Invasive Software Composition

Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised 2005-2010 by C. Kessler.

Invasive Composition: A Fragment-based composition technique

Invasive composition adapts and extends components at hooks by transformation

Literature:
Uwe Assmann: Invasive Software Composition, Springer, 2003. Chapter 4

The Component Model of Invasive Composition

The component is a fragment container (fragment box)
a set of program elements

Uniform representation of
a fragment
a class, a package, a method
a set of fragments
an aspect
metadata
a composition program

Fragment Components Have Hooks

Hooks are variation points of a component: fragments or positions, which are subject to change

Software variation points
Method entries/exports
Generic parameters
...

The composition interface of a component

Implicit Hooks In Software

Example: Method Entry / Exit

Method.entry
meth.entry
abc, cde...
Method.exit
meth.exit

Given by the programming language

Declared Hooks

Declared hooks are declared by the component writer as code parameters

Declarations
Declared Hooks for Generic Types

```java
<< ClassBox >>
class SimpleList {
    genericTType elem;
    SimpleList next;
    genericTType getNext() {
        return next.elem;
    }
}
```

Placeholder item (hooked program element) will be removed by the binding.

Declared Hooks for Generic Modifiers

```java
/* @hook Modifier MY */
public print() {
    System.out.println("Hello World");
}
```

Component methodComponent = cs.createMethodBox();
Hook modif = methodComponent.findHook("MY");
if (parallelVersion) {
    modif.bind("synchronized");
} else {
    modif.bind(" ");
}

synchronized public print () {
    System.out.println("Hello World");
}
```

Declared Hooks for Generic Statements

```java
public print() {
    /* @hook Statement MY; */
}
```

Component methodComponent = cs.createMethodBox();
Hook statement = methodComponent.findHook("MY");
if (StdoutVersion) {
    statement.bind("System.out.println("Hello World");");
} else {
    statement.bind("FileWriter.println("no way");");
}

public print () {
    System.out.println("Hello World");
}
```

The Composition Technique of Invasive Composition

Invasive Composition adapts and extends components at hooks by transformation

An invasive composition operator treats declared and implicit hooks uniformly

Binding hooks by invasive composition

component.findHook("MethodEntry").extend("print("enter m");");
component.findHook("MethodExit").extend("print("exit m");");

MethodEntry
MethodExit
Difference between Declared and Implicit Hooks

Declared hooks refer to a dummy program element
- e.g., generic type variable, statement placeholder, markup in names, or markup comment.
- Visible in unparsed source code
- Binding will remove the declared hook from the fragment box.

Implicit hooks refer to a point in the program's source code
- Only defined by convention (programming language constructs)
- Only as target for code insertion.
- Not visible in unparsed source code.
- Binding will not remove the implicit hook.

What Can You Do With Invasive Composition?

already seen:
- generic types
- generic modifiers
- generic statements

Invasive composition can improve COTS component adaptation

When static relations have to be adapted
- inheritance relationship
- import relationship

When delegation pointers have to be inserted

When physical unity of logical objects is desired
- Integration of multiple roles / feature groups into one class
  (- subject-oriented programming)

When to Use What?

Use Invasive Extension
- for non-foreseen extensions that should be integrated
to develop aspect-orientedly
- to adapt without delegation

Use Inheritance
- for consistent side-effect free composition

Use Delegation
- for dynamic variation
### Inheritance Composes a Superclass with a Mixin

A *mixin* is a class (i.e., a set of features) by which a superclass can be extended to derive a subclass. The mixin class itself is final, i.e., cannot be subclassed.

Extension can be used for inheritance (extend by mixin)

**inheritance** :=
- copy first superclass
- extend with methods and attributes of second superclass (i.e., the mixin)

### Invasive Connections of COTS components

**Client**

**Library**

Blackbox connection with glue code

**Invasive Connection**

### Connect COTS components

**import** book_store

**public class** Customer

**public** Customer (**String** server)

```java
// allocate new server:
book_store bs = get_book_store(server);

// call the services:
bs.select(...);
bs.buy(...);
```

### Invasive Document Composition for XML

**Example: List Entry/Exit**

- `<UL>`
- `<LI>`...
- `<LI>`...
- `</UL>`

Hooks are given by the XML schema

**<UL>**

- `<LI>`...
- `<LI>`...
- `<LI>`...
- `<LI>`...
- `</UL>`

### Hook Manipulation for XML

```java
XMLcomponent.findHook("ListEntry").extend("<LI>... </LI>"téxy);
XMLcomponent.findHook("ListExit").extend("<LI>... </LI>"téxy);
```

### Composers can be used for AOP

*Complex composers distribute aspect fragments over core fragments*

**Distributors** extend the core

**Distributors** are more complex operators, defined from basic ones
Different Forms of Grayboxes

Shades of Gray

Invasive Composition and Information Hiding

Invasive Composition modifies components at well-defined places during composition.
Less information hiding than in blackbox approaches
More information hiding than in whitebox approaches (such as source-level OOP)
This leads to graybox components.
The gray-ness (degree of information hiding) can be varied.

Refactoring is a Whitebox Operation

Refactoring works directly on the AST/ASG.
Similar: Source-level program transformations
Attaching / removing / replacing fragments anywhere
Whitebox reuse

Refactoring builds on transformation of abstract syntax trees

Refactoring
Class A
Class B
New Class AB

Invasive Composition Builds On Transformation Of Implicit Hooks

Composer
Invasively transformed code

Invasively transformed code

Modifying Implicit Hooks is a Light-Gray Operation

Aspect weaving and view composition works on implicit hooks (= join points)
Set of implicit hooks = implicit composition interface
Parameterization as Darker-Gray Operation

Templates (generic types) work on declared hooks
Declared composition interface

Composition with declared hooks
Refactorings
Transformations

Invasive Composition Builds On Transformation on Declared Hooks

Composer
Invasively transformed code

Systematization Towards Graybox Component Models

Composition with declared hooks
Composition with implicit hooks
Refactorings
Transformations

Composition and Functional Interfaces

Composition vs Functional Interfaces

Invasive Composition removes Composition Interfaces
**Connect COTS components**

```java
import book_store;
public class Customer {
    public Customer (String server) {
        // allocate new server:
        book_store bs = get_book_store(server);
        // call the services:
        bs.select(...);
        bs.buy(...);
    }
}
```

**Functional Interfaces are Generated from Composition Interfaces**

```java
import book_store;
import org.omg.CORBA.*;
public class Customer extends CORBA.client {
    public Customer (String server) {
        // initialize CORBA:
        ORB orb = ORB.init(args);
        // allocate new server:
        book_store bs = orb.string_to_object(server);
        // call the services:
        bs.select(...);
        bs.buy(...);
    }
}
```

**Invasive Software Composition as Composition System**

- **Component Model**
- **Composition Technique**
- **Composition Language**

**Invasive Composition: Component Model**

- Graybox components instead of black box ones
- Composition interfaces with declared hooks
- Implicit composition interfaces with implicit hooks
- The composition programs produce the functional interfaces
- Resulting in efficient systems, because superfluous functional interfaces are removed from the system
- Content: source code
- Binary components also possible, poorer metamodel
- Aspects are just a new type of component
- Fragment-based parameterisation of components

**Invasive Composition: Composition Technique**

- Adaptation and glue code: good, composers are program transformers and generators
- Aspect weaving
  - Supported
  - No special languages required
- Extensions:
  - Hooks can be extended
  - Metamodeling employed
- Not yet scalable to run time

**Invasive Composition: Composition Language**

- Various languages can be used
  - Imperative (e.g., Java in COMPOST) vs. declarative
- Product quality improved by metamodel-based typing of compositions
- Metacomposition possible
  - Architectures can be described in a standard object-oriented language and reused
- An assembler for composition
  - Other, more adequate composition languages can be compiled
What Have We Learned

Conclusions for ISC

- Fragment-based composition technology
- Graybox components
- Producing tightly integrated systems
- Components have a composition interface
  - From the composition interface, the functional interface is derived
  - Composition interface is different from functional interface
  - Overlaying of classes (role model composition)

Unification of Software Development Techniques

- With the uniform treatment of declared and implicit hooks, several technologies can be unified:
  - Generic programming
  - Connector-based programming
  - Refactorings
  - Inheritance-based programming
  - View-based programming
  - Aspect-based programming

Invasive Composition as Composition System