Refactoring is the behaviour-preserving change of (a program's) design.

Move Class (to another package)

```java
package a;
package b;

class A {
    B b;
}
class B {
}
```

access violation

Move Class (to another package)

```java
package a:
class A {
    public void m(Object o)
        { ... }
    void m(String s) { ... }
}

class B {
    new A().m("a");
}
```

class B {
    new A().m("a");
}

class B {
    new A().m("a");
}

class B {
    new A().m("a");
}

change of static binding
Move Class (to another package)

```java
package a;

public class A {
    public void m() { ... }
    public void n() {
        m();
    }
}

class B extends A {
    void m() { ... }
}

package b;

change of dynamic binding

class C extends B {
    void m() { ... }
}
```

Basic program elements

A program $P$ consists of
- a set $D$ of declared entities and
- a set $R$ of references to elements of $D$.

Every $d \in D$ has
- a declared accessibility, $d.\alpha$ and
- a location, $d.\lambda$, in $P$.

Every $r \in R$ has
- a location, $r.\lambda$, in $P$.

Constraint variables

- $d.\alpha$, the declared accessibility of a declared entity
  - $d.\alpha \in A = \{ \text{absent, private, package, protected, public} \}$
  - absent $<$ private $<$ package $<$ protected $<$ public
- $d.\lambda$, the location of a declared entity
- $r.\lambda$, the location of a reference
- $\alpha : L \times L \rightarrow A$, a function computing required accessibility

Constraint variables example

```java
class A {
    double d;
}

class B {
    A a;
    { a.d = 1.0; }
}
```

Constraint rules

- accessing:
  - $d.\alpha \geq \alpha(r.\lambda, d.\lambda)$
  - $d.\alpha \geq \alpha(a.d.\lambda, d.\lambda)$
  - $d.\alpha \geq \alpha(B, A) = \text{package}$
Constraint rules

- inheritance:
  \[ \text{binds}(r, d) \quad \text{rtype}(r, t) \]
  \[ d.\alpha \geq \alpha(t, d.\lambda) \]

- subtyping:
  \[ \text{overrides}(d', d) \lor \text{hides}(d', d) \]
  \[ d'.\alpha \geq d.\alpha \]

- dynamic binding:
  \[ \text{overrides}(d', d) \]
  \[ d.\alpha \geq \alpha(d', \lambda) \]

- arrays:
  \[ T[\cdot].\alpha = T.\alpha \]

- multiple declarations such as
  \[ i.\alpha = j.\alpha \]

- admissible access modifiers:
  \[ d.\alpha \in A(d, d.\lambda) \]

- main methods, imported elements, several top-level types in a compilation unit, …

Basic idea

- Generate accessibility constraints from program as is.
  - The variable assignments derived from the program as is present a solution of the constraint set.
- Assign variables new values that reflect the intended refactoring.
- Check whether constraint system is still solved; if not and user permits, try to solve it.
- Write back changed values, or deny refactoring.

Example: Move Class

```java
package a;

class A {
    private double d;
    B b;
    { b.d = 1.0; }
}

class B extends A {
}
```

Example: Pull Up Method

```java
package a;

public class A {
    public void m(Object o) {...}
    void m(String s) {...}
}

class B extends A {
    void n() {
        m("abc");
    }
}
```

```java
package b;

class A {
    public void m(Object o) {...}
    void m(String s) {...}
}

class B extends A {
    void n() {
        m("abc");
    }
}
```
What we have

- program transformations …
  - ... resulting in a compiling program
  - ... and not altering the program's behaviour

Steimann, Thies: ECOOP (2009)

Mutant generation is non-trivial

```
class A {
    private void m(String s) {...}
}
class B extends A {
    void m(Object o) {...}
    void main() {
        new A().m("abc");
    }
}
```

Will it compile?
Will it likely change the program's behaviour? (be non-equivalent)
Will multiple mutants point to same missing test case? (redundant mutants)

Steimann, Thies: ICSE (2010)

What we need

- program transformations …
  - ... resulting in a compiling program
  - ... and not altering the program's behaviour

Steimann, Thies: ECOOP (2009)

Constraint-based mutant generation

```
class A {
    private void m(String s) {...}
    void m(Object o) {...}
}
class B {
    void main() {
        new A().m("abc");
    }
}
```

Constraint-based mutant generation

```
class A {
    private void m(String s) {...}
    void m(Object o) {...}
}
class B {
    void main() {
        new A().m("abc");
    }
}
```

The Class A must not be declared with protected accessibility level.
Otherwise: Compiler error

Steimann, Thies: ICSE (2010)
Constraint-based mutant generation

A. \( \alpha \neq \text{protected} \)

\[
\text{class A} \\
\quad \text{private void } m(String s) \{ \ldots \} \\
\quad \text{void } m(Object o) \{ \ldots \}
\]

A. \( \alpha \geq \alpha(B, \lambda, A, \lambda) \)

class B \\
\quad \text{void main}() \\
\quad \quad \{ \text{new A().m("abc");} \}

A. m(String). \( \alpha < \alpha(B, \lambda, A, \lambda) \)

Constraint-based mutant generation

A. m(String). \( \alpha \leq \alpha(B, \lambda, A, \lambda) \)

Otherwise:

Change of binding

Evaluation: Mutant filtering

![Graph showing mutant filtering results]

Evaluation: Runtime

![Graph showing runtime results]

Evaluation: Completeness

- complete iff the constraint rules model the language specification completely
- constraint rules tested by thousands of automatic refactoring applications on open source projects and automatic check for unchanged behaviour
  - with the help of the compiler
  - with the help of the provided JUnit-Tests

use tests to check the completeness of an approach that checks the completeness of (other) tests
Incorporating other constraint rules

Type Constraints [Tip et al. 2003]

```java
class A {
    String m(A a) {...}
}
class B extends A {
    String m(A a) {...}
}
```

A.m(A) overrides B.m(A) so the parameter types of both methods must be equal.

```java
new B().m(new A())
```

Enhancing Mutant Reports

```java
public class A {
    String m(String s) { return "a"; }
}

class B extends A {
    String m(Object o) { return "b"; }
}
```

```java
@Test
public void testM() {
    // TODO Auto-generated test object
    B b = null;
    // TODO Auto-generated test parameter
    String param1 = null;
    // TODO Auto-generated test expectation
    String expected = null;
    Assert.assertEquals(expected, b.m(param1));
}
```

Thank you!
Questions?