CHAPTER 5

Integration of MDSE in your Development Process

MDSE is process-agnostic, i.e., it neither provides nor enforces any specific development process, but it can be integrated in any of them. Furthermore, MDSE *per se* does not define which models must be used in each step of the development process, at what abstraction level, how they must be related, and so on. It is up to each organization to define the right methodology (*right* in terms of the organization's context and requirements) to successfully apply MDSE.

Nevertheless, as happens with the introduction of any complex technology in an organization, adoption of MDSE is not straightforward. In fact, one of the typical failure reasons for MDSE projects was to think that you build a code generator and start increasing your productivity tenfold the day after. Integration of MDSE must be a step-by-step process that takes into account your specific organizational context.

This chapter first provides some general considerations on the adoption of MDSE and then discusses how MDSE can be merged with five well-known software development approaches of various kinds.

5.1 INTRODUCING MDSE IN YOUR SOFTWARE DEVELOPMENT PROCESS

We have witnessed how some organizations interested in giving MDSE a try drop it after an initially failed attempt or are unable to draw all the benefits that supposedly come with the use of MDSE in practice.

As with any other technology adoption, organizational, managerial, and social aspects (as opposed to simply technical ones) are the main factors of failure [34]. Neglecting to consider these aspects hampers the successful adoption and efficient application of a new MDSE-based process, eluding the benefits that modeling could bring, and even worse, discouraging its use in future projects.

Some aspects to take into account in your first MDSE project are common sense: start with a small project, choose a non-critical one (so that there is no additional pressure for the development team), make sure you have the commitment of the management to support you when inevitably some problems arise during the project, get some external help (e.g., an expert MDSE consultant to help you iron out the details of the adoption process), and so forth.

Nevertheless, other aspects are a little bit more subtle, and therefore, deserve more careful attention. In the following, we comment on what we believe are two of the major challenges in the adoption of MDSE: (i) the imbalanced distribution of efforts and rewards among team members, and (ii) the lack of socio-technical congruence.

5.1.1 PAINS AND GAINS OF SOFTWARE MODELING

The adoption of a model-based process introduces new tasks and roles into the development process. These additional tasks are expected to be beneficial for the quality and productivity of the overall team. However, at the personal level, not all members benefit equally from the change. In fact, some members, depending on their assigned role, can perceive modeling as detrimental since it imposes additional burdens that benefit other people in the team but not themselves. Therefore, they may not see MDSE as cost effective: they suffer the cost but they may not realize the positive overall impact.

As an example, when precise and up-to-date UML models are available, correctness of changes performed during the software maintenance is clearly improved. This will benefit the subteam in charge of that task, but the chore of developing those models would fall on the analysis subteam responsible for creating such models. When both activities are performed by the same team members, they easily realize that the extra effort of creating the UML models (the "pain") at the beginning pays off in the end since they benefit from using those same models (the "gains") when performing maintenance tasks. However, when the activities are performed by two different groups, the group in charge of doing the models does not directly benefit from those models and may doubt that their effort is worthwhile.[1]

Therefore, when the work distribution among the members separates the pains and gains of modeling, we must adopt policies that recognize (and compensate for) those differences. Otherwise, we may face motivation problems during the performance of the pain activities which would limit the benefits we may get with the gain ones.

5.1.2 SOCIO-TECHNICAL CONGRUENCE OF THE DEVELOPMENT PROCESS

The technical requirements of the MDSE-based process may disrupt the social elements of the team due to the introduction of new roles, skills, coordination patterns, and dependencies among the team members.

All these elements must be carefully considered to ensure that the organization fits the socio-technical requirements of the new MDSE process, where fit is defined as the match between a particular design (social structure) and the organization's ability to carry out a task under that design [16]. For instance, the organization must be able to match the roles of the target process

[1]The difficulties of introducing an application/process that provides a collective benefit but not a uniform individual benefit is well known, e.g., see [31] for a study for the groupware domain.

on the actual team members, ensuring that each member has the skills required to play that role. A good socio-technical congruence is a key element in the software development performance.

As an example, imagine the extreme case where an organization with no modeling experience decides to adopt a full MDD approach with complete automatic code generation. This change has a lot of (direct and indirect) effects on the development team and its social organization. Therefore, to succeed in this endeavor, the team needs to be willing and able to adopt new roles (e.g., the platform expert, language engineer, transformation developer, domain expert) with the corresponding new skills (e.g., proficiency in modeling, model transformation, and language development techniques) and dependencies (the designer depends on the platform expert for knowing the platform capabilities, on the language engineer for the language to use when defining platform-specific models, on the transformation developer for transforming platform-independent models into platform specific ones, etc.). This is not a trivial evolution and requires time and means (e.g., to take training courses) but it is a necessary condition for the successful adoption of MDSE.

5.2 TRADITIONAL DEVELOPMENT PROCESSES AND MDSE

Under the banner of "traditional" development processes, we group all development processes that prescribe to follow the sequence of "classical" phases (requirement elicitation, analysis, design, implementation, maintenance, etc.) in any software development project, regardless whether these activities are performed following a waterfall model, spiral model, an iterative and incremental model (like in the well-known Unified Process [37]), etc. This is in contrast with the development processes described in the next sections which depart significantly from this schema.

In general, all traditional processes are already model-based, i.e., they consider models as an important part of the process and propose to use different kinds of models to represent the system in the successive stages of the development process. In this sense, most include the notion of requirement models, analysis models, and design models. What MDSE brings to these processes is the possibility of going from *model-based* to *model-driven*, i.e., the opportunity to use MDSE techniques to (at least partially) automate the transition between the different phases of the development process. Model-to-model transformations may be used to refine the models in the early phases, while model-to-text transformations may be employed to generate the actual software implementation at the end of the process.

5.3 AGILE AND MDSE

Agile methods are a family of development processes that follow the principles of the Agile Manifesto,[2] like Scrum, XP, and others. The Agile Manifesto proposes to center the development around four main principles:

[2]http://agilemanifesto.org

1. individuals and interactions over processes and tools;
2. working software over comprehensive documentation;
3. customer collaboration over contract negotiation; and
4. responding to change over following a plan.

These values are implemented in a dozen principles related to the frequent delivery of working software, tight co-operation between customers and developers, continuous attention to the good design of the software, and so forth.

So far, Agile and MDSE have not often been used in combination. The Agile community tends to see MDSE as a superfluous activity that has no place in their view of *working software* as the only measure of project progress. Their main criticisms of MDSE (summarized by Stephen Mellor) are that models:

- don't run (i.e., they are not working software);
- can't be tested;
- are just documentation; and
- require extra work and alignment (i.e., they are not adequate for late requirement changes).

However, these criticisms are basically resulting from a limited understanding of the concept of model. If one interprets *models as sketches* (you draw them on your whiteboard and then you just throw them away) or *models as blueprints* (aimed at directing and planning the implementation before actually building it) then it is true that this view of modeling is not agile.

However, we have already seen in Chapter 3 that *models can be executable artifacts*. In this case, models are neither just pictures nor guidelines. They are meant to be an active part of the system's implementation and verification. They are built under the assumption that design is less costly than construction, and thus, it makes sense to put an effort on their realization, because then construction will come (almost) for free. In this sense, models are working software. Therefore, you can definitely be agile while developing with model-driven techniques.

Some initial proposals show how these two paradigms can interact. The most relevant is the *Agile Modeling* (AM) [4] initiative lead by Scott W. Ambler.[3] Simply put, AM is a collection of modeling practices that can be applied to a software development project in an effective and light-weight (i.e., agile) manner. The goal is to avoid "modeling just for the sake of modeling." The principles of AM can be summarized in the following:

- *Model with a purpose*. Identify a valid purpose for creating a model and the audience for that model.

[3]http://www.agilemodeling.com

- *Travel light*. Every artifact will need to be maintained over time. Trade-off agility for convenience of having that information available to your team in an abstract manner.

- *Multiple models*. You need to use multiple models to develop software because each model describes a single aspect/view of your software.

- *Rapid feedback*. By working with other people on a model you are obtaining near-instant feedback on your ideas.

- *Favor executable specifications*.

- *Embrace change*. Requirements evolve over time and so do your models.

- *Incremental change*. Develop good enough models. Evolve models over time in an incremental manner.

- *Working software is your primary goal*. Any (modeling) activity that does not directly contribute to this goal should be questioned.

- *Enabling the next effort is your secondary goal*. Create just enough documentation and supporting materials so that the people playing the next game can be effective.

These agile modeling practices and principles are the basis of what we could call an *Agile MDD* approach where the focus is on the effective modeling of executable models to transition from models to working software automatically in the most agile possible way.

Note that not only can we integrate MDSE practices in an agile development process but the inverse is also true. We can reuse agile techniques to improve the way we perform MDSE and build MDSE tools. Why not apply eXtreme Programming (XP) and other agile techniques when designing new modeling languages, models, or model transformations? If so many studies have proven the benefits of eXtreme Programming, why not have eXtreme Modeling as well?

5.4 DOMAIN-DRIVEN DESIGN AND MDSE

Domain-driven design (DDD) [22] is an approach for software development based on two main principles:

1. The primary focus of a software project should be the domain itself and not the technical details.

2. Complex domain designs should be based on a model.

As such, DDD emphasizes the importance of an appropriate and effective representation of the problem domain of the system-to-be. For this purpose, DDD provides an extensive set of design practices and techniques aimed at helping software developers and domain experts to share and represent with models their knowledge of the domain.

Clearly, DDD shares many aspects with MDSE. Both argue the need of using models to represent the knowledge of the domain and the importance of first focusing on platform-independent aspects (using the MDA terminology) during the development process. In this sense, MDSE can be regarded as a framework that provides the techniques (to model the domain, create DSLs that facilitate the communication between domain experts and developers if needed, etc.) to put DDD in practice.

At the same time, MDSE complements DDD (Figure 5.1) by helping developers to benefit even more from the domain models. Thanks to the model transformation and code generation techniques of MDSE, the domain model can be used not only to represent the domain (structure, rules, dynamics, etc.) but also to generate the actual software system that will be used to manage it.

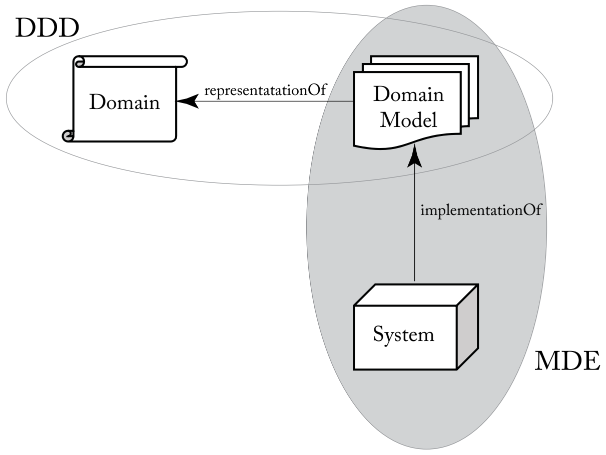

Figure 5.1: Relationship between DDD and MDD.

5.5 TEST-DRIVEN DEVELOPMENT AND MDSE

Test-driven development (TDD) [8] relies on the test-first philosophy. This means, the developer first writes an executable test case that will be used to check the new functionality to be implemented. If the test fails (which is supposed to happen since the new functionality has not yet been coded), the developer writes the code needed to pass the test (i.e., which in fact "forces" to implement the functionality). Once the new code passes the full test suite, the code can be refactored and the process starts again.

MDSE can be integrated into a TDD process at two different levels, depending on whether models will be used as part of a code generation strategy to automatically derive the system implementation or not.

5.5.1 MODEL-DRIVEN TESTING

When models are used to specify the system but the system is not automatically generated from them, models can be used to derive the tests that the system's implementation will need to pass in order to ensure that it behaves as expected (i.e., it behaves as defined in the model). This is known as model-based testing. As shown in Figure 5.2, a test generation strategy is applied at the model level to derive a set of tests. This set is subsequently translated into executable tests that can be run on the technological platform used to implement the system. Testing strategies depend on the kind of model and test to be generated. For instance, constraint solvers can be used to generate test cases for static models while model checkers can be used to generate relevant execution traces when testing dynamic models.

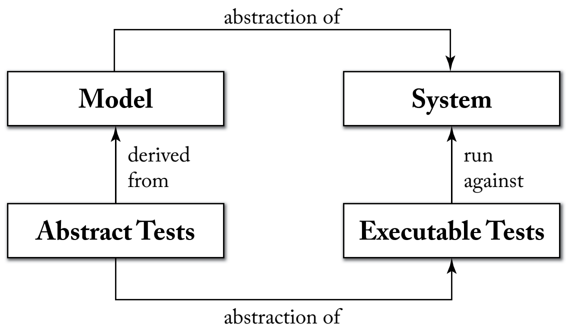

Figure 5.2: Model-based testing.

5.5.2 TEST-DRIVEN MODELING

When software is derived from models then there is no need to test the code (of course, assuming that our code generation is complete and that we trust the code generator). Instead, we should test the models themselves. In this sense, some approaches for the test-driven development of modeling artifacts have been proposed, e.g., [69]. These approaches follow the same philosophy: before developing a model excerpt, the modeler needs to write the model test that will be used to evaluate the new functionality at the modeling level. Then this new functionality is modeled and the test is executed to ensure that the model is correct. This practice requires models to be executable (see Chapter 3). Chapter 10 discusses some tools that help in the exhaustive and systematic testing of models and/or in their validation.

5.6 SOFTWARE PRODUCT LINES AND MDSE

Software product lines [58] are becoming mainstream for the systematic development of similar systems. The main idea of software product lines is to design a family of systems instead of one single system. A family of systems defines a set of systems that share substantial commonalities but also exhibit explicit variations. In this respect, productivity and quality improvements achieved by software product lines originate from the reuse of so-called core assets, i.e., elements shared by all systems of a family. In a nutshell, software product lines can be seen as a course-grained software reuse mechanism which allows us to reuse the common parts but also manages variations between the systems of a family. In order to do so, software product line engineering makes a clear distinction between domain engineering, i.e., the development of reusable core assets within a domain, and application engineering, i.e., assembling of core assets for building the actual systems.

Model-driven engineering techniques have proven to be effective for supporting software product line engineering [38]. One of the most interesting question when it comes to software product lines and model-driven engineering is how to formulate the knowledge of a software product line by using models. Especially, the variability aspect is in this context of main interest as modeling languages are quite often lacking variability as a first-class language concept. In the past years, a plethora of variability modeling approaches have been presented to elevate this shortcoming. In general, two types of modeling approaches can be considered: annotative and compositional [38]. Annotative modeling approaches aim to represent all elements of a system family within one model description, and in addition explicit activation or deactivation steps are needed for representing a concrete system. In contrast, compositional modeling approaches are using a set of model fragments which have to be combined to represent a concrete system.

One of the most promising combinations of software product lines and model-driven engineering has been proposed within the scope of feature-oriented software development (FOSD). The main idea of FOSD is to decompose a family of software systems in terms of its features [5]. Features may be considered as visible aspects of the software system from a user's viewpoint or more generally as characteristics of the domain. From a set of features of a software system family, the systems can be generated by selecting certain features.

One approach to specify a feature-based decomposition of a system family is the usage of feature models. Feature models express the commonalities and variabilities among the systems of a software product line. These models organize the so-called features in a hierarchical structure. In this context, the basic relationships between a feature and its children are: mandatory relationships, optional relationships as well as XOR, OR, and AND groupings. In addition, cross-tree relationships may be introduced for describing inclusion/exclusion relationships of features residing in different parts of the feature model. By selecting certain features, a configuration or concrete system can be produced. As a feature model is a formal model in the sense that the relationship types clearly define which combinations of feature selections are valid or invalid, there is a systematic derivation of configurations supported. Furthermore, this supports the completion of a partial configuration, i.e., a partial selection of features which should be in a system, by reasoning which features are needed in addition to the selected ones to end up with a valid configuration.

An example feature model is shown in Figure 5.3. In particular, this model defines an abstract system consisting of different A-D features. Please note that there is a mandatory A feature which has to be selected for any system, but there is also an optional D feature which can be chosen independently of any other feature. Furthermore, for some features all children may be selected (OR) or only one of them may be selected (XOR). Finally, there is also a cross-tree constraint shown which states: if feature B1 is selected then feature C1 has to be selected as well.

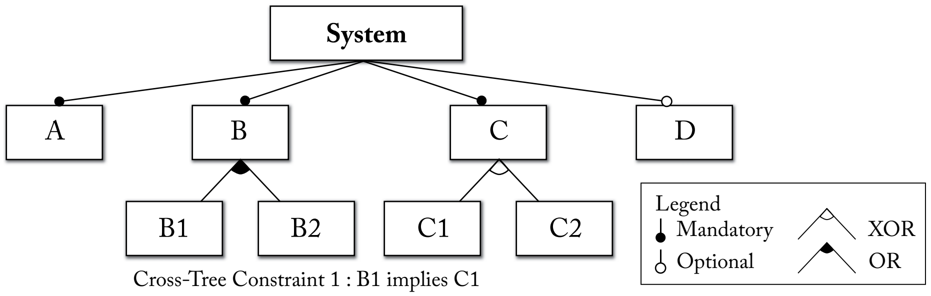

Figure 5.3: Example feature model.

Besides feature models, a plethora of other modeling languages for modeling software product lines have been proposed. In addition, several model-based techniques for the automated processing of variability models, including the automated analysis of variability models, derivation of configurations from the variability model of a software product line, and generation of test cases for derived products have been developed. Thus, the combination of model-based techniques and software product line engineering constitutes a promising reusability approach for software systems. Not surprisingly, this has been already explored in the past. For instance, Software Factories [30] combine domain-specific modeling and FOSD to exploit a structured collection of related software assets and processes that include reusable code components, documentation, and reference implementations to build up new applications.