Practical Refactoring-Based Framework Upgrade

Ilie Ţavga, 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
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).

Examples inspired by Demeyer et al., 2005

Johnson and Foote, 1998
Framework Refactoring Breaking a Plugin

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
Comebacks: Refactoring Inverses on Adapters

Version 1

Refactorings History

Adapter Generator

input

generate

Version 2

Version 3

Evolution

Plugin

Framework

Adaptation Layers

Plugins
Adapter Design Pattern (Object Version)

```java
Adapter

    request() {
        adapter.specificRequest();
    }

Adaptee

    specificRequest()

Target

    request() {
        target.request();
    }

Client

    use

GoF, 1995
```
Refined Adapter Design Pattern (Object Version)

```java
use AdapteeImpl
  request();

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

Adaptee
  specificRequest();

Target
  request();

Client
  target.request();

GoF, 1995
Black-Box Interface Adapter

**Framework Version 2**

- **<<AdapteeImpl>>** TokenPacket
- **Node getCreator()**

**Adaptation Layer Version 1**

- **<<Target>>** IPacket
- **Node whoCreated()**

- **<<Adapter>>** IPacketAdapter
- **Node whoCreated()**

**Plugin Version 1**

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

---

**use**

**adapted**
White-Box Interface Adapter

Framework Version 2

<<Client>>
PrintServer

use

print (packet.getCreator());

<<Target>>
IPacket

Node getCreator()

Adaptation Layer Version 1

<<Adaptee>>
IPacket

Node whoCreated()

<<Adapter>>
IPacketAdapter

Node getCreator()

Plugin Version 1

<<AdapteeImpl>>
LAN

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

### Framework Version 1

**PrintServer**

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

**Node**

- `getName()`
- `getDescription()`

### Plugin Version 1

**LAN**

```java
Node n = new SecureNode();
...
utils.debug(n);
```

**SecureNode**

- `getDescription()`
- `//encryption info`

---

Ilie Şavga, Michael Rudolf, Sebastian Götz and Uwe Aßmann
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()
Method Capture Solved: White-Box Class Adapter

```
<<Client>>
PrintServer

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

<<AdapteeImpl>>
SecureNode

// encryption info
String getDescription()

<<Target>>
Node

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

<<Adapter>>
NodeAdapter

// run-time method dispatch
String getName()
  adapter.getName();

<<Adaptee>>
Node

String getName()
  adapter.getName();
```
Exhaustive API Adaptation

Framework -> Plugins

Plugins -> Framework

Adapter Cache

Framework

Adaptation Layers

Plugins
Framework<->Plugin Object Exchange

**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

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

Refactorings History

Adapter Generator

input

generate

Version 2

Version 3

Evolution

Framework

Application

Adaptation Layers

Plugins

Refactorings History

Adapter Generator

input

generate

Application
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 \(\leq\)

- static optimizations
- run-time optimizations
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
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 soundness

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

Henkel and Diwan, 2005
Dig at al., 2008
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
References


Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts.. Addison-Wesley, 1999.


GoF: E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.


SalesPoint homepage. www-st.inf.tu-dresden.de/SalesPoint/v3.1/index.html


Theorem 1. \textit{CbAddClass} is a comeback of \textit{AddClass}.

\textbf{Proof.} • Prop 1: \textit{CbAddClass} is constructed using exactly one refactoring \textit{(RemoveClass)} and, because it satisfies the preconditions of that refactoring (they are the same), behavior is preserved.

• Prop 2: The precondition of \textit{RemoveClass} has to evaluate to true for the program changed by \textit{AddClass}. Let the changed program be $P'$.

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

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

$\Leftrightarrow$ \[
(T \land T \land (T \lor T)) \Leftrightarrow T
\]

The last derivation step is performed using the post of \textit{AddClass}.

• Prop 3: The precondition of \textit{AddClass} has to evaluate to true for the program changed by \textit{AddClass} and \textit{RemoveClass}. Let the changed program be $P''$.

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

\[
(\text{IsClass}(c) \land (\text{Superclass}(c) = \text{superclass}))
\]

$\Leftrightarrow$ \[
(P'' \models \text{IsClass}(\text{superclass})) \land
(P'' \not\models \text{IsClass}(\text{class})) \land \forall c \in \text{subclasses.}
\]

\[
((P'' \models \text{IsClass}(c)) \land (P'' \models (\text{Superclass}(c) = \text{superclass})))
\]

$\Leftrightarrow$ \[
T \land T \land T \land T \Leftrightarrow T
\]

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

1. **AddMethod**\((\text{class, method, Method(subclass, method)})\):
   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**\((\text{subclass, method})\): Remove \(\text{method}\) from \(\text{subclass}\).

The precondition of **CbPushDownMethod**:

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

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

*Prop 2.* The post of 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 CbPushDownMethod is satisfied by the program changed by PushDownMethod.

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

\[\square\]
**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.
Create Adaptation Layer: AddAdapter

**AddAdapter**(\textit{class})

1. \textit{AddClass}(\textit{Delegatee}(\textit{class}), \textit{Delegatee}(\textit{Superclass}(\textit{class})), \emptyset): 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 \textit{superclass} of \textit{class}.

2. \textit{AddInstanceVariable}(\textit{class}, \textit{DField}(\textit{class}), \textit{Delegatee}(\textit{class})): Add a (protected) delegation variable to the class.

3. \(\forall v \in VariablesDefinedBy(\textit{class}) \setminus \{DField(\textit{class})\}\).

   \textit{MoveInstanceVariable}(\textit{class}, v, \textit{Delegatee}(\textit{class})): Move all but the delegation variable of \textit{class} to the class created in step 1.

4. \(\forall m \in \{d | DefinesSelector(\textit{class}, d)\}\).

   \textit{MoveMethod}(\textit{class}, m, \textit{DField}(\textit{class}), m): Move all methods, defined in \textit{class}, to the class of its delegation variable. For each method, \textit{MoveMethod} creates a method in the original class, which forwards to the moved method.


