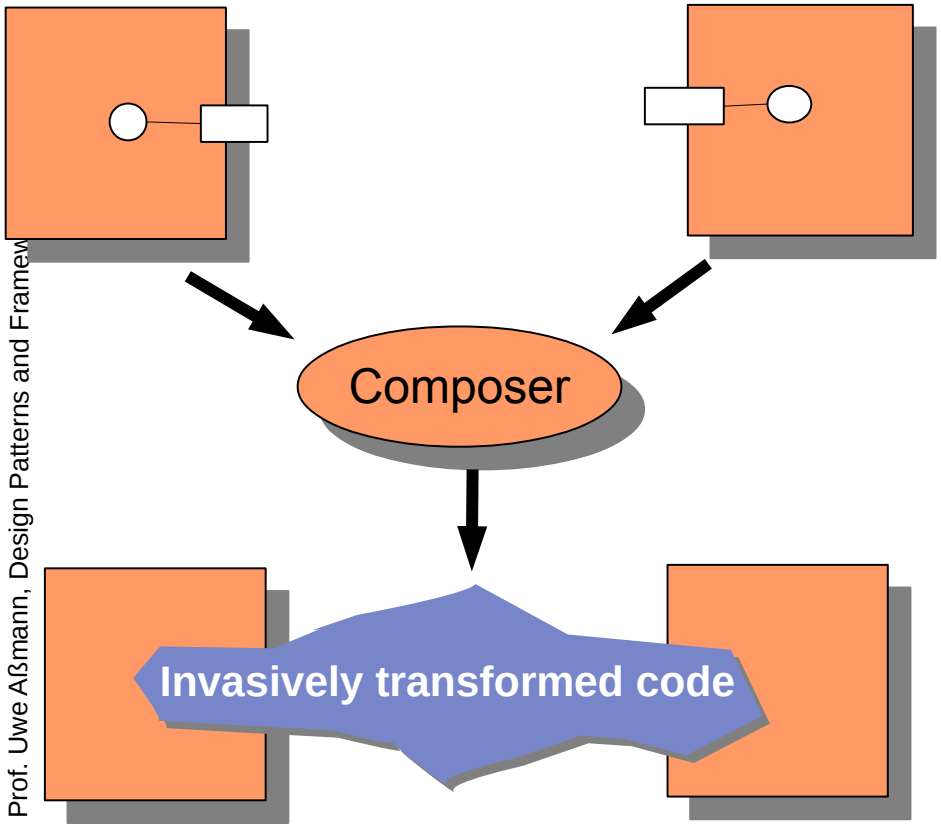
- A **composition system** is a two-level composition algebra, whose elements (called components) have a composition interface (hooks, ports)



Invasive Software Composition



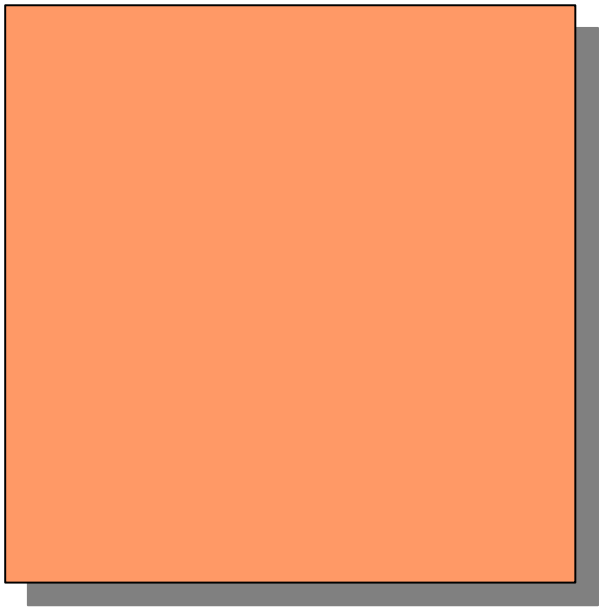
Invasive Composition as Hook Transformations



Invasive Composition adapts and extends components at hooks by transformation (2-level composition algebra)

The Component Model of Invasive Composition

- The basic element is a **fragment component (fragment box)**, a set of program elements
- May be
  - a class
  - a package
  - a method
  - an aspect
  - a meta description
  - a composition program



Boxes have Hooks

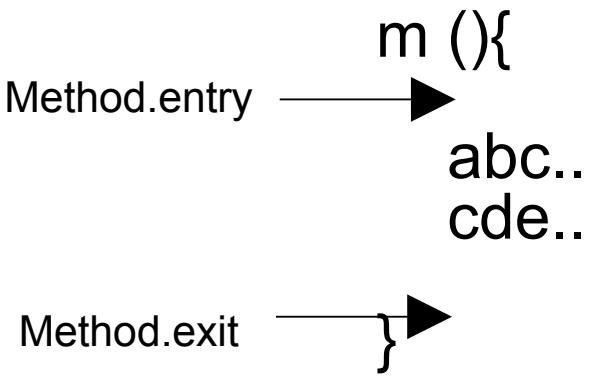
Hooks are arbitrary fragments or spots in a fragment component

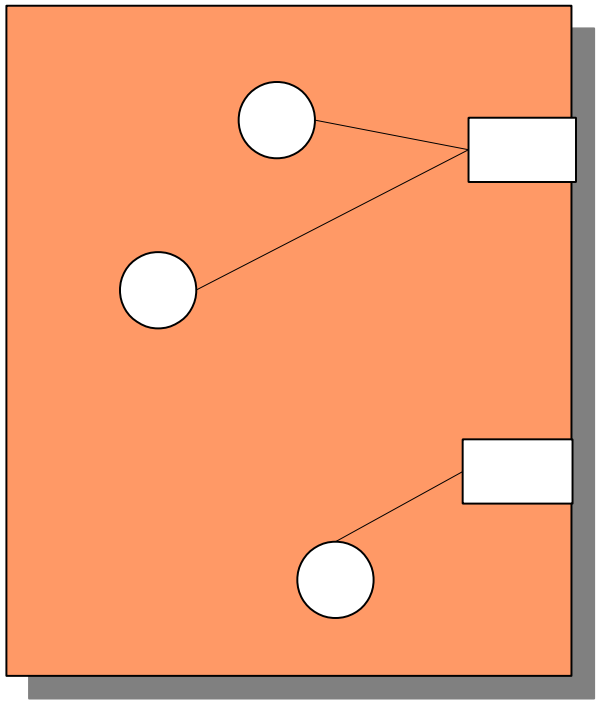
which are subject to change

- beginning/end of lists
- method entries/exits
- generic parameters

Implicit Hooks (aka Static Join Points)

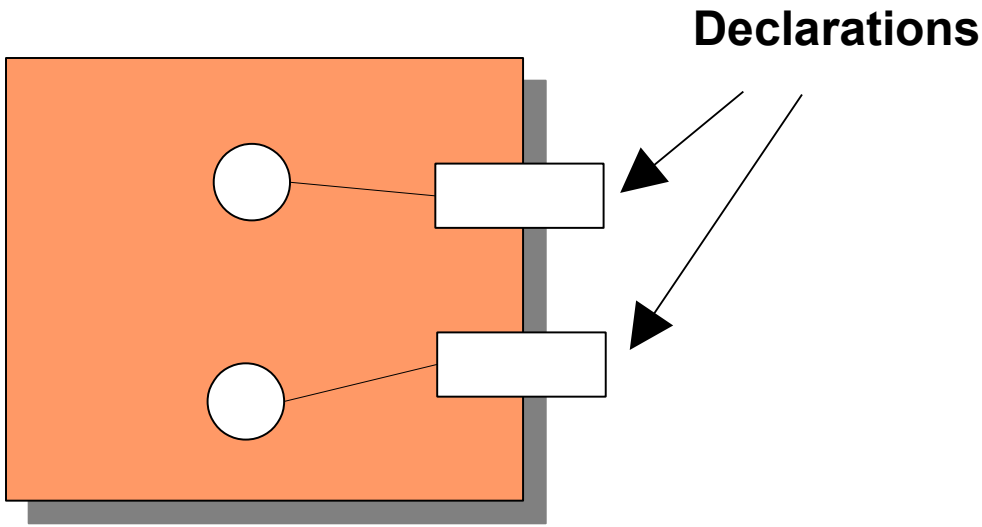
- Given by the programming language, the DTD or Xschema
  - Example Method Entry/Exit





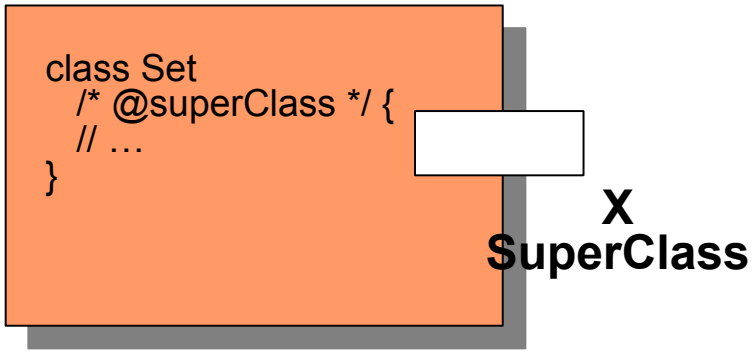
Declared Hooks (Generic Parameters)

Declared Hooks are declared by the box writer as variables in the hook's tags.



Declaration of Hooks

- by special keywords
- by markup tags
- Language Extensions (keywords..)
- Standardized Names
- Comment Tags



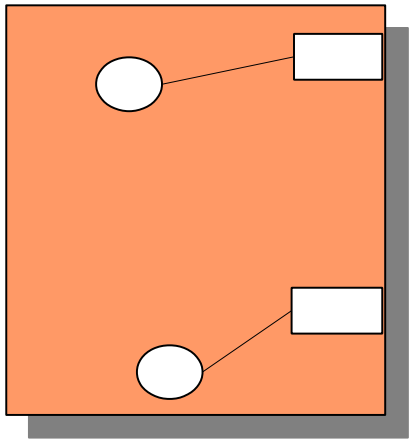
<superclasshook> X *</superclasshook>*

```
class Set extends genericXSuperClass {  }

class Set  /* @superClass */ {
   // …
}
```

The Composition Technique of Invasive Composition

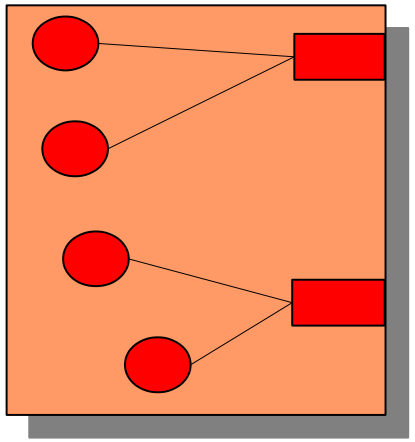
Invasive Composition
adapts and extends
components
at hooks
by transformation

Composition on Implicit Hooks



MethodEntry

MethodExit



MethodEntry

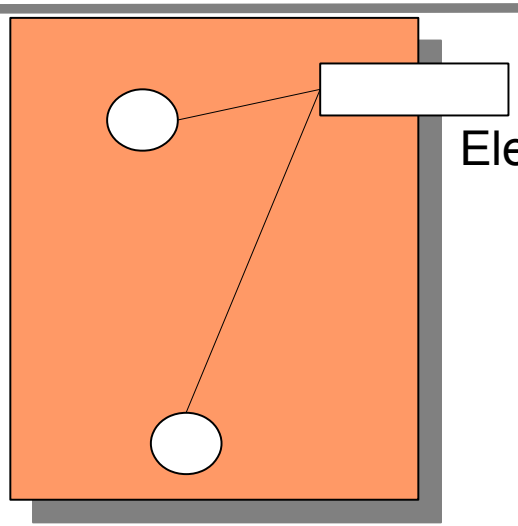
MethodExit

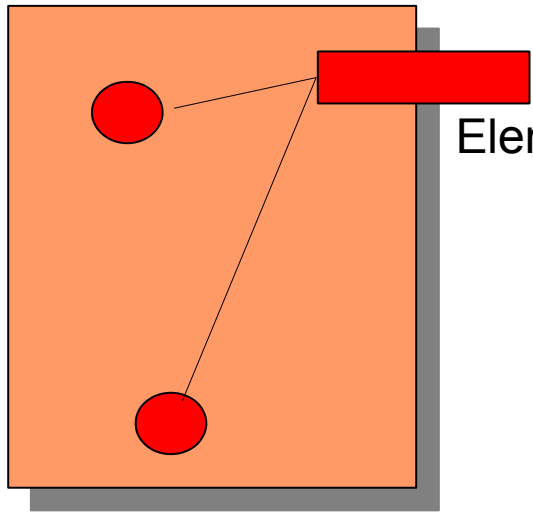
```
m (){

    abc..
    cde..

}
```

```
m (){
    print(“enter m”);
    abc..
    cde..
    print(“exit m”);
}
```

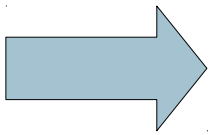
```
box.findHook(„.MethodEntry“).extend(“print(\”enter m\”);”);

box.findHook(„MethodExit“).extend(“print(\”exit m\”);”);
```

Composition on Declared Hooks





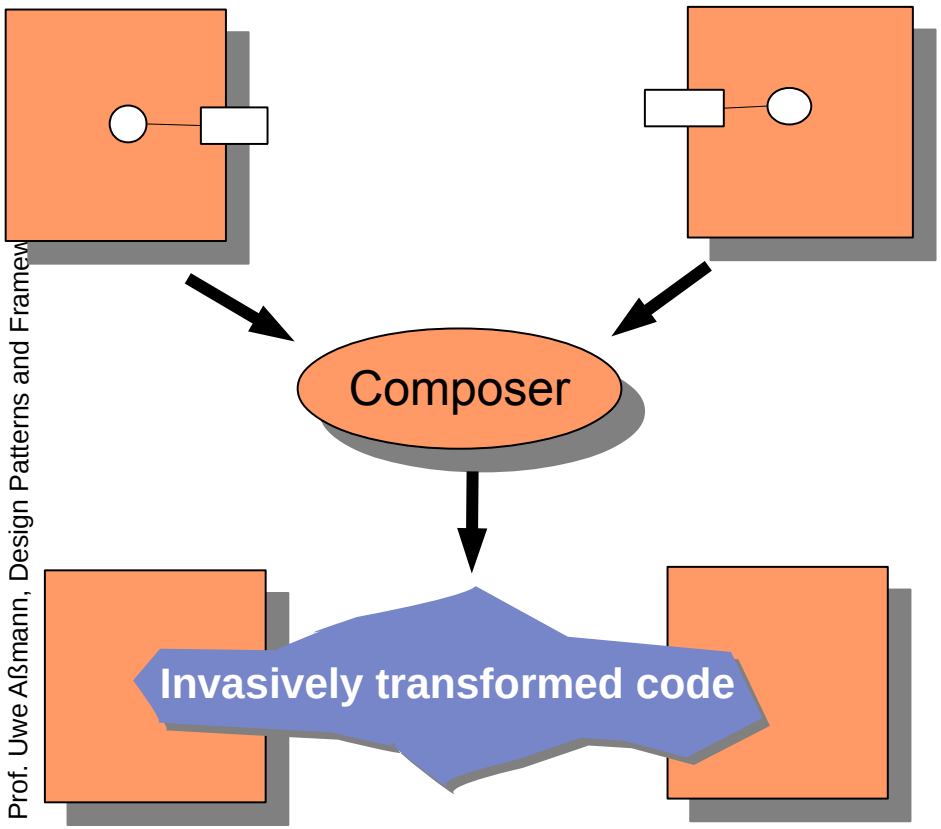

```
List(<hook>Element</hook>) le;

....
le.add(new <hook>Element</hook>());

...
```



```
List(Apple) le;

....
le.add(new Apple());

...
```

box.findHook(„Element“).bind(“Apple”);

Invasive Composition as Hook Transformations



- Invasive Composition works uniformly on
  - declared hooks
  - implicit hooks
- Allows for unification of
  - Inheritance
  - Views
  - Aspect weaving
  - Parameterization
  - Role model merging

The Composition Language of Invasive Composition

- As a composition language, arbitrary languages can be used
  - Standard languages (Java)
  - XML
  - Rule languages
- Meta-composition possible
  - composition classes, methods

Atomic and Compound Composition Operators

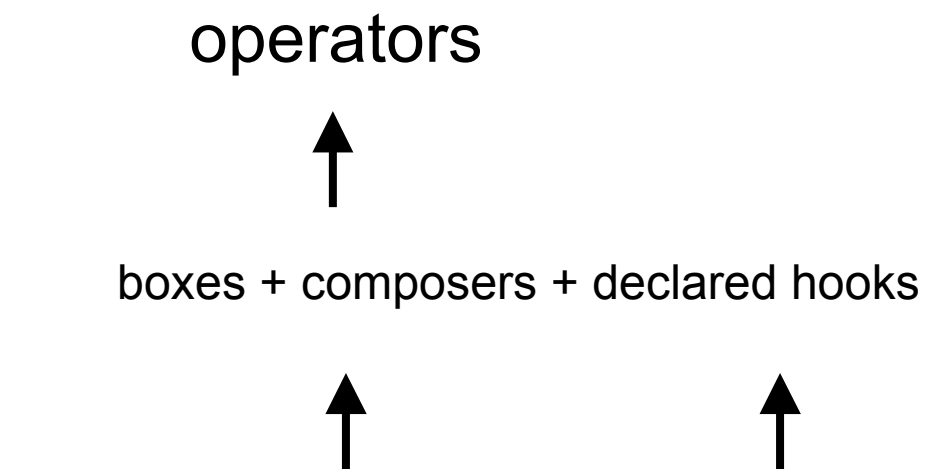
- **bind** hook (parameterize)
  - generalized generic program elements
- **rename** component, rename hook
- **copy** component
- **extend**
  - extend in different semantic versions

Compound composition operators:

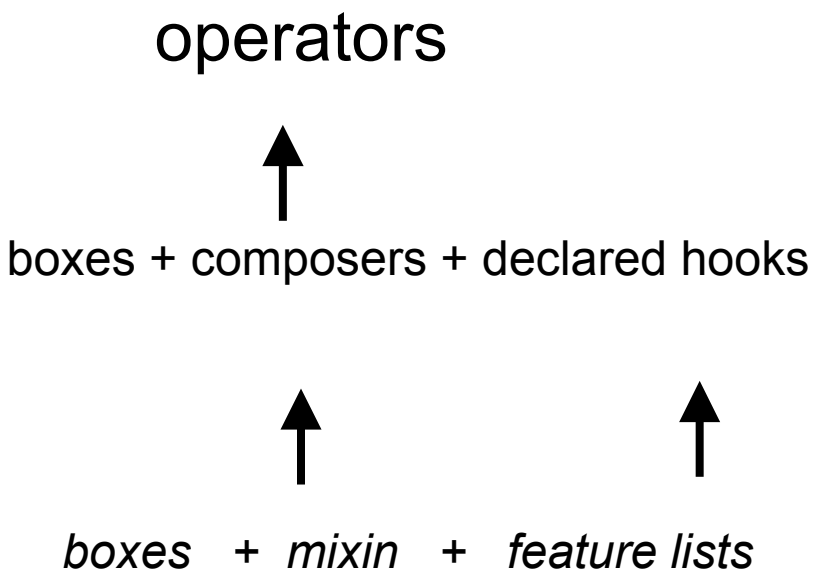
- **inheritance**
- **views**
  - Class merge
  - Role model merge
  - Package merge
  - Intrusive data functors
- **connect**
- **distribute**
  - aspect weaving

33.4.2 What Can You Do With Invasive Composition?

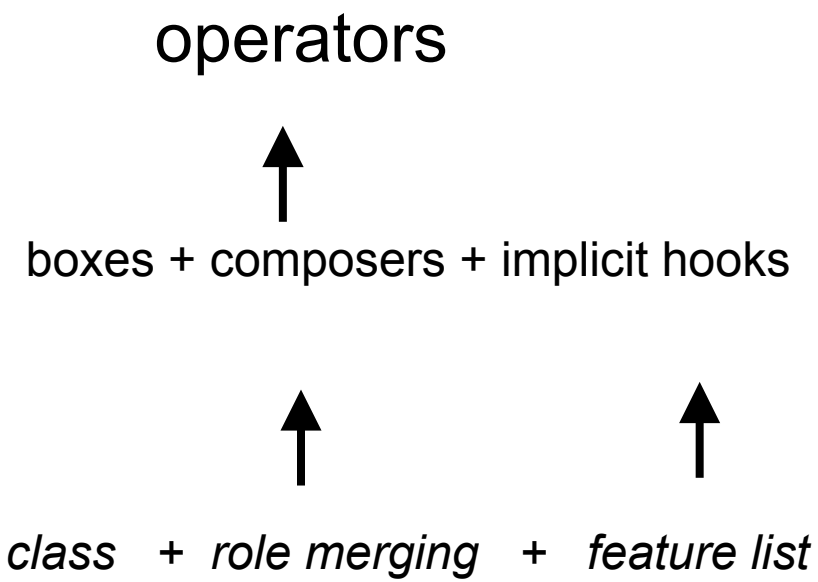
Composers Generalize Connectors



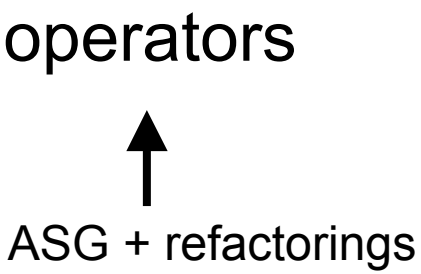
Composers Generalize Inheritance Operators



Composers Generalize Role Model Merge



Refactorings are Operators on the ASG



Refactoring Can Be Regarded As Primitive Composition