6 Contracts and Scenarios in the Software Development Process

Summary: Software development processes play an important role in the successful and timely delivery of software. There are different approaches to software development processes—more traditional approaches following and extending the spiral model (e.g., the Rational Unified Process—RUP) and more agile approaches such as Scrum, that try to integrate ideas from extreme programming (XP) and extreme modeling. Regardless of the software development process, the application of contract and scenario techniques is possible in all major tasks involved with software development (analysis, design, implementation, and testing).

Keywords: Unified Process (UP), Rational Unified Process (RUP), Scrum, Agile, Agile models, Quality assurance

So far, we have followed a product-oriented perspective on contracts and scenarios. In Chap. 2, we presented the technical foundations of contracts as we understand them in this book. Chapter 3 related the general principles of contracts to the techniques available in UML, and presented guidelines about how to formulate contracts. In Chap. 4 we presented the technical foundations of scenarios, while Chap. 5 related the general principles of scenarios to the techniques available in UML (e.g., use case diagrams and interaction diagrams) and presented guidelines about how to formulate use cases, use case diagrams, and interaction diagrams.

The emphasis of Chaps. 3 and 5 was on relating general concepts to the techniques available in UML. Implicitly, we assumed that the presented techniques could be applied successfully in the early phases of the software development process. But what is a software development process? In general, a software development process is a set of activities that are needed to transform a user's requirements into a software system. A software development process [Jacobson99]:

- Provides guidance to order a team's activities.
- Directs the tasks of individual developers and of the team as a whole.
- Specifies what artifacts should be developed.
- Offers criteria for monitoring and measuring a project's products and activities.

One general model for describing software development processes is the waterfall model as described by Boehm [Boehm76], which specifies the production of software as a sequence of phases. Numerous variations of this model exist, some even

offering the possibility of jumping back to a previous phase. Typical phases according to the waterfall model are requirements analysis and definition, design, implementation, testing, deployment and operation, and maintenance. In the waterfall model, the end of a phase also marks an important milestone at which certain products and documents must be available. For example, the analysis and design phase ends with the availability of the software requirements specification. In real projects [Parnas85] this is problematic, as only parts of the requirements can be captured accurately at the beginning of the project. In a real-world project, the requirements change and evolve in the course of the project. The same problem occurs in the later phases of the software development process.

It is now commonly understood (both in industry and academia) that software development processes have to be iterative. However, some quite different approaches to the software development processes can still be identified. The spiral model [Boehm88] is an enhancement of the classical waterfall model, as it supports an iterative approach; that is, the phases as defined by the waterfall model can be passed several times. However, the waterfall model still concentrates on the production of one defined product, rather than considering several versions—such activities would be postponed to the maintenance phase, or in the case of major revision of a product would lead to a new product.

On the contrary, in an incremental approach, differing versions of the software are considered right from the beginning, The STEPS approach [Floyd89], as well as other more agile process models such as Scrum, explicitly support incremental development. This seems to be a key approach to managing complexity, as the planning horizon always is the next version; that is, the next increment. This approach gives us the additional opportunity to roll out products earlier, as a version of a software product is always considered to be fully functional and stable—but with missing functionality that has to be added in later versions of the product. Nevertheless, if this version supports the users' work, it still makes sense to make it available to them (obviously, roll-out can also take place when all increments are consolidated at the end of the process).

The Unified Process (UP) [Jacobson99] is an example of an iterative process. It is quite popular, as it is tightly tied to UML; that is, the UP is the surrounding process for the development and refining of UML models. The UP still is a rigid process with clearly defined workflows, and with typical activities known from classical software development processes. Basically, the UP is a more modern version of the spiral model. The Rational Unified Process (RUP) [Kruchten00] is a specialized form of the UP. While the UP remains at a somewhat general level, the RUP is a web-enabled product of Rational Software, Inc. According to Rational [Rational03b]:

> The Rational Unified Process, or RUP, is a web-enabled set of software engineering best practices that provide you with guidance to streamline your team's development activities. As an industry-wide process platform, RUP enables you to easily choose the set of process components that are right for your specific project needs. You will achieve more predictable results by unifying your team with common processes that improve communication and create a common understanding of all tasks, responsibilities, and artifacts. On one cen-

> tralized web exchange, Rational Software, platform vendors, tool vendors and domains experts provide the process components you need to be successful.

On the other hand, agile methods and techniques such as XP [Beck99a, Beck99b], Feature Driven Design [Palmer02], DSDM [Stapleton97], the Chrystal family of methodologies [Cockburn02], and Scrum [Schwaber02] are gaining widespread use. Currently, no agreement exists on what the "agile" concept actually refers to. Nevertheless, it has been widely acknowledged that the introduction of the extreme programming method [Beck99b] was the starting point for various agile software development approaches. The "Agile Software Development Manifesto" [AgileManifesto02] states a number of values that are the driving forces for developing techniques and for organizing software development processes:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

As McCauley [McCauley01] and Glass [Glass01] state, there is a need for both agile and process-oriented methods, as there is no software development model that suits all imaginable purposes. A good overview of agile methods can be found in Abrahamsson et al. [Abrahamsson02].

Scrum [Schwaber02] is an example of a software development process that aims at supporting the construction of software in an agile way. We have chosen Scrum in this book as it is one of the most promising developments in the field of agile models. There are some developments under way [XPScrum03], such as refactoring, pair programming, and collective code ownership, that are attempting to integrate the Scrum management practices with XP practices [Cockburn02].

In this chapter, we give an overview of the Rational Unified Process (RUP) as well as of the Scrum process, in order to give the readers an idea of typical up-to-date software process models. We prefer the RUP over the UP, as the RUP is more dynamic and open to changes. As this book focuses on quality, we also describe the role of quality assurance in these software development processes; that is, which technical and methodological approaches are followed to enhance software quality. For each typical activity in a software development process, we will show how contracts and scenarios can be applied and how they fit into the specific processes (RUP and Scrum).

6.1 An Overview of the Software Development Process

6.1.1 The Rational Unified Process (RUP)

The RUP is an iterative approach for object-oriented systems. The RUP is use-case driven; that is, use cases are a central part of the modeling requirements and of building the foundation for a system. The RUP is divided into four phases (see

Fig. 37), called “inception”, “elaboration”, “construction”, and “transition”. Each phase is split into iterations, each of which serves the purpose of refining the expected results of a phase in an iterative way. It is important to understand that the deployment of the software product is in the transition phase; that is, there is no incremental software development where a preliminary version with, for example, 40% of the intended functionality is available—at project half-time. We will give a short overview of the phases and key workflows—a more detailed description can be found in Abrahamsson et al. [Abrahamsson02] and Kruchten [Kruchten00].

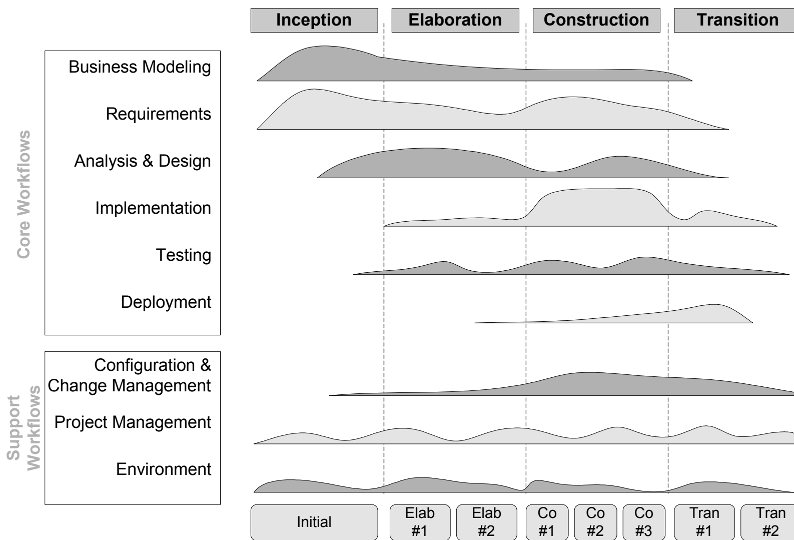

Fig. 37. The Rational Unified Process (RUP)

Inception: During the inception phase, the vision of the end-product and its business application—that is, the scope of the project—is defined. The outcomes of the inception phase are a number of artifacts: a vision document (the project requirements, key features, and main constraints), a use case model (which contains all of the identified use cases and actors), an initial business case, an initial risk assessment, and a project plan (which contains the main phases and iterations).

Elaboration: In general, elaboration is the process of developing something in greater detail. For the RUP, this means that all necessary activities and required resources are planned. All features are specified and the architecture is designed. The outcomes of the elaboration phase are a number of artifacts: a use-case model (quite complete, with all actors and use cases identified), supplementary requirements (to capture nonfunctional requirements or requirements that are not related to use cases), a description of the software architecture, an executable architectural prototype, a revised risk list, and a development plan for the overall project.

Construction: In this phase, building of the product and evolution of the vision, architecture, and plans takes place. At the end of the construction phase, a first release of the product has to be available. The outcomes of this phase are the aforementioned software product, integrated on an adequate platform, the user manuals, and a description of the current release.

Transition: Transition is the process of delivering the product to the customer. This process encompasses manufacturing, delivering, training, supporting, and maintaining the product until users are satisfied.

Each phase can be carried out in several iterations. Kruchten [Kruchten00] gives an example of the timely distribution of the phases for a 2-year project: "inception", 2.5 months; "elaboration", 7 months; "construction", 12 months; and "transition", 2.5 months.

In general, throughout these phases (see Fig. 37) a number of workflows are taking place. The RUP distinguishes between core workflows and supporting workflows. The core workflows are Business Modeling, Requirements, Analysis and Design, Implementation, Testing, and Deployment. The supportive workflows are Configuration and change management, Project Management, and Environment. The contribution of particular workflows varies over time; that is, it depends on the actual phase and iteration of the project. As you can see in Fig. 37 , for example, the emphasis of the deployment workflow is on the transition phase, while deployment does not play any role during the inception and elaboration phases.

Business Modeling: The general goal of this workflow is to understand the structure and dynamics of the organization in which a system is to be deployed. Furthermore, current problems in the target organization and improvement potentials have to be identified. Another important goal of this workflow is to ensure that all stakeholders taking part in the project have a common understanding of the target organization. Finally, system requirements have to be derived in order to support the target organization.

Requirements: The main goal is to establish and maintain agreement with the customer and other stakeholders on what the system should do. The gathering and documentation of requirements has to provide system developers with a better understanding of the system requirements, has to define the boundaries of the system, has to provide the basis for planning the technical contents of iterations, and has to provide a basis for estimating the cost and time needed to develop the system.

Analysis and Design: The purpose of this workflow is to translate the requirements into a specification that describes how to implement the system. The goal is to derive an analysis model and to later transform it into a design model; that is, a model that can be directly used in implementation. We do not discuss how to

differentiate between analysis and design models, but we refer to Kruchten [Kruchten00], who presents the RUP view on this issue.

Implementation: The implementation workflow serves a number of purposes. First of all, the general organization and packaging in terms of systems and subsystems has to be defined. Furthermore, the implementation workflow is responsible for implementing the system; that is, providing the necessary classes and components. The testing of individual units, as well as system integration, completes the goals of this workflow.

Testing: The major goal of this workflow is to ensure that the developed software is correct; that is, that it fulfills the specification. Additionally, the testing workflow has to identify any defects, and ensure that all discovered defects are addressed before the software is deployed.

Deployment: The general purpose of deployment is to turn the finished software product over to its users. This involves activities such as beta testing, packaging of the software for delivery, software distribution, software installation, and the training of the end users.

Configuration and Change Management: The purpose of configuration and change management is to track and maintain the integrity of evolving project assets; that is, of all artifacts that are created and changed during the project.

Project Management: The project management workflow as understood by the RUP provides a framework for managing software-intensive projects, provides practical guidelines for planning, staffing, executing, and monitoring projects, and provides a framework for managing risk. The RUP explicitly excludes issues such as managing people, budgets, and contracts.

Environment: This supportive workflow is responsible for providing the development organization with processes and tools. This support includes tool selection and acquisition, process configuration, and improvement, as well as typical technical services to support the process.

6.1.2 Scrum

The Scrum methodology is a typical representative of so-called agile methods. The main reasons why we have chosen Scrum as an alternative approach here are as follows:

- Scrum is a kind of project meta-model that allows us to decide how analysis, design, implementation, and testing have to be carried out—this is not tied to any specific process or method, but is free to the applicants of this meta-model. It is therefore possible to use techniques from the XP toolbox, as well as to introduce proven methodologies or techniques from the RUP.

- Scrum represents a modern style, the agile style, of software development that is suitable for small to medium-sized projects, where incremental software development with easy-to-grasp tasks and functionality is the core planning mechanism. The approach delivers software to the customer in an incremental way and therefore introduces productive feedback mechanisms early in the software development process.
- Scrum integrates well with agile-style software development. Efforts are under way [XPScrum03] to integrate Scrum with XP.
- We believe that scenario-based and contract-based prototyping offer optimal payback in an incremental project setting, as—due to the incremental nature of behavioral changes—contracts are invalidated, outdated, have to be extended, and so on. Both contracts and scenarios support this process, as we will describe in Sect. 6.3.

The Scrum process is person-centered (in the positive meaning of the agile manifesto [AgileManifesto02], as it concentrates on how the team members should function in order to produce the system flexibly. Scrum, as already mentioned, does not define specific software development techniques for the implementation phase—it can be applied regardless of the implementation techniques to be used (object-oriented development or classical module-oriented software development).

The general assumption that underlies the Scrum process is that the development of a system is unpredictable due to a number of variables (e.g., the requirements, the time frame, the resources, and the technology), which are likely to change during the process. According to the proponents of Scrum, this implies a flexible process that can adapt itself according to the changes that take place. Figure 38 visualizes the Scrum process.

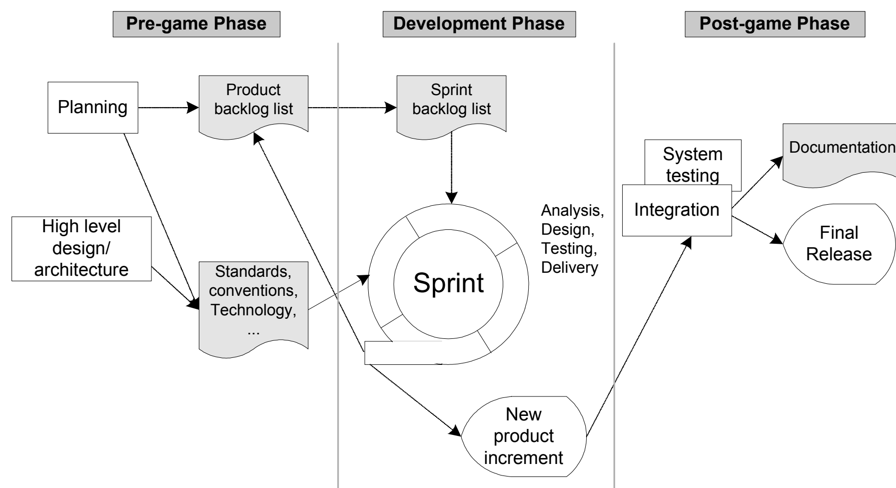

Fig. 38. The Scrum Process

The Scrum process consists of three major phases—the pre-game phase, the development phase, and the post-game phase.

Pre-Game Phase: In the pre-game phase, two major activities take place. (1) Planning activities have to be carried out in order to define the system to be developed. This is achieved by creating a backlog list; that is, a list that contains all known requirements. The backlog list is changed throughout the entire software development process—as soon as new knowledge is available, the backlog list is improved to reflect the current requirements for the product. The backlog list is a central planning instrument, as it serves as the principal means of updating estimations and establishing priorities about the requirements to be implemented in the next increment.

The pre-game phase also incorporates an architecture phase, in which the high-level architecture of the system is developed and changed; that is, changes to the backlog list are also reviewed from a design perspective. The main emphasis here is on how additional or clarified requirements make changes necessary in the design.

Development Phase: The development phase is carried out in Sprints. Sprints are iterative cycles in which the functionality is developed or enhanced to produce new increments. Each Sprint includes the traditional phases of software development; that is, requirements elicitation, analysis, design, implementation, testing, and delivery. A Sprint typically lasts for 30 days. It is current practice that the functionality of an increment is determined in a Sprint planning meeting, where customers, users, and management decide upon the goals and the functionality of the next Sprint. During execution of the Sprint, there are daily Scrum meetings that are organized to keep track of the progress of the Scrum team continuously, to discuss problems, and to decide what has to be done next. A Sprint is concluded with a Sprint review meeting, held on the last day of the Sprint, where the Scrum team presents the results (i.e., the working product increment) to the management, customers, and users. The Sprint review is used to assess the product increment and to make decisions about subsequent activities. It may also happen that changes are made to the direction of the system being built.

Post-Game Phase: The post-game phase is entered when agreement is reached among the stakeholders that no more requirements are left or should be included in the release. In the post-game phase, typical deployment activities such as integration, system testing, and documentation take place.

6.1.3 Observations

In the previous sections, we have briefly presented the Rational Unified Process (RUP) as well as the Scrum approach. While the RUP is a typical representative of more classical software development approaches (in the tradition of the spiral model), Scrum is a “new age” approach to support agile concepts. Some observations that are of importance in the remainder of this chapter are as follows:

- Typical activities, which are already known from the waterfall model or from the spiral model, can also be found in the RUP and in the Scrum software development process—these activities are requirements elicitation, analysis and design, implementation, testing, and deployment. The RUP has additional supportive workflows that are not directly related to core software development (configuration and change management, environment, and project management), and a business modeling workflow that also goes beyond core software development. Business modeling as understood by the RUP concentrates on finding out the structure and dynamics of the organization in which a system is to be deployed. Furthermore, current problems in the target organization and the potential for improvement have to be identified.
- Both approaches are iterative, which means that, for example, the developed specifications, models, and implementations evolve over time—hopefully in a consistent way. In the RUP, the iterations are concentrated on the main products of the actual phase; that is, on the analysis model during the analysis and design phase and on implementation in the construction phase. In the Scrum model, it is obligatory that one iteration covers requirements elicitation, analysis, design, implementation, and testing, which means that usually in every iteration changes to the analysis model, the design, the implementation and to the test environment are necessary.
- The RUP (and also the more general Unified Process—UP) rely heavily on use cases and on the UML. Therefore, our approaches to prototyping with contracts and scenarios are applicable in principle, as both components form part of the UML, by means of the OCL for contracts and interaction diagrams for scenarios.
- Scrum is independent of any specific implementation techniques. It is therefore feasible to integrate contract approaches and scenario approaches—regardless of the technology used (UML-based or not)—into the Scrum process.

6.2 Quality Assurance and Software Development Processes

The emphasis of this book is on providing and explaining techniques to enhance the quality of software products, with a focus on contracts and scenarios. Nevertheless, we will briefly describe how the software development processes described in this chapter attempt to enhance the quality of software from the point of view of the process.

Quality Assurance in the Rational Unified Process (RUP): The RUP is a software development process that is also suitable for large teams. Nevertheless, the RUP does not anticipate a team or a distinctive workflow to ensure quality (as is the case is the V model [VModel03]). The RUP distinguishes between product quality and process quality. Process quality (according to [Kruchten00]) is the degree to which an acceptable process was implemented and adhered to during the manufacturing of the product. The RUP itself focuses on verifying whether a product

meets the expected quality level. For this purpose, a test workflow is available that tries to ensure a proper quality level. In the RUP, testing is a workflow that occurs in parallel with the implementation workflow. The RUP systematizes the dimensions of testing—the quality dimension (e.g., reliability and performance), the stages of testing (e.g., the unit test and the integration test), and the types of test (the benchmark test, the function test, and the load test)—and provides a principal test model that can be used in any project. A test model defines what will be tested and how it will be tested. A test model includes test cases, test procedures, test scripts, test classes and components, and notes, and is the basis for carrying out testing. At the general process level, the RUP tries to ensure quality by means of iterative development for the distinctive phases of the software development process.

Quality Assurance in the Scrum Process: At the process level, the most important aspect for quality assurance is the daily Scrum meeting during a Sprint. These daily meetings ensure that the entire development team is on track, that they work focused for the next increment; that is, they try to work off the actual Sprint backlog list.

Another important issue is the continuous involvement of the stakeholders (users, managers, and clients). The main difference compared to classic methods is that the stakeholders get a better feeling about the product capabilities, as they can directly see at the end of a Sprint what could be achieved, and how the software behaves or misbehaves. This is a valuable input for the next planning phase; that is, for adapting and enhancing the product backlog and for planning the next Sprint backlog.

According to the Scrum approach, it is important that there are no interruptions from outside during a Sprint. It is therefore not possible to change a Sprint backlog during execution of a Sprint. We think that this is an important issue, as efficient software production can only take place when the software development team has the time to concentrate on the problems to be solved, without being forced to continuously react to exterior inputs.

In the development phase of Scrum in particular, there no specific development techniques are favored. Nevertheless, as Scrum is an agile process model it perfectly complements agile software development approaches such as XP [Beck99a]. As already mentioned, work is under way [XPScrum03] to integrate XP into the Scrum process model. There are some XP practices that have a positive impact on software quality, such as test-driven development (write tests before you implement the software to be tested), continuous refactoring, continuous integration (code is integrated into the code-base as soon as it is ready), and onsite customer integration (the customer has to be present and available full-time for the team).

6.3 Contracts and Scenarios in the Software Development Process

In the previous sections, we have presented and discussed two typical software development processes, the Rational Unified Process (RUP) and the Scrum approach. While the RUP is a consequent development of the spiral model, the Scrum approach is a representative of an agile process that can easily be complemented by agile techniques.

Regardless of the type of software development model, a number of tasks always occur in software development processes—requirements elicitation, analysis, design, implementation, and testing. In the following sections, we will briefly describe the role of scenarios and contracts in the software development process. We will concentrate on the scenario and contract approaches because we have described them in the early chapters of this book. In this chapter, we will not take necessary tool support into consideration—the discussion about tool support is postponed until Chap. 7. The description in this section has the following pattern:

- *The role of contracts:* This subsection describes how contracts can be applied to the specific task.
- *The role of scenarios:* This subsection describes how scenarios can be applied to the specific task.
- *Methodological issues:* In this subsection we discuss general methodological issues; that is, issues that are independent of any specific software development model.
- *RUP/Scrum issues:* In this subsection we discuss (if applicable) special issues that are related to particular aspects of the RUP or Scrum process.
- *Example:* In this subsection we show, by means of our mobile agent example (see Appendix A), how to use contracts and scenarios.

6.3.1 Contracts and Scenarios for Analysis

Analysis is the task of finding out what exactly a software program should do. This involves tasks such as gathering and documenting requirements. The gathering and documenting of requirements takes place at different levels of abstraction. From the point of view of a customer, features or a list of features are typically the subject matter of interest. For software developers, the detail level of feature lists is too coarse—therefore, the software requirements have to be captured and described in much more detail. With regard to the RUP, the analysis task comprises the requirements workflow as well as parts of the analysis and design workflow. With regard to the Scrum process, the analysis task takes place during the pregame phase (planning and enhancement of the backlog list) and during the development phase—here, an explicit analysis task is anticipated.

The Role of Contracts: During analysis, general, business-related rules are often captured and documented. On the whole, we believe that the use cases to be de-

veloped are the driving force for the subsequent development of business rules. Use case descriptions usually have sections that describe preconditions and post-conditions for use cases, and separate tabular representations of business rules (see [Kulak00]). During the analysis process, a model (by means of a class diagram) is developed to capture core elements and their relations. In the course of defining and refining this model, assertions have to be added step by step. As we pointed out earlier in this book, it is not appropriate to capture the assertions at the use case level by means of a formal language such as OCL; the same applies to the business rules. Nevertheless, when a certain degree of maturity is reached in the analysis phase, these formal approaches can be used to capture the assertions (in use cases and business rules) in a more precise way.

The Role of Scenarios: Scenarios play an important role in capturing the requirements of a system—this is true for systems in which business processes play an important role. The approach is not so suitable, in our opinion, for capturing the requirements of typical software tools, such as graphics editors or word processors. The technique can be used from first rough sketches of what has to be achieved right up to detailed descriptions of a scenario. In Chap. 4, we presented a tabular representation for scenarios that is valuable during the analysis task, as additional information can be added in the course of development of the scenarios. For selected use cases, a more formal description by means of interaction diagrams can be developed.

Methodological Issues: As described above, scenarios and contracts are developed in parallel, where the scenarios are the driving force for capturing and documenting the system and software requirements. Figure 39 shows a principal process that contract-related and scenario-related activities follow during analysis.

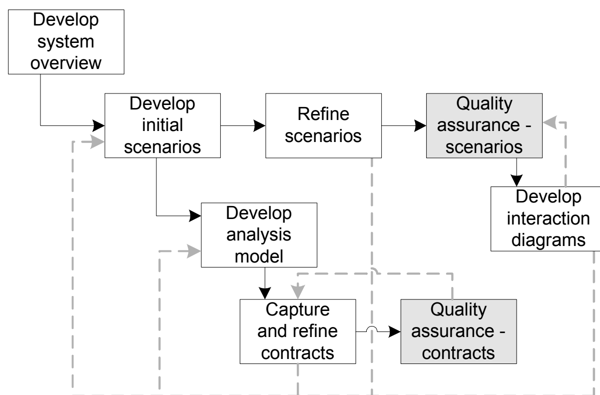

Fig. 39. Contracts and scenarios for analysis

Development of the system overview has something to do with finding out what the customer wants in the larger perspective—What's the business case? What is the aim of the project? What does the customer want the product to do in general? One principal idea about how to capture and document the system overview is to use high-level use case diagrams. The use of this approach helps to identify system boundaries as well as the actors needed to fill in more details.

On the basis of this general understanding of the system, initial scenarios are identified and described. At this stage of the project, it is important not to go into too much detail. It is more important here to identify all of the relevant scenarios and to understand them in principle. It is a good idea at this point to concentrate on good names for scenarios, as well as on good overview descriptions.

After these initial scenarios have been approved by all of the stakeholders who are interested in this system, the scenarios are refined and an analysis model of core parts of the system is developed. The refinement of scenarios, as well as the development of an analysis model, is itself an iterative process, and it might also require us to step back, and to enhance and change the initial set of scenarios, due to additional insights gained during the analysis phase (this explicit iteration leads back to other tasks, as shown by the dotted line in Fig. 39). The process of refining scenarios means to incrementally fill in the missing parts of a textual scenario description (see Sect. 4.4.1); that is, the basic course of events, alternative paths, preconditions, postconditions, and so on. The more insight that the analysts gain in the system to be developed, the more elaborate the analysis model will be. The development of the analysis model comprises the definition of core elements of the system as well as their relationships. This analysis model can also be enhanced using contracts that describe—at the class level (invariants) or at the method level (preconditions and postconditions)—aspects of the behavior of the elements. Refining the analysis model in this way, as well as refining the scenarios, leads to changes (visualized in Fig. 39 using dotted lines). In this context of constant iterations and changes, we believe that contracts are an excellent tool, as they support consistent changes—that is, either you find out that the changes that you make violate existing contracts and therefore are not allowed, or you find out that the contracts have to changed in order to capture the behavior more precisely.

Quality assurance for scenarios and contracts is an important issue. In Chaps. 3 and 5, we developed a number of guidelines for the development of assertions and scenarios. It has to be assured that the contracts and scenarios adhere to these guidelines. These quality assurance tasks definitely have a loop back to the "refine scenario" task, as well as to the "capture and refine contracts" task. We will model this explicitly in order to emphasize it better.

For selected scenarios, event interaction diagrams will be derived on the basis of textual description. For cost reasons, this can only take place for selected scenarios. In order to ensure good quality, the developed interaction diagrams also have to adhere to the developed guidelines—especially the guidelines for drawing interaction diagrams (see Sect. 5.4.4).

RUP/Scrum Issues: The process described in the above subsection can be applied for the RUP as well as for the Scrum approach. Nevertheless, we believe that the

payback in the Scrum approach is even higher. In the RUP approach, the analysis task is carried out in several iterations, but always for the entire scope of the project, while with the Scrum approach new requirements are added in each iteration. This incremental approach followed by Scrum imposes additional challenges on the analysis task, as new requirements always lead to changes of requirements or of the analysis model. These changes can more easily be made when parts of the behavior are captured using assertions, as it is possible to find out in a structured way (by systematically reviewing existing and newly added assertions) whether the changed model is still consistent.

Example: The system overview can best be provided by means of a use case diagram. Figure 64 shows such an example for our deployment example (see Appendix A). Usually, such a diagram is developed in several iterations—the diagram as presented in Fig. 64 shows the result of this process. The next important step is the development of initial scenarios; that is, of scenarios that only contain the overview. The following descriptions show the overview for selected scenarios:

Scenario Name:	Install Agent
Overview:	Installs an agent along with its initial configuration at a specified host, and ensures that all agent and native libraries necessary for execution at the target system are transferred to the gateway server.

Scenario Name:	Deploy Agent Libraries
Overview:	Deploys the agent code library, and optionally the user interface library, for the agent in the Java code repository (JCR) at the administration server (AS) to the JCR at the gateway server, if it is not yet available there.

Scenario Name:	Deploy Native Libraries
Overview:	Deploys all required native libraries that are available to the agent in the native code repository at the administration server (AS) to the native library repository (NLR) at the gateway server (GS).

During the process of refining the scenarios, additional parts are filled in: “Basic Course of Events”, “Alternative Paths”, “Exception Paths”, “Trigger”, “Assumptions”, “Preconditions”, “Postconditions”, and “Related Business Rules”. In Sect.

4.4.1 we described the structure and expected contents. Below you can see the result of the iterative process; that is, a well-described scenario:

Scenario Name:	Deploy Native Libraries
Overview:	Deploys all required native libraries that are available to the agent in the native code repository at the administration server (AS) to the native library repository (NLR) at the gateway server (GS).
Basic Course of Events:	1. The AS retrieves descriptions of all available native libraries for the agent to be installed from the NLR at the AS. 2. The AS sends a description of each native library to the GS. 3. The GS returns a list of libraries that are missing in the NLR of the gateway server. 4. For each missing library (see the previous step), the AS retrieves the native library from the NLR at the AS (based on the description) and transfers it to the GS. 5. The GS stores each received native library in its local NLR and acknowledges proper receipt of the library to the AS.
Alternative Paths:	None.
Exception Paths:	Throughout the scenario, any errors are logged and the system tries to carry on with the installation process. If any missing library cannot be installed properly, the GS signals an error to the AS.
Trigger:	The AS receives the request to deploy the native libraries of a specific agent to a specified target system.
Assumptions:	» The GS is up and running. » No version conflicts can occur, as the version information for a native library is coded into the library name.
Preconditions:	» All native libraries required by an agent are at least available for one platform (e.g., the Win32 platform).
Postconditions:	» Each library whose proper deployment was acknowledged by the GS is guaranteed to be available in the NLR of the GS.
Related Business Rules:	» Naming conventions for native libraries: Each native library name consists of the name of the library plus a version number. The version

	number is obligatory and consists of one or two digits. The first digit represents the major release number, and the second digit the minor release number. The library extension terminates the name of the library. The library extension has two to four letters and is separated from the rest of the name by a dot (e.g., “dll” is a valid extension for a Win32 library). » Compatibility of native libraries: Each native library is backward compatible with any library with a lower minor version digit. Native libraries with differing major version digits are incompatible.

The understanding gained during elicitation and documentation of the use cases enables us to develop an analysis model; that is, a class diagram that shows the main elements of our problem. The analysis model is shown in Fig. 62. On the basis of the knowledge that is now available, we can define a number of initial contracts for this system:

```
Agent
inv: javaArchive != null
inv: nativeLibrary.canLoad()

GatewayServer::storeNativeLibrary(NativeLibrary lib)
pre: !nativeLibraryRepository.contains(lib)
post: nativeLibraryRepository@pre +1 = nativeLibraryRepository.size()
post: nativeLibraryRepository.contains(lib)
```

These activities are all carried out in parallel and must be accompanied by the quality assurance tasks as shown in Fig. 39; that is, quality assurance of scenarios and of guidelines. It makes sense to describe some selected scenarios in more detail using interaction diagrams (Fig. 35 and Fig. 36 are examples of interaction diagrams for the installation of an agent and for the deployment of native libraries).

6.3.2 Contracts and Scenarios for Design

During analysis and requirements elicitation, the emphasis of the activities is on understanding and capturing the problem. In design, a general architecture (e.g., client–server) and an associated design are developed that consider and reflect all functional and nonfunctional requirements. Important aspects to be considered during design are [Jacobson99] as follows:

- Acquisition of an in-depth understanding of the requirements and technologies that are relevant for this project.
- Decomposition of the implementation work into manageable segments that can

be handled by different implementation teams.
- Definition of the major interfaces between subsystems.

The Role of Contracts: As contracts partly capture functional requirements, it is important to transform the contracts developed during analysis (see the previous section) into design. The design process itself is iterative in nature, and starts with the analysis model, which is transformed step by step into the physical design, with different class names, additional or changed interfaces, divided functionality, and so on. Each change in the design model also has to take into consideration the changes in the contracts. Consideration of the contracts in redesign tasks ensures that the design model remains compatible with the analysis model (from the point of view of the behavioral specifications).

The Role of Scenarios: While contracts are a good basis with which to ensure that the specification of selected classes and interfaces is adhered to, scenarios allow us to reflect whether the developed design model can be used in the context of a specific application, where core parts of the functionality (in the sense of how to use the system) are captured in the scenarios.

Methodological Issues: Figure 40 sketches the principal process during design, with an emphasis on contracts and scenarios.

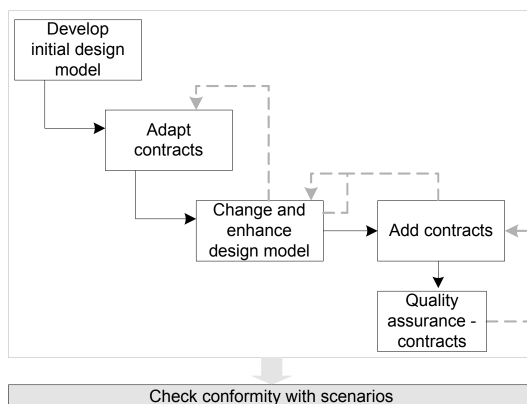

Fig. 40. Contracts and scenarios for design

On the basis of the analysis model, an initial design model has to be derived. In the course of deriving the design model, the contracts have to be adapted to this model—usually by means of simple adaptations, such as name changes, the division of contracts, and so on. Obviously, this task can also be carried out step by step, the contracts being adapted systematically in each step.

During the changes to the design process, enhancements and more details are added to the design model. This is a highly iterative approach (depicted explicitly in Fig. 40) and it naturally implies changes in the contracts. Additionally, by gaining a deeper understanding of the problem domain, additional contracts can be added to the existing ones. This is also an iterative approach, and it has to be ensured that added contracts do not conflict with the additional ones developed during analysis. Quality assurance for contracts is an important issue. In Chap. 3 we developed a number of guidelines for the development of contracts. It has to be assured that the contracts adhere to these guidelines. These quality assurance tasks definitely have a loop back to the "add contracts" task. We will model this explicitly in order to emphasize it better.

In parallel with these activities, the designers always have to check whether their changes conform to the requirements for the system. Parts of the functional requirements are captured by means of scenarios. The designers therefore always have to check whether their design is valid in the context of the developed scenarios.

RUP/Scrum Issues: The role of contracts and scenarios during design is the same as depicted in Fig. 40—regardless of the type of process followed. The RUP, as well as the Scrum approach, are iterative by nature, which means that any changes applied to the design of the system have to be checked in terms of contracts and scenarios. This is even more important in the Scrum approach, as an entire Sprint—in which, for some reason, design decisions might be lost—usually takes place between two design enhancements: as long as contracts were formulated for these decisions, they won't be lost after all.

Example: Figure 62 shows a typical analysis model of our problem domain. During design, this model probably will be changed. Figure 41 shows portions of a design model based on the above-mentioned analysis model. Note that we have only presented one small portion of the design model—the overall design model would be far more complex.

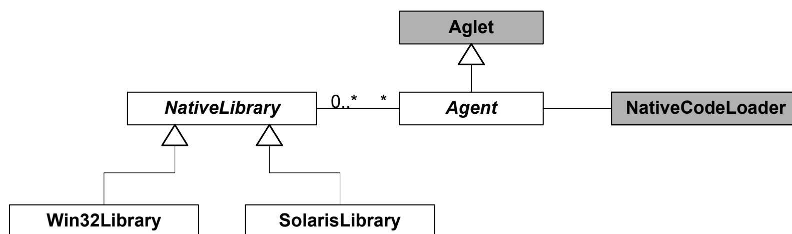

Fig. 41. A partial design model of an agent system

In this design model, the Agent class is now a subclass of an Aglet class [Lange98]; that is, an agent of a specific agent platform. Furthermore, each Agent references a

NativeCodeLoader that is responsible for loading a required native library from the GatewayServer (as described in Appendix B, our deployment process ensures that all required libraries are at least available at the GatewayServer). During the analysis process, we developed (superficially) some contracts for the Agent class:

```
Agent
inv: javaArchive != null
inv: nativeLibrary.canLoad()
```

From a design point of view, the Agent class does not necessarily hold a reference to a nativeLibrary at any time, but this association is established when the agent needs this library for the first time during execution. In this event, the library is loaded by the NativeCodeLoader (either directly from the NativeLibraryRepository at the Host or from the NativeLibraryRepository at the GatewayServer). Additionally, we can also say that a native library (once loaded) may not be altered. These insights can be captured using the following contracts:

```
Agent
inv: javaCodeArchive != null
inv: nativeLibrary != null implies nativeLibrary.canLoad(platform)

Agent::setNativeLibrary(NativeLibray library)
pre: nativeLibrary = null

NativeLibrary
inv: libraryName != null
inv: platform != null

NativeLibrary::canLoad(PlatformDesc platform)
post: result = true implies getPlatform() = platform
```

The precondition in the setNativeLibrary method ensures that this method can only be called once (assuming that no other method resets the nativeLibrary variable to null—but even in this erroneous case, it would be acceptable to reset the nativeLibrary). We could also capture additional assertions in the newly introduced NativeCodeLoader class:

```
NativeCodeLoader::getNativeCode(String name, String platform)
post: result != null
```

This postcondition reflects the fact that a NativeCodeLoader must always find a valid NativeLibrary for a given name and a given platform. This is a restrictive requirement for the getNativeCode method, and it could not be accomplished by the method in a general way. However, in our specific context the postcondition is valid and correct. This example also shows that the contracts expressed in a class need not necessarily be independent of any context. The higher the potential for reusability, the less specific are the assertions formulated. On the other hand, specific contracts describe the specification of a class much better and therefore provide more “guarantees”.

6.3.3 Contracts and Scenarios for Implementation

The general goal of the implementation phase is to transform the derived design in such a way that the individual system components are executable in a certain environment. Typical activities during implementation are (1) refinement of design and algorithms, (2) coding, and (3) testing at the unit level. The major artifacts developed during implementation are the source code of the developed classes, as well as the test classes needed for unit testing. Additionally, the results of the unit-testing activities must also be available.

The Role of Contracts: Contracts play an important role, as they guide the implementers during implementation and unit testing. In the implementation phase, invariants and postconditions are important, as each implementation has to show that it meets these assertions.

The Role of Scenarios: In our opinion, scenarios play a minor role during implementation (compared to the role of contracts). They act as a source of clarifications concerning functional requirements.

Methodological Issues: Figure 42 sketches the principal process during implementation, with an emphasis on contracts (scenarios are not included, as they do not have a distinctive role).

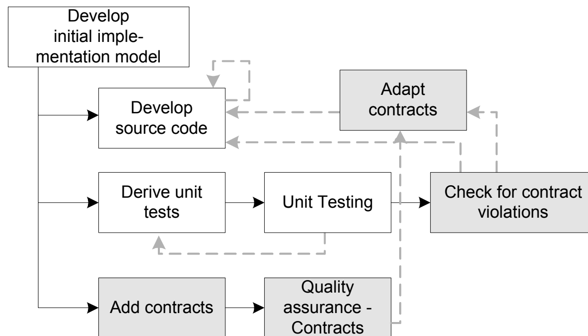

Fig. 42. Contracts and scenarios for implementation

On the basis of the design derived in the design phase, an initial implementation model has to be derived. In our case, the implementation model not only contains the set of classes developed during design but also the contracts initially captured during analysis and refined during design. Unit tests are derived before, during, or after the development of the source code for a unit. Both activities (developing the source code and deriving the unit tests) should be contract-driven:

- The implementation can rely on the contracts—other sources of information (scenarios and requirements documents) also have to be considered. Nevertheless, the implementer need not handle situations that are explicitly captured by the preconditions, although he or she always has to ensure that the implementations do not violate any invariants of the postconditions.
- The contracts specified for a unit (e.g., a class) are an excellent source for deriving test cases. Every precondition reduces the possible combinations of input values and should therefore be used carefully in order to check whether the implementation conforms to the specification in general, and to the postconditions and invariants in particular.

The unit-testing activities may reveal errors that indicate inadequacies in the implementation, as postconditions or invariants are violated. If this is the case, the source code has to be changed accordingly to reflect the correct behavior (in the sense of the specified contracts) of the class. It may also happen that the implementation is acceptable but the specified contracts are wrong. Therefore, it is also necessary to adapt (alter or enhance) the contracts in order to better reflect the behavior of a class.

Additionally, it will always be the case that while developing the source code and deriving the unit tests, new contracts come to mind as the implementers gain a deeper understanding of the problems to be solved. Obviously, these contracts should be added to the existing contracts, insuring that they comply with the quality requirements for contracts and that they do not conflict with existing contracts.

RUP/Scrum Issues: The role of contracts and scenarios during implementation is the same as depicted in Fig. 42—regardless of the type of process followed. The RUP, as well as the Scrum approach, are iterative by nature, which means that any changes have to be checked in terms of contracts.

Example: During the design process, we developed some contracts for the Agent class:

```
Agent
inv: javaCodeArchive != null
inv: nativeLibrary != null implies nativeLibrary.canLoad(platform)

Agent::setNativeLibrary(NativeLibray library)
pre: nativeLibrary = null

NativeLibrary
inv: libraryName != null
inv: platform != null
```

On the basis of this information, the implementer has to ensure that the constructors of the Agent class always set the nativeLibrary instance variable to null. Additionally, we can use the precondition information to construct proper test cases for unit testing.

6.3.4 Contracts and Scenarios for Testing

The goal of testing (beyond ordinary unit tests) is validation of the system, and the detection of as many errors as possible. Integration tests and system tests are the most important types of test.

According to Jacobson et al. [Jacobson99], integration test cases are used to verify that the components interact properly with each other after they have been integrated into a build. Most integration tests can be derived from scenarios, as they describe how the entities of the system interact.

System tests [Jacobson99] are used to test whether the system functions properly as a whole. Each system test primarily tests combinations of scenarios under different conditions. These conditions include different hardware configurations, different levels of system loads, different numbers of users, and so on.

The Role of Contracts: As already mentioned in the introduction to this section, contracts play a subordinate role, as the driving force for developing integration and system tests is the developed scenarios. Nevertheless, the specified contracts support this test process. In case of thorough unit testing during implementation, violations of postconditions should not occur. However, as the individual classes are now tested together, it will happen that some classes violate preconditions, or that special (valid) constellations occur that lead to errors in the postconditions or invariants. In the case of violations of preconditions, the caller has not fulfilled its contract, while in case of postcondition violations the callee has not fulfilled its contracts—therefore the sources of the errors can more easily be traced. It will also happen that errors occur that are not covered by existing assertions. In this case, it has to be decided whether new assertions should be included in order to capture these error situations.

The Role of Scenarios: Scenarios are the primary source for composing interaction tests. Regardless of the representation, the scenarios are used to construct test scripts. Standardized descriptions of scenarios (e.g., using interaction diagrams) facilitate this process. Typically, type scenarios (see Sect. 4.2.4) are used—therefore, from one scenario, a number of test scripts that are suitable for integration or system testing can be extracted.

Methodological Issues: Figure 43 sketches the principal process during testing.

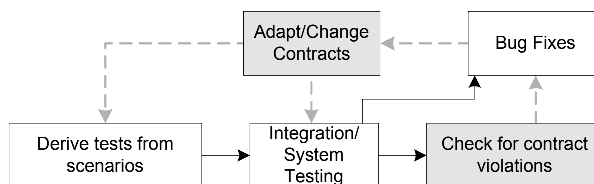

Fig. 43. Contracts and scenarios for testing

The scenarios have to be worked through systematically, in order to derive tests from scenarios for integration and system testing. Typically, one scenario yields a number of different tests. The integration and system testing activities will reveal errors in the implementation. In the case of contract violations, either the contracts have to be adapted (due to incorrectly specified assertions) or the implementation has to be changed. Of course, errors can also occur that are not related to any contract. In these cases, the errors have to be fixed and possibly new assertions have to be added that better reflect the requirements associated with the discovered errors.

RUP/Scrum Issues: The role of contracts and scenarios during implementation is the same as depicted in Fig. 43—regardless of the type of process followed. The RUP, as well as the Scrum approach, are iterative by nature, which means that any changes have to be checked in terms of conformance with the underlying specification and the specified contracts.