Electronic Notes in Theoretical Computer Science 82 No. 8 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 20 pages

Extending Java to dynamic object behaviors ⋆

Lorenzo Bettini   Sara Capecchi   Betti Venneri

Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
{bettini,capecchi,venneri}@dsi.unifi.it

Abstract

Class inheritance and dynamic binding are the key features of object-oriented programming and they permit designing and developing complex systems. However, standard class inheritance is essentially static and cannot be directly employed for modeling dynamic object behaviors. In this paper we propose a linguistic extension of Java, called Dec-Java, that is partially inspired by the *decorator* design pattern. This extension permits easily separating the basic features of objects (that are likely not to change during the application) from their behaviors (that, instead, can be composed dynamically at run-time). Thus, Dec-Java enables a dynamic extension and specialization of object responsibilities.

1 Introduction

Object-oriented programming has been successfully used in designing and developing complex software systems. Specific domain entities can be structured in *classes* that play the role of *software modules* and permit abstracting the most significant features, possibly hiding their actual representations and implementations. In this context, class *inheritance* [6,22] is a key feature since it provides means for code reusability and, from the type perspective, it allows the programmer to address flexibility in a safe way.

However, class inheritance is essentially a static mechanism: the relation between a parent and a derived class is established statically and once for all: should this relation be changed, then the program has to be modified and recompiled. The only dynamic feature is represented by *dynamic binding*, i.e., the dynamic selection of a specific method implementation according to the run-time type of an object. This may not suffice for representing the dynamic

⋆ This work has been partially supported by EU within the FET - Global Computing initiative, project AGILE IST-2001-32747 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

©2003 Published by Elsevier Science B. V.

evolution of objects that behave differently depending on their internal state, the context where they are executing or the entities they interact with.

All these possible behaviors may not be completely predictable in advance and they are likely to change after the application has already been developed and used. Thus, apart from the problem of dynamically extending object behaviors, we have also to consider the problem of application scalability to new contexts and domains. While trying to forecast all the possible evolutions of system entities, classes are often designed with many responsibilities, most of which are basically not used. Furthermore, the number of subclasses tend to grow dramatically when trying to compose different functionalities into single modules.

In order to overcome the problems due to the static nature of inheritance, several solutions have been proposed, that we classify into three groups:

- design patterns [9],
- extensions of class-based languages either with new basic concepts [16] or with new features [21,18,7],
- solutions inspired by the object-based model [13,11,20].

In this paper we address the above issue in the specific context of class-based languages, that "form the main stream of object-oriented programming" [1]. The proposed solution is a combination of the three approaches listed above. We propose an extension of Java [2], based on [5] and called *Dec-Java*, with a dynamic object behavior extension/specialization mechanism inspired by the *Decorator* design pattern [9]. This new mechanism directly exploits *delegation* to achieve part of the dynamic flexibility that is typical of object-based models.

We aim at providing a mechanism that separates in different modules the basic features of entities (representing their structure) from the additional ones representing their run-time behaviors. At run-time, these modules can be dynamically composed in order to manipulate object responsibilities and behaviors.

Thus, we achieve a dynamic specialization of methods, which is an alternative to standard inheritance. Inheritance lets the programmer redefine methods belonging to the superclass: a method in the derived class can be completely overridden (i.e., the superclass implementation is never called from within the redefined method). Instead, in our solution, methods are really specialized and not overridden, in the sense that their meaning can only be extended and not totally twisted (and so methods do not perform unexpected actions).

Let us observe that method specialization would be very useful in many situations. Typically many classes in a framework require a method to be specialized, not redefined: for instance, the method `paint` in a GUI framework such as Java AWT, is a callback method that is invoked by the framework when the contents of a window have to be redrawn on the screen. The programmer is required to extend this method in order to take care of drawing

the contents that are specific of the application, while standard window items (menu, toolbars, button, etc.) are drawn by the implementation of the superclasses in the framework. Thus, the programmer has to explicitly call `super.paint()` in his redefined method, otherwise the window will not be correctly redrawn. However, there are no means to require/check this, but (informally) documenting the framework.

Summarizing, we think that our proposal has two main useful features:

- we achieve a good degree of flexibility in dealing with object run-time behaviors, without any substantial type reclassification (as required in other approaches, see, e.g., [7,19]);
- basic features are (logically and physically) separated from object roles, thus class hierarchies do not tend to explode in number and size.

Finally, the extended Java here proposed is translated into standard Java. We remark that the same extension can be applied to other object-oriented languages, such, e.g., C++, since we do not exploit any specific Java feature.

The paper is organized as follows. Section 2 introduces the new constructs of Dec-Java, and applies them to the design of a case study. Section 3 discusses the main features of our approach. Section 4 presents the translation of Dec-Java programs into standard Java programs. Sections 5 and 6 conclude the paper by relating our solution to other proposals presented in the literature.

2 New linguistic constructs for extending object behavior

In this section we introduce the new linguistic constructs that permit dynamically extending the behavior of objects. In order to motivate their introduction, we firstly consider a case study: the design of an interactive game (similar to the classic *Cluedo* detective game). Then, we define new linguistic constructs that characterize the extended language Dec-Java w.r.t. Java. Finally, we discuss Dec-Java features by applying the new constructs to our case study.

2.1 The case study: a Cluedo-like game

We want to design an interactive game that consists in bringing the guilty of a crime to light. The game starts with an introduction of the characters and an explanation of the circumstances in which the crime happened. Every player has:

- a set of ‘skills’ (e.g., observation, agility, etc.) each of which has an associated score that can be modified during the game (for instance it can increase because of a good result or decrease due to a specific action);
- a list of suspects: at the beginning this list contains all the characters; during the game, characters will be dropped from this list one by one until

only the guilty remains;

- a list of weapons among which there is the one used for the crime;
- a list of clues and evidences collected during the game.

The game can be described as a graph of states: when the player gets to a state, for instance the state `LivingRoom`, he can interact with the environment (e.g., open a door, add a clue to the list, etc.). However, the interaction with the state is not the same for everyone; for instance a player that has a high observation score can see more details and thus the environment shown to him is richer in objects than the one shown to others with a lower score. Conversely, the authority score determines the number of allowed actions and the number of states that can be reached by the next transition. This way, *the behaviors associated to each state during a game depend on the player's characteristics at that specific moment*: a state can be seen differently by distinct players and by the same player in different stages of the game.

We would like to structure the possible states of the game so that they can be enriched at run-time with new elements, in such a way that the view of the current state provided to the player can be dynamically built in a smooth way.

A possible approach would be to widely use "**if** statements" inside the state classes, in order to build the right view of the current state according to the player's skills and clues. However, this is not an elegant solution and does not scale well to new versions of the game: to add new clues, skills or actions it would be necessary to change the class representing the state or to define a new subclass representing the changes in the game. Conversely, the use of multiple inheritance would cause an explosion of subclasses in order to represent all possible states.

Thus, it seems to be difficult to model the situation described above with the standard features supported by object-oriented programming. Basically, such mechanisms remain totally static. Our aim is to find a different solution that permits changing the environment a player is allowed to interact with in a specific situation, according to his current characteristics. This solution should also permit easily extending the class hierarchies to represent possible evolutions of the game (e.g., new states, new characters, new skills, etc.) without burdening state and player classes.

2.2 The key idea

We propose to introduce three new constructs in order to achieve a dynamic extension/specialization mechanism of the object behaviors. The basic idea, applied to our case study, can be summarized as follows: at run time an object representing a state (called *component*) is embedded in another object (called *decorator*) that associates to the state the behaviors related to the player's dynamic situation (Figure 1).

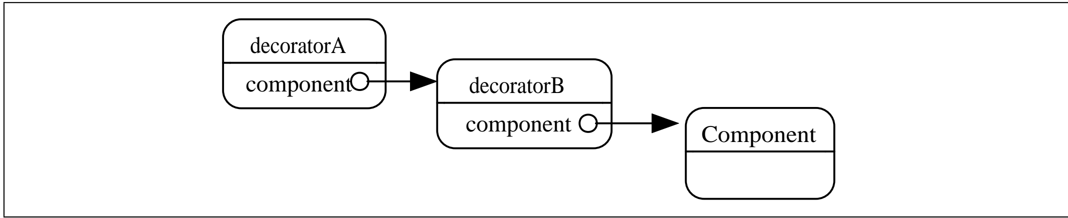

Fig. 1: Decorator instances.

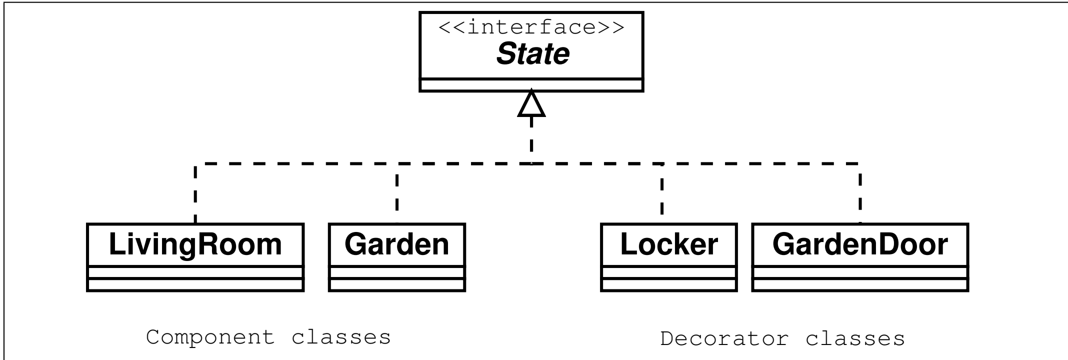

Fig. 2: Some classes of the hierarchy including both the Component and the Decorator classes.

A Decorator object's interface is conforming to the one of the Component so the power of polymorphism can be exploited (Figure 2): we can assign to a variable `S` of type `State` a Decorator object that embeds an instance of (a subclass of) `State`. Every time a client invokes a method m on `S` that belongs to the interface `State`, `S` forwards the method call to its attribute `component` possibly adding code after the forwarded method invocation has returned. This forwarding is implemented through the *delegation* mechanism, i.e., the implicit parameter *this* is bounded to the Decorator object even after the forwarding of the method to its attribute component. Decorator objects can also be recursively nested in order to represent all the evolutions of the `State` instances (Figure 1).

These class and object structures are very similar to those proposed in the design pattern *decorator* [9] (this is the main reason why we use the term "decorate"). However, the decorator pattern implements a *consultation* mechanism instead of delegation [12]: the implicit parameter *this* is bound to the object the method call has been forwarded to. The difference between delegation and consultation is depicted in Figure 3. We observe that delegation is a more powerful mechanism since dynamic binding is not lost during the method call forwarding and permits dynamic object extension as opposed to simple object composition. As a main consequence, our solution is not merely an implementation of the decorator pattern. Unfortunately, in the literature, the term delegation is given different interpretations [1]; in this paper we will use delegation with the meaning described above.

In Figure 2 we distinguish two kinds of classes: the first group repre-

[1] Indeed, in [9], delegation is used with the meaning of consultation.

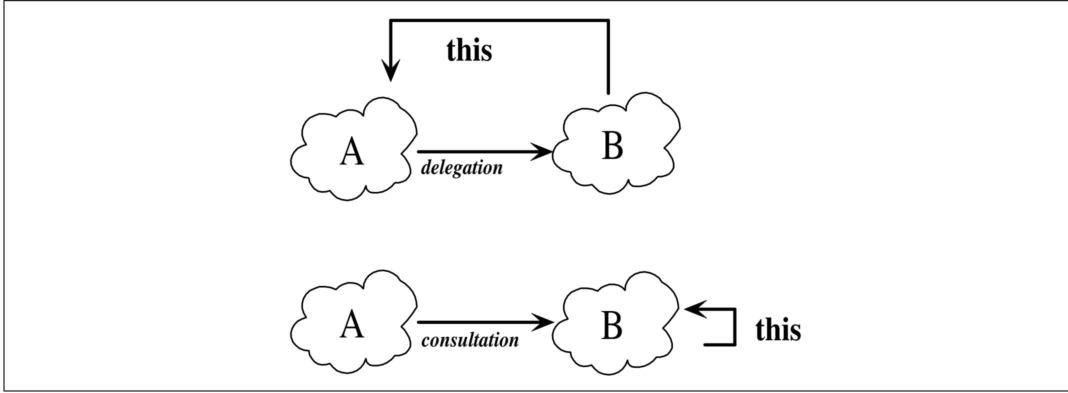

Fig. 3: Consultation and delegation mechanisms.

sents the states of the game and implements their basic features (Component classes); the second group contains classes representing possible evolutions of states (Decorator classes). The instances of the classes of this second group will be composed at run time to create a view of the state that fits the dynamic characteristics of the player.

To this aim we need to define new constructs that allow us to:

- declare that we want to develop a decorator structure based on an interface I (like the one in Figure 2);
- define Decorator classes whose instances can extend, at run time, behaviours of states;
- define Component classes whose instances can be decorated.

2.3 Description of Dec-Java

Following the key idea discussed in the previous section, we propose to extend the language Java by adding the following three new constructs:

(i) **decorate** I, where I is any Java interface. This statement is used to develop a class hierarchy like the one in Figure 2, based on I; after the statement **decorate** I, it is possible to use the **decorator_of** and **decorated_in** constructs to define, respectively, Decorator classes and Component classes.

(ii) **class** A **decorated_in** I, if A has been defined as a class implementing I (i.e., it assumes **class** A **implements** I and **decorate** I). Its meaning consists in that class A is a Component class of I, that is, instances of class A can be decorated at run time by instances of Decorator classes of I. Notice that, from a typing point of view, A is a subtype of I.

(iii) **class** D **decorator_of** I $\{$*bodyD*$\}$. In *bodyD* the programmer:
- can add fields and new methods w.r.t. I;
- can implement some methods belonging to I;
- does not write out any constructor: the appropriate constructor will be

automatically generated as shown in Table 1.[2]
This construct states that instances of class D can be used to decorate, at run time, instances of type X where X is either a Component class of I or another Decorator class of I.

Moreover, a new subtyping relation, namely D is a subtype of I, is introduced, since the class D is considered as a concrete class implementing I. Notice that not all methods of I need to be implemented in $\{bodyD\}$. However, this does not cause run time errors (e.g., "message-not-understood") because instances of D can only be used to decorate Component instances (or, in turn, Decorator objects) of a class implementing (all methods of) I.

Concerning object creation, an instantiation of a Decorator class can only be of the shape **new** $D(x)$ where x is an istance of a class X such that **class** X **decorated_in** I or **class** X **decorator_of** I (nested decorations).

Let us consider now the case of a method invocation m, belonging to the interface I, on an instance of the Decorator class D:

```
obj = new D(x);
obj.m();
```

This invocation will execute firstly $x.m$, that is the implementation of m in the class X, and then the specialization of m in D. In the case of X being in turn a Decorator class of I, this invocation will cause a chain of calls of m, starting from the Component class back to each Decorator class, according to the order of the nested decorators. This mechanism of specialized nested calls will be clear in Section 4, presenting the translation of Dec-Java constructs into Java.

The extended language so obtained is called Dec-Java.

2.4 Using Dec-Java in the case study

Now we can analize the use of the new constructs by applying them to our case study. Listing 1 shows the code related to the example introduced in the previous section. The methods that will be "decorated" (i.e., specialized) are those declared in the interface `State`. There is a method, `display`, for displaying the state to the player, and a method `show_places` that shows the next places that can be reached from the current room.

For instance, `LivingRoom` represents a simple state (i.e., a Component class of `State`) that can be decorated by Decorator classes, such as `Locker`, `LockerKey` and `Garden` in Listing 1.

[2] For the sake of simplicity we chose not to deal with constructors defined by the programmer in this presentation. However, the extension to explicit constructors is trivial.

```
public interface State {
  void display();
  void show_places();
}

decorate State

public class LivingRoom implements State {
  public void display() { print("LivingRoom"); }
  public void show_places() { show(BEDROOM); }
}

class LivingRoom decorated_in State

public class Locker decorator_of State {
  private State contents;
  private boolean open = false;
  public void setOpen() { open = true; }

  public void display() {
    print("Locker");
    if (open)
      contents.display();
    else
      print("closed");
  }
}

public class LockerKey decorator_of State {
  public void display() { print("Locker key"); }
}

public class GardenDoor decorator_of State {
  private boolean shut = true;
  public void setShut(boolean b) { shut = b; }

  public void display() { print("Garden door"); }
  public void show_places() {
    if (! shut)
      show(GARDEN);
  }
}
```

Listing 1: Some classes from the implementation of the game.

Now the code that is responsible for creating the state for the player, will create the living room in the following way:

```
state = new LivingRoom();
gardendoor = new GardenDoor(state);
locker = new Locker(gardendoor);
state = locker;
state.display();
state.show_places();
```

If at some point the player gains a high observation score, then the locker key will become visible, and so the state will be enriched with that element:

```
state = new LockerKey(state);
state.display();
```

Now the player can use the key to open the locker, so the state will be redrawn:

```
locker.setOpen();
state.display();
```

Finally, if the player gains a high score in strenght, then he becomes able to open the garden door, and thus the reachable states will be now more:

```
gardendoor.setShut(false);
state.show_places();
```

Summarizing, the above solution has the following features:

- the view of the current state that is provided to the player can be manipulated at run time and can be easily kept updated;
- the class hierarchy can be easily extended with new classes for states or decorators, without affecting the existing ones; a class representing a state can be designed without worrying about its possible evolutions or different versions: they can be defined in Decorator classes and composed at run time.
- the decorators are still independent from the components (i.e., rooms in the game), thus the actual composition of the room can be reorganized in different instances of the game;
- the class representing the player is not burdened with the management of these evolutions.

Finally, let us remark an important design choice: since we are interested in dynamically changing the behavior of objects, we are only considering methods that do not return any value, i.e., **void** methods. Methods that return values, i.e., functions, can be considered instrumental to methods and for these methods the standard dynamic binding seems to be sufficient. However, the extension for dealing with return values is straightforward.

3 Programming with Dec-Java

The new constructs introduced by Dec-Java extend Java to a new programming framework. In this section we discuss its features, its advantages and its possible uses.

Given a class A that implements an interface I the constructs **decorate** I and **class** A **decorated_in** I permit dynamically extending the behaviors associated to objects of type A. The essential features of this extension can be summarized as follows:

- an extended instance may have additional behaviors compared with the original one (these behaviors are the ones implemented by methods defined in Decorator classes);
- as regards methods belonging to the interface I, the behavior of an extended instance consists of the behavior of the original instance plus the added behavior;
- the extensions can be composed at run-time, by manipulating Decorator objects.

Thus the dynamic "decoration" of an instance has two effects:

(i) as for the functionalities offered to clients, the instance is extended: its interface, understood as the set of messages that can be called on it, is enlarged with the methods of the extension (i.e., methods implemented in the Decorator classes);

(ii) as for the methods belonging to the interface I, instance's behavior is specialized (i.e., its behavior + "decoration").

The second point is interesting because there is no tool, so far, for automatically programming a real dynamic specialization of methods (as hinted in the introduction).

3.1 Representing the evolution of instances

Representing the evolution of object behaviors is a crucial problem in the development of object-oriented software. The objects belonging to the real world are extremely mutable entities and must be described by means of static tools such as classes and inheritance. With Dec-Java we can statically design features that represent object behaviors that can be dynamically composed so that they can adapt to different run-time contexts. Indeed, instance behaviors can dynamically change due to several factors:

- an object can be seen in a different way by its clients or must behave differently according to a specific run-time context. For instance, a person is a customer for a shop, a patient for a doctor, a member for an association. Every time the instance `JohnSmith` of class `Person` gets in touch with other instances of `Person` it should behave accordingly. We should avoid

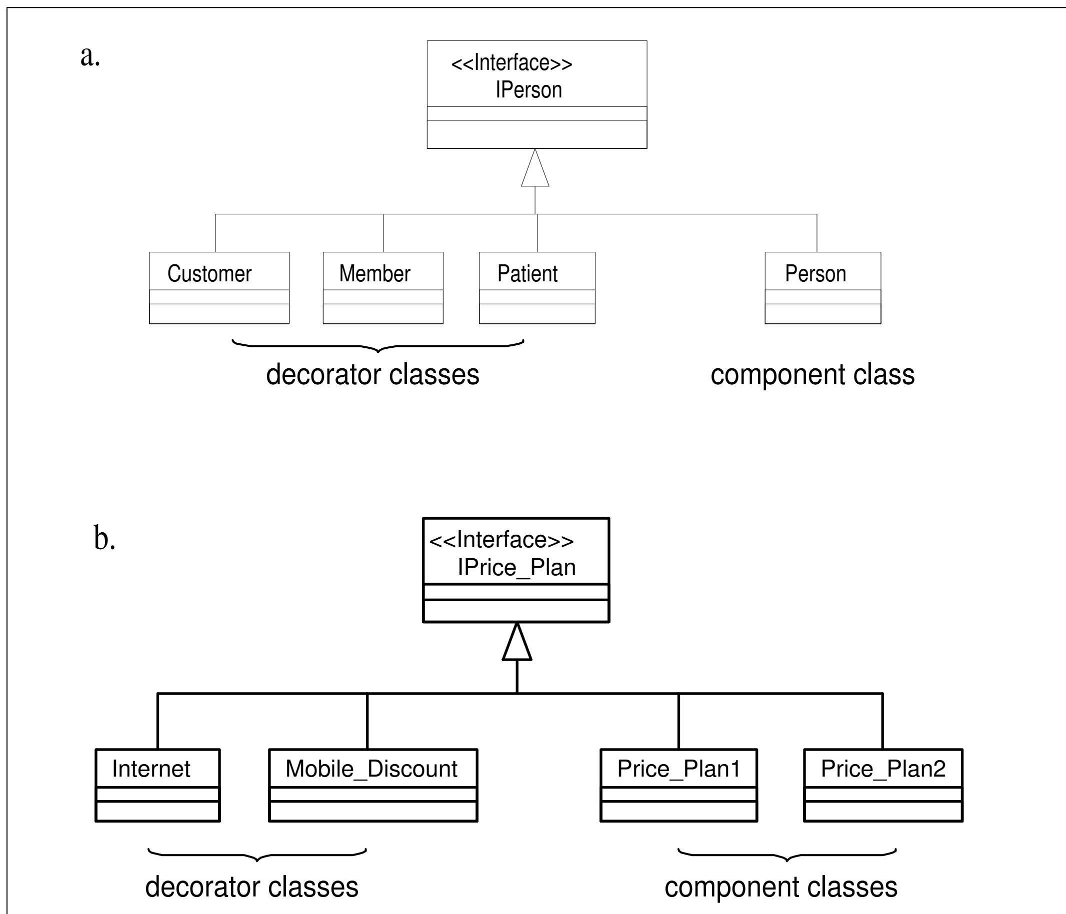

Fig. 4: Examples of class hierarchies with Dec-Java.

burdening the class `Person` with all the functionalities needed to hold all the roles listed above. The solution is to separate the essential features of a person, encapsulated in the class `Person`, from the other features that will be implemented in Decorator classes (Figure 4.a). The case study presented in Section 2.3 is another example: the states evolve at run time depending on the player's characteristics.

- some of the entities belonging to application domains are extremely dynamic so it is very difficult to manage them by means of classes and inheritance. For instance, the phone price plans and the related options (discounts, internet, etc.) continuously change. A customer's plan can be seen as the result of a specific price plan plus some option combination that can change according to the customer's choice. In Figure 4.b we show a solution that allows a separation of price plans from options implementing them in classes `PricePlan` and Decorator classes respectively. The instances of these classes will be composed at run time according to the customer's choices and changes.

3.2 Representing extensions in dedicated modules

The encapsulation of extensions in separated modules (Decorator classes) makes the design more extensible and the code even more reusable.

First of all, we can design the essential features of instances separately from their dynamic evolutions (these will be encapsulated in Decorator classes). This way, there is both a logical (separate modules) and a temporal separation from the design of basic elements and the design of evolutions and accessories related to a set of instances: we can avoid mistakes that are usually made when trying to forecast all possible evolutions of objects (e.g., burdening classes with too many attributes and methods).

Secondly, class hierarchies can be extended more easily just by adding new classes (Component classes or Decorator classes) as in the schemes illustrated in Figure 4.

4 From Dec-Java to Java

In this section we present the semantics of Dec-Java by providing its translation into plain Java. First of all, we introduce some conventions that we use in the sequel.

An environment ξ is a pair of two sets (sub-environments), (ξ_J, ξ_D). ξ_J is a set of standard Java class and interface definitions. ξ_D is a set of triples of the form $\langle I, I_D, D_I \rangle$, where I is the name of an interface, and I_D and D_I are, respectively, the names of the interface and the class generated by the translation w.r.t. I.

We will use ρ with subscripts for denoting a method's parameter type, in particular we will denote a list of method signatures as $m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j})\ ^{j \in J}$, where J is a set of indexes. When we need to refer to methods declared in a specific interface I we will informally write $m_i(\rho_1\ x_1, \ldots, \rho_{m_i}\ x_{m_i})\ ^{m_i \in \textbf{interface}\ I}$. As hinted in the previous section, we only consider **void** methods, so we do not need to explicitly represent the return value type in signatures.

We now define the interpretation function $[\![\ ]\!]$, that, given a Dec-Java definition and an environment ξ, produces a new environment ξ'. More formally, $[\![d]\!]_\xi = \xi'$ means that d is a Dec-Java definition (i.e., a class or interface or one of the three new constructs introduced in the previous section), and ξ' is produced by adding to ξ new classes and interfaces derived from the interpretation of d.

Clearly, the definition of $[\![\ ]\!]$ is trivial in the simple case of d being either a Java class or interface definition:

$$[\![\textbf{class}\ A\ \{bodyA\}]\!]_{(\xi_J,\xi_D)} = (\xi_J \cup \{\textbf{class}\ A\ \{bodyA\}\}, \xi_D)$$
$$[\![\textbf{interface}\ I\ \{bodyI\}]\!]_{(\xi_J,\xi_D)} = (\xi_J \cup \{\textbf{interface}\ I\ \{bodyI\}\}, \xi_D)$$

Now we consider the interesting cases when d is one of the new constructs introduced by Dec-Java. In order to keep the presentation of the translation

simpler, we do not address error handling explicitly.

4.1 decorate

The construct **decorate** I is used for initializing a decorator hierarchy structure based on the interface I. The result is the creation of a new interface I_D, related to I, that contains the same methods as I but with an additional parameter to implement the delegation mechanism. The original method and the new method with the same signature but for the additional parameter are called *twin methods*. Finally, a new class D_I is also created to implement the specialization of methods automatically. Formally,

$$[\![\textbf{decorate } I]\!]_{(\xi_J,\xi_D)} = (\xi'_J, \xi'_D)$$

where ξ'_J and ξ'_D are so defined: let **interface** $I\ \{m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j})\ ^{j\in J}\} \in \xi_J$ then

$$\xi'_J = \xi_J \cup \{\textbf{interface } I_D\ \{bodyI_D\}\} \cup \{\textbf{class } D_I \textbf{ implements } I_D\ \{bodyD_I\}\}$$
$$\xi'_D = \xi_D \cup \langle I, I_D, D_I\rangle$$

where

$$bodyI_D = m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j}, I_D\ x)\ ^{j\in J}$$

$bodyD_I =$
I_D component;
$D_I(I_D\ c)\{\text{component} = c;\}$
$m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j}, I_D\ x)\ \{\text{component}.m_j(x_1, \ldots, x_{m_j}, x);\}\ ^{j\in J}$

4.2 decorator_of

The construct **class** A **decorator_of** I $\{bodyA\}$ is used to define Decorator classes. Then

$$[\![\textbf{class } A \textbf{ decorator_of } I\ \{bodyA\}]\!]_{(\xi_J,\xi_D)} = (\xi'_J, \xi_D)$$

where ξ'_J is so defined: let $\langle I, I_D, D_I\rangle \in \xi_D$, then

$$\xi'_J = \xi_J \cup \{\textbf{class } A \textbf{ extends } D_I \textbf{ implements } I\ \{bodyA'\}\}$$

The Decorator class A can implement (decorate) some of the methods of the interface I, here denoted by $m_k\ ^{k\in K}$; moreover, it can introduce new methods, here denoted by $m_j\ ^{j\in J}$.

The body of the new class, $bodyA'$, is obtained from the original body, $bodyA$, according to the transformation presented in Table 1. Informally speaking, for each method m_j a new twin method is generated and the body of the original method m_j is replaced with a call to the twin method, passing **this** as the additional parameter (this way we implement the delegation mechanism).

The body of the twin method m_j is obtained by applying the following substitution σ to the original body:

- each invocation of a new method introduced by A is replaced with a call to

$$
\begin{array}{l}
bodyA = \\
\quad <\textit{field definitions}> \\
\quad m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j})\ \{body_j\}\ ^{j \in J} \\
\quad m_k(\rho_1\ x_1, \ldots, \rho_{m_k}\ x_{m_k})\ \{body_k\}\ ^{k \in K}
\end{array}
$$

$$
\begin{array}{l}
bodyA' = \\
\quad <\textit{field definitions}> \\
\quad A(I_D\ c)\ \{\mathbf{super}(c);\} \\
\quad m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j})\ \{m_j(x_1, \ldots, x_{m_j}, \mathbf{this});\}\ ^{j \in J} \\
\quad m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j}, I_D\ x)\ \{\sigma(body_j)\}\ ^{j \in J} \\
\quad m_i(\rho_1\ x_1, \ldots, \rho_{m_i}\ x_{m_i})\ \{m_i(x_1, \ldots, x_{m_j}, \mathbf{this});\}\ ^{m_i \in \mathbf{interface}\ I} \\
\quad m_k(\rho_1\ x_1, \ldots, \rho_{m_k}\ x_{m_k}, I_D\ x)\ \{\mathbf{super}.m_k(x_1, \ldots, x_{m_k}, x); \sigma(body_k)\}\ ^{k \in K}
\end{array}
$$

where

$$
\sigma = \{{}^{x.m_i(e_1,\ldots,e_{m_i},x)}/_{m_i(e_1,\ldots,e_{m_i})}\ ^{m_i \in \mathbf{interface}\ I},\quad {}^{m_j(e_1,\ldots,e_{m_j},x)}/_{m_j(e_1,\ldots,e_{m_j})}\ ^{j \in J}\}
$$

Table 1: Translation of the body of a Decorator class.

the corresponding twin method, passing also the additional parameter x of type I_D;

- each invocation of a method belonging to the interface I is replaced with the invocation of the corresponding twin method called on the additional parameter x (again passing x as the additional parameter).

Let us observe that all methods m_i belonging to the interface I are implemented in $bodyA'$; namely, each m_i calls its twin method.

Regarding these twin methods, we have to distinguish two cases:

- methods that are not decorated (implemented) by A are simply inherited from D_I;
- the methods m_k (belonging to I and decorated by A) are modified both in the signature, where the parameter $I_D\ x$ is added, and in the body; namely, firstly the super method m_k of D_I is called and then the substitution σ is applied to the original body.

4.3 decorated_in

The construct **class** A **decorated_in** I adapts an existing class to the decorator hierarchy structure. The transformation of the class is transparent to the old clients. Formally,

$$
[\![\mathbf{class}\ A\ \mathbf{decorated_in}\ I]\!]_{(\xi_J,\xi_D)} = (\xi'_J, \xi_D)
$$

where ξ'_J is so defined: let

- $\langle I, I_D, D_I \rangle \in \xi_D$ and
- **class** A **implements** I $\{bodyA\} \in \xi_J$,

$$
\begin{array}{l}
bodyA = \\
\quad < \textit{field definitions} > \\
\quad m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j})\ \{body_j\}^{\ j \in J} \\
\quad m_i(\rho_1\ x_1, \ldots, \rho_{m_i}\ x_{m_i})\ \{body_i\}^{\ m_i \in \textbf{interface}\ I}
\end{array}
$$

$$
\begin{array}{l}
bodyA' = \\
\quad < \textit{field definitions} > \\
\quad m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j})\ \{m_j(x_1, \ldots, x_{m_j}, \textbf{this});\}^{\ j \in J} \\
\quad m_j(\rho_1\ x_1, \ldots, \rho_{m_j}\ x_{m_j}, I_D\ x)\ \{\sigma(body_j)\}^{\ j \in J} \\
\quad m_i(\rho_1\ x_1, \ldots, \rho_{m_i}\ x_{m_i})\ \{m_i(x_1, \ldots, x_{m_j}, \textbf{this});\}^{\ m_i \in \textbf{interface}\ I} \\
\quad m_i(\rho_1\ x_1, \ldots, \rho_{m_i}\ x_{m_i}, I_D\ x)\ \{\sigma(body_i)\}^{\ m_i \in \textbf{interface}\ I}
\end{array}
$$

where

$$
\sigma = \{{}^{x.m_i(e_1,\ldots,e_{m_i},x)}/_{m_i(e_1,\ldots,e_{m_i})}\ {}^{m_i \in \textbf{interface}\ I},\ {}^{m_j(e_1,\ldots,e_{m_j},x)}/_{m_j(e_1,\ldots,e_{m_j})}\ {}^{j \in J}\}
$$

Table 2: Translation of the body of an adapted class.

then

$$
\begin{array}{l}
\xi'_J = \\
\quad (\xi_J - \{\textbf{class}\ A\ \textbf{implements}\ I\ \{bodyA\}\}) \cup \\
\quad \{\textbf{class}\ A\ \textbf{implements}\ I, I_D\ \{bodyA'\}\}
\end{array}
$$

The body of the modified class, $bodyA'$, is obtained from the original body, $bodyA$, according to the transformation presented in Table 2. Basically, all the methods originally defined in the class A are inserted in the generated class and their bodies are changed so that they simply implement the delegation mechanism by calling the corresponding twin methods. These introduced twin methods contain the original bodies of the corresponding methods after applying the same substitution σ described in Section 4.2.

4.4 Translating a program

We define a Dec-Java program as a set of definitions $d_1 \ldots d_k$ where d_i is either a Java class or interface definition or one of the added construct (**decorate** I, **class** A **decorator_of** I $\{bodyA\}$, **class** A **decorated_in** I). Thus, a program can be defined as a sequence of definitions, that is $P = d\ P_1$ where P_1 can be empty (denoted by **nil**). So the interpretation of a Dec-Java program can be defined recursively as follows:

$$
\begin{array}{l}
[\![\textbf{nil}]\!]_\xi = \xi \\
[\![d\ P]\!]_{(\xi_J, \xi_D)} = [\![P]\!]_{(\xi'_J, \xi'_D)} \text{ where } [\![d]\!]_{(\xi_J, \xi_D)} = (\xi'_J, \xi'_D)
\end{array}
$$

Finally, let us observe that for any Dec-Java program P, $[\![P]\!]_\xi = (\xi'_J, \xi'_D)$, where ξ'_J is a Java program. Thus, the translation from Dec-Java to Java, here denoted by $\mapsto$, can be simply defined by interpreting a Dec-Java program in the empty environment and then by taking the obtained ξ'_J as the result. More formally, for any Dec-Java program P,

$$
P \mapsto \xi'_J \text{ if and only if } [\![P]\!]_{(\emptyset,\emptyset)} = (\xi'_J, \xi'_D).
$$

For lack of space, we omit here a complete formalization of static and dynamic semantics of Dec-Java. These topics will be presented in a future extended version of this paper, where main theoretical properties are proved, namely type safety, subject reduction and correctness of the translation into plain Java.

5 Related works

In order to highlight the main features of our approach we compare our solution with alternative approaches and mechanisms.

Both Dec-Java and the *mixin based* approach [4,8,3] encapsulate extensions in classes making them reusable. A *mixin* (a class definition parameterized over the superclass) can be viewed as a function that takes a class as a parameter and derives a new subclass from it. The same mixin can be applied to many classes (the operation is known as *mixin application*), obtaining a family of subclasses with the same set of methods added and/or redefined. As for standard inheritance, the main difference with Dec-Java is that when we compose one or more mixins with a class, a new class, and therefore a new type, is created, while, in Dec-Java, the composition is moved at instance level so neither new classes nor new types are created.

We observe that our mechanism of method specialization offers a similar computational "feel" as the *Beta* inheritance [15] that is designed to avoid the replacement of a method by a completely different method in subclasses (as via standard overriding). A Beta virtual method in a class A can be seen as a function of its redefined version in a subclass B: an **inner** statement in the body of the method in class A calls its redefined version in B. Thus, the **inner** call mechanism along a hierarchy of subclasses has the same effect of a chain of method-invocations along a composition of nested Decorator instances in Dec-Java. However, there is a basic difference between the two approaches: Dec-Java decorations are applied to objects and can be dynamically composed in several ways, while Beta inheritance is a static mechanism concerning class definition.

More in general we can observe that the dynamic flexibility achieved by Dec-Java relies in that it enables a run-time decoration of objects, while both mixin and Beta approaches allow to statically "decorate" code (classes).

Concerning *delegation*, we remark that the delegation mechanism shown in Section 2.2 is used by *prototype-based* languages to share code. Prototype-based languages [14] drop the notions of class and inheritance to achieve a higher dynamic flexibility at the cost of the detriment of safety. The lack of a static type system easily leads to run time errors such as "message-not-understood". Because of these problems, prototype-based languages had not the same success as class-based languages. However, there have been many attempts to integrate the delegation mechanism in class based languages to achieve a higher dynamic flexibility [11,13]. These integrations turned out

to be quite hard to use. Our solution can be seen as a partial integration of delegation in a class-based language: this mechanism is restricted to the methods belonging to a specific interface. This way, we achieve part of the dynamic flexibility typical of prototypes (we can decide at run-time which features are associated to an instance), while maintaining the static safety of a class-based approach. Moreover, the new constructs are quite easy to use for a Java programmer.

Regarding the *design pattern* approach, we have already mentioned that the mechanism implemented in Dec-Java is inspired by the design pattern *decorator* [9]. In fact, the constructs introduced in Dec-Java can be used as a tool for the automatic development of this pattern whose implementation requires additional programming. However, the mechanism presented here goes beyond the functionalities offered by the pattern decorator.

The main difference is that the consultation mechanism is used by the pattern decorator, while delegation characterizes our solution (see Section 2.2). As a consequence, dynamic binding can be exploited during the method call forwarding in Dec-Java, while this is not a feature of the pattern decorator.

Another difference is in the definition of Decorator classes. Our implementation allows the programmer to add code to methods belonging to the decorated interface, but this code is automatically inserted after the call of the same method on the component attribute. In the decorator pattern the code can be added before or after this call. We took this decision to be sure that the code of the component method is always executed in order to get a real specialization for methods belonging to the decorated interface. Of course, we can extend Dec-Java with mechanisms for explicitly stating whether the added code has to be executed after or before the component's method.

Finally, *Fickle* [7] implements a dynamic object *reclassification* (i.e., objects can change class membership at run-time) in order to represent object evolutions and changes. For example, an instance of class `Student` can change class at run time becoming an instance of class `Employee`. Our approach is different in that we let incrementally add features to a kernel of essential features that do not change (these are the ones belonging to the Component classes). In Fickle an object loses the features that were typical of the previous role, when it is reclassified:

(i) an instance of class `Student` loses its role if it is reclassified as an `Employee` while, a person, could be a student and an employee at the same time; with Dec-Java we can decorate an instance of class `Person` with two Decorator instances, one of type `Student` and one of type `Employee`;

(ii) two instances of different classes can be reclassified to the same class without considering their previous differences (a mechanic and a teacher should not be reclassified to the same `Soldier` class during a war). In our opinion, their skills should be preserved because their performances as soldiers and their usefulness are highly influenced by their former job.

More in general, Dec-Java is oriented to model a dynamic notion of *non-exclusive roles* [10] rather than a dynamic change of mutually exclusive types.

6 Conclusions

We presented an extension of Java, Dec-Java, supporting a dynamic reconfiguration of objects w.r.t. to their behaviors. Namely, any instance of type I (interface) can be dynamically decorated by specializing (extending) the methods declared in I. A prototype implementation of Dec-Java has been developed in [17] in order to experiment with performance and applicability of our solution. In particular, this implementation includes other features in Decorator and Component classes, such as fields, constructors and methods returning values, that have been avoided in this paper for simplicity. More in general, the present work is a first step towards a working extension of Java, exploiting several mechanisms for achieving a reasonable degree of dynamic flexibility in object manipulation.

The specialization mechanism implemented by Dec-Java and the standard inheritance are both tools that permit code extension and reuse. Concluding, we want to point out their main differences, in order to underline the cases where Dec-Java constructs are a significant alternative to standard inheritance.

First of all, inheritance permits specializing and extending classes while Dec-Java works on instances. As a consequence, when creating a subclass we also define a new type. Instead, decorating an object at run-time simply specializes and extends its features without creating a new type. Inheritance is a static tool that lets us classify objects by means of the subtype relation. This benefit is lost when we decorate instances at run-time because of the dynamic nature of this action: we gain flexibility losing the equivalence specialization = subtyping. Thus, when the extensions are very mutable or numerous and it is not important to include them in the definition of a type, it is better to define Decorator classes instead of subclasses.

Secondly, inheritance and Dec-Java implement two different kinds of specialization, as explained in the introduction; namely, Dec-Java supports a real extension/specialization of methods. Finally, Dec-Java permits encapsulating extensions in modules so they can be reused while, using inheritance, if we want to add the same code to two classes we need to write it for each subclass.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

[3] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In R. Guerraoui, editor, *Proceedings ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer-Verlag, 1999.

[4] Gilad Bracha and William Cook. Mixin-Based Inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).

[5] Sara Capecchi. Il pattern Decorator per l'estensione dei linguaggi orientati agli oggetti. Master's thesis, Dip. Sistemi e Informatica, Università di Firenze, 2002.

[6] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: $Fickle_{\|}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

[8] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM Press, 1998.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object Oriented Software*. Addison Wesley, 1994.

[10] G. Ghelli and D. Palmerini. Foundations for extensible objects with roles, extended abstract. In *Proc. of the 6th Workshop on Foundations of Object-Oriented Languages (FOOL)*, 1999.

[11] JavaSoft. The Glascow Model. 1997. Available at `www.javasoft.com`.

[12] Gunter Kniesel. Delegation for Java: API or Language Extension? Technical Report IAI-TR-98-5, University of Bonn, May 1997.

[13] Gunter Kniesel. *Darwin - A Unified Model of Sharing for Object-Oriented Programmning*. PhD thesis, University of Bonn, 1999.

[14] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In Norman Meyrowitz, editor, *Proc. of OOPSLA*, volume 22, pages 214–223. ACM Press, 1987.

[15] Ole Lehrmann Madsen, Birger Mller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[16] Mira Mezini. *Variational Object Oriented Programmnig Beyond Classes and Inheritance*. PhD thesis, College of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997.

[17] Marco Naldini. Realizzazione di un'estensione di Java verso l'evoluzione dinamica degli oggetti. Master's thesis, Dip. Sistemi e Informatica, Università di Firenze, 2003. Forthcoming.

[18] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of Object Behavior Using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.

[19] Manuel Serrano. Wide Classes. In R. Guerraoui, editor, *Proceedings ECOOP'99*, volume 1628 of *LNCS*, pages 391–415, Lisbon, Portugal, 1999. Springer-Verlag.

[20] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.

[21] John Vlissides. Subject-Oriented Design. *C++ Report*, February 1998.

[22] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, 1990. Expansion of Oct 4 OOPSLA '89 Keynote Talk.