Reverse Inheritance: Improving Class Library Reuse in Eiffel

Ciprian-Bogdan Chirila chirila@cs.upt.ro
Pierre Crescenzo Pierre.Crescenzo@unice.fr
Philippe Lahire Philippe.Lahire@unice.fr

Goals

- To reuse Eiffel class libraries when their source code is not available or is copyrighted;
- To avoid maintaining entire class libraries by modifying their existing source code.

Solution

Reverse inheritance (RI) class relationship offers several facilities for **reorganizing class hierarchies** in the context of Eiffel language: creating a common superclass, factoring common features, inserting a class into an existing hierarchy.

Eiffel programming language was chosen for implementing an expressive and orthogonal RI because of reasons dealing with language symmetry and consistency:
- The presence of adaptation mechanisms;
- The existence of multiple inheritance class relationship;
- Covariant feature redefinition;
- The lack of overloading enables feature identification by unique names;
- The uniform way of using features implemented by memory (attributes) or by computation (methods).

In order to respect the duality of ordinary inheritance of Eiffel, reverse inheritance can be **conforming** or **non-conforming**. Conforming inheritance/reverse inheritance keeps the type conformance relationship between subclass and superclass while non-conforming does not.

Reverse Inheritance Based Solutions

Capturing Common Functionalities

By creating a superclass using reverse inheritance there is no need to modify the source of subclasses, because in the new superclass must be specified the list of all its subclasses and the list of common features. This capability of RI allows creating a **common interface** which helps **manipulating in an homogeneous way** classes from different hierarchies. Also new subclasses can be created by ordinary inheritance as descendants of the newly created superclass.

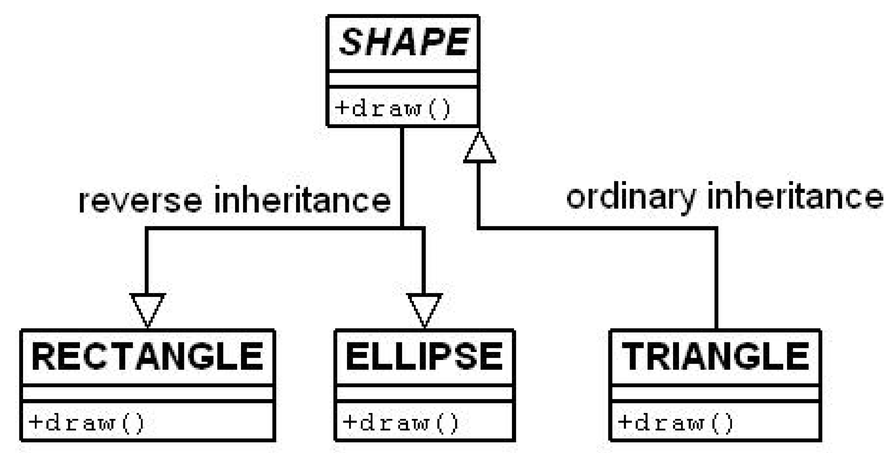

Figure 1: Capturing Common Functionalities

Problems:
- **Name conflicts** [Ped89,LHQ94] arise when two features having the same semantics have different names - called lost friends [Sak02] and when two features having the same name but different semantics - called false friends;
- **Signature incompatibilities** have to take into account incompatibilities related to parameter and return types, parameter number and order, assertions: preconditions, postconditions and invariants.

Benefits:
- Avoid modifying the original class hierarchy;
- Factoring the common features in one place on the hierarchy, in the superclass;
- Creating a common interface which helps manipulating the subclasses in an uniform manner;
- Extending the class hierarchy with a new subclass by ordinary inheritance.

Inserting a Class into an Existing Hierarchy

In figure 2 is presented a typical situation in which the design of an existing class hierarchy have to be changed and a new class have to be inserted between already existing two ones. **To add retroactively a new layer of abstraction** in a class hierarchy is a natural practice when the model of the application has to be adapted to new contexts or when the model evolves.

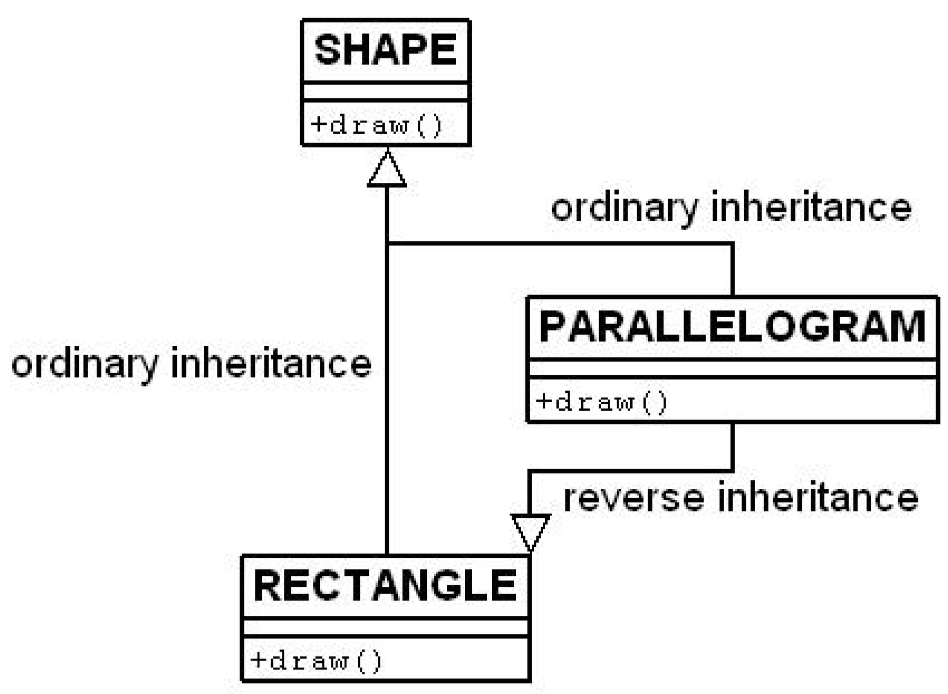

Figure 2: Inserting a Class Into an Existing Hierarchy

Benefits:
- Preserving the original classes untouched;
- Still refining the class hierarchy;
- Easily canceling the modifications.

Expressiveness of Exheritance

RI offers several mechanisms for reaching the goal of reusability:
- **Factoring Features** - selects common features to be factored in the superclass;
- **Exheriting Implementation** - selects one implementation from one subclass to be available in the superclass;
- **Renaming** - solves the name conflict problems or abstracts / generalizes the name of a feature;
- **Parameter and Assertion Adaptations** - solves the signature incompatibilities problems by giving more general assertions.

```
class RECTANGLE
 feature
  fgcolor, bgcolor: INTEGER;
  draw is do ... end
  floodfill is do ... end
  perimeter: INTEGER is do ... end
  print_figure(xcenter, ycenter: INTEGER) is
  do ... end
  display is
   require fgcolor > 64 and bgcolor < 192
  do ... end
 end -- class RECTANGLE
```

```
class ELLIPSE
 feature
  fgcolor, bgcolor: INTEGER;
  draw is do ... end
  floodfill is do ... end
  circumference: INTEGER is do ... end
  print_figure(x1, y1, x2, y2, color: INTEGER) is
  do ... end
  display is
   require fgcolor /= bgcolor
  do ... end
 end -- class ELLIPSE
```

```
deferred foster class SHAPE
 exherit
  RECTANGLE
   rename perimeter as boundary
   undefine draw, floodfill, boundary
   adapt print
  end
  ELLIPSE
   rename circumference as boundary
   undefine draw, boundary
   adapt print
   move floodfill
  end
  all -- all exheritable features
```

```
-- class SHAPE continued
feature
 print(x1, y1, x2, y2: INTEGER) is
  require stronger
 adapted
  {RECTANGLE}.print((x1+x2)/2,(y1+y2)/2)
  {ELLIPSE}.print(x1,y1,x2,y2,0)
 do
  -- possible implementation
 end
end -- class SHAPE
```

Example 1: Example of Figure 1 of the Extension of RI in Eiffel

```
class SHAPE
 feature
  draw is do ... end
 end -- class SHAPE

class RECTANGLE
 inherit
  SHAPE
   redefine draw
  end
 feature
  draw is do ... end
 end -- class RECTANGLE
```

```
class PARALLELOGRAM
 inherit
  SHAPE
   redefine draw
  end
 exherit
  RECTANGLE
   redefine draw
  end
  all -- all exheritable features
 feature
  draw is do ... end
 end -- class PARALLELOGRAM
```

Example 2: Example of Figure 2 of the Extension of RI in Eiffel

The **foster** keyword marks a class as being source in a RI class relationship.

From Inheritance/Exheritance Hierarchies to Ordinary Inheritance

In order to point that our approach is feasible we will show that each semantical construct discussed earlier can be expressed using a pure Eiffel language. The intermediate compilable code may contain a modified copy of the original source code. Modifications are mostly performed at syntactical level on a copy, leaving the behavior unchanged.

```
deferred class SHAPE
 feature
  fgcolor, bgcolor: INTEGER;
  draw is deferred end
  floodfill is
  do
   -- copy implementation of feature floodfill from class Ellipse
  end
  boundary: INTEGER is deferred end
  print(x1, y1, x2, y2, color: INTEGER) is do ... end
  display is
   require fgcolor > 64 and bgcolor < 192 and fgcolor /= bgcolor
  do ... end
 end -- class SHAPE
```

```
class RECTANGLE
 inherit
  SHAPE
   rename boundary as perimeter
   redefine floodfill, perimeter, print_figure,
   display
  end
 feature
  draw is do ... end
  floodfill is do ... end
  perimeter: INTEGER is do ... end
  print(x1, y1, x2, y2, color: INTEGER) is
  do
   old_print_figure((x1+x2)//2, (y1+y2)//2)
  end
  display is
   require else fgcolor > 64 and bgcolor < 192
  do ... end
  old_print_figure(xcenter, ycenter: INTEGER) is
   do ... end
 end -- class RECTANGLE
```

```
class ELLIPSE
 inherit
  SHAPE
   rename boundary as circumference
   redefine floodfill, circumference,print_figure
   display
  end
 feature
  draw is do ... end
  floodfill is do ... end
  circumference: INTEGER is do ... end
  print_figure(x1, y1, x2, y2, color: INTEGER) is
  do
   old_print_figure(x1, y1, x2, y2, 0)
  end
  display is
   require else fgcolor /= bgcolor
  do ... end
  old_print_figure(x1, y1, x2, y2: INTEGER) is
   do ... end
 end -- class ELLIPSE
```

Example 3: Example of Figure 1 Using Ordinary Inheritance Only

```
class SHAPE
 feature
  draw is do ... end
 end -- class SHAPE

class PARALLELOGRAM
 inherit
  SHAPE
   redefine draw
  end
 feature
  draw is do ... end
 end -- class PARALLELOGRAM
```

```
class RECTANGLE
 inherit
  PARALLELOGRAM
   redefine draw
  end
 feature
  draw is do ... end
 end -- class RECTANGLE
```

Example 4: Example of Figure 2 using Ordinary Inheritance Only

Features *old_print_figure* from classes *RECTANGLE* and *ELLIPSE* are the renamed versions of the original ones. This is necessary because in Eiffel there is no overloading. The new *print_figure* feature will have a common signature and in the implememntation each will call the original version using adaptations for parameters.

Perspectives

Reverse inheritance can help Eiffel class reusability by redesigning existing class hierarchies. The new class relationship is built symetrically from ordinary inheritance, so is not a concept hard to understand and to use, for designers. The special adaptation mechanisms included in RI semantics, do not represent a severe deviation from the philosophy of the language.

One of the perspectives regarding the semantics of reverse inheritance is to fully integrate it into Eiffel programming language. In order to fulfill this target, a formal model of the foster class must be proposed and a translation schema have to be designed which will be the core of a translator that generates compilable Eiffel code. Another perspective of the RI class relationship is to test it in practice like on the classes of Eiffel Kernel library.