Transformation issues found in the JRRT's implementations

Melina Mongiovi
Advisor: Rohit Gheyi

Federal University of Campina Grande, Brazil
melina@copin.ufcg.edu.br

1 Bugs

Listing 1.1. Before refactoring.

```
public abstract class A {
  public abstract void m();
}

abstract class B extends A {}
```

Listing 1.2. Resulting program.

```
public abstract class A {}

abstract class B extends A {
  public abstract void m();
}
```

Fig. 1. After pushing down method m from class A to class B using JRRT (03/feb/2013), the class A needs no longer be abstract but the transformation does not remove the modifier.

Listing 1.3. Before refactoring.

```
class A {
  private int f = 10;
}
```

Listing 1.4. Resulting program.

```
class A {
  private int f = 10;

  public int setF(int f) {
    return this.f = f;
  }

  public int getF() {
    return f;
  }
}
```

Fig. 2. JRRT (03/feb/2013) encapsulates a private field.

Listing 1.5. Before refactoring.

```
public abstract class A {
  public abstract int m();
}

class B extends A {
  public int m() {
    return 0;
  }
}
```

Listing 1.6. Resulting program.

```
public abstract class A {}

class B extends A {
  public int m() {
    return 0;
  }
}
```

Fig. 3. Applying the push down method refactoring to move method m from class A to a concrete class B using JRRT (03/feb/2013), removes it from the program.

Listing 1.7. Before refactoring.

```
public abstract class A {}

abstract class B extends A {
  public abstract int m();
}

class C extends A {}
```

Listing 1.8. Resulting program.

```
abstract public class A  {
  abstract public int m();
}

abstract class B extends A
{}

abstract class C extends A
{}
```

Fig. 4. Applying the pull up method refactoring to move method m from class B to class A using JRRT (03/feb/2013), makes class C abstract.

Listing 1.9. Before refactoring.

```
public abstract class A {
  public abstract int m();
}

abstract class B extends A {
  public int m() {
    return 0;
  }
}
```

Listing 1.10. Resulting program.

```
public abstract class A {}

abstract class B extends A {
  public int m() {
    return 0;
  }
}
```

Fig. 5. Applying the push down method refactoring to move method m from class A to an abstract class B using JRRT (03/feb/2013), removes it from the program.

Listing 1.11. Before refactoring.

```
public interface A {}

abstract class B implements A {
  public abstract int m();
}
```

Listing 1.12. Resulting program.

```
public interface A {}

abstract class B implements A {}
```

Fig. 6. Applying the push down method refactoring to move method m from class B to interface A using JRRT (03/feb/2013), removes the method.

Listing 1.13. Before refactoring.

```
public class A {
  public int f;
}
```

Listing 1.14. Resulting program.

```
public class A {
  private int f;
  private int getF() {
    return f;
  }
  private int setF(int f) {
    return this.f = f;
  }
}
```

Fig. 7. Encapsulating field A.f JRRT (03/feb/2013), creates a set method returning the field.

Listing 1.15. Before refactoring.

```
public class A {
  public int k(long a) {
    return 1;
  }
}

abstract class B extends A {
  public abstract int m(int a);
}

class C extends A {}
```

Listing 1.16. Resulting program.

```
public abstract class A {
  public int k(long a) {
    return 1;
  }
  public abstract int m(int a);
}

abstract class B extends A {}

abstract class C extends A {}
```

Fig. 8. Applying the pull up method refactoring to move method m from class B to class A using JRRT (03/feb/2013), makes classes A and C abstract.