Hygienic methods - introducing HygJava

Jarosław Kuśmierek¹
Viviana Bono²

¹Warsaw University ²Torino University

TOOLS 2007
Zurich - 25 June 2007
1 **Introduction**
- Background - obvious facts
- Expected properties of programming languages

2 **Problems**
- Nature of the problems
- Examples of problems
- Difficulties in specifying properties of programs

3 **Our solution**
- General idea
- Example
- IDE Support
- Evaluation
Systems grow larger and larger:

- Almost no system is build from scratch. Every system is build on top of libraries.
- Almost no system is written, deployed and left as it is. Every system evolves and it is modified.
- Many systems are deployed to different sites, with different versions of libraries they depend on.
Systems grow larger and larger:

- Almost no system is build from scratch. Every system is build on top of libraries.
- Almost no system is written, deployed and left as it is. Every system evolves and it is modified.
- Many systems are deployed to different sites, with different versions of libraries they depend on.
Systems grow larger and larger:

- Almost no system is build from scratch. Every system is build on top of libraries.
- Almost no system is written, deployed and left as it is. Every system evolves and it is modified.
- Many systems are deployed to different sites, with different versions of libraries they depend on.
Systems grow larger and larger:

- Almost no system is build from scratch. Every system is build on top of libraries.
- Almost no system is written, deployed and left as it is. Every system evolves and it is modified.
- Many systems are deployed to different sites, with different versions of libraries they depend on.

These facts are obvious, however...
We need to be sure that:

- the code will also work with newer versions of libraries the code depends upon;
- the code we write will always mean what it meant to us (i.e., no accidental actions will occur).
We need to be sure that:

- the code will also work with newer versions of libraries the code depends upon;
- the code we write will always mean what it meant to us (i.e., no accidental actions will occur).
Nature of the problems

Lack of flexibility in the syntax

A new method introduction and a method override are sometimes not distinguishable.

The references are ambiguous

References from method overrides and method calls to method definitions are not precise.
Nature of the problems

Lack of flexibility in the syntax
A new method introduction and a method override are sometimes not distinguishable.

The references are ambiguous
References from method overrides and method calls to method definitions are not precise.
Introduction

Problems

Our solution

Nature of the problems
Examples of problems
Difficulties in specifying properties of programs

Variants of problems in different languages

Java

Code can stop working or start working differently than expected.

C#

It is significantly better with regard to versioning. Problems are less likely to occur, however they still can.

Eiffel

Presents an alternative approach. There are no situations with ambiguous overrides, only problems at compile time.
Introduction

Problems

Our solution

Nature of the problems

Examples of problems

Difficulties in specifying properties of programs

Variants of problems in different languages

Java

Code can stop working or start working differently than expected.

C#

It is significantly better with regard to versioning. Problems are less likely to occur, however they still can.

Eiffel

Presents an alternative approach. There are no situations with ambiguous overrides, only problems at compile time.
### Variants of problems in different languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Code can stop working or start working differently than expected.</td>
</tr>
<tr>
<td>C#</td>
<td>It is significantly better with regard to versioning. Problems are less likely to occur, however they still can.</td>
</tr>
<tr>
<td>Eiffel</td>
<td>Presents an alternative approach. There are no situations with ambiguous overrides, only problems at compile time.</td>
</tr>
</tbody>
</table>
A possible problem for the implementation of an interface

A class can accidentally contain a method with the same name and signature as one belonging to the interface to be implemented. However those two methods might have a different meaning (specification).
A revised version of a base class can contain a new method which will accidentally conflict with an existing method in subclasses.
Subtyping is dangerous

How do we use subtypes

If we change the type of a variable to a subtype, then the existing code in which it may occur in a covariant (read-only) position will still work. Additionally, it can be expanded by referencing new features accessible in the subtype. But this is not always true, e.g. sometimes in C# such approach fails.

```java
{ Class1 x;
  x = someExpression;
  ...
  x.met1();
  x.met2();
}
```
Subtyping is dangerous

How do we use subtypes

If we change the type of a variable to a subtype, then the existing code in which it may occur in a covariant (read-only) position will still work. Additionally, it can be expanded by referencing new features accessible in the subtype. But this is not always true, e.g. sometimes in C# such approach fails.

```java
{   Class2 x;
    x= someExpression;
    ....
    x.met1();
    x.met2();
}
```
Subtyping is dangerous

How do we use subtypes

If we change the type of a variable to a subtype, then the existing code in which it may occur in a covariant (read-only) position will still work. Additionally, it can be expanded by referencing new features accessible in the subtype. But this is not always true, e.g. sometimes in C# such approach fails.

```java
{ Class2 x;
  x = someExpression;
  ....
  x.met1(); // This call can unexpectedly be bound to another method
  x.met2();
}
```
With mixins problems are more likely to occur

When it comes to a languages with a mixin construct, the probability of conflict raises, therefore different mechanisms were developed in order to overcome these difficulties.
Name clashes during the composition of mixins

With mixins problems are more likely to occur

When it comes to languages with a mixin construct, the probability of conflict raises, therefore different mechanisms were developed in order to overcome these difficulties.

MixedJava & MixGen

Contain sophisticated rules which define visibility of the identifiers, and define which mixins override methods from which mixins. Allow many different introductions interleaved with their implementations. MixGen extends the MixedJava rules to arbitrary first class generics. But the subtyping relation is not transitive!
Introduction

Problems

Our solution

Nature of the problems

Examples of problems

Difficulties in specifying properties of programs

Name clashes during the composition of mixins

With mixins problems are more likely to occur

When it comes to languages with a mixin construct, the probability of conflict raises, therefore different mechanisms were developed in order to overcome these difficulties.

MixedJava & MixGen

Contain sophisticated rules which define visibility of the identifiers, and define which mixins override methods from which mixins. Allow many different introductions interleaved with their implementations. MixGen extends the MixedJava rules to arbitrary first class generics. But the subtyping relation is not transitive!

Traits

Disallow the composed traits to contain the same method, making the code not compile with newer version of libraries.
**Question:**

How can we state properties about a method $m_1$ declared in a class $C$ referencing some foreign code? We need to state some assumptions about the referenced methods belonging to other classes. But which assumptions?
Question:
How can we state properties about a method $m_1$ declared in a class C referencing some foreign code? We need to state some assumptions about the referenced methods belonging to other classes. But which assumptions?

In existing languages -
A referenced class Y contains a method $m_2$, which has the property $P_1$...
Question:

How can we state properties about a method $m_1$ declared in a class $C$ referencing some foreign code? We need to state some assumptions about the referenced methods belonging to other classes. But which assumptions?

In existing languages - like Java

A referenced class $Y$ contains a method $m_2$, which has the property $P_1$...

... and the ancestors of class $C$ do not introduce methods named $m_3$, $m_4$, which are present in class $C$ and could have been overridden.
Question:

How can we state properties about a method $m_1$ declared in a class $C$ referencing some foreign code? We need to state some assumptions about the referenced methods belonging to other classes. But which assumptions?

In existing languages - like C#:

A referenced class $Y$ contains a method $m_2$ inherited from ancestor $Z$, which has the property $P_1$...

... and class $Y$ will not reintroduce method $m_2$, inherited from its ancestor $Z$. 
Question:

How can we state properties about a method $m_1$ declared in a class $C$ referencing some foreign code? We need to state some assumptions about the referenced methods belonging to other classes. But which assumptions?

In existing languages - in general

A referenced class $Y$ contains a method $m_2$, which has the property $P_1$...

... and none of the above mentioned problems will accidentally change the way the method $m_1$ works.
Hygienic identifiers

Distinct Notions

1. method introduction
2. method implementation
3. method call
Hygienic identifiers

**Distinct Notions**
1. method introduction
2. method implementation
3. method call

**Precise Bindings**
- binding from 2 to 1 non ambiguous - by explicit paths
Hygienic identifiers

Distinct Notions

1. method introduction
2. method implementation
3. method call

Precise Bindings

- binding from 2 to 1 non ambiguous - by explicit paths
- binding from 3 to 1 non ambiguous - by explicit paths
package p;

class A
{
    String getName();
    implement String p.A.getName() {...}
}

class B extends A
{
    implement String p.A.getName() {...; super(); ...}
}

class C extends A
{
    String getName();
    implement String p.C.getName() {...}
    implement String p.A.getName() {...}
}

c C obj = new C();
obj.p.A.getName();
package p;

class A
{
    String getName();
    implement String p.A.getName() {...}
}

class B extends A
{
    implement String p.A.getName()
    {
        ...; super (); ... }
}

class C extends A
{
    String getName();
    implement String p.C.getName() {...}
    implement String p.A.getName() {...}
}

class C obj = new C();
obj.p.A.getName();
package p;

class A
{
    String getName();
    
    implement String p.A.getName() {...}
}

class B extends A
{
    implements String p.A.getName()
    {
        ...; super (); ... }
}
class C extends A
{
    String getName();
    
    implement String p.C.getName() {...}
    implement String p.A.getName() {...}
}
package p;

class A
{
   String getName();
   implement String p.A.getName() {...}
}

class B extends A
{
   implement String p.A.getName()
   {
       ...; super (); ... 
   }
}
class C extends A
{
   String getName();
   implement String p.C.getName() {...}
   implement String p.A.getName() {...}
}
package p;

class A
{
    String getName();
    implement String p.A.getName() { ... }
}

class B extends A
{
    implement String p.A.getName() 
    {
        ...; super (); ... 
    }
}
class C extends A
{
    String getName();
    implement String p.C.getName() { ... }
    implement String p.A.getName() { ... }
}
C obj = new C();
obj.p.A.getName();
How to make hygienic life easier with the support of some IDE

Automated prefixing

A tool can automatically expand non-hygienic code to a hygienic version, using Java-like expansion rules.
How to make hygienic life easier with the support of some IDE

**Automated prefixing**
A tool can automatically expand non-hygienic code to a hygienic version, using Java-like expansion rules.

**Prefix hiding**
A tool can display the code in an abbreviated form, showing the “full path” on demand.
Advantages & Disadvantages

What we lose

- Without the IDE support it is hard to write code
- The code is more coupled: it is not possible to move the method introductions. It might look like breaking the “encapsulation principle”. However, we think that moving method introductions correspond to changing the interface of the superclass from the point of view of the subclasses.
Advantages & Disadvantages

What we loose

- Without the IDE support it is hard to write code
- The code is more coupled: it is not possible to move the method introductions. It might look like breaking the “encapsulation principle”. However, we think that moving method introductions correspond to changing the interface of the superclass from the point of view of the subclasses.

Jarosław Kuśmieren Viviana Bono

Hygienic methods - introducing HygJava
Advantages & Disadvantages

What we lose

- Without the IDE support it is hard to write code
- The code is more coupled: it is not possible to move the method introductions. It might look like breaking the “encapsulation principle”. However, we think that moving method introductions correspond to changing the interface of the superclass from the point of view of the subclasses.

What we gain

- Perfect resilience to versioning.
- Much simpler type checking and compiler. Especially with mixins. Easier to understand.
- Theoretical properties can be stated and proven without negative assumptions.
Advantages & Disadvantages

What we loose

- Without the IDE support it is hard to write code
- The code is more coupled: it is not possible to move the method introductions. It might look like breaking the “encapsulation principle”. However, we think that moving method introductions correspond to changing the interface of the superclass from the point of view of the subclasses.

What we gain

- Perfect resilience to versioning.
- Much simpler type checking and compiler. Especially with mixins. Easier to understand.
- Theoretical properties can be stated and proven without negative assumptions.
Advantages & Disadvantages

What we lose

- Without the IDE support it is hard to write code
- The code is more coupled: it is not possible to move the method introductions. It might look like breaking the “encapsulation principle”. However, we think that moving method introductions correspond to changing the interface of the superclass from the point of view of the subclasses.

What we gain

- Perfect resilience to versioning.
- Much simpler type checking and compiler. Especially with mixins. Easier to understand.
- Theoretical properties can be stated and proven without negative assumptions.