Advances in Software Development

Scientific Editor

Jakub Swacha

Conferences organized by
Polish Information Processing Society:

VIII edition of the Congress of Young IT Scientists

**XV edition of the Polish Conference
on Software Engineering**

XX edition of Real Time Systems

were supported
by the **Ministry of Science and Higher Education**
within the program related to the implementation
of tasks of science dissemination
(Decision No 1064/P-DUN/2013 on 24/07/2013)

POLISH INFORMATION PROCESSING SOCIETY

Advances in Software Development

Scientific Editor
Jakub Swacha

Warszawa 2013

Authors

Barbara Begier – ***CHAPTER 1,*** *Walery Susłow, Michał Statkiewicz –* ***CHAPTER 2,*** *Szymon Kijas, Andrzej Zalewski –* ***CHAPTER 3,*** *Jakub Swacha, Karolina Muszyńska, Zygmunt Drążek –* ***CHAPTER 4****, Bartosz Wilk, Marek Kasztelnik, Marian Bubak –* ***CHAPTER 5,*** *Mariusz Jarocki, Agata Półrola, Artur Niewiadomski, Wojciech Penczek, Maciej Szreter –* ***CHAPTER 6,*** *Bogumiła Hnatkowska, Radosław Tumidajewicz –* ***CHAPTER 7,*** *Tomasz Straszak, Michał Śmiałek –* ***CHAPTER 8,*** *Anna Derezińska, Piotr Trzpil –* ***CHAPTER 9,*** *Michał Żebrowski, Andrzej Ratkowski –* ***CHAPTER 10,*** *Patryk Czarnik, Jacek Chrząszcz, Aleksy Schubert –* ***CHAPTER 11,*** *Janusz Zalewski –* ***CHAPTER 12,*** *Marek J.Greniewski –* ***CHAPTER 13***

Reviewers

Wojciech Cellary, Piotr Czapiewski, Zbigniew Czech, Włodzimierz Dąbrowski, Norman Fenton, Tracy Hall, Jason Ho, Zbigniew Huzar, Magne Jørgensen, Stanisław Kozielski, Bev Littlewood, Leszek A. Maciaszek, Lech Madeyski, Andrzej Marciniak, Tomasz Muc, Ngoc Thanh Nguyen, Łukasz Radliński, Martin Shepperd, Andrzej Stasiak, Zdzisław Szyjewski, Lech Tuzinkiewicz, Marek Valenta, Bartosz Walter

Scientific Editor

Jakub Swacha

Technical Editor

Łukasz Radliński

ISBN 978-83-7518-597-3

Edition: I. Copies: 200. Publishing sheets: 9,95. Print sheets: 12,4.
Publisher, print and binding: PPH ZAPOL, al. Piastów 42, 71-062 Szczecin

Contents

Preface

One hundred and seventy years have passed since *Taylor's Scientific Memoirs* published Ada Lovelace's notes, including what is considered to be the first program ever written. These days, when writing computer programs is a developed industry, and software engineer – a renowned profession, a lot of thought and effort is put into finding better ways of gathering software requirements, designing, developing and testing software, as well as managing development teams and teaching software engineering. This monograph presents the recent developments from researchers and practitioners specializing in these areas.

In chapter one, Barbara Begier shows that teaching Scrum methodology at the university encounters difficulties which can be eluded by restraining the teaching only to selected elements of Scrum. She illustrates her proposal with observations from her own work with students.

Chapter two is devoted to prevention of conceptual errors in system design. The authors analyze their experiences with graduate student team projects based on the domain-driven design approach and the Iconix methodology and provide interesting suggestions.

The third chapter presents a model for capturing architectural decisions, tailored for documenting the evolution of service-oriented systems. It not only enables tracking architectural decisions made as the project progresses, but also allows to document changes made to artefacts developed earlier.

Chapter four provides guidelines for the process of adaptation of open-source software in form of an updated process framework for open-source software acquisition and some remarks drawn from its application in two international projects: BalticMuseums 2.0 and BalticMuseums 2.0 Plus.

The fifth chapter presents a feasibility study of applying the Feature Model to develop tools for automatic eScience environment configuration using a prototype implementation. It also describes an architecture of an extensible framework automating various deployment and component installation tasks based on the Feature Model.

Chapter six describes PlanICS 2.0, an automatic web service composition system which separates between an abstract and a concrete planning phase, provides flexibility, and allows to handle services that do not publish their internal semantics.

The seventh chapter presents a new approach to the automatic generation of test cases, in which they are created on the base of business rules expressed in a structured natural language using a tool developed especially for this purpose.

In chapter eight, the concept for the Requirements Driven Software Testing (ReDSeT) tool is described. The tool allows for automatic integrated test generation based on different types of requirements. It also allows to create relations between tests by combining different types of requirements,

Chapter nine examines application of mutation testing to test cases evaluation for ASP. NET MVC-based web applications. Several new specific mutation operators were proposed for this purpose. The presented experimental evaluation results of the proposed approach are generally positive.

The tenth chapter contains an interesting case study of differential evaluation of Service Oriented Architecture implementation in an organization that consist of two separate parts. The authors examine critical success factors of SOA implementations and conclude with some general observations.

In chapter eleven, a design for a formalisation of the whole set of about two hundred Java bytecode instructions is presented. The instructions are grouped based on the way they operate on the runtime structures. The authors aim to create a platform where both real programs could be verified and metatheoretical properties of the language could be shown.

Chapter twelve presents the idea of web-based laboratories for teaching software engineering, as implemented at Florida Gulf Coast University. This topic gives opportunity to discuss issues related to the concepts of Lewis Mumford's megamachine, Marshall McLuhan's medium as a message, and Clayton Christensen's disruptive technology.

The last chapter provides us with the history of development of system modeling approaches, beginning with the concepts of Henryk Greniewski.

Jakub Swacha
Szczecin, September 2013

Chapter 1
Using Scrum or Scrumbut?

Developing the next generation of IT professionals requires paying more attention to modern methodologies of software development and training students in using them. The increasing popularity of Scrum process bears a need to train students in using this agile methodology at the faculty of computing. The aim of this chapter is to show that teaching Scrum at the university encounters difficulties – only some elements of Scrum methodology are easy to be applied by students. The described real life experience refers to students' software projects implemented using Scrum process. Various departures from Scrum, met in practice, are described. Detailed observations and conclusions drawn during project classes and supervision of student's dissertation are presented.

1.1.Introduction

Scrum, a framework of the software process [5, 6], has gained wide recognition and software companies make frequent use of this agile methodology. This trend is also observed in Poznan (Poland) in the last two years. Thus Scrum should be present in an educational process of future programmers and project managers. Ken Schwaber and Jeff Sutherland, the authors of Scrum, are also the co-authors of Agile Manifesto [3]. At the author's faculty, Scrum is initially presented in lectures of software engineering during the first-circle studies. Aiming to master's degree, students are taught *software project management*, referred at first to any IT project [4] and then especially to agile methodologies [2, 5]. A need to balance agility and discipline is also emphasized [1]. In response to students' expectations, the author has decided to supervise project development using Scrum during project classes in the winter semester of the academic year 2012/13. Students willingly started their projects trying to apply Scrum methodology. Unfortunately, this experience turned out to be rather *Scrumbut* [7] in practice than Scrum. Not all principles and recommended practices of Scrum have been applied. Various aspects of Scrum, applied or ignored by students, are described in this chapter.

The list of compatibilities of students' conduct with recommendations given by the Scrum authors has been developed and included into this chapter. Paralelly, the list of divergences from Scrum rules and one more, containing misinterpretations and problems with Scrum, have been developed, too. To work out these lists, the author has made use of an opinion given by the professional certified Scrum Master. The Scrum authors say that *Scrum exists only in its entirety* [8]. It means that all principles, roles, events, and rules should be met and respected. There is no Scrum if only some Scrum ideas are followed. Thus the questions may be: Are the students able to work in Scrum? How to teach and use Scrum methodology in students' projects? The described experience may help answering these questions and open a discussion about how to teach students agile methodologies and how to practice software process using Scrum. Conclusions are given in the last section.

1.2.About Scrum itself

The Scrum framework is briefly described below as a reference from the further parts of this chapter to show many departures from Scrum rules made by students. The content of this section is based on several documents published by the Scrum authors, Ken Schwaber and Jeff Sutherland [5, 6, 8]. They co-presented Scrum first at the OOPSLA conference in 1995 but Scrum has evolved from this time. Scrum framework is known as a tool to manage effectively software projects. It helps achieving two opposite, at the first sight, aims: high quality of a product and at the same time high productivity in software development. How is it possible to combine these usually conflicting elements together?

The answer to this question consists of all elements and rules of a Scrum process. Its crucial elements are: constant cooperation with a customer represented by a key user called Product Owner, iterative-incremental model of software development, relatively short iterations called sprints (1−4 weeks), requirement management using priorities assigned by a Product Owner to each item in the *Product Backlog*, based on these priorities selection of requirements for each sprint (*Sprint Backlog*), frequent issues (*Increments*) making possible a fast feedback from a customer's side, work in small teams (3−9 persons), team's autonomy during a sprint, possibility to stop a sprint or even a whole project any time.

Scrum as a framework of a software process consists of roles, events, artifacts, and rules. The rules of Scrum bind together associated roles, required and then developed artifacts, and events, starting from sprint planning. The Scrum team consists of a Product Owner, development team, and a Scrum Master. Every second student wants to play a role of a Scrum Master although a person playing this role is expected to be a team servant instead of a supervisor of a team. The success in Scrum results from: skills in team work, easy (direct) communication, responsibility of team members, and their mutual trust. It is not evident a priori if all team members share these abilities.

There is an assumption in Scrum process that a team is self-organizing (nobody from outside tells each team member what she/he should do) and cross-functional (each team member shows various skills necessary to create a product increment). Testing is performed alternately by team members (not by external testers). Everyone in team must understand and share the definition of "*Done*" which refers to each item of the Sprint Backlog. In other words, team members know *what it means for work to be complete*. Then a visible effect of synergy seems to be very encouraging to apply this methodology.

As mentioned above, all work in Scrum is divided into sprints (iterations). A sprint is a time-box having its own specified goal. It is *a forecast by the development team about what functionality will be in the next increment and the work needed to deliver that functionality* [8]. Each sprint contains specified development work and several events taking place to meet all requirements, including quality requirements, and this way to satisfy the Product Owner. There are several meetings required in Scrum, including Sprint Planning, Daily Scrum (short stand-up meeting every day), Sprint Review, and Sprint Retrospective. Burn-down charts (derived from Kanban method and containing at the beginning all tasks to do during a given sprint) are used to monitor the sprint progress. The term "sprint" emphasizes that the work will be done as soon as possible (sprinters run faster than other racers).

Each sprint starts from the *Sprint Planning Meeting* intended to state what will be done in this sprint and how will the chosen work get done? The

Daily Scrum is a 15-minute meeting early morning and it takes place every day. Team members are then expected to say: *What has been accomplished since the last meeting? What will be done before the next meeting? What obstacles are in the way?* Sometimes the sprint is cancelled before its planned end if the Product Owner desires to do so. For example, if business or technology conditions change. But it happens rarely. At the end of the sprint two meetings take place: *Sprint Review* to present what has been done during that sprint and to discuss the current content of the Product Backlog, and *Sprint Retrospective* to inspect the work of the team and potentially to create a plan of improvements to be enacted during the next sprint.

Scrum authors' emphasize that *every Scrum role, rule, and time box is designed to provide the desired benefits and address predictable recurring problems* [7]. So applying Scrum means using all its recommended elements − otherwise there are only so called *Scrumbuts*, like "*We use Scrum but we can't build a piece of functionality in a month, so our sprints are 6 weeks long*" [7]. For example, students say "We use Scrum, but we cannot often communicate with our Product Owner (or meet every day) because each of us lives elsewhere and we have to attend lectures". Implementing only parts of Scrum is possible but the result is not Scrum [8].

1.3.Introducing Scrum process − starting with artifacts concerning requirements

Students of the second semester of courses leading to the master's degree have taken part in the described experiment. Several lectures about Scrum development process have preceded project classes so students have already heard about Scrum process framework, its sprints, roles, and produced artifacts. 52 students worked in 13 teams during the winter semester. There were mostly four people in each team: 11 four-person team, one with three persons, and another one with 5 students. Most of students were eager to practice Scrum methodology because most of them work in software firms which, in turn, try to apply the promising Scrum methodology. Project classes took place every two weeks to present developed artifacts, progress of work and to plan next sprint. Students have implemented their projects mainly in the meantime between classes.

Furthermore, the author has been a mentor of two dissertations having *Scrum* in their titles in the academic year 2012/13. The first one started in March so it is too early for conclusions. Another one entitled "*Using Scrum methodology in a real software project*" has already been finished. It required a lot of mentor's help − many conclusions presented in subsequent parts of this chapter are concerned with this experience. Both dissertations refer to the real software projects. Their authors and many other students imagine themselves as future Scrum Masters.

Students have had freedom to decide who will work in one team and what software application will be implemented (each team worked on different subject). Each team pointed out its *Scrum Master* although nobody was sufficiently experienced with Scrum methodology. The author has been an advisor, supervisor, and a witness of work realized by students in Scrum. At first, a subject and title of an application for each team were specified. Only three teams had the true (real) *Product Owner*. But Product Owners coming from academia did not know Scrum methodology sufficiently and did not understand properly their own role in a software process. It was the reason that an author has partially played also a role of the Product Owner. Sometimes a team itself had to specify requirements in the recommended forms.

All developed documents were checked by the author. At the beginning students wrote their initial ***Product Backlog*** in a form of a table, containing expected program functions, features, and other non-functional requirements for all sprints. Its each row describes one requirement and contains:

- identifier of a requirement (to refer to it during tests and Sprint Review),
- a statement expressed in natural language (a statement expressing one requirement),
- priority expressed by a number form the range <1, 20>.

The number of initial requirements placed in the Product Backlog by each team and other details are presented in Table 1.1. Students have been instructed that there are expected to specify about a dozen requirements as a minimum. Initially some students formulated less than 12 initial requirements. Even in the engineering thesis for three intended sprints there were only 7 requirements at the very beginning (the engineering four-person team is marked as Eng in Table 1.1). With the author's help this number has been doubled soon. Before start of the first sprint, there were 8 to 15 rows in each Product Backlog, as shown in Table 1.1. Most of its content described functional requirements. Some students wrote short gerund clauses in it instead of full sentences expressed in natural language. The gerund clauses are less comprehensive for other people.

Table 1.1. Initial number of requirements in a Product Backlog, number of requirements selected for the first sprint, and number of sprint tasks

Team	I1A	I1B	I1C	I1D	I1E	I1F	I1G	I2A	I2B	I2C	I2D	I2E	I2F	Eng
Product Backlog	12	8	12	12	13	11	12	15	13	16	10	12	8	14
First Sprint Backlog	3	9	3	16	4	4	6	5	9	6	5	4	2	9
Tasks to do in the first sprint	21	10	7	18	13	11	11	7	34	26	12	11	2	24

In Scrum the Product Backlog is dynamic. It is never completed – it may constantly change. Then general requirements were transformed into *user stories*. First, each record in the Product Backlog has been expanded into several sentences, and then to the most popular form (although not strictly required by the Scrum authors), like "As a <*user type*> I want to <*do some action*> to <*achieve desired results*>" or "As a <*role*> I want <*goal/desire*> so that <*benefit*>". In several teams, the number of requirements in the Product Backlog has been significantly increased already during this transformation and became more numerous (ca 20−100%) than before. Priority from the range <1, 20> was assigned to each requirement. Then the content of each Product Backlog has been ordered decreasingly on a base of assigned priorities.

The following mistakes have been made by students in the above activities:

- There are few requirements at the beginning, not enough for 2−3 sprints.
- Requirements have no identifiers.
- Requirements are not expressed in statements of natural language (they are gerund clauses).
- A sentence describing requirement in an initial form of a Product Backlog is not simple but the compound one so it expresses more than one requirement.
- Identifiers assigned to requirements are changed during development process.
- There is too small range of priorities, like <1, 3>, for example.
- Priorities are wrongly assigned to requirements – their values do not help selecting them properly for the first sprint.
- Requirements included in the Product Backlog are not ordered decreasingly.

The listed above mistakes are not very serious. Their reason is a lack of experience. Some fault is also at the side of Product Owners who have no experience with Scrum methodology or even any methodology. Furthermore, software developers do not often consider the user's point of view (they are not aware of it). These critical remarks concern an application of any agile methodology. The Scrum Guide contains no instructions concerning the recommended number and form of requirements, including their identifiers. Scrum authors emphasize a need to express requirements clearly, to assign priorities to them and then to list them on that basis in a decreasing order. *Correctly specified higher ordered items of the Product Backlog are clearer and more detailed than lower ordered ones* [8]. In practice, the requirement must be expressed clearly for anybody (customer and team members) so several sentences, including suggestions concerning test designs, should be stated in every case. Identifiers are helpful during testing and release of a software product – a consumer has to be sure that all his/her requirements have been met.

Then each team developed the ***Sprint Backlog*** as a subset of their Product Backlog. Students formulated a goal of the sprint and tried to estimate an effort/time needed to implement the selected requirements. For implementation in the first sprint, there were selected from ¼ of a total number up to almost all requirements specified in the Product Backlog. One team selected only 2 requirements (from 8 in the whole) for the first sprint. In the meantime, the number of requirements recorded in the Product Backlog has been already increased. That's why, for example, in the case of I1D team, the number of requirements selected for the first sprint is greater than the initial number of all requirements.

Then the selected requirements have been decomposed into ***tasks*** to do during the first sprint. At this point students tended to consider mainly software construction and express only strictly programming tasks. They forgot to include tasks of designing, testing, inspections, writing test scenarios, etc. At last all work for the first sprint has been specified. Tasks had their own identifiers related to requirements and sprint (each task identifier consisted of three elements separated by dots: sprint number, requirement id, and task id). It is not strictly required in the Scrum Guide but it is helpful during a sprint review to summarize all results and to assess contribution of team members.

1.4.Monitoring sprint progress and its results

To monitor a sprint progress, expressed as a total work remaining to do, the *burn-down chart* has been developed by a Scrum Master of each team. At the beginning, the burn-down chart shows the work forecast (number of tasks that should be performed in the time estimated for a given sprint), and then the real time consumed by task performance. According to Scrum principles, the development team tracks their total work during short daily meetings – it should be enough to sum-up their efforts and monitor the progress. There were several mistakes made in some developed charts. At first, no axis of a burn-down chart has been described (and no other legend has been provided) – its y-axis (ordinate) should describe tasks to be performed and its x-axis describes days of sprints or days of whole software project (all sprints).

The most important Scrum artifact is an ***increment***. Each development team was obliged to show results of the sprint. The *scenario of acceptance tests* has been developed by each team and then performed at the end of a given sprint. Such scenarios are not required in Scrum. They are typical rather for the waterfall model than Scrum. But there was no other formal definition of

"Done" and no more stringent criteria for high quality of software product were formulated. In most cases, the author played a role of a purchaser who made technical acceptance of the first (and then next) increment.

One half of test scenarios had to be rewritten because of their improper content. Some students show poor creativity in producing possible scenarios of using their software product in practice as it should be during the test acceptance. The following remarks were made at this point:

- A title of a given scenario does not express/illustrate its content. So it may be incomprehensible for a customer.
- Scenario has no identifier and no name at all.
- There is a lack of initial information describing a given scenario, like references to requirements and context of test implementation.
- The word "user" is overused instead of naming a user role precisely, like *client*, *officer*, *pupil*, *clerk*, etc. – students have problems how to name these roles.
- Too general terms were used like *data* or *information* instead of *teacher's personal data* or *attributes of a book*, for example.
- Some expressions are imprecise. For example, it is not specified directly if data come from the database or from a user in the given case.
- Expected results of a test are laconically worded, like *results are displayed on a screen*.
- Trivial unit tests are described instead of including sophisticated cases.
- Content of a scenario is too general, not referred especially to a tested application so it may be applied to any software product.
- There are only few steps of a scenario; there are no combination of several options required to obtain expected results so mostly only one option or function is tested.

Despite the burn-down chart illustrating a progress in a given sprint, each team prepared at its the end *a burn-down chart* which illustrated a forecast for the whole software project (divided into 3 sprints) and the current implementation of requirements (including increment of the given sprint).

Then two other Scrum events have been described − the *Sprint Review* and the *Sprint Retrospective*. The following mistakes have been noticed in reference to Sprint Review and Sprint Retrospective:

- Not all requirements included in the Sprint Backlog were implemented (mostly 2 requirements were left).
- Lack of communication in team resulted in using different identifiers of the same requirement and/or task.
- It was no exact explanation which requirements/tasks were finished completely.
- It was no explanation and/or justification why some requirements were not met.
- Priorities were assigned improperly in the Product Backlog so the selection of requirements for the first sprint was wrong.
- Some students did not see any difference between burn-down chart for one sprint and for the whole software project.
- There was no precise distinction between the Sprint Review and a Sprint Retrospective. The Sprint Retrospective was not intended to assess the mode of work and possibilities of its improvement as it should be.

Any document containing mistakes had to be improved. So there were subsequent versions of elaborated documents. Each document had its title, author, date of edition, version number, purpose, and addressee.

The modified Product Backlog was presented at the end of the sprint. Modifications were modest – there were 2−3 new requirements added to previous content of the Product Backlog. Priorities assigned to requirements have been often changed.

1.5.Conformity and observed incompatibilities with Scrum

Observations show that students (first of all weak students) tend to simplify problems instead of solving them and also try to ignore and/or pass over some Scrum events. So a problem of applying so called *ScrumButs* [7] appears as mentioned in one of previous sections. People say, for example: *We use Scrum, but having a daily Scrum meeting every day is impossible in our case*. During their studies, some students are accustomed to defend somehow their weak progress at work or a lack of required knowledge. Then they continue this practice at work. Thus it seems to be important to gather and explain them all departures from the Scrum process framework. The results presented below are divided into three parts describing respectively: What was done correctly? What was completely wrong in applying Scrum? What misinterpretations and problems have occurred?

Activities undertaken correctly, it means performed in accordance with Scrum:

- Whole development process is realized in sprints.
- Team size is small enough; there are 3–5 persons in it.
- One team member becomes the Scrum Master.
- A team has the Product Owner although he/she is not a person representing business. For example, the Product Owner is an academic teacher responsible for another course.
- Meeting intended for sprint planning takes place at the beginning of each sprint. It results in a Sprint Backlog.
- Sprint goal is specified at the meeting of a sprint planning.
- Requirements recorded in the Sprint Backlog are decomposed into tasks.
- A forecast is made to estimate time (in hours or person-days) required for each identified task.
- *Sprint Review* takes place at the end of each sprint.
- Burn-down chart shows a progress of work and work to do during each sprint.
- Records concerning the time really consumed for implementation of each task are made at the end of each sprint.
- *Retrospective* meeting takes place at the end of a sprint.
- The new release (issue) is ready at the end of each sprint.

Evident **failures** (undertaken activities diverge significantly from those recommended and required in Scrum):

- Team work does not take place every day.
- Daily meetings do not take place every day (usually twice a week).
- One student nominates himself to be a Scrum Master although he/she has no experience with Scrum.
- A team has no real Product Owner (then the lecturer plays this role in several cases).

- Scrum Master instead of the Product Owner makes transformation of requirements into the most popular form in Scrum (As a <role> I want <to do some action> so that <benefit>).
- Scrum Master should teach the product Owner to make the above transformation but he didn't because of the lack of knowledge and experience.
- Scrum Master is the only one who communicates (not frequently!) with the Product Owner – in Scrum all team members are supposed to communicate with him/her.
- Team members do not work at the same place so direct communication is problematic.
- Sprint goal is not always placed on a table visible for all team members. Student team does not work in one room so it doesn't make sense to place any inscription in room (other solution is to put such inscription on a monitor screen of each team member).
- Scrum Master decomposes personally sprint requirements into tasks to do.
- There is no exact definition of "*Done*" for the Scrum team to assess when work on the product increment is complete.
- Daily meetings last longer than 15 minutes because other problems are discussed during it.
- Product does not work correctly even at the end of the last sprint (in some teams).
- Product Owner limited himself/herself to play a role insufficiently for Scrum purposes and does not participate in Sprint Review and Sprint Retrospective.
- There are breaks between sprints so the next sprint does not begin exactly when the previous one is finished. Some work is done between sprints.
- Retrospective meeting takes place before the Sprint Review (in some cases).
- Product issues are delayed. A delay of the first sprint causes further delays. Usually one member in a team causes a delay but then all team is delayed with work.

Misinterpretations and other problems:

- Scrum Master tries to assign tasks to team members, so he/she plays rather a role of a manager than a servant-leader for the Scrum team − team should be self-organized (team members should select tasks to do by themselves and then be responsible for them).
- A team is not cross-functional as it is expected in Scrum so tasks are assigned to persons who are experienced or specialized in some desired activities like graphics, for example.
- One sprint of a student project lasts longer than 1−4 weeks but there are only several days or even hours of true work on it during that time.
- Product Owner does not understand his role – he/she adds no new requirements during whole development process and specifies them in a free (traditional) form but not as it is expected in Scrum. Then the Product Backlog is not dynamic.
- Each requirement has no unique identifier – such identifiers are not strictly required in Scrum but they are very useful in the case of a student project. Otherwise it would be almost impossible to control requirements, their implementation, and dynamic changes.
- Team members show their results and tests during daily meetings − daily meetings in Scrum are intended to discuss what should be done (tasks to do) not how it has been done already.
- The Scrum Master creates the burn-down chart on a piece of paper, modifies it from time to time, and presents it to the teacher during project classes. Burn-down chart should be modified every day of work and be shown on a table accessible for all team members.
- Product Owner (the author of this chapter is the only exception in playing that role) rarely participates in Scrum events; he/she does not track the development process.
- Increment of a given sprint is not free of defects.
- Retrospective meeting takes place without participation of a Product Owner − students have problems to meet him/her frequently.
- Product Owner does not specify his/her suggestions concerning test scenarios – he/she is not obliged to do it but test cases are welcome in user stories; otherwise students have problems how to implement testing and to ensure quality of a product.

- Tasks assigned to team members require unequal effort of work so some people are more exploited than others.
- Product acceptance is planned at the end of the last sprint – final acceptance is typical for a waterfall model (also in students' work).
- Students do not take enough care for software quality – they limit software quality to meet main functional requirements and to ensure small density of errors.
- Records concerning productivity of team members and a team as a whole are made using a tool like Redmine, for example. Each Scrum Master willingly makes Gantt charts to show project progress although these charts are typical for traditional, task-level project management. In Scrum, managers track requirements not tasks. But translating Product Backlog to the Gantt report is acceptable also by the Scrum authors – it does not require much effort and may be helpful in practice.
- Although some students try to hide their problems, several problems were reported, like: too many tasks to do in one sprint, imprecise specification of requirements, communication problems in team, swapping some files without others' knowledge about it, incompatibility with an applied design pattern, problems with code integration, etc.

Motivation of team members weakens when the time goes on. When all Scrum rules are fully respected then team members are enough motivated to do their work. Team may work in its own tempo, terms are met, and no special motivation is needed. But it works so in the case of a professional firm. Students have no financial motivation. Work goes better when their product is produced for the real life purchaser also without any financial gratification.

1.6.Observations and conclusions

Real life experiences with Scrum at the technical university have been described. Using Scrum, students have implemented their projects from the very beginning to the end – such approach is not often practiced in their semester projects. Students have learnt that applying Scrum does not mean only to name iterations as sprints, a list of requirements as the Product Backlog, and a team leader as the Scrum Master. The team work performed in the chosen methodology is valuable itself. Students have learnt to keep everything visible to all parties and to track progress referred to the specified requirements – in Scrum software developers plan and then report just requirements.

Despite all failures, misinterpretations, and incompatibilities with Scrum, listed in the previous section and known to all participants at the end of semester, students have gained considerable experience in Scrum methodology. They have become aware of Scrum itself and its rules including their required entirety to be applied. Many Scrum rules have been applied although many haven't. Partially it is a result of an organization of studies – students have to share their attention and time between several subjects. Scrum authors recommend the work of a whole team on one software project in one room, every day during the whole week. Daily work and stand-up meetings are not possible in the case of students. Some of them are able to communicate each other using Skype, for example. Other difficulties with Scrum refer to students' attitude and skills. They have no habit to specify requirements precisely, they do not practice daily cooperation with a customer, they do not realize solid testing, etc., although all these elements are basic for any agile methodology.

The author was able to assess the developed artifacts but not to monitor a real sprint progress and Scrum events like daily and other required meetings. Students have trained writing artifacts and short documents required in Scrum. Thus they have not only implemented the code of increments. A discussion on how to apply Scrum correctly took place during project classes − these issues have been superficially attractive for students.

Responsibilities of a Scrum Master require special attention [5]. He/she is supposed to remove barriers between developers and a Product Owner and sometimes to teach him/her how to express and change requirements dynamically and to meet project objectives through Scrum. He/she also tries facilitating creativity and empowerment of team members. Scrum Master improves productivity of the team by applying and improving engineering practices and ensuring enough communication between software developers and the Product Owner. All these require a lot of experience and are difficult for practitioners even in software firms applying Scrum.

Nowadays students are aware of an importance of applied methodology. Many students work in software firms and participate in projects implemented for customers from Germany, United Kingdom, and USA. In these countries at least some customers are ready to cooperate with software developers. So students are eager to train one of agile methodologies. Academic teachers know, in turn, that software engineers should be able to find a job on the global job market, not only a local one.

Some Scrum elements are easier to introduce and realize them than others. Scrum exists only in its entirety. Implementing only some its elements is not Scrum itself – just *Scrumbuts* appear this way. And it has happened in the described cases. Scrum authors warn software developers against such practices. Students should be aware of it. Project management applied in Scrum ranks among so called radical management methods which extend its applications on other disciplines. Scrum can be taught only empirically. At university, any practical experience in this area is valuable.

Projects implemented at the university are still driven mostly by the traditional methodology based on a waterfall model – there are no real customers, no software purchasers and users, then some artifacts are assessed, and the final acceptance takes place at the end of a semester. An applied methodology seems to be the minor problem − this is wrong in the author's opinion. Usually the only stated question is how to solve a given technical problem but not who requires this solution and for what. Then no proper methodology is applied or even no methodology at all. Several students told me that they dream to be employed in a real firm where an agile methodology is applied instead of being forced to work in "yesterday" terms. Experience with Scrum may help realizing it.

References

[1] Boehm B., Turner R.: Balancing Agility and Discipline, Addison-Wesley, Boston 2004.

[2] Highsmith J.: Agile project Management, Addison-Wesley, Boston 2004.

[3] Manifesto for Agile Software Development, Agile Alliance, http://agilemanifesto.org, 2001.

[4] Phillips J.: IT Project Management. On Track from Start to Finish (Polish 3rd edition), Helion, Gliwice 2011.

[5] Schwaber K.: Agile Project Management with Scrum, Microsoft Press, Redmond, 2004.

[6] Schwaber K., Sutherland J.: Software in 30 Days, John Wiley & Sons, Hoboken NJ 2012.

[7] Schwaber K., ScrumButs and Modifying Scrum, http://www.scrum.org/ScrumBut, accessed 25.03.2013.

[8] Sutherland J., Schwaber K.: The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game, on-line at: http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf, recently accessed 15.02.2013.

Chapter 2
Prevention of conceptual errors in the system design

The authors analyzed their experience in the running graduate student team projects of small-scale information systems (IS) based on the domain-driven design approach and the Iconix methodology. The documented project failures clearly indicate the need of an improvement of the conceptual layers of software design because conceptual errors have significantly decreased the quality of the software designed and have extended the duration of the project. The suggested organizational and methodological solution to this problem is a mandatory concept mapping of the interest area of the project and a mandatory maintaining of a checklist that has specially been designed for the detection of the selected categories of conceptual errors. Additionally, a rich image of the general IS concept and a problem domain glossary are recommended. The authors anticipate that the proposed set of tools will result in streamlining of the educational design process.

2.1.Introduction

Modeling and analysis is one of the key steps in the manufacturing process of information systems (IS) [1]. A properly carried out analytical process allows one to successfully design and implement the code, it reduces problems at the implementation stage and ensures the efficiency of the software produced. A thorough analysis of information systems is a key factor in the product quality and, therefore, a key factor in the success of IT projects [1].

The complexity of modern computer systems that operate in a dynamic environment requires a good understanding of a large number of complex issues in a design team. For this reason, there is a number of ordered (standard) software manufacturing processes (known as the software development process) [2], such as the Rational Unified Process (RUP), the Open Unified Process and the Iconix Process[1]. In these processes, at the modeling and design stages, the Unified Modeling Language (UML) is used that enables an accurate description of the IS structure and its functionality.

However, despite the existence of such a precise and powerful tool as the UML, designing of information systems remains a difficult task. Young designers with limited experience face two major types of problems in the use of UML in practice. The first problem is to master UML elements and rules so as not to make any formal errors. For each software engineer, this is only a temporary problem, which disappears after a short time of using the language, despite its complexity. The second problem is more serious as its source is of a behavioral-cognitive nature, which relates to the organization and control of the complex cognitive and emotional activity of a designer [4].

A good design of the architecture and functionality of IS requires designers to use precise concepts that describe the elements and processes of the system, and requires them to understand the role of IS in the business process, especially in tasks which are not intuitive for final users. What is even more important for designers is also to possess an ability to take the users' point of view. For larger teams of designers, there exist specializations in this matter, but the ability of precision conceptual planning is required of every designer, regardless of his/her specialty. Technological support for designers can be provided by validation of conceptual schema. There are a number of research teams that are currently working on such validation methodologies [5]. An example of a European team is a group of IS conceptual modeling at the Technical University of Catalonia-Barcelona[2].

In the conceptual layer, the creation of strictly diagrams such as a domain model, a conceptual database model or a use case diagram [3] can cause conceptual errors resulting in severe defects in the subsequent stages of the design, in the implementation of IS, and in a serious reduction of the quality of the final product [6]. Some industrial software development processes are iterative, which helps to eliminate most of the errors in the subsequent phases of an IS design. Note that in such cases, conceptual errors still cause a prolongation of the production process. In addition, other manufacturing processes, which are useful for smaller projects (e.g. Iconix), do not possess an iterative

[1] http://iconixprocess.com/
[2] Conceptual Modeling of IS research group, http://guifre.lsi.upc.edu/

nature. They are also often less formal: in these methods, even some serious conceptual errors may escape one's notice, resulting in a faulty software.

Educational projects deserve separate attention. Upgrading of computer studies should result in the increasing role of specialized design classes [7]. The problem is that the specificity of the projects carried out under the plan of study and a rigid frame of semester hours does not allow one to use an iterative approach and forces de facto a manufacturing process that is similar to the waterfall model. Significant time limitations and a small design team do not allow real iterations, especially in single thesis projects. This situation highlights the role of a conceptual model validation in a didactic IS project because the requirement for an advanced conceptual analysis increases the chance of the success of a project (course credit) and leads to an increase in the quality of the acquired knowledge and skills.

2.2. The research material, the idea of extending the instrumentation

The authors have been teaching advanced software engineering classes for several years. Classes are conducted in accordance with a project method which is similar to actual IT projects, and address issues related to the production of selected systems at the level of a small business management system, such as a car rental, a craft shop or a dispensary. Projects are carried out mainly in accordance with the Iconix process, due to its moderate formalization and its average level of complexity of formal requirements, which young designers can fulfill during one-semester classes. It is important to emphasize that the project tasks were performed by graduate students, who already hold bachelor's degrees (in Poland, licentiate or engineer) and possess some initial experience in computer programming and software engineering (because of their industrial practice and diploma projects).

Project documentation, accumulated over last five semesters, which describes those activities, allowed the authors to note a regular occurrence of some conceptual problems in students' projects. It was discovered, as a result of the study [8], that these problems were caused by shallow modeling and limited understanding of the problem being solved. In this research, the identified types of conceptual errors were grouped according to their impact on the project success.

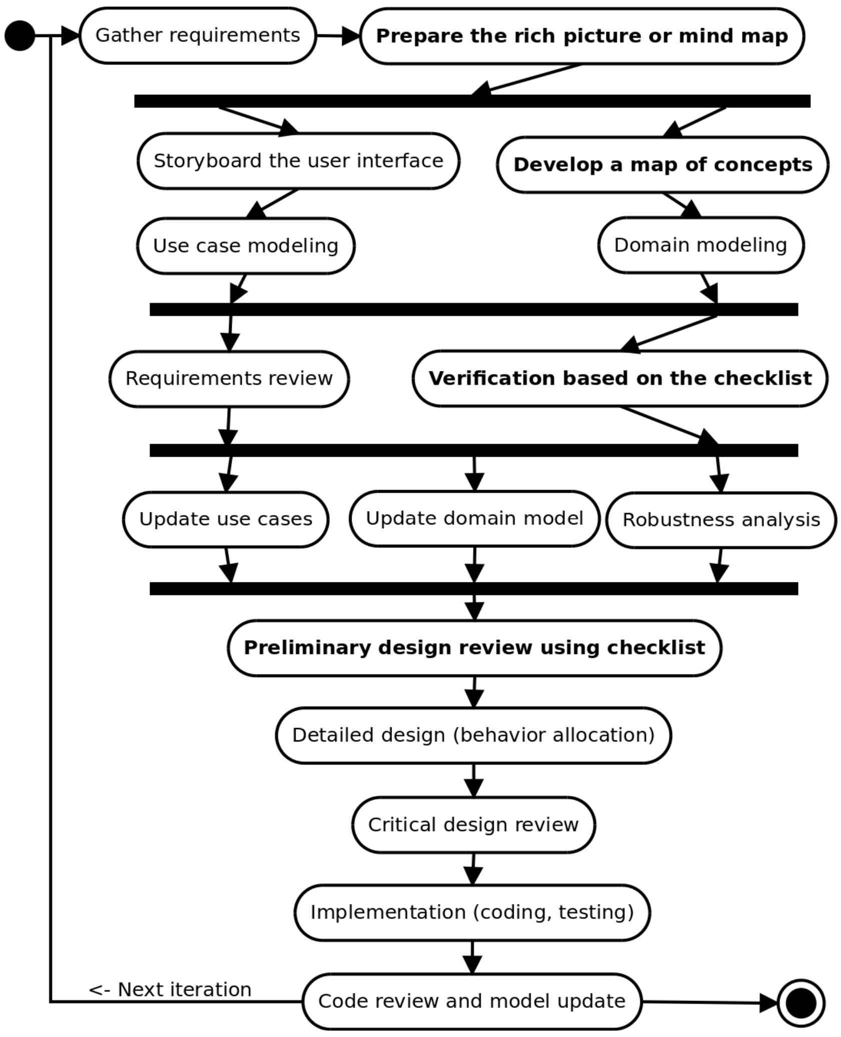

Gather requirements
Prepare the rich picture or mind map
Storyboard the user interface
Develop a map of concepts
Use case modeling
Domain modeling
Requirements review
Verification based on the checklist
Update use cases
Update domain model
Robustness analysis
Preliminary design review using checklist
Detailed design (behavior allocation)
Critical design review
Implementation (coding, testing)
<- Next iteration
Code review and model update

Fig. 2.1. Roadmap of extended Iconix process. Source: own

These observations allowed us to suggest a conceptual error preventing technique based on an extension of the project instrumentation. While remaining in the Iconix manufacturing process, a designer should use additional analytical tools, which are well-known from the areas of knowledge engineering.

In our opinion, the tools can be as follows: a conceptual map, which fits well the scale and objective of the project, and a checklist designed to verify the absence of some categories of conceptual errors in the project. As an additional tool, we can recommend a rich picture of the whole IS and a glossary for the problem domain. The authors' proposal, therefore, is focused on a modification and an extension of the Iconix design process for those items that are indicated in Figure 2.1. It should be noted that architectural decisions are made after the use of case modeling, starting from the robustness diagram.

2.3. Example of the use of conceptual maps in the IS project

One of the most effective error preventing techniques in student projects turned out to be concept mapping [9], that had to be done before the actual start of work on the IS project. In terms of software engineering, it may be said that conceptual modeling of a problem should take place at the planning stage of a project (in the strategic phase). The main task of the concept map (a kind of semantic web) in this context is a "synchronization of the vocabulary" of all the project participants. In our experience, the CmapTools software[3], developed by The Institute for Human and Machine Cognition, proved to be a useful and user-friendly tool for developing a concept map with students.

Figure 2.2 shows a sample map of concepts done in order to prepare the project team to work on a particular IS. This map is set in the context of the project goal, which provides the creation of a database web application supporting auctions conducted over the Internet. This map is not a representation of any "absolute knowledge" about the problem domain, but it is rather a look at this domain agreed by the project participants. An ideal situation would be to aim for encyclopedic representation of the problem, but in a real situation, one can have satisfaction from some simplified mapping of issues. It is important that the map includes recorded ontological arrangements adopted by the project participants.

The students were generally well familiar with the domain of the project generally because of the high popularity of websites like allegro.pl and ebay.com, but work on the map shown in Figure 2.2 presented for the young

[3] The IHMC CmapTools program, http://cmap.ihmc.us/

designers some very important features of auctions. For example, a lot of discussion was caused by the problem of bringing Offers, Prices and Goods into association. The same team discussed a lot about whether Bidding concerns "Auction Item" and whether during the auction one can use the "Market Price" concept of the Goods. By working on this map, the team could quickly figure out the problem situation and move on to specify the major processes that had to be assisted by IS. The map also proved to be useful in any analysis of use cases, and when the team was building the data model of the auction house. The main positive result of using this map in the project proved to be a consistent vocabulary, which can easily be seen in the documentation, as well in the text-based UI design.

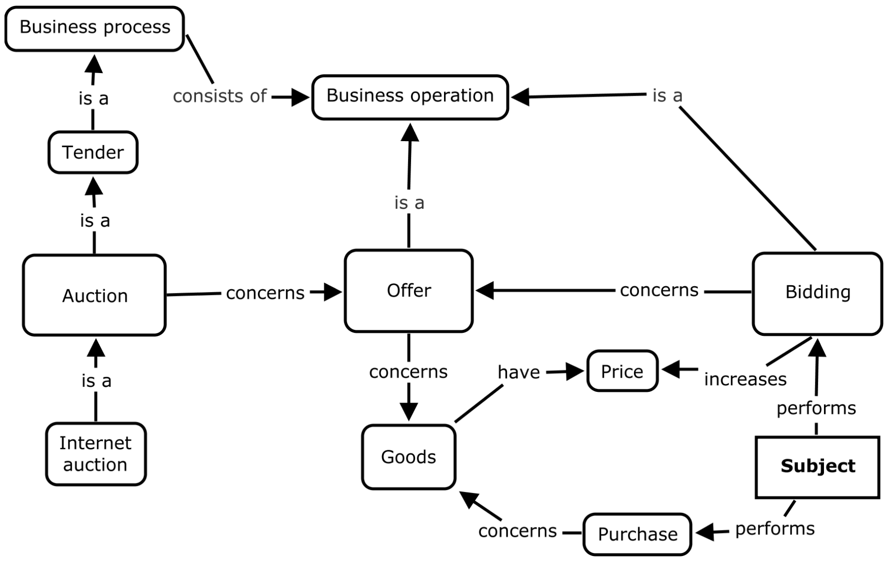


Fig. 2.2. Concept map for the Internet Auction House. Source: own

The domain dictionary was another effective technique to prevent errors in student projects, in regard to our experience. Such a template-based dictionary, prepared by the team, can contain the same information about the components of the problem domain and about their relationships in the form of a structured text. This dictionary may even contain information, that would not normally be included in a concept map, e.g.: restrictions, operating parameters, minimum requirements, etc. The dictionary, prepared in a tabular form, can be much more informative and at the same time as readable as a map of concepts.

Table 2.1. Sample template of project dictionary

Concept	Related to the concepts	Parameters	Description[4]
Auction	Offer, Tender	The Term, Type	This is a public sale in which the buyer of a thing is the person who offers the highest amount.
Offer	Auction, Goods, Bidding	Amount, Time	A formal proposal for concluding the contract, the one-time transaction in the context of auctions, submitted by the potential buyer.
Subject	Bidding, Goods, Purchase	ID, Status	Auction Participant who has specific privileges in the system, which can start auctions as a buyer, participate, or submit offers.
Goods	Auction, Subject, Offer	Description, Value	Subject of the auction and potentially of transactions, described in the IS.

2.4. Example of using a checklist

The main advantage of a checklist as a tool is its simplicity of use, regardless of whether it is prepared in a traditional paper form or in the form of a computer program. A precise checklist is condensed expert knowledge given in an easily accessible form. The checklist will be very helpful for a correctness verification of UML models, if it contains a specified and sufficiently large number of possible "crisis" situations, which are not obvious from the point of view of the modeling language rules.

Figure 2.3 shows an example of a use case diagram that should describe the functionality of the Auction House. This diagram contains a number of common conceptual mistakes. It was created by the students without the use of a checklist. Some of errors from Fig. 2.3 can be easily overcome by using checklist questions.

[4] Based on http://oxforddictionaries.com/

One can additionally note in the Diagram 3, the mistake of an incorrect designation of the relationship direction, which replaces the cause with the optional effect: the use cases of "Bid item" and "Buy item".

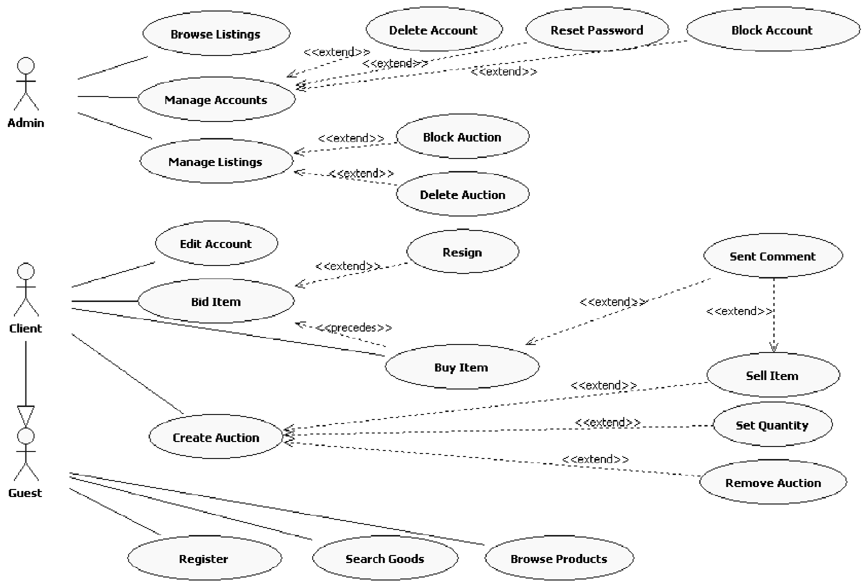


Fig. 2.3. Use case diagram for the Auction House, developed without an adoption of a checklist (see Table. 2.2). Source: own

Table 2.2 contains a fragment of our checklist used to detect defects in the use cases diagrams without targeting a specific type of IS. The questions included in the list (column no. 2, Question) are ranked in the ascending order according to the level of a semantic difficulty from the designer's point of view. The template, adopted by the authors for building the questions, is as follows: being → action → object, which is similar to the order resulting from a natural grammar that reflects the evolutionary cognitive priority [10].

Table 2.2. Some checklist questions that were verified during our studies

Nr	Question	Answer
1	Do specified actors cover the business model of the project?	Yes (no errors associated with the lack of modeling of select-ed categories of users).
2	Are all the use cases unique?	Probably not. Cases “Browse Listings” and “Browse Prod-ucts” should be the same, unless the Administrator needs some additional powerful features when viewing a list of items. However, the Guest and the Client, in every known Auction House implementation, can browse the offer, which means they can see the parameters of the items (goods) and the conditions of sale. In most cases, one cannot browse the items in an isolation from the offer.
3	Are the selected objects related to appropriate actors and ac-tions?	No. There are errors associated with ontology. The most im-portant item in most cases is the auction, because it is a foun-dation for the business model. But, in the diagram, there are other two different objects: goods and items, which are basi-cally represented as synonyms of an auction. However, in the practice of auction houses, it is impossible to operate with goods without manipulating the auctions; goods are here only the components of auctions. Hence, you cannot search for goods while not searching for auctions, but in order to create an auction, you must add an item/product/goods. In this case, an error is also related to the third point checklist.
4	Do use cases that are acces-sible to various actors possess the same flow?	Not applicable, there is no improper sharing of use cases.
5	Do specified use cases cover the necessary functionality in the established business mod-el?	Probably not. There are no cases that are complementary to some of the existing ones, for example, there is no “Unlock Account” for the existing “Block Account”. There is also a deficiency of some use cases, preferred from a business point of view, for example realizing adjudication of disputes, man-aging of advertising, managing of personal auctions. Moreo-ver, if we do model viewing and searching activities as two separate use cases, the Administrator also should have the use case of "Search Auctions".
6	Is the inher-itance of use cases and actors justified?	Yes, there is no faulty inheritance of use cases, but the inher-itance of actors is dubious. The customer should have the use cases of "Search Goods" and "Browse Goods" (rather than "Search Auctions" and "Browse Auctions"), but the use case of "Register" should be blocked to him: this is not indicated on the diagram.

A negative answer to even a single question indicates the need to amend the use case diagram. Using mentioned above checklist, we can easily find, and thus correct the errors in the analyzed diagram. In column 3 (Answer) of Table 2.2, there are described errors found in the diagram shown in Figure 2.3.

2.5. Example of the use of a rich picture

An initialization of the IS project involves taking strategic decisions on appropriate priorities and elaborating of the system conception. Thus, from the beginning it is advisable to agree on the main concepts and an outline of the system architecture that is accessible to all stakeholders. The authors propose to capture a first look at the mechanisms of the future information system using the unconventional diagram shown in Figure 2.4.

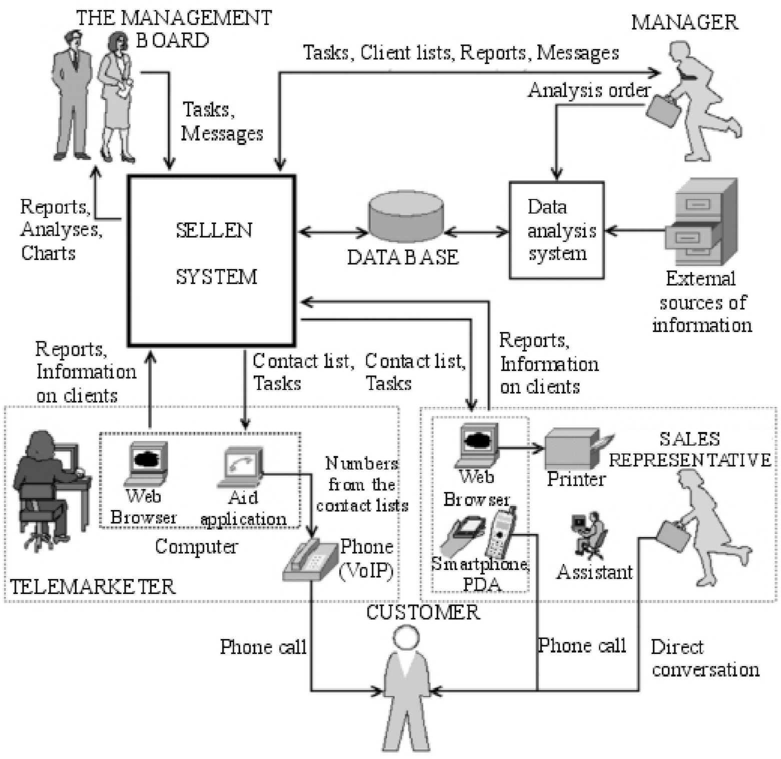


Fig. 2.4. Rich picture of Trade Agency. Source: [11].

This diagram in fact visualizes data flows between the major modules of the system and its environment. The arrows between the pictures are very essential for the diagram because the main lines indicate data flows, and the inscriptions on them identify the transmitted documents or data sets. In terms of software engineering, it is a contextual diagram. The advantage of the technique suggested is that the information perspective is suited to start discussions with the client, sponsor, or shareholder using the same concepts. Knowledge about the project issues, included in the perspective proposed, can serve as inspiration of a further functional analysis of the system. This technique is a part of the systems engineering methodology [1]. It provides a mechanism for learning about complex topics or ill-defined problems by drawing their detailed ("rich") representation.

The situation represented in Figure 2.4 applies to the information system of Trade Agency. This figure tells us about the planned services provided by the system and about its users. The subtitles increase the informativity and in a specific situation allow one to start building the data model of the system.

2.6. Conclusions

The practice of running graduate students' team projects of small-scale information systems (as part of software engineering teaching) points to the perception of the existence of problems at the stage of the conceptual modeling of IS. Many of these problems can be solved with relatively uncomplicated extensions of the analytical instrumentation.

As our practice has shown, using some extra conceptual perspective, one can minimize this problem. In an academic environment, the following tools have proved to be very useful: a conceptual map or a project domain dictionary, a checklist for a conceptual verification of the functionality and structure of the system proposed, and a rich picture of IS. A relatively easy use of these tools has led to an observable improved quality of projects architecture, to a more functional and useful UI, and to a compressed implementation of information systems.

The proposed tools proved to be effective as an enhancement of the Iconix process in the case of a cascade production of small-scale information systems. However, there are no contraindications to use this approach in the iterative production: the team will not gain in terms of time in completing the project, but it may gain in terms of the product quality by creating a better IS.

The authors of this chapter have had an opportunity to test their proposals in an educational environment, in running graduate students' team projects, and therefore among those who have software development knowledge, but with very limited practical experience.

Bibliography

[1] Cecelja F., Manufacturing information and data systems: analysis, design and practice, Penton Press, London, 2002.

[2] Kruchten P., The rational unified process: an introduction, Addison-Wesley Longman Publishing Co., Inc., Boston 2003.

[3] Rosenberg D., Stephens M., Use case driven object modeling with UML. Theory and practice, Apress 2007.

[4] Sternberg R.J., Cognitive psychology, Wadsworth Publishing, Belmont, 2008.

[5] Tort A., Olivéa A., An approach to testing conceptual schemas, Data & Knowledge Engineering, Vol. 69, Issue 6, June 2010, p. 598-618.

[6] Beynon-Davies P., Information systems development: an introduction to information systems engineering, Macmillan, London, 1998.

[7] Gabryel P., Polskie uczelnie europejskie, Rzeczpospolita, 7.10.2010.

[8] Statkiewicz M., Susłow W., Klasyfikacja błędów konceptualnych, popełnianych przez studentów kierunku informatyka w trakcie modelowania i projektowania systemów informatycznych, Zeszyty Studia i Materiały Polskiego Stowarzyszenia Zarządzania Wiedzą, nr. 36, 2010, s. 199-212.

[9] Węgrzyn A., Węgrzyn E., Technologia mappingu jako wsparcie nauczyciela w przekazie wiedzy, W: Uczelnia oparta na wiedzy/Red. Gołębiowski T., Mierzejewska B., Fundacja Promocji i Akredytacji Kierunków Ekonomicznych, Warszawa 2005, s. 239-252.

[10] Buss D. M., Evolutionary psychology: the new science of the mind, MA: Omegatype Typography, Inc., Boston, 2008.

[11] Jeż S., Susłow W., Developing the computerization concept of the commercial agency by combining analytical perspectives, Studies and Materials in Applied Computer Science, Vol. 3, No. 4, 2011, pp. 41-48.

Chapter 3

The decision making model for design of service - oriented systems

The evolution of service-oriented systems is an intensive process of modifications, additions and removals of services and their compositions, driven by ever changing business requirements. As such systems rarely reach a stable state and are subject to many changes during their lifetime, documenting these changes and their rationale can facilitate making further changes and extensions. In this chapter, we present a model for capturing architectural decisions specially tailored for documenting the evolution of service-oriented systems. It follows an intuitive process of decision-making during the evolution of services and their compositions. It enables architectural decisions made during a single evolution step to be traced, and allows changes made to artefacts developed in earlier evolution steps to be documented. Our model includes the set of relations that allow for the auto-detection of previous decisions, which could then be reused when implementing the change. Additionally, the set of relations provides a mechanism for tracing changes of requirements and identifying the impact of such changes. The model has been validated on a real world example.

3.1. Introduction

Modern organisations must perpetually modify their computer systems in order to keep up with frequently changing or emerging business requirements. This makes a system's modifiability and evolvability a primary concern for the architecture stakeholders. Service-oriented architectures deliver a system design paradigm that is particularly suitable for rapidly evolving systems. They address the concern of modifiability and evolvability by enabling the development of a new functionality by composing a new functionality from the existing services. Such a development takes place mainly at an architectural level.

In order to develop further changes, it is necessary to comprehend a system's architecture in its current state. Architectural knowledge is necessary for this comprehension. In a system that is subject to frequent changes, architectural knowledge is created as subsequent modifications are developed. Models and supporting tools are needed to facilitate capturing architectural knowledge together with the development of changes (compare [2]).

The argument presented in this chapter is as follows:

- The evolution of a service-oriented system, similarly to software architecture [3], can be represented as a set of architectural decisions;
- Although some general models that support capturing architectural knowledge and the decision-making process have already been developed (e.g., [1], [2]), a model specially tailored to service-oriented systems would assist architects more effectively than these general models, where it supports the intuitive process of decision-making during the evolution of services and their compositions;
- The model postulated above has been presented in this chapter. It enables:
    - capturing architectural decisions that have been made in consecutive evolution steps – together with the models of the entities of service-oriented systems, modified by these decisions (e.g. service compositions are modelled in BPMN);
    - defining relations between decisions that follow a cascading structure of changes made to the services and their compositions (i.e. change to a service may force changes to services operation, which in turn may force changes to service composition, which in turn may require changes to services etc.;);
    - defining relations that show the evolution of architectural decisions and of the requirements that drive these decisions;
    - tracing the decision process that takes place in a single evolution step;
    - tracing requirements, architectural decisions and the artefacts modified by them during consecutive evolution steps – provides a versioning mechanism for decisions and artefacts
    - auto-detection of previous decisions that could be reused when implementing changes.

Finally our concept has been verified on a real world example.

The rest of the chapter has been organised as follows: the evolution capturing model is presented in section 2, its application has been illustrated on an example in section 3, related work and contribution of this chapter is discussed in section 4, and finally the outcomes and further research outlook is presented in section 5.

3.2. Documenting the evolution of service-oriented systems

The Evolution Documentation Model consists of two basic components:

SOA System Model (section 3.2.1) – a set of models representing the components of service-oriented systems (business processes, services, service operations and their internal logic, service compositions) at various levels of detail. This model is based on a conceptual model of the SOA system proposed in [4].

Evolution Capturing Model (section 3.2.2) – documenting the changes introduced by the evolution steps. Such changes may concern services, operations, service compositions and detailed models describing the implementation of services. The evolution model provides a traceability mechanism for SOA System Models and architectural decisions, a mechanism for the auto-detection of reusable decisions, and also facilitates impact analysis and capturing the architectural knowledge emerging during the development of changes

3.2.1. SOA System Model

Service-oriented systems implement one or more business processes, whose activities are supported by suitable business services. These services, in turn, comprise a number of operations. The latter may be implemented as a service composition (composed of other service operations) or, in the case of elementary services, as a piece of a source code. These dependencies have been reflected in the SOA System Model (Fig. 3.1). The model comprises the following entities:

The set of "Business Processes" supported by a service-oriented system. These BPMN models (class Flow) abstract from the implementation details such as service compositions, services definitions, interfaces, operations, operation arguments, etc. Each business process is associated with a set of tasks (class Task), which are also included in the workflow represented in BPMN.

The set of models that represent services used to support business processes. These models form a cascading, recursive structure as a model of a service is connected with a number of operations, each of which can be either an invocation of a basic (non-composed) service operation, or of a service composition, etc. The set of models that represent services is represented by the following classes:

- Service consists of a set of operations (represented as associations with an operation). Therefore, service is a kind of container, or simply a label for the set of its operations.
- Operation is an entity in which computation actually takes place.
- Service Composition: is the model in BPMN that expresses the workflow composed of the invocations of operations (operations belonging to various services – internal and provided by the external providers). Service composition should be assigned to the service operation that actually provides its input and output interface.

The set of low-level, detailed models (typically in UML) and executable code. Note that these models may concern only basic services developed in-house, or in the possession of the system's owner.

It is worth emphasising that the SOA System Model reflects the structure of real world, service-oriented systems, which is particularly noticeable in the relation between services and their operations. We also assume that the tasks can be one-to-one associated with service-operations, which implement them in a service-oriented system. This imposes a certain rigour both on business analysts and SOA system designers, which is needed to make business even closer to IT.

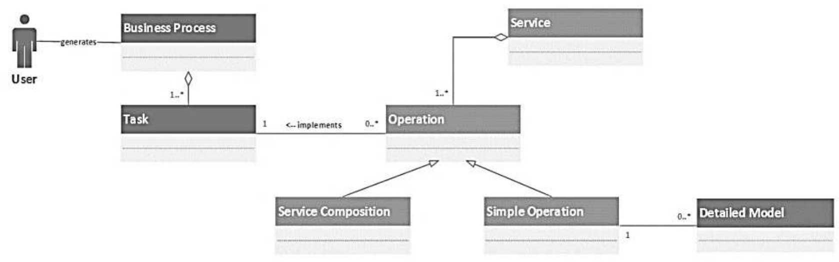


Fig. 3.1. The conceptual model of service-oriented system

3.2.2. Evolution Capturing Model

The changes made to a service-oriented system are of a cascading structure, i.e. a change to a service may force changes to services operation. These, in turn, may force changes to service composition, which may then require changes to services etc. The Evolution Capturing Model (Fig. 3.2) documents evolution as a set of "Evolution Steps". Each Evolution Step is motivated by an RFC document (Request for Change), which specifies the requested change, describes its motivation, business and, if needed, technical context (contains business process models). The step itself comprises a hierarchy of architectural decisions, capturing the changes made to the models of the SOA System Model. Such a cascading effect is reflected by the triggers relation.

Architectural decisions contain the following properties: input – artefact before modification, outcome – artefact after modification, as well as artefact alternatives considered during a change's development (alternatives) and requirements that should be met by the decision. A detailed description of the evolution-capturing model with several types of relations between model entities and their formal definitions (not included in Fig. 3.2) has been presented in the rest of this section. An example of the evolution capturing model is presented in Fig. 3.4.

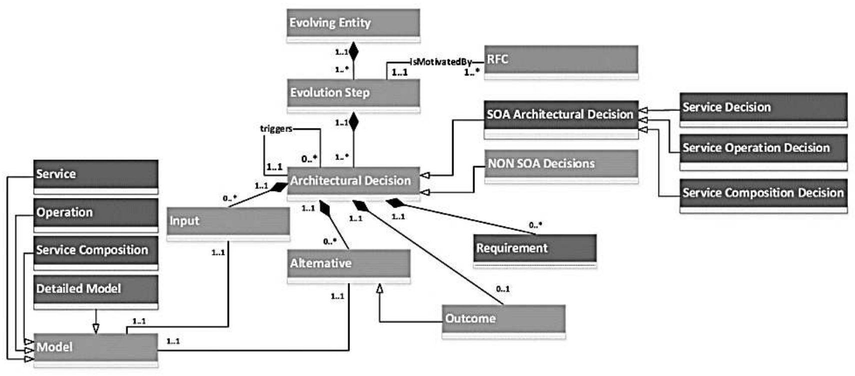


Fig. 3.2. The conceptual model of the Evolution Documentation Model
for a service-oriented system

Definition 1 – Entities of the Evolution Documentation Model:

Model – Let SOM be a set of operation models, let SRM be a set of service models, let SCM be a set of service composition models and let DTM be a set of detailed models. Model is the sum of: SOM, SRM, SCM and DTM:

$$M = SRM \cup SCM \cup SOM \cup DTM$$

Rationale: Model is a set of all the models that evolve during the evolution of a service-oriented system.

Evolving Entity {Service Composition, Service or Service operation} – Let EE be an Evolving Entity $EE = \{(n, d / n, d \in String\}$ where n is the name of an evolving service, service composition or service operation and d a description.

Evolution Step – Let ES be a set of evolution steps $ES = \{(n, d / n, d \in String\}$ where n is a name and d a description.

Request For Change – Let RFC be a set of RFCs $RFC = \{(n, d / n, d \in String\}$ where n is a name and d a description.

Architectural decision – Let AD be a set of architectural decisions $AD = \{(n, d, s / n, d, s \in String\}$ where n is a name, d a description and s state = {being solved, resolved} (being solved – decision has been created, but it has not been resolved yet, resolved - decision has not been resolved yet).

Architectural decision input – Let IN be a set of architectural decision inputs $IN = \{(n, d, m / n, d \in String, m \in M\}$ where n is a name, d is a description and m is the model before evolution.

Architectural decision alternatives – Let A be a set of architectural decision alternatives $A = \{(n, d, m, \{p\}, \{c\} / n, d, \{p\}, \{c\} \in String, m \in M\}$ where n is a name, d is a description, {p} is the set of pros, {c} is the set of cons and m is the model after evolution.

Outcome – Let O be a set of outcomes $O = \{(n, d, m, \{p\}, \{c\} / n, d, \{p\}, \{c\} \in String, m \in M\}$ where n is a name, d is a description, {p} is the set of pros, {c} is the set of cons and m is the model after evolution.

Requirement – Let R be a set of requirements $R = \{(n, d / n, d \in String\}$ where n is a name and d is a description. Defined requirements have to be met by outcome of decision.

Definition 3 – *Contains* relations:

Let $\prec$ be a contains relation. Contains relation links architectural decisions with their components: inputs, alternatives, outcomes and requirements. Additionally it links decisions with evolution steps and evolution steps to the evolving entity. The contains relation is expressed by grammar of UML.

Definition 4 – *IsMotivatedBy* relation:

Let $\mathrm{isMotivatedBy} \subseteq \mathrm{ES} \times \mathrm{RFC}$ be an *isMotivatedBy* relation defined between the evolution step and RFC. An *isMotivatedBy* relation is not reflexive and not transitive.

Rationale: An *isMotivatedBy* relation describes a connection between an evolution step and the RFCs that motivate it.

Definition 5 –*Triggers* relations:

Let $triggers \subseteq AD \times AD$ be a triggers relation defined between two architectural decisions. If $(AD_1\ triggers\ AD_2)$, we also say that *AD_1 triggers AD_2* and AD_2 is triggered by AD_1.This relation shows the cascading of the decision-making process and means that AD_1 forces the need for making AD_2. A triggers relation is transitive: $AD_1\ triggers\ AD_2 \cup AD_2\ triggers\ AD_3 \Longrightarrow AD_1\ triggers\ AD_3)$ and not inverse: $AD_1\ triggers\ AD_2 \Longrightarrow \neg(AD_2\ triggers\ AD_1)$.

Rationale: A triggers relation shows the flow of a decision process – it captures the cascading structure of a decision-making process.

Definition 6: Service Composition Evolution Tree

The Service Composition Evolution Tree has been defined as follow:

$$\mathbb{T} = (EE \cup ES \cup AD \cup IN \cup A \cup O \cup R, \prec, triggers)$$

Definition 7 – *isMet* and *isNotMet* relation

Let $isMet \subseteq R \times (A \cup O)$ be a *isMet* relation defined between requirement and alternatives or outcomes and express that requirement is met by the outcome or alternative. Let $isNotMet \subseteq R \times (A \cup O)$ be a *isNotMet* relation defined between a requirement and alternatives or outcomes and express that requirement is not met by the outcome or alternative. Neither of these relations are reflexive and transitive.

Rationale: The *isMet* relation means that a requirement is met by an alternative or outcome, otherwise a *isNotMet* relation should exist. The *isMet* relation is default.

Definition 8 – *Modifies* relation:

A *modifies* relation is a relation between architectural decisions made in subsequent evolution steps. It can be detected automatically if both decisions modify the same model (i.e. If decision AD modifies model M and produces model M', and in the next evolution step decision AD' modifies M' and produces M" then: $AD'\ modifies\ AD$). We say that *AD' modifies AD* and AD is modified by AD'. A modifies relation is not inverse: $AD_2\ modifies\ AD_1 \Longrightarrow \neg(AD_1\ modifies\ AD_2)$.

Rationale: The *modifies* relation shows the evolution of an architectural decision–it represents the sequence of successive versions of the decision in subsequent evolution steps.

Definition 9 – *Overrides* relation:

An *overrides* relation is a relation between requirements in subsequent evolution steps. It can be defined between requirements connected to decisions in the modifies relation, i.e. If decision AD_2 modifies decision AD_1, then requirement $R_i(AD_2)$ can be in an overrides relation with $R_j(AD_1)$, then: $R_i(AD_2) overrides\ R_j(AD_1)$. We can say that: $R_i(AD_2)$ overrides $R_j(AD_1)$ and $R_j(AD_1)$ is overridden by $R_i(AD_2)$. An overrides relation is transitive: If $AD_3\ modifies\ AD_2\ \cup\ AD_2\ modifies\ AD_1 \cup\ AD_3\ modifies\ AD_1$ then

$R_i(AD_3)\ overrides\ R_j(AD_2) \cup R_j(AD_2)\ overrides\ R_k(AD_1) \Longrightarrow$
$R_i(AD_3)\ overrides\ R_k(AD_1)$ and not inverse:
$R_i(AD_2)\ overrides\ R_j(AD_1) \Longrightarrow \neg(R_j(AD_1)\ overrides\ R_i(AD_2)$.

Rationale: The *overrides* relation represents the evolution of a requirement–it designates the road of successive versions of the requirement in subsequent evolution steps.

Definition 10 – Context:

The context of an architectural decision AD$_n$ is a set of all outcomes and requirements of decisions made before decision AD$_n$ and triggered it. Let $AD_m, AD_n \in AD$ (AD – Architectural decision), $O_1, \dots, O_n \in O$ (O – Outcome), $R_{11}, \dots R_{n\infty} \in R$ (R – Requirement), context is represented by:

$$Context(AD_n) = [O_n, R_{n1}, \dots, R_{n\infty}] \cup \bigcup_{AD_m\ triggers\ AD_n} [O_m, R_{m1}, \dots, R_{m\infty}]$$

$$Context(AD_n) \equiv Ctx\ ([O_1, R_{11}, \dots, R_{1\infty}], \dots, [O_n, R_{n1}, \dots, R_{n\infty}])$$

Rationale: Context represents decision path which includes all outcomes and requirements of decisions that have an impact on the considered decision and itself.

Definition 11 –Evolution context:

The evolution context of an architectural decision AD$_n$ contains all the outcomes and requirements connected to decisions that AD$_n$ modifies (being in a modifies relation with AD$_n$). *Let* $AD_m, AD_n \in AD$ (AD – Architectural decision), $O_1, \dots, O_n \in O$ (O – Outcome*),* $R_{11}, \dots R_{n\infty} \in R$ (R – Requirement), Evolution Context is represented by:

$$EvolutionContext(AD_n) = [O_n, R_{n1}, \dots, R_{n\infty}] \cup \bigcup_{AD_n\ modifies\ AD_m} [O_m, R_{m1}, \dots, R_{m\infty}]$$

$$EvolutionContext(AD_n) \equiv EvCtx\ ([O_1, R_{11}, \dots, R_{1\infty}], \dots, [O_n, R_{n1}, \dots, R_{n\infty}])$$

Rationale: Evolution context represents history of the evolution of an architectural decision and includes all the outcomes and requirements of previous versions of the decision and the decision itself.

Definition 12 – Current Context:

Let AD – Architectural Decision, O – Outcome, R – Requirement, $AD_n \in AD$ then:

$$CurrentContext(AD_n) = [O_n, R_{n1}, \dots, R_{n\infty}]$$

Rationale: The CurrentContext of a decision represents the outcome and the set of requirements connected to it.

Definition 13 – *IsStronglyCompatibleWith* relation:

Let AD – Architectural Decision, A – Alternative, O – Outcome, R – Requirement, $AD_m, AD_n \in AD, AD_n\ modifies\ AD_m$, $AD_m \prec O_m$, $AD_m \prec R_{mi}$, $\forall_{i\in\mathbb{N}}\ O_m isMet\ R_{mi}$

and $CurrentContext(AD_m) = [O_m, R_{m1}, \dots, R_{mi}]$ where $i \in \mathbb{N}$ then:

$$[R_{n1}, \dots, R_{ni}] \equiv CurrentContext(AD_m) \backslash O_m \Rightarrow O_m\ isStronglyCompatibleWith\ AD_n$$

Rationale: The *isStronglyCompatibleWith* relation means that the previous version of the decision could be reused, and all the requirements of the considered decision and its previous version must be exactly the same.

Definition 14 – *IsCompatibleWith* relation:

Let AD – Architectural Decision, A – Alternative, O – Outcome, R – Requirement, $AD_n, AD_m \in AD$, $AD_n\ modifies\ AD_m$, $AD_m \prec O_m, AD_m \prec R_{mi}$ where $i \in \mathbb{N}$ and $\forall_{i\in\mathbb{N}}\ O_m isMet\ R_{mi}$ then:

$$\exists(R_{nk}\ overrides\ R_{ml}\ where\ k, l\ \lesssim i)$$
$$\wedge \neg\left(\neg(R_{nk}\ overrides\ R_{ml}) \vee \neg(R_{nk}\ \equiv\ R_{ml})\right)$$
$$\Rightarrow O_m\ isCompatibleWith\ AD_n$$

Rationale: The *IsCompatibleWith* relation means that the previous version of the decision is eligible for reuse, and at least one requirement has to be overridden, but not identical. The architect has to verify whether the overridden requirement is met by the considered decision. If all the requirements are met, the decision could be reused.

Definition 15 – *IsIncompatibleWith* relation:

Let AD – Architectural Decision, A – Alternative, O – Outcome, R – Requirement, $AD_n, AD_m \in AD$, $AD_n\ modifies\ AD_m$, and $AD_m \prec O_m$ then:

$$\neg(O_m\ isStronglyCompatibleWith\ AD_n) \wedge \neg(O_m\ isCompatibleWith\ AD_n) \Rightarrow O_m\ isIncompatibleWith\ AD_n$$

Rationale: The *isIncompatibleWith* relation means that the previous version of a decision being considered cannot be reused.

3.2.3. Modeling Methodology

The Evolution Capturing Model is created as follows: the decision-making process is captured as a hierarchy of architectural decisions; this structure is reflected by the "*triggers*" relation (see def. 5). The hierarchical decision-making is summarised by the context of a decision (see def. 10) that facilitates making further decisions.

The single evolution step could contain a number of decisions and consequently a number of hierarchies built out of them. The RFC (or RFCs) motivates each evolution step (see def. 4).

An architectural decision evolves when its context is changed. This may be caused by removing a change or adding requirements related to that decision. These changes are usually motivated by RFCs (new business requirements). The evolution of architectural decisions is represented by the "*modifies*" relation (see def. 8) in our model. In turn, changes in requirements are represented by the "*overrides*" relation (see def. 9). Finally, the history of the evolution of a decision is captured as a evolution context (see def. 11).

A decision can be changed many times. It is very probable that the context and requirements of the decision could be similar or even identical in various evolution steps. Our model allows such situations to be detected, which can result in the decision being reused (see. see def. 13, 14, 15).

3.3.Example

Our approach has been illustrated using the example of the evolution of service composition. This composition describes the structure of service developed for the automation of internet payments. The architecture of this composition has been shown in Fig. 3.3(a).

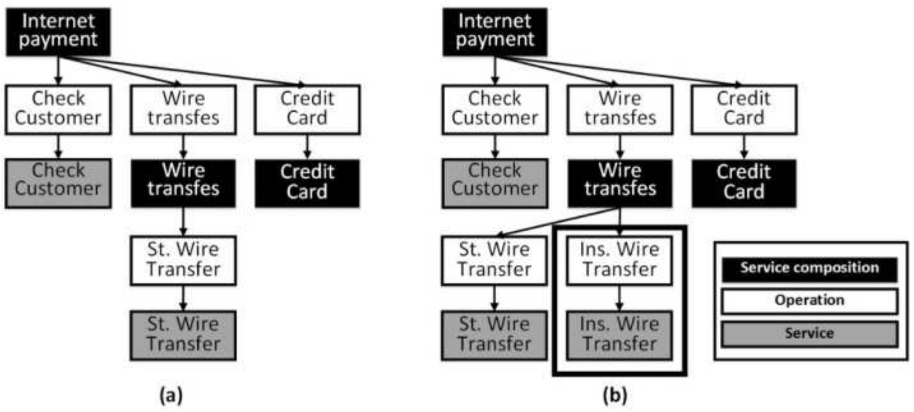


Fig. 3.3. The architecture of "internet payment" service composition: a) before evolution; b) after the first evolution step (added entities have been shown inside the red frame).

We consider three evolution steps in our example. In the first evolution step, "internet payment" service composition has to be extended by support for instant wire transfers. In the second evolution step, the performance requirement to handle 100 concurrent users has been increased to 200 users, and the architect decided that all the service operations for payment making have to support 200 transactions per second. In the third evolution step, the performance requirement to handle 200 concurrent users has been limited to 50 users (the number of active users has been limited and the use of high performance services has been very expensive). The architect decided that the number of supported transactions in case of all of the service operations for payment making have been limited to 50.

The service composition of "internet payment" has been modified only in the case of the first evolution step and has been illustrated in Fig. 3.3(b).

A model representing the evolution of the "internet payment" service composition is presented in Fig. 3.4.

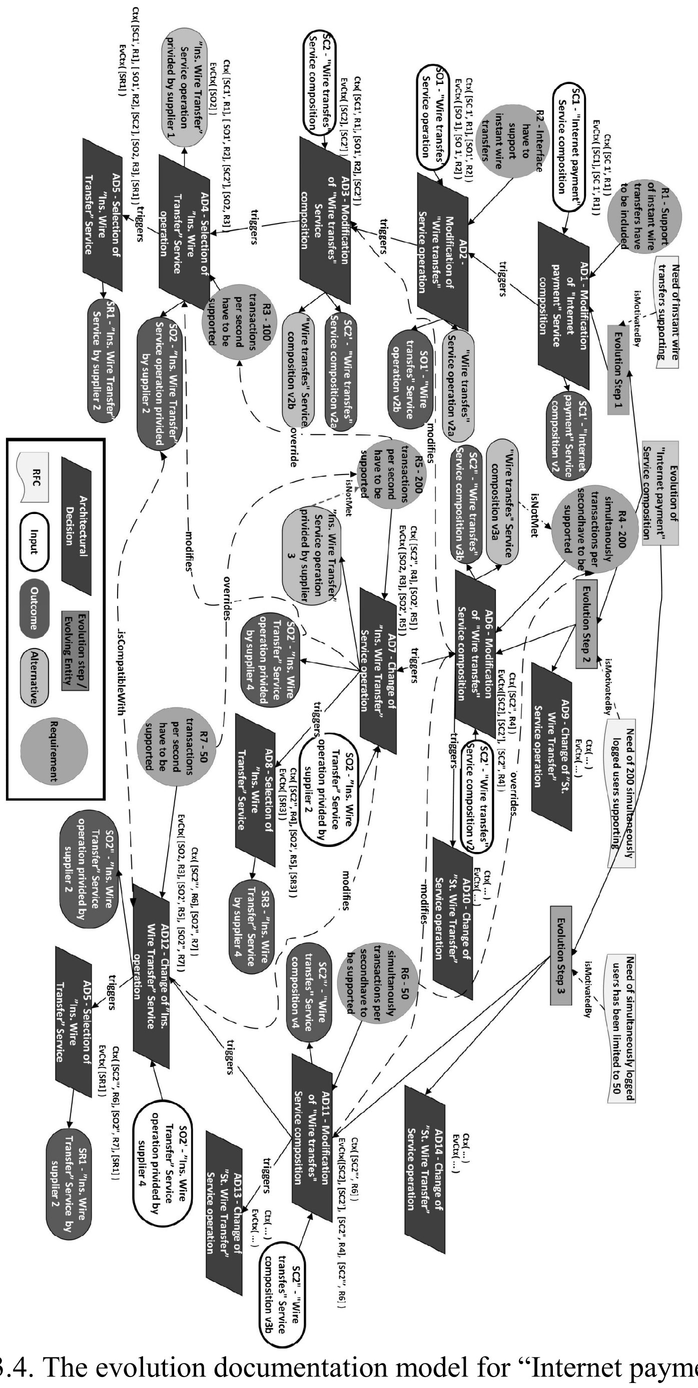


Fig. 3.4. The evolution documentation model for “Internet payment”.

The structure of decision-making of three evolution steps for our example has been presented in Fig. 3.4. We can observe that its hierarchical structure is reflected by the *triggers* relation. This allows for tracing the decision making-process, which is captured as the context, e.g. context for AD4 – "Selection of 'inst. Wire Transfer' Service Operation" looks as follows (compare Fig. 3.4): *Context (AD4) ≡ Ctx ([SC1', R1],[SO1', R2], [SC2'],[SO2, R3])*

Having more than one captured evolution step, we can observe the evolution of architectural decisions and their requirements, which is reflected by the modifies and overrides relations. The evolution of them is captured as evolution context, and allows for their evolution to be traced. An example of evolution context for AD12 – "Change of 'inst. Wire Transfer' Service Operation" looks like the follows (compare Fig. 3.4):

EvolutionContext (AD4) ≡EvCtx ([SO2, R3],[SO2', R5], [SC2'', R7])

Our model provides the mechanism for detecting reusable decisions. Let consider decision AD4 –"Selection of 'inst. Wire Transfer' Service Operation" and its modifications: AD7 and AD12 – "Change of 'inst. Wire Transfer' Service Operation" (compare Fig. 3.4 and section 3.2.3):

$$AD7\ modifies\ AD4\ \land AD12\ modifies\ AD7$$

AD4 had to meet only one requirement R3 – "100 transactions per second have to be supported", the modification of AD4 was forced by a change of this requirement to 200 transactions per second (by requirement R5), and then in the third evolution step was reduced in to 50 transactions per second (by requirement R7):

$$R5\ overrides\ R3\ \land R7\ overrides\ R5$$

Therefore using definition 15 (see section 3.2.2) we can conclude that outcome of AD4 is eligible for reuse as an outcome of AD12, and if $O_{AD12} isMet\ R3\ then$:

$$O_{AD4}\ isCompatibleWith\ AD12$$

It means that O_{AD4} can be used as an outcome of $AD12$ and that the sub-tree of AD4 can be used as a sub-tree of AD12 (see Fig. 3.4).

Decisions that could be reused as an outcome of AD12 are detected automatically; nevertheless, compliance with the requirements has to be verified manually. For example, if AD4 would be replaced with AD7, then AD4 will be incompatible with AD12 and could not be reused as an outcome of AD12.

Only in the case of an *isStronglyCompatibleWith* relation is the detection of reusable decisions done automatically.

3.4.Related Work and Discussion

The evolution of service-oriented systems can be documented as sets of architectural decisions, changed or newly made in order to design the required changes.

Textual approaches to document architectural decisions do not support evolution or only document chosen properties describing evolution [5], [6]. A similar situation is in the case of approaches based on relations between decisions [6], [7]. These limitations have already been observed and presented by the authors of [8]. Moreover, they made their own approach based on two types of attributes: annotated decisions to record the history and the status of a decision at a given time; and recorded relationships between design decisions and between design decisions and artefacts. In [11] authors proposed a meta-model for the decision model including additional attributes to manage the evolution of architecture design decisions. The most advanced approaches, such as [1], [2], are based on a diagrammatic representation of ADs. The model described in [1] includes the classification of ADs as defined by architect topic groups, four refinement levels [1], a set of relations between ADs and their components and decision-making models. However, this model is designed more for supporting decision-making during the construction of SOA systems.

Another diagrammatic approach is MAD 2.0 [2], which has been developed by our team. MAD 2.0 has been designed to support architect practitioners working on system evolution. It does not impose any predefined classification or hierarchy of architectural decisions and assumes a limited number of kinds of relations between architectural decisions. The extended version of MAD 2.0, including a definition of the context of a decision, has been presented in [10].

In [9], the authors propose documentation framework for ADs based on four viewpoints organising and categorising information on architectural decisions. The evolution of architectural decisions is represented by the Decision

Chronology viewpoint representing all the versions of every architectural decision.

The goal of our research was to develop models specialised for capturing the evolution of services and their compositions as a set of architectural decisions. Our model documents evolution as a set of evolution steps, each of these steps includes hierarchical structures of ADs reflecting the intuitive process of decision-making during the evolution of services and service composition. Documenting evolution steps and evolution of ADs is not possible in the case of MAD 2.0 [2] and the model presented in [1]. Moreover, our approach documents the evolution of the modelling artefacts (e.g. service compositions are captured as BPMN models), i.e. artefacts are changed as a result of architectural decisions. Linking architectural decisions with the artefacts that have been modified by them facilitates access and a comprehension of architectural knowledge and its reuse.

Detecting previous versions of architectural decisions that could be reused in a current evolution step, and tracing changes in architectural decisions and of the requirements related to them are new features. Reusing a previous version of the decision results in reusing all of the sub-decisions. A definition of requirements exists in MAD 2.0, but documenting and tracing requirements is not supported.

Our model is compatible with the viewpoints set out in [9]:

- “Decision Detail Viewpoint” is represented by architectural decisions and properties assigned to them, such as: alternatives, outcomes, inputs and requirements;
- “Decision Relationship Viewpoint” is represented by the hierarchical structure of architectural decisions, which is designated by the triggers relation;
- “Decision Stakeholder Involvement Viewpoint” – in the context of our model, the decision maker is understood as the software architect;
- “Decision Chronological Viewpoint” is represented by the modifies relation, which designates subsequent versions of architectural decisions;

3.5.Summary and Outlook

A novel model has been proposed for documenting the evolution of services and their composition based on capturing architectural decisions. Our approach includes a set of formally defined relations, context and evolution context, and provides the following features: mechanisms for tracing the evolution of architectural decisions; requirements and modified artefacts in subsequent evolution steps; tracing the decision-making process for a single evolution step; versioning evolved artefacts and mechanisms for auto-detecting previous versions of architectural decisions that could be reused. The concept has been illustrated on a real world example. The approach that has been proposed in this chapter will be included into the evolution methodology for service-oriented systems developed by our team. Therefore, the research outlook includes:

- The development and refinement of formal definitions and integrity constraints of the model;
- The development of a software tool supporting the model;
- Carrying out further and more extensive validation.

This work was sponsored by the Polish Ministry of Science and Higher Education under grant number 5321/B/T02/2010/39.

Bibliography

[1] Zimmermanna O., Koehlera J., Leymannb F., Polleya R., Schustera N.: Managing architectural decision models with dependency relations, integrity constraints, and production rules, Journal of Systems and Software, vol. 82, No. 8, pp. 1249-1267, 2009

[2] Zalewski A., Kijas S., D. Sokołowska.: Capturing Architecture Evolution with Maps of Architectural Decisions 2.0, ECSA 2011, Essen, Germany, Lecture Notes in Computer Science, vol. 6903, pp. 83-96, 2011.

[3] Bosch J., Jansen A.: Software Architecture as a Set of Architectural Design Decisions, 5thWorking IEEE/IFIP Conference on Software

Architecture (WICSA'05), pp. 109-120. IEEE Computer Society, 2005.

[4] Cardellini V., Casalicchio E., Grassi V., Iannucci S., Lo Presti F. and Mirandola R.: MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems, IEEE Transactions on software engineering, vol. 38, No. 5, Sep/Oct 2012.

[5] Tyree J., Akerman A.: Architecture Decisions: Demystifying Architecture, IEEE Software, vol. 22, No 2, pp. 19-27, 2005.

[6] Kruchten P., Lago P., van Vliet H.T.: Building up and Reasoning about Architectural Knowledge, QoSA2006, Springer-Verlag LNCS 4214, pp. 43-58, 2006.

[7] Wang K. Sherdil, Madhavji N.H.: ACCA: An Architecture-centric Concern Analysis Method, 5th IEEE/IFIP Working Conference on Software Architecture, 2005.

[8] Capilla R., Nava F., Tang A.: Attributes for Characterizing the Evolution of Architectural Design Decisions, Third International IEEE Workshop on Computing & Processing, pp. 15 – 22, 2007.

[9] van Heescha U., Avgerioua P., Hilliard R.: A documentation framework for architecture decisions, The Journal of Systems and Software, vol. 85 pp. 795–820, 2012

[10] Szlenk M., Zalewski A., Kijas S.: Modelling architectural decisions under changing requirements, Proceedings of the Joint 10th Working Conference on Software Architecture & 6th European Conference on Software Architecture, pp. 211-214. IEEE Computer Society, 2012

[11] Capilla R., Nava F., Dueñas J.C.: Modeling and Documenting the Evolution of Architectural Design Decisions, Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops, IEEE DL, 2007.

Chapter 4

Managing the adaptation of open-source software: the examples of BalticMuseums 2.0 and BalticMuseums 2.0 Plus

The acquisition of open-source software often includes a non-trivial adaptation of existing solutions. In this chapter we would like to provide some suggestions on how the process of adaptation should be managed based on our observations from development of two international projects: BalticMuseums 2.0 and BalticMuseums 2.0 Plus. We discuss an updated version of our framework for open-source software acquisition that gives due focus to the process of software adaptation and relate how this approach proved in real-world conditions during the development of the two aforementioned projects.

4.1.Introduction

With the growing popularity of open-source software [12, p. 895], and a variety of solutions available for many typical purposes, it becomes increasingly important to devise schemes that would provide guidelines for selection of software and improve efficiency of the OSS acquisition process.

What makes the issue complicated, is that often ready-made open-source solutions do not provide the exact functionalities the users require. As a result, some level of adaptation is needed in order to make the users satisfied.

This chapter is devoted to the problem of managing OSS acquisition when adaptation phase is expected. We propose an updated version of our own framework that provides guidelines on how to combine the adaptation with the other stages of acquisition, and how to perform them taking the adaptation into consideration. We describe the framework in the context of BalticMuseums 2.0 and BalticMuseums 2.0 Plus (later referred to as the “BM projects”), two international projects whose realization led to development of the framework, and provided examples of its use.

This chapter is organized as follows: we start with a short description of the BM projects, and then explain why it endorsed OSS solutions. The central section of this chapter describes in detail an updated version of the FEChADO framework [19]. Experiences from applying the framework within the subsequent BM projects are then described. Finally, conclusions are drawn.

4.2.BalticMuseums 2.0 and BalticMuseums 2.0 Plus projects

BalticMuseums 2.0 ("Joint development of cross-border information products for South Baltic Oceanographic Museums") and BalticMuseums 2.0 Plus ("Implementation of eGuides with cross-border shared content for South Baltic Oceanographic Museums") are international projects realized within the South Baltic Cross-border Co-operation Programme 2007-2013, and part-financed from the European Regional Development Fund [1].

The main objective of the BalticMuseums 2.0 project is the promotion and effective use of the natural heritage stored in the oceanographic museums by means of cross-border tourism information tools, in order to increase their attractiveness and competitiveness, especially for international tourists. The first among the detailed aims of the project is the development of a multilingual online platform, which enables a common presentation of tourist information by the museums – participants of the project [20, pp. 237-238].

The main objective of the BalticMuseums 2.0 Plus project is to develop multilingual content describing exhibits of the museums – partners of the project, and make it available for the tourists via multimedia eGuides. The first among the detailed aims of the project is the development of an effective content management system for storing and sharing various eGuide content [21].

The two projects are realized by an international consortium consisting of two scientific institutions – The University of Applied Sciences in Stralsund and the University of Szczecin, and four oceanographic museums – the German Oceanographic Museum in Stralsund, Gdynia Aquarium, Lithuanian Sea Museum in Klaipeda and the Museum of the World Ocean in Kaliningrad.

4.3. Reasons for using open-source software

Both BalticMuseums 2.0 and BalticMuseums 2.0 Plus projects contain significant components which require obtaining software suitable for specific needs. The use of open-source software was not imposed on the project consortium by the rules of the South Baltic program or any other institution, but it was assumed by all the consortium members as one of the core characteristics of the accepted approach since the very beginning of the first of the projects. There were various reasons for using open-source rather than proprietary software in BM projects. We can classify them in two dimensions: according to the time perspective (short or long-term), and according to the aspect (organizational, technological, and financial).

The short-term reasons pertain to the period of project development and the ability of attaining the appointed goals within planned time and budget. The long-term reasons refer to possible consequences after finalization of the projects, including long-term sustaining of the projects' results. The advantages expected by the BM projects partners from using open-source software in relation to different aspects are given in Table 4.1.

Table 4.1. The advantages expected from using open-source software.

Area	Advantages
Short-term	
Technological	High level of maturity: software developed for years High level of security: many testers involved, quick updates for identified weaknesses
Organizational	Active developers' and users' communities, available solutions for known problems
Financial	No license fees
Long-term	
Technological	Well-documented interfaces: easier combination with future systems
Organizational	Long-term sustainability: high chance that software will be updated No dependence on specific software supplier in case of future modifications
Financial	No paid upgrades needed

Source: own elaboration.

4.4.Methodology for open-source software adaptation

One of the principal advantages of open-source software is the availability of the source code and the right to modify it [8]. This right needs not be used if the obtained software fully meets the specified requirements. However, if any of the main requirements is not met, it becomes very useful, as it allows to modify the software so that it meets the missing requirements.

As open source software adaptation is a repeatable process, a framework can be defined and then applied to support its efficient execution. Quite surprisingly, even though there is a number of formalized approaches to the problem of selecting open source software (see Table 4.2 for examples), none of them addresses exactly the kind of problem that the managers of BM projects, among others, met: to acquire the most appropriate open-source solution considering it will still have to be adapted.

Although among 20 different approaches for open-source software evaluation listed by K.-J. Stol and M. A. Babar [17, pp. 390-391], there are at least two ([6],[10]) that take software developers perspective, they address the situation of building a new software system using open-source components rather than adapting an existing open-source software system for specific purposes.

Hence there was a number of open-source software systems acquired in the BM projects, most of which required adaptation (see Tables 4.3 and 4.4 further on), such framework developed naturally. It was formed and refined on the experiences of the BalticMuseums 2.0 project and applied in the BalticMuseums 2.0 Plus project.

We call it FEChADO, which is an acronym of the six steps of the proposed procedure for open-source software acquisition and adaptation [19]:

1. Find available solutions,
2. Evaluate solutions from the list,
3. Choose the most appropriate solution,
4. Adapt the solution,
5. Develop new modules,
6. Obtain users' feedback.

The respective stages will be described in the subsections to follow. Note that the description below may differ in details with the original presentation of the framework, as it contains later improvements; it may therefore be referred to as FEChADO 1.1, versus 1.0 presented in [19].

Table 4.2. Methods for selection of open source software

Method	Stages
QSOS by Atos Origin [11]	1. Define evaluation templates 2. Evaluate open source solutions 3. Qualify specific usage contexts 4. Select most relevant solutions
Context-Dependent Evaluation Methodology by M. Cabano, C. Monti, and G. Piancastelli [5]	1. Context analysis, which defines the necessities and requirements 2. Preliminary selection, which addresses the most critical metrics 3. Filtered selection, which estimates the remaining products by complete set of metrics
Open source software evaluation process by D. A. Wheeler [22]	1. Identify candidates 2. Read existing reviews 3. Compare the leading programs' attributes to the needs 4. Analyze the top candidates in more depth
Software Assessment Phases according to Business Reading Rating Model [4]	1. Quick assessment filter 2. Target usage assessment 3. Data collection & processing 4. Data translation
Evaluation through Prototyping by R. Carbon et al. [6]	1. Initial requirements analysis 2. OSS candidate selection 3. Iteration planning 4. Iterative prototype development 5. Final evaluation

Source: own elaboration.

4.4.1. Finding available solutions

The first stage consists of two phases: (1) the requirements have to be specified; (2) available solutions that supposedly match these requirements are listed. The initial list of suggested requirements may be based on characteristics of known software similar to that to be obtained. Regardless of whether there are examples of similar software, a brainstorming session involving both designers and representative users can be held to obtain the list.

The suggested requirements are then evaluated by a larger number of end-users, who can also propose new requirements at this stage. Either a CAPI (computer assisted personal interview) [2, pp. 136-139] or a CASI (computer-assisted self-interview) [14] technique of survey is proposed for this task, with the latter being faster, and the former allowing to obtain extra information and avoid misunderstandings.

The results of questionnaires are then analyzed in order to produce a list of requirements arranged in three groups: (a) core requirements (must be met), (b) additional requirements (should be met), (c) special requirements (some end-users believe they should be met). This list is once more presented to the end-users for correction and, possibly, re-evaluation.

Having obtained the list of requirements, we can proceed to the next phase: the search for candidate solutions that are supposed to match the requirements. The search should start from open-source project portals, such as sourceforge.net and freecode.com, and web search engines. Many search phrases should be examined, as different words may be used to describe equivalent functionalities.

The titles of software found first may also be used as search phrases in order to find competing solutions, or even reviews and feature comparison tables that may be very helpful also in the evaluation stage.

The search is discontinued after no more matching solutions can be found or the number of the candidate solutions exceeds an assumed threshold.

4.4.2. Evaluating found solutions

There are four phases in the evaluation stage: (1) Specifying evaluation criteria; (2) Preliminary evaluation; (3) Main evaluation; (4) In-depth evaluation.

The evaluation criteria are specified by assigning measurable fulfillment levels to the respective requirements. Only one fulfillment level must be defined for each criterion: *acceptable*; no solution failing to achieve this level should be chosen over one that achieves it. Additional levels can be defined, so that solutions that pass the *acceptable* level could be compared between each other. Their number may depend on requirements of the method used in the main evaluation phase.

Although the evaluation criteria may be given weights in this phase (if the weights are used at all), it is reasonable to postpone it until the main evaluation, as the procedure for obtaining the weights may be an element of the method used in the third phase.

The evaluation criteria should be arranged, depending on their measurability, into the following groups:

a) objective and easily measurable, which can be evaluated based only on software fact sheets, without need for testing demo versions or careful reading of documentation,
b) objective and not easily measurable, which can be evaluated only after careful reading of documentation or by testing demo versions, but without need for configuring the software as for intended use and without involvement of the end-users,
c) subjective, that can only be evaluated by questioning the end-users after letting them use the configured software.

The preliminary evaluation (phase two) has the goal to shorten the candidate list by removing software that does not attain the acceptable level for objective and easily measurable criteria based on the core requirements. The removed candidate solutions will not be considered anymore, so it is important that the source of information used at this stage is reliable.

The aim of the main evaluation phase is to obtain a limited number of solutions to be considered for the final choice.

First, the candidate solutions are evaluated by experts on all objective criteria based on the core requirements. Then, as in the previous phase, the list is pruned by removing solutions that do not attain the acceptable level.

Next, the list of candidates is arranged according to how well they meet the criteria, so that a short list (2-5 items) can be formed that will be considered further. Various methods may be used for this purpose, e.g., Hasse diagram [3], outranking methods (e.g. ELECTRE) [15, pp. 49-73], or hierarchy processes (e.g., AHP) [16].

The in-depth evaluation takes into consideration all criteria, including the subjective ones. Its scheme is akin to the main evaluation phase: the candidate solutions are first evaluated (yet this time by the end-users), the solutions failing to achieve the acceptable level are discarded, and the remaining ones are arranged, using a method of choice, into three ordered lists: one obtained taking into consideration only the criteria based on core requirements, the second – combined core and additional requirements, and the third – all requirements (i.e., including the special ones).

4.4.3. Choosing the most appropriate solution

The goal of stage 3 is to select a single solution that will be adapted and then adopted. There are only two phases in this stage: Discussion of evaluation results and Making the choice.

First, a meeting should be organized to discuss the evaluation results. The following stakeholders should take part in it: sponsors of the project (actual decision makers), members of the development team, external experts having knowledge of the software on the short list – especially if there are no people highly qualified in a specific software within the development team, and representatives of the end-users – especially those who took part in the in-depth evaluation phase.

The meeting should start with presentation of the evaluation results. Then, during the discussion, members of the respective groups should provide information important for making the choice, e.g.:

- the end-users who participated in the in-depth evaluation should point to drawbacks or special advantages of respective solutions,
- the invited experts should confirm, that the mentioned drawbacks are not due to wrong configuration or misusage,
- the members of the development team should declare if they are capable of fixing the mentioned drawbacks or adding specific advantages to candidate solutions lacking them, and estimate required resources,
- the sponsors of the project should declare the resources they are willing to contribute so that the limits of planned improvements can be defined for respective candidate solutions,

- the end-users should state which of the candidate solutions they would accept, either as they are, or after planned improvements.

The final decision on choosing the solution for adoption is made by the sponsors of the project. It should be based on the results of the evaluation process and the conclusions of the discussion.

4.4.4. Adapting the solution

There are three phases in the adaptation stage: (1) Acquiring the core of chosen solution, installing it with all the necessary prerequisites, and configuring it; (2) Acquiring, installing, and configuring the required add-ons; (3) Applying the planned code modifications.

Regarding phase 1 and 2, it is important to make sure that: (a) up-to-date versions of software are acquired and installed; (b) software is properly configured; (c) a due amount of tests is run before proceeding to the modification phase, so that problems discovered later can be attributed to the modifications rather than wrong configuration.

Every source code modification, regardless of its size (even if it consists of merely few new lines of code), should be explained in the technical documentation of the system, in terms of its scope, purpose, relation to other modifications, assumed conditions and possible risk factors. Regression tests are required to assure that the modification does not jeopardize stability or security of the system. If it is found to degrade system performance, an attempt to optimize the relevant code should be made.

Managing the adaptation does not end with the tests, but it continues for the entire lifespan of the system. As the modifications are applied to a certain version of the obtained software, they have to be reapplied whenever the original software is updated. A special procedure for this purpose must be defined in the technical documentation of the system, and it is convenient to prepare a script to automate the process.

Sometimes, the update of the original software may render the modifications non-applicable in their original form, as the updated source code may miss the context of the modification. This is why no update should be made to the modified software before checking if it is possible to apply the fix to the updated version. Of course, system backup should be done before every update so that reverting to the working version is possible.

If the modification cannot be applied or applying it to the updated system causes its malfunction, the problem should be investigated by an expert who should produce one of the following solutions:

a) decide that the modification is no longer needed as the issue it fixes is already solved in the updated version of the software;
b) provide a walk-around to apply the modification to the updated version of the software;
c) decide that the modification is still needed but cannot be applied as it is, and suggest, taking into consideration the benefits of updating the system, whether it is necessary to re-implement the modification and proceed with the update, or the current version of the system should still be used.

4.4.5. Developing new modules

Whenever the nature of the modifications and the architecture of the original system allows it, they should be developed as new modules. In this form, they are easier to apply, test, and less prone to compatibility problems in case of future patches of the system core.

In order to achieve seamless integration with the system, the new modules should be developed in full accordance with the guidelines defined in the system's technical documentation and respective API's.

If the modifications are of general nature, they could be of interest for other users. It is then advisable to contribute the new module to the open-source community, if the organization's internal regulations permit distribution of software developed by its employees as open source.

In case some of the modifications are suitable only for the intended system users, the new module can still be published, provided it is pruned from such elements. Depending on their kind and number, it may be accomplished by:

- turning these elements into a profile of module configuration settings,
- forking the module into internal (full) and external (limited) versions,

- developing an additional module, only for internal usage.

4.4.6. Obtaining users' feedback

The end-users' opinions on the adapted system should be gathered throughout its lifespan. A simple web form could be used to facilitate the feedback process.

All received opinions should be read by an appointed person, so that the management could be aware of the users' attitude to the system. Contrasting opinions should be resolved via discussion with the involved users. Bug reports and feature requests should be sorted into three groups:

- pertaining to the implemented modifications – should be passed to the internal development team;
- pertaining to the original system, but considered crucial for its use in the organization – should also be passed to the internal development team;
- pertaining to the original system, and not considered crucial for its use in the organization – should be passed to the original system developers.

4.5.Experiences from the BalticMuseums 2.0 project

The BalticMuseums 2.0 project resulted in development of five software systems, of which four were adaptations of open-source systems (see Table 4.3), the remaining one (the Panorama Manager [13]) being a closed-source solution as no open-source functional equivalent had been found.

Note that open-source software was also used in this project to set-up the system environment (*Debian, MySQL, PHP), support the management of the* project (Redmine, DokuWiki), and other purposes (e.g., GIMP), yet none of these was adapted in the sense described in this chapter, they were merely configured.

We shall use the first of the systems mentioned in Table 4.3 (the OIP) for an illustration of open-source software adaptation process. It was also the first system developed within the BM projects, so although we present it as an example of application of FEChADO, in reality it served as an inspiration and a model for the framework.

Table 4.3. Adaptations of open-source systems in the BalticMuseums 2.0 project

System	Based on	Scope of adaptation
Online Information Platform	Drupal	New website template developed, a few modifications applied to the source code of Drupal
Mobile Online Information Platform	Mobile Tools module for Drupal	New mobile website template developed
Kids' Zone	Drupal	New website template developed, multimedia-related scripts added
Online Ticketing System	Ubercart module for Drupal	New website template developed, multiple modifications applied to the source code of Ubercart module

Source: own elaboration.

First, a list of suggested requirements was assembled. The procedure started with a brainstorming session involving IT specialists and representatives of end-users. Then, a survey of similar systems was performed to enrich the list. Next, questionnaires based on the compiled list of requirements were presented at a meeting to the end-users, which resulted in several modifications of the requirements. As there are four museums involved in the BM projects, all principal decisions are made by consensus. After discussion and consultation, the four museums agreed on the proposed list of requirements.

Subsequently, using a pair-wise-comparison-based approach modeled according to the AHP method [16], the respective requirements were assigned weights by the representatives of end-users.

After it was decided that the OIP would be developed on the basis of a Content Management System, a list of 100 popular systems was prepared, based on various web sources, including Wikipedia.

The evaluation and choice stages differed somewhat from the scheme described in sections 3.2 and 3.3. The evaluation was wholly performed by a team of IT specialists with large experience in CMS configuration and administration. They filtered the candidate system list, leaving only those systems that could pass the requirements and were familiar to the team members. The latter criterion was motivated by the will to have at least one person in the development team that was skilled in configuration and administration of the chosen CMS. As a result, only three systems remained on the list: Drupal, Joomla!, and TYPO3.

The three systems were then evaluated on a set of criteria derived from the list of requirements. The points awarded in respective areas were summed and the systems were ranked according to the aggregate. Drupal topped the ranking and was approved by the project management for acquisition. There was no in-depth evaluation and no discussion.

Although the planned adaptation consisted only of developing a new website template, several requirements could not be met without several small modifications of the Drupal source code. Consequently, a relatively sophisticated update guide had to be prepared. The applied changes were, however, too scarce and too scattered to justify developing a new module.

The feedback obtained from the users revealed multiple flaws of the system. The most important complaints concerned low performance and the way content structure could be defined. The former was a consequence of the used template design approach and was solved by a complete reimplementation of the template, but it could only be identified no sooner than late test phase on the production server. The latter was a consequence of internal Drupal workings, and there was nothing that could be done about it at that stage of development, but it could have been identified earlier if an in-depth evaluation had been performed, and then it could have affected the choice of the system.

4.6.Experiences from the BalticMuseums 2.0 Plus project

In the BalticMuseums 2.0 Plus project, only two software systems were developed, both of which were adaptations of open-source software (see Table 4.4). Again, a number of open-source systems was also used to set-up the system environment*, support the management of the* project, and other purposes, without real adaptation.

Table 4.4. Adaptations in the BalticMuseums 2.0 Plus project

System	Based on	Scope of adaptation
eGuide Content Sharing System	ResourceSpace	Modifications applied to the source code of ResourceSpace
Photo Competition Management System	Drupal	New website template developed

Source: own elaboration.

We shall use the first of the systems mentioned in Table 4.4 (the eGCSS) as an example of applying FEChADO in its mature form to the open-source software adaptation process.

The work on the system started with a project meeting, at which first a brainstorming session was held, and then, through discussion and consultation, an initial list of suggested requirements was assembled. On the basis of this list a questionnaire was prepared and distributed to a larger number of end-users in a form of a computer-assisted self-interview. The survey results were processed to form a list of requirements, divided into core and additional.

In the first attempt, the development team tried to complete stages one and two of FEChADO using CMSmatrix. It is a web-based tool allowing to compare over 1200 content management systems, using 145 criteria grouped in ten categories: (1) system requirements, (2) security, (3) support, (4) ease of use, (5) performance, (6) management, (7) interoperability, (8) flexibility, (9) built-in applications, (10) commerce [7]. A mapping of the eGCSS requirements to these criteria was made [9, pp. 61-73], and then, a list of ten content management systems that ranked best, according to the chosen criteria, was obtained from CMSmatrix,.

A closer examination of these systems revealed that in reality none of them is suitable for the intended use without significant modifications. The failure can be attributed to the lack of relevant criteria in the CMSmatrix set, too general criteria in the CMSmatrix set (e.g., “multimedia management”), and different meaning of criteria in the CMSmatrix than in the system requirements specifications (despite similar names). Even though the mapping assigned respective requirements to CMSmatrix criteria best matching them, the analogy turned out to be too weak. Although CMSmatrix could be a useful tool for a CMS selection, it failed to provide support in the selection of a digital asset management system, in spite of it being a CMS of specific kind.

In the second attempt, the development team resorted to find candidate solutions using web search engines. Note that at the time of the searching, the topic of open-source digital asset management was not as well-researched as it is today, with valuable reviews available in the Internet (see, e.g., [18]).

Six objective criteria were defined based on the core requirements of the eGCSS. Every system found was examined on fulfilling them, so the phases of listing candidate solutions and preliminary evaluation were not separated. As a result, the obtained list contained only seven systems: ResourceSpace, Cyn.in, TYPO3 with DAM extension, OpenKM, Alfresco, NotreDAM, and EnterMedia.

The seven candidate solutions were then evaluated by experts on the six objective criteria based on the core requirements, and a ranking was produced using Hasse diagram [3]. In the next step, weights were assigned to the respective criteria, and a second ranking was produced, this time based on weighted scores.

Three top solutions were chosen for in-depth evaluation: ResourceSpace (which ranked first without considering the weights, and second considering them), OpenKM (which ranked first considering the weights), and TYPO3 (which ranked second without considering the weights).

After the three systems were installed and configured, the end-users were allowed to use them for the intended tasks – they uploaded multimedia resources, organized them into collections, added metadata, searched for them using various criteria, and downloaded them.

Surprisingly to the development team, the first impressions of all the three systems were very negative. The users complained on both the functionality and the user interface of the systems, each of them having flaws of its own. Because of such negative attitude, an approach based on the lesser evil principle was assumed: to choose the system that would require least adaptation to become acceptable for the end-users. The reported issues were therefore classified according to a scheme presented in Fig. 4.1.

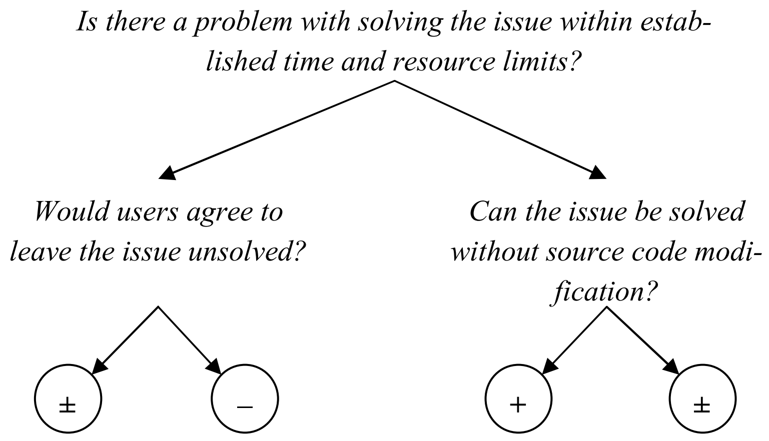

Is there a problem with solving the issue within established time and resource limits?
Would users agree to leave the issue unsolved?
Can the issue be solved without source code modification?
±
–
+
±

Fig. 4.1. Issue classification scheme

Source: own elaboration.

The in-depth evaluation lasted for a period of more than two months, during which, different systems were considered as the best choice. Eventually, ResourceSpace was selected, even though it differed from the requirements specification to a comparatively large extent. Its flaws, however, were considered either negligible or relatively easy to fix. The choice was made after discussion at a project meeting.

The problems with ResourceSpace were serious, as it lacked several requirements from the specification, e.g. content hierarchy, granting users access rights depending on the content's institutional owner, and allowing specific users to edit resource metadata while forbidding them to delete that resource.

After profound investigation of ResourceSpace functionalities, such as themes, metadata-dependent access rights and search filters, it was found possible to emulate the missing functionalities using those that were available, except for the last issue, which was solved by source code modification.

The adapted system was moved to the production server and opened for normal use. As a larger number of end-users got into contact with the system, many complaints were received, most of which caused merely by misunderstanding of how the system should be used. It led to an additional training, and preparation of a shorter, more comprehensible version of the system manual.

There were, however, some technical issues reported. For instance, although unprivileged users were unable to delete other users' content (after the fix described earlier), they were still able to remove it from other users' collections, leaving it in the repository, but rendering virtually invisible (solving it required further modification of the system source code).

Due to the changes in the source code, a system update guide was written that included instructions on how to reapply the fixes after updating ResourceSpace. So far, it has caused no maintenance problems.

4.7.Conclusions

Adaptation should be considered as an important element of the OSS acquisition. The framework described in this chapter can be very useful for the project management as a scheme for carrying out the acquisition process with due attention paid to the opportunities of software adaptation.

The framework is flexible, and the proposed stages of the procedure should not be treated as required, but as suggested. They can be combined or even skipped, if the circumstances permit.

The two provided examples of use, respectively from the early and final stages of the BM projects (as well as the framework) development, emphasize the consequences of improper selection of candidate solutions, significance of the in-depth evaluation phase, and how the possibility of adaptation could be taken into consideration during evaluation, as well as the importance of gathering and reacting to the users' opinions.

Acknowledgements

This chapter was written with the financial support from the European Union.

Bibliography

[1] BalticMuseums 2.0 & 2.0 Plus Projects Website, http://www.balticmuseums.org, 2013 [accessed 15.04.2013].

[2] Beam G.: The Problem with Survey Research, Transaction Publishers, New Brunswick, 2012, pp. 136-139.

[3] Brüggemann R., Halfon E.: Theoretical base of the program “Hasse”, GSF, Neuherberg, 1995.

[4] Business Readiness Rating for Open Source, http://docencia.etsit.urjc.es/moodle/file.php/125/OpenBRR_Whitepa per.pdf, 2005 [accessed 15.04.2013].

[5] Cabano, M., Monti, C., Piancastelli, G., Context-Dependent Evaluation Methodology for Open Source Software in J. Feller, B. Fitzgerald, W. Scacchi, A. Sillitti (eds.) Open Source Development, Adoption and Innovation, Springer, New York, 2007, pp. 301–306

[6] Carbon R., Ciolkowski M., Heidrich J., John I., Muthig D.: Evaluating OpenSource Software through Prototyping, in: St.Amant K., Still B. (eds.), Handbook of Research on Open Source Software: Technological, Economic, and Social Perspectives, Information Science Reference, Hershey/New York, 2007, pp. 269-281.

[7] CMSmatrix, http://www.cmsmatrix.org, 2012 [accessed 7.02.2013].

[8] Free Software / Open Source: Information Society Opportunities for Europe?, Working group on Libre Software, http://eu.conecta.it/paper/paper.html, 2000 [accessed 15.04.2013].

[9] Komorowski T.: Wspomaganie podejmowania decyzji w zakresie wyboru systemu zarządzania dokumentami (CMS/DMS), „Studies & Proceedings of Polish Association for Knowledge Management” 56, 2011, pp. 61-73 [in Polish].

[10] Majchrowski A., Deprez J.: An operational approach for selecting open sourcecomponents in a software development project, in: O’Connor R., Baddoo N., Smolander K., Messnarz R., Software Process Improvement, Springer-Verlag, Berlin/Heidelberg, 2008, pp. 176-188.

[11] Method for Qualification and Selection of Open Source software (QSOS) version 1.6, Atos Origin, 2006, http://master.libresoft.es/sites/default/files/Materiales_MSWL_2010_2011/Project%20Evaluation/materiales/qsos-1.6-en.pdf, [accessed 15.04.2013]

[12] Midha V., Palvia P., Factors affecting the success of Open Source Software, “Journal of Systems and Software” 85(4), 2012, pp. 895-905.

[13] Miluniec A., Drążek Z., Komorowski T., Muszyńska K., Swacha J.: A novel approach to panoramic gallery management on the example of Balticmuseums 2.0 website. Forthcoming.

[14] Olsen R., Sheets C., Computer-Assisted Self-Interviewing (CASI), in: Lavrakas P. J. (ed.), Encyclopedia of Survey Research Methods , SAGE Publications, Thousand Oaks, 2008.

[15] Roy B.: The outranking approach and the foundations of Electre methods, “Theory and decision” 31, 1991, pp. 49-73.

[16] Saaty T. L.: The Analytic Hierarchy Process, McGraw-Hill, New York, 1980.

[17] Stol K.-J., Babar M. A.: A Comparison Framework for Open Source Software Evaluation Methods, in: Ågerfalk P., Boldyreff C., González-Barahona J. M., Madey G. R., J. Noll (eds.), Open Source Software: New Horizons, Springer, Notre Dame 2010, pp. 389-394.

[18] Sarwan N.: Review of Available Open Source DAM Software, http://www.opensourcedigitalassetmanagement.org/reviews/available-open-source-dam, 2013 [accessed 15.04.2013].

[19] Swacha J., Muszyńska K., Drążek Z.: An outline of development process framework for software based on open-source components, Proceedings of the 14th International Conference on Enterprise Information Systems, vol. 2, SciTePress, 2012, pp. 183-186

[20] Swacha J., Muszyńska K., Komorowski T., Drążek Z.: Development and maintenance of a multi-lingual e-Tourism website on the example of BalticMuseums 2.0 Online Information Platform, “Information Management”, 3, 2011, pp. 237-246.

[21] Swacha J.: Koncepcja systemu współdzielenia treści dla elektronicznych przewodników na przykładzie projektu BalticMuseums 2.0 Plus, „Studies & Proceedings of Polish Association for Knowledge Management” 56, 2011, pp. 207-217 [in Polish].

[22] Wheeler, D. A., How to Evaluate Open Source Software / Free Software (OSS/FS) Programs, http://www.dwheeler.com/oss_fs_eval.html [accessed 15.04.2013.

Chapter 5

Software for eScience: from feature modeling to automatic setup of environments

To increase our productivity when setting up various software environments, we try to reduce the complexity of configuration tasks by managing components at different levels of abstraction and by automating the process. This is particularly important (in terms of performance) when the configuration is not the direct objective of our activity. When deploying environments for eScience applications the researcher's main interest lies in executing experiments and obtaining results, not in tedious fine-tuning of the computational platform itself. Tackling the challenge of automatically setting up environments for *in-silico* experiments is the main motivation behind the discussion presented in this work. When facing such a task, clear representation and processing of component dependencies poses a challenge. In this work the Feature Model notation, popular in the Software Product Line methodology and successfully applied to configuration modeling, is examined for this purpose. This chapter presents a feasibility study of applying the Feature Model to develop tools for automatic environment configuration using a prototype implementation. The presented discussion has led the authors to further extend this idea, covering a wider range of applications. The chapter describes the architecture of an extensible framework automating various deployment and component installation tasks based on the Feature Model.

5.1.Introduction

eScience [1, pp. 93-40] is fast becoming a popular approach to scientific research. However, new research paradigms such as simulation and data intensive processing bring new challenges. Preparing an eScience application execution environment is a complex and time-consuming process, frequently requiring configuration of numerous cooperating components. Such components may include applications and datasets; hence their deployment requires extensive knowledge in the area of OS administration, cloud platforms, communication protocols and others.

Following analysis of requirements and review of available technologies it seems clear that there is need for a tool which would enable selection of eScience application prerequisites in a simple and intuitive way, and then deploy them in a given environment. The tool presented here combines the use of the *Feature Model* for modeling the component domain with a *Provisioning* system for application deployment. The *Feature Model* is a representation of product feature relationships, known for its broad use in science and industry, including *Software Product Lines*.

The objective of this chapter is to present a feasibility study of applying the *Feature Model* to modeling eScience application component dependencies, implemented as a tool for automatic deployment of execution environments. The architecture described in Section 5.2 was implemented as a prototype system using tools and libraries presented in Section 5.3 and then evaluated using a case study (Section 5.4). Based on this evaluation a generalization of the idea is proposed (Section 5.5). We also try to address the following question: how appropriate is the presented approach for configuring environments for a wider range of applications (not limited to eScience), with a broader spectrum of installation methods (not only *Provisioning Tools*) Validation of the architectural concept results in the design of a expansible software production line framework which better fulfills the presented evaluation criteria. Related work is presented in Section 5.6.

5.2.Description of the proposed solution

eScience applications consist of many components, often using separate technology stacks. In the framework of the VPH-Share project [11] examples of such applications include @neurIST [16] (simulation of brain aneurisms), euHeart [21] (human heart simulations), Virolab [5, p. 8] (virtual virological laboratory) and VPHOP [28] (prediction of osteoporotic bone fracture risk). Here, we deal with components being a process of an operating system, applications deployed in application containers, interpreted code written in various interpreted programming languages and various types of databases, each of which needs to reside on a virtual machine being a part of an execution environment. The wide variety of components calls for a generic scripting approach for installation of prerequisites. We assume that each component is associated with an installation unit comprised of deployment scripts and a set of configurable attributes. The user selects components which form the deployment configuration. This approach gives flexibility while also introducing the need to cope with component dependencies, analysis and visualization. Achievements of the *Software Product Line* appear to be particularly helpful for this purpose. The *Feature Model* – successfully applied in the *Software Product Line* – seems to match the gap between configuration element domain modeling and execution environment instantiation mechanisms.

The *Feature Model* [4, p. 3] is a notation used to define a domain of objects as a set of features. By building a tree-like hierarchy (parent-child relationships) and defining types of sibling relationships (and, or, xor), it organizes dependencies and helps to identify commonalities and variabilities [1, p. 7]. It allows also for representing dependencies which do not fit into a hierarchical model (cross-tree constraints) and, in some variants, defining even more complex requirements (extended Feature Models) [3, p. 3]. By using a simple mapping to well-known decision problems, the *Feature Model* is well suited for representing dependency logic [10, p. 3] (it is understandable to a computer). There are various implementations of *Feature Model* operations which simplify configuration processes, enable automatic completion of decisions and support error detection [3, p. 16]. Owing to the simplicity of the graphical *Feature Model* representation and its hierarchical construction (which allows for reducing model complexity), it is also understandable to a human and well suited to visualization. Furthermore there are ready-to-use *Feature Model* visualization tools.

The *Software Product Line* (SPL) [1, pp. 4-6] is a group of methods which describe the process of organizing software creation in a way that allows for increase the reusability of artifacts and leads to partial automation of the software product creation. The problem addressed by SPL methods has much in common with automation of eScience application deployment. In the process of building a production line it becomes necessary to map the product feature model to the architecture of a production line and to define product instantiation methods. However, product instantiation is one of the less frequently studied activities in the domain of software product lines [5, p. 1]. Therefore, an interesting challenge is to examine the advantages and drawbacks of the selected mapping method as well as the mechanism of eScience application instantiation using a prototype tool.

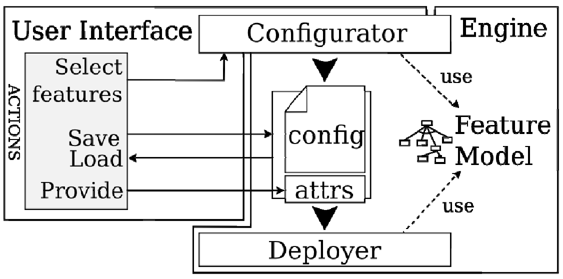


Fig. 5.1. Tool architecture. Model configuration links *Configurator* and *Deployer* modules.

The architecture of a tool which implements the presented features comprises two main components, presented in Figure 5.1. The *Configurator* is a module associated with the user interface, allowing for stepwise selection of *Feature Model* elements. The system automatically eliminates potentially conflicting decisions from the decision space. The selected *Feature Model* elements are supplemented by a set of attributes and passed to the *Deployer* module. The *Deployer*'s task is to deploy the configuration on a dedicated experiment execution machine. The *Deployer* validates the configuration and creates an acyclic dependency graph based on relationships defined in the *Feature Model*. This graph is sorted topologically to obtain the order of installation. It should be emphasized that each feature in the model is mapped to at most one installation unit and therefore each model element represents either an installation unit or a group of other features.

5.3.Choice of technology

In order to choose an appropriate approach to automatic configuration of the execution environment a study of the available technological solutions has been performed [14, pp. 1-32]. Three classes of tools were taken into account – *Distributed Shell, Unattended Installation* and *Provisioning Tools*. Evaluation was influenced by factors such as applicability in private cloud infrastructures, capability for simultaneous configuration of multiple Virtual Machine instances and redeployment potential. Given such criteria the most promising solutions appear to belong to the group of *Provisioning Tools*. That is why a provisioning tool was chosen to support the presented prototype. The suitability of four popular provisioning tools was evaluated: Bcfg2 [17], CFEngine [18], Chef [19], Puppet [24]. Comparison criteria included ease of integration, availability of ready-to-use installation packages, essential compatibility with the Java Enterprise Edition technological stack, support for various operating systems and type of license (relevant for the VPH-Share project). As each of the reviewed tools represents a slightly different approach and each might be useful in performing specific tasks, it is hard to compare them directly. Nevertheless the Chef platform seems to be the best fit for the selected criteria due to its full support for Windows, a straightforward Java API and a large user community providing ready-to-use installation scripts. Using Chef consists of two main tasks: maintaining an installation unit repository and performing deployment. The Chef repository is comprised of so-called *cookbooks:* packages containing scripts and requisites needed to perform installation. Deployment is performed by Chef on the basis of an ordered list of cookbooks with associated attributes.

We have also evaluated tools for automatic analysis of dependencies represented by the *Feature Model*. The use of libraries implementing various *Feature Model* operations enables auto-completion of configuration decisions, conflict detection etc. [2, p. 16]. The reviewed libraries (SAT4J [25], JavaBDD [23], Choco [20], AHEAD [15], FaMa [22], SPLAR [26]) implement the above mentioned tasks at different levels of abstraction and perform model operations on the basis of two fundamental Feature Model representation mappings – SAT (Boolean Satisfiability Problem) or BDD (Binary Decision Diagram) trees [9, p. 3]. As there are differences with regard to processing efficiency and memory utilization depending on the choice of data structure, the BDD-based approach was chosen due to its better performance in operations associated with interactive configuration of a single model. The prototype implementation bases on the SPLAR library due to its ease of use and the ability to reuse visualization code fragments of a related open-source tool – SPLOT [26].

5.4.Tool and architecture evaluation

The presented tool, called *Cloudberries*, was evaluated in a case study. It was integrated with the web portal as well as with the private cloud infrastructure of the VPH-Share project and validated using the euHeart [21] application as a deployment testbed. The structure of euHeart allowed us to completely automate its deployment. The environment configuration process consists of twelve steps, including locating files in the target operating system, creating user accounts, granting user rights, etc. Although the process is quite simple, configuration steps require some administrative knowledge. Time savings depend on individual skill and are hard to measure directly. Most deployment steps can be easily expressed as cookbooks (Chef installation units) and mapped to Feature Model elements. Cookbooks were created for such components as python, python-pip, xvfb, wine, libxp6, openjdk-6-jdk, python-dev and libxslt1-dev. Procedures specific to euHeart installation were collected in a separate cookbook.

In the course of evaluating the tool we came to some conclusions representing the common ground between implementation and architectural design. Using the current mechanism of system extension (based on cookbooks) it is almost impossible to automate creation of a new Virtual Machine instance. This is due to several limitations which cannot easily be bypassed. One of the most significant problems is that all Chef deployment scripts are invoked on an already-existing machine available via the SSH protocol. Although Chef can be integrated with a cloud stack, this is not a procedure which can be performed by the user on their own, e.g. by providing an implementation for a new feature. Even if VM instantiation could be represented as an installation unit, passing attributes between installation units would remain an issue. Some attributes, such as the IP address of the machine newly allocated by the cloud hypervisor, cannot be defined by the user and have to be produced by the system.

Although the presented architecture is performing its function well, it has some shortcomings. First, the installation scheduling process is imperfect as it is based on the feature dependency model. Rules which govern coexistence of components in a single environment are not necessarily connected with the order of their deployment in a single *product* (deployed application). In particular, a situation in which product feature dependencies form a cycle should be taken into account. Such cycles prevent us from considering dependency relationships as a chronological order of deployment. Secondly, in the current situation there is no way to define a flow of attributes between installation units. An installation unit should be allowed to base its behavior on the product of the previous unit. Moreover, the architecture does not allow for sharing attributes and, consequently, the same attributes have to be specified multiple times for different installation units. Finally, all of the installation units are subordinated to a tool which manages their sequential execution (Chef) – this limits potential avenues of expansion to mechanisms built into that tool.

By limiting the model-architecture mapping to the feature-component correspondence, we lose the flexibility of defining product features inherent in classic approaches to feature-based domain analysis [9, pp. 35-39]. However, thanks to automatic mapping between features and installation rules, we gain the flexibility of expanding our production line, which is clearly an advantage of the presented approach. A question now arises: how to modify the architecture to gain compromise

5.5.Research result: a refined architecture

Overcoming the drawbacks presented in the previous section would allow us to broaden the range of supported applications and create a more generic solution. As shown in Figure 5.2.A., the proposed refined architecture is composed of a dynamically modifiable, layered core, which includes a production line, a user interface and modules managing its lifecycle.

Feature Layer dependencies define sets of features which may constitute a correct product configuration. The Feature Model validation is more complex due to the influence of installation dependencies (*Deployment Layer*). An element of the Feature Model is treated as a feature of product configuration and a representation of an installation unit at the same time. Each installation unit provides an implementation of the feature instantiation as a plug-in for the *Provider Layer.* Installation may be implemented as arbitrary behavior using a chosen programming language. Within the *Deployment Layer* the installation dependency graph is defined as a set of nodes (elements of the Feature Model and so-called *ports*), along with installation dependency edges. The graph defines the order of executing installation units representing elements of Feature Model while ports define types of information transported between them (Figure 5.2.B.). Each port is either an attribute transmitted between installation units or a production line state. Each port has a type, zero or more producers and zero or more consumers. Each element of the Feature Model may have zero or more input ports and zero or more outputs (Figure 5.2.B). An input port is an attribute provided prior to unit installation. An output port defines an attribute produced by the unit. A port without a producer is processed as an attribute provided by the system user. Each producer of a single port has to be part of a different configuration: in a single cycle of installation each port is produced only once, with the exception of production line states (these may have multiple producers invoked in an unspecified order). The *Provider Layer* is where the implementation of installation units is provided in the form of plug-ins (e.g. invocation of Chef). Each element of the Feature Model is associated with a single *provider*. Each *provider* has to satisfy the interface defined by input and output ports in the *Deployment Layer.*

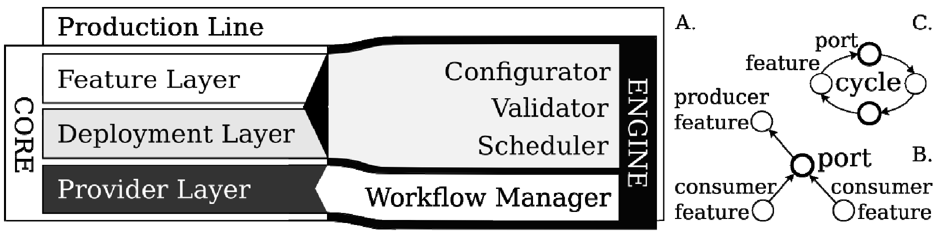


Fig. 5.2. Refined architecture – production line framework (A). Fragments of an installation dependency graph – passing a parameter between feature installation units via a so-called *port* (B); cyclical dependency (C).

These three layers allow for dynamic expansion of the production line and are complemented by several other modules. The *Product configurator:* the Feature Model configuration panel for the end user. The user selects components, defines the required attributes and instantiates the product. The *Model configurator:* an administration panel used for production line expansion. It supports modification of Feature Model and installation dependency graph as well as installing *provider* plug-ins. The *Validator:* a module for assessing the correctness of the Feature Model. Validation is performed on the basis of an installation dependency graph defined in the *Deployment Layer*. The *Validator* seeks cycles in the graph (Figure 5.2.C.) and checks if all the features belonging to each cycle also belong to any correct model configuration. The complexity of this operation is dependent on the complexity of cycle detection and validation of partial model configurations (dependent on implementation of the Feature Model operations). If a model configuration containing all of the cycle elements exists, feature selection has not been performed properly and the feature model is incorrect. The *Validator* also validates the dependency graph by analyzing nodes which do not represent a production line state. Each node must have no more than a single producer in a single model configuration. The *Scheduler:* a module scheduling an installation order on the basis of dependencies in the *Deployment Layer.* In order to create a schedule the *Scheduler* performs topological sorting of dependency subgraph which is limited to the current configuration. The *Workflow manager:* the module responsible for managing running installations and passing attributes between units.

Defining an installation dependency graph clearly introduces some overhead in the process of creating new Feature Model elements. However, scanning for configurations containing elements whose installation procedures cannot be ordered chronologically is necessary to ensure the correctness of the model. Therefore, introducing this additional formal description allows us to reduce the likelihood of errors in modeling the configuration domain.

5.6.Related work

This section presents some different approaches to mapping the Feature Model to product line architectures and product instantiation. In [13] automatic creation of Java Enterprise Edition family applications is described. Configuration files used by the base production line components (Spring framework Object Factory configuration, application container Deployment Descriptor) are generated on the basis of a model configuration. The authors propose a solution tightly coupled with a specific technology stack and much less generic than the one presented in this chapter. The instantiation process is partially manual, which also differs from our solution. Nevertheless, an interesting aspect of this approach is the mechanism of component state probes enabling monitoring of installation progress and application lifecycle. The study described in [7] presents an approach to automatic creation of applications based on aspect programming. Features are mapped to so-called Object Teams – modules grouping sets of classes whose behavior can vary depending on the aspect. The solution is limited to developing software which is subsequently compiled. One idea worth pursuing in future work seems to be the substitution of behavior on the basis of requirements which are represented as features. Much like the presented work this study requires defining attributes. The authors of [8] present some ideas which may drive further research. In their paper Feature Model elements are also mapped to components, however each component consists of a set of states and a definition of its inner architecture, expressed as a Feature Model. Component construction may vary between revisions. It is worth noticing that dependencies between component states are considered at the Feature Model level, which may be a good alternative to the dependency graph concept described in Section 4 of this chapter. The FArM system presented in [12] aims to provide a transformed FM where each feature can be implemented in an architectural component. At the beginning of the process model elements are described by additional semantics (e.g. Quality Features) which enable the system to choose an appropriate approach to treating each individual feature. The notion of the *provider* presented in this work (Section 4) can be adapted to specify any semantics within the rules of production line operation.

5.7.Conclusions and future work

Although the tool presented in this work is a prototype, it has proven successful in the realization of its assigned task. Cloudberries links the benefits of software provisioning with the specific requirements of the scientific community, increasing the productivity of researchers. Moreover, as shown in this chapter, eliminating some architecture shortcoming allows us to design a generic framework of an expansible production line with a much wider range of applications. The developed solution constitutes an approach to mapping the Feature Model to a production-line architecture, somewhat different than those presented in other publications. Careful comparison and evaluation of its benefits will be the subject of further work. Planned research will focus on the selection on the best approach to modeling dependencies and automatic compilation of applications comprised of different types of components.

Acknowledgement

The work is partially based on an M.Sc. thesis [14] and was partially funded by the EC ICT VPH-Share Project [27] and the corresponding KI AGH grant (contract no. 269978).

References

[1] Belloum A., Inda M. A., Vasunin D., Korkhov V., Zhao Z., Rauwerda H, Breit T. M., Bubak M., Hertzberger L. O.: Collaborative e-Science Experiments and Scientific Workflows, IEEE Internet Computing, Volume:15, Issue: 4, DOI: 10.1109/MIC.2011.87, pp. 39 - 47, 2011.

[2] Benavides D.: On the automated analysis of Software Product Lines using Feature Models. A framework for developing automated tool support, PhD thesis, Department of Computer Languages and Systems, ETSI Informática, University of Seville, Spain, 2007.

[3] Benavides D., Segura S., Ruiz-Cortés A.: Automated analysis of feature models 20 years later: A literature review, Information Systems, Volume 35 Issue 6, DOI: 10.1016/j.is.2010.01.001, pp. 615-636, 2010 .

[4] Benavides D., Trinidad P., Ruiz-Cort´s A.: Automated Reasoning on Feature Models, Advanced Information Systems Engineering: 17th International Conference, Proceedings, DOI: 10.1007/11431855_34, pp. 491-503, 2005 .

[5] Bosch J., Högström M.: Product Instantiation in Software Product Lines: A Case Study, Second International Symposium on Generative and Component-based Software Engineering, DOI: 10.1007/3-540-44815-2_11, pp. 149-163, 2000.

[6] Bubak M., Gubala T., Kasztelnik M., Malawski M.: Building Collaborative Applications for System-level Science, Advances in Parallel Computing, Vol. 18, 2009, High Speed and Large Scale Scientific Computing, DOI: 10.3233/978-1-60750-073-5-299, 2009.

[7] Hundt C., Mehner K., Pfeiffer C., Sokenou D.: Improving Alignment of Crosscutting Features with Code in Product Line Engineering, Journal of Object Technology, Volume 6, Number 9, pp. 416-436, 2007.

[8] Jansen S., Brinkkemper S.: Modelling Deployment using Feature Descriptions and State Models for Component-Based Software Product Families, Proceedings of the Third international working conference on Component Deployment, DOI: 10.1007/11590712_10, pp. 119-133, 2005.

[9] Kang K., Cohen S., Hess J., Novak W., Peterson A., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-90-TR-021, 1990.

[10] Mendonça, M.: Efficient Reasoning Techniques for Large Scale Feature Models, PhD thesis, School of Computer Science, University of Waterloo, 2009 .

[11] Nowakowski P., Bartynski T., Gubala T., Harezlak D., Kasztelnik M., Malawski M., Meizner J., Bubak M.: Cloud Platform for Medical Applications, eScience, 2012.

[12] Sochos P., Riebisch M., Philippow I., The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines, Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, DOI: 10.1109/ECBS.2006.69, pp. 308-318, 2006.

[13] White J., Schmidt D. C., Czarnecki K., Wienands C., Lenz G., Wuchner E., Fiege L.: Automated Model-based Configuration of Enterprise Java Applications, 11th IEEE International Enterprise Distributed Object Computing Conference, DOI: 10.1109/EDOC.2007.22, 2007.

[14] Wilk B.: Installation of complex e-Science applications on heterogeneous cloud infrastructures, MSc thesis, Faculty of Electrical Engi-

neering, Automatics Computer Science and Electronics, AGH Kraków, 2012.
[15] AHEAD: http://www.cs.utexas.edu/~schwartz/ATS.
[16] @neurIST: http://www.aneurist.org.
[17] Bcfg2: http://trac.mcs.anl.gov/projects/bcfg2 .
[18] CFEngine: http://cfengine.com.
[19] Chef: http://www.opscode.com/chef.
[20] Choco: http://www.emn.fr/z-info/choco-solver.
[21] EuHeart: http://www.euheart.eu.
[22] FaMa: http://isa.us.es/fama.
[23] JavaBDD: http://javabdd.sourceforge.net.
[24] Puppet: http://docs.puppetlabs.com.
[25] SAT4J: http://www.sat4j.org.
[26] SPLAR, SPLOT: http://www.splot-research.org.
[27] VPH-Share: http://www.vph-share.eu.
[28] VPHOP: http://www.vphop.eu.

Chapter 6
PlanICS 2.0 – a web service composition system

Distributed web services with well-defined interfaces enable building complex functionalities from simpler ones. An automatic web service composition prepares an execution plan specifying how to reach a given goal, fitting the services together and choosing an optimal provider for each required service type. PlanICS 2.0 is a web service composition system implementing our original approach aimed at providing flexibility at the level of modelling the reality in which the web services operate, and enabling to handle the services that do not publish their internal semantics, but communicate only by simple query/answer entries. PlanICS 2.0 separates between an abstract and a concrete planning phase, where the former deals with service types while the latter with their concrete instances, thus making the matching more efficient. Another distinguishing feature of the system consists in defining a computation engine as an independent block, which enables to compute plans using any suitable approach. Currently, two engines, based on a genetic algorithm and an SMT-solver, have been implemented. This chapter presents PlanICS 2.0 at a general level, comparing it also to related solutions from the area of automated web service composition.

6.1.Introduction

Automatic composition of web services [2, 1, 12] is a relatively fresh research area, gaining momentum as a web service-based infrastructure is becoming more and more popular. The problems to be solved are of very broad scope: syntactic matching of different description languages and approaches, dealing with semantic differences, high complexity associated with a large number of distributed services, various formulations of goals to be reached, etc. PlanICS 2.0 is a system implementing our original approach which solves the composition problem in some clearly separated stages. Fig. 6.1. shows the general PlanICS 2.0 architecture. The information about the services is stored in the following way: an ontology, managed by the ontology provider, contains a system of classes describing the types of the services as well as the types of the objects they process, while the service registry keeps an evidence of real-world web services, registered accordingly to the service type system. PlanICS 2.0 uses a state-based approach, which means that there are states (worlds) representing (partial) 'snapshots' of the reality, and services transforming them by modifying object attributes and adding new objects. Composition is thus understood as searching for a set of services capable to process certain states in a desired way.

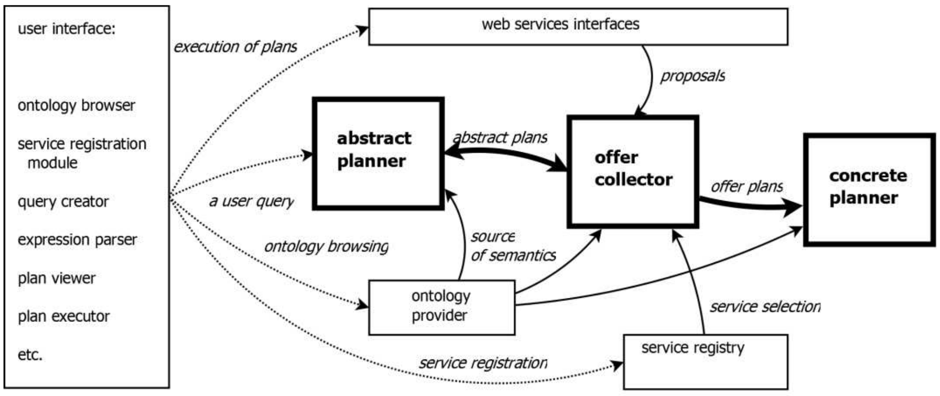


Fig. 6.1. A diagram of PlanICS 2.0 system architecture. The bold arrows correspond to computation of a plan, the thin arrows model the planner infrastructure, the dotted arrows represent the user interaction.

The user expresses a goal by a query, referring to objects and adding constraints, and defining an initial world to start with and an expected world to be reached. The system searches for a service composition transforming a subset of the initial world into a superset of the expected world. The latter, obtained by executing services according to a plan, is called a final world.

The composition process looks as follows: in its first stage, an abstract planner produces a (context) abstract plan, matching services at the level of input/output types. In the second stage, this plan is used by an offer collector, i.e., a tool which queries real-world services. The result is an offer plan containing concrete offers produced by service instances of appropriate types. In the third stage, the offers are searched by a concrete planner in order to find the best solution maximising a quality function.

PlanICS 2.0 has been revised and extended comparing to its previous edition [4, 5], making it easier to adapt to real-world applications (a comparison of the two versions is provided in the final section). This chapter gives an overview of PlanICS 2.0 in a way strict but informal, because of the space limitations. The rest of the chapter is structured as follows. In Section 6.3 the basic notions are introduced, necessary to describe the key topic of planning in Section 3.

6.2. Related work

The research in the area of automatic web service composition started very briefly after web services themselves became an important part of the modern IT. Many different approaches have been put forward, with several aims, ideas, and solutions. Here, we briefly describe the state-of-the-art in the field.

The Entish system [1] and the WSMO/SESA project [15] are two approaches particularily close to PlanICS 2.0 by sharing the idea of using ontologies with a formal semantics for representing knowledge about services. The major common features shared with Entish are discovering service capabilities by web communication, multi-stage planning, and using a similar service description language (a restricted quantification has been introduced to PlanICS 2.0). WSMO is similar to PlanICS 2.0 with respect to expressing the goal as a system state and using mediators/proxies for communicating with real-world services. PlanICS 2.0 differs from both the systems by a service model, extending the IOPE descriptions with mapping inputs/outputs of services to states of the transformed worlds, and an automatic conversion of the planning problem to the abstract domain (for planning in service types). Another difference is that our system does not (yet) execute services.

Among other approaches to service composition, [11] tackles the problem as a logic-based program synthesis using theorem provers. A semi-automatic composition is described in [13], and special languages to describe plans were proposed in [6] and [10]. An important group of methods formulates service compositions in terms of AI-planning. One of the most commonly used planning approaches is STRIPS/PDDL, used for example in [9].

Testing of the developed solutions is sometimes problematic as there are still not sufficiently many available real-world services. Thus, own testbeds were developed, enabling composition testing while setting the parameters of services [3]. PlanICS 2.0 also implements such a tester [7, 14].

6.3.Basic notions

Below we introduce basic notions for describing the planning stages of PlanICS 2.0.

6.3.1. Objects, object types, ontology

One of the main assumptions of our approach is that all the web services in the domain of interest as well as the objects processed by the services can be strictly classified in a hierarchy of *classes*, organised in an *ontology* (the ontologies are encoded using the OWL language [8]). All the classes are derived from the base class *Thing*. There are three direct descendants of *Thing*, namely *Artifact*, *Service*, and *Stamp*. The rest of the ontology modelling the domain of interest can be designed in an arbitrary way, but not violating the rules presented below.

The branch of classes rooted at *Artifact* is composed of the types of objects the services operate on, while the branch rooted at *Stamp* contains types of special-purpose objects aimed at confirming service executions and describing certain execution features (like a price or an execution time). Each object type definition consists of a number of typed attributes specifications, with the set of types including integer and real numbers, boolean values, dates and references to other objects. An object of a given type is an instance of the appropriate class.

The rules of class inheritance: a subtype class contains all the attributes of its parent classes, and optionally introduce some more. Multi-base inheritance is also allowed. The names of the attributes are unique within the ontology.

Valuations of objects, worlds: An object valuation is a function that assigns to each attribute of the object a value from the respective domain. A world is a set of objects together with their valuations. If partial valuations for a set of objects are specified only, then they define a set of worlds, of elements determined by all the possible assignments for the missing values, covering the respective domains. By a sub-world of a world w we mean a restriction of *w* to some subset of objects from *w*. Given two objects *o*, *o'* and their valuations v_o, $v_{o'}$ we say that $v_{o'}$ is compatible with v_o if the type of *o'* is either the same as the type of *o* or is a subtype of that type, and the values of v_o and $v_{o'}$ are the same for all the common attributes of the objects.

Consequently, a world w is compatible with a world w if there exists a one-to-one mapping between the objects of *w* and *w'* such that each object from *w'* is compatible with the object of *w* it corresponds to.

6.3.2. Services

A key notion of the approach is that of a *service*. We assume that each service processes a set of objects, possibly changing values of their attributes, and produces a set of new (additional) objects. The types of services available for planning are defined as elements of the branch of classes rooted at *Service*. Each service type stands for a description of a set of real-world services of some common features.

The common features of the services of a given type are described using the attributes introduced by the *Service* class. These attributes are: *in*, *inout*, and *out* aimed respectively at specifying sets of objects the service of a given type requires to execute (leaving them unmodified), processes while the execution (possibly modifying) and produces as its result, *preCondition* and *postCondition* (*pre* and *post*, for short) aimed at specifying the conditions these objects are to satisfy, and *inquiry*, *offer*, and *assign* enabling to describe the interaction with a real-world service of this type (an intuition behind the service description is presented in Fig. 6.2.). Technically, the values of *pre* and *post* are Boolean formulas (encoded in strings, similarly as the other features of services; an interpretation of these strings is the main task of the PlanICS 2.0 parser) being combinations of expressions over attributes of the objects from *in*, *inout* and *out* (respecting type limitations) and functions from a certain set applied to these attributes. In turn, the values of *inquiry* and *offer* are sets of (typed) parameters specifying respectively the data to be sent to a real-world service of the given type, and the data which will be received as an answer. The value of the *assign* attribute is a set of assignments specifying a relation between the contents of *inqury* and *offer* and the attributes of objects from the sets *in*, *inout* and *out*. The values of all the above attributes are kept in the ontology as the valuation of a special instance of the corresponding class, called a *metaservice*.

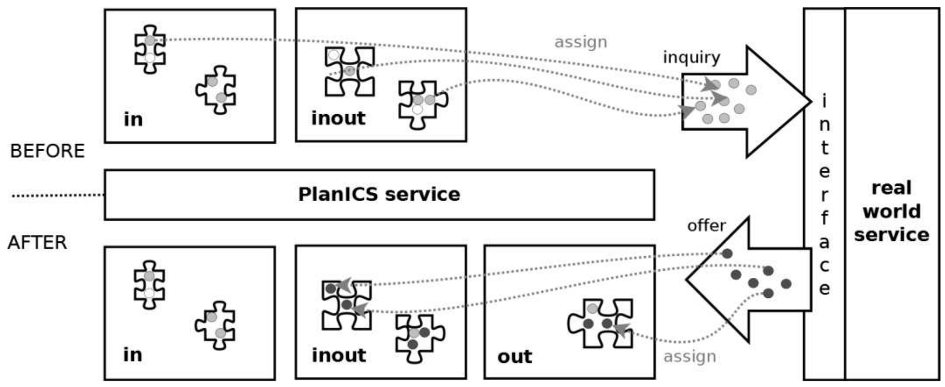


Fig. 6.2. PlanICS 2.0 service model. The boxes correspond to in, out, inout of a service, the puzzle- shapes model objects, the dots within them - their attributes.

The inheritance rules for the classes from the *Service* branch are the same as for the rest of the ontology. However, additional rules are needed to describe a computation of effective values of the attributes of metaservices being instances of derived classes. So, the formulas *pre* and *post* of such a metaservice are conjunctions of the corresponding formulas of all the ancestors up to the root of the hierarchy, and the formula specified explicitely. Similarly, the sets *in*, *inout*, *out*, *inquiry*, *offer*, and *assign* are unions of the appropriate sets for all the ancestors, and the set given explicitely. However, if the same object name is used in a set in a parent and a child class, then the one from the descendant must belong to a class derived from that from the ancestor, which means that it overrides the corresponding object from the parent specification.

Semantics: A service type *s* is understood as a pair of world sets, called the *input* and the *output worlds*, respectively. The input worlds consist of objects given by the union of the sets *in* and *inout*, of the valuations determined by the *pre* formula. Similarly, the output worlds are determined by the union of the sets *in*, *inout*, *out*, and the *post* formula.

A service type *s* can transform a world *w* if some its sub-world is compatible with some input world of *s*. The result of such a transformation is a world *w'* composed of the set of objects obtained by enriching that of *w* by the objects produced by *s* (i.e., given by its *out*). The valuation of each object $o \in w \cup w'$ not used as the *inout* parameter of *s* is the same in *w* and in *w'*, the valuations of all the remaining objects from $w \cap w'$ (i.e., used as the *inout* parameters of *s*) can by different from these in *w* only if this is implied by the assign attribute of *s*, and the objects from *w' \ w* (produced by *s*) have their valuations assigned in a way resulting from the assign attribute of *s*. The valuations satisfy also the *post* formula of *s*. By a *transformation sequence* we mean a sequence of service types such that the first service type is able to transform a given world, and each subsequent service type is able to transform the result of the previous transformation.

Service registry: The *service registry* is an element of the system which keeps an evidence of real-world web services, registered by their providers accordingly to the service types given by the ontology. Each entry of the register corresponds to one real-world service (however, the service provider can register its functionality using a number of PlanICS 2.0 service registry entries, for example to declare its compatibility to several PlanICS 2.0 services), and is a tuple containing an unique identifier of the service (assigned by the system), a type of the service (taken from the ontology), its specific *pre* and *post* that express conditions to be satisfied by the data the service receives and returns (the conditions are therefore logical formulas over the components of *inquiry* and *offer* for the appropriate class), and an *offerBinding* program responsible for interacting with the real-world service and obtaining this way offers satisfying requirements of interest.

6.4.Planning

Planning is the core functionality of PlanICS 2.0. In this section, we describe the complete planning process, starting from the user query and going through all the planning stages.

6.4.1. A user query

A task the user expects from the system to perform is given in the form of a user query specification. It resembles a service definition, i.e., contains typed objects in the *in*, *inout*, and *out* sets, as well as *pre* and *post* formulas over their attributes. As in the case of the service types, an interpretation of a user query specification is a pair of world sets. The initial worlds are determined by objects from *in* and *out*, and the *pre* formula of the query, while the expected worlds are defined by its *in*, *inout*, *out*, and the *post* formula.

A user query specification can introduce additional constraints on the world obtained as a result of composition (the final world) in order to limit the number of objects of a particular type (*cardinality constraint*) or aggregated (by aggregating we mean taking sum, minimal or maximal value) values of certain object attributes (*aggregate constraint*). Moreover, a *quality function* enables to specify criteria for evaluating the quality of a plan (e.g., the minimal cost, the minimal time, or some more complicated expression over objects from the final world).

Thus, every transformation sequence able to transform some initial world into a final world, which contains some expected world and meets all the additional constraints, and satisfies the user query, is called a *user query solution*.

6.4.2. Abstract planning

The first stage of planning is performed by an *abstract planner*. Its main goal is to determine which service types can potentially cooperate to satisfy the user query, and thereby to reduce the number of interactions with services in the subsequent planning phases. The planning in this phase looks as follows. First, the *pre* and *post* formulas from the service types and the user query specifications are converted to *abstract formulas*. That is, according to a formally defined transformation, the expressions involving the object attributes are simplified and each attribute value is substituted with the predicate *isSet* or *isNull*. These predicates determine the cases when the value needs to be known or remains unassigned, respectively. The transformation considers also the relations between reference attributes of objects and (in part) cardinality restrictions specified in the user query. Then, a search is performed and a *context abstract plan* (*CAP*) is produced. It specifies which service types need to be applied over which objects, in order to satisfy the user query. By the contexts we mean mappings between objects in the worlds, and the objects being service parameters.

6.4.3. Collecting offers

The next stage of the process consists in collecting offers. This is done by an *offer collector* on the basis of the context abstract plan, but using *full* (not abstract) *condition formulas*.

The offer collector communicates with the registered real-world web services of appropriate types (using their *offerBinding*s to this aim), collecting offers for each service type present in the context abstract plan. More precisely, the offer collector sends to each appropriate *offerBinding* the constraints on the data we are potentially able to sent to the service in the *inquiry*, and on the data we expect to receive in the offer in order to keep on building a potential plan (checking earlier whether these constraints do not contradict the specific limitations of the service specified in its *pre* and *post* in the registry). The *offerBinding* program determines, by way of an interaction with the real-world service, the possible variants of the service execution satisfying the constraints mentioned, and returns them in the form of formulas. The pair of formulas, the first of which specifies the actual constraints on the data the real-world service "agrees" to receive, and the second - the constraints on the data it declares to return in consequence, is called a *proposal* of the service.

The offer collector works recursively, using the proposals collected earlier to obtain the next ones, and memorising the bindings between them. The result of its work is an *offer plan* of the nodes representing sets of worlds implied by the proposals and mapped to the worlds of CAP, and of the edges corresponding to real-world services of appropriate types.

6.4.4. Concrete planning

The last stage of the planning process is concrete planning, taking an offer plan and finding a *concrete plan* - i.e., a sequence of real-world services (corresponding to a CAP sequence) and data to be sent to these services which form together a scenario maximising the quality function. Potential consecutive phases of planning, such as executing concrete plans or regenerating them partially when the execution fails, are currently not covered by our research.

6.4.5. Implementation, algorithms

At the current stage of the PlanICS project, abstract planners have been implemented with the associated infrastructure. Concerning ontology, the OWL modeling approach is used with available implementations. Two abstract planners have been developed so far: one based on a translation to Satisfiablity Modulo Theorems (SMT) [7] and another using Genetic Algorithms approach [14]. A generator of benchmarks has been implemented, allowing to scale several parameters such as number of services, maximum number of processed objects and object attributes, etc.

6.5.Conclusions and future work

PlanICS 2.0 offers a complete solution to automatic web service composition, distinguished by a multi-stage planning and focused on an easy adaptation to existing models of web services. Our main objective was to create a reasoning system based on ontologies modelling selected aspects of business processes, implemented as web services. A flexible semantic model allows to use the system in various domains in order to achieve a goal - automatic web service composition. Additional advantages of the approach are a reduction of the search space by a classification of the services, and dynamic discovering their capabilities suitable for the created composition. A special attention is paid to transforming a composition problem to other problems solvable by well-known effective reasoning methods.

Comparing PlanICS 2.0 to its previous edition [4, 5], the improvements consist in: (1) extending the service descriptions by parts related to proxy communication (*inquiry*, *offer*) and relation of the processed world with the service input and output (*assign*), (2) introducing a concise multi-proposal representation of *offerBinding*, (3) separating between collecting offers and planning with offers, (4) extending the query language with a restricted quantification over objects and attributes as well as with a quality measure, (5) defining a formal conversion from the concrete to the abstract planning domain, and (6) providing two implementations of the abstract planning engines based on genetic algorithms and SMT-solvers. The offer collecting and concrete planning phases are still to be implemented as a next stage of the development.

Acknowledgements

The research described in this chapter has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.

Bibliography

[1] S. Ambroszkiewicz. Entish: A language for describing data processing in open distributed systems. Fundam. Inform., 60(1-4):41-66, 2004.

[2] M. Bell. Introduction to Service-Oriented Modeling. Wiley & Sons, 2008.

[3] E. Cho, S. Chung, and D. Zimmerman. Automatic web services generation. In HICSS, pages 1-8. IEEE Computer Society, 2009.

[4] D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, and J. Skaruz. HarmonICS - a tool for composing medical services. In ZEUS, pages 25-33, 2012.

[5] D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, M. Szreter, and A. Zbrzezny. PlanICS - a web service compositon toolset. Fundamenta Informaticae, 112(1):47-71, 2011.

[6] M. Klusch, A. Gerber, and M. Schmidt. Semantic web service composition planning with OWLS-XPlan. In Proc. of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web, pages 55-62. AAAI Press, 2005.

[7] A. Niewiadomski, W. Penczek, and A. Półrola. Towards SMT-based Abstract Planning in PlanICS Ontology. In Proc. of KEOD'13, to appear, 2013.

[8] OWL 2 web ontology language document overview. http://www.w3.org/TR/owl2-overwiew/, 2009.

[9] J. Peer. A PDDL based tool for automatic web service composition. In Proc. of the Second Intl Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR), pages 149-163. Springer Verlag, 2004.

[10] S. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for web service composition. In Proc. of the 11st Int. World Wide Web Conference (WWW'02), 2002.

[11] J. Rao, P. Küngas, and M. Matskin. Logic-based web services composition: From service description to process model. In Proc. of the IEEE Int. Conf. on Web Services (ICWS'04), pages 446-453. IEEE Computer Society, 2004.

[12] J. Rao and X. Su. A survey of automated web service composition methods. In Proc. of the 1st Int. Workshop on Semantic Web Services and Web Process Composition (SWSWPC'04), volume 3387 of LNCS, pages 43-54. Springer- Verlag, 2004.

[13] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic compositions of web services using semantic description. In Proc. of the Int. Workshop 'Web Services: Modeling, Architecture and Infrastructure' (WSMAI'03), pages 17-24, 2003.

[14] J. Skaruz, A. Niewiadomski, and W. Penczek. Automated abstract planning with use of genetic algorithms. In Proc. of GECCO'13, to appear, 2013.

[15] Web Service Modelling Ontology D2v1.0. http://www.wsmo.org/2004/d2/v1.0, 2004.

Chapter 7

Test case generation on the base of business rules described in structural natural language

Requirement specification is a key artefact in any software development process. Among others, it may include business rules written in natural language. Requirement specification is the basis for a design and coding of a software system. It is also used for preparing tests verifying a software product against different kinds of requirements. Testing artefacts include test cases, mainly created by testers who interpret requirement specification for that purpose. In this chapter an approach to automatic generation of test cases is presented. The test cases are created on the base of business rules expressed in a structured natural language, more specifically in SBVRSE notation. A tool developed for that generates abstract test cases expressed in natural language in tabular form or as a set of English sentences. Generated test cases are assumed to support a tester in creation concrete test cases which can be applied for manual or automatic tests.

7.1.Introduction

Business rules belong to important artefacts of requirement specification. On one side they provide definitions of terms used in a specific domain, on the other – they represent constraints on the structure and behaviour of that domain. Business rules are usually defined in natural language understandable for all involved parties. High quality software should be consistent with such defined business rules.

Checking if the software meets business rules for specific domain is a concern of software verification [5], and it is usually performed as a part of dynamic analysis and functional testing. Testing – regardless the way it is done (manual, automatic) – requires a set of test cases to be prepared, i.e. “a set of inputs, pre conditions, expected results, and post conditions” [6]. Test cases can be abstract or executable. Abstract test cases define test inputs and outputs with their specific input values and expected results. For example, supposing that a component for calculating an average value of two numbers is being tested, an abstract test case can specify the following values: 6, 14 (test data) and 10 (test result). Abstract test cases cannot be directly executed against the subject of testing because they are on a different level of abstraction. That is the main difference between abstract and executable test cases – the latter can communicate directly with the subject of testing [1].

In this chapter an approach to automatic test generation of abstract test cases is proposed. The test cases are generated on the base of business rules expressed in SBVRSE. Test cases are expressed in natural language and presented in a form of English sentences or as a table. Such test cases can support a tester in creation executable test cases which can be applied for manual or automated tests. The set of business rules considered during the generation process is limited to structural assertion and integrity constraint rules.

This chapter is organized as follows. Section 7.2 presents the adopted classification of business rules, and points out those being the subject of generation. Section 7.3 shortly describes existing testing patterns, from which some were implemented in generation tool. The architecture of the tool and the way of its operation is presented in Section 7.4. Section 7.5 contains a case-study which demonstrates the tool capabilities. Section 7.6 concludes the chapter, and presents directions of further works.

7.2.Business rules

Each organization is a complex organism which has to obey outer or inner regulations, standards, and policies, known as business rules. Business rules are statements that either define or constrain a specific aspect of organization functioning [7].

Because business rules are under business jurisdiction and should be understood especially by that side, they are expressed (especially at the first stage) in a natural language.

7.2.1. Business rules classification

Many different classifications of business rules exist in the literature. However, we use the classification proposed by Business Rules Group [7]. This classification splits the business rules into 3 categories which are further divided into subcategories. The presentation below lists all kinds of business rules but gives short explanations only for those we are interested in:

- Structural assertions:
  - Business terms – elements of business glossary that need to be defined, e.g. invoice
  - Common terms – elements of business glossary which commonly known meaning, e.g. car
  - Facts – relationships between terms:
    - Attribute – a feature on a given term, e.g. colour
    - Generalization – represents "as-is" relationship, e.g. car is a specific case of vehicle
    - Participation – represents semantic dependency between terms, e.g. student *enrols* for courses
      - Aggregation – represents "a whole-part" relationship, e.g. book consists of pages
      - Role – describes way in which one term may serve as an actor, e.g. customer may be *a buyer in* a contract
      - Association – used when other kinds are inappropriate
- Action assertion – concerns some dynamic aspects of the business
  - Division according to class:
    - Condition
    - Integrity constraint – an assertion that must always be true (invariants).
    - Authorization
  - Division according to type: Enabler, Timer, Executive
  - Division according to role: Action Controlling Assertion, Action Influence Assertion
- Derivation: Mathematical calculation, Inference

7.2.2. SVBR and supporting tools

Semantics of Business Vocabulary and Business Rules (SVBR) is an OMG standard that allows describing business rules in a convenient way (in semi-natural language) which can be transformed to other representations, easier for computer processing [8]. SBVR separates the representation of business rules from their meaning, what enables adoption of that standard for different natural languages. At that moment only English is supported (SBVRSE – SBVR Structured English).

SVBR uses two vocabularies. The first – named *Vocabulary for describing business vocabularies* – defines all notions from a specific domain (terms and facts); the second – *Vocabulary for describing business rules* – static and dynamic constraints based on previously defined facts and terms.

It is worth to mention that SVBR is flexible enough to specify all elements from UML class diagram (classes with attributes, associations, and compositions with multiplicities, generalizations).

Nowadays there are only a few tools supporting SBVR. Mostly they are text editors, e.g. SBearVeR [9], SBVR Visual Editor [10]. One of the most interesting is VeTIS [8], elaborated as a plug-in for Magic Draw. It is able to transform business models defined in SBVR to UML class diagram accompanied with a set of OCL constraints.

7.3.Testing patterns

Testing patterns represent strategies which testers can use during verification process. The main popular testing patterns are [1]:

1. *Boundary Value Analysis*, BVA – used in a context of singular variable which range of possible values is constrained. In a context of a number variable VBA suggests checking it with minimal, maximal, one above the maximal, and one below the minimal values.
2. *Equivalence Class Partitioning*, ECP – addresses a need of checking potentially big number of input values with a limited number of test cases. An equivalence class represents such sets of input values which are treated by an application in similar way.

3. *Combinatorial Analysis*, CA – aims at finding undesirable interactions among input values. In typical cases, testing of all possible combinations of such values is impossible.
4. *Fuzz testing,* FZ – used mainly to check the safety of testing object. This method checks how the objects react on (big amount of) improper data.

7.4.Proposed approach to test case generation

The approach to test case generation used in the developed tool is an implementation of Model Based Testing (MBT). MBT organizes the testing process with four main layers of abstraction [1]: the System Under Test (SUT), the model of SUT (simplified representation of the test object), the abstract test case and the concrete test case – see Fig. 7.1.

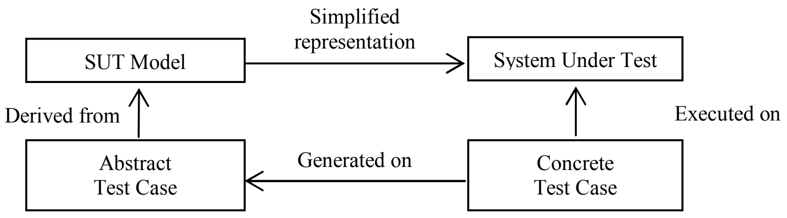


Fig. 7.1. MBT general scheme

Due to the fact that the approach focuses on using business rules expressed in SBVR notation our implementation of MBT makes use of two SUT models. The first one is the SBVR business rules set which gets transformed into a UML class diagram with additional OCL constraints by Magic Draw software with VeTIS plug-in. The second SUT model (called the Inner SUT Model) gets constructed on the fly by the developed tool itself based on the information retrieved from the input XML file (representation of UML class diagram). The Inner SUT Model is used as the actual input data for the developed test case generation algorithm. The proposed approach is presented on Fig. 7.2.

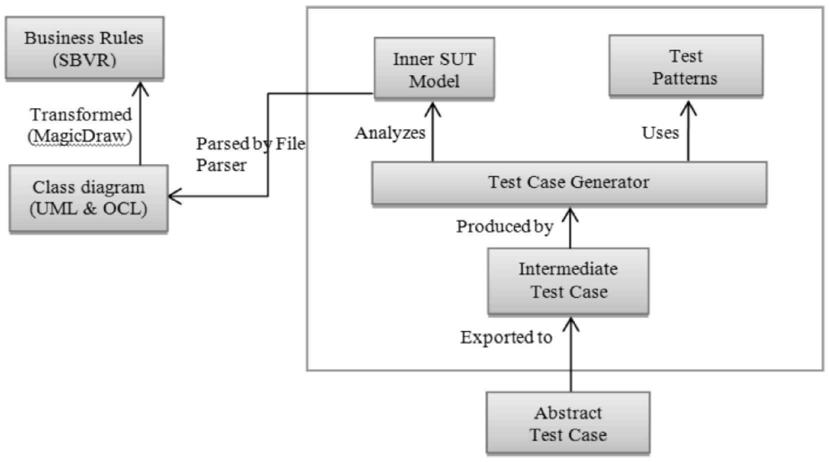


Fig. 7.2. MBT implementation in the proposed approach

The generation algorithm makes use of implemented test patterns (BVA) and applies them to various aspects of the Inner Model which selected based on their applicability beforehand. The product of the algorithm is a set of abstract test cases which can be used by testers to produce concrete test cases.

Inner SUT Model construction

The first phase of test case generation is focused on constructing the Inner SUT Model based on the input data provided by the class diagram stored in the XML file. This activity is performed by the *File Parser* component.

In Inner SUT Model primitive properties are modeled in terms of name, type, and visibility. Associations are modeled in a similar manner with the exception that the type which is stored is actually the name of the class at the other end of the association. Also the value range for the association's multiplicity is remembered. The current stage of implementation focuses on binary associations which are related to the limited functionality of VeTIS plug-in. Also the OCL constraints applied to number properties and their values are processed by *File Parser*.

Test case generation algorithm

Tests are generated by *Test Case Generator* component. This component uses defined test patterns for that purpose. At that moment *Test Case Generator* provides only Boundary Value Analyzer which is an implementation of the Boundary Value Analysis test pattern. For every analyzed boundary value two test cases are generated: one which verifies the boundary value which lies within the accepted range and one which verifies the value exceeding the range. In corner cases, like 0 or * for association multiplicities, tests are not generated.

7.5.Case study

The proposed method is illustrated with a simple case study elaborated on the base of [8]. The input for the tools is two basic vocabularies expressed in SBVRSE.

Vocabulary for describing business vocabularies is presented below (a part of it).

```
account
  Definition: a formal contractual relationship established to provide for regular banking or brokerage or business services; "he asked to see the executive who handled his account"
account balance
  General_concept: number
atm
  Definition: Acronym for automated teller machine, a machine at a bank branch or other location which enables a customer to perform basic banking activities (checking one's balance, withdrawing or transferring funds) even when the bank is closed.
balance check
bank
card
digit
  General_concept: integer
```

This vocabulary contains 15 terms and 18 facts as well as 12 synonyms for facts.

Vocabulary for describing business rules defined on the base of terms and facts is defined below (a part of it).

```
bank controls account
  Synonymous_form: account is_controlled_by bank
user owns account
  Synonymous_form: account is_owned_by user
...
It is necessary that each transaction has exactly one quote.
It is necessary that each withdrawal has exactly one quote.
It is necessary that each quote of_the transaction
is_not_greater_than 1000.
...
```

This vocabulary contains 28 constraints among which 10 are integrity constraints.

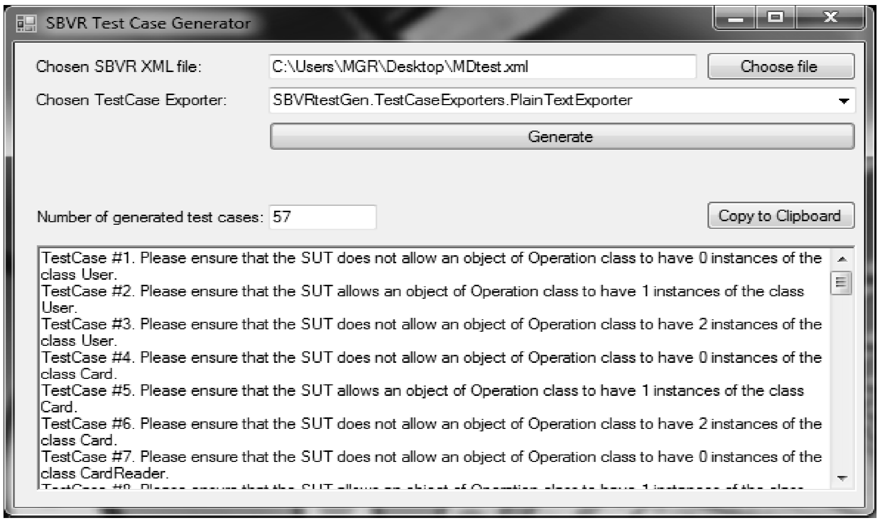
Both glossaries were rewritten in the VeTIS tool, and imported to the Magic Draw. Next, the VetTIS tool was used for generation of a UML class diagram with OCL constraints. The generated diagram was exported to an XML file.

Depending of the test exporter used, test cases are presented in English or as a table. The results (part of it) are presented in Fig. 7.3.

For the considered example 57 general test cases were generated. Test cases come to two groups: those verifying multiplicities of associations, and those connected to attribute values. Below there is an example of test cases created for integer attribute, presented in a table 7.1.

Table 7.1. example of test cases created for integer attribute

ID	Class	Property	Value	Allowed
54	Withdrawal	Quote	1000	Yes
55	Withdrawal	Quote	1001	No
56	Withdrawal	Quote	1	Yes
57	Withdrawal	Quote	0	No



Chosen SBVR XML file: C:\Users\MGR\Desktop\MDtest.xml

Chosen TestCase Exporter: SBVRtestGen.TestCaseExporters.PlainTextExporter

Number of generated test cases: 57

```
TestCase #1. Please ensure that the SUT does not allow an object of Operation class to have 0 instances of the
class User.
TestCase #2. Please ensure that the SUT allows an object of Operation class to have 1 instances of the class
User.
TestCase #3. Please ensure that the SUT does not allow an object of Operation class to have 2 instances of the
class User.
TestCase #4. Please ensure that the SUT does not allow an object of Operation class to have 0 instances of the
class Card.
TestCase #5. Please ensure that the SUT allows an object of Operation class to have 1 instances of the class
Card.
TestCase #6. Please ensure that the SUT does not allow an object of Operation class to have 2 instances of the
class Card.
TestCase #7. Please ensure that the SUT does not allow an object of Operation class to have 0 instances of the
class CardReader.
```

Fig. 7.3. General test cases prepared by the tool

7.6.Conclusions

This chapter presents a method to generic test case generation on the base of business rules expressed in natural language. The proposed approach is a specialization of MBT, in which SBVRSE specification is treated as original SUT model. This SUT model is further transformed into an UML class diagram with OCL constraints, which XML version feeds the developed tool. The tool uses test patterns (at that moment BVA only) to prepare intermediate test cases. These test cases are assumed to support testers with test ideas. They can be also the input for generation of concrete test cases. At that moment integrity constrains (invariants) built upon structural assertions are taken in consideration. In the future we plan to extend the set of considered business rules. It seems that the most promising are authorization rules as well as condition rules (preconditions for operations). The case study showed the usability of proposed approach.

There are several examples in which UML diagrams are used for test generation [2,3,4] but none of them directly address the problem of business rules. Typically, SUT model in these approaches is a kind of finite state machine. For example in UML-CASTING the space of possible states of testing model is prepared on the base of class diagram and state machine diagram, but state machine diagram is a crucial one [2]. This representation of SUT model allows generating test cases that transit through possible states of testing model. The other example of MTB is those presented in [12]. Here test cases are generated on the base of BPMN diagrams. In this approach SUT model is represented as a Petri net modeling the space of its states [12]. Our approach to test generation at that moment uses only static diagrams as SUT model. It can be changed in the future, when dynamic business rules will be taken into consideration.

Bibliography

[1] Page A., Johnston K., Rollison B.,"How we test software at Microsoft", Microsoft Press, 10.12.2008, ISBN 9780735624252

[2] Van Aertryck L., Jensen T.,"UML-CASTING: Test synthesis from UML models using constraint resolution", In Proc. AFADL'2003, http://www.irisa.fr/triskell/AFADL2003/actesAFADL2003/test04.pdf

[3] Pickin S., Jard C., Traon Y.L., Jéron T., Jézéquel J-M., Le Guennec A., "System Test Synthesis from UML Models of Distributed Software", IEEE Transactions on Software Engineering, Vol. 33, Iss. 4, pp. 252-269, 2007

[4] Lugato D., Bigot C., Valot Y., "Validation and automatic test generation on UML models: the AGATHA approach", International Journal on Software Tools for Technology Transfer, Vol. 5, Iss. 2-3, pp. 124-139, 2004

[5] Graham D., Veenendaal E., Evans I., Black R., "Foundations of Software Testing: ISTQB Certification", Cengage Learning Business Press, 2008

[6] "IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.12-1990", Institute of Electrical & Electronics Engineering, 1991

[7] "Defining Business Rules - What Are They Really?", The Business Rules Group, 2000, http://www.businessrules-group.org/first_paper/BRG-whatisBR_3ed.pdf

[8] “Creating UML&OCL Models from SBVR. Business Vocabularies and Business Rules. VeTIS User Guide”, Kaunas University of Technology, 2009
[9] SBeaVeR, http://sbeaver.sourceforge.net/, 2006
[10] “SBVR Visual Editor”, http://sbvrve.sourceforge.net/, 2009
[11] Apfelbaum L., Doyle J.,“Model Based Testing”, Software Quality Week Conference, Maj 1997, http://www.geocities.com/model_based_testing/sqw97.pdf
[12] Buchs D., Lucio L.and Chen A., “Model Checking Techniques for Test Generation from Business Process Models”, Lecture Notes in Computer Science, Volume 5570/2009.

Chapter 8

Acceptance test generation based on detailed use case models

Tests performed in order to verify compliance of a software system with customer expectations cover different areas. Some of them verify the functionality, other – the business domain logic, the non-functional characteristics or the user interface. Usually they are done separately, but on the same functional areas. This chapter presents the concept for the Requirements Driven Software Testing (ReDSeT) tool, which allows for automatic integrated test generation based on different types of requirements. Tests are expressed in newly introduced Test Specification Language (TSL). The basis for functional test generation are detailed use case models. Furthermore, by combining different types of requirements, relations between tests are created. The constructed tool acknowledges validity of the presented concept.

8.1.Introduction

The main goal of a software development project is to deliver a software product that meets the expectations of the customer. Verification of compliance with the requirements of the stakeholders is possible by carrying out acceptance tests [1] . Acceptance testing is the process of comparing the system under development to its requirements and needs of its users. These tests are performed usually by the customer through comparing the system's operation to the original contract between the stakeholders and the developers. This contract should be understandable for the stakeholders and at the same time precise enough for the developers to produce efficient software.

To describe the expected functionality of the software system, use cases are commonly used [2] . Use cases describe interactions between external actors and the system, which lead to specific goals according to the given scenarios. Such requirements are supposed to be satisfactory to define the tests, that will be used during acceptance testing.

To improve the development of tests from use cases, a number of automatic test generation mechanisms were proposed. Examples of such approaches can be found in work by El-Attar and Miller [3] , Gutiérrez et al. [4] , and Nebut et al. [5] . Beside use cases, requirements specifications contain other types of requirements, that describe different aspects of the desired software. These requirements also should be verified by executing corresponding tests. Some work has been done on the generation of tests based on business rules (see Junior et al. [6]), GUI requirements (see Bertolini and Mota [7]), and even on non-functional requirements (see Dyrkom and Wathne [8]).

All these mechanisms use model transformation forming the area of *Model-based testing (MBT)*, which is an evolving technique for generating a suite of test cases from requirements [9] . Although different types of tests are generated from requirement models describing the same software system usually they are not related, because they verifies different aspects of the system.

This chapter describes the idea of automatic generation of different types of tests integrated in functional test cases and test scenarios executed during acceptance testing. These tests are generated on the basis of interrelated requirements describing many aspects of the developed software system, which makes this idea MBT compliant. The element that integrates different types of testing is the functional test case corresponding to the use case scenario as shown in Fig. 8.1.

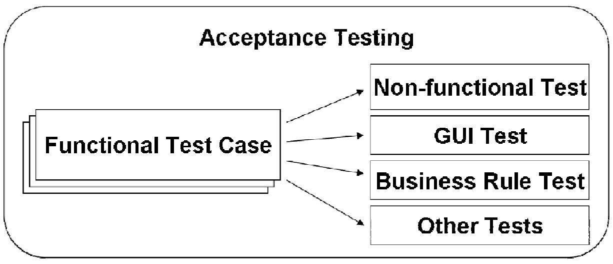


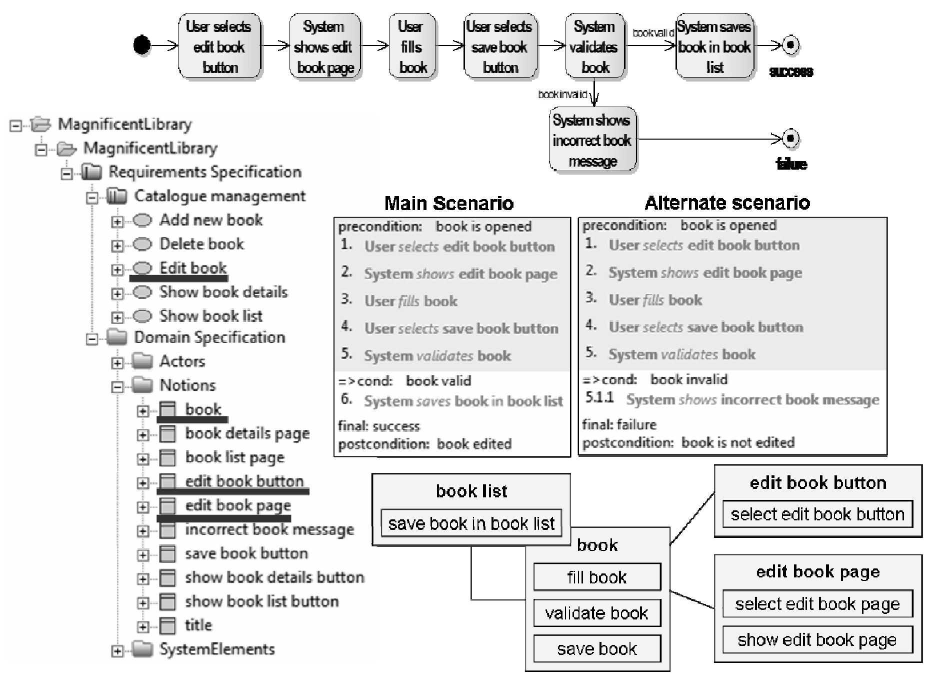
Fig. 8.1. Acceptance test suite based on functional test cases

This concept is based on the test metamodel defined as the Test Specification Language (TSL) and implemented within the ReDSeT tool (Requirements Driven Software Testing). The tests are generated automatically based on the requirements specification created with RSL (Requirements Specification Language) [10] . As RSL gives a notation for precise use case scenarios, generation of test cases verifying the system behaviour is significantly facilitated. Additional information contained in scenario sentences (notions from the domain vocabulary) and other related requirements allows for generation of tests of different types. All the tests generated on the basis of RSL-based requirements form a complete test suite for acceptance testing.

8.2.Detailed requirements expressed in RSL

As in other test generation solutions, the basis for automatic generation of tests is the precise specification of requirements. As mentioned above, the described solution is based on the requirements specification created with RSL. The main features of this language are: clear separation of descriptions of the system's behaviour and descriptions of the system's domain. Functional requirements can be presented in three equivalent forms: structured text with hyperlinks to domain elements, activity diagram and sequence diagram. It allows for precise specification of requirements, which is understandable even for ordinary people who do not have technical expertise. The language has a precise specification of its syntax and semantics [10] with methods of its use explained e.g. by Śmiałek et al. [11] . Fig. 8.2 shows an example requirements specification, created in RSL.

All the elements of a requirement specification are grouped in packages in a tree structure. Simple requirements described with the free text can be used to define business rules or non-functional aspects of the system. Use cases describing the functionality of the system are described with structured scenarios. Scenarios are consist of numbered sentences in a simple grammar SVO(O). These sentences are constructed with notions stored in the domain vocabulary. This is illustrated with two scenarios (main and alternative) of the *Edit book* use case. The same information is presented in the form of an activity diagram that is generated automatically from the scenarios.



Main Scenario

precondition: book is opened
1. User *selects* edit book button
2. System *shows* edit book page
3. User *fills* book
4. User *selects* save book button
5. System *validates* book

=>cond: book valid
6. System *saves* book in book list

final: success
postcondition: book edited

Alternate scenario

precondition: book is opened
1. User *selects* edit book button
2. System *shows* edit book page
3. User *fills* book
4. User *selects* save book button
5. System *validates* book

=>cond: book invalid
5.1.1 System *shows* incorrect book message

final: failure
postcondition: book is not edited

Fig. 8.2. Example of detailed use cases expressed RSL

The notions are referred to in scenario sentences through hyperlinks (*book, book list, edit book button, edit book page*) and are presented on a notion diagram, that is similar to a class diagram. The relationships between notions, and notion operations are defined automatically according to the scenario sentences where these notions appear or are defined manually by the requirements engineer. The notions and their operations used in use case scenarios describe the business logic and the user interface elements.

All the requirements can be related. To depict relations between use cases, a special *invoke* relationship is used. It allows to determine under what conditions and in which step of a use case scenario another use case is to be called (see Śmiałek et al. [11] for more details).

RSL is based on a formal metamodel. This allows for automatic processing of information contained in the requirements specification. We will use this characteristics of RSL for generating test cases.

8.3.Automating test generation

To define acceptance test suite and to ensure accurate and automatic transition from RSL-based requirements to tests, Test Specification Language (TSL) was developed. This language is based on a metamodel defined in EMF (Eclipse Modeling Framework) [12] and is out of scope of this chapter.

The main idea of TSL is to provide the notation for reusable tests, that are understandable for non-technical people and precise enough for detailed verification of the software system. All tests are grouped in a tree structure, named the Test Specification (see Fig. 8.3), that groups tests assigned to a specific release of software. Each test contained in the test specification represents a procedure for software verification for a single requirement. Such a verification is made by examining all the check points defined inside a test.

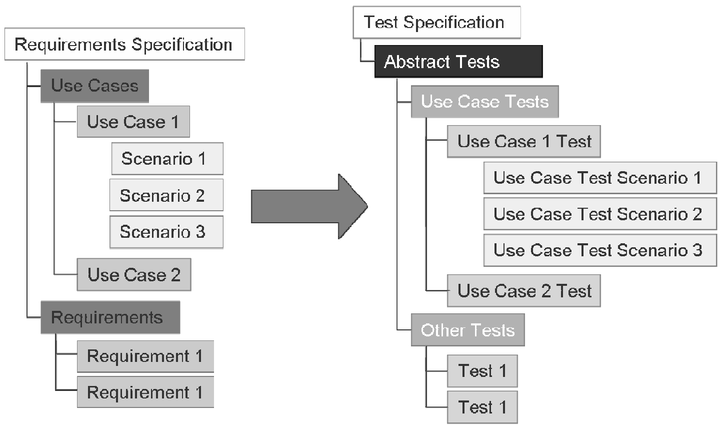


Fig. 8.3. Test generation based on the requirements specification

The basic structure of a TSL test specification consists of two packages: Abstract Tests and Concrete Tests. The first of these includes tests generated directly from the requirements specification: mostly Use Case Tests but also other related tests of other types. A use case test corresponds to a use case, and includes test scenarios, as shown in Fig. 8.3.

A use case test scenario includes the initial condition (a precondition sentence) that must be met before the execution of actions described in this scenario and the final condition (a postcondition sentence) that describes the desired state of the system after the scenario is executed.

Every use case test scenario, generated from an RSL use case scenario, is a sequence of actions forming a dialogue between the primary actor and the system. Every such action is expressed by a single sentence in a simple subject-verb-object (SVO) grammar (see Graham [13] for an original idea). These sentences describing single actions can have check points assigned. In addition to action sentences, two additional sentence types were introduced: condition and control sentences. They are used in a scenario to express the flow of control between alternative scenarios of the same use case as well as between scenarios of different use cases (see [13]).

An important feature of the requirements specified with RSL, is the possibility to create relationships between requirements. Due to generation of test specifications on the basis of these requirements, relationships between tests are also created. Relations binding use cases depicted as *invoke* are transferred to become relations between use case tests. This brings information from which step of the use case test scenario and under what conditions, a scenario of another use case test should be called. Other requirements' relationships are transferred to relationships binding other tests than use case tests.

8.4.Instantiating concrete tests

A scenario of a use case test determines the conditions, steps and check points that will be subject to verification for the use case implementation. Such algorithms will be used in acceptance testing after placing them in test scenarios and assigning specific test data values.

Test scenarios are grouped by the second package in the basic structure of the test specification (see Fig. 8.4). They are defined by a test engineer as a set of ordered instances of use case test scenarios, that are named functional test cases. Functional test case is composed of ordered steps in form of SVO sentences. Each step can contain check points with assigned test data values and can be related with test cases of other types. Test cases of other types are automatically created during instantiation. They are related with functional test case the same as other tests are related with use case test scenarios and particular scenario sentences.

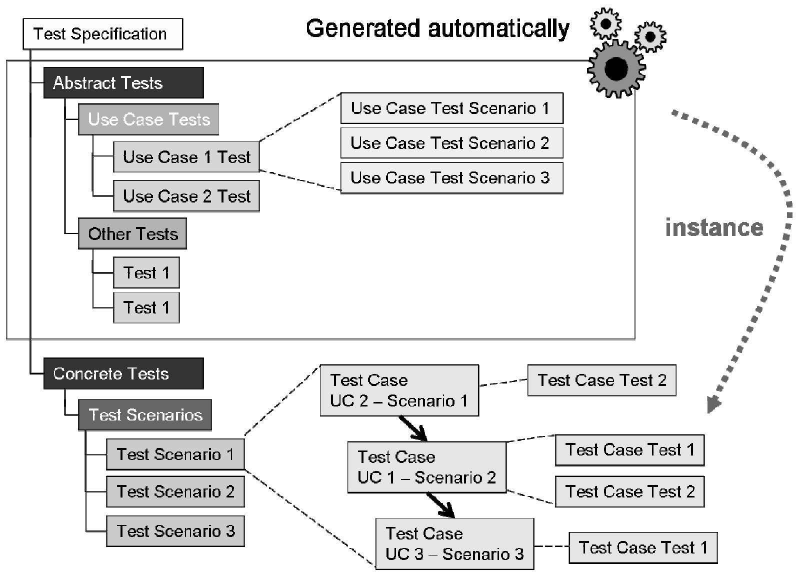


Fig. 8.4. Test scenarios composed of concrete test cases

A test scenario constructed with test cases builds also the context for the test data. The initial test data values are set by the test engineer as the precondition values of the test scenario. Test data values describe basic business objects as well as GUI elements. The test data in the scope of one test scenario are passed between test cases as its' precondition and postcondition values. Test data values are changing according to the functionality and the business logic that are under tests. Although the test cases cannot be formally related to each other, within the test scenarios they simulate business processes that are implemented in the system being tested.

Particular use case test scenarios, can be instantiated as test cases in the scope of other test scenarios with different test values. It makes the tests reusable.

The described above idea of creating test cases allows for verifying of system's application logic. Use case test cases form the skeleton of a test specification. This can be supplemented with other types of testing, such as business logic, GUI or non-functional tests, depending on the content of the requirements specification. The extension of the TSL metamodel allows to attach to the use case test scenario steps other specialized tests. Such a test specification would be complete for acceptance testing.

8.5.Tool support

The tool supporting the described idea of automatic test generation based on requirements is called ReDSeT (Driver Software Testing Requirements). It is based on the Eclipse Rich Client Platform. This enables integration with ReDSeeDS tool (www.redseeds.eu [14]) which provides advanced editors for requirements (for use cases, notions and other requirements) described with RSL. The generated test specification can be included in the same Eclipse project as the requirements specification and code. This enables integration of activities at different stages of the software development project.

ReDSeT tool perspective is divided into several areas. The Test Specification Browser allows to manage the test specification, which is organised in a tree structure. The current test cases and the test scenarios are presented in the Test Editor area. The Detailed Test View is dedicated for viewing check points and editing test values contained in all the types of tests.

The tool does not allow to edit the tests generated automatically based on the requirements. It is expected that all the necessary information about the check points will be derived from the specification requirements. If there is a need to clarify the test, the requirements should be modified and then the test specification should be regenerated. This assures strict compliance of the tests with the requirements.

The ReDSeT tool is designed as a set of integrated plug-ins, which are responsible for: automatic generation of tests, test management, use case tests viewer, test scenario editor for composing sequence of functional test cases and editors for test data values. Test viewers and test data value editors for tests of other types may be attached to the tool as the additional plug-ins.

The repository of the test specification is based on the EMF technology. The TSL metamodel can be easily extended to handle other types of tests which are adapted to different types of requirements associated with use cases. As the repository of the test specification XML file is used. It gives the technical capabilities for easy extraction of test scripts for acceptance tests execution.

8.6.Conclusions

The proposed idea and the ReDSeT tool bring a complete solution for creating acceptance testing for the systems that are focused on user-system interaction. The basis for creation of a set of test scenarios are detailed use cases. Requirements defined in RSL significantly facilitate automatic test generation, and TSL allows for expressing interrelated tests of different types in a way that is comprehensible to the audience responsible for acceptance testing.

The automatically generated tests can be re-used in various test scenarios. The work on the preparation of the test specification can begin during the requirements formulation stage, and regeneration of tests allows to reach test complexity corresponding to detail level of final requirements.

It can be noted that the proposed method is based on black box testing and is independent of the implementation technology of the system under test. Since RSL and TSL are based on the metamodels, the whole idea is close to Model Based Testing. Traces from requirements to test cases are planned to be used for generating requirements with tests coverage reports. These traces will be subject to further research on regression test selection.

Relying TSL on the EMF-compliant metamodel and constructing the ReDSeT tool as an Eclipse plug-ins allows for easy extension of the described solution. To support another types of tests, the metamodel and appropriate editor plug-ins should be developed. In the future it is planned to extend the solution with detailed tests for the business logic and graphical user interface.

It is also planned to extend the tool with the mechanism for extracting of test scripts as input for tools that automates test execution (e.g. IBM Rational Functional Tester, Selenium). It would bring a complete solution for detailed use case based testing.

Bibliography

[1] Myers G. J., Sandler C., Badgett T.: The Art of Software Testing. Wiley Publishing, 3rd edition, 2011.
[2] Cockburn A.:. Writing Effective Use Cases. Addison-Wesley, 2000..

[3] El-Attar M., Miller J.: Developing comprehensive acceptance tests from use cases and robustness diagrams, RE, 15(3), 285–306,09.2010.

[4] Gutiérrez J. J., Escalona M. J., Mejías M., Torres J.: An approach to generate test cases from use cases, ACM, Proc. ICWE ’06, 113–114, 2006.

[5] Nebut C., Fleurey F., Le Traon Y., Jézéquel J.: Automatic test generation: A use case driven approach, IEEE Transactions on Software Eng., 32, 140–155, 2006.

[6] Mendes Bizerra Junior E., Silva Silveira D., Lencastre Pinheiro Menezes Cruz M., Araujo Wanderley F.J.: A method for generation of tests instances of models from business rules expressed in ocl, IEEE Latin America Transactions, 10(5), 2105–2111, 2012.

[7] Bertolini C., Mota A.: A framework for gui testing based on use case design, IEEE, Proc. ICSTW ’10, 252–259, 2010.

[8] Dyrkorn K., Wathne F.: Automated testing of non-functional requirements, ACM, Proc. 23rd OOPSLA Companion ’08, 719–720, 2008.

[9] Dalal S. R., Jain A., Karunanithi N., Leaton J. M., Lott C. M., Patton G. C., Horowitz B. M.: Model-based testing in practice, ACM, Proc. ICSE ’99, 285–294, 1999.

[10] Kaindl H., Śmiałek M., Wagner P., Svetinovic D., Ambroziewicz A., Bojarski J., Nowakowski W., Straszak T., Schwarz H., Bildhauer D., Falb F., Brogan J. P., Mukasa K. S., Wolter K., Kavaldjian S., Szép A., Kalnina E., Kalnins A.: Requirements specification language definition, ReDSeeDS Project Deliverable D2.4.2, www.redseeds.eu, 2009.

[11] Śmiałek M., Ambroziewicz A., Bojarski J., Nowakowski W., Straszak T.: Introducing a unified requirements specification language, Proc. CEE-SET’2007, Soft. Eng. in Progress, 172–183. Nakom, 2007.

[12] Steinberg D., Budinsky F., Paternostro M., Merks E:. EMF: Eclipse Modeling Framework 2.0, Addison-Wesley Prof. , 2nd edition, 2009.

[13] Graham I. M.: Task Scripts, Use Cases and Scenarios in Object-Oriented Analysis, Object-Oriented Systems, 3(3), 123–142, 1996.

[14] Śmiałek M., Bojarski J., Nowakowski W., Ambroziewicz A., Straszak T.: Complementary use case scenario representations based on domain vocabularies, LNCS, MODELS'07, 4735, 544–558, 2007.

[15] Śmiałek M., Straszak T.: Facilitating transition from requirements to code with the ReDSeeDS tool, IEEE, 20th Requirements Engineering Conference, 321–322., 2012.

Chapter 9
Mutation testing of ASP.NET MVC

Mutation testing deals with assessing and improving quality of a test suite for a computer program. The range and effectiveness of the method depends on the types of modifications injected by mutation operators. We have checked whether mutation testing technique can be used to evaluate test cases for ASP. NET MVC-based web applications. Several new specific mutation operators were created and discussed. The operator judgment was experimentally verified with the mutation tool implementing the operators in the Common Intermediate Language (CIL) of .NET. The results show that mutation testing can be successfully applied to an application running on a web server, but execution times of functional tests can be long.

9.1.Introduction

Mutation testing is a process that can be used to measure quality of a test suite for a computer program [4]. It is based on injecting deliberate mistakes into the application code and testing the modified program to gain information about insufficient and missing tests. Algorithms used to create modifications (mutation operators) can be devoted to general features of a programming language such as logical expressions, or object-oriented characteristics. However, specific application technology, such as web processing also requires comprehensive testing, which could be verified with the mutation approach. The ASP.NET MVC programming environment was chosen for evaluation. This framework is a set of libraries for creation of easily-tested web applications using the Model-View-Controller design pattern [5,11].

We proposed several specialized mutation operators that can be applied in the ASP.NET MVC applications at the Common Intermediate Language (CIL) code originated from the C# source code. The operators were implemented in the mutation tool and experimentally evaluated. In experiments two common methods of application testing were taken into account: unit tests and functional tests run in a web browser.

9.2.Related work

Mutation testing was applied for different general purpose languages as well as specific domain languages [4]. Mutation operators related to .NET platform were developed at two code levels, with changes provided into C# source code or into lower level of the Common Intermediate Language (CIL).

General purpose structural mutation operators are implemented in the Nester tool [9]. The simple C# code modification rules are defined in regular expressions or XML document and can result in invalid mutants. The tool is not further developed. PexMutator [10] cooperates with the Pex extension of the Microsoft Visual Studio. It injects several structural changes into Intermediate Language. The mutated code is verified with tests automatically generated by Pex. CREAM (CREAtor of Mutants) was the first mutation testing tool dealing with object-oriented mutation operators for C# programs [1, 2]. Faults are injected into the C# code in the form of a syntax tree which is an output of the parser analysis. The current - third version supports 8 standard and 18 object-oriented mutation operators of C#. Mutations of Intermediate Language of .NET for programs originated from C# are introduced by the ILMutator prototype [3]. It implements 10 object-oriented and C# specific mutation operators.

Mutation testing was considered for web applications based on the ASP.NET Web Forms [6]. Though, applications using this former library have less test facilities and do not support the MVC pattern that is fundamental for mutation operators aimed at ASP.NET MVC. Advantages and disadvantages of integration and unit testing of ASP.NET MVC are discussed in [12].

9.3.Mutation operators for ASP.NET MVC framework

The ASP.NET MVC framework is a set of libraries supporting building of highly testable Internet applications based on the MVC (Model-View-Controller) architectural pattern [5, 11]. It combines programming paradigms common to Ruby on Rails, such as conventions over configuration, model binding and code simplicity, with the ASP.NET web technology of Microsoft (running on .NET framework).

The MVC architectural pattern separates an application into three main components: the model, the view, and the controller. In the framework, URL requests are mapped to controller classes and their methods. The controller handles and responds to user input and interactions. The controller performs operations on the model, and forwards a response e.g. a view to the user. Action methods (also called 'actions') are controller methods that can handle HTTP requests. They are recognized by their return type – deriving from *ActionResult*. The platform manages and calls specific actions to handle incoming requests.

Views are components providing generic data for presentation of web pages. In the framework, views are files returned by controller actions. The files consist of HTML code, combined with the source code of an imperative language of .NET - usually C#.

Model objects implement the logic for the business data domain. They often cooperate with the data base that stores the model data.

Separation of components and loose coupling of controllers with the execution platform encourage application testability. In unit tests, we can create controller objects, call their methods and verify results.

Mutation operators devoted to selected features of a programming technology should take into account various criteria, such as:

- a place of a change can be easily identified in the code,
- a code modification can be straightforwardly realized,
- a modified code is not detected by all tests,
- a mutation mimics a mistake that can be commonly made by a software developer.

We propose six new mutation operators for ASP.NET MVC that can be implemented at the CIL level. Selected mutation operators are illustrated by examples in the C# code corresponding to actual CIL code on which the mutation operators operate. In other cases code examples are omitted due to brevity reasons. Full examples are available in the thesis [13]. The following sections present mutation operators grouped by area of application.

9.3.1. Modifications of Model Binding

Values of client requests can be automatically adjusted to action parameters. A request is passed to a method if its name is identical to the name of the action parameter.

CAPN - Change Action Parameter Name is a mutation operator that changes the name of an action parameter. The name is substituted by a dummy name such as “mutatedParameterName#”, where # stands for an order number (Listing 9.1). In consequence, a request value for the action parameter will not be found during a mutant execution, unless a default value was defined. The result of this mutation depends on the parameter type. If the parameter is of reference type, it will be set to *null* and will probably cause a fault of the method. In case of a value parameter, an exception will be raised immediately.

This mutation can be easily introduced in the intermediate language. It is more complicated when applied in the C# code due to usage of optional parameters. In C# the whole project has to be searched for occurrence of the method calls (expected in unit tests) in order to ensure a compliable code.

```
// Before mutation - C# code
public ActionResult Edit(string name)            { ... }

// After mutation - C# code
public ActionResult Edit(string mutatedParameterName1)   { ... }
```

Listing 9.1. Example of CAPN operator - Change Action Parameter Name

9.3.2. Modifications of Action Attributes

There are two kinds of C# attributes that are placed before action methods: method selectors and filters. A programmer can use attributes delivered by the platform or create their own attributes.

Method selectors are used for identification of an action which will be executed after a request delivery. One of such attributes is *ActionNameAttribute* that changes a default action name, which is the name of a method, into a given name.

Filters make actions to be constrained with additional restrictions. Filter attributes can be placed before a controller class, thus influencing all actions of the controller. Among other filters of the framework, we can use *AuthorizeAttribute* for an action that has to be authorized, or *HandleErrorAttribute* stating what should be done when an exception was raised.

Attributes have influence on application execution only if it is executed on a server. Therefore the most obvious tests that verify usage of attributes are functional tests run in a web browser. Using unit tests a presence and a state of an attribute can be verified.

SWAN - Swap Action Names could be a mutation operator that swaps names of two actions through interchange of *ActionName* attributes. In result, in all cases when one action should be executed another action is raised. In order to have a consistent code, both actions should have the same number of parameters of the same types. Moreover, action names can be checked by a compiler, e.g. while calling *RedirectToAction* method, and the mistake can be easily detected.

RAAT - Remove Authorize Attribute - is a mutation operator that removes *Authorize* attribute placed before an action or a controller. Therefore the action or all actions of the controller can be called by an anonymous client.

This mutation checks an important feature of an application concerning its security. In many programs, it is easy to be applied both in C# and CIL code. However, *Authorize* attribute can be extended by inheritance with additional functionality or other authorization policy. In such cases the removal of the attribute should be waived.

9.3.3. Modifications of Action Results

An action of a controller returns a value describing a server answer to a client request. There are different types of such answers inherited from the *ActionResult* class, for example: *ViewResult* - a view is generated, *RedirectResult* - redirection of a client to another address, *JsonResult* - a return value is in JavaScript Object Notation, *FileResult* - a file is returned. Methods of controller support creating of these answers.

An application changes its behavior if a value returned by an action is modified. The mutation is limited for the cases when the return value inherits from the *ActionResult* class, which is a typical solution.

RVRA - Replace View with RedirectToAction - is a mutation operator that changes an object returned by a controller action; *RedirectToActionResult* is returned instead of *ViewResult* (Listing 9.2). The mutation can be detected by tests that check a type of an object returned by an action.

```
// Before mutation - C# code
public ActionResult ViewOrRedirect(object obj)
{   return base.View(obj);    }

// After mutation - C# code
public ActionResult ViewOrRedirect(object obj)
{  return base.RedirectToAction("Index");   }
```

Listing 9.2. Example of RVRA operator - Replace View with RedirectToAction

CRAT - Change RedirectToAction Target - is a mutation operator that changes a target action being a redirection method call parameter. The mutation can be implemented by substitution of a string identifying a target action.

In the selected solution the name is substituted by a dummy action *"MutatedIrrelevantActionName*". Usage of a dummy action is easy to be implemented and it is irrelevant whether the action redirected to exists or not. The action will not be found and will cause an error when run on a web server. However in unit tests this will not be the case and a user must check the *ActionResult* object for valid action name.

9.3.4. Modifications of Route Mapping

URL routing is used for mapping incoming URL requests to the appropriate controllers and their actions. The routing engine parses variables defined in the URL and the framework passes the parameter values to the controllers.

CMRA - Change MapRoute Address Pattern - is a mutation operator that changes an URL address. The string defining the URL pattern is substituted by a dummy one, e.g. "*MutatedString*". Therefore the route will be not corresponding to any incoming request. One of other existing routes will be used and as a result the appropriate controller might not be found.

The basic rule of the mutation is easily implemented. However, there are many overloaded forms of the *MapRoute* method. Extension of the mutation operator to all of them requires investigation of many possible parameter combinations. The CMRA operator is reasonable for bigger projects with many routes applied. In a small project a routing mistake can be easily detected by a developer.

9.4.Experimental evaluation of ASP.NET MVC mutation operators

Mutation experiments on the above discussed mutation operators were performed with the VisualMutator tool [13]. This tool was developed as a Visual Studio extension and provides an expansible framework for mutation testing at the CIL level. Tight coupling with the Visual Studio development framework makes the mutation testing process efficient, as the program under test is compiled only once and mutants can be generated fast.

Two subjects based on the ASP.NET MVC platform were evaluated in experiments (Table 9.1). Their open source code is available on the codeplex.com service. The first subject is NerdDinner [8] - an open source project that helps Internet people plan get-togethers. It utilizes the authorization system based on the Open ID standard and local accounts. The application also uses Bing search engine, geolocation and RSS feeds. NerDinner is distributed with a set of unit tests. The second subject of experiments is MVC Music Store [7] - a store which sells music albums online. This application was tested with functional test cases that run in a web browser implemented as control instructions of the WebDriver library.

Table 9.1. Subjects of experiments

	NerdDinner		Music Store	
	Applic.	Test cases	Applic.	Test cases
LOC without comments	730	461	195	61
Type number	71	20	27	2
Method number	399	156	172	17

The basic metrics of the applications and their test cases are summarized in Table 9.1. The metrics were measured with the NDepend tool. In Music Store, big samples of exemplary data included in the program were omitted.

Results of the test ability to detect faults injected by the mutation operators are shown in Table 9.2. In case of NerdDinner mutants of only two operators were killed by unit tests. Operators RVRA and CRAT modify results returned by actions, which is usually covered by unit tests. Equally important is verification of route mapping (CMRA) that is not covered by the tests designed for the application. Other mutants are not easily killed by unit tests unless the reflection mechanism was applied.

Tests of Music Store run were more effective in killing mutants. They required less code (Table 9.1) but were run in the web browser and took more time. An average test time of a mutant was equal 1.9 s for NerdDinner with unit tests run with NUnit, whereas 26.2 s for Music Store mutants run in the ASP.NET Development Server and functional tests executed with the assistance of VS MsTest.

Table 9.2. Mutation testing results

Mutation operators	NerdDinner		Music Store	
	mutant number	killed	mutant number	killed
CAPN	25	0	14	7
RVRA	37	22	18	6
CRAT	11	5	10	3
SWAN	15	0	8	6
RAAT	13	0	4	1
CMRA	3	0	1	1
Sum	104	27	55	24

9.5.Conclusions

We have shown how the mutation testing approach can be applied for the ASP.NET MVC-based web services.

Efficiency of a unit test suite in the respect of the considered mechanism verification was not very high (mutation score about 26%). However, it is difficult to cover by unit tests all mechanisms utilized by an application run on a server. Better mutation results (44%) with mutants killed of all fault types gave test cases run in a web browser but their execution times were significantly longer.

In many cases, the functionality of presented operators can be approximated by standard and object mutation operators for C# language. However it can be assumed that part of possible programmer error space will not be covered in that case, due to differences of ASP.NET MVC-based application and standard desktop application. The specific operators for the platform should operate on higher level of abstraction, making use of concepts of the platform and the language. This puts them to good use along standard and object operators. Nevertheless, usability of each operator should be analyzed to avoid duplicating functionality of classic operators.

Some faults, as e.g. injected by RAAT operator, can be easily detected by test cases run in a web browser. On the other hand simple unit tests do not kill such mutants, which might be treated as equivalent in their context. These mutants can be killed by unit tests with the usage of meta-programming techniques, such as reflection, that allow investigating and modifying a program during its run. An open question remains whether an application has to be run on a web server or should we accept usage of meta-programming in unit tests. In the first case, the long execution time might make the entire process impossible to use efficiently with large number of tests or mutants. In the latter case, interesting consequences of such a decision emerge. The mutant equivalence is then relative, depending on the testing approach. If we allow the usage of meta-programming techniques, no mutant with changed code can be considered equivalent, as the modification can always be detected by static analysis and not program behavior.

The VisualMutator tool is currently extended with selected standard and object-oriented mutation operators for C# language. It is also planned to be used in evaluation of automatically developed mutation-based test cases.

Bibliography

[1] CREAM – Creator of Mutants, http://galera.ii.pw.edu.pl/~adr/CREAM/ [access: 2013].

[2] Derezińska A., Szustek A.: Object-oriented testing capabilities and performance evaluation of the C# mutation system, Szmuc T., Szpyrka M., Zendulka J. (eds.), LNCS, vol. 7054, Springer, 2012, pp. 229-242.

[3] Derezińska A. Kowalski K.: Object-Oriented Mutation applied in Common Intermediate Language programs originated from C#", Proc. of IEEE 4th Inter. Conf. Software Testing Verification and Validation Workshops (ICSTW), IEEE Comp. Soc., 2011, pp. 342 - 350.

[4] Jia Y., Harman M.: An analysis and survey of the development of mutation testing, IEEE Transactions of Software Engineering, Vol. 37, No. 5 Sep/Oct 2011, pp. 649-678.

[5] Madeyski L., Stochmialek M.: Architectural design of modern web applications, Foundations of Computing and Decision Sciences, Vol. 30, No 1, 2005, pp. 49-60.

[6] Mansour N., Houri M.: Testing web applications, Information and Software Technology, vol. 48, issue 1, Jan 2006, pp. 31-42.

[7] MVC Music Store, http://www.asp.net/mvc/tutorials/mvc-music-store.

[8] NerdDinner, http://nerddinner.com [access: 2013].

[9] Nester, http://nester.sourceforge.net [access: 2013].

[10] PexMutator, http://pexase.codeplex.com/wikipage?title=PexMutator [access: 2013].

[11] Sanderson S.: Pro ASP.NET MVC 2, Apress, 2010 .

[12] Smirnow A.: Automated testing of ASP.NET MVC applications, Methods & Tools, Vol. 20, No 1, 2012, pp. 13-18.

[13] Trzpil P.: Mutation testing in ASP.NET MVC, Bach. Thesis, Institute of Computer Science, Warsaw University of Technology, 2012.

Chapter 10
SOA System Evolution Differential Evaluation

This chapter describes evaluation of SOA implementation in an organization that consist of two parts - mobile and fixed telecommunication operator. Both parts started to implement SOA in one point of time using similar procedures and principles. After a few years of implementations both parts reached different levels of SOA maturity and different results of SOA rollout. The chapter contains differential evaluation of these implementations. On the basis of the case are examined critical success factors of SOA implementations and some general observations are concluded.

10.1. Introduction

Service Oriented Architecture [7] reached certain level of maturity and became reliable approach to integrate complex IT systems. Many organizations adopted SOA principles and that has created an opportunity to evaluate real results of implementation of theoretical concepts. This chapter contains a study of SOA concept implementation in an organization in quite specific circumstances that enable comparing analysis of evaluation of SOA based system. Evolution begins in similar starting points in two similar parts of the organizations but reaches two substantially different states. The research described in this chapter determines success factor of SOA evolution in this particular situation and states general thesis on SOA evolution.

10.2. Related work

In [2] authors are analyzing different methods of evaluation of software architecture. Another approach represents SACAM method introduced in [8], that is devoted to compare different architectures. However, none of above takes into account SOA properties of analyzed architecture.

In [4] authors are presenting benefits of implementing SOA architecture comparing to non SOA approach on experimental application. Authors are focusing on technical and architectural benefits. Comparison described in this chapter concentrates on business benefits of SOA architecture. Business benefits should be expressed in suitable metrics that shows business value that SOA delivers. Metrics for evaluating SOA quality are described in [5] and [6].

This chapter considers SOA system development as an long-term and evolutional process. Broader context of such approach is described in [10].

10.3. Organization Context

During the early phase of SOA implementation, the two considering organizations operated as a separate companies, formally belonging to one capital group. The first organization was a mobile operator, the second – an operator of fixed telephony and Internet. A brief summary of the characteristics of the analyzed organizations actual for the considering moment are presented in Table 10.1. Although many elements of this two organizations was comparable or even shared in those days – the telecommunication industry, ownership relations, exchange of personnel – both companies remained different in terms of internal organization and external relations.

The fixed part "inherited" an extensive organizational structure and resources, not necessarily meeting the actual needs of the business. It was of a significant meaning to be subject of regulations office, on many issues limiting freedom of action. The mobile operator started without any burden. The company was built from the ground up, from the beginning tailored to the needs arising from the ongoing business model.

Table 10.1. Basic facts about considering companies

	Mobile	Fixed
Year of establishing	1991	1991*
Employment in 2005	2730	27500
Clients at the end of 2005	8,5 mln	10,5 mln

* Privatizations of a state-owned company

At the start of the implementation of the SOA, the two companies already had existing IT environment - see Table 10.2. In the area of SOA implementation, the overall level of both companies pursue at this time comparable business functions supported by IT.

Table 10.2. Year of building main IT systems in considering companies

	Mobile	Fixed
CRM	2003 – 2005	2002 – 2003
Billing	1997 – 1998	1999 – 2003
Self care	2003 – 2004	2004 – 2005

Customer Domain functions was chosen for the first SOA services. The implementation of SOA projects in both organizations were very similar – see Table 10.3.

The CRM environment in the fixed operator was built around a number of COTS components. The expansion of the CRM solution included integration with the billing system and with the main networking systems (OSS). Integration solution was based on IBM MQ Series. Integration services were used only to implement the functions controlled by the CRM. The environment was build centrally around CRM.

Table 10.3. Basic facts about SOA development

	Mobile	Fixed
Decision about SOA	2002	2003
First group of services delivered	In 2003 for SMS self-service area	In 2004 for SelfCare area
Integration technology	Software AG webMethods 6	Software AG webMethods 6
Main operational IT systems covered by SOA solution	CRM, Billing, OSS, Self care	CRM, Billing, OSS, Self care

In the second phase, the implementation of self-service solutions (IVR and Portal) took place, followed by introducing the second integration platform based on web-Methods EAI. The existing integration services have been transferred to the new plat-form. Finally, the period of early SOA architecture ended with an integrated environment in which the three major functional areas (CRM including self-service, billing and OSS) were integrated on a common integration platform. It is worth noting that the integration services, which if implemented, largely took the nature of point-to-point. This was due to low flexibility of major systems (billing, OSS), which prepared own APIs, which strongly underlined the specificity of their internal implementation. It had a certain impact on the further development of SOA.

Introducing SOA in the mobile operator had two main phases. The main objective of the first phase was to build the enterprise integration platform and the development of a standard method for SOA approach. The effect was the optimization of the processes using different, heterogeneous IT systems. Additional value was to gain knowledge of how long it should take to build a new solution in the context of future projects. Another important result of the work was to identify the principles of capacity planning for the integration platform. At that time, methods and standards for SOA solutions was developed and adopted. In the second phase of the SOA introducing, the integrated environment has been enhanced with the newly implemented system – CRM. Unlike the fixed operator, in that case CRM used existing services.

10.4. Results

10.4.1. Architectural perspective

Abstract pattern of SOA can be presented in the form of layered architecture, as pro-posed by the Open Group (see Fig. 10.1.) [9]. Comparison of SOA environments will be carried out on the basis of indicators for selected layers:

- The operating system layer, which includes all custom or packaged application assets in the application portfolio running in an IT operating environment, supporting business activities. Those systems include: existing applications and solutions package ERP and CRM packages, custom monolithic existing applications and legacy applications.

- The Service Component Layer, which contains software components, each of which provides the implementation or realization for services and their operations, hence the name Service Component. The layer also contains the Functional and Technical Components that facilitate a Service Component to realize one or more services.
- The service layer (services) which consists of all the services defined within the SOA. This layer can be thought of as containing the service descriptions for business capabilities and services as well as their IT manifestation during design time, as well as service contract and descriptions that will be used at runtime.

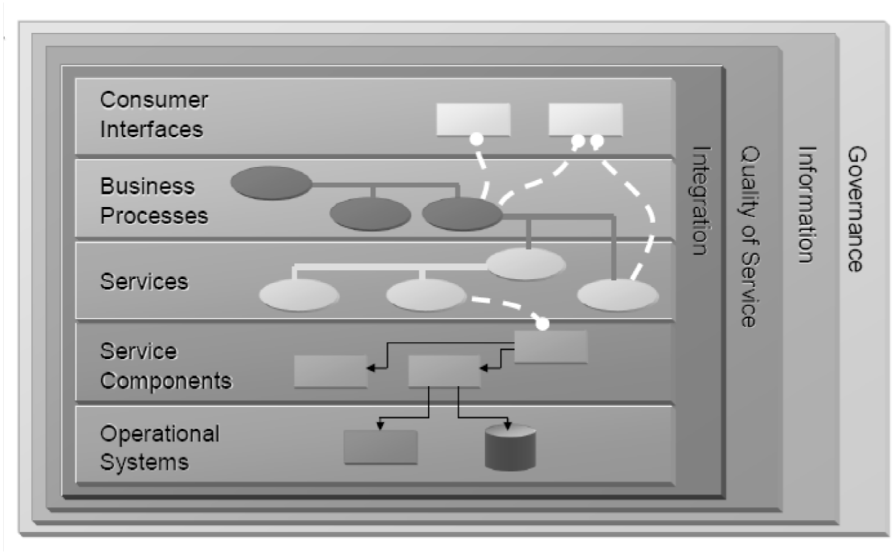


Fig. 10.3.SOA architecture pattern from the Open Group. Source: [9]

Comparison of the current state of both SOA environments is presented using two groups of characteristics. The first group includes the basic measures for SOA architecture:

- number of systems in the environment for Operational Systems Layer – a system is considered as belonging to the environment, when it provides its functions and/or use of functions of other systems;
- number of services for Services Layer – a service is considered as a function of operational system (one or more) that is made available to

consumers (from Business Process and/or Consumer Interfaces Layer);

- number of services reused by at least one consumer – service is reused in the process supported by more than one operational system (see Fig. 10.2 case A);
- number of services realizing point to point integration (between two operational systems) – occurs when a single feature of one operational system is available to one other system (see Fig. 10.2 case B);
- number of services composing functions from a few operational systems for exactly one consumer – composing functions of several operational systems for only one system (see Fig. 10.2 case C).

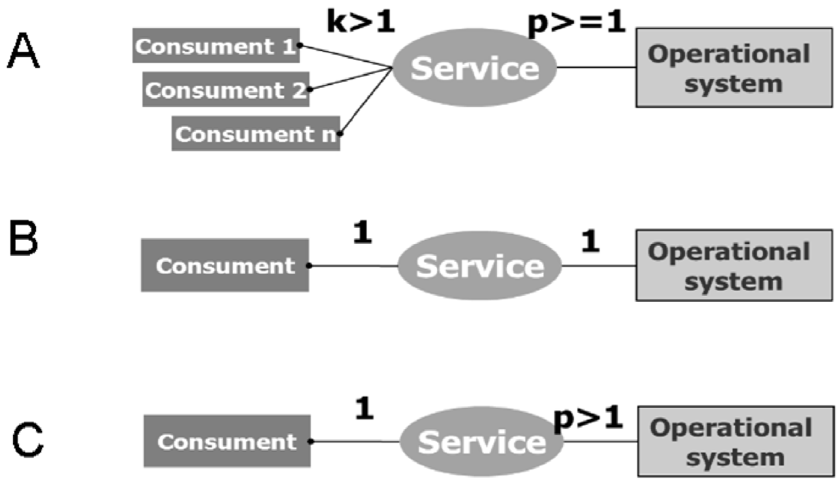


Fig. 10.4. SOA measures: A – of services reused by at least one consumer; B – services realizing point to point integration; C – services composing functions from a few operational systems for exactly one consumer.

Summary of measurements for both environments is presented in Table 10.4.

Table 10.4. Main architecture measures

	Mobile	Fixed
Operational systems quantity	68	71
Services quantity	462	380
Share of services reused by at least one consumer	59%	29%
Share of services realizing point to point integration (between two operational systems)	25%	52%
Share of services composing functions from a few operational systems for exactly one consumer	16%	19%

The second group of measured characteristics concerns the use of services during the business processes in the IT environment. The test statistics have been collected through the EAI infrastructure, during one month period. The results are presented in Table 10.5.

Table 10.5. Main runtime measures

	Mobile	Fixed
All services requests quantity	835 mln	13 mln
Share of requests per most used service	32%	15%
Share of requests per 10 most used services	65%	61,5%
Share of requests per 20 most used services	80%	75,5%

The main difference in runtime measure is visible at the first sight – the difference in the requests quantities can be observed. Nevertheless, some additional explanation has to be made. It should be underlined that heavy service usage in the mobile operator is the consequence of the customers' usage model. The mobile customers use the growing number of mobile devices for self service functions, exposed by the operator. Each of these functions work with SOA service in the background IT infrastructure. The fixed internet customers from the fixed operator do not practice the self service functions widely.

10.4.2. Non architectural perspective

Non architectural perspective shows that costs of IT development in fixed part are significantly higher (per function point) than in mobile part. Also average Time To Market (development time from request to go-live) is much longer in fixed than in mobile.

There is also meaningful difference in organizational culture between mobile and fixed part. Mobile part as an organization is generally more agile and changeable than fixed part.

As parts of one corporation the mobile and fixed have common software engineering methods, IT development procedures and architectural principles. There is also centralized enterprise architecture organization that controls both IT environments.

10.5. Conclusions

Development of SOA based IT system has evolutionary nature. System evolves by slight changes which superposition in long term builds significant change. Collected data can help in identification important factors that influent such evolution.

Service reusability was chosen as main SOA quality metrics. It has business value justification – the more frequent service is reused the more cost effective SOA implementation is. From that point of view, better quality of SOA architecture was gained in mobile part.

Architectural measures presented in Table 10.4 shows that in both organization, similar effort was put in building services (similar number of services), however in fixed part most of services were misused as point-to-point integration technology. This usage is against to SOA paradigm [7] stating that service should present agnostic behavior. This observation suggest that improper approach to SOA analysis and design exists in fixed part, because it does not take into account SOA paradigms but treat services as pure technology assets.

Runtime characteristics gathered in Table 10.5 can be partly explained by difference in behavior of mobile part customers, that was described in section 10.4.1. But even taking this fact into account there is still significantly bigger overall usage of services in mobile than in fixed environment. This fact can be explained by better design of services that are designed in reusable fashion. Another possible reason is better organized and accessible services directory in mobile part.

There is clear correlation between architectural and organizational aspects of the mobile and fixed SOA evolution results. More agile and adaptive mobile part is better in using SOA based system and gains better results that fixed part. But it is not possible to find a causal relationship. This conclusion is consistent with Architectural Business Cycle [11] that states that architecture influent business on the same level as business influent architecture.

10.6. Further work

Authors believe that above presented observations are general and can be applied in any IT environment adapting SOA. Further work on the following subject is focused on drilling down data and finding another correlations and observations on SOA success factors in this particular case.

Another branch of our research is to attempt creating methodology of comparative evaluation of SOA architecture in organization (or more general any fashion architecture). Such evaluation can be useful in finding specific success factors causing that in one part or in one environment SOA is better than in another. This situation occurs in organizations that parallel develop a few environments or may have inherited different environment after major transformations (e.g. fusion of two companies). The methodology should help to identify disjoint but comparable parts.

Bibliography

[1] Aier S., Bucher T., Winter R, Critical Success Factors of Service Orientation in Information Systems Engineering, Business & Information Systems Engineering, 2011.

[2] Muhammad A. B., Liming Z., Ross J. A Framework for Classifying and Comparing Software Architecture Evaluation Methods. In Proceedings of the 2004 Australian Software Engineering Conference (ASWEC '04). IEEE Computer Society , 2004.

[3] Zhou N., Zhang, L.: Analytic Architecture Assessment in SOA Solution Design and its Engineering Application.. In: ICWS : IEEE, S. 807-814, 2009.

[4] Offermann P, Hoffmann M, Bub U, Benefits of SOA: Evaluation of an implemented scenario against alternative architectures, Workshops Proceedings of the 12th IEEE International Enterprise Distributed Object Computing Conference, EDOC, 2009.

[5] Karthikeyan T., Geetha J.: A Study and Critical Survey on Service Reusability Metrics, International Journal of Information Technology and Computer Science (IJITCS) vol 4, 2012.

[6] Aier S., Ahrens M., Stutz M., Bub U.: Deriving SOA Evaluation Metrics in an Enterprise Architecture Context. In Service-Oriented Computing - ICSOC 2007 Workshops, Elisabetta Nitto and Matei Ripeanu (Eds.). Lecture Notes In Computer Science, Vol. 4907. Springer-Verlag, Berlin, Heidelberg 224-233, 2009.

[7] Erl T.: SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl). Prentice Hall PTR, 2007.

[8] Stoermer, C., Bachmann, F., Verhoef, C.:. SACAM: The Software Architecture Comparison Analysis Method (Technical Report CMU/SEI-2003-TR-006). Pittsburgh: Software Engineering Institute, Carnegie Mellon University. http://www.sei.cmu.edu/library/abstracts/reports/03tr006.cfm , 2003.

[9] The Open Group SOA Reference Architecture, Technical Standard, The Open Group, 2011.

[10] Zalewski, A., Szlenk, M., Kijas, S.: An Evolution Process for Service-Oriented Systems. Computer Science Journal, vol. 13, no. 4, pp. 71-86. AGH University of Science and Technology , 2012.

[11] Bass L., Clements P., Kazman R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA., 1998.

Chapter 11

CoJaq: a hierarchical view on the Java bytecode formalised in Coq

We present a design for a formalisation of the whole set of 200 Java bytecode instructions. It consists of a concise set of abstract, generic instructions that can be specialised to obtain any particular bytecode instruction. In this way one can work with a manageable set of instructions to prove general facts about the Java bytecode, but at the same time all the bytecode instructions are available to enable direct verification of actual bytecode programs. A considerable part of the design has been realised in Coq proof assistant.

11.1. Introduction

There are many tools to manipulate programs depending on their semantic properties (e.g. program translators, code refactoring tools, code optimisers) but their trustworthiness is usually based on producer's reputation. In order to avoid this, but guarantee correctness of such a tool one needs a formalisation of the semantics of the programming language the tool works with.

In this chapter we provide a detailed and in a considerable part realised design of a formalisation of the Java Virtual Machine language (JVML) semantics. The key motivation for this project, called *CoJaq*[5], is to create a platform whereboth real programs can be verified and metatheoretical properties of the language can be shown. In this way the properties are proved about the same language as the actual programs are verified. Thus the need for modelling of the language features, that may lead to inaccuracies or may be impaired by inadequate treatment of the subject issue, is eliminated.

Formalisation of a real programming language is difficult. It is so since (a) its informal description spans over hundreds of pages and it is easy to omit

[5] Available at http://cojaq.mimuw.edu.pl

some details that are surrounded with pragmatic details that have little to do with the actual behaviour; (b) crucial properties of the language are sometimes expressed implicitly and should be inferred from hints spread over many pages; (c) the descriptions in natural language are often ambiguous; (d) formalisation of such large entities easily goes beyond human comprehension so a non-trivial structuralisation effort is needed to achieve the final result; and (e) in case interactive theorem provers are used as support, their limitations can be reached. In fact, such a formalisation effort can be viewed as an implementation of an interpreter for the subject language in the anguage of the formalisation, being either set theory, arithmetic or Coq etc. logic. As a result formalisations require significant effort very similar to the effort of programmers.

The key achievements of the presented formalisation are:

1. Almost full Java bytecode instruction set has been modelled in Coq. This creates a platform that can be used at the same time to verify programs and to make feasible metatheoretical proofs for the language.
2. The semantics is extensive and detailed – although it is not yet complete, it covers a significant number of instructions and contains simplified formalisation for all aspects of the JVML in such a way that it can be extended to cover full functionality in future versions.
3. A static semantic based upon types of values is developed. This static semantics is proved to be sound and complete with regard to the dynamic one.
4. We proved a general theorem that programs for which Hoare-style logic rules apply at each step are partially correct.

The last two features were provided for the instructions that use only frame runtime structures (i.e. structures such as operand stack and local variables table, but not heap). We omit their details in the this chapter.

The chapter is structured as follows. Section 11.2 describes the key design features of our formalisation. In Section 11.3 we report the related work and we conclude in Section 11.4.

11.2. Key Ideas

A formalisation of an industry standard specification for a programming language is a costly task so it should be done with a particular set of requirements in mind. Here is a summary of the requirements we worked with during the formalisation of *CoJaq* and their impact on the design.

Multiple specification interpretations The formalisation should take into account the possibility to interpret the original specification in different yet plausible ways. The actual implementations of the standard may differ in their operation since they have taken a different approach on the implementation of a particular notion described in the specification (e.g. they use a different scheduler which has the impact on the way programs are executed). The formalisation we want to obtain should allow one to prove correct statements that describe the operation of all plausible implementations.

To obtain this, we use three techniques. First, the semantics is written in the small-step fashion. The small-step fashion is more appropriate since the natural language semantics is also formulated in the small-step fashion. Moreover, the formulation of many metatheoretic properties (e.g. immutability, purity etc.) is easier in this style. Second, the semantics is formalised as a relation. In this way the semantics we propose can be non-deterministic in places where the description is, while this would be impossible in case the formulation in the form of function was chosen. Third, we use extensively the module system of Coq to separate different aspects of the virtual machine (see Fig. 11.1).

Manageable set of instructions In case a metatheoretical property should be proved for a system like this, one needs to make many proofs by induction over the structure of the language. When the language is big one must consider a big number of cases. This can be avoided when similar instructions are grouped in a hierarchical structure. With such a layout many cases can be discharged on higher levels of the hierarchy resulting in smaller proofs. We took this approach in our formalisation and present it in more detail below.

Static semantics The correct operation of the JVML semantics strongly relies on the assumption that bytecode instructions have arguments of appropriate types. Therefore, the operation of each instruction is accompanied by a careful description of the types for its input and results. In our formalisation we separated the description of the types from the description of the actual operation of the instructions and provide two different relations for the two aspects, proving their correspondence. We believe that many proofs can be simplified due to the choice since they will not have to manage the typing information.

Hoare-like logic We provided a systematic method of using Hoare-style logic rules in a shallow embedding fashion. In addition we proved a general theorem that programs for which these rules apply at each step are partially correct.

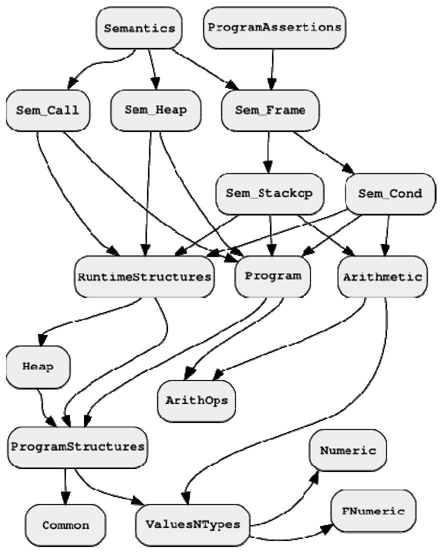


Fig. 11.1. Module dependencies in CoJaq

11.2.1. Hierarchy of Instructions

Although the number of bytecode instructions is very large, one can see that many instructions are similar to one another. We can distinguish the following simple situations contributing to the proliferation of instructions.

- A set of instructions performing the same operation for different data types, e.g. `iload`, `fload`, `aload`, and so on. Each instruction loads a local variable value and pushes it on the operand stack, but a single instruction is applicable only to its own type of values (`int`, `float`, and `ref`, respectively).

- "Shorthand instructions" are defined for some most widely used argument values, as predicted by the JVML designers. An example could be the set of instructions `iload_0`, `iload_1`, `iload_2`, `iload_3`.
- Instructions related to arithmetic and comparison often behave in similar way and differ only in an arithmetic operator. For example, `iadd`, `isub`, `imul`, and `idiv` all perform binary arithmetic operations on integers. They all pop two operands from the stack and push back one resulting value.

Each of the aforementioned cases can be simply "compressed" back to a single parametrised instruction. The examples mentioned in cases 1 and 2 can be covered by an abstract *load* instruction parametrised by a data kind (e.g. `int`) and a variable number. This one abstract instruction covers 25 JVML instructions. The case 3 can be factorised into an abstract instruction, parametrised by a data kind and a function on that kind, i.e. an abstract *binop* instruction with parameters such as addition, subtraction etc. This factorisation idea was used e.g. in Bicolano [9], where the number of instructions was reduced by almost 40%.

In [4] we decided to go one step further and factorise instructions according to runtime structures they operate on. We divided JVML instructions into twelve parametrised abstract instructions. In this chapter, we refine this approach by organising the instructions into a hierarchy that is also followed by the definition of operational semantics. The hierarchy is presented in Fig. 11.2. It is realised in Coq as a number of (non-recursive) inductive types. The topmost one is `TInstruction` with 6 constructors: `I_Throw`, `I_Monitor`, `I_Invoke`, `I_Return`, `I_Heap`, and `I_Frame`. The first four represent instructions with specific access to JVM data. The fourth one, `I_Heap`, represents instructions that operate on the heap without modifying the call stack (i.e. variants of get, put, new and array access). The fifth one, `I_Frame`, represents the largest family of instructions that operate on data in the method frame at the top of the method call stack of a thread.

11.2.2. Hierarchical Definition of Semantic

Our operational semantics follows the hierarchy of instructions. It can be seen already in a ``big picture" view of Coq modules, Fig. 11.1, where the structure of modules implementing semantics resembles the hierarchy of instruction representation. The hierarchy is designed in such a way that the relations defining semantics of abstract instructions which are lower in the instruction hierarchy operate on a smaller fragment on the JVM state – the one accessed by real instructions represented by the abstract one.

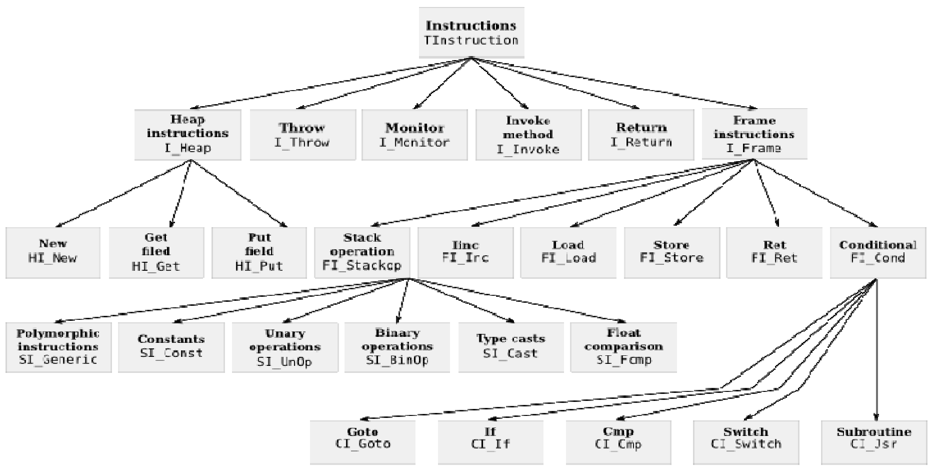


Fig. 11.2. Hierarchy of instruction abstractions

The hierarchical structure of semantics has at least three advantages. First of all, it prevents code duplication, as otherwise the `step` relation e.g. for all the `I_Frame` instructions would have almost identical premises corresponding to extracting the suitable fragment of the JVM state. Second, when proving some properties of the semantics, the necessarily large proof is also hierarchically organised and easier to manage than a big monolithic one. Moreover, if the property at hand is not relevant for a large part of instructions, chances are that many of the irrelevant instructions will be discharged at a high level of semantic hierarchy, e.g. one would discharge the whole `I_Frame` branch of the `step` relation and not many separate instructions one by one. The third advantage is the possibility to develop some proof techniques like VCGen, Hoare logic etc. only for fragments of the semantics, if the whole semantics is too complex to cover. The hierarchical structure of the semantics provides again, a natural delineation between parts to do and to ignore.

11.2.3. Program Verification

A systematic JVML program verification can be performed in the following way: one writes formulas that describe the states between every two consecutive bytecode instructions and then proves that starting from a state satisfying the formula before an instruction if the semantic step of the instruction is taken then the resulting state satisfies the formula after the instruction.

Consider the program in Fig. 11.3. It consists of initial assignments of constants to local variables and a loop that calculates the sum of first *n* odd numbers, which is equal to n^2. Its fragments in *CoJaq* looks as follows:

```
Definition code: TCode := codeFromList
  [(* 0*) I_Frame (FI_Stackop (SI_Const KInt zero));
   ...
   (* 6*) I_Frame (FI_Load KInt var0); (* start of the loop *)
   ...
   (*18*) I_Frame (FI_Cond (CI_Goto (offsetFromPosition 6%nat)));
   (*19*) I_Frame (FI_Load KInt var2) (* after the loop *)
   (*20*) I_Return (Some KInt) (* return *) ].
```

First of all, note that labels in bytecode are positions in bytes, whereas in the Coq they are consecutive numbers. Second, the *CoJaq* code is parametrised by *n*, which stands for 50 in Java and JVML. The proof of program correctness is of course done for arbitrary (but small enough) *n*.

For the proof we need to express properties describing the state before instructions of the program, e.g:

```
Definition s8_prop frame := pcToPosition (frameGetPC frame) = 8%nat
    /\ exists i, exists r, stack_values frame [n; i]
    /\ var_value frame var0 i /\ var_value frame var1 n
    /\ var_value frame var2 r /\ r = i*i /\ 0 <= i /\ i <= n.
```

The above definition says (i) that the program counter of the current frame is at position 8, (ii) that the values on the operand stack correspond to the values of appropriate local variables, and (iii) that the abstract loop invariant is satisfied, i.e. `r=i*i`, where `i` is in the appropriate range.

Once the state properties are defined, we prove a number of lemmas about transitions, e.g.

```
Lemma trans_7_8: forall frame frame', s7_prop frame →
                        SF.stepFrame code frame frame' → s8_prop frame'.
```

The proofs consist mostly in unfolding definitions, decomposing conjunctions and inverting inductive relations. They can be largely automated.

After proving transition lemmas, we can establish the partial correctness of the program, i.e, when it is started in the initial state and arrives after instruction 19 then the operand stack holds n^2:

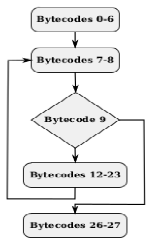
```
Theorem partial_correctness: forall frmF,
  pcToPosition (frameGetPC frmF) = 20%nat →
    SF.stepsFrame code frame0 frmF →
      exists res, frameGetLocalStack frmF = [(VInt res)] /\
             Num.toZ res = (n * n).
```

In this way we proved the desired property of the bytecode program in Fig. 11.3.

```
public static int m() {
  int i = 0;
  int n = 50;
  int r = 0;
  while (i < n) {
   r = i + r;
   i++;
   r = i + r;
  }
  return r; }
```

(a)

```
public static int m();
   0: iconst_0       // int i = 0;
   1: istore_0
   2: bipush 50      // int n = 50;
   4: istore_1
   5: iconst_0       // int r = 0;
   6: istore_2
   7: iload_0
   8: iload_1
   9: if_icmpge 26   // i >= n
  12: iload_0
  13: iload_2
  14: iadd           // i + r
  15: istore_2       // r = ...
  16: iinc 0, 1      // i++
  19: iload_0
  20: iload_2
  21: iadd           // i + r
  22: istore_2       // r = ...
  23: goto 7         //end of loop
  26: iload_2        // r is returned
  27: ireturn
```



(c) (b)

Fig. 11.3. An example of a method. (a) The Java source code of a method, (b) the corresponding bytecode, and (c) the control flow graph of the bytecode.

11.2.4. Missing Features in the Formalisation

The current version of *CoJaq*, does not cover all the details of the language. We do not handle all the aspects of advanced multithreading and Java memory model, 64-bit types, exceptions, and native methods.

In terms of the bytecode instructions, we do not handle at the top level of the categorisation the cases of `I_Throw` and `I_Monitor`. At lower levels of the formalisation we miss also the formalisation of `tableswitch` and `lookup-switch`, `jsr` and `ret`, as well as some of 64-bit instructions.

11.3. Related Work

We only present the most important formalisations here due to the space limit. A systematic reduction of a large set of JVML instructions to a small one by means of abstraction was given by Yelland [11]. He proposed a language μJVM with a modest set of instructions that transform program continuations. Next, a translation was provided for the actual bytecode instructions. The language of μJVM works on a different level of abstraction than *CoJaq* as instructions in *CoJaq* correspond in a hierarchical way to instructions in the JVML, while in the case of μJVM a translation is required.

Early efforts to mechanically formalise the JVML was done by Pusch [10], in Specware project [5], and by Bertot [3]. Leroy [7] proposed a more mature formalisation. It focuses on JavaCard version of JVML and offers a Coq formal proof that the JVML verifier preverification procedure are correct. Klein and Nipkow [6] proposed probably the most extensive work on the JVML verification. They defined in Isabelle/HOL a model of Java called Jinja and a formalisation of the JVM language model with 15 instructions covering low-level control flow, integer numeric operations, classes, arrays, methods, exceptions, casts, and bytecode subroutines.

A considerable fragment (over 70 instructions) of the JVML was modelled in largely flat fashion by Pichardie [9] in Coq. The work was similar in spirit to the one of Bertelsen [2] and modelled directly the instructions. The semantics was done both in the small-step and big-step fashion and the two were proved equivalent. This was probably the most ambitious and largely successful attempt to make a formal account of the full bytecode instruction set.

Atkey [1] formalised a fragment of JVML in Coq so that program extraction can be used to extract JVM implementation in Ocaml. In this way it is possible to efficiently validate the operational semantics in Coq against real JVMs and test if the results obtained in the two environments agree.

11.4. Conclusions

We grouped in our formalisation the JVML instructions based upon the way they operate on the runtime structures. In this way we obtained a hierarchical decomposition of the Java instruction set and formalised in Coq a big part of it. We believe that in this way it will be possible to both prove metatheoretic properties of the JVML and prove correctness of particular programs. The formalisation consist currently of over 7 KLOC of Coq files.

Acknowledgements

This work was partly supported by Polish government grant N N206 493138.

Bibliography

[1] R. Atkey. CoqJVM: An executable specification of the JVM using dependent types. Proc. of TYPES 2007, vol. 4941 LNCS, pp 18–32. Springer, 2008.

[2] P. Bertelsen. Dynamic semantics of Java bytecode. Future Gener. Comput. Syst., 16(7):841–850, 2000.

[3] Y. Bertot. Formalizing a JVML verifier for initialization in a theorem prover. Proc. of CAV'01, vol. 2102 LNCS, pp 14–24. Springer, 2001.

[4] J. Chrząszcz, P. Czarnik, and A. Schubert. A dozen instructions make Java

[5] bytecode. ENTCS, 264(4):19–34, 2011.

[6] A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. Proc. of DISCEX ’00. , pp 403–410, vol. 2, Los Alamitos, CA, USA, 2000. IEEE.

[7] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Trans. Program. Lang. Syst., 28(4):619–695, 2006.

[8] X. Leroy. Bytecode verification on Java smart cards. Softw. Pract. Exper., 32(4):319–340, 2002.

[9] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from http://mobius.inria.fr.

[10] D. Pichardie. Bicolano – Byte Code Language in Coq. http://mobius.inria.fr/bicolano. Summary appears in [8], 2006.

[11] C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In Rance Cleaveland, ed, Proc. of TACAS ’99, vol. 1579 LNCS, pp 89–103. Springer, 1999.

[12] Ph. M. Yelland. A compositional account of the Java virtual machine. In Proc. of POPL’99, pp 57–69, New York, NY, USA, 1999. ACM.

Chapter 12

Web-based Software Engineering Labs for Embedded and Cyberphysical Systems

To the memory of Professors
Leon Łukaszewicz,
Zdzisław Pawlak,
Roman Trechciński, and
Władysław M. Turski

This chapter presents an idea of web-based but still hands-on laboratories for teaching software engineering principles, as implemented in courses on development of embedded and cyberphysical systems in the Software Engineering Program at Florida Gulf Coast University. Since the concept of the labs is pretty straightforward and its technological principles have been outlined in previous publications, the paper touches upon technology and related pedagogical issues only briefly. Instead, attention is paid to the motivation for such labs and, even more so, to the significance and consequences of using new technology, in general, and applying it in education, in particular. Issues related to the concepts of Lewis Mumford's megamachine, Marshall McLuhan's medium as a message, and Clayton Christensen's disruptive technology are discussed in relation to the remote labs. Lab's pervasive and ubiquitous nature leads to the game changing concept of a lab-by-wire.

12.1. Introduction

In computing disciplines, hands-on software labs have always been essential at all levels of education, whether in computer science, computer engineering, software engineering, or information systems. With the advent of the Internet, virtual labs became popular allowing students to do work remotely, without the need of being physically present in the laboratories. However, virtual labs have been criticized that they were not necessarily effective in courses on embedded, real-time, or cyberphysical and other systems involving the operation of digital devices controlled by microprocessors or microcontrollers, where observing the effects of the software being developed on the device's operation is essential to the learning process.

It is relatively obvious that technology exists today, which allows students and other types of users do the software development work in their homes or offices, or maybe even on their mobile phones, and upload the executables to the remote devices to run them according to specifications. However, for a variety of reasons, this is not being done very effectively, or even actively pursued, in education. The thesis of this paper is that the respective change in educational practices is imminent, and it will have all signs of not just a major shift but that of a disruptive move, which will make a dramatic breakthrough in our lives as educators in computing disciplines.

The paper shows an example of a lab, in which students use a variety of platforms and networking protocols to develop software remotely to remotely control various embedded devices, and provides a broader perspective on the use of similar technologies in contemporary college education..

The rest of the chapter is structured as follows. The remaining part of this section begins with an overview of remote labs in other engineering and science disciplines. This is followed by a brief discussion of the nature of embedded/cyberphysical systems and an outline of the motivation for creating remote labs in software engineering. The next section presents the lab itself, as developed at FGCU, and outlines briefly the lessons learned and plans for expansion. The final section addresses the significance of remote labs and consequences of new technologies for the learning and teaching processes. The Conclusion section ends the paper.

12.1.1. Remote labs overview

The concept of an online or remote lab is derived from the idea of distance learning and has been present in education for over twenty years. A remote lab is a physical lab accessible and operated via the Internet. The first known attempt to address the issue of remote access to lab experiments dates back to 1991 [1].

Since then online labs have been used in science and engineering courses in a number of disciplines, from physics and chemistry to mechanical and electrical engineering, and other fields. Multiple survey papers have been published over the years [2-6] and two recent volumes of related articles describe current status of remote labs [7-8].

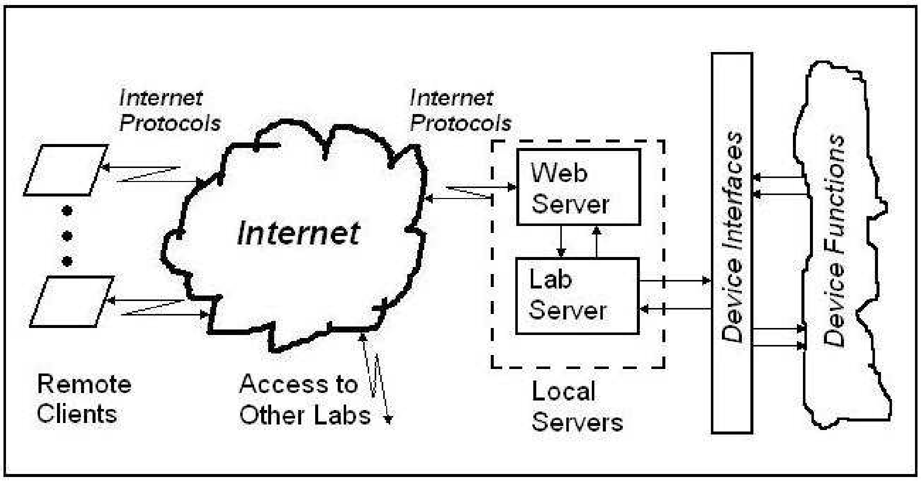


Fig. 12.1. Architectural components of a remote lab.

A simplified architecture of an online lab is shown in Figure 12.1. Multiple remote clients can access and potentially control remote devices via the Internet, utilizing respective network protocols. Device access is scheduled via their lab servers and respective interfaces, in coordination with traditional web servers providing Internet connectivity.

Nowadays, such labs are quite widespread in higher education and are generally divided into three overlapping categories: batch, sensor and interactive labs [8, p. 138]. *Batch labs* are labs where experiments are completely defined before the beginning of the experiment and run without additional intervention of a user. *Sensor labs* are those labs, which allow for one-way data transmission, but not full interaction, during experiments, so a user can monitor or analyze real-time data streams without influencing the phenomena being studied. *Interactive labs* are those, in which the user has the ability to fully interact with the experiment by monitoring and controlling selected parameters or behavior of the experiment during its execution.

These web-based or online labs are also *hands-on*, although this term may be a bit confusing. The confusion may be related to the proximity of equipment and student, implied by the term “hands-on”. Then, of course, the terms are contradictory. But when one talks about functionality, functions of a hands-on lab can be provided equally well by a web-based lab, which has been noticed before by some other authors [9].

From the perspective of Software Engineering, it is crucial to realize that all types of labs mentioned above are *non-invasive*, that is, software controlling the experiment never changes, only the parameters of the experiment can change via the interaction of a user with the experiment control software.

12.1.2. Embedded and cyberphysical systems

Embedded computer systems have their roots in control systems, long before digital computer control has been conceived [10]. The essential structure of a simple control system is shown in Figure 12.2. It consists of a *controller* and a controlled object (commonly termed a *plant*). The *sensor* installed on a plant delivers a measurement signal to the controller, which on this basis, and a prescribed plan (*setpoint*), takes a decision what value of a control signal to apply to an object via an *actuator*, to counteract potential variations caused by disturbances.

A typical example of this sort of control system, which we are all familiar with, either from our homes, offices or cars, is a temperature controller, otherwise known as a thermostat. Historically, the oldest known device, which applies this type of control principle is the Ktesibios water clock (third century B.C. [11]), stabilizing the water level to let it flow with constant rate, out of the tank to another tank at the lower level, to mark the passage of time.

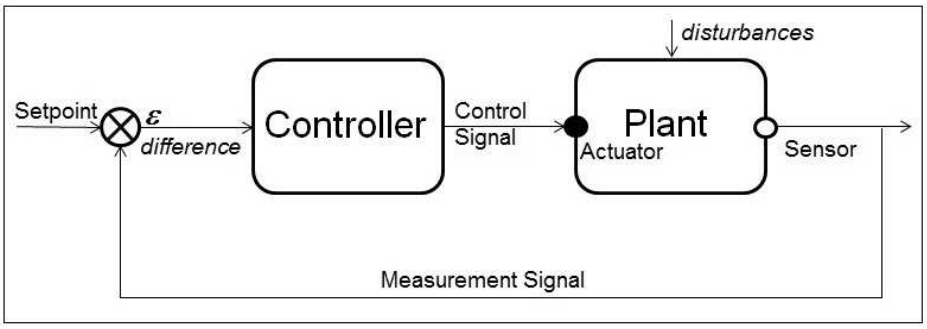


Fig. 12.2. Diagram of a simple control system.

When one relaxes some of the assumptions of a feedback controller, it is easy to derive the following variations of the control system:

- data acquisition system, when one breaks the control signal line, leaving only the measurement signal reach the controller (then the setpoint becomes irrelevant);
- programmed controller, when one breaks the feedback loop, leaving only control signal reach the plant and the setpoint to set parameters applied to the controller.

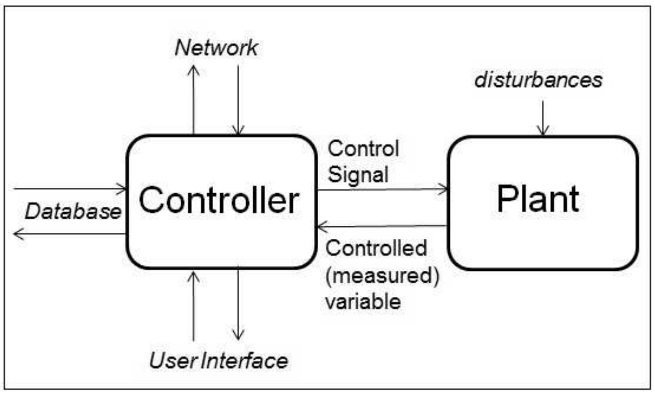


Fig. 12.3. High-level model of an embedded/cyberphysical system.

With the development of digital technologies, control systems became miniaturized and directly embedded into plants, that is, controlled objects. At the same time, they were expanded to include operator (user) interface, and their parameter values, such as a setpoint, expanded into more sophisticated data sets, soon to reach the size of true databases. This expanded embedded control system's structure is shown in Figure 12.3.

Once it became possible and desirable to implement network connectivity feature in embedded controllers, we have faced a new breed of systems called cyberphysical systems, which are in fact old embedded control systems enhanced by connectivity mechanisms. This is reflected in Figure 12.3 by a network interface.

Thus, for the purpose of this paper, a cyberphysical system can be defined as a computing system with access to physical measurement and control instruments as well as with connectivity to the Internet. As shown in Figure 12.3, in addition to a physical process and network interfaces, also user and database interfaces are present.

12.1.3. Motivation and nature of the remote software engineering lab

While the drive by science and engineering educators to access instruments and experiments remotely has led to a significant progress in designing and establishing remote labs, it is the fact of the matter that these labs have not been used in courses in software engineering. In this view, it would make sense to point out what kind of motivations do exist to implement remote labs in this discipline. We look into the following four aspects of the motivational context: professional, educational, innovational and societal.

Professional context. Professional motivation for developing such lab comes from a combination of two different perspectives: needs of the software engineering programs and trends in the design and application of embedded and cyberphysical systems.

First of all, modern software intensive embedded and cyberphysical systems are applied in the most demanding real-time safety-critical applications, such as flight control systems, accelerator control, road vehicle control, etc. They are all distributed and for proper operation require very different programming techniques than traditional systems. Typical software engineering curricula, however, rarely include respective methodologies of software development for such systems. If they do, their courses mostly concentrate on the specification and design aspects of software for distributed systems, but stop short of including thorough treatment of implementation and testing issues.

Secondly, the need for workforce with respective type of skills has been documented in professional practice. The most spectacular case had actually occurred as long ago as in 1997 during the Pathfinder mission to Mars [12]. In brief, a robotic device, which landed on Mars, got stuck due to an unidentified software problem, later during the mission recognized as, so called, priority inversion. Then, engineers at the ground control center corrected the software and re-uploaded it remotely to the device. This example gives an incredible boost to the need of acquiring respective knowledge and skills by software engineers.

Educational context. Well entrenched in their academic positions, college educators may not realize it, yet, that perhaps in the next decade or two the universities, as we know them, may disappear from the face of the earth. Even though this may seem a statement too dramatic to say now, it is certain that the digital generation will take over, sooner or later.

Both on the earth and in the sky there are clear signs of this threat. In the sky, that is, in cyberspace, there are very distinctive examples, such as `coursera.com`, of a forthcoming overturn in effective teaching on-line. Thousands of libraries around the world are converging towards a single huge library named `google.com`. On the earth, on the other hand, one can quote numerous articles, from the newspapers, professional magazines and research journals, on the rapidly growing phenomenon of online learning. Web-based labs have to become an integral part of this trend.

Societal motivation. It may not be immediately obvious, but it should be made absolutely clear that the World Wide Web, as we know it now, bas been created exactly for the purpose of remote access to the labs. The very first paper published on this technology was written, as an internal report, by two scientists from the European Organization for Nuclear Research, CERN, in Geneva, proposing the creation of a protocol, which would allow sharing data and equipment usage among physicists around the world, working on high-energy physics experiments [13].

The Large Hadron Collider (LHC) built and operated at CERN, and known from recent discoveries of the tiniest elementary particles known to the humankind, has remote control centers, from which one can design and control high-energy physics experiments. An example of such center at Fermilab, near Chicago, has been described in [14] and is shown in Fig. 12.4.

Fig. 12.4. LHC control room at Fermilab (photo by the author).

Innovation context. Among the recent (2012) Top Ten innovations predicted by IEEE Spectrum [15] to likely have the biggest impact in the forthcoming years, there were a number of significant technologies, among them:

- exoskeleton for paraplegics, which would retire a wheelchair solution
- bionic eye that allows the blind to see just with a pair of sunglasses
- 3-D printing that allows creating complicated mechanical structures without a manufacturing facility
- 3-D chips that would further increase computing speed and power
- 4G Long Term Evolution (LTE) networks that increase bandwidth by the order of magnitude, and more.

What is amazing, however, and sad, is that this and a broader list of fourteen technological hits does not include any single entry, which would closely relate to education. This fact must prompt some response from educators to both verify the accuracy of the claims and take respective action.

Need for remote labs. Overall, there is an unquestionable need to create a software development education laboratory to apply methodologies for implementation and testing of embedded and cyberphysical systems, as well as to enable web-based development and testing (as opposed to traditional local development), by expanding remote access to operation.

One must remember that the primary objective of a remote software engineering lab is to let students learn by developing software remotely and then uploading it to the remote devices for testing and debugging. This is significantly different from traditional web-based or online laboratories, which only allow conducting remote experiments, without any changes to software operating the remote device.

Thus, traditional labs can be called *non-invasive*, since the student is not intended to change the software, which runs the instrument or device, perhaps, with a few exceptions limited to selection of modules but never with complete change of software running the device.

In software engineering labs, the situation is completely different, because the essence of a lab is for the student to design, implement and test a software module on the equipment, accessible locally or remotely. Consequently, the notion of remote lab must be extended by a concept of *invasive labs*, where, as in software engineering, new software, good or bad, but developed by a student, is to be uploaded to the remote instrument or device to let he student learn software engineering principles to become a professional.

Although the concept of a lab operating this way is not extremely new, to the author's knowledge, until now there has been no single course offered in the U.S. universities, which would involve using such a lab on a full scale basis.

12.2. FGCU's web-based software engineering lab

Given the strong motivation for developing the labs, as outlined in the previous section, a web-based real-time software engineering lab with hands-on features has been created at FGCU and used on experimental basis in project courses. General overview of the lab and its progress has been presented previously [16-21], so below only a brief summary is given followed by the lessons learned thus far and an overview of the most recent developments.

The hands-on feature is emphasized, which means that the functionality of an experiment is preserved whether the lab experiments are done over the web or with student's physical presence in the lab.

12.2.1. Lab overview

The architecture of the lab follows the one shown in Figure 12.1. A number of lab stations, interfaced to the Internet through various technologies, are accessible from web clients. Students obtain access to the stations according to schedule via a course webpage. Lab stations are diversified to offer various technological platforms, to help meet the criteria what lab is best suited to learn specific concepts.

The assumption in selection of equipment and software for lab stations is that the platform clearly articulated computing concepts important in embedded and cyberphysical systems. Following this principle, a respective platform has been selected for each station, as illustrated in Table 12.1.

Table 12.1. Summary of Existing Lab Stations

Vendor	***Wind River***	***Parallax***	***WDL Systems***	***National Instrum.***	***Atmel***
Hardware	PowerPC	Multicore	Vortex86	Any	μC
Bus	Serial	USB	USB	Zigbee	I2C
OS	VxWorks	Bare	Windows CE	Any	Bare
Program. Language	C/C++	TinyBasic	C#	G	C
IDE	Work-bench	Hydra	Visual Studio	Labview	AVR Studio
Protocol	FTP/RPC	HTTP	TCP/UDP	Data Sockets	HTTP
Web Technol.	CGI	ASP.NET	.NET Framework	RT Lab	PHP
Application	Device Testing	GUI	Control & GUI	Sensor Network	DAQ

Stations have been designed to cover entire array of embedded and cyberphysical systems development, including:

- hardware architecture and bus architectures
- operating systems

- programming languages and integrated development environments
- network protocols and web technologies, as well as
- applications.

The essential point in operating all lab stations is their dual interface: one for operators, which allows conducting experiments or tests just like in all remote labs described in literature, and another for developers, which would let uploading the executables and run them over the network. This is illustrated in Figures 12.5 and 12.6, for a web game development station [19].

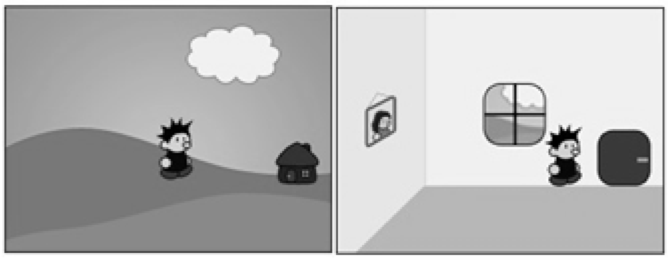
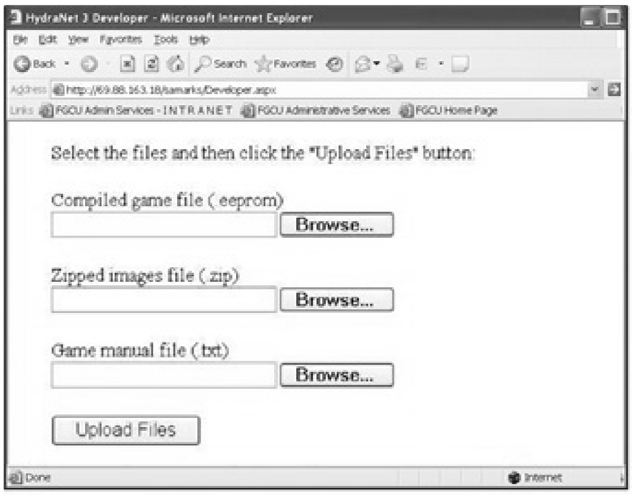


Fig. 12.5. Sample screenshot of a "Hide and Seek" web game.

This station allows development of an Internet game running on an embedded computer based on a Parallax multicore chip, installed on a board located in the lab. The operation of the game, which is partially shown on the interface in Fig. 12.5, involves participation of up to four players connected via the network to the game server. Each player can move the kid on the screen and score points according to the rules of the game.

When the developer is ready to deploy a new version of the game, or upload an update fixing the bugs, he can stop the execution warning the players, first, and then use the developer's interface shown in Fig. 12.6, for uploading. The interface for this particular station involves uploading the executable, uploading new images and a modified game manual, if necessary. Thus, the lab station has full capability adequate for its use in a course.


HydraNet 3 Developer - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://69.88.163.18/samarks/Developer.aspx
Links FGCU Admin Services - INTRANET FGCU Administrative Services FGCU Home Page
Browse...
Browse...
Browse...
Upload Files
Done
Internet

Select the files and then click the "Upload Files" button:

Compiled game file (eeprom)

Zipped images file (zip)

Game manual file (txt)

Fig. 12.6. Example of a developer's interface for a web game.

12.2.2. Lessons learned

The lab has been in operation for the last three years on an experimental basis, with a more comprehensive use in a course on Embedded Systems Programming offered in the Spring 2013 semester. Overall, lab stations have been used as needed in various upper level project courses. For example, teaching lower level concepts, in a course on Embedded Systems, is well served by the labs covering hierarchical items listed in rows from (1) to (5) of Table 12.1, while teaching concepts of cyberphysical systems in a Computer Networks course is well covered by items in rows (6) through (8) from that table.

During this period, a number of issues came up and experiences have been gained, which can be roughly divided into three categories: pedagogy, technical issues, and administrative and organizational challenges. Respective observation have been discussed extensively in previous publications [17] and [19], so here only a handful of problems are mentioned.

Pedagogy. Pedagogy is a crucial factor in offering and use of all engineering labs, not only those accessible online. In this regard, many different issues come into play due to the fact that the remote labs are unsupervised and asynchronous.

First, it must be made clear that including the labs in a course enhances the learning process. This seemed to work for two reasons: (1) the remote labs speed up the process of acquiring knowledge of concepts and techniques, and (2) the remote labs broaden the horizons of knowledge in software development, because students are forced to include into the picture elements of interactions with multiple additional components, such as networks and people.

Second, emphasis on conducting the later phases of the software development cycle, implementation and testing, via the web, makes the learning process more attractive, because of the opportunity to make actual observations in real time how the developed software performs.

Third, elements of pedagogy which work in teaching embedded systems with conventional labs, that is, enforcing knowledge by a sequence of demo, exercise, assignment and project, do not seem obstructed by the move to web-based labs.

Ultimately, the question of intellectual value of each project has to be addressed before it is included in the course: what is the contribution of a lab to the course objectives? For a discussion of several other observations related to pedagogy, the reader is referred to [17] and [19].

Technical issues. Numerous technical issues have been identified, which fall into two major categories: availability of network connections and continuity of operation. Additional observations related to technical issues can be found in [17], [19].

Multiple embedded devices with Internet connectivity, which are essentially additional computers on the university enterprise network, constitute potential vulnerability, which can be exploited by malicious users. For security reasons, most software ports except port 80 are blocked by network administrators and computing services. The significance of this problem has not been anticipated when the project started, and depending on the project's scale and scope they must be appropriately addressed sufficiently early in the life cycle.

To provide the continuity of operation of such a lab is an enormous challenge, due to the (non)-availability of technicians. One solution is to designate students from senior level courses to become station custodians. This costs less than full-time technicians and is potentially even more practical regarding responsiveness, however, can be considered only as a temporary solution, since students who graduate have to be replaced by the next generation, and this requires additional time and extra funds for training.

Administrative and organizational challenges. Administrative problems are also significant and are by nature mostly beyond the control of the lab offerors. The major problem is course enrollment and tuition payment. The nature of remote labs is to make them available for access from remote locations worldwide. This fact multiples the number of potential users by a factor hard to estimate, because anyone with respective prerequisites or academic credentials can become a legitimate user. However, due to the state and university regulations, students who want to take a course offered with such remotely accessible lab may face multiple difficulties. There may be many years to come before this issue will find some satisfactory resolution.

Another significant problem appears to be the training of faculty to include these labs in their courses. This is related to compensating faculty who want to participate in the development of such labs. Whether developing or adopting the lab, the process is challenging and imposes additional burden on instructors who are willing to face the changing world. It is certain that both governments and universities have to find ways to fund faculty development to follow such inevitable trends.

Other administrative and organizational aspects of web-based labs exist, of which the most important are the following:

- One of the main advantages of web-based labs, opportunity of sharing equipment, is raising a question of paying for maintenance.
- If the lab is initially playing well its role, there is an essential question on sustainability to continue operation when the funding period expires.
- One of the most crucial factors in building and expanding the lab is faculty motivation, especially critical when external universities are involved.
- Educating faculty about the benefits and logistics of web-based labs seems to be necessary. One vehicle to achieve this goal is faculty workshops.

12.2.3. Evolution of the lab stations

Expansion of the software engineering program by adding new courses constantly demands development of new contents and new lab stations to satisfy course requirements. Additionally, the need of filling the pipeline with high-quality high school students entering college prompts for teacher training in respective technologies, which in turn results in adding new stations.

Given that pedagogy is the essential driving factor in every course or lab design, in case of this project, there are three major educational determinants of the lab station contents:

- knowledge of software design issues for cyberphysical systems, including an appropriate design methodology and design notation;
- principles of remote implementation and remote debugging and testing of an application;
- remote execution of an application to control a remote device.

These three stages translate directly into the organization of the lab, with corresponding three different components in mind for adding new stations.

Assuming that two particular topics in embedded and cyberphysical systems need attention nowadays, robotics and security, respective lab stations that are meant to address related needs are briefly outlined below.

NAO robot. The robotics device selected for use in lab expansion is the NAO humanoid robot (Fig. 12.7) [22]. Its development environment named NAOqi could be used simply to create desktop applications, but with clever use of new web protocols a web accessible version of the NAO software can be created at a distance and remotely uploaded to the robot.

WebSockets, which are a new web interface created alongside HTML 5 and supported by most of modern web browsers, provide full duplex communication between the client and server. This enables a responsive web application to use the external server to communicate with the robot's onboard computer. Voice and image communication are also possible, which open completely new opportunities in remote robot control and learning.

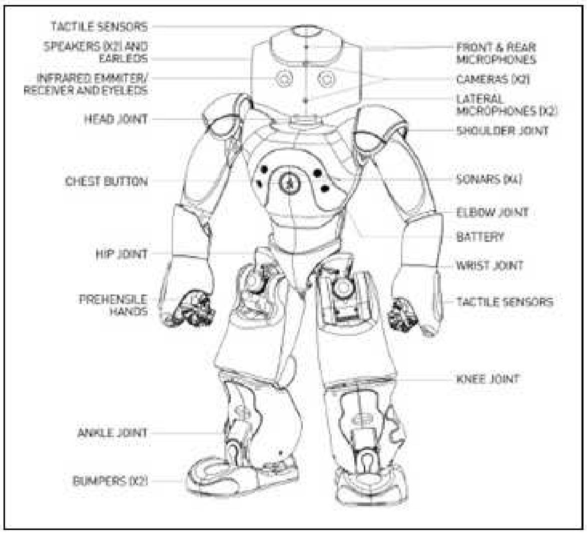


Fig. 12.7. NAO humanoid robot [22].

Raspberry Pi. Embedded systems security has many facets and can be taught in a number of ways. For this lab, Secure Shell (SSH) and Secure Socket Layer (SSL) protocols have been selected as teaching vehicles. To prepare the remote system for proper use by students, two components have to be devised accordingly: the target platform and an external device connected to it.

The target platform chosen for this project is Raspberry Pi, a very inexpensive credit-card size computer from the non-profit Raspberry Pi Foundation [23]. It has a number of useful features for an embedded system, including eight general-purpose input/output (GPIO) pins, with support for I2C, SPI, and UART protocols, and two USB ports. The Raspberry Pi has support for a few different operating systems, among them Android and a version of Debian Linux adapted for the Pi, called Raspbian.

The remote device controlled by the Raspberry Pi is a very simplistic remote controlled car (Fig. 12.8). It can move forward and backward with variable speed. The components for the car include a USB Wi-Fi dongle and a USB webcam for vision (marked by a right arrow), and a number of pins (marked by left arrow) for attaching external devices, including a motor controller for controlling the speed and direction of the car, etc.

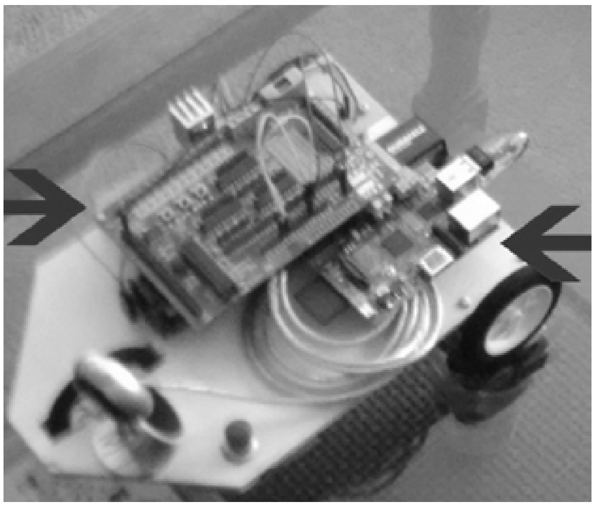


Fig. 12.8. Pi rover prototype.

12.3. Significance of remote labs

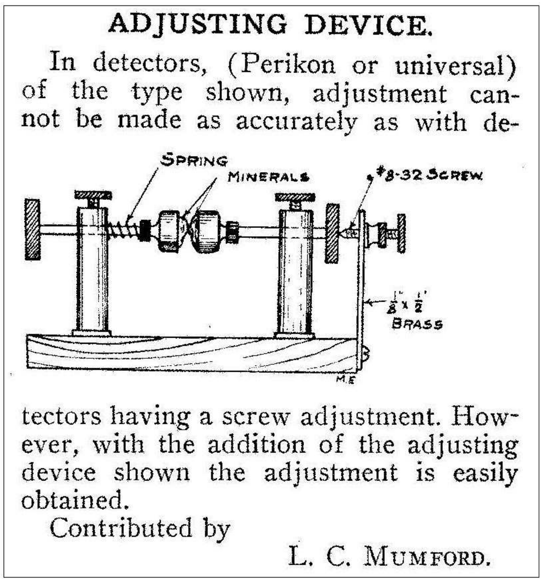
In addition to the local perspective at the educational institution, where such labs have tremendous impact on course offers and actual delivery, the labs have great significance from a broader perspective due to their innovative nature. It is not enough to answer the typical questions how this type of labs fits into the educational model. There is a more fundamental question how this type of phenomenon plays in the historical process of technological and societal change. The current section tries to shed some additional light on the emergence of remote labs, discussing the subject from a completely non-technical points of view.

12.3.1. Megamachines, gigamachines, and more

Observing the speed and pervasiveness with which modern computing technologies penetrate the society, what immediately comes to mind is the unprecedented scale of their usage. The number of Facebook users, according to company's quarterly report, recently reached 1,11 billion per month (March 2013 data), with 665 million active users each day on average in March. As reported by Digitaltrends, the number of active mobile phones will exceed the world population in 2014, reaching 7.4 billion devices [24]. Even more importantly, from the perspective of embedded and cyberphysical systems, according to the Chief Scientist of the U.S. Air Force, by 2025 there will be 7 trillion IP enabled devices in existence [25], all forming a humongous ecosystem that would need a well-educated workforce.

But it's not only the growing population of devices or users, which is astonishing and unprecedented. It is also the size of certain individual devices, which is reaching amazing proportions. To an extent, this issue has been quantified around 70 years ago by Lewis Mumford, with his concept of a *megamachine* [26]. Mumford, himself an inventor, whose first published improvement appeared over a 100 years ago (Fig. 12.9), used this term to describe the size of some large-scale endeavors, with an example of building the pyramids. Referring to this concept he termed it "a shorthand reference to the entire technological complex", one that the Egyptians invented for harnessing the manpower to erect and maintain the pyramids.

In a more contemporary world, the concept of a megamachine is very fertile and can be applied to devices, such as the Large Hadron Collider, Space Shuttle, Boeing Dreamliner, etc., but not only in technology, in other sectors of the society as well. For example, building art objects by Christo and Jean-Claude, such as the Valley Curtain in Rifle, Colorado, in the early 1970's, or wrapping up the Reichstag in the 1990's or making The Gates in New York City's Central Park, in 2005, where they installed 7,503 vinyl gates along 23 miles of pathways have all the symptoms of a megamachine. Thus, the megamachines are present not only in technology and can be viewed as social structures or organizations identified by complexity and particular challenges of scale.


SPRING
MINERALS
#8-32 SCREW
1/8" x 1/2" BRASS
M.E.

ADJUSTING DEVICE.

In detectors, (Perikon or universal) of the type shown, adjustment cannot be made as accurately as with detectors having a screw adjustment. However, with the addition of the adjusting device shown the adjustment is easily obtained.

Contributed by

L. C. MUMFORD.

Fig. 12.9. Mumford's first invention [27].

Incidentally, Mumford quantified the size of a megamachine, using the prefix "mega", to reflect the machine's size, not really intending it but adhering to its meaning as 10^6 components. What was hard to count back then, and could be only estimated for the pyramids, is much easier to account for in contemporary societies.

As stated by Ozaki [28], "A commercial aircraft typically involves roughly 200,000 parts" – the number probably much lower than number of parts in Boeing 787 – "ten times more than a car, which involves 20,000 parts.". It is amazing to see how much the number of car parts has increased over a century: "In the early 1900's it took only 700 parts for workers at Ford Motor Company to produce a Model T." [29]

Thus, when one speaks of the Internet based endeavors, such as social networks or mobile phones, the term one could use is *gigamachines*, and with the advent of cyberphysical systems it goes into the *teramachines* territory. This is where the remote labs have to be placed regarding the size of the venture, maybe forming a lab cloud, and have to be judged in this context.

Talking further about megamachines as social structures, Mumford looked closer at the role of a *clock* across societies and made one very crucial observation that: "The clock is not merely a means of keeping track of hours, but of synchronizing the actions of men." Its domination in our lives evolved over the ages to the point that: "Abstract time became the new medium of existence. organic functions themselves were regulated by it: one ate, not upon feeling hungry, but when prompted by the clock; one slept not when one was tired, but when the clock sanctioned it."

When the scale goes up, it has a prevailing impact on human behavior. Thus, clock brings us to "the medium is the message" principle.

12.3.2. The medium is the message

It was Marshall McLuhan who wrote in [30] and later reformulated it in [31] that:

> "Today, after more than a century of electric technology, we have extended our central nervous system itself in a global embrace […]. Rapidly, we approach the final phase of the extensions of man – the technological simulation of consciousness, when the creative process of knowing will be collectively and corporately extended to the whole of human society, much as we have already extended our senses and our nerves by various media."

In his analysis, glasses are extending our vision, vehicles extend our legs, printing extended our speech, telegraph, telephone and newer media extend our nervous system, and electronic media extend our consciousness.

In this view and in relation to these observations he coined the phrase *the medium is the message*, arguing that there is a separation between the content of the medium and the medium itself. What is actually in the contents is not important, it's the medium that contains, conveys, and *is* the message. What led McLuhan to this observation is his previous study on the printing press [32]. This is where he formulated the essential thought on the impact of the invention of printing on the evolution of modern society in western civilization. The advent of the printing press gave rise to the revolution in communication. Following this, the emergence of a rational thought in Enlightenment was actually a consequence of introducing the printing press.

In this view, it is easy to state that cell phones, as extension of our consciousness, are determining our behavior. The clock is the technology that has been enforcing our behaviors for ages. If, indeed, the medium is the message, the clock’s message is time. The medium impacts the message to the extent that ultimately it becomes a message itself.

To illustrate how web-based labs are actually moving into this direction, let’s bring up a simple scenario from a web-based course. When Jim and Joe, who are taking such a course on Embedded Systems cannot resolve a dispute, whose program to control a remote robotic arm, in response to an assignment, is better, they decide to go to a Starbucks coffee shop. Then, they logon to the robotic device in the lab, upload their programs in sequence, and see whether Joe’s program, which is moving the arm faster, or Jim’s program, which is moving the arm not so fast but smoothly, actually meets the requirements more precisely. What happens is exactly using the medium as the message.

12.3.3. Disruptive technology

When one looks carefully into the evolution of technologies over the ages, it may be surprising that Lewis Mumford considered the mechanical clock as the most important invention of mankind and Marshall McLuhan might have said the same about the invention of a printing press. Both inventions were ranked higher than that of a steam engine. Another interesting observation, which comes in parallel, is that it is rarely obvious or discussed what technologies were failing due to the introduction of new inventions.

Bowen & Christensen ([33], p. 43) give the following interesting examples from the computing field:

> IBM dominated the mainframe market but missed by years the emergence of minicomputers, which were technologically much simpler than mainframes. Digital Equipment dominated the minicomputer market with innovations like its VAX architecture but missed the personal-computer market almost completely.

The same is true about non-computer companies that missed the boat when newer technologies became to appear. They include: Sears giving a way to Walmart, Blockbuster that gave a way to Netflix, newspapers that are getting collectively bankrupt giving a way to digital media, and so on. What we do not realize is that the same may happen to traditionally viewed universities, which are one of the most conservative institutions on earth, still enjoying their structures shaped centuries ago.

This brings us to the concept of *disruptive technologies*, the term introduced in [33]. Generally speaking, a disruptive technology means the technology that has a potential to disrupt the markets, because they have not been prepared for its introduction. The term does not mean just innovation, but innovation that has a disruptive impact on the markets.

In a broader perspective, such a disruptive technological event was the introduction of print by Gutenberg, although it took years to change the markets, but that was proportionate to the slow pace of adopting innovations at that time. The same observation can be made about invention of a steam engine, which ultimately led to the industrial revolution. A less obvious example, not necessarily perceived as a disruptive technology, was the introduction of a clock, which drastically changed our lives, as noticed by Lewis Mumford.

It is important to note that the concept of a disruptive technology may have a different meaning depending on the context. For instance, as has been noticed in [34], Ryszard Kapuściński gives an example of an old technology of a "cheap, light, plastic container" that had drastically a tremendous impact on everyday life in Africa [35]. Namely, when plastic containers replaced heavy metallic vessels to carry water, not only women but also children could carry them and there were no worries of losing or breaking such, since they could be immediately replaced.

The nature of the concept has been quickly adopted in computing by peer-to-peer networks [36] and more recently in the military [37]. As reported by Keefe [37], among a couple of dozens technologies categorized as disruptive and important to the U.S. military, including night vision systems, autonomous robotic devices, free electron lasers, etc., one which is surprisingly listed as a category is *training*. This statement gives web-based labs a double edge: one as a true technological advancement and another as a game-changing educational vehicle.

12.4. Conclusion

As the data of the U.S. Department of Labor indicate, the projected growth of the demand for software engineers for the next decade is at the level of 30%, much faster than the average for all other occupations [38]. A large part of the profession will have to deal with embedded and cyberphysical systems, because of their significance to the nation's wealth and security. Thus, it makes a big difference whether we educate these engineers with old or new technologies and how do we shape their attitudes.

The real question we should ask ourselves, as educators, is: How can we make the existing and forthcoming Internet-based technologies beneficial in transferring knowledge? How to make sure that they create lasting cultural and intellectual values? Finding the answers becomes more urgent every day, since with the advent of mobile phones it appears like the entire world collapsed into the fifth dimension, the cyberspace. With the pervasive nature of web technologies and their proliferation, web-based labs can make a dramatic change in our lives as engineers.

The presented project is original in software engineering and embedded/cyberphysical systems education, because it allows students and developers not only to operate remotely the lab devices via web interface, but also program the devices from remote locations and remotely test the software (if necessary, with a webcam) without ever entering the lab physically. Thus, the project is game changing, because if such labs proliferate, this will ultimately cause a significant expansion of the ways students of engineering and computing disciplines can learn online. If the education market happens to respond to this challenge, web-based labs may become a new disruptive technology, one can term *lab-by-wire*. It may cause a real breakthrough in education, contributing to the potentially dramatic change one can already observe coming, that traditional universities will cease to exist.

Although the concept of a lab operating this way is not extremely new, to the author's knowledge there has been no single course offered, yet, in the U.S. universities, which would involve using such a lab on a full scale basis. This paper, placing remote labs in a broader historical and societal context, urges the profession to take a closer look at creating fully operational remote labs that allow students to conduct hands-on tasks from a distance, never physically coming to the lab.

There are, of course, multiple unknowns in the entire endeavor. For example, it is unclear from this project whether, in addition to learning specific concepts of software development, web-based labs help in acquisition of problem solving skills and application of critical thinking. A more targeted research is needed to lead to specific observations and respective conclusions.

In summary, there is no doubt that as much as the significance of online education is being recognized in the computing community [39], the significance of remote, web-based labs should be also brought to our attention. Both the interest of the US military [37] and the recent establishing of an IEEE Working Group 1876 with a mission to develop a standard for online laboratories [40] testify to this fact.

Acknowledgment

This work has been done under grants from the National Science Foundation (NSF), Award No. DUE-0632729 and Award No. DUE-1129437. Views expressed herein are not necessarily those of the NSF. Additional support has been provided by a grant SBAHQ-10-I-0250 from the U.S. Small Business Administration (SBA). SBA's funding should not be construed as an endorsement of any products, opinions, or services. Supplementary funding has been provided by NASA through the Florida Space Grant Consortium and through Summer 2010 Faculty Fellowship for the author at the Ames Research Center.

Bibliography

[1] Aburdene M.F., Mastascusa E.J. and Massengale R.: A Proposal for a Remotely Shared Control Systems Laboratory. In: *Proc. FIE'91, 21st Annual Frontiers in Education Conference*, Purdue University, West Lafayette, Ind., September 21-24, 1991, pp. 589-592.

[2] Ma J. and Nickerson J.V.: Hands-on, Simulated and Re-mote Laboratories: A Comparative Literature Review. *ACM Computing Surveys*, Vol. 38, No. 3, Article No. 7, 2006.

[3] Gravier C., Fayolle J., Bayard B., Ates M. and Lardon J.: State of the Art about Remote Laboratories Paradigms: Foundations of Ongoing Mutations. *Intern. Journal of Online Engineering*, Vol. 4, No. 1, pp. 19-25, 2008.

[4] Cooper M. and Ferreira J.M.M.: Remote Laboratories Extending Access to Science and Engineering Curricular, *IEEE Trans. on Learning Technologies*, Vol. 2, No. 4, pp. 342-353, 2009.

[5] Guimarães E.G., Cardozo E., Moraes D.H. and Coelho P.R.: Design and Implementation Issues for Modern Remote Laboratories, *IEEE Trans. on Learning Technologies*, Vol. 4, No. 2, pp. 149-161, 2011.

[6] Tawfik M., Sancristobal E., Martin S., Diaz G. and Castro M.: State-of-the-Art Remote Laboratories for Industrial Electronics Applications, *Proc. TAEE'12, Conference on Technologies Applied to Electronics Teaching*, Vigo, Spain, June 13-15, 2012, pp. 359-365.

[7] Azad A.K.M., Auer M.E. and Harward V.J. (Eds.): *Internet Accessible Remote Laboratories: Scalable E-Learning Tools for Engineering and Science Disciplines*. IGI Global, Hershey, Penn., 2011.

[8] Zubía J.G. and Alves G.R. (Eds.): *Using Remote Labs in Education: Two Little Ducks in Remote Experimentation*. University of Deusto, Bilbao, Spain, 2011.

[9] Bochicchio M.A. and Longo A.: Hands-On Remote Labs: Collaborative Web Laboratories as a Case Study for IT Engineering Classes, *IEEE Trans. on Learning Technologies*, Vol. 4, No. 4, pp. 320-330, October-December 2009.

[10] Zalewski J.: Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences. *Annual Reviews in Control*, Vol. 25, pp. 133-146, 2001.

[11] Mayr O.: *Zur Frühgeschichte der technischen Regelungen*, Oldenburg Verlag, München, 1969 (English translation: *The Origin of Feedback Control*, MIT Press, Cambridge, Mass., 1970).

[12] Reeves G.E.: *What Really Happened on Mars: Authoritative Account*. URL: http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html; 1997
[13] Berners-Lee T. and Cailliau R.: *WorldWideWeb: Proposal for a HyperText Project*. CERN, Geneva. URL: http://www.w3.org/Proposal.html; 1990
[14] Harms E. et al.: LHC@FNAL - A New Remote Operations Center at Fermilab, *Proc. ICALEPCS07, Intern. Conference on Accelerator and Large Experimental Physics Control Systems*, Knoxville, Tenn, October 15-19, 2007, pp. 23-25.
[15] *Top Tech 2012. IEEE Spectrum's prediction of the tech that will make news this year*. URL: http://spectrum.ieee.org/at-work/innovation/top-tech-2012
[16] Zalewski J.: Cyberlab for Cyberphysical Systems: Remote Lab Stations in Software Engineering Curriculum, *Proc. ICEL 2013, 4th Intern. Conference on e-Learning*, Ostrava, Czech Rep., July 8-10, 2013, pp. 1-7.
[17] Zalewski J.: Web-based Labs for Cyberphysical Systems: A Disruptive Technology. *Proc. WCCE2013, 10th IFIP World Conference on Computers in Education*, Toruń, Poland, July 1-5, 2013, Vol. 2, pp. 89-97.
[18] Zalewski J.: Hand-on Software Engineering Labs via the Web: Game Changing in Online Education. *TransNav – Intern'l Journal on Marine Navigation*, Vol. 7, No. 2, pp. 93-100, June 2013.
[19] Zalewski J.: Lab-by-Wire: Fully Web-based Hands-on Embedded Systems Laboratory, *Proc. EDUCON 2013, IEEE Global Engineering Education Conference*, Berlin, Germany, March 13-15, 2013, pp. 928-933.
[20] Zalewski J.: A Comprehensive Embedded Systems Lab for Teaching Web-based Remote Software Development, *Proc. CSEET 2010, 23rd Annual IEEE Conf. on Software Engineering Education & Training*, Pittsburgh, Penn., March 9-12, 2010, pp. 113-120.
[21] Daboin C. and Zalewski J.: Lab Station for Remote Measurement and Control in Teaching Real-Time Embedded Systems and Software Engineering, *Proc. WRTP'09, 30th IFAC Workshop on Real-*

Time Programming, Mrągowo, Poland, October 10-12, 2009, pp. 43-48.

[22] Aldebaran Robotics. *NAO Humanoid Robot - Key Features*. URL: http://www.aldebaran-robotics.com/en/Discover-NAO/Key-Features/hardware-platform.html

[23] Richardson M. and Wallace S.: *Getting Started with Raspberry Pi*, O'Reilly, Sebastopol, Calif., 2012.

[24] Pramis J.: *Number of Mobile Phones to Exceed Word Population by 2014*. February 28, 2013. URL: http://www.digitaltrends.com/mobile/mobile-phone-world-population-2014/

[25] Maybury M.Y.: Air Force Cyber Vision 2025, Invited Talk, *CSIIRW-8, 8th Cyber Security and Information Intelligence Research Workshop*, Oak Ridge, Tenn., January 8-10, 2013.

[26] Mumford L.: *Technics and Civilization*. Harcourt, Brace & Co., New York, 1934, pp. 12-18.

[27] Mumford L.: Adjusting Device, *Modern Electrics*, Vol. 3, No. 6, p. 324, September 1910.

[28] Ozaki T.: Open trade, closed industry: the Japanese aerospace industry in the evolution, In: *Globalization and Economic Nationalization in Asia*, A.P. D'Costa, Ed., Oxford University Press, 2012, p. 147.

[29] Carbaugh R.J.: *International Economics. 14th Edition*, Cengage Learning, Independence, Kentucky, 2011 p. 59.

[30] McLuhan M.: *Understanding Media: The Extensions of Man*, Mentor, New York, 1964.

[31] McLuhan M. and Fiore Q.: *The Medium is the Massage*. Bantam, New York, 1967.

[32] McLuhan M.: *The Gutenberg Galaxy*, Routledge, London, 1962.

[33] Bower J.L and Christensen C.M.: Disruptive Technologies - Catching the Wave, *Harvard Business Review*, Issue 2128, pp. 43-53, January 1995.

[34] Mitchell W.J.: *ME++ – The Cyborg Self and the Networked City*. MIT Press, Cambridge, Mass., 2003.

[35] Kapuściński R.: *Heban*, Czytelnik, Warszawa, 1998 (English translation: *The Shadow of the Sun*, Knopf, New York, 2001).

[36] Oram A.: *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly, Sebastopol, Calif., 2001.

[37] Keefe J.C.: Disruptive Technologies for Weapon Systems, *WSTIAC Quarterly*, Vol. 7, No. 4, pp. 3-7, November 2007.

Chapter 13

From relatively isolated system to object approach – the story of system development & modeling tools

The very first theoretical approach to the principle of systems and processes modeling named relatively isolated system was presented by Henryk Greniewski in 1956. Farther development of the idea was an introduction of kind of object techniques to design an information system for Ericson company by Ivar Jacobson about 1965. The first implementation of relatively isolated system in computer programming was carried out for Simula 67 simulation language. Smalltalk (1976) brought in genuine breakthrough into implementation of the object ideas into programming languages. Graphical object language like UML is a computer aided tool for systems and processes modeling, developing the prime ideas of relatively isolated system.

13.1. Introduction

Business modeling (generally system modeling) approach with object oriented technology plays very imported role in today business engineering, information system design, and so on. The history of it is practically very short. Early models were built with so called boxes and connecting them into a set. There were two categories of boxes: one named a black box and the other white box. The functionality of boxes named as black was not formally described or unknown. The name white box concerns boxes with functionality formally described or known.

In 1956 Henryk Greniewski (from University of Warsaw) presented the very first theoretical approach to the principle of systems and processes modeling introducing the concept of the relatively isolated system [1]. Two years later, Jay W. Forrester of Harvard University (who earlier specialized in flight simulation and modeling) published results concerning of usage box technique to simulate behavior of industrial dynamics [2]. The second step was performed by the Swedish team (directed by Ivar Jacobson [3] – information system design with prototype of object approach, for Ericson company) around 1965, and the third by the Norwegian team (authors of Simula 67 [4] - extension of Algol 60, first simulation language proposing concepts of primitive classes and objects: Ole-Johan Dahl & Kristen Nygaard of Norwegian Computing Center). Next step in development of the object concepts was the first object oriented programming language Smalltalk [5]; it was the beginning of further object oriented programming languages like Java, C# and others. The languages like UML [6] (Unified Modeling Language) providing options for modeling of business processes, being under continued elaboration, seem to be the latest development up to now.

13.2. Relatively isolated systems

The concept of the relatively isolated system is not new (its definition was presented in 1956 in Namur at First International Congress of Cybernetics with further research results published in 1960 [7]); it has been (somewhat tacitly) used in science for many centuries, at least since the time of Hippocrates' Corpus, but never presented openly. The need for an overt and at the same time exact use of that concept is self-evident as regards cybernetics. Moreover, its precise use is of importance in the logic of induction and the logic of analogies.

What is implied by that concept? Reference, to an *absolutely isolated system* - generally mean a system which:

- is not influenced by the rest of the Universe, and
- exerts no influence on the rest of the Universe (whether any such system actually exists is not immediately relevant).

By a *relatively isolated system* [7] we mean any system (a box or set of boxes – using above mentioned terminology) and only such which has the following two characteristics:

- it is influenced by the rest of the Universe, but only in certain specified ways called inputs, and

- it influences the rest of the Universe, but only in certain specified ways called outputs.

These conceptions, originally so simple, have to be subjected to certain complications: the influence of the system upon itself (e.g., self-correction) has to be taken into consideration, involving acceptance of the fact that some (but not all) outputs of the system may at the same time be inputs (feedback coupling).

The first basic concept considered, then, is that of a relatively isolated system. The concept of such a system, the notions of input and of output are abstract notions somewhat difficult to formulate with adequate precision. They can be formulated precisely if they are treated as what are called "primitive term", and if a certain set of postulates which infuse meaning into those concepts is adopted. That is to say, they can be formulated precisely by the method used in elementary geometry to give precision of meaning e.g., to the concepts of point and plane. For our purpose, however, such postulates would be over complicated; better to do with a somewhat simplified explanation, closing our eyes to certain essential difficulties.

Every *input* and every *output* of a given relatively isolated system is associated with:

- its *calendar*, i.e., a certain set moments, or intervals of time, of at list two elements, and
- its *repertory*, i.e., a certain set of distinguishable states.

In a given *relatively isolated system*, every *input* and every *output* adopts one, and only one, distinguishable state in any moment of its *calendar*. The function establishing a relation between the elements of the calendar of a given input (output), and the distinguishable states belonging to the repertory of that input (output) is called the *trajectory* of that input (output). Instead of using the expression “a distinguishable state of an input”, we may use briefly *stimulus*, and “a distinguishable state of an output”, we may use shorten to *reaction*. It should be stressed - *repertory* of each *input* (*output*) consists of at least two distinguishable states. To every paired input and output of a specific system a certain non-negative number of time units is ascribed, representing the time necessary for the reaction to take place – called reaction time or *time-lag*.

Four types of relatively isolated systems are introduced: differentiations are made between *reliable* and *unreliable* systems, on the one hand, and *prospective* and *retrospective* systems, on the other, each of these dichotomy classification is independent of the other.

- Prospective reliable systems – the present distinguishable state of any *output* is always univocally determined by past and present distinguishable states of all the *inputs* of a given system.
- Prospective unreliable systems - the present distinguishable state of any *output* is always univocally determined by past and present distinguishable states of all the *inputs* of a given system, with a constant probability greater than 50 per cent. The condition is called the principle of local Pseudo-determinism.
- Retrospective reliable systems – any past (but sufficiently remote from the present) distinguishable state of any input is always *univocally* determined by present and past (but not prior to the input state in general) distinguishable states of all outputs. The condition is called the principle of local Para-determinism.
- Retrospective unreliable systems - any past (but sufficiently remote from the present) distinguishable state of any input is always *univocally* determined by present and past (but not prior to the input state in general) distinguishable states of all outputs, with a constant probability greater than 50 per cent. The condition is called the principle of local Para-determinism. By a *determinator* of any *output* of a given prospective system is meant a function which assigns reaction to stimuli, and by *paradeterminator* of any *input* of a given retrospective system is meant a function which assigns stimuli to reaction.

If in any system the direction of time is reversed, the inputs replaced by outputs and the outputs by inputs, then a prospective system becomes a retrospective one (and vice versa, a retrospective system becomes a prospective one). In prospective systems, the present state of the outputs is determined (logically or probabilistically) by the past and present states of the inputs; in retrospective systems, the situation is the opposite: the past state of the inputs is determined by the recent past and present states of the outputs. The theory of relatively isolated systems is marked by duality: every theorem referring to the prospective systems has one, and only one - corresponding (dual) theorem referring to the retrospective systems, and vice versa; if one of such two theorems is proved, the other can be proved by simple transformations.

The concept of the theory of relatively isolated system - include principles of coupling different isolated systems (further called subsystems – equivalent of boxes). Three types of couplings are distinguished: parallel coupling, serial coupling and feedback coupling of subsystems. Let us take in consideration a pair of subsystems (or two boxes). If two subsystems are in a state of coupling with each other we have one of situations described below:

- The parallel coupling of two subsystems means one or more pairs of inputs, in which: (a) the first element of the pair being an input to the first subsystem and second element of the pair being an input to the second subsystem had been connected, and (b) the both inputs belonging to the pair have the same repertory.
- The serial coupling of two subsystems means: (a) one output of the first subsystem is connected to one input of the second subsystem, and (b) the connected input and output have the same repertory.
- The feedback coupling of two subsystems means: (a) one output of first the subsystem is connected to one input of the second subsystem, and (b) one output of the second subsystem is connected to one input of the first subsystem, and (c) the both connected pairs of input and output have the same repertory respectively.

The reaction of connected subsystems to feedback activity can be negative or positive. Negative feedback means of maintaining a state not remote from that of equilibrium, which often occurs in nature, in technology and in organized structures. Positive feedback operates opposite to the negative.

It happens that a given system is *feedback coupled with itself*; i.e., some of its outputs are at the same time among its inputs. In such – and only – such cases, we say that system in question is self-coupled. Such case is equivalent to *memory* option of the given system. It can be treated as the *internal state* of the system.

The *analysis* of the given existing complex relatively isolated system aims at identifying its component parts which in turn are simple relatively isolated systems, and studying couplings connecting them.

Typical *synthesis* of a relatively isolated system occurs as follows:

- the task is to build a new system satisfying certain specified conditions;
- a certain range of simple relatively isolated systems is given;
- the planned system is built by coupling systems belonging to the given range of simple systems;
- the task sometimes proves insoluble. Then, the compromise solution is sought: the task is reformulated so as to become (probably) solvable; for this purpose either the initial conditions are weakened or the given range of simple systems is augmented;
- there are often more solutions than one. In such a case, all the solutions are usually studied and the optimum one chosen (from the point of view of costs, efficiency, speed of operation, etc.).

Since the term *model* has many different meaning it seems advisable to define in which sense it will be used here. Let us assume that a certain relatively isolated system is given which will be called *the original*. By *model*, we shall here mean a system which is as little complicated as possible and which functions in a manner analogous to the original. *Model construction* will mean the designing or physical construction of a model.

13.3. The object oriented model

The object model has been proven as applicable to in a wide variety of problem domains [8] (Air traffic control, Animation, Command and control system, Computer integrated manufacturing, Office automation, Robotics, User interface design, and so on). Object oriented approach is based on a number of ideas:

- O*bject* is the relatively isolated system (the object is *encapsulated*) with its attributes and methods.
- A*ttributes* represent internal states of the object. The data of internal states are used to remember values characteristic for object.
- Contacts are possible only with object content by *interfaces* (input/outputs) of the object. Each interface of the object is bound with one or more methods of the object.

- M*ethods* represent internal functionality of the object (they perform transformation of input messages – supplied by an interface and attributes into output messages and new values of attributes). The objects with methods controlling their behavior (for example object meets homeostatic conditions), are named self-controlled or active objects. The other objects type, are named passive objects – they need external control provided by other object or objects.
- Objects belong to families named *classes*. Each class can be treated as a pattern to create objects. Number of objects belonging to the given class can change from one (singleton family) to many. Sets of attributes and methods and interfaces are the same for all objects created and belonging to given class.
- Two objects can connect (coupled) each other if they possess the same interface. All objects belonging to same class have the same interface. In case they belong to different classes they can use additional external interface with the given implementation. It should be noticed, each object can implement different interfaces. Therefore, object needs to be supplemented with the *interfaces* as a tool for objects interconnecting. Additional interface (which can be added to an object) is bound with one or more methods. Each method has name only without its implementation. Therefore it is called *abstract* method.
- Interface of given name can have different implementation (for example "to open" has one implementation for door opening and second implementation for barrel opening – the name of method is the same but its implementation differs). Some specialists call this property *polymorphism*.
- The *inheredity* of classes (or interfaces) gives the possibilities to extend both attributes and methods. The *derivate* class has attributes and methods of its parent class and additional ones created by operation of inheredity.
- Connectivity between objects (so called *association*) is possible if both objects has the same type of interfaces (it concerns both method and its implementation), as was noticed above. Objects can be *associated one to one*, or *one to N*, or *M to N*. Communication between associated objects is performed by the *message*. Each message contains data (pa-

rameters) and calls for a method or methods in an addressed object. Message obtained by the object sets new data for some attributes and causes performance of one or more indicated methods. As a result of performed one or more methods, the object generates message containing the values of some attributes and can send it back to the object source of previous message.
- Security options controls users' rights for change values of attributes and call methods. If an option *public* precedes name of a class, an object, an attribute or a method – it means that the name is accessible and visible for each user. If an option *private* precedes name of a class, an attribute or a method – it means that the name is accessible and visible for user having rights access to given name. The last option *protected* precedes name of a class, an object, an attribute or a method – it means that the name is accessible and visible for user having rights access to given name and their heritages. An additional security option is *final* – it blocks operation of inheredity of a class, an object, an attribute or a method. The *final* precedes a name of a blocked element.
- White box technique – means use of inheredity of classes or interfaces because a derivative class has known structure and implementation. It is a static technique.
- Black box technique – means use of composition of objects belonging to different classes. In this technique a message is sent from one object to another – both objects have the same interfaces. Such a message activates corresponding method of the destination object. The object's composition can be built from a string objects belonging to different classes. It is a dynamic technique.

13.4. Introduction to the development of the object approach

Objects are omni present. Starting from the late 1960s with programming languages such as Simula 67, they have pervaded every domain of software technology. So why have objects proliferated like a contagious virus? At least for two reasons:

- First, objects are good abstractions of real-world entities, thus are ideal for modeling of , distributed systems,
- Second, objects can yield suitable components for fostering modularity and re-use in building quality, complex applications.

However, object orientation is not a panacea that can cure all design and development problems. There is still considerable confusion and controversy for such key concepts as encapsulation, inheritance and polymorphism. There is no complete theory of object orientation, based on simple and well-defined concepts as well. Some theories (i.e. Abadi and Cardelli's – *Sigma Calculus*) proposing understanding of objects have emerged, but they are incomplete, concentrating on a few aspects such as polymorphism and overriding only.

Thus, object orientation is more an engineering approach than a well-founded science [9]. But it is rich enough to help solve problems in many domains, including user interface, programming, modeling, re-use and co-operation. The central concept in the object approach is that of the object; an object associates data and processes in to single entity, leaving only the interface - that is the operations that can be performed on the object - visible from outside. This approach is not new; it first appeared in the language Simula, designed as a Structured programming language for simulating parallel processes. The classes of Simula made abstraction possible by hiding the implementation and creating increasingly complex entities. The abstract aspect of the object approach, which enables a data structure to be hidden by the allowable operations for that structure, was formalized in the 1970s in the theory of abstract data types. The basic idea is to define a set of functions (for example the stack functions push, pop and top) for manipulating a set of types (for example stacks of integers) and to give a formal specification of these and of their relations as a set of axioms, leaving any implementation to be decided later.

In parallel with this formalization of abstract types, the language Smalltalk was developed; this, like Simula, implemented the object concept in the form of classes but in addition brought in message passing, taken from the actor concept. Smalltalk used in terms of generalization concept of inheritance to structure the classes hierarchically. Smalltalk was the real source of inspiration for the object approach; it was in the course of its development that the idea of a multi-window system, with graphical objects dragged in and dropped out, first arose.

Three classes Model/View/Controller (named MVC) were used to construct user interface in Smalltack-80. After Smalltalk-80 the relations between generalization and inheritance were developed extensively in so called design patterns [10]. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, suggested that practically the number of important patterns is limited [10]. They operated on three groups of patterns for creation, structures and activates, altogether 23 patterns only. Survey of Software Pattern Collections by Scott Henninger & Victor Corrêa [11], worked out nearly 15 years later - shows existences of 170 pattern entities (collections and individual patterns not in a collection) with a total of 2,241 patterns. Hopes concerning usage of design patterns to construct application software have been overestimated. Object approach gives new ideas to analyzing and designing of information systems. The new methodology of information system analyzes and designs with use of specialized languages like UML and so on, is the one of main results of object approach.

13.5. Unified Modeling Language

Object oriented approach to system modeling and design, named *Unified Modeling Language* (*UML*) is a graphical language. UML has a group of parents. The most important are: Grady Booch, James Rumbaugh and Ivar Jacobson [12]. The Object Management Group (OMG) – organization established in 1989 is responsible for standardization and further development of the language. The *UML* or rather its software realization (a processor), is a powerful tool to design and analyze relative isolated systems, i.e., information systems. UML operates a group of artifacts, named: *class diagram*, *objects diagram*, *use case diagram*, *sequence diagram, state machine diagram*, *component diagram*, *package diagram*, *deployment diagram*, *collaboration diagram*, and so on. UML functionality, carried on by UML processor, is extended by specialized languages like *Business Process Modeling Notation*

(BPMN)[6]. Some examples of the UML processor possibilities (to create, analyze, improve and simulate relative isolated systems) are presented below:

- Capture user's requirements by focusing on who (*actor*) wants to do what (use case) with the system: Identify functional requirements through users' scenario development; Specify use case details; Identify and document the interactions between use cases and actors with flow of events.
- Business Analyst draws and produce professional workflow diagram. UML also includes set of analytical tools for analysis and improves business process. Last but not least, instead of printing diagrams without arrangement, Business Analyst can produce tidy and up-to-date report automatically: Write operational procedure; Write step-by-step procedure for all business process tasks.
- Business process improvement helps organizations to optimize their current business processes in a systematic way. It is mainly concerned with desirable outcome in better resource management. Various tools, such as: *Business Process Modeling Notation* designer, assist business analysts in decision-making.
- Handy business rule editor enables users to describe, analyze and manage business operation and business logic as well. In model diagrams, the relationships among terms can be visualized.
- Capture software requirements with *UML use case diagram* and write use case flow. Elaborate and design user interactions with sequence diagram. Edition provides full use case modeling toolset and all UML diagrams for system analyst to design software.
- A UML Diagram elaborates system design decision and simplifies the communications between Project Manager, Software Architect, System Analyst, Designer and Developer.
- The simulation software helps to model business processes - with a visual language adopted worldwide — *Business Process Modeling Notation* (BPMN). Process simulation helps to spot potential *bottlenecks* for optimization.

[6] *Business Process Modeling Notation (BPMN) - www.bpmn.org.*

- Design conceptual/logical/physical database with *ER Diagram* (ERD)[7]: Generate or update database from ERD; Reverse engineer ERD from legacy database (Oracle, MS SQL, MySQL, PostgreSQL and more); Synchronization between class diagram and ERD; Generate and reverse database; In addition to visualizing database with ERD is also possible.
- Comfortable modeling environment containing: Intuitive user-interface boosts modeling productivity; The unified modeling interface is ready for cross-platform corporate environments (i.e. Windows, OSX and Linux).

13.6. Conclusions

The idea of relatively isolated system finds its continuation and further formalization in object approach but still waits for its' theoretical background.

Bibliography

[1] Henryk Greniewski, *Logique et cybernetique*, Actes du 1-er Congres International de Cybernetique, Namur 1956.

[2] Jay W. Forrester, Industrial Dynamics – a major breakthrough for decision Makers, Harvard Business Review, Vol. 36 No 4 pp 37 – 66.

[3] Ivar Jacobson, Maria Ericson and Agneta Jacobson, The Object Advantage – business process reengineering with object technology, Addison-Wesley Publishing Company, 1994.

[4] Ole-Johan Dahl and Kristen Nygaard, Class and subclass declarations. In Proceedings from IFIP TC2 Conference on Simulation Programming Languages, Lysebu, Oslo, ed.: J. N. Buxton, pages 158-174. North Holland, May 1967.

[7] ER Diagram (ERD) - *www.umsl.edu.*

[5] Adele Goldberg & Alan Kay, ed. (March 1976). Smalltalk-72 Instruction Manual. Palo Alto, California: Xerox Palo Alto Research Center.

[6] Grady Booch, Ivar Jacobson & James Rumbaugh (2000) OMG Unified Modeling Language Specification, Version 1.3 First Edition: March 2000.

[7] Henryk Greniewski, *Cybernetics without mathematics*, Pergamon Press and PWN, Warsaw 1960.

[8] Ian Graham, Alan O'Callaghan, Alan Cameron Wills, *Object-Oriented Methods – Principles & Practices*, Third Edition Pearson Education Ltd., 2001.

[9] Invar Jacobson, Maria Ericson and Agneta Jacobson, *The Object Advantage – business process reengineering with object technology*, Addison-Wesley Publishing Company, 1994.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1st Edition published by Pearson Education, Inc., publishing as Addison-Wesley, 1995.

[11] Scott Henninger & Victor Corrêa, Surveying Software Pattern Collections 1-1-2007 – DigitalCommons, University of Nebraska - Lincoln.

[12] Grady Booch, James Rumbaugh, Ivar Jacobson, *UML User Guide*, Addison-Wesley, 1999.

Authors and affiliations

***Barbara Begier* – Chapter 1**
Institute of Control and Information Engineering, Faculty of Electrical Engineering, Poznan University of Technology, barbara.begier@put.poznan.pl

***Walery Susłow* – Chapter 2**
Department of Computer Engineering, Faculty of Electronics and Computer Science, Koszalin University of Technology, walery.suslow@tu.koszalin.pl

***Michał Statkiewicz* – Chapter 2**
Department of Computer Engineering, Faculty of Electronics and Computer Science, Koszalin University of Technology, michal.statkiewicz@weii.tu.koszalin.pl

***Szymon Kijas* – Chapter 3**
Institute of Automatic Control and Computational Engineering, Software engineering, Warsaw University of Technology, s.kijas@elka.pw.edu.pl,

***Andrzej Zalewski* – Chapter 3**
Institute of Automatic Control and Computational Engineering, Software engineering, Warsaw University of Technology, a.zalewski@ia.pw.edu.pl

***Jakub Swacha* – Chapter 4**
Institute of Information Technology in Management, Faculty of Economics and Management, University of Szczecin, jakubs@wneiz.pl

***Karolina Muszyńska* – Chapter 4**
Institute of Information Technology in Management, Faculty of Economics and Management, University of Szczecin, karolina.muszynska@wneiz.pl

***Zygmunt Drążek* – Chapter 4**
Institute of Information Technology in Management, Faculty of Economics and Management, University of Szczecin, drazek@wneiz.pl

***Bartosz Wilk* – Chapter 5**
AGH Krakow, ACC Cyfronet, b.wilk@cyfronet.pl

***Marek Kasztelnik* – Chapter 5**
AGH Krakow, ACC Cyfronet, m.kasztelnik@cyfronet.pl

***Marian Bubak* – Chapter 5**
AGH Krakow, Department of Computer Science and ACC Cyfronet, bubak@agh.edu.pl

***Mariusz Jarocki* – Chapter 6**
Faculty of Mathematics and Computer Science, University of Lodz, jarocki@math.uni.lodz.pl

***Agata Półrola* – Chapter 6**
Faculty of Mathematics and Computer Science, University of Lodz, polrola@math.uni.lodz.pl

***Artur Niewiadomski* – Chapter 6**
Institute of Computer Science, Siedlce University of Natural Sciences and Humanities, aniewiadomski@gmail.com

***Wojciech Penczek* – Chapter 6**
Institute of Computer Science, PAS and Siedlce University of Natural Sciences and Humanities, penczek@ipipan.waw.pl

***Maciej Szreter* – Chapter 6**
Institute of Computer Science, Polish Academy of Sciences, mszreter@ipipan.waw.pl

***Bogumiła Hnatkowska* – Chapter 7**
Institute of Informatics, Faculty of Informatics and Management, Wroclaw University of Technology, Bogumila.Hnatkowska@pwr.wroc.pl

***Radosław Tumidajewicz* – Chapter 7**
Institute of Informatics, Faculty of Informatics and Management, Wroclaw University of Technology, radek@zacnie.net

***Tomasz Straszak* – Chapter 8**
Institute of Theory of Electrical Eng., Measurement and Information Systems Faculty of Electrical Engineering, Warsaw University of Technology, t.straszak@iem.pw.edu.pl

***Michał Śmiałek* – Chapter 8**
Institute of Theory of Electrical Eng., Measurement and Information Systems Faculty of Electrical Engineering, Warsaw University of Technology, smialek@iem.pw.edu.pl

***Anna Derezińska* – Chapter 9**
Institute of Computer Science, Faculty of Electronics and Information Technology, Warsaw University of Technology, A.Derezinska@ii.pw.edu.pl

***Piotr Trzpil* – Chapter 9**
Institute of Computer Science, Faculty of Electronics and Information Technology, Warsaw University of Technology

***Michał Żebrowski* – Chapter 10**
Orange Polska, michal.zebrowski@orange.com

***Andrzej Ratkowski* – Chapter 10**
Institute of Control and Computation Engineering, Warsaw University of Technology, a.ratkowski@elka.pw.edu.pl

***Patryk Czarnik* – Chapter 11**
Institute of Informatics, Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, czarnik@mimuw.edu.pl

***Jacek Chrząszcz* – Chapter 11**
Institute of Informatics, Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, chrzaszcz@mimuw.edu.pl

***Aleksy Schubert* – Chapter 11**
Institute of Informatics, Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, alx@mimuw.edu.pl

***Janusz Zalewski* – Chapter 12**
Dept. of Software Engineering, Whitaker College of Engineering, Florida Gulf Coast University, zalewski@fgcu.edu

***Marek J.Greniewski* – Chapter 13**
Institute of Computer Science, Maria Sklodowska-Curie Warsaw Academy, marek@greniewski.pl