Generic Traits for the Java Platform

Lorenzo Bettini and Ferruccio Damiani

Dipartimento di Informatica, Università di Torino, Italy

CINA, July, 2014.
To appear in PPPJ 2014.

Traits

- introduced by Schärli et al., in SQUEAK/SMALLTALK
  - play the role of *units for behavior fine-grained reuse*, in order to counter the problems of class-based inheritance with respect to code reuse
- a trait is
  - a set of methods, independent from any class hierarchy;
  - the common methods can be factored into a trait;
  - traits can be composed to form other traits or classes.

Traits

- introduced by Schärli et al., in SQUEAK/SMALLTALK
  - play the role of *units for behavior fine-grained reuse*, in order to counter the problems of class-based inheritance with respect to code reuse
- a trait is
  - a set of methods, independent from any class hierarchy;
  - the common methods can be factored into a trait;
  - traits can be composed to form other traits or classes.
- Traits can be composed in arbitrary order
- composed traits must resolve possible name conflicts explicitly.

XTRAITJ

- a JAVA-like language for *pure trait-based programming*;
- the declarations of object type, behavior and instance generation are completely separated:
  - *Interfaces*, as pure types, defining only method signatures.
  - *Traits*, as pure units of behavior reuse, defining only methods.
  - *Classes*, as pure generators of instances, implementing interfaces by using traits, and defining fields and constructors.

Design choices

- Full interoperability with Java and its type system
- This allows to reuse all Java libraries and an incremental adoption of traits
- Full IDE support (Eclipse); IDE tooling is crucial for Agile programming

Design choices

- Full interoperability with Java and its type system
- This allows to reuse all Java libraries and an incremental adoption of traits
- Full IDE support (Eclipse); IDE tooling is crucial for Agile programming

XTEXT & XBASE

- XTEXT: a framework for language implementations with full Eclipse integration
- XBASE: a reusable JAVA-like expression language
  - interoperable with the JAVA type system (including generics)
  - clean and "less noisy" JAVA-like syntax, with some bonus features (type inference, lambda expressions, extension methods)

Pure Trait programming

- no class hierarchies and consequently no class-based inheritance;
- multiple inheritance is obtained using traits;
- the composite unit has complete control over the composition and must resolve conflicts explicitly.

- Traits are not types;
- to maximize the opportunity for reuse

Pure Trait programming

- no class hierarchies and consequently no class-based inheritance;
- multiple inheritance is obtained using traits;
- the composite unit has complete control over the composition and must resolve conflicts explicitly.

- Traits are not types;
- to maximize the opportunity for reuse

Avoid statically unintentional name clashing.

Promote code reuse

- programs may look more verbose than standard class-based programs;
- however, the degree of reuse provided by traits is higher than class-based hierarchies;
- the distinction of each programming concept in a separate entity makes each component reusable in different contexts, in an unanticipated way;
- on the contrary, class hierarchies need to be designed from the start with a specific reuse scenario in mind.

Traits and operations

- a trait consists of
  - *provided methods* (methods defined in the trait),
  - *required methods* (abstract methods assumed to be available in a trait or a class using the trait),
  - *required fields* (fields assumed to be available in a class using the trait),
  - the required fields and the required/provided methods can be directly accessed in the body of the trait's provided methods.
- Traits do not specify any state.
- Traits are building blocks to compose classes and other, more complex, traits.
- A suite of trait composition operations allows the programmer to build classes and composite traits.
- The composite unit has complete control over conflicts that may arise during composition and must solve them explicitly.

XTRAITJ classes

- A class in XTRAITJ can
  - implement interfaces by using traits
  - can define fields and constructors
  - but it cannot define methods

XBASE for method bodies

- an extensible and reusable statically typed expression language
- a Java with "less noise"
  - type inference, lambda expressions, extension methods
- integrates completely with Java and Eclipse JDT
- full access to Java type system (including generics)

XBASE for method bodies

- an extensible and reusable statically typed expression language
- a Java with "less noise"
  - type inference, lambda expressions, extension methods
- integrates completely with Java and Eclipse JDT
- full access to Java type system (including generics)

Example

```
val personList = newArrayList(
        new Person("James", "Smith", 50),
        new Person("John", "Smith", 40),
        new Person("James", "Anderson", 40),
        new Person("John", "Anderson", 30),
        new Person("Paul", "Anderson", 30))
personList.filter[firstname.startsWith("J")].
    sortBy[age].take(3).map[surname + ", " + firstname].
        join("; ")
```

Examples of traits

```
trait T1 {
    void upd(int n);   // required method
    void inc(int n) {  // provided method
        if (n > 0) this.upd(n) else errs=errs+1
    }
    int errs;   // required field
}
```

Examples of traits

```
trait T1 {
    void upd(int n);   // required method
    void inc(int n) {  // provided method
        if (n > 0) this.upd(n) else errs=errs+1
    }
    int errs;   // required field
}

trait T2 {
    void upd(int n) { // provided method
        counter = counter + 1;
    }
    int counter;   // required field
}
```

XTRAITJ classes

- A class in XTRAITJ can
  - implement interfaces by using traits
  - can define fields and constructors
  - but it cannot define methods

XTRAITJ classes

- A class in XTRAITJ can
  - implement interfaces by using traits
  - can define fields and constructors
  - but it cannot define methods

```
interface ICounter {
    /* Increment the counter by n, if n > 0.
     * Otherwise, increment the number of errors */
    void inc(int n);
}
```

XTRAITJ classes

- A class in XTRAITJ can
  - implement interfaces by using traits
  - can define fields and constructors
  - but it cannot define methods

```
interface ICounter {
    /* Increment the counter by n, if n > 0.
     * Otherwise, increment the number of errors */
    void inc(int n);
}
class Counter implements ICounter uses T1, T2 {
    int counter = 0;
    int errs = 0;

    Counter(int counter) {
        this.counter = counter;
    }
}
```

The complete example

```
trait T1 {
    void upd(int n);   // required method
    void inc(int n) {  // provided method
        if (n > 0) this.upd(n) else errs=errs+1
    }
    int errs;   // required field
}
```

```
trait T2 {
    void upd(int n) { // provided method
        counter = counter + 1;
    }
    int counter;   // required field
}
```

```
interface ICounter {
    /* Increment the counter by n, if n > 0.
     * Otherwise, increment the number of errors */
    void inc(int n);
}

class Counter implements ICounter uses T1, T2 {
    int counter = 0;
    int errs = 0;

    Counter(int counter) {
        this.counter = counter;
    }
}
```

Flattening semantics

```
trait T1 {
    void upd(int n);   // required method
    void inc(int n) {  // provided method
        if (n > 0) this.upd(n) else errs=errs+1
    }
    int errs;   // required field
}
class Counter implements ICounter uses T1, T2 {
    int counter = 0;
    int errs = 0;

    Counter(int counter) {
        this.counter = counter;
    }
}
```

```
trait T2 {
    void upd(int n) { // provided method
        counter = counter + 1;
    }
    int counter;   // required field
}
```

Flattening semantics

```
trait T1 {
    void upd(int n);   // required method
    void inc(int n) {  // provided method
        if (n > 0) this.upd(n) else errs=errs+1
    }
    int errs;   // required field
}
class Counter implements ICounter uses T1, T2 {
    int counter = 0;
    int errs = 0;

    Counter(int counter) {
        this.counter = counter;
    }
}
```

```
trait T2 {
    void upd(int n) { // provided method
        counter = counter + 1;
    }
    int counter;   // required field
}
```

```
// "flattened" version
class Counter implements ICounter {
    int counter = 0;
    int errs = 0;

    Counter(int counter) { ... }

    void inc(int n) {
        if (n > 0) this.upd(n)...
    } // from T1

    void upd(int n) {
        counter = counter + 1;
    } // from T2
}
```

The symmetric Sum operation

- Merges two traits to form a new trait
- Requires the summed traits to be
  - disjoint (no identically named provided methods)
  - consistent (identically required fields and methods must have the same type)

Example of sum

```
trait T1 {
    void upd(int n);   // required method
    void inc(int n) {  // provided method
        if (n > 0) this.upd(n) else errs=errs+1
    }
    int errs;   // required field
}
```

```
trait T3 {
    void upd(int n);   // required method
    void dec(int i) {
        if (i>0)
            upd(-i)
    }
}
```

```
trait TCounter uses T1, T3 {
    void reset(int i) {
        if (i>0)
            upd(-i) // required by T1, T3
        else
            errs=errs+1 // required by T1
    }
}
```

Full JAVA generics support

```
trait TStack<T> {
    List<T> collection; // required field

    boolean isEmpty() { return collection.size() == 0; }
    void push(T e) { collection.add(0, e); }
    T pop() {
        if (isEmpty())
            return null;
        return collection.remove(0);
    }
}
```

Full JAVA generics support

```
trait TStack<T> {
    List<T> collection; // required field

    boolean isEmpty() { return collection.size() == 0; }
    void push(T e) { collection.add(0, e); }
    T pop() {
        if (isEmpty())
            return null;
        return collection.remove(0);
    }
}

class CStack<T> implements IStack<T> uses TStack<T> {
    List<T> collection = new ArrayList();

    CStack() {}
    CStack(Collection<T> c) {
        collection.addAll(c);
    }
}
```

Including bounded and *F*-bounded quantifications

```
trait TStack<T> {
    List<T> collection; // required field

    boolean isEmpty() { return collection.size() == 0; }
    void push(T e) { collection.add(0, e); }
    T pop() {
        if (isEmpty())
            return null;
        return collection.remove(0);
    }
}

class CStackOfSetsOfComparable<T extends Comparable<T>>
            implements IStack<Set<T>> uses TStack<Set<T>> {
    List<Set<T>> collection = new ArrayList();
}
```

And generic methods

With the addition of functional types and lambdas (thanks to XBASE)

```
trait T1<T> {
    Iterable<T> iterable;

    // function type: (T) => R
    <R> List<R> mapToList((T) => R mapper) {
        val result = new ArrayList<R>();
        for (e : iterable)
            result += mapper.apply(e);
        return result;
    }
}

class C1<String> uses T1<String>{...}

val c = new C1()
// inferred as List<Integer>
val sizes = c.mapToList([ element | element.size ])
// inferred as List<String>
val upper = c.mapToList([ element | element.toUpper ])
```

Support for JAVA annotations

- Can annotate:
  - Traits provided methods
  - Class fields
  - It makes no sense to annotate requirements

Support for JAVA annotations

- Can annotate:
  - Traits provided methods
  - Class fields
  - It makes no sense to annotate requirements

Integration with JAVA frameworks

For example, JUnit and Dependency Injection frameworks that rely on annotations

Write JUnit tests in XTRAITJ

```
trait TStackTestCase {
    IStack<String> fixture;

    @Test void testNotEmpty() {
        assertFalse(fixture.isEmpty())
    }
    @Test void testContents() {
        assertEquals("foo", fixture.pop())
        assertEquals("bar", fixture.pop())
        assertNull(fixture.pop())
    }
}

trait TStackAsArrayListSetup {
    IStack<String> fixture;
    @Before void setup() {
        fixture = new CStack<String>(newArrayList("foo", "bar"))
    }
}

trait TStackAsLinkedListSetup {
    IStack<String> fixture;
    @Before void setup() {
        fixture = new CStack<String>(newLinkedList("foo", "bar"))
    }
}
```

Method Alias

The expression `T[`**alias** `m1` **as** `m2]`, where the method `m1` must be provided by `T` and the method `m2` must not be declared by `T`, denotes the trait obtained from `T` by adding a new provided method that is a copy of `m1` with name `m2`

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
```

Method Alias

The expression `T[`**`alias`** `m1` **`as`** `m2]`, where the method `m1` must be provided by `T` and the method `m2` must not be declared by `T`, denotes the trait obtained from `T` by adding a new provided method that is a copy of `m1` with name `m2`

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}

trait TAuxFactorial uses TFactorial[alias mRec as mAux] {}
```

Method Alias

The expression `T[alias m1 as m2]`, where the method `m1` must be provided by `T` and the method `m2` must not be declared by `T`, denotes the trait obtained from `T` by adding a new provided method that is a copy of `m1` with name `m2`

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}

trait TAuxFactorial uses TFactorial[alias mRec as mAux] {}
```

corresponds to the flattened version:

```
trait TAuxFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
    int mAux(int n) { if (n==0) 1 else n*mRec(n-1) }
    // alias does not rename methods in bodies
}
```

Renaming

The expression `T`[**rename** `n1` **to** `n2`], where the method (or field) `n1` must be declared by `T` and the method (resp. field) `n2` must not be declared by `T`, renames references and declaration.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
```

Renaming

The expression `T`[**rename** `n1` **to** `n2`], where the method (or field) `n1` must be declared by `T` and the method (resp. field) `n2` must not be declared by `T`, renames references and declaration.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}

trait TRenFactorial uses TFactorial[rename mRec to mNew] {}
```

Renaming

The expression `T`[**rename** `n1` **to** `n2`], where the method (or field) `n1` must be declared by `T` and the method (resp. field) `n2` must not be declared by `T`, renames references and declaration.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}

trait TRenFactorial uses TFactorial[rename mRec to mNew] {}
```

corresponds to:

```
trait TRenFactorial {
    int factorial(int n) { return mNew(n) }
    int mNew(int n) { if (n==0) 1 else n*mNew(n-1) }
    // references are renamed as well
}
```

Restrict

The expression `T`[**restrict** `m`], where the method `m` must be provided by `T`, denotes the trait obtained from `T` by making `m` a required method.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
```

Restrict

The expression `T`[**restrict** `m`], where the method `m` must be provided by `T`, denotes the trait obtained from `T` by making `m` a required method.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}

trait TRestrictFactorial uses TFactorial[restrict mRec] {}
```

Restrict

The expression `T[`**`restrict`** `m]`, where the method `m` must be provided by `T`, denotes the trait obtained from `T` by making `m` a required method.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}

trait TRestrictFactorial uses TFactorial[restrict mRec] {}
```

corresponds to:

```
trait TRestrictFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n); // made required
}
```

Combining operations

For example, to simulate method overriding

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
trait TNewFactorial
        uses TFactorial[alias mRec as mOldRec, restrict mRec] {
    // since TFactorial.mRec is restricted, this is not a conflict
    int mRec(int n) {
        println("factorial of " + n); return mOldRec(n)
    }
}
```

Combining operations

For example, to simulate method overriding

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
trait TNewFactorial
        uses TFactorial[alias mRec as mOldRec, restrict mRec] {
    // since TFactorial.mRec is restricted, this is not a conflict
    int mRec(int n) {
        println("factorial of " + n); return mOldRec(n)
    }
}
```

corresponds to (note the dynamic binding semantics):

```
trait TNewFactorial {
    int factorial(int n) { return mRec(n) } // invokes the new version
    int mRec(int n) {
        println("factorial of " + n); return mOldRec(n)
    }
    int mOldRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
```

Private methods

- Qualifying a method `m` as **private** in a trait `T`, hides the name `m` and permanently binds the method `m` to the trait.
- The actual name of a **private** method is immaterial.

The following two trait declarations are equivalent.

```
trait T1 {
    private void update() { ... }
    void m() { update(); }
}

trait T2 {
    private void foobar() { ... }
    void m() { foobar(); }
}
```

Method hiding

The expression `T`[**hide** `m`], where the method `m` must be provided by `T`, denotes the trait obtained from `T` by making the method `m` **private** to the trait.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
trait THideFactorial uses TFactorial[hide mRec] {
    // since TFactorial.mRec is hidden, this is not a conflict
    String mRec(boolean b) { return "something completely different"; }
}
```

Method hiding

The expression `T`[**hide** `m`], where the method `m` must be provided by `T`, denotes the trait obtained from `T` by making the method `m` **private** to the trait.

```
trait TFactorial {
    int factorial(int n) { return mRec(n) }
    int mRec(int n) { if (n==0) 1 else n*mRec(n-1) }
}
trait THideFactorial uses TFactorial[hide mRec] {
    // since TFactorial.mRec is hidden, this is not a conflict
    String mRec(boolean b) { return "something completely different"; }
}
```

corresponds to:

```
trait THideFactorial {
    int factorial(int n) { return <boundname>(n) }
    private int <boundname>(int n) { if (n==0) 1 else n*<boundname>(n-1) }
    String mRec(boolean b) { return "something completely different"; }
}
```

Translation into Java

- We do not implement "flattening" directly,
- we translate traits into JAVA with object composition and method delegation.
  - Each method body is translated into exactly one single JAVA method body.
  - Our JAVA code generation is compositional in the presence of alteration operations.
  - No code duplication.
- Similar strategy adopted by other implementation.
- Drawback: overhead due to method forwarding.

Generated JAVA code

- The generated JAVA code has no dependency on XTRAITJ,
- it only depends on XBASE library and Google Guava library.
- These two libraries are less than 2 MB.
- The generated JAVA code can run on any JAVA platform, and
- thanks to the reduced size of the required JAVA libraries, it can be installed on JAVA devices, such as, Android devices.

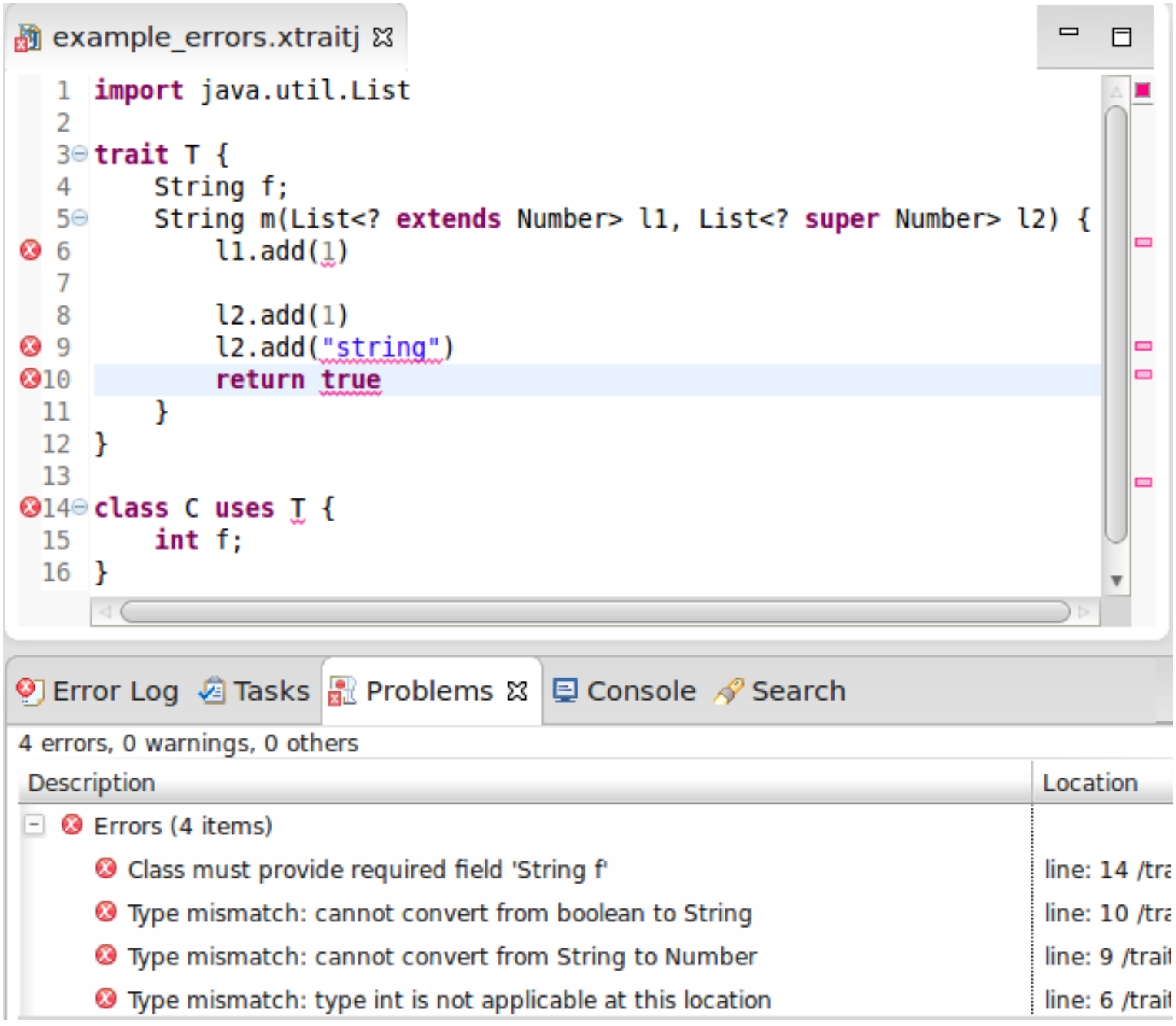
Generated JAVA code

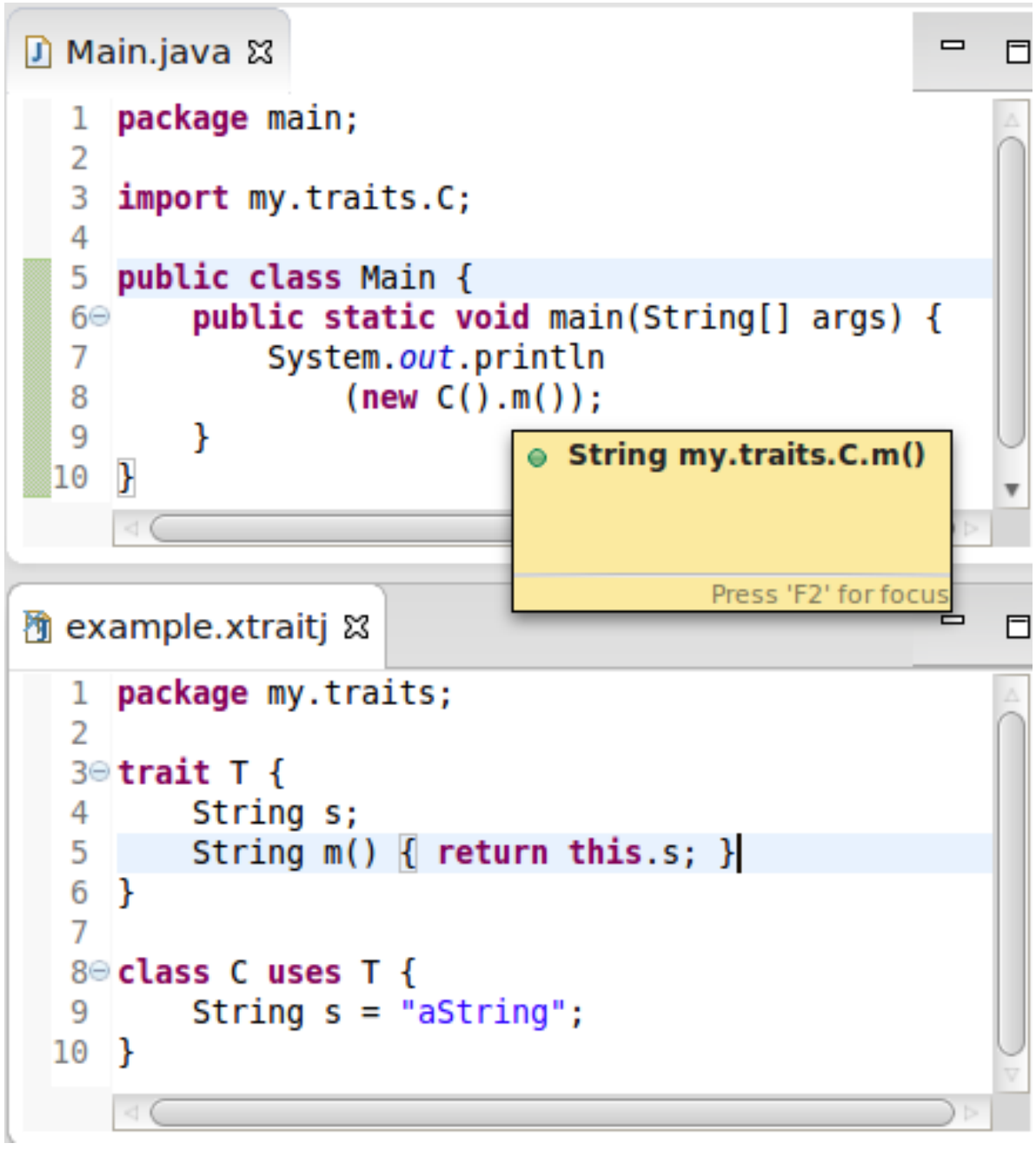
- The generated JAVA code has no dependency on XTRAITJ,
- it only depends on XBASE library and Google Guava library.
- These two libraries are less than 2 MB.
- The generated JAVA code can run on any JAVA platform, and
- thanks to the reduced size of the required JAVA libraries, it can be installed on JAVA devices, such as, Android devices.

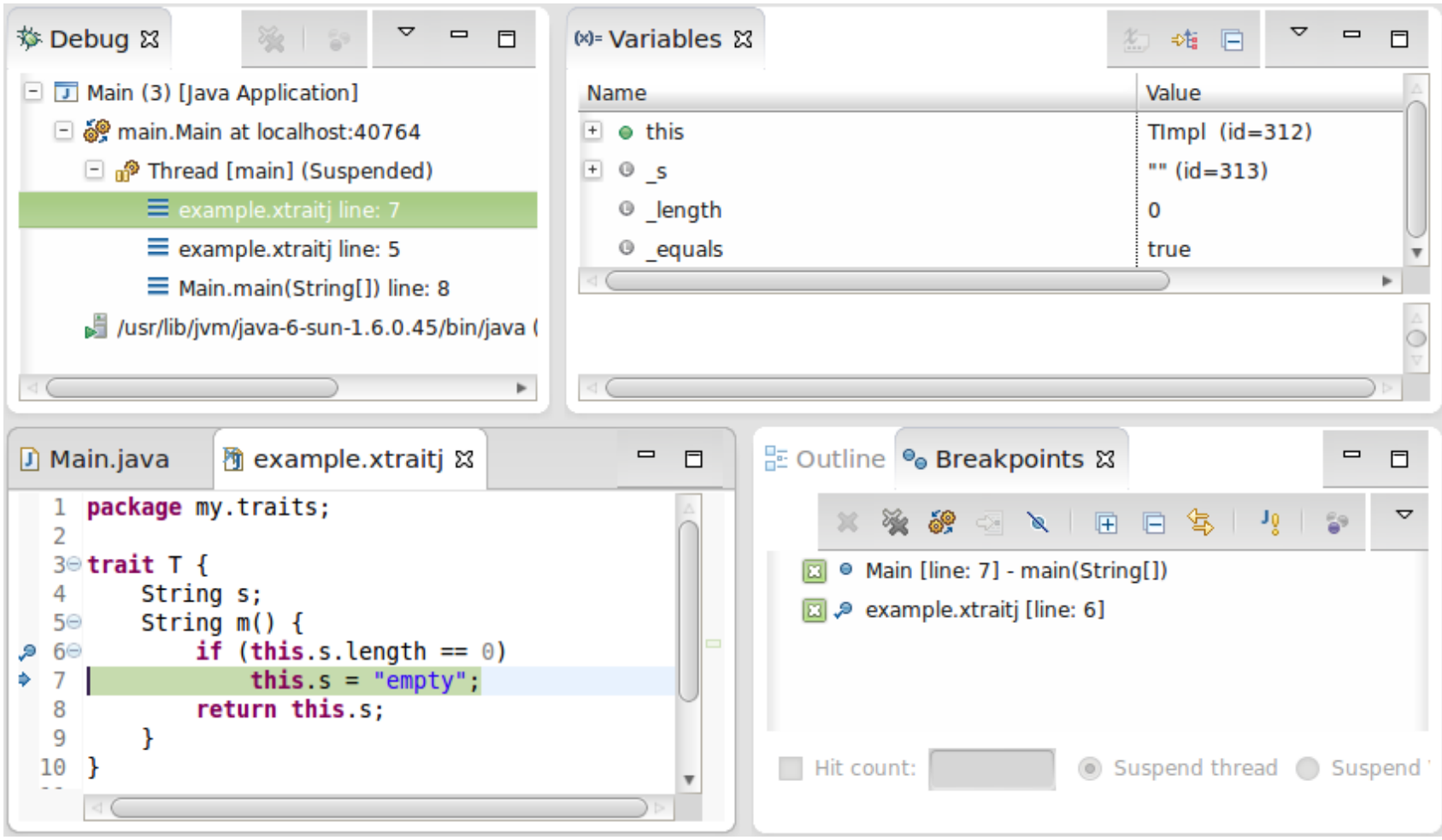
JAVA code and XTRAITJ code coexist and interoperate.

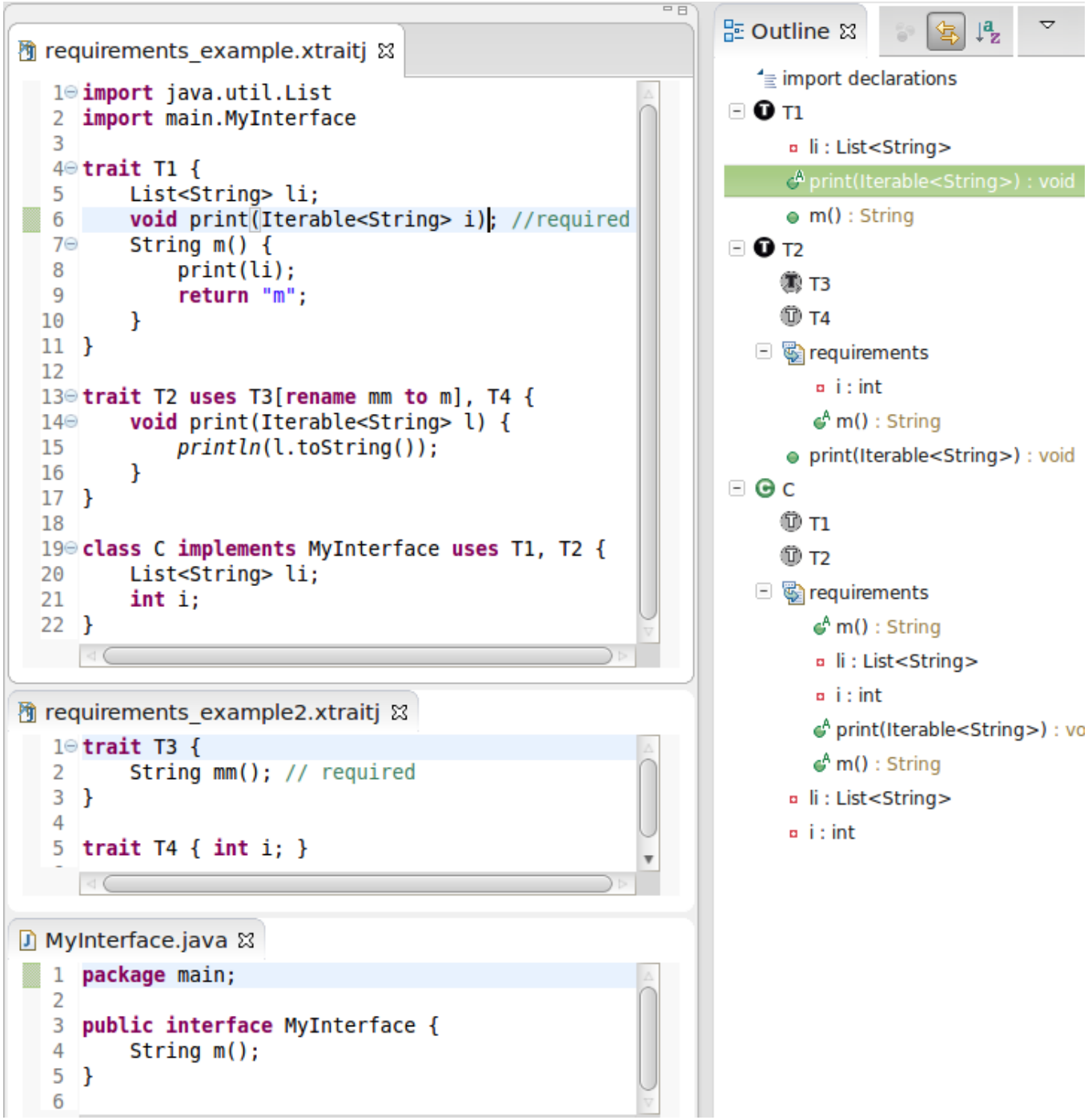
Benefits of XBASE

- Typing of method bodies is done by XBASE: no need to re-implement the JAVA type system;
- Complete integration with JAVA libraries;
- No need to define interfaces in XTRAITJ: just define a JAVA interface and use it in a XTRAITJ program;
- Eclipse IDE tooling integrated with JDT
- Debugging: debug XTRAITJ programs directly!


example_errors.xtraitj
1 import java.util.List
2
3 trait T {
4 String f;
5 String m(List<? extends Number> l1, List<? super Number> l2) {
6 l1.add(1)
7
8 l2.add(1)
9 l2.add("string")
10 return true
11 }
12 }
13
14 class C uses T {
15 int f;
16 }
Error Log
Tasks
Problems
Console
Search
4 errors, 0 warnings, 0 others
Description
Location
Errors (4 items)
Class must provide required field 'String f'
line: 14 /tra
Type mismatch: cannot convert from boolean to String
line: 10 /tra
Type mismatch: cannot convert from String to Number
line: 9 /trait
Type mismatch: type int is not applicable at this location
line: 6 /trait


Main.java
1 package main;
2
3 import my.traits.C;
4
5 public class Main {
6 public static void main(String[] args) {
7 System.out.println
8 (new C().m());
9 }
10 }
String my.traits.C.m()
Press 'F2' for focus
example.xtraitj
1 package my.traits;
2
3 trait T {
4 String s;
5 String m() { return this.s; }
6 }
7
8 class C uses T {
9 String s = "aString";
10 }


Debug
Main (3) [Java Application]
main.Main at localhost:40764
Thread [main] (Suspended)
example.xtraitj line: 7
example.xtraitj line: 5
Main.main(String[]) line: 8
/usr/lib/jvm/java-6-sun-1.6.0.45/bin/java (
Variables
Name
Value
this
TImpl (id=312)
_s
"" (id=313)
_length
0
_equals
true
Main.java
example.xtraitj
1 package my.traits;
2
3 trait T {
4 String s;
5 String m() {
6 if (this.s.length == 0)
7 this.s = "empty";
8 return this.s;
9 }
10 }
Outline
Breakpoints
Main [line: 7] - main(String[])
example.xtraitj [line: 6]
Hit count:
Suspend thread
Suspend


requirements_example.xtraitj
1 import java.util.List
2 import main.MyInterface
3
4 trait T1 {
5 List<String> li;
6 void print(Iterable<String> i); //required
7 String m() {
8 print(li);
9 return "m";
10 }
11 }
12
13 trait T2 uses T3[rename mm to m], T4 {
14 void print(Iterable<String> l) {
15 println(l.toString());
16 }
17 }
18
19 class C implements MyInterface uses T1, T2 {
20 List<String> li;
21 int i;
22 }
requirements_example2.xtraitj
1 trait T3 {
2 String mm(); // required
3 }
4
5 trait T4 { int i; }
MyInterface.java
1 package main;
2
3 public interface MyInterface {
4 String m();
5 }
6
Outline
import declarations
T1
li : List<String>
print(Iterable<String>) : void
m() : String
T2
T3
T4
requirements
i : int
m() : String
print(Iterable<String>) : void
C
T1
T2
requirements
m() : String
li : List<String>
i : int
print(Iterable<String>) : vo
m() : String
li : List<String>
i : int

Future work

- [1], refactoring JAVA class hierarchies into traits.
- Dynamic trait replacement [2].
- [3], a compositional proof systems for the verification of pure traits.
- Parametric Traits [4]: (not generic: more similar to C++ templates).


L. Bettini, V. Bono, and M. Naddeo.
A trait based re-engineering technique for Java hierarchies.
In *PPPJ*, pages 149–158. ACM, 2008.


L. Bettini, S. Capecchi, and F. Damiani.
On flexible dynamic trait replacement for java-like languages.
Science of Computer Programming, 78(7):907–932, 2013.


F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer.
Verifying traits: an incremental proof system for fine-grained reuse.
Formal Aspects of Computing, 2013. In press.


L. Bettini, F. Damiani, and I. Schaefer.
Implementing type-safe software product lines using parametric traits.
Science of Computer Programming, 2013. In press.

Software

http://xtraitj.sourceforge.net

Git repository:
https://github.com/LorenzoBettini/xtraitj

Eclipse update site:
http://sourceforge.net/projects/xtraitj/files/updates

Software

http://xtraitj.sourceforge.net

Git repository:
https://github.com/LorenzoBettini/xtraitj

Eclipse update site:
http://sourceforge.net/projects/xtraitj/files/updates

Thanks!