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 redifinition;
- The lack of overlapping 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 all the underlying features from 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 [Pod99,LII99] arise when two features having the same semantics have different names - called friends [Sal92] 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.

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

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.

Example 3: Example of Figure 1 Using Ordinary Inheritance Only

Example 4: Example of Figure 2 Using Ordinary Inheritance Only

Example 5: Example of Figure 3 Using Ordinary Inheritance Only

References

Example 6: Example of Figure 4 Using Ordinary Inheritance Only

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.

Example 3: Example of Figure 1 Using Ordinary Inheritance Only

Example 4: Example of Figure 2 Using Ordinary Inheritance Only

Example 5: Example of Figure 3 Using Ordinary Inheritance Only

Example 6: Example of Figure 4 Using Ordinary Inheritance Only

References