Lappeenranta University of Technology

School of Business and Management

Master's Degree Programme in Computer Science

Tuomo Timonen

APPLYING SOFTWARE PERFORMANCE ENGINEERING METHODS TO DEVELOPMENT OF IT DEVICE MANAGEMENT SYSTEMS

Examiners: Professor Jari Porras

M.Sc Sami Mäkiniemelä

Supervisors: Professor Jari Porras

M.Sc Sami Mäkiniemelä

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto

School of Business and Management

Tietotekniikan koulutusohjelma

Tuomo Timonen

Applying software performance engineering methods to development of IT device management systems

Diplomityö

2016

84 sivua, 24 kuvaa, 4 taulukkoa

Työn tarkastajat:	Professori Jari Porras
	DI Sami Mäkiniemelä

Avainsanat: Software performance engineering, suorituskyky, laitehallinta, web-sovellus

Ohjelmiston suorituskyky on kokonaisvaltainen asia, johon kaikki ohjelmiston elinkaaren vaiheet vaikuttavat. Suorituskykyongelmat johtavat usein projektien viivästymisiin, kustannusten ylittymisiin sekä joissain tapauksissa projektin täydelliseen epäonnistumiseen. Software performance engineering (SPE) on ohjelmistolähtöinen lähestysmistapa, joka tarjoaa tekniikoita suorituskykyisen ohjelmiston kehittämiseen. Tämä diplomityö tutkii näitä tekniikoita ja valitsee niiden joukosta ne, jotka soveltuvat suorituskykyongelmien ratkaisemiseen kahden IT-laitehallintatuotteen kehityksessä. Työn lopputuloksena on päivitetty versio nykyisestä tuotekehitysprosessista, mikä huomioi sovellusten suorituskykyyn liittyvät haasteet tuotteiden elinkaaren eri vaiheissa.

ABSTRACT

Lappeenranta University of Technology

School of Business and Management

Master's Degree Programme in Computer Science

Tuomo Timonen

Applying software performance engineering methods to development of IT device management systems

Master's Thesis

2016

84 pages, 24 figures, 4 tables

Examiners: Professor Jari Porras
M.Sc. Sami Mäkiniemelä

Keywords: Software performance engineering, software performance, device management, web application

Software performance is a pervasive quality of software that is affected by everything from design and implement to environment in which the software is run. Performance issues are a serious problem in many projects leading to delays, cost overruns and even complete failures. Software performance engineering (SPE) is a software oriented engineering approach that provides methods to develop software that meets its performance goals. This master's thesis researches SPE methods and selects the ones suitable for solving performance issues during development of two IT device management systems. The outcome is an updated version of the development process currently in use that takes software performance challenges into account during different stages of software lifecycle.

ACKNOWLEDGEMENTS

I am using this opportunity to express my gratitude to Jari Porras, Sami Mäkiniemelä and Ville Kinnunen for their guidance in the process of writing this thesis. Also, I would like to thank Julia Virkkala for invaluable help and mental support during the work. I would not have been able to to do this without your help!

I thank my family, friends, colleagues and everyone who supported me through writing this thesis and my life in general!

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
CMD	Command Prompt
CMS	Configuration Management System
CPU	Central Processing Unit
CSV	Comma Separated Values File
DOM	Document Object Model
EG	Execution Graph
ETW	Event Tracing for Windows
GUI	Graphical User Interface
HTTP(S)	The Hypertext Transfer Protocol (Secure)
IIS	Internet Information Services
IT	Information Technology
LQN	Layered Queuing Network
MDM	Mobile Device Management
MSP	Managed Service Provider
NFR	Nonfunctional Requirement
OMG	Object Management Group
OS	Operating System
PA	Performance Assertions
PeRF	Performance Requirements Framework
PMF	Performance Measurement Framework

QN	Queuing Network
RoI	Return on Investment
RPC	Remote Procedure Call
SDM	Semantic Data Model
SIG	Softgoal Interdependency Graph
SMS	Short Message Service
SPA	Stochastic Process Algebra
SPE	Software Performance Engineering
SPEM	Software Process Engineering Metamodel
SPN	Stochastic Petri Nets
SQL	Structured Query Language
UI	User Interface
UML	Unified Modeling Language
UML-SPT	UML Profile for Schedulability, Performance and Time
VB	Visual Basic
XML	Extensible Markup Language

1 INTRODUCTION

Ensuring that software systems meet their performance expectations can be a difficult task, especially when the number of clients, complexity of the software and diversity of software deployment environment grows. A small on-premises software installation that serves few hundreds of clients has different resource requirements compared to a cloud-based installation that provides services for tens of thousands of clients worldwide. Regardless, the software should function similarly in both situations and offer satisfying user experience.

A survey (Compuware, 2006) indicates that 80% of European IT (Information Technology) executives know that customer satisfaction can be affected by performance issues and other related effects. However, over 70% of them said that problems were actually reported by customers rather than in-house monitoring systems.

Performance is a pervasive attribute of software systems. It is affected, for example, by the design, implementation, runtime environment and workload. Increased workload generates variable load on different parts of the software system. An operation that is extremely fast on lower number of users may slow down significantly along with the number of concurrent users. A new feature or modification to an existing feature may have a severe impact on performance, particularly if the change is made to a critical part of the code.

Having proper tools and practices, including understanding how to utilize those is vital in the above-mentioned case. Software performance engineering (SPE) is systematic software oriented engineering approach to assist development of applications that meet performance requirements. It provides means for tracking performance during the development process and help to prevent unexpected performance problems late in the lifecycle. (Smith, 2001)

1.1 Background

This Master's thesis is done for a company developing solutions for IT and mobile device lifecycle management system with integrated asset management, configuration management, and lifecycle management features. The company has two main products: a cloud-based mobile device management (MDM) system and a configuration management system (CMS).

The MDM solution is hosted at company's own cloud servers. It provides mobile device management functionalities, such as inventory data collection, configuration deployment, application installation and real-time location tracking. The CMS offers information technology (IT) asset's lifecycle management from purchase and initial installation, through use and maintenance to retirement of devices including features such as software asset management, license management, incident management and security management. Using the products, system administrators can see the status of their managed IT environment, generate variety of reports and run maintenance operations on managed devices from the web-based management console.

The CMS is mainly targeted for managed service providers (MSPs) that host the product in their own premises and sell configuration management as a service to their own customers. Configurations of the software runtime environment tend to change between different customers and the number of managed devices per instance may differ from hundreds to tens of thousands of devices.

Lately, the company has been growing steadily. New features have been implemented and the number of users and managed devices has been increased constantly. This has resulted in situations where applications performance has received increasing attention.

1.2 Goals and restrictions

The goal of this Master's thesis is to improve practices on detecting and resolving performance issues before the product is released to production, thus, improving quality and reducing the development effort and costs. The focus is on the server-side components.

Prior to this work, the company has no unified means for analyzing product performance. There are some manual methods for inspecting performance of specific parts of the system, but they require temporal changes to codebase and strong knowledge of the architecture. For example, developers may add temporary instrumentation points to measure how long it takes to execute a piece of code. These methods are neither well documented nor consistent across developers

To begin with, the initial goal was to design and implement a performance measurement framework that would visualize current performance using a traffic light. The green light means that performance is within acceptable limits and the red light indicates presence of performance problems. Additionally, the framework should be able to point of the origin on the problem. The initial plan was to implement solely measurement-based tools to monitor performance of existing features.

Quickly, it became obvious that this was not sufficient to solve performance issues. Initial discussions with other developers in addition to literature reviews pointed out that performance is more of a fundamental issue. It is relatively easy to measure execution times, especially when having access to the source code. However, fixing performance issues late in the development is complicated and expensive. Previous experiences pointed out that it may take longer to fix an issue than it took to implement the feature at first place.

Based on the above-mentioned concerns, the following main research question was constituted:

- How to integrate software performance analysis into company's software development process?

Additionally, the following sub-questions were derived to support the main question:

- What software aspects should be measured?
- When and how to make measures?
- How to get feedback from performance analysis activities back to the software development process as early as possible?

The research done in this thesis follows the practical approach. The goal is to study the literature and find out solutions that solve similar problems. Solutions are mirrored against company standards, current model of operation and well-proven best practices. The most applicable solutions should be adopted in everyday product development work.

1.3 Structure of the thesis

The remainder of this thesis is structured as follows:

- Chapter 2 presents the architecture of the CMS and MDM system and introduces the most critical features from performance point of view. It also highlights the best-known performance issues of the systems.
- Chapter 3 presents the development process used to develop the products. It discusses how the process has evolved over the years and presents issues with the current model of operation.
- Chapter 4 casts a glance on the literature. It introduces the Software Performance Engineering a software development approach to develop software that meet its performance objectives. This chapter goes into details theory behind software performance analysis, various SPE process models and techniques to examine software system's performance.
- Chapter 5 takes the process, presented in chapter 3, and introduces practices from SPE to different stages of the process.
- Chapter 6 wraps things up with conclusions.

2 SYSTEMS OVERVIEW

The configuration management system and the cloud-based mobile device management system are device management systems. The systems consist of the server-side application and the client applications installed on managed devices. The client communicates with the server over HTTPS (Hypertext Transfer Protocol Secure) and runs different tasks. The server provides a web-based user interface for system administrators to see status of their environment and generate various reports. The CMS, which is installed on customer's own premises, offers comprehensive IT asset's lifecycle management capabilities from purchase to retirement of devices. The MDM system is a simplified version of the CMS providing mobile device management features for Android, iOS and Windows phone devices from the cloud.

This chapter presents these systems in detail. The introduction starts from the big picture and goes well into details by describing how different components take a place in the domain and affect each other. Additionally, identified performance challenges are described to set some baseline requirements for the work.

2.1 Architectural overview

The architecture of the CMS and MDM systems is presented in figure 1. The products run on a Microsoft Windows platform. The key parts are the front-end server(s) and the Microsoft SQL Server (Structured Query Language) database. The web server hosts ASP.NET C# applications that run on Microsoft Internet Information Services (IIS). These applications serve as entry points to the system by providing:

- a web-based graphical user interface (GUI) for users and administrators
- communication interfaces for client applications running on managed devices
- application programming interfaces (APIs) for other entities that communicate with the system.

A front-end server runs various background daemons that process the data in the systems (e.g. generate scheduled reports and perform maintenance jobs). Daemons can be, for example, individual executables (C#), Visual Basic (VB) or Windows command line (CMD) scripts, or SQL Server stored procedures.

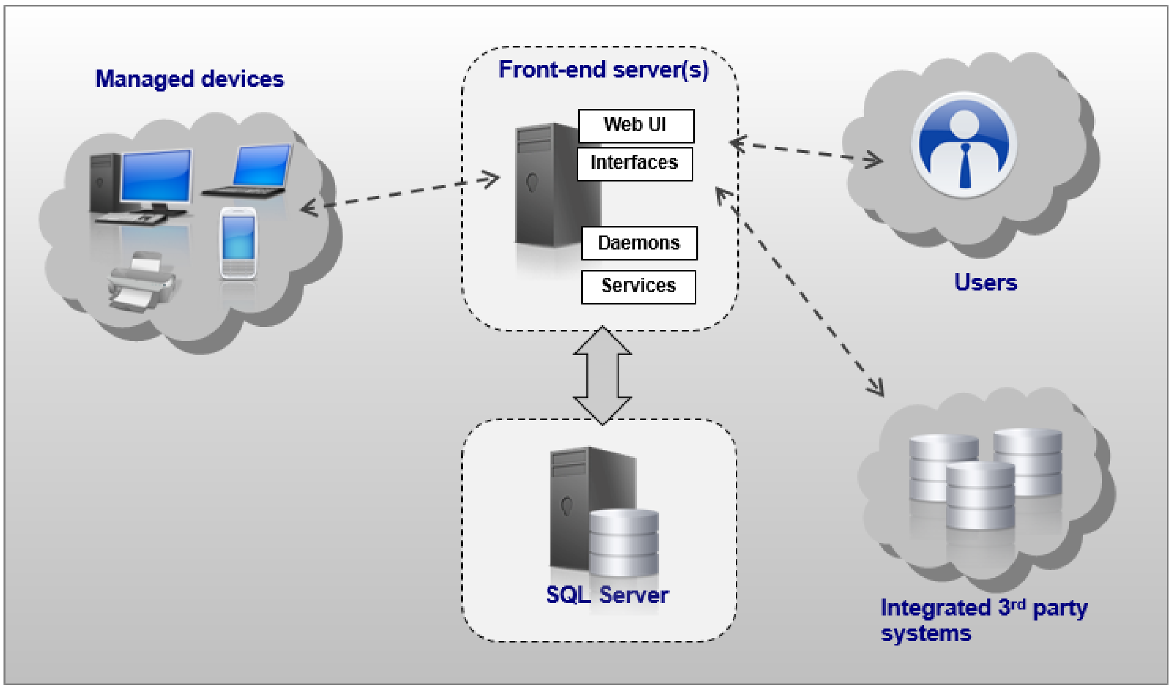


Figure 1: Architecture of the device management systems

Additionally, the server runs Windows services implemented in C#. These services run in the background and provide asynchronous queues for different actions, for example, sending emails, wake-up requests and Short Message Service (SMS) messages. Finally, yet importantly, the Microsoft SQL Server database has a large role from performance standpoint. It serves as both long and short-term data storage for all other components. Features such as, client-server communication, user interface, integrations and background tasks generate constant load the database server.

2.2 Performance critical features

This chapter introduces the features that have been identified to have the largest impact on system's performance.

2.2.1 Client-server communication

Each managed device has a client application installed that starts automatically when the device is powered on and runs silently in the background. The client polls the server periodically to update its status, queries for pending jobs and sends inventory data. Implementation of the client varies between platforms. For desktop operating systems (Windows, Linux and OSX), the client is a C++ application. For Android, the client is a Java application. Apple iOS and Windows Phone use MDM capabilities built in to those platforms, and therefore, the in-house client application is not needed.

Common for all platforms is that the underlying communication protocol consists of XML (Extensible Markup Language) fragments transmitted over HTTPS. The protocol specific APS.NET web handles processes incoming requests by parsing contents into an in-memory DOM tree (Document Object Model) authenticates and identifies the device, and update device data to the SQL Server database.

Client's polling interval can be configured. By default, the desktop client connects to the server once per hour and updates its configurations once every 12 hours. The next connection time is calculated based on the previous, which leads roughly even distribution over the time. However, there are few exceptions, which are presented in the next chapter.

As summarization of the client-server communication, table 1 presents the average number of incoming messages the server must process over an hour. Example calculations are done for a desktop client. It represents a best-case scenario that contains only the basic messages generated by the client with default polling intervals.

Table 1: Message volumes generated by the client and scheduler

Clients	Client messages	Config updates	Avg. msg / min	Avg. msg/ sec
1000	3000	83	51	1
10 000	30000	833	514	9
30 000	90000	2500	1542	26
100 000	300000	8333	5139	86

Server's capability to process consecutive requests becomes critical when the number of clients increases. With 1000 clients, the server must handle an average of one message per second, but with 30 000 clients there will be 26 messages per second. That is 26 XML fragments and even more SQL queries every second - continuously. A few seconds of slowdown in server-side processing can cause severe congestion.

2.2.2 Device inventory data processing

Inventory data collection is one of the most important features of a device management system. Inventory data consists of, for example, hardware details, information about installed applications and software usage reporting. There are two main sources from inventory data: managed devices and integrated third party systems. The client collects inventory data from managed devices, and connectors gather data from third party sources. Collected inventory data is sent to the server's ASP.NET inventory handlers. Inventory data flow consists three phases:

1. Read incoming data (HTTPS/XML).
2. Decompress if needed and add to the queue.
3. Import data from queue to database.

Received inventory data is queued for further processing. Some inventory data is sent as compressed archives and must be decompressed by the handler before it can be processed. Inventory import daemon/service processes inventory queues inserts data to the database. Inventory data imports are major performance concern in larger environments, because incoming files can be large and it may take a while to update the database.

2.2.3 Scheduled background tasks

As aforementioned, scheduled tasks, also known as daemons, process a lot of data in the background. For example, file scan inventory data consists of a full list of executables found from a target computer. The list may contain tens or even hundreds of thousands of entries, each containing file names, file sizes and other useful metadata. Inventory import process inserts raw data to the database as is. In order to be useful, this data must be further processed. A customer may want to know whether a specific application (e.g. Notepad++ v.1.0.8.0) or software bundle (Microsoft Office 2007 Enterprise) is installed in the environment. Such software identification may take a while and is rather resource demanding operation, and therefore, a background task is needed that will process the job during off-peak hours when system use is lower. Device management systems consist of lots of similar background daemons, which have undisputed effect on application performance.

2.2.4 User interface

Web user interface (UI) is an ASP.NET web application. It is the primary interface for system administrators and operators, from which they can see the status of their entire IT infrastructure, generate various reports, distribute application and configurations, and perform other maintenance operations. UI contains lots of dynamic content that is retrieved from the database when a user opens a page or report.

2.2.5 Integrations and connectors

Integrations with third-party systems and services have increased over the years. Data is transferred from device management systems to other systems (e.g. service management systems and financial systems) and vice versa. Data is used to complement available reports.

There are two ways to do the integrations:

1. Connectors
2. REST-based API

Connectors are in-house applications that gather desired data from third party sources and send it to the device management system. Connector data is sent to server's ASP.NET handlers from which it goes to the import queue. REST-based API (ASP.NET) provides two-way interface for custom third party system integrations.

2.2.6 Summary of performance critical features

Previous chapters described client-server communication, inventory data imports, background tasks, user interface and integrations as the largest factors that affect performance of the systems. Although they all represent different scenarios there are similarities, which specify requirements for the performance analysis. It should be possible to:

- **Track HTTP requests**
    a. Number of incoming requests
    b. Type of the request (e.g. client message type or name of the web page)
    c. Processing time on server-side
    d. Timestamp
- **Track SQL queries**
    a. Identifier (e.g. entire SQL query or name of the SQL Server stored procedure)
    b. Execution time
    c. Timestamp
- **Queue statuses**
    a. Number of items in queue
    b. Throughput

In addition, it is important to link executed SQL query to associated HTTP request. This provides additional information about the system because database issues may have system-wide effect on application performance.

2.3 Known performance issues

The biggest known performance issues fall under the following categories:

1. Incorrect or inefficient implementation
2. Request congestion and load accumulation
3. Locks and synchronization issues
4. Diversity of runtime environments

Every now and then, there are bugs in the code that cause performance issues. On the other hand, the code might work as expected but does it too slow. This behavior might be inconspicuous in a small environment but causes problems in a larger environment.

Request congestion and load accumulations are other critical issues. For example, Monday morning tends to be problematic in large environments in which all users are roughly on the same time zone. Computers have been turned off during the weekend. At the Monday morning, many devices are powered on within short period. Because the client has not updated its status for days, it will immediately poll the server and send inventory data. This results in as a significant peak load on the server.

Database locks and other synchronization methods are related to the first category, but deserve emphasis due to recently detected issues. A table in the database can be modified simultaneously by more than one component. For example, exclusive database locks are used to synchronize such operations. Unexpected issues may emerge if an SQL query that holds the lock has problems to complete.

The final category is related solely to the CMS. A customer of the CMS can be a managed service provider having thousands of devices from hundreds of their customers or a smaller company having only a hundred of devices. The CMS has to scale to meet the needs of both. Because of this, it has to support different infrastructure configurations. Additionally, it supports multi-instance environments, which means that MSPs can run separate application instances in the same physical server on behalf of different customers.

3 DEVELOPMENT PROCESS

Software development process a collection of different activities that lead to a software product. Software development processes vary between software development companies and software products. Different companies have their own processes that are suitable for them. Type of the software product has an influence on the process. Products having a long lifecycle require different activities compared to ones with short lifespan (Cortellessa, et al., 2011).

This chapter introduces the development process used in the company. Firstly, a general example of a software development process is described. Then, the glance is cast on how software development process used in the company has evolved over the time. Finally, the current state of the process is described.

3.1 Overview of general software development process

Software development process can be expressed with a software development process model. Different software development processes contain different set of activities. However, there are fundamental development stages that are common for every software development process (Cortellessa, et al., 2011):

1. **Requirement specification**

   During this stage, customers and developers define software products functional and operational constraints by specifying all the requirements of the system.

2. **Software design and implementation**

   During design and implementation stage, the software product is produced according to its specifications. Software models (e.g. architecture and design models) are created and the software is implemented based on those models.

2. **Software verification and validation**

   After software is implemented it moves to verification and validation phase. This stage ensures that the software meets its original requirements. Verification and validation is usually accomplished by demonstrating the software to the customers.

3. **Software evolution**

   Software moves into evolution stage after the first version has been deployed. This stage manages the changes to the software.

Figure 2 illustrates an iterative software development process that follows these stages. An iterative process runs concurrently to quickly develop an initial version of the software. Later the initial version is refined through several iterations, each producing a new version of the software. Recently, software evolution has become more important due to progresses in the software development processes. (Cortellessa, et al., 2011).

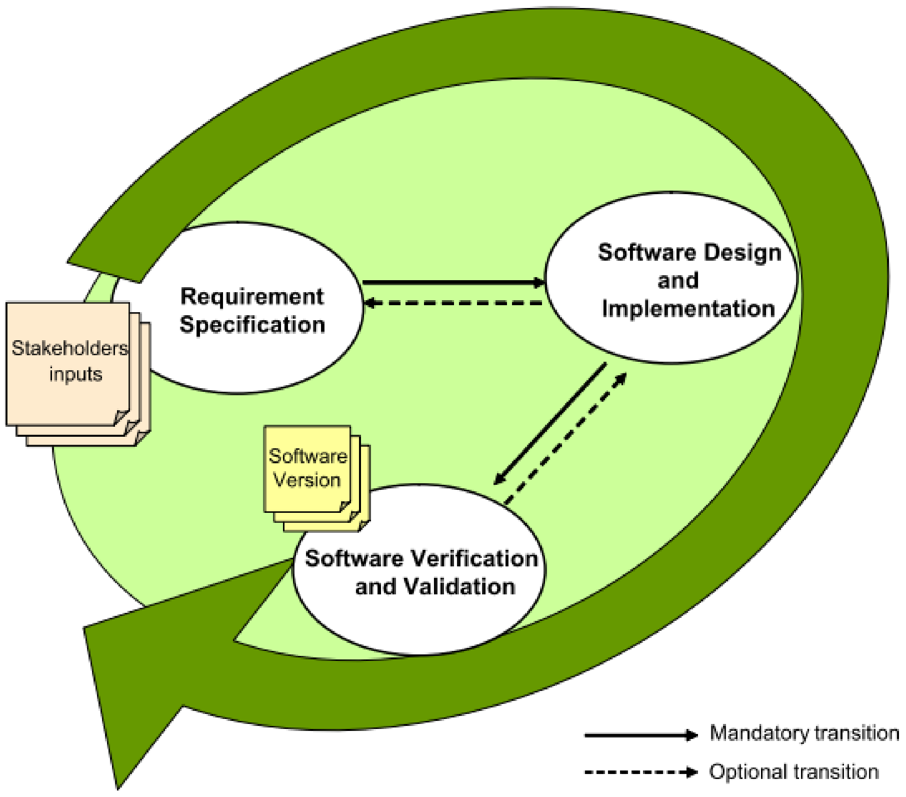

Requirement Specification
Software Design and Implementation
Stakeholders inputs
Software Version
Software Verification and Validation
Mandatory transition
Optional transition

Figure 2: An iterative software development process (Cortellessa, et al., 2011)

The development process described in the following chapters an iterative process. Developed products have a relatively long lifespan with periodical consecutive releases. For example, the CMS has been developed over 10 years.

3.2 Agile software development

The Software development process the company used to develop its product has evolved over the time. The process used to be rigid which caused problems. Development cycles for new features used to be lengthy. It took excessively long until stakeholders and customers were able to review the changes. If a feature was inadequate, it had to be fixed. Fixing severe deficiencies, which require architectural changes, took remarkably long time.

Projects used to have strict deadlines set in advance before the development had begun. Additionally, resources and required feature set was defined beforehand. This operational model was suitable for small improvements, but lead frequently to issues with larger projects. Firstly, initial planning of release dates with complete feature sets was found to be an infeasible task. Secondly, customer needs are ambiguous and tend to change over the time. Thirdly, business position and technology keeps shifting during a long running project. Today mobile devices may be a hot topic, but tomorrow it might be something else. Therefore, it was identified that instead of waiting for over six months until complete feature set is out, some parts of it should be implemented, reviewed and possibly released at more rapid pace.

It is recognized that software development is an empirical, nonlinear process, because of the change occurring during the time of developing a product. Such empirical process requires frequent, short feedback loops, which can better react on rapid changes. Agile software development is about feedback and change. It attempts to overcome above mentioned challenges. (Laurie & Alistair, 2003). The original "Manifesto for Agile Software Development" (Beck, et al., 2001) states that valuable outcome requires daily co-operation between business people and developers throughout the project. This way the team can deliver working software frequently and hence keep customers satisfied. This reasoning was identified within the company, and therefore, the process has been shifting towards agile software development methods.

3.3 Current model of operation

The software development process the company used to develop its products is presented in figure 3. The product backlog is the tool used by the product owner to keep track of the features that customers, stakeholders or other contributors want. The backlog contains a short description of each new feature, minor improvement and bug fix. These backlog items are also known as user stories. User stories are prioritized so that the most important ones are on the top. Prioritization is done by product owner along with product management personnel. User stories that are near the top are also more refined as they will be earlier in the making. It is not reasonable to specify features far into the future as it is not certain when those features will be on the table. User stories are further refined in a weekly backlog grooming meeting in which product owner, product management and developers are present. For example, relative workload estimate is defined for user stories during the meeting.

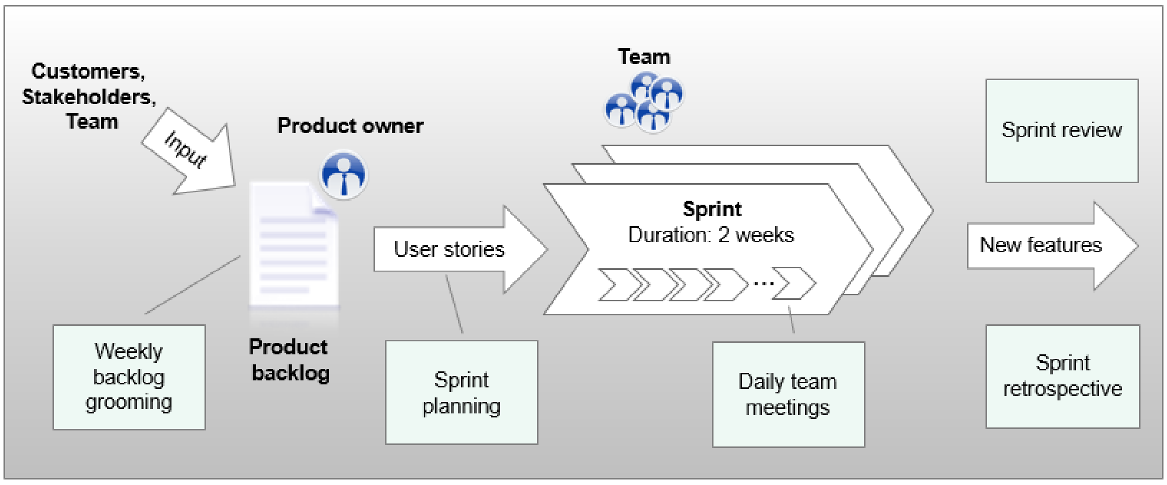


Figure 3: Development process overview

Development is done in two-week sprints. During a sprint developers work independently and iteratively to implement user stories. Each sprint is preceded by a sprint-planning meeting where the team alongside with product owner selects the highest priority user stories to sprint backlog. The spring is followed by a review meeting, where the completed user stories are reviewed to rest of the company personnel, and sometimes for customers too. After the review, the team meets in sprint retrospective session in which the sprint is reviewed in order to identify lessons learned. This is used to improve upcoming sprints. Sprint retrospective is followed by the planning of the next sprint. Usually, a new release comprises of multiple sprints. In such case, the last sprint before the release is dedicated to release and integration testing.

User stories completed during a sprint follow the workflow presented in figure 4. Stories in the backlog, either product or sprint, remain in *TO DO* state. The end state for stories is either

DONE (implementation, review and testing ready) or *REJECTED*. Generally, user stories in the backlog do not contain detailed requirement specifications. There might be some high-level functional requirements from the customers and stakeholders, many of which are related to common usability. Non-functional requirements and precise functional requirements are seldom available. The development team draws the requirement specifications based on available information and their expertise when they pull out a user story from the backlog and start working on it. This works because all developers have long history with the company. The situation would be different with a junior team.

The user story is divided into tasks, each representing an independent piece of work to be done (e.g. “Add new column X to table Y” or “Add a button to the GUI”). This is rather straightforward process, but communication between development team and product owner is particularly important at this stage. Communication is the way the team requests feedback on the work in progress. Moreover, the work in progress can be demonstrated to customers to achieve valuable feedback. This is done during the *IN PROGRESS* state.

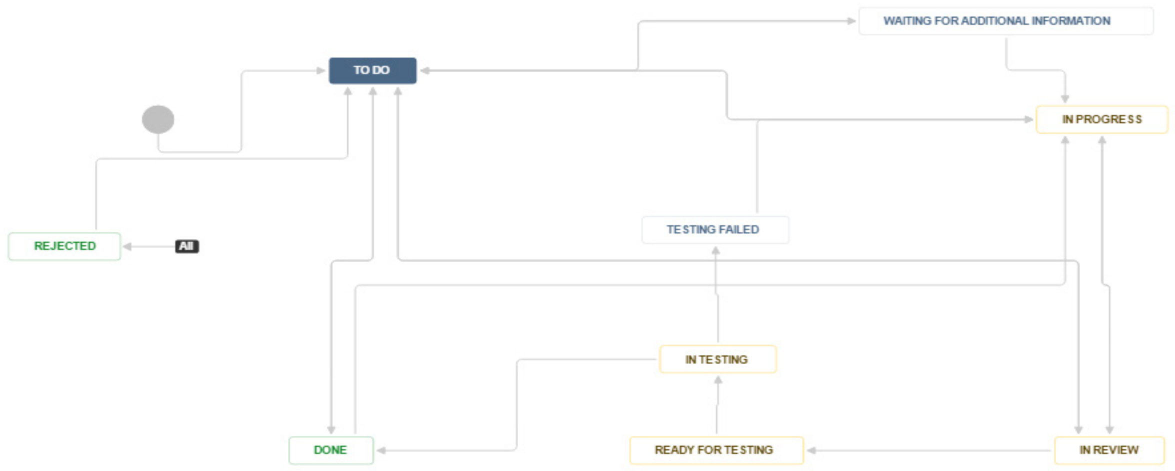


Figure 4: The lifecycle of a user story

When a developer thinks the user story is ready, it moves to *IN REVIEW* state. Peer review is performed by another member of the development team. Peer review consists of the code evaluation, also known as code review, and quick functional overview that aims to:

- catch obvious bugs as early as possible
- share knowledge among developers

- evaluate implementation choices.

Next, the user story moves to verification phase. Similarly compared to peer review, testing is performed by some personnel who is a member of the development team. The main goal of this phase is to verify whether the implemented feature meets its original requirements. Additionally, the tester takes an overview of the feature and evaluates its usability in general.

3.4 Challenges with the current model of operation

Perhaps the biggest problem with the current model of operation is that performance goals and other nonfunctional attributes of software are not considered enough during the design and development phases. This results in the fact that the performance issues are detected late in the development. Usually, when the work is almost ready or when the feature already released to the customers.

Naturally, this means that comprehensive measurements of application performance are neither implemented nor performed. In addition to that, automation is not involved in performance testing. Doing performance analysis in the current process has to be done during a separate project that incorporates lots of overhead.

4 SOFTWARE PERFORMANCE ENGINEERING

Traditionally, software development focuses on functional correctness meaning that non-functional requirements such as software performance are considered later in the development process. This style of development is known as the "fix-it-later" approach. Software system complexity has increased over the years while the relative number of performance experts has decreased. This combined with the commonness of the "fix-it-later" methodology lead to serious problems with many software products. Critical performance issues usually evolve from early design choices, many of which cannot be corrected with hardware that is more powerful. The software must be designed from start to meet its performance objectives. (Smith, 2001) (Cortellessa, et al., 2011)

Software performance engineering is systematic software oriented engineering approach to assist development of applications that meet performance objectives. Providing a collection of methods and tools, it spans throughout the software development lifecycle (Woodside, et al., 2007) (Smith, 2001). This chapter describes what SPE is and how it can be taken into consideration during software development.

4.1 Definition of software performance

Literature contains several definitions for the software performance. Smith and Williams describe it as a make-or-break quality for software that can be observed as software systems responsiveness and scalability (Smith & Williams, 2003). In other words, it can be seen from user's point of view as the response time and throughput of the software system (Smith, 2001).

Woodside et al. consider software performance as a pervasive quality that is affected by everything from software design and implementation to environment in which the software is run; hence, making it difficult to understand. Performance issues alone are causes of serious problems in many projects, leading to delays, cost overruns and even complete failures. Although performance issues can be critical, they are seldom documented (Woodside, et al., 2007).

Similarly, Balsamo et al. describe software performance as a runtime attribute of software systems. Moreover, they add "software performance is the process of predicting (at early phases of the lifecycle) and evaluating (at the end) whether the software system satisfies the user performance goals. From the software point of view, the process is based on the availability of software artifacts that describe suitable abstraction of the final software system." These artifacts are, for example, software requirements, and architecture and design documents. (Balsamo, et al., 2002)

Fortier and Michel define that software performance is a part of software systems quality of service, considering some performance attributes (e.g. response time, ease of use, reliability and fault tolerance) as qualitative measures, which are hard to measure with quantitative manner. Quantitative measures are easier to understand, because they can be presented with quantifiable variables (e.g. numbers). Performance quantities, (e.g. usability) are qualitative versus quantitative measures, meaning that they can be observed, but not measured precisely. To be scientifically precise software performance measurement should focus on quantitative qualities. (Fortier & Michel, 2003)

As can be seen from the above-mentioned definitions, performance is ambiguous attribute of software. What is the performance like? Tough the question looks simple it is hard to answer as performance can be viewed from different perspective, especially when considering complex software systems. Performance is not a single measure, or value, but combination of many.

4.1.1 Performance indices

Performance indices are defined to conduct performance measures in quantitative manner. Traditional performance indices are (Balsamo, et al., 2004) (Fortier & Michel, 2003):

- **Response time:** Time required for a request to travel from a specific source to a specific destination and back within the software system.
- **Throughput:** Rate of tasks the system, or part of it, is capable of execute per unit of time**.** For example, number of SQL statements per second.
- **Utilization:** The time that the system, or part of it, is busy. For example, how much is the CPU (Central Processing Unit) utilization.

Performance indices can be divided further into two categories: user- and system-oriented measures. Response time is a user-oriented measure, which means it can be observed directly by the user. User-oriented measures are not accurate over the time, because of the number of variables involved, and their dependence on system resources. For example, response time for a web application may be affected by the network characteristics (e.g. throughput, bandwidth and latency) and server utilization. Therefore, user-oriented measures are typically measured as averages over the time. System-oriented resources, on the other hand, determine system capabilities and are more accurate (Fortier & Michel, 2003). In addition to traditional indices, new software systems and platforms, such as mobile devices, have brought need for new indices. For example, power consumption is a major performance factor for handheld devices (Balsamo, et al., 2004).

4.1.2 Time

Time is the most fundamental unit in performance measurement (Fortier & Michel, 2003). It shows up in many different contexts. Server-side processing time, response time, and intervals are all different measures of time and present the same physical quantity in different orders of magnitude. Processing time is usually measured in milliseconds, and response time and interval for example in seconds. This is an important characteristic of time and must be considered when measuring it.

4.1.3 Events

Although time is an important quantity, it is only useful when used to measure something within the software system. Therefore, time is usually tied to events. Events are entities in the system, which are interesting from performance point of view (Fortier & Michel, 2003).

Figure 5 illustrates example set of events from a web application. Events are bound to time. The time describes when an event occurs, what is the duration of it, and also what is the interval between subsequent events. The events have relationship and hierarchy between each other. Initially, *page_load* event consists of four individual events: *init*, *query_db*, *build_obj*, and *resp.* Understanding *Page_load* event requires knowledge on these individual pieces. Moreover, subsequent events depend on their predecessors. *build_obj* cannot start before data is retrieved from the database. If the database transaction halts, the subsequent actions cannot proceed.

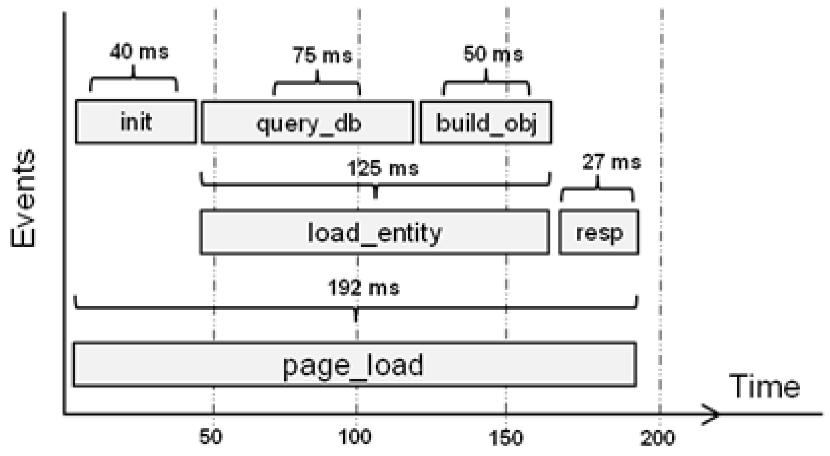


Figure 5: Example events from a web application

Knowing all events of interest and their relations is vital to make performance analysis as effective as possible. Event data can be used to determine (Fortier & Michel, 2003): How to make measures, when to measures and what to measure.

4.1.4 Sampling and instrumentation

Sampling in software performance analysis is the process of measuring system events of interest. There are different means to do sampling, each suitable for different situations. Method selection should be based, for example, on monitoring overhead and accessibility. (Fortier & Michel, 2003)

Hardware monitoring requires the ability add instrumentation to the system under study. Hardware instrumentation can be done by extracting and analyzing signals from the system, which means it is only possible to measure items or actions within the system that are accessible for monitoring. Signals can be extracted, for example, by adding custom designed hardware to the system. It is important that the monitoring itself does not interfere with the system. (Fortier & Michel, 2003)

Software monitoring, on the other hand, requires that the system under study provides means for accessing systems hardware and software resources, for example, system clocks and different timers. Modern operating systems provide this information. Software monitoring is typically used for trace monitoring and sampling, in which the code contains additional elements that allows code's execution to be monitored at run time. Software monitoring collects typically (Fortier & Michel, 2003):

- How often a segment of code (e.g. function or interface call) is run?
- How long it took to run the segment of code?
- How much of the total system time the code segment took?

Hybrid monitoring is a combination of hardware and software monitoring. It utilized both measurement techniques to gather extensive instrumentation data of the system. To get that data the hybrid monitoring must have access to systems hardware and software resources. Additionally, it may require synchronization of different hardware and software domains. (Fortier & Michel, 2003)

4.2 SPE process models

The SPE umbrella consists of two general approaches: measurement-based approach and model-based approach. These are usually associated with different stages of the software lifecycle. The measurement-based approach, also known as late-cycle measurement-based approach, focuses on running and measuring the software to investigate possible performance issues (Woodside, et al., 2007). In turn, model-based approaches use common software modeling techniques to predict the impact of early architectural and design decisions (Smith, 2001).

Although the above-mentioned approaches are considered different, they are not exclusive. In fact, it is recommended to use these techniques in conjunction. Performance measurements provide valuable information on system's overall performance, but can be used also to validate performance models. Moreover, model-based approaches can be used also throughout the software lifecycle from early-cycle prediction to evaluation at the end. This chapter presents different SPE process models to integrate SPE methods into software development (Smith, 2001) (Balsamo, et al., 2004) (Woodside, et al., 2007) (Smith & Williams, 2003)

4.2.1 The eight step performance modeling process

The eight step performance modeling process, presented in figure 6, starts from identification of critical and significant application scenarios. Critical scenarios are those associated with performance expectations or performance requirements. Significant scenarios, on the other hand, do not involve performance requirements, but may have an impact on critical scenarios, especially when they occur simultaneously. Identification can be done, for example, by analyzing use cases, user stories and service-level agreements. The second step is to identify workload for each individual scenario. Workload is usually derived from marketing data. It may consist of the desired number of total and concurrent users, and data volumes and transaction volumes. (Microsoft Corporation, 2004)

An example workload for the CMS could be:

- The CMS needs to support 10 concurrent administrators browsing the user interface.
- The CMS should be able to handle 100 concurrent software installations.

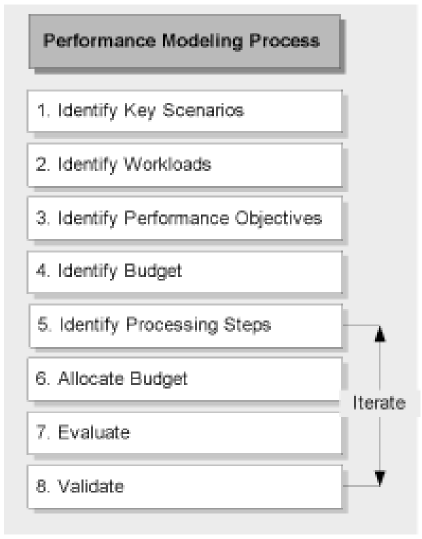

Performance Modeling Process
1. Identify Key Scenarios
2. Identify Workloads
3. Identify Performance Objectives
4. Identify Budget
5. Identify Processing Steps
6. Allocate Budget
7. Evaluate
8. Validate
Iterate

Figure 6: Performance modeling process (Microsoft Corporation, 2004)

Performance objectives determined by business requirements are quantifiable goals for the application. Performance objectives should be written for each performance scenario identified during the first step. Usually, performance objectives involve the following performance indices (Microsoft Corporation, 2004):

- **Response time:** Startup page must be displayed in less than 1 second.
- **Throughput:** The database must support 100 transactions per second.
- **Resource utilization**: CPU, memory, disk and network resource consumption.

If the application has a long lifetime, workload requirements are likely to change over the time. Thus, performance objectives should not be based only on the initial workload requirements, service-level agreements and response times, but also take into account the future growth. (Microsoft Corporation, 2004)

The fourth step is dedicated to identification of budgets. Budgets specify limitations for the corresponding scenarios. If a given budged is exceeded, the application fails to meet its performance objectives. The budged is usually specific by either execution time or resource utilization (Microsoft Corporation, 2004), for example:

- **Execution time:** Page_Load event must not take longer than 1 second.
- **Resource Utilization:** Memory consumption of the client application must not exceed 100 MB.

In addition to common resources such as CPUs, available memory, network I/O and disk I/O, there are other dimensions that may effect on the available budget. Some resources (e.g. CPUs and memory) may be shared among other applications or dependent on underlying software or hardware limitations (e.g. maximum number of inbound connections). From the project perspective, time and costs are notable constraints also. (Microsoft Corporation, 2004)

During fifth step, scenarios are itemized and divided into separate processing steps. This helps identification of critical points within the application that may require custom instrumentation logic to provide actual execution costs and timings. Unified Modeling Language (UML) use cases and sequence diagrams can be used to assist this step. Table 2 shows an example of processing steps of an order submission system (Microsoft Corporation, 2004).

Table 2: Example processing steps (Microsoft Corporation, 2004)

Processing Steps
1. An order is submitted by a client.
2. The client authentication token is validated.
3. Order input is validated.
4. Business rules validate the order.
5. The order is sent to a database server.
6. The order is processed.
7. A response is sent to the client.

Next, scenarios are refined even further. The budged defined for a scenario is spread across the processing steps so that the scenario meets the performance objectives. It is important to note that some of the budget may apply to only one processing step within a scenario, but some of it may apply across multiple scenarios. Execution time and resource utilization are considered during this step. (Microsoft Corporation, 2004)

- **Assigning execution time**

This is accomplished by assigning a portion of the budged for each processing step. If execution time is not known, it is possible to simply by dividing the total time equally between the steps. At this point, most of the allocated values are predictions, which will be re-evaluated after measuring actual time, and therefore, they do not have to be perfect. However, reasonable degree of accuracy is desirable to stay on the right track. Budged allocation shows whether each step has sufficient execution time available. For those that look questionable, it is important to conduct some further experiments, for example with prototypes, to verify actual state prior to proceed. (Microsoft Corporation, 2004)

- **Assigning resource utilization**

When assigning resources to processing steps, there are four important things to consider (Microsoft Corporation, 2004):

1. Find out the cost of the materials. For example, how much API X1 does costs in comparison to API Y2.
2. Find out the budged allocated for hardware. This budged defines how much resources

are available.

3. Find out what hardware systems are already in places and can be utilized.
4. Know the functionality of the application. For example, some applications may utilize more CPU and some may require more network capacity due to continuous web access.

The results from the previous step should be evaluated during step 7 before actual prototyping and testing. Early evaluation helps to modify the design, revise requirements or change the budget allocations before spending unnecessary time and effort. Firstly, verify whether the budget is realistic and meets the objectives. In case of a performance problem, the model should identity possible resource hot spots. Then, alternatives that are more efficient should be evaluated starting from the high-level design choices down to the code-level specifics. Finally, it is time to evaluate whether there are any tradeoffs involved. Is productivity, scalability, maintainability or security traded for performance? (Microsoft Corporation, 2004)

Validation should be an ongoing activity during the process. Validation involves creation of prototypes and measurement of actual budgets of scenarios by capturing performance metrics. Collected data is used to validate the models and estimates. Accuracy of the validation evolves during the project. Early on, validation is based on prototypes, thus results may be inaccurate. Later, measurements can be more precise because validation can be done against actual code. (Microsoft Corporation, 2004)

4.2.2 The software performance engineering process

Figure 7 presents a software engineering process introduced by Smith (Smith, 2001). Similarly compared to the eight-step performance modeling process, it advocates performance modeling throughout the software lifecycle to identity potential performance problems early in the development. On the high level, two segments. The right-hand side consists of early-cycle performance prediction activities and the left-hand side involves performance measurements used to verify and validate modeling results against working software.

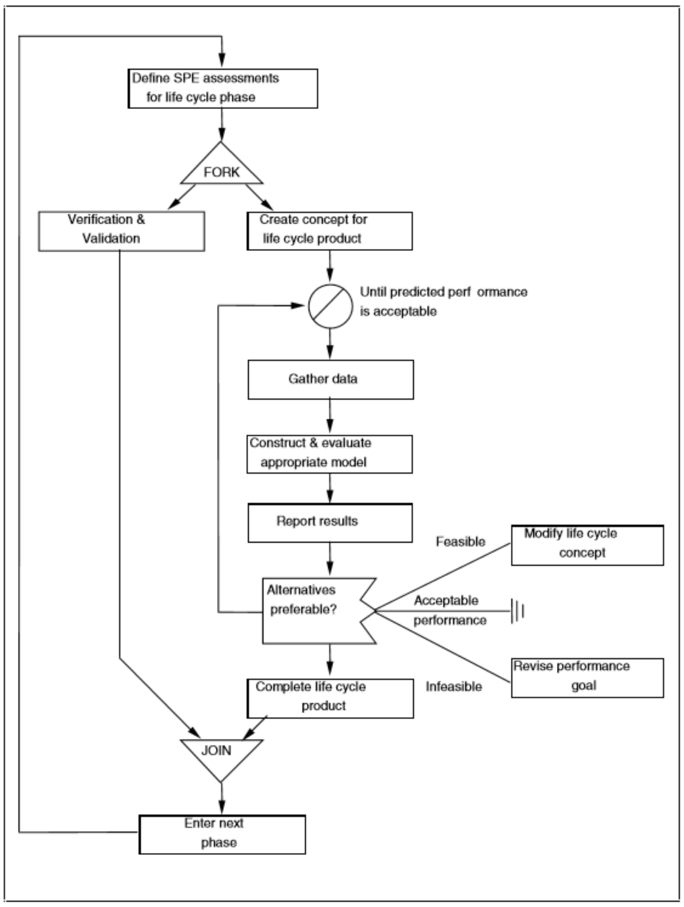

Define SPE assessments for life cycle phase
FORK
Verification & Validation
Create concept for life cycle product
Until predicted perf ormance is acceptable
Gather data
Construct & evaluate appropriate model
Report results
Feasible
Modify life cycle concept
Alternatives preferable?
Acceptable performance
Revise performance goal
Infeasible
Complete life cycle product
JOIN
Enter next phase

Figure 7: The software performance engineering process (Smith, 2001)

Starting from the prediction-side, the first step is to define the SPE assessments for the current ongoing lifecycle phase. This data is used to determine whether the software meets its quantitative performance objectives. The performance objectives may wary, depending on the developed system, from overall responsiveness as seen by the users to more specific response time and throughput requirements, which are both certain measures of responsiveness. Additionally, efficiency in terms of resource usage may be considered in this stage if some important computer resource requirements must be satisfied. (Smith, 2001)

The second step is to create the concept for the lifecycle product, which changes during the development process. Early the concept is the architecture, the requirements and high-level designs for satisfying those. Later in the development the concept is, for example, the software design, implemented algorithms and code. During this phase SPEs general design principles, patters and anti-patterns are used to create responsive designs. (Smith, 2001)

Next, performance engineers estimate lifecycle concepts. They collect performance data to create performance predictions. This is achieved by utilizing performance models which are based on projected typical usage performance scenarios and software components, in addition to best-case, worst-case and failure scenarios. The process moves to next phase if the model results indicate satisfying performance. If not, the models indicate critical components whose resource usage should be further analyzed. This iteration constructs more detailed performance models that help to further refine the design concepts. Performance engineers report results with possible alternative strategies and expected improvements to developers who review those. If an alternative is found to be feasible, developers modify the concept according to it. If not, the original performance objective is modified to reflect degrade in performance. (Smith, 2001)

As aforementioned, results from performance models are predictions of the system performance. Therefore, it is vital to verify that the performance models present the actual software execution and to validate the predictions against measured performance data. If measured results differ from predictions, performance models must be re-calibrated and updated to represent the actual behavior of the system. This validation and verification phase should begin early, based on early prototypes and continue throughout the lifecycle. (Smith, 2001)

4.2.3 Q-Model

The Q-Model presented by Cortellessa et al. is based on a conventional waterfall software development process. Additionally, the Q-Model takes inspiration from the familiar V-model for software validation (Cortellessa, et al., 2011). The waterfall model, presented in figure 8, implements the fundamental stages for software development process described in chapter 3.1.

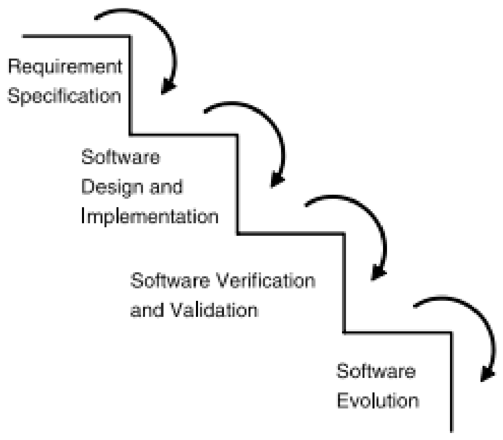

Requirement Specification
Software Design and Implementation
Software Verification and Validation
Software Evolution

Figure 8: The waterfall software development process (Cortellessa, et al., 2011)

The waterfall model presents a sequential development process in which the progress is flowing downwards through series of steps. Each step produces software artifacts that further describe the software under development. Artifacts from the previous step are used as an input for the next step. In traditional software analysis, software models can be used to produce and better understand these artifacts. Similarly, performance models can be created to produce performance related artifacts (e.g. performance objectives) from each step (Cortellessa, et al., 2011), for example:

1. Software requirement specification phase produces the requirement specifications document, including performance requirements.
2. Performance models can be created based on the requirements specifications.
3. Performance models can be used to further refine performance requirements and analyze software designed software.

The Q-Model refines the waterfall model by applying similar process for all of its steps. The result is presented in figure 9. Each software development phase on the left-hand side is connected to the corresponding performance analysis activities on the right-hand side through the performance model generation activities. The bottom section represents the implementation of the software and monitoring of its behavior. (Cortellessa, et al., 2011).

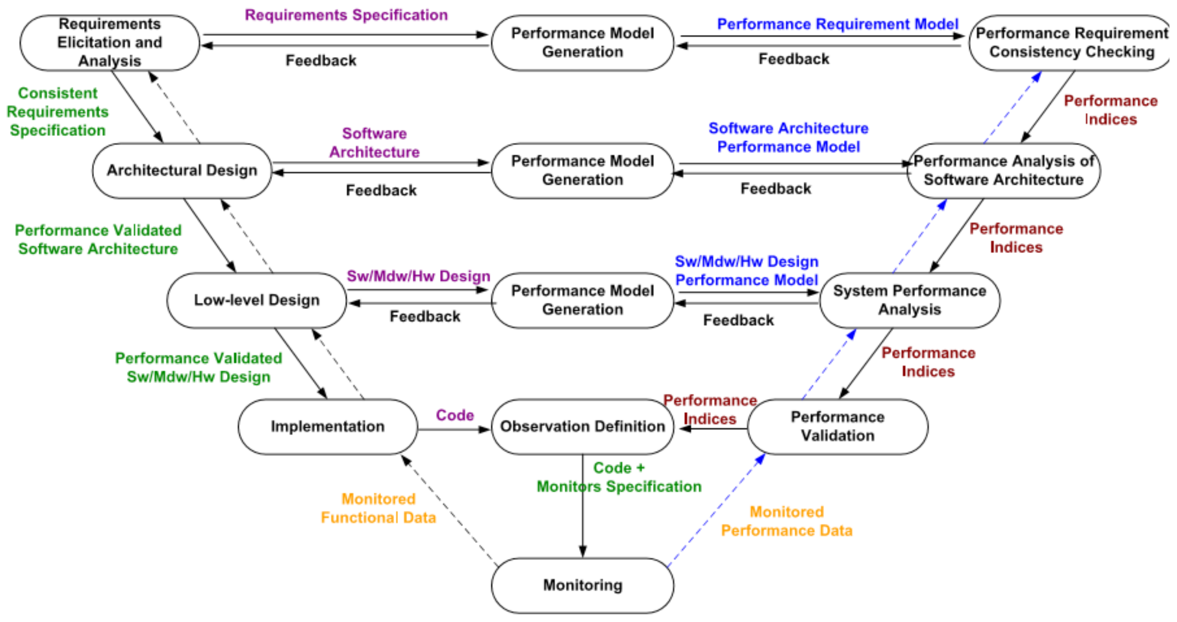

Requirements Elicitation and Analysis
Requirements Specification
Feedback
Performance Model Generation
Performance Requirement Model
Feedback
Performance Requirement Consistency Checking
Consistent Requirements Specification
Performance Indices
Software Architecture
Architectural Design
Feedback
Performance Model Generation
Software Architecture Performance Model
Feedback
Performance Analysis of Software Architecture
Performance Validated Software Architecture
Performance Indices
Sw/Mdw/Hw Design
Low-level Design
Feedback
Performance Model Generation
Sw/Mdw/Hw Design Performance Model
Feedback
System Performance Analysis
Performance Validated Sw/Mdw/Hw Design
Performance Indices
Implementation
Code
Observation Definition
Performance Indices
Performance Validation
Code + Monitors Specification
Monitored Functional Data
Monitored Performance Data
Monitoring

Figure 9: The Q-Model for a waterfall process (Cortellessa, et al., 2011)

The Q-Model maps the common development process stages with the following notation (Cortellessa, et al., 2011):

- Requirement specification stage is renamed to requirement elicitation and analysis.
- Software design and implementation stage is partitioned into three stages: architectural design, low-level design and implementation.
- Software verification and validation stage is partitioned and represented by the middle and the right-hand side of the model.

Transitions in the Q-Model are described as follows (Cortellessa, et al., 2011):

- **Downstream vertical arrows on the left-hand side**

  Represent the transfer to the next stage. Transfer to the next stage is not allowed before appropriate performance analysis activities are completed. For example, before software architecture is valid from performance point of view corresponding performance models must be made and analyzed.

- **Downstream vertical arrows on the right-hand side**

  Represent the information flow between performance analysis activities. The previous stage produces performance boundaries that maybe be used as a reference values for the next phase. For example, performance constrains from architectural stage (e.g. "the maximum number of simultaneous database connections is 100") are the architectural limits that must be considered during low-level design.

- **The lowest horizontal arrows**

  On the left side the arrow represents the definition of suitable observation functions based on the running code. On the right side it presents the validation of performance indices using the observation functions.

- **The bottom vertex**

  This arrow presents the monitoring activity that receives what to monitor from observation definition process that is based on the executing code and the performance indices to validate.

- **Upstream vertical arrows on both side**

  Performance problems may arise during later stages of the process. The monitoring activity at the bottom provides feedback for the both sides. Because some issues may not be corrected without re-executing the previous stages, the feedback traverses up along both sides inducing changes on the corresponding software artifacts and performance models.

4.2.4 Converged SPE process

In their paper, Woodside et al. (Woodside, et al., 2007) express their view on the state of the software performance engineering, and they are not very satisfied about it. Current performance processes require heavy effort, and therefore, are not suitable for everyone. There are no standards in performance measurement and there is a semantic gap between software performance and software functionality. Lack of trust and understanding on performance models is a common problem. Lastly, performance modeling is effective but often costly. Models are approximations that may leave out important details, and are difficult to validate. Presented solution is to combine measurement and modeling methods into a single Performance Knowledge Base.

Before going into the process, the SPE domain considered in this chapter consists of the following SPE activities summarized in figure 10 and described below. (Woodside, et al., 2007):

- **Identify performance concerns** including important system operations (use cases) and resources. Resources are system elements that offer services for other parts of the system. Resources have limited capacity and include hardware (e.g. CPU and I/O), logical resources (e.g. buffers and locks) and processing resources (e.g. processes and threads).

- **Define and analyze requirements**. This activity requires identification of operational profile, different workload intensities, delay and throughput requirements and system behavior. Operational profile describes a subset of system operations important from the performance point of view. Workloads define frequency of the system operations and behavior is defined by various scenarios (e.g. UML behavior notation or execution graphs).

- **Performance prediction** by modeling the interaction of the behavior with available resources. Consider scenarios, architecture and detailed design.
- **Performance testing** on entire system or part of it under different load conditions.
- Use performance models to predict the effect of changes and new features during **product maintenance and evolution period.**
- Perform **total system analysis** where the planned software system is considered in the complete and final deployed system.

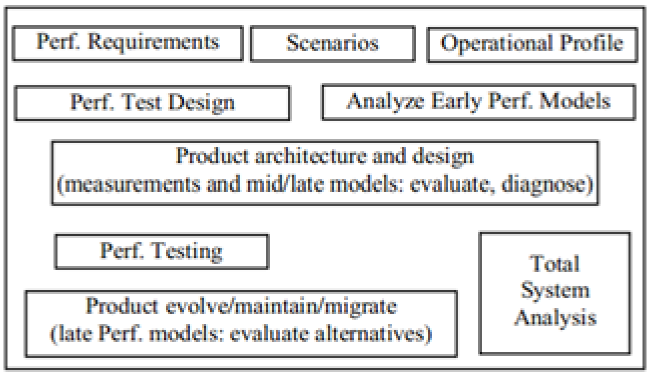
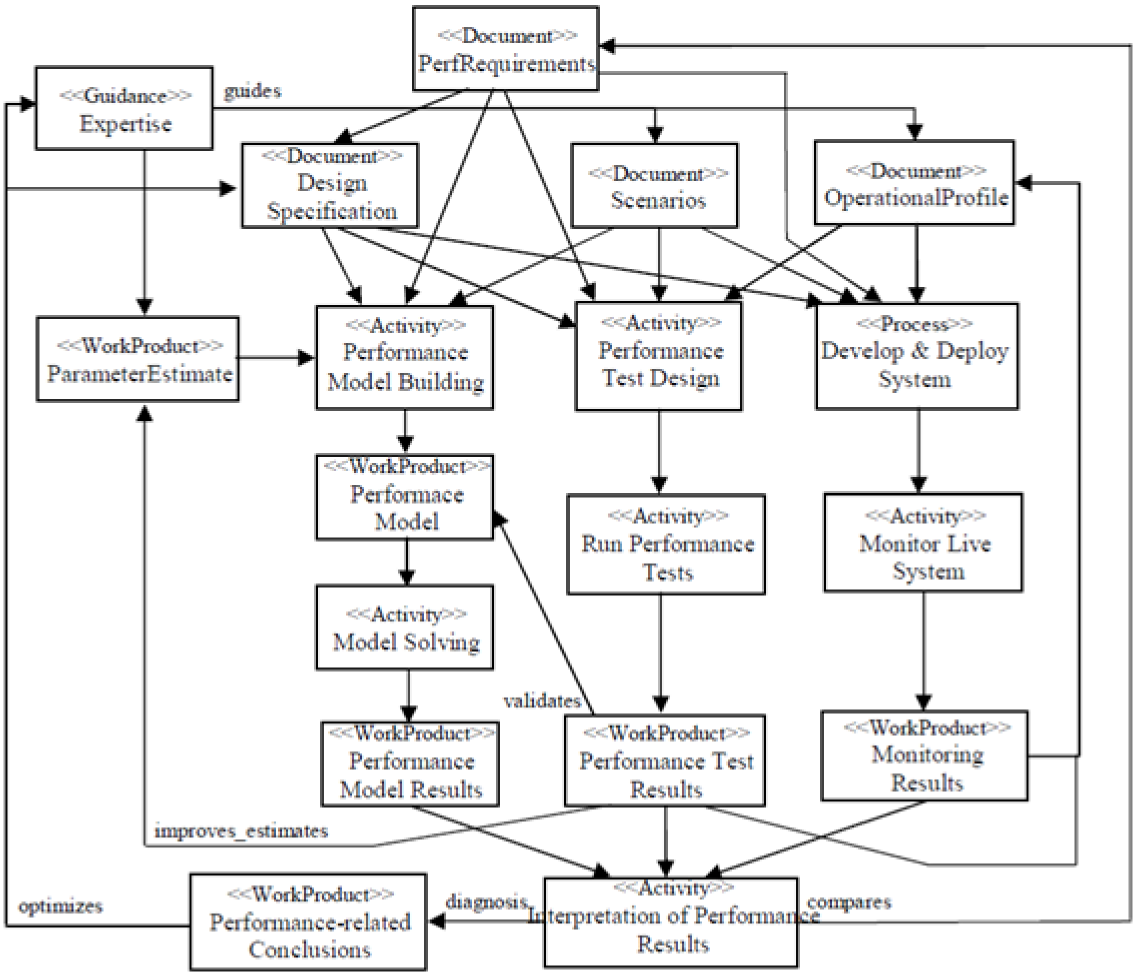

Perf. Requirements
Scenarios
Operational Profile
Perf. Test Design
Analyze Early Perf. Models
Product architecture and design
(measurements and mid/late models: evaluate, diagnose)
Perf. Testing
Total System Analysis
Product evolve/maintain/migrate
(late Perf. models: evaluate alternatives)

Figure 10: SPE activities (Woodside, et al., 2007)

Figure 11 presents the converged SPE process that converges shattered knowledge of different kinds and from various sources into a single domain model. The left-hand side of the process consists of concepts related to the performance modeling, and the right-hand side consists of performance measurement activities, where distinction is made between performance tests in a laboratory environment and live production system monitoring. (Woodside, et al., 2007)



<<Document>> PerfRequirements
<<Guidance>> Expertise
guides
<<Document>> Design Specification
<<Document>> Scenarios
<<Document>> OperationalProfile
<<WorkProduct>> ParameterEstimate
<<Activity>> Performance Model Building
<<Activity>> Performance Test Design
<<Process>> Develop & Deploy System
<<WorkProduct>> Performace Model
<<Activity>> Run Performance Tests
<<Activity>> Monitor Live System
<<Activity>> Model Solving
validates
<<WorkProduct>> Performance Model Results
<<WorkProduct>> Performance Test Results
<<WorkProduct>> Monitoring Results
improves_estimates
optimizes
<<WorkProduct>> Performance-related Conclusions
diagnosis
<<Activity>> Interpretation of Performance Results
compares

Figure 11: Domain model for the converged SPE process (Woodside, et al., 2007)

The notation of the converged SPE process is based on the Software Process Engineering Metamodel (SPEM) standard (OMG, 2008). "At the core of SPEM is the idea that a software process is a collaboration between abstract active entities called *ProcessRoles* (e.g., use case actors) that perform operations called *Activities* on concrete entities called *WorkProducts*. Documents, models, and data are examples of *WorkProduct* specializations. Guidance elements may be associated to different model elements to provide extra information." (Woodside, et al., 2007)

Similarly to SPE process introduced in the previous chapters, the converged model incorporates performance model building and solving early in the development to get initial performance figures out of the design. Performance tests are used to validate and to enhance the models, and to ensure that original performance requirements are met. In addition, iteration improves team expertise and provides valuable feedback for the future work.

4.3 Performance requirements

Identification and analysis of performance requirements is a key part of every SPE process presented. Use case and risk analysis, architecture and system design, performance modeling and performance measurements are all dependent different data prerequisites. There data requirements for performance are: performance objectives, workload definitions, software execution characteristics, execution environment descriptions, and resource usage estimates. (Smith, 2001)

Software performance is evaluated by comparing gathered performance data against performance objectives. Precise and qualitative metrics are vital to determine whether performance objectives are met. As aforementioned, performance objectives usually include response time, throughput and utilization requirements. (Smith, 2001)

Workload requirements specify the performance scenarios. Initially, scenarios specify operations that are the most frequently used. Later, scenarios also cover resource intensive operations. Scenarios can be divided into interaction workload definitions (e.g. number of concurrent users), batch workload definitions (e.g. programs on critical path, their dependencies and data volumes). (Smith, 2001)

Software processing steps identity software components most likely to be invoked, invoke frequency and execution characteristics per scenario. The execution environment defines the computer system configuration (e.g. CPU, memory and I/O). Resource usage estimates the available resource budged, that is how much system resources each performance scenario requires. (Smith, 2001)

4.3.1 Challenges for managing performance requirements

There are several challenges involved when working with data requirements. The most important ones are listed below:

- **Performance requirements have a global impact on the software throughout the development process**

  It is not possible simply to add a new module to fix performance issues. It may require significant changes to different parts of the system. Therefore, performance requirements should be considered system wide throughout the development process. (Nixon, 2000)

- **Trade-offs among requirements and implementation techniques**

  Conflicting and interacting nonfunctional requirements (NFRs) and implementation choices can potentially lead into trade-offs in the final product. For example, comprehensive response time optimizations may decrease flexibility for future changes. (Nixon, 2000)

- **Variety of implementation techniques**

  During development, several choices must be made between alternative implementation techniques available, each having different performance characteristics. (Nixon, 2000)

- **Incompleteness of the specification**

  Requirement specifications can be too abstract. Design approaches and algorithms can be still open at early stages. Additionally, environment and its components to be used may be undecided leading to uncertain final computational requirements. (Petriu & Woodside, 2002)

- **Unawareness of the production workload intensity**

  For example, the number of end users may be unknown during the performance requirement specification stage. (Petriu & Woodside, 2002)

- **Lack of investment in obtaining performance requirements**

  Requirements are often not obtained in any dept, or validated for realisms, consistency and completeness. Developers tend to underrate importance of performance requirements. (Smith, 2001)

- **Identification of performance scenario for new functions**

  Performance scenarios can be easily derived for systems replacing previously implemented systems. However, it may be difficult to identify scenarios for new functions. Web applications the number of potential users is may be very high and highly variable. (Smith, 2001)

- **Identification of processing steps for existing systems**

  Identification of software processing steps relatively easy for new applications, but more difficult for the software that has been already been built. (Smith, 2001)

- **Shifting user satisfaction**

  As systems evolve, users tend to demand more of them. At the same time, responses should be faster. Moreover, users do not know precisely how to use the system and tend to expect the system to modify their way to work. (Smith, 2001)

4.3.2 Performance Requirements Framework (PeRF)

The Performance Requirements Framework (PeRF) (Nixon, 2000) is a framework for managing performance requirements. It consists of structured, systematic collections of information, called catalogues of knowledge, to deal with the large body of knowledge (e.g. performance concepts and development techniques). At any time, developers can utilize the information that will help to deal with the large number of decisions involved. The premise of the framework is that "performance and development knowledge must be organized". (Nixon, 2000)

Inputs for a process that utilizes PeRF to manage performance requirements include: Catalogues of knowledge of particular domain, performance requirements and functional requirements, priorities for the organization and the system, expected workload, development techniques, and iterations and trade-offs among performance requirements. Respectively, the process produces the following outputs: a record of development decisions made, reasons for the decisions, list of performance requirements that are met and to what extent, a record of the iterations and trade-offs among performance requirements, priorities, workload, decisions, and rejected alternatives. Optionally, PeRF may produce a prediction of performance of the system. (Nixon, 2000)

PeRF treats all performance requirements as NFRs, and represents those using the NFR Framework (Mylopoulos, et al., 1992), a framework for representing and utilizing nonfunctional requirements. NFR Framework provides a process-oriented approach to meeting quality objectives. It is based on the view that "the quality of a software product is largely determined by the quality of the process used to construct it". Development decisions justified are based on whether or not they help to meet certain NFRs. (Nixon, 2000)

NFR Framework calls NFRs softgoals due to their nature. They may not be fully satisfiable and tend to change. Definition of NFRs is often ambiguous. Meeting them may be matter of balancing trade-offs. Softgoals are then analyzed, interrelated and prioritized. Based on this, the impact of decisions upon NFRs is determined. All this data is represented in softgoal interdependency graphs (SIGs). An example of an initial performance goal and a refinement is presented in figure 12. (Nixon, 2000)

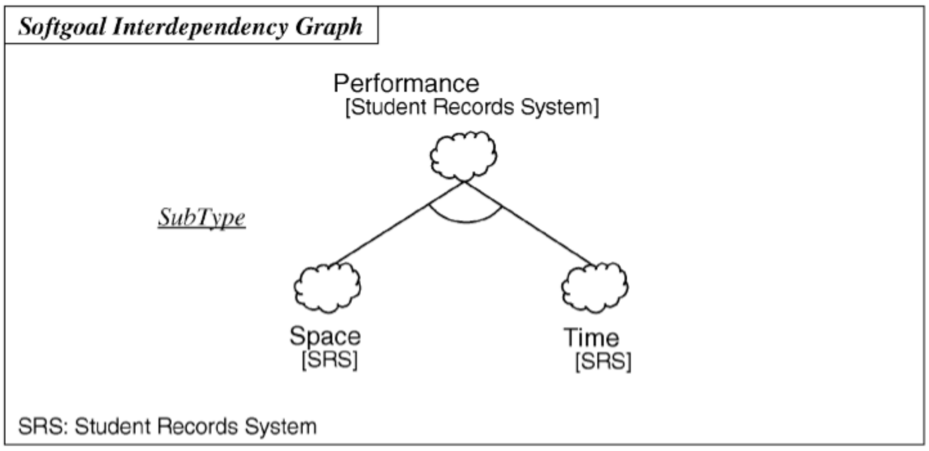


Figure 12: An initial performance softgoal and a refinement. (Nixon, 2000)

Performance [Student Records System] is a NFR softgoal stating that there should be good performance for the student record system. All softgoals have an NFR type (e.g. Performance or Security) and one or more parameters (e.g. [Student Records System]). PeRF extends the Performance type in the NFR Type Catalogue by providing a Performance Type Catalogue presented in figure 13. The Performance Type consists of subtypes Time and Space that describe time and space performance requirements for the system. Additionally, Time and Space can be refined by their own subtypes. PeRF defines decomposition as the process of refining softgoals into other softgoals. Developers use their expertise to decide what to things refine and how much refinement is actually needed. (Nixon, 2000)

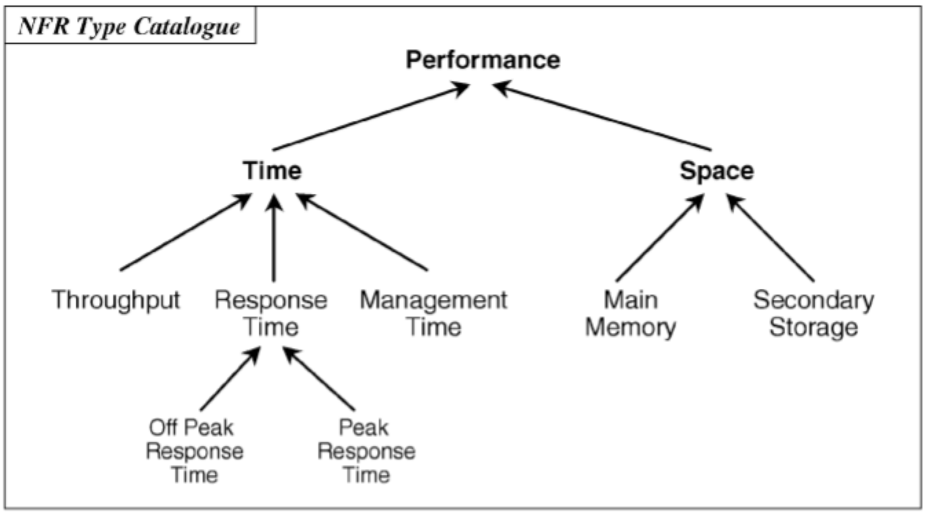


Figure 13: The performance type catalogue. (Nixon, 2000)

Figure 13 represents interdependency between the parent and its offspring. Performance[SRS] is refined downwards into Space[SRS] and Time[SRS] subgoals. Respectively, Space and Time contribute upwards to Performance. The interdependency presented in figure 17 can be read as “Space[SRS] AND Time[SRS] SATISFICE Performance[SRS]”. In addition to aforementioned AND contribution, PeRF defines OR, MAKES, BREAKS, HELPS and HURTS contributions to determine the state of the parent when its subgoals are satisfied. (Nixon, 2000)

PeRF provides similar tools for prioritizing softgoals, stating reasons for development decisions, representing implementation techniques used to realize softgoals, evaluating impact of decisions, and catalogues of methods that offer and organize techniques and principles needed in software development. Figures 14 and 15 present catalogues of decomposition and operationalization methods. These catalogues document the process and provide a kind of roadmap to follow.

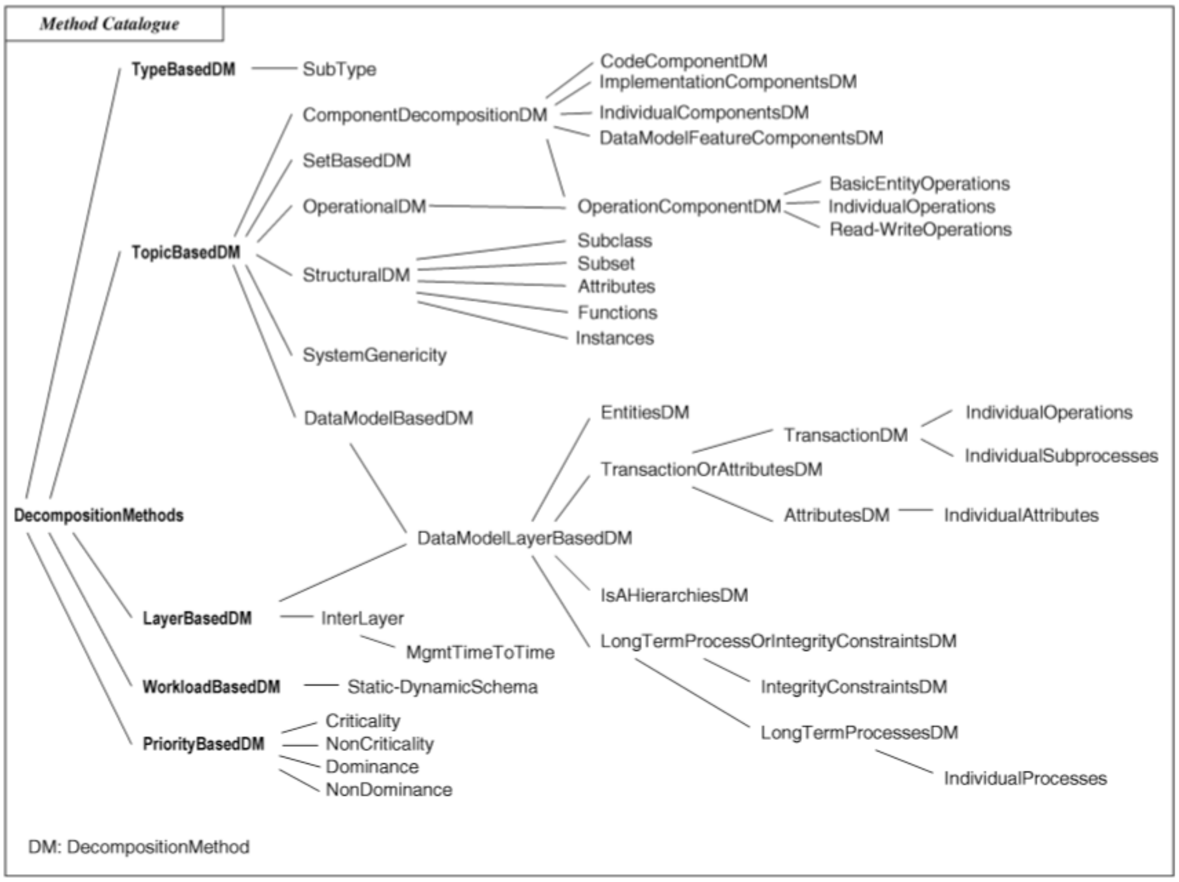


Figure 14: Catalogue of decomposition methods (Nixon, 2000)

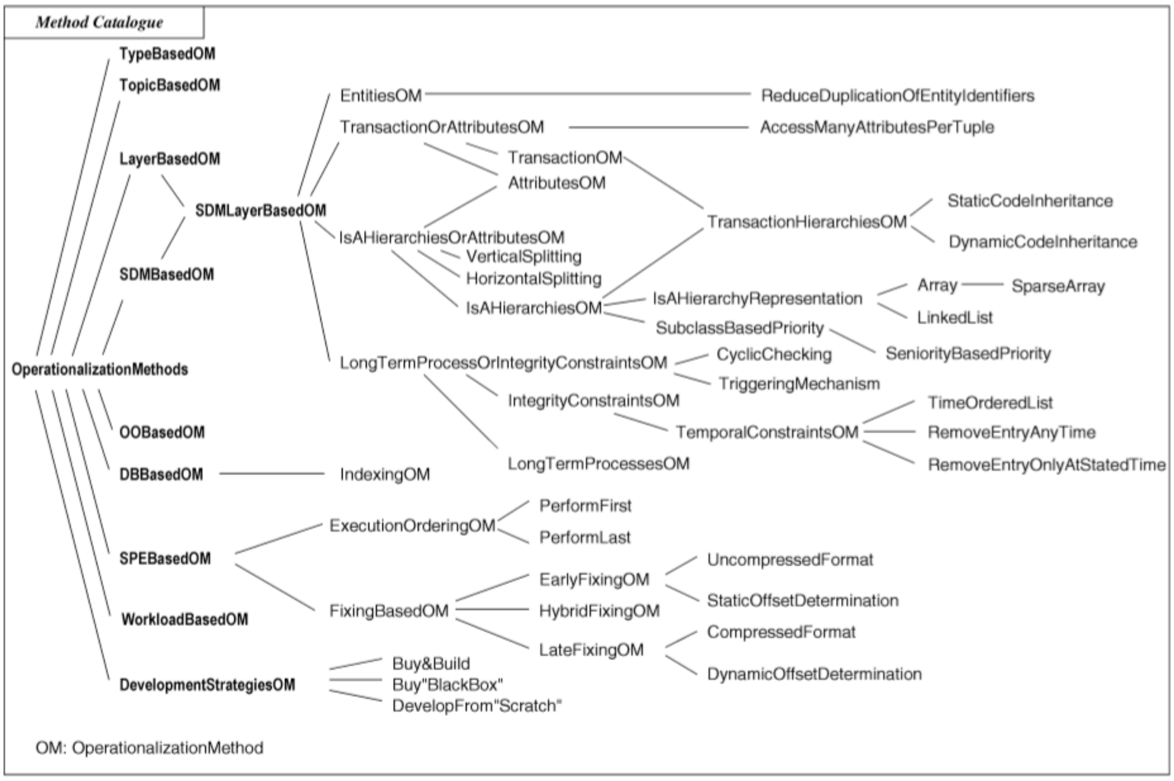


Figure 15: Catalogue of operationalization methods (Nixon, 2000)

Methods in the catalogues are grouped in a way that left side consists of more coarse-grained groups. The first subgroup level shows the major groupings of methods. These deal with the structure of softgoals (type, topic, layer and priority), sources of methods (e.g. SPE, Semantic Data Models (SDMs), databases and object-oriented systems), system characteristics such as workload, and implementation strategies. Subgroups become more precise when moving to the right. For example, if the source of the method is SPE, then next layer shows particular SPE principles. Similarly to softgoal refinement, developers use their experience and expertise to determine which methods to utilize in which context. (Nixon, 2000)

PeRF is a comprehensive framework, which provides plenty of tools to manage performance requirements. This chapter introduced few key concepts that would be useful for this work. Initially, performance critical features should have their performance requirements defined. Then, developers may use composition methods to concretize those abstract high-level requirements. Going through this process hopefully encourages developers to perform similar actions in the future, and build experience and confidence on the process.

4.4 Performance modeling notations

Software modeling is a process of describing software by using ad-hoc models. Software description consists of static (modules and components) and dynamic (behavior at run time) aspects of a software system. There are many different notations available to describe either the statics or dynamics, but in this context, the behavioral description is more interesting because performance is an attribute of the system dynamics. (Cortellessa, et al., 2011)

In software modeling, UML has become the standard modeling language for complex computer systems. Yet, software performance modeling lacks such widely adopted standards. Gap between notations for modeling the software and notations for modeling software performance is apparent. The multiplicity of performance notations allows developers to select the most suitable one for each application domain and model. On the other hand, the use of different notations may require different definitions for the same concept and obtained results may become hard to compare between different models. This chapter briefly introduces some of the popular notations for performance modeling. (Cortellessa, et al., 2011)

Object Management Group (OMG), a consortium behind UML, has standardized UML Profile for Schedulability, Performance and Time (UML-SPT) (OMG, 2005), an extension to enable performance and scheduling analysis of UML models. The profile enables quantitative analysis and predictive modeling by specifying means to annotate quantitative information into UML models (Cortellessa, et al., 2011). For example, figures 16 and 17 describe a simple scenario modeled with UML sequence diagrams with performance annotations.

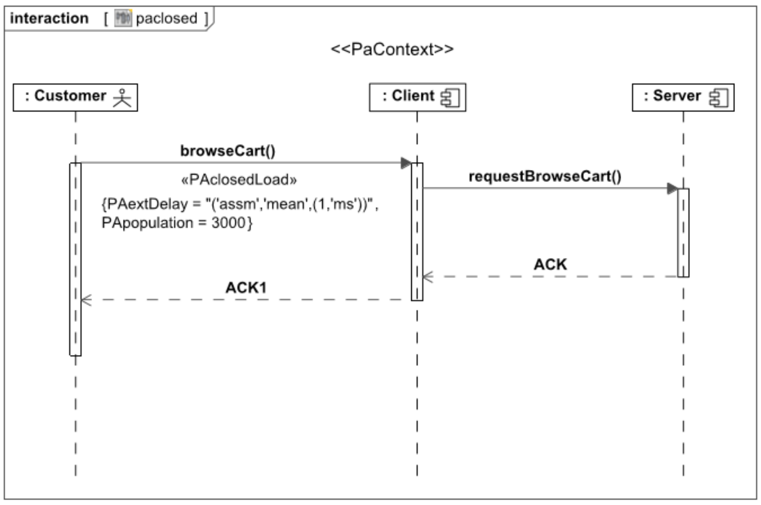


Figure 16: A UML sequence diagram describing the workload (Cortellessa, et al., 2011)

In this example, performance context («PAcontext» stereotype) describes a browse cart activity requested by customers. In figure 16, workload for the scenario is defined by «PAclosedLoad» stereotype with a fixed number of customers (PApopulation = 3000) with each spending an assumed mean external delay of 1 ms between requests. In figure 17, «PAstep» stereotype models a step in the scenario. The requestBrowseCart activity is annotated with a tag that indicates measured mean service time demand of 5 ms (PAdemand). (Cortellessa, et al., 2011)

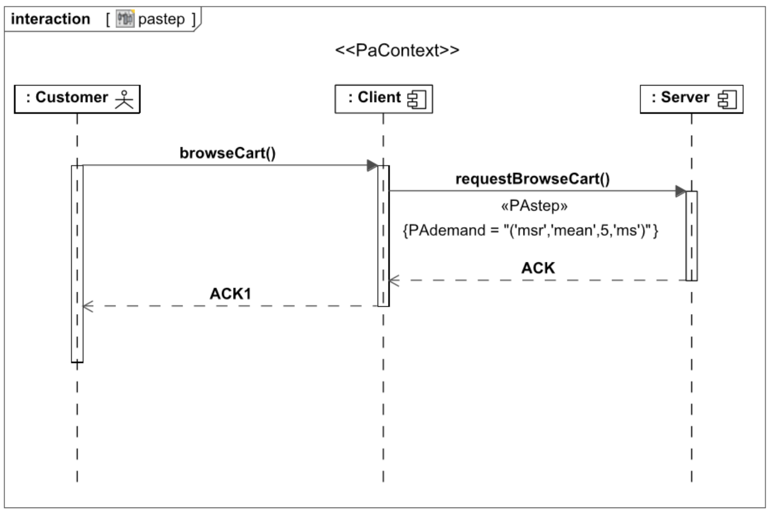


Figure 17: A UML sequence diagram describing a step (Cortellessa, et al., 2011)

As mentioned earlier there are many different kinds of notations available. For sake of this chapter interested may refer to (Cortellessa, et al., 2011) and become acquainted with Markov processes, Queuing Networks (QNs), Layered Queuing Networks (LQNs), Execution Graphs (EG), Stochastic Petri Nets (SPNs) and Stochastic Process Algebras (SPAs).

Previously listed notations require definition and construction of specific models that then needs to be solved with appropriate methods. Another approach is to use simulation models. A simulation model is software that describes the dynamic behavior of a target system. The model can be used to collect behavioral traces or execution times of simulated operations. Simulation models can be implemented to simulate any specified behavior at different levels of abstraction. Results produced by the model can be very accurate if the software is instrumented properly. The main drawback of the simulation is that it the model requires thorough design, implementation and verification work to ensure it can be substituted to the real system. (Cortellessa, et al., 2011)

4.5 Performance measurement frameworks

The previous chapter described some of the performance modeling notations used to predict software performance before actual implementation. After parts of the software system are implemented, either as prototypes or as final components, it is possible to conduct performance measures to analyze performance of the software.

The literature contains many reports of different performance measurement techniques. The frameworks can be divided into two categories: *black-box* and *white-box* frameworks. Black-box solutions are aimed for software system which internal behavior is unknown. Usually such a system is composed of software from many different vendors without access to source code (Aguilera, et al., 2003). White-box solutions, also referred as annotation-based monitoring schemes (Sigelman, et al., 2010), require custom instrumentation points inserted into critical parts of the code.

To begin with, httperf (Mosberger & Jin, 1998) and NetLogger (Tierney, et al., 1998) are early contributions associated with the topic. httperf is a simple client-side tool for measuring web server performance. It does not require source code access or extensive application specific knowledge. It can be configured to send requests to a server at fixed rate. Increasing the rate at which requests are sent an analyst can detect when the server becomes saturated, and reached its maximum throughput. NetLogger, on the other hand, provides custom event logging framework used to collect event lifelines out from application components and operating system components. NetLogger calls that generate logs must be implemented to specific parts of the application.

Many newer frameworks utilize similar principles with httperf and NetLogger, but overcome limitations of their predecessors. For example, httperf is capable of producing accurate client-side statistics but is unable to pinpoint sources of server-side issues. NetLogger's principle is to log as much information as possible under realistic conditions (Tierney, et al., 1998). This is problematic because incautious addition of logging generates too many events, concealing actually interesting events and complicating detection of true issues (Reynolds, et al., 2006a).

The rest of this chapter comprises few other performance measurement frameworks starting from the ones requiring no access to the source code, and followed solutions utilizing source code instrumentation.

4.5.1 Black-box techniques

WAP5 (Reynolds, et al., 2006b), mBrace (van der Zee, et al., 2009), Chopstix (Bhatia, et al., 2008) and Whodunit (Chanda, et al., 2007) are example of performance analysis frameworks that do not require source code instrumentation but are able collect adequate data for performance analysis.

Whodunit and mBrace enable action-based measurement by tracing the flow of data through multi-tier web applications in Apache environment. Whodunit provides custom wrappers for critical system functions (e.g. *pthread_mutex_lock*, *event_add* and *send/receive*) and some modifications to MySQL server that will track the transaction context used to identify the current action (Chanda, et al., 2007). mBrace relies on customized Apache modules that will process incoming HTTP requests. Modules will identify the request, track used CPU cycles and log results to the database. Additionally, mBrace provides custom MySQL client library that exposes an interface for executing traced SQL queries. (Chanda, et al., 2007)

WAP5 provides an interposition library, LibSockCap, to capture a trace of all networking system calls. Similarly to Whodunit's trace collection library, LibSockCap consists of system call wrappers to log all socket-API activity. WAP5 uses trace reconciliation and causal message linking algorithms to constitute trace logs. (Reynolds, et al., 2006b)

Chopstix is a diagnostics tool that continuously collects low-level OS events (e.g. *CPU utilization*, *I/O operations*, *page allocations* and *locking*) using a data collector component and aggregates raw data at multiple timescales (e.g. 5 minutes or 1 hour). Then, the data can be visualized using the visualization component. For example, Chopstix is able to answer the question "what was the system doing last Wednesday around 5pm when the ssh prompt latency was temporarily high, yet system load appeared to be low?" (Bhatia, et al., 2008).

4.5.2 Source code instrumentation techniques

This chapter introduces Magpie (Barham, et al., 2004), Dapper (Sigelman, et al., 2010), Pip (Reynolds, et al., 2006a) and performance assertions (PA) (Vetter & Worley, 2002) which are techniques that utilize source code instrumentation.

Magpie is Microsoft's contribution and provides end to end request tracking capabilities. Magpie's instrumentation is built on Event Tracing for Windows (ETW). ETW logs contain events collected from kernel to track CPU utilization and disk I/O, the WinPcap library to capture transmitted and received packages, and custom application and middleware instrumentation to capture application specific behavior. Magpie request parser links related events together in order to track individual requests throughout the system. (Barham, et al., 2004)

Dapper is Google's performance tracing utility, which is successfully deployed in their production environment. Dapper is able to trace the flow of requests through large-scale distributed environment. Dapper's core instrumentation is restricted into a small set of common threading, control flow and RPC (Remote Procedure Call) libraries, which makes it transparent to application developers. Dapper minimizes the overhead by optimizing trace collection and by recording only a fraction of all traces (sampling). (Sigelman, et al., 2010)

Pip provides a set of tools for source code instrumentation and checking application behavior. The tool chain consists of a custom middleware library that generates instrumentation automatically. In addition to that, programmers may add more annotations to anywhere they want. Expectations that define expected application behavior are fundamental requisites for Pip's operation. Pip checks all traced behavior against the expectations to expose structural errors and performance problems. All recorded traces are stored in an SQL database and can be visualized via GUI. (Reynolds, et al., 2006a)

Similarly, PA system provides capability for developers to assert performance expectations to the code. The main conceptual compared to Pip is that PA combines instrumentation and performance expectations into single notation. In other words, when a developer annotates an individual code segment with performance assertions using the PA language (e.g. *pa_start(&pa, '$nInsts / $nCycles > 0.8');* **<code>** *pa_end(pa);*) the PA runtime automatically configures any necessary instrumentation. PA limits the amount of data that must be processes during performance analysis by highlighting only those parts of the code that fails to meet the defined expectations. (Vetter & Worley, 2002)

4.5.3 Summary of contributions

This chapter wraps up contributions introduces in the previous sections. Each solution is listed along key features and other highlights valuable in this context. The summary is shown in table 3. One important thing to note is that the table does not list low overhead as a key feature, because it seems to be a common target for all. Each solution attempts to have negligible performance impact on monitored system.

Table 3: Summary of performance measurement frameworks

Name	Key features / design goals	Highlights
WAP5	• Black-box solution • Reveal causal structure and timing of communication • Wrappers for system functions	• Trace request path and timing (causal structure of communication) • Modified middleware / system library
mBrace	• Black-box solution • Request-based monitoring for multi-tier web applications • MySQL query profiling	• Use of custom HTTP modules • Link SQL queries to HTTP requests
Chopstix	• Black-box solution • Collect low-level OS events • Maintain comprehensive long time history logs • Continuous monitoring in production environment	• Collection of low-level OS events and • Could be used in all operating systems (not only Linux) • Use of sampling (sketches) to reduce overhead
Whodunit	• Black –box solution • Request-based monitoring for multi-tier applications • MySQL query profiling • Wrappers for system functions	• Link SQL queries to HTTP requests • Modified middleware / system library
Magpie	• No need to propagate request identifiers through the system • Use of event logs (ETW) • Scalability	• Windows / IIS / SQL Server environment • Trace request path, timing and resource demands
Dapper	• Application-level transparency • Ubiquitous deployment • Continuous monitoring in production environment • Scalability	• Use of sampling to reduce overhead • Minimized data collection • Can be used as a general monitoring tool • Application transparent tracing

Pip	• Compare actual behavior against expected behavior • Detect structural errors and performance problems • Automatic instrumentation	• Application behavior analysis • Understand expected behavior during development
PA	• Assert performance exceptions directly to source code • No additional instrumentation needed • Highlight only parts of code that fail to meet the expectations	• Concentrates only on true problems and discards others • Understand performance goals during development

4.6 Best practices

Smith and Williams present 24 best practices for SPE, which are divided into four categories: *project management*, *performance modeling*, *performance measurement* and *general techniques*. Selection of the best practices is based on their experience in SPE techniques, interviews and discussions, and observations from real life scenarios of companies that are successfully applying SPE in their development process. (Smith & Williams, 2003)

4.6.1 Best practices in project management (1-11)

The best practices in project management focus on risk management, integration of SPE into development process and ensuring that SPE actions are executed. This chapter contains 11 useful advices (Smith & Williams, 2003):

1. Perform an early estimate of performance risk.

At first, the level of performance risk should be estimated. Is it possible that failing to meet the performance goals due to inexperienced developers, use of new technologies or working with tight schedule may jeopardize the success of the project? If so, the project has a performance risk.

The following guidelines help to estimate the performance risk (Smith & Williams, 2003):

i. Identify potential performance risks.

ii. Determine impact of identifier risks. Focus on probability of happening and the severity of damage that may occur.

iii. Rank risks based on their anticipated impact to address them systematically.

2. **Match the level of SPE effort to the performance risk.**

After possible risks are identified and estimated, the level of risk should be used to determine level of effort required to put into SPE activities during the project. Used effort defines the costs. For a low-risk project the effort might be 1% of the budged, but for high-risk project it might be up to 10% of the total budget.

3. **Track SPE costs and benefits.**

The costs and benefits of applying SPE should be documented carefully. This is important to justify need of SPE efforts in the future, because successfully applied SPE is often invisible. Some managers have asked: “Why do we have performance engineers if we don’t have performance problems? “ With good documentation, the return of investment (ROI) can be calculated for applied SPE efforts.

4. **Integrate SPE into your software development process and project schedule.**

To be successful, SPE should be integrated into the development process. This helps to avoid relying too much on certain individuals, because those individuals may be unavailable for the task or leave the company. Additionally, this helps to ensure that SPE goals are kept in mind and not neglected due to tight schedule.

5. **Establish precise, quantitative performance objectives and hold developers and managers accountable for meeting them.**

When SPE is integrated into the process, the next practice is to define precise and quantitative performance objectives. Well-defined performance objectives can be compared against performance modeling results throughout the development process to determine whether the project has a risk not to meet the performance goals. Identification of performance issues throughout the process aids to perform appropriate remedial actions early.

“The end-to-end time for completion of a ‘typical’ correct ATM withdrawal performance scenario must be less than 1 minute, and a screen result must be presented to the user within

1 second of the user's input." is considered as a well-defined performance objective. On the other hand, "The system must be efficient" is not, because it is too imprecise. Some systems may additionally have multiple different performance objectives based on system load. For example, response time objective for 1000 users might be 1 second, but 2 seconds for 5000 users.

6. **Identify critical use cases and focus on the scenarios that are important to performance.**

Use cases describe systems' behavior when an actor (e.g. end-user) uses the system and the system performs the associated action. A use case is a critical use case if the failing of performance goals makes the system unusable or less usable from actor's point of view. Every complex system contains several different uses cases, but only small subset of these is critical from performance point of view. "A small subset of the use cases (<20%) accounts for most of the uses (>80%) of the system. There performance is dominated by these heavily used functions". It is impracticable and costly to focus on all possible uses cases. Performance analysis should focus on uses cases that are executed frequently, are critical from user's point of view, or those that are executed infrequently, but whose performance is particularly critical when executed.

7. **Perform an architecture assessment to ensure that the software architecture will support performance objectives.**

One of the earliest decisions made in a software development project is the architecture design. Performance cannot be added into the architecture afterwards. Because architectural designs are the most costly to fix afterwards, performance must be taken into the design from the very beginning. Smith & Williams add that based on their experience "performance problems are most often due to inappropriate architectural choices rather than inefficient coding. By the time the architecture is fixed, it may be too late to achieve adequate performance by tuning." Additionally, well-defined architecture cannot guarantee meeting performance goals, but a bad architecture can prevent that.

8. **Secure the commitment to SPE at all levels of the organization.**

Software performance is not just a developmental concern; it's a fundamental matter for the entire organization. Software developers are usually aware of the performance and anxious to fix it. SPE commitment issues usually arise from the middle management, because they constantly trying to satisfy many conflicting goals. Keeping project in schedule and costs under control are just a few to mention. Successful adoption of SPE requires full commitment from the middle management as well as other company personnel.

9. Establish an SPE center of excellence to work with performance engineers on project teams.

SPE described by Smith and Williams relies heavily on performance modeling. At early phases of a project, the developers can utilize basic SPE techniques to build simple performance models to assist architectural and design decisions. Later on, performance models become more complex and more detailed. Thus, advanced SPE skills are required to conduct studies that are more detailed. For this purpose, the company should nominate personnel from the development organization to be responsible for performance management. Performance managers should have enough authority in the organization to demand changes when they are needed. Additionally, they are responsible for:

- Tracking and communicating performance issues.
- Establishing SPE process for identifying and solving issues that prevent accomplishment of performance goals.
- Building performance expertise and assist other developers with SPE related issues.
- Create a risk management plan.
- Ensure that SPE activities are properly accomplished.

10. Ensure that developers and performance specialists have SPE education, training and tools.

SPE consists of wide set of methods, each suitable for different project and development phase. Additionally, performance engineering requires various modeling and measurement tools. Successful SPE adoption requires proper expertise to know when and how to utilize the methods and tools. This helps to build confidence on SPE among developers.

11. Require contractors to use SPE on your products.

Use of SPE should also be enforced if the company utilizes external contractors in developing their products to avoid unpleasant surprises. These same practices apply when using a contractor.

4.6.2 Best practices in performance modeling (12-16)

As mentioned earlier, SPE utilizes performance modeling techniques, for instance, to model the software architecture and design. There are total of five best practices for performance modeling (Smith & Williams, 2003):

12. Use performance models to evaluate architecture and design alternatives before committing to code.

13. Start with the simplest model that identifies problems with the system architecture, design, or implementation plans then add details as your knowledge of the software increases.

One challenging part of software development is balancing between quality attributes like performance, availability and security. Often trade-offs are made in the development when such attributes conflict with each other. Situations like this should be solved early in the development process to avoid refactoring when the code is already written. Simple performance models are cheap to build and evaluate before the software is implemented. Evaluating performance issues early allows changes to architecture and design alternatives without complex and expensive refactoring. Later in the development, as more details are known, those simple performance models can be expanded to evaluate more complex performance data.

14. Use best- and worst-case estimates of resource requirements to establish bounds on expected performance and manage uncertain in estimates.

Software architecture, design and other details of the software are still vague. Hence, it is impossible precisely estimate system resource requirements for the software execution. SPE performance models rely upon such estimates. Therefore, SPE uses various strategies, for example, best- and worst-case strategy to produce predictions of the best-case and worst-case performance thresholds. If thresholds are unsatisfactory, a new alternative is required. On the other hand, if results are satisfactory, software development can continue with confidence. If the thresholds are in between, models can be used to identify critical components, and analyses can focus on those.

15. **Establish a configuration management plan for creating baseline performance models and keeping them synchronized with changes to the software.**

Software design evolves during software development. SPE artifacts, such as performance scenarios and models, evolve along with the software. Changes to SPE artifacts should be managed similarly to changes to software design and code. The software configuration management ensures that baseline SPE artifacts evolve with the software. The configuration plan specifies identify rules for artifacts, baseline establishment criteria for an artifact and instructions what to do when making a change. It is not mandatory for every system, but it is highly recommended for safety-critical systems.

16. **Use performance measurements to gather data for constructing SPE models and validating their results.**

Performance modeling at early stages of design and development usually produces estimates. As soon as parts of the system are implemented they can be measured to resolve actual resource usage of key components. Measurements substantiate model predictions. Additionally, measurements may highlight details that were originally omitted from the model because they were thought to be negligible. Those may have critical impact on performance. Therefore, critical components should be measured as early as possible throughout the development process.

4.6.3 Best practices in performance measurement (17-19)

The previous chapter concluded that performance measurement should be used to substantiate performance model predictions. The best practices for conducting performance measurements and performance testing include the following three instructions (Smith &

Williams, 2003):

17. Plan measurement experiments to ensure that results are both representative and reproducible.

Performance measurements should be representative and reproducible. Representative performance measurement results precisely reflect all of the factors affecting software performance: the workload, the software itself and its execution environment. Perfect measurements are very hard to conduct and they require a lot of effort to design and execute. Therefore, unimportant details should be omitted to reduce design effort and execution overhead. Reproducibility means that when measurements are repeated the results are the same or very similar. Reproducibility increases confidence in the results.

The traditional scientific method can be applied to performance measurements. The following steps are vital to success:

- Select appropriate test cases that accurately represent the factors affecting software performance. Build a test plan and prioritize tests to be run.
- Collect only the necessary data. Too much data may ruin the result because it may conceal essential results. Too little data, on the other hand, may render measurement inaccurate or even useless.
- Collect and coordinate measurement results carefully. Synchronize data collection and the granularity of the results, and follow the prioritized test plan.

18. Instrument software to facilitate SPE data collection.

Standard SPE measurement tools can be supplemented by adding instrumentation probes at key points of the code. Firstly, well-done instrumentation of the code generated detailed traces with all events of interest, such as event sequence and frequency. This helps to collect and report exactly the data required. Secondly, instrumentation can be used to measure precise measurements. Results from standard measurement tools rarely meet the desired data granularity requirements. Lastly, instrumentation allows full control on measurement process. Performance measurement with standard tools is not just a one click task. It requires several execution steps and experienced personnel to analyze the results. Instrumented code enables to turn on and off measurement of selected features as needed.

18. Measure Critical components early and often to validate models and verify their predictions.

As previously stated, SPE should be considered early in the development process. Similarly, performance measurement should be started on critical components as soon as possible and continued throughout the process. This ensures that performance models are up-to-date and changes do not void the models.

4.6.4 Best practice techniques for SPE (20-24)

The last five best practices are techniques for co-operating SPE are (Smith & Williams, 2003):

20. Quantify the benefits of tuning versus refactoring the architecture design

Performance improvements can be obtained by fine tuning an existing system. Those are unlikely to be as good as of a well-designed system. Additionally, fine tuning results are often mistakenly believed as SPE accomplishments, which they are not. This failure in thinking can be avoided by comparing what was achieved with tuning to what could have accomplished by using SPE.

21. Produce timely results for performance studies.

SPE results should be composed and presented as soon as possible. Software development is fast-paced and key architectural and design decisions are done in timely fashion. If SPE results are not available when they are needed, the key decisions are likely already made.

22. Produce credible model results and explain them.

It is essential that developers and other members of the project team have confidence in performance models and the performance engineer's skills in using them. Confidence in performance models can be enhanced by explaining the models:

- How they represent the software execution?

- How the model results are interpreted?
- How the early performance models are capable of predicting software performance and identify problems?

23. Produce quantitative data for thorough evaluation of alternatives.

Whenever a problem is detected it should be presented with alternatives for solving it. When the project knows the actual costs and benefits of alternative solutions they can make the best choice among those.

24. Secure cooperation and work to achieve performance goals.

"The purpose of SPE is not to solve models, to point out flaws in either designs or models, or to make predictions-it is to make sure that performance requirements are correctly specified and that they are achieved in the final product." Therefore, everyone accompanying the project should have a common goal of developing a product that meets desired quality constraints.

5 SPECIFICATIONS FOR THE UPDATED SOFTWARE DEVELOPMENT PROCESS

This chapter incorporates SPE techniques, presented in the previous chapter, into the development process presented in chapter 3, which is used to develop the Configuration Management and Mobile Device Management systems presented in chapter 2.

5.1 Requirements for the specification

Current development process is built on agile practices. Developers and development teams work independently to design, implement and test user stories in order of priority. This should be taken into account when making changes to the process. Furthermore, the code base has been under development for a decade. Products contain many features some of which have not been changed for years. On the other hand, new features are being implemented on daily basis. The updated process should provide guidelines how to work with old features and what to do when implementing new ones.

The company wishes to receive various benefits from the updated process. Firstly, target is to reduce costs. Performance issues should be detected early in the making because it is cheaper to remove defects earlier than later in the software lifecycle. Secondly, performance measurements are needed in order to estimate effects of change in performance and verity that application's performance is within the established performance goals. This implies that performance requirements must be written down. Thirdly, it should be possible to conduct performance measures in production environment also without noticeable effect on daily usage. Lastly, Performance analysis should be able to help to create a baseline for up-to-date system requirements.

5.2 Updated software development process

Presented in figure 18 is the software development process model from chapter 3. It highlights three sections that are important from SPE point of view. Each incorporates different SPE practices to that phase of the process. These are:

1. **Product management**
2. **Agile implementation**
3. **Verification and validation**

As can be seen from the figure, product management is present throughout the process to support and supervise other activities. Similarly, verification starts already during user story planning and continues until new version is released. The following subchapters discuss these in more detail.

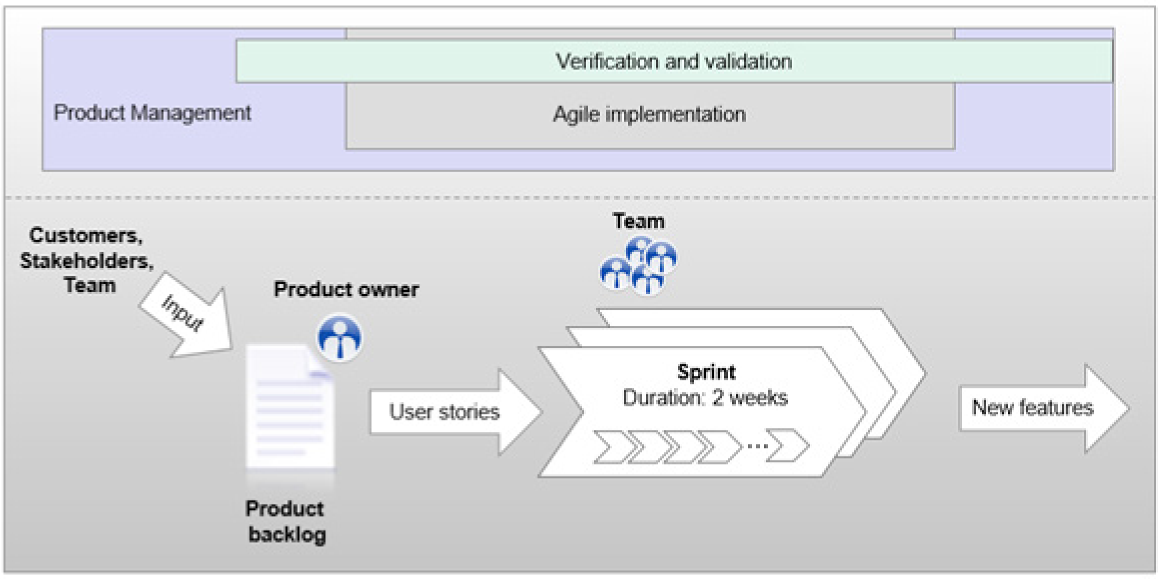


Figure 18: Updated software development process model from chapter 3

5.2.1 Product management

The product management in the company is responsible for the product strategy, roadmap and feature specifications together with sales staff, customers, stakeholders and developers. They define in which direction the product development will need to take in order to satisfy customer needs, including finding new uses for the product or creating new features to increase profits or sales. They write new user stories to the product backlog and maintain priorities of user stories on the backlog. If a user story does not contain sufficient information about the use case, they acquire required information from corresponding personnel.

From SPE point of view, as mentioned in chapter 4.6, product (project) management is mainly responsible for supervising that software performance concerns are taken into account throughout the software development lifecycle. They drive the development by securing commitment to the subject at all levels of the organization, and ensure that employees have sufficient training and tools.

Additionally, product management is responsible to identify critical user stories (scenarios) that are important from the performance point of view. This is the first concrete new task for them. It is accomplished by evaluating whether a user story involves a performance risk. If the user story involves a performance risk performance objectives, target workload and budget must be presented. If there is no risks, definition of target workloads and performance objectives are not needed. Each user story on the backlog should have the following new attributes defined:

- **Performance risk: Yes / No**
- **Performance objectives**
- **Target workload**
- **Budged**

These attributes are later in the process used to determine necessity to carry out subsequent SPE activities. Table 4 lists example attributes for performance critical components presented in chapter 2. Further examples of budgets, workload requirements and performance objectives are presented in chapter 4.2.

Table 4: Performance attributes for user stories

	ASP.NET client handler	**ASP.NET GUI**	**Inventory data import**	**Daemons/ Services**
Performance objectives	Response time	Response time	Throughput	Throughput
Workload	Clients	Users	Clients	Managed devices Users
Budged	Resource utilization Execution time	Resource utilization Execution time	Resource utilization Execution time	Resource utilization Execution time

A common thing for the products is that there are many similar events occurring (e.g., the client sends a message to server or a user opens an ASP.NET web page). Thus, common system-wide performance attributes are needed to simplify this phase. A common performance objective could be *"A web page should be displayed in less than 1 second."* This objective is suitable for a large portion of pages. However, for a complex page, product management could write an exception stating that the response time requirement should be increased to 2 seconds.

5.2.2 Agile implementation

As aforementioned, development teams work independently and iteratively to implement the user stories in a sprint. Figure 19 presents steps through which each user story goes. In addition, under each step the figure contains new SPE related activities. Complexity of these steps varies depending on whether user story is a small improvement or a more complex feature.

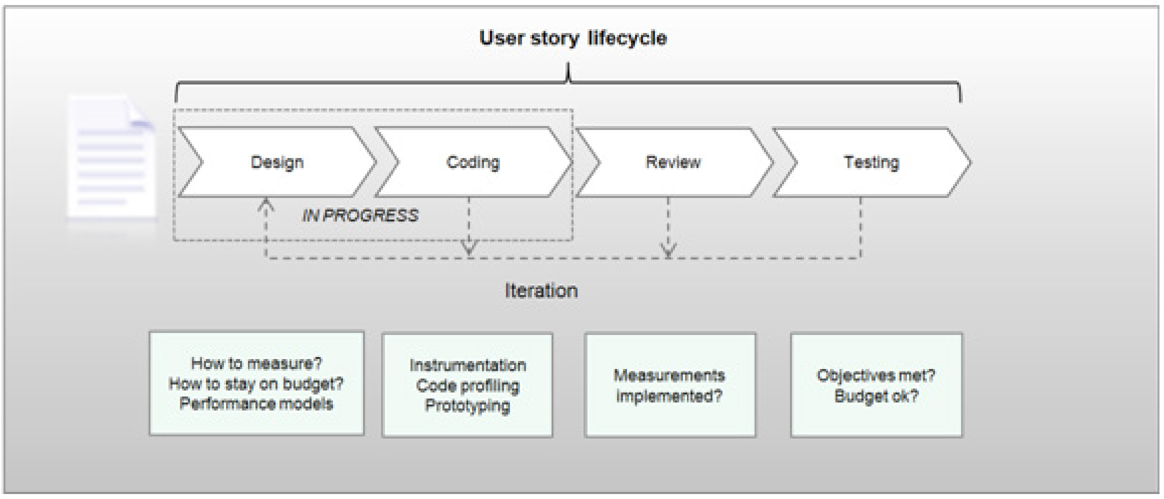


Figure 19: User story lifecycle in a sprint

Implementation starts with a design phase. Designs are usually drawn on a piece of paper or on a blackboard. In some cases, prototyping is also used. However, use of software modeling notations, such as UML, is not a common practice. Therefore, design and construction of performance models can be left out until use of software modeling notations becomes common practice during development. At that point in time, software models should be annotated with quantitative information and analyzed from performance point of view.

If a user story involves a performance risk, developers must design how to make measures and how to stay on given budget. At any given time, if performance objectives or budged requirements turn out to be unachievable they need to be revised. Performance objectives define which performance indices to measure. Development team's goal is to design the measures (e.g. code instrumentation points) that can be used to verify that the feature meets its objectives. Similarly, staying on budget must be measurable (e.g., how much memory is used).

There are three important tasks, which need to be performed during the coding phase. Firstly, all new code should be well covered by adequate instrumentation points. Secondly, performance-profiling tools (e.g. ANTS Performance Profiler[1] or Visual Studio Profiling Tools[2]) should be used during development. These tools are useful to analyze performance issues and other unusual behavior. Lastly, prototyping should be continued throughout this phase for the understanding potential architectural advantages and disadvantages. While reviewing the code, peer reviewers check that source code is properly instrumented. Ultimately, before the user story is marked as complete, it has to be validated against its performance objectives and given budged.

5.2.3 Verification and validation

As aforementioned, verification and validation starts already during product management phase when user stories are verified to contain all required attributes. Verification is a part of the backlog grooming sessions. During development phase, development teams test individual user stories before marking them as complete. Practically, each new product release consists of several stories, usually something between 5 to 50 different user stories. New features may or may not have an effect on existing features, and therefore, integration and release tests have to be executed before release.

Performance tests should be included as part of this phase immediately after common performance objectives and budget requirements are defined and the first version of performance measurement framework, presented later on this chapter, is implemented and first instrumentation points have been added to the code base.

[1] http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/
[2] https://msdn.microsoft.com/en-us/library/z9z62c29.aspx

In order to be adequate, the test framework must address the following concerns. Firstly, it should be able to repeat a particular sequence of actions. For example, open all web forms one by one, measure response times, and compare values against previously collected data. Secondly, it should be possible to simulate specific load conditions (e.g. 5000 managed devices / 100 users versus 20000 managed devices / 500 users) over and over again, making it possible to collect performance measures over a time frame (e.g. 1 hour) and compare collected data against previous results. Such simulation is not feasible with genuine devices because of mere numbers, and therefore, requires some kind of load simulation utilities to be used.

5.3 Performance Measurement Framework

This chapter discusses the specification for the Performance Measurement Framework (PMF). The PMF consists of several components that can be used in conjunction to analyze software system's performance in internal test environments as well as customer's production environments. The main goals for the PMF are:

1. **Collect performance indices, such as response times, processing times and throughput out from performance critical features presented in chapter 2.2.**
2. **Collect hardware resource usage out from web and SQL servers.**
3. **Store collected data and make it possible to link the data together making it possible to analyze what is actually going on in the system.**

5.3.1 The big picture

The concept behind the Performance Measurement Framework is depicted in figure 20. In order to meet the goals, the framework needs ability to collect performance data out from the following components:

1. ASP.NET IIS applications
2. C# services
3. Scheduled background daemons
4. Server hardware resources

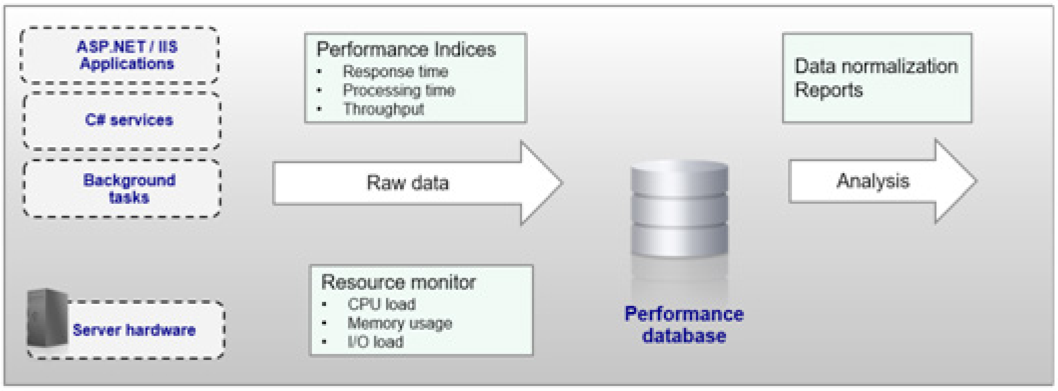


Figure 20: The concept behind the Performance Measurement Framework

The impact to system's performance while collecting performance data must be kept constantly in mind. The overhead caused by data collection should have negligible impact on the monitored systems performance. To achieve this data is collected and stored as is, and all further data processing should be done later on as offline analysis. In order to minimize the overhead caused by the data collection, it should be possible to enable it by feature basis. For example, if it looks like inventory queues are constantly full; the PMF could be enabled for inventory imports only. Likewise, if one is interested in client-server communication, one can enable performance measurement for it.

5.3.2 Instrumentation techniques for ASP.NET

This chapter specifies software-monitoring techniques for collecting performance indices out from the ASP.NET applications. By doing this, the PMF is able to cover the following performance critical features:

1. Client-server communication
2. User interface
3. Inventory data handlers
4. Web service and connector interfaces

- **Performance index:** response time
- **Instrumentation technique**

How long it takes to process a request by the ASP.NET *HttpApplicaton* pipeline? This can be done by measuring spent time between *BeginRequest* and *EndRequest* events (Microsoft, 2016a). Additional instrumentation is required when tracking requests through different layers of the web application. This can be achieved by adding more detailed custom instrumentation points to critical parts of the code. Additional instrumentation points can be linked to a specific request by storing a unique request identifier to ASP.NET *HttpContext*.

- **Request identification**

Each HTTP request can be identified by the web page part of the URL.

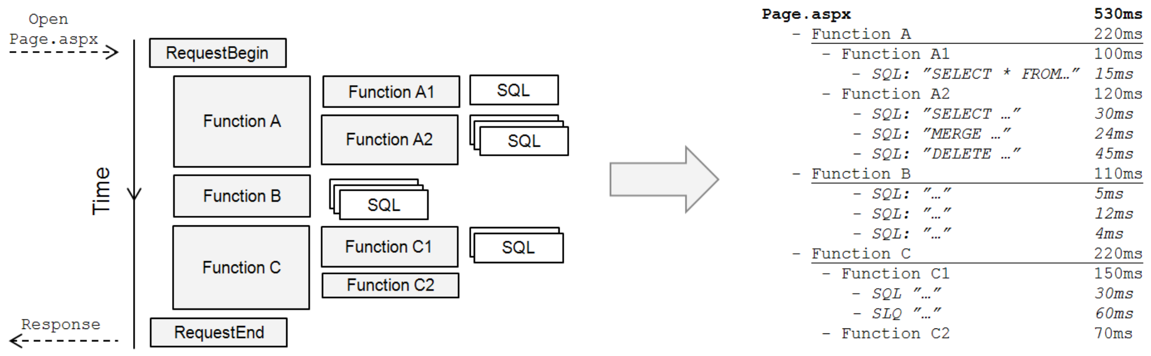
For example:

- http://www.site.com/views/<u>device_list.aspx</u>
- http://www.site.com/handlers/<u>client.ashx</u>

A single platform specific ASP.NET handler processes most of the client-server communication. Different messages are defined either XML elements or XML attributes defined in the XML elements. Therefore, in order to message level identification requires XML parsing.

- **An example of collected data**

Presented in figure 21 is an example of an instrumented ASP.NET application. Instrumentation starts from *RequestBegin* event and ends after *RequestEnd* event. During the request, seven instrumentation points were bypassed and nine SQL queries were executed altogether.



```
Page.aspx                          530ms
  - Function A                     220ms
    - Function A1                  100ms
      - SQL: "SELECT * FROM…"      15ms
    - Function A2                  120ms
      - SQL: "SELECT …"            30ms
      - SQL: "MERGE …"             24ms
      - SQL: "DELETE …"            45ms
  - Function B                     110ms
      - SQL: "…"                   5ms
      - SQL: "…"                   12ms
      - SQL: "…"                   4ms
  - Function C                     220ms
    - Function C1                  150ms
      - SQL "…"                    30ms
      - SLQ "…"                    60ms
    - Function C2                  70ms
```

Figure 21: An example of an instrumented ASP.NET HTTP request

5.3.3 Instrumentation techniques for C# Windows Services

This chapter specifies software-monitoring techniques for collecting performance indices out from Windows services implemented using C#. This covers some of the inventory data import types and other service based queues.

- **Performance indices:** throughput and execution time
- **Instrumentation technique**

Services expose interfaces for other applications to use. For example, ASP.NET inventory data handlers send inventory data to inventory service's interface. Service application can be instrumented by measuring how long it takes to process a service call. When needed, more specific instrumentation points can be added for subsequent function calls. Additional instrumentation points can be linked to a specific task by storing a unique request identifier to the thread domain.

- **Identification**

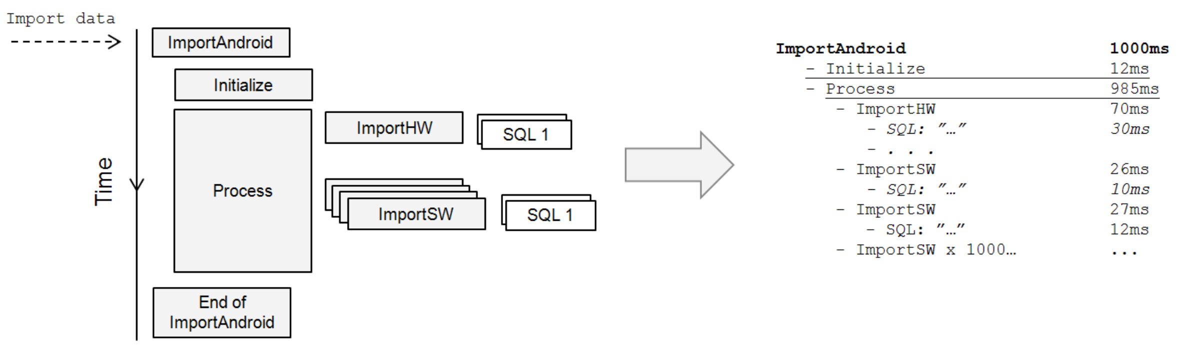
Each service call can be identified by the name of the interface method.

For example:

- public void ImportAndroidInventory
- public void ImportiOSInventory

- **An example of collected data**

Figure 22 presents an example of collected data from an instrumented service call. In the example, interface method is known as *ImportAndroid* and it consists of two first level children function calls and multiple second level ones. It should be noted that inventory data might contain thousands of installed software entries. Additionally, the PMF should keep track about number of jobs in each queue. This information can be used to calculate how many files are being processed within a period.



```
ImportAndroid                        1000ms
  - Initialize                       12ms
  - Process                          985ms
     - ImportHW                      70ms
        - SQL: "…"                   30ms
        - . . .
     - ImportSW                      26ms
        - SQL: "…"                   10ms
     - ImportSW                      27ms
        - SQL: "…"                   12ms
     - ImportSW x 1000…              ...
```

Figure 22: An example of an instrumented service call

5.3.4 Instrumentation techniques for inventory import scripts

This chapter specifies software-monitoring techniques for collecting performance indices out from the inventory import scripts. This covers some of the inventory import types.

- **Performance indices:** throughput and execution time

- **Instrumentation technique**

Inventory import scripts process inventory folders one by one. If folder contains a file, it is read to memory and written to the database using SQL Server stored procedure. Instrumentation can be done by measuring how long it takes to read a file from the disk and how long it takes to execute the stored procedure.

- **Identification**

Inventory imports can be identified by name of the inventory folder and name of the stored procedure.

For example:

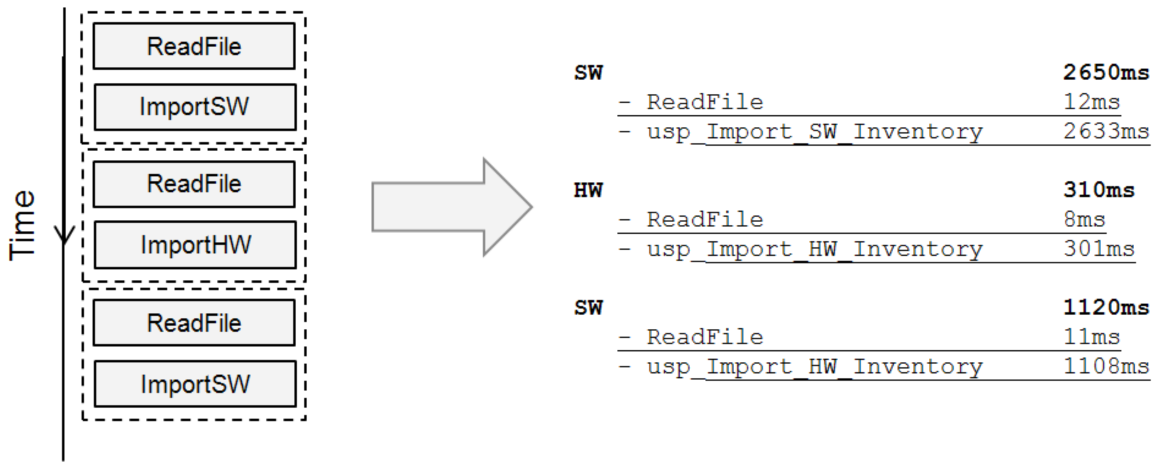
  - FileScan & dbo.usp_import_filescan

  - HWScan & dbo.usp_import_hwscan

- **An example of collected data**

As depicted in figure 23, output generated by an instrumented inventory import script is rather straightforward containing only the read and write operations as described earlier.

Additionally, PMF should keep track about number of files in each inventory import folder. This information can be used to calculate how many inventory files are being processed within a period.



```
SW                                       2650ms
   - ReadFile                            12ms
   - usp_Import_SW_Inventory             2633ms

HW                                       310ms
   - ReadFile                            8ms
   - usp_Import_HW_Inventory             301ms

SW                                       1120ms
   - ReadFile                            11ms
   - usp_Import_HW_Inventory             1108ms
```

Figure 23: An example of an instrumented inventory import script

5.3.5 Instrumentation techniques for background daemons

This chapter specifies software-monitoring techniques for collecting performance indices out from the background daemons.

- **Performance index:** execution time
- **Instrumentation technique**

Background tasks are run by a custom C# executable known as scheduled task engine. Instrumentation of background tasks, such as, SQL Server stored procedures on other scripts can be done simply by measuring how long it takes to execute. Instrumentation code can be added to the scheduled task engine.

- **Identification**

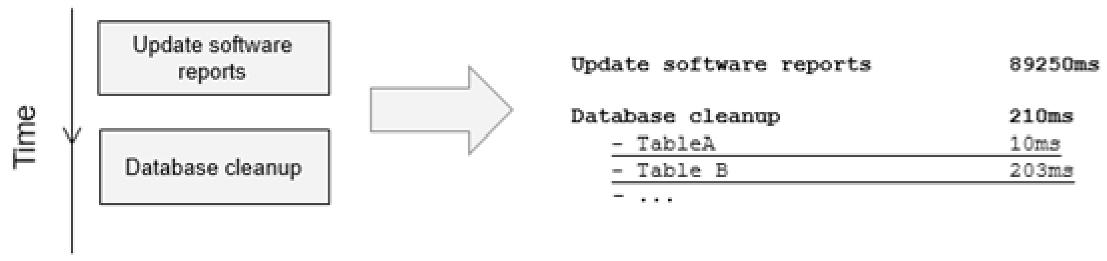
Background tasks can be identified by their unique name.

For example:

- Update software reports and database cleanup

- **An example of collected data**

As depicted in figure 24, output of instrumented background tasks is very straightforward.



```
Update software reports          89250ms

Database cleanup                 210ms
   - TableA                      10ms
   - Table B                     203ms
   - ...
```

5.3.6 Resource monitor

PMF's software monitoring features should be adequate for software behavior and problem analysis. However, sometimes the code may work as expected, but there are hardware bottlenecks that cause the problems. Underpowered hardware is known issue in multiple several customer cases. To detect such problems, the PMF must collect hardware resource usage from the web and SQL servers. Collected resource usage data should be timed so that it can be linked to specific actions in time. For example, the user interface slows down at Monday 8AM. How high were CPU load and memory usage, and which HTTP requests and background tasks were running at that time?

The PMF should collect data out from the following hardware resources:

1. **Overall CPU usage (%)**
2. **Overall memory usage (MB and %)**
3. **Disk activity (Read/Write B/sec)**
4. **Per process resource usage**
    - Processes: IIS process (w3wp.exe), SQL Server process (sqlservr.exe), custom service executables
    - Collected data: CPU usage (%), Memory usage (MB and %) and Disk activity (Read/Write B/sec)

5.3.7 Performance measurement database

The PMF should store collected performance data into a centralized performance measurement database. Later on, performance analysts can write queries against the database in order to generate reports and figures about system state at any given time. The products under analysis already utilize SQL Server database as temporary and long-term data storage. There are existing utilities that can be used to write data to SQL Server database. Thus, use of SQL Server database is obvious choice. Writing lots of data to the database might have an impact on database server performance, and therefore, performance database should be configurable on different physical server than monitored system's database is running on.

5.3.8 Utilization in production environment

Performance problems may occur in customer's production environment. Resolving such issues is very time consuming and expensive. The PMF should be easy to enable when such problems arise. It can collect valuable data out from the system that is not possible currently. In in-house test environment, it is straightforward task to configure a custom test server to be used as SQL Server performance database. In customer's production environment, it is not that easy. Hence, use of SQL Server as performance database is not preferable option in production environment. The solution is to write performance data into a local file. Instead of using centralized SQL Server database, different components generate local comma separated values files (CSV). These files can then be compressed and sent the product support for further analysis.

5.3.9 Common libraries

Target products consist of many similar components, for example, ASP.NET C# web applications and C# services. It is unreasonable to copy the same source code to multiple places. The PMF should define common language-based libraries containing the core instrumentation utilities are available for different application types:

- Common C# profiling library
- Common scripting functions for profiling

5.4 The deployment steps for the updated process

This chapter discusses the phasing and prioritization of previously introduced additions to the software development process. Applying all changes at once is not feasible due to various reasons.

Firstly, it will take quite a while to deploy new practices and implement the framework. Such a project is risky and not very agile. Secondly, making all changes at once may slow down or halt other development activities. Similar problems have been identified in the past while implementing major projects or introducing new development methods as discussed in chapter 3.

Lastly, feedback from the process and other development personnel is valuable in order to improve the process iteratively. Reacting to the feedback is easier when changes are introduced gradually. The steps and the order in which these steps are to be implemented are described in the following sub-chapters.

5.4.1 Step 1: Product and backlog management

The first step is to put the backlog and product management in order. Identification of performance risks (per user story), and definition of performance objectives, target workload and budged is mandatory for subsequent steps. JIRA[3] the software development tool used by the company should be updated to have the new attributes for user stories. Henceforth, each new user story needs to be evaluated from performance point of view before it can be selected for development.

In addition to that, developers should get necessary training on the topic before actual implementation. A performance-profiling tool should be selected and started to use as daily basis during development. Commitment to the topic should be assured during backlog grooming, sprint planning and daily team meetings.

[3] https://www.atlassian.com/software/jira

5.4.2 Step 2: Performance Measurement Framework for ASP.NET

The next step is the implementation of the Performance Measurement Framework for ASP. NET server applications. This chapter will not go into implementation details such as whether to use already existing components or build a custom solution from scratch. Nevertheless, ready-made solutions should be considered if a suitable can be found and it can be fitted to this purpose.

For example, MiniProfiler (MiniProfiler, 2016) is an easy to use profiling utility for ASP. NET. It supports HTTP request-based tracing, SQL query profiling and a pragmatic step instrumentation that can be used to add custom instrumentation points to the code. It provides a simple client-side interface for inspecting collected traces and built-in support for storing them into SQL Server database

After data collection technique is selected and implemented, instrumentation points should be added to the source code. During the first phase, it is enough to measure duration of the HTTP request and trace corresponding SQL queries. This provides adequate information to verify that the PMF for ASP.NET works as expected. More detailed instrumentation points can be added later if needed to the critical parts of the code. Collected data needs to compliant with the specifications presented in chapter 5.3.2.

5.4.3 Step 3: Implement Resource Monitor

The third step is to implement the Resource Monitor. As mentioned in the previous chapter, possibility to use ready-made solutions should be carefully investigated. For example, Windows Performance Monitor (Microsoft, 2016c) can be used to log Windows performance counter data for later analysis. Windows performance counters can provide all aforementioned hardware resource usage.

If Windows Performance Monitor is not suitable for this purpose, it is possible to utilize *System.Diagnostics.PerformanceCounter* class (Microsoft, 2016b) that can be used to read Windows performance counters programmatically from C# code. Additionally, performance counter data can be collected remotely meaning that one custom-made Performance Monitor application is capable of collecting data from multiple servers.

5.4.4 The future steps

The renewal of the software development process is an iterative process. To carry out the first three steps already takes time and provides valuable feedback to the process. Some additions might have to be made and these should be carried out at immediately before proceeding to the next steps. Thus, it is not feasible to make detailed plans for subsequent steps.

The remaining steps in order of importance are:

1. Enable performance monitoring in production environment.
2. Add support for remaining performance critical features to the PMF.
3. Add more instrumentation points to ASP.NET code.
4. Automate performance tests with load simulation framework.

6 CONCLUSIONS AND FUTURE WORK

Software performance analysis is essential part of software development to any software company. Software performance is present, as described in this paper, throughout the software lifecycle: from requirement analysis to design and development to testing to software maintenance. Software performance issues usually stem from early architectural and design choices and have severe impact on customer experience and success of the business. Fixing performance related issues late in the lifecycle is usually time consuming and expensive. Still, software performance issues are rarely considered early in the development.

Presented in this paper are solutions with witch a software company can take performance into consideration during the software development. Software performance engineering, a systematic software oriented engineering approach to develop software that meets its performance objectives, provides methods for different stages of the development process. Each of which is valuable by itself, but when used in conjunction it can substantially improve quality and cost-efficiency of the software.

The methods presented allow valuable enhancements to software development processes, but not all of them are suitable for every software company. Selection of methods must be made based on company's current model of operation. Introducing too many additions at once not only requires significant changes to the current way of doing things but may also nullify the benefits from using the SPE in the first place. Therefore, it is important that companies determine their individual level of effort required to put into SPE activities during the projects. This paper proposed one real-life example of a development process with carefully chosen SPE enhancements. The result is an updated end-to-end process model that is agile, obeys current model of operation and is relatively light-weight to implement. It will be put to use in the near future.

Doing this thesis aroused discussions about non-functional software requirements in general, of which software performance is only one. There are other important non-functional requirements, such as security, reliability and usability. For example, security of a software system is heavily dependent on early design choices. The approach presented in this paper could therefore also be applied with some modifications to other non-functional software attributes.

REFERENCES

Aguilera, M. K. et al., 2003. *Performance debugging for distributed systems of black boxes*. New York, ACM.

Balsamo, S., Di Marco, A., Inverardi, P. & Simeoni, M., 2002. *Software performance: State of the art and perspectives,* s.l.: Dipartimento di Informatica, Universita dell' Aquila.

Balsamo, S., Di Marco, A., Paola, I. & Marta, S., 2004. *Model-Based Performance Prediction in Software Development: A Survey*. s.l., IEEE.

Barham, P., Donnelly, A., Isaacs, R. & Mortier, R., 2004. *Using Magie for request extraction and workload modeling.* s.l., OSDI '04: 6th Symposium on Operating Systems Design and Implementation.

Beck, K. et al., 2001. *Manifesto for Agile Software Development.* [Online] Available at: http://agilemanifesto.org [Accessed 10 7 2014].

Bhatia, S., Kumar, A., Fiuczynski, M. E. & Peterson, L., 2008. *Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems*. San Diego, ACM Press.

Chanda, A., Cox, A. L. & Zwaenepoel, W., 2007. Whodunit: *Transactional Profiling for Multi-Tier Applications.* Lisboa, ACM Press.

Compuware, 2006. *Application Performance Management Survey,* s.l.: s.n.

Cortellessa, V., Di Marco, A. & Inverardi, P., 2011. *Model-Based Software Performance Analysis.* s.l., Springer Berlin Heidelberg.

Fortier, P. & Michel, H., 2003. *Computer Systems Performance Evaluation and Prediction.* s.l.:Elsevier.

Laurie, W. & Alistair, C., 2003. Agile software development: it's about feedback and change. In: *Computer.* s.l.:IEEE, pp. 39 - 43.

Microsoft Corporation, 2004. *Improving .Net Application Performance and Scalability (Patterns & Practices).* s.l.:s.n.

Microsoft, 2016a. *ASP.NET Application Life Cycle Overview for IIS 7.0.* [Online] Available at: https://msdn.microsoft.com/en-us/library/bb470252.aspx [Accessed 14 February 2016].

Microsoft, 2016b. *MSDN: PerformanceCounter Class (System.Diagnostics).* [Online] Available at: https://msdn.microsoft.com/en-us/library/system.diagnostics.performance-counter(v=vs.110).aspx [Accessed 26 2 2016].

Microsoft, 2016c. *Windows Performance Monitor.* [Online] Available at: https://technet.microsoft.com/en-us/library/cc749249.aspx [Accessed 26 2 2016].

MiniProfiler, 2016. *MiniProfiler: A Simple buteffective mini-profiler for .NET and Ruby..* [Online] Available at: http://miniprofiler.com [Accessed 26 2 2016].

Mosberger, D. & Jin, T., 1998. *httperf—a tool for measuring web server performance.* New York, ACM SIGMETRICS Performance Evaluation Review.

Mylopoulos, J., Chung, L. & Nixon, B., 1992. *IEE Transactions on Software Engineering,* Issue Volume: 18, Issue: 6, pp. 183-497.

Nixon, B. A., 2000. Management of Performance Requirements for Information Systems. *IEEE Transactions of Software Engineering,* Issue Volume:26, Issue: 12, pp. 1122-1146.

OMG, 2005. *UML Profile For Schedulability, Performance and Time.* [Online] Available at: http://doc.omg.org/formal/2005-01-02.pdf [Accessed 13 10 2015].

OMG, 2008. *Software & Systems Process Engineering Metamodel (SPEM).* [Online] Available at: http://www.omg.org/spec/SPEM/ [Accessed 31 January 2015].

Petriu, D. & Woodside, M., 2002. *Analysing Software Requirements Specifications for Performance.* Rome, s.n.

Reynolds, P. et al., 2006a. *Pip: Detecting the Unexpected in Distributed Systems.* San Jose, NSDI.

Reynolds, P. et al., 2006b. *WAP5: Black-box Performance Debugging for Wide-Area Systems.* Edinburg, Proceedings of the 15th international conference on World Wide Web.

Sigelman, B. H. et al., 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,* s.l.: Google, Inc..

Smith, C. U., 2001. *Origins of Software Performance Engineering: Highlights and Outstanding Problems.* s.l., Springer Berlin Heidelberg.

Smith, C. U. & Williams, L. G., 2003. *Best Practices for Software Performance Engineering.* Dallas, TX, Computer Measurement Group.

Tierney, B. et al., 1998. *The NetLogger Methodology for High Performance Distributed Systems Performance Analysis.* Chicago, IL, IEEE.

van der Zee, A., Courbot, A. & Nakajima, T., 2009. *mBrace: Action-based Performance Monitoring of Multi-Tier Web Applications.* Vancouver, Computational Science and Engineering, 2009. CSE '09. International Conference on (Volume:2).

Vetter, J. S. & Worley, P. H., 2002. *Asserting Performance Expectations.* Los Alamitos, IEEE Computer Society Press.

Woodside, M., Franks, C. & Petriu, D. C., 2007. *The Future of Software Performance Engineering.* Minneapolis, MN, IEEE Computer Society, pp. 171 - 187.