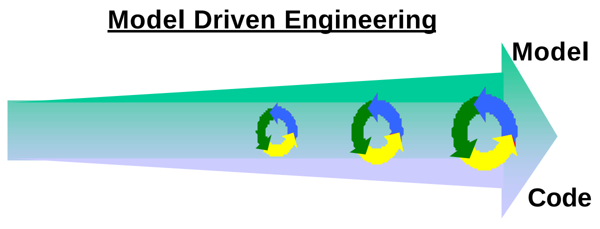
Guarantee permanent Model/Code consistency:

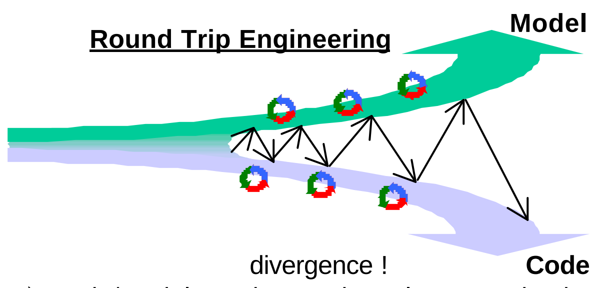
"Model driven Engineering" (MDE) versus "Roundtrip engineering" (RTE)

White Paper

ó SOFTEAM 2000



convergence.
➡ code/model consistency is permanently maintained



divergence !
➡ code/model consistency is no longer maintained!

Presentation

The model is the representation of the application which is applied throughout the development process. The code is only a complement to the model, three-quarters of which is generated, whilst the remaining quarter is inserted into the model itself. *MDE* (Model Driven Engineering or Model Driven Development) is the only approach which guarantees total code/model consistency throughout modifications and iterations of the development process. Everything is carried out at model level and the code is only a projection of this model. Moreover, model/code translation rules can be totally adapted so as to obtain code that always complies with the specific quality requirements of any project.
The *Round Trip Engineering* (RTE) provides a partial and unsatisfactory solution which, due to its successive iterations of code modification and reverse-engineering, very quickly breaks the model/code link, and requires a large amount of manual work to obtain consistency on the model and on the code separately.

With the MDE approach, the model thus covers all development from analysis to coding, and provides a common vision of the application which is shared by all participants. CASE tools guarantee model consistency, handle group work and automatically produce documentation, code, database diagrams and all development products.

The association of model and CASE tool thus provides all the necessary means to master software development at model level. Automation provided by the tool must, however, guarantee that overall consistency is maintained and that all development is monitored. It must ensure that the different parts of a model, the various activities of development teams and the different parts automatically generated are constantly updated and standard.

It is on this fundamental aspect that the MDE approach proves its superiority to the so-called "Round Trip Engineering" techniques.

The Objecteering CASE tool guarantees these automation and consistency maintenance characteristics, thus providing full handling of the MDE. The MDE allows you to take full advantage of the model input and to maximize development automation, thus increasing its overall quality.

After a brief description of the advantages of MDE on a complete development cycle, this "white paper" gives a more detailed technical description of internal technologies and services provided.

Advantages of "Model Driven Engineering" (MDE)

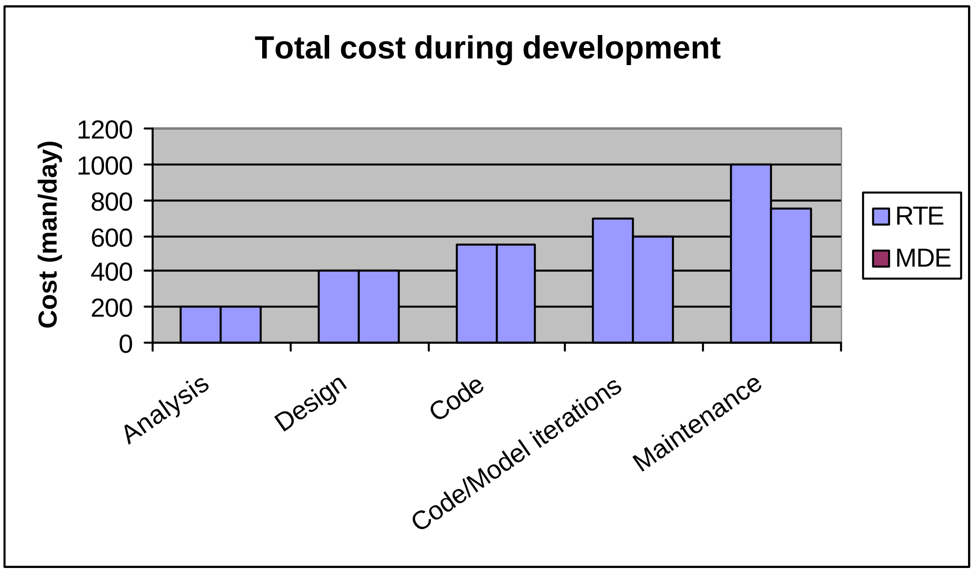


Figure 1 – *Total development costs*

Figure 1 compares the costs of development carried out using an "MDE" type approach or an "RTE" type approach, on the basis of an average size project with 700 man/days (to which a year's maintenance is added), and of average complexity.

In the initial stages, the advantages are not clear. However, as soon as the maintenance of model/code consistency is required (design/code iterations or incremental development, for example), the MDE approach shows a significant saving in cost.

In fact, once model/code consistency is broken when using the RTE approach, developers essentially work at code level. They no longer benefit from the UML model nor from CASE tool capacities.

With the MDE approach, the project, at the end of development, is delivered with a model, documentation and updated code. This facilitates maintenance, where work is always carried out at UML model level.

The MDE approach saves around 30% on productivity compared to Round Trip Engineering on such a project. This saving increases according to the size of the project.

With the MDE approach, all software development can be carried out from an application model. From the initial analysis to the final coding, the model constitutes the reference model, of which each participant has a different view, depending on the level of detail he is interested in. With the help of an appropriate CASE tool, the model ensures four levels of simultaneous consistency:

- A validity check of entered information to avoid building incorrect models
- Internal consistency management between parts of the model and different users, so as to reflect element changes, according to all views and on all dependent parts
- "External" consistency management, so as to guarantee that development work products generated (code, documentation, data models) always conform to the model
- Traceability on model development and on development work products generated during the life cycle

In this way, from initial analysis to maintenance, the model is the main tool representing an application. This advantage is further enhanced by UML, which is the standard model, understood and shared by everyone.

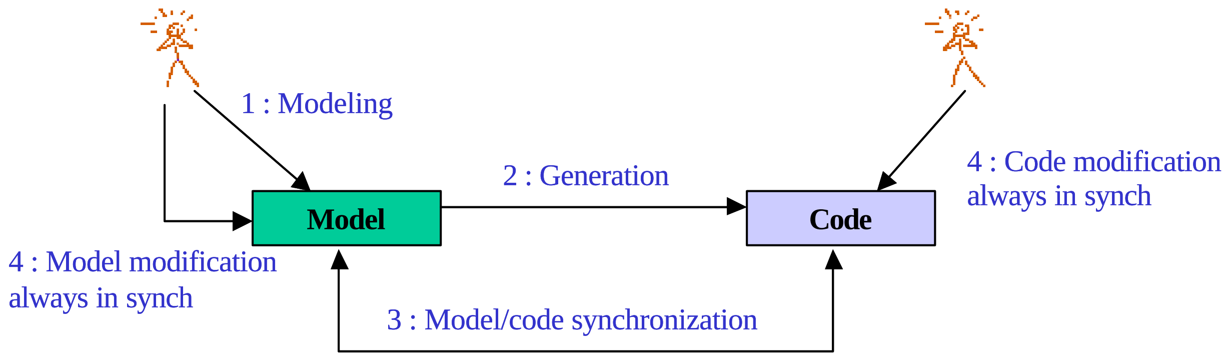

1 : Modeling
2 : Generation
Model
Code
4 : Code modification always in synch
4 : Model modification always in synch
3 : Model/code synchronization

Figure 2 – *The MDE saves the work done on the model and code and maintains model/code consistency*

The MDE approach consists of guaranteeing permanent model/code consistency, whilst with the Round Trip Engineering approach, the corresponding model is put together according to the code.

When in the modeling phase or in simple cases of code generation, the RTE approach might appear to be similar to the MDE approach. However, as soon as iterations between the model and the code appear, and when specification or design increments are introduced, then the benefits of the MDE approach are obvious. (cf Figure 2).
In fact, the MDE approach guarantees permanent model/code consistency. Modifications made at analysis or design model level will be, on the one hand, carried out on a model which corresponds exactly to what is implemented in the code and, on the other hand, will cause an automatic code update.
Even the most sophisticated RTE technologies cannot guarantee the maintenance of such consistency. This means that code on which a large amount of updating and testing has been carried out will never be regenerated from a model, since RTE would cause too many changes in the code to be able to preserve all the investment in terms of tests and development.

Table 1 – *MDE/RTE: comparison of effects during model/code synchronization*

		Model Driven Engineering (MDE)	***Round Trip Engineering (RTE)***
Model modification	Code update	Operation automatically guaranteed by generation	Semi-automatic operation: manual updates or checks are necessary
	Result	The code is proven standard to the model, compatible to the former state but introducing modifications carried out on the model.	The model is disconnected from the code or a new code is created on which update operations and previous tests must be redone.
Code modification	Model update	The model is kept standard to the code. The unchanged model elements are preserved.	By "reverse engineering", a "physical image " model of the code is reconstituted.
	Result	The model represents a faithful vision of the code, but remains on the level of the initial representation	A physical model is introduced into the initial model. This model is different from previous representations

Technically speaking, the two approaches are very different. Round Trip Engineering is essentially based on code reverse engineering techniques, whilst MDE uniquely identifies each and every model element and keeps permanent traceability of this, element even when code is generated.
This traceability is used to preserve the semantics of a defined element at model level and to separate those parts of code manually entered by the programmer from those parts of code which result from generation modeling. Difficult modifications (such as the re-naming, creation or destruction of elements), which generally show up the failings of the RTE techniques, can also be handled.

MDE implies a realization approach: it obliges everything linked to the model (for example, the definition of an association between two classes) to be carried out at this level, and everything which is related to code definition (for example, coding a call to a Java library) to be done in code typed model extensions in Java. MDE also guarantees that your programming rules (for example, translation mode for a code association) will be systematically applied to the whole application. Our clients, some of whom have been using this technique for 10 years, have had the best code quality and certification results by using this technique. In fact, both quality in terms of traceability and the application of programming rules are automatically ensured. These factors result in a high level of maintenability of applications.

"Model Driven Engineering" techniques are more sophisticated than "Round Trip Engineering" techniques. This is why few editors provide true coverage of this approach. Due to the quality of its technological choices, SOFTEAM has been able to a develop its Model Driven Engineering type CASE tool approach, with OBJECTEERING.

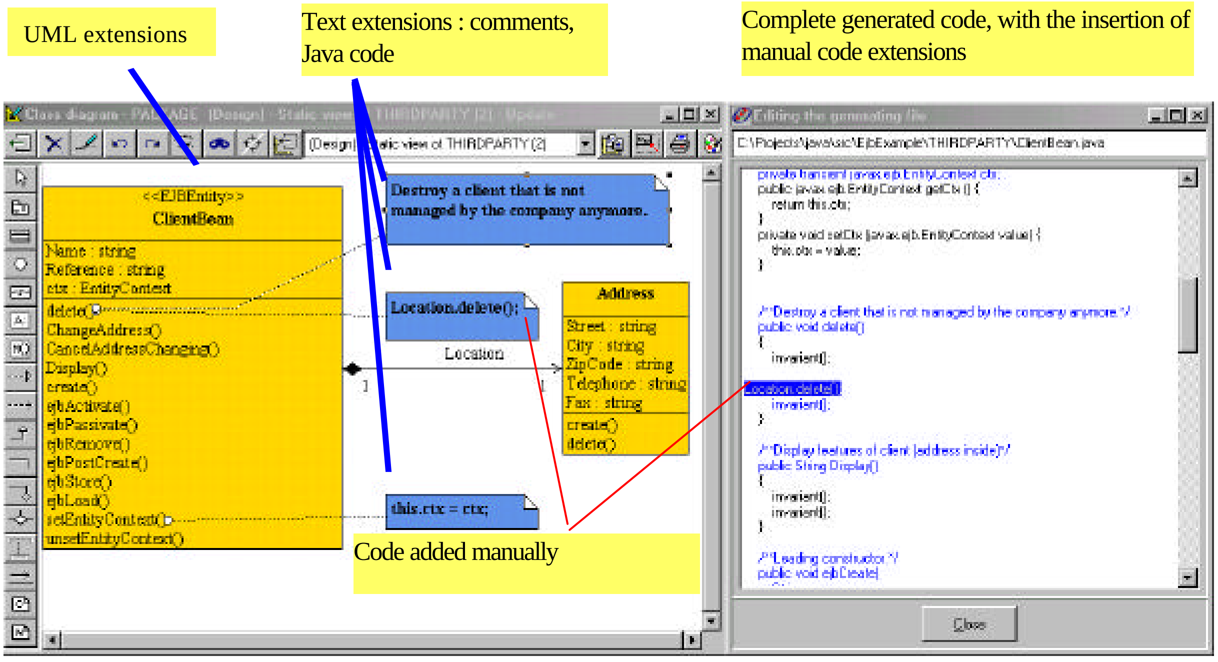
The five functions handling "Model Driven Engineering"

The Objecteering CASE tool handles MDE through the five basic services presented below. These services ensure permanent consistency for the model and all the development work products associated with it.

1: Universal identification

In this way, throughout the lifecycle of the element in question and the modifications made to it (modification, exchange, code generation, etc.) it will always have the same identity. For example, if a developer creates a class attribute on a company's Objecteering site, this attribute will be identified in a unique and definitive way. If this attribute is exchanged between different sites, and returns at a later stage to the initial model, then Objecteering will recognize the identity of the attribute and will update it, whatever changes may have been made to it in the interim.
Using this identifier, it is possible to trace a model element through all the development work products (code, documentation, etc.) in which it appears.

2: Storing code complements in the UML CASE tool repository



```
private transient javax.ejb.EntityContext ctx;
public javax.ejb.EntityContext getCtx () {
   return this.ctx;
}
private void setCtx (javax.ejb.EntityContext value) {
   this.ctx = value;
}


/**Destroy a client that is not managed by the company anymore.*/
public void delete()
{
   invariant();
Location.delete()
   invariant();
}

/**Display features of client (address inside)*/
public String Display()
{
   invariant();
   invariant();
}

/**Loading constructor.*/
public void ejbCreate()
```

Figure 3 –*Code is seen as an extension of the model*

Code extensions which are added to the mode, are stored in the Objecteering repository. They appear in the explorer or in Objecteering/UML Modeler editors as specialized notes ("C++", "Java", etc.). Objecteering code generation thus produces, for example, 70% of the final code deduced from the model, and transfers 30% of the code manually entered but already stored in the Objecteering repository. (These proportions were measured by one of our clients on his project. Certain users exceed the proportion of 80% code automatically generated).
For example, if the sources are destroyed or lost, Objecteering regenerates the whole application. With this capacity, design patterns, which produce code in the repository, can also be automated. In this way, the proportion of code deduced can easily be increased.

3: Permanent management of model/code liaison

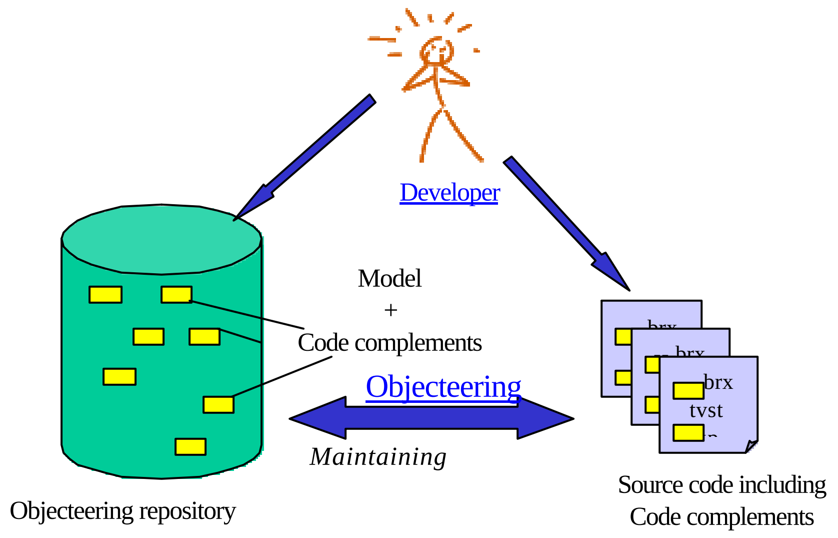


Figure 4 – *The developer can intervene either on the model or on the code*

Objecteering provides a permanent maintenance mechanism for consistency between the model and the generated code. In this way, users can modify either the generated code, using editors external to Objecteering (such as the C++ or Java development environments), or the model in the Objecteering CASE tool. Objecteering detects differences, and synchronizes the repository and the code. It is principally based on the *universal identification* mechanism, so as to transfer external modifications to the part of the model in question.

4: Capacity for total parameterization of "development work product" generation from the model

Presented in the White Paper on UML profiles, (you can download it at www.objecteering.com) Objecteering/UML profile builder is a tool which has total Objecteering generator parameterization and adaptation abilities. Certain users with specific requirements regarding generated code forms, or with programming quality rules, will adapt Objecteering generators, in order to obtain code which corresponds exactly to their needs. The user can also define the generation target libraries of their choice, so as to parameterize basic type generation or lists, containers, sets, etc at the lowest level.
Once this has been done, the development team has a total guarantee with regard to respecting coding forms, as well as code/model conformity. Quality audits carried out on Objecteering users' work by external auditors mentioned the exceptional quality of the code produced by the CASE tool.

5: Permanent consistency checks on models entered

In order to obtain correct code, the model must be accurately entered. Model quality and consistency are fundamental if high-quality code is to be generated. Objecteering carries out more than 300 consistency checks in real time, to guide users and guarantee high-level model quality. For example, Objecteering checks accessibility between packages, avoids mutual dependencies and helps organize models correctly.
Objecteering code generation is, therefore, based on a sound model, in order to produce correct, well-structured and easily maintainable code.

"Roundtrip Engineering": an unsatisfactory solution

Many UML CASE tools provide a "roundtrip engineering" type solution. This consists of generating code (Java, C++, etc.) based on a model before subsequently completing code sources manually. Modified code can then be analyzed by the CASE tool, in order to reconstruct as best as possible a model which corresponds to the code in question.
Let us take as an example the "design" model shown in Figure 5 and detail the "Commercial" class. Java generation on the "Commercial" class of such a model (example of generation carried out by Objecteering) will map an oriented association to "accessor" methods, to attributes which implement links and to the "import" of associated classes, etc..

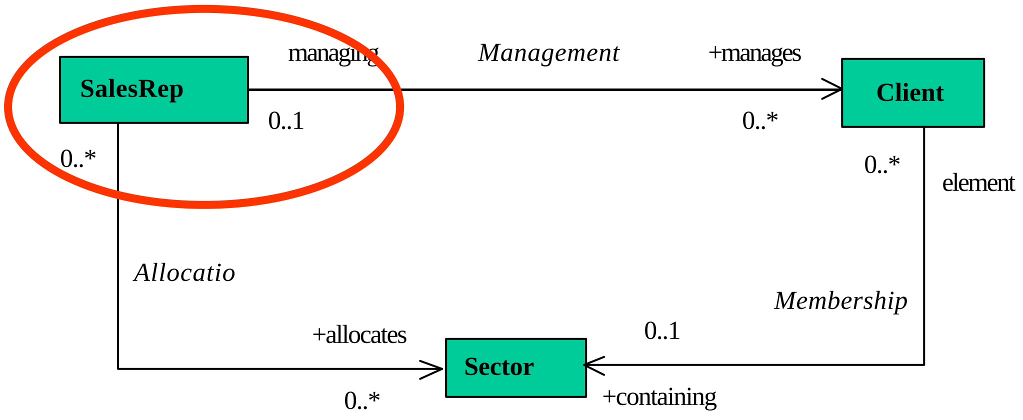


Figure 5 – *Example of a design model*

Java code in the following form is then obtained:

```
import java.util.Vector;
public class SalesRep {
     public Client getManages(int i) {...} //Management -> manages Client
     public Vector getManages() {...}
     public void setManages(int i, Client newClient) {...}
     public void setManages(Vector newClients) {...}
     public Secteur getAllocates(int i) {...}// -> allocates Sector
     public Vector getAllocates() {...}
     public void setAllocates(int i, Sector newSector) {...}
     public void setAllocates(Vector newSectors) {...}
     private Vector manages = new Vector();
     private Vector allocates = new Vector();         }
```

If this code is reversed in UML, a model like that shown in Figure 6 (i.e., a physical, Java-oriented model) is obtained.

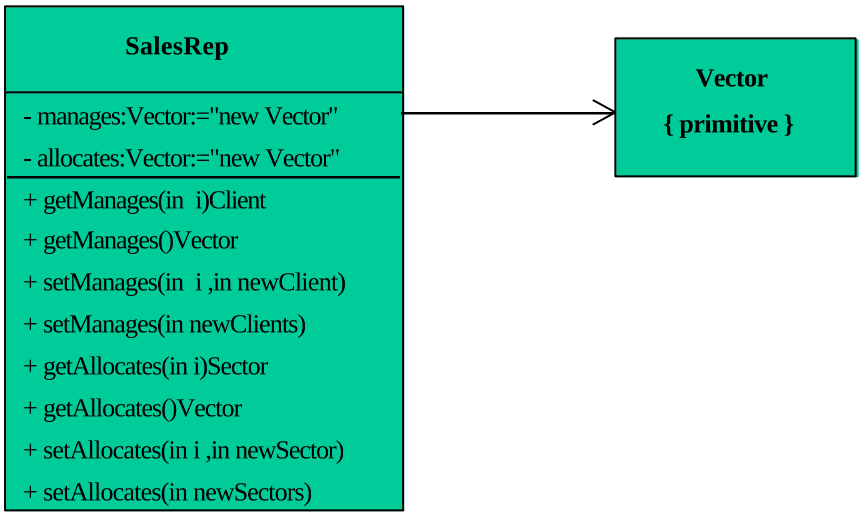

SalesRep
- manages:Vector:="new Vector"
- allocates:Vector:="new Vector"
+ getManages(in i)Client
+ getManages()Vector
+ setManages(in i ,in newClient)
+ setManages(in newClients)
+ getAllocates(in i)Sector
+ getAllocates()Vector
+ setAllocates(in i ,in newSector)
+ setAllocates(in newSectors)
Vector
{ primitive }

Figure 6 – *Result obtained by reversing Java generation of the "SalesRep" class*

As illustrated above, the initial conceptual or logical model has been lost. The model's value has in fact been lost : the model has become a mere graphic representation of the code, and has lost its high level abstraction resulting from software analysis or design.Code/model and model/code transformations are not symmetrical.
The problem becomes worse still if, for example, other modifications have been made, such as class name changes. If this is the case, the Java reverse does not recognize that the same class is concerned and builds a model which contains both the former class and the new class.
Even the most sophisticated tools cannot entirely synchronize themselves with generated code through this process. They cannot, for example, distinguish between the renaming of an element and the destruction followed by the creation of a new element, if these operations are carried out in the source code.
Those people who use this process do not update their model with regard to the code, since once the code has been produced, tested and validated, they refuse to reverse the code in order to generate new and different code, which would then necessitate the re-testing and re-validating of the entire code.
A study carried out amongst CASE tool users in the United States showed that in 1998 very few users obtained, at the end of the development phase, a model consistent with the code.

How to make the most of the "Model Driven Engineering" approach

Permanent model/code consistency

Using the MDE approach, the model is always consistent with the code. The model maintains its level as defined by the designer, but corresponds exactly with what has been automatically translated into code. It provides high level views, which can, for example, be used to visualize the general structure of a large-scale application, or indeed to understand an application during its maintenance phase. Dependency links between packages, for example, represent an exact synthesis of the dependencies which exist between the classes of the corresponding code.
With this approach, all development work products are well synchronized, and consistency is maintained between the central model, generated documentation and code.

Develop in your favorite environment

According to the development stage and the environments used, the developer will either use the UML Objecteering CASE tool or the programming environment. The CASE tool maintains consistency between sources and the model, whilst retaining an up-to-date repository with regard to sources. The model is thus a guide and not a constraint, in cases such as typically coding or fine-tuning code, where the user can often prefer to use his own environment.

Use the power of model operations

A modification at model level can lead to the update or generation of hundreds of lines of code. The UML CASE tool is considered a highly efficient editor of the application itself. For a user, manually modifying parts of code automatically generated by the CASE tool would be the equivalent of "patching" assembling code generated by a compiler. We present here two example of simple model handling, translated by the subsequent update of the generated code.

.1 Example 1: drag & drop on an association between two classes

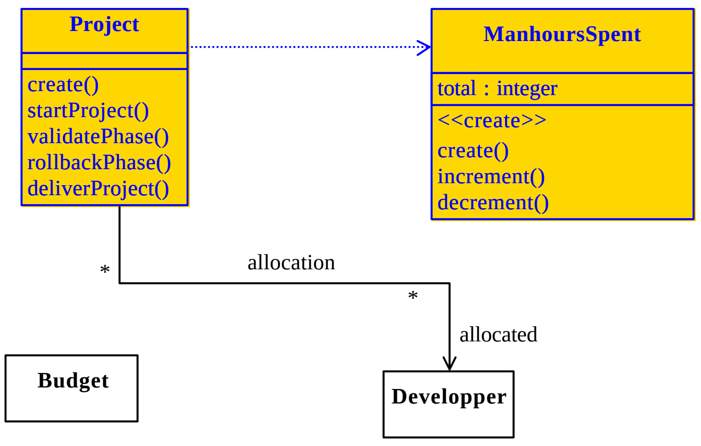


Figure 7 – *Initial model in Objecteering*

Let's take the model shown in Figure 7. This model corresponds to the following code for the "Project" class, regarding the "allocation" association. It is made up of "accessor" Java methods, of attributes which implement the association and of import declarations on the necessary classes.

Code relative to the "allocation" association in the "Project" class :

```
….
public class Project
{
   Vector allocated = new Vector();
   public ProjectManagement.project.Developper getAllocated (int i) {
      return (ProjectManagement.project.Developper)this.allocated.elementAt(i);
   }
   void setAllocated (int i, ProjectManagement.project.Developper value) {
      this.allocated.setElementAt(value, i);
   }
   void appendAllocated (ProjectManagement.project.Developper value) {
      this.allocated.addElement(value);
   }
   void eraseAllocated (ProjectManagement.project.Developper value) {
      this.allocated.removeElement(value);
   }
   void eraseAllocated (int i) {
      this.allocated.removeElementAt(i);
   }
   public int cardAllocated () {
      return this.allocated.size();
   }
….
```

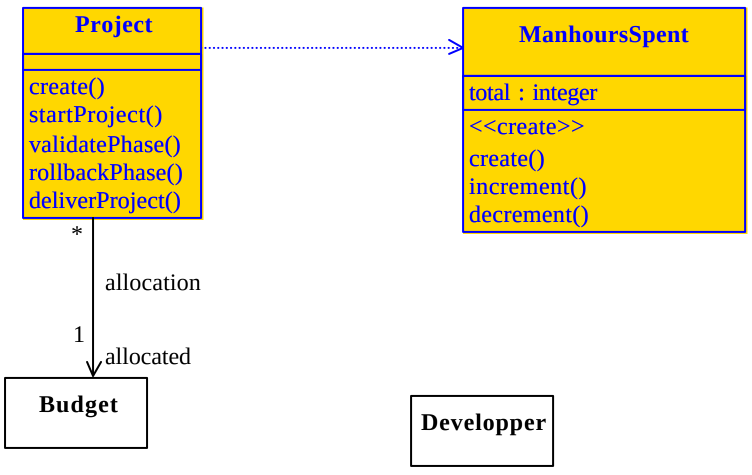


Figure 8 – *Modified model*

Let us now carry out in Objecteering a drag and drop of the end of this association towards another class ("Budget") and then change the multiplicity (Figure 8).

Modified code after this operation has been carried out :

```
…
public class Project
{
   ProjectManagement.project.Budget allocated;
   public ProjectManagement.project.Budget getAllocated () {
      return this.allocated;
   }
   void setAllocated (ProjectManagement.project.Budget value) {
        this.allocated = value;
    }
    public int cardAllocated () {
        if ( this.allocated == null ) return 0;
        else return 1;
    }
…
```

This simple operation on the model caused around twenty lines of Java code to be modified, thereby producing new code, guaranteed as being correct and up-to-date with regard to the model on which it is based.

Example 2: changing the name of a base class

Let's imagine that we have defined a "date" class designated as being "primitive", and massively used to type the attributes and parameters of other classes in a large-scale application. This "date" class could, therefore, find itself referenced hundreds of times throughout the application. Objecteering manages these dependency links and constantly maintains consistency. If we then change the name of the "date" class to "Date", Objecteering will update all references in the model, and all the graphic editors will reflect the name change. Once Java code has been regenerated, it will be made consistent with all the code produced.

Model quality greatly enhanced

The model is and always remains an exact image of the code during software development. In this way, all services produced at model level provide advantages at the level of generated code. All work carried out on the model, all improvements made to its quality, will have a positive impact on the code. This approach adds to the quality of the model. We provide here two examples of services which benefit the quality and overall productivity of software development, but it should be pointed out that any number of similar examples exist.

Example 1: Metrics

Objecteering provides a metrics service on the model. This service is used to provide model quality measurements, such as indications of complexity, modularity and the independence of different model parts. For example, a model can be reworked in the earliest stages of development in order to improve its indicators, or afterwards during maintenance phases in order to pinpoint problem areas in an application. In the first case, this work produces well-structured, better quality code. In the second case, metrics provide a good overview of the implemented application.

Example 2: Automating design patterns

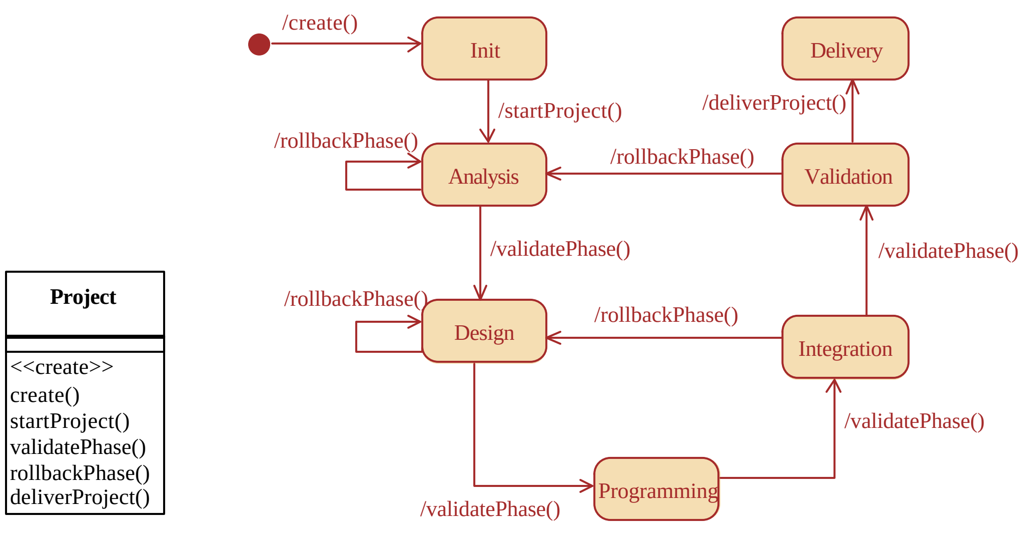
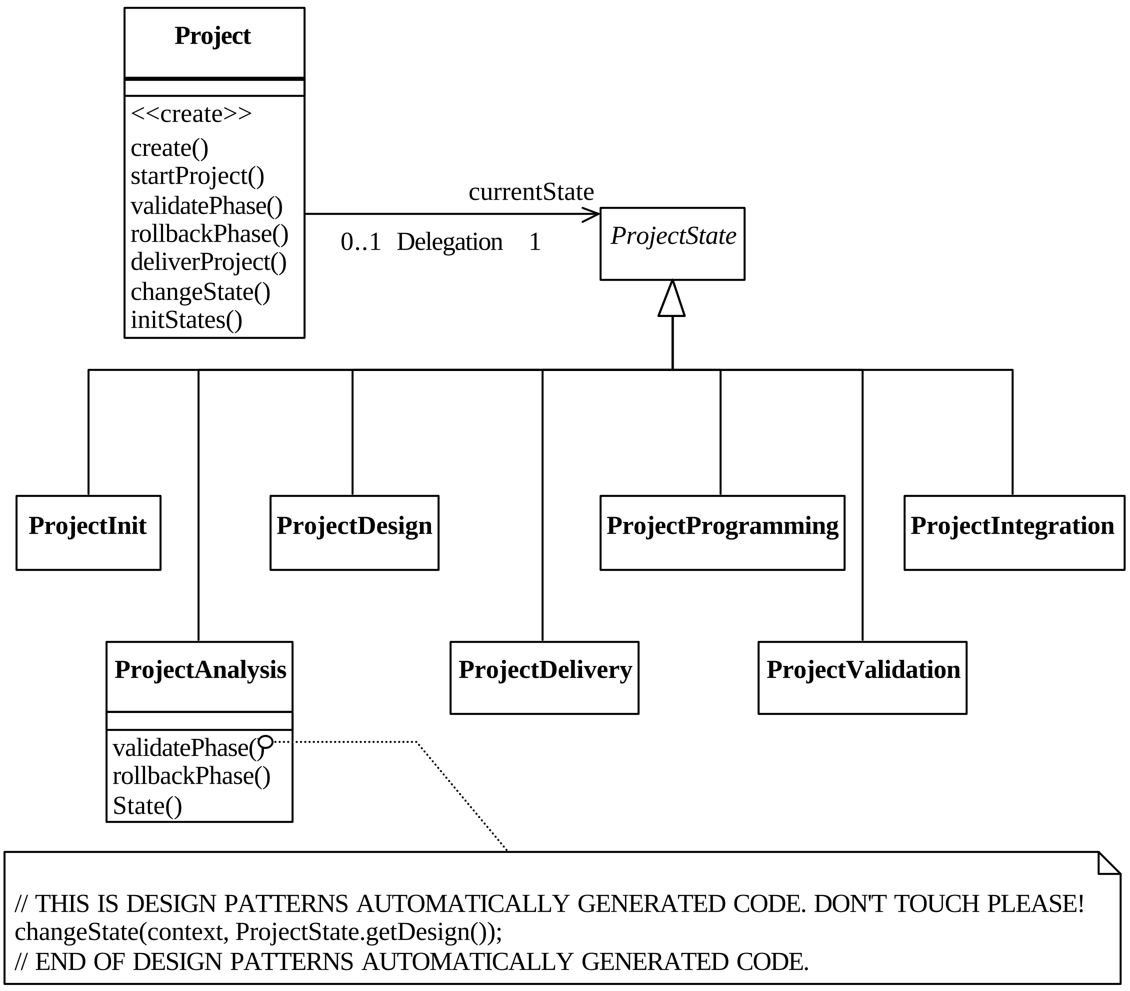


Figure 9 – *Initial model: a class and its state machine*

Objecteering provides services to automate the most commonly used design patterns. The Objecteering/UML Profile Builder tool can be used to automate other design patterns or parameterize those patterns which have already been automated. Figure 9 shows an initial model, while Figure 10 illustrates the transformed model after the application of the "state" pattern.



```
// THIS IS DESIGN PATTERNS AUTOMATICALLY GENERATED CODE. DON'T TOUCH PLEASE!
changeState(context, ProjectState.getDesign());
// END OF DESIGN PATTERNS AUTOMATICALLY GENERATED CODE.
```

Figure 10 – *Applying an automated design pattern to the model*

The fact that the CASE tool stores code complements in its repository means that automated design patterns will not only transform the model, in order to create a technical design model close to implementation, but will also insert code in the typed notes associated with method, to provide a model, whose code generation is virtually complete (Figure 10). Thus we can see in Figure 10 that the "Project" class in Figure 9 has been improved by a set of new methods and by a graph of classes dealing with each possible state of the "Project" class. This is due to the application of the "state pattern". Using this technique coupled with the code generation, more than 70% of the code is generated, sometimes reaching almost 90%.

Based on an initial model, the combination of design patterns and automated generation thus greatly increases productivity, whilst providing ready-made design solutions, which guarantee well-constructed software.

Conclusion

With the MDE approach, the model, used to represent an application, also becomes a development tool. Every intervention at the highest level of the model has an immediate impact on the code, thus immediately providing a correct application with up-to-date documentation. Maintenability, quality, robustness and speed of development are just some of the advantages of this approach.

Compared with Round Trip Engineering, the Model Driven Engineering approach increases productivity by more than 30%, provides greater flexibility during iteration or incremental development phases and guarantees that development quality rules are closely followed.