Implementing an MDA solution with Modelio for the development of military information systems

Client case study - DCNS

Emmanuel Grivot, Jérôme Veillard – DCNS

Philippe Desfray – SOFTEAM, Modeliosoft

The strategic importance of productivity, quality and rapid time to market drive all companies providing information systems to seek innovative solutions in order to optimize production. With this objective in mind, DCNS has developed an internal information system development process that combines a component-oriented approach with the implementation of the UML and MDA technologies using Modelio, and the use of aspect-oriented development environments (AOP). Productivity gains of 30% are expected on very large systems with strong quality constraints.

Overview

The successful implementation of an MDA approach in an industrial development context cannot be taken for granted. While it is relatively simple to carry out a demonstration of model transformation and code generation to illustrate the capacities of the MDA technology, the efficient implementation of MDA within an organization requires a global approach, adapted organization and suitable tools. This case study presents the implementation of an MDA approach within a large development team, split over several sites, using the Modelio tool suite. For this, a dedicated approach was put in place, combining MDA with aspect-oriented application development, and enabling the optimization and clarification of both the approach and the production process.

The DCNS context

DCNS' primary mission is the construction of ships and submarines for use by the military. DCNS has 12 200 employees split over 21 sites in France, and an annual turnover of 2,4 billion Euros. 30% of its turnover is generated through international and cooperative projects.

Figure 1 – *Examples of products constructed by DCNS*

Controlling a zone, securing airspace, protecting one or several buildings - these are all missions assigned to "Combat Systems" (CS), which are complex sets of arms and sensors. The Combat Management System (CMS) is responsible for coordinating all this equipment, thus playing the central role in the CS. The CMS analyzes gigantic information flows from receptors in real time, and provides sailors with an overall vision of the operational situation. It allows the positions and movements of a naval force to be visualized, and enables all sensors to be supervised and controlled, and arms system to be controlled.

As in all modern systems, IT plays a more and more important role in ships and submarines, both of which have a large amount of electronic equipment to coordinate. The development of a CMS takes between three and five years, with the number of lines of codes varying between 5 million and 15 million. CMS systems have to be maintained for at least 20 years. Optimizing the production and management of combat management systems is, therefore, a key issue for DCNS.

Besides increased productivity, realization times and quality, DCNS' objectives also include an improvement in the agility of the system and the development process. This means being well able to implement changes, whether these be functional (services provided by the CMS) or technical (technical capacities, implemented technologies).

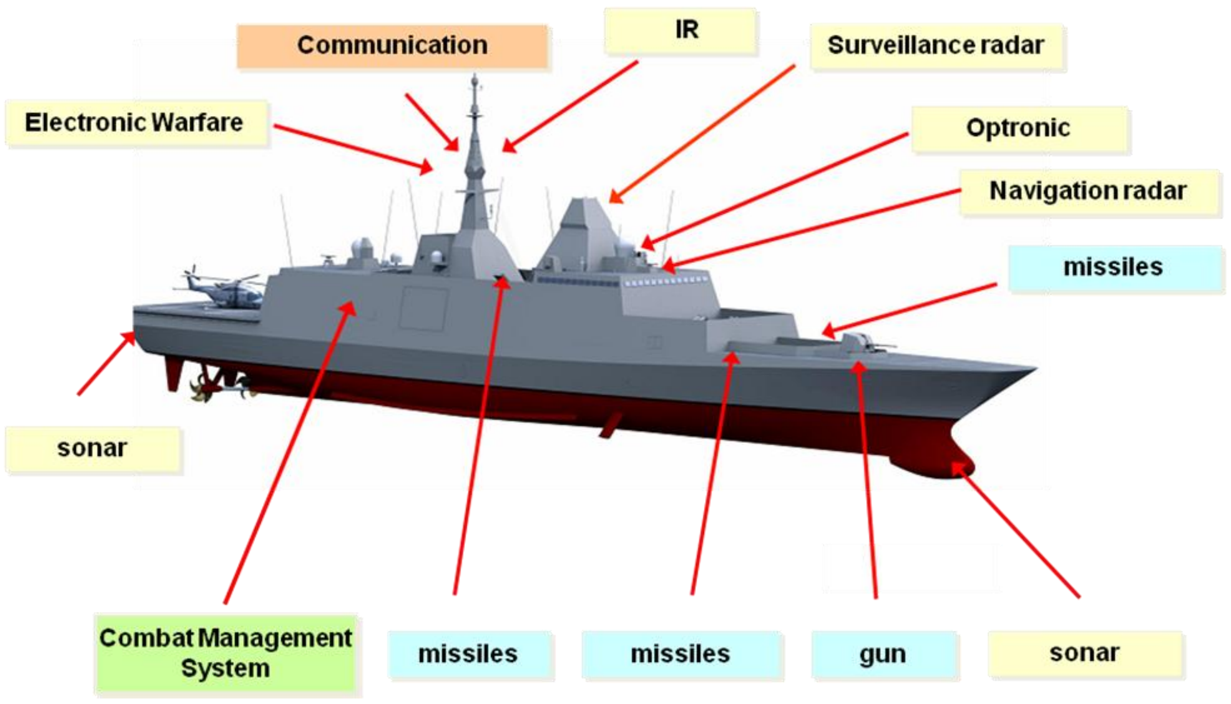


Figure 2 – *Equipment of a FREMM frigate*

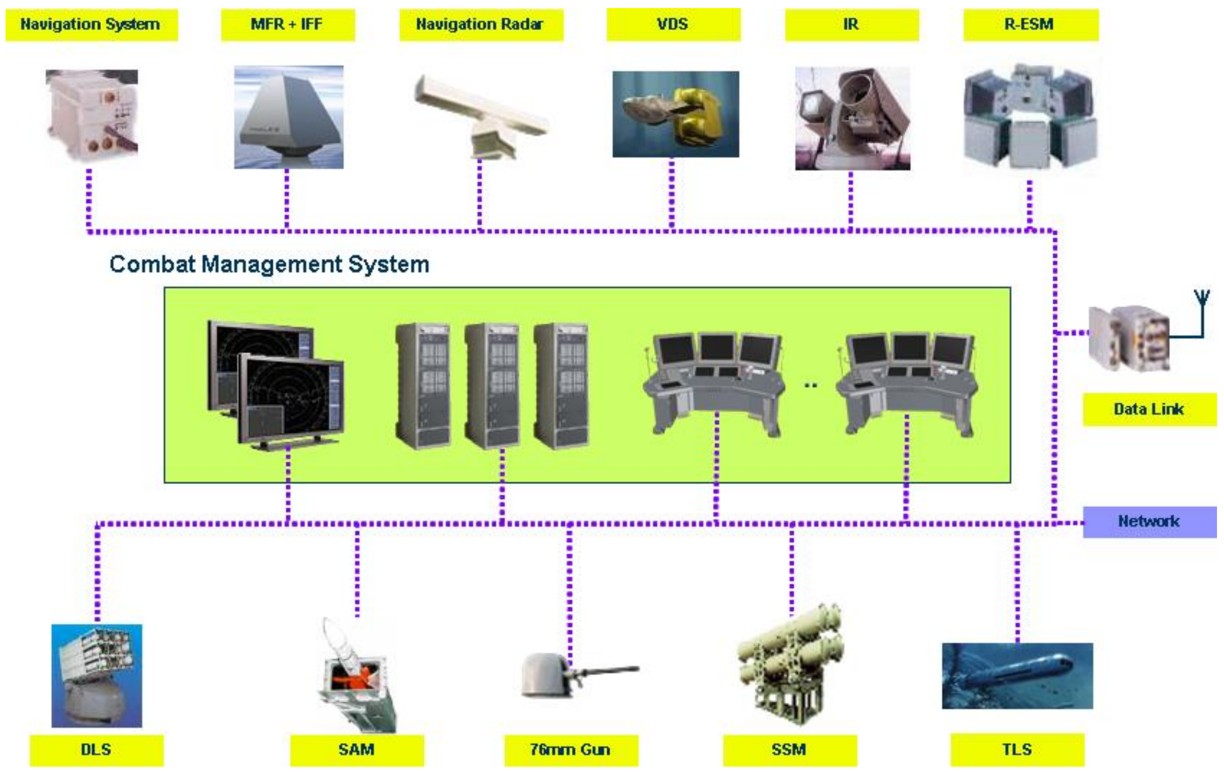


Figure 3 – *Characteristic structure of a CMS*

Optimizing the MDA approach by combining it with aspect-oriented programming

Ten years ago, DCNS set up an "MDA" approach using the Objecteering tool (www.objecteering.com). DCNS is now in the process of migrating this process to Objecteering's successor, Modelio (www.modeliosoft.fr).

DCNS uses a component-oriented approach, based on a component model named JACOMO (Java Component Model). JACOMO associates a service-oriented approach and an event-oriented approach (SOA and EDA) in its architectural vision. This approach structures modeling and defines a framework for the model-driven approach from specification right through to implementation and testing.

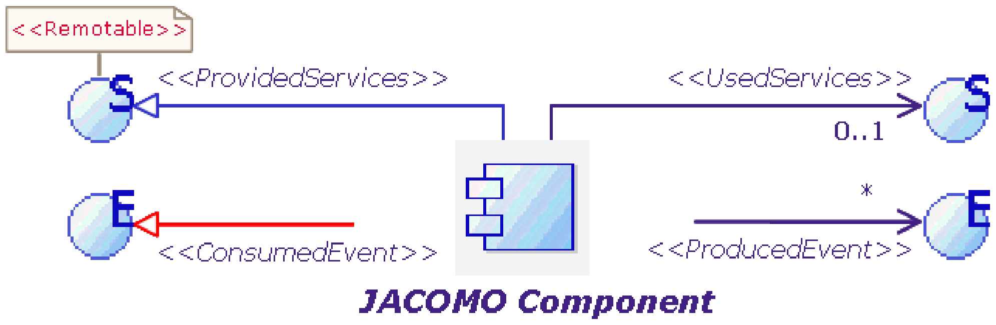


Figure 4 – *The JACOMO component*

A dedicated UML profile and specific modeling wizards, built in Objecteering and improved in Modelio, enable a large part of development and testing work to be guided and automated.

Initially, the entire model and all technical code complements were defined in Objecteering, before undergoing model transformation and code generation operations dedicated to DCNS' CMS architectural frameworks. This solution presented the advantages of an MDA approach, but was subsequently perfected in the aim of increasing development agility, by delegating many technical and architectural aspects to aspect-oriented development environments (the weaving technique). In most programs, less than 30% of the code corresponds to "functional" code that carries out the services required of the system. The rest of the code corresponds to technical aspects, initialization management, distribution management, storage management, and so on. This technical complexity is traditionally found in models used for code generation.

Aspect-oriented programming enables users to focus on functional aspects. Non-functional needs are expressed through annotations (meta-data), which indicate those aspects to take into account. Code weaving tools ensure that the annotated code is enriched with code specific to the referenced aspects. The distribution of (remote) applications is, for example, managed by a specific aspect stereotyped <<Remotable>>, and added, through weaving, to the parts of the code annotated for this aspect (@remotable). Weaving thus diminishes the need for code generation and makes implementation significantly simpler.

The model is lighter, since all the technical code has been removed. Annotations are clarified at model level, and simply transferred back into the generated code. The code is then enriched through weaving, using the dedicated technical code.

Figure 5 presents the combination of different levels of model (in the MDA sense), the equivalent annotated Java, and a weaving stage, producing the complete final code in accordance with the annotations. Design and development activities take place either in the model and/or in the annotated Java code, with the weaving stage automatically producing the final result.

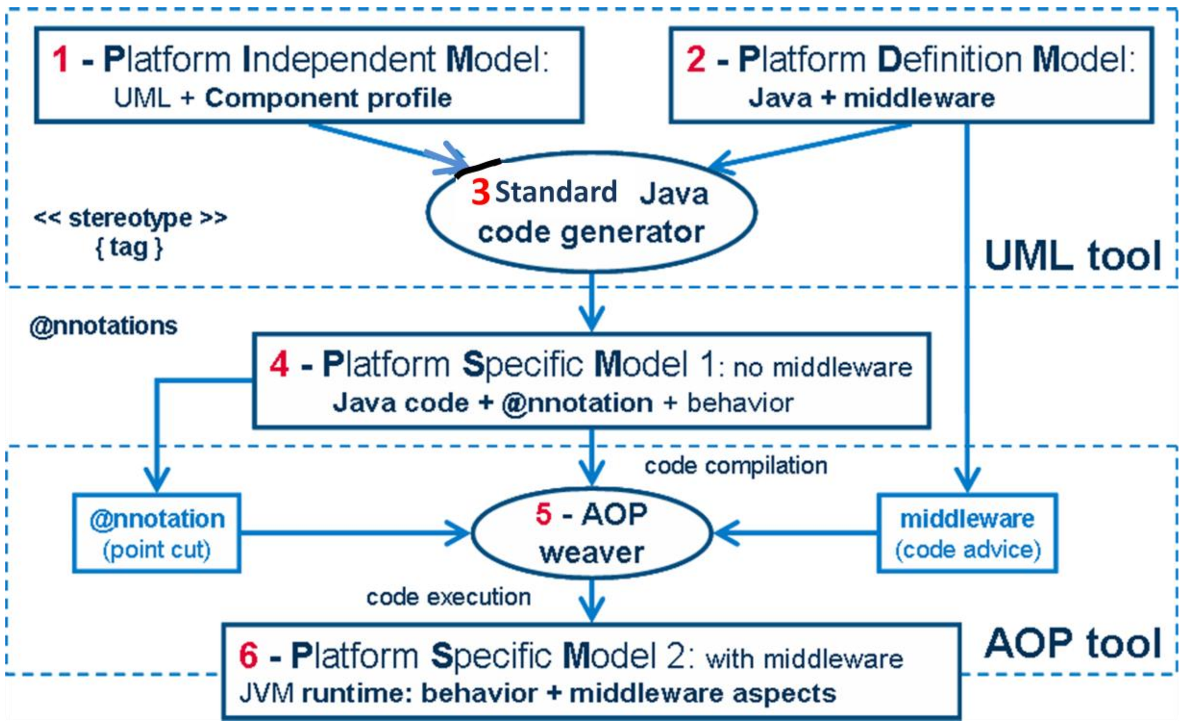

1 - Platform Independent Model:
UML + Component profile
2 - Platform Definition Model:
Java + middleware
3 Standard Java code generator
<< stereotype >>
{ tag }
UML tool
@nnotations
4 - Platform Specific Model 1: no middleware
Java code + @nnotation + behavior
code compilation
@nnotation (point cut)
5 - AOP weaver
middleware (code advice)
code execution
6 - Platform Specific Model 2: with middleware
JVM runtime: behavior + middleware aspects
AOP tool

Figure 5 – *Combination of the MDA and AOP approaches in a tooled approach*

Implementation example with the Modelio tool

As an example, let's look at the notion of tracks, which corresponds to the radar detection and subsequent analysis of a moving mobile. The "track" notion is a business notion, which is extremely important for combat systems. This information is managed and displayed in real time, providing critical data to the users of a combat system.

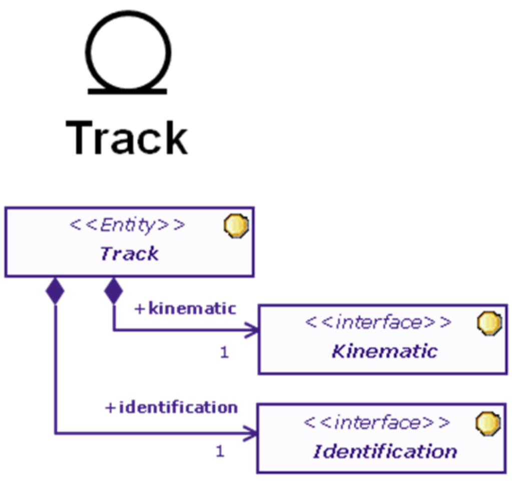


Figure 6 – *The "Track" business notion*

A track is defined by a kinematic and by identification data. Based on this business data, the designer decides to define a JACOMO component to manage it. A specialized Modelio wizard is used to support this modeling (specific UML profile).

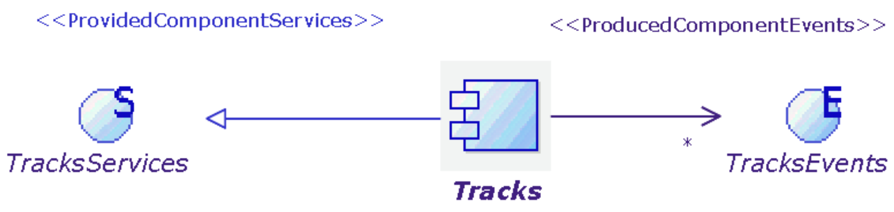
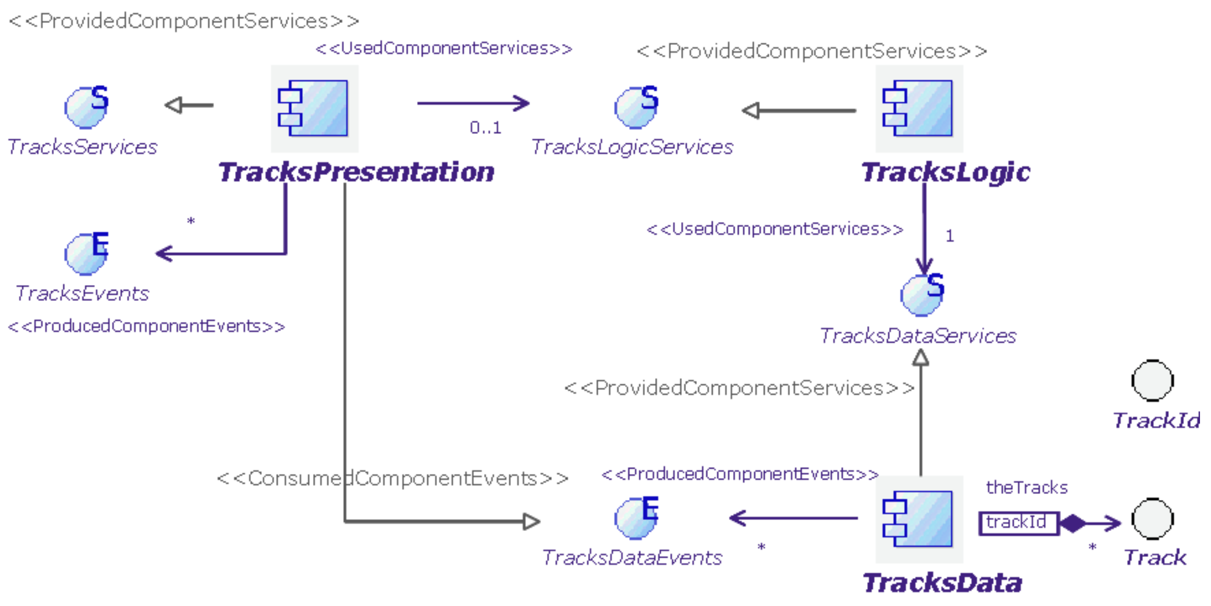


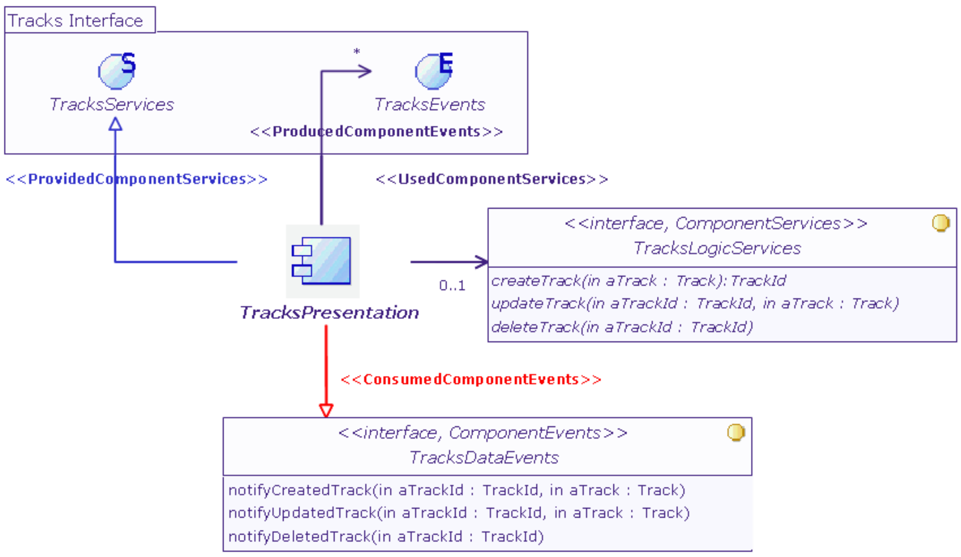
Figure 7 - *The "Track" JACOMO component*

The wizard developed in Modelio enables the architecture model shown in Figure 8 to be built. This architecture model breaks down the "Tracks" component into three layers, presentation, logic and data, which take into account the fact that the user must be able to create, read, update and destroy a "Track" (CRUD).



<<ProvidedComponentServices>>
<<UsedComponentServices>>
<<ProvidedComponentServices>>
TracksServices
TracksPresentation
0..1
TracksLogicServices
TracksLogic
*
TracksEvents
<<ProducedComponentEvents>>
<<UsedComponentServices>>
1
TracksDataServices
<<ProvidedComponentServices>>
TrackId
<<ConsumedComponentEvents>>
<<ProducedComponentEvents>>
theTracks
trackId
TracksDataEvents
*
*
TracksData
Track

Figure 8 – *Three JACOMO components share the management of "Track" over three levels*



Tracks Interface
TracksServices
*
TracksEvents
<<ProducedComponentEvents>>
<<ProvidedComponentServices>>
<<UsedComponentServices>>
TracksPresentation
0..1
<<interface, ComponentServices>>
TracksLogicServices
<<ConsumedComponentEvents>>
<<interface, ComponentEvents>>
TracksDataEvents

```
createTrack(in aTrack : Track): TrackId
updateTrack(in aTrackId : TrackId, in aTrack : Track)
deleteTrack(in aTrackId : TrackId)
```

```
notifyCreatedTrack(in aTrackId : TrackId, in aTrack : Track)
notifyUpdatedTrack(in aTrackId : TrackId, in aTrack : Track)
notifyDeletedTrack(in aTrackId : TrackId)
```

Figure 9 – *More detailed view of a component created by the Modelio wizard*

This wizard also produces the model and structure of black box and white box tests on the "TracksData" component. These tests are based on the open source "JUnit" library.

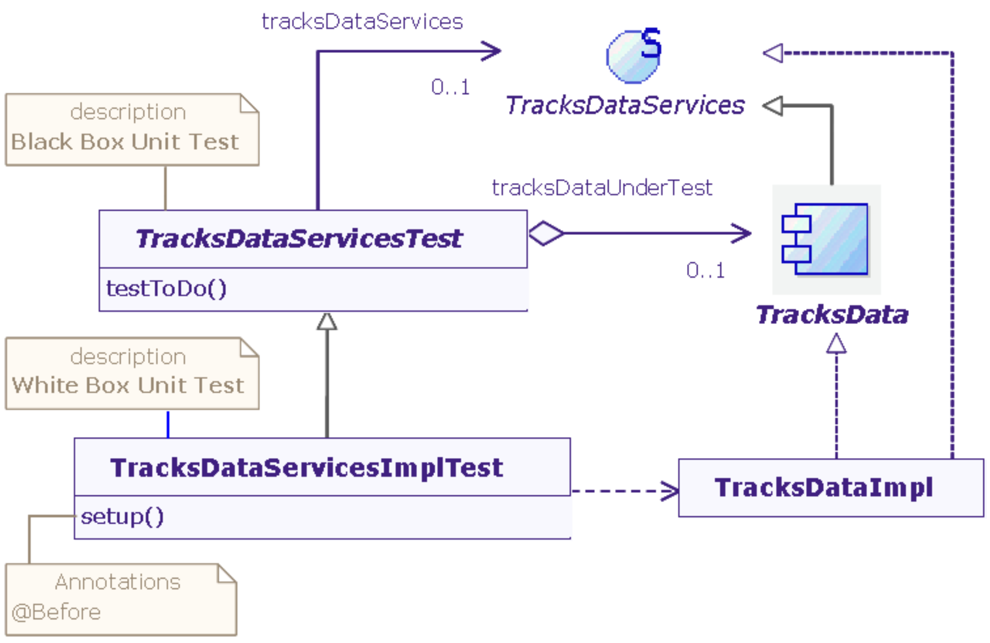
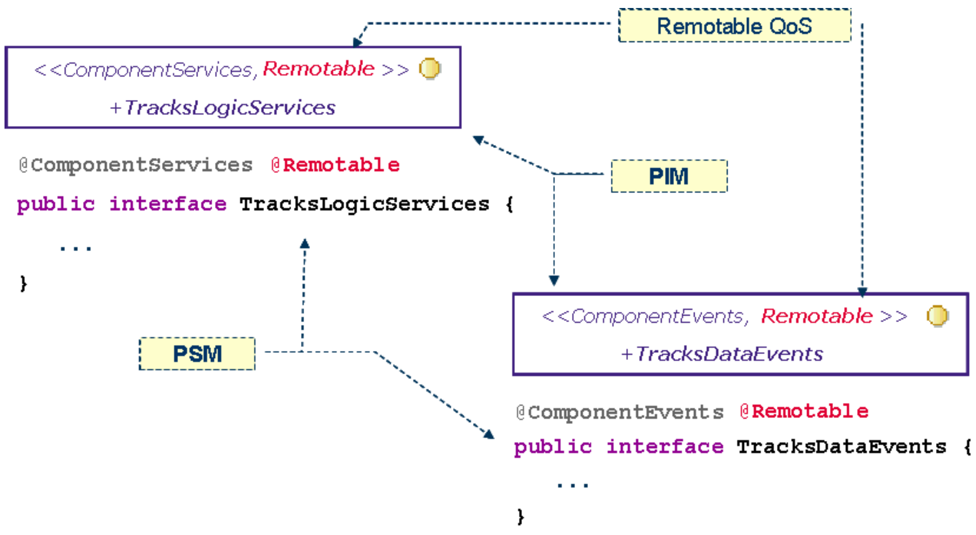


Figure 10 – *Models of white box tests and black box tests of "TracksData"*

On the basis of these models, the developer adds stereotypes according to the technical choices made (for example "Remotable", "Persistable" or "Asynchronous"), and then completes his model with operations and functional Java code added to these operations. This can be entered either in Modelio at model level, or in Eclipse, which is better adapted to coding. It is then managed in round-trip mode, which enables perfect synchronization with the model.

The stereotypes linked to the technical choices are simply translated as Java annotations in the code produced (@Remotable, @Persistable, @Asynchronous). The level of abstraction of the code is therefore very close to that of the model (it is a simple bijection).

Figure 11 shows an example of PIM to Java PSM transformation, with the <<Remotable>> stereotype translated as a @Remotable annotation.


Remotable QoS
<<ComponentServices, Remotable >>
+TracksLogicServices
PIM
PSM
<<ComponentEvents, Remotable >>
+TracksDataEvents

```
@ComponentServices @Remotable
public interface TracksLogicServices {
    ...
}
```

```
@ComponentEvents @Remotable
public interface TracksDataEvents {
    ...
}
```

Figure 11 – *The <<Remotable>> stereotype transformed into a @Remotable Java annotation*

With this approach, a simple redeployment of components enables a monolithic application running on a single JVM to be transformed into a client server application running on different machines.

Results obtained and perspectives with these techniques

The Objecteering tool has been used by DCNS since 2000 to support their model-driven development. The JACOMO profile and the first wizards were put in place in Objecteering in 2004. Migration to Modelio took place in 2010, in order both to migrate existing models, and to migrate and extend the JACOMO profile, wizards and MDA transformers. Everything is based on the standard Java code generation provided off-the-shelf by Objecteering or Modelio, with its code/model synchronization system. The implementation of the tooled JACOMO process with Modelio is relatively simple, requiring no specific code generators (customization of the Modelio Java code generator proved to be sufficient).

The Objecteering and then Modelio tools are also used to automatically produce DoD (Department of Defense) documents: "SRS" (Software Requirements Specifications), "IRS" (Interface Requirements Specifications) and "SDD" (Software Design Documents).

More than 1200 JACOMO components have been developed using the MDA JACOMO approach, and more recently aspect weaving.

Roughly 40% of the code results from generation from UML models, with 60% being manually coded inside methods.

For a component, 17% of code lines concerns the definition of interfaces, 48% implementation and 35% testing.

All "framework" code is in the modeling tool (Objecteering then Modelio), including unit test code, with the obvious exception of the technical code obtained by aspect weaving.

Conclusion

Large systems are increasingly functionally complex, making it essential to avoid adding accidental complexity due to technical environments. In order to clearly distinguish between complexity and complication, the joint use of a model-driven approach (MDE) and aspect-oriented development (AOP) avoids mixing up technical and functional problems, making it much easier to master each individual type of complexity.

Model-driven development has enabled DCNS to raise the level of abstraction in analysis and design, and to ensure their continuity and traceability right through to the code. Aspect-oriented programming has enabled us to extract technical code from models (up to 70% of the code). The model supports our development process (JACOMO approach, unit tests) and allows the automation of our code production. By combining MDE and AOP, we are able to apply MDA and to make our systems evolve, both on a functional level (changes at model level) and on a technical level (changes at aspect and weaving levels). We estimate that the entire approach should bring us productivity gains of between 20% and 30% throughout the CMS development cycle.

The deployment of the Modelio tool has brought us all the advantages of its improved ergonomics, as well as its integrated Subversion configuration management.