Bearbeitet von
Vittorio Cortellessa, Antinisca Di Marco, Paola Inverardi

1. Auflage 2011. Buch. xii, 190 S. Hardcover
ISBN 978 3 642 13620 7
Format (B x L): 15,5 x 23,5 cm
Gewicht: 473 g

Weitere Fachgebiete > EDV, Informatik > Software Engineering
Zu Inhaltsverzeichnis
schnell und portofrei erhältlich bei

beck-shop.de
DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung beck-shop.de ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.
Chapter 4
Software Lifecycle and Performance Analysis

This chapter is aimed at illustrating performance modeling and analysis issues within the software lifecycle. After having introduced software and performance modeling notations, here the goal is to illustrate their role within the software development process. In Chap. 5 we will describe in more details several approaches that, based on model transformations, can be used to implement the integration of software performance analysis and software development process.

After briefly introducing the most common software lifecycle stages, we present our unifying view of software performance analysis as integrated within a software development process (i.e. the Q-Model). Without losing generality we consider the traditional waterfall process as a reference software process. However, many considerations introduced in this chapter can be exported to other process models.

4.1 Software Lifecycle

A software process is a set of activities that are suitably combined in order to produce a software system. Different set of activities and different ways of combining such activities lead to different software processes. However, there are some fundamental common stages that can be identified in every software process, where each stage includes a set of well-defined activities. In practice such stages identify different abstractions or maturity levels of the software under development.

Requirement specification focuses on the functionalities of the system and on its operational constraints. At the end of this stage, all the functionalities of the software system and the constraints on its operation are identified and specified. In this stage customers and software engineers collaborate to produce a document collecting all the requirements of the system. Such a document can be the basis of a contract among customers and developers since it defines the software application the developers have to produce for the customers.

Software design and implementation deals with the production of the software system according to its specifications. During this stage several models (or, more
generally, artifacts), describing the system at different levels of details, are produced. Typically they are architectural models and low level design models. The implementation can be obtained through a refinement process of such models.

Software verification and validation is a stage aimed at (more or less formally) proving that the software system conforms to the requirements and constraints identified in the specification stage, and at demonstrating that the system meets the customer expectations.

Each of these stages produces one or more software artifacts that represent the software system. Also, in each stage a set of activities that operate on the artifacts can be devised, in order to achieve the expected development process results. In the next section we detail these stages within the framework of a software process model.

With the recent progresses in the software development processes, the lifecycle time after the software has been deployed is becoming ever more crucial. Software evolution is the stage that manages the changes to the software product. It starts after the delivery of the software system since software is ever more subject to changes required, for example, by evolving needs of the customers or by changes in the running context/environment. However, for the sake of readability, we do not deal with software evolution in this chapter.

Many software processes exist that combine such stages in different ways. Different types of software system may need different software processes. Moreover, each industrial organization might have its own software process that it follows during software development. In order to describe the software lifecycle a software process model is used. In the following we briefly mention two of them and we refer to classical software engineering books for a comprehensive presentation of the topic [112, 56].

The waterfall process model organizes and details the lifecycle stages sequentially, as shown in Fig. 4.1, where we limit the illustration to the common stages described above.

Another popular software process model category is the iterative one. They carry on the specification, implementation and validation activities concurrently in order
Fig. 4.2  An iterative process model

to quickly produce an initial version of the software system that can then be refined through iterations, as illustrated in Fig. 4.2. This kind of development process model has been recently promoted by the so-called agile development community [78].

For our purposes it is important to note that, independently of the considered software process, for each stage there can be one or more analysis tasks that concern the software artifacts involved in the stage. These analysis tasks are either specific tasks as part of the main stage or part of the overall software validation process. Let us, for example, consider the requirement stage. During the requirement stage there can be an analysis task that allows for improving the elicitation and the understanding of requirements as well as their correctness and completeness. Once the requirements are specified there can also be an analysis task that aims at validating the set of specified requirements with respect to the customer expectations.

4.2 Performance Analysis Within the Lifecycle

The aim of this section is to couple performance analysis with the development lifecycle, thus sharing the classical view of Software Performance Engineering of addressing performance concerns while developing the software system.

The starting point is the existence of a set of non-functional requirements, specifically performance ones. The goal of any development process that intends to satisfy such requirements is to start performance analysis as early as possible on the available software artifacts, possibly supported by suitable models. However, the use of
these models for performance analysis is analogous to the use of behavioral models for functional analysis. Namely they serve the purpose of the analysis at the concerned abstraction level with no intent to be considered predictive with respect to the performance of the final system.

Let us consider a conventional waterfall software development process. The first stage deals with requirements specification. At this stage the non-functional requirements are specified together with any operational constraints. During this stage, performance models can be built as any other model, mainly during the requirements engineering stage and in order to elicit and better understand the performance requirements. The same kind of reasoning applies to the architecture design and further down to the implementation and deployment, which can represent the last step where performance analysis reduces to simulate and/or monitor the actual behavior of the implemented system.

In this section we tackle a more detailed level of abstraction with respect to the description provided in the previous section. Therefore, we refine the concept of stages introduced before as phases of the lifecycle. The refinement logic is illustrated here below.

Taking inspiration from the familiar V-model for software validation, we customize this view toward performance analysis, thus obtaining what we will call the Q-model in the following. Figure 4.3 illustrates our view.

The left-hand side represents common development phases, that is: requirements elicitation and analysis, architectural design, detailed design and implementation. The right-hand side represents the performance analysis activities that can be carried on at each specific development phase.

With respect to the common stages described in the previous section, here we can consider the following mapping: (i) requirement specification stage has simply been rephrased as the requirement elicitation and analysis phase, (ii) software design and implementation stage has been partitioned in architectural design, low-level design and implementation phases, (iii) software verification and validation stage is represented by the middle and right-hand side of the figure, as will be illustrated here below.

In the middle, performance model generation activities connect each development phase with the corresponding performance analysis activity. Basically such intermediate activities derive from specific software artifacts the corresponding performance model. For example, the architectural design phase is connected to the performance analysis of software architecture through a performance model generation step that, starting from a software architecture specification, produces the corresponding performance model. Feedback arrows complement each horizontal connection and denote the feedback that the performance analysis can produce for the corresponding software development phase.

The connecting vertical arrows along the development path (i.e. the left-hand side) represent the successful completion of a phase and the transfer to the next development phase. In the Q-model a phase is complete only after appropriate performance analysis activities (i.e. the horizontal path for that phase). The connecting
Fig. 4.3 Q-model for a waterfall process
vertical arrows along the performance analysis path (i.e. right-hand side) represents the information that analysis activities transfer to the next phase activities. For example, the performance bounds obtained at the architectural phase may represent reference values for the analysis at the low-level design phase. Like in any V-model, upstream vertical arrows appear in both paths, and they represent backward paths that might be traversed in case of problems at lower phases that cannot be fixed without re-executing the previous phases.

The lowest part of the Q-model deals with the implementation of the system and with the monitoring of its actual behavior. In this case the horizontal line denotes the process of defining suitable observation functions on the running code that may allow for performance indices validation. The bottom vertex is the monitoring activity that receives information on what to monitor, on the final executing code, from the observation definition process that also depends on the performance indices to validate. The monitoring phase provides feedback to both the implementation and the
Fig. 4.6 Annotated component diagram
performance validation analysis activities. The feedback can then vertically travel along both lateral sides, thus inducing changes backwards on the software artifacts and on the performance models, respectively. On the horizontal paths, it is worthwhile remarking that the feedback process starts from performance analysis activities; it can have effect on the generated model, and it can induce changes that must be reflected at the corresponding development level. We will see in Chap. 7 that this feedback process is not straightforward and still represents a challenging research issue.
Fig. 4.9 Queueing Network model
4.3 A Simple Application Example

As an illustration of the Q-model let us consider the e-commerce example previously introduced in Chap. 2.

The use case diagram in Fig. 4.4 represents the artifact modeling the e-commerce system at the level of requirements specification. We have annotated the link connecting the Customer actor to the BrowseCatalog functionality to express a response time requirement, that is: a Customer should not wait more than 8 seconds to access the Catalog. From the analysis point of view we can interpret this limit either as an average or as an upper bound.\(^1\)

In this case since we are dealing with just one non-functional requirement we are neither producing a performance model from the requirement specification nor performing any performance analysis for consistency checking.

While proceeding in the software development process, Figs. 4.5 and 4.6 show, respectively, a flat and an annotated UML component diagram of the example, namely a static view of the e-commerce example software architecture. The flat and annotated UML sequence diagrams of Figs. 4.7 and 4.8, respectively, represent the dynamic view of the same architecture.

\(^1\)The capability of annotating UML diagrams with additional information (such as performance parameters and indices) is provided from the UML profiling technique that has been described in Chap. 2.
Annotations at the architectural phase can represent different performance-related data. For example, in Fig. 4.6 the resource demand of the service to read the status of a catalog is annotated on the corresponding component interface, and yet the same component CatalogServer is annotated with its policy of scheduling for pending requests. Similarly, in Fig. 4.8 the workload originated from triggering the service of browsing the catalog is annotated on the first message of the scenario represented by the UML sequence diagram.

Following the path on the right-hand side of Fig. 4.3, we notice that our initial performance requirement on the BrowseCatalog functionality in Fig. 4.4 is reflected in the annotation of the BrowseCatalog interface delay in Fig. 4.6. The latter annotation also refines the original requirement, in that the time limit is interpreted in the annotated component diagram as an average value.

If we focus on the architectural phase of the development process of Fig. 4.3, and we run the horizontal path leading from the Architectural Design to a Software Architecture Performance Model, we can generate a Queueing Network (QN) model from the previously introduced artifacts (i.e. the set of annotated UML models of the e-commerce example). The QN structure is shown in Fig. 4.9.

In order to perform analysis on this model, its parameterizations must be completed. As will be discussed in Chap. 6, depending on the available information the analysis can be totally or partially symbolic and it is usually oriented, at this development phase, at comparing alternative architectural designs.
Fig. 4.12  Layered Queueing Network model
4.3 A Simple Application Example

Following the development process (i.e. the left-hand side of Fig. 4.3), the detailed design of the e-commerce application is produced in terms of algorithms and data structures of each software component. This step may terminate with the construction of flat and annotated UML deployment diagrams of the software system, as shown, respectively, in Figs. 4.10 and 4.11. Annotations here may represent the low-level scheduling policy of a deployment host (e.g. the one of the operating system running on the specific host).

From these additional artifacts a further (more detailed) performance model can be generated, possibly using a different modeling notation, such as the Layered Queueing Network shown in Fig. 4.12. This model reflects the architectural decomposition of the system and its deployment structure. Hence it contains more information than the Queueing Network shown above, because it has been generated in a later development phase. The evaluation of this model produces performance indices that should be compared with the initial performance requirement.

As illustrated above, our focus in this book is on the process of producing performance models, by means of model transformations, from the software artifacts produced during the development process. In Chap. 5 we will describe in detail several approaches to build these model transformations.

In this chapter we have described the performance analysis in the context of a waterfall software development process. In order to address other software process models we need to provide an idea on how to generalize the Q-model previously described.

Let us recall that at each phase the software artifacts represent the system at different levels of abstraction, whereas the ultimate target of performance analysis is always to satisfy the initially formulated performance requirements. Therefore the performance models generated at each phase aim at the same kind of quantitative analysis, no matter what the name is assigned to the development phase.

Hence, the quantitative analysis can be always based on model transformation that, opportunely defined, generated the performance model at the same level of abstraction of the source development artifact. This amounts to saying that we need to attach at each software artifact the missing information (i.e. the model annotations illustrated above) and a corresponding model transformation that enables the performance model generation. Then the triple \(<\text{software artifact}, \text{missing information, model transformation}>\) can be freely embedded in different software processes. Thus, thanks to model transformation techniques, today we can concentrate on how to produce, for each significant software artifact of our development process, a model that allows the quantitative analysis to be made typical of the development activity the software artifact refers to.
Model-Based Software Performance Analysis
Cortellessa, V.; Di Marco, A.; Inverardi, P.
2011, XII, 190 p., Hardcover
ISBN: 978-3-642-13620-7