

Chapter 1

Introducing Requirements-Driven Modelling

Requirements play a pivotal role in software development because they express the needs of the customer. A quality software system can emerge only when the real needs of the client are discovered. However, this is not enough. A typical software development project faces the problem of translating the user needs into a working system. These problems are dealt with by hundreds of books on various aspects of software design and pertaining to the plethora of software development technologies we can choose from. Related activities produce important artefacts that are treated as primary in software development: design models and code. Software design and coding directly contributes to the final system, and thus their results are treated as first-class citizens in the world of software development.

By contrast, requirements engineering is treated as a much less crisp and precise field of software development [33]. Requirements are treated as secondary artefacts for software developers as they cannot be translated directly into code. They are formulated as paragraphs of text structured to some extent, but still usually quite ambiguous and necessitating disambiguation during the later stages of development. They are obviously important but they do not contribute directly to the final effect. Their contribution is indirect and is treated like a craft rather than as a discipline of engineering. Various books on requirements engineering concentrate a lot on communication with the client and the psychological aspects of requirements elicitation (which is obviously good). This also includes notations (languages, templates, guidelines) with different levels of precision. However, there can be seen a lack of approaches to formulate requirements in a way that would allow for automation in their translation into code.

In this chapter, we introduce an approach to requirements engineering where requirements are treated as first-class citizens [70], contributing directly to the production of the final code. In this approach, requirements are formulated as models [17]. These models are intended to be comprehensible even by “ordinary” people (not software developers). At the same time, these models are formulated in a language that is precise enough (has precise semantics) to be able to generate meaningful and usable design models and code.

1.1 Why Model Requirements?

If we could try to imagine an ideal dream of a software project manager, it would most likely look as in Fig. 1.1. In this dream, requirements agreed-upon with the user and written down in whatever format, would automatically transform themselves into ready executable code of the target system [37, 148, 155, 179]. All that is needed is an automaton that would encapsulate all the knowledge on the target executable environment, logical and physical architecture, user interface design and so on. Moreover, this automaton would need to be able to process natural language and disambiguate it. Having such a tool, we would only need to press a button (“make a snap”) and voilà—here come the executables of the system.

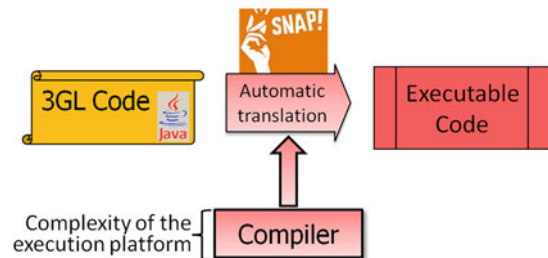
Of course, this is just a dream. . . Or maybe not? Maybe we can bring this dream closer to reality? First, let us take a look at Fig. 1.2. It shows a similar situation but at a much lower level of abstraction. Here we transform a program written in a 3rd Generation Language (3GL; like C++, Java or C#) into executable code. This is done by an automaton known to all programmers and is called the compiler [3]. This automaton encapsulates the knowledge about the hardware and the execution platform (machine code, bytecode, registers, memory access, basic I/O access and so on). During compilation, this knowledge is automatically “injected” into the executable program and merged with the programming constructs that could be expressed in the specific 3GL. The resulting code can be executed directly by the execution environment (in a specific operating system, processor type, virtual machine, etc.). The knowledge of this execution environment is to a large extent abstracted away in the 3GL code. As a result, the source code is significantly less complex (or: easier to comprehend) than the machine code.

We can argue that the two situations are different. The 3GL code has precise syntax and semantics (meaning) that determines its interpretation for execution (during runtime). By contrast, the requirements are too ambiguous. They do not use a syn-

Fig. 1.1 From requirements to code: ideal scenario



Fig. 1.2 From 3GL code to executable code



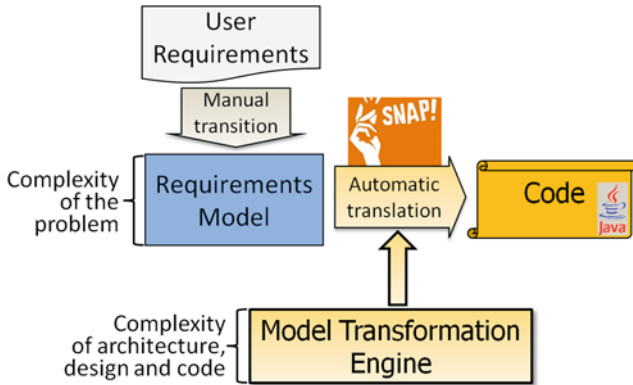


Fig. 1.3 From requirements to code: a more realistic scenario

tactically and semantically precise language. Thus, it is not possible to develop an automaton that would be able to process requirements and produce executable code. Obviously, this is true for requirements written in natural language and using semi-formal diagrams, even when certain templates and constraints on the requirements structure are applied. However, we can think of a much more realistic scenario, as illustrated in Fig. 1.3.

In this scenario, the less structured user requirements are manually transformed into syntactically and semantically precise requirements models. These models capture the complexity of the problem using a high-level modelling language. This includes the desired functionality of the system (application logic, user interface) and a detailed description of the problem domain. The main issue here is to be able to capture all such knowledge using this high-level language with “enough” precision. Provided that the precision is satisfactory, we will be able to develop a model transformation engine (cf. a compiler) that produces a full code of the desired system. This engine, by analogy to the compiler, would encapsulate the technological details of the target programming platform. This includes the target architectural solutions, detailed design decisions and intricacies of specific coding guidelines. Such technological knowledge is injected during translation and is combined with the description of the problem (the requirements models) to produce fully operational code.

This last scenario (see again Fig. 1.3) refers to a clear separation of two kinds of software complexity formulated by Fred Brooks [26, 27]. Back in the 1980s, Fred Brooks distinguished the “essence” from “accidents” in programming. His definition was that “the **essence** of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions” and “**accidental** tasks arise in representing the construct in language”. In our case, the essential complexity is expressed through the requirements models. These models contain the definition of the problem domain (data and relations) and the desired

functionality (application and domain logic algorithms with their invocations). The accidental complexity is expressed through the model transformation engine and contains all the details of the target software and hardware platform.

The distinction made by Fred Brooks gives us a hint of the direction in which we should move in our approach to software development. We should hide as much accidental complexity as possible and promote the essential complexity. Compilers (Fig. 1.2) indeed hide some of the accidental complexity. In the later parts of this book, we provide an introduction to how to tackle the problem of hiding the remaining “accidents” through constructing model transformation engines (Fig. 1.3). To develop such an engine seems a much more complex task than to develop a compiler. However, these two tasks have much in common. In both cases, we need a source language which has a formal syntax. We also need translation rules that are based on the semantics of the source language. This semantics defines how to express the (high level) constructs of the source language through the (low level) constructs of the target language.

3GL compilers allow us to express programming constructs at a significantly higher level than machine/assembly code. However, this is done at a cost of less programming freedom and usually worse performance. This is because the compilers apply uniform translation rules to all the 3GL constructs and produce uniformly structured and often not perfectly optimised output code. All the detailed decisions on how to structure the output code were made by the compiler constructors. Despite this, we do not really want to go back to the era of assembly language programming. The advantage of abstracting away the complexity of the execution platform is much higher than the minor disadvantages that were mentioned. Though, in some cases, whenever it is necessary (e.g. for performance reasons), some subroutines can be programmed in assembly language.

Still, we can try to move up the ladder of abstraction in software development. We would like to define a language that abstracts away not only the execution platform but the whole software technology platform. This includes the architectural design (physical and logical architecture), approaches to persist data, ways to exchange data through the user interface, divisions into programming units (packages, classes), approaches to pass control between programming units (class dependencies) and so on. The appropriate translator for this language would need to capture and unify specific decisions in these areas. Thus, the developers that would use the new high-level language would have a lot less freedom in making these decisions than when they would program in Java or other 3GL. This would be the cost of “programming” at the level of the problem domain (or: requirements [31, 69, 150]). The question would be whether this cost would pay off just like in the case of 3GL to machine code translators. Another question is how to develop such new translators and this is what we want to answer in this book.

In the meanwhile, the reader would probably appreciate some more concrete justification on why we should bother about modelling requirements in a semantically precise way. Let us illustrate what we mean, through an elementary example. In Fig. 1.4, we can see a tiny requirements model. This model uses a notation that should be familiar to many requirements engineers. It contains a single use case

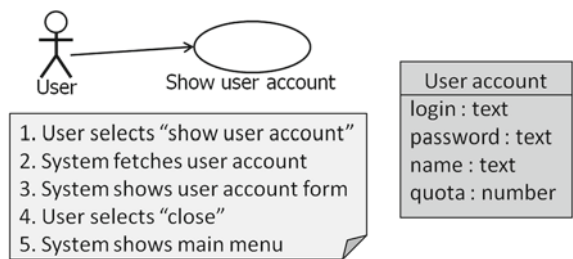


Fig. 1.4 Tiny example: source requirements model

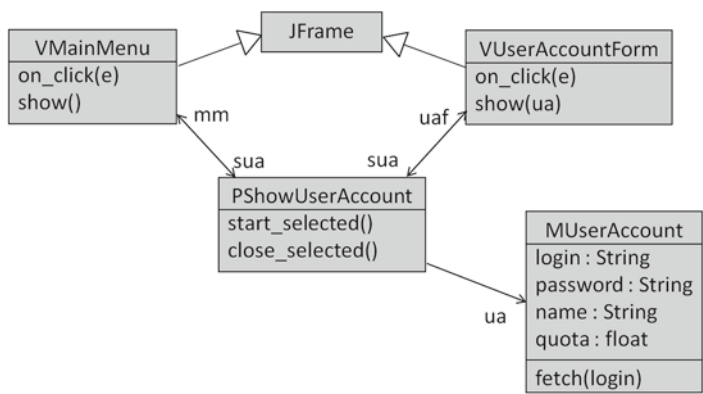


Fig. 1.5 Tiny example: target code structure model

(the oval) and a single actor (the sticky figure). This simple notation was invented by Ivar Jacobson and dates back to the beginning of the 1990s [2, 12, 34, 44, 78, 96, 137, 141]. The use case diagram is supplemented by a textual scenario (sentences 1–5) and a single definition of the domain notion (“user account”) used in this scenario. The notion definition uses standard UML class notation [24, 52] and contains four attributes (information components) of the notion.

The diagram in Fig. 1.4 is a requirements model that we claim is precise enough to generate working code. Obviously, this generated code should be governed by certain design decisions that would be uniformly and automatically applied. Here we do not explain these rules in detail, instead we describe code that we would like to have as an implementation of the source requirements model. The structure of this code is presented in Fig. 1.5. This is a UML class diagram that depicts four classes with their operations and relationships between them. A careful reader might notice right away that we have applied an architectural pattern to this code structure.¹ It is the Model-View-Presenter (MVP) pattern that is becoming more and more popular

¹ The examples in this chapter use very simplified Java with an imaginary programming framework. Here we want to abstract away from any specific Java technology.

in contemporary programming frameworks. This pattern consists of three layers. The view layer (classes starting with a “V”) contains code responsible for exchanging data and commands with the user through the user interface. The classes in this layer specialise in more general window frames. The presenter layer (classes starting with a “P”) handles the application logic in terms of sequences of events happening in the dialogue between the user and the system (including internal actions of the system). The model layer (classes starting with an “M”) handles the domain logic that includes data processing algorithms and storing (persisting) the data.

The code structure in Fig. 1.5 should be substantial enough to handle the simple functionality defined in Fig. 1.4. Moreover, we may intuitively feel that there can be specific rules determined for obtaining this code structure from the requirements model. There can be seen specific traces from the elements of the requirements model to the elements of the code model. For instance, the notion of “user account” is reflected in two classes in code: VUserAccountForm and MUserAccount.

Of course, the structure alone is not enough to implement the functionality (scenario) from Fig. 1.4. We need some dynamic code. This dynamics can be presented using a UML sequence diagram as in Fig. 1.6. This diagram is a translation of the scenario into a sequence of messages passed between objects in a running system. These messages are equivalent to calling class methods either synchronously or asynchronously. The current diagram reflects design decisions on how to structure

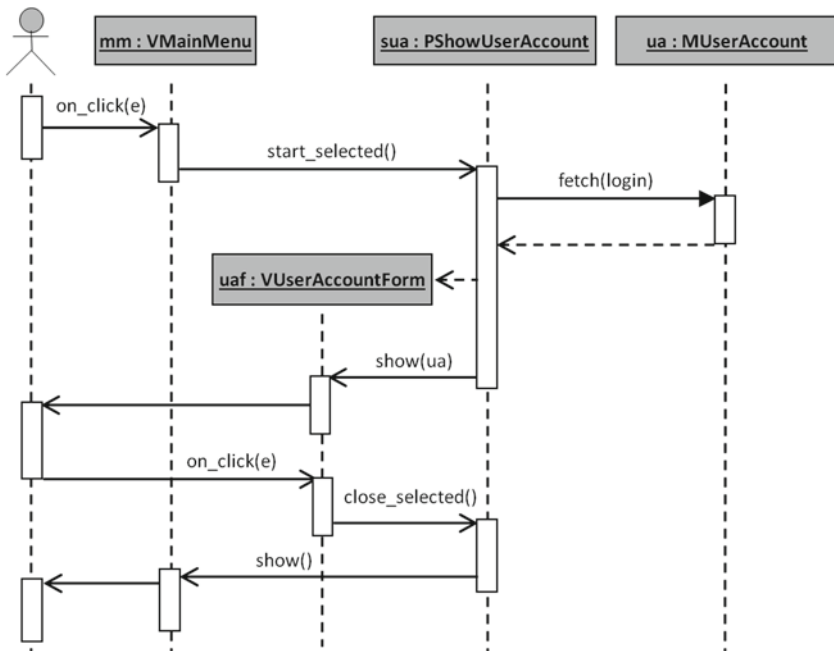


Fig. 1.6 Tiny example: target code dynamics model

code within individual methods in code. For instance, the method “start_selected” in PShowUserAccount contains a call to “fetch”, a constructor call for VUserAccountForm and a call to “show”. Similarly to the case of the code structure, we can intuitively see certain traces from the sequence of sentences in our example use case scenario to the sequence of messages in the sequence diagram. For instance, the sentence “User selects ‘show user account’” can be traced to the sequence of two messages: “on_click” and “start_selected”.

The class diagram from Fig. 1.5 and the sequence diagram from Fig. 1.6 already contain much of the target complexity. However, the final code has more details, as presented in Fig. 1.7. The presenter (“PShowUserAccount”) code is a direct translation of the dynamics from the sequence diagram. However, the other class methods contain code that adds more details. The view classes have specific code to show individual widgets on the screen (here: a very simplified widget rendering framework). The model class contains code to fetch persisted data from the database (here: a very simplified inline SQL). In reality, this code would be significantly more complex but we have removed much of the technical details for the sake of simplifying this example. Despite this simplification, it can be seen that at this stage, the target platform (technology) details were introduced. However, the individual instructions in code can be intuitively traced back to the initial requirements model.

In the remainder of this book, we show that it is possible to automate the path sketched in this simple example. The main prerequisite for this automation is the ability to define the source requirements models precisely. This means using precise syntax that allows to assure the coherence of the models similar to the coherence of code. Moreover, we need a precise definition of the semantics that would explain the

```
class VMainMenu extends JFrame{
    PShowUserAccount sua;
    void on_click(Event e) {
        sua.start_selected();
    }
    void show() {
        show_option("Show User Account");
    }
}
```

```
class VUserAccountForm extends JFrame{
    PShowUserAccount sua;
    void on_click(Event e) {
        sua.close_selected();
    }
    void show(ua: MUserAccount) {
        show_text(ua.login);
        show_text(ua.name);
        show_number(ua.quota);
    }
}
```

```
class PShowUserAccount {
    VMainMenu mm; VUserAccountForm uaf;
    MUserAccount ua;
    String login;
    void start_selected() {
        ua.fetch(login);
        uaf.show(ua);
    }
    void close_selected() {
        mm.show();
    }
}
```

```
class MUserAccount {
    String login, password, name;
    float quota;
    void fetch(String l) {
        "SELECT * FROM user_accounts
        WHERE login=l"
    }
}
```

Fig. 1.7 Tiny example: target code

translation of requirements elements into specific code constructs. Assuring strict precision of requirements involves additional effort in this phase of software development. However, this effort should pay off with the possibility to apply automatic translation into fully working code. In the above example, we have not considered many other aspects of this translation which are explained in detail in the subsequent chapters. Thus, although considerable effort is needed to develop the transformation, this transformation can be reused many times similar to how we “reuse” transformations into machine code within compilers.

So, how do we answer the question “why model requirements?” There can be at least two reasons (see Fig. 1.8). First, requirements models have very good communication capabilities. Using visual models, we can make requirements more comprehensible to the business people who order and use software. Visual models can thus be used by less formal human readers. The second reason is more important, as its effects can have a significant impact on the productivity of the developers. Requirements models can be made formal enough to be able to transform them automatically into other artefacts like design models and code. Also, we can assure their coherence through implementing certain automated validation mechanisms. The possibility to generate code directly from requirements means a significant rise in abstraction for activities that produce running code. This rise of abstraction is similar to that of 3GL programming in relation to assembly language programming.

However, a significant problem we face is to design a language that is comprehensible to “ordinary people” and at the same time gives enough “power” to serve as high-level code for software developers. This language must have a precise but understandable syntax (grammar) and its constructs must possess strict meaning (semantics) in terms of code generated from them. The following two sections provide an overview of these two aspects.

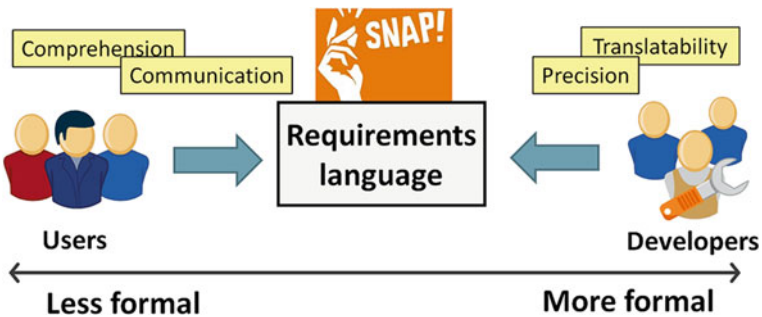


Fig. 1.8 Combining good communication with precision

1.2 Making Requirements Precise

1.2.1 Writing Good Stories

To answer the question of how to make requirements precise we start with an analogy that may look odd at first sight. We start by explaining how to write a good adventure novel. First, we have to have a good story. It should have an exciting action with many possible resolutions. Preferably, the story should end with a happy ending but there always has to be a possibility of the story ending sadly. Moreover, we should place this action in an interesting environment. This should include the characters that take part in the story, placed within the nature (forests, animals, etc.) and the products of technology (buildings, vehicles, etc.) that surround them. The best novel writers try to create a coherent new (future, alternative, sci-fi) environment, or try to reflect the real environment existing some time in history or at the present.

One of the best examples of a coherent environment created by a talented writer is the Middle-earth. J.R.R. Tolkien has described it in much detail throughout several of his works like “Hobbit”, “The Lord of the Rings” and “Silmarillion”. The environment of the Middle-earth consists of many different intelligent species (men, hobbits, dwarfs and so on), a specific landscape with different landmarks (cities, mountains, rivers), specific technical capabilities (weapons, means of transport, etc.) and other features of the characters like their extra-natural capabilities. Being a good novel writer, Tolkien gradually reveals to us the whole environment of the Middle-earth. This is done along with the different stories being told. A good novel is thus a balanced combination of stories and the description of a coherent environment. This is illustrated in Fig. 1.9 where the descriptions of various events (forming the story) are interweaved with the descriptions of the environment.

In order to be able to understand the environment of the Middle-earth better, Tolkien provides us also with a sketch of its map. However, his text is so precise and

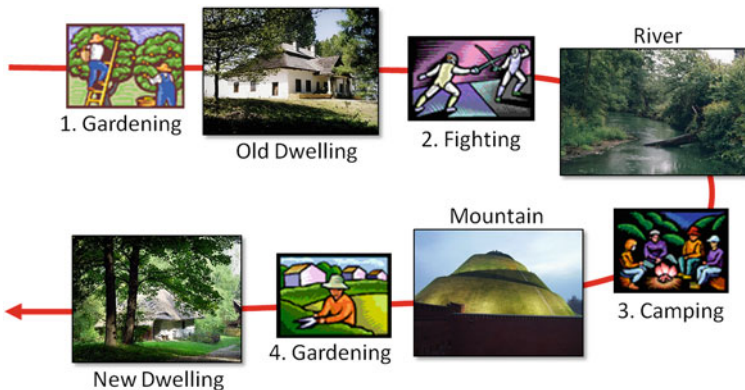


Fig. 1.9 Story and environment combined in a novel

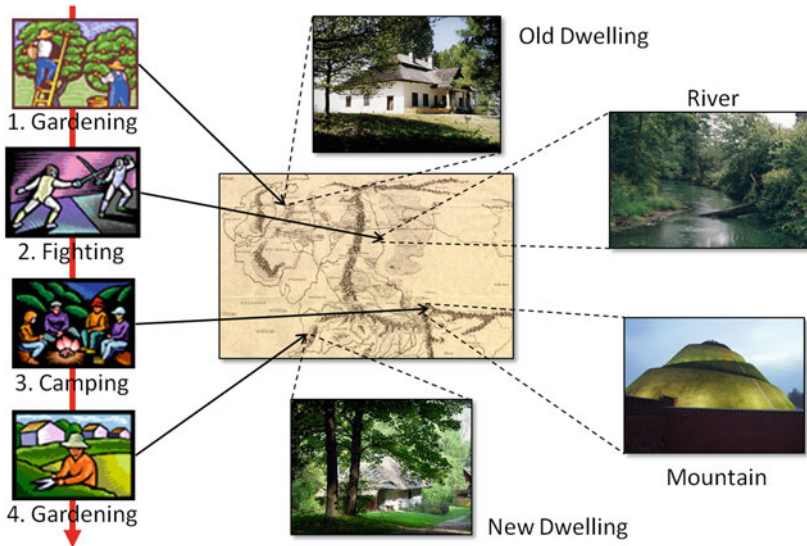


Fig. 1.10 Making the environment coherent with the story

coherent that many other authors have developed various other descriptions of the Middle-earth. This includes atlases with detailed maps of various areas of Middle-earth, encyclopaedias explaining all the notions, and vocabularies explaining all the terms in various Middle-earth languages. In fact, from Tolkien's various works on the Middle-earth, there has been "extracted" a coherent (quasi-historical) description of the whole environment. This can be generalised in that a good adventure novel should provide an environment that can be described with a coherent conceptual framework, e.g. using a map. An illustration of this can be found in Fig. 1.10. Now, all the events from Fig. 1.9 are extracted to form the story. The events refer to specific places on a coherent map of the territory. Each of the places is described in a vocabulary of places (rivers, mountains, dwellings, ...). What is also important is that different events happening at the same place should be positioned correctly in relation to other events.

1.2.2 Writing Stories About Software

After this short digression on adventure stories, let us get back to dull business systems. With the above example, it is possible to write a story that is understandable and at the same time—coherent and precise [147, 181]. Stories themselves are understandable because this is the most fundamental way in which people have been communicating for ages. Furthermore, stories about reality (testimonies, history reports) can be verified by cross-checking their coherence with the environment (domain). Thus, a natural choice for making requirements for business systems understandable

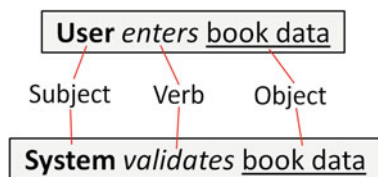


Fig. 1.11 Sentences in a story

and precise would be to base them on stories. How can we write such stories for software [7, 147]?

We can write them from the point of view of the system’s user. Then, the story should tell about the dialogue between the user and the system. The story can consist of simple sentences, as those shown in Fig. 1.11 [20, 61]. These sentences are indicative and contain a subject, a verb and an object. We call them Subject-Verb-Object (SVO) sentences. The SVO structure should be sufficient to present the interactions between the users (when the subject is a user) and the system (when the subject is the system). However, it is obviously not sufficient to explain the context for the interactions. Thus, requirements specifiers often tend to insert such explanations as continuations of sentences with such (or similar) structure. For instance, we receive a complex sentence like, “The user enters book data, where book data contains the book title and the author”. This can be compared to writing an adventure novel where the story is interweaved with the descriptions of the environment. In another part of the story (e.g. 20 pages later in the requirements document), we can have a sentence like, “The system validates book data (author, title, issue date) by comparing the issue date with the author’s lifespan”. This leads to significant confusion when we want to define “book data”. Both definitions do not match—is it the same kind of data we are talking about in both cases? To make things coherent, we need to use the same technique as Tolkien and his followers did—create a detailed map of the territory, coherent with the story.

Note that we have clearly emphasised the three sentence parts—the subject is in bold, the verb is in italic and the object is underlined. This emphasis is important, because we want to relate the verbs and the nouns to their centralised definitions. This will make the “environment” coherent with the “story” as in our discussion on the Middle-earth, as illustrated in Fig. 1.12. It can be seen that both sentence objects

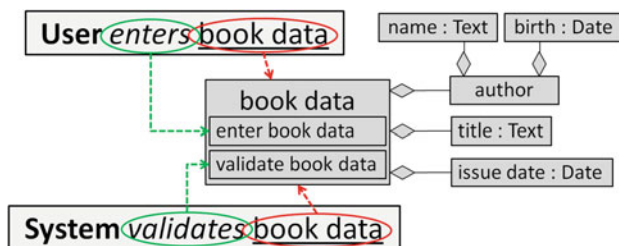


Fig. 1.12 Extending story sentences with domain definitions

refer to one centrally defined domain element (here: “book data”). Moreover, the sentence verbs refer to appropriate domain statements (here: “enter book data” and “validate book data”) contained in the domain element (verb phrases). The domain element definition is complemented by aggregated attributes (here: “author”, “title” and “issue date”). The references from the SVO sentences can be seen as hyperlinks [82] leading to a central “wiki” definition. We should note that this “wiki” contains definitions of not only the nouns (“book data”) but also the verb phrases associated with the nouns (“validate book data”).

Several SVO sentences can be formed into a story which we can also call a scenario. An example can be found in Fig. 1.13. There we find in fact two scenarios. One of them ends with a “happy end” (the book data are saved) while the other one ends with a failure (an error message is shown). Both scenarios have the same beginning (sentences 1–5) and what distinguishes them are the condition sentences (book data either valid or invalid). These condition sentences can be compared to the dilemma of a scenario writer for a TV “soap opera” (or an adventure novel writer). She might wonder how to resolve a specific key scene in an episode. Depending on this resolution, the plot might go into several different directions. The issue for software system scenarios is that we need to specify all the possible “plots”...

When writing scenarios, we maintain their coherence with the domain definition [10, 20–22, 152, 165]. This definition grows through adding new notions and new verb phrases. The notions are taken not only from the problem domain (like “book data”) but also from the domain of the user-system dialogue [177], which includes window frames (see «frame» in Fig. 1.13), button triggers (see «trigger») and message boxes (see «message»). This forms a vocabulary of notions to be used coherently within

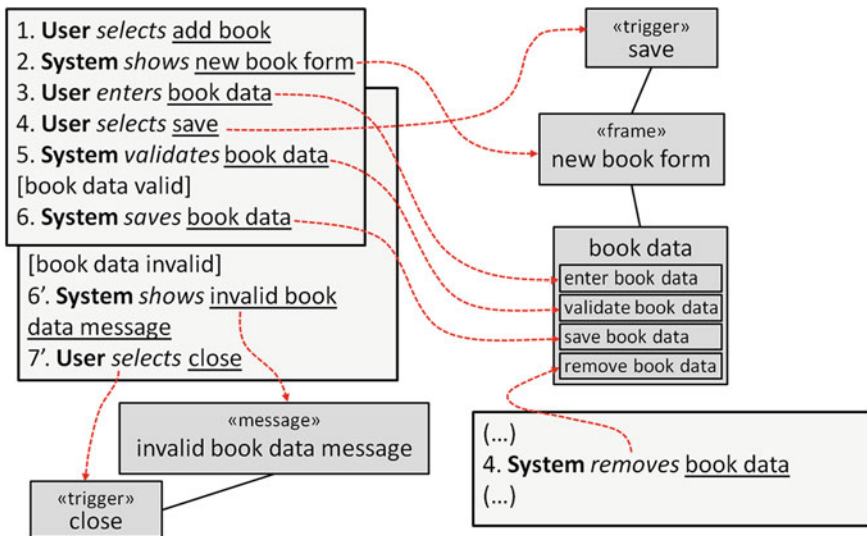


Fig. 1.13 The stories and their vocabulary

various scenarios throughout the whole requirements model. This is illustrated in Fig. 1.13 by showing another (perhaps distant) scenario (at the bottom) which also refers to the same domain notion as that referred to from the first two scenarios. We can associate the various notions through relationships which makes also the vocabulary internally coherent. For instance, we can indicate which of the domain data elements (here: “book data”) should be presented in which of the window frames (here: “new book window”).

Several similar scenarios can be combined to form a use case. Use cases were introduced earlier in an example in the previous section but here we present them more formally. There can be found many definitions of what use cases are in the literature. Our definition tries to summarise them and concentrates on three main features of use cases.

Definition A use case is a complete piece of functionality that possesses the following characteristics:

1. It starts with the interaction of an outside actor with the system or assumes the possibility of such an interaction to start it.²
2. It contains several scenarios that constitute sequences of interactions of an outside actor (or actors) with the system, and replies (or requests) of the system.
3. It ends by reaching a goal of some value to the outside actor or failing to reach that goal (despite trying).

Use cases are defined in relation to outside actors. Outside actors represent roles that groups of people or outside systems play in relation to the currently considered (specified) system. A representative of an outside actor can come into interaction with the current system in accordance with the use cases related to this outside actor. Note that the definition speaks of “outside actors” and not simply “actors” (as is done in the literature and software engineering practice). This is due to certain confusion that can occur for less-experienced modellers. They sometimes model actors as internal elements within the modelled system (or even as the system itself). The word “outside” makes it unambiguous that actors are never part of the current system.

This definition can be analysed through the example found in Fig. 1.13. We did not name the use case in the figure, but it is obvious that the name could read as “Add new book”. This name also specifies its goal. The use case (all of its possible scenarios) starts with a specific interaction (selecting the “add book” button). It contains several SVO sentences forming a sequence of interactions between the user (cf. outside actor) and the system. It ends either by reaching the goal (saving “book data”) or failing to reach it (presenting an error message). Note that after reaching the goal, the system is in a stable state from the point of view of the user interface and internal system transactions. This means that—for instance—the user can start the use case again.

² The assumption of possible initial actor interaction is explained in Chaps. 2 and 4. This pertains to use cases invoked from within other use cases (see Fig. 2.27 and rules P9 and P10).

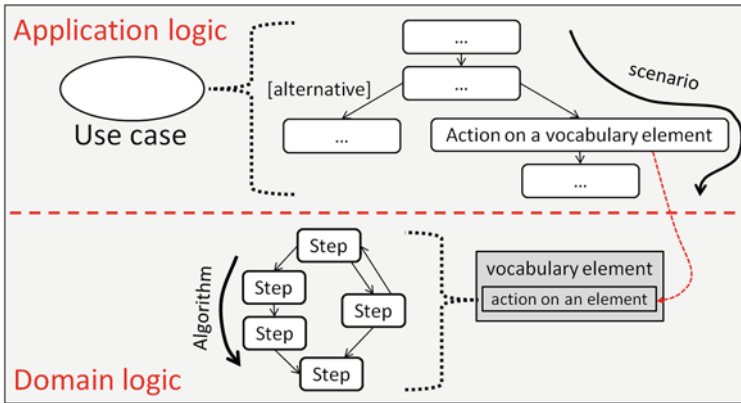


Fig. 1.14 Use cases: stories forming the application logic that refers to domain logic

The functionality defined through use case scenarios can be also called the application logic. This is in contrast to the domain logic illustrated in Fig. 1.14. The application logic contains all the possible alternative scenarios of user–system interaction. Actions in the application logic refer (“hyperlink”) to specific actions in the domain logic. The domain logic is organised around the vocabulary of notions as presented above. The verb phrases (actions) that are contained in the notions can be defined in more detail to form the domain logic. This includes specific notations to denote data processing algorithms (e.g. how to “validate book data” or how to “calculate mean book price”). We do not discuss these notations in detail here because they are specific to the considered domains (cf. domain-specific notations mentioned in the next section).

It should be emphasised that the presented notation for the requirements models is domain agnostic. This means that it can be applied to any typical business domain, thus making it universal. The domain vocabulary is constructed within the specification and is then used to define the domain logic [47]. The notation is simple and understandable by a “layman” (not proficient in software engineering, e.g. a business person). It consists mainly of simple SVO sentences and domain element definitions. Its main characteristic is strict relation between the application logic (use case scenarios) and the domain logic (verb phrases). We call this notation the Requirements Specification Language (RSL) [83, 118].³ This language allows us to implement the “realistic” code translation scenario described in the previous section. For this purpose, we need to define the meaning of RSL constructs in terms of application logic and equivalent code. This is introduced in the next section.

³ The idea of requirements modelling started with the Requirements Modelling Language proposed by Greenspan et al. [63, 64]. More recently, Helming et al. proposed the Unified Requirements Modelling Language [16, 68]. Yet another approach was proposed by Beatty and Chen [13].

1.2.3 How About Quality Issues?

Note that use cases and the vocabulary identify two types of requirements: functional requirements and vocabulary requirements (domain definitions). For a requirements specification to be complete, we need to define quality requirements (known also as non-functional requirements). The first two types of requirements define the way the system functions, as seen by the outside user and defines what data are processed and exchanged with the user. The quality requirements influence the internals of the system. Architects use specific design solutions based on the required performance, reliability or security. Moreover, they might need to apply certain constraints to the target technology (e.g. a specific operating system or UI framework has to be used).

Quality requirements may highly influence the “accidental” aspects of the target code. Thus, they need to be carefully considered when developing translation rules for functional and vocabulary requirements. For some of the quality requirement types, we can determine precise guidelines as to which architectural patterns or specific software development frameworks should be applied. For instance, the requirement that the application should be used on mobile devices determines many elements of its architecture and various detailed design solutions. In case of other types, some general guidance can be applied, and only later verified when some parts of the system are ready. This is typically the case for performance requirements. Experienced architects can use specific design solutions that are known to fulfil the desired system performance (e.g. response time). However, there is no way to assure this in advance at design time.

In this book, we assume the average quality requirements that influence the translation from requirements models to code. In particular, we assume that the system is available through the Internet and thus a web application is needed. We also assume that the response time can be average, and typical architectural solutions for web application will be satisfactory. In case there are special quality constraints set on the target system, the translation rules presented throughout this book should be updated. This may, for instance, influence the code structure and its distribution between processing nodes (physical/virtual machines). It may also involve developing code for different UI platforms or using a certain security framework.

In general, this may be reflected in several variants of translation programs that produce code from requirements, selectable depending on the specific quality requirements. We may also think about automating this process, and selecting appropriate transformation procedures depending on the values of the quality requirements. For this purpose, we would need a precise modelling language for quality metrics and a schema for parameterising code generation depending on these metrics. This topic is still well beyond the current state-of-the-art and thus we will not elaborate on it in this book. However, it seems a promising direction worth a significant research effort in the future.

1.3 What Is the Meaning of Requirements Models?

Being a language (and specifically—a modelling language), RSL communicates some meaning. To understand this meaning we need to define its semantics. Semantics is a necessary complement of the indexsyntax syntax (grammar) for any language (including natural languages). To explain the semantics of a software language, we can use various methods. In all of these methods, we need to specify the given language in terms of simple concepts which have a known meaning (semantics). For instance, to explain the semantics of a programming language, we can introduce a simple automaton (with memory, processing capabilities, etc.). Then we can show how specific constructs of the given language work during runtime in terms of operations of this automaton (called operational semantics). However, this approach—although precise and formal—is usually hard to understand and use in practice.

A practical way of defining the semantics of a language is to specify the rules of translating specifications (programs) in this language into specifications (programs) in another language. Obviously, this other language has to have its semantics already defined. This way of defining semantics is called translational semantics [91, 146]. We use this approach to specify the semantics of RSL in Chap. 4. We offer rules for translating RSL into Java (also explaining some constructs using UML). This is helpful for constructing language translators. At the same time, it allows to understand the meaning of specific language constructs by software engineers proficient in 3GL programming.

However, it is also important to explain the meaning of RSL to domain experts not proficient in software development. In this case, we should provide some account on RSL constructs in terms of the behaviour of the specified systems as seen from the outside. This would include the navigation and appearance of the user interface and changes in the system state related to the domain (business) model. Again, these semantics can be defined through some target language understandable to the reader of the semantics definition (translational semantics). Yet, this definition might not necessitate the level of precision as that for translating into, e.g. Java.

1.3.1 Requirements Explained Through Observable Behaviour

In this introduction to semantics, we start with the approach based on defining the system's observable behaviour. It allows us to be less formal. We explain use cases, scenarios and vocabulary elements by translating them into sequences of user interface elements that would appear to the user and changes in domain objects handled by the system. A summary of this approach can be found in Fig. 1.15. It consists of three parts: (1) control flow semantics, (2) individual sentence semantics and (3) vocabulary element semantics. The first two parts pertain to the application logic of the system to be built. The third pertains to the domain logic (see the previous section for explanation of the distinction between application logic and domain logic). Control

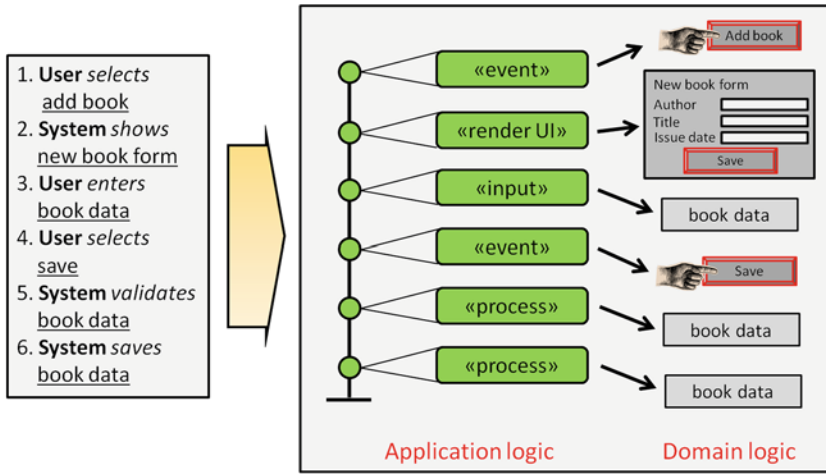


Fig. 1.15 Software language semantics explained through observable behaviour

flow semantics defines the sequence in which the individual interactions are to be executed. In Fig. 1.15, it is depicted with a series of dots along a line, ending with a “stop bar”. In our simple example, we have only one straight line because there are no alternatives: we have a single scenario. Each of the dots represents a single action equivalent to a single sentence in a scenario. Each of these sentences has a specific meaning, which can be a user-triggered event, a UI rendering action or a data processing action. Detailed semantics of each of the actions can be determined by examining the vocabulary elements linked from the associated scenario sentences. These details include, e.g. the specific button to be pressed, the layout of a window frame to be displayed or the specific data processing algorithm to be executed.

The semantics of use cases and especially their control flow semantics was not the first concern of the literature on this subject [143, 170]. This is because use cases were not intended for automatic translation into design and code. However, for our purposes we need to be clear about this. The “use case language” has to define precisely how a set of use cases may be “executed” in a running system. This is illustrated in significant detail in Fig. 1.16. We can see two use cases related through a relationship denoted with «invoke». This is complemented by two scenarios of the first of the two use cases (“Show book list”). The second use case is the already presented “Add new book”. Note that we do not use the well-known «include» and «extend» relationships because of their ambiguous control flow semantics, criticised in the literature [57, 97, 108–110, 143]. Instead, we use a new kind of relationship for which we introduce precise control flow semantics.

The control flow of this RSL model is explained through the diagram on the right of Fig. 1.16. It shows how the individual scenario actions can be executed. There are two alternative paths for “Show book list” starting after sentence 3 (see the left part of the diagram). Two SVO sentences denote two alternative decisions by the actor—to

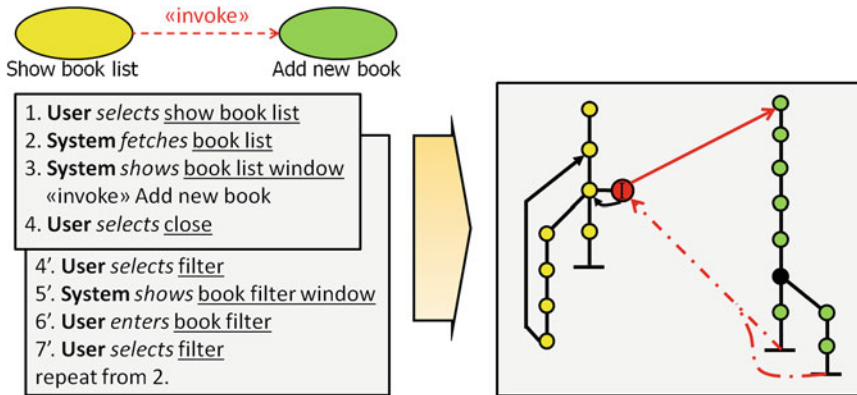


Fig. 1.16 Control flow semantics of use cases

either select “close” or “filter”. In “Add new book”, there are also two possible paths (compare with Fig. 1.13) but the alternative is of a different kind. This time it does not depend on the user’s decision but on the system’s state and data processing. This is denoted by the black dot in the right part of the control flow diagram. The dot can be expanded into several (here: two) alternative conditions (here: data are valid or invalid) leading to selection of one of the outgoing paths.

The control flow diagram in Fig. 1.16 also presents the semantics of «invoke». An invocation relationship exists between two use cases whenever an appropriate «invoke» sentence is present in a scenario. In our example, “Add new book” is invoked within sentence 3 of “Show new book”. The control flow semantics of this RSL construct resembles a procedure call. Control is first passed to the first sentence of the invoked use case. After reaching one of the final sentences of the invoked use case, control is passed back to the invoking use case. However, the return of control repeats the action associated with the invocation sentence. In our example, the action to “show book list window” is performed after returning from “add new book”.

Having explained the control flow, we need to explain the meaning of individual actions, associated with SVO sentences. In general, we can distinguish four types of sentences as presented in Fig. 1.17. Their general meaning depends on their subject and their object. Syntactically, the subject of an SVO sentence can be either one of the outside actors or the system. Moreover, the object of an SVO sentence can be a trigger or a domain element. Based on this division, we can define the four types of SVO sentences.

1. **User-to-system-event.** These sentences denote interactions of an actor that passes control to the system. This includes selecting menu options, pressing buttons or other such events.

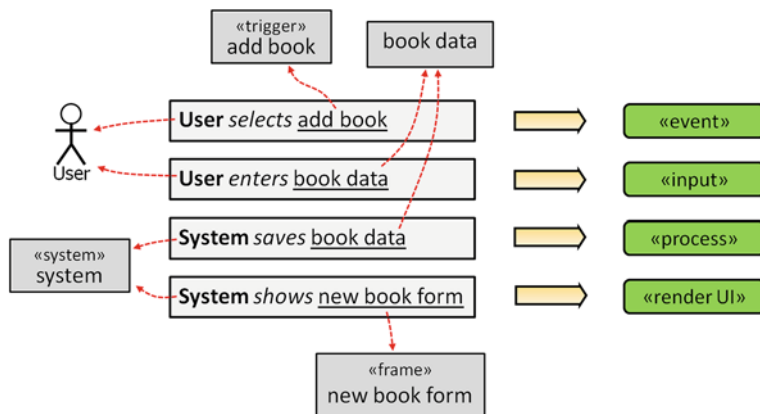


Fig. 1.17 SVO sentence semantics

2. **User-to-system-input.** These sentences denote passing data from the user to the system. These data are compliant with some domain element and its components (attributes).
3. **System-to-user.** These sentences denote presenting some message or data to the user. Most often, these data are rendered through some graphical user interface. These data can be read-only or editable in some part (subject to a further user-to-system-input sentence).
4. **System-to-system.** These sentences denote performing some processing related to specific domain elements that may include performing calculations, changing system state or retrieving persistent data.

The exact meaning of a specific SVO sentence depends on the actual vocabulary elements that are linked from this sentence. This is illustrated for system-to-user sentences in Fig. 1.18. This example shows the effect of executing “System shows new book form”, having a specific definition of “new book form”. This «frame» has an associated domain element (“book data”) and a «trigger» (“save”). These two elements are rendered on the window frame as shown on the right.

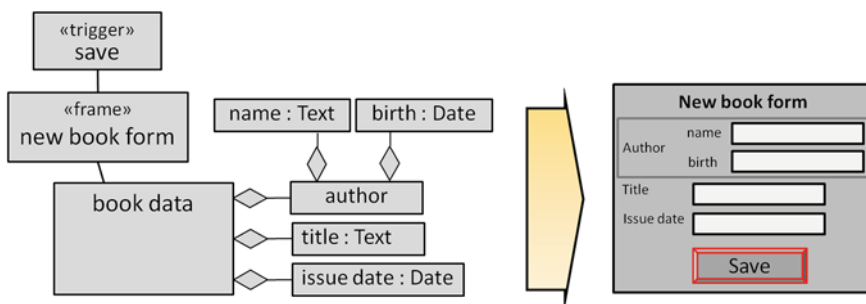


Fig. 1.18 Vocabulary element semantics

Two other types of SVO sentences are quite easy to explain without a separate illustration. A user-to-system-event sentence is a reaction to interacting with the specific «trigger» (e.g. “save” in our example) rendered on a window frame in a preceding system-to-user sentence. In turn, a user-to-system-input sentence denotes editing the fields rendered on a window frame, based on the associated domain element (e.g. “book data”). Care should be taken to assure coherence of user-to-system sentences with the vocabulary. The previously rendered window frame must have the necessary elements available for editing or triggering.

Finally, the system-to-system sentences have their meaning dependent on the actual contents of appropriate verb phases. The contents should specify appropriate algorithms for processing, accessing and changing the state of data within the system. This is the domain-specific part of a requirements model. For this purpose, a domain-specific language can be developed or some general-purpose language with known semantics can be used. In case of a lack of such language, we are left with specifying the domain logic algorithms informally and leave them out of the automatic transformation path. This is a very broad topic of domain-specific languages and is out of the scope of this book. However, all the techniques for defining software modelling languages that we describe can be used also for such domain-specific languages.

1.3.2 Requirements Explained Through Translation into Java

So far, we have used elements observable by the user to explain the meaning of requirements models. Now let us get back to the idea of defining requirements semantics by offering equivalent 3GL (Java) code. This approach is fully formal and can be used directly to construct automatic transformation engines. In this introductory section we present the basics of the approach, and the details are given in Chap. 4.

The first step is to define the translational framework, illustrated in Fig. 1.19. The translational framework assumes a certain code structure to which the requirements

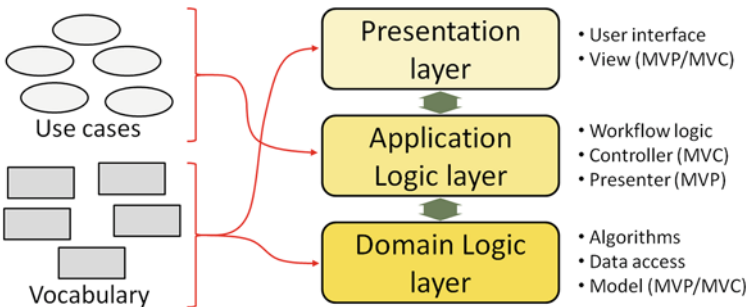


Fig. 1.19 General translational framework for RSL

models are translated. Here we choose a standard three-layered architecture consisting of Presentation, Application Logic and Domain Logic. This division is equivalent to the architectural patterns of Model-View-Controller (MVC), Model-View-Presenter (MVP), etc. This approach was earlier informally introduced in Sect. 1.1. The presentation layer is responsible for accepting and presenting data to the user. The application logic layer contains code that pertains to the workflow of the application which includes different flows of user-system interaction. The domain logic contains code that does the actual data processing (algorithms) and accesses persistent data.

Figure 1.19 shows that use cases with their scenarios are translated mostly into the code within the application logic layer. This is obvious in the face of what we have already explained about the semantics of use cases. Use case scenarios can serve as good controllers of the application logic. In turn, the vocabulary requirements influence mostly the other two layers. This is also obvious from the previous descriptions of their semantics. The domain logic layer reflects the realisation of verb phases with the associated algorithms. These algorithms process data as defined by domain elements and their attributes. The presentation layer reflects the visualisation of the same data to the user. The data are shown within specific UI elements that are also defined in the vocabulary and relate to the domain elements.

Figure 1.19 provides an informal overview of the translation; more details are required. A summary of several formal translation rules is presented in Fig. 1.20. Here we use a UML class diagram to present the code structure. Moreover, the class operation methods are expanded to show their code (in this example—only one of the methods). The arrows with numbers show how RSL constructs are translated into code constructs.

1. Use cases are translated into classes of the Presenter (application logic) layer.
2. User-to-system-trigger sentences are translated into operations of the Presenter class translated from the current use case.

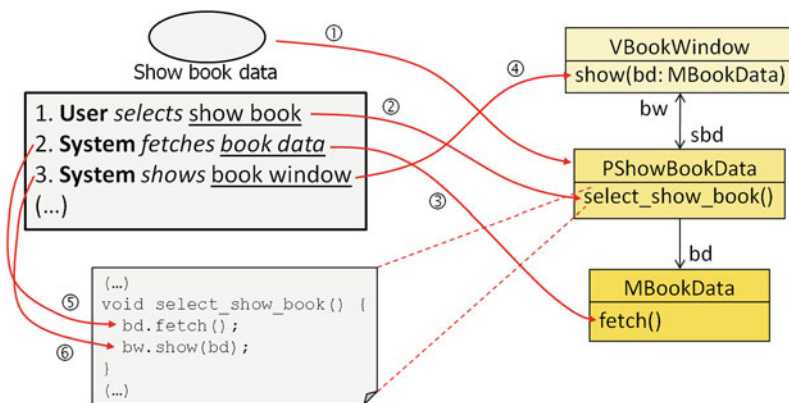


Fig. 1.20 Use case translational semantics

3. System-to-system sentences are translated into operations of the Model (domain logic) class associated with the given domain element.
4. System-to-user sentences are translated into operations of the View (presentation) class associated with the given UI element.
5. System-to-system sentences are translated into procedure calls referring to appropriate operations in the Model layer. These calls are generated within the method generated with rule 2.
6. Stem-to-user sentences are translated into procedure calls referring to appropriate operations in the View layer. These calls are generated within the method generated with rule 2.

Note that the presented translational framework uses simple programming constructs. It is not scalable for larger systems and was meant only for these introductory notes. Our rules were also not presented in full detail and precision. In the actual detailed presentation in Chap. 4, we use a more sophisticated framework, which will also contain more rules and consider various configurations of RSL scenario sentences and domain elements. However, even with the current simple introduction, we can see that the intuitions from Sect. 1.1 can be substituted with very specific translation rules.

1.4 Towards Model-Driven Requirements Engineering

All the elements presented in the previous sections lead to constructing a new approach to software engineering. We call this approach Model-Driven Requirements Engineering (MDRE) [15, 114–116], although perhaps it would be more apt to call it Requirements-Oriented Programming. The first name emphasises the way we specify requirements—through constructing requirements models. The second name concentrates on direct contribution of requirements (models) to the final code. It also indicates that programming activities can be shifted closer to the level of precisely specified requirements. Let us now analyse how this new approach may change the way in which software is built.

1.4.1 “Traditional” Software Development

We will start with a typical (“traditional”) software development process [103, 132, 159], as illustrated in Fig. 1.21. This process is far from the “dream scenario” of Fig. 1.1. However, the goal is the same—to get from the user requirements (top left) to executable code (bottom left). To reach this goal, we need to go through several steps. First, we need to determine the detailed functionality of the system and describe the problem domain in terms of the data that need to be handled by the system. This forms the detailed software requirements. Having these details, we can

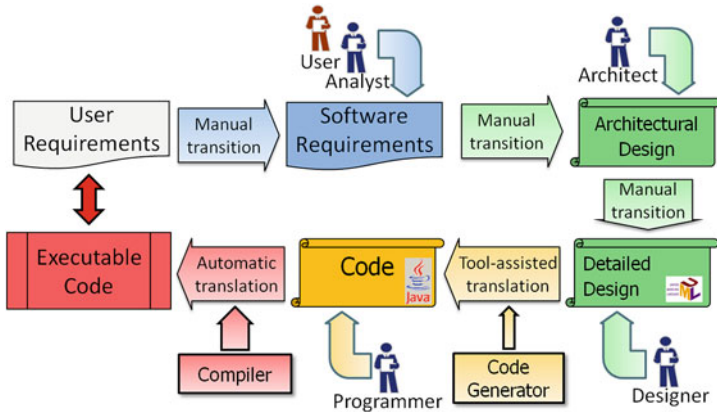


Fig. 1.21 Traditional software development

start design activities. We develop the high-level architectural specifications, and then determine the details of the individual components. This detailed design can then be implemented in 3GL code. Finally, we obtain the executables through compiling this code. This ends a typical path—user requirements to software requirements to architectural design to detailed design to 3GL code to executable code. The cycle can be repeated several times throughout a single project resulting in an iterative lifecycle. Of course, we have greatly simplified the lifecycle, leaving out important activities like quality assurance (testing) and deployment. Instead we concentrate on representing a gradual shift from artefacts at a high level of abstraction (user requirements) to those at the lowest level of abstraction (executable code).

This briefly sketched process involves various roles which use different techniques and tools to produce intermediate and final artefacts. Software requirements are normally created by analysts that cooperate with the users. This is usually a manual transition from less precise user requirements. Often, some elements of modelling are used, like use case diagrams to denote units of functional requirements and class diagrams to denote domain elements. However, most often such specifications rely on standard templates and natural language descriptions with some rigour in terms of the structure of use case descriptions. Moreover, it is not often that the vocabularies are made fully consistent with the functional specifications. The most frequently used tools to specify requirements are traditionally word processors. In many cases, this is supported by special-purpose CASE (Computer Aided Software Engineering) tools for requirements management [29, 30]. These tools can facilitate the management of requirements units, assigning attributes to requirements and tracing between user requirements and software requirements. To draw use case diagrams, another group of CASE tools is used—UML modelling tools [184].

The software requirements are taken by architects and turned into architectural specifications [40]. This is again traditionally a manual process. During this step, architects “inject” their knowledge on architectural frameworks and technological solutions. This knowledge turns use cases and domain elements into, e.g. components

and interfaces according to a chosen architectural framework. It can be noted that the architectural decisions are to an important extent influenced by the quality (non-functional) requirements. For instance, the choice of a specific UI framework can depend on the usability and portability requirements.

Architectural specifications can be developed using various tools. Often, these are simple graphical applications chosen ad hoc to draw diagrams showing the deployment of components or relationships between interfaces. Sometimes, a more rigorous approach is taken and a UML modelling tool is used to draw deployment and component diagrams.

The architectural specification is then subject to further design activities performed by the designers. They specify the details of individual components (subsystems). This mainly consists of dividing component code into units (classes) that realise the component interfaces. Sometimes, the component, internal dynamics in terms of interactions between runtime objects is specified. This also includes the design of detailed algorithms for especially complex data processing code. Detailed design specifications are more and more often developed using UML CASE tools [129]. This is due to their capability of generating code skeletons from class models. Class diagrams are used for documentation purposes (as a visual “map of code”) and thus their role becomes significant.

Designers are often the same people as the programmers. Thus, they often simultaneously design and program their components. This is obviously done using typical IDEs (Integrated Development Environments) [88] that contain code editors, compilers, debuggers and execution environments. The compilers form the final, most automated step in “traditional” software development. Sometimes, the IDEs are integrated with UML editors and generators. In this way, design models and code can be developed hand-in-hand, thus making code more visual and understandable.

1.4.2 Model-Driven Software Development

Typical software development can be seen as mainly a manual process. There exist some elements of code generation but we would certainly desire much more automation. This has changed with the advent of Model-Driven Software Development/Engineering (MDSD/E) [18, 25, 178]. Originally, the idea of MDSD was launched by the Object Management Group (OMG)⁴ and called Model-Driven Architecture (MDA)⁵ [56, 66, 92, 106, 111, 127, 167]. The underlying principle is that the intermediate artefacts in the software development process are models that are transformed to produce other, more detailed models. For example—generating the user interface models from higher-level domain model [73]. MDA introduces three basic levels at which models are produced.

⁴ <http://www.omg.org/>.

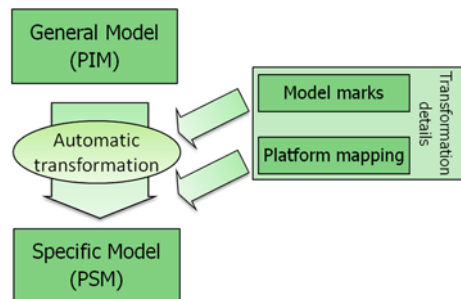
⁵ <http://www.omg.org/mda/>.

- *Computation-Independent Model (CIM)*. “A computation-independent model is a view of a system from the computation-independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification”.
- *Platform-Independent Model (PIM)*. “A platform-independent model is a view of a system from the platform-independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type”.
- *Platform-Specific Model (PSM)*. “A platform-specific model is a view of a system from the platform-specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform”.

MDA in its specification is not precise as to what the exact boundaries are between these layers. This also pertains to defining what a “platform” is. The definition offered by the MDA Guide [111] is rather imprecise, although it gives some important hints. “A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented”. It can be argued that the division between platform-independent and platform-specific design is quite foreign to “traditional” software developers. Thus, the postulate to create an artefact that is “platform-independent” sounds somewhat artificial to them. This might be the reason that MDA did not really spread widely in software engineering (at least not as widely as it was initially expected and hoped for).

Nevertheless, the MDA approach introduces a fundamental concept of gradual and automated transition from models that are close to the problem domain to models that are close to the target code. This transition is made by means of model transformation, as illustrated in Fig. 1.22. The less detailed (general) models (e.g. at the PSM level) can be transformed to more detailed (specific) models (e.g. at the PIM level) by adding certain details inserted through the transformation. What is important is that these inserted additional details can be configured to some extent—the models can be marked with additional information. If several PSMs are the

Fig. 1.22 General model transformation scheme



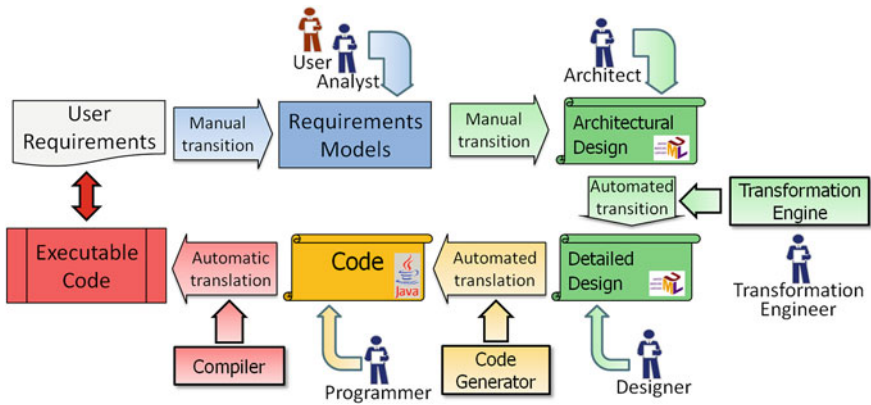


Fig. 1.23 Model-driven software development

potential targets of the transformation, it can be configured based on these markings. This configuration determines the kinds of mappings to the target platform elements that will be used in the transformation. In this way, models leading to code for different target platforms can be produced.

MDA transformations can be used in the software development process and thus form the MDSD lifecycle as shown in Fig. 1.23. We retain the same process structure as in Fig. 1.21. The MDA layers of CIM, PIM and PSM can be mapped onto Software Requirements, Architectural Design and Detailed Design respectively. This is not an exact mapping but such division into layers and phases is familiar to software developers not acquainted with MDA. In the MDSD approach, we concentrate on using models during all of these three main phases of software development. However, note that we have not introduced any automation in the transition from Requirements Models (cf. CIM) to Architectural Models (cf. PIM). Such transitions are not supported in industrial reality. Practically all of model transformation approaches concentrate on transitions from PIMs to PSMs. This means shifting between design-level models and not reaching as far back as the requirements models.

MDSD activities are performed by Architects and Designers. However, the role of designers is significantly reduced because the design models are to a large extent generated automatically. Instead, there needs to be introduced the role of the Transformation Engineer who is responsible for developing and maintaining transformation programs executed within Transformation Engines. The role of the Architects is to develop a general (platform independent) model and mark it with platform-specific decisions. Then, the transformation should generate a platform-specific detailed design model. Using MDA transformations we can also generate code. This can add to standard CASE tool code generation capabilities and relieve Programmers from developing many standard and repeatable code fragments.

MDSD can be practiced using various tooling environments. The most popular are probably the various model transformation engines embedded into UML modelling environments. Being the most widely used modelling language, UML is the obvious choice for developing models also in the MDSD process. Thus, UML tool producers have developed various model transformation modules. They introduce model transformation languages usually specific to a given tool. Moreover, there can be found several model transformation environments that are based on standardised model transformation languages [38]. These environments are external to modelling tools but interface with their model repositories [168]. We discuss this in detail in Chap. 5.

When discussing the applicability of MDSD in practice, we can compare Fig. 1.23 with Fig. 1.21. The complexity of both processes is similar or one may even infer that MDSD is in fact more laborious. This impression can be argued as incorrect [112, 113] but it may be one of the reasons for the limited success of typical MDSD approaches. MDSD would need to remove some phases in software development in order for software developers to see the real benefits. This necessitates much more automation and transformations coming right from requirements.

1.4.3 Software Development with DSLs and Model-Driven Requirements

To shorten the path from requirements to code, we can try to remove or simplify some of the phases. This is usually done when agile methodologies are applied. Often, the detailed requirements are simplified to informal user stories combined with sketched domain models as the exact requirements are worked out during an iterative development process when the actual system is examined by the users. Moreover, the design activities are also reduced and most of the design decisions are made directly during coding. The great advantage is that we result with a system that is developed precisely according to what the user needs—this is verified on-the-go in very short validation cycles. However, this approach certainly influences scalability and maintainability of systems. Further maintenance and extensions are compromised due to lack of proper documentation and diagrams showing the code structure and “how it works”.

Software developers consider detailed requirements and design documentation as overhead. Writing code directly contributes to the final effect, while writing models does not. However, it is obvious that for large systems to be maintainable, we need design documentation. The model-driven approach brings a solution to this dilemma, where the design models become first-class artefacts and not overhead. Developing a model pays off through automatic transformation down to code. But, we would like to have even more: to be able to construct a single model at a high level of abstraction (as close to requirements as possible) and translate it directly into code.

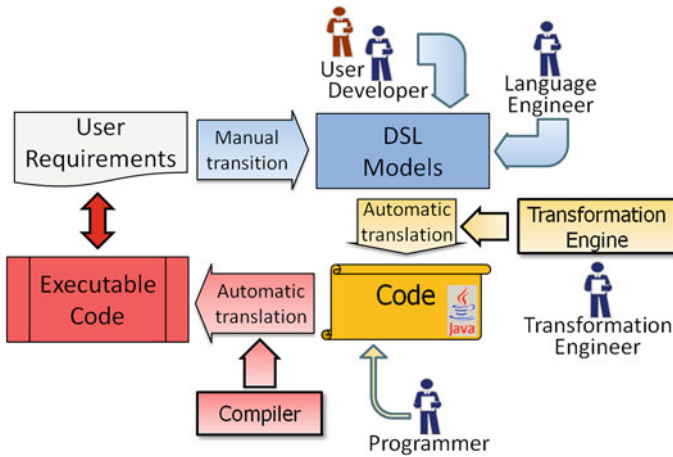


Fig. 1.24 DSL-based software development

The desire to shorten the software development path is reflected in the idea of Domain-Specific Languages (DSL) [36, 54, 89]. Such languages allow us to create models that are specific to a given problem domain and quickly transform into working code. This significantly reduces the number of steps in the process, as illustrated in Fig. 1.24. We no longer need architects and designers, instead, we have only the Developer role that cooperates with the User to produce DSL models. Also, the role of 3GL programmers is significantly reduced or even not needed as most or all of the target code is generated automatically through the transformation engine.

In the DSL-driven process, two additional roles are important. This is the Transformation Engineer and the Language Engineer. We have already discussed the first. The second role is necessary to develop and maintain the language in which the users and developers create their domain-specific models. This is crucial because the language has to be expressive and understandable but also formally precise (cf. language semantics). Such languages are distinct for specific problem domains and develop together with these domains. Moreover, they usually combine elements of domain logic and application logic. A specific DSL may be created for only a single system, and within the domain of a single business organisation.

The greatest advantage of the DSL approach is that in the realm of a suitable problem domain, significant productivity gains can occur [62]. This is mainly achieved when a family of similar systems is needed (e.g. for different clients), leading to the idea of Software Product Lines (SPL) [60, 131, 138, 171]. A fast process to generate similar (variable) systems can be established through defining similar source models based on a DSL. For isolated systems, productivity gains are not very impressive as additional effort is needed to develop a DSL (Language Engineer) and then to develop model transformations to the target platform (Transformation Engineer). This effort pays off significantly only when several similar systems using the same DSL and

transformation are developed. To develop a DSL and associated transformations is also a significant investment which forms a barrier for using this approach.

For the above reasons, it would be beneficial if we had a general-purpose (unified) language that would be able to substitute the plethora of domain-specific ones. This language could be used in a wide range of domains and for various application types. Moreover, we could develop many transformation variants for different target platforms. These transformations could be reused many times and thus a unit cost of their development could be spread among many projects throughout many business organisations. This idea finally brings us to Model-Driven Requirements Engineering. In this approach, we want to develop models in a unified Requirements Specification Language and transform them directly to code.

Figure 1.25 illustrates this kind of process. It is similar to the previous one, driven by DSLs. This time RSL is the source language for automatic transformations. Moreover, the Language Engineer is not needed because RSL is a unified language and is maintained across many projects and many organisations. The main consequence of using RSL is that not all the code is generated from requirements models. This is due to lack of constructs to express the domain logic (data processing). In DSLs, these constructs were developed as part of a specific language. As a result, three possible scenarios for data processing formulation are possible: (1) it is coded directly in 3GL; (2) it is expressed using a DSL integrated with RSL; (3) it is formulated in an RSL algorithmic extension (which necessitates additional research). These three scenarios are discussed in detail in Chap. 7. In this book, we mostly concentrate on the first one, where the generated code is clearly marked with places where data processing code should be supplied by 3GL programmers. The marking can be performed using detailed design models showing the map of necessary updates. This means a compromise between full code generation, limited

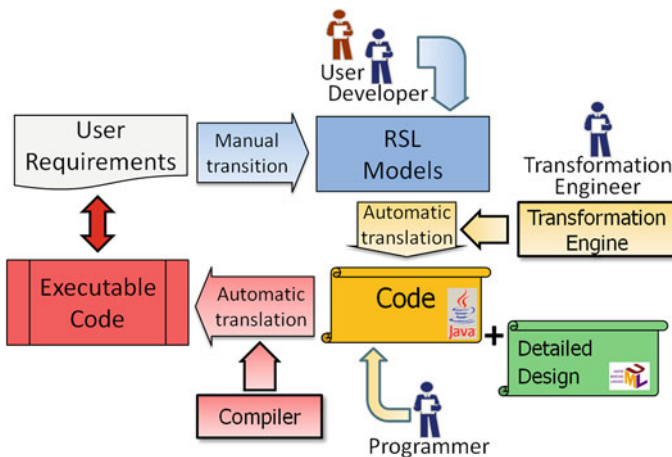
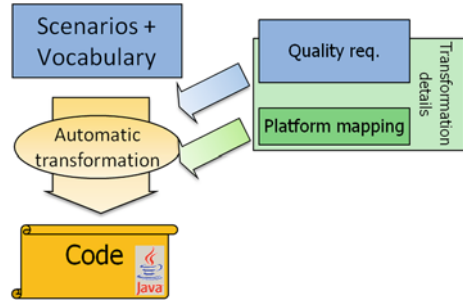


Fig. 1.25 Model-driven requirements software development

Fig. 1.26 RSL to code transformation scheme



to specific domains (DSLs) and universality of approach with a more sophisticated process (standard MDSD).

Technically, an RSL-based MDRE transformation can be implemented as a typical MDA transformation according to Fig. 1.22. The big shift however is that the transformation source is equivalent to a CIM. A PSM together with working code is thus generated directly from the CIM. This was not even present in the DSL-based approach where the source models expressed in a DSL should be placed somewhere between a CIM and a PIM (they are predominantly too technical to call them “requirements”). Instead of model marking and platform mapping, the transformation can be controlled with quality requirements models as indicated in Fig. 1.26. This is an important characteristic of this transformation. The architectural and detailed design decisions pertaining to the target platform are embedded in the transformation and marked through the values of quality metrics. For instance, a different target platform may be chosen by the transformation if a mobile interface is required and a different one for a web-based interface.

Both the DSL-based and the RSL-based approach promise significant gains in productivity. However, in the first case, each specific problem domain has to be equipped with a Domain-Specific Language. Moreover, model transformations have to be developed that transform from the DSL-compliant models to the target platform code. In the second case, both the language (RSL) and the transformations are developed for various problem domains and various target platforms independent of the potential problem domains. This obviously reduces the effort in a specific software project or for a specific software product line. However, regardless of the approach, several new competences are needed. First, we need to understand how software modelling languages are constructed in terms of their syntax and semantics. Second, we need to know how to develop model-to-model and model-to-code transformations. The remainder of this book provides detailed guidelines in these two areas.

From Requirements to Java in a Snap
Model-Driven Requirements Engineering in Practice
Smialek, M.; Nowakowski, W.
2015, XXIII, 352 p. 295 illus., Hardcover
ISBN: 978-3-319-12837-5