Experiences from Adding Reverse Inheritance to Eiffel

Informal talk at the PLE workshop at ECOOP 2014, Uppsala, Sweden, 28. 7. 2014

Markku Sakkinen
Department of Computer Science and Information Systems, University of Jyväskylä

Based on research performed with:

Philippe Lahire
I3S Laboratory, University of Nice – Sophia Antipolis and CNRS (France)

Ciprian-Bogdan Chirila
Politehnica University of Timisoara (Romania)

Different kinds of evolution

Maybe most often, an evolution step is at least intended to be a pure enhancement, i.e., to preserve full backward compatibility.

However, this does not always succeed, but the new features necessitate some changes to old features.

Adding a new keyword may invalidate old programs which use that word as an identifier.

- In some languages, new standard classes may cause similar problems.
- This did not happen in Algol 60, because keywords were lexically distinguished from identifiers.
- In PL/1, a keyword and an identifier are never legal in the same syntactic position.

In other cases, evolution includes deliberate changes to some language features that are deemed to need improvement or even to be obsolete.

We wanted to make a pure enhancement to Eiffel.

- Even to be able to program exactly the same class hierarchies by ordinary or reverse inheritance, or a mixture of both.

Evolution of inheritance in C++ (for comparison)

The story of C++ began by adding classes and inheritance (‘derivation’) to C.

- No other changes to the language.
- Probably intended to have no harmful interactions with existing language mechanisms.

Pitfall: pointer arithmetic (and arrays).

- A pointer of type ***Myclass** can also point to objects of any subclass of **Myclass**.
- Because those objects are usually larger, pointer arithmetic (and array indexing) goes wrong.
- Even when I last looked at C++, this severe defect had not been corrected.

Access levels for superclasses ('base classes') are a useful feature, which is found in almost no other language, but…

- The default is **private**, while **public** could be preferable (not just implementation inheritance).

Multiple inheritance (MI) was added to C++ later.

- The critical case is *fork-join* or *diamond inheritance* (a superclass inherited over more than one path).
- The default was chosen to be replication of the common superclass part.
- That actually makes sense with private, but not with public inheritance.
- To cause sharing of the superclass part, **virtual** inheritance must be used.
- This was defined so that many non-trivial combinations of virtual and non-virtual inheritance just cannot be defined sensibly.

Why Reverse Inheritance (RI)?

- 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).
- Especially library classes often have very large interfaces, but a particular program may need only a small part of the offered methods.
- Why not offer also a generalisation mechanism in a language?
- There have been proposals in this direction for the integration of heterogeneous databases (and other systems)
  - The motivations and solutions are rather different — not discussed now.

Possible disadvantages

- The language becomes larger and more complex — are the advantages worth that?
- “With OI you don’t know the descendants of a class, and with RI you don’t know even all its ancestors.” (Peter Grogono)
  - However, you know all other classes on which a given class *depends* by inheritance.
- 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.
- It was claimed by some critics that reverse inheritance makes separate compilation impossible.
  - We have not studied whether this is quite true or not.
  - At least in Eiffel, fully separate compilation of classes is not generally possible anyway.

Why Eiffel?

Our main goal was not the evolution of Eiffel, but we had to choose some particular language for the first concrete definition and implementation of RI.

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).
  - More so than Java, e.g. concerning basic data types.
- Much attention paid to software engineering — in particular “Design by Contract”™.
- Multiple inheritance and and genericity (semantic, not just syntactic as in C++) from the beginning.

Interesting inheritance mechanisms

- An abstract (deferred) feature can be implemented (effected) in a descendant class either as a method or an attribute (if it has a result type and no parameters).
- Features can be renamed in inheritance.
- In *repeated (fork-join) inheritance*, (re)naming 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 quite 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.
- First standardised by ECMA in 2005, now also an ISO standard.

Pragmatic

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

Previous research

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.

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; I was not yet aware of the paper by Lawson *et al*..
- 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.: Towards fully-fledged reverse inheritance in Eiffel. *SPLST'09 & NW-MODE'09 - Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering* (Peltonen, J., ed.), Tampere University of Technology, 2009, 132–146.

- 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).

An enhanced version with the same title published in *Nordic Journal of Computing* 15:1 (2013), 32–52 (final version accepted in 2010).

Later publications

Chirila, C.-B., Sakkinen, M., Lahire, P., Jurca, I.: Reverse Inheritance in Statically Typed Object-Oriented Programming Languages. *Proceedings of The 4th InternationalWorkshop on MechAnims for SPEcialization, Generalization and Inheritance, MASPEGHI'10* (2010), 20–24.

- Discusses C++, Java and C#.

Chirila, C.-B.: *Generic Mechanisms to Extend Object-Oriented Programming Languages: The Reverse Inheritance Class Relationship* (2010). PhD thesis, University Politehnica of Timisoara.

- Very thorough, especially on the implementation of RI-Eiffel.

Nørmark, K., Thomsen, L. L., Thomsen, B.: Object-oriented programming with gradual abstraction. 41-52 *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12* (Warth, A. ed.), ACM, 2012, 41–52.

- From the viewpoint of dynamic, prototype-based languages.
- Independent, the above literature not referenced.
- The authors are now writing an article that has comparisons with the class-based approaches.

Basic ideas

Single exheritance (SE)

- 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 (parent) S, N *can* be defined to be also a subclass (heir, child) of S.
  - If there are several parents, this applies to each of them separately.
  - In other cases, exheriting *inherited* features of E may be questionable for program comprehension, but it is often useful.

Multiple exheritance (ME)

- N is defined as a superclass of several existing classes $E_1, \ldots, E_m$.
- Only those features common to all E_is can be exherited.
- 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 more complicated adaptations, e.g. conversions of parameters and results, may be needed.
- It seems consistent to allow such adaptations also in SE.

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).
  - Especially preconditions can be problematic, as recognised already by Lawson et al.

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.

Of course, the syntactic definitions will not be the same in the two cases.

Rule 5: Minimal Change of Inheritance Hierarchy
Introducing a foster class must neither delete nor create inheritance relationships (ancestor-descendant relationships) between previously existing classes.

It would be possible and even useful to allow the *creation*.

- But it would slightly change the behaviour of some programs in some particular situations.

Rule 6: Exheritable Features
The features $f_1, \ldots, f_n$ of the respective, different classes $C_1, \ldots, C_n$ 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 $f_1, \ldots, f_n$ can be either immediate or inherited.

For methods, the preconditions must also be considered.

Rule 7: No Repeated Exheritance
Two different features of the same class must not be exherited to the same feature in a foster class.

The equivalent effect *is* possible in Eiffel with RI, but we consider it problematic.

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.
- If an effective (implemented) feature is made deferred in the inheriting class, the keyword **undefine** 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.
- If an effective feature remains effective in the exheriting class, the keyword **moveup** is used.

- The case is different for an *amphibious* feature (see later).

Many adaptations 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.)

```
deferred foster class FIGURE
  exherit
    CIRCLE
      redefine location
      adapt location
      end
    RECTANGLE
      redefine location
      rename display as draw
      end
  all -- all exheritable features
  feature
    location: GEN_POINT
    adapted {CIRCLE}
      to Result.x := Precursor.x/10, Result.y := Precursor.y/10
      from Precursor.x := result.x*10, Precursor.y := result.y*10
  end
end -- class FIGURE
```

Adding a class with both reverse and ordinary inheritance

This is likely to be the more common situation when (and if) RI is used in real-life programming.

- But it had not been considered at all in the papers preceding ours.

We call such a foster class *amphibious*.

- All of its *features* need not be amphibious.

If RI is added to a single-inheritance language, a foster class is necessarily amphibious, unless its children were originally root classes.

- ME is possible only if all children have the same parent.

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.

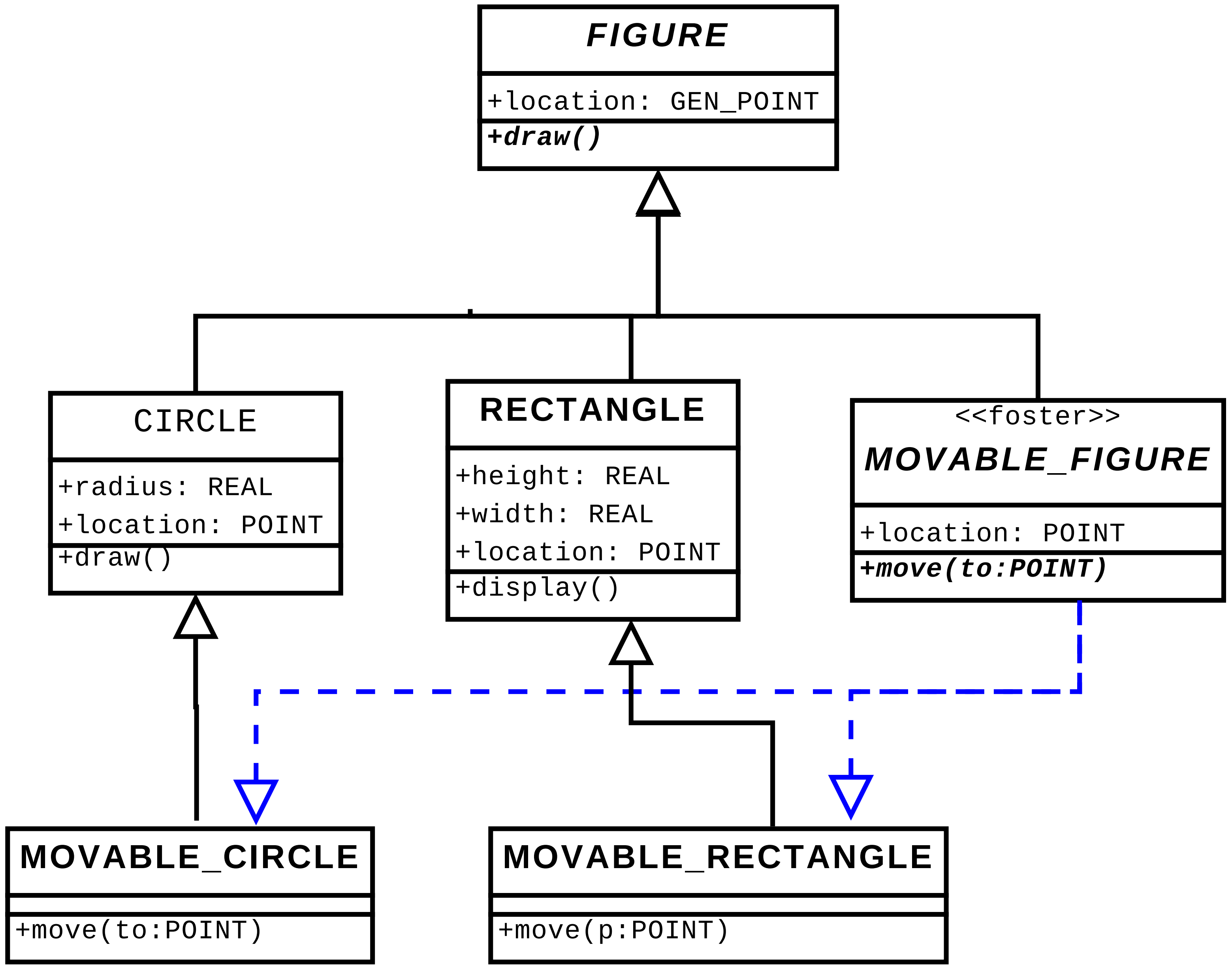
FIGURE
+location: GEN_POINT
+draw()
CIRCLE
+radius: REAL
+location: POINT
+draw()
RECTANGLE
+height: REAL
+width: REAL
+location: POINT
+display()
<<foster>>
MOVABLE_FIGURE
+location: POINT
+move(to:POINT)
MOVABLE_CIRCLE
+move(to:POINT)
MOVABLE_RECTANGLE
+move(p:POINT)

Experiences

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

- Most of RI-Eiffel was also implemented by transformation into standard Eiffel.
  - Non-trivial adaptations of methods and attributes needed special run-time tricks.
- 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 be discussed only briefly or not at all even in the journal paper.
- We had to admit that incompatible assertions can prevent the multiple RI of some methods.
  - But incompatible assertions can similarly prevent the multiple OI of whole classes.
- The required, consistent “reverse thinking” was sometimes surprisingly difficult to us even at later stages of the work.

- We tried to keep the spirit and style of standard Eiffel.
- The task would have been much easier if we had not maintained full backward compatibility with existing Eiffel.
- Some significant weaknesses and some smaller inconveniences of Eiffel were revealed.
  - Most importantly, the unrestricted possibilities to duplicate and unify features in MI can lead to illogical structures that cannot be handled in any consistent manner.
  - It appeared that the simple **select** clause is not sufficient to disambiguate dynamic binding for too spaghetti-like MI.
  - It also appeared that recent Eiffel compilers could not handle even Meyer's old ”intercontinental drivers” example correctly.

Probably, trying to add any large orthogonal enhancement to an existing language will highlight some weak points and inconsistencies of that language.

Further application possibilities for RI

Interface inheritance, not offered by any well-known language (for classes).

- Exheriting all public features of a class as abstract (deferred) yields the interface of the class.

"RI induced by OI"

- When a new class N is defined by multiple (ordinary) inheritance (MI) from the existing classes E_1 and E_2, one or more common features may be detected.
- In many languages, it is not allowed to unify features from E_1 and E_2, unless they originate in a common superclass (ancestor).
- A new common superclass S can be created to which those common features are exherited.

Bridging the gap between *subobject-oriented* (as in C++) and *attribute-oriented* (as in Eiffel) inheritance.

- In a language based on subobject-oriented inheritance.
- Any single feature or any set of features of a class can be made into a subobject by exheriting them into the same foster class.

The gap between the usual *explicit inheritance* and *implicit subtyping* (e.g. Cardelli) can be bridged similarly.

- Our Rule 5 must be relaxed to allow defining new inheritance relationships between existing classes.
- Of course, that will be possible only if they already have an implicit subtyping relationship.
- We could even allow two or more existing classes to be declared identical (as suggested in my 2002 paper).

Continuation of the research

Already performed (very little), ongoing and possible future work:

- Applying RI-Eiffel in practice to get experience about its advantages and disadvantages.
- Defining RI extensions for other suitable languages.
  - The MASPEGHI'10 paper was a modest beginning.
  - Cooperation with Nørmark et al. on dynamic languages seems possible.
- Designing a new language having ordinary and reverse inheritance as mechanisms of equal status.