Modern Refactoring

II Escola de Informática Teórica e Métodos Formais

Volker Stolz[1], Larissa Braz[2], Anna M. Eilertsen[3],
Fernando Macías[1], Rohit Gheyi[2]

Western Norway University of Applied Sciences,
Universidade Federal de Campina Grande,
University of Bergen, Norway

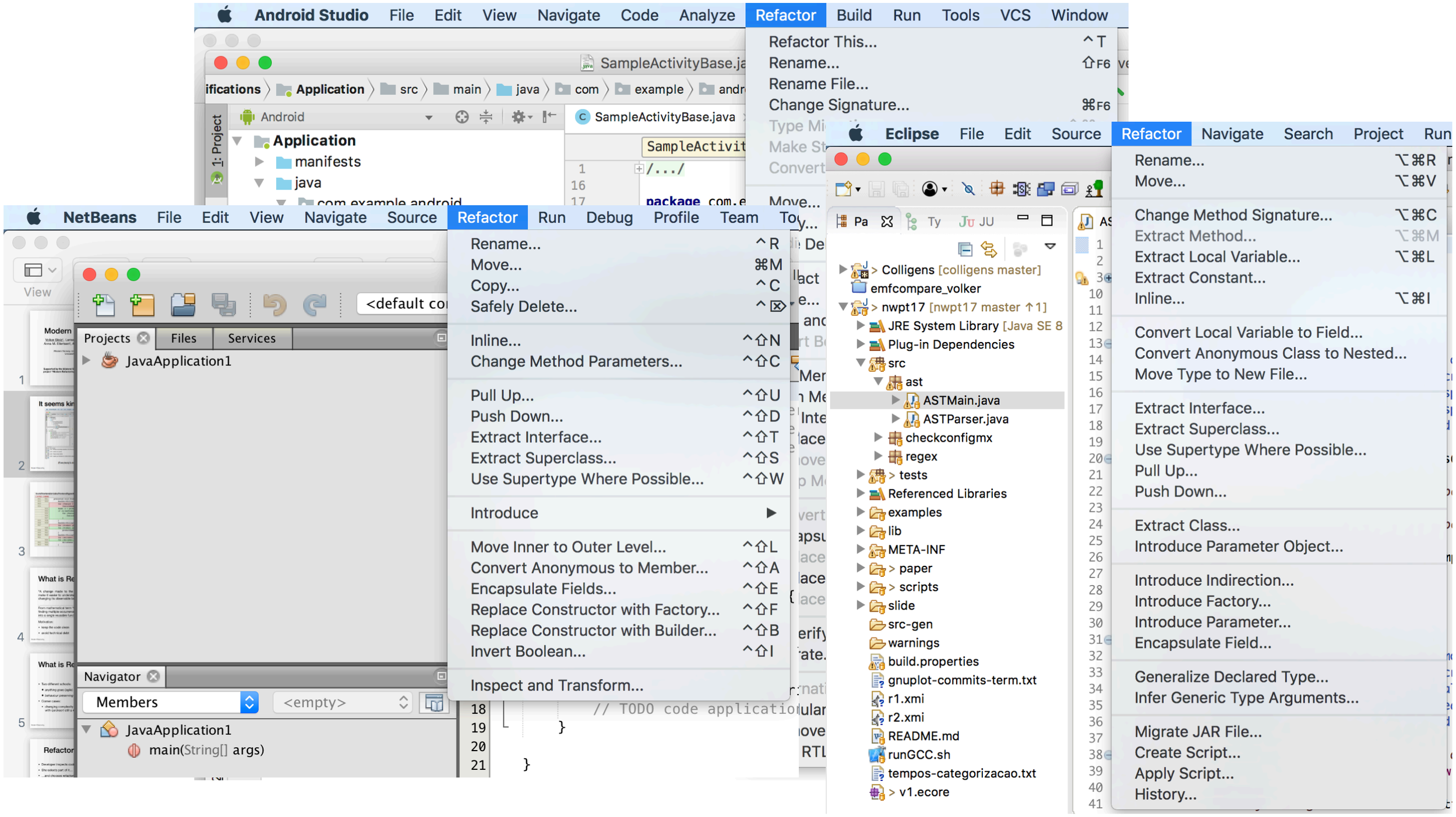
Supported by the bilateral SIU/CAPES project "Modern Refactoring" 2017/18

Høgskulen på Vestlandet

Overview

- What are refactorings? What are they good for?
- Examples in common IDEs
- Implementation of refactorings
- Research challenges & prerequisites

It seems kinda important...

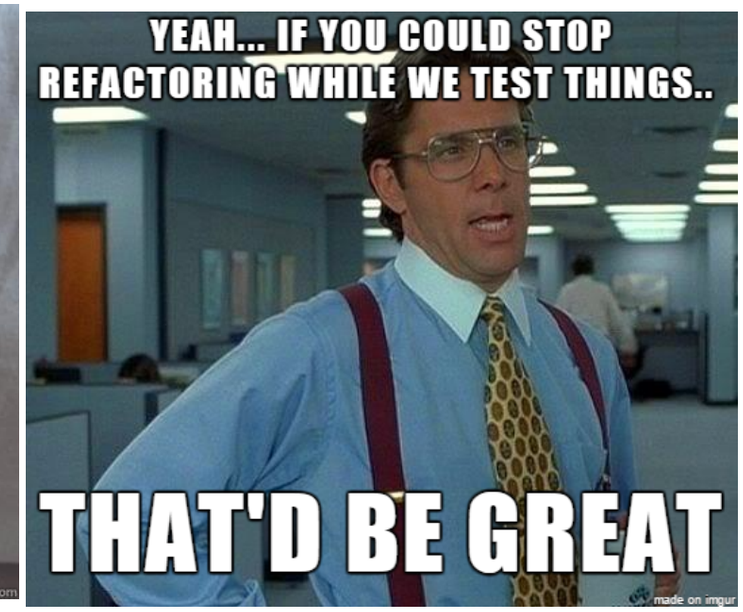


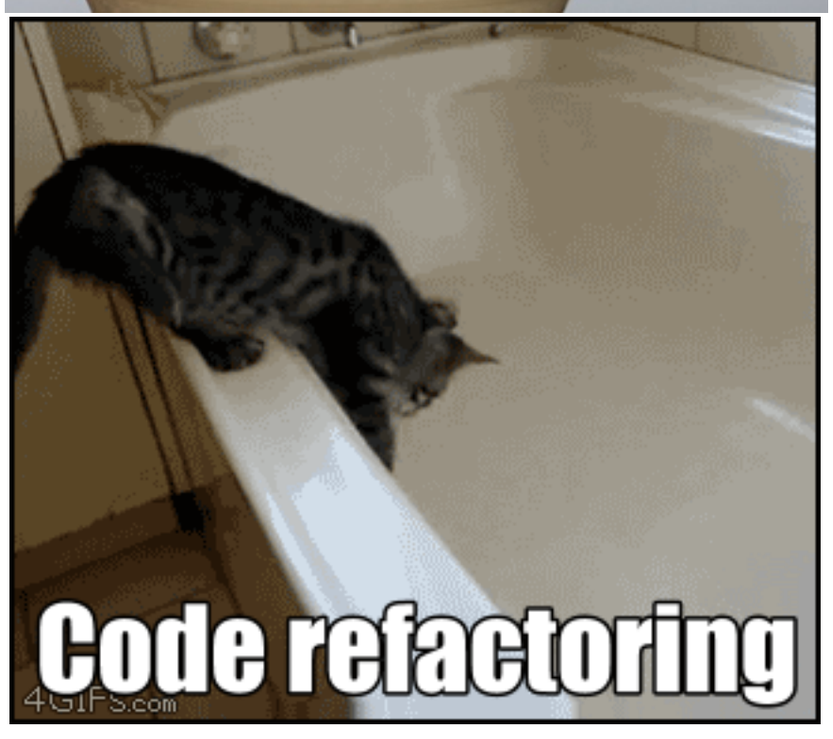
(Everybody's doing it; you should as well!)

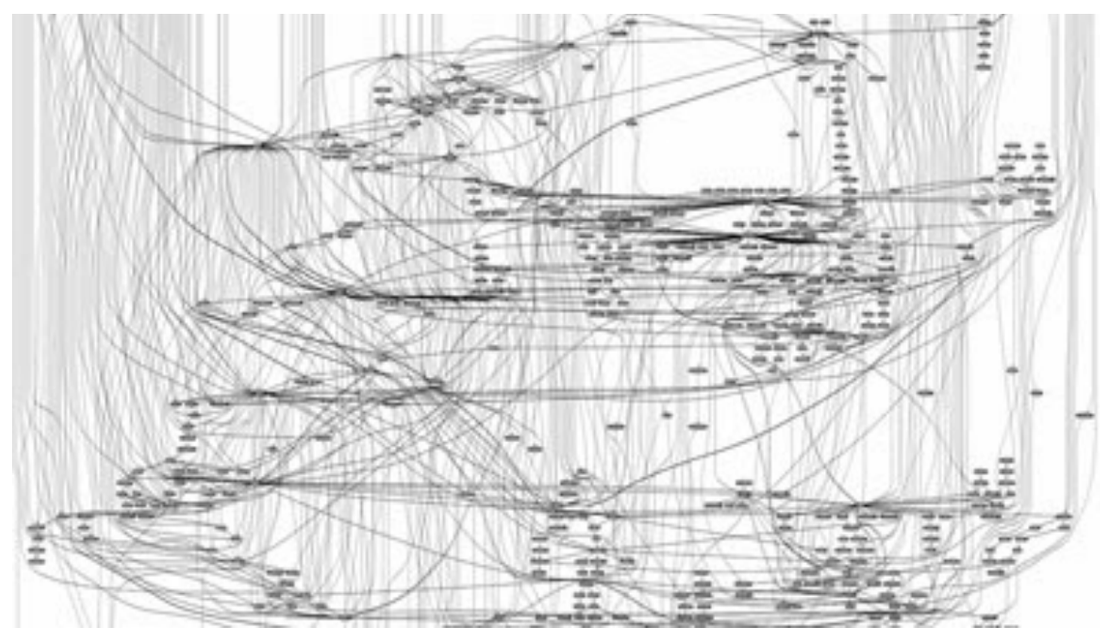
trunk/frontend/src/abs/frontend/typechecker/TypeChecker.jadd

r19783	r19896	
951	951	`protected void Product.typeCheck(SemanticErrorList e) {`
952	952	`HashSet<String> featureNames = new HashSet<String>();`
953		`for (Feature f : getModel().getProductLine().getFeatures()) {`
954		`featureNames.add(f.getName());`
	953	`Model m = getModel();`
	954	`if (m.hasProductLine()) {`
	955	`for (Feature f : m.getProductLine().getFeatures()) {`
	956	`featureNames.add(f.getName());`
	957	`}`
955	958	`}`
956	959	`HashSet<String> productNames = new HashSet<String>();`
957		`for (Product prod : getModel().getProducts()) {`
	960	`for (Product prod : m.getProducts()) {`
958	961	`productNames.add(prod.getName());`
959	962	`}`
960	963	`HashSet<String> deltaNames = new HashSet<String>();`
961		`for (DeltaDecl delta : getModel().getDeltaDecls()) {`
	964	`for (DeltaDecl delta : m.getDeltaDecls()) {`
962	965	`deltaNames.add(delta.getName());`
963	966	`}`

Refactoring: ~~how to do it?~~ Why does everyone hate it?







What is Refactoring? (1)

“A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour” [Fowler]

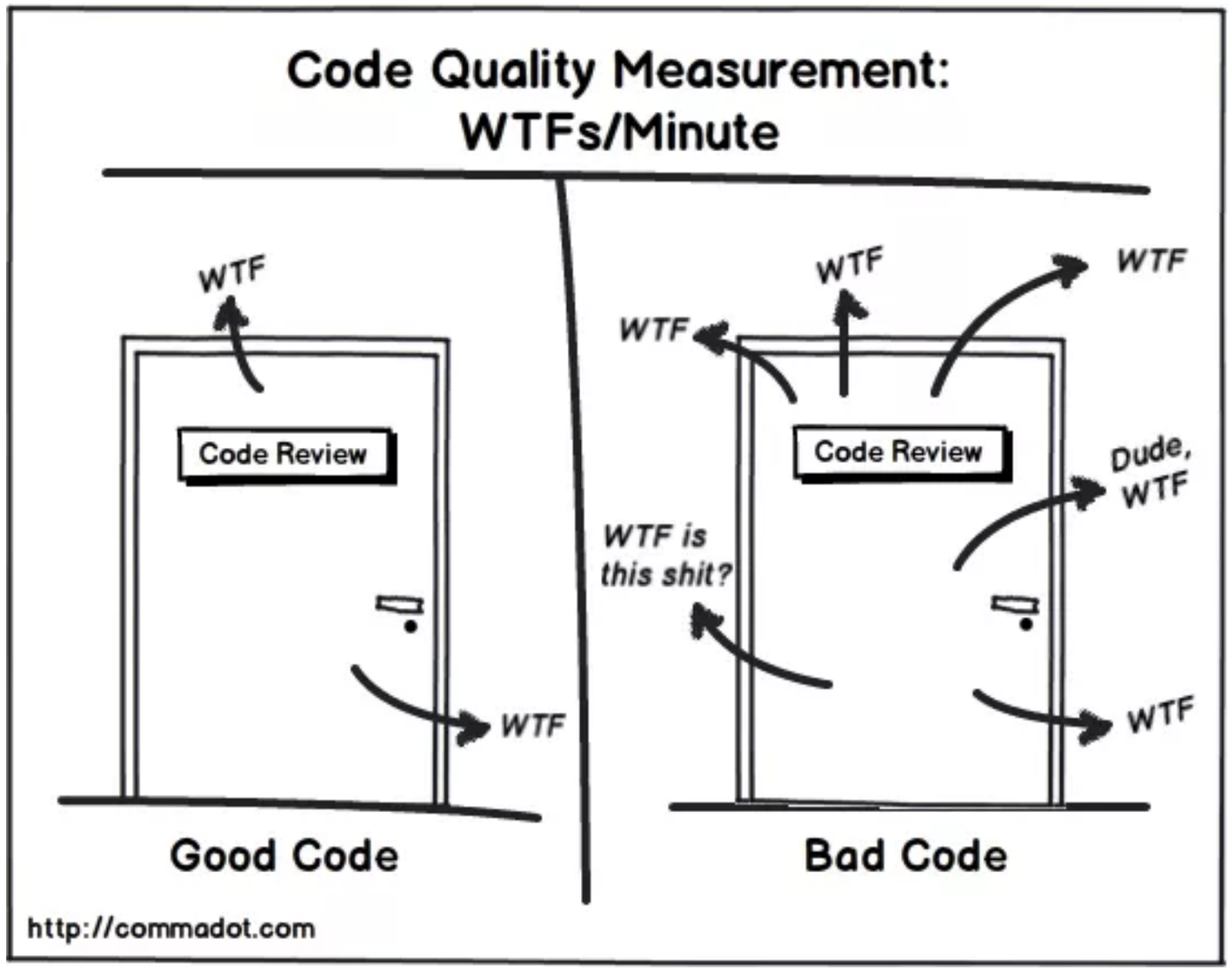
From mathematical term “factor”:
finding multiple occurrences of similar code and *factoring* it into a single reusable function

---

Motivation:

- keep the code clean
- avoid technical debt

Motivation

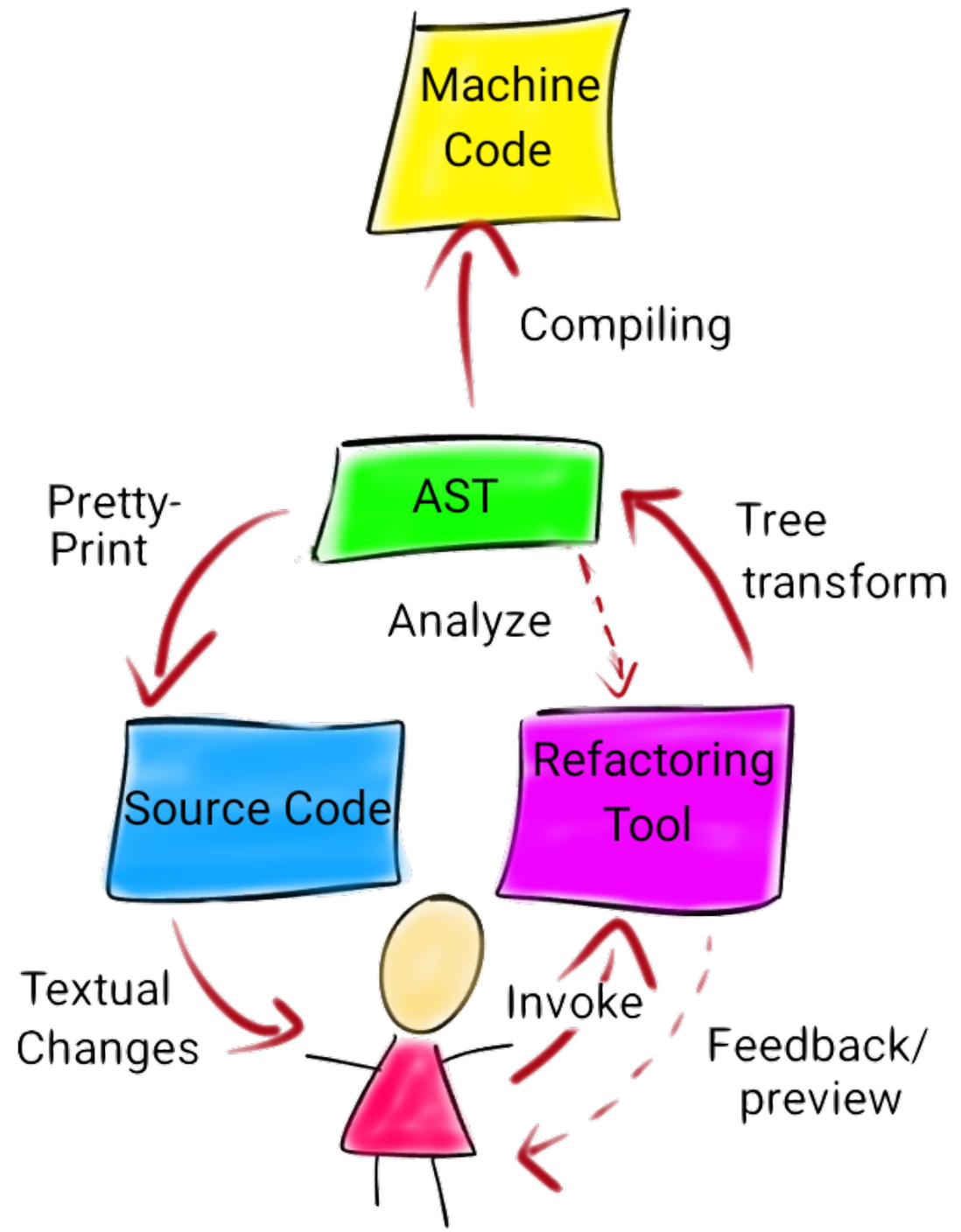

Code Quality Measurement:
WTFs/Minute
WTF
Code Review
WTF
Good Code
WTF
WTF
WTF
Code Review
Dude,
WTF
WTF is
this shit?
WTF
Bad Code
http://commadot.com

What is Refactoring? (2)

- Two different schools:
  - *anything goes* (agile)
  - *behaviour preserving*
- Corner cases:
  - changing complexity class, e.g. replacing bubble sort with quicksort still a refactoring?

Refactoring Process

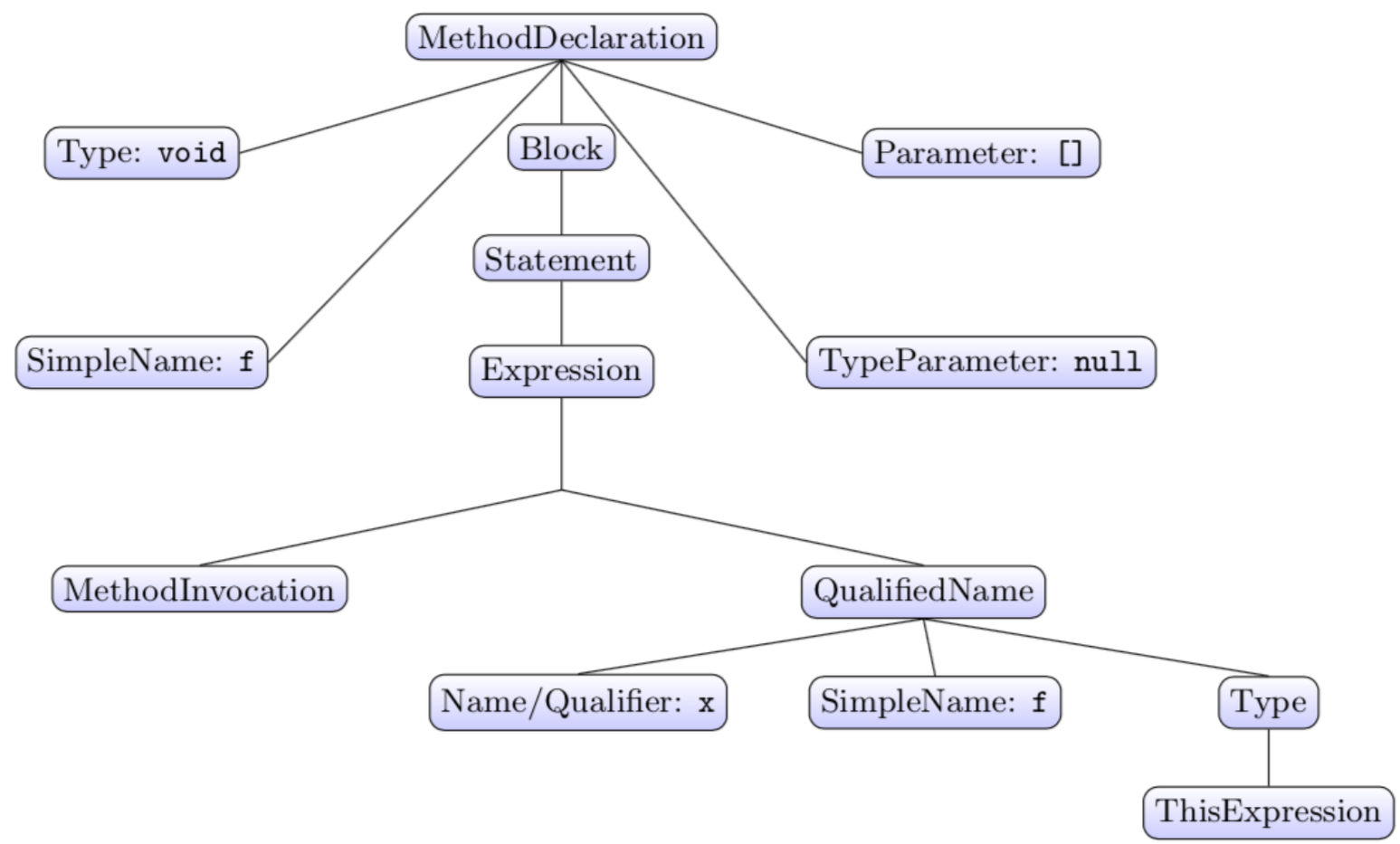
- Developer inspects code.
- She selects part of it…
- …and chooses refactoring action from menu.
- Refactorings usually modify the Abstract Syntax Tree (AST) in memory…
- … and then synchronize the source code file.



Abstract Syntax Tree (AST)

- In-memory representation of parsed source code
- Semantic information available (Where was this variable declared? What are the superclasses?)

```
1  public class C {
2          public X x = new X()
3          public void f(X x) {
4                  x.m(this);
5          }
6  }
```

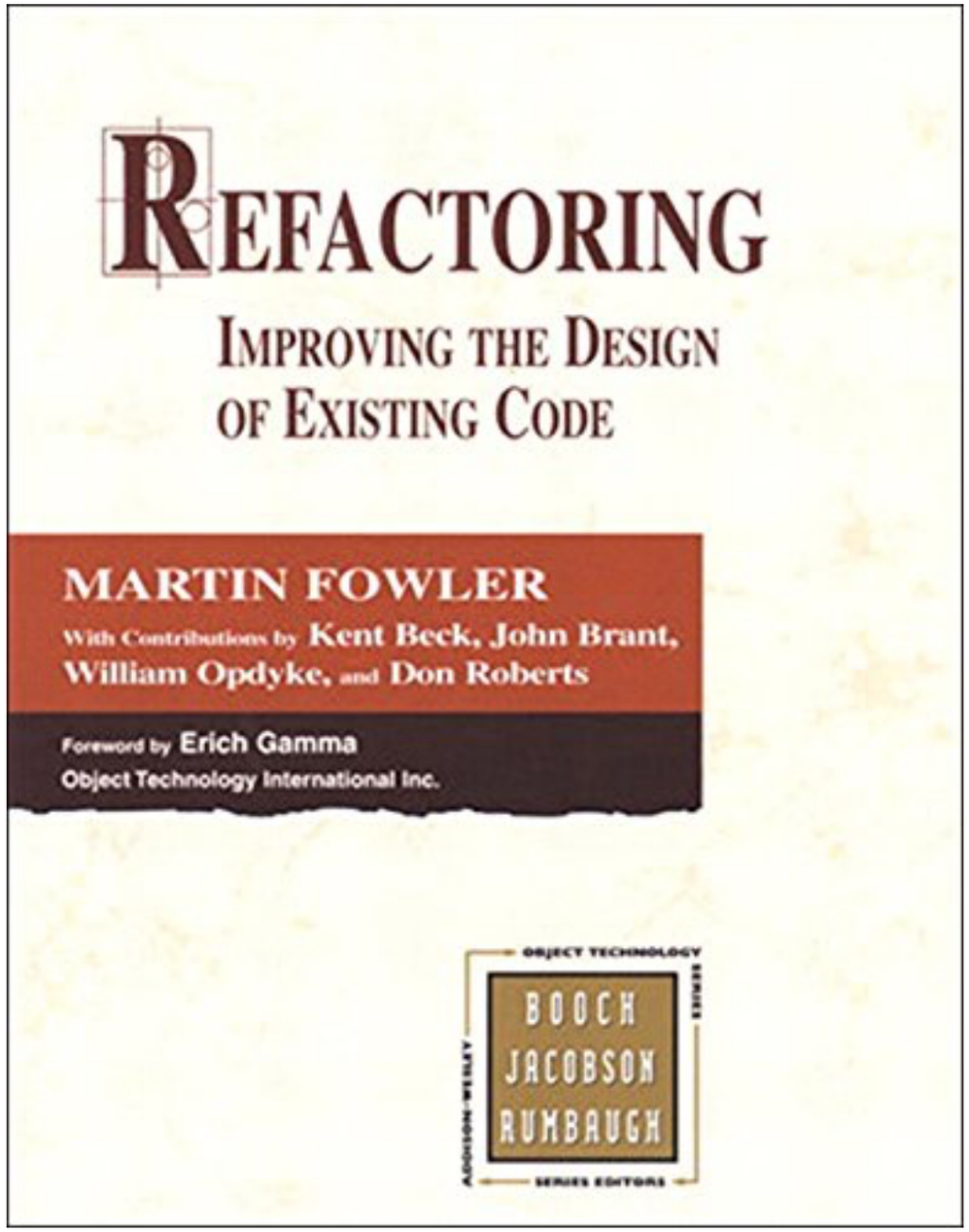


Refactoring: Origins

- Opdyke's PhD thesis [1992]
- Smalltalk Refactoring Browser [Roberts, Brant, Johnson '97]
- "Refactoring: improving the design of existing code" [Fowler '99]

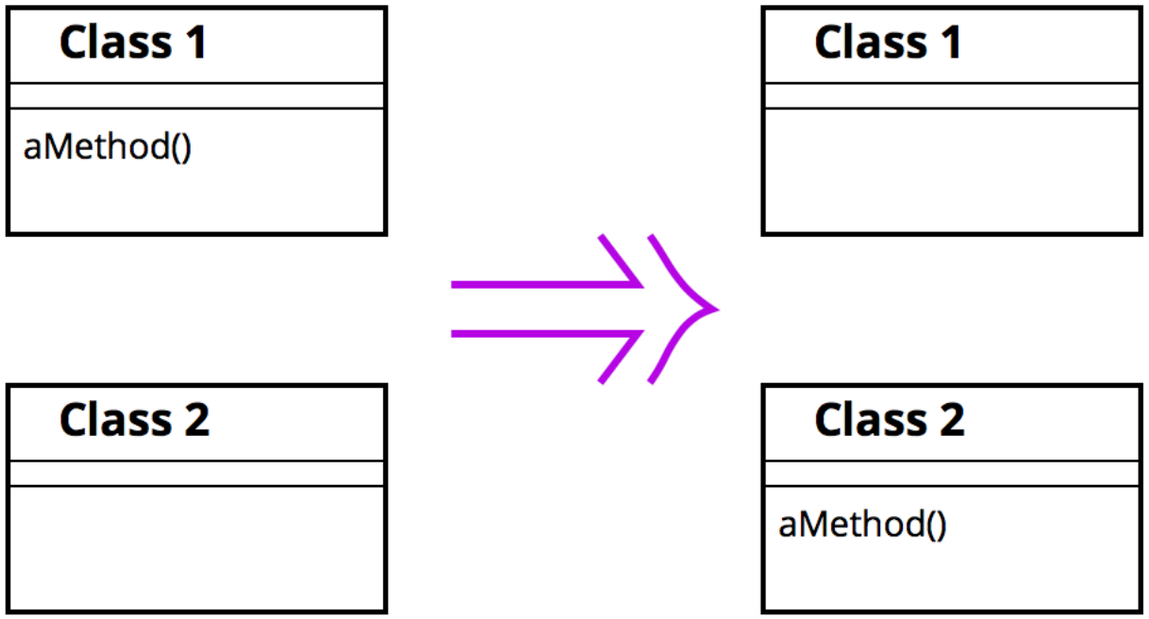
- 30% of changes are refactorings [Soares et al., 2011]
- Extract Method most popular — but performed manually [Murphy et al., 2006]

Literature



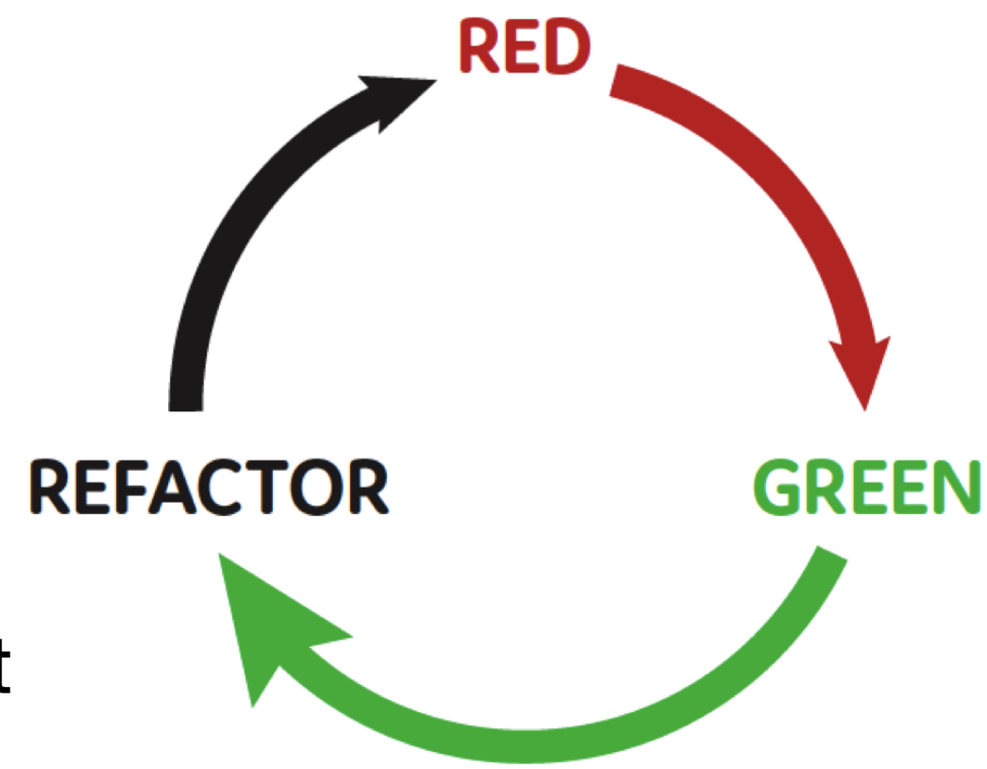
Refactoring: Improving the Design of Existing Code
Martin Fowler with Kent Beck, John Brant, William Opdyke, Jon Roberts

Addison Wesley, 1999



Adoption of Refactorings

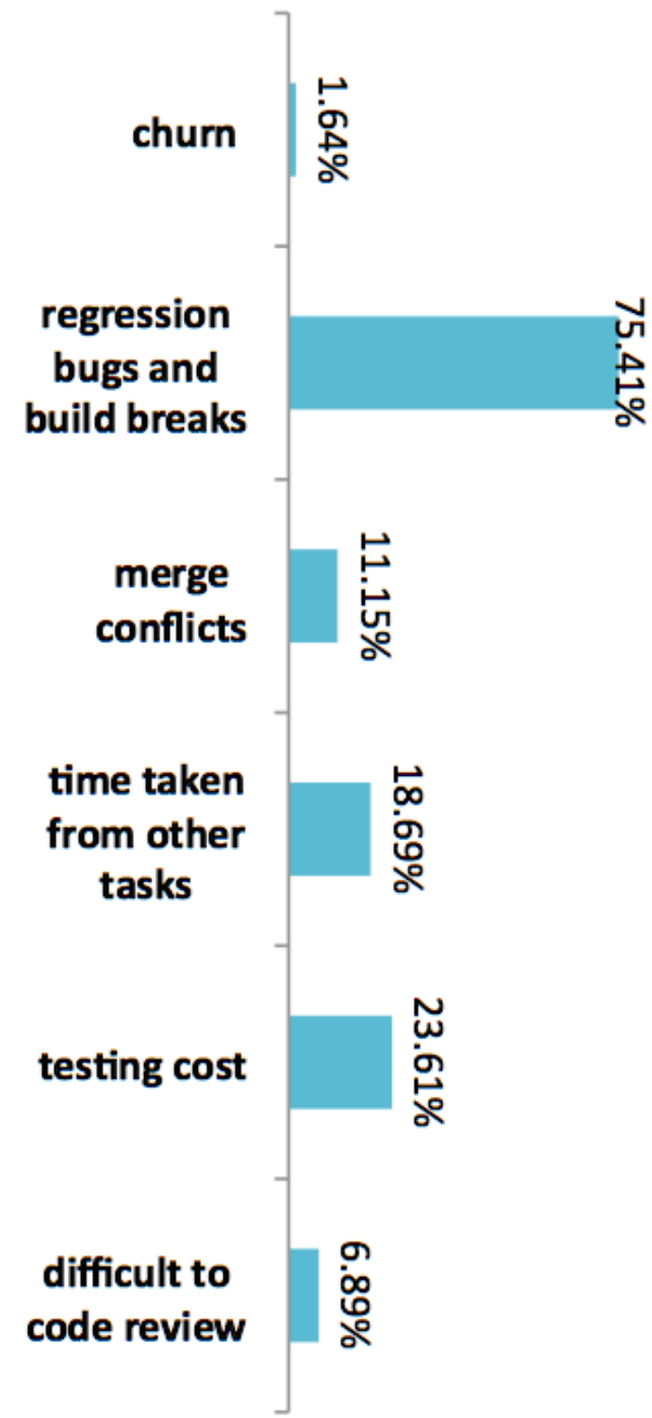
- Agile: fully embraced refactorings
- Developers usually sceptical of automated changes
- Study: developers more confident when they can predict changes
- Problem in OO languages: refactoring touches on multiple contexts



The agile workflow

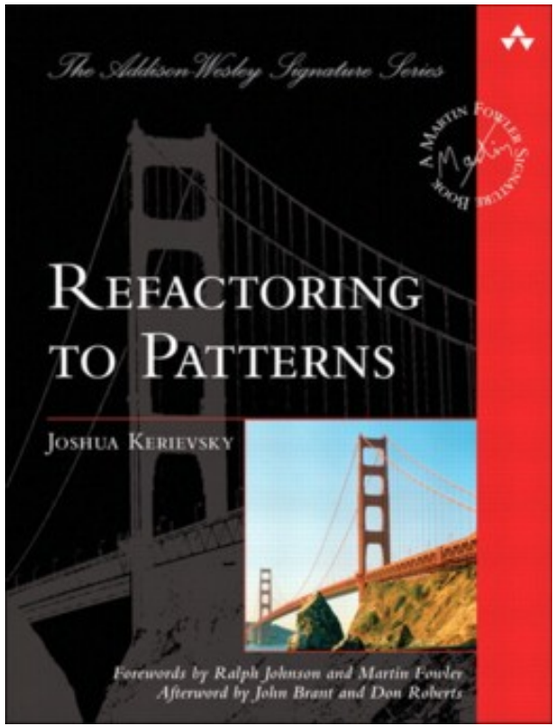
Adoption: Software Engineering Studies

- Kim et al. (2010): survey on more than 300 engineers who had used refactoring during Microsoft Windows development
- Tempero et al. (C.ACM, 2017):
  - Survey on 3785 developers in 2009
  - They understand benefits of refactoring, but they see costs and risks as well.



Related Topics: Patterns

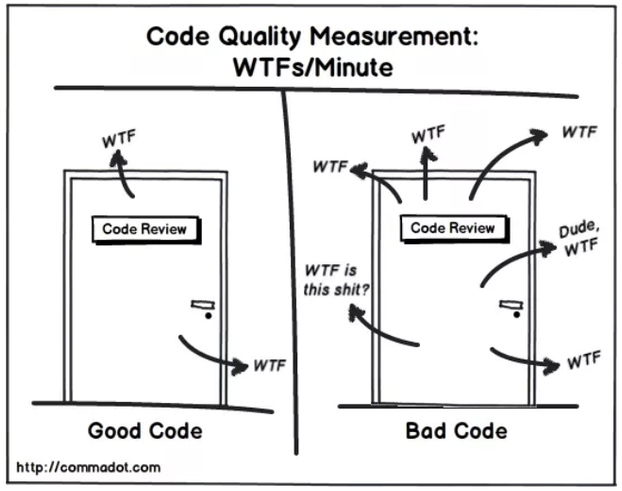
- “Design Patterns: Elements of Reusable Object-Oriented Software” [Gamma, Helm, Johnson, Vlissides, 1994]
- “Refactoring to patterns” [Kerievsky, 2005]



- “Anti-patterns” and “code smells”: indicators of design deficiencies
- Ignoring exceptions (*AP)*, magic strings (*AP),* repeated code (*CS*), long functions (*CS*)
- Detection partially automated
- Refactoring to more structured solutions

Software Quality Metrics

- How “good” is your code?
- Often subjective, but some guidelines:
  - high cohesion/low coupling between classes
  - long method body
  - class with too many methods
- Refactorings affect those metrics:
  - Extract Method *reduces* length of method and cyclometric complexity…
  - …but obviously increases *number of methods*.



Software Quality Metrics (2)

- Tools like Findbugs, Checkstyle, SonarQube identify problems
- Developers still need to act on that info
- Problem with automation:
  - large search-space
  - often many (overlapping) possibilities
  - Extract Method ↔ Inline Method "competing" against each other
- Our attempt: Kristensen/Stolz, "Search-based composed refactorings", NIK 2014

```
class C {
  A a; B b; boolean bool;

  void method(int val) {
    if (bool) {
      a.foo();
      a = new A();
      a.bar();
    }

    a.foo();
    a.bar();

    switch (val) {
    case 1:
      b.a.foo();
      b.a.bar();
      break;
    default:
      a.foo();
    }
  }
}
```

Reducing Coupling

Before

```
class C {
  A a; B b;
  X x;
  void m() {
    x.y.foo();
    x.y.bar();
  }
}

class X {
  Y y;
  /* ... */
}

class Y {
  void foo(){ ... }
  void bar(){ ... }
}
```

After

```
class C {
  A a; B b;
  X x;
  void m() {
    x.fooBar();
  }
}

class X {
  Y y;
  /* ... */
  void fooBar() {
    y.foo();
    y.bar();
  }
}

class Y ... /* unchanged */
```

- Coupling Between Object Classes (CBO) of class C improves from 4 to 3…
- …but sometimes introduces additional coupling into the receiving class!

Related Topics: Source Code Rejuvenation

“**Source Code Rejuvenation**”
[Pirkelbauer, Dechev, Stroustrup ’10]

- automated migration of legacy code
- leverages enhanced program language/library facilities
- “*reverse (some forms of) (software) entropy*”
- “*preserves <u>or improves</u> a program’s behavior*”

Source Code Rejuvenation

	Source Code Rejuvenation	**Refactoring**
Transformation	Source-to-source	Source-to-source
Behavior preserving	Behavior *improving*	Behavior preserving
Directed	yes Raises the level of abstraction	no
Drivers	Language / library evolution	Feature extensions Design changes
Indicators	Workaround techniques / idioms	Code smells Anti-patterns
Applications	One-time source code migration	Recurring maintenance tasks

From: Pirkelbauer, Dechev, Stroustrup, SOFSEM 2010

Source Code Rejuvenation

```
vector<int> vec;

// three consecutive push backs

vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
```

Inefficient!

Sizeof() what again?!

```
// copying from an array
int a[] = {1, 2, 3};
vector<int> vec(a,a+sizeof(a)/sizeof(int));
```

Now isn't that pretty:

```
// rejuvenated source code in C++0x

vector<int> vec = {1, 2, 3};
```

Refactoring in IDEs

- All major IDEs support some form of refactoring
- Here: C, C++, Java
- Special case: command line tools for scripting (Go?)
- Support for scripting languages like Python, JavaScript, …
- Refactoring of UML models
  (semantical overlap with OO-refactoring)

Tool Support for Java

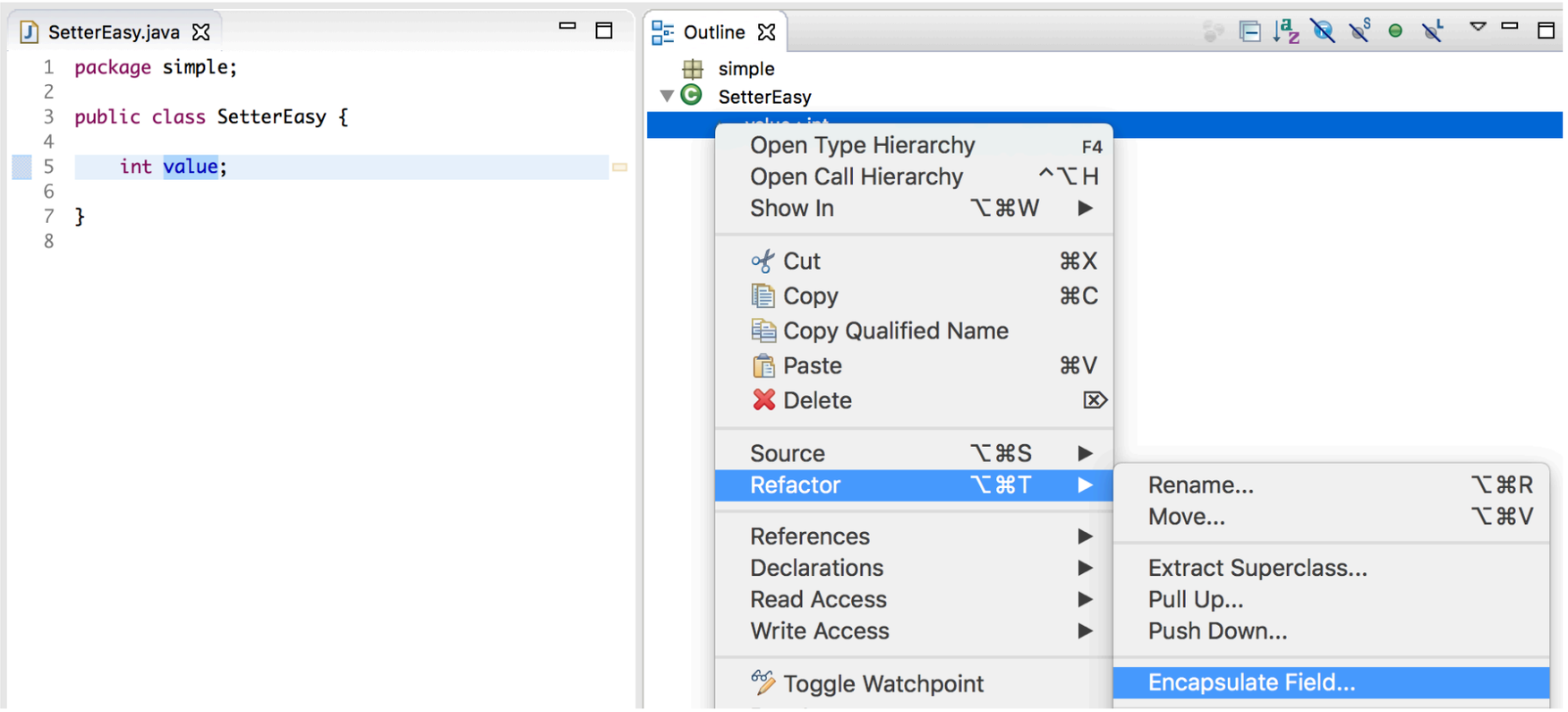
- Common IDEs:
  - Eclipse JDT
  - IntelliJ (Android)
  - NetBeans
- Other object-oriented languages similar:
  - Visual Studio

Refactoring: Common Java Examples

Encapsulate Field: avoid direct field access

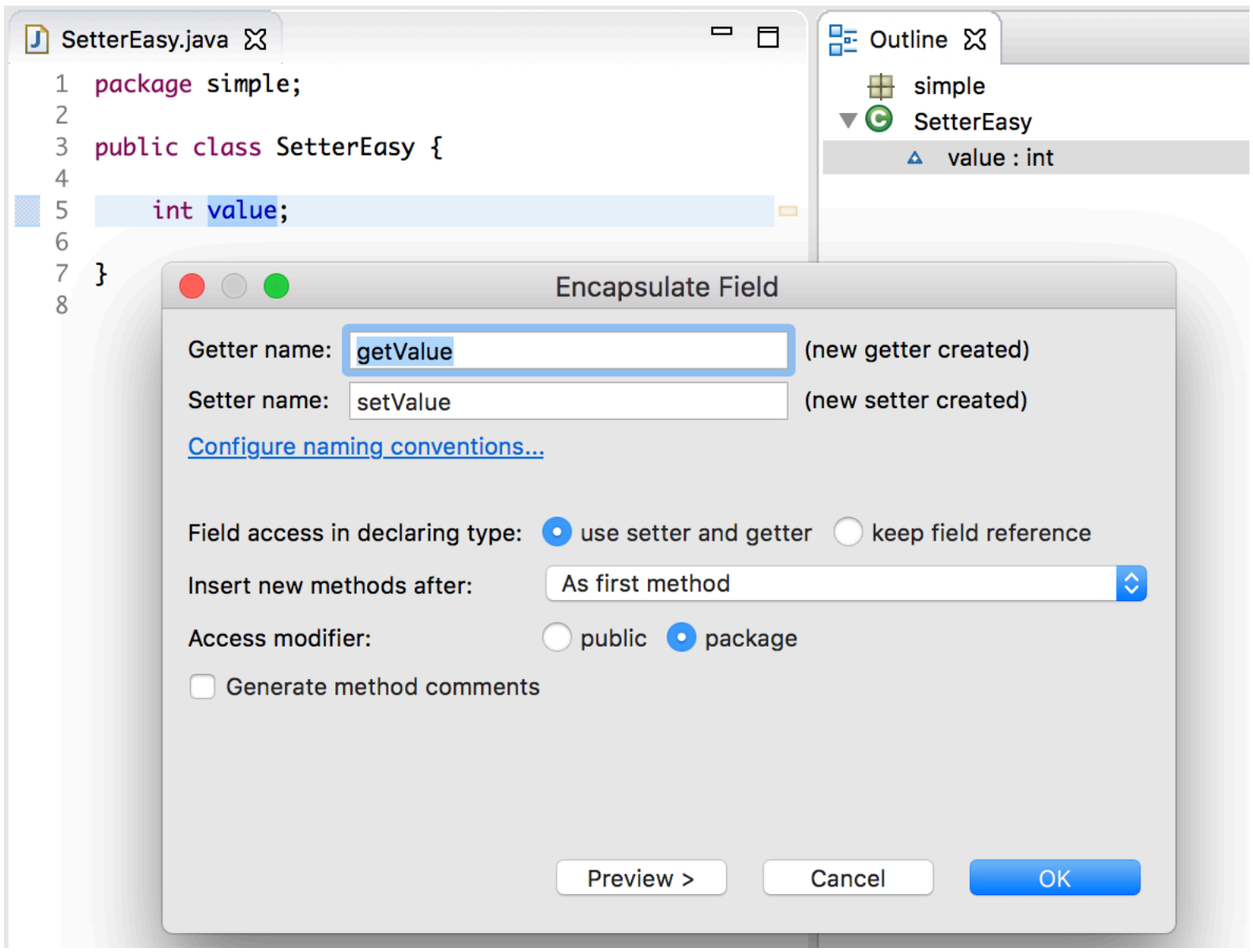
1) introduce setter & getter methods;
2) replace all field accesses with calls to new methods;
3) make field private.

Encapsulate Field



Right-click on a field and find the “Refactor” menu.

Encapsulate Field



IDEs will often have a helpful dialog, because further input is required.

Encapsulate Field

SetterEasy.java

```
1  package simple;
2
3  public class SetterEasy {
4
5      private int value;
6
7      int getValue() {
8          return value;
9      }
10
11     void setValue(int value) {
12         this.value = value;
13     }
14
15 }
16
```

Outline

- simple
- SetterEasy
  - value : int
  - getValue() : int
  - setValue(int) : void

SetterEasy.java

```
1  package simple;
2
3  public class SetterEasy {
4
5      int value;
6
7  }
8
```

Outline

- simple
- SetterEasy

Open Type Hierarchy F4
Open Call Hierarchy ^⌥H
Show In ⌥⌘W ▶

Cut ⌘X
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete ⌦

Source ⌥⌘S ▶
Refactor ⌥⌘T ▶

References ▶
Declarations ▶
Read Access ▶
Write Access ▶

Toggle Watchpoint

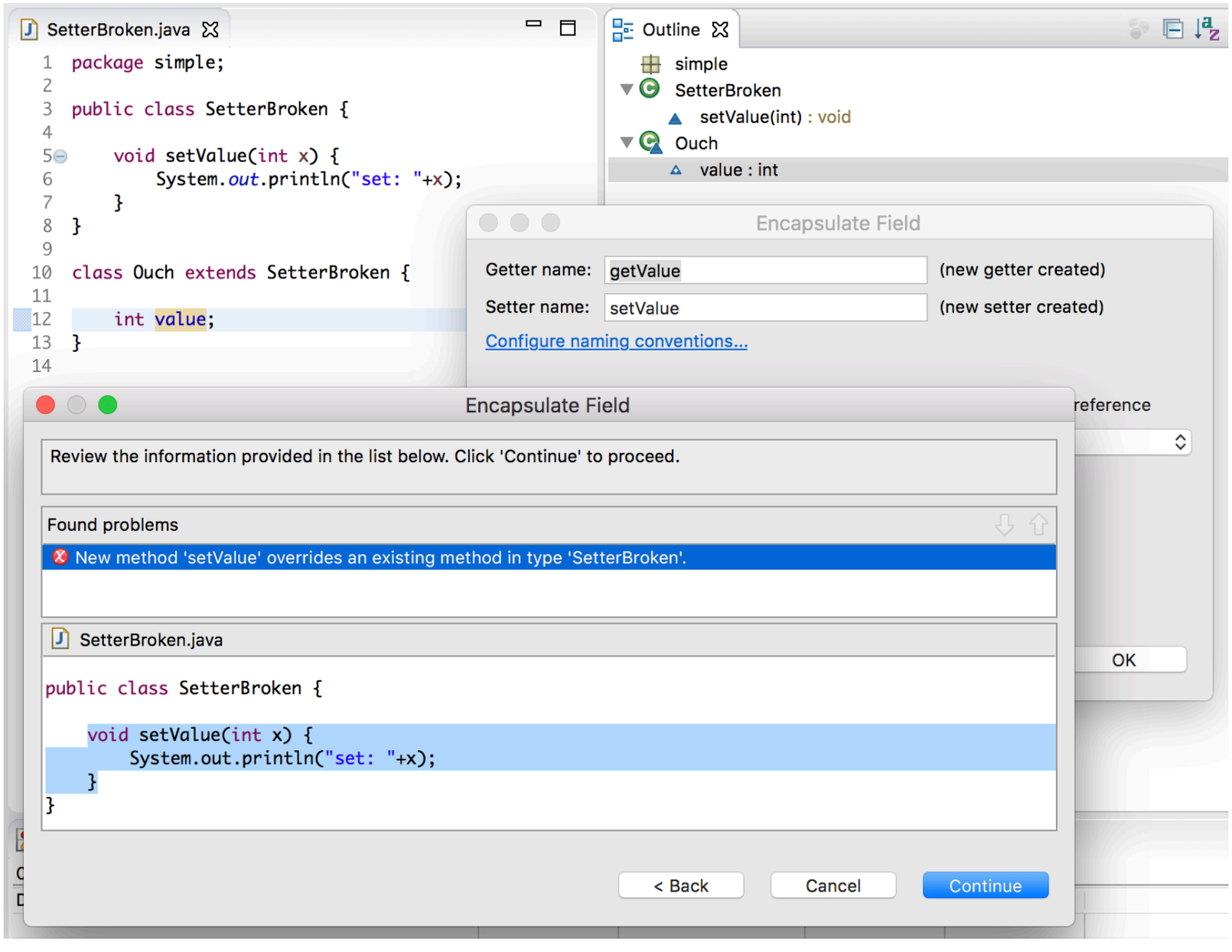
Rename... ⌥⌘R
Move... ⌥⌘V

Extract Superclass...
Pull Up...
Push Down...

Encapsulate Field...

Enjoy your result!

Encapsulate Field



IDEs will even try to be helpful!

Refactoring: Common Java Examples

Encapsulate Field: avoid direct field access
1) introduce setter & getter methods;
2) replace all field accesses with calls to new methods;
3) make field private.

Let's assume you have to *program* this refactoring.
Can you see what happens if you swap steps 2 & 3?
We will come back later to that.

Refactoring: Extract Local Variable

Before

```
1  public void f() {
2    a.b.c.d.m();
3    a.b.c.d.n();
4    a.b.foo(a.b.c.d);
5    a.b.bar();
6    a.b.c.d.m();
7    }
```

After

```
public void f() {
  D temp = a.b.c.d;
  temp.m();
  temp.n();
  a.b.foo(temp);
  a.b.bar();
  temp.m();
}
```

Compute complex (expensive) expression only once.

Extract Local Variable: Formally

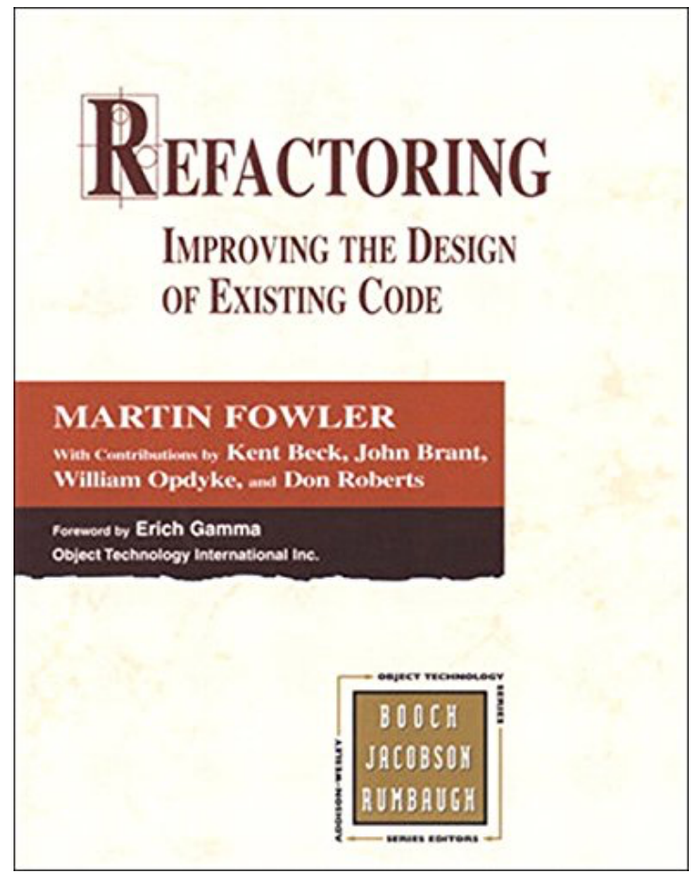
input : e – an expression of non-void type E
: S – a selection, as a list of consecutive statements
: *context* – the outermost, non-type scope containing S
output: *context* with e extracted to a local variable in S

```
v ← fresh variable name;
for s ∈ S do
    in s replace all occurrences of e with v;
end
add a new variable declaration E v = e context just before S;
```

Other OO Refactorings

Fowler has a *pretty loooong list* of refactorings in his book:

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Duplicate Observed Data
- Dynamic Method Definition
- Eagerly Initialized Attribute
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Module
- Extract Subclass
- Extract Superclass
- Extract Surrounding Method
- Extract Variable
- Form Template Method
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method
- Inline Module
- Inline Temp
- Introduce Assertion
- Introduce Class Annotation
- Introduce Expression Builder
- Introduce Foreign Method
- Introduce Gateway
- Introduce Local Extension
- Introduce Named Parameter
- Introduce Null Object
- Introduce Parameter Object
- Isolate Dynamic Receptor
- Lazily Initialized Attribute
- Move Eval from Runtime to Parse Time
- Move Field
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Recompose Conditional
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Middle Man
- Remove Named Parameter
- Remove Parameter
- Remove Setting Method
- Remove Unused Default Parameter
- Rename Method
- Replace Abstract Superclass with Module
- Replace Array with Object
- Replace Conditional with Polymorphism
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation With Hierarchy
- Replace Delegation with Inheritance
- Replace Dynamic Receptor with Dynamic Method Definition
- Replace Error Code with Exception
- Replace Exception with Test
- Replace Hash with Object
- Replace Inheritance with Delegation
- Replace Loop with Collection Closure Method
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
- Replace Nested Conditional with Guard Clauses
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Record with Data Class
- Replace Subclass with Fields
- Replace Temp with Chain



https://refactoring.com/catalog/

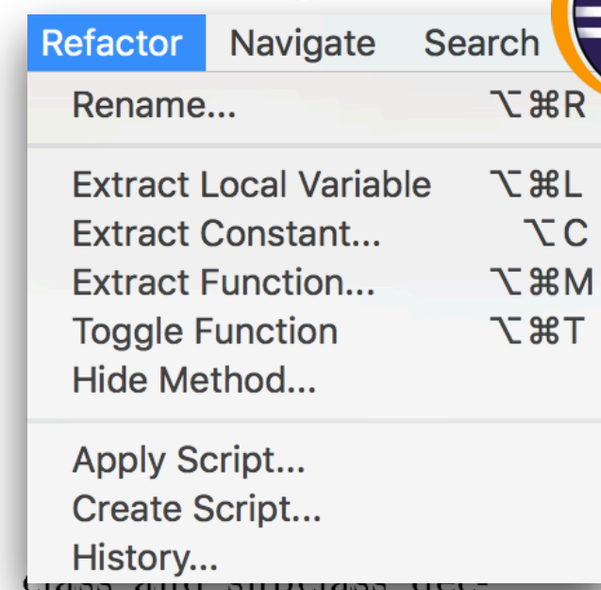
Eclipse Refactoring: General Lifecycle

- Tooling in Eclipse through the *Language Toolkit (LTK)*
- Refactoring launched by user or script on a context
- Initial check whether refactoring is applicable in this context: `checkInitialConditions()`
- Configuration details provided by user interactively or through script
- Check if parameters are okay: `checkFinalConditions()`
- Calculate changes: `createChange()`
- Preview dialog; change application after confirmation
- Record action in Undo-history

Refactoring for C/C++

- We will see that processing C/C++ has it's very own special challenges…
- …refactoring even more.
- Opdyke: first thesis on "refactoring" in 1992
  - proposed a number of refactorings for C++
  - with conditions to guarantee behaviour preservation
- "Refactoring" first showed up in 1984

Opdyke: Refactoring Example



A. *change_class_name*

Change the name of a class.

Arguments: class C, string S.

This name change is reflected throughout the program (i.e. in class and subclass declarations, constructor and destructor functions, and the declarations of instances of the class).

Preconditions:

1. $\forall$ class $\in$ Program.classes,
   class.name $\neq$ S.
   (the new name doesn't clash with an already existing class)

Remember how you studied logic in your first year?

The precondition ensures distinct class names (satisfying program property two). Changing the name of a class does not change its behavior (satisfying program property seven). Other program properties are trivially preserved.

[1] W. Opdyke, "Refactoring Object-Oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

Challenges of Refactoring C Systems

- Typical C file contains not only **terminals** defined in the C grammar but also **preprocessor directives**
- **Conditional compilation:** include code selectively depending on the value of conditions evaluated during compilation.
- Example:

```
#ifdef A
int x = 0;
#else
int x = 1;
#endif
```

Code **before** preprocessing

A Preprocessing Odyssey…

- The C preprocessor takes as input a C file with directives and outputs pure C code, where directives have been stripped out and substituted accordingly:

```
#ifdef A
int x = 0;
#else
int x = 1;
#endif
```

A is enabled

```
int x = 0;
```

A is disabled

```
int x = 1;
```

Code **before** preprocessing

Code **after** preprocessing

- Refactoring needs to deal with preprocessor directives!

Challenges of Refactoring C Systems

- Tools that work on preprocessed C code have many disadvantages:
  - allowed refactorings very restricted;
  - directives can be irrecoverable lost or may no longer apply.
- If directives are irrecoverable, the code may become unmaintainable.

A. Garrido and R. Johnson. 2002. Challenges of Refactoring C Programs. In Proceedings of the 5th International Workshop on Principles of Software Evolution. ACM, 6–14.

A. Garrido and R. Johnson. 2013. Embracing the C Preprocessor during Refactoring. Journal of Software: Evolution and Process 25, 12 (2013), 1285–1304.

Code That Keeps You Awake At Night:

Disciplined vs undisciplined conditional annotations

```
#ifdef A
int x = 0;
#else
int x = 1;
#endif
```

Disciplined

```
int x =
#ifdef A
  0;
#else
  1;
#endif
```

Undisciplined

(Don’t write that code!)

Undisciplined Conditional Annotations Refactoring Example

```
if ( COND_1
#ifdef EXP
    ⊕ COND_2
#endif
){
   STMTS
}
```

```
int test = COND_1;
#ifdef EXP
test = test ⊕ COND_2;
#endif
if (test) {
   STMTS
}
```

(→) `test` is not used in the code

However, there is no automatic tool to apply them

Refactoring improves code quality

F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca: Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. Transactions on Software Engineering (2017). To appear.

Research Challenges

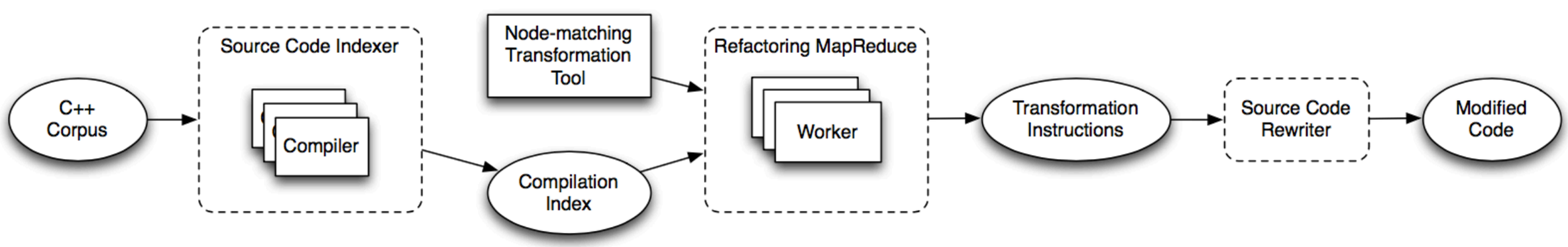
- Refactoring at Scale
- Correctness of Refactorings
- Repository mining
- Refactoring *models*
- Rearchitecting (forward engineering)
- (Rearchitecting to concurrency)
- (Machine Learning for refactoring)

Refactoring at Scale

- Refactoring of large code bases
- Example: migration to new APIs
- Semi-Automated refactoring:
  - candidates identified automatically, e.g. during Continuous Integration
  - automated application of suggestions to developers (e.g. through ticketing system)

ClangMR at Google

- Moving from old APIs to new ones…
- …or to C++11 standard constructs.
- Source code rejuvenation!
- “Large-Scale Automated Refactoring Using ClangMR”, Wright *et al.*, ICSM 2013.



ClangMR: Example & Statistics

```
void SplitStringUsing(const string& full,
                      const char* delimiters,
                      vector<string>* result);

void foo() {
  string input;
  vector<string> output;
  ...
  SplitStringUsing(input, "-", &output);
}
```

old internal API

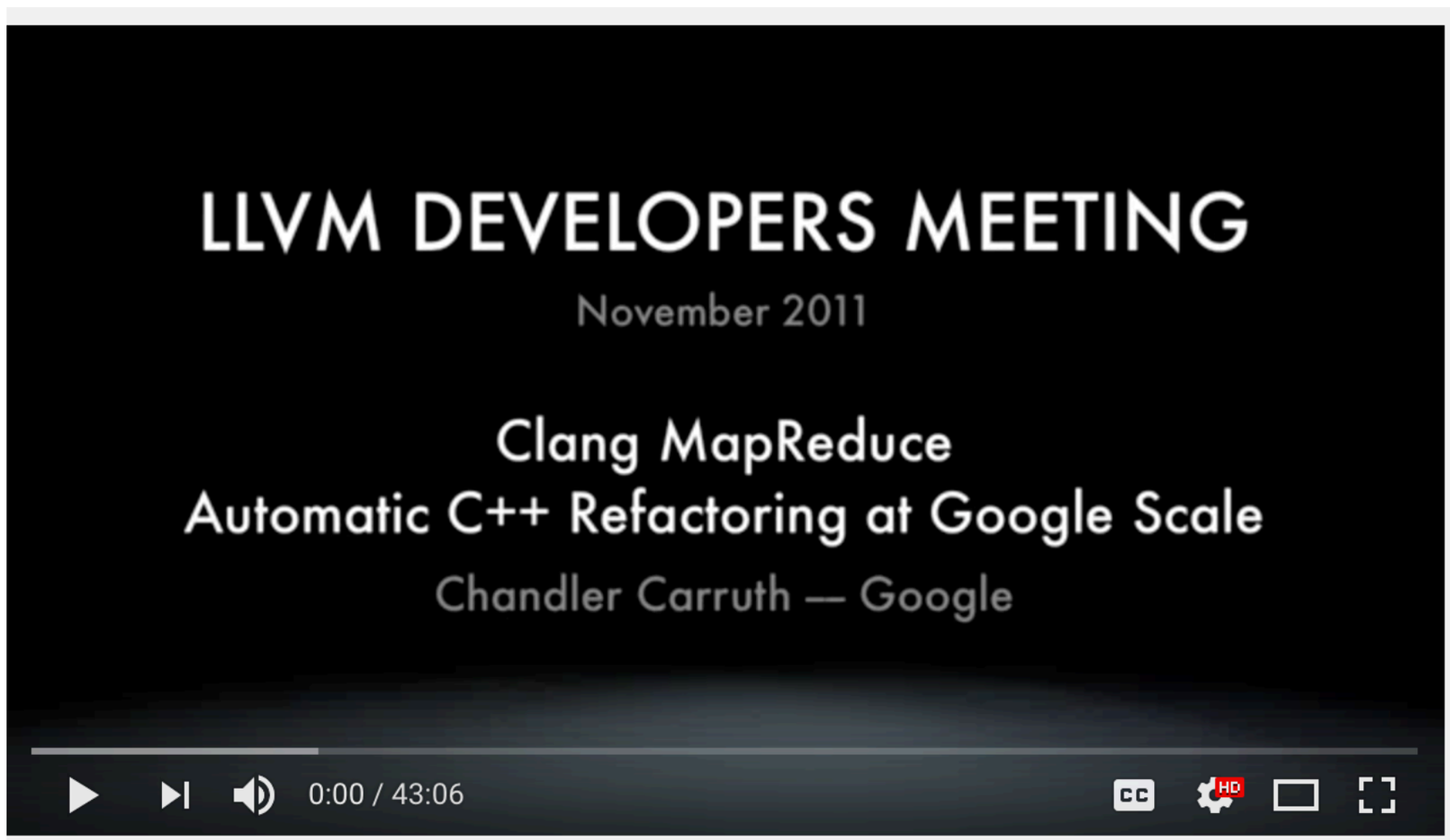
```
namespace strings {
template <typename Delimiter, Predicate>
Split(StringPiece text, Delimiter d, Predicate p);

struct SkipEmpty {
  bool operator()(StringPiece sp) const {
    return !sp.empty();
  }
};
}

void foo() {
  string input;
  ...
  vector<string> output =
    strings::Split(input, "-",
                   strings::SkipEmpty());
}
```

new `strings::Split()` API

- 45.000 callers
- 35.000 transformed
- 3.100 chunks
- largest part reviewed within 2 minutes

ClangMR Video

https://www.youtube.com/watch?v=mVbDzTM21BQ



Correctness of Refactorings
Is it really a refactoring?

"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its **observable** behaviour" [Fowler]

Challenge:
Can we predict the effect of applying a refactoring?

What could possibly go wrong here?

trunk/frontend/src/abs/frontend/typechecker/TypeChecker.jadd

r19783	r19896	
951	951	protected void Product.typeCheck(SemanticErrorList e) {
952	952	HashSet<String> featureNames = new HashSet<String>();
953		for (Feature f : getModel().getProductLine().getFeatures()) {
954		.getName());
	953	Model m = getModel();
	954	) {
	955	for (Feature f : m.getProductLine().getFeatures()) {
	956	featureNames.add(f.getName());
	957	}
955	958	}
956	959	HashSet<Stri oductNames = new HashSet<String>();
957		for (Produ prod : getModel().getProducts()) {
	960	for (Product prod : m.getProducts()) {
958	961	productNames.add(prod.getName());
959	962	}
960	963	HashSet<String> deltaNames = new HashSet<String>();
961		for (DeltaDecl delta : getModel().getDeltaDecls()) {
	964	for (DeltaDecl delta : m.getDeltaDecls()) {
962	965	deltaNames.add(delta.getName());
963	966	}

A Problem With Extract Local Variable…

```java
public class C {
    public X x = new X();
    {//initializer
       x.myC = this;
    }

    public void f(){
        x.n();
        x.m();
        x.n();
    }
}
```

```java
public class X{
    public C myC;

    public void m(){
        myC.x = new X();
    }

    public void n(){
        System.out.println(
         this.hashCode());
    }
}
```

Output:
1735600054
21685669

← Method `n()` called on two different objects

Refactored…

```java
public class C {
    public X x = new X();
    {//initializer
      x.myC = this;
    }

    public void f(){
        X temp = x;
        temp.n();
        temp.m();
        temp.n();
    }
}
```

```java
public class X{
    public C myC;

    public void m(){
        myC.x = new X();
    }

    public void n(){
        System.out.println(
         this.hashCode());
    }
}
```

Output now:
1735600054
1735600054

← Method `n()` called on *the same* object!
(changed behaviour)

Does Refactoring Need A Safety Net?

```java
public class C {
    public X x = new X();
    {//initializer
      x.myC = this;
    }
    public void f(){
        X temp = x;
        assert temp == x;
        temp.n();
        assert temp == x;
        temp.m();
        assert temp == x;
        temp.n();
```

```java
public class X{
    public C myC;

    public void m(){
        myC.x = new X();
    }

    public void n(){
        System.out.println(
         this.hashCode());
    }
}
```

Output now:
1735600054
Exception in thread ”main” java.lang.AssertionError

An Isolated Incident? Meet Extract-And-Move Method

```java
public class C {
    public X x = new X();
    {//initializer
      x.myC = this;
    }
    public void f(){
        x.bar(this);
    }
}
```

```java
public class X{
   ...
    void bar(C c){
        this.n();
        assert this == c.x;
        this.m();
        assert this == c.x;
        this.n();
    }
}
```

- Combination of two refactorings
- Extract has same effect of evaluating target only once
- But: different implementations actually get this right by using additional parameter instead of `this`!

Test Case Generation for Refactored Code

- General issue: need test cases for (modified) code
- Here: refactoring often introduces new units that need to be subjected to additional tests
- No direct call to refactored method in tests
  → less likely to discover fault
- High branch coverage
  → better chances of discovery

[Alves et al., 2017: Test coverage of impacted code elements for detecting refactoring faults: An exploratory study]

Correctness of Refactorings Is it implemented correctly?

- Implementation of refactorings often difficult
- Many syntactical elements in AST:
  - missing corner cases
  - mistakes
- Eclipse JDT bugtracker issues on refactorings
- Nice separation between UI and parameter objects
- What was the *specification* of your refactoring anyway?

Correct implementation

- Schäfer *et al.*: JRRT (based on JastAdd)
- Natural language description vs. implementation
- (Issue: refactoring with same name slightly different implemented)

```
class A {

  void m() {
    int f = 23;
    ...
  }
}
```

⇒

```
class A {
  int f;
  void m() {
    f = 23;
    ...
  }
}
```

```
class A {
  int f(int y) {
    if(y <= 1)
      return 1;
    int x = y;
    return f(y-1) * x;
  }
}
```

⇒ ↯

- Example: Promote-variable-to-field
- Not always correct/possible

Better Implementations

Algorithm 1 PROMOTE TEMP TO FIELD(d : *LocalVar*)

Input Language: Java
Output Language: Java ∪ locked dependencies

1: ⌊SPLIT DECLARATION⌋(d)
2: $d' \leftarrow$ new **private** field of same type and name as d
3: make d' **static** if d is in static context
4: ⌊INSERT FIELD⌋(hostType(d), d')
5: **for all** uses u of d **do**
6: lock u onto d'
7: lock reaching definitions of u
8: REMOVE DECL(d)

Implemented through JastAdd meta-compilation system

```
public void VariableDeclaration
  .promoteToField()
{
  split();
  Modifiers mods = new Modifiers("private");
  if(inStaticContext())
    mods.addModifier("static");
  TypeAccess ta = type().createLockedAccess();
  FieldDeclaration f
    = new FieldDeclaration(mods, ta, name());
  hostType().insertField(f);
  for(VarAccess va : uses()) {
    va.lock(f);
    va.lockReachingDefs();
  }
  flushCaches();
  remove();
}

public void VariableDeclaration
  .doPromoteToField()
{
  Program root = programRoot();
  promoteToField();
  root.eliminate(LOCKED_NAMES, LOCKED_FLOW);
}
```

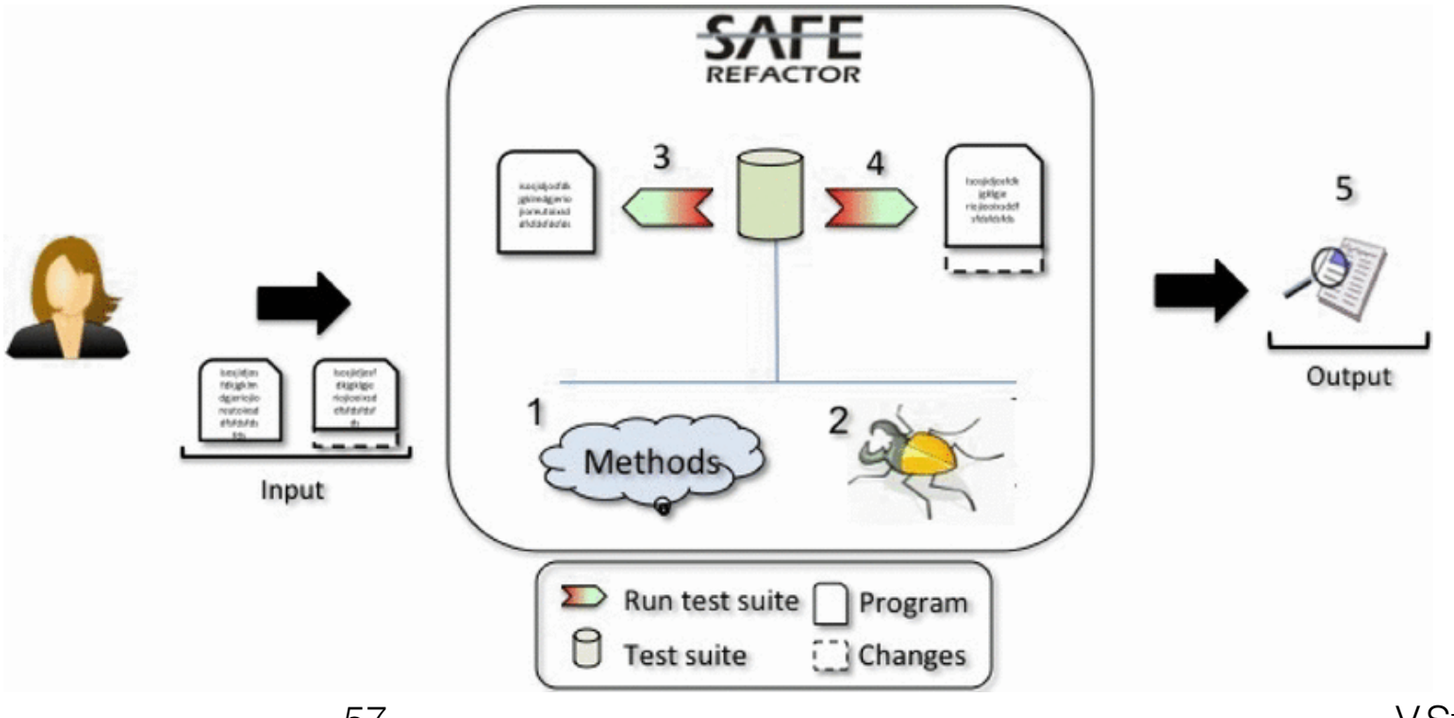
Repository Mining

- Analyse (open source) repositories
- Example: GitHub (good API?)
- Analyse individual changes:
  - classify type of change — development, or refactoring?
  - “understand” commit message?
  - behaviour preserving?

Repository Mining

G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, T. Massoni:
Analyzing Refactorings on Software Repositories

- compare two versions of the software via test suite
- transformation considered a refactoring if no behavioural difference detected between source and target programs
  - requires execution
  - classification?



Repository Mining

Average of:
refactoring frequency, granularity, scope, and test coverage

Subject	Refactoring (%)	Low Level (%)	Local (%)	Average Test Coverage (%)
JHotDraw	20.32	55	61	35
ArgoUML	33	70	75	30
jEdit	23.48	45	60	27
SweetHome 3D	29.30	52	62	35
HSQLDB	25.70	71	80	31
Average	27.27	63.83	71.05	30.95

Soares et al.: Analyzing Refactorings on Software Repositories

Refactoring Models

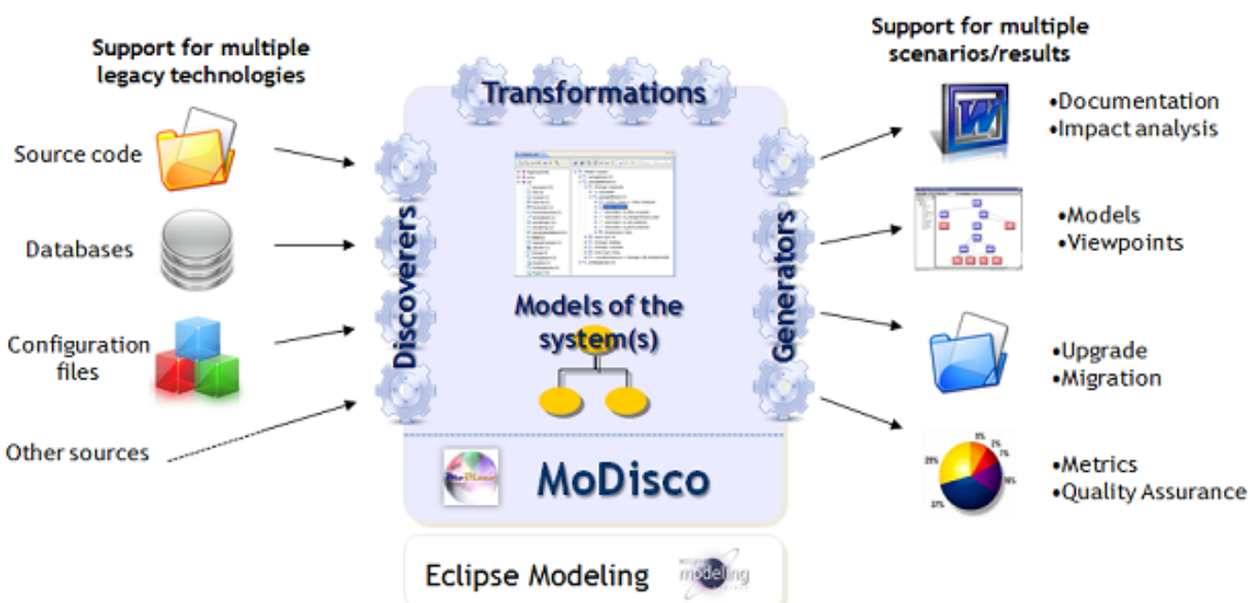
- Refactoring = Model Transformation
- Refactoring models — we're working e.g. on UML diagrams.
- Refactoring *through* models — we don't work on the source, but an intermediate representation.
  Example: class diagram derived from Java code.
- Refactoring *meta*models and instances.

Refactoring in MDSE

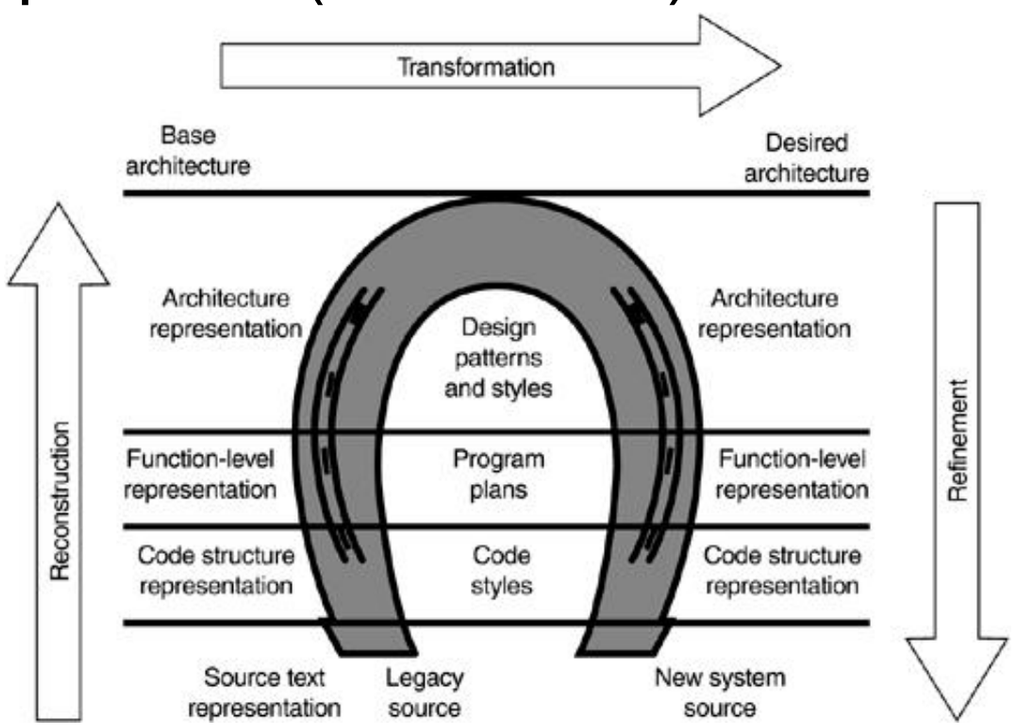
- Based on Graph Theory
- Higher level of abstraction, hence simpler reasoning
- Refactorings as model-to-model transformations
- Applicable to ***general-purpose*** modelling languages (e.g. UML) and ***domain-specific*** modelling languages
- Applicable to low-level or big-scale refactoring (reverse engineering and modernisation)
- Easier specification and visualisation of results

Reverse Engineering of Legacy Systems

- MoDisco Text-to-Model transformations



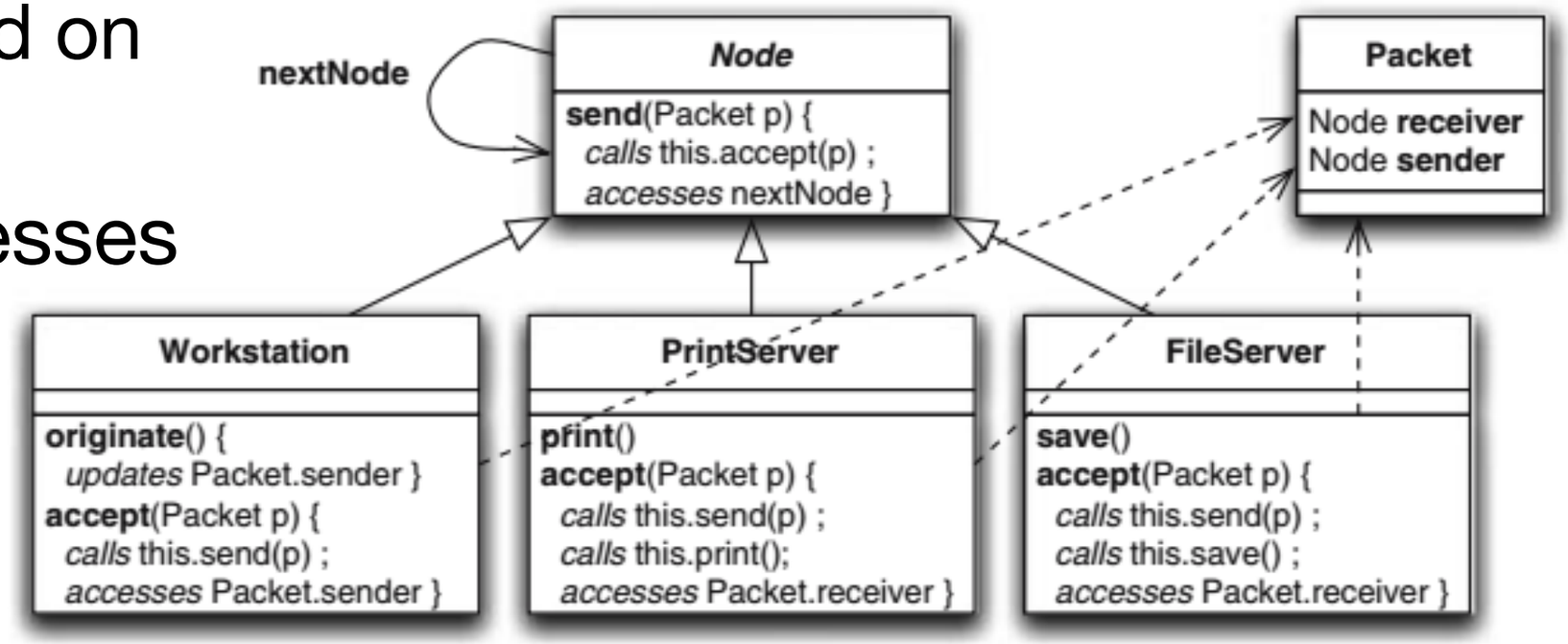
- Three-phase reengineering process (horseshoe)



Source: Erik Philippus

Model-based Refactoring of Source Code

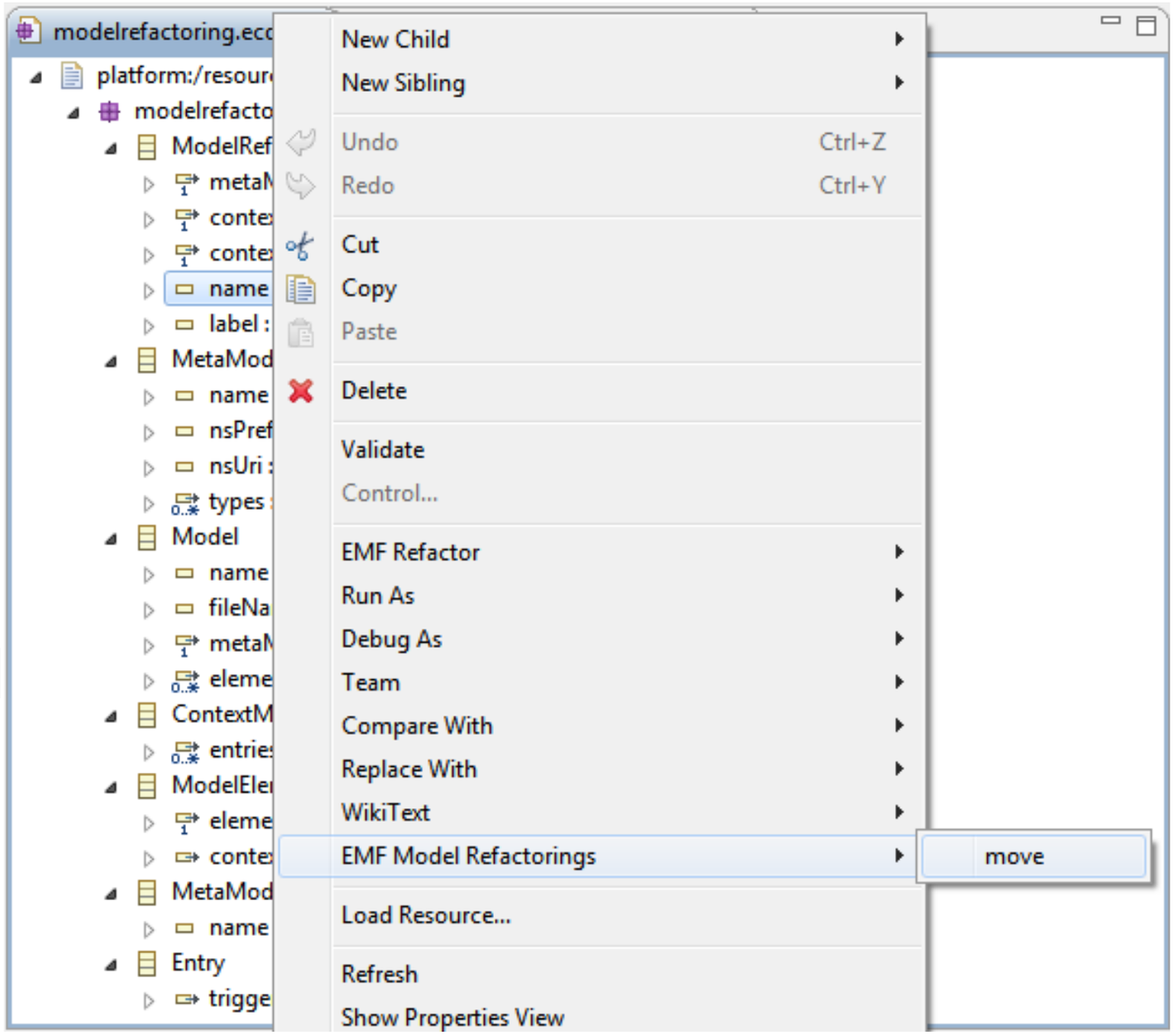
- Based on the analysis of annotated UML models
- Coordination of refactoring operations based on dependencies
  - Variable accesses
  - Method calls



T. Mens, G. Taentzer, O. Runge: "Analysing refactoring dependencies using graph transformation." *Software and Systems Modeling,* 2007

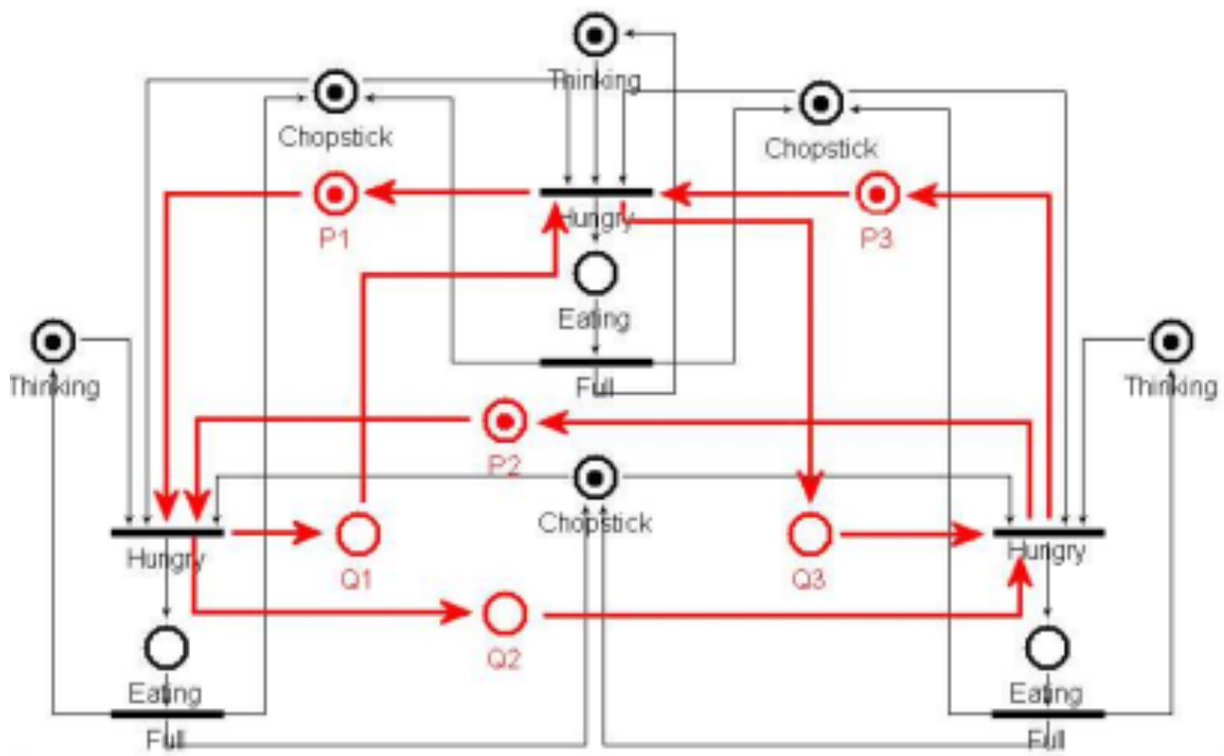
Refactoring of General-Purpose Languages

- Support in EMF through EMF Refactor
- Refactorings based on class diagrams (Ecore metamodels)
- Extensible framework for any EMF-based model



Source: eclipse.org

Refactoring of DSMLs

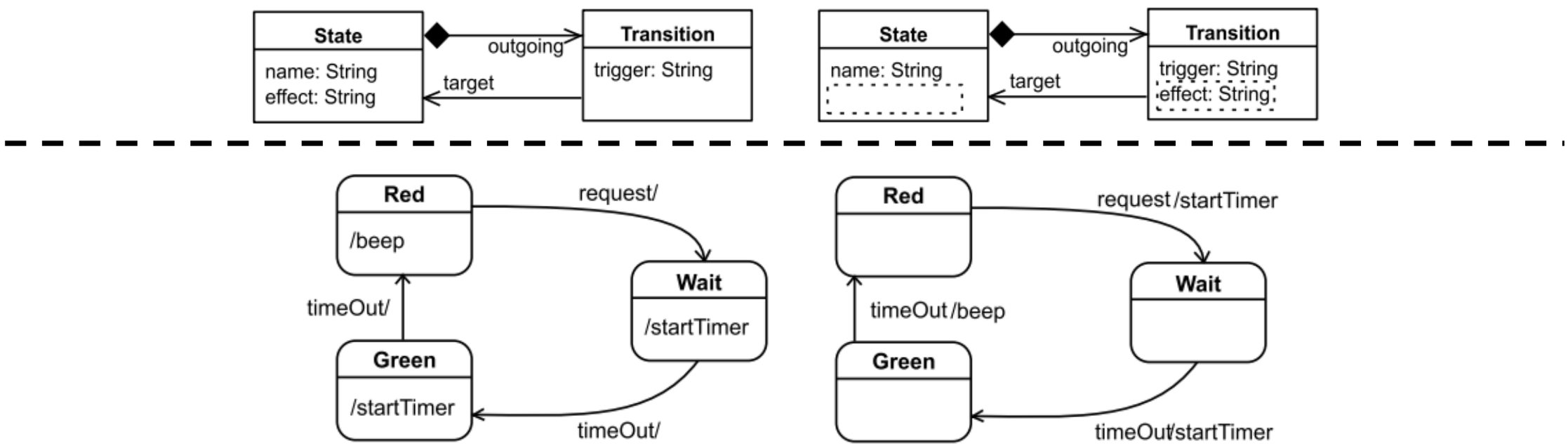



```
strategy freeStarvation()
{
    declare dstList:modelList;
    declare dst1,dst2,p,q:model;
    declare static num:Integer;
    dstList:=findOutConnections();
    assert(dstList.size()==2);
    dst1:=dstList.get(0);  dst2:= dstList.get(1);
    p:=createModel("InitMarker","P"+intToStr(num));
    q:=createModel("Place", "Q" + intToStr(num));
    num:=num + 1;
    addConnection(dst1,p);  addConnection(p,dst2);
    addConnection(dst2,q);  addConnection(q,dst1);
}
```

J. Zhang, Y. Lin, J. Gray: “Generic and domain-specific model refactoring using a model transformation engine”, 2005

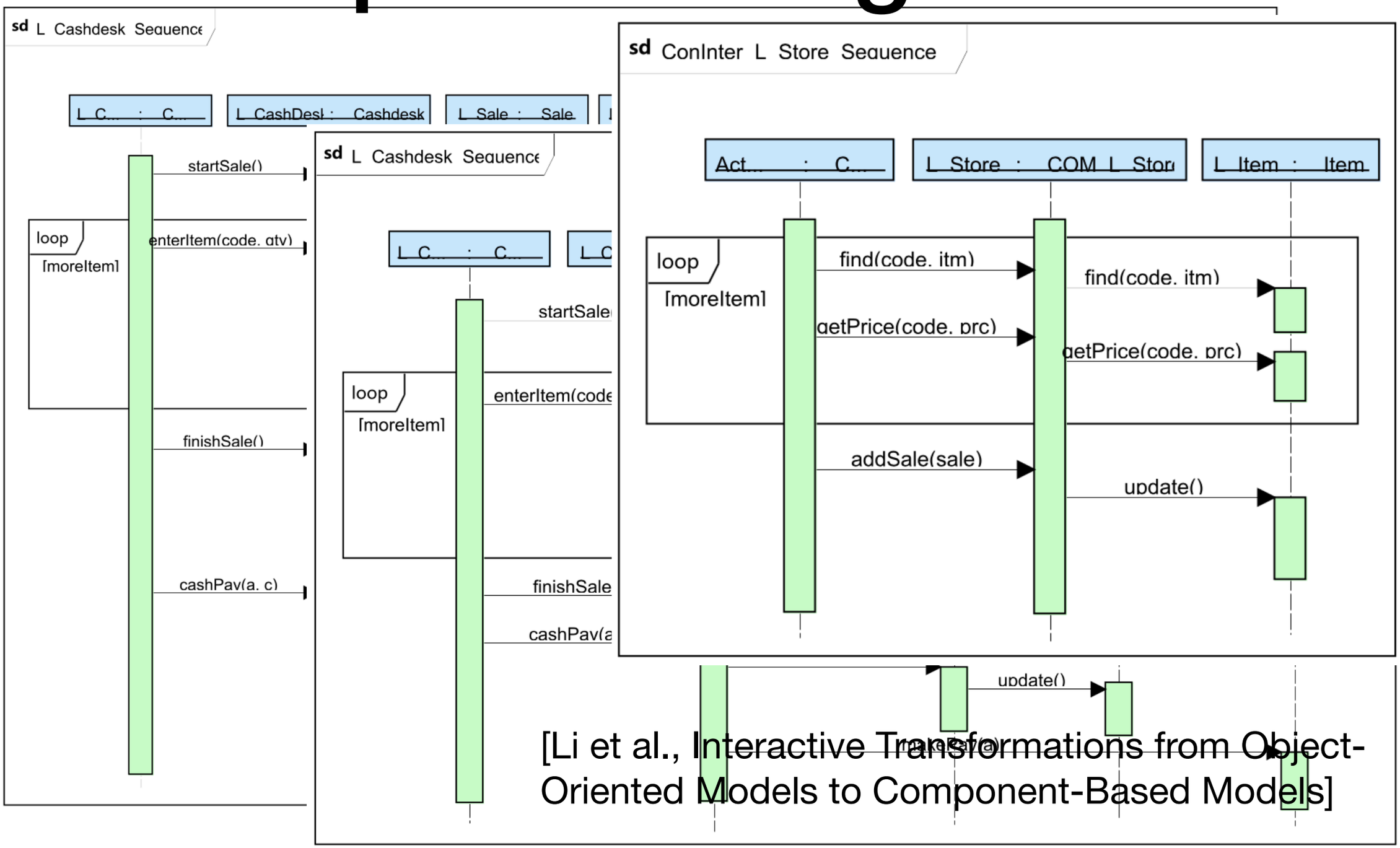
Refactoring of Metamodels and Their Instances

- AKA co-evolution AKA model migration



Mantz et al. “Co-evolving meta-models and their instance models: A formal approach based on graph transformation”, 2015

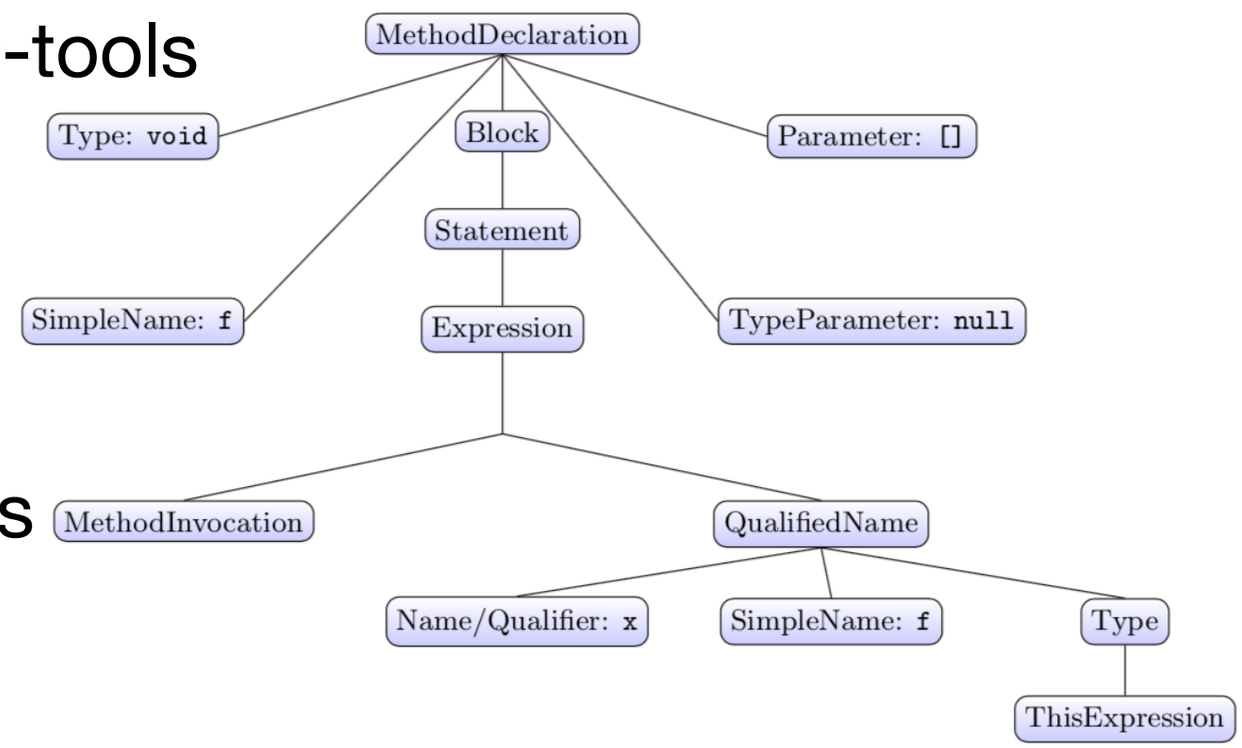
Refactoring an UML Sequence Diagram



[Li et al., Interactive Transformations from Object-Oriented Models to Component-Based Models]

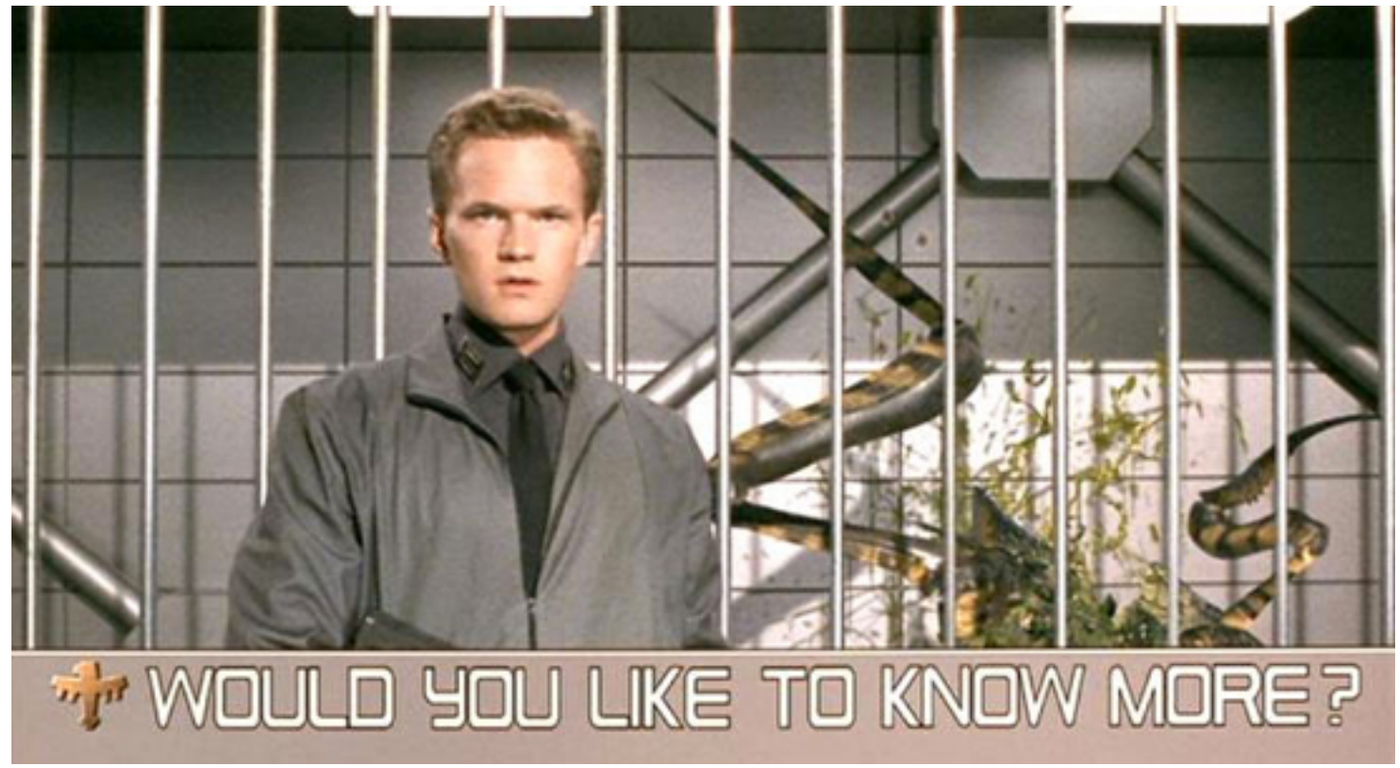
Refactorings = Model Transformation?

- Many existing approaches for graph (tree!) -based models
- AST = model
- Can use OCL and other MDD-tools to describe transformations
- Limitation:
  - many syntactical elements → complex transformations
  - formal reasoning difficult
    - *refactoring always correct* vs. *instance of refactoring correct* (with proof obligations)



Interested?

- Refactorings are an important software engineering topic.
- Tool support for refactoring *always* needs improvement.



- Requirements:
  - *compiler construction* (to work with programs as input: syntax & semantics, grammars & parsing, ASTs, typing rules)
  - *logic/discrete maths* (∀,∃,∈,… to read & write specifications)
  - optional: *static analysis* (information flow analyses etc., also useful for security analysis)
  - most of all: interest in *coding*!

Bibliography (1)

Foundations:

- W. F. Opdyke. Refactoring object-oriented frameworks. Technical Report GAX93-05645, University of Illinois at Urbana-Champaign, 1992.
- D. Roberts, J. Brant, and R. Johnson. A refactoring tool for SmallTalk. Theory and Practice of Object Systems, 3(4):253–263, 1997.
- M. Fowler. Refactoring: improving the design of existing code. Addison-Wesley, 1999.
- E. Gamma, J. Helm, R. Johnson, R. Vlissides: “Design Patterns: Elements of Reusable Object-Oriented Software”, 1994
- J. Kerievsky: “Refactoring to patterns”, Addison-Wesley, 2005
- M.O’Keeffe and M.Ó.Cinnéide. Search-based refactoring: An empirical study. J. of Software Maintenance and Evolution, 20(5):345–364, 2008.
- P. Pirkelbauer, D. Dechev, B. Stroustrup: Source Code Rejuvenation Is Not Refactoring. SOFSEM 2010
- M. Kim, T. Zimmermann, N. Nagappan: A field study of refactoring challenges and benefits. Intl. Symp. on the Foundations of Softw. Eng. ACM, 2010
- G.C. Murphy, M. Kersten, L. Findlater: How are Java software developers using the Elipse IDE? IEEE Software 23(4), 2006.
- E. Tempero, T. Gorschek, L. Angelis: Barriers to Refactoring. Communications of the ACM, Vol. 60 No. 10, 2017

Bibliography (2)

“Modern” Reading:

- M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In 34th Intl. Conf. on Software Engineering (ICSE 2012). IEEE, 2012.
- E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. Software Engineering, IEEE Transactions on, 38(1):5–18, 2012.
- J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust must be earned. IEEE Software, 32:80–83, 2015.
- G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, T. Massoni: Analyzing Refactorings on Software Repositories. SBES 2011
- G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. IEEE Software, 27(4):52–57, 2010.
- M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba. Making refactoring safer through impact analysis. SCP, 93:39–64, 2014.
- T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. Software & Systems Modeling, 6(3):269–285, 2007.
- M. Schäfer and O. de Moor. Specifying and implementing refactorings. In Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’10. ACM, 2010.
- A. M. Eilertsen, A. H. Bagge, and V. Stolz. Safer refactorings. In Proc. of the Intl. Symp. On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), LNCS. Springer, Oct 2016.

Bibliography (3)

- F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca: Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. Transactions on Software Engineering (2017).
- A. Garrido and R. Johnson. 2002. Challenges of Refactoring C Programs. In Proceedings of the 5th International Workshop on Principles of Software Evolution. ACM, 6–14.
- A. Garrido and R. Johnson. 2013. Embracing the C Preprocessor during Refactoring. Journal of Software: Evolution and Process 25, 12 (2013), 1285–1304.
- H. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. Large-scale automated refactoring using ClangMR. Intl. Conf. on Software Maintenance. IEEE, 2013.
- E.L.G. Alves, T. Massoni, P.D. de Lima Machado: Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. J. Systems and Software 123, 2017.
- D. Li, X. Li, Z. Liu, V. Stolz: Interactive Transformations from Object-Oriented Models to Component-Based Models. FACS 2011, LNCS, Springer 2011.
- F. Mantz, et al.: Co-evolving meta-models and their instance models: A formal approach based on graph transformation. Science of Computer Programming 104 (2015): 2-43.
- J. Zhang, Y. Lin, J. Gray: Generic and domain-specific model refactoring using a model transformation engine. Model-driven Software Development 23 (2005).