32. Practical Refactoring-Based Framework Upgrade with Comeback!

Ilie Șavga, Michael Rudolf, Sebastian Götz, Uwe Aßmann
20.10.2008   GPCE’08: Nashville, Tennessee
Restructuring a Sign

John Thompson, hatter, makes and sells hats for ready money.

Benjamin Franklin, as cited in Kerievsky, 2004
"A large program that is used undergoes continuing change or becomes progressively less useful."

_Lehman's first law_

Lehman and Belady, p. 250

"As a large program is continuously changed, its complexity ... increases unless work is done to maintain or reduce it."

_Lehman's second law_

Lehman and Belady, p. 253

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

Behavior-preserving yet structural-improving

Fowler et al., 1999, xvi

Opdyke, 1992; Roberts, 1999
Software Frameworks and Plugins

A software framework is a software component that embodies a skeleton solution for a family of related software products and is instantiated by modules containing custom code (plugins).

Example inspired by Demeyer et al., 2005

Framework Version 1

```
Node
String getName ()
void broadcast (String msg)
```

LAN

```
void enter (Node node)
{
    node. broadcast(node.getName() + " enters the network");
}
```

Plugin Version 1

Examples inspired by Demeyer et al., 2005

Johnson and Foote, 1998
Refactorings comprise 75-97% of application-breaking API changes.

Dig and Johnson, 2005; Dagenais and Robillard, 2008; Schäfer et al., 2008; Şavga et al., 2008;
Adapter is a well-known design pattern to bridge a software component mismatch. 

GoF, 1995;

“The most structured way to deal with component evolution and upgrading, which is likely to result in new mismatches, arguably is by applying adaptation techniques.”

Becker et al., 2004
Comebacks: Refactoring Inverses on Adapters

Version 1

Version 2

Framework

Adapters

Plugins

TU Dresden, 20.10.2008

Ilie Șavga, Michael Rudolf, Sebastian Götz and Uwe Aßmann
Comebacks: Refactoring Inverses on Adapters

Version 1

- Refactorings History
  - input

Version 2

- Adapter Generator
  - generate

Version 3

- Framework
- Adaptation Layers
- Plugins

Evolution
Adapter Design Pattern (Object Version)

```
use Adapter

Client

Target

Adaptee

specificRequest()

Adapter

request() {
    adapter.specificRequest();
}

Target

request()

Client

target.request();
```

GoF, 1995
Refined Adapter Design Pattern (Object Version)

```
use Adapter

Client
request()

target.request();

Target
request()

request() { 
adaptee.adapter.specificRequest();
}

AdapteeImpl
request()

Adaptee
-specificRequest()

adapted

GoF, 1995
```
The Comeback! Black-Box Interface Adapter

Framework Version 2

<<AdapteeImpl>>
TokenPacket
Node getCreator()

<<Adaptee>>
IPacket
Node getCreator()

Adaptation Layer Version 1

<<Target>>
IPacket
Node whoCreated()

<<Adapter>>
IPacketAdapter
Node whoCreated()

Plugin Version 1

<<Client>>
LAN
packet.whoCreated();

adapted
use

use

use
The Comeback! White-Box Interface Adapter

Framework Version 2

<<Client>>
PrintServer

print (packet.getCreator());

use

<<Target>>
IPacket

Node getCreator()

Adaptation Layer Version 1

<<Adaptee>>
IPacket

Node whoCreated()

adapted

<<Adapter>>
IPacketAdapter

Node getCreator()

Plugin Version 1

<<AdapteeImpl>>
LAN

packet.whoCreated();
A White-Box Framework (Re-use by Inheritance)

Framework Version 1

PrintServer

// encryption info
String getDescription()

use

Node
debug (Node n){
    Log.write(n.getName());
} String getName()

use

Plugin Version 1

LAN

Node n = new SecureNode();
...
String getName()

use

SecureNode

// encryption info
String getDescription()
Fragile Base Class Problem: Method Capture

Framework Version 2

PrintServer
debug (Node n){
    Log.write(n.getDescription());
}

Node
String getName()
String getDescription(){
    this.getName();
}

Plugin Version 1

LAN
Node n = new SecureNode();
...
utils.debug(n);

SecureNode
//encryption info
String getDescription()

Steyaert et al, 1996; Mikhajlov and Sekerinski, 1998
Method Capture Solved with Comback! White-Box Class Adapter

Framework Version 2

<<Client>>
PrintServer

Debug(Node n){
Log.write(n.getDescription());
}

call

<<Target>>
Node

String getName()
String getDescription(){
this.getName();
}

Adaptation Layer Version 1

<<Adaptee>>
Node

String getName(){
adapter.getName();
}

adapted

<<Adapter>>
NodeAdapter

//run-time method dispatch

Plugin Version 1

<<AdapteeImpl>>
SecureNode

//encryption info
String getDescription()
Exhaustive API Adaptation: Management of Adapters

Call Framework -> Plugins

Call Plugins -> Framework

Adapter Cache

Framework

Adaptation Layers

Plugins
**Framework Version 1**

```java
public class PacketFactory{
    private Hashmap packetHistory;
    public Packet createPacket(PacketType type, INode creator){
        Packet pckt = new Packet(type);
        pckt.setCreator(creator);
        packetHistory.save(pckt);
        return pckt;
    }
}
```

**Plugin Version 1**

```java
SecureNode node = new SecureNode();
PacketType pt = PacketFactory.getEncryptedPacketType();
...
Packet packet = PacketFactory.createPacket(pt, node);
```
Wrapping and Unwrapping of Parameter Objects in the Adaptation Layer

New Framework

```java
public class PacketFactory{
    private HashMap packetHistory;
    public Packet createPacket(PacketType type, INode creator){
        Packet pckt = new Packet(type);
        pckt.setCreator(creator);
        packetHistory.save(pckt);
        return pckt;
    }
}
```

Adaptation Layer

Old Plugin

```java
SecureNode node = new SecureNode();
PacketType pt = TypeFactory.getEncryptedPacketType();
...
Packet packet = PacketFactory.createPacket(pt, node);
```
Tool Validation: the ComeBack!

ComeBack! homepage: http://comeback.sf.net
Side-by-Side Plugin Execution

**Version 1**
- **Refactoring History**
- **Adapter Generator**

**Version 2**
- **Application**

**Version 3**
- **Framework**
- **Adaptation Layers**
- **Plugins**

Evolution
Java-based frameworks: SalesPoint and JHotDraw

- application-driven refactoring detection
- no backward compatibility concern but 85%
- comebacks specified and executed, remaining changes adapted manually

**Effectiveness**: all refactorings adapted

**Performance**: up to 6.5% overhead ≤

- static optimizations
- run-time optimizations
Challenges and Limitations

Adapter pattern limitations
• no field refactorings
• no comebacks for refactorings implying this
• limited recovery of deleted methods

Object structure assumptions
• abusive reflective calls
• default serialization

Non-available refactoring info
• querying Eclipse refactoring log
• investigating the use of CVS
Intrusive Refactoring-Based Adaptation

CatchUp!: intrusively adapting plugins
- refactoring record-and-replay on application sources
  + re-use of Eclipse refactoring info
  - requires plugin sources and implies new application release

ReBA: intrusively adapting frameworks
- compensating refactorings for combining old and new APIs
  + preserve object identities; low performance overhead; recovering deleted implementation
  - no prove of soundess

- (both): context-dependent (delete M and rename to M);
  no white-box adaptation (accidental overriding possible);
  Java-specific transformations
Conclusions

Comeback-based approach is rigorous and practical:
• refactorings treated as formal specification of syntactic change
• automatic and transparent API adaptation for most of application-breaking changes
• side-by-side plugin execution and fairly acceptable performance overhead (in tested applications)

At least, a short-term solution
Questions and Further Discussion
**Theorem 1.** *CbAddClass* is a comeback of *AddClass*.

**Proof.**
- Prop 1: *CbAddClass* is constructed using exactly one refactoring (*RemoveClass*) and, because it satisfies the preconditions of that refactoring (they are the same), behavior is preserved.

- Prop 2: The precondition of *RemoveClass* has to evaluate to true for the program changed by *AddClass*. Let the changed program be $P'$.

\[
P' \models (\text{IsClass}(\text{class}) \land \\
(\text{ClassReferences}(\text{class}) = \emptyset) \land \\
((\text{Subclasses}(\text{class}) = \emptyset) \lor \\
\text{IsEmptyClass}(\text{class})))
\]

\[\iff (P' \models \text{IsClass}(\text{class})) \land \]
\[\quad (P' \models (\text{ClassReferences}(\text{class}) = \emptyset)) \land \]
\[\quad ((P' \models (\text{Subclasses}(\text{class}) = \emptyset)) \lor \\
\quad (P' \models \text{IsEmptyClass}(\text{class})))
\]

\[\iff T \land T \land (T \lor T) \iff T
\]

The last derivation step is performed using the assertions transformed by the *post* of *AddClass*.

- Prop 3: The precondition of *AddClass* has to evaluate to true for the program changed by *AddClass* and *RemoveClass*. Let the changed program be $P''$.

\[
P'' \models (\text{IsClass}(\text{superclass}) \land \\
\neg \text{IsClass}(\text{class}) \land \forall c \in \text{subclasses}. \\
(\text{IsClass}(c) \land (\text{Superclass}(c) = \text{superclass})))
\]

\[\iff (P'' \models \text{IsClass}(\text{superclass})) \land \]
\[\quad (P'' \not\models \text{IsClass}(\text{class})) \land \forall c \in \text{subclasses}. \\
\quad ((P'' \models \text{IsClass}(c)) \land \\
\quad (P'' \models (\text{Superclass}(c) = \text{superclass})))
\]

\[\iff T \land T \land T \land T \iff T
\]

The last derivation step is performed using the *post* of *AddClass* composed with the preconditions and *post* of *RemoveClass*. □
**CbPushDownMethod**\( (\text{class}, \text{subclass}, \text{method}) \) is defined as a set of refactorings executed in two steps:

1. **AddMethod**(\( \text{class}, \text{method}, \text{Method}(\text{subclass}, \text{method}) \)): Add to the class \( \text{class} \) the method \( \text{method} \), which is semantically equivalent to the method with the same name defined in \( \text{subclass} \).
2. **RemoveMethod**(\( \text{subclass}, \text{method} \)): Remove method from \( \text{subclass} \).

The precondition of **CbPushDownMethod**:

1. \( \text{IsClass}(\text{class}) \)
2. \( \text{IsClass}(\text{subclass}) \)
3. \( (\text{Superclass}(\text{subclass}) = \text{class}) \)
4. \( (\text{Superclass}(\text{Delegatee}(\text{subclass})) = \text{Delegatee}(\text{class}) ) \)
5. \( \text{DefinesSelector}(\text{subclass}, \text{method}) \)
6. \( \neg \text{DefinesSelector}(\text{class}, \text{method}) \)
7. \( (\neg \text{UnderstandsSelector}(\text{class}, \text{method}) \lor (\text{LookUpMethod}(\text{class}, \text{selector}) \equiv \text{Method}(\text{subclass}, \text{method}))) \)
Theorem 2. \textit{CbPushDownMethod is a comeback of PushDownMethod.}

\textit{Proof.}  
• Prop 1. For each used refactoring its precondition is satisfied. For \textit{ChangeType}: type safeness property is preserved by assertions 1–4 of the \textit{CbPushDownMethod} precondition. For \textit{AddMethod}: the newly added method is not yet defined locally and is semantically equivalent to any overridden function (satisfied by assertions 5–7). For \textit{RemoveMethod}: the \textit{subclass} overrides a semantically equivalent \textit{method} from \textit{class} after executing \textit{AddMethod} in the previous step, so \textit{method} can be safely removed from \textit{subclass}. Since the preconditions of all used refactorings are satisfied, behavior is preserved.

• Prop 2. The \textit{post} of \textit{PushDownMethod} (not shown) reflects the appearance of the method in the subclass and its removal from the superclass. It can be shown that the precondition of \textit{CbPushDownMethod} is satisfied by the program changed by \textit{PushDownMethod}.

• Prop 3. The assertions of the precondition of \textit{PushDownMethod} ensure that: \textit{class} and \textit{subclass} exist; \textit{method} is defined in \textit{class} and not redefined in \textit{subclass}; no private variables of \textit{class} are accessed from \textit{method}. The first two assertions are also assertions of the \textit{CbPushDownMethod} precondition and are not changed (i.e., remain satisfied) after its execution. The definition of \textit{method} in \textit{class} and not in \textit{subclass} is implied by the execution of \textit{AddMethod} and \textit{RemoveMethod}. The last assertion is satisfied by keeping the access mode of the delegation field protected (see Step 2 of \textit{AddAdapter}). \hfill \Box
CbExtractSubclass(class, subclass, method) is defined as:

1. CbPushDownMethod(class, subclass, method)
2. CbAddClass(subclass, class, Subclasses(class))

The precondition of CbExtractSubclass is a conjunction of the precondition of CbPushDownMethod and that of CbAddClass evaluated with regard to the post definition of CbPushDownMethod.

**Theorem 3.** CbExtractSubclass is a comeback of ExtractSubclass.

**Proof.** As CbExtractSubclass is defined as a sequence of two comebacks CbPushDownMethod and CbAddClass, its three comeback properties can be proven by induction on the previous two proofs. \(\square\)
AddAdapter(class)

1. AddClass(Delegatee(class), Delegatee(Superclass(class)), ∅): Create an empty class with the unique name returned by the renaming function. Its superclass name is the value of the renaming function for the superclass of class.
2. AddInstanceVariable(class, DField(class), Delegatee(class)): Add a (protected) delegation variable to the class.
3. ∀v ∈ VariablesDefinedBy(class) \ {DField(class)}.
   MoveInstanceVariable(class, v, Delegatee(class)): Move all but the delegation variable of class to the class created in step 1.
4. ∀m ∈ {d|DefinesSelector(class, d)}.
   MoveMethod(class, m, DField(class), m): Move all methods, defined in class, to the class of its delegation variable. For each method, MoveMethod creates a method in the original class, which forwards to the moved method.


