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."

Fowler et al., 1999, xvi

Behavior-preserving yet structural-improving

Examples inspired by Demeyer et al., 2005

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).

Johnson and Foote, 1998

Framework Version 1

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

Plugin Version 1

```
LAN
  void enter(Node node)
  node.broadcast("enters the network")
```

examples inspired by Demeyer et al., 2005

"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."

Fowler et al., 1999, xvi

Behavior-preserving yet structural-improving

Examples inspired by Demeyer et al., 2005

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).

Johnson and Foote, 1998

Framework Version 1

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

Plugin Version 1

```
LAN
  void enter(Node node)
  node.broadcast("enters the network")
```

examples inspired by Demeyer et al., 2005
Framework Refactoring Breaking a Plugin

Plugin Version 1

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

Framework Version 1

Node

String getName() {
}

void broadcast(String msg) {
}

LAN

Adapter Version 1

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

Adapter Version 2

Node

String getName() {
}

void broadcast(String msg) {
}

Adapter is a well-known design pattern to bridge a software component mismatch.

Comebacks: Refactoring Inverses on Adapters

Version 1

Version 2

Adaptation Layers

Refactoring-Driven Adaptation

Adapter is a well-known design pattern to bridge a software component mismatch.

“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

GoF, 1995;

Dig and Johnson, 2005; Dagenais and Robillard, 2008; Schäfer et al., 2008; Şavga et al., 2008.

Refactorings comprise 75-97% of application-breaking API changes.
Adapter Design Pattern (Object Version)

use Adapter

request()

adapter.specificRequest();

Target

request()

Target

request()

Client

target.request();

Adaptee

specificRequest()

Adaptee

specificRequest()

AdapteeImpl

request()

GoF, 1995

Refined Adapter Design Pattern (Object Version)

use Adapter

request()

adapter.specificRequest();

Target

request()

Target

request()

Client

target.request();

Adaptee

specificRequest()

Adaptee

specificRequest()

AdapteeImpl

request()

GoF, 1995

The Comeback! Black-Box Interface Adapter

Framework Version 2

<<AdapteeImpl>>

TokenPacket

Node whoCreated();

<<Adaptee>>

IPacket

Node whoCreated();

<<Client>>

LAN

packet.whoCreated();

<<Target>>

IPacket

Node getCreator();

adapted

Adaptation Layer Version 1

<<Target>>

IPacket

Node whoCreated();

<<Target>>

IPacket

Node getCreator();

Plugin Version 1

<<Client>>

LAN

packet.whoCreated();

The Comeback! White-Box Interface Adapter

Framework Version 2

<<AdapteeImpl>>

PrintServer

print (packet.getCreator());

<<Adaptee>>

IPacket

Node whoCreated();

<<Client>>

LAN

packet.whoCreated();

<<Adaptee>>

IPacket

Node whoCreated();

<<Target>>

IPacket

Node getCreator();

adapted

Adaptation Layer Version 1

<<AdapteeImpl>>

PrintServer

print (packet.getCreator());

<<Adaptee>>

IPacket

Node whoCreated();

<<Target>>

IPacket

Node getCreator();

Plugin Version 1

<<AdapteeImpl>>

PrintServer

print (packet.getCreator());

<<Adaptee>>

IPacket

Node whoCreated();

<<Target>>

IPacket

Node getCreator();
A White-Box Framework (Re-use by Inheritance)

Framework Version 1

PrintServer

String getName()

Node

String getDescription()

Plugin Version 1

LAN

Node n = new SecureNode();

utils.debug(n);

SecureNode

// encryption info

Fragile Base Class Problem: Method Capture

Framework Version 2

PrintServer

String getName()

Node

String getDescription()

Plugin Version 1

LAN

Node n = new SecureNode();

utils.debug(n);

SecureNode

// encryption info

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());

<<Target>>

Node

Debug (Node n)

Log.write(n.getDescription());

<<Adapter>>

NodeAdapter

// run-time method dispatch

<<Adaptee>>

Node

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

Adaptation Layer Version 1

<<Adaptee>>

Node

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

<<Adapter>>

NodeAdapter

// run-time method dispatch

<<Adaptee>>

Node

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

Exhaustive API Adaptation: Management of Adapters

Call Framework -> Plugins

Call Plugins -> Framework

Framework

Adapter Cache

Adaptation Layers

Plugins
Framework<>Plugin Object Exchange

Plugin Version 1

SecureNode node = new SecureNode();
PacketType pt = PacketFactory.getEncryptedPacketType();
...
Packet packet = PacketFactory.createPacket(pt,node);

Framework Version 1

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;
    }
}

Wrapping and Unwrapping of Parameter Objects in the Adaptation Layer

Old Plugin

SecureNode node = new SecureNode();
PacketType pt = TypeFactory.getEncryptedPacketType();
...
Packet packet = PacketFactory.createPacket(pt,node);

New Framework

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;
    }
}

Tool Validation: the ComeBack!

ComeBack! homepage: http://comeback.sf.net

Framework Binaries

Refactoring History
Prolog Engine
Adapter Fact Base
pretty printing

ComeBack!

Side-by-Side Plugin Execution

Version 1
Version 2
Version 3
Evolution

Adaptation Layers
Frameworks
Plugins
Application
Adapteer Module
Refactorings History
Adapter Generator
generate
input
Case Studies

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

Department of Computer Science, Technische Universität Dresden

References

CBAddClass: Proof

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 be true for the program changed by AddClass. Let the changed program be P'.

\[
P' = (\text{Class}(\text{superclass}) \land 
-\text{IsClass}(\text{class}) \land \forall c \in \text{subclasses}, 
(\text{IsClass}(c) \land (\text{Superclass}(c) = \text{superclass})))
\]

\[
\Rightarrow (P' \Rightarrow \text{IsClass}(\text{superclass})) 
\Rightarrow \text{TATATA}(\text{T} \Rightarrow \text{T})
\]

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

CBPushDownMethod: Definition

CBPushDownMethod(class, subclass, method) is defined as a set of refactorings executed in two steps:

1. AddMethod(class, method, Method(subclass, method))
   
   Add to the class class the method method, which is semantically equivalent to the method with the same name defined in subclass.

2. RemoveMethod(subclass, method) Remove method from subclass.

The precondition of CBPushDownMethod:

- \text{IsClass}(\text{class})
- \text{IsClass}(
  \text{subclass})
- \text{Superclass}(\text{subclass}) = \text{class}
- \text{Delegates}(\text{subclass}, \text{method})
- \text{Delegates}(\text{class}, \text{method})
- \text{UnderstandsSelect}(\text{subclass}, \text{method})
- \text{UnderstandsSelect}(\text{class}, \text{method})

TU Dresden, 20.10.2008
Ilie Şavga, Michael Rudolf, Sebastian Götz and Uwe Allmann
CbPushDownMethod: Proof

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.

CbExtractClass: Definition

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.

Creating Adaptation Layer: AddAdapter

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)}:
   Move all but the delegation variable of class to the class created in step 1.
4. ∀ m ∈ {DDefinesSelector(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.

References

Molhadore homepage.