A Small Extension to Java for Class Refinement

Muga Nishizawa and Shigeru Chiba
(Tokyo Institute of Technology, Japan)
Class Refinement for Software Evolution

- Extend an existing class without modifying its source code

```java
class Point {
    int x, y;
    void setX(int nx) {
        x = nx;
    }
    void setY(int ny) {
        y = ny;
    }
}

refine Point {
    void setX(int x) {
        System.out.println("x :" + x);
        super.setX(x);
    }
    void print() {
        System.out.println(x +", " + y);
    }
}
```
Dynamic Class Refinement

- Turn refinement on and off during runtime

```java
class WebApp {
    static Session context;
    String price(int itemNo) {
        return toYen(super.price(itemNo)) + " yen";
    }
}

refine WebApp when context.lang == "jp" {
    String price(int itemNo) {
        return toYen(super.price(itemNo)) + " yen";
    }
}
```
The GluonJ Language

- Static/dynamic class refinement for Java
- Design goals
  - A normal Java syntax + Java annotations
    - No new language syntax
    - Programming with a normal Java IDE + compiler
  - Manageable runtime type errors
    - Runtime type errors should occur at only limited places
    - Compile-time type checking to a certain degree
Manageable Runtime Type Errors

- Without this, dynamic-refinement causes runtime type errors at too many places

```java
class Point {
    int x, y;
    int setX(int nx) { x = nx; }
}
Point p = ... ;
p.add(new Observer());
p.setX(0);
p.print();
```

This causes a type error if Main.debug is false.

```java
refine Point when Main.debug {
    void print() { ... }
    List observer;
    void add(Observer o) { .. }
    void notify() { .. }
    void setX(int ny) { notify(); ... }
}
```
Syntax of GluonJ

- It is described as a subclass of the original class.

```java
@Glue class Logging {
    @Refine static class BookLogger extends Book {
        @Override int getPrice() {
            System.out.println(price);
            return super.getPrice();
        }
    }
}
```

An @Glue class groups @Refine classes related to each other

The @Refine class refines a Book class.

Indicates the Book class
Calling an Appended Method

- To call a newly appended method, an instance of its target class must be cast:

```java
Book b = ...;
((BookPrinter)b).print();
```

From the type of an original class to the type of an @Refine class

```java
@Glue class Printing {
    @Refine static class BookPrinter extends Book {
        void print() {
            System.out.println("book[" + title + "," + price + "]");
        }
    }
}
```
A class refinement is directly applied to the bytecode of the original class definition.

```java
class Book {
    int getPrice() {
        System.out.println(price);
        return orig_getPrice();
    }
    int orig_getPrice() {
        return price;
    }
}
```

Overridden by @Refine

super.getPrice() is replaced

The original getPrice() is renamed.
An @Refine class is a standard subclass
Enjoy a code assist by a Java IDE
- A Java IDE recognizes methods/fields appended by @Refine classes
Dynamic-Refinement

- Effective while the specified method is running.
- Methods/fields are appended to all instances of the target class during that time.

```java
@Cflow("void BookStore.getBookPrice(Book)")
@Glue class CflowPrinting {
    @Refine static class BookPrinter extends Book {
        void print() {
            System.out.println("book[" + title + "," + price + "]");
        }
    }
}
```

Print() is appended only while getBookPrice() is running.
• For manageable type errors
  • Only explicit type casts may throw a type error.

```java
class Point {
    int x, y;
    int setX(int nx) { x = nx; }
}

Point p = … ;
PointEx pe = (PointEx)p;
pe.add(new Observer());
pe.setX(0);
pe.print();
```

```java
refine PointEx when Main.debug {
    void print() { … }
    List observer;
    void add(Observer o) { .. }
    void notify() { .. }
    void setX(int ny) { notify(); … }
}
```
Bytecode instrumentation (again)

- All methods/fields in an @Refine class are appended.

- Cast opeartor
  - Before type cast, insert a code for checking whether or not the @Refine class is effective.
  - replace all occurrences of the type name of refinement classes with their target class names.

- Methods
  - Throw NoSuchMethodError if the @Cflow condition is not satisfied.
Coding constraints

• For manageable type errors
  • Once an explicit type cast succeeds, a type error never happens
    • because the @Refine class is effective during the lifetime of the value.

• The value never escapes from the dynamic scope where the @Refine class is effective.

• Constraints
  • The value after the type cast cannot be saved in a non-local storage such as a field.
  • This is statically checked.
Coding Constraints (cont.)

Let $G$ is an @Glue class associated with @Cflow and let $G$ include a @Refine class $R$

- 1. A field of the type $R$ appears only within $G$.
- 2. The type $R$ is not the return type or one of the parameter types of the method specified as the argument to @Cflow.
- 3. $R$ is not an exception type (i.e. a subclass of Throwable). It is not the parameter to a catch clause.
- 4. The refinement class $R$ does not override a static method in its original class.
Related Work

- Class Refinement
  - E.g. eJava [Warth’06], AspectJ [Kiczales’01]
  - ContextJ [Costanza’06], CaesarJ [Aracic’06]

- Extended Java languages
  - AspectJ5, JBoss AOP, AspectWerkz
  - Ad-hoc design with Java annotations
    - Thus such extensions are not understood by a normal Java IDE
Conclusion

- **GluonJ**
  - Static/dynamic class refinement for Java
    - Class refinement is useful for software evolution

- **The Design goals of GluonJ**
  - A normal Java syntax + Java annotations
    - No new language syntax
    - Programming with a normal Java IDE + compiler

- **Manageable runtime type errors**
  - Runtime type errors should occur at only limited places
  - Compile-time type checking to a certain degree
Thank you
Class Refinement

- An effective technology for software evolution
- Its concept is similar to subclassing, mixins
  - It allows separating an extension to a class as a module
- E.g.
  - Changing an existing method in its class
  - Adding new methods, fields, interfaces to an existing class
The Benefit of Refinement

• Unlike subclassing and mixins, it **directly modifies** the definition of a class

Subclassing

```
class A

class SubA
```

```
A a = new SubA();
```

Refinement

```
class A

A refinement for A
```

```
A a = new A();
```

• Editing an original source code is **NOT** necessary
  • A client does not need to explicitly create an instance of the extended version of that class
Various Refinements

- **Static-refinement**
  - It allows statically changing the structure of a class
  - E.g. eJava [Warth’06], AspectJ [Kiczales’01]
  - E.g. ContextJ [Costanza’06], CaesarJ [Aracic’06]

- **Dynamic-refinement**
  - It allows switching refinements to a class according to dynamic contexts

- These features are useful for modularly extending and customizing a program