Using the PAC-Amodeus Model and Design Patterns to Make Interactive an Existing Object-Oriented Kernel

Thierry Duval — François Pennaneac'h
IRISA
Campus de Beaulieu, F-35042 Rennes cedex
{Thierry.Duval, Francois.Pennaneach}@irisa.fr

Abstract

This paper presents an efficient way to provide a graphical interactive visualisation to a non-interactive existing object oriented application. Assuming that the initial application uses an 'Abstract Factory' pattern (GoF87) in order to create new objects, our aim is achieved by using the PAC-Amodeus model and extending the existing objects to create intermediate components, using object oriented techniques: inheritance, polymorphism and dynamic binding, using the 'Proxy' pattern (GoF207). Although our field of interest is physical and behavioural simulation, the techniques developed in this paper can be applied to any non–interactive object oriented existing kernel. Then, we present a complete simulation example 'Bugs life' to illustrate the use of our method. Finally, we point out the limits of our approach, and we suggest new directions for further work.

1: Introduction

Quite often, a specialist of a particular area is able to structure an application as a hierarchy of classes (Java, C++ or Eiffel classes for example), without taking any care about graphical user-interface. This is particularly true in the field of simulation, as in [3], but also for any existing object oriented non–interactive kernel.

For example, if a simulation specialist wants to transform his initial simulation, he may have the possibility to use a particular development environment, such as those described in [5] or [11], which will provide an interactive visualisation to the simulation. But in this case, he will have to re-design and to re-write his simulation in order to adapt it to this kind of environment.

The other solution is to adapt the existing simulation software in order to obtain an interactive simulation. So, our aim is here to find the best way to provide a graphical user-interface to this kind of application. A very important thing is to let this initial application unchanged, in order to allow the programmer to make it evolve. This point is interesting for any category of existing application.

Therefore, we propose an efficient way to use a particular software architectural model for interactive applications: the PAC-Amodeus model [10]. This model is derivated from PAC [1] (which is quite different from MVC [8]) and ARCH [4], and is very well suited for the reuse of an existing object kernel [2]. More precisely, it ensures that the final application will respect software engineering concepts such as the independence of the simulation kernel from the physical graphic representation.

In this paper, we propose to combine efficiently some well known design patterns described in the 'Gang of Four' book [7] with the PAC-Amodeus model, in order to obtain an interactive simulation without modifying a non-interactive existing simulation, as it has already been discussed in [6]. We

avoid code duplication between the initial simulation and the interactive one, and we discuss how to minimise code duplication among the new components we create.

2: Useful design patterns for making evolution possible and easy

A typical simulation application consists in a scheduler managing simulation objects, asking each of them to evolve at each simulation step. So, most of these applications can be built using the 'Template Method' pattern (GoF325). For example, each simulation object will have to provide an 'evolve' method in its programming interface, some families of simulation objects will perhaps define this method in terms of call of some abstract methods that will have to be defined in their subclasses. The use of such a pattern is not absolutely necessary, but it helps in fixing a general architecture for the simulation, so that it is easier to concentrate on the behaviours of the simulation objects as they can be defined in subclasses.

Another very interesting pattern is the 'Abstract Factory' pattern (GoF87), because it eases the iterative prototyping of a simulation. The 'Abstract Factory' is in charge of providing new simulation objects to any software component of the simulation. Then, thanks to this pattern, it becomes very easy to provide more efficient subclasses of these simulation objects classes, without changing any of the software components of the simulation: the only thing needed is to provide a new 'Factory' able to provide new simulation objects. The use of this pattern is absolutely necessary to our transformation method, it is the only particular mechanism required to our simulation applications in order to be able to migrate easily toward interactive simulations.

3: The Human–Computer Interface models

Now, we are going to make a point about the PAC and PAC-Amodeus models, before trying to use them as a combination of design patterns.

3.1: The PAC model

PAC [1] is a multi-agents model which main principles are the notions of faceted agents and recursive decomposition. An interactive system can be modelled as a hierarchy of PAC agents, as shown in figure 1.

A PAC agent has three complementary views:

- the Presentation: it defines the agent behaviour as it can be seen by an end-user, it manages both inputs (the end-user actions) and outputs (the end-user perception),
- the Abstraction: it defines the features of the agent independently from its possible presentation, it is the Functional Core of the agent,
- the Control: its purpose is to serve as a bridge between Presentation and Abstraction for the maintain of consistency, and it manages the communications between PAC agents and between its Abstraction and Presentation associated components.

In fact, although at first glance this model looks very much like the MVC model [8], PAC is quite different from it. First, its presentation features are concentrated in the same Presentation component, which is not the case with MVC, which puts the input features in its Controller component and the output features in its View component. Second, and even more important, the Abstraction components are not aware of the Control and Presentation components, which is not the case

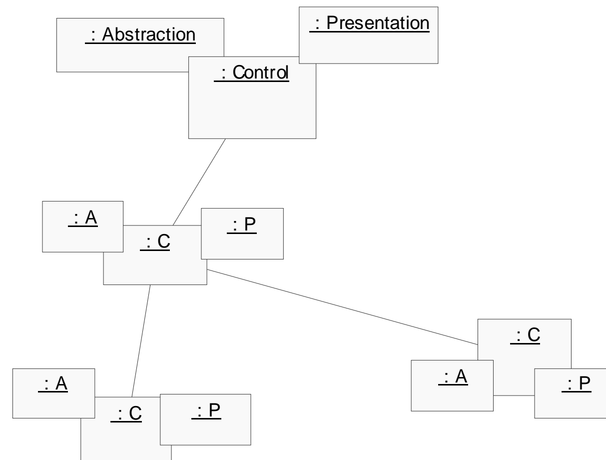

Figure 1. PAC modelling of the software components of an interactive system

with MVC where the Model component knows about the View and the Controller components. So, PAC ensures a better separation than MVC between the features of a non-interactive object oriented kernel and the interactive components of the complete application.

3.2: The PAC-Amodeus model

The PAC-Amodeus model [10], described in figure 2, shares its main components with the ARCH model [4]. It describes its main component, the Dialogue Controller component, as a hierarchy of PAC agents.

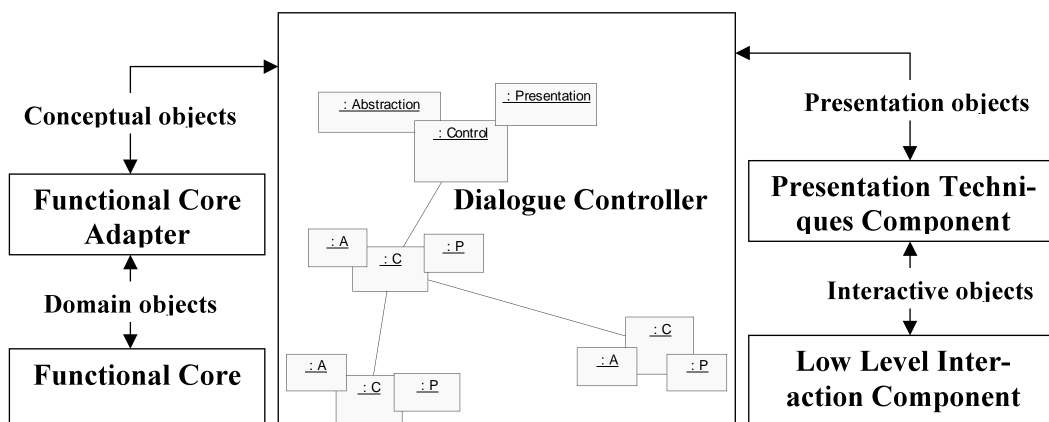

Figure 2. PAC-Amodeus modelling of the software components of an interactive system

The Functional Core contains the main features of the domain of the application. The Functional Core Adapter serves as a bridge between domain objects and conceptual objects exported toward the end-user. The Dialogue Controller manages the scheduling of the tasks and the dialogues threads, using a PAC agents hierarchy, it ensures the correspondences between the conceptual objects and the presentation objects. The Presentation Techniques Component defines the correspondence rules between the presentation objects and the interactive objects. Last, Low Level Interaction Compo-

nent stands for the hardware and software targeted platform for the effective implementation of the interactions.

4: Principle of the transformation method

4.1: The initial application

The example we are going to use to illustrate the method is shown in figure 3: an object `A1` which sends the `move` message to an object `A2`. The problem is here to insert these components most efficiently within a PAC-Amodeus modelling, and without modifying these initial components.

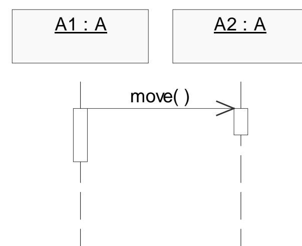

Figure 3. Initial relation between two simulation objects

4.2: An original PAC-Amodeus implementation

We use common object oriented techniques: inheritance, polymorphism and dynamic binding, such as the ones we find in C++, Eiffel or Java. We combine well known design patterns in order to describe our transformation method, as proposed in the chapter 7 of [9], but quite in a different way, as our method does not rely on the 'Observer' pattern (GoF293).

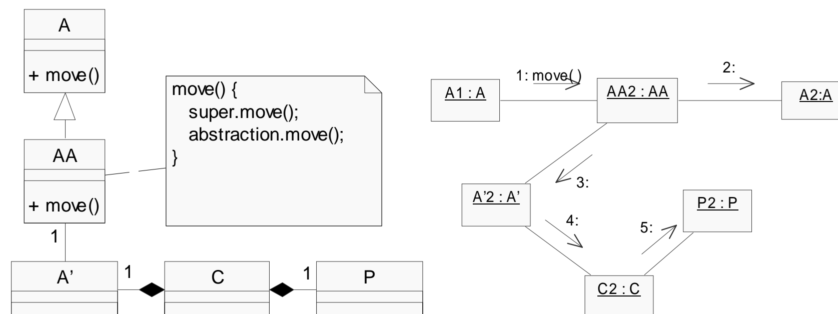

Figure 4. Transformation method using the PAC-Amodeus model

A way to implement that is to make the Functional Core Adapter components inherit from the initial Functional Core ones, and to adjoin them a way to access the abstractions of the PAC components.

In fact, it is what is described in the 'Proxy' pattern (GoF207). Our case study is described in figure 4: the `AA` class is the 'Proxy' and the `A` class is the 'RealSubject', with a particular implementation: `AA` inherits from `A`, and redefines the `move` method, in order to call its inherited

one before asking the A' abstraction to move, then A' asks C to move, and finally C asks its presentation P to move also. Here, using inheritance instead of delegation allows an optimal reuse of the existing code, as the interface of AA will be the same as the interface of A, so we have to redefine the only methods that have an impact on the visualisation of the simulation objects.

This works fine, and the use of the 'Abstract Factory' pattern enables to provide an instance of the AA class instead of an instance of the A class when needed by the simulation software components.

The only problem is that this method is quite heavy: we have to create four new classes for each existing class to be modified. Of course, some of these classes can be written in quite a systematic way, and most of the methods of the Adapter classes and of the Abstraction classes are mainly relays toward other classes methods.

4.3: An optimisation of the implementation

We propose an optimisation of this implementation. First, PAC-Amodeus authorizes empty adapter components, so, if there are no special needs, as in our case, we can empty this level. Second, PAC allows some agents to lack one or two facettes, so here, we can suppress the abstraction facette of the PAC agents.

Then, the solution is to create the new control components by inheritance from the initial simulation application ones, as shown in figure 5 .

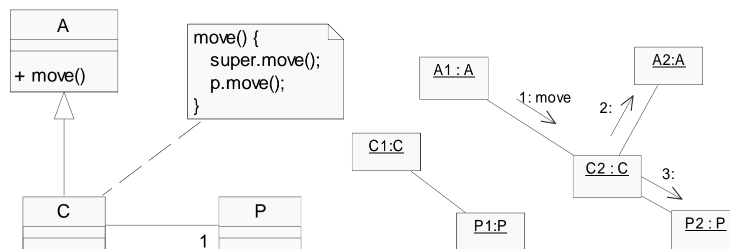

Figure 5. Optimisation of the transformation method

With our small example, the C control component inherits from the A application component, then we provide to it a P presentation component. We must also let the control components know each other, while each initial simulation application component believes in fact that it is in touch with another initial object.

So, the C control inherits from A of a method that decides to ask another simulation object instance of the A class to move. In fact, C is controlling the situation, as it makes the choice to let its inherited method to do its job. In fact, A now makes reference to an instance of C, so the move method which is invoked is a method redefined in C, which will invoke the A move method, before asking its P presentation component to refresh its visualisation.

5: A case study: 'Bugs life' simulation with Java

Now, here is a simulation example where entities (we call them 'bugs') initially evolve within a 2D (non graphical) environment. We used the 'Template Method' pattern (GoF325) to structure our application: a scheduler asks these entities to 'live'. An abstract class describes with a method what is to 'live': a succession of calls to deferred methods: 'move', 'grow up', 'eat', 'get tired',

sometimes 'reproduce', and if 'too old' or 'too weak', 'die'. The concrete subclasses implement these methods.

This simulation application is written in Java.

5.1: The application

Several components are parts of this application:

- bacteria (bugs' food),
- a reserve of bacteria,
- several sorts of bugs (basic, erratic, sniffer, hungry, long cruiser, cannibal),
- a population of bugs, which makes them live,
- a life area, within which bugs live, and which owns a reserve of bacteria and a population of bugs,
- a factory which aim is to provide the new simulation objects to any of the software components of the simulation,
- a life application, which creates all the simulation components, and then schedules them.

The bacteria know that they are within a reserve, so when their energy level is getting too low, they signal it to the reserve which suppress them.

The bugs know they form a population and they are evolving in a life area. They ask this area for finding food and for staying within its bounds. Like the bacteria, when their energy is getting too low, they send a message to their population which suppress them.

The life application is in charge with the creation of these entities, using an appropriate factory. It makes them live by the way of the population, as long as it contains bugs. Then, a main program is in charge with the creation of the good factory and the life application.

5.2: Initial requirements

To ensure future consistency, we suggest to define some interfaces to let our programs manipulate references toward instances of classes which implement these interfaces.

Here are our interfaces:

- `Bacteria`,
- `Bug`,
- `Reserve`,
- `Population`,
- `LifeArea`,
- `Factory`,
- `Life`.

In practice, the `Bug` interface inherits from the `Bacteria` interface, and the `Population` interface inherits from the `Reserve` interface. It is not for a polymorphism use but simply for code reuse.

5.3: Implementation of the initial simulation

As shown in figure 6, the main classes we create to achieve our simulation are:

- ABacteria: implements the Bacteria interface,
- ABug: implements the Bug interface, (and inherits from ABacteria): this is an abstract class which factorises attributes and common methods for all the bugs; thanks to its "live" method which calls several abstract methods, it is this way we implement here the "Template Method" pattern,
- ABasic, AErratic, ASniffer, AHungry, ALongCruiser and ACannibal: they all inherit (more or less directly) from the ABug class, they are the concrete classes for bugs,
- AReserve: implements the Reserve interface, manages Bacterias,
- APopulation: implements the Population interface (and inherits from AReserve), manages Bugs,
- ALifeArea: implements the LifeArea interface, manages a Population composed of Bugs and a Reserve composed of Bacterias,
- AFactory: implements the Factory interface, provides the instances of the previous concrete types,
- ALife: implements the Life interface, creates all the simulation components and schedules them.

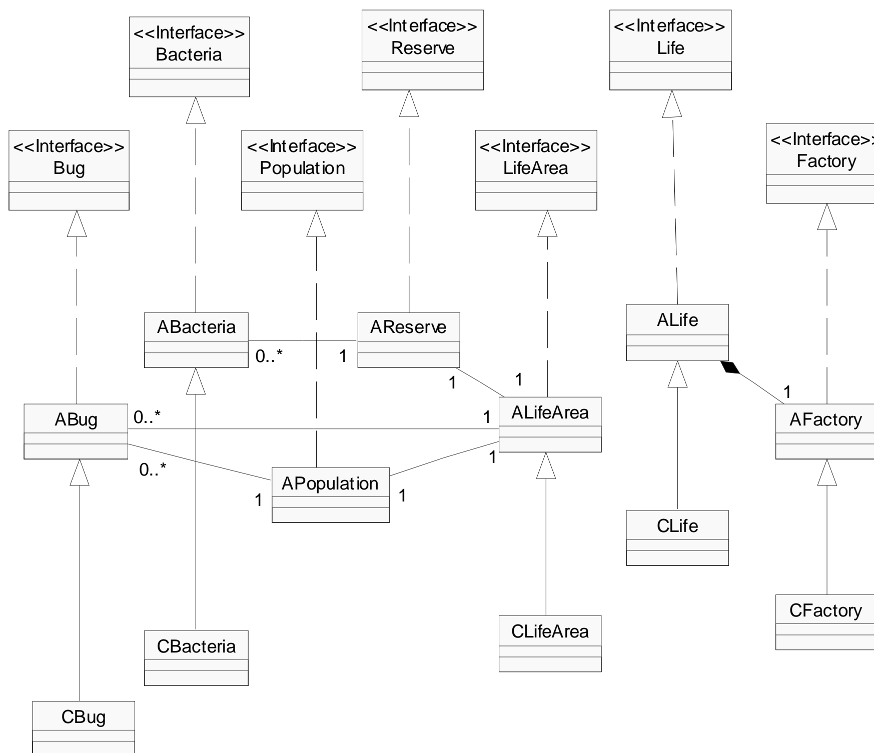

Figure 6. Partial UML modelling of the classes in our simulation

Starting from this initial non-interactive simulation, we are going to apply our transformation method to obtain an interactive simulation.

5.4: Implementation of the interactive simulation

First, we need to define the CBug interface, which extends the Bug interface. Then we create the following classes:

- CBacteria: implements the Bacteria interface,
- CBasic, CErratic, CSniffer, CHungry, CLongCruiser and CCannibal: they all implement the CBug interface and they extend respectively ABasic, AErratic, ASniffer, AHungry, ALongCruiser and ACannibal,
- CLifeArea: extends AZoneDeVie,
- CFactory: extends AFactory, provides the instances of the new concrete types,
- CLife: extends ALife, manages the other Human–Computer Interface components.

There is no use to define new classes inheriting from AReservoir and APopulation, as these two classes can manage CBacteria and CBug instances.

A resulting partial UML model is presented in figure 6. This would be the effective model if there was only one category of bugs: then the ABug would be a concrete one, and there would be also a concrete CBug class. As the real situation is a little bit more sophisticated, the detailed model of this hierarchy of bugs will be discussed further, and will be illustrated in figure 8.

```
public class CBasic extends ABasic implements CBug {
   private PBug presentation ;
   ...
   public void setPosition (final Point position) {
      super.setPosition (position) ;
      presentation.setLocation (getPosition ()) ;
   }
   public void die () {
      super.die () ;
      if (dead ()) {
         presentation.erase () ;
      }
   }
   ...
}
```

Figure 7. Typical methods of a control component

Typically, the methods redefined in these control components are like the ones presented in figure 7: they ensure a redirection toward the inherited part, and they ensure also consistency between the inherited component (the initial application component) and the presentation associated component. It is important to notice that this control component does not make any assumption about the success or fail of the methods it relays to its inherited application part: for example the setPosition method proposes a new position by calling the inherited setPosition method, and then asks the presentation associated component to update its position by giving it a new position obtained by a call to the getPosition method, as it does not know if the proposed position has been effectively reached.

Several other control classes are needed to obtain the final interactive simulation, but they do not directly interact with the classes of the initial simulation.

Last, only three classes are needed for the visualisation of our bugs: `PBacteria`, `PBug` and `PLifeArea`, as all the different types of bugs share the same presentation, parameterized with a particular color. The only important thing to know about these classes is that each of them has the knowledge of its associated control component. It is important because a presentation facette may receive events from an end-user that it has to send to its associated control, in order to allow interaction.

5.5: Problem of an existing hierarchy

We have here a little problem due to Java simple inheritance mechanism: as shown in figure 8, there is no particular relation between `CErratic` and `CBasic` as there is between `AErratic` and `ABasic`, so the work that has been done once when writing `CBasic` has to be repeated for `CErratic`, `CSniffer`, `CHungry`, `CLongCruiser` and `CCannibal`.

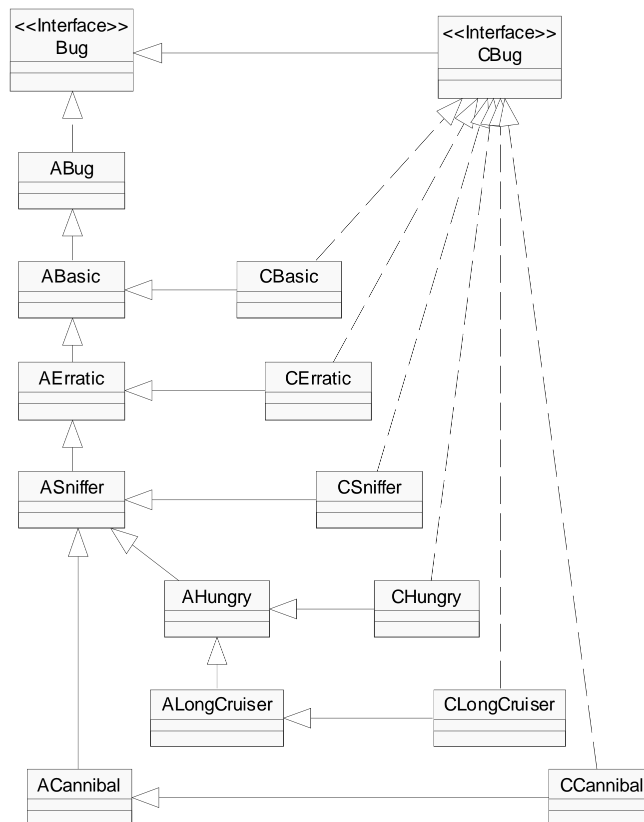

Figure 8. Inheritance problems in the case of an existing hierarchy

For example, C++ and Eiffel would allow `CErratic` to inherit from `AErratic` and `CBasic`, which would reduce the work to be done. In the case of C++, it would be virtual inheritance because of diamond inheritance.

So, another solution is here to create a quite bigger class, named for example `CGenericBug`, and which encapsulates an instance of some real initial `Bug`. Using the Java RTTI mechanism,

this class is parameterized with the name of that initial bug in order to be able to make the good object instantiation. The first problem is then that with such a delegation, each method of the initial object has to be defined in this new class, not only the methods implied with graphics. The second problem, more important, is that the code of all these methods is not so systematic as in the case of inheritance: it is not enough to systematically ask the encapsulated object to invoke the correct method, and then eventually to update the presentation associated object. The reason is that the initial methods invoked can be some combinations of methods that all can have presentation effects. So, the compound methods have to be totally re-written, as in the initial code, but now in order to invoke the newly defined elementary methods. It works (we have also implemented this 'generic' class), but it clearly leads to code duplication between the initial simulation and the interactive one.

What would be great would be to be able to declare that this `CGenericBug` class should dynamically inherit from the initial class which name is provided as a parameter: it would then be possible for other components to dynamically instantiate the good object with correct inheritance. The code of this class would be as simple as the code for `CBasic`, and would have to be written only once. Of course, it should be necessary that the dynamical inheritance should at least ensure that the obtained class would inherit from a particular class, for example `ABug`, or would implement a particular interface, for example `CBug`.

6: Other Implementations with C++

Two other implementations of this simulation application have been realized using the C++ language: both share the same simulation non–interactive kernel. These applications differ only from the visualisation they offer: one uses X-Window as graphics, the other uses OpenInventor.

The main advantage with C++ for such a method is that it eases the development process, allowing multiple inheritance, which reduces the code to be written. An UML subset of this C++ bugs hierarchy is shown figure 9.

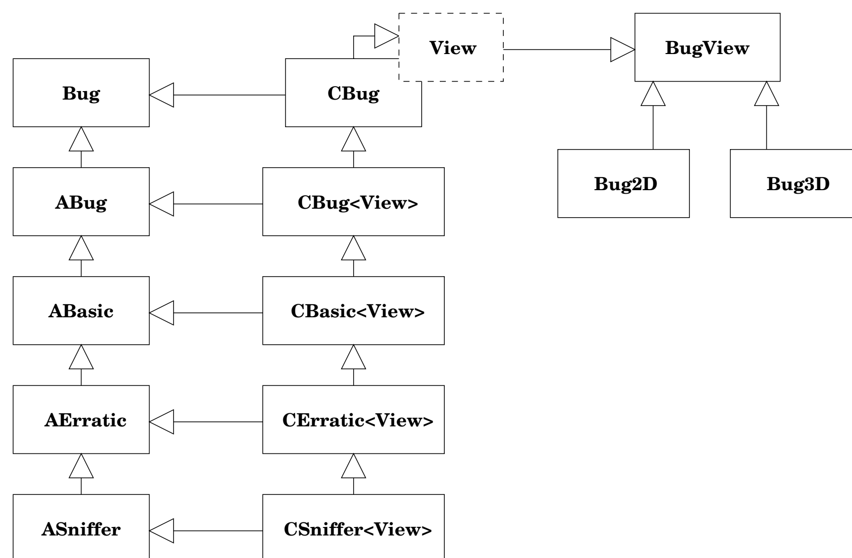

Figure 9. Multiple inheritance for an existing hierarchy

Here, for example the `CErratic` class inherits from both the `Erratic` class and the `CBasic` class, so the only methods to be written within this class are one constructor and the destructor. We also use C++ template inheritance to share the same control components for the 2D and the 3D applications: a control component will inherit from a particular visualisation class. The effective instantiation is decided within the corresponding subclasses of our `Factory` class : `CFactory2D` and `CFactory3D`. It would be possible to use delegation instead of inheritance, but inheritance is more appropriate if we want to redefine some methods of the visualisation classes within the control classes in order to take into account the end-user interactions.

The main drawback of this multiple inheritance comes from diamond inheritance, so we have to use C++ virtual inheritance, which is sometimes cumbersome.

Another solution is to use C++ template inheritance to create only one control class for all the bugs, as shown figure 10. The added parameter of this class is the bug type, and this class inherits also from this parameter.

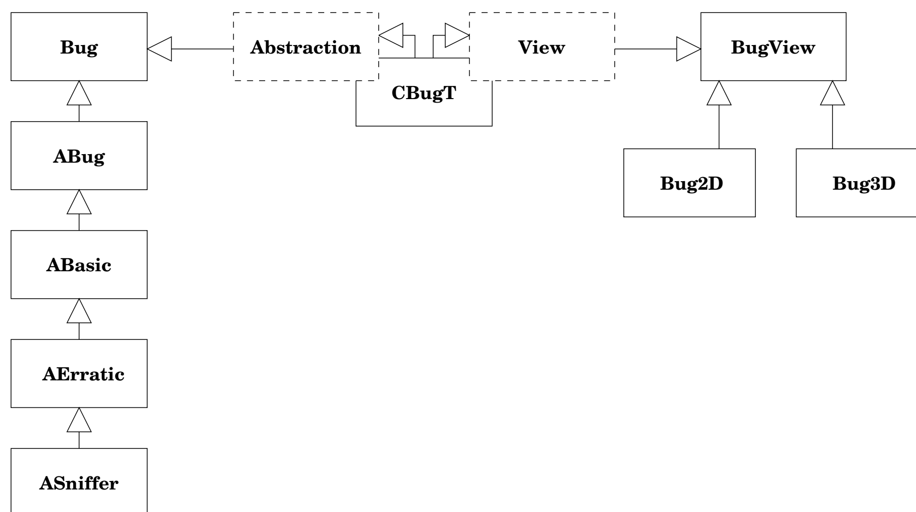

Figure 10. Template inheritance solution to control an existing hierarchy

This solution is particularly powerful as it enables to create only one template control class which will here be instantiated in six different ways in each of the associated `CFactory2D` or `CFactory3D` classes.

7: Conclusion

We have shown here that it is possible to transform efficiently an existing simulation application into an interactive one. This transformation has been realized using the PAC-Amodeus model by combining two design patterns: the creational ‘Abstract Factory’ pattern and the structural ‘Proxy’ pattern.

The main interest of our method is that the initial simulation application is absolutely not modified by our transformation process. In fact, this initial application has been written as an independent Java package, named `Application`, which is used by a `Control` package, also using a `Presentation` package. There are even two more packages: `GenericControl` and `Applet`, both use some features of the `Application`, `Control` and `Presentation` packages.

The main limitation of our approach is about the transformations needed to be repeated in case of inheritance of some classes of the initial simulation, because of Java simple inheritance. This problem does not arise with C++ or Eiffel, because they offer multiple inheritance and they are able to manage diamond inheritance. C++ even offers template inheritance, which reduces greatly the code to be written in our case of bugs inheritance. So, with Java, in the case of our example, it would be possible to use the 'Strategy' pattern (Gof315) in order to decouple the bugs strategies from the visualisation problems: the inheritance would then only concern the behaviour, this would not have to be adapted for the interactive simulation. But we are not sure that this pattern could always resolve that kind of problem with different simulation applications.

Another possibility with Java would be to be able to declare dynamically that a class can inherit from one subclass of a particular class: according to the name of this class, a controller could then declare that it inherits from it, and then its code would be as light as the `CBasic` one, and as powerful as the `CGenericBug` one, as it can be with C++ template inheritance. As template inheritance modelling is possible with UML, such a solution could also be offered using an UML case tool with appropriate code generation.

References

[1] Coutaz J. *Interfaces homme-ordinateur, conception et réalisation*. Dunod informatique, 1990.

[2] Coutaz J., Nigay L., and Salber D. Agent-based architecture modelling for interactive systems. Critical Issues in User Interface Engineering, P. Palanque & D. Benyon Eds., Springer-Verlag: London Publ., ISBN 3-540-19964-0, pages 191-209., 1995.

[3] Cremer M., Demir C., Donikian S., Espie S., and McDonald M. Investigating the impact of aicc concepts on traffic flow quality. In *5th World Congress on Intelligent Transport Systems*, Séoul, Corée du sud, octobre 1998.

[4] The UIMS Workshop Tool Developers. Arch : A metamodel for the runtime architecture of an interactive system. SIGCHI Bulletin, 24, 1, pp. 32-37, 1992.

[5] Donikian S., Chauffaut A., Duval T., and Kulpa R. Gasp: from modular programming to distributed execution. In *Computer Animation'98, IEEE, Philadelphie, USA*, juin 1998.

[6] Duval T. and Nigay L. Implémentation d'une application de simulation selon le modèle pac-amodeus. In *IHM'99*, pages 86–93, 1999.

[7] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns : Elements of reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] Goldberg A. Information models, views and controllers. *Dr Dobb's Journal*, july 1990.

[9] Jézéquel J.M., Train M., and Mingins C. *Design Patterns and Contracts*. Addison-Wesley, October 1999.

[10] Nigay L. Conception et modélisation logicielle des systèmes interactifs : application aux interfaces multimodales. Thèse de Doctorat de l'Université de Grenoble 1, IMAG, 1994.

[11] Reignier P., Harrouet F., Morvan S., Tisseau J., and Duval T. Arévi : A virtual reality multiagent platform. In *Proceedings of the First International Conference on Virtual Worlds (VW'98), Paris, Lecture Notes in Computer Science, Artifial Intelligence series (LNCS/AI 1434)*, juillet 1998.