

Chapter 2

Software Product Lifecycles: What Can Be Optimized and How?

Abstract The chapter discusses lifecycle models for software development in more detail. These include build-and-fix, waterfall, incremental, object-oriented and spiral. We present a more detailed description of the lifecycle models application for software development. We compare benefits and shortcomings of the models discussed. We confirm that there is no “silver bullet”, i.e. a universal lifecycle model equally applicable to any software product. Consequently, lifecycle model choice is dependent upon product size and scope; each project requires a unique combination of features. In crisis, we recommend to combine prototyping with the other models that we discussed in order to achieve a common understanding of the key product features and to reduce project risks. The lifecycle model choice determines project economics, time to market, product quality and overall project success. However, the product success essentially depends on human factors, which include common vision of the critical product functions, transparent communication and feedback. We analyze applicability of the lifecycle models to large-scale, mission-critical software systems, which is essential in crisis. Finally, we introduce a methodology, which includes a spiral-like lifecycle and a set of formal models and visual tools for software product development. The methodology helps to optimize the software product lifecycle, which is mission-critical in crisis. The methodology is applicable to large-scale, complex software products for heterogeneous environments.

Keywords Software lifecycle · Lifecycle model · Software development methodology

1 Introduction

The previous chapter gave a brief review of a number of lifecycle models used in software development, such as build-and-fix, waterfall, incremental, object-oriented, spiral, and a few others.

This chapter presents a more detailed description of the lifecycle models application to software development. It includes discussion of their benefits and

shortcomings. It analyses applicability of the lifecycle models to large-scale, mission-critical software systems, especially in a crisis.

Some of the models are more straightforward, others require a number of iterations. Our deeper investigation of the models will still conclude that there is no crisis-proof “silver bullet” for lifecycle models. However, we will arrive to certain recommendations of combining and adjusting the models in order to succeed in crisis software development.

Project success is usually determined not only by the lifecycle model, or by a combination of models, but also by a number of human factors, which may help or hinder a common understanding of the key product features by the client and the developer. We will cover these human-related factors in more detail in Chap. 5.

In order to optimize software product lifecycle, which is mission-critical in crisis, we will introduce a methodology that includes a spiral-like lifecycle and a set of formal models and visual computer-aided tools for software product development.

This chapter is organized as follows. Section 1 discusses the abbreviated and straightforward lifecycle models, such as build-and-fix and waterfall. Section 2 presents an overview of simple iterative models, such as incremental and prototyping. Section 3 describes more complex software lifecycle models; these are spiral, synchronize and stabilize, and object-oriented. Sections 4 and 5 contain an overview of an enhanced software development methodology, which provides lifecycle optimization and sequentially elaborates the deliverables for mission-critical software products in crisis. The conclusion summarizes the results of the chapter.

Let us have a look at the software development lifecycles in more detail.

2 Simple Lifecycles: Brief and Straightforward

One of the models of software development lifecycle discussed previously is the build-and-fix (see Fig. 1). This is a model of incomplete lifecycle. Because of its simplicity, the build-and-fix is not suitable for large and complex projects, which have a size of over 1 KLOC. So, the build-and-fix model may be a possible option for a software solution, which is downsized by crisis. However, it is only applicable in case of a trivial product with clear requirements.

The other model discussed previously is rapid prototyping (see Figs. 4 and 5). It is also somewhat limited, despite the fact that it includes all the necessary stages of the lifecycle. These are analysis and specification of requirements, preliminary and detailed design, implementation, unit testing, integration, product testing, maintenance, and retirement. The limit of the rapid prototyping is lack of self-consistency. Actually, its testing phase, both for the individual modules and the prototype as a whole, yields to a low quality code. The prototype documentation is usually insufficient and incomplete, and the resulting code is not a software product, since it only simulates the key functionality and certain aspects of the future software system of operational quality.

Fig. 1 Build-and-fix model

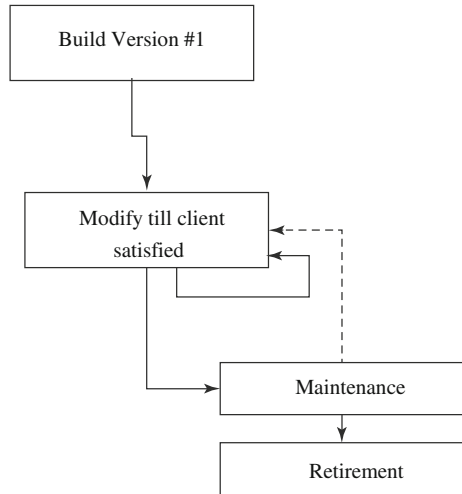
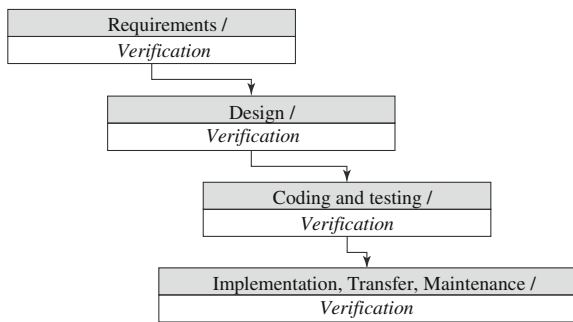


Fig. 2 Waterfall model



The waterfall model presented in Fig. 2 is fully applicable to large-scale and mission-critical software systems; however, it has some limitations, since it has a limited agility to meet the crisis conditions. In particular, the waterfall model requires discipline and organization, operational knowledge of CASE tools as the project team needs to produce a large number of documents and to communicate intensively. The documentation should meet the standards, which follow the agreement with the customer. Any product document, such as operations manual, should follow specific templates, since documentation is a critically important part of any waterfall-based software product. Let us recall that the product is not only the code but also a large amount of documentation required for competent and stable maintenance. The product documentation has a special value for the maintenance personnel, as they usually read the code produced by other developers, and their task is to detect and to fix the remaining defects. The documentation is mission-critical for the waterfall model, because developers follow the lifecycle based on document-driven milestones conditions. For example, as soon as the

required detailed design documents for the product are ready, the milestone is reached and the developer proceeds to the implementation phase of the lifecycle.

The following three models (see Figs. 3, 4 and 5) are important to understand how to organize the lifecycle of software systems, including large-scale and mission-critical anti-crisis solutions. In contrast to the waterfall pattern, these three models are focused on multiple passing through the stages of the lifecycle. The product functionality is usually incremented after each pass.

Fig. 3 Waterfall-based V-model

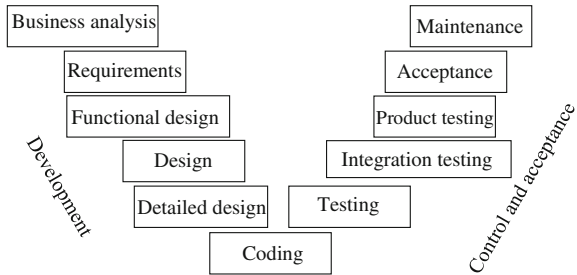
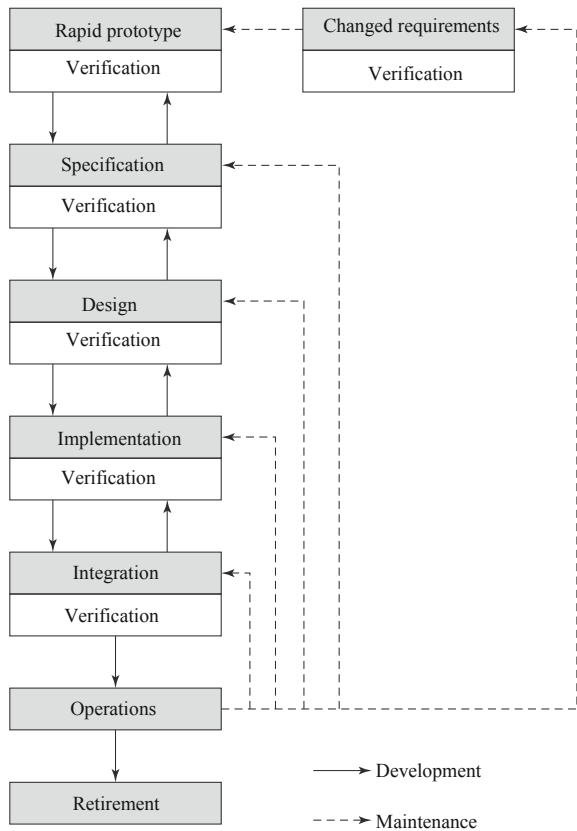


Fig. 4 Rapid prototyping model



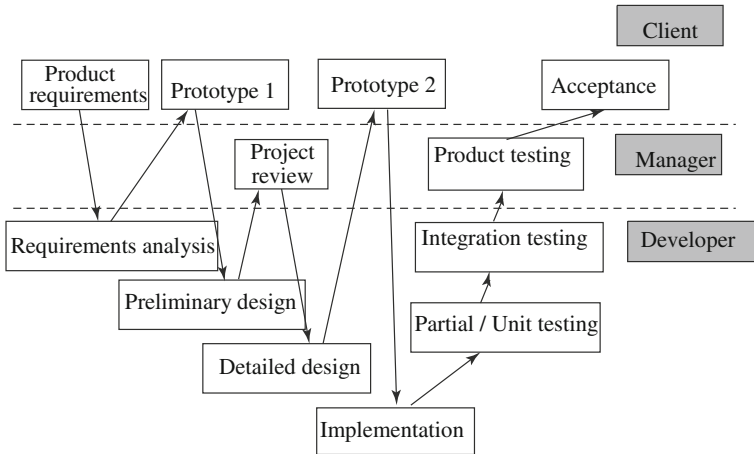


Fig. 5 Rapid prototyping-based “shark teeth” model

Naturally, it is challenging to implement certain kinds of the software systems in a single pass. However, the waterfall is adequate for a problem domain with clear and stable requirements, which is straightforward to document and design. The waterfall approach is applicable mostly for government agencies and military software applications.

3 Simple Iterative Lifecycles: Incremental and Prototyping

The iterative lifecycle models provide loop-based elaboration of the software product. They assume that several loops are required in order to build a product release. Each of these loops usually includes all stages of the software product lifecycle. The iterative models are often easier to follow in terms of discipline, as they do not usually require developing a full product functionality or a complete product documentation after each stage.

One of these iterative models is the incremental model (see Fig. 6). What are its key features? According to the model, while building a project plan, the product is divided into a sequence of releases. The lifecycle stages, which precede product transfer to client, involve a number of releases. These are iterations of the development cycle, each of which provides an operational product. Therewith, in case the product does not require a revolutionary transformation of the previous releases (i.e. functionality builds up smoothly), each release is transferred to the customer as an operational product, though it has a limited functionality. In addition, every lifecycle stage delivers the required product documentation, so that each product release is operational and utilizable.

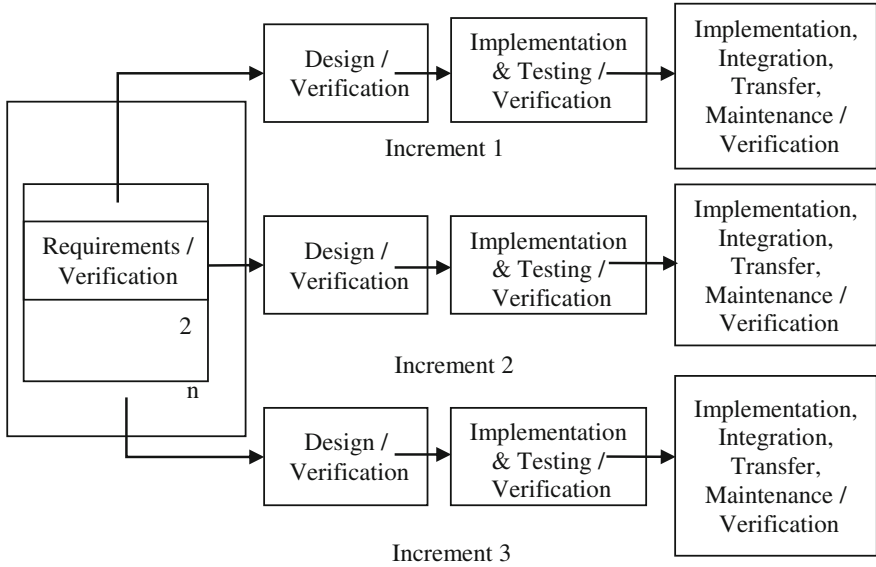


Fig. 6 Incremental model

For instance, in case of an online store, we can initially simplify the interface associated with the purchase of products. The first release may not have a choice of delivery options (e.g. by sea and by air), and it may have a single kind of delivery with a fixed rate. Later releases can also include more details to support credit card payments, such as a dedicated server for client authentication and transaction processing. However, even the first release yields to a fully operational product, though it is rather simple in terms of functions available.

Thus, the idea of the incremental lifecycle is to supply an operational product to customer as soon as possible. In some cases, this can be a suitable solution in terms of crisis management. For the incremental model, the project plan typically specifies the sequence and schedule of functionality transfer to the customer; it may also include a maintenance plan, which specifies technological and functional constraints for each release.

Another feature of the incremental model is a relatively smooth transfer of new functionality. Each release is a clearly separated functional block, and it contains a number of modules. Naturally, these modules do not exist by themselves and are related to some other modules. They can inherit certain properties of these other modules; they also can interact with semantically related modules through the interfaces provided. Thus, it is desirable that within every release, each interacting module is relatively small and self-consistent, i.e. it has a relatively small amount of interaction points with the other modules. By keeping the modules and the releases relatively small and self-consistent, the incremental lifecycle provides a smooth transfer of the new functionality to the customer.

Modular software design ensures minimum connectivity between the modules, so that each relatively small and functionally separate task is located in a separate software module. The same modularity principle usually holds true for each of the incremental releases. However, since the functionality is implemented and introduced gradually, the new modules and releases will interact with the existing ones, so we need to test their interfaces. Therefore, if the product requires a revolutionary functional change, which significantly influences its previous releases, it usually causes a number of problems. Thus, we can get a local crisis in development instead of a stable incremental release plan.

The higher-level source of such a local crisis can be poor design and inadequate planning. As for the lower-level sources, we can identify at least two of them at this point. First, there is a significant problem associated with inheritance. It may happen that a number of modules in the operating release is to change in a significant way. Therewith, we have to redesign and rebuild a significant percentage of the previous release structure, and to rewrite all the documentation required. Of course, any new release brings certain changes to the previously built modular structure. However, with one revolutionary functional update, we have to make such a large number of changes in design and implementation, that it nearly nullifies all the efforts to produce the previous releases. In fact, this local development crisis is comparable to complete redevelopment of the product from scratch in a build-and-fix manner. In this case, the functionality developed for the previous releases would be largely rebuilt, and the time and labor to create this functionality would be lost. In this respect, revolutionary development typically results in a local crisis of an incremental lifecycle.

An incrementally developed product should have a scalable architecture in terms of release updates. For example, a web service-based component architecture usually scales up well in terms of adding new modules or expanding existing ones. However, there are software architectures, such as a file server, that do not support similar scalability equally well. Therefore, we should consider the features of a particular model at the early stages of project planning and high-level architectural design in order to adapt to the technical constraints and to avoid a local crisis in product development.

For the developers, the incremental model provides evolutionary interaction with the customer and greatly simplifies their relations, because the core modules that implement the business logic of the application often vary slightly. The new releases only add functionality. Therewith, maintenance of an incremental product is usually sufficiently smooth and relatively inexpensive.

However, a possible disadvantage of the incremental model is that it is not suitable for quite a number of software products, which initially require a full-featured implementation. Let us assume that there is a number of customers, who need a full-featured online store, which includes a 3D catalog, credit card payment support, and a variety of electronic payment gateways. Certain clients would also ask to monitor delivery, as it is implemented at their competitor portals, and it is convenient and useful. If the product initially requires full functionality, we should probably consider some other model, such as waterfall, which allows a

single pass implementation. Of course, in case of waterfall certain project risks are higher; however, they will be discussed further in relation with the spiral model, which is designed to deal with them. Therefore, there is a number of constraints for the incremental model, and it is clear that this model it is not suitable for every product.

Another drawback of the incremental model is that the resulting software product should provide a stable upgrade path for its development. That is, the efforts spent for functional updates with each product release should clearly exceed the redundant efforts for the high-level release reconfiguration. Such reconfiguration efforts should not have a significant negative impact on the performance of the next product release. The incremental model does not support a revolutionary, unstable path of the software upgrade; it also has no mechanisms for risk assessment.

Depending on customer or market constraints, a number of software products requires revolutionary changes in the product concept itself, such as fundamental principles that underlay the functional requirements, project plan and product release policy. If the customer requirement changes are frequent, spontaneous and dramatic, and there is no way to adjust these requirements so that they become evolutionary, it may turn out that every other release the developer has to create a new product almost from scratch rather than to reuse a significant portion of the previous one. Thus, the incremental approach degrades to build-and-fix. Moreover, in contrast to build-and-fix, which is an incomplete lifecycle model, the developer has to re-implement the entire lifecycle for each release. For each release, the developer has to specify complete requirements, to do the software design, i.e. to produce a large number of diagrams, including data flow diagrams, use cases, class diagrams etc. The developer also has to develop a new test plan, including product testing scenarios and their sequence, acceptance test cases and a number of other artifacts. Moreover, the end user and administrator documentation requires significant changes. The new product probably has a different setup procedure, user interfaces, usage scenarios, error codes and so on. The developer has either to rebuild all these documentation artifacts or to create new ones. Thus, the developer has to rework the product using a more bulky and complex approach than a trivial build-and-fix, which includes a full-scale documentation and artifact reviews for each software lifecycle phase. Therefore, the software production is likely to result in a local crisis, since it involves a huge amount of bulky overheads. Thus, the incremental model is unacceptable for a product that quickly goes beyond the original concept, no matter how large the product is.

In case of a predictable upgrade path of the product, the previous release is naturally included into the next one. At the same time, a special document, release notes, is issued, which includes a list of additions to the previous release. Release notes document also contains important information about the new release of the software product. It addresses customer's maintenance service, who detect, localize and fix errors; it also guides customer's end users on their moving from the previous release to the next one.

Figure 6 shows a view of the incremental lifecycle model. It is clear that each subsequent release includes the functionality of all the previous ones. Thus,

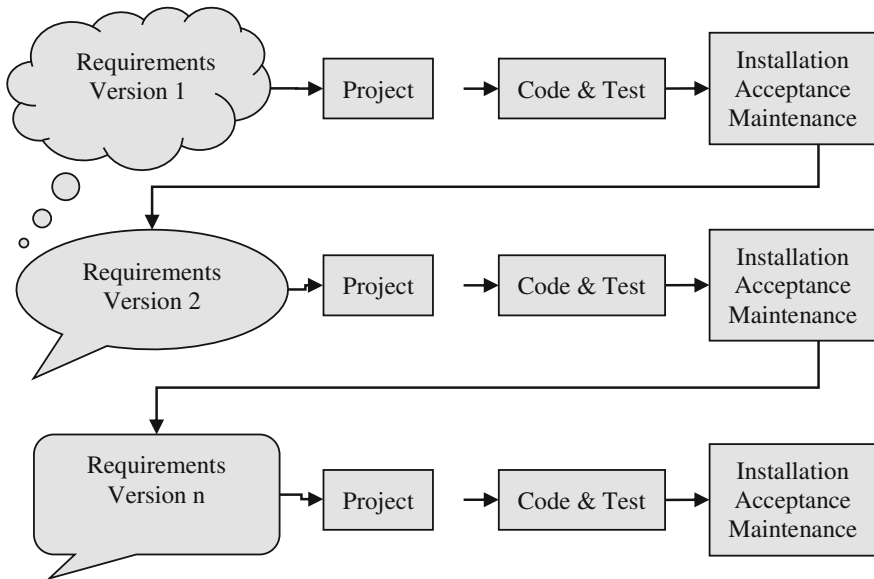


Fig. 7 Evolutionary model

functionality increases smoothly, and so that each of the following releases absorbs previous ones and adds certain new features. For incremental product development, production of the new release includes verification of all the lifecycle stages, such as requirements analysis, requirements specification, design etc. Thus, the main stages of the software lifecycle are the same for each release. The incremental model fits evolutionary introduction of product functionality.

Figure 7 shows a specific form of incremental development model, which is called evolutionary. It provides a gradual transition from the previous release to the next one; each release elaborates functionality rather than builds it up. The rest lifecycle processes are similar to the incremental model.

4 Complex Iterative Lifecycles: Spiral, Synch-and-Stabilize and Object-Oriented

Another iterative approach to software systems lifecycle is the so-called spiral model introduced by Boehm [1]. According to the approach, each iteration consists of four phases (Fig. 8):

- (1) determine the goals for product and business objectives, understand the constraints, suggest possible alternatives;
- (2) evaluate the alternatives by risk analysis and prototyping;

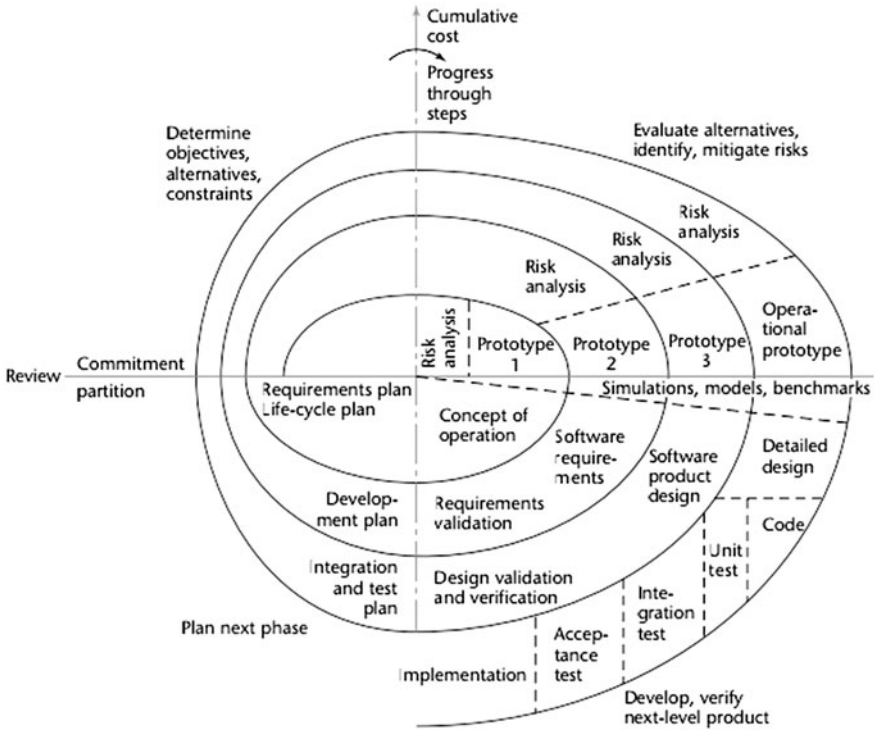


Fig. 8 Spiral model (© 1988 IEEE)

- (3) develop a product by detailed design, coding, unit testing and integration;
- (4) plan for the next iteration, including product development, implementation and delivery to customer.

The above four stages: determine—evaluate—develop—plan, are often represented graphically as a spiral.

This model is suitable for projects with significant risks. Some other lifecycle models also address risk assessment. In crisis, the projects tend to become more risky. Additional crisis-related risks may include delays of funding, communication challenges in the project team, especially in case it is distributed. In fact, in the spiral model, risk analysis happens each iteration.

Each phase of the spiral model usually repeats; there is often three to four iterations. However, the exact number strongly depends on the “convergence” of the project. In certain cases, the number of iterations is difficult to predict; it may also happen that after the risk assessment it is not feasible to continue the project. This may result in additional expenses; however, in crisis conditions it is required to recognize that the project team is not able to ship the fully functional product of the

required quality level within the deadlines. A possible solution of this problem is the contradiction management matrix-based approach; we discuss it in the next chapter.

Each cycle includes four basic phases: determine, evaluate, develop and plan.

The first phase includes an outline of the objectives for the current iteration, possible alternatives to achieve these objectives, and the constraints for each alternative. Further, evaluation of the alternatives follows; risk assessment is among the key activities. Risk assessment is a complex process; it requires specific knowledge. In crisis, risk experts often have to make decisions in case of uncertainty, insufficient resources and incomplete information. Prototyping helps to reduce some of the risks and to understand the others better; this chapter discusses prototyping in more detail below. After risks identification, risk mitigation plan follows, which specifies the ways to reduce risk consequences or to continue the project with the risks that exist. Then, the implementation phase begins, which starts from coding and testing of the functions required in the iteration, and which ends with the integration and testing of the partial product developed for the current loop of the spiral. Afterwards, based on development postmortem and the existing resources, the next loop of the spiral is planned.

Risk analysis often includes a number of significant uncertainties, which are unlikely disclosed by the customer. Risk analysis usually requires a large amount of expensive labor, so spiral model is generally feasible for large-scale projects. The spiral model is suitable for the so-called in-house projects, where the developer and the customer collaborate within same enterprise. Typically, in large corporations, there is a dedicated IT company, such as Gazprom Inform as a part of Russian Gazprom group of companies. The spiral model is a suitable solution for such enterprises, since the developer and the customer belong to the same large-scale corporation. In case of in-house development, there is usually an adequate transfer of the sensitive information required to assess the risks, and risk assessment results are reliable. Moreover, the in-house development with spiral model is cost-effective, and so this is a recommended option for crisis software development.

The spiral model is somewhat similar to iterative models, such as incremental and evolutionary. However, spiral model is fundamentally different from a number of other models because of explicit risk assessment. Certain lifecycle models can be combined with the others, especially with the rapid prototyping. Spiral model also includes prototyping, which usually assists in risk analysis and evaluation. Rapid prototyping helps a developer to discuss possible product alternatives with the customer. As compared to a full-scale software product, a prototype is relatively cheap and easy to produce. A prototype behavior is usually functionally similar to the product; however, it is limited in terms of quality attributes, such as performance, reliability, security and so on, and in terms of documentation. In crisis, we recommend to combine every lifecycle stage with rapid prototyping, including the early stages, such as analysis and design. Prototyping is a quick and a low-cost way to mitigate a number of project risks. Rapid prototyping simplifies decision-making prior to the release production, so the product transfer occurs timely even in crisis conditions, though the functionality maybe somewhat limited.

The spiral model requires risk analysis; it identifies and classifies the project risks. Developers need to mitigate the most serious project risks, i.e. they have to find a way to reduce their impact on project schedule, budget and product functionality. In case it is impossible to mitigate critical risks, the project manager may decide to terminate the project.

What are the advantages of the spiral model? First, it ensures a smooth transition of the product to the customer. Therewith, it is possible to reuse the product, even under initially high project risks, or in a crisis.

Based on risk analysis, quality assurance metrics are set. Release-based product transfer and risk assessment assist for maintainability. Despite the high costs of risk assessment, the spiral model provides a relatively cost-effective maintenance, which is the most expensive part of the lifecycle. Thus, in terms of full lifecycle the spiral model is often affordable.

The drawbacks of the spiral model originate from high costs of risk assessment. It is applicable for in-house projects.

The spiral model is theoretically applicable to relatively small projects; however, given the substantial costs of the risk assessment, it is more suitable for large-scale ones.

The spiral model requires high level of expertise in risk assessment. In case the development team has no internal risk experts, they have to hire third-party professionals.

The next model we are going to discuss is the synchronize and stabilize model, which is somewhat similar to the Microsoft Solution Framework (MSF) methodology. The next chapter gives a more detailed description of MSF. Due to significant complexity and specific knowledge, skills and CASE tools mastery required, the model is not widespread outside of Microsoft.

This is an iterative model, and the functionality is usually delivered in releases, from essentials to desired requirements, which is similar to incremental model. Each iteration includes planning, design, development, synchronization, integration and stabilization.

According to the model name, the key processes in this software lifecycle are synchronization and stabilization. The synchronization process, however, refers not only to integration of the deliverables produced by the project team, but also to product conformance checking against the requirements specification. The purpose of this phase is to detect and to record as many defects as possible, and to do this as early as possible. However, the model does not suggest immediate correction of the defects recorded in the synchronization phase. Instead, the defects recorded are prioritized by severity and fixing cost, and the list of the defects to fix is produced.

Later on, in the stabilization phase, all the defects detected previously and included into the list are fixed, and the product release for current iteration is produced. The main objective of the stabilization phase is to produce a release with a stable behavior. Therefore, each release is intensively tested in order to meet the threshold values for key quality attributes, such as performance, availability, security and so on. The model uses the idea of sequential functionality build-up;

each release results in an operational software product. The final product usually requires three to four incremental software releases.

Synchronize and stabilize processes are a part of each release. Synchronization process is followed by integration: the individual product modules developed by programmers are assembled in order to make a product release. The integration process is accompanied by frequent and extensive testing, which potentially results in a fast delivery of the product release. The stabilization process ends when all critical errors found in test are fixed.

Thus, synchronize and stabilize are the two interrelated processes, which result in software product release if performed consistently. The final step before release transfer is its “freezing”, i.e. saving its configuration.

Let us consider the advantages of the synchronize and stabilize model. First, they come from early and frequent testing. Why is this useful? We mentioned earlier that the defects in the product must be detected as early as possible. The later a defect is detected, the more effort is required to fix it. It may also happen that a defect found in one of the modules affects the operation of the adjacent modules, larger product components, or even the entire product. Furthermore, the defect fixes often crosscut through a number of product artifacts, since they affect not only the code but also the documentation. The documentation is often a crosscutting concern, since it usually influences not only the defective module but also its interaction with the other modules. Of course, it is possible to localize and fix even a logical defect of a top-level module, which is responsible for the overall business logic of the software product. However, such a fix will often influence a large number of the dependent modules and the related documentation. Thus, frequent and early testing is a positive solution and a potential advantage of the synchronize and stabilize model.

However, this advantage often has a side effect: intensive testing may lead to quite a large labor overhead, since it requires specific software, methods and skills. In this case, much time is wasted for synchronize and stabilize processes, which, in fact, do not add any new functionality. Although, in case of proper use, frequent and early testing leads to an exponential increase in product quality, since the number of errors found in testing decreases exponentially; it also provides a better maintainability and customer satisfaction.

Another advantage of the synchronize and stabilize is continuous product interoperability. This is usually guaranteed by partial testing of the product modules at their early development stage. Even before the first stabilization round is over, there exists an operational version of a partial product, which has been thoroughly tested. That is, before the release integration, each combination of potentially interactive modules has already been tested. Continuous product interoperability is vital in case of mission-critical and large-scale systems, which usually combine a huge number of modules that interact in a complex way. For example, the Oracle e-Business Suite, which is an enterprise resource planning system, contains about two dozens of subsystems for planning and management of different kinds of resources: HR, financials, documents and so on. Thus, it is quite challenging to ensure quality and efficiency of such a system without continuous product interoperability.

One more important advantage of this model is that an operational product exists immediately after the initial release. Due to frequent and early testing and continuous interoperability, the initial release is not merely a prototype, but rather an operational quality product with all the documentation required. This results in faster ROI, smoother transfer and better maintainability, which are mission-critical in crisis.

The other possible advantage of the model is that the product becomes potentially better adjustable to the crisis requirement changes. For example, we can adjust the less critical functions and even certain aspects of the low-level architecture by adapting the structure and functions of the modules for the future releases. This may help in future release adjustment, because we can adapt the later releases considering the product shortcomings in terms of architectural design and functionality of the earlier releases.

Additionally, the developer can identify requirement inconsistencies and try to resolve them with the customer in progress of early releases, long before the final release is ready. This approach can significantly reduce the redesign costs for the later releases and can be a positive solution for crisis. Customer's engagement into pre-release testing phase may become an additional source of crisis agility.

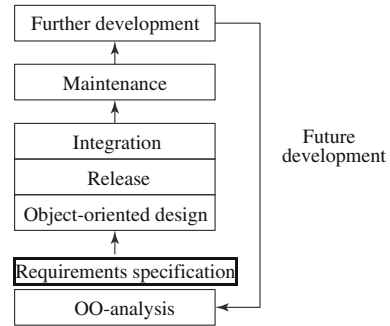
The synchronize and stabilize model is flexible and therefore potentially prospective. However, it has a number of complex processes, with the key indicators somewhat difficult to measure and control. Its major drawback is a hardly predictable amount of time for the synchronization and stabilization processes. These processes are designed to add product quality; however, they do not add any new functionality, and in case of immature team, this may result in critical overall performance dropdown.

Under the synchronize and stabilize model, the cycles of integration and testing must take place frequently; in some cases, they occur on a weekly basis. This suggests that every iteration should not only add new functionality but also synchronize and stabilize the intermediate releases, also known as builds. So, frequent build production requires not only new functionality development to match the product specifications but also comprehensive testing of the documentation and code changes with specific methods and CASE tools, in order to detect and fix defects. Therefore, the short intervals between the builds require extremely high productivity and operational knowledge of the methodology in order to be able to add new functions and to test the quality. Otherwise, the developers spend too much time in test, and they have not enough time to add the required functionality to the build. The benefits of this model are often hard to implement, especially in crisis, as they require specialized training and costly staff.

One more lifecycle model to consider, the object-oriented model, is the most dynamic and concurrent. Figure 9 shows the fountain model, a subtype of object-oriented model, which we are going to discuss further.

What are the features of the object-oriented model? The above-mentioned models contain isolated, clearly separated lifecycle stages. These are: requirements analysis, requirements specification, preliminary and detailed design, implementation and unit testing, integration, acceptance testing and maintenance, and

Fig. 9 Fountain
(object-oriented) model



retirement. Waterfall model gives most clear example of this lifecycle stages separation: every next lifecycle stage may start only after the document has been signed, which certifies that the previous lifecycle stage is complete. Conversely, the object-oriented model features intensive interaction between the lifecycle phases. Moreover, there is a significant phase overlap between requirements analysis and requirements specification, and sometimes also between analysis and design, which generally refer to separate phases of the other lifecycle models.

Another important feature of the object-oriented model is its iterative nature. Software product is produced in loops, which often allow returns to the previous lifecycle phases. For example, the phase of object-oriented design often includes a backtrack to the phase of object-oriented analysis. More specifically, analysis of scenarios of product behavior is based on use case diagrams, which are deliverables for product design stage.

Figure 9 shows that the design, analysis and specification phases, as well as design and implementation, are closely related; moreover, returns to the previous phases are possible.

What are the benefits of the object-oriented model? It fits well into the state-of-the-art object-oriented approach to software development, which has been adopted by a large number of industrial programming languages, such as C++, Java and C#. Therefore, object-oriented model is widely used in the production of mission-critical and large-scale systems. This is so because the object-oriented approach allows scalable design of software products due to inheritance and abstraction principles. Based on primitive classes, a small size product can scale up to a large and a complex one.

However, there is a number of features of the object-oriented approach, resulting from principles of inheritance and polymorphism, which may, in case of undisciplined development, lead to local crises in the design and implementation, specifically for large-scale and mission-critical software systems. In particular, concerning the use of inheritance, a bulky and complex class hierarchy may lead to such a situation that, for instance, due to an inaccurate initial problem statement, the system redesign will dramatically modify the entire class hierarchy. This is known as the “fragile” base class problem; it requires complete hierarchy redesign, including code updates for the topmost hierarchy classes that contain the high-level

logic and the problem domain-specific features. For complex problem domains, it may occur that the initial design does not scale up, and that a serious redesign of the entire “fragile” hierarchy is required. This redesign usually results in significant labor costs and product delivery delays. In this sense, such a potential benefit of the object-oriented model as inheritance may result is a local software development crisis.

Another potential source of a local crisis is the dynamic method call based on the fundamental object-oriented concept of polymorphism. The object-oriented polymorphic functions are potentially powerful and resource efficient, as they can uniformly handle heterogeneous parameters. However, these parameters are instantiated only at runtime, which means that it is impossible to test the product for all possible scenarios of the polymorphic function calls. This may result in unpredictable and severe faults that usually lead to critical product malfunctions, such as system crash, data loss, unexpected behavior with system hanging or freezing, and so on.

Thus, the object-oriented model, based on a number of promising concepts, such as inheritance and polymorphism, can degenerate into build-and-fix in large-scale and mission-critical projects, especially under lack of development discipline and organizational maturity. Conversely, well-disciplined and mature development and persistent implementation of standards for coding, testing and documenting usually help to avoid the local crises of the object-oriented model. Therewith, it becomes clear that the root cause of the crisis in software product development is largely dependent upon human-related factors.

5 Managing Lifecycles: Flexible Methodologies

In addition to lifecycle models, there is also a set of lifecycle approaches based on the flexible methodologies such as Agile (Fig. 10), Scrum (Fig. 11), and eXtreme Programming or XP (Fig. 12). Chapter 3 discusses these in more detail. Note that the processes of the software development methodologies are parallel to the phases of the lifecycle models. The methodologies, unlike the lifecycle models, are usually applicable to the projects, which feature greater uncertainty, more risk, i.e. to crisis conditions of software product development. The other aspect of the methodologies is managerial; in addition to practices of software product development they also include a number of project management techniques.

We have discussed a number of lifecycle models—build-and-fix, waterfall, spiral, rapid prototyping, incremental, synchronize and stabilize, and object-oriented—in terms of their applicability for crisis software development. The build-and-fix model is usually suitable for crisis in case of product downsizing, as it works well for small projects with a predictable development lifecycle. The waterfall model is better applicable to large-scale and mission-critical systems. However, waterfall projects require a disciplined management as they are

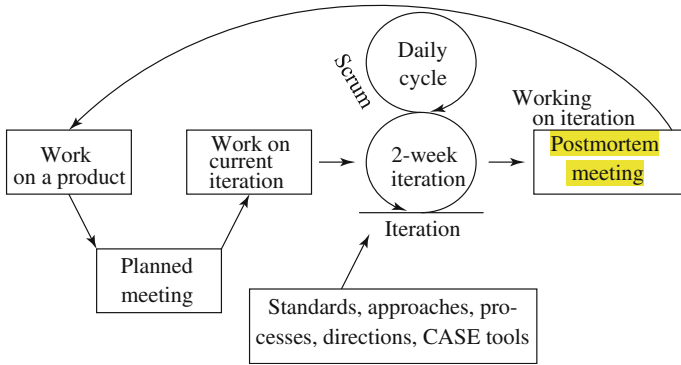


Fig. 10 Agile lifecycle

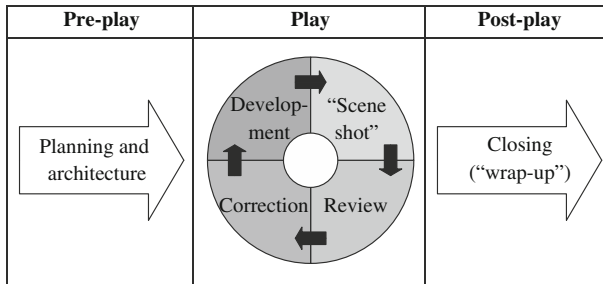


Fig. 11 Scrum lifecycle

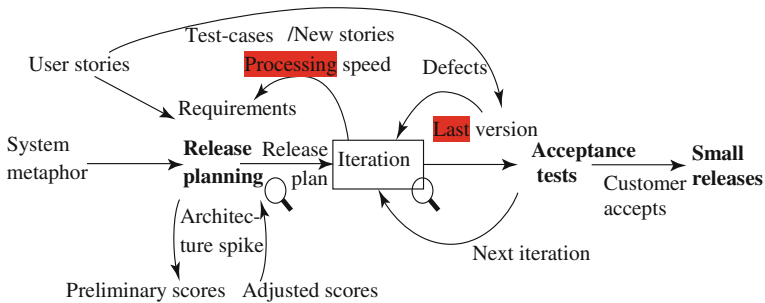


Fig. 12 Extreme programming lifecycle

document-driven. In case of crisis, due to one-pass development of the software it is quite likely that the product does not meet the requirements of the customer.

Rapid prototyping, if used “as is”, may tempt developers to reuse a quickly developed, untested, unreliable and undocumented prototype code as a product; this imposes extra risk constraints for any crisis implementation. However, prototyping potentially results in fast and economically efficient consensus on the customer

requirements. Thus, we recommend using prototypes in crisis, in combination with the other models, such as spiral or waterfall, to promote maintainability and early return on investment.

The same considerations are applicable to the incremental model, as the product can gradually update to meet the requirements of the customer. However, due to evolutionary process of incremental product development, which requires an open architecture, the model is hard to use with an innovative product as it can easily degenerate into build-and-fix. Synchronize and stabilize model is risk-based and potentially crisis-adaptive; however, it is very sensitive to specific and complex testing technologies and tools. In case of crisis conditions, the spiral model is better suitable for in-house projects, as it requires specific knowledge on risk assessment. The object-oriented model provides iteration and parallelism; it also provides a better resource flexibility and thus is essential for crisis conditions. However, under a poor discipline the object-oriented projects are likely to degenerate into an expensive and unpredictable build-and-fix lifecycle.

6 Optimizing the Lifecycle: Enhanced Spiral Methodology

Every lifecycle stage of the software system development can be optimized, including requirement analysis, product specification, design, implementation, maintenance and retirement. To optimize the lifecycle, i.e. to adapt it for crisis conditions, a complex methodology is required. This section focus is the basic outline of the optimization methodology for the product lifecycle, which includes a set of models, methods, CASE tools and practices. The methodology is process-based, and it has six stages, each of which produces certain deliverables in terms of software product components and their connectors. At the most abstract level, these are key concepts of the product and certain relationships between these concepts. Next, high-level architectural modules and interfaces follow; these are elaborated later on as classes and methods to access these classes. The lowest abstraction level is for data objects and their relationships.

The optimization methodology for the software development lifecycle is based on close integration of models, supporting methods and computer-aided tools. The models for problem domain and computing environment are built on rigorous formal theories [2–6]. The models for other lifecycle stages are more heuristic and pragmatic. Therewith, the supporting development toolkit contains both traditional CASE tools and the so-called “lower” level tools, which integrate the formal model and the software product components.

The process diagram of the methodology for optimized software product development is to a certain extent similar to the spiral lifecycle model (Fig. 13). The methodology provides iterative bidirectional component-based development of open, expandable heterogeneous software products in global environment; it supports data consistency and integrity control. Heterogeneity involves architectural and structural aspects. The architectural heterogeneity means that the methodology

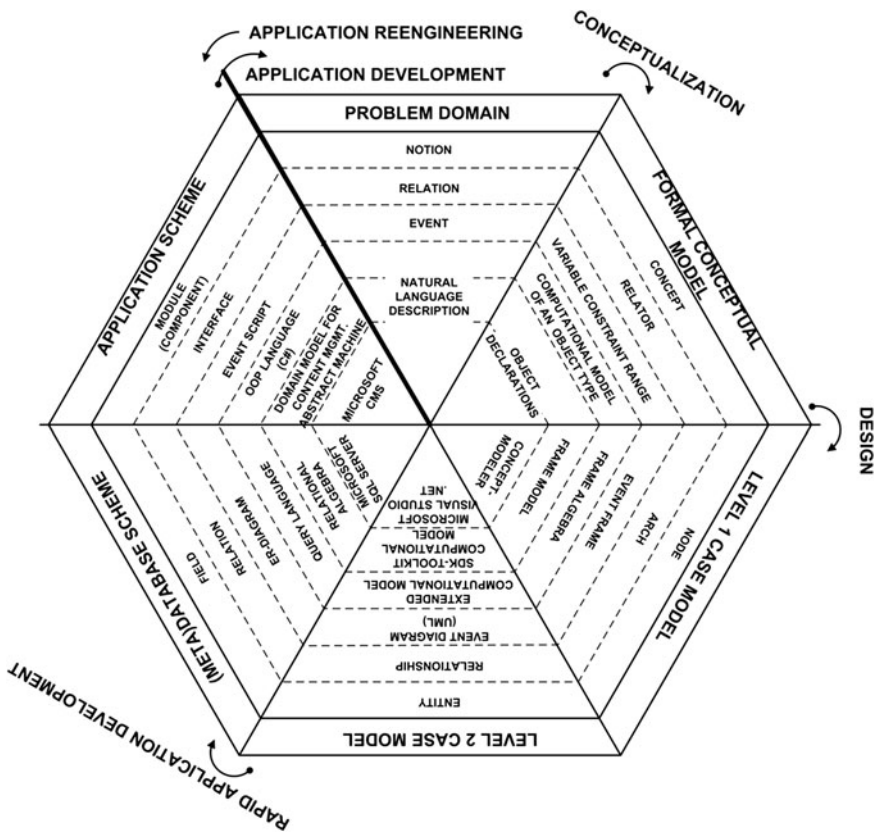


Fig. 13 Process diagram of the software development lifecycle

allows for integration of the modules or software subsystems, which are based on different architectures, such as mainframes, file servers, client servers and clouds. The structural heterogeneity means that the methodology allows for integration of the modules or software subsystems, which manage different kinds of data objects, such as relational databases, audio and video data and scanned documents.

During the software development lifecycle, the components of the heterogeneous software systems are transformed from problem domain concepts to formal model data entities. Further, by means of the software toolkit, which includes ConceptModeller [7] and content management system [8–10], the product is transformed into a complex semantic network and object-oriented warehouses managed by an abstract machine and represented by a virtual machine at the CASE level. Finally, we arrive to a well-formed layout of software product component interfaces managed by an internet portal superstructure. The development levels are elaborated in terms of entities, relationships, languages for content definition and management, and software tools.

A family of the formal object models for data representation and management supports the methodology for software development lifecycle. These models incorporate fundamental methods of finite sequences, variable domains, semantic networks and other theories [10–14].

The methodology for software development lifecycle provides the following features:

- (i) Rigorous object models of heterogeneous software products, their elements and families, and their environments;
- (ii) Integration of formal models, industry-standard technologies and CASE tools for software development by means of the innovative “middleware” tools.

Both advantages were implemented for representation and management of the integrated data and metadata.

Currently, the focus of mathematical and conceptual modeling, analysis and design of the software products shifts the lifecycle paradigm of the software development from object-oriented to pure object approach, i.e. from IT to computing. Computing is a relatively new research area; it models complex, heterogeneous, changeable and interactive problem domains in terms of objects and their environment [10].

7 Organizing the Lifecycle: Sequential Elaboration

The major purpose of the methodology is multi-factor optimization of the model for software development lifecycle, which, in crisis, is mission-critical for both product quality and project success. The key optimization factors for the software development lifecycle are: time, budget, requirements conformance and quality attributes, such as product performance, maintainability, security and the like.

Therewith, specific features of our understanding of the term “optimization” are the following ones. First, we do not mean optimization in common mathematically rigorous terms; instead, we select the best (or sometimes even a good enough) option out of a finite number of discrete values rather than a maximum of a continuous function. Second, the priority of the factors is dependent on the software project scale and scope. Third, the optimization factors are typically measurable and have certain metrics, such as number of code lines and defect removal rate. For each possible software solution, we can calculate the scenario-dependent optimization parameter values based on the above metrics and certain priorities. The resulting indicative values make the basis for better justified project management decisions, which include project plan estimates.

Naturally, in case of crisis, especially for mission-critical, large-scale, complex and heterogeneous products, it seems reasonable to use the above lifecycle methodology for data representation and management at the analysis and conceptual design stages.

We developed visual CASE tools to support the formal models for data representation and management and the processes of the product lifecycle phases, such as analysis, design, implementation, integration and maintenance. Specific workflow management tools based on document management system support the lifecycle processes for the product development. For each lifecycle phase of the product development, depending on the lifecycle model type and on the project scale and scope, these workflow management tools assist in generation and processing of certain document types, such as project plan, requirements checklist and unit test report.

Since the book is aimed at crisis management of the lifecycle processes for software product development, let us limit our scope to the overview of the methodology, i.e. models, metrics, methods, and tools, and focus on certain examples; the methodology itself is covered in more details in [9, 10, 15].

During the requirement analysis phase, optimization often results in generating requirements checklist, which is a simplified and less formal document, than the detailed product specification. However, irrespective of the type of the specification document, it should contain the lifecycle model chosen for the product development. The lifecycle model is a global parameter, which critically influences the product development plan.

The product development process is a sequential elaboration of the functional specification for the software product. In the above case, the product conceptual model is instantiated to obtain a more detailed product specification, which is elaborated later in the lifecycle. Further, we implement the architecture of the databases and other subsystems, which make the software product. In crisis, the software development lifecycle for mission-critical, large-scale, heterogeneous products is usually iterative, evolutionary and incremental, and every iteration provides further elaboration of the product functions (Fig. 13). In essence, the process outline is an improved spiral lifecycle model of software product development. However, in a number of cases, this process outline is elaborated depending on the product scale and scope or on the “project triangle” crisis optimization in terms of time, budget and functions. For instance, such a crisis optimization may result in the lifecycle reduced to a waterfall, where the software development is limited to a single pass through all of the lifecycle phases, or even to a build-and-fix model with incomplete lifecycle and simplified product documentation.

The further product lifecycle is optimized and elaborated in terms of system architecture, key technologies and development environment, which includes CASE tools and programming languages. The lifecycle optimization and elaboration process also addresses the existing software environment. The product developed should have certain quality attributes; for instance, it should be predictable, reliable, maintainable and, ideally, reusable.

In crisis, it is critically important to keep in mind that the lifecycle phase impact into the project economics is uneven. For example, the maintenance phase is the most expensive and challenging; it requires over 60 % of project time and budget [16].

Consequently, maintenance phase planning should be very accurate. However, coding contribution into the product lifecycle expenses is minimal. Consequently, coding planning should not usually take a long time. A well-justified combination of the software development methods and tools is essential for low-cost crisis development.

In certain cases, such as test termination after reaching a satisfactory error threshold, it is the project manager who makes the decision; however, the other cases, such as software retirement, usually require multi-side project evaluation.

Object-based approaches to software development often help to create interactive, distributed, open and expandable software products; they range from classical object-oriented to active objects and “pure” objects. As we know, according to the object-based approaches, the lifecycle phases of product development are flexible and have dynamically adaptive borderlines. However, even in crisis conditions, the object-based approaches require a disciplined management based on quantitative software engineering metrics and processes.

CASE tools help to validate the software in order to meet product specifications; they are often based on rigorous mathematical foundations, such as reliability statistical analysis and formal logics. Such CASE tools require a moderate level of mathematical training as they are typically designed for analysts and developers of a medium qualification level.

In crisis, essential preconditions of a product success include frequent functional prioritizing and sequential incremental elaboration.

Project specifications should be rigorous, logically correct and consistent, non-contradictory, complete in critical functional coverage, and transparently traceable.

For each lifecycle phase, software engineering requires rigorous and disciplined processes for the product development; clients and developers should strictly follow them. In crisis, developers should also follow development and documentation standards; otherwise, product development is at risk of becoming an unmanageable informal anarchy with an unpredictable result. That is why we suggest a methodology as a set of interrelated processes, methods and tools, which guides development of a requirement-matching, maintainable and high quality software even under such crisis challenges as changeable requirements, “on the fly” budget adjustments and other similar uncertainties.

The lifecycle optimization methodology is based on a thoroughly selected and tested set of models, software engineering methods and tools; it has been practically approved for developing large-scale, complex, heterogeneous and distributed software products.

The implementations of the methodology embraced a number of enterprises, such as ITERA International Group of Companies, including nearly 150 companies of over 20 countries and over 10,000 employees, the Institute of Control Problems of Russian Academy of Science, Russian Ministry for Industry and Energy, and a few others [17, 18].

8 Conclusion

In this chapter, we discussed certain lifecycle models of software development. These were build-and-fix, waterfall, incremental, object-oriented, spiral and a few others.

We also presented a more detailed description of the lifecycle models application to software development. We compared benefits and shortcomings of the models discussed. Some of the models that we discussed were one-pass and straightforward, others required a number of iterations. One key conclusion that we made was that there was no “silver bullet”, i.e. a universal lifecycle model, equally applicable to any software product. That is why the lifecycle model choice was dependent upon product size and scope, and each project required a unique combination of features. In crisis, we recommended to combine prototyping with any of the other models discussed in order to achieve a common understanding of the key product features and to reduce project risks. The lifecycle model choice determined project economics, time to market, product quality and overall project success.

Another major takeaway we made is that the product success essentially depended on a number of human-related factors, which included vision of the critical product functions, transparent communication, feedback and a few others. We cover these human-related factors in more detail in Chap. 5.

We also analyzed applicability of the lifecycle models to large-scale, mission-critical software systems, especially in a crisis.

Finally, we introduced a methodology, which included a spiral-like lifecycle and a set of formal models and visual CASE tools for software product development. The methodology was designed to optimize the software product lifecycle. This is mission-critical in crisis; we cover the implementations in more details in Chap. 4.

The methodology was applied to large-scale, complex software products and to heterogeneous environments. In the next chapter, we present more details on the product development methodologies in terms of processes, roles and artifacts.

References

1. Boehm, B.: A Spiral model of software development and enhancement. *IEEE Comput.* **21**(5), 61–72 (1988)
2. Amemiya, M., Arikawa, S., Ishizuka, M., Ueno, H., Okuno, H., Kithashi, T., Koyama, T., Saeki, Y., Shimura, M., Shirai, Y., Tanaka, H., Tanaka, Y., Tamura, K., Tsujii, Y., Tsuji, S.: *Knowledge Representation and its Use*. Ohm Press (1987)
3. Brookshear, J.G. *Computer science: An overview* (10th ed.), Addison-Wesley, 2003
4. Rosen, K.: *Discrete Mathematics and Its Application*, 7th edn. McGraw-Hill (2011)
5. MackKay, D.J.C.: *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press (2003)
6. Sipser, M.: *Introduction to the Theory of Computation*. Cengage Learning (2013)

7. Zykov, S.V.: ConceptModeller: a frame-based toolkit for modeling complex software applications. In: Baralt, J., Callaos, N., Chu, H.-W., Savoie, M.J., Zinn, C.D. (eds.) Proceedings of the International Multi-Conference on Complexity, Informatics and Cybernetics (IMCIC 2010), vol. I, pp. 468–473. Orlando, FL, USA. 6–9 Apr 2010
8. Nilsson, N.: Principles of Artificial Intelligence. Morgan Kaufmann, San Francisco (1980)
9. Sommerville, I.: Software Engineering, 9th edn. Addison-Wesley, 790 p. (2011)
10. Wolfengagen, V.E.: Applicative Computing. Its Quarks, Atoms and Molecules. Jurinfo-R, Moscow, 62 p. (2010)
11. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* **2**(8), 613–6412 (1978)
12. Minsky, M.: A framework for representing knowledge. The psychology of computer vision. In: Winston P.H. (ed.) McGraw-Hill (1975)
13. Barendregt, H.P.: The lambda calculus (rev. ed.), *Studies in Logic*, 103, North Holland, Amsterdam (1984)
14. Cheney E.W., Kincaid D.R.: Numerical Mathematics and Computing, 6th edn. Brooks/Cole (2007)
15. Ziegler, C.: Programming System Methodologies. Prentice Hall Inc, Englewood Cliffs, N. J. (1983)
16. Zykov, S.V.: Enterprise content management: bridging the academia and industry gap. In: Proceedings of the International Conference on Information Society (i-Society 2007), vol. I, pp. 145–152. Merrillville, Indiana, USA. 7–11 Oct 2007
17. Zykov, S.V.: The integrated methodology for enterprise content management. In: Proceedings of the International of the 13th International World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2009), pp. 259–264. Orlando, FL, USA. 10–13 July 2009
18. Zykov, S.V.: An integral approach to enterprise content management. In: Callaos, N., Lesso, W., Zinn, C.D., Zmazek, B. (eds.) Proceedings of the International 11th International World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2007), vol. I, pp. 212–216. Orlando, FL, USA. 8–11 July 2007



<http://www.springer.com/978-3-319-42965-6>

Crisis Management for Software Development and
Knowledge Transfer

Zykov, S.V.

2016, XXIII, 133 p. 37 illus., 13 illus. in color., Hardcover

ISBN: 978-3-319-42965-6