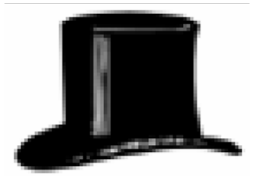
TECHNISCHE UNIVERSITÄT DRESDEN

32. Practical Refactoring-Based Framework Upgrade with *Comeback!*

Ilie Şavga, 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

Refactoring and Software Evolution

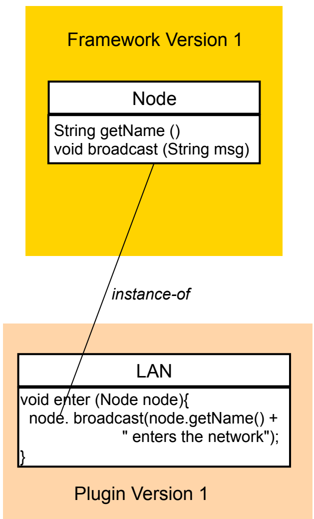
"A large program that is used undergoes continuing change or becomes progressively less useful."
Lehman's first law Lehman and Belady, p. 250

"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
Opdyke, 1992; Roberts, 1999

Software Frameworks and Plugins

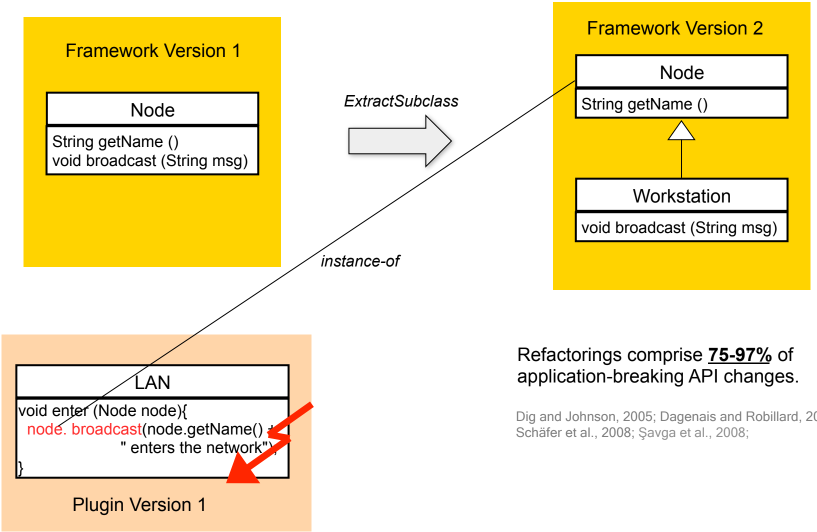


A software framework is a software component that embodies a skeleton solution for a family of related software products and is instantiated by modules containing custom code (*plugins*).

Johnson and Foote, 1998

Examples inspired by Demeyer et al., 2005

Framework Refactoring Breaking a Plugin



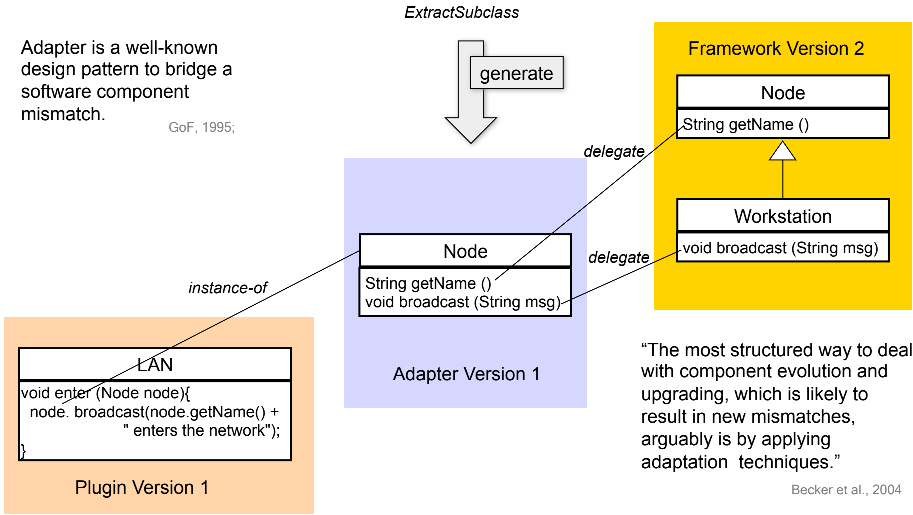
Refactorings comprise **75-97%** of application-breaking API changes.

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

Refactoring-Driven Adaptation

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

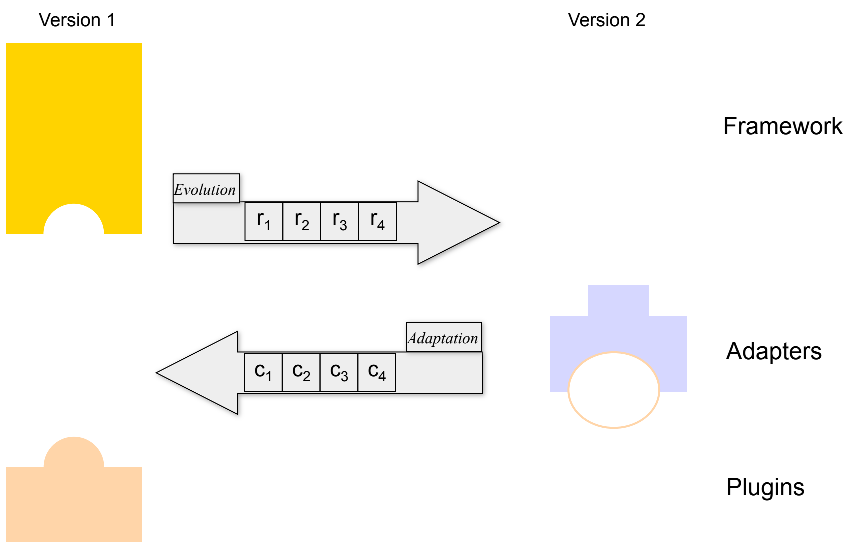
GoF, 1995;



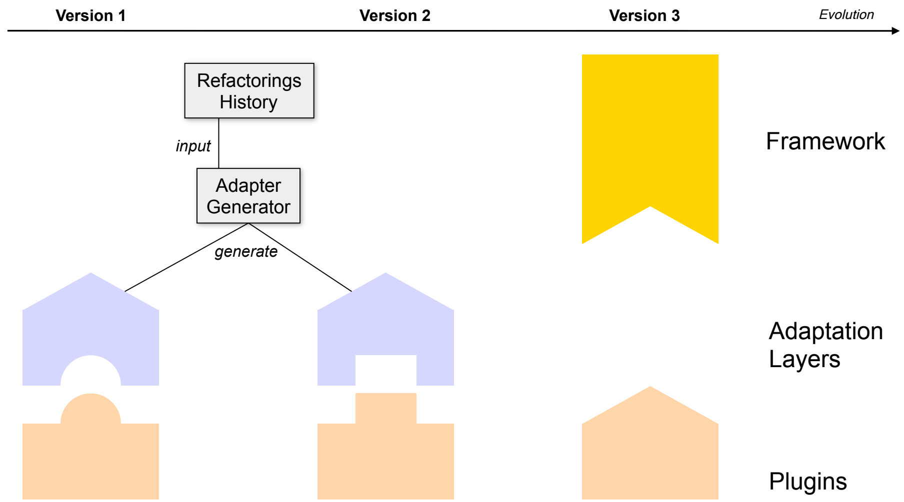
"The most structured way to deal with component evolution and upgrading, which is likely to result in new mismatches, arguably is by applying adaptation techniques."

Becker et al., 2004

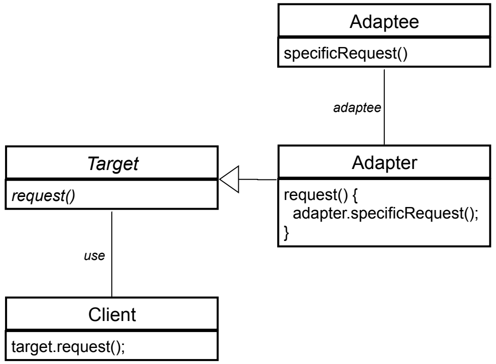
Comebacks: Refactoring Inverses on Adapters



Comebacks: Refactoring Inverses on Adapters

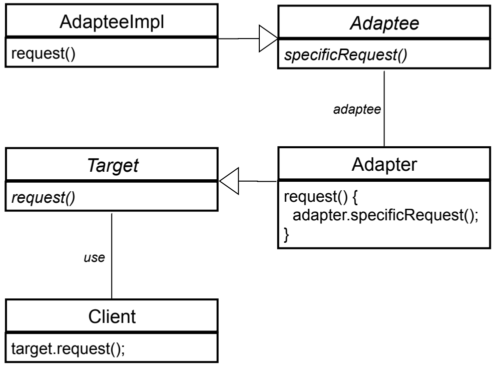


Adapter Design Pattern (Object Version)



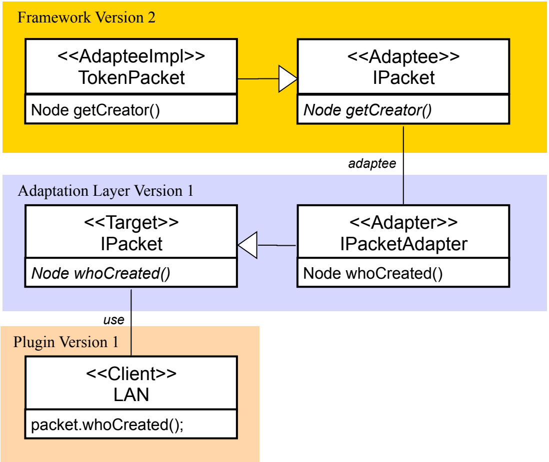
GoF, 1995

Refined Adapter Design Pattern (Object Version)

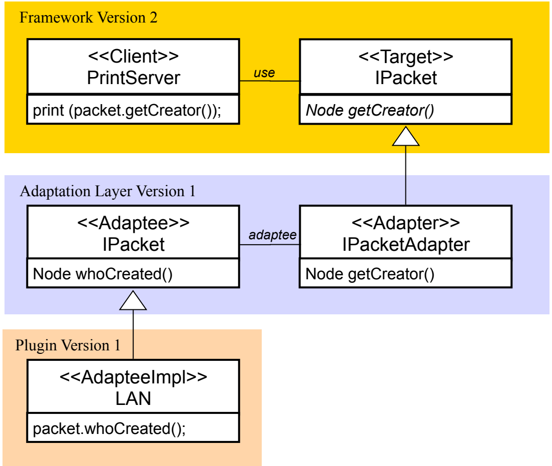


GoF, 1995

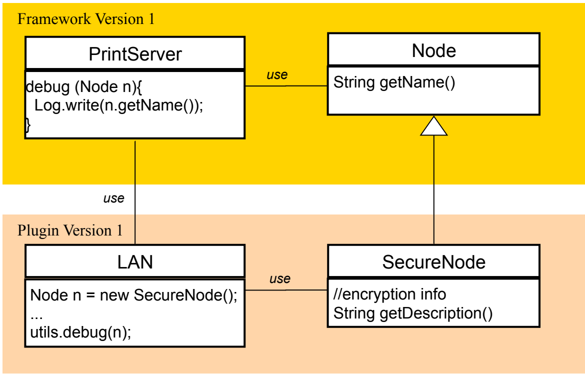
The Comeback! Black-Box Interface Adapter



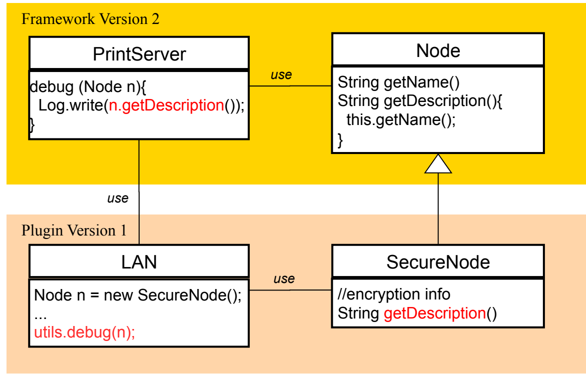
The Comeback! White-Box Interface Adapter



A White-Box Framework (Re-use by Inheritance)

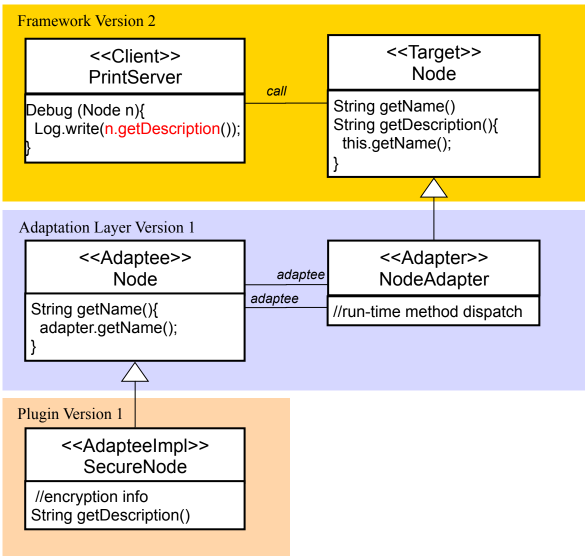


Fragile Base Class Problem: Method Capture



Steyaert et al, 1996; Mikhajlov and Sekerinski , 1998

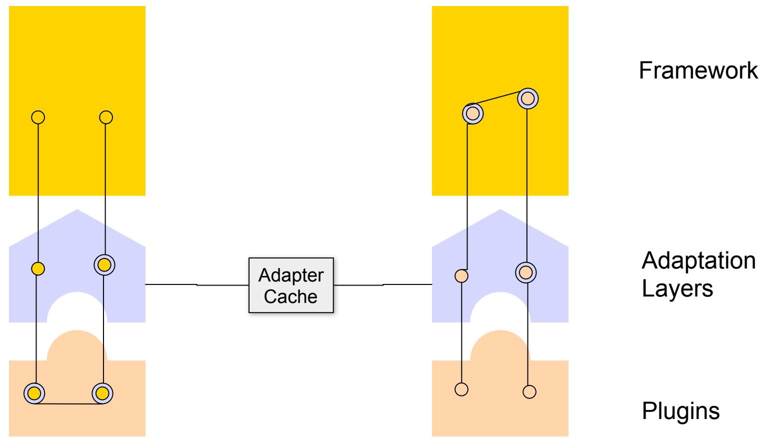
Method Capture Solved with Comback!White-Box Class Adapter



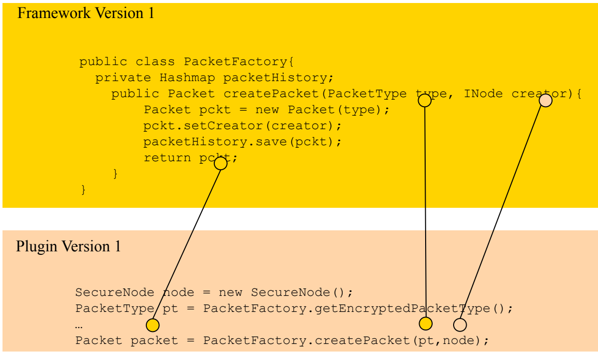
Exhaustive API Adaptation: Management of Adapters

Call Framework -> Plugins

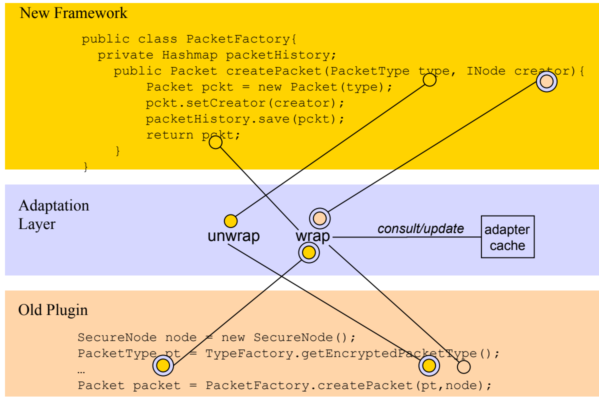
Call Plugins -> Framework



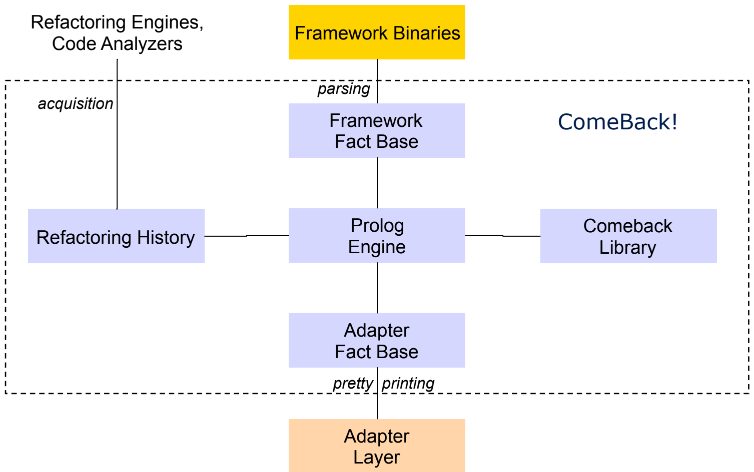
Framework<->Plugin Object Exchange



Wrapping and Unwrapping of Parameter Objects in the Adaptation Layer

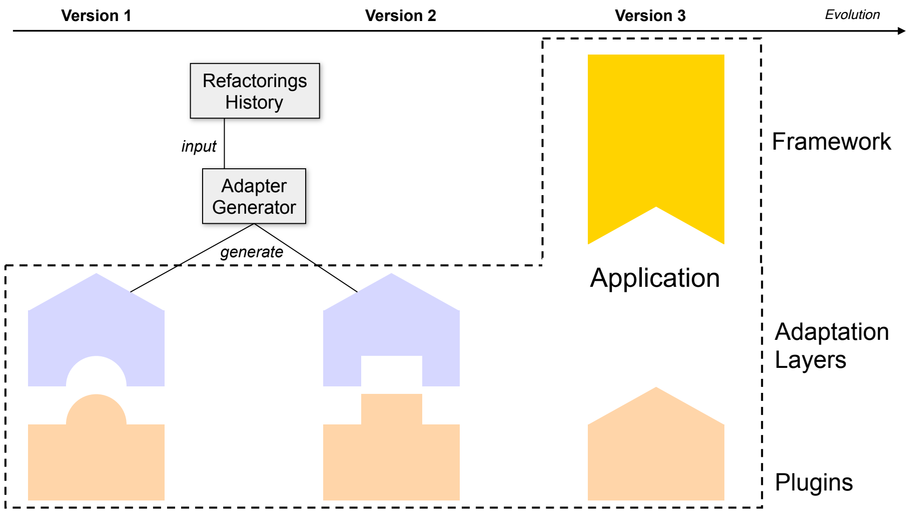


Tool Validation: the ComeBack!



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

Side-by-Side Plugin Execution



Case Studies

Java-based frameworks: SalesPoint and JHotDraw — SalesPoint; 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

- quering Eclipse refactoring log
- investigating the use of CVS

Intrusive Refactoring-Based Adaptation

CatchUp!: intrusively adapting plugins — Henkel and Diwan, 2005

- 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 — Dig at al., 2008

- 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

References

M.M. Lehman and L.A.Belady. Program evolution: processes of software change. In *APIC Studies in Data Processing*, San Diego, CA, USA, 1985.

W. F. Opdyke. Refactoring Object-Oriented Frameworks. *PhD thesis*, Urbana-Champaign, IL, USA, 1992.

D. B. Roberts. Practical analysis for refactoring. *PhD thesis*, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1999.

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

R. Johnson and B. Foote. Designing reusable classes. Journal of Object-Oriented Programming, 1(2):22–35, June 1988.

S. Demeyer, F. V. Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. D. Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, M. El-Ramly. The LAN-simulation: A refactoring teaching example. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 481–490, New York, NY, USA, 2008. ACM.

Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 471–480, New York, NY, USA, 2008. ACM.

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

S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an engineering approach to component adaptation. In R. H. Reussner, J. A. Stafford, and C. A. Szyperski, editors, Architecting Systems with Trustworthy Components, volume 3938 of Lecture Notes in Computer Science, pages 193–215. Springer, 2004.

L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. Lecture Notes in Computer Science, 1998.

Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages and Applications, pages 268–285. ACM Press, October 1996.

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

JHotDraw homepage. www.jhotdraw.org.

CORBA homepage. http://www.corba.org

Microsoft COM homepage. http://www.microsoft.com/Com/default.mspx.

J. Camara, C. Canal, J. Cubo, and J. Murillo. An aspectoriented adaptation framework for dynamic component evolution. In *3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 59–71, 2006.

I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA '95: Proceedings of the tenth annual conference on Objectoriented programming systems, languages, and applications*, pages 426–438, New York, NY, USA, 1995. ACM Press.

S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361, London, UK, 2000. Springer-Verlag.

I.Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press.

K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance, page 359, Washington, DC, USA, 1996. IEEE Computer Society.

J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.

D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In ICSE'08: International Conference on Software Engineering, Leipzig, Germany, May 2008.

R. Keller and U. Hoelzle. Binary component adaptation. *Lecture Notes in Computer Science*, 1445:307–318, 1998.

S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *XP'02: Proceedings of Extreme Programming Conference*, pages 182–185, 2002.

Robert K. Yin. Case Study Research: Design and Methods. SAGE Publications, December 1994.

I. Savga and M. Rudolf. Refactoring-based adaptation of adaptation specifications. In SERA'08: Best Paper Selection Proceedings of Software Engineering Research, Management and Applications, Prague, Czech Republic, August 2008. Springer Verlag.

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

$$\begin{aligned} & P' \models (IsClass(class) \wedge \\ & (ClassReferences(class) = \emptyset) \wedge \\ & ((Subclasses(class) = \emptyset) \vee \\ & IsEmptyClass(class))) \\ \Leftrightarrow\ & (P' \models IsClass(class)) \wedge \\ & (P' \models (ClassReferences(class) = \emptyset)) \wedge \\ & ((P' \models (Subclasses(class) = \emptyset)) \vee \\ & (P' \models IsEmptyClass(class))) \\ \Leftrightarrow\ & \top \wedge \top \wedge (\top \vee \top) \Leftrightarrow \top \end{aligned}$$

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''.

$$\begin{aligned} & P'' \models (IsClass(superclass) \wedge \\ & \neg IsClass(class) \wedge \forall c \in subclasses. \\ & (IsClass(c) \wedge (Superclass(c) = superclass))) \\ \Leftrightarrow\ & (P'' \models IsClass(superclass)) \wedge \\ & (P'' \not\models IsClass(class)) \wedge \forall c \in subclasses. \\ & ((P'' \models IsClass(c)) \wedge \\ & (P'' \models (Superclass(c) = superclass))) \\ \Leftrightarrow\ & \top \wedge \top \wedge \top \wedge \top \Leftrightarrow \top \end{aligned}$$

The last derivation step is performed using the *post* of *AddClass* composed with the preconditions and *post* of *RemoveClass*.

□

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*:

1. $IsClass(class) \wedge$
2. $IsClass(subclass) \wedge$
3. $(Superclass(subclass) = class) \wedge$
4. $(Superclass(Delegatee(subclass)) = Delegatee(class)) \wedge$
5. $DefinesSelector(subclass, method) \wedge$
6. $\neg DefinesSelector(class, method) \wedge$
7. $(\neg UnderstandsSelector(class, method) \vee (LookUpMethod(class, selector) \overset{\alpha}{\equiv} Method(subclass, method)))$

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*.
- 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*).

□

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. $\forall v \in VariablesDefinedBy(class) \backslash \{DField(class)\}$. *MoveInstanceVariable(class, v, Delegatee(class)):* Move all but the delegation variable of *class* to the class created in step 1.
4. $\forall m \in \{d | DefinesSelector(class, d)\}$. *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.

References

S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 166–177, New York, NY, USA, 2000. ACM Press.

G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop on (IWPSE'04), pages 31–40, Washington, DC, USA, 2004. IEEE Computer Society.

C. Gorg and P. Weisgerber. Detecting and visualizing refactorings from software archives. In IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension, pages 205–214, Washington, DC, USA, 2005. IEEE Computer Society.

L. Zou and M.W. Godfrey. Detecting merging and splitting using origin analysis. In WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering, page 146, Washington, DC, USA, 2003. IEEE Computer Society.

F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution, page 126, Washington, DC, USA, 2003. IEEE Computer Society.

P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06), pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, ECOOP, volume 4067 of Lecture Notes in Computer Science, pages 404–428. Springer, 2006.

D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In ICSE, 2007.

Molhadoref homepage. https://netfiles.uiuc.edu/dig/MolhadoRef.

C. Bitter. Using Aspect-Oriented Techniques for Adaptation. Master Thesis, TU Dresden, 2007.

T. Mens and S. Demeyer. Evolution metrics. In Proc. Int. Workshop on Principles of Software Evolution, 2001.

L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Quality-driven software reengineering. Journal of System Software, 66(3):225–239, 2003.

F. Bannwart and P. Müller. Changing programs correctly: Refactoring with specifications. In J. Misra, T. Nipkow, and E. Sekerinski, editors, FM, volume 4085 of Lecture Notes in Computer Science, pages 492–507. Springer, 2006.

T. Mens, N. V. Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations: Research articles. *Journal of Software Maintenance and Evolution:Research and Practice*, 17(4):247–276, 2005.

N.V.Eetvelde. A Graph Transformation Approach to Refactoring. University of Antwerp. PhD. Thesis, 2007.