

Towards Fully-fledged Reverse Inheritance in Eiffel

Markku Sakkinen

Department of Computer Science and Information Systems, University of Jyväskylä

Philippe Lahire

I3S Laboratory, University of Nice – Sophia Antipolis and CNRS

Ciprian-Bogdan Chirila

Politehnica University of Timisoara

Presentation at SPLST 2009, Tampere, 26. 8. 2009

Motivation

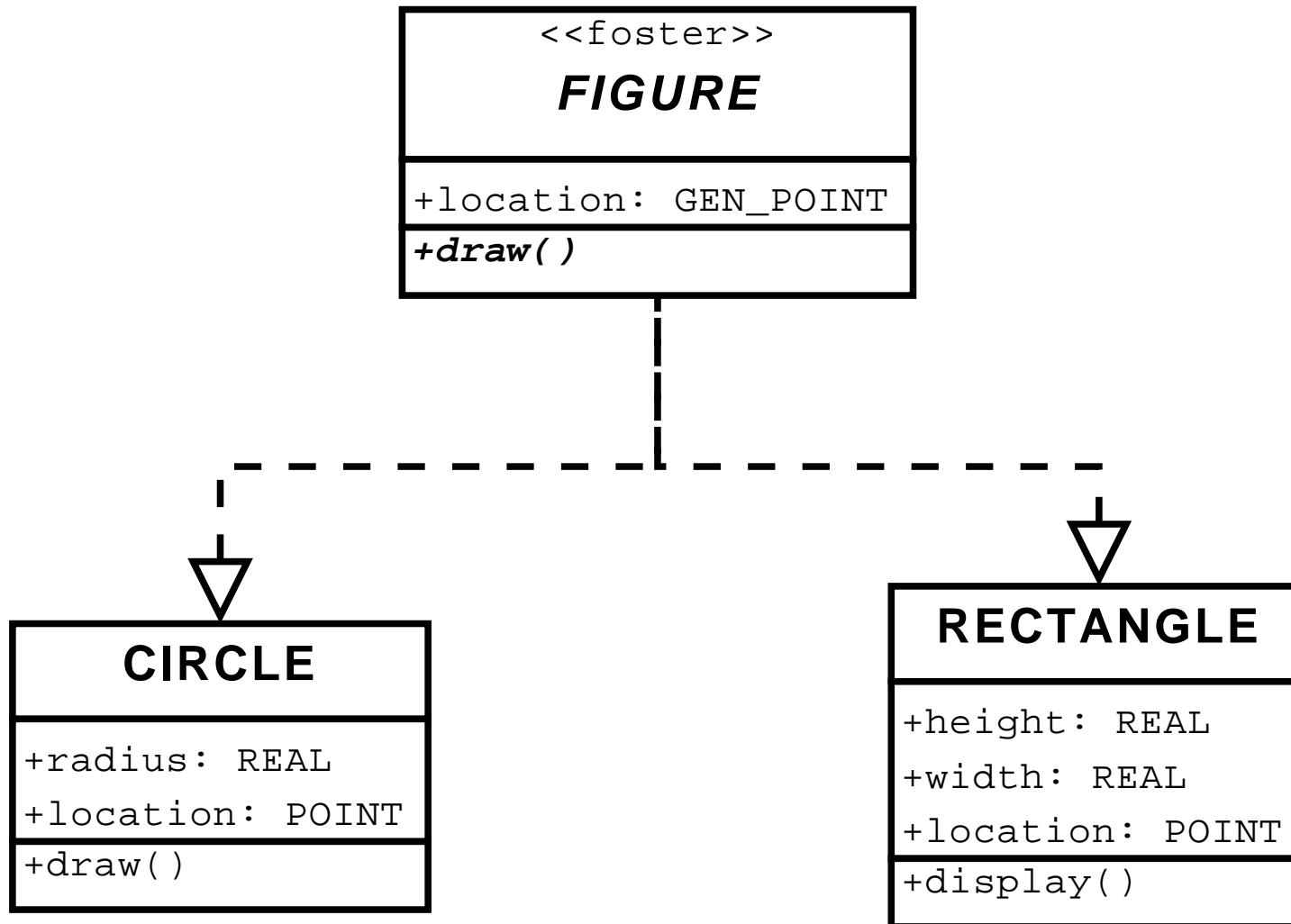
From object-oriented modelling and programming

- Generalisation (bottom-up) often more natural than specialisation (top-down).
- Existing languages support only the latter by ordinary inheritance (OI).
- Generalisation is possible by refactoring (modifying existing classes).
- Such refactoring is often undesirable or even impossible (e.g. standard class libraries).
- Why not offer also a generalisation mechanism, reverse inheritance (RI), in a language?

From the integration of heterogeneous databases (and other systems)

- The existing databases may represent partially the same things differently.
- A global system is desired that that gives a homogeneous view on the different subsystems.
- Homogenising the primary databases is out of the question! (A schema change of a database that cannot be stopped for maintenance is problematic and risky.)
- The same *objects* can appear in several subsystems.

A simple example



Previous research

Schrefl, M., Neuhold, E.J.: Object class definition by generalization using upward inheritance. *Proceedings of the Fourth International Conference on Data Engineering*, IEEE Computer Society (1988), 4–13.

- “upward inheritance”
- Generalisation classes on a different layer and slightly different from ordinary classes.
- Adaptation mechanisms.

Pedersen, C.H.: Extending ordinary inheritance schemes to include generalization. *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, ACM Press (1989), 407–417.

- A seminal paper, although with some errors.
- Illustrates the ideas with a simple hypothetical language.
- Also *implements* clause.

Lawson, T., Hollinshead, C., Qutaishat, M.: The potential for reverse type inheritance in Eiffel. *Technology of Object-Oriented Languages and Systems (TOOLS Europe'94)* (1994), 349–357.

- “reverse inheritance”, “foster class”
- A rather thorough proposal for an enhancement of Eiffel.
- Several non-trivial problems explained and at least partially solved.
- Foster classes are different from ordinary classes, mainly in the implementation.

Qutaishat, M., Fiddian, N., Gray, W.: Extending OMT to support bottom-up design modelling in a heterogeneous distributed database environment. *Data & Knowledge Engineering* **22** (1997), 191–205.

- An extension of OMT.
- Rather extensive bibliography (almost only DB literature).

Sakkinen, M.: Exheritance — class generalization revived. *Proceedings of the Inheritance Workshop at ECOOP 2002* (Black, A.P., Ernst, E., Grogono, P., Sakkinen, M., eds., Publications of Information Technology Research Institute **12**, University of Jyväskylä (June 2002), 76–81.

- “exheritance”
- Continues and corrects Pedersen’s work.
- Many new ideas briefly presented, quite optimistic.

Chirila, C.B., Crescenzo, P., Lahire, P.: A reverse inheritance relationship for improving reusability and evolution: The point of view of feature factorization. *Proceedings of The 3rd International Workshop on MechAnims for SPEcialization, Generalization and Inheritance – MASPEGHI’04* (Lahire, P., et al., eds.), Sophia-Antipolis, France, Laboratoire I3S (2004), 9–14.

- Inspired by the previous paper.
- RI could affect existing classes.
- Lead to cooperation.

Sakkinen, M., Chirila, C.B., Lahire, P.

- Continues mainly from Lawson et al. (1994) and Sakkinen (2002).
- Several submitted versions before this one.
- A much more complete suggestion for Eiffel.
- Also an almost complete implementation by automatic transformation to standard Eiffel (i.e., refactoring).

Basic ideas

Single exheritance

- New class N defined as a superclass (parent) of existing class E.
- Corresponds to OI from N to E, but *dependency* is the opposite.
- Either all features, none, or any subset are exherited to N.
- N can be later used for both OI and RI.
- If E has a superclass S, N *can* be defined also to be a subclass (heir) of S. In a single-inheritance language, it *must*.
- In other cases, exheriting *inherited* features of E may be questionable.

Multiple inheritance

- N is defined as a superclass of several existing classes E_1, \dots, E_m .
- Only those features common to all E_i s can be inherited.
- Some adaptation of features may be needed:
 - The “same” feature can have different names in different classes.
 - The same name can denote different features.
 - The types of the same attribute may be different.
 - The signatures of the same method may be different.
 - Some other adaptations may be needed.

Exheritance of methods

- Exheritance as abstract (deferred) is safe.
- Exheriting also the implementation (body) is often questionable or impossible. Impossible if not also all other features needed (recursively) by the method are exherited.
- Pre- and postconditions must be taken into account (at least in Eiffel).

RI induced by OI

- When a new class N is designed by multiple inheritanced (MI) from the existing classes E_1 and E_2 , a common feature F (or several) may be detected.
- In many languages, it is not allowed to unify features from E_1 and E_2 , unless they are inherited from a common superclass (ancestor).
- A new common superclass N can be created by RI.

Why Eiffel?

General reasons

- A language liked by many researchers, although not common in the industry.
- One of the oldest still living OOPLs of the Scandinavian school (e.g., with static typing).
- Thoroughly designed from scratch as a “homogeneous” language (no hybrid).
- Much attention paid to software engineering — in particular “Design by Contract”™.
- MI and genericity (semantic, not just syntactic as in C++) from the beginning.

Interesting inheritance mechanisms

- An abstract feature can be implemented (effected) in a subclass either as a method or an attribute (if it has a result type and no parameters).
- Features can be renamed in inheritance.
- In *repeated inheritance* (multiple inheritance paths from the same ancestor), renaming determines whether a feature is shared or replicated.
- Features can be unified in MI even when they are not inherited from a common ancestor.
- Even direct repeated inheritance (the same parent several times) is allowed.
- Covariant type/signature redefinition is allowed for both methods and attributes — unsafe, requires run-time type checking.
- Recently, *non-conforming* inheritance has been added (similar but not the same as protected or private inheritance in C++).

Evolution

- Eiffel 3 had many enhancements and changes from previous versions.
- The current version is significantly further developed (also incompatible changes).
- A rather large and complicated language — but depends on what it is compared with.
- Standardised by ECMA in 2005, now also an ISO standard.

Pragmatic

- Previous work and authors' previous knowledge.
- Availability of suitable tools.

Main principles

In approximate order of importance, some especially for Eiffel.

Rule 1: Genuine Extension

Eiffel classes and programs that do not exploit reverse inheritance must not need any modifications, and their semantics must not change.

Rule 2: Full Class Status

After a foster class has been defined, it must be usable in all respects as if it were an ordinary class.

Rule 3: Invariant Class Structure and Behaviour

Introducing a foster class as a parent of one or several classes using reverse inheritance must not modify the structure and behaviour of those classes.

Rule 4: Equivalence with Ordinary Inheritance

Declaring a reverse inheritance relationship from class A to class B should be equivalent to declaring an ordinary inheritance relationship from class B to class A.

Rule 5: Minimal Change of Inheritance Hierarchy

Introducing a foster class must neither delete direct inheritance relationships (parent-heir relationships) nor create *any* inheritance relationships (ancestor-descendant relationships) between previously existing classes.

Rule 6: Exheritable Features

The features f_1, \dots, f_n of the respective, different classes C_1, \dots, C_n are exheritable together to a feature in a common foster class if there exists a common signature to which the signatures of all of them conform, possibly after some adaptations. Each of the features f_1, \dots, f_n can be either immediate or inherited.

Rule 7: No Repeated Exheritance

Two different features of the same class must not be exherited to the same feature in a foster class.

Basics of our approach

‘RI-Eiffel’ and ‘RI-UML’

‘foster class’

In a *new* language with both OI and RI, the ‘foster’ keyword would be needed no more than a ‘heir’ or ‘subclass’ keyword.

Standard Eiffel keywords for inheritance: **inherit**, **rename**, **redefine**, **undefine**. If a deferred (abstract) feature is effected in the inheriting class, no keyword is used.

RI-Eiffel keywords for exheritance: **exherit**, **rename**, **redefine**, **moveup**, **adapt**.

An effective feature becomes deferred in the exheriting class by default; no keyword is used.

Adaptations would be possible, but probably not very useful in OI. They require special handling at run time. They must not cause side effects (by Rule 3).

Adding a root class as a parent

The universal root class **ANY** can be ignored in most cases.

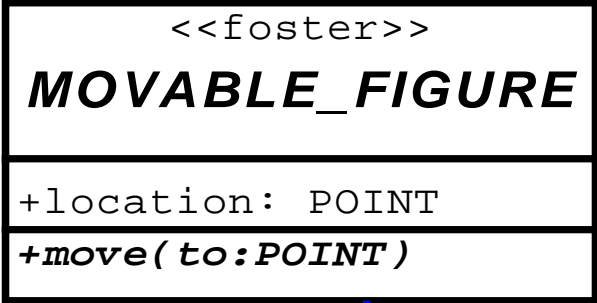
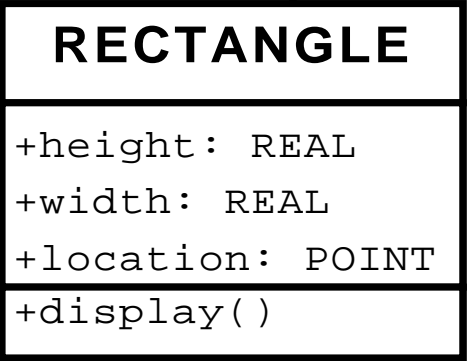
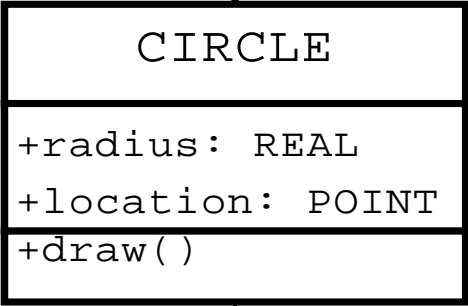
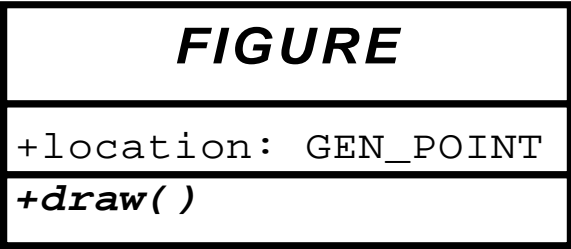
```
01 deferred foster class FIGURE
02   exherit
03   CIRCLE
04     redefine location
05     adapt location
06   end
07 RECTANGLE
08   redefine location
09   rename display as draw
10   end
11 all -- all exheritable features
12 feature
13   location: GEN POINT
14   adapted CIRCLE
15     to x := result.x/10, y := result.y/10
16     from x := result.x*10, y := result.y*10
17   end
18 end -- class FIGURE
```

Adding a class with both reverse and ordinary inheritance

An *amphibious* foster class; the difference to a root class concerns only its amphibious *features*.

It seems natural that the inherited version of an amphibious feature is the default in the foster class.

(Re)naming determines which inherited and exherited features are actually taken as versions of the same feature.



Conclusion and perspectives

We have succeeded to define reverse inheritance as an extension to Eiffel, covering practically the whole language.

- Very many mechanisms had to be taken into account, although not all were found to have an effect on RI.
- Genericity, pre- and postconditions and many other things could not be discussed in this paper.
- We tried to keep the spirit and style of standard Eiffel.
- The task would have been much easier if we had not maintained full compatibility with existing Eiffel.

Additional useful application possibilities for RI, e.g.:

- *Interface inheritance*, not offered by any well-known language — simply by inheriting all public features of a class as abstract (deferred).
- Bridging the gap between *subobject-oriented* (as in C++) and *attribute-oriented* (as in Eiffel) multiple inheritance: any set of attributes of a class can be made into a subobject by inheriting them into the same foster class.

Negative effects:

- “With OI you don’t know the descendants of a class, and with RI you don’t know even all its ancestors” (Peter Grogono).
- The set of features that a parent class inherits in RI is not as straightforward as the set of features that a child class inherits in OI.
- The language becomes larger and more complex.

Another negative effect claimed by some critics:

- Reverse inheritance makes separate compilation impossible.
- At least in Eiffel the separate compilation of classes is not generally possible anyway.

Ongoing and possible future work:

- Implementation of most of RI-Eiffel by transformation into standard Eiffel (adaptations cannot be done without modifying the Eiffel compiler).
- Applying RI-Eiffel in practice to get experience about its advantages and disadvantages.
- Defining RI extensions for other suitable languages.
- Designing a new language having ordinary and reverse inheritance as mechanisms of equal status.