Refactoring and Beyond

Refactoring as a Basis for a Unified Operator-Based View on Software Engineering

Prof. Dr. Uwe Aßmann
TU Dresden
Lehrstuhl Softwaretechnologie
07-1.0, Jan 21, 2008
Obligatory Literature


► Non-obligatory:
A refactoring is a semantics-preserving, but structure-changing transformation of a program.

Often, the goal is a design pattern.
Refactorings are Harmless Evolution Operations

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

Evolution = Refactorings + Enhancements
A Little History

- 80s: Broad-spectrum languages (CIP)
- System REFINE
- 1992 William Opdyke coined the term *refactoring*
- 1997, Karlsruhe University started a refactoring tool
  - Based on Walter Zimmer's thesis “Design patterns as operators”
  - Idea: a refactoring is a *semantics preserving operator*, transforming class graphs to class graphs
  - A refactoring operator can be implemented as a static metaprogram
- 1998, during Zimmer's work was reimplemented into the Together CASE tool, the world-wide first CASE tool with refactoring support
Classes of Refactorings

- **Rename Entity**
  - Problem: update all references on definition-use-graph

- **Move Entity**
  - Move class feature (attribute, method, exception,...)
  - Problem: shadowing of features along scoping

- **Split Entity or Join Entity**
  - Method, class, package
  - Problem: updating of references

- **Outline Entity (Split Off) or Inline Entity (Merge)**
  - Method, generic class
  - Problem: introduction of parameters
Example: Rename Refactorings in Programs

How to change the name of variable Foo and keep the program consistent?
Refactor the name `Person` to `Human`:

class Person { .. }
class Course {
    Person teacher = new Person("Jim");
    Person student = new Person("John");
}

class Human { .. }
class Course {
    Human teacher = new Human("Jim");
    Human student = new Human("John");
}
Definition-Use Graphs (Def-Use Graphs) as a Basis of Refactorings

- Every language and notation has
  - Definitions of items (define the variable Foo)
  - Uses of items (references to Foo)
- This is because we talk in specifications about *names of objects* and their use
  - Definitions are done in a data definition language (DDL)
  - Uses are part of a data manipulation language (DML)
- Starting from the abstract syntax, the *name analysis* finds out about the definitions, uses, and their relations (the *Def-Use graph*)
  - Def-Use graphs exist in every language!
  - How to specify the name analysis, i.e., the def-use graph?
Refactoring on Def-Use Graphs

- For renaming of a definition, all uses have to be changed, too
  - We need to trace all uses of a definition in the Def-Use-graph
  - Refactoring works always on Def-Use-graphs

- Refactoring works always in the same way:
  - Change a definition
  - Find all dependent references
  - Change them
  - Recurse handling other dependent definitions

- Refactoring can be supported by tools
  - The Def-Use-graph forms the basis of refactoring tools

- However, building the Def-Use-Graph for a complete program costs a lot of space and is a difficult program analysis task
  - Every method that structures the Def-Use-Graph benefits immediately the refactoring
  - either simplifying or accelerating it
From Refactoring to Software Composition
Soon One Can See...

- There are other operators in modern software engineering approaches
  - **Connectors** are operators
    - Architecture systems
    - The connect components at ports
  - **Inheritance** are operators
    - [Steyaert 96 OOPSLA Reuse operators]
    - [Braha&Cook 90 OOPSLA mixin based inheritance]
    - “mixins” compose superclasses with mixins
  - **Parameterizations** are operators
    - [GenVoca/Batory parameterization as composition]
  - **Role Model merges** are operators
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 activity is an operator in an algebra
  - Implementation: library
- How do the elements of the algebra look:
  - Connectors: Components with port
  - Inheritance: Classes with feature list
  - Refactorings: Abstract syntax graph
  - Parameterizations: Abstract syntax graph templates (fragment templates)
- Can there be a component model for all of them?
  - Solution: graybox components
Invasive Software Composition with Graybox Components

... preview onto the summer (CBSE course)
Composition

Component Model
Composition Technique
Composition Language
Invasive Composition as Hook Transformations

Invasive Composition adapts and extends components at hooks by transformation
The Component Model of Invasive Composition

- The basic element is a 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 box 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

```
Method.entry
m (){ abc.. cde.. }
Method.exit
```
Declared Hooks are declared by the box writer as variables in the hook’s tags.
Declaration of Hooks

- Markup Tags
- Language Extensions (keywords..)
- Standardized Names
- Comment Tags

```<superclasshook> X </superclasshook>

class Set extends genericXSuperClass { }

class Set /* @superClass */
Invasive Composition adapts and extends components at hooks by transformation
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");");
List(Element) le;

....
le.add(new Element());
...

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

- Using standard languages (Java)
  - and XML itself
- Meta-composition possible
  - composition classes, methods
- Other composition languages are possible
What Can You Do With Invasive Composition?
Atomic and Compound Composition Operators

- **bind** hook (parameterization)
  - generalized generic program elements
- **rename** component, rename hook
- **remove** value from hook (unbind)
- **extend**
  - extend in different semantic versions

- **inheritance**
- **views**
  - Intrusive data functors
- **connect** (bind hook 1 and 2)
- **distribute**
  - aspect weaving
Composers Generalize Connectors

operators

boxes + composers + declared hooks

boxes + connectors + ports
Composers Generalize Inheritance
Operators

operators

boxes + composers + declared hooks

boxes + mixin + feature lists
Composers Generalize Role Model Merge

operators

boxes + composers + implicit hooks

class + role merging + feature list
Refactorings are Composition Operators on the ASG

operators

↑

ASG + refactorings
Refactoring Can Be Regarded As Primitive Composition

- Component Model
- Abstract Syntax Graphs
- Composition Language
- Composition Technique
- Static Megaprogramming
- Transformation
The Uniform Operator Model of Invasive Software Composition
Operations on Different Levels

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

Refactorings
Transformations
Operations on Different Levels

- 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

Invasively transformed code
Invasive Composition Builds On Transformation Of Implicit Hooks

Composer

Invasively transformed code
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
Consequences of Uniform Operators For Software Engineering
The Dimension of Refactoring
A Uniform Operator-Based View on Two Dimensions of SE
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 Operations 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)
► Dangerous
  ▪ Enhancements
  ▪ Modifications

Split up development steps into harmless, additive, and dangerous ones
Use Harmless Steps in Two Dimensions

Build: Compositions

Inheritance
Symmetries
Connectors
SemanticPreserving
Compositions
Enhancements

Refactorings and Enhancements

Time
Beyond Refactoring

▶ What started as history, is now ending up in a concept
  ▪ 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 (*beautiful structures*) 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 everywhere
   - Semantics-preserving (refactorings)
   - Symmetries (conformant inheritance)
   - Syntax-preserving
   - Idempotents

► Validate algebraic features
   - Program analysis
   - Contract checker
   - Regression test
   - diff

► 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

Requirements

$R_k$ $\rightarrow$ $R_{k+1}$

$D_k$ $\rightarrow$ $D_{k+1}$

$I_k$ $\rightarrow$ $I_{k+1}$

Design

Automated Code
The End

www.easycomp.org
www.the-compost-system.org
recoder.sourceforge.net

Book “Invasive Software Composition”
Springer, Feb 2003