46. Invasive Software Composition (ISC)

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de
Version 14-0.2, June 14, 2014

1. Invasive Software Composition - A Fragment-Based Composition Technique
2. What Can You Do With Invasive Composition?
3. Universally Composable Languages
4. Functional and Composition Interfaces
5. Different forms of grey-box components
6. Evaluation as Composition Technique
Obligatory Literature

- ISC book Chap 4
- www.the-compost-system.org
- www.reuseware.org
Other References


Invasive Software Composition

- Adds a full-fledged composition language to generic and view-based programming
- Combines architectural systems, generic, view-based and aspect-oriented programming
46.1. Invasive Software Composition - A Fragment-Based Composition Technique
Software Composition

Component Model

Composition Technique

Composition Language
**Invasive Software Composition**

Invasive software composition parameterizes and extends fragment components at change points (hooks and slots) by transformation.

- A **fragment component** (snippet components) is a fragment group (fragment container, fragment box) with a composition interface of change points.

- A fragment component is a uniform container for:
  - A plain fragment
    - a class, a package, a method
  - A generic fragment (group)
  - A fragment group
    - an advice or an aspect
    - a composition program
The Component Model of Invasive Composition

**Change points** of a fragment component are fragments or positions, which are subject to change.

- Fragment components have change points
- A *change point* can be
  - An *extension point* (hook)
  - A *variation point* (slot)
  - A *query point* (out port)
- Example:
  - Extension point: method entries/exits
  - Variation point: Generic parameters
Hooks

- A **hook (extension point)** is given by the component's language
- Hooks can be implicit or explicit (declared)
  - We draw implicit hooks *inside* the component, at the border
- Example: Method Entry/Exit
- Between hooks and their positions in the code, there is a **hook-fragment mapping**

```
m (){  
  abc..  
  cde..  
}  
```
A Hook can Relate to Many Code Points

► A hook can relate to many code points (1:n-hook-fragment mapping)
► Example:
  ► Method Entry refers to a code point at the beginning the method
  ► Method Exit refers to n code points before return statements

```java
boolean m (){
    abc..
    cde..
    if (cond) {
        return true;
    } else {
        return false;
    }
}
```
**Slots (Declared Hooks)**

- A slot is a variation point of a component, i.e., a code parameter.
- Slots are most often declared, i.e., declared or explicit hooks, which must be declared by the component writer.
  - They are implicit only if they designate one single program element in a fragment.
  - We draw slots as crossing the border of the component.
- Between slots and their positions in the code, there is a slot-fragment mapping.
Invasive Software Composition queries, parameterizes and extends fragment components at implicit and declared change points (hooks and slots) by transformation.

An invasive composition operator treats declared and implicit hooks uniformly.
**The Composition Technique of Invasive Composition**

- A composer (composition operator) is a static metaprogram (program transformer)
Object-Oriented Metamodeling of Composers

- In the following, we assume an object-oriented metamodel of fragment components, composers, and composition languages.
- The COMPOST library [ISC] has such a metamodel (in Java)
- Composers work on Composables (Changepoints or Boxes)
Bind Composer Universally Parameterizes Fragment Components

• Like in BETA, for uniformly generic components

```
Box component = readBoxFromFile("m.java");
component.findHook("mod").bind("synchronized");
component.findHook("mid").bind("f();");
```
Slot Binding State Diagram

Unbound Slot

(Fully) Bound Slot

Bind with fragment

Remove fragment
Extend Operator Universally Extends the Fragment Components

```
component.findHook(“MethodEntry“).extend(“print(“enter m”);”);
component.findHook(“MethodExit“).extend(“print(“exit m”);”);
```

```
m (){  
    abc..  
cde..  
}  

m (){  
    print(“enter m”);  
    abc..  
cde..  
    print(“exit m”);  
}  
```
**Merge Operator Provides Universal Symmetric Merge**

- The **Extend** operator is asymmetric, i.e., extends hooks of a fragment component with new fragment values.
- Based on this, a symmetric **Merge** operator can be defined:
  \[
  \text{merge(Component C1, Component C2) := extend(C1.list, C2.list)}
  \]
  - where list is a list of inner components, inner fragments, etc.

- Both extend and merge work on fragments:
  - Extend works on all collection-like language constructs
  - Merge on components with collection-like language constructs
Hook Extension State Diagram

Unbound Hook

Extend with fragment

Remove fragment

Extended Hook

Remove fragment

Extend with fragment
Query Operator Delivers Fragments out of the Fragment Component

```java
int findoutAge(Person p)
{
    if (p == null) return 19;
    abc..
    result = cde..
    if (result == 0) return 10;
}
```

```
{ if (p == null) return 19;
    if (result == 0) return 10;
}
```

"{ if (p == null) return 19;
    if (result == 0) return 10;}" \(\Leftarrow\) component.findHook("ContractQuery").query();
## Basic Composition Operators

<table>
<thead>
<tr>
<th>Approach</th>
<th>Composables</th>
<th>Composers</th>
<th>Variation/Extension points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>extend</td>
<td></td>
<td>Implicit member list</td>
</tr>
<tr>
<td></td>
<td>merge</td>
<td></td>
<td>Open definitions</td>
</tr>
<tr>
<td>Slots</td>
<td>bind</td>
<td></td>
<td>Variation point</td>
</tr>
<tr>
<td></td>
<td>unbind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hooks</td>
<td>Extend</td>
<td></td>
<td>Extension point</td>
</tr>
<tr>
<td>Fragment out port</td>
<td>query</td>
<td></td>
<td>Query point</td>
</tr>
</tbody>
</table>
Applied to Classes, Invasive Extension Integrates Feature Groups

- The Extend operator integrates feature groups and roles into classes (role merge)
  - because a feature group can play a role
- The semantics of invasive extension lies between inheritance and delegation
Invasive composition unifies generic programming (BETA) and view-based programming (merge composition operators)

- By providing bind (parameterization) and extend for all language constructs

```
/* @genericMYModifier */ public print() {
    // <<MethodEntry>>
    if (1 == 2)
        System.out.println("Hello World");
    // <<MethodExit>>
    return;
    else
        System.out.println("Bye World");
    // <<MethodExit>>
    return;
}
```

Hook h = methodComponent.findHook("MY");
if (parallel)
    h.bind("synchronized");
else
    h.bind(" ");
methodComponent.findHook("MethodEntry").bind("");
methodComponent.findHook("MethodExit").bind("");

synchronized public print () {
    if (1 == 2)
        System.out.println("Hello World");
    return;
    else
        System.out.println("Bye World");
    return;
}
You Need Invasive Composition

- Adaptation of static relations
  - Inheritance relationship: multiple and mixin inheritance
  - Delegation relationship: When delegation pointers have to be inserted
  - Import relationship
  - Definition/use relationships (adding a definition for a use)
  - When templates have to be expanded in a type-safe way

- When physical unity of logical objects is desired
  - Invasive extension and merges roles into classes
  - No splitting of roles, but integration into one class

- When the resulting system should be highly integrated
  - When views should be integrated constructively
When To Use What?

- Deploy Inheritance
  - for consistent side-effect free composition

- Deploy Delegation
  - for dynamic variation
  - Suffers from object schizophrenia

- Deploy Invasive Extension
  - for views: non-foreseen extensions that should be *integrated*
  - For foreseen views with extension points
  - For aspect-oriented development with cross-cutting
  - to adapt without delegation
Composition Programs

Basically, every language may act as a composition language, if its basic operators are bind and extend.

Imperative languages: Java (used in COMPOST), C, ..
Graphical languages: boxes and lines (used in Reuseware)
Functional languages: Haskell
Scripting languages: TCL, Groovy, ...
Logic languages: Prolog, Datalog, F-Datalog
Declarative Languages: Attribute Grammars, Rewrite Systems
Homogeneous Composition Systems

• A composition system is called **homogeneous**, if it employs the same composition language and component language.
• Otherwise, it is called **heterogeneous**.
• In a homogeneous composition system, metacomposition is staged composition.
• A **point-cut language (cross-cut language)** is a form of composition language.
46.2. What Can You Do With Invasive Composition?
**Invasive Composition**

Adds a full-fledged composition language to generic and view-based programming.

Combines architectural systems, generic, view-based and aspect-oriented programming.
Universally Generic Programming

- ISC is a fully generic approach
- In contrast to BETA, ISC offers a full-fledged composition language
- Generic types, modifiers, superclasses, statements, expressions,...
- Any component language (Java, UML, ...)

```java
class SimpleList {
    genericTType elem;
    SimpleList next;
    genericTType getNext() {
        return next.elem;
    }
}
```
Universal Constructive View Programming

• ISC is a uniform and universal view-programming approach
  • The Extend operator realizes open definitions for all language constructs: methods, classes, packages
  • The Merge operator realizes symmetric composition for all language constructs
• Additionally, ISC offers a full-fledged composition language
Invasive Connections

- In contrast to ADL, ISC offers invasive connections [AG00]
- Modification of inheritance relations possible

Blackbox connection with glue code
Invasive Architectural Programming

[ISC] shows how invasive connectors achieve tightly integrated systems by embedding the glue code into senders and receiver components
Separation of Topological from Transfer Aspect

Connection A

Connection B
Port Binding State Diagram

Topologically Bound Port

Unbound Port

(Fully) Bound Port

Topological Connector

Full Connector

Unlinker

Deconnector

Transfer Deselector

Transfer Selector

Prof. U. Aßmann, CBSE
Gate Objects: Glue Separate
Invasive Connection

Embedding communication gate methods into a class
Invasive Connection

Embedding glue code into sender methods

Sender

<< ClassBox >>

<< MethodBox >>

Pack Arguments

Send

Receiver

<< ClassBox >>

<< MethodBox >>

Unpack Arguments

Receive
Universal Inheritance and Mixins

- Extension can be used for inheritance, mixins
- In contrast to OO languages, ISC offers tailored inheritance operations, based on the extend operator
- Mixins can be used to simulate static roles

- **inheritance** :=
  - copy first super class
  - extend with second super class
- **mixin_inheritance** :=
  - Bind superclass reference
**Mixin Inheritance Works Universally for Languages that don't have it**

- Invasive composition can model mixin inheritance uniformly for all languages
  - e.g., for XML
- inheritance :=
  - copy first super document
  - extend with second super document
Invasive Document Composition for XML

- Invasive composition can be used for document languages, too [Hartmann2011]
- Example List Entry/Exit of an XML list
- Hooks are given by the Xschema

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

List.entry → <UL>
              <LI>... </LI>
              <LI>... </LI>

List.exit → </UL>
Hook Manipulation for XML

<UL>
  <LI>... </LI>
  <LI>... </LI>
  <LI>... </LI>
  <LI>... </LI>
</UL>

XMLcomponent.findHook(“ListEntry“).extend(“<LI>... </LI>”);
XMLcomponent.findHook(“ListExit“).extend(“<LI>... </LI>”);
Universal Weaving for AOP (Core and Aspect Components)

- Complex composers distribute aspect fragments over core fragments
- *Distributors* extend the core
- Distributors are more complex operators, defined from basic ones
- Static aspect weaving can be described by distributors, because hooks are static
  - ISC does not have a dynamic joinpoints
  - Crosscut specifications can be interpreted
Distributors are Composition Programs

Core (Algorithm)

Debugging Aspect

Persistency Aspect

Distributor

Persistency

Debugging
Distributors Weave Relations between Core and Aspect
Invasive Model Composition with Reuseware

Editor specification
46.3 Reuseware, a Meta-Composition System to Build Composition Systems

Universally Composable Languages with for universal genericity and extension
**Universally Composable Languages**

**Universally composable**: A language is called *universally composable*, if it provides universal genericity and universal extensibility
- The language has to be enriched with an invasive component model

**Reuse language**: Given a metamodel of a core language L, a metamodel of a universally composable language can be generated (the Reuse-L)
- The Reuse language describes the composition interfaces of the components, an important part of the component model
- The component model can be composed by metamodel composition

**Slot and Hook model**: added (or even generated) from the core language metamodel
- realizes universal composability by defining *slots* and *hook constructs*, one for each construct in the core language
The core and the reuse language have two levels:

- Language-neutral knowledge...
- Language-specific knowledge...

Structure of a Universally Composable Language

Core

Reuse

UL

UH

Core

L

Slots for L

Hooks for L

Component model

Names

Reuse-L
Reuse-UML, a Universally Composable Language

- an extension of UML with slot and hook model
Reuse-XML, a Universally Composable Language

- an extension of XML with slot and hook model

Component model

Language-neutral knowledge ...

Language-specific knowledge ...

Core

UL

Core XML

Het

Reuse

H

Slots for XML

Hooks for XML

Names

Reuse-XML
Reuse-XML, a Universally Composable Language

- an extension of XML with slot and hook model

Language-neutral knowledge ...

XML-specific knowledge ...

Reuse-XML Component model

Core XML

UL

UH

Slots for XML

Hooks for XML

Names
The Reusewair Technology

- [www.reuseware.org](http://www.reuseware.org) (Phd of Jakob Henriksson, 2008)
- *Reusewair* was the world-wide first technology and tool to build reuse languages (component models) and composition systems for *text-based languages*
  - Grammar-based (EBNF)
  - Generic strategy for applying composition operators on components (based on Design Pattern Visitor)
  - Composition tools, type checker, come for free
The Reuseware Tool

- www.reuseware.org (Phd of Jendrik Johannes, 2010)
- **Reuseware** is a tool to build reuse languages (component models) and composition systems for *text-based* and *diagramm-based* languages
  - Eclipse-based
  - metamodel-controlled (metalanguage M3: Eclipse e-core)
  - Plugins are generated for composition
  - Composition tools come for free
  - Textual, graphic, XML languages
- Framework instantiation is supported for variation and extension
- Jobs open!
The SkAT Tool

- Phd of Sven Karol, 2014
- SkAT is a tool to build reuse languages (component models) and composition systems for text-based and diagram-based languages
  - Based on Reference-Attribute-Grammar (RAG)
  - And metamodels (metalanguage M3: Eclipse e-core)
  - Declarative composition constraints control the composition
  - Composition tools come for free
  - Textual, graphic, XML languages
- Framework instantiation is supported for variation and extension
- Jobs open!
46.4. Staging of Composition: Composition and Functional Interfaces
Composition vs Functional Interfaces

Composition interfaces contain hooks and slots static, based on the component model at design time.

Functional interfaces are based on the component model at run time and contain slots and hooks of it.
Functional Interfaces are Generated from Composition Interfaces

2-stage process

Composition Interface (Boxes with Declared Hooks)

Functional Interface (Classes or Modules with Methods)
Execution of a Composition Program

A composition program transforms a set of fragment components step by step, binding their composition interfaces (filling their slots and hooks), resulting in an integrated program with functional interfaces.
The Stages of ISC

- Produces code from fragment components by parameterization and expansion
- The run-time component model fits to the chip
Standard COTS models are just models for binary code components.
Component Models on Different Levels in the Software Process

Another stage can be introduced by XML model composition from which Java code is generated [Johannes 10]
**Staging**

- With a universal composition system as Reuseware, stages can be designed (stage design process)
- For each stage, it has to be designed a universally composable language:
  - component models
  - composition operators
  - composition language
  - composition tools (editors, well-formedness checkers, component library etc.)
46.5) Different Forms of Greyboxes (Shades of Grey)
Invasive Composition and Information Hiding

- Invasive Composition modifies components at well-defined places during composition
  - There is less information hiding than in blackbox approaches
  - But there is...
  - ... that leads to greybox components
Refactoring is a Whitebox Operation

- Refactoring works directly on the AST/ASG
- Attaching/removing/replacing fragments
- Whitebox reuse
Modifying Implicit Hooks is a Light-Grey Operation

- Aspect weaving and view composition works on implicit hooks (*join points*)
- Implicit composition interface

Composition with implicit hooks

Refactorings Transformations
Parameterization as Darker-Grey Operation

- Templates work on *declared hooks*
- *Declared composition interface*
Systematization Towards Greybox Component Models

Composition with declared hooks

Composition with implicit hooks

Refactorings Transformations
Refactoring Builds On Transformation Of Abstract Syntax
Invasive Composition Builds On Transformation Of Implicit Hooks

Composer

Invasively transformed code
Invasive Composition Builds On Transformation on Declared Hooks

Composer

Invasively transformed code
46.6 Invasive Software Composition as Composition Technique
**Invasive Composition: Component Model**

- Fragment components are graybox components
  - 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 special type of component

- Fragment-based parameterisation *à la* BETA
  - Type-safe parameterization on all kinds of fragments
**Invasive Composition: Composition Technique**

- Adaptation and glue code: good, composers are program transformers and generators
- Aspect weaving
  - Parties may write their own weavers
  - No special languages
- Extensions:
  - Hooks can be extended
  - Soundness criteria of lambdaN still apply
  - Metamodelling employed
- Not yet scalable to run time
Composition Language

- Various languages can be used
- 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
Conclusions for ISC

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

- COMPOST framework showed applicability of ISC for Java
  - (ISC book)
- Reuseware Composition Framework extends these ideas
  - For arbitrary grammar-based languages
  - For metamodel-based languages
- http://reuseware.org
Invasive Composition as Composition System

Component model
- Source or binary components
- Greybox components
- Composition interfaces with declared an implicit hooks

Composition technique
- Algebra of composition operators
- Uniform on declared and implicit hooks
- Complex composition operators can be defined by users

Standard Language
Composition language
What Have We Learned

With the uniform treatment of declared and implicit hooks and slots, several technologies can be unified:

- Generic programming
- Connector-based programming
- View-based programming
  - Inheritance-based programming
- Aspect-based programming
- Refactorings
The End