Organizing a large model-driven software development project

Case study

Philippe Desfray –Modeliosoft

Philippe Vlaemynck - Modeliosoft

www.modeliosoft.com

Abstract

Activities such as configuration and version management, integration, validation and team organization and cooperation are all too often neglected by development teams, but remain crucial if a project is to succeed. This white paper is a reminder that numerous tools exist to help assist and automate these activities, notably in the world of open-source applications. An example illustrates how these tools can be used in conjunction with a modeling tool (Modelio), in order to obtain successful model-driven development.

Introduction

How can cooperative work by a large team on large-scale models be organized? This is a key question is essential if the advantages of modeling and model-driven development are to be fully exploited. Aside from the technical points linked to the features of both the modeling tool and the various other tools used, the question of organization is fundamental, and the entire development team must be on board if a good level of collective discipline is to be obtained.

Success can only be achieved if participants can clearly see the advantages:

- The developer must be able to see the gains, notably with regard to his/her productivity and simplified cooperation with his/her partners.
- The project must obtain a clearly measured and perceptible qualitative gain, visible to both project managers but also to higher levels of management.

This white paper addresses the most delicate aspects of large-scale project management:

- Teamwork
- Configuration management
- Version management
- An operational vision of model-driven development
- Software integration
- Validation

It presents an example of implementation, where the tools and approach put in place result in significant everyday gains for all development participants. The ROI (return on investment) in terms of efficiency and quality can be measured daily.

This example is based on the modeling, teamwork support, configuration and version management, model-driven development and customization features of the Modelio tool. It describes how different complementary tools (all open-source) are implemented in order to provide an efficient solution for a particular organization. With the focus firmly on the practical, no theory, general solutions or universal processes are provided. Each reader is free to use the information and ideas contained in this white paper to develop possible implementations in his/her own context.

The example presented does not provide an ideal situation, but rather a real one, including compromises and perfectible elements. It is implemented through a desire to achieve optimum gains with limited means. There is, therefore, no obligation to be "model-driven", or to automate all procedures. The focus is firmly on finding practical solutions with immediately perceptible gains. This approach is the result of numerous iterations, and takes into account remarks made, difficulties encountered and improvements suggested by all participants. To us, a good process is a process that is regularly used, generally accepted and whose gains are well understood by all concerned.

The conclusion of this white paper summarizes the gains made, and explains how to go further and put in place a solution that is specific to each person’s context.

The context

How many projects really apply a model-driven approach?

The answer to this question is far too few! There are several reasons for this:

- developers do not spontaneously adhere to the model-driven approach
- most modeling tools do not provide the services necessary to the real support of teamwork and configuration and version management
- too few invest in the set-up of an adapted organization and tools
- and so on...

Since its creation, Softeam has provided a tooled solution for model-driven development (Objecteering, from 1991), which has been constantly consolidated ever since. The Modelio tool is the result of our know-how, with tried and tested practical solutions now implemented by many of our clients.

In the example used in this white paper, we describe how the Modelio development team itself is organized, and detail which tools have been put in place to support this approach.

Several teams contribute to the development of Modelio, and development is split into several projects. Developments use different technologies according to the project in question (C++ for the heart of the tool, Java for the graphical user interface, and MDA projects for Modelio modules).

One person is dedicated to integration activities, and a separate team manages validation. Tool maintenance (for the current tool version) is carried out by a team separate from the development team, which works on the future version.

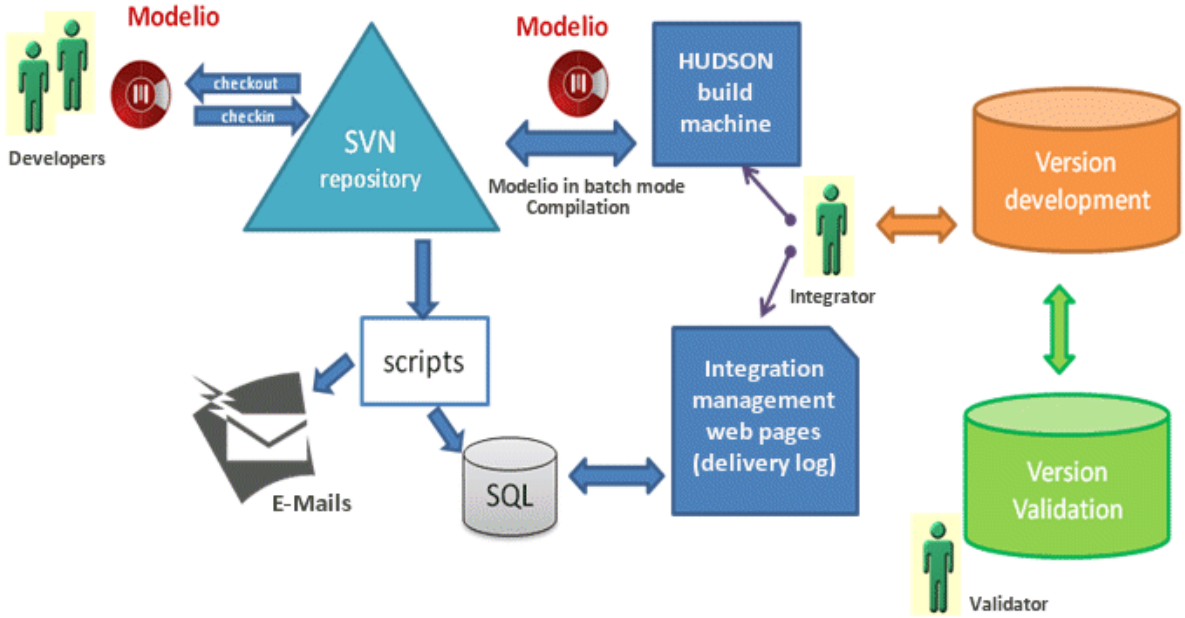

Figure 1 - *Organization of a project*

The tools used are as follows:

- The Modelio tool: Modeling, code generation (www.modeliosoft.com)
- The Modelio Java Designer, Modelio C++ Designer, Modelio MDA Designer and Modelio Teamwork Manager modules
- Modelio extensions dedicated to supporting the development process (specific Java modules): Integration with other tools, automation of procedures, automatic generation on our internal model management framework
- Eclipse: (www.eclipse.org) Open-source tool for Java developments
- Visual Studio: Microsoft tool for C++ developments
- Subversion: (www.subversion.tigris.org) Open-source tool for configuration and version management. This tool is used via the native Modelio coupling provided by the Teamwork Manager module.
- Mantis: Open-source bug management tool. Web application. (www.mantisbt.org)
- Testlink: Open-source test management tool (http://sourceforge.net/projects/testlink/)
- Hudson: Open-source automated application production tool. (http://hudson-ci.org/)

Complementary work was carried out to configure and couple these various tools.

Tooling, procedures and organization

One "toolkit" for all developers

Within a Modelio project, developments are closely linked and require that access conflicts on models and code be carefully managed. Participants must communicate and know what other people are doing, in order to work on their current tasks in the best conditions. On C++ or Java-type development projects, delivery cyles are short : several "commit" operations per day.

However, developments on "module" projects are not closely linked. Participants work autonomously on longer cycles, and the Subversion tool is used more to manage versions than to manage concurrent access.

C++ developments are separated into several projects, whereas Java developments are grouped into one single database. In our example, there are 2800 classes. The current performance of the Modelio Java generator enables the application to be quickly produced (1’40 pour les 2800 classes) and recompiled several times a day. Six people work on the Java project in multi-user mode with Modelio.

The same development tools are deployed on each workstation. This set of tools, known as the "toolkit", is versioned and managed by the integration manager under the supervision of the head of product development, who decides on updates for developer workstations.

Participants must not adapt their own individual environments (for example, with specific plugins or different versions). They are informed of tool updates via an RSS flow, and have at their disposal a script to update the toolkit when necessary.

This point is fundamental to the success of the whole approach. All the development environments are the same, meaning that developers can easily intervene in other people's developments. The integration procedures are also made significantly easier.

This necessity for discipline initially met with resistance on the part of the developers, each of whom enjoyed having his/her own specific tools. However once the toolkit had been tried and tested, the gains obtained quickly rallied participants to the approach.

Each version of Modelio is versioned with the toolkit that was used in its development.

Delivery/publication procedures

The development process as a whole is broken up into "projects" (in the Modelio sense of the term), which are homongeous cooperating entities. The project contains the model and code repository, and is subject to dedicated configuration and version management. "Model components" supported by the Modelio tool are used to manage deliveries between projects in a similar way to two classic development projects (code), where the first delivers what has been produced in the form of libraries to the second, which then uses them and updates them at its own rhythm.

Only the model is managed in configuration. With Modelio, the code is entirely deduced from the model and from code complements on operations. Everything is stored in the Modelio repository. Since code is rapidly deduced from the model, it is possible but pointless (and more complex) to manage code versions at the same time.

One SVN repository is created per project. This avoids any problems with size, simplifies branch and evolution management, enables independent life-cycles for each project, lets each project be configured individually (modules, versions, model components, …), and limits annoying situations of excessive "locks".

Version and configuration management happens at Class (or Actor, Use Case, …) level. With the Modelio tool, participants reserve parts on which they are working (check-out) and then deliver and free up these parts according to the progress they have made (commit, check-in).

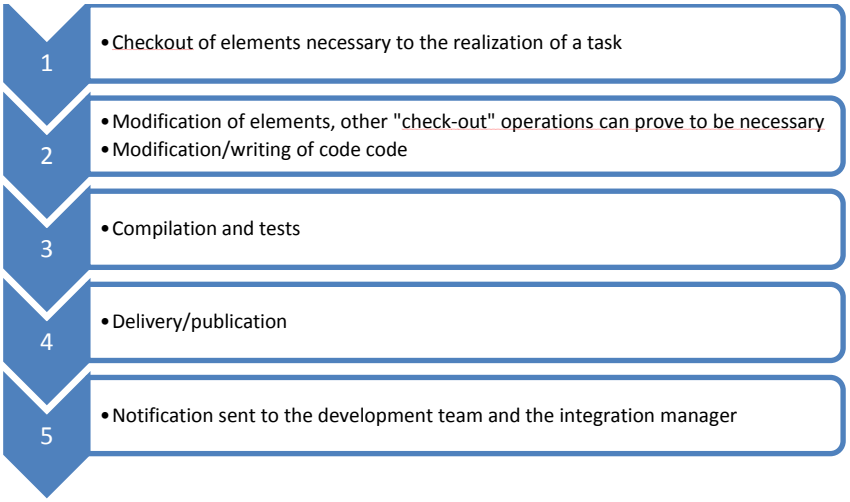

Figure 2 – *Steps necessary to the realization/modification of an application element*

Other than the Modelio Teamwork Manager and Subversion tools, cooperative work requires that participants be disciplined. When realizing or modifying part of the application designer/developer has the following responsibilities:

- To "checkout" the model elements he/she has to modify, while minimizing the number of locked elements.
- To modify these elements while respecting the methodological rules regarding the implementation of an approach which focuses on the model (model-driven) versus an approach which focuses on the code (round-trip). These approaches and rules are described in the dedicated chapter in this white paper.
- To "deliver" or "publish" his/her modifications in order to make them available to the entire team.
- To guarantee the "compilability" of his/her modifications before their "delivery" or "publication".
- To test his/her modifications before their "delivery" or "publication".
- To document his/her delivery (what has been modified, why, is there an impact on the validation process, and so on).

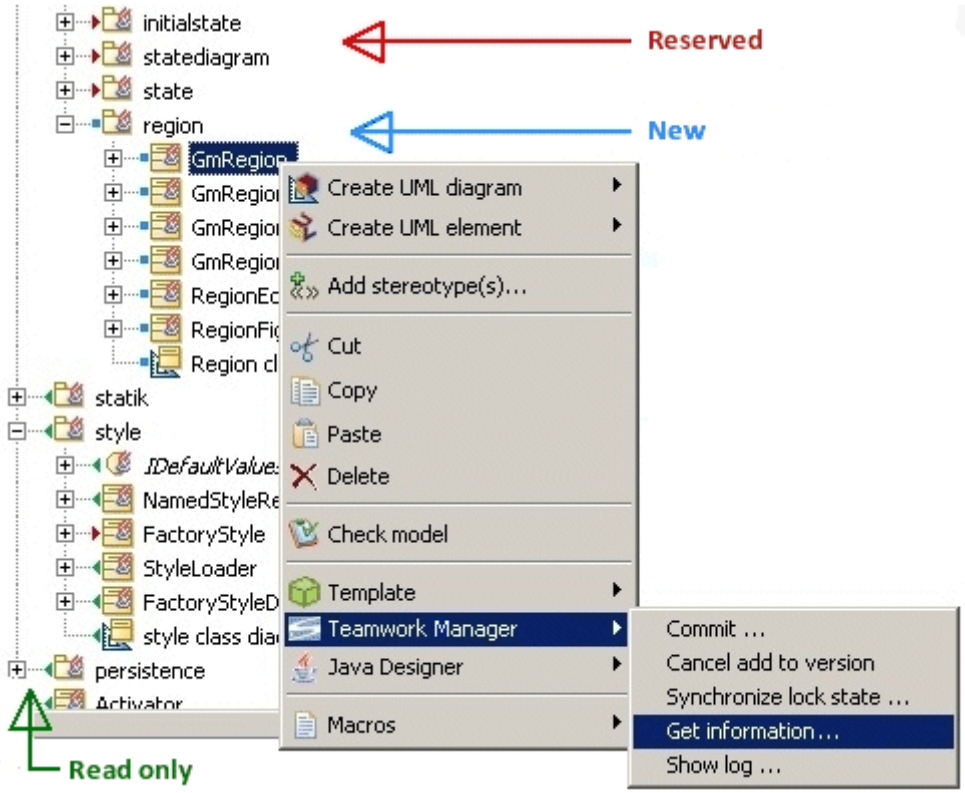

Figure 3 - *Modelio displays the state of model configuration and enables Check-in and Check-out operations in SVN*

When delivering or publishing modifications, a participant must describe what he/she has done. This task is made easier by the toolkit, which provides a dedicated form that must be completed for every " check-in" or "commit" of a modification (Figure 2). Concise information (for example, "Correction of bug 3765" or "Addition of a move() function on graphical objects") are useful for the entire team. Conversely, vague information (for example, "Modification of the GraphManager" class) is useless.

A participant must also indicate whether or not there is an impact on the validation process, in other words, how functional behavior has been modified (change of command, new menu, …).

The parties concerned (participating developers, the integration team, the validation team) are informed by email of the evolution, which they can then take into account.

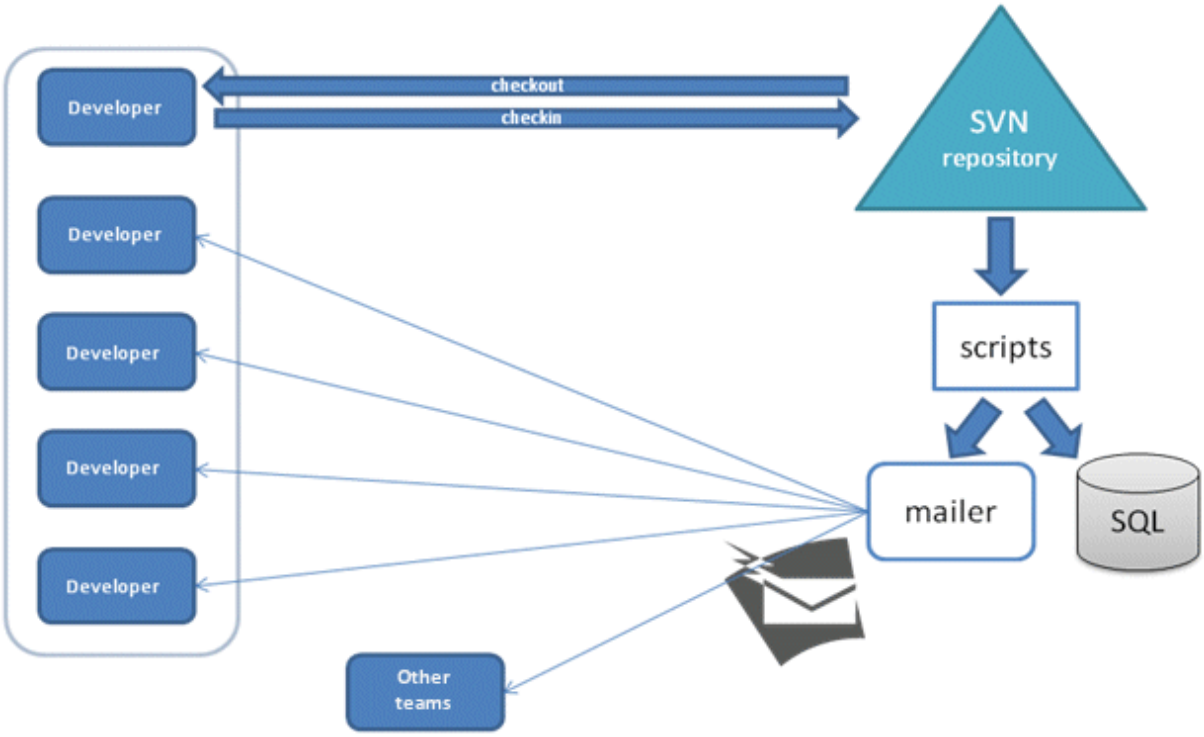

Figure 4 – *Process linked to the check-in/check-out of work*

In practice, for six simultaneous participants on a project of 2800 classes, the locking mechanism causes very few problems. The SVN "add" mechanism supported by Modelio Teamwork Manager enables the creation of new parts of models within the general model without using the lock mechanism, for as long as the participant has not delivered his initial work.

All the information entered in the form (Figure 5) is stored in a relational database (MySQL). This means that useful searches can subsequently be made on the history of work carried out, typically to find out who has modified what, why and which developments have taken place. This type of information is naturally present in the Subversion tool "logs", but the search scope is too limited.

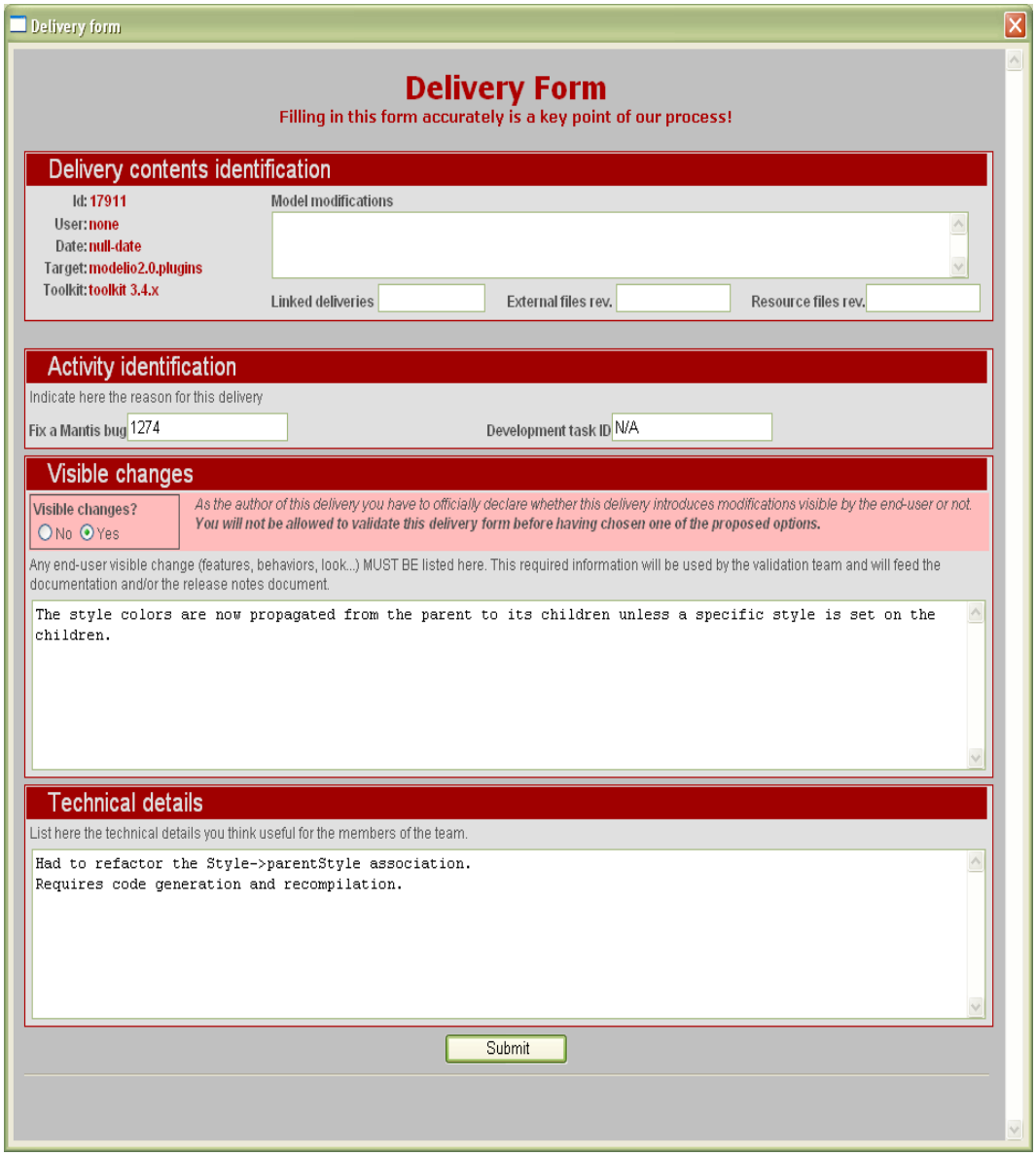

Figure 5 – Application code and model delivery form

Different workspaces to manage major development stages

When a bug is corrected, the developer must be able to access the bug management tool (Mantis), in order to update the status of the bug in question. Other concerned participants are also informed of the correction. This update could have been coupled to the completion of the form shown in Figure 5, at the cost of integrating other tools.

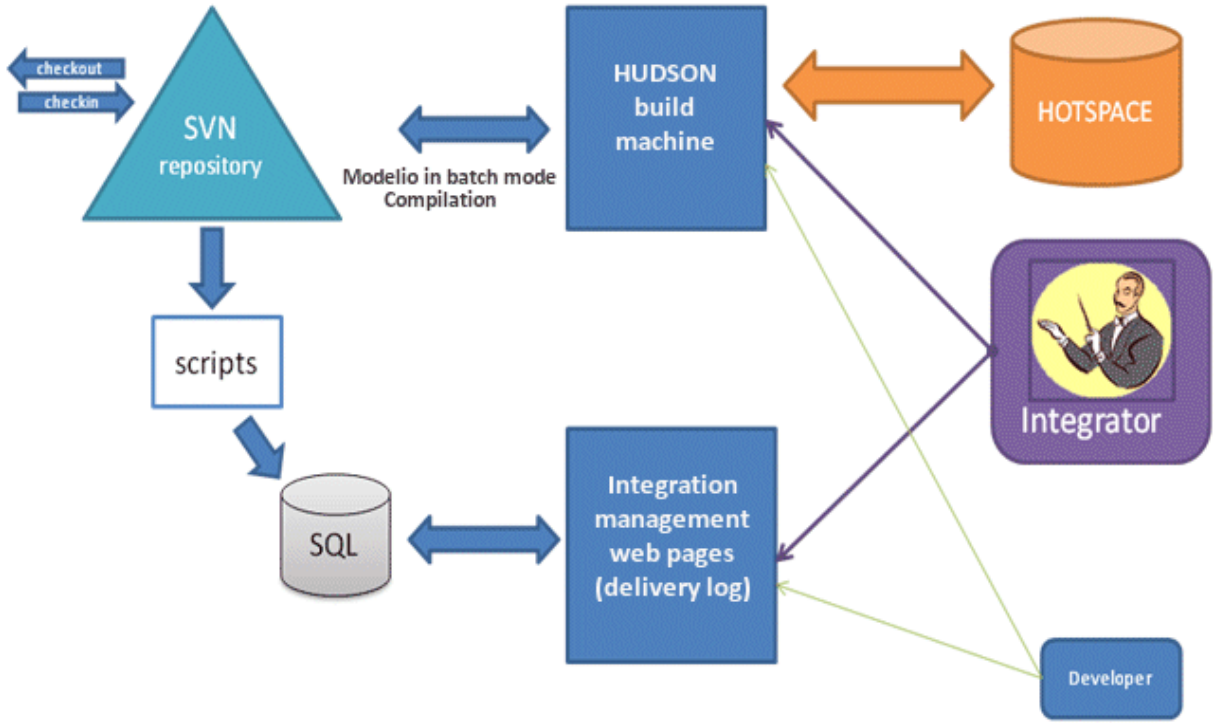

Figure 6 – *Processus d'intégration*

The integration manager plays a key orchestration role in the process, as he has an overall view of deliveries. Deliveries from designers and developers are automatically assembled using the Hudson tool, which permanently rebuilds the application in its most recent development/delivery state. Assembled application elements are placed in an area called the "hotspace". By definition, the application in the hotspace is unstable. It has no identified version number (since it is simply an ongoing development of the last stable version), and is not suitable for validation.

If a build of a new application fails, Hudson informs all participants who are potentially concerned, in other words, all those who have "committed" in SVN since the last build.

The integration manager supervises the permanent construction of applications, and can intervene where necessary, notably in collaboration with developers if the "build" is not correct.

He decides, with the approval of the head of product development, to switch applications to the delivery/validation area ("frozen space"). At this stage, deliveries are versioned (SVN tags are created) and provided to the validation team. The validation team knows what has to be validated in this new version thanks to Mantis and the delivery database.

The Modelio product development team uses Modelio to carry out its developments (bootstrap). This approach consists of always using the latest version of the tool, in other words making the

team’s "toolkit" evolve by using the version of Modelio that is currently in the "frozen space". In this way, possible problems with built versions are quickly detected by developers, who are extremely motivated to correct them as quickly as possible. In other words, developers are continually testing the reliability of the latest version.

Version branch management is always complex. Branches must be reduced to the strict minimum required. The Modelio product typically has a maintenance branch (current version) and a future version branch. Bug corrections made in the current version must also be made in the future version. Modelio provides a very useful model diff/merge tool for this merge. For Java, it is also possible to generate the code on both branches, to run a diff/merge operation via the dedicated textual tool on the source code and then to re-import (round-trip) the result into the model.

Modelio features used for this development

The Modelio/Teamwork Manager module coupled with Subversion

The Modelio tool is coupled with the popular open-source Subversion tool, enabling work carried out by a group of participants on a model to be managed, and also providing assistance with configuration and version management activities. Modelio guarantees the permanent integrity of the central repository through a safe locking system, whereby the elements modified by each participant are locked, and enabling parts of the model to be reserved (check-out) and delivered (commit, check-in). With Modelio,elements that can be locked are packages, but also classes, use cases, actors, etc., as these are the lowest level for autonomous work by participants.

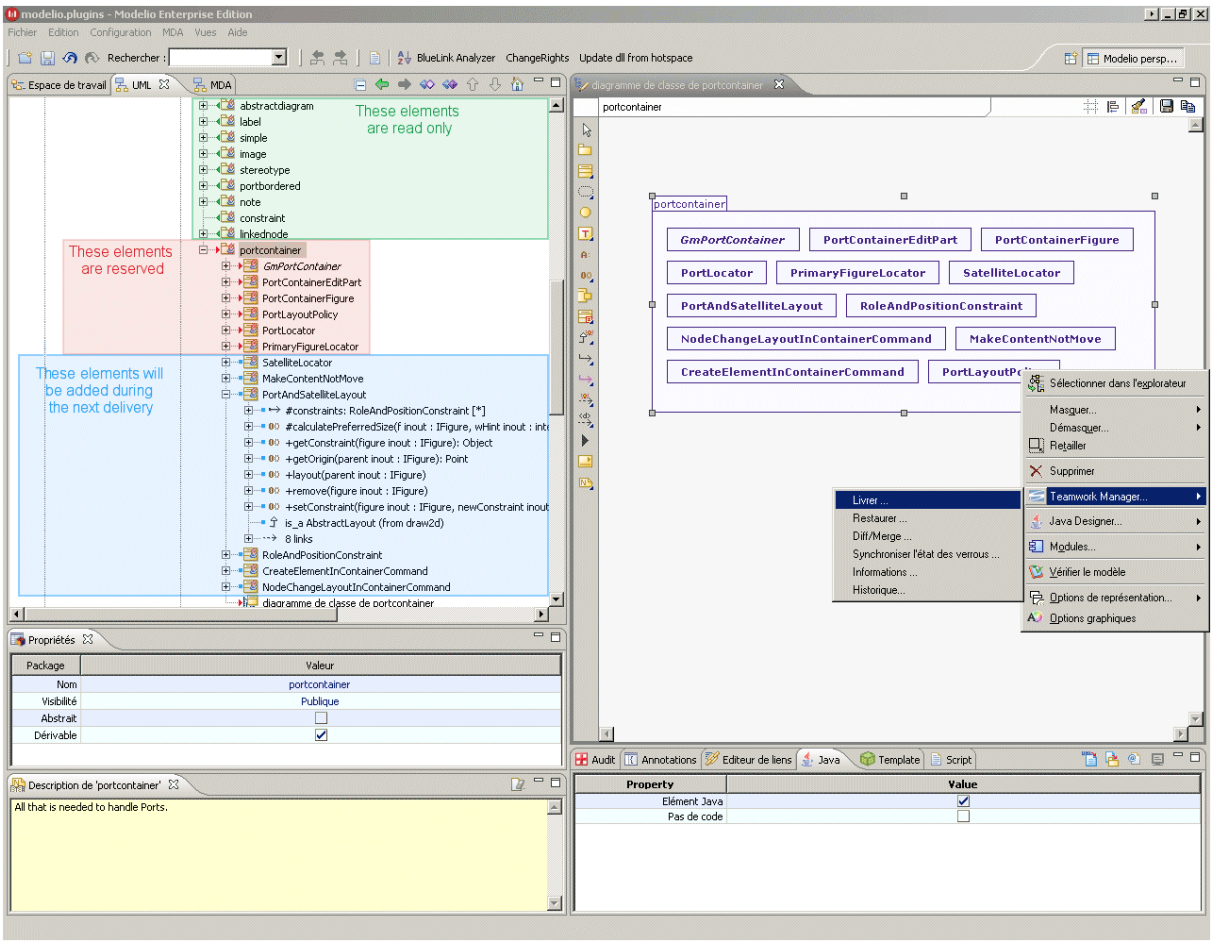

Figure 7 – *Modelio manages the delivery states of different model elements and provides SVN services*

The centralized administration of cooperative Modelio projects manages the consistency of each participant's environment (modules used, version of tools and modules, libraries used). This ensures that all participants are working in a homongenous environment, at the same level as the rest of the team.

Model components

In Modelio, model components are independent, identified and coherent parts of a larger model, packaged into a single file. Model components enable developers or teams of developers working on different Modelio projects to set up mechanisms for the delivery/reception of model parts.

Model components increase the efficiency of large teams, by reducing the size of the different model extracts involved in a project, and the number of people working on any given model part. This means that each development team can progress in its own teamwork environment and according to its own schedule, without affecting the work of the other project participants. It is interesting to note in practice how this breakdown into components, which is sometimes seen as a constraint during the initial set-up, rapidly improves the general architecture of the application.

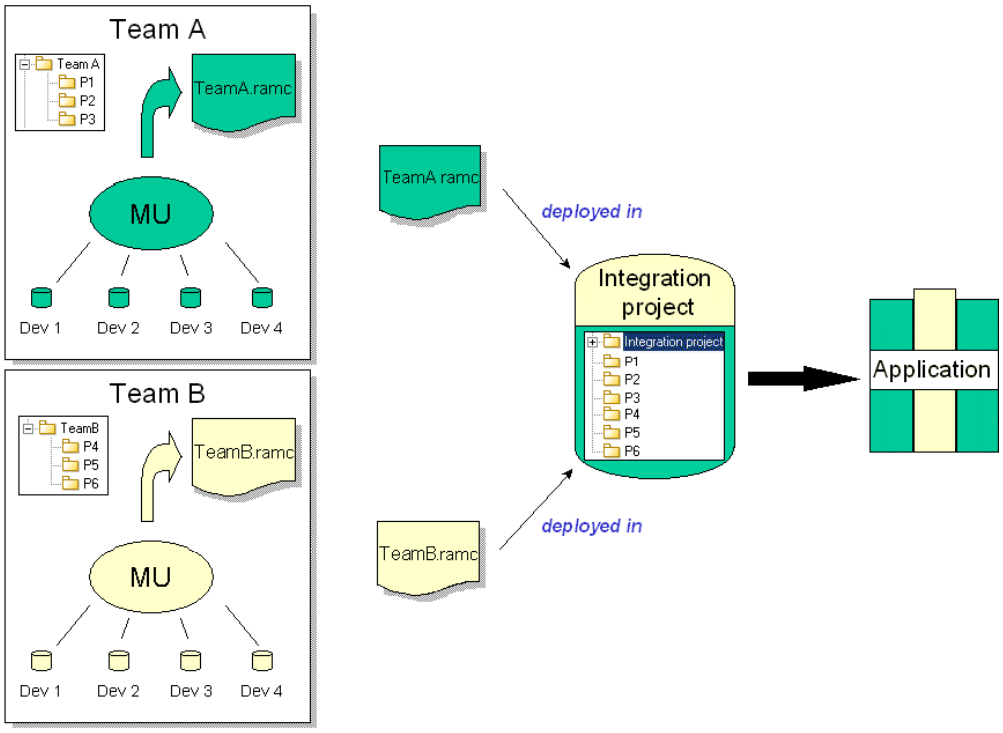

Figure 8 – *Example of organization using Modelio model components*

Customizing Modelio Teamwork Manager with the "PEM" module

The Modelio Teamwork Manager module can be extended through a "companion" module called PEM (Process Extension Module). With the "MDA Designer" module, simply develop the proposed extension points in Java (see the "Develop a process extension for Modelio Teamwork Manager" tutorial - http://www.modeliosoft.com/en/tutorials/modelio-tutorials.html). This module enables the Teamwork Manager module to be perfectly adapted to each organization's functional process, and the process to be automated. It provides Java entry points (operations) to be redefined. These will be called before and after the main check-in/check-out operations: precheckin() ; precommit() ; postcommit() …

In this way, customization can send an email, store information in databases, run a compilation, run tests, and so on. This customization depends on the maturity of an organization and on its methodology. In order to have a good balance between control of the process and flexibility of use, and also to encourage the adhestion of participants to the process, the benefit of the controls must counter-balance the annoyance of the constraints.

The model diff/merge function

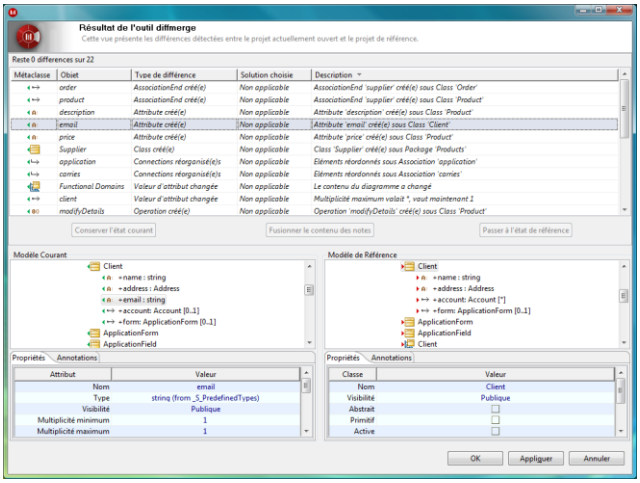

Figure 9 – *Modelio provides a model diff/merge service*

To ensure that teamwork is complete at model level, the same services and facilities must be available on models as on sources. All developers are familiar with the utility tools for source diff/merges, which present differences and manage changes at source code level, in an extended text editor. Modelio provides the same feature at model level, enabling users to compare a reference model to another model and present the result of the merge decisions. This operation is clearly made easier if the projects have been broken down into separate layers and themes using Modelio organization mechanisms (Project/Sub-Project, model component) and UML organization mechanisms (Package).

Modeling split into three levels

According to the realization phase in question (analysis, design, detailed design/coding), the model parts implemented are not identical.

During the analysis phases, models are used to build the application's dictionary, define its requirements, realize use cases, construct design models and use all UML's modeling capacities. Model construction and documentation generation are the main results of this phase.

The design phase focuses on realization, building productive models, essentially class models, which concentrate on main architectural themes and the system's essential classes. The design model produces the application code, either through integral generation via Modelio's MDA technologies, or through code generation in "model-driven" mode.

The coding/detailed design phase completes the general design model in "round-trip" mode, in other words by massively using the features of the Eclipse/Java environment and by permanently resynchronizing the model.

Code generation: model-driven and round-trip

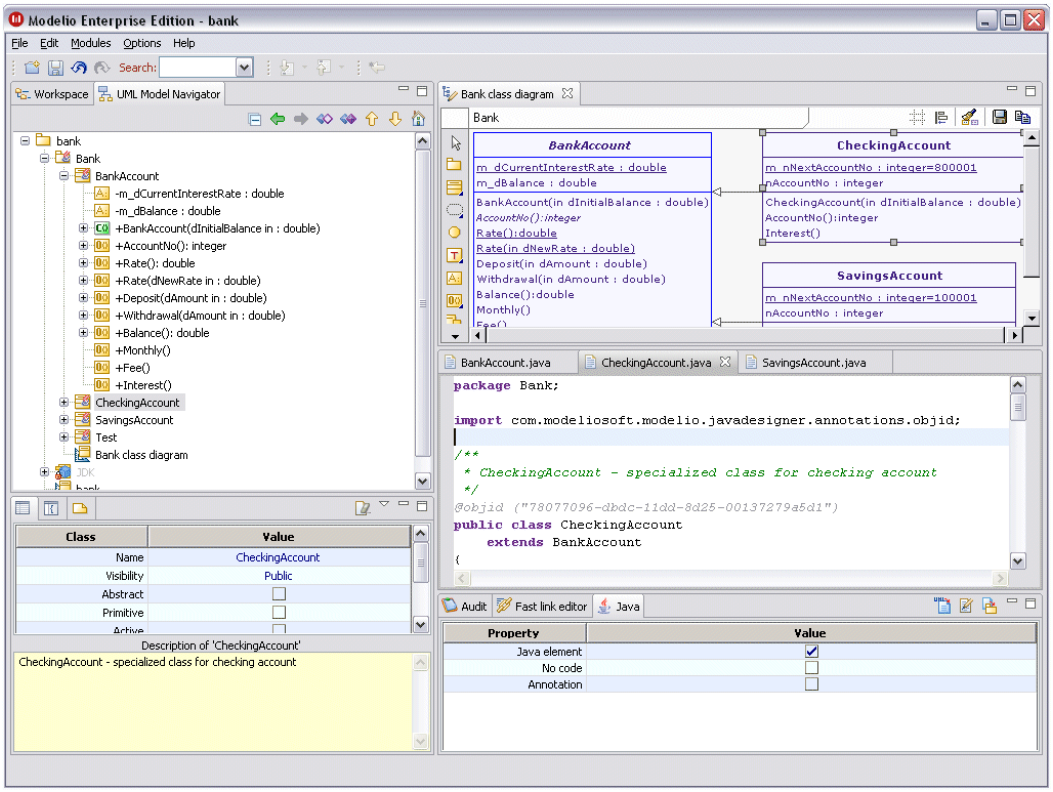

Figure 10 – *Modelio Java Designer permanently manages UML model/Java code consistency*

The model-driven approach must balance the advantages and disadvantages of model orientation or code orientation. Modeling everything before coding can be excessive in the case of small technical Java classes; however, a code-centric approach can result in the production of huge amounts of often badly constructed code. For example, Eclipse is a great tool for productivity, with code "refactoring" features and practical completion mechanisms. However, completion (for example, when Java methods are called) transgresses all design rules, such as the respect of encapsulation, architectural layers or the avoidance of mutual dependencies, quickly resulting in a large amount of badly structured code.

Modelio enables users to choose between two model-driven approaches:

- Model-driven: The model is developed and then the code generated, with authorization to write method code between markers in the Java code. Modelio is therefore in charge, with Eclipse (in the case of Java) being used to complete the skeleton of product code. In theory, this is the recommended mode, since it guarantees that model-level design is respected.

- Round-trip: The code is freely altered, and Modelio resynchronizes the model with the code. In this case, Eclipse is in charge, with Modelio adapting the model to the code (Java reverse). This mode is popular among developers, since they are free to take full advantage of the power of the Java environment.

Modelio retains all the information – model and code complements – in its repository. At any time, all the code can be regenerated from Modelio. In this way, whether the approach is "model-driven" or "round-trip", the reference remains the model, which means that only the model need be version-managed.

The strict "model-driven" mode is used during the design phase. Design does not go into detail concerning highly specific Java classes, typically in the case of GUI construction. For example, an "ActionListener" class will not be modeled. The GUI is simply built using Eclipse features, possibly using specific complementary tools. However, the main dialog classes are modeled and then generated, along with the generic GUI classes, and the rest is carried out in Eclipse and then reversed in Modelio (using the "Update from source" command).

Implementation uses the "round-trip" model, which is more flexible for the programmer, while keeping design classes in model-driven mode.

Developers must be able to see the time saved through modeling. They must also be disciplined in their respect of the "model-driven" approach with regard to classes resulting from the design phase. In "round-trip" mode, everything is based on the responsibility of the developer, who must not do everything with code, thereby encountering the afore-mentioned disadvantages (absence of design, architectural decay). Giving developers responsibility is often the best way of getting them on board. Audit processus and light-weight checks of real practises must be put in place, for example through the proximity of project managers with developers, "peer reviews" and so on.

All developers have two screens linked to their workstation : one screen is dedicated to development (Eclipse-Java or Visual Studio-C++), while the other is used for modeling (Modelio). Using Eclipse perspectives and constantly switching from code to model turned out to be annoying.

MDA generation features dedicated to an architecture

Like the development of any IT system, Modelio is based on dedicated architecture. The systematic and repetitive aspects of the architecture lead to automation from specific modeling to code, in order that an architectural principal judged to be optimal after intial studies and implementation be systematically applied and automated. Aside from guaranteed quality and increased development performance, this approach enables an architectural principle to be re-applied to all concerned cases after an evolution, meaning the entire application can be modernized with a minimum of effort.

For our application target, which is the Modelio modeling tool itself, the dedicated MDA transformers and generators allow the following to be automatically produced from the metamodel (definition of the supported model through a specialized class diagram : UML, BPMN, requirements modeling, …):

- management of the persistence of this model
- construction of the model's "explorer" view
- creation of dialog boxes for different model elements
- model consistency checks

- production of its program handling interface (Java API)
- implementation of different functional mechanisms such as lock management, model diff/merge mechanism
- …

Modelio diagram editors are not automatically produced, since each has important particularities, but a model "pattern" is particularly helpful when programming editors, as it allows the developer to concentrate exclusively on the specific parts to be coded.

Impact analysis on models

With Modelio, users can determine for any given model element which model elements are linked to it. All possible dependencies (type, import, association, inheritance , …) are analyzed to determine which elements depend in whatever way on the analyzed element. An impact analysis diagram is then automatically produced. This diagram provides complementary analysis, useful in thoroughly understanding the causes of identified dependencies. This feature allows developers to see what consequences a change to a given model element would have on the entire application, which is particularly useful during the maintenance phase.

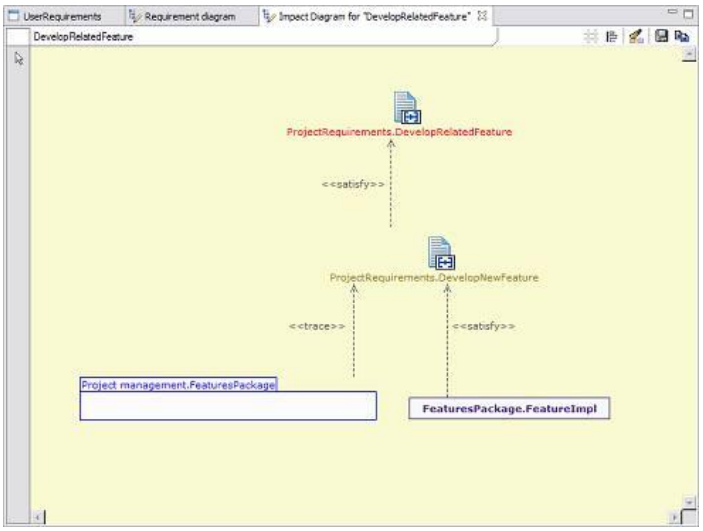

Figure 11 – *Impact analysis from a requirement in Modelio*

Conclusion

While it is relatively easy to show examples of model transformation or code generation resulting in impressive automation and immediate productivity, in practice this advantage is very often damaged by an incapacity to generalize mechanisms across a team, and to implement it within cooperation, configuration and version management procedures.

Successfully implementing teamwork is fundamental, since failure to do so means that there is no real team or team synergy. Configuration and version management questions are also decisive in terms of the mastery of the software produced. If one adds to this the advantages of a model-centric approach and MDA automation, the qualitative and production gains are maximized. Let’s not forget

that that pure coding only represents something like 30 to 40% of the work carried out on the realization of a software application, and that the cost of the realization phase is often around 20% of the total cost of possessing the software, including evolutive and corrective maintenance operations. A clear and tooled model-driven development process including cooperation and configuration and version management aspects will maximize the gains made during the maintenance phase, and will facilitate participant interchangability. However, code automation will facilitate the pure development phase, and the model-centric approach will increase the quality of the analysis and design phases, as well as the consistency of the whole, across all phases.

It is clear that the tools and procedures presented in this white paper strengthen and confirm that several CMMi levels of maturity have been reached. For example, in the definition of CMMi maturity levels, software configuration management appears as a key point from level 2 onwards, and the definition and improvement of the development process is an essential part of level 3.

Even though tools are important, the definition and appropriation of adapted procedures by all participants is essential to the success of a development project.

Implementing this approach within an organization requires an initial investment in the definition of an organization and its roles, and in the putting in place of the necessary tools. This investment will have paid for itself in a few months. Modeliosoft can provide consulting services on organization and process, and on the set-up and configuration of Modelio, in order to obtain a version and configuration repository and a centralized teamwork management. For more information, please contact sales@modeliosoft.com.