e-Informatica

software engineering journal

2007 volume 1 issue 1

e-Informatica

e-Informatica

software engineering journal

e-Informatica

Wrocław University of Technology

Editors

Zbigniew Huzar (*Zbigniew.Huzar@pwr.wroc.pl*)
Lech Madeyski (*Lech.Madeyski@pwr.wroc.pl, http://madeyski.e-informatyka.pl/*)

Wrocław University of Technology
Institute of Applied Informatics
Wrocław University of Technology, 50-370 Wrocław, Poland

Graphics by Michał Stanek and Piotr Przybył
Computer typesetting by Piotr Przybył

e-Informatica Software Engineering Journal
http://www.e-informatyka.pl/e-Informatica/

Printed in a camera ready form

OFICYNA WYDAWNICZA POLITECHNIKI WROCŁAWSKIEJ
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISBN 987-83-7493-305-6

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Order No. 130/2007.

Editorial Board

Contents

Empirical Evaluation of Novel Approaches to Software Engineering

It is a pleasure to present to our readers the first issue of the e-Informatica Software Engineering Journal (ISEJ).

The idea to establish the e-Informatica Software Engineering Journal as a new scientific journal has been considered by Polish academic environment for several years. Finally, it appeared that we are able to start the international software engineering journal with strong support from many recognized researchers and practitioners in Europe who agreed to join the Editorial Board. We would like to express our gratitude to all those involved in many international software engineering conferences, e.g. International Conference on Product Focused Software Process Improvement (PROFES), International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP) or International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). We also want to thank SIEMENS (*http://www.siemens.pl/*) for sponsoring the journal and many outstanding students involved in e-Informatyka project.

The mission of the e-Informatica Software Engineering Journal is to be a prime international journal to publish research findings and IT industry experiences related to theory, practice and experimentation in software engineering. The scope of e-Informatica Software Engineering Journal includes methodologies, practices, architectures, technologies and tools used in processes along the software development lifecycle, but particular stress is laid on empirical evaluation.

There is evidence that software engineering researchers undertake relatively little empirical validation of their research [1, 2, 3]. Therefore the aim of the journal is to put a strong emphasis on empirical evaluation of novel approaches to software engineering. The journal's emphasis is in line with the ENASE series of conferences started by Leszek Maciaszek and Lech Madeyski (members of the editorial board) and Zbigniew Huzar (Editor-In-Chief) in 2006.

The first issue of the e-Informatica Software Engineering Journal includes five papers carefully reviewed by Editorial Board members, as well as by external reviewers, and then selected by the editors. Addressing the raised empirical validation issue, the first of the papers includes an empirical evaluation of refactoring technique. The second article explores some of the basic tenets of eXtreme Programming (XP) and agile methodologies and presents an analysis of an interview with two of the proponents and early participants in the "Agile revolution", Chet Hendrickson and Ron Jeffries. The third paper analyses to what extent the CMMI process areas can be covered by XP, and where adjustments of XP have to be made. The last two papers do not fall into an agile track. The forth paper identifies a program verification problem which is caused by the loose conventional object typing/subtyping, introduces object type graphs in which object component interdependencies are integrated into object types, and shows how the problem existing in conventional object type systems can be easily resolved. The last paper presents a user-centered approach to modelling business processes applying structured use case descriptions. You can download the abstracts and entire articles from the journal web site.

We look forward to receiving quality contributions from researchers in software engineering for the next issue of the journal.

Editors
Zbigniew Huzar
Lech Madeyski

References

[1] R. L. Glass, I. Vessey, and V. Ramesh. Research in software engineering: an analysis of the literature. *Information & Software Technology*, 44(8):491–506, 2002.

[2] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Trans. Software Eng.*, 31(9):733–753, 2005.

[3] M. V. Zelkowitz and D. Wallace. Experimental validation in software engineering. *Information & Software Technology*, 39(11):735–743, 1997.

e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

Agile Methods and CMMI: Compatibility or Conflict?

Martin Fritzsche*, Patrick Keil*

*Technische Universität München

fritzscm@in.tum.de, keilp@in.tum.de

Abstract

During the last years, agile methods like eXtreme Programming have become increasingly popular. Parallel to this, more and more organizations rely on process maturity models to assess and improve their own processes or those of suppliers, since it has been getting clear that most project failures can be imputed to inconsistent, undisciplined processes. Many organizations demand CMMI compliance of projects where agile methods are employed. In this situation it is necessary to analyze the interrelations and mutual restrictions between agile methods and approaches for software process analysis and improvement. This paper analyzes to what extent the CMMI process areas can be covered by XP and where adjustments of XP have to be made. Based on this, we describe the limitations of CMMI in an agile environment and show that level 4 or 5 are not feasible under the current specifications of CMMI and XP.

1 Introduction

Organizational maturity indicators like CMMI levels, SPICE ratings or specific ISO standards have become increasingly important for software development.

Customers or organizations that set up a distributed project often rely on them when selecting suppliers, since the results of these assessments and audits can serve as a 'signal' for their process maturity [8, 19].

In large organizations there are policies which enforce that all parts of the organization have to achieve certain maturity levels.

At the same time, agile methods continue to gain currency. This has also been true for larger projects, e.g. Cockburn and Highsmith cite successful agile projects with up to 250 people [6] and even for outsourcing and offshoring projects [10, 24, 26].

This leads to the challenge that, on the one hand, organizations often rely on CMMI as an indicator for process maturity (which is supposed to translate into product quality), on the other hand agile methodologies like XP [3], Scrum [25], Lean Development [23] or the Crystal methods [3] get more prominent.

It has been shown that projects that use agile methods with certain adjustments can achieve CMMI level 2 or even 3 [2, 17]. But from the various reports of successful agile projects it doesn't become clear how agile methods contribute to the fulfillment of process areas, where they have to be adjusted and where they are in conflict with CMMI goals.

Research should be conducted on how agile methods can be adapted to reach certain CMMI levels. This paper is meant as a starting point which reveals where adjustments have to be made.

Therefore, this paper takes a qualitative approach to analyze in how far agile methods support or conflict with CMMI process areas, where adjustments have to be made and if organizations employing agile methods can reach conformity with certain CMMI levels. After analyzing XP, we derive general statements and theses about the comparability and compatibility of CMMI and agile methods.

1.1 Related Work

Several authors have discussed the compatibility of CMMI and agile methods. Paulk [21] analyzes how XP can help organizations to reach the SW-CMM goals. While his work gives good insights into the interrelations between XP and CMM, the use of the now outdated SW-CMM limits the results. Our approach extends his work since we do explicitly show which process areas are in conflict with agile methods.

Kane and Ornburn [18] analyze which CMMI process areas are covered by XP and Scrum. Especially those areas related with process management are not considered by these two methods. Therefore, the authors propose tailoring of XP and Scrum to satisfy these goals. Unfortunately, most of the findings are not clearly derived. In addition, it is not discussed whether certain process areas are not addressed by agile methods or whether they are in conflict.

Finally, Turner and Jain [27, 28] show how CMMI can help to successfully implement agile methods. The difference to our approach is that we want to analyze how agile methods support CMMI and not vice versa.

2 Agile Methods

As an answer to the challenges of modern software development which in many cases cannot be tackled by 'traditional' processes, different 'lightweight' approaches have been established since the mid 1990ies that can be subsumed under the brand 'Agile Methods' [3, 6]. They ' "allow for creativity and responsiveness to changing conditions" [8] by emphasizing customer participation, quick reaction to requirements' changes and continuous releases [7, 14]. Some of them are rather a collection of techniques and activities than complete process models with precise definitions of roles, products, activities etc. But there are some methods, e.g. eXtreme Programming (XP) [3] or SCRUM [25], which are widely employed in projects of various sizes. Some concepts and ideas from the agile space have even been introduced into 'heavyweight' process models [1].

The characteristics of agile methods are elaborately defined in the twelve principles behind the agile manifesto [4, 5, 9]:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable develop-ment. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity - the art of maximizing the amount of work not done – is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These principles specify the four agile values [9] and provide a good summary of the intentions and ideas of agile methods.

3 Compatibility of Agile Methods with CMMI Requirements

3.1 CMMI – an Overview

The Capability Maturity Model for Software (CMM) [22, 15] developed by the Software Engineering Institute (SEI) has had a major influence on software process and quality improvement around the world [20]. Based on the first version released 1991, the Capability Maturity Model – Integrated (CMMI) has been presented in 2000, integrating CMM for Software (SW-CMM), the Capability Model for Systems Development (EIA/IS 731) and the CMM for Integrated Product Development (IPD-CMM).

Software Process Improvement (SPI) assumes that a well-managed organization with a defined engineering process is more likely to produce software that consistently meets the users' requirements within schedule and budget than a poorly managed organization with no such engineering process. "In other words, the project failure is usually a process failure" [8]. CMMI – as SPI's "de facto method" [29] – describes managerial processes to attack software development difficulties at five maturity levels:

1. initial
2. managed
3. defined
4. quantitatively managed
5. optimizing

It is important to note that the CMMI process models do not contain prescriptive processes that can be used right out of the box. Instead, CMMI provides a way to assess the state of an organization's ability to build software in a repeatable, predictable way [8].

Applying CMMI as a means to increase process capabilities is an organization-wide challenge. Herbsleb et.al. show that the average time for an organization to move up one level is between 21 and 37 months [13]. Over three quarters of the organizations reported that implementing any key SPI activity took longer than expected. But the effort pays off since "software process management maturity is positively associated with project performance" [16].

In order to reach a certain level, an organization has to fulfill all process areas of that level as well as those of lower levels. A process area is a summary of all requirements for a certain topic, e.g. project management, organizational training or causal analysis and resolution. To satisfy a process area all of its associated goals – specific ones and generic ones – have to be met. Specific goals apply to a process area and address the unique characteristics that describe what has to be implemented to satisfy the process area. To meet a specific goal CMMI suggests a set of specific practices. A specific practice is an activity that is considered important in achieving the associated specific goal. Generic goals are called "generic" because the same goal statement appears in multiple process areas. In the staged representation, each process area has only one generic goal. To meet a generic goal, CMMI suggests a set of generic practices. Generic practices provide institutionalization to ensure that the processes associated with the process area will be effective, repeatable, and lasting [15].

3.2 An Approach to Analyze the Coverage of Process Areas by Agile Methodologies

Our goal is to determine which of the CMMI process areas are supported by agile methods, where adjustments need to be made and which process areas are in conflict. In order to do so we analyzed every process area and all of its specific goals in detail [11]. The specific practices are only expected model components, meaning that their use is recommended but not necessary. CMMI states that they can be replaced by alternative practices. In fact, agile methods often employ different approaches than those suggested by CMMI. Therefore we concentrate on the analysis of the goals, using the practices only as guidelines and always looking for possible alternative ways of implementing the goals.

We also analyze the two generic goals ("institutionalize a managed process" and "institutionalize a defined process") and the generic practices, but only in general terms and not in conjunction with particular process areas. The reason for this omission is that agile methods do not directly address institutionalization practices. Institutionalization is a topic which has to be considered on the organizational level while agile methods only regard project level. Results in a detailed analysis of generic practices would be very limited.

For the coverage of specific goals, process areas and generic practices, a rating system is applied:

- Conflicting (–)
- Not addressed (0)
- Partially supported (+)
- Supported (++)
- Largely supported (+++)

"Largely supported" means that the agile method's practices, if employed correctly, satisfy the major part of the respective model component. "Supported" and "partially supported" describe a restricted coverage and "not addressed" reflects that there is no coverage at all. These ratings do not imply that the respective CMMI goals cannot be attained. They merely point out that additional practices have to be introduced to fully satisfy the CMMI requirements. "Conflicting" on the other hand indicates that the respective CMMI goal cannot be reached with the agile method being used. This rating is given if there are no possible extensions that do not interfere with the method's basic practices or the agile principles. To differentiate between "not addressed" and "conflicting", we therefore always had to check whether the agile method could be extended to reach the CMMI goal without interfering with the method's basic practices or contradicting to the principles stated in the agile manifesto.

3.3 Applying the Approach to eXtreme Programming

In this chapter we apply our approach to XP and show the interrelations and conflicts between XP and the CMMI process areas and all of their associated specific goals. To not go beyond the scope of this paper we will condense the analysis. [11] provides a more detailed presentation and a discussion of Scrum.

3.3.1 Analysis of Process Areas and Their Specific Goals

Requirements management – Manage requirements (+++)

Understanding of the requirements is obtained through the integration of the customer into the team and the resulting intensive communication with the customer. The project participants' commitment to the requirements is obtained in the planning phase. Changes of requirements are quickly exchanged and discussed. Even if traceability of requirements is not an explicit goal of XP, it is supported by stories, tasks, functional tests that detect inconsistencies between project work and requirements, and by unit tests. XP's practice of throwing away story cards that already have been realized can prove to be problematic. To better implement this process area story cards should be kept. Thus, traceability can be extended by keeping record of previous story cards and old versions of the documentation.

Project planning (+++)

Establish estimates (+++)

Estimates for stories and tasks are established and can be corrected during the project. The estimates' precision is increased through a short planning horizon due to short iterations.

Develop a project plan (+++)

The project plan is established through XP's release and iteration plans that evolve throughout the project. Therefore long term plans remain vague and only short term plans are detailed. Risks are identified, training needs are planned and the involvement of all relevant stakeholders is assured if XP is applied correctly.

Obtain commitment to the plan (+++)

Commitment to the release and iteration plans is obtained through the high involvement and responsibility of all team members.

Project monitoring and control (+++)

Monitor project against plan (+++)

Schedule and estimates are monitored by the tracker. Information on the project's progress is gathered by the use of measures. The intensive communication among the team members and with the customer helps to convey that information. Milestones are checked against the schedule by functional tests. The strict system of short iterations and the regular commitments to the plan make it easier to monitor the project against the baseline.

Manage corrective action to closure (+++)

Issues that demand corrective actions are informally collected and analyzed. Corrective actions can be adjustments of the method and also of the functionality that will be realized. In addition new iterations always offer good opportunities to make adjustments.

Supplier agreement management (0)

This process area is not addressed by XP. We believe that the method can be extended to fulfill the goals of this process area. However, involving suppliers could be problematic for agility if it hinders iterative development. There are cases where supplied components are needed to obtain functioning software at the end of an iteration. It can pose a critical problem if they are not available at that point.

Measurement and analysis (+)

Align measurement and analysis activities (+)

The only measurement objective is progress control. Measurements and analysis procedures are defined by the tracker. XP provides no specific guidelines for these tasks.

Provide measurement results (++)

The measurement data is obtained through intensive communication within the team. The tracker analyzes the data and conveys the results to the team using wall charts. The data is usually not permanently stored. However, there are many tools available for effort estimation and tracking for agile teams. By using these tools the measurement data and results can be stored permanently without too much effort.

Process and product quality assurance (+)

Objectively evaluate processes and work products (+)

XP doesn't demand an explicit evaluation of processes, work products and services against the applicable process descriptions. The only instrument of controlling that the method is applied in the right way is the coach who guides the team in the use of XP.

Provide objective insight (+)

Quality issues can be easily communicated in an XP team. The work of the coach supports this specific goal. However, there are no strict guidelines for the resolution of noncompliance issues and the establishing of records of quality assurance activities.

Configuration management (+++)

Establish baselines (+++)

Configuration items are code, design, tests and requirements. The use of a configuration management system is recommended since continuous integration relies heavily on it. Baselines are established regularly through functional tests. In addition, baselines are created at the end of each iteration.

Track and control changes (+++)

Changes are controlled and tracked through various practices like pair programming, tests, customer collaboration, etc.

Establish integrity (+++)

XP enforces continuous integration. Code is easy to read because of coding standards and therefore its own description. Audits are informally performed through pair programming, customer involvement and testing.

Requirements development (++)

Develop customer requirements (++)

The customer elicits requirements and specifies them in story cards and functional tests. The developers often support him in these tasks. The requirements specification however remains quite vague. Details have to be discussed directly with the customer during development.

Develop product requirements (++)

Customer requirements are refined into product requirements. These are specified using task cards. They remain relatively vague too.

Analyze and validate requirements (++)

An analysis of requirements is carried out in a well-defined way. The programmers consult the customer during requirements elicitation. In addition, the acceptance of changing requirements and the use of iterations allow constant analysis and validation of requirements. Operational concepts and scenarios are established using functional tests. However there is no in depth requirements analysis up front.

Technical solution (+++)

Select product-component solutions (+++)

Alternative solutions are explored at the beginning of the project through prototypes and later on through refactoring and iterative development.

Develop the design (+++)

A design as simple as possible is developed. Code is used as a design document. Design is carried out iteratively.

Implement the product design (+++)

XP employs a variety of implementation practices, e.g. refactoring, coding standards, pair programming. A product support documentation is developed if it is requested by the customer.

Product integration (+++)

Prepare for product integration (+++)

XP employs continuous integration and since integration steps are performed very often, a thorough preparation is critical.

Ensure interface compatibility (+++)

Interface compatibility is ensured by running all tests at each integration step.

Assemble product components and deliver the product (+++)

Component assembly and delivery is carried out. The use of continuous integration and direct customer involvement further helps to achieve this goal.

Verification (+++)

Prepare for verification (+++)

Verification is carried out through intensive testing. The preparation is therefore concentrated on this topic. A test framework should be used and hence according preparation activities executed. Furthermore XP employs a test-first approach. All tests have to be written before the code.

Perform peer reviews (+++)

Peer reviews are implicitly always part of XP. Pair programming, refactoring and the principle of collective code ownership imply constant peer reviews.

Verify selected work products (+++)

Methods for verification are mainly peer reviews and testing, which both are performed constantly.

Validation (+++)

Prepare for validation (+++)

Validation is performed in XP projects through customer participation and frequent releases. The main criterion for validation is acceptance by the customer.

Validate product or product components (+++)

The customer constantly validates the work done by the team. This is possible because he is integrated into the team. In addition he validates the deliveries at the end of each iteration. This may result in additional or changed requirements specified by the customer. The enormous influence of the customer improves the chances that the product is suitable for use in its intended operating environment.

Organizational process focus (–)

This process area isn't addressed because it applies to the organization while XP only applies to a project. It even is in conflict with XP: like in other agile methods, adjustments are often done during a project. These improvements, however, are limited to the current project since they shall not be documented. Knowledge about improvements is linked to people. Other projects can benefit if people are moved between projects. But the problem is that in big organizations there are too many projects. In that case such a practice cannot let all of them benefit from a particular project's experience. In addition the information is not permanent since people can retire or change organization. The conflict can be eased by establishing organization-wide repositories storing best practices of previous projects or by institutionalizing the exchange of lessons learnt between projects.

Organizational process definition (0)

Organizational training (++)

Establish an organizational training capability (++)

Training is carried out by XP during the exploration phase. Therefore an XP project requires organizational training capabilities. Pair programming and coaching can also be regarded as training, so XP further enhances the organization's training capabilities.

Provide necessary training (++)

As stated above, training is carried out explicitly during the exploration phase and implicitly during the whole project through coaching and pair programming. Through the latter, there are however deficiencies regarding the establishment of records and the assessment of training effectiveness.

Integrated project management (++)

Use the project's defined process (0)

Coordinate and collaborate with relevant stakeholders (+++)

XP integrates and coordinates developers, customer, testers, and management.

Use the project's shared vision for IPPD (+++)

XP contributes a lot to the project members' integration and their close collaboration. This and the intensive communication within the team help to establish a shared vision.

Organize integrated teams for IPPD (0)

Risk management (+++)

Prepare for risk management (+)

XP doesn't explicitly state how risk management is to be conducted. But XP projects surely make some sort of preparation.

Identify and analyze risks (+++)

XP enforces the identification and analysis of risks during the planning phase.

Mitigate Risks (+++)

The flexibility gained by the use of short iterations is a potent instrument to mitigate risks.

Integrated teaming (+++)

Establish team composition (+++)

XP establishes a self-organizing cross-functional team in which all relevant stakeholders are integrated.

Govern team operation (+++)

Team operation is governed through a clear definition of the different roles, pair programming, collective ownership of the code and the focus on cooperation and communication.

Integrated supplier management (0)

Decision analysis and resolution (–)

Turner [27] points out that the ability to adapt quickly to new situations is preferred by agile methods to a formal evaluation process. XP identifies and evaluates alternatives informally and not in the way CMMI suggests.

Organizational environment for integration (+)

The issues of this process area are addressed at project level but not at the organizational level.

Provide IPPD infrastructure (++)

XP establishes the basis for this specific goal through the introduction of tools, intensive communication and cooperation. By promoting the abilities to communicate and cooperate as well as leadership skills the method further supports this goal.

Manage people for integration (+)

Leadership mechanisms are democratic within the development team. However the customer and the big boss have authority to decide on high level issues.

Organizational process performance (–)

XP focuses rather on individuals than on issues that are as process oriented as this process area. Turner [28] points out that the idea of measuring a process and maintaining baselines and models is in conflict with the agile manifesto.

Quantitative project management (–)

Statistical methods have their focus on defined processes and not on individuals since quantitative analyses need a static baseline. Therefore, statistical methods are in conflict with agile principles. Furthermore, they rely on the law of big numbers and on averaging out effects in large teams. Since most agile software projects are small the use of statistics is questionable.

Organizational innovation and deployment (–)

Process improvements and adaptations are made only within projects and not documented, so that they cannot be propagated to the whole organization. This topic relies heavy on "organizational process focus", a process area that is in conflict with XP.

Causal analysis and resolution (0)

3.3.2 Generic Practices

Establish an organizational policy (0)

Plan the process (0)

Provide resources (+)

This practice is conducted only regarding a few process areas.

Assign responsibility (+++)

The role model assigns responsibilities to certain team members. In addition the developers take responsibility for particular tasks during the project.

Train people (+++)

Training is conducted during the exploration phase. Furthermore pair programming and coaching is employed to train people.

Manage configurations (++)

A configuration management system is employed. The configurations of code, tests, design and requirements are managed. For protocols of test cases, measurement data, release and iteration plans configuration management isn't planned.

Identify and involve relevant stakeholders (+++)

All relevant stakeholders are part of the team.

Monitor and control the process (++)

This generic practice is implemented for all project-related process areas due to XP's fulfillment of the process area "project monitoring and control". To realize it for all processes and not only for project-related processes, measures for monitoring actual performance of the process have to be established.

Objectively evaluate adherence (+)

The coach is XP's only instrument to support this generic practice. However by implementing the process area "process and product quality assurance" which isn't in conflict with XP it would be possible to fulfill this practice for all process areas.

Review status with higher level management (++)

Frequent releases enable reviews by the management.

Establish a defined process (0)

Collect improvement information (–)

Improvements are deliberately not documented by XP and therefore this generic practice cannot be implemented. This conflict could be solved by properly documenting process changes in a project and making them available to other projects in the organization. In addition, process improvement information might be easily captured during iteration planning and via postmortem analyses.

3.4 Coverage of Process Areas by Agile Methodologies

In 3.3., we showed in detail which of the CMMI process areas are supported by XP and which are in conflict. In this section, we give a summary on the coverage of CMMI process areas by XP and Scrum.

All of the seven process areas of CMMI level 2 are attainable by both methods. From the fourteen process areas of level 3 only two are in conflict. Three out of the four process areas of level 4 and 5 are also in conflict. Of the twelve generic practices only one was rated as in conflict.

The results indicate that level 2 can be attained without major adaptations. The same is true for level 3 with the exception of two process areas. It is however practically impossible to reach level 4 and 5 with XP and Scrum without making changes to the methods that contradict agility.

Mainly those process areas that deal explicitly with process improvement ("organizational process focus", "organizational process performance", "quantitative project management" and "organizational innovation and deployment") are in conflict with agile methods. Also the generic practice "collect improvement information" deals explicitly with process improvement and is in conflict. In addition "decision analysis and resolution" interferes with Scrum and XP due to the demand of a formal evaluation process.

The major part of the process areas can be attained by agile methods. But often, the methods have to be extended by additional practices to fully satisfy the process areas. The coverage of all process areas and generic practices is shown in Table 1 and Table 2, with those written in italics where XP and Scrum differ.

Process area	XP	Scrum
2.1 Requirements management	+++	+++
2.2 Project planning	+++	+++
2.3 Project monitoring and control	+++	+++
2.4 Supplier agreement management	0	0
2.5 Measurement and analysis	+	+++
2.6 Process and product quality assurance	+	0
2.7 Configuration management	+++	0
3.1 Requirements development	++	++
3.2 Technical solution	+++	0
3.3 Product integration	+++	0
3.4 Verification	+++	0
3.5 Validation	+++	+++
3.6 Organizational process focus	-	-
3.7 Organizational process definition	0	0
3.8 Organizational training	++	+
3.9 Integrated project management	++	+++
3.10 Risk management	+++	+++
3.11 Integrated teaming	+++	+++
3.12 Integrated supplier management	0	0
3.13 Decision analysis and resolution	-	-
3.14 Organizational environment for integration	+	+
4.1 Organizational process performance	-	-
4.2 Quantitative project management	-	-
5.1 Organizational innovation and deployment	-	-
5.2 Causal analysis and resolution	0	0

Table 1: Coverage of CMMI process areas by XP and Scrum

Generic practice	XP	Scrum
2.1 Establish an organizational policy	0	0
2.2 Plan the process	0	0
2.3 Provide resources	+	+++
2.4 Assign responsibility	+++	+++
2.5 Train people	+++	+++
2.6 Manage configurations	++	0
2.7 Identify and involve relevant stakeholders	+++	+++
2.8 Monitor and control the process	++	++
2.9 Objectively evaluate adherence	+	0
2.10 Review status with higher level management	++	+++
3.1 Establish a defined process	0	0
3.2 Collect improvement information	-	-

Table 2: Coverage of CMMI generic practices by XP and Scrum

There are only minor differences between the ratings of Scrum and XP. Scrum, not addressing development activities, gets lower ratings than XP in accordant process areas ("configuration management", "technical solution", "product integration" and "verification"). On the other hand, Scrum performs slightly better in process areas dealing with project management ("measurement and analysis" and "integrated project management for IPPD") and according generic practices ("provide resources" and "review status with higher level management").

Our analysis shows that XP and Scrum cover only project related, but not process related process areas.

3.5 Interrelations between Agile Methods and Process Maturity Models

CMMI evaluates an organization as a whole and its development processes. In contrast, an agile method is (a framework or sometimes only a fragment of) one individual development process. Thus, the concepts are not comparable per se. Their focus is different, but still they have interrelations. Paulk summarized that CMM is a method for software management whereas agile approaches are methods for software development [21]. They not only can coexist, but they even support each other [12].

We are convinced that CMMI is an appropriate way to improve processes also in an agile environment. Checking an agile method's coverage of the process areas reveals shortcomings in the approach and thereby improvement potentials. However, process improvement with CMMI can only be carried out up to a certain degree since there are several process areas which are in conflict with agile principles. Some process areas of level 3 and most of level 4 and 5 are unattainable without sacrificing some agile bedrocks. This would weaken the agile method and eliminate several of its benefits. Also, such actions would be contradictory to the aim of CMMI, i.e. improving the process by making the agile method as good as possible and not turning it into a different kind of method which isn't agile anymore. So we conclude that the best improvement approach in an agile environment is to stop at CMMI level 3.

Implicitly, we conclude that CMMI levels have to be judged considering the process model employed in an organization. But also, like for every traditional process, refining the agile processes needs to be regarded as an ongoing, success-critical task.

4 Conclusion

We analyzed in detail which CMMI process areas can be covered by Scrum and XP. We identified process areas where the methods have to be adjusted to fulfill CMMI goals. Some process areas were in conflict with the two methods and agile principles in general. Most of the process areas can be fulfilled using agile methods. However some are clearly in conflict. Through the use of CMMI, shortcomings of agile methods can be identified. We therefore come to the conclusion that process improvement with CMMI can also be carried out when using agile methods. However, since some process areas, mainly those of the maturity levels 4 and 5, are in conflict with agile principles, agile methods can be applied without any major adaptations up to level 2 and up to 3 with some minor changes described in this paper. Extending the project focus of agile methods to an organization-wide perspective would help to make use of the existing concepts of ongoing process-improvement.

If these concepts are employed in agile environments, agile methods will further gain acceptance. But today, an obstacle for process improvement with CMMI is the difficulty to carry out assessments of projects which use agile methods. The specific practices suggested by CMMI often differ from agile approaches. Assessors therefore encounter serious problems when trying to analyze a project. To remedy this situation a catalogue of practices and sub-practices that are typically used by agile methods to implement CMMI goals should be developed.

Here, we only discussed XP and Scrum. To make our results more general, further agile methods should be analyzed as well. In addition, concrete guidelines should be established which show how agile methods can be enhanced to fully cover all the process areas that are not in conflict. For this our work can be seen as a starting point.

References

[1] V-Modell XT Portal. http://www.v-modell-xt.de.

[2] D. J. Anderson. Stretching Agile to fit CMMI Level 3 - the story of creating MSF for CMMI Process Improvement at Microsoft Corporation. In *AGILE*, pages 193–201, 2005.

[3] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2nd edition, 2004.

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development, accessed in 2006. http://AgileManifesto.org/.

[5] A. Cockburn. *Agile Software Development.* Addison-Wesley, 2002.

[6] A. Cockburn and J. Highsmith. Agile Software Development: The People Factor. *IEEE Computer*, 34(11):131–133, 2001.

[7] D. Cohen, M. Lindvall, and P. Costa. Agile Software Development: A DACS State-of-the-Art Report. Technical report, 2003. http://www.thedacs.com/techs/agile/agile.pdf.

[8] M. Doernhoefer. Surfing the net for software engineering notes. *SIGSOFT Softw. Eng. Notes*, 31(1):5–13, 2006.

[9] C. Dogs and T. Klimmer. *Agile Software-Entwicklung kompakt.* mitp-Verlag, 2005.

[10] M. Fowler. Using an Agile Software Process with Offshore Development, accessed in 2005. http://www.martinfowler.com/articles/agileOffshore.html.

[11] M. Fritzsche. Agile Methoden im industriellen Umfeld. Master's thesis, Technische Universität München, 2005.

[12] H. Glazer. Dispelling the Process Myth: Having a Process Does Not Mean Sacrificing Agility or Creativity. *CrossTalk: The Journal on Defense Software Engineering*, (14):27–30, 2001.

[13] J. D. Herbsleb, D. Zubrow, D. Goldenson, W. Hayes, and M. C. Paulk. Software Quality and the Capability Maturity Model. *Commun. ACM*, 40(6):30–40, 1997.

[14] J. Highsmith. Extreme Programming: Agile Project Management Advisory Service White Paper, accessed in 2005. http://www.cutter.com/freestuff/ead0002.pdf.

[15] S. E. Institute. Capability Maturity Model Integration (CMMI), Version 1.1 (CMMI-SE/SW/IPPD/SS, V1.1). Technical report, Software Engineering Institute, Carnegie Mellon University, 2002.

[16] J. J. Jiang, G. Klein, H.-G. Hwang, J. Huang, and S.-Y. Hung. An exploration of the relationship between software development process maturity and project performance. *Inf. Manage.*, 41(3):279–288, 2004.

[17] T. Kähkönen and P. Abrahamsson. Achieving CMMI Level 2 with Enhanced Extreme Programming Approach. In *PROFES*, pages 378–392, 2004.

[18] D. Kane and S. Ornburn. Agile Development: Weed or Wildflower? *CrossTalk: The Journal on Defense Software Engineering*, 2002.

[19] P. Keil. Principal agent theory and its application to analyze outsourcing of software development. In *EDSER '05: Proceedings of the seventh international workshop on Economics-driven software engineering research*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[20] M. C. Paulk. Using the Software CMM With Good Judgment. *ASQ Software Quality Professional*, 1(3), 1999.

[21] M. C. Paulk. Extreme Programming from a CMM Perspective. *IEEE Software*, 18(6):19–26, 2001.

[22] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber. Capability Maturity Model, Version 1.1. *IEEE Softw.*, 10(4):18–27, 1993.

[23] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[24] R. S. Sangwan and S. P. Masticola. Model-Driven Rapid Application Development: A Framework for Agile Development in Outsourced Environments. Technical report, Siemens Corporate Research, 2004.

[25] K. Schwaber and M. Beedle. *Agile Software Development with Scrum.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[26] M. Simons. Internationally Agile, accessed in 2005. http://www.informit.com/articles/article.asp?p=25929.

[27] R. Turner. Agile Development: Good Process or Bad Attitude? In *PROFES*, pages 134–144, 2002.

[28] R. Turner and A. Jain. Agile Meets CMMI: Culture Clash or Common Cause? In *XP/Agile Universe*, pages 153–165, 2002.

[29] R. van Solingen. Measuring the ROI of Software Process Improvement. *IEEE Software*, 21(3):32–38, 2004.

e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

An Empirical Evaluation of Refactoring

Dirk Wilking*, Umar Farooq Khan*, Stefan Kowalewski*

**Embedded Software Laboratory, RWTH Aachen University*

`wilking@informatik.rwth-aachen.de`, `umar.khan@ixi.informatik.rwth-aachen.de`, `kowalewski@informatik.rwth-aachen.de`

Abstract

This paper presents a process evaluation for the agile technique of refactoring based on the language C. The basis for this evaluation is made up by an experiment which is targeted on the aspects of increased maintainability and modifiability. Although the maintainability test shows a slight advantage for refactoring, results show no significant strength here. Concerning modifiability, the overhead of applying refactoring appears to even weaken other, positive effects. The analysis of secondary variables provides hints on advantages of the refactoring technique like reduced resource consumption and a reduced occurrence of complicated control structures.

1 Introduction

Maintenance of software is reported as a serious cost factor [24]. One solution proposed to reduce maintenance effort is refactoring [8] which is a method to continuous restructure code according to implicit micro design rules. Its new aspects are the smooth integration into an existing development process where it is used continuously in the background. Developers are forced to think about their code structure and to identify parts which "smell" - which is the best description that can be given for this subjective concept. After identification, the according code is changed based on a catalogue of change steps referring to the problem. These steps range from renaming of variables, extraction of methods to the extraction of complete classes from the existing code.

Refactoring is assumed to positively affect non-functional aspects, presumably extensibility, modularity, reusability, complexity, maintainability, and efficiency as stated in [24]. However, additional negative aspects of refactoring are reported, too. They consist of additional memory consumption, higher power consumption, longer execution time, and lower suitability for safety critical applications.

Most research concerning refactoring is done on the technical side in order to apply refactoring in a computer aided way. The general aim here is to either integrate a new technical aspect to refactoring like languages ([15, 21]), to support refactoring by a tool for analysis ([22, 30, 31, 35]) or to support the actual execution of refactoring ([10, 23]). Empirical evidence of the effect of refactoring is rarely to be found. One example for an empirical evaluation is the influence of refactoring on changeability as evaluated in [12] reporting a lower change effort. Other empirical results provide a taxonomy for bad smells as presented in [22].

Experience reports show a mixed picture of refactoring. One application of refactoring is reported to show non-satisfactory results [28]. It is reported that bad tool support along with the size of a legacy system created the problems. A code evolution analysis [19] investigates one of the main artifacts minimized by refactoring: copied code (code clones). It states that not every code clone should be subject to refactoring and that for some clones, appropriate refactorings are missing. One successful examination in terms of an increase in program performance due to refactoring is reported in [13]. A secondary, nonetheless interesting aspect mentioned there is the compliance to the design principle of information hiding after the application of refactoring.

Concerning agile methods in general, a limited empirical evaluation took place so far [2]. Most work has been done for pair programming [1] as this seems to be the most important aspect of extreme programming. As in addition refactoring is a major technique which can be used on its own, this report presents an experiment which intends to help assessing agile methods and this technique more precisely.

2 Design of the Experiment

The general approach followed by this experiment consisted of a group of 12 students using the same requirements specification to develop a program. Six students used refactoring continuously during development while the rest was asked to continuously document each function programmed. The assignment to a treatment group was done at random. The later treatment is regarded as a placebo in order to omit a Hawthorne effect [29] and to apply the same level of disturbance to this control group.

As the effects of refactoring were assumed to have an impact on non-functional aspects, two hypothesis were of special interest. The first one was the effect of refactoring on maintainability. Regarding this aspect, a direct evaluation method as proposed in [17] which is mainly based on a metric definition was not done. As in this case participants were available, a measurement with the help of the participants was done. Maintainability was tested by randomly inserting defects into the code and measuring the time needed to fix them (thus classified as corrective maintainability [3]). The second hypothesis was an improved modifiability caused by refactoring. In order to test this, small additions were added to the specification as new requirements and the time and physical lines of code (LOC) needed to implement them were measured.

2.1 Variables and Measurement

The independent variable of this experiment was the treatment which was a single, dichotomous factor. Either a participant was assigned to the refactoring or to the documentation treatment. In order to control the execution of the particular treatment, a simple tool was established disturbing every participant every 20 minutes. During each disturbance, the participant was asked to either work on a refactoring checklist or to document the last functions he programmed. In the case of documentation, changing the code was prohibited during this step.

One dependent variable of this experiment was the LOC metric together with the time to implement a new version based on additional features. LOC is considered to be a rough measure for the size of the resulting product. Both were used to measure the additional effort a developer needed to add new, unmentioned features to his code. These two thus were regarded as an indicator for system modifiability.

For maintainability, a special test was prepared. It consisted of a time measurement for the fixing task of randomly induced syntactical and semantical failures. These were directly created in the participants source code by randomly removing lines of source code. The tests consisted of a short description of the failure (in case of a semantical failure) and the measuring consisted of the time needed to locate and fix them. The measuring was done in seconds and supervised by a member of the chair.

A measurement of a difference in the abstract syntax tree is currently executed in order to assess a general difference in the micro structure of the different versions (cf. [14, 18, 20]).

2.2 Hypothesis

The main hypothesis of an improved modifiability for different versions measured by the time t was formalized by

$$H_0 : \bar{t}_{mod_{Ref}} \geq \bar{t}_{mod_{Doc}}$$

with $\bar{t}_{mod_{Ref}}$ being a version's mean development time for the refactoring group and $\bar{t}_{mod_{Doc}}$ being the according value for the documentation group. Thus, the resulting alternative hypothesis was

$$H_1 : \bar{t}_{mod_{Ref}} < \bar{t}_{mod_{Doc}}$$

Concerning corrective maintainability, the corresponding hypothesis was that the measured time for maintainability $\bar{t}_{main}$ during the maintainability test was greater for the documentation group leading to the null hypothesis of

$$H_0 : \bar{t}_{main_{Ref}} \geq \bar{t}_{main_{Doc}}.$$

The expected hypothesis thus was

$$H_1 : \bar{t}_{main_{Ref}} < \bar{t}_{main_{Doc}}.$$

2.3 Procedure

The execution of the experiment started with a video introduction explaining the microcontroller, the development environment, and the general conditions of the experiment. Only the last video was different for each participant group as it either explained the refactoring or documentation task. By using videos it was made sure that each participant received the same introduction and that no treatment group was favored. After that, an initial survey was carried out in order to assess the participant's overall programming knowledge and knowledge about refactoring. In order to avoid motivation effects refactoring was named reorganization within the documents and videos. Additionally, the participants were asked to develop the software without any additional software engineering techniques to avoid interference of other factors. At last, the participants were not told what kind of measurement was done in the end in order to avoid preparation for requirement additions in terms of architecture.

The development started by reading the requirements document which was the same for all participants. After that, programming started until each requirement was implemented. The development task consisted of a game based on a reaction and a memorization part. The reason for this type of application was the low domain knowledge required. In addition, different types of hardware programming were needed in order to use the buttons (with debounce), LCD, LEDs and interrupts.

Concerning the execution of refactoring, only a subset of applicable refactoring steps was chosen with the addition of macro refactorings as discussed in [9] and [10]. The reason for excluding certain refactorings is the utilization of the language C. Only non-object oriented programming features were used during this experiment.

As the participants were not supposed to be accustomed to refactoring, a special, controlled execution was intended. First, the frequency of refactorings was set to a rate of 20 minutes. This was done to assure continuous refactoring together with a reminder of executing refactoring at all. The disadvantage of this approach are the occasions where a refactoring was initiated without the actual need for it. As the execution of refactoring steps was uncommon and the perception of bad smells was not based on participant experience, a checklist based on [8] was used in order to control both aspects. The execution of a refactoring is regarded non problematic whereas the detection of bad smells is subject to personal interpretation because of the nature of this term. Thus, only an informal description of this basic concept was given.

The final code size differed between individuals and was not affected by the treatment. The size ranged from 745 to 2214 lines of code. For each version, an acceptance test was executed checking the basic functionality and new features which were added. In case of an imprecise requirement definition, the implementation was accepted in the way the participant understood the requirement.

2.4 External Conditions and Limitations

The time span for this experiment was 3 months. During this time, all participants worked on the tasks until they finished them or the maximum of 40 hours was reached. Each participant worked in a different room and a simple room management was done as only three different rooms were available. The event that a participant wanted to work and no room available could be circumvented by this. Files were separated on network drives so that no participant could see the results of the other. The complete development environment was accessible in each room and participants worked on their own. Interruption sometimes occurred, but the frequency was not very high. For questions, an instant messaging server was setup and all messages were logged which was known and had to be accepted by the students.

2.5 Participants

The experiment was carried out with twelve graduate students. All of them were students at the RWTH Aachen University. The experiment had been advertised on the university's mailing list, notice boards and in the courses. Applications from 14 students had been received of whom 12 students had been selected randomly. Their field of study was mainly computer science, with one participant working in the field of mechanical engineering. All participants were paid and received a forty hour student helper contract. The students had programming knowledge of Java, whereas the language C was new to some. As mentioned above, refactoring was new to them except for one student who had practical knowledge.

As explained in [33], this type of participants is sufficient for evaluating basic effects or an initial hypothesis. In addition, [16] states that at least last-year software engineering students have a comparable assessment ability compared to professional developers and in [4] no general difference could be found for different programming expertise between these groups.

2.6 Technical Background

The experiment used an ATMEL ATmega16 microcontroller clocked with 6MHz as development platform. The software was written in C and developed with WINAVR 2 and ATMEL AVR Studio 3. For the LCD programming, an additional C-header was given to the students as this was regarded standard. Some tools were used in the background which comprised the disturber mentioned above and a code gathering tool which copied the code base every time a compilation was done. This last step was done in order to study code evolution.

3 Validity

This section critically examines practices and ancillary conditions. The procedure, measurements and theoretical concepts are structured as proposed in [36].

3.1 External Validity

Although in general students can be regarded as average programmers, they do not represent the often demanded professional developers. As stated above, they are regarded sufficient to show an effect within an initial method evaluation [33]. Regarding the treatment, the use of additional, unmentioned features can be regarded as in favor for refactoring. The event of changing requirements and thus the need for new features is not regarded artificial but normal industrial development. Regarding the environment, especially the lack of an object oriented language might have changed the influence of refactoring. This is not regarded as an artificial interaction because refactoring is regarded a method that can be applied in general to improve the design of a program. Technical factors like an exceptional good development environment or a method specific language might blur a method's effect and thus the lack of it is not regarded problematic.

3.2 Internal Validity

One of the major internal threats is the application of refactoring itself consisting mainly of a checklist and a periodical call for the application of it. This artificial treatment was chosen because of the high control. The downside of this is that the concept of a bad smell may require much more experience than provided by the checklist and that the application of refactoring may require a higher degree of freedom for an individual developer than allowed by such a list.

History and maturation are not regarded a threat as there is no repeat in the sense of reoccurring tasks or measurements except for the maintainability test were the code knowledge may have increased for each test. As an additional precaution, the main measurement tasks (additional requirements and failure inducing) were not known to the participants so that they could not prepare their code for this.

As some participants could not fulfill the requirements for all versions, they may have suffered from demoralization effects. But because of the fact that each participant worked on his own and no results where revealed to others, social threats are regarded a minor threat.

Concerning the communication between participants, only a contract specifying the participants duties and rights could be used as controlling device. As the development took a few weeks per participant, the possibility of private communication could not be eliminated.

3.3 Construct Validity

A clear theory in the sense of an abstraction of the effects is not easy to define for refactoring. There are several effects which are accumulated in the term refactoring. One of the major points is the abstract design principle of "once and only once" suggested by the inherent term "factor" [8]. Another effect might be the constant rereading and rethinking of existing code. By this, a continuous awareness of all parts of the source code might by achieved revealing positive effects like simpler reuse of code and faster navigation. This might be considered a constant reviewing process, too. One last aspect is an implicit effect of refactoring with the existence of a good structure being indirectly postulated. This effect may force developers to maintain a certain quality for every part of the code which may not be the case for non-refactoring based development.

However, by following the combined approach of bad smell and collection of refactoring steps, the common usage of this technique is adopted and its general influence assessed. Consequently, an abstract construct was not used.

Concerning the outcome expected to be caused by refactoring, only the variables of maintainability and modifiability were measured. For other non-functional aspects like modularity, reusability, complexity, and efficiency no direct measurement construct could be found. The quality of the actual measurement consisted only of a single variable for maintainability, whereas multiple measurements in the form of LOC and time were done for modifiability.

The generalizability of the treatment suffers from the hard 20 minute interrupts. One the one hand, as refactoring is executed on demand when a problem has been discovered, this treatment is artificial. On the other hand, it is the only way of assuring a constant execution. In addition, the reminder of looking for bad code aspects is regarded helpful for unexperienced participants. The general idea of changing bad code continuously thus is considered as maintained.

3.4 Conclusion Validity

Concerning the experiment's power, the low number of participants ($n = 6$) is a problematic point. Power is described in [5] as the probability of rejecting the null hypothesis and thus directly describes how good the experiment can show an effect. As the importance of that aspect may be exaggerated (compare [26] to [25]) given the quality of variables for empirical software engineering, the value for n still is too low. As described in [7], a bootstrap power calculation can be done by sampling (with replacement) a higher number of participants based on the original data. Table 1 depicts the probability p of showing a difference of the mean fixing time of 12 seconds or more. This can be regarded a rough indicator, as a only point estimator is used and 12 seconds is a rather low difference (five percents regarding the mean fixing time of 240 seconds). Regarding a refactoring group size of 48 participants, the experiment starts to have an appropriate probability of showing the expected effect. An interesting application of this sample size oriented power calculation is proposed in [32] suggesting a continuous review of an experiment's power.

N:	6	12	24	48
$p(d \geq 12)$	0.68	0.74	0.83	0.91

Table 1: Power calculation of a difference in means of 12 seconds for different sample sizes N

As the hypothesis and the assumed effects of refactoring have been clearly stated, "fishing for results" may only occur for secondary variables for this experiment. Nevertheless, these variables are investigated and interpreted, as they may give ideas for other effects caused by refactoring. Their unreliable nature (significant results cannot be regarded as such) is emphasized in the text.

The reliability of the measures is difficult to assess. LOC is always a point of discussion, but it nevertheless can be regarded a rough measure for system size. The measurement of relative time (compared to the first, full version) used to assess modifiability has the advantage that it includes the main benefit expected for refactoring: a decrease of effort when adding features. In addition, this variable is simple to measure. The special test for maintainability which randomly induces failures into the participant's code simulates the same effect as a real case of corrective maintainability: a system failure is reported, its cause has to be found in the code and it has to be fixed. Its reliability is regarded above average as time is used as main variable and the failure creation is based on a random process.

4 Analysis

4.1 Main Hypothesis

4.1.1 Maintainability

The measurement of maintainability, which consisted of a random insertion of 15 syntactical and 10 non-syntactical errors, was measured in seconds. The errors were created by removing lines of code randomly. The resulting error were divided into syntactical and non-syntactical nature. Because of the randomization and the rather uncommon test method, a more detailed rating of the severeness of an error was note feasible. The results were gathered for all twelve participants and the corresponding mean error correction times were aggregated into the box plot of figure 1. Here, a minor advantage for the refactoring treatment can be seen, but the results were not significant when a bootstrap test was executed for $\alpha = 0.05$. The assumption of better maintainability thus cannot be answered according to this, but the slightly lower value for the refactoring treatment lead to the impression of only a minor effect of refactoring.

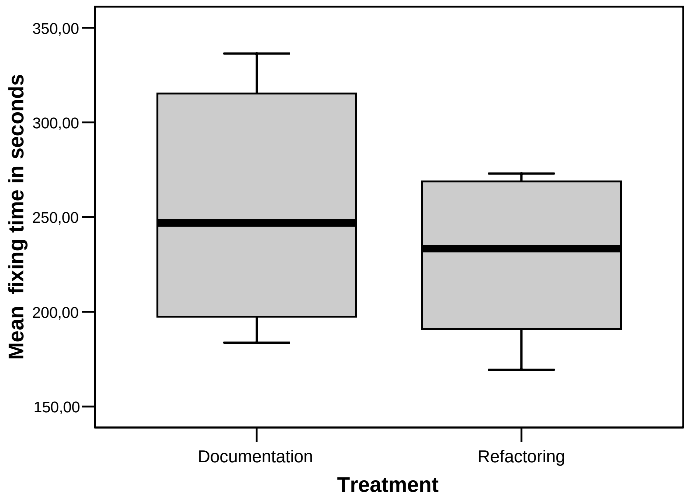


Figure 1: Box plot of mean fixing time of each participant divided by treatment group, 6 data points per group

4.1.2 Modifiability

Concerning modifiability, the measurement consisted of an additional implementation of minor, new requirements added to the main task. The effect of each addition was evaluated by counting the lines of code that were added, changed, and deleted for a version and by measuring the time needed to fulfill the new requirements. It must be noted that due to the different performance of the participants only 10 results were included for version 1.1, and only 9 participants could be included for version 1.2 and 1.3.

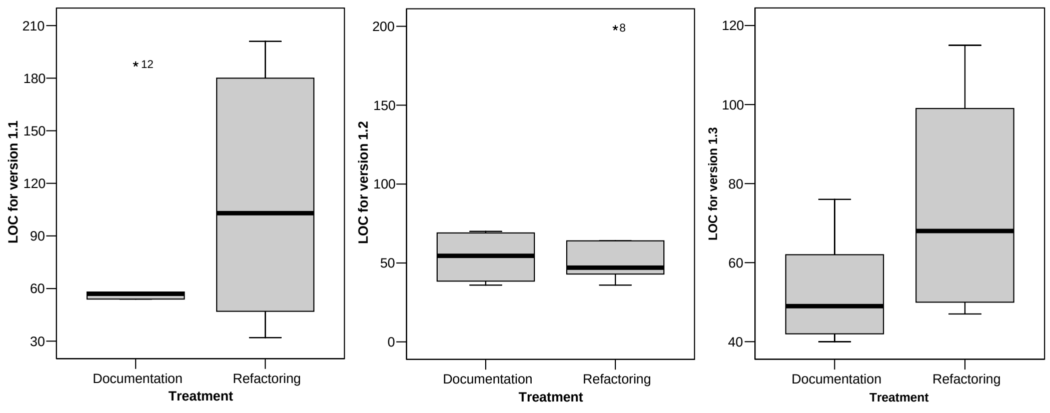


Figure 2: Box plots for changed LOC per version categorized by treatment, 6 data points per treatment

Figure 2 displays the change needed for each development version. Changing incorporates the actions of addition, deletion and modification of a line of source code. Concerning the difference in LOC, it becomes obvious that the refactoring treatment contradicts the initial assumption that refactoring has a benefit on system modifiability. The median of the changed lines for the refactoring group is above that of the control group in two cases.

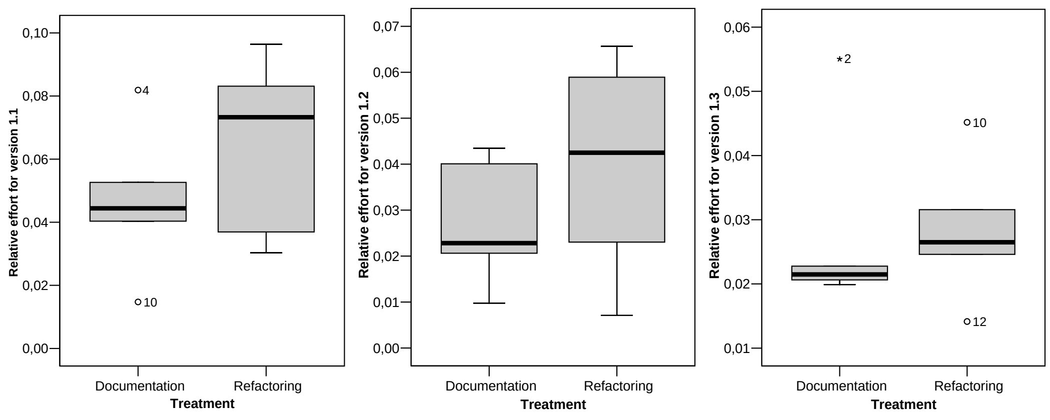


Figure 3: Box plots for fraction of development time compared to first version per modification categorized by treatment, 6 data points per treatment

The observation of figure 2 from above is supported by the time measurement for each treatment group as presented in figure 3. Although these two variables are linked together (more lines of code will take longer to write), the overall impression of additional effort for refactoring is strengthened. In this case, refactoring has a bad effect on all three versions.

Regarding the main hypotheses of better maintainable systems and a better overall modifiability, these results could not show an effect in favor of these non-functional aspects, but rather give a hint on strong side effects of refactoring. An additional interpretation of these main results is given in section 5.

4.2 Analysis of Secondary Variables

Regarding the execution of the experiment, additional variables were measured during the programming procedure. One interesting aspect was the memory usage of a program which was reported by the compiler after each compile cycle. The memory types for the system consisted of SRAM and flash-RAM. SRAM is used for heap memory allocation, as stack memory and for initialized and non-initialized data fields. Flash is mainly used as program memory, meaning that the text of a program is stored here.

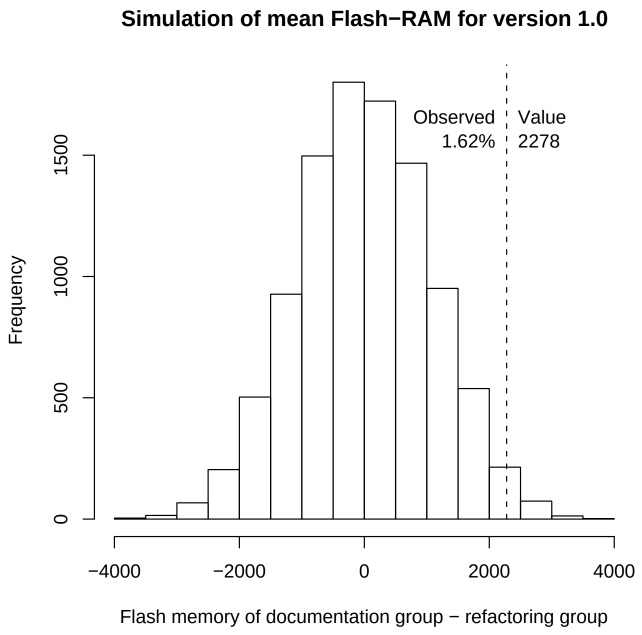


Figure 4: Bootstrap simulation of mean memory difference

When comparing the results of both groups, the difference of the mean flash memory usage for both groups was significantly different. While the documentation group needed less SRAM, the usage of flash was higher as shown by the bootstrap simulation [6] for the mean memory usage of both groups of figure 4. The simulation compares the observed memory difference to differences created by randomized groups. It starts by randomly dividing the observed memory values in two groups of the same size as in the original experiment. From each of these groups, the mean value is calculated and the values are subtracted. This is repeated 10000 times and the results are given in form of the histogram in figure 4. The original value of 2278 is rarely observed (only 1.62%) which can be interpreted as a non-random occuring event. Thus, memory consumption might be effected by refactoring. The implication of this observation and possible causes for this difference are explained in section 5.

An advantage of bootstrap is that for randomization of groups, the values measured during the experiment are taken. Thus, it reuses (bootstraps) its own data to compare the values to a more problem specific population. Compared to t-test and u-test, assumptions concerning the distribution of the data are lower making it more usable for smaller experimental groups [34].

4.3 Analysis of Refactoring Techniques

Another data source originates from the checklists of the participants. Here, each time a student was disturbed, the refactoring techniques applied during the process had to be checked. Based on the frequency of usage, a ranking of the importance for each refactoring technique could be created as shown in figure 5.

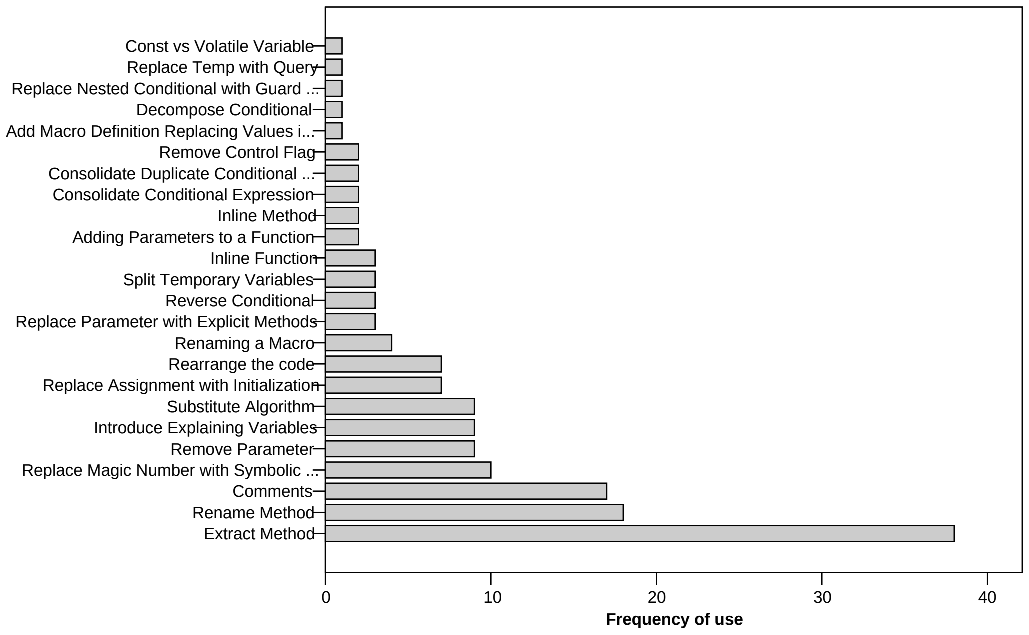


Figure 5: Accumulated occurrences of refactoring techniques for 6 participants

Regarding the techniques used, only some techniques may be of importance for average programming tasks. The extract method principle of aggressively dividing code blocks into smaller chunks appears to be by far the most important technique when refactoring is applied. Additionally, better naming schemes for methods appear to be important which might be understood as a change on the semantic level of the code. The addition of code comments seems to head for the same goal by giving a better explanation for blocks of source code. One single refactoring may have a high ranking only because of the C language programming: replacing a magic number with a symbolic constant. Here again, a better explanation seems to be the aim of the refactoring technique.

This refactoring list may give hints on tool support for refactoring. One problem with this list is that even the most important technique (extract method) will be difficult to implement, as it requires syntactical knowledge of the source code for the according tool in oder to do the refactoring. This is regarded non standard for most source code editors.

4.3.1 Difference in Metrics

In order to describe the structural change that is caused by refactoring, the McCabe metric of cyclomatic complexity is used. Figure 6 shows the cyclomatic complexity plotted against the according lines of code for each function and each treatment group. While the midpoints of both groups do not differ, the number of functions with high cyclomatic complexity appears to be higher for the documentation group. About 11% of the functions created in the control group had a complexity of more than 10, while only 3% of the functions created with active refactoring had a higher value than 10. This may be a hint on the principle of simple design constituting one of the goals of refactoring.

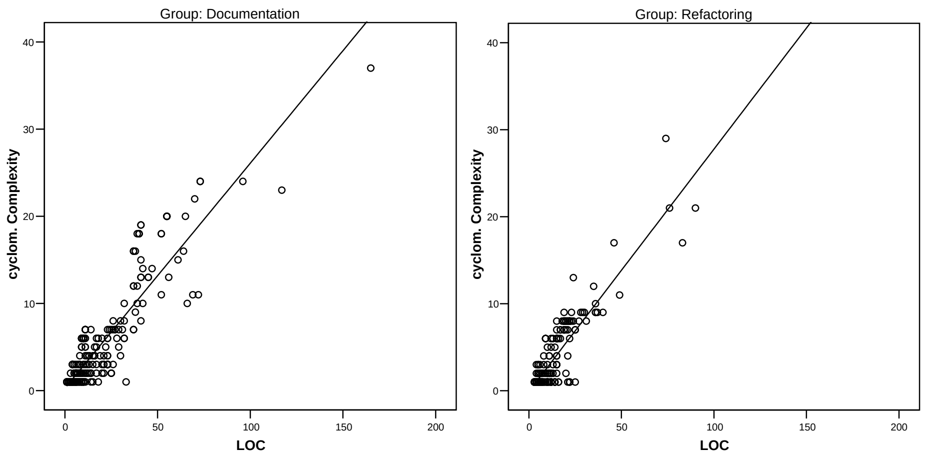


Figure 6: Cylomatic complexity versus LOC scatter plot for functions of both groups

5 Interpretation

The direct effect of an increased maintainability and a better modifiability caused by refactoring could not be shown within this experiment. Although rigid control of the application of refactoring techniques took place, the resulting system did not seem to have a better structure in terms of ability to understand the structure faster for maintainability. Modifiability, which might benefit from the idea of "once and only once", and simplicity, did not seem be of significantly different, either. Instead, the results rather hint to an overhead when refactoring is applied leading to actually more effort when new requirements are added to a system. The question arising from that overhead is if the accumulated time needed for refactoring pays off in bigger systems with more complex architectural aspects. For short projects, the probable benefit of refactoring may reveal itself too late and the resulting overhead may be a waste of time. For long projects, refactoring may have a more positive effect.

Regarding other variables measured during this experiment, the aspect of lower memory usage for program memory is a positive side effect. The basic principle of "once and only once" directly pays off as similar code is reused more often or, in other words, copied code for similar programing tasks is omitted. As this was not part of any hypothesis, this observation has to be regarded carefully.

The main criticism regarding this experiment is its size. The time frame of 40 hours is more than in other experiments, but not sufficient in terms of process assessment. The number of 12 participants is low, too, but as the modifiability results point into the opposite direction, the length of the experiment is regarded more problematic. One other source of criticism might be the use of refactoring without unit tests ([11, 27]). As this can be regarded a major technique to control side effects when a refactoring is executed, it is most often regarded a necessary addition to refactoring. It was omitted, because of the effect this kind of testing might have on software development. Its application would have made an evaluation of refactoring as a single factor more difficult.

One last argument against the experiment is that expert developers would constitute a much better evaluation basis. Their knowledge concerning better system design and areas of "smells" might increase the effect of refactoring. This argument is somewhat misleading, as first of all the effect of refactoring should occur even in the case of average programmers. In addition, using experts in the sense of 1% of available developers appears as an unrealistic modification compared to normal software development.

6 Conclusions and Future Works

6.1 Future Works

Regarding the long term effect of refactoring, a more indirect approach may be beneficial in the future. For example, instead of the execution of refactoring, the effects countered by refactoring might be subject to investigation. The habit "of copy and paste code" may be regarded as development laziness. If the occurrence of this behavior could be shown, the negative effects on the system might be measured leading to an indirect justification for refactoring. Another point closely related to the general application of agile methods is whether the first development solution found is the optimal one in terms of simplicity, understandability, performance, future-applicability and so on. If shortcomings in this area can be shown, refactoring might be the technique to give an increase in these variables. Another starting point for research is the underlying aim of refactoring: what are the reasons to change code, when to change it and, ultimately, do these reasons accumulate for a given code basis? This would need a formalization of the subjective term "smell".

A rather different approach more related to code evolution or metrics as proposed in [17] is done on an additional data source collected during the experiment. It consists of the complete code basis of every participant which was saved every time a compilation was executed. The idea here is to analyze the abstract syntax tree of the code in order to find the cause for a specific refactoring.

6.2 Conclusions

In this paper, a controlled experiment is presented assessing the effect of refactoring on non-functional aspects. However, a general effect of refactoring on maintainability or modifiability could not be shown. Instead, an overhead for the modifiability aspect seems to exists as refactoring itself needs a certain amount of time for its execution. A positive aspect of refactoring might be found in the "once and only once" design principle, as this seems to reduce the memory requirements of a system. As an addition, the three most important refactoring found during this experiment appear to be "extract method", "rename method", and "comments" which might be a starting point for basic refactoring support in software tools. In addition, a different approach to assess the importance of refactoring is presented focusing on indirect assumptions of why refactoring is applied and what problems it might solve.

References

[1] P. Abrahamsson and J. Koskela. Extreme Programming: A Survey of Empirical Data from a Controlled Case Study. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE04)*, 2004.

[2] P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen. New directions on agile methods: a comparative analysis. *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 244–254, 2003.

[3] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett. Understanding and Predicting the Process of Software Maintenance Releases. In *Proceedings of the 18th International Conference on Software Engineering*, 1996.

[4] J.-M. Burkhardt, F. Deétienne, and S. Wiedenbeck. Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Software Engineering*, 7:115–156, 2002.

[5] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences.* Lawrence Erlbaum Associates, 1988.

[6] A. C. Davison. *Bootstrap Methods and their Application.* Cambridge University Press, 1997.

[7] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap.* Chapman & Hall/CRC, 1998.

[8] M. Fowler. *Refactoring. Improving the Design of Existing Code.* Addison Wesley, 1999.

[9] A. Garrido and R. Johnson. Challenges of refactoring c programs. *IWPSE: International Workshop on Principles of Software Evolution*, 2002.

[10] A. Garrido and R. Johnson. Refactoring c with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, 2003.

[11] B. Georgea and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46:337–342, 2004.

[12] B. Geppert, A. Mockus, and F. Rößler. Refactoring for changeability: A way to go? In *11th IEEE International Software Metrics Symposium (METRICS 2005).*

[13] B. Geppert and F. Rosler. Effects of refactoring legacy protocol implementations: A case study. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium on (METRICS'04)*, pages 14–25, Washington, DC, USA, 2004. IEEE Computer Society.

[14] D. M. German. An empirical study of fine-grained software modifications. In *20th IEEE International Conference on Software Maintenance, 2004*, 2004.

[15] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, 2004.

[16] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects – a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5:201–214, 2000.

[17] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance (ICSM02)*, 2002.

[18] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[19] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM Press.

[20] R. Leitch and E. Stroulia. Understanding the economics of refactoring. In *The 7th International Workshop on Economics-Driven Software Engineering Research*, 2005.

[21] H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2003. ACM Press.

[22] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance (ICSM03)*, 2003.

[23] B. McCloskey and E. Brewer. Astec: a new approach to refactoring c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–30, New York, NY, USA, 2005. ACM Press.

[24] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004.

[25] J. Miller. Statistical significance testing – a panacea for software technology experiments? *The Journal of Systems and Software*, 73:183–192, 2004.

[26] J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks. Statistical power and its subcomponents – missing and misunderstood concepts in empirical software engineering research. *Journal of Information and Software Technology*, 1997.

[27] M. Müller and O. Hagner. Experiment about test-first programming. *IEE Proceedings Software*, 149(5):131–136, October 2002.

[28] M. Pizka. Straightening spaghetti code with refactoring? *Software Engineering Research and Practice*, 2004.

[29] L. Prechelt. *Kontrollierte Experimente in der Softwaretechnik*. Springer, 2001.

[30] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, New York, NY, USA, 2005. ACM Press.

[31] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, 2001.

[32] J. L. Simon. *Resampling: The New Statistics*. Resampling Stats, 1999.

[33] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč. Challenges and recommendations when increasing the realism of controlled software engineering experiments. *ESERNET 2001–2003, LNCS 2765*, pages 24–38, 2003.

[34] J. B. Todman and P. Dugard. *Single-Case and Small-N Experimental Designs: A Practical Guide to Randomization Tests*. Lawrence Erlbaum Associates, 2000.

[35] B. Walter and B. Pietrzak. Multi-criteria detection of bad smells in code with uta method. In H. Baumeister, M. Marchesi, and M. Holcombe, editors, *Extreme Programming and Agile Processes in Software Engineering, XP 2005*, Sheffield, UK, 2005. Springer.

[36] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering. An Introduction.* Kluwer Academic Publishers, 2000.

e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

Programming in the eXtreme: Critical Characteristics of Agile Implementations

Gerald DeHondt II*, Alan Brandyberry*

*Management & Information Systems Department, Kent State University

gdehondt@kent.edu, abrandyb@kent.edu

Abstract

The prevalence of systems development project failures has been well documented. eXtreme Programming (XP) is a software development methodology that seeks to eliminate many of the shortcomings of cumbersome life cycle oriented traditional methodologies. We explore some of the basic tenets of XP and Agile methodologies and present the thoughts of two of the proponents and early participants in the "Agile revolution", Chet Hendrickson and Ron Jeffries. We analyze this interview utilizing an interpretive field study employing a hermeneutical circle technique. Our analysis suggests some of the characteristics of XP implementations are more critical than others. We propose a more concrete definition of what XP represents and suggest areas for future research.

1 Introduction

Modern software development efforts often follow traditional software development methodologies such as the Systems Development Life Cycle (SDLC), with organizations typically making certain adjustments to bring these traditional methodologies in line with their organizational needs. While utilizing this method is acceptable, it often leads to cost overruns and longer development cycles than originally anticipated, and user requirements that remain unmet. These types of problems can wreak havoc with corporations' IT strategy and planning functions, as well as have significant negative impact on the business. Recently, certain "Agile" development techniques have been introduced that seek to provide shorter development cycles, with the associated cost savings.

Specifically, eXtreme Programming (XP) is an Agile development technique which emphasizes rapid and frequent feedback to the customers and end users, unit testing, and continuous code reviews. By focusing on rapid iterations of simpler code, XP seeks to identify and resolve potential pitfalls in the development process early, leading to projects that remain focused on the ultimate goal – timely delivery of a well-designed and tested system that meets customer requirements.

XP breaks down a project into sub-projects, each including planning, development, integration, testing and delivery [21]. Developers work in small teams with customers as active team members. Features are implemented iteratively during each development cycle with joint decision making occurring between the customer and the rest of the development team. Agile software-development methods use human- and communication-oriented rules in conjunction with light, but sufficient, rules of project procedures and behavior [8]. They rely on planning, with the understanding that everything is uncertain, to guide the rapid development of flexible systems of high value [20, 21]. eXtreme Programming has been described in terms of the values that support it: communication, feedback, simplicity, courage, and respect [4]. This methodology works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and tune the practices to their unique situation.

2 Extreme Programming Begins

The eXtreme Programming Methodology was developed and first implemented by Kent Beck at DaimlerChrysler in 1996 as a way to accelerate development efforts, while producing better software for their Chrysler Comprehensive Compensation (C3) Project. The C3 Project had run into an impasse. Previous efforts to develop a new payroll system – one that would support over 86,000 international employees from union represented employees to upper management – had failed, and in early 1996 it was decided to start over from scratch [46]. As background, the new system was to replace three separate legacy systems that had been in place for over twenty years supporting the various payroll groups. Originally, Chrysler had purchased a payroll system from a leading industry vendor. After many months of effort, it was determined that neither this package, nor any on the market, could handle the complexities of the current payroll structure. At this point, it was determined that the system would have to be designed and built from scratch. The previous implementation attempt had used a traditional software development methodology, the "Waterfall" method. The team had found this method too complex and cumbersome and realized they needed to approach the problem in a different way. This is consistent with the previous research of Merisalo-Rantanen, Tuunanen, and Rossi [35] as one of the cases they studied suggested that traditional methodologies were too restrictive and slow, and hence evolved their corporate system development model into an XP-like framework. Additionally, Levina and Ross [28] suggest that having stringent development methodologies might prove detrimental to the success of more creative and innovative application development projects.

At this point, the team decided to apply the practices of eXtreme Programming to develop the C3 System. Over about the course of a year, they were able to architect, design, code, test and implement a system that supported the Chrysler Comprehensive Compensation structure. This is quite a feat when considering that the previous iteration of the project languished for a number of years before being deemed a failure and turning to XP. Fowler [14] argues that these lightweight, or Agile, approaches are necessary due to the high overhead and resource intensiveness of the existing and dominant methods of software development.

There is ample room for improvements to be made in the prevailing software development methodologies. It is well documented that systems development failures are all too common. Although different definitions of what constitutes a failure undoubtedly yield different rates, estimates of failure rates on systems development projects have been purported to range from 50 percent to 75 percent [12, 16].

eXtreme Programming, however, is not without its detractors. As a significant departure from traditional development methods, many corporations are hesitant to risk development projects to a relatively new development technique, especially one that significantly departs from conventional methods. Adoption of an Agile Methodology will also likely pose several challenges for organizations steeped in the traditional systems development methodologies, since the two software development methods are grounded in opposing concepts [37].

3 Background

An Information Systems Development Methodology has been defined as an organized collection of concepts, methods, beliefs, values and normative principles supported by material resources [22]. Initial efforts at formalizing and planning systems development efforts began with "traditional" methods such as the "Waterfall" [6, 23], which then evolved into the Systems Development Life Cycle – a commonly used systems development methodology. However, these traditional methods take a phased approach to systems development, requiring that one phase be completed prior to beginning the next phase [24] and the product is not delivered until the whole linear sequence has been completed. This effectively means that the first day that functionality can be delivered to the customer is the last day of the project (except for the ongoing system maintenance).

3.1 Satisfying Customer Requirements

Extensive upfront planning is the basis for predicting, measuring, and controlling problems and variations during the development lifecycle. The traditional software development approach is process-centric, guided by the belief that sources of variations are identifiable and may be eliminated by continually measuring and refining processes [9]. The primary focus is on realizing highly optimized and repeatable processes.

One measure of an organization's development process maturity is the Capability Maturity Model (CMM). The Capability Maturity Model is a project of the Software Engineering Institute (SEI) at Carnegie Mellon University seeking to identify best practices that may be useful in helping organizations increase the maturity of their software development processes [43]. Organizations will achieve a designation from 1 through 5, based on the repeatability, definition, management, and optimization of their software development processes. This process itself though, is very cumbersome. Even stripped to the bare essentials, the CMM comprises 52 primary goals and 18 key process areas [38]. As one official in the CMM project at Carnegie Mellon University noted, "You can be an [highest CMMrated] organization that produces software that might be garbage." [31] Focusing on the process used to develop and deliver software may not always lead to systems that meet customer requirements.

One of the primary drivers of XP is the focus on delivering the features the customer wants. Under traditional software development methods, planning and control accomplished by a command and control style of management provide the impetus for developing a software product [21]. Traditional methodologies assume that problems are fully specifiable, and that an optimal and predictable solution exists for every problem [7]. In the instance that there is a change in design or user requirements over the course of the project – a situation all too common in real-world corporate development environments – the methodology begins to break down and efforts become focused on reworking previous design and development activities. These changing requirements cause additional rework in the project. Gopal, Mukhopadhyay, and Krishnan [17] found that clients who request rework later in the lifecycle increase rework – and cause project delays – considerably, with requirements volatility having an even stronger influence on rework. A methodology not appropriately suited to changing requirements causes delays in system delivery.

Traditional methodologies are also too set and too full of inertia to be able to respond quickly enough to a changing environment to be viable in all cases [12]. Existing heavy-weight processes tend to be predictive and slow [45] and are unable to efficiently react to changing circumstances. It is often suggested that these traditional methodologies are useful in situations where requirements are well-known and unlikely to change but that assertion is also challenged by some in the XP community. They can also be used when significant project management overhead must be incurred, as in large, business-critical systems built with teams in excess of 50 members [14]. Lightweight or Agile methods, on the other hand, are seen to be more code-oriented, more people-oriented, and more adaptable to change suited to relatively small projects with rapidly changing requirements [19]. Others, however, find significant scalability in XP through project decomposition [4].

These evolving requirements are often viewed as an inherent problem of software development in traditional methodologies [47]. In a sense, taking one step forward and two steps back. On the other hand, the Agile community views requirements changes as an opportunity to provide software that can enhance the customer's competitiveness in a rapidly evolving marketplace [47]. Development teams that can handle these changes will produce software that is more useful to the customer, leading to more satisfied customers. XP seeks to support timely and economical development of high-quality software that meets user requirements at the time of delivery. XP utilizes an iterative approach that is helpful in developing, modifying, and maintaining systems more quickly and more successfully [2, 5]. It is these short iterations that provide the flexibility to accommodate the changes requested by the customers and allows the customer to increase competitiveness in the market [47]. In fact, XP seeks to implement the simplest design that will satisfy current user requirements [29] without attempting to anticipate future design or user requirements. The iterative nature of this methodology enables it to be tolerant of changes in requirements [4].

3.2 Organizational Influence

As the traditional SDLC has become embedded and institutionalized in organizations as the standardized method of systems development, any changes to a new approach will require a shift in the organizational norms. This can be one of the most difficult obstacles to implementing XP. Based on the team approach of XP [29], those with individual influence under the traditional methodologies will have to acquiesce for the greater good of the team. Organizations using heavier methodologies typically had trouble adopting the incremental release approach of Agile Methodologies because of the implications of the core practices: simple design, testing, refactoring, and continuous integration [12]. These core practices require that everything be available. For example, a daily build means that the testing suite must also be ready daily, which also has implications for continuous integration and refactoring. Specifically, Project Managers have been viewed as the overall leader, with significant authority to make decisions impacting the project. In an XP environment, the project managers give up much of their decision-making authority to the team [35]. This "Method Adaptation" [34] is the process or capability of agents, through responsive changes in and dynamic interplays between contests, intensions, and method fragments, to determine a system development approach for a specific project situation.

It is generally accepted that there is no single process that will be applicable to all projects [47]. There are a number of best practices, techniques, and experiences that developers can use in appropriate situations. Software development teams with leaders that understand the situations in which particular processes are applicable are more likely to be successful within an Agile environment. Remember, Agile processes are not silver bullets. Because these assumptions are not met in all development environments, it is possible to extend Agile processes to address their limitations. These extensions can involve incorporating principles and practices associated with traditional development processes into Agile processes. Users of Agile processes need to ensure that practices based on invalid, environmental assumptions are modified accordingly.

3.3 Values and Practices of XP

XP, at its core, rests upon the following values: communication and feedback, simplicity, and responsibility [47]. Project success, and delivering software that meets the customer requirements, requires frequent, face-to-face communication between developers and customers. Not only is this explicit communication, but also delivering working code incrementally at frequent intervals. Ultimately, this is the best check to demonstrating understanding of customer requirements. Simplicity is embodied by delivering only solutions that meet current user needs. XP does not attempt to design or anticipate for future needs. Ultimately, the customer is concerned with addressing current needs, and is not interested in what may occur later. Finally, the responsibility of delivering high-quality code rests with the developers of the system. Agile methods focus on people as the primary drivers of development success [10]. They are those closest to the solution and should be the most knowledgeable about how the solution will be implemented.

These values are expanded into the twelve practices of the XP Methodology: planning, small releases, metaphors, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, on-site customers, coding standards, and 40-hour week [3]. Similar values are embodied in the Agile Manifesto [11]:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These values implement the best practices of previous Systems Development Methodologies representing an evolutionary process growing out of a natural and useful way to develop software [35]. They provide a number of instances in their study where XP-like methods simply evolved as logical ways of doing things. This evidence provided support for their claim that XP is more of a packaging of best practices, rather than a totally new way of doing things, formalizing habits that work in certain organizational settings for delivering better software.

This evidence remains consistent with the core XP beliefs, especially given the flexibility that the methodology provides in adapting the values to a given situation. These beliefs include user involvement, simple code, iterative development, and courage. McKeen, Guimaraes, and Wetherbe [33] argue that user participation improves the quality of the system in several ways such as providing a more accurate and complete assessment of requirements, expertise about the user organization the system will support, avoiding development of unacceptable or unimportant features, and improving user understanding of the system as it is being developed. XP engages the end users in the IS development process.

Kent Beck [4] states that the programming strategy of XP is to keep the code easy to modify. Iterative development requires that each developer must integrate and release code at least once a day after passing all the unit tests or completing a smaller part of planned functionality [25]. This continuous integration detects compatibility problems early and ensures everyone is working with the latest version of the system.

Along with the iterative nature, XP incorporates rigor into the methodology. XP stipulates that developers follow all of the practices to realize the benefits of Agile Development [47]. McBreen [32] points out that XP requires enormous discipline to implement, and in fact, some projects may find it difficult to adopt a true XP-compliant process.

Another of the main beliefs of eXtreme Programming is courage – for developers to rework their code when it doesn't perform as expected, to stick to the chosen approach when the going gets tough, and to make the hard decisions required to deliver the software [1]. In order for XP to work, corporations – or at least their IT departments – must be willing to take the necessary risks to depart from conventional techniques, and have the courage to continue along this path during the difficult times. IT departments must be able to adapt and implement new techniques when the situation is called for, as opposed to holding on to old methods that produce the same inadequate results. Organizations must be willing to change and implement new techniques to develop better software.

Exhibiting the fortitude required to undertake some of these efforts will allow additional flexibility – agility – in the development of the system that in the end will provide a better system, more flexible to changing user requirements, and implemented in a fashion that meets the goals of the organization.

Previous research by Merisalo-Rantanen, Tuunanen, and Rossi [35] provide anecdotal evidence that both developers and internal and external customers were satisfied with the results of their projects developed using eXtreme Programming. A number of other studies and experience reports also provide claims for the successful use of eXtreme Programming and Agile Methods [18, 30, 39, 42, 44, 46] in practice. Additionally, experimental studies [15, 36] have focused on particular techniques used within eXtreme Programming such as pair programming and test-driven Development (TDD). With the relative recency of eXtreme Programming as an approach to software development, results of a survey by Rumpe and Schroder [40] show that this methodology is still in the "hype phase". However, numerous opinion pieces and several surveys clearly demonstrate the growing popularity of Agile Methodologies [37].

3.4 A Broad-based Perspective of XP Implementations

The literature on eXtreme Programming is primarily focused on review of the practices and techniques, and how this methodology differs from traditional methodologies. In this context, there have been a few examples of anecdotal studies of individual projects and experience reports focusing on a small number of projects. This study reviews a multitude of projects from the practical experiences of two of the initial implementers of these practices. The purpose of this work is to gain insight across a number of projects from these practitioners based on their implementation experiences. It is from this broad-based perspective that we can learn the appropriate positioning of eXtreme Programming within an organization's systems development toolbox.

This interview focuses on the types of organizations that may successfully implement XP techniques, including some methods to integrate XP into an organization focused on rigid, top-down development methodologies, further explanation of how XP differs from traditional methodologies, and provides a means of identifying a true XP Project, including ways to determine whether the practices are really being used.

As is commonly seen when innovations are introduced, there is some confusion as to what is and is not XP. In casual conversations with people generally knowledgeable in the area of systems development the perception that XP is using "no methodology" at all is sometimes expressed. This confusion is probably due predominantly to the emphasis on values [4] that, on the surface, appear to lack the structure that other methodologies emphasize. What is misunderstood is that these values must be manifested in principles and then actual practices. This allows substantial flexibility in how an organization actually implements XP. Although this is viewed as a strength by XP proponents, it also contributes to the confusion around XP. There exists no objective test as to whether an organization is employing XP. As the interview subjects suggest, there are cases where XP has been identified as the development methodology that others have questioned. Obviously an organization has not truly implemented XP just by uttering the words "eXtreme Programming". In the interview in the following section we engage these experienced practitioners in a general discussion of XP to attempt to discover the themes that they express as being critical to XP implementation. We then employ a hermeneutic circle technique to extract the critical characteristics of XP implementations.

4 Methodology

This analysis is an interpretive field study as used by Fitzgerald et al [13] and Sauer and Lau [41] when studying the practical use of a method. This has been suggested as an appropriate research method for explorative and descriptive types of research by Klein and Myers [26]. It examines the experiences of two well-known eXtreme Programming practitioners, Chet Hendrickson and Ron Jeffries. They were original team members of the seminal XP Project at Chrysler Corporation in 1996, and have since provided consulting and training services to Fortune 500 companies in the appropriate use and implementation of the eXtreme Programming methodology. They have also spoken at numerous academic and practitioner conferences expounding their knowledge of eXtreme Programming. The question and answer discussion presented in the next section was derived from a series of interviews and follow-up contacts. Gerald DeHondt facilitated these interactions.

4.1 Interview

DeHondt: What kind of organizational model will be most appropriate to implementation of eXtreme Programming techniques?

Jeffries: No project is completed with traditional software development methodologies; it is completed in spite of traditional software development methodologies. It is a combination of the process and the team that leads to successful projects. If it is the wrong combination, the organization will try to kill it.

DeHondt: In these instances, how would you get the organization to change?

Jeffries: A politically safe method is needed for implementing XP in organizations.

This can be accomplished by increasing user feedback to the development staff. In this, we focus on 8–10 stories per month. At the end of the month we determine how many we were able to successfully complete. From there we can adjust the timeline either up or down.

There also needs to be accountability. One of the things agile methods do is provide an API into the organization to produce clear unambiguous information.

DeHondt: Will XP work in all environments? For example, many organizations are focused on top-down, structured, rigid development methodologies. In these instances, how would XP be adapted?

Jeffries: XP has been used in a number of rigorous environments. NASA has developed projects using agile methodologies, McKeen implemented a SmallTalk project, and it has been used by an embedded medical equipment manufacturer for a product requiring FDA approval.

One of the key points is whether the project is utilizing Test Driven Development (TDD) and integrating customer acceptance testing into their development process.

We also want to make sure we are appropriately completing the tasks. For example, if we are 90 percent complete on all of our tasks, we haven't completed any. On the other hand, it would have been better to spend the time to complete 90 percent of the tasks completely.

DeHondt: How does XP differ from traditional development?

Jeffries: What we do is break up the project into smaller pieces. This way we can more effectively monitor whether we are on task with the smaller piece. The error bars will be smaller in this situation.

Consider driving a car. As we drive down the road, we are continuously making adjustments to our direction to stay in our lane. If we made corrections at longer intervals we would risk going off the road before changing our direction. This is the same thing with software development. We are continuously monitoring the project progress by getting feedback from the customer. If we are going a little off course, customer feedback will allow us to change direction before we go off the road. We are able to respond to the environment faster. We are able to learn faster.

DeHondt: There have been some cases where XP methods have not worked, and the projects have failed. Are there any common reasons why an XP project would fail?

Jeffries: In the instances where whatever you did didn't work, maybe these projects weren't using XP, only saying they were. If a project says they are using XP, but not any of the practices, then it is not XP.

In order for projects to be using XP, they need to be able to ship code at any time. In the Chrysler project, at any time, we could provide the customer with the latest build for installation and testing. This is XP.

You also want to get negative work as close to zero as possible. We don't want to have rework when customer requirements aren't properly implemented. By having the customer work closely with us, we are able to monitor and ensure that their requirements are being implemented as expected.

Hendrickson: With project status reporting, we are able to more closely monitor work completed and provide this information to the customer. Do you want to hear the truth about the project, or do you want to hear what you want?

When companies look at project completion measures, there are a number of measures that can be used to gauge project completion. These could include percent of time spent, percent of money expended, or other criteria. XP will look at percent of deliverables satisfied and are we are building what the customer asked for.

We want to minimize the process overhead not related to writing code.

Jeffries: The project needs to have a good communication feedback loop with the customer, be able to test quickly – this may even include an automated test cycle – simple design, code to support the test cases, small chunks of code, and integrated code.

One of the problems encountered is that teams don't know how to ship code regularly. They should be able to provide the latest build of the system at any time.

Hendrickson: Using traditional software development methodologies, a company may spend three months on the analysis phase and three months on the design phase. At this point, the project has been silent for six months. At that point the determination is made if they are on track.

Jeffries: In XP, the architecture and design grow along the way. XP seeks to ensure that the software is working as intended along the way, and the design is good. This will allow the software to change if the customer requirements change.

DeHondt: Now let's shift gears a little to team attributes. The literature has looked at XP and proposed that it is successful because it is staffed with highly qualified people. In this type of an environment, it may be the people that cause projects to succeed, not necessarily the process. What type of attributes does a team require to be successful at an XP project?

Jeffries: A good XP team will have people who are thoughtful about what they did, with good coaching, and determination. The customer has to know what they want in the software. There also needs to be support from the highest levels of the organization. Ultimately, they need to be good people, who are good programmers.

XP succeeds because of commitment from the company, team, management, and the customer. The team needs to embrace and execute the practices of XP, monitor and adjust. There needs to be codification of best practices and feature by feature analysis. The development process needs to be molded to the situation with the agile development bias towards simplicity and action.

Companies that deliver software on time and on budget do not outsource.

Hendrickson: Agile focuses on people and delivering code to implement features. It breaks down a project into smaller iterations focused on delivering working functionality or requirements. It places emphasis on communication with the customer to determine if the code developed meets the customer's requirements.

Agile is also good at handling evolving requirements, whereas the traditional software development methodologies specify everything up front.

Jeffries: XP also requires everything to be available – the development environment, test environment, everything that will be needed to quickly move through the process and develop working code that meets customer requirements.

Hendrickson: Some XP projects will fail; it is just that these projects will fail sooner with XP. In traditional approaches, it may take longer to realize that the project is not viable. At that point, more time and resources have been expended.

5 Analysis and Discussion

We employ a meaning categorization and hermeneutical interpretation form of analysis as described by Kvale [27]. In the review of the literature in the first section of this research we identified the values, principles, and practices that have been commonly associated with XP implementations. We begin our analysis by viewing each of these as a general theme and shared symbolic vocabulary. We then iteratively explore the interview for the apparent level of importance of each theme by determining the frequency and context in which each theme appeared. In the tradition of hermeneutical circles we iteratively interpreted the meaning of the whole and the meaning of the parts or themes with the goal of deepening our understanding of the meaning within (a *circulus fructuosis*) [27]. We will discuss these identified dominant themes in the derived order of importance (based on number of occurrences and implied criticality) to the overall meaning.

5.1 Continuous Delivery of Working Software

This theme was repeated often and in different ways during the interview. They emphasize completing tasks rather than having many tasks partially completed. They state that projects using XP "need to be able to ship code at any time". Another related concept introduced is measuring percent of deliverables satisfied as the primary metric of project completion. A clear sentiment on this issue is their articulation that "teams don't know how to ship code regularly. They should be able to provide the latest build of the system at any time". Similarly, they state, "Agile focuses on people and delivering code to implement features. It breaks down a project into smaller iterations focused on delivering working functionality or requirements”. There is a clear message throughout the discussion that delivering working software continuously is critical to utilizing XP.

5.2 Customer-driven Process

The concept that the customer must be involved and must, in fact, drive the XP process is expressed several times during the interaction. This is exemplified by statements such as “The customer has to know what they want in the software” and “XP succeeds because of commitment from the company, team, management, and the customer”.

5.3 Communication

“Do you want to hear the truth about the project, or do you want to hear what you want?” This statement underscores the importance of good communication in XP implementations. Other statements such as the “project needs to have a good communication feedback loop with the customer” and their reference to “increasing user feedback to the development staff” also demonstrate the importance of this concept.

5.4 Incremental Design and Acceptance of Changing Requirements

These are two related themes that are referenced by the subjects in several instances. We view them as related since the reason incremental design is important is that requirements do often change. Their analogy to driving a car (needing constant adjustment) is related to this theme. They also refer to breaking up projects into smaller pieces, implying separate design cycles for these functions. This theme may be well summated in the statement, “Agile is also good at handling evolving requirements, whereas the traditional software development methodologies specify everything up front”.

5.5 Continuous Testing

The subjects specify that “One of the key points is whether the project is utilizing Test Driven Development (TDD) and integrating customer acceptance testing into their development process”. They also make reference to being able to test quickly as well as utilizing automated testing.

5.6 Other themes

There were several other themes that were identified by this process. Both the number of occurrences of the theme and the implied criticality lead us to believe that these are less critical (though not necessarily unimportant) themes when trying to conceptualize those few important characteristics that would generally be considered essential. These themes include: the incremental implementation of XP, use of “stories”, XP is lightweight, team-talent, all-constituents need to sign on, and that XP may fail but fail sooner.

5.7 Hermeneutical Interpretation of “the whole”

Each of the identified themes appears to have its own individual importance and, at least to these practitioners, certain themes seem to be more critical to the process than others. Additionally, their interaction with each other and less critical XP themes is also significant. For instance, continuous testing is critical if you are going to be able to ship the latest build of a software project at any time. An attempt to synthesize these critical themes into a single defining statement yields the following result:

XP is fundamentally the continuous delivery of incremental working software that is customer-driven and dependent on communication, incremental design, acceptance of changing requirements, and continuous testing.

Although this definition may be incomplete in some ways since XP is possibly more philosophy than methodology, it may serve as a useful starting point in bridging the gap of understanding in what is meant by the term eXtreme Programming. Certainly, it adds a way of expressing the XP concept in the more concrete terminologies that many methodologists prefer.

6 Conclusion

The views of these participants in the Agile revolution hold that eXtreme Programming has substantial benefits when applied appropriately and they report many experiences that support this view. There also needs to be consideration given to whether the project actually implemented XP processes and practices. Investigations of failed “XP” projects indicate that these projects were not, in fact, implementing the values and principles, but simply stating that they were an XP project.

As a relatively new methodology there is need for substantial research into the concept and its implementation. We have attempted to focus the discussion of “what is XP?” on several critical themes. However, this work has been based on the perspective of two individuals. No matter how experienced and well-versed they may be, studies that synthesize a broader array of viewpoints will also be important. Critical questions include: Are there project or environmental characteristics that make other XP themes more (or less) critical? Are there certain “brands” of XP where common sets of themes are utilized together? Are failures due to incomplete implementation of XP concepts, project characteristics independent of methodology, or other factors? The authors suggest performing follow-up work on comparable paired projects – based on application requirements, project size, or project type – that use eXtreme Programming and traditional methodologies as their development approach. This could allow comparability of the methods to similar situations and help determine factors in each methodology or particular project interaction that highlight strengths and weaknesses of each approach.

Many XP proponents seem to imply that XP would be the best choice for all development projects. Proponents of other methodologies seem more willing to say that there are certain project types that a particular methodology is better suited for than others.

An important question that needs to be answered is whether there are project characteristics that make XP more or less likely to succeed. Are there characteristics that would make a particular project unsuitable for implementation using XP?

Some people believe XP succeeds because of the above-average skill level of the team. This study reveals placing the overall attitude of the team as a more salient feature to project success than the aptitude of the team.

References

[1] S. Ambler. The Extreme Programming Software Process Explained. *Computing Canada*, 26:24, March 2000.

[2] V. Basili and A. Turner. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering*, 1(4):390–396, December 1975.

[3] K. Beck. *Extreme Programming Explained.* Addison-Wesley, Boston, 2000.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change 2nd Ed.* Addison-Wesley, Boston, 2004.

[5] B. Boehm. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):22–42, 1988.

[6] F. Brooks. *The Mythical Man Month: Essays on Software Engineering.* Addison-Wesley, Reading, MA, 2003.

[7] S. Cavaleri and K. Obloj. *Management Systems: A Global Perspective.* Wadsworth Publishing Copany, California, 1993.

[8] A. Cockburn. *Agile Software Development.* Addison-Wesley, Boston, 2002.

[9] A. Cockburn and J. Highsmith. Agile Software Development: The Business of Innovation. *IEEE Computer*, pages 120–122, September 2001.

[10] B. Conrad. Taking Programming to the Extreme. *Infoworld*, page 24, July 2001. July 21, 2000. available online at http://www.infoworld.com/articles/mt/xml/00/07/24/000724mtextreme.html.

[11] W. Cunningham. Manifesto for Agile Software Development. *[WWW document]. URL http://www.agilemanifesto.org/principles.html*, n.d./2006.

[12] J. Erickson, K. Lyytinen, and K. Siau. Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research. *Journal of Database Management*, 16:88–100, October–December 2005.

[13] B. Fitzgerald, N. Russo, and T. OKane. An Empirical Study of System Development Method Tailoring in Practice. *Proceedings of the Eighth International Conference on Information Systems*, pages 187–194, 2000.

[14] M. Fowler. The New Methodology. *[WWW document]. URL http://www.martinfowler.com/articles/newMethodology.html*, December 2005.

[15] B. George and L. Williams. A Structured Experiment of Test-Driven Development. *Information and Software Technology*, 46(5):337–342, April 2004.

[16] E. Germain and P. Robillard. Engineering-Based Processes and Agile Methodologies for Software Development: A Comparative Case Study. *Journal of Systems and Software*, 75:17–27, 2005.

[17] A. Gopal, T. Mukhopadhyay, and M. Krishnan. The Role of Software Process and Communication in OffShore Software Development. *Communications of the ACM*, 45(4):193–200, March 2002.

[18] J. Grenning. Launching Extreme Programming at a Process Intensive Company. *IEEE Software*, 18(6):27–33, November–December 2001.

[19] B. Henderson-Sellers and M. K. Serour. Creating a Dual-Agility Method: The Value of Method Engineering. *Journal of Database Management*, 16(4):1–23, October–December 2005.

[20] J. Highsmith. *Agile Software Development Ecosystems.* Addison-Wesley, Boston, MA, 2002.

[21] J. Highsmith. *Cutter Consortium Reports: Agile Project Management: Principles and Tools*, volume 4(2). Cutter Consortium, Arlington, MA, 2003.

[22] R. Hirschheim, H. Klein, and K. Lyytinen. *Information Systems Development and Data Modeling.* Cambridge University Press, New York, 1995.

[23] R. Hirschheim, H. Klein, and K. Lyytinen. *Information Systems Development and Data Modeling: Conceptual and Philosophical Foundations.* Cambridge University Press, Boston, MA, 2003.

[24] J. Hoffer, J. George, and J. Valacich. *Modern Systems Analysis and Design.* Addison-Wesley, Boston, MA, 1998.

[25] S. Joosten and S. Purao. A Rigorous Approach for Mapping Workflows to Object-Oriented IS Models. *Journal of Database Management*, 13(4):1–19, October–December 2002.

[26] H. Klein and M. Myers. A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems. *MIS Quarterly*, 23(1):67–93, March 1999.

[27] S. Kvale. *Interviews: An Introduction to Qualitative Research Interviewing.* Sage Publications, Thousand Oaks, 1996.

[28] N. Levina and J. Ross. IT From the Vendors Perspective: Exploring the Value Proposition in Information Technology Outsourcing. *MIS Quarterly*, 27(3):331–364, September 2003.

[29] L. Lindstrom and R. Jeffries. Extreme Programming and Agile Development Methodologies. *Information Systems Management*, 21(3):41–52, 2004.

[30] P. Manhart and K. Schneider. Breaking the Ice for Agile Development of Embedded Software: An Industry Experience Report. *Proceedings of the 26th International Conference on Software Engineering*, pages 378–386, 2004.

[31] N. Matloff. Globalization and the American IT Worker. *Communications of the ACM*, 47(1):27–29, November 2004.

[32] P. McBreen. *Questioning Extreme Programming.* Addison-Wesley, Thousand Oaks, 2003.

[33] J. McKeen, T. Guimaraes, and J. Wetherbe. The Relationship Between User Participation and User Satisfaction: An Investigation of Four Contingency Factors. *MIS Quarterly*, 18(4):427–451, 1994.

[34] A. Mehmet, F. Harmsen, K. van Slooten, and R. Stegwee. On the Adaptation of an Agile Information Systems Development Method. *Journal of Database Management*, 16(4):24–40, October–December 2000.

[35] H. Merisalo-Rantanen, T. Tuunanen, and M. Rossi. Is XP Just Old Wine in New Bottles. *Journal of Database Management*, 16(4):41–61, October–December 2000.

[36] M. Mller. Preliminary Study on the Impact of a Pair Design Phase on Pair Programming and Solo Programming. *Information and Software Technology*, 48(5):335–344, May 2006.

[37] S. Nerur, R. Mahapatra, and G. Mangalaraj. Challenges of Migrating to Agile Methodologies. *Communications of the ACM*, 48(5):73–78, May 2005.

[38] M. Paulk. Extreme Programming from a CMM Perspective. *IEEE Software*, 18(6):19–26, November–December 2001.

[39] C. Poole and J. W. Huisman. Using Extreme Programming in a Maintenance Environment. *IEEE Software*, 18(6):42–50, November–December 2001.

[40] B. Rumpe and A. Schroder. Quantitative Survey on Extreme Programming Projects. *Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002), Alghero, Italy, May 26-30*, pages 95–100, 2002.

[41] C. Sauer and C. Lau. Trying to Adopt Systems Development Methodologies: A Case Based Exploration of Business Users Interests. *Information Systems Journal*, 7(4):255–275, October 1997.

[42] P. Schuh. Recovery, Redemption, and Extreme Programming. *IEEE Software*, 18(6):33–40, November–December 2001.

[43] Software Engineering Institute. Capability Maturity Model for Software. *[WWW document]. URL http://www.sei.cmu.edu/cmm/*, January 2006.

[44] W. Strigel. Reports from the Field: Using Extreme Programming and Other Experiences. *IEEE Software*, 18(6):17–18, November–December 2001.

[45] T. Chau and F. Maurer and G. Melnik. Knowledge sharing: Agile methods vs. tayloristic methods. *Proceedings of the 12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 03)*, pages 302–307, 2003.

[46] C. Team. Chrysler Goes to "Extremes". *Distributed Computing*, pages 24–28, October 1998.

[47] D. Turk, R. France, and B. Rumpe. Assumptions Underlying Agile Software-Development Processes. *Journal of Database Management*, 16(4):62–87, October–December 2005.

e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

A User-Centered Approach to Modeling BPEL Business Processes Using SUCD Use Cases

Mohamed El-Attar*, James Miller*

*STEAM Laboratory – Electrical and Computer Engineering Deparment, University of Alberta

melattar@ece.ualberta.ca, jm@ece.ualberta.ca

Abstract

BPEL is being widely used to specify business processes through the orchestration, composition and coordination of web services. It is now common practice to begin the process of modeling the "workflows" within a set of BPEL business processes using UML Activity Diagrams since they can be automatically mapped down onto BPEL code. However activity diagrams were not intended to explicitly model user goals and interactions with external systems offering web services. However, since the chief purpose of BPEL business processes is to first and foremost provide services to their users, using activity diagram modeling alone will not allow an E-commerce analyst to explicitly capture and model the users' goals. In this paper we propose an approach to solve this issue; initially model BPEL business processes using Use Cases to capture users' perspective, and to systematically develop activity diagrams from Use Case models. A Travel Agency system case study is presented illustrates the feasibility of the proposed approach.

1 Introduction

Web services offer their users an efficient means to solicit and research publicly available services. A user maybe interested in acquiring the best deal on a particular service or a product from a number of competitors that offer that service or product. For example, a customer interested in purchasing a particular book will be interested in obtaining the best price from a number of book vendors. Alternatively, users can be interested in the collaboration of a number of web services to attain a higher level goal. For example, a user can be interested in a set of web services provided by couriers that can interact with each other to provide tracking and history details of a current shipment. BPEL processes can be created to specify the invocation order of web services to achieve the desired goal. Using BPEL, a great deal of interaction information between web services and the BPEL process user can be specified, commonly known as defining a business process. Every BPEL business process has a purpose to achieve; this purpose is usually to provide a service to the process's user. It is not necessary that the user must be human; the user of a business process can be another system. In any case, it is the responsibility of an E-commerce analyst to define BPEL business processes that provide the services that are in demand.

BPEL provides support to specify complex business processes that contain sequences, loops, conditions, exception handling, variable declarations and data editing. In essence, a BPEL business process defines a workflow. Therefore, it is common practice to model these workflows using the Unified Modeling Language (UML) activity diagrams, since it can be mapped directly onto BPEL code. In this paper, an activity diagram that represents a BPEL business process will be referred to as BPEL activity diagram. A BPEL activity diagram needs to posses a great deal of quality. A high quality BPEL activity diagram can be defined as one that accurately represents the workflow required to satisfy a business requirement. In the analysis phase, an E-commerce analyst will develop BPEL activity diagrams directly from a set of business requirements. This might be problematic for the following reasons:

- Activity diagrams are not geared towards capturing the user-centric perspective of business workflows. It is important to model such a perspective since the user is the principle beneficiary of the BPEL business process. Therefore, it is crucial to understand the means by which the user(s) will interact with the host system(s) during the execution of a BPEL business
- Activity diagrams do not capture the intricacies of interactions occurring between web services (external systems) and the host system that runs the BPEL business process process.
- It is common to define a set of related services using a set of BPEL business processes. This concept is illustrated through the Travel Agency System case study presented in Section 4. Activity diagrams are not designed to provide a mechanism to discover common activities, interactions and sub-services provided by a set of related BPEL services. Not being able to discover and factor out such commonalities might result in unnecessary effort required for implementing redundant functionalities at the end system; and since BPEL processes potentially have highly extended execution periods, these redundancies can be significant.

Therefore, in addition to knowing what the goals are, it is essential to know how the user will utilize the BPEL business process. It can be argued that understanding the user's participation in the business process will actually yield to creating higher quality activity diagrams, in the sense that the activity diagrams created will more accurately represent the user's involvement in the business process and the involvement of the available web services. To combat the issues presented above, we propose using Use Cases to model the user's perspective. Whereby, consequent activity diagrams can be developed based upon the Use Case models. Use Case modeling has become the de-facto technique for modeling user-centric systems. A Use Case model will detail the intricacies of interactions that occur between the BPEL business process users and the web services provided by external systems. Use Case modeling prompts E-commerce analysts to consider common behavior and sub-services allowing the development of simpler and more modular systems.

There are two main requirements to develop high quality BPEL activity diagrams. Firstly, it is crucial to develop a high quality Use Case model. It is intuitive that if one artifact is developed based on another artifact, the quality of the source artifact will extensively influence the quality of the resultant artifact. Utilizing the Structured Use Case

Descriptions (SUCD) form to develop Use Case descriptions can help achieve this goal. Use Cases described in the SUCD form will be referred to as SUCD Use Cases. Secondly, an equally important requirement is to provide a systematic transition between the Use Case model and the corresponding set of BPEL activity diagrams. This can be achieved by using the AGADUC (Automated Generation of Activity Diagrams from Use Cases) process. AGADUC is a systematic approach to transform Use Case models into UCADs (Use Case Activity Diagrams). UCAD is a notation introduced in [9] that represents the workflows embedded in SUCD Use Cases. The UCADs produced can be considered as BPEL activity diagrams upon which the implementation of the BPEL processes will be based. A principle advantage of using AGADUC is that it is supported by the tool AREUCD (Automated Reverse Engineering of UC Diagrams). Automating the transformation process reduces the time and effort required to develop BPEL activity diagrams. Therefore, E-commerce analysts can further focus on the development of high quality Use Case models. Furthermore, automating the transformation process will ensure consistency between Use Case models and activity diagrams by eliminating errors injected by analysts, and eliminate "inspection-like" effort required to ensure such consistency. An overview of the AGADUC process is shown in Figure 1.

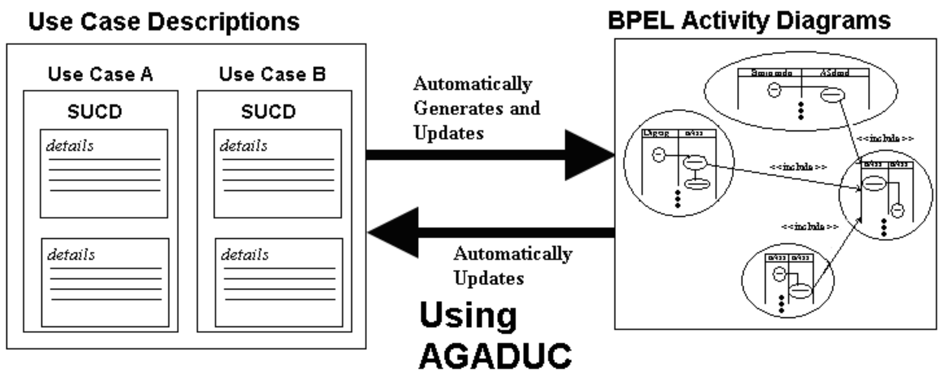


Figure 1: An overview of the AGADUC process

Activity diagrams are traditionally used to model BPEL processes since activity diagrams are designed to support the specification of workflows. Workflows that may contain loops, branches and conditions, and concurrent flows, and can be easily visualized using activity diagrams. E-commerce analysts have avoided adopting Use Cases to model their BPEL process, since Use Case models are not intended for expressing workflows, but rather actor-system interactions. It is rather cumbersome to express workflow features using Use Cases in a concise and understandable manner. Traditional Use Cases are described using unstructured natural language that makes it impossible to automatically extract and represent their embedded workflows using activity diagrams. SUCD helps overcome the limitations endured when using Use Cases to model workflows. As will be discussed in great detail in Section 3, SUCD contains enough structure to support the concise modeling of workflows. There will be no additional effort in creating the corresponding activity diagrams, since AREUCD automatically performs this operation. Therefore, SUCD allows

E-commerce analysts to utilize the user-centered approach to modeling their BPEL processes without sacrificing the benefits of visualizing their BPEL processes using activity diagrams!

The remainder of this paper is structured as follows: Section 2 discusses previous work related to modeling BPEL processes. Section 3 introduces the SUCD structure and the AGADUC process and how it can be utilized to describe BPEL business processes and automatically generate BPEL activity diagrams. In Section 4, a Travel Agency System case study is presented to illustrate feasibility of our proposed approach, and to demonstrate the application of the AREUCD tool. Section 5 concludes and discusses future work.

2 Related Work

Schader et el. [3] and Aagedel et al. [13] have shown through a number of case studies the limitations of using activity diagrams for business process modeling. Dumas et al. [4] extended this work by examining the expressiveness and adequacy of UML Activity Diagrams with respect to specify workflows in particular. The result of their study shows that activity diagrams possess features that allow for capturing situations arising in practice that usually cannot be captured by most Workflow Management Systems. However, their study also revealed that activity diagrams suffer from limitations inherited from statecharts, which stem from the fact that activity diagrams are a special type of state machines.

Korherr et al. [5] presented an extension to the UML Activity Diagram. The extension allows for modeling process goals and performance measures in a visual manner by the definition of a UML profile. The profile allows the extended activity diagrams to be mapped onto BPEL. Mantell also created a UML profile that supports the systematic transformation of activity diagrams to BPEL code [8]. Mantell's work featured tool support that automates the transformation process and generates skeletons of BPEL code.

In 1997, it was argued by Keung et al. that there was little contribution made to make goals explicit during the modeling of business processes [7]. Today, a large number of business modeling languages have been introduced such as the Business Process Modeling Language [6], the Event-driven Process Chain [11] and UML Activity Diagrams [10], however none of them provide means to model business process goals [1]. Towards achieving this objective, this paper proposes a use-centered approach to modeling BPEL business process using Use Cases.

3 The Structural Elements of SUCD

SUCD is in large based on the structure presented by [2]. Use Cases described using the SUCD structure contain five structured main sections, these are: (a) Use Case Name, (b) Basic Flow, (c) Alternative Flows, (d) Subflows and (e) Extension Points. Meanwhile, other sections in a Use Case description that do not require structure are described using natural language. There have many templates presented in the literature for describing Use Cases. This section will provide details about the SUCD structural elements that are relative to supporting the specification of workflows. A complete description of the SUCD structure and a mini-tutorial can be located at [12]. The systematic mapping of the SUCD structural elements to activity diagram notation and the underlying transformation algorithm is also presented in this section. The formalized mapping rules also serves as a mechanism to ensure inter-diagrammatic consistency between the Use Case model and the activity diagrams.

The following list outlines the structural elements of SUCD that support the specification of workflows which will be described in detail in the subsequent sections:

- Headers and Actions
  - Transforming Headers and Actions
  - Swimlanes
  - Nested Activity Diagrams
- Concurrency and Loop Support using the 'RESUME' and 'AFTER' Statements
- Condition Evaluation and Branching Support using 'AT' and 'IF' Statements

3.1 Headers and Actions

The basic building block comprising all structured components is headers. A header contains a number of actions that carryout certain behavior. The name of the header indicates the behavior that is carried out by its actions. A header is comprised by a pair of matching tags. An 'opening' tag is comprised of curly brackets that contain the header's name <header> prefixed with the keyword 'BEGIN'. Its corresponding 'closing' tag must contain the same header name and is prefixed with the keyword 'END'. A header's enclosed actions are normally listed in bullet form. For example, in a library system, a header may represent the actions required to enter information regarding how a new library member:

```
{BEGIN enter member information}
- Librarian → enters member's name
- Librarian → enters member's address
- Librarian → enters member's phone number
{END enter member information}
```

This header Enter Member Information contains three actions. In this paper, performing a header indicates that all of its enclosed actions are performed. Each header inside a Use Case must have a unique name. It can be easily deduced that the purpose behind performing the three actions shown above is to enter a member's information into the system. Moreover, all three actions must be performed to carryout the underlying purpose of the header. A header groups together a set of actions that must all be performed in order to carryout complete and meaningful behavior.

A header may contain other lower-level headers that comprise parts of the behavior required to carryout the main behavior represented by the higher-level header. Therefore, a Use Case description will contain a virtual tree of headers, whereby actions become the roots (see Fig. 2). A high-level header may have actions of its own. Performing a higher-level header forces the all of its lower-level headers in addition to its own actions to be performed. When a lower-level header is completely performed, its higher-level header resumes performance. Actions listed under a header represent the actual behavioral details

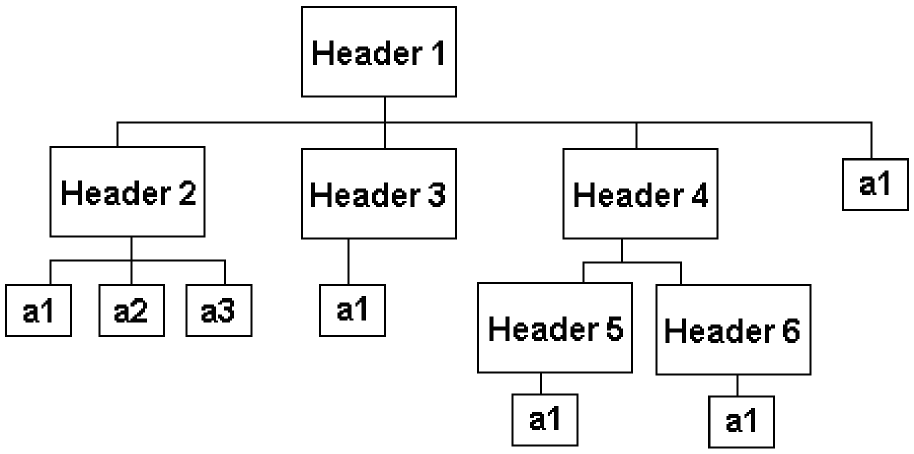


Figure 2: Headers in a Use Case descriptions form a virtual tree structure

of a Use Case. Actions must be listed in bullet form and described using natural language. Listing actions in bullet form will allow analysts and designers to trace back design artifacts and decisions to individual actions in a Use Case description. Only one actor may perform an action, unless the action is performed by the system itself. The name of the actor that performs a given action is prefixed to that action. For the Enter Member Information header shown above, the Librarian actor performs all three actions shown. Actions that are performed by the system itself are prefixed with the keyword 'SYSTEM'.

3.1.1 Transforming Headers

Since headers are the basic building blocks for the three types of flows, it is only appropriate to start with the transformation of headers to activity diagram elements. Each action enclosed in a header is directly represented by an activity in an activity diagram. The Enter Member Information header presented in section 3.1 is translated into the following activities as shown in Fig. 3.

3.1.2 Swimlanes

Swimlanes are used to associate each activity with an actor. This assigns the responsibilities of each actor. In SUCD Use Cases, each action is designated an actor, unless it is performed by the system (where the keyword 'SYSTEM' is used). Hence, assigning each action to the appropriate swimlane is straightforward (see Fig. 3).

{BEGIN enter member information}
- Librarian→ Enter member's name
- Librarian→ Enter member's address
- Librarian→ Enter member's phone number
- SYSYEM → Store member's information

{END enter member information}

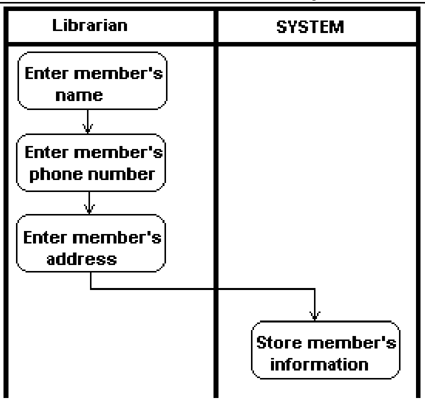


Figure 3: Headers and Swimlanes

3.1.3 Nested Activity Diagrams

Each activity diagram maybe nested. Nesting in activity diagrams is used to model a hierarchy between the activities and to manage the complexity of the activity diagram. Nesting activity diagrams do not change the semantics behind the activity diagrams. Therefore, whether the activity diagrams were nested or not, should not change the underlying concepts and workflows presented by the diagrams. Hence, there is no 'hard and fast' rule as to what sections in an activity diagram should be nested. Nesting sections of an activity diagram is a judgment call made by the E-commerce analysts and designers. The proposed Use Case description structure only provides guidelines to what can be nested.

SUCD uses headers only to show what can be nested, as supposed to what should be nested. A set of actions can be abstracted to show their corresponding header as an activity. Lower-level headers can be abstracted to show corresponding higher-level header as an activity. Assuming a header named Enroll New Member that is composed of three lower-level headers; Enter Member Information, Enter Record into Library Database and Produce Library Card For New Member. As shown in figure 4, the lower-level headers can be abstracted to show the higher-level header as an activity. The Enter Member Information header presented in Section 3.1 is composed of actions. The actions can be abstracted to show its header as an activity (see Fig. 4).

3.2 Concurrency and Loop Support

Activity diagrams have features such as forking and joining to support activity synchronization. As already discussed, forking and joining execution flows in activity diagrams are modeled using the 'RESUME' and 'AFTER' statements in the Use Case descriptions.

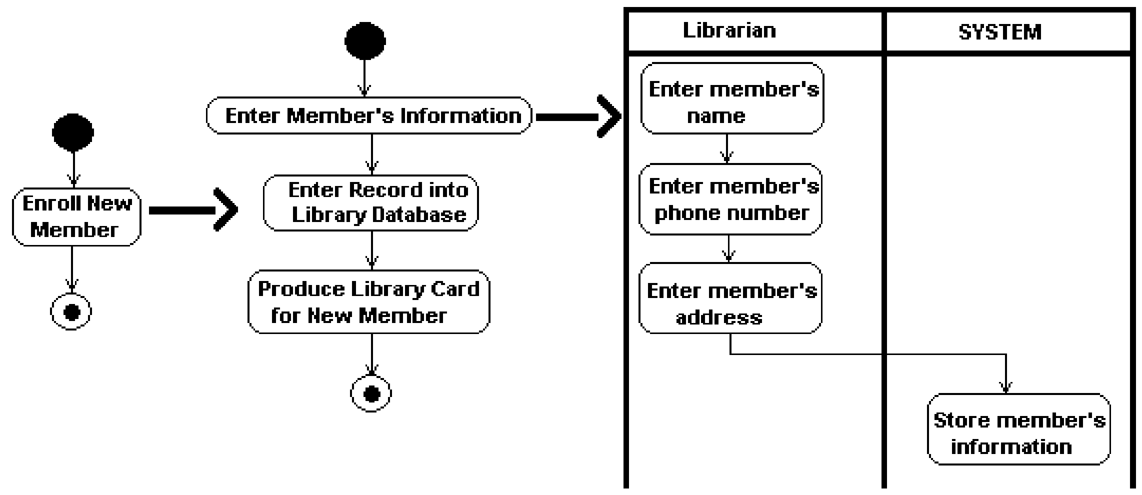
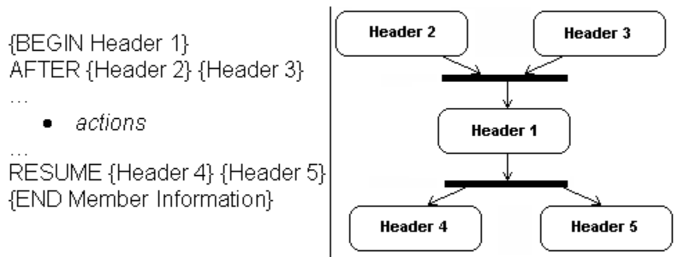

Enroll New Member
Enter Member's Information
Enter Record into Library Database
Produce Library Card for New Member
Librarian
SYSTEM
Enter member's name
Enter member's phone number
Enter member's address
Store member's information

Figure 4: Different nesting levels for presentation purposes

A header can explicitly state the header(s) to be performed next. This can be achieved using the 'RESUME' statement. The 'RESUME' statement consists of the keyword 'RESUME' followed by a list of header(s) that will follow. This is used to model the concept of flow forking. Finally, a header may explicitly state the headers that must be completed before it can commence. This is achieved using the 'AFTER' statement. Similarly, the 'AFTER' statement consists of the keyword 'AFTER' followed by a list of headers that need to be completed first (see Fig. 5):



```
{BEGIN Header 1}
AFTER {Header 2} {Header 3}
...
  • actions
...
RESUME {Header 4} {Header 5}
{END Member Information}
```

Figure 5: Forking and joining using SUCD

3.3 Supporting Condition Evaluation and Branching

Workflows may contain decision points where a condition is evaluated. Decision points are indicated using the 'AT' statement. The 'AT' statement consists of the keyword 'AT' followed by a header where the stated conditions are evaluated. By default, the conditions are evaluated for each action within the specified header. Alternatively, the conditions can be evaluated only at certain actions within the specified header. This is achieved by specifying these actions after the 'AT' statement.

Conditions evaluated at each 'AT' statement are indicated using 'IF' statements. An 'IF' statement consists of the keyword 'IF', followed by a condition described in natural language.

An 'AT' statement will cause the creation of a decision diamond following the activity(s) that represent the header(s) listed in the 'AT' statement. An 'IF' condition will be displayed as an activity diagram condition at the corresponding decision diamond (see Fig. 6). A 'RESUME' statement is used afterwards to specify the action where the flow will be heading towards.

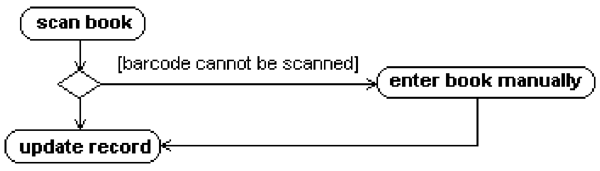


Figure 6: Modeling conditions and braches

3.4 Formalizing the SUCD Structure

It is essential for the grammar and constructs of the SUCD structure to be formalized. Formalizing the SUCD structure will provide a strict guideline to Use Case authors in composing Use Case descriptions, so that there is no disagreement or ambiguity as to what is allowed and what is not. The grammar of the SUCD structure is defined below in E-BNF (see Table 1). Due to space restrictions, only the grammar of SUCD's higher structural constructs are shown below, while the grammar of minor structural constructs are excluded. However, the entire E-BNF is located at [12].

3.5 The AGADUC Process Mapping Rules

In order for the AGADUC process to be tool supported, the mapping rules for transforming SUCD Use Cases in activity diagrams must be formalized. A complete specification of the implementation of the mapping rules will require many pages in length. A summarized pseudo code version of the mapping rules are presented below (see Fig. 7).

The mapping process is carried out by four main algorithms [12]. The first algorithm is responsible for scanning through the headers and actions of a SUCD Use Case and creating a hierarchy of activities that represent this hierarchy. Algorithm 1 will also assign activities representing actions to their corresponding swimlanes. Finally, the first algorithm will set control flow link between the actions of a header, and save any information regarding RESUME and AFTER statements detected. The second algorithm is responsible for building and maintaining a list of swimlanes detected from the actions. The third algorithm is responsible for creating synchronization bars and creating control flow links that connect activities with the synchronization bars. The execution of this algorithm depends

S ::= UseCaseDescrption+ Actor+
Actor ::= Abstract? ActorName Implements? Specializes?
UseCaseDescrption ::= NameSection BasicFlowSection? AlternativeFlowSection? SubflowsSection? ExtensionPointsSection?
NameSection ::= 'Use Case Name:' Abstract? UseCaseName Implements? Specializes?
Abstract ::= 'ABSTRACT'
Implements ::= 'IMPLEMENTS' UseCaseName
Specializes ::= 'SPECIALIZES' UseCaseName
BasicFlowSection ::= 'Basic Flow:' 'BEGIN Use Case' Header* 'END Use Case'
Header ::= 'BEGIN' HeaderName '' AfterStatement? Contents* ResumeStatement? 'END' HeaderName ''
AlternativeFlowsSections ::= 'Alternative Flows:' AF*
AF ::= AtStatement IfStatement AFHeader

Table 1: E-BNF grammar for the SUCD structure

on the information saved earlier regarding the RESUME and AFTER statements. The third algorithm requires that Algorithm 1 must be executed first. Finally, the fourth algorithm is responsible for creating the decision diamonds and conditions according to the AT and IF statements. The fourth algorithm requires that Algorithm 1 and Algorithm 3 are executed first. Due to space restricitions, only Algorithm 1 is shown below; the remaining Algorithms can be found at [12].

4 Business Traveler Case Study

The following case study is used to demonstrate how SUCD Use Cases can be used to by the AGADUC process to generate UCADs. The case study is about a simplified Business Traveler System that allows employees of a certain company to travel using the best offers for plane ticket(s) and travel insurance. The system provides three BPEL business processes. The first business process allows an employee to retrieve the best plane ticket offer from two web services provided by American Airlines and Delta Airlines. The second business process allows an employee to receive the best travel insurance offer for an upcoming trip from two web services provided by Northern Insurance and Pacific Insurance. When searching for the best plane ticket offer or the best travel insurance offer, the employee's travel status must be checked. The travel status of an employee allows the company to

Algorithm 1: Transforming *headers* and *actions* into *activities*.
Input: The description of a SUCD Use Case.
Output: A hierarchy of *activities* based on the given hierarchy of the given *headers* and their *actions*.

```
Top_ Activity = NULL
Current_Parent_Activity = NULL
LET Statement = getFirstStatement(SUCD Use Case)

WHILE (EOF is not reached)
{
      IF Statement = Opening header tag
            Detect header name
            Create an activity and set its name as:
            Activity.name = header.name
            IF (Top_Header found)
                  Set activity as child of Current_Parent_Activity
                  Set activity as Current_Parent_Activity

            ELSE Set activity as Top_ Activity
                  Set activity as Current_Parent_Activity

      ELSE IF Statement = Closing header tag
            Set Current_Parent_Activity as
getLastActivity().getParent()

      ELSE IF Statement = action
            Detect action name
            Create an activity and set its name as:
            Activity.name = action.name

                  Detect action's actor
                  Create currentSwimlane = action.actor
                  Check for currentSwimlane (perform Algorithm 2)
                  currentSwimlane = swimlane reference returned from
Algorithm 2
                  Set activity.swimlane = currentSwimlane
                  currentSwimlane.addAcitivty (activity)
                  Set activity as child of Current_Parent_Activity
                  IF Not First or Last action in
Current_Parent_Activity
                        Create flow link from previous action in parent
to self
                  ELSE IF Last action in Current_Parent_Activity
                        Create flow link from previous action in parent
to self
         Look for any RESUME statements following it. If a RESUME
         statement exists, the save the headers specified as a property
         of that action.
                  ELSE IF First action in Current_Parent_Activity
         Look for any AFTER statements before it. If an AFTER statement
         exists, save the headers specified as a property of that
         action.

      ELSE //Statement is just an unstructured statement

      Statement = getNextStatement();
```

Figure 7: Algorithm 1. Detecting activities and creating main flow links

determine which class that employee can use to travel, such as: business, first or economy class. The travel status also allows the company to determine the corresponding travel insurance package that the employee is entitled to receive. The travel status checking process is performed by the third BPEL process called "Check Employee Travel Status". The Use Case diagram of the system is presented below in Figure 8.

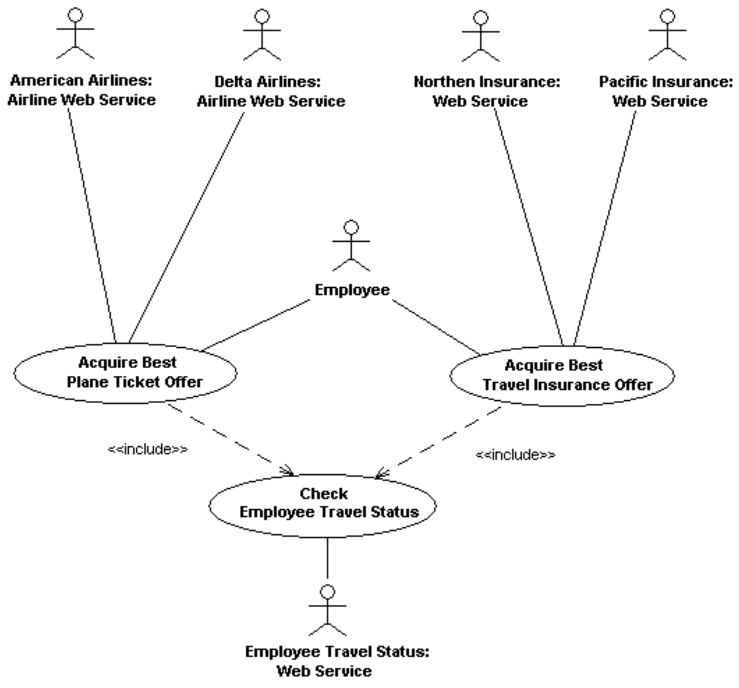

American Airlines: Airline Web Service
Delta Airlines: Airline Web Service
Northen Insurance: Web Service
Pacific Insurance: Web Service
Employee
Acquire Best Plane Ticket Offer
Acquire Best Travel Insurance Offer
<<include>>
<<include>>
Check Employee Travel Status
Employee Travel Status: Web Service

Figure 8: The Travel Agency System Use Case Diagram

As discussed in Section 3, BPEL business processes represent services that are offered by a system to its user(s). Therefore, the BPEL business processes are represented as Use Cases. The web services and the user of the BPEL processes interact with the "Use Cases" to attain their services. Therefore the web services and users are modeled as actors.

This case study is an expansion of the BPEL business process example presented in [14]. In [14], only one BPEL business process was discussed which combined the "Acquire Best Plane Ticket Offer" and "Check Employee Travel Status" BPEL processes shown below. The reason the "Check Employee Travel Status" BPEL process was created in our case study is to allow the "Acquire Best Travel Insurance Offer" to use its offered service. Therefore, using a Use Case driven approach, it was possible to identify and factor out common behavior, which will help avoid the implementation of any redundant functionality and save development costs.

The BPEL activity diagram for the "Acquire Best Travel Insurance Offer" BPEL process is presented in Fig. 9. The BPEL business process was then implemented. Therefore, in this case study we will shed the light on the "Acquire Best Travel Insurance Offer" BPEL process since it was discussed in [14], to show that using a Use Case driven approach can produce the same BPEL activity diagram. Moreover, this case study will show how using Use Cases can overcome the limitations suffered by using the traditional development approach presented earlier in Section 1. Once again due to space resatrictions, the Use Case descriptions for the "Acquire Best Plane Ticket Offer" BPEL process is presented below using the SUCD structure, while the Use Case description of the "Check Employee Travel Status" BPEL process can be located at [12].

Use Case Name:
Acquire Best Plane Ticket Offer

Brief Description:
This Use Case describes a simple business process that selects the best airline flight ticket offer. The business process is carried out as a web service. Currently, there are two competing Airline companies that have subscribed to this web service, namely (a) American Airlines, and (b) Delta Airlines.

Preconditions:
The Employee must have approval to travel.

Basic Flow:
{BEGIN Use Case}

{BEGIN Receive the initial request}
- Employee → requests to search for the best plane ticket offer
- Employee → specifies his/her name
- Employee → specifies the destination
- Employee → specifies the departure date
- Employee → specifies the return date

{END Receive the initial request}

{BEGIN Prepare the input for the Employee web service}
- SYSTEM → retrieves the information inputted by the Employee and prepares for the Employee Travel Status Web Service

{END Prepare the input for the Employee web service}

{BEGIN Retrieve the employee travel status}
- SYSTEM → sends the Employee Travel Status Web Service the Employee information to check for the Employee's travel status
- INCLUDE Check Employee Travel Status

{END Retrieve the employee travel status}

{BEGIN Prepare the input for both Airline web services}
- SYSTEM → uses information provided by the Employee and the Employee Travel Status Web Service to prepare inquiry requests for both Airlines

RESUME {Acquire plane ticket offer from American Airlines} {Acquire plane ticket offer from Delta Airlines}
{END Prepare the input for both Airline web services}

{BEGIN Acquire plane ticket offer from American Airlines}
- American Airlines: Airlines Web Service → retrieves information about the requested plane ticket(s)
- American Airlines: Airlines Web Service → checks for tickets availability
- American Airlines: Airlines Web Service → if the requested ticket(s) are available the web service sends back an offer for the ticket(s). Otherwise, if the tickets were unavailable, the web service sends back a response indicating that
{END Acquire plane ticket offer from American Airlines}

{BEGIN Wait for a callback from American Airlines}
- SYSTEM → waits for a response from the American Airlines Web Service {END Wait for a callback from American Airlines}

{BEGIN Acquire plane ticket offer from Delta Airlines}
- Delta Airlines: Airlines Web Service → retrieves information about the requested plane ticket(s)
- Delta Airlines: Airlines Web Service → checks for tickets availability
- Delta Airlines: Airlines Web Service → if the requested ticket(s) are available the web service sends back an offer for the ticket(s). Otherwise, if the tickets were unavailable, the web service sends back a response indicating that
{END Acquire plane ticket offer from Delta Airlines}

{BEGIN Wait for a callback from Delta Airlines}
- SYSTEM → waits for a response from the Delta Airlines Web Service
{END Wait for a callback from Delta Airlines}

{BEGIN Select the best offer}
AFTER {Wait for a callback from American Airlines} {Wait for a callback from Delta Airlines}
- SYSTEM → after receiving a response from both Airlines web services, the SYSTEM selects the best offer
{END Select the best offer}

{BEGIN Return the offer}
- SYSTEM → returns to the Employee a response indicating the best offer for the requested plane tickets
{END Return the offer}

{END Use Case}

Postconditions:
A response must be provided to the Employee with the best offer for the requested plane tickets. If no tickets were available, a response must be provided to the Employee indicating that.

Special Requirements:
An internet connection must be available for the BPEL process to operate.

Upon inputting the SUCD Use Cases (all three of them) into the tool AREUCD, the AGADUC process will be performed and the UCADs for the SUCD Use Cases are generated. Due to space limitations, only the UCADs of the “Acquire Best Plane Ticket Offer” and “Check Employee Travel Status” BPEL processes are shown below (see Fig. 9 and Fig. 10).

In contrast with using the activity diagram for the “Acquire Best Plane Ticket Offer” business process, it can be deduced that describing the business process using a SUCD Use Case have provided an explicit representation of the user goals and have also provided much more details about the interactions between the BPEL process and the user and other web services.

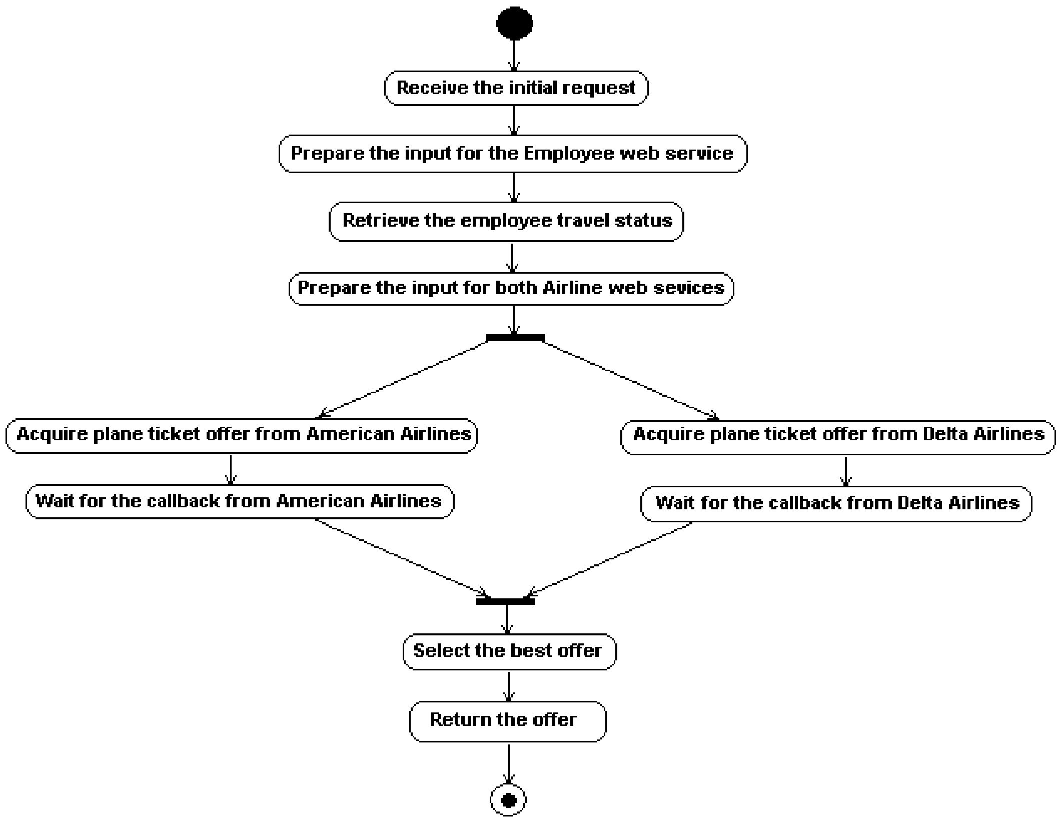

Receive the initial request
Prepare the input for the Employee web service
Retrieve the employee travel status
Prepare the input for both Airline web sevices
Acquire plane ticket offer from American Airlines
Acquire plane ticket offer from Delta Airlines
Wait for the callback from American Airlines
Wait for the callback from Delta Airlines
Select the best offer
Return the offer

Figure 9: The resulting BPEL activity diagram for the “Acquire Best Plane Ticket Offer” BPEL business process

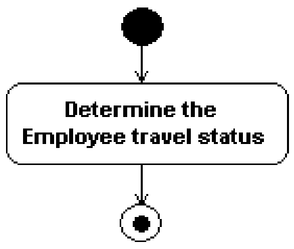


Figure 10: The resulting BPEL activity diagram for the "Check Employee Travel Status" BPEL business process

5 Conclusion

In this paper we presented proposed a user-centered approach to modeling BPEL business processes. The approach is based on using Use Cases to explicitly model the services offered to the business processes' users. Traditionally, BPEL business processes are modeled using UML activity diagrams only, which did not support the explicit modeling of user goals. Another advantage of using Use Cases is that the interaction intricacies between the business processes and the web services offered by other systems can be captured. Use Case modeling provides a high level overview of the services that are provided by a set of related BPEL business processes. This will allows E-commerce analysts to discover common services and functionality which in turn will save time and effort by avoiding the implementation of redundant functionality.

Modeling BPEL business processes using activity diagrams appeals to E-commerce analysts since activity diagrams is an excellent technique to model and visual workflows. Moreover, it can be directly mapped to BPEL code to kick start the implementation phase. Traditional Use Cases are described using unstructured natural language. Describing workflows concisely using unstructured natural language is difficult since workflows contain features such as loops, conditions and branches and concurrent flows. In this paper, we utilize the SUCD structure to describe Use Cases. The SUCD structure features structural elements that allow E-commerce analysts to describe the BPEL workflows concisely and accurately. Using the AGADUC process, which is implemented by the AREUCD tool, E-commerce analysts will be able to effortlessly generate activity diagrams that accurately represent the workflows described by SUCD Use Cases. This allows E-commerce analysts to utilize a user-centered approach to model their BPEL processes without losing the advantages of activity diagrams.

Future work can be directed towards developing a Use Case driven approach to systematically generate test suites that will check the validity of the BPEL processes.

References

[1] A-W Scheer. *ARIS – Business Process Modeling*. Springer Verlag, 1999.

[2] J. Aagedal and Z. Milosevic. ODP enterprise language: An UML perspective. In *In Proc. of The 3rd International Conference on Enterprise Distributed Object Computing*. IEEE Press, 1999.

[3] K. Bittner and I. Spence. *UC Modeling*. Addison-Wesley, 2002.

[4] M. Dumas and A. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proc. of the UML 2001 Conference*, 2001.

[5] B. P. M. Initiative. *BPMI: Business Process Modelling Notation – Specification v1.0*, November 2004.

[6] B. Korherr and B. List. Extending the UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL. In *2nd International Workshop on Best Practices of UML (BP-UML'06)*. Spinger Verlag, Lecture Notes in Computer Science, November 2006.

[7] P. Kueng and P. Kawalek. Goal-based business process models: creation and evaluation. *Business Process Management Journal*, Volume 3(1):17–38, 1997. MCB Press.

[8] B. List and B. Korherr. An Evaluation of Conceptual Business Process Modelling Languages. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06)*. ACM Press, April 2006.

[9] M. B. M. B. Juric and P. Sarang. *Business Process Execution Language for Web Services. Second Edition*. PACKT Publishing, 2006.

[10] J. M. M. El-Attar. AGADUC: Towards a More Precise Presentation of Functional Requirements in Use Case Models. In *4th ACIS International Conference on Software Engineering, Research, Management and Applications*, 2006.

[11] K. Mantell. *From UML to BPEL – http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel*, September 2003.

[12] I. Object Management Group. *UML 2.0 Superstructure, http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf*, November 2006.

[13] M. Schader and A. Korthaus. Modeling business processes as part of the BOOSTER approach to business object-oriented systems development based on UML. In *Proc. of The Second International Enterprise Distributed Object Computing Workshop*. IEEE Press, 1998.

[14] STEAM laboratory website at the University of Alberta. *http://www.steam.ualberta.ca/main/research_ areas/Requirements_ Capture.htm*, Dec 2006.

e-Informatica Software Engineering Journal, Volume 1, Issue 1, 2007

Program Verifications, Object Interdependencies, and Object Types

Cong-Cong Xing*

Department of Mathematics and Computer Science, Nicholls State University

cmps-cx@nicholls.edu

Abstract

Object types are abstract specifications of object behaviors; object behaviors are abstractly indicated by object component interdependencies; and program verifications are based on object behaviors. In conventional object type systems, object component interdependencies are not taken into account. As a result, distinct behaviors of objects are confused in conventional object type systems, which can lead to fundamental typing/subtyping loopholes and program verification troubles. In this paper, we first identify a program verification problem which is caused by the loose conventional object typing/subtyping which is in turn caused by the overlooking of object component interdependencies. Then, as a new object typing scheme, we introduce *object type graphs* (OTG) in which object component interdependencies are integrated into object types. Finally, we show how the problem existing in conventional object type systems can be easily resolved under OTG.

1 Introduction and Related Work

Although much of the recent year's work on object-oriented programming (OOP) has focused on large entities such as components, environments, and tools, investigations on issues related to object-oriented languages themselves are still an on-going research and many new improvements can be expected. In particular, typing and program verification are still a critical issue and a problem-prone area in the formal study of object-oriented languages, especially when type-related subjects, such as subtyping and inheritance, are considered. In the contexts of OOP theory research, there are three major lines: Abadi-Cardelli's ς-calculus [2], Fisher-Mitchell's lambda calculus of objects [14, 19, 18, 4], and Bruce's PolyTOIL [7, 6]. The type systems of all these calculi are *conventional* in the following sense: the major behavior indicator of objects – object component interdependencies – is not reflected in object types.

The result of not having such component interdependency information represented in object types is that two behaviorally distinct objects which deserve to be typed differently, may have the same type. For example, let objects a and b be defined, using the ς-calculus [2] notation, as follows: $a \stackrel{\text{def}}{=} [l_1 = 1, l_2 = 1]$, $b \stackrel{\text{def}}{=} [l_1 = 1, l_2 = \varsigma(s : \mathit{Self})s.l_1]$ where s is the self variable and *Self* is the type of s. The behavioral difference between a and b can be revealed by the following computations: Suppose we would like to update l_1 in a to 2.

It is easy to see that before and after this updating operation, the "status" of l_2 in a remains the same, namely, $a.l_2 = 1$ and $(a.l_1 \Leftarrow 2).l_2 = [l_1 = 2, l_2 = 1].l_2 = 1$[1]. However, when the same operation (updating l_1 to 2) is applied to b, the "status" of l_2 in b would be changed after the operation, namely, $b.l_2 = 1$ but $(b.l_1 \Leftarrow 2).l_2 = [l_1 = 2, l_2 = \varsigma(s : \mathit{Self})s.l_1].l_2 = 2$ due to the fact that l_2 "depends on" l_1 (l_2 calls l_1) in b. In conventional type systems, this behavioral difference between a and b is not captured in their types; a and b are of the same type: $[l_1 : int, l_2 : int]$. As a result, elusive programming errors and program verification problems will inevitably occur when subtyping is considered (as shown in the next section).

In this paper, we introduce, as a new way to represent object types, *object type graphs* (OTG) in which object component interdependency information is abstractly revealed, and show that OTG provides an effective support for program verifications. Section 2 presents a program verification problem caused by object typing. Section 3 defines a formal object-oriented language **TOOL** in which object component interdependencies are to be studied. Sections 4 and 5 define OTG and typing/subtyping under OTG respectively. Section 6 demonstrates how the program verification problem shown in Section 2 can be resolved under OTG. Section 7 concludes this paper.

There are some research work in the literature that are (somehow) related to our work. *Behavioral subtyping* is introduced in [20]. Although object behavior and subtyping are the common interests in both [20] and our paper, our typing approach is fundamentally different from that in [20] where object interdependencies are not considered. *Labeled types* and *width subtyping* are proposed in [3, 4, 19], where the type of a method is labeled by a set of methods that it uses. While the idea of labeled types is somewhat related to our idea of object interdependency, they differ substantially in quality and in quantity. For example, the notion of object interdependency is precisely defined in our work whereas the issue of method usages is not formally addressed in labeled types. Furthermore, in our work, object interdependencies fully participate and decisively reshape object subtyping whereas in labeled types the method usages information is barely used in object subtyping. The notion of *object state typing* can be found in, for example, [9, 21]. Just like [20] (as opposed to our work), this approach deals with the issue of object behavior and subtyping in a fundamentally different way from ours, which makes it orthogonal and complemental to our approach.

2 The Problem and Motivation

Points with additional attributes (e.g., color points [5, 8, 15], movable points [2, 4, 15]) have been an interesting study-case in the fundamental research of object-oriented languages. Here, we observe a new problem that is associated with movable points. We first present this problem on a theoretical basis and then demonstrate it using Java.

[1] $a.l_1 \Leftarrow 2$ means that field/method l_1 in a is updated to 2.

2.1 ς-calculus Description of the Problem

We stipulate that a point is *colored* (or *non-colored*, respectively), if this point (object) has a (or has no, respectively) color attribute. Let us consider non-negative movable points[2]. For 1-d movable points, we assume that all points greater than 1 are colored points and all other points are non-colored points (Figure 1). For 2-d movable points, similarly, we assume that all points with a distance from the origin greater than 1 are colored points and all other points are non-colored points (Figure 2). This assumption can be easily extended for higher-dimensional points.

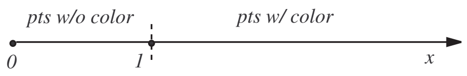


Figure 1: 1-d Colored and non-colored points

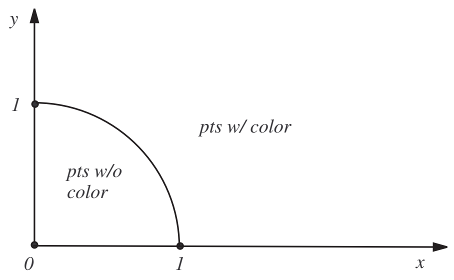


Figure 2: 2-d Colored and non-colored points

For instance, using the ς-calculus (second-order) notation [2], we can define a 1-d non-colored movable point and a 1-d colored movable point as follows:

$$p_{1n} \stackrel{\text{def}}{=} \begin{bmatrix} x = 0.5 \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ dist = \varsigma(s\!:\!Self)s.x \end{bmatrix},$$

$$p_{1c} \stackrel{\text{def}}{=} \begin{bmatrix} x = 2.0 \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ dist = \varsigma(s\!:\!Self)s.x \\ clr = blue \end{bmatrix},$$

where *mvx* moves the point to a new position on the x-axis and *dist* returns the dis-

[2]For the sake of simplicity, we only consider non-negative points here. The case for negative points can be easily duplicated with slight changes.

tance from the origin to the current position of the point. The $\Leftarrow$ is the method updating/overriding operation in ς-calculus. The intentions of fields x and clr are obvious.

To characterize the behaviors of 1-d movable points, we define the following types:

$$P \overset{\text{def}}{=} \varsigma(\mathit{Self}) \begin{bmatrix} x : \mathit{real} \\ \mathit{mvx} : \mathit{real} \to \mathit{Self} \\ \mathit{dist} : \mathit{real} \end{bmatrix},$$

$$CP \overset{\text{def}}{=} \varsigma(\mathit{Self}) \begin{bmatrix} x : \mathit{real} \\ \mathit{mvx} : \mathit{real} \to \mathit{Self} \\ \mathit{dist} : \mathit{real} \\ \mathit{clr} : \mathit{color} \end{bmatrix},$$

$$NCP \overset{\text{def}}{=} P,$$

where P is the type of all 1-d movable points, CP is the type of 1-d colored points, and NCP is the type of 1-d non-colored points. Given the objects and types defined as the above, it is easy to check that in conventional object type systems, we have $p_{1n} : NCP$, $p_{1c} : CP$, $CP <: P$, and $NCP <: P$.

Now, suppose we would like to write a program, ms ("move and see"), which takes a 1-d point and moves it along the x-axis. Due to the co-existence of colored and non-colored points on the x-axis, the movement cannot be arbitrary. We specify the behavior of ms as follows: (a) ms moves the argument point to its right a certain amount of distance if the argument point is colored (so that it will not mix with non-colored points). (b) ms moves the argument point to its left half of the distance from the origin to the current position of the argument point if the argument point is non-colored (so that it will not mix with colored points). (c) Let p' be the newly resulted point in cases (a) and (b). In case (a), ms uses the property $p'.\mathit{dist} > 1$ of p' to carry out the computation $\arcsin(1/p'.\mathit{dist})$; in case (b), ms uses the property $p'.\mathit{dist} \leq 1$ of p' to carry out the computation $\arcsin(p'.\mathit{dist})$. Because of subtyping and subsumption, inevitably, ms will take higher dimensional points as its arguments. To ensure that ms works fine with higher dimensional points, we require that, in such cases, the higher dimensional point be moved (right or left) along the x-axis, and the amount of distance to be moved follows the same guideline stated above. For example, given a 2-d point p with coordinates (x, y), if p is colored (which means $\sqrt{x^2 + y^2} > 1$), we move it to the right along the x-axis over a distance $\delta > 0$. The distance from the origin to the new position of the point then would be $\sqrt{(x+\delta)^2 + y^2} > \sqrt{x^2 + y^2} > 1$, indicating that the point is still in the colored point area on the x-y plane. If p is non-colored (which means $\sqrt{x^2 + y^2} \leq 1$), we move it to the left along the x-axis half of x. The distance from the origin to the new position of the point then would be $\sqrt{(\frac{1}{2}x)^2 + y^2} < \sqrt{x^2 + y^2} \leq 1$, indicating that the point is still in the non-colored point area. Thus the specification of the program ms is sound and feasible.

With little effort, we can write *ms* as follows:

```
$ms \overset{\text{def}}{=} \lambda(p : P)$
    if $(p.dist > 1)$                        // $p$ is colored
        $sin^{-1}(1/(p.mvx(\delta)).dist)$         // $\delta > 0$
    else                                     // $p$ is non-colored
        $sin^{-1}((p.mvx(-\frac{1}{2}p.x)).dist)$
    endif
```

Figure 3: The function *ms*

Now, the question we have is: does *ms* perform to its specification with all permissible arguments? Or simply, is *ms* reliable? Can we verify its correctness?

It is easy to check that *ms* works as expected with p_{1n} and p_{1c}. We now define one colored 2-d point and two non-colored 2-d points as follows:

$$p_{2c} \overset{\text{def}}{=} \begin{bmatrix} x = 2.0 \\ y = 2.0 \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ mvy = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.y \Leftarrow s.y + i) \\ dist = \varsigma(s\!:\!Self)\sqrt{(s.x)^2 + (s.y)^2} \\ clr = blue \end{bmatrix},$$

$$p_{2n} \overset{\text{def}}{=} \begin{bmatrix} x = 0.5 \\ y = 0.3 \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ mvy = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.y \Leftarrow s.y + i) \\ dist = \varsigma(s\!:\!Self)\sqrt{(s.x)^2 + (s.y)^2} \end{bmatrix},$$

$$p'_{2n} \overset{\text{def}}{=} \begin{bmatrix} x = 0.5 \\ y = \varsigma(s\!:\!Self)\frac{1}{4(s.x)} \\ mvx = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.x \Leftarrow s.x + i) \\ mvy = \varsigma(s\!:\!Self)\lambda(i\!:\!real)(s.y \Leftarrow s.y + i) \\ dist = \varsigma(s\!:\!Self)\sqrt{(s.x)^2 + (s.y)^2} \end{bmatrix}.$$

Note that p_{2c} and p_{2n} can be regarded as "free" 2-d points since their x and y fields are independent each other, whereas p'_{2n} can be regarded as a "constrained" 2-d point since its y coordinate depends on its x coordinate. Also note that p'_{2n} is a legitimate non-colored point since its coordinate is $(0.5, 0.5)$ which shows that the distance from the origin to this point is less than 1. Moreover, note that although p_{2c}, p_{2n}, and p'_{2n} are defined from scratch, they could have been defined through inheritance from (the classes of) p_{1c} or p_{1n} in class-based object-oriented languages (as shown in the next subsection).

Under conventional object type systems, p_{2c} and p_{2n} have types

$$CP_2 \stackrel{\text{def}}{=} \varsigma(\mathit{Self}) \begin{bmatrix} x : \mathit{real} \\ y : \mathit{real} \\ \mathit{mvx} : \mathit{real} \to \mathit{Self} \\ \mathit{mvy} : \mathit{real} \to \mathit{Self} \\ \mathit{dist} : \mathit{real} \\ \mathit{clr} : \mathit{color} \end{bmatrix}$$

and

$$NCP_2 \stackrel{\text{def}}{=} \varsigma(\mathit{Self}) \begin{bmatrix} x : \mathit{real} \\ y : \mathit{real} \\ \mathit{mvx} : \mathit{real} \to \mathit{Self} \\ \mathit{mvy} : \mathit{real} \to \mathit{Self} \\ \mathit{dist} : \mathit{real} \end{bmatrix}$$

respectively, and p'_{2n} has the same type as p_{2n}. That is, $p'_{2n} : NCP_2$. Furthermore, $CP_2 <: P$ and $NCP_2 <: P$, so $ms(p_{2c})$, $ms(p_{2n})$ and $ms(p'_{2n})$ all type-check.

It is easy to check that $ms(p_{2c})$ and $ms(p_{2n})$ work just fine. What about $ms(p'_{2n})$? It is supposed to return the degree of an angle. Unfortunately, the execution of $ms(p'_{2n})$ produces a run-time error, as outlined below: The current position of p'_{2n} is $(0.5, 0.5)$ with $p'_{2n}.\mathit{dist} = \sqrt{0.5^2 + 0.5^2} < 1$. So it is moved to the left $\frac{0.5}{2} = 0.25$ units of distance resulting in another point, say, p''_{2n}. The position of p''_{2n} is $(0.25, \frac{1}{4 \times 0.25}) = (0.25, 1)$ and the distance from the origin to p''_{2n} is $p''_{2n}.\mathit{dist} = \sqrt{0.25^2 + 1^2} > 1$. The execution $\mathit{sin}^{-1}(p''_{2n}.\mathit{dist})$ thus crashes because sin^{-1} is undefined for argument greater than 1.

What goes wrong is clear: when the x-coordinate of p'_{2n} is moved (decreased), its y-coordinate is *implicitly* moved too (increased) due to the interdependency between x and y ($y = \frac{1}{4(s.x)}$). The combination of these two movements makes p'_{2n} (a non-colored point) go into the colored point area of the x-y plane, resulting in a point with distance greater than 1 and creating semantics confusions. The importance of object component interdependencies to object behaviors can be seen clearly here. Conceptually, for ms to safely fulfill its specifications, it should not take an arbitrary point as its argument. Any points in which some methods depend on x and affect dist at the same time, for example p'_{2n}, will potentially make the behavior of ms unpredictable and endanger the execution of ms when they are submitted to ms. Thus, allowing points like p'_{2n} to be submitted to ms is a *"wrong idea"*, in the sense that $ms(p'_{2n})$ does not work as specified and therefore ms is unreliable.

How can we fix this problem? Is the function ms composed incorrectly? Is there a way to rewrite ms so that we can prove that ms works as specified for all permissible arguments? It seems unlikely. Note that ms is written with P as the type of its argument. ms cannot foresee what kind of extra methods there are in its actual arguments. When p'_{2n} is submitted to ms, p'_{2n}'s y-coordinate is *invisible* to ms. ms does not know the existence of the y-coordinate, and of course, has no way of knowing the interdependencies between y and other methods and the ensuing behavior of p'_{2n}. This is especially the case if p'_{2n} is constructed via inheritance from p_{1c} or p_{1n}. This situation causes the behavior of ms (with various permissible arguments) unpredictable, and is *inevitable* in OOP supported by conventional object type systems.

2.2 Java Version of the Problem

To show that the problem exists not only in object-based languages, but in classed-based languages as well, we present a Java version of the problem with two running scripts in Figure 4.

Classes `P`, `CP`, `CP2`, and `NCP2` correspond to types P (and NCP), CP, CP_2, and NCP_2 respectively. Similarly, objects `p1n`, `p1c`, `p2n`, `p2c` and `p2na` correspond to points p_{1n}, p_{1c}, p_{2n}, p_{2c}, and p'_{2n} respectively. `MPP` and `MPP1` are two applications that use these points. Due to the "class-serves-as-type" feature of Java, the Java version of the problem is twisted a bit: The types of `p2n` and `p2na` are `NCP2` and `NCP2a` respectively. These two types are not the same but enjoy a subtyping relationship `NCP2a` $<:$ `NCP2`. This is different from ς-calculus where p_{2n} and p'_{2n} have the same type, but does not affect the illustration of the problem.

Note that in class `NCP2a` of Figure 4, in order to faithfully implement the desired fact that "y-coordinate depends on x-coordinate", we need to use the combination of the field `y` and the method `y()` to *simulate* it. This is due to the imperative feature of Java. Field `y`, as an instance variable, once acquires a value, will evaluate to the same value each time it is evaluated. So field `y` does not "depend on" anyone in this sense. Then how can we code "y-coordinate depends on x-coordinate"? The use of an auxiliary method `y()` which depends on `x` (as desired) comes into help.

From the execution script of `MPP`, we can clearly see that submitting the "constrained" point `p2na` to the function `ms` causes a run-time bug, which demonstrates that the type `NCP2a` of `p2na` should *not* be regarded as a subtype of the type `P` although `NCP2a` is inherited (indirectly) from `P`. Considering that all `ms(p1c)`, `ms(p2c)`, `ms(p2n)` work fine and all the classes (types) of the three objects `p1c`, `p2c`, `p2n` are inherited (indirectly) from `P` too, we need to distinguish (all) inheritances in Java so that some inheritances (e.g., those as `CP`, `CP2`, and `NCP2`) may imply subtyping and others (e.g., those as `NCP2a`) do not. This can be done by using object interdependency as a measurement. Unfortunately, Java thinks "all inheritance is subtyping". What is more interesting is that due to the way in which Java handles `NaN` (Any arithmetic operation involving `NaN` and other operands produces a `NaN`, but any relational operation involving `NaN` and other operands produces either true or false[3].), this run-time bug can become hidden and difficult to find if the relevant expression is (deeply) involved with other computations. `MPP1` is such an example; by just examining the execution script of `MPP1`, it is hard to tell that `ms(p2na)` has actually caused a run-time bug.

[3]There are other means in Java to make the "illegal value" `NaN` legal, e.g., `(int)(Math.asin(2))` evaluates to 0, which could also help to conceal the `NaN` run-time bugs.

```
// class P. Note that this is also class
// NCP since NCP is defined as P.
public class P {
  protected  double x = 0.5;

  public double getx()
  { return x; }

  public void mvx(double i)
  {  x = x+i; }

  public double dist()
  { return getx();}
}


// class CP, inherited from P
public class CP extends P { String clr = "blue";
  public CP()
  { x = 2.0;}
}


// class CP2, inherited from CP
public class CP2 extends CP {
   protected double y;

   public CP2()
   { y = 2.0;}

   public double gety()
   { return y; }

   public void mvy(double i)
   { y = y+i; }

   public double dist()
   { return Math.sqrt(getx()*getx() + gety()*gety());}
}


// class NCP2, inherited from P
public class NCP2 extends P {
   protected double y;

   public NCP2()
   { y = 0.3;}

   public double gety()
   { return y; }

   public void mvy(double i)
   {  y = y+i; }

   public double dist()
   { return Math.sqrt(getx()*getx() + gety()*gety());}
}


// class NCP2a, inherited from NCP2. Need the combination
// of y and y() to simulate "y depends on x".  Note that
// "y depends on x" is what we want to do, without the use
// of y(), fields x and y would be independent
 public class  NCP2a extends NCP2
 {
   public NCP2a()
   { y = y(); }          // calling y() to get
                         // value for y

   public double y()     // implementation of
   { return 1/(4*x);}    // "y depends on x"

   public double gety()
   { y = y();            // calling y() to get
     return y;           // value for y
   }
}
```

```
// Application that uses P, CP, CP2, NCP2, and NCP2a
 public class MPP {
 public static void ms(P p)
 { if (p.dist() > 1)
   {System.out.println("  This is a colored point");
    p.mvx(1);                 // move p as specified
    System.out.println("  The result is: "+Math.asin(1/p.dist()));
   }
   else
   {System.out.println("  This is a non-colored point");
    p.mvx(-0.5*p.getx());     // move p as specified
    System.out.println("  The result is:  "+Math.asin(p.dist()));
   }
 }
 public static void main(String args[])
 {  P p1n = new P();
    CP p1c = new CP();
    CP2 p2c = new CP2();
    NCP2 p2n = new NCP2();
    NCP2a p2na = new NCP2a();

    System.out.println("making call ms(p1n)..."); ms(p1n);
    System.out.println("making call ms(p1c)..."); ms(p1c);
    System.out.println("making call ms(p2n)..."); ms(p2n);
    System.out.println("making call ms(p2c)..."); ms(p2c);
    System.out.println("making call ms(p2na)..."); ms(p2na);
 }
}

// Application that uses P, CP, CP2, NCP2, and NCP2a
public class MPP1 {
   public static void ms(P p)
   {   System.out.print("  Check to see if the result > PI/4:");
       if (p.dist() > 1)
        { p.mvx(1);                  // move p as specified
          if (Math.asin(1/p.dist()) > (Math.PI)/4)
             System.out.println ("   yes");
          else
             System.out.println ("   no");
        }
       else
       {  p.mvx(-0.5*p.getx());    // move p as specified
          if (Math.asin(p.dist()) > (Math.PI)/4)
             System.out.println ("   yes");
          else
             System.out.println ("   no");
        }
   }
   public static void main(String args[])
   { // omitted, same as the part in MPP }
}
```

```
C:\MyJavaPrograms\Point\movable pt problem>java MPP making call ms(p1n)...
  This is a non-colored point
  The result is:  0.25268025514207865
making call ms(p1c)...
  This is a colored point
  The result is: 0.3398369094541219
making call ms(p2n)...
  This is a non-colored point
  The result is:  0.40118821299725976
making call ms(p2c)...
  This is a colored point
  The result is: 0.2810349015028136
making call ms(p2na)...
  This is a non-colored point
  The result is:  NaN

C:\MyJavaPrograms\Point\movable pt problem>java MPP1 making call ms(p1n)...
  Check to see if the result > PI/4:   no
making call ms(p1c)...
  Check to see if the result > PI/4:   no
making call ms(p2n)...
  Check to see if the result > PI/4:   no
making call ms(p2c)...
  Check to see if the result > PI/4:   no
making call ms(p2na)...
  Check to see if the result > PI/4:   no
```

Figure 4: Java Code of the Movable Point Problem

Summarizing what is described in this section, we can state the problem as follows:

- In OOP supported by conventional object type systems, there is no way to implement programs like ms reliably and verify its correctness.

Motivated by this problem, we propose, in the subsequent sections, a new typing scheme for objects.

3 A Simple Typed Object-Oriented Language

To illustrate our approach, we define a simple typed object-oriented language (**TOOL**) in this section.

3.1 Syntax

The terms and types of **TOOL** are defined as follows.

$$\begin{aligned}
M &::= x \mid \lambda(x{:}\sigma).M \mid M_1M_2 \mid M.l \mid M.l{\Leftarrow}\varsigma(x{:}\mathcal{S}(A))M' \\
&\quad\mid [l_i = \varsigma(x{:}\mathcal{S}(A))M_i]_{i=1}^n \\
\sigma &::= \kappa \mid t \mid \sigma_1 \to \sigma_2 \mid \mu(t)\sigma \mid A \mid \mathcal{S}(A) \\
A &::= \iota(t)[l_i(L_i){:}\sigma_i]_{i=1}^n \quad L_i \subseteq \{l_1,\dots,l_n\} \text{ for each } i
\end{aligned}$$

Terms in **TOOL** are standard λ-terms and ς-terms [2]. In particular, $[l_i = \varsigma(x : \mathcal{S}(A))M_i]_{i=1}^n$ represents an object, $M.l$ represents method invocation, and $M.l \Leftarrow \varsigma(x : \mathcal{S}(A))M'$ represents method updating.

Types in **TOOL** are standard ground type, function type, recursive type, and the newly proposed object type. In object type $\iota(t)[l_i(L_i){:}\sigma_i]_{i=1}^n$, ι is the self-type binder, each method l_i has type σ_i, and L_i is the set of *links* of l_i (defined in the next subsection). $\mathcal{S}(A)$ denotes the self type induced by the object type A. $A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^n$ if and only if $A = [l_i(L_i){:}\sigma_i(\mathcal{S}(A))]_{i=1}^n$.

We provide a simple example to illustrate the syntax of types and terms. Let

$$A \stackrel{\text{def}}{=} \iota(t)\begin{bmatrix} l_1(\{l_2, l_3\}) : t \\ l_2(\emptyset) : int \\ l_3(\{l_2\}) : int \to int \end{bmatrix}.$$

It specifies that l_1, l_2, and l_3 are of self type (associated with A), int, and $int \to int$ respectively. The sets of links for l_1 and l_3 are $\{l_2, l_3\}$ and $\{l_2\}$. l_2 has no links. An object of type A could be

$$a \stackrel{\text{def}}{=} \begin{bmatrix} l_1 = \varsigma(s{:}\mathcal{S}(A))s \\ l_2 = 1 \\ l_3 = \varsigma(s{:}\mathcal{S}(A))\lambda(x{:}int)(x + s.l_2) \end{bmatrix}.$$

3.2 Definition of Links

Links are used to signify the structure of component dependency of objects. Informally, in object type $\iota(t)[l_i(L_i) : \sigma_i]_{i=1}^n$, $l_j \in L_i$ means that the value of method l_i depends (partially) on the value of method l_j. The link mechanism makes the types of objects in **TOOL** substantially different from that in conventional object type systems.

Definition 1 (Link) Given an object $a = [l_i = \varsigma(s : \mathcal{S}(A))M_i]_{i=1}^n$, (1) l_i is said to be ***dependent*** on $l_j (i \neq j)$ if there exists a M such that $a.l_i$ and $(a.l_j \Leftarrow \varsigma(s : S(A))M).l_i$ evaluate to different values; (2) l_i is said to be ***directly dependent*** on $l_j (i \neq j)$ if (a) l_i is dependent on l_j, and (b) if all such $l_k (i \neq k, j \neq k)$ where l_i is dependent on l_k and l_k is dependent on l_j, are removed from a, l_i is still dependent on l_j; (3) The set of ***links*** of l_i (or equivalently, of M_i with respect to object a), denoted by $L(l_i)$ (or equivalently, by $L_a(M_i)$), contains exactly all such l_j on which l_i is directly dependent.

Example 1 Take the object a and its type A defined at the end of section 3.1, by the definition of links, we see that the links of the methods in a are:

$$\begin{array}{l} L(l_1) = L_a(s) = \{l_2, l_3\} \\ L(l_2) = L_a(1) = \emptyset \\ L(l_3) = L_a(\lambda(x\!:\!int)(x + s.l_2)) = \{l_2\} \end{array}$$

which match the corresponding link specifications in type A.

4 Object Type Graphs

4.1 Definitions

To reveal the structure of object component interdependencies more clearly and facilitate the study of object subtyping and behaviors, we introduce a graphical representation of object types – object type graphs. We define directed colored graphs first.

Definition 2 (Directed Colored Graph) A ***directed colored graph*** G is a 6-tuple (G_N, G_A, C, sr, tg, c) consisting of: (1) a set of ***nodes*** G_N, and a set of ***arcs*** G_A; (2) a ***color alphabet*** C; (3) a ***source map*** $sr : G_A \to G_N$, and a ***target map*** $tg : G_A \to G_N$, which return the source node and target node of an arc, respectively; and (4) a ***color map*** $c : G_N \cup G_A \to C$, which returns the color of a node or an arc.

Definition 3 (Ground Type Graph) A ***ground type graph*** is a single-node colored directed graph which is colored by a ground type.

Definition 4 (Function Type Graph) A ***function type graph*** $(s, G_1, G_2)_{(G_N, G_A, C, sr, tg, c)}$ is a directed colored graph consisting exactly of a ***starting node*** $s \in G_N$, and two type graphs G_1 and G_2, such that, (1) $c(s) = \to$; (2) there are two arcs associated with the starting node s, *left* arc $l \in G_A$ and *right* arc $r \in G_A$, such that $c(l) = in$, $c(r) = out$;

l connects G_1 to s by $sr(l) = s_{G_1}$, $tg(l) = s$, and r connects s to G_2 by $sr(r) = s$, $tg(r) = s_{G_2}$, where s_{G_1} and s_{G_2} are the starting nodes of G_1 and G_2, respectively; (3) G_1 and G_2 are disjoint; (4) if there is an arc $a \in G_A$ with $c(a) = rec$, then $sr(a) = s_{G_i}$, $tg(a) = s$, $c(s_{G_i}) = \rightarrow$, $i = 1, 2$.

Definition 5 (Object Type Graph) An ***object type graph*** $(s, A, R, L, S)_{(G_N, G_A, C, sr, tg, c)}$ is a directed colored graph consisting exactly of a ***starting node*** $s \in G_N$, a set of ***method arcs*** $A \subseteq G_A$, a set of rec-colored arcs $R \subseteq G_A$, a set of ***link arcs*** $L \subseteq G_A$, and a set of type graphs S, such that (1) $c(s) = self$. (2) $\forall a \in A$, $sr(a) = s$, $tg(a) = s_F$ for some type graph $F \in S$, and $c(a) = m$ for some method label m; $c(a) \neq c(b)$ for $a, b \in A$, $a \neq b$. (3) $\forall r \in R$, $c(r) = rec$, $tg(r) = s$, $sr(r) = s_F$ for some $F \in S$, and $c(s_F) = self$. (4) $\forall l \in L$, $sr(l) = s_F$, $tg(l) = s_G$ for some $F, G \in S$, and $c(l) = bym$ for some method label m.

Remarks: Directed colored graph is the foundation of graph grammar theory [10, 11, 12, 13, 22]. Object type graphs are adapted from directed colored graphs. Ground type graphs are trivial. Function type graphs are straightforward. They need to be defined because an object type graph may include them as subgraphs. An object type graph is formed by a starting node s and a set S of type graphs with each $F \in S$ being connected to s by a method arc that goes from s to F. The starting node s is colored by *self* and is used to denote the self type. The method interdependencies are specified by arcs in L. If $L(m)$ is the set of links of method m, then for each $l \in L(m)$ there is an arc (colored by *byl*) that goes from l to m. Recursive object types are specially indicated by rec-colored arcs in R.

For the sake of brevity, we drop the subscripts in $(s, G_1, G_2)_{(G_N, G_A, C, sr, tg, c)}$ and $(s, A, R, L, S)_{(G_N, G_A, C, sr, tg, c)}$ whenever possible throughout the paper.

4.2 Examples of Object Type Graphs

We now provide some examples to illustrate the definition of object type graphs.

Example 2 In Figure 5, A, B, and C are the type graphs for ground types *int*, *real*, and *bool* respectively. D is the type graph for function type $int \rightarrow int$ and E is the type graph for $(int \rightarrow real) \rightarrow (real \rightarrow int)$.

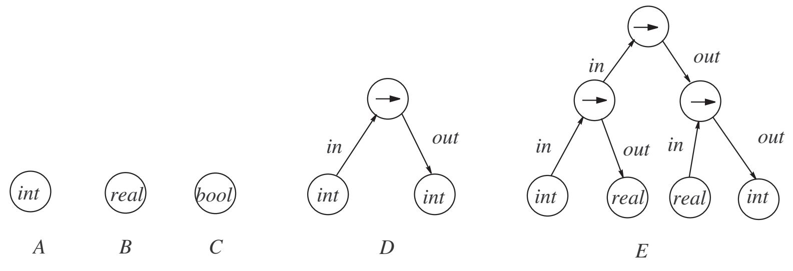


Figure 5: Examples of Ground Type Graphs and Function Type Graphs

Example 3 In Figure 6, graph A denotes the object type $[x\!:\!int, y\!:\!int]$, where methods x and y are independent of each other. Graph B denotes the type $[x\!:\!int, y(\{x\})\!:\!int]$ where y depends on x. Note that the direction of the link arc in B is from x to y, (not from y to x), signifying the fact that changes made to method x will affect method y.

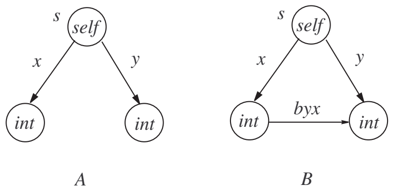


Figure 6: Examples of Object Type Graphs

Example 4 In Figure 7, graph C represents the object type $\mu(t)\iota(s)[a : int, b : t, c : s]$. Method a is of type int; method b is of recursive object type C. Method c is of the self type induced by the object type C. Note the structural difference between the type of b and the type of c revealed in the type graph[4]. Graph D represents the type of a simplified 1-d movable point $[x = 1, mvx = \varsigma(s\!:\!\mathcal{S}(D))\lambda(i\!:\!int)(s.x \Leftarrow s.x + i)]$. The facts that mvx depends on x and returns a modified self are indicated by the byx-colored arc and the out-colored arc in D.

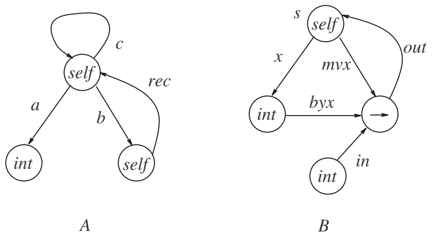


Figure 7: Examples of Object Type Graphs

Example 5 Two more object type graphs are shown in Figure 8. They are the types of some variations of point objects. Graph A is the type of the object

$$\begin{bmatrix} x = 1, \\ m_1 = \varsigma(s\!:\!\mathcal{S}(A))\lambda(i\!:\!int)p \\ m_2 = \varsigma(s\!:\!\mathcal{S}(A))\lambda(i\!:\!int)s \end{bmatrix}$$

[4]This structural setting, potentially, will allow the type of c to remain as self type and the type of b to be changed after some operations on graph C are performed.

where p is some point object of type A. Graph B is the type of the object

$$\begin{bmatrix} x = 1 \\ y = 2 \\ d = \varsigma(s{:}\mathcal{S}(B))(s.x + s.y)/2 \\ e_1 = \varsigma(s{:}\mathcal{S}(B))\lambda(p{:}B)(p.x = s.x \ \wedge p.y = s.y) \\ e_2 = \varsigma(s{:}\mathcal{S}(B))\lambda(p{:}\mathcal{S}(B))(p.x = s.x \ \wedge p.y = s.y) \end{bmatrix}.$$

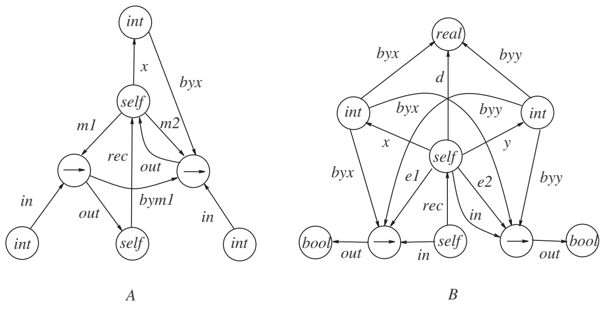


Figure 8: Examples of Object Type Graphs

5 Object Typing/Subtyping Under OTG

We now investigate the issue of typing/subtyping under OTG. We first define object subtyping through a series of definitions and then present the typing/subtyping rules with a brief discussion. Note that OTG is just another way (a graphical way, specifically) to represent object types. There is a natural 1-1 correspondence between OTG and the normal textual representations of object types in **TOOL**. So the typing rules presented in this section naturally apply to object type graphs. What makes OTG significant is its facilitation of the formulation of object subtyping with the presence of links in object types (as addressed below).

Definition 6 (Type Graph Premorphism) Let $\mathbf{\Phi}$ be the set of ground types. Given two type graphs $G = (G_N, G_A, C, sr, tg, c)$ and $G' = (G'_N, G'_A, C', sr', tg', c')$, a ***type graph premorphism*** $f : G \to G'$ is a pair of maps $(f_N : G_N \to G'_N, f_A : G_A \to G'_A)$, such that (1) $\forall a \in G_A$, $f_N(sr(a)) = sr'(f_A(a))$, $f_N(tg(a)) = tg'(f_A(a))$, and $c(a) = c'(f_A(a))$; (2) $\forall v \in G_N$, if $c(v) \in \mathbf{\Phi}$, then $c'(f_N(v)) \in \mathbf{\Phi}$; otherwise $c(v) = c'(f_N(v))$.

Definition 7 (Base, Subbase) Given an object type graph $G = (s, A, R, L, S)$. The ***base*** of G, denoted by $Ba(G)$, is the graph $(s, A, t(A), L)$, where $t(A) = \{tg(a) \mid a \in A\}$. A ***subbase*** of G is a subgraph $(s, A', t(A'), L')$ of $Ba(G)$, where $A' \subseteq A$, $L' \subseteq L$, $t(A') = \{tg(a) \mid a \in A'\}$, and for each $l \in L'$ there exist $a_1, a_2 \in A'$ such that $sr(l) = tg(a_1)$ and $tg(l) = tg(a_2)$.

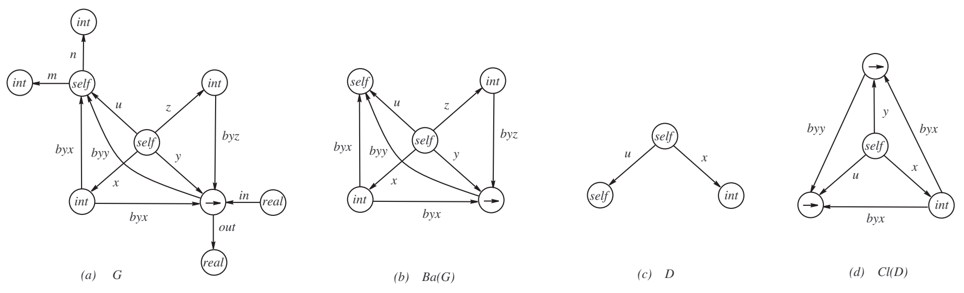


Figure 9: (a) An object type graph G; (b) The base $Ba(G)$ of G;
(c) A subbase D of G; (d) The closure $Cl(D)$

Definition 8 (Closure, Closed) The ***closure*** of a subbase $D = (s, A', t(A'), L')$ of an object type graph $G = (s, A, R, L, S)$, denoted by $Cl(D)$, is the union $D \cup E_1 \cup E_2$, where (1) $E_1 = \{l \in L \mid \exists a_1, a_2 \in A'$ with $tg(a_1) = sr(l),\ tg(a_2) = tg(l)\}$, and (2) $E_2 = \{l, h, a, t(l) \mid l, h \in L,\ a \in A,\ a \notin A',\ tg(l) = sr(h) = tg(a)$, and $\exists a_1, a_2 \in A'$ such that $tg(a_1) = sr(l),\ tg(a_2) = tg(h)\}$. A subbase D is said to be ***closed*** if $D = Cl(D)$.

Definition 9 (Covariant, Invariant) Given an object type graph (s, A, R, L, S). Let $t(A) = \{tg(a) \mid a \in A\}$. For each $v \in t(A)$, if v is not incident with any links, or if v is the target node of some links but not the source node of any links, then v is said to be ***covariant***; otherwise, v is said to be ***invariant***.

Definition 10 (Object Subtyping) Given two object type graphs $G = (s_G, A_G, \emptyset, L_G, S_G)$ and $F = (s_F, A_F, \emptyset, L_F, S_F)$. $F <: G$ if and only if the following conditions are satisfied: (1) There exists a premorphism f from $Ba(G)$ to $Ba(F)$ such that $f(Ba(G)) = Cl(f(Ba(G)))$. That is, $f(Ba(G))$ is closed. (2) For each node v in $f(Ba(G))$, let u be its preimage in $Ba(G)$ under f, $F_v \in S_F$ be the type graph with v as its starting node, and $G_u \in S_G$ be the type graph with u as its starting node. (i) If v is invariant, then $F_v \cong G_u$. (ii) If v is covariant, then $F_v <: G_u$.

Remarks: Type graph premorphism is adapted from graph morphism which is a fundamental concept in algebraic graph grammars [13, 10, 22, 11, 12]. It preserves the directions and colors of arcs and the colors of nodes up to ground types. The base of an object type graph singles the method interdependency information out of the entire object type graph so that the structure of the method interdependencies can be better studied. The closure of a subbase captures the complete behavior of the subbase by including, in addition to all methods and links in the subbase, a set E_2 of methods (and associated links) outside of the subbase in the following way: for any method l in E_2, (1) l depends on some methods inside the subbase, and (2) there exist some methods inside the subbase that depend on l. An example of base, subbase, and closure is shown in Figure 9. Object subtyping is defined using the ideas of type graph premorphism, base, subbase, closure, and variance property. It first ensures that the behavior of a subobject (indicated by method interdependencies) is the same as that of a superobject through the closure requirement.

Then, it uses the variance information of each method to check the subtyping feasibility of each method type (graph) in a subobject with its counterpart in a superobject[5]. Note that in the definition of object subtyping, we only consider the case $R = \emptyset$ (i.e., no recursive object types). The case $R \neq \emptyset$ requires complicated graph grammar operations and is beyond the scope of this paper.

The typing/subtyping rules of **TOOL** are shown in Table 1. The rules that are affected by links are (TObj) and (TUpd). Note that in these rules, the set of links computed from terms are checked against the set of links specified in types.

$$\frac{}{\emptyset \rhd \diamond}(\mathrm{TC}\emptyset) \qquad \frac{\Gamma \rhd \sigma \quad x \notin dom(\Gamma)}{\Gamma, x\!:\!\sigma \rhd \diamond}(\mathrm{TCVar}) \qquad \frac{\Gamma \rhd M : \sigma \quad x \notin dom(\Gamma)}{\Gamma, x\!:\!\tau \rhd M : \sigma}(\mathrm{T}x)$$

$$\frac{\Gamma \rhd \diamond}{\Gamma \rhd \kappa}(\mathrm{TyCons}) \qquad \frac{\Gamma \rhd \sigma \quad \Gamma \rhd \tau}{\Gamma \rhd \sigma \to \tau}(\mathrm{TyFun})$$

$$\frac{\Gamma \rhd \sigma_i \quad \forall i \in \{1, \ldots, n\}}{\Gamma \rhd \iota(t)[l_i(L_i)\!:\!\sigma_i(t)]_{i=1}^{n}}(\mathrm{TyObj},\ L_i \subseteq \{l_1, \ldots, l_n\} \text{ for each } i)$$

$$\frac{\Gamma \rhd \diamond \quad x\!:\!\sigma \in \Gamma}{\Gamma \rhd x\!:\!\sigma}(\mathrm{TVar}) \qquad \frac{\Gamma, x\!:\!\sigma \rhd M\!:\!\tau}{\Gamma \rhd \lambda(x\!:\!\sigma).M : \sigma \to \tau}(\mathrm{TAbs}) \qquad \frac{\Gamma \rhd M\!:\!\sigma \to \tau \quad \Gamma \rhd N\!:\!\sigma}{\Gamma \rhd MN : \tau}(\mathrm{TApp})$$

$$\frac{\Gamma, s\!:\!\mathcal{S}(A) \rhd M_i\!:\!\sigma_i \quad L_i = L_a(M_i) \quad \forall i \in \{1, \ldots, n\}}{\Gamma \rhd a : A}\left(\mathrm{TObj},\ \begin{array}{l} a = [l_i = \varsigma(s\!:\!\mathcal{S}(A))M_i]_{i=1}^{n} \\ A = \iota(t)[l_i(L_i)\!:\!\sigma_i(t)]_{i=1}^{n} \end{array}\right)$$

$$\frac{\Gamma \rhd M : A \quad j \in \{1, \ldots, n\}}{\Gamma \rhd M.l_j : \sigma_j(A)}(\mathrm{TInv1},\ A = \iota(t)[l_i(L_i)\!:\!\sigma_i(t)]_{i=1}^{n} = [l_i(L_i)\!:\!\sigma_i(\mathcal{S}(A))]_{i=1}^{n})$$

$$\frac{\Gamma \rhd s : \mathcal{S}(A) \quad j \in \{1, \ldots, n\}}{\Gamma \rhd s.l_j : \sigma_j(A)}(\mathrm{TInv2},\ A = \iota(t)[l_i(L_i)\!:\!\sigma_i(t)]_{i=1}^{n} = [l_i(L_i)\!:\!\sigma_i(\mathcal{S}(A))]_{i=1}^{n})$$

$$\frac{\Gamma \rhd N\!:\!A \quad \Gamma, s\!:\!\mathcal{S}(A) \rhd M\!:\!\sigma_i \quad L_i = L_N(M) \quad i \in \{1, \ldots, n\}}{\Gamma \rhd N.l_i \Leftarrow \varsigma(s\!:\!\mathcal{S}(A))M : A}(\mathrm{TUpd},\ A = \iota(t)[l_i(L_i)\!:\!\sigma_i(t)]_{i=1}^{n})$$

$$\frac{\Gamma \rhd \sigma}{\Gamma \rhd \sigma <: \sigma}(\mathrm{SRefl}) \qquad \frac{\Gamma \rhd \sigma <: \tau \quad \Gamma \rhd \tau <: \delta}{\Gamma \rhd \sigma <: \delta}(\mathrm{STran}) \qquad \frac{\Gamma \rhd a\!:\!A \quad \Gamma \rhd A <: B}{\Gamma \rhd a\!:\!B}(\mathrm{SSump})$$

$$\frac{\Gamma \rhd \sigma' <: \sigma \quad \Gamma \rhd \tau <: \tau'}{\Gamma \rhd \sigma \to \tau <: \sigma' \to \tau'}(\mathrm{SFun})$$

$$\frac{\Gamma \rhd G_A <: G_B}{\Gamma \rhd A <: B}\left(\mathrm{SObj},\ \begin{array}{l} G_A \text{ and } G_B \text{ are the OTGs of } A \text{ and } B \text{ respectively} \\ A = \iota(t)[l_i(L_i)\!:\!\sigma_i(t)]_{i=1}^{n}, B = \iota(t)[l'_i(L'_i)\!:\!\sigma'_i(t)]_{i=1}^{n'} \end{array}\right)$$

Table 1: Typing and Subtyping Rules for **TOOL**

We would like to emphasize that the purpose of object type graphs is to facilitate the formulation and reasoning of object subtyping when method interdependencies are considered in object types. This can be seen in the object subtyping rule (SObj) where the determination of $A <: B$ for object types A and B depends on whether their object type graphs G_A and G_B have a subtyping relationship which, in turn, can be decided by

[5]Ground subtyping and function subtyping which are involved in object subtyping are standard as in the literature.

the Definition 10. (Definition 10 suggests an immediate algorithm for how to compute $G_A <: G_B$.)

6 Verification of the Program *ms* under OTG

We have shown, in section 2, that under conventional object type systems, there is no way to code the function ms satisfactorily in the sense that we are unable to prove that ms performs to its specification for all permissible arguments. In this section, we show that this problem can be easily resolved under OTG typing/subtyping. That is, we show that ms can be coded reliably under OTG typing/subtyping and prove that it performs to its specification in all situations.

Given the code of ms in Figure 3 and under the OTG notation, the type of the point p_{1n} (which is also the type of the parameter in the function ms) and the type of the point p'_{2n} are depicted as P and Q'_{2n} in Figure 10[6]. Let f be the premorphism from base $Ba(P)$ to base $Ba(Q'_{2n})$, $f(Ba(P))$ and its closure $Cl(f(Ba(P)))$ are also shown in Figure 10. By the OTG object subtyping definition (Definition 10), we can see that $Q'_{2n} \not<: P$ because $f(Ba(P)) \neq Cl(f(Ba(P)))$ (i.e., $f(Ba(P))$ is not closed). Hence, p'_{2n} cannot be viewed as having type P and $ms(p'_{2n})$ does not type-check. The run-time error of $ms(p'_{2n})$ is therefore prevented by type checking at compile-time. Hence, the code of ms in Figure 3 is safe under the OTG typing/subtyping.

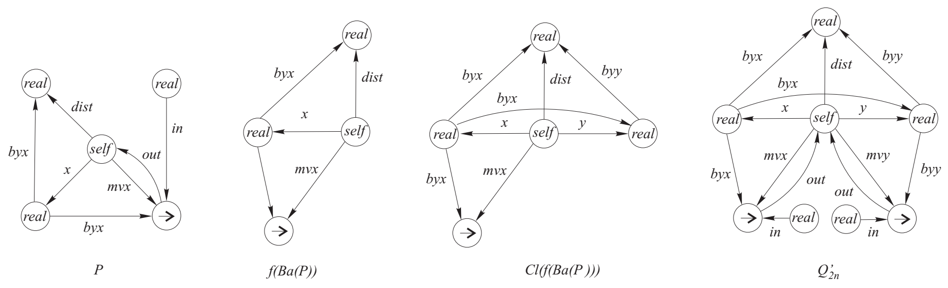


Figure 10: Resolution of the Movable Point Problem in OTG

To fully revisit of the movable point problem in the context of OTG, the type graphs of p_{1c}, p_{2c}, and p_{2n} are depicted in Figure 11 as Q_{1c}, Q_{2c}, and Q_{2n}, respectively. We can easily check, using Definition 10, that $Q_{1c} <: P$, $Q_{2c} <: P$, and $Q_{2n} <: P$ all hold. This shows that the desired executions $ms(p_{1c})$, $ms(p_{2c})$, and $ms(p_{2n})$ are all supported by OTG typing/subtyping scheme.

From Figure 10 and Figure 11, we see that the type of p'_{2n} and the type of p_{2n} are different under OTG (as opposed to the same in conventional type systems). The fact that method y depends on method x in p'_{2n} and method y does not depend on method x in p_{2n} (i.e., p_{2n} and p'_{2n} have different behaviors) is faithfully captured in their type graphs

[6]For the sake of conciseness, some unimportant links that do not affect the result of illustration, such as the link from method *dist* to method *mvx*, are not shown in Figure 10.

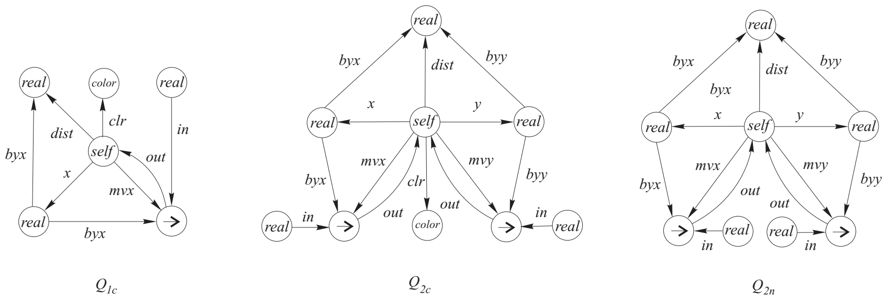


Figure 11: Types of p_{1c}, p_{2c}, and p_{2n} in OTG

as the presence/absence of a link from method x to method y. Indeed, this distinction is necessary in order to prevent run-time errors such as those caused by $ms(p'_{2n})$. This observation leads to the following proposition.

Proposition 1 *Let A be the type of an object a in which there is a link between method x and method y. Let B be the type of an object b which is modified from a by deleting the link between method x and method y. Then $A \neq B$.*

Also note that in Figure 10 and Figure 11, we have $Q'_{2n} \not<: Q_{2n}$ (we can easily verify this by Definition 10). This disallowance of subtyping is also necessary in order to statically prevent similar run-time errors caused by $ms(p'_{2n})$. Thus,

Proposition 2 *Let A and B be as specified in Proposition 1. Then $A \not<: B$.*

We now show the correctness of ms in Figure 3 under the OTG typing scheme. We assume that all arguments (1-d points, 2-d points, ...) submitted to ms are "correctly" coded. In particular, if p is an n-dimensional point with coordinates $x_1, \ldots, x_n$, then its method *dist* must have $\sqrt{x_1^2 + \cdots + x_n^2}$ as the body; and its method *mvx* must have $\lambda(i\!:\!real)s.x \Leftarrow (s.x + i)$ as the body; how other methods in p are coded is irrelevant to the proof. This is a reasonable assumption, for if p is coded "incorrectly" or arbitrarily (say, p's *dist* body is $\sqrt{x_1^2 + 4x_2^2 + \cdots + n^2 x_n^2}$), then there would be no way to expect what kind of behavior ms can have with p as its argument.

To facilitate the proof, we rewrite the functional program ms in Figure 3 equivalently into an imperative one in Figure 12, where a holds the computation result. We would like to prove, under the framework of Hoare logic (e.g. [16, 17]), that the two Hoare triples

$$(\!|p.dist > 1 \wedge p\!:\!P|\!)ms(p)(\!|p.dist > 1 \wedge p\!:\!P|\!)$$
$$(\!|p.dist \leq 1 \wedge p\!:\!P|\!)ms(p)(\!|p.dist \leq 1 \wedge p\!:\!P|\!)$$

are valid for any point p of type P in Figure 10. The first triple specifies that ms keeps a colored point in the colored point area after moving it. The second triple specifies that ms keeps a non-colored point in the non-colored point area after moving it. Before proving the validity of the triples, we prove a lemma first. Let colored points and non-colored points be defined as in section 2, we can show that

```
$ms \overset{\text{def}}{=} fun(p : P)$ {
    $real\ a$;
    **if** ($p.dist > 1$){
        $p.mvx(\delta)$;    // $\delta > 0$
        $a = sin^{-1}(1/p.dist)$;
    }
    **else** {
        $p.mvx(-\frac{1}{2}p.x)$;
        $a = sin^{-1}(p.dist)$;
    }
}
```

Figure 12: The Imperative Version of the Program *ms*.

Lemma 1 *Given an n-dimensional point p, if p is a non-colored point and is of type P in Figure 10, then after being moved, along the x-axis and towards the origin, half of the projection of the distance from the origin to p's current position over the x-axis, p is still in the non-colored point area in the space.*

Proof: Without loss of generality, we assume that the coordinates of p are $x_1, x_2, \ldots, x_n$ ($n > 1$) with x_1 being the x-coordinate, x_2 being the y-coordinate, ... Since p is a non-colored point, we have $\sqrt{x_1^2 + \cdots + x_n^2} \leq 1$. After p is moved as specified, its x-coordinate would be changed to $\frac{1}{2}x_1$. Since p is n-dimensional and $n > 1$, the actual type of p must be a subtype of P. By the definition of OTG subtyping (Definition 10), we know that the x-coordinate change of p will *not* affect any other coordinates $x_2, \cdots, x_n$ of p because all $x_2, \cdots, x_n$ occur in the method *dist* of p and *dist* appears in type P[7]. Thus, $x_2, \ldots, x_n$ all retain their old values after p's move. Therefore, the distance from the origin to the new position of p is $\sqrt{(\frac{1}{2}x_1)^2 + x_2^2 + \cdots + x_n^2} < \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \leq 1$, indicating the p is still in the non-colored point area. □

The validity of the second Hoare triple is given in Theorem 1 below. The proof of the first Hoare triple is similar and omitted.

Theorem 1 *Given the program ms in Figure 12, the Hoare triple*

$$(\!|p.dist \leq 1 \wedge p{:}P|\!)ms(p)(\!|p.dist \leq 1 \wedge p{:}P|\!)$$

is valid.

Proof: The proof, shown in Figure 13, is an application of the standard imperative program verification rules (see e.g. [17]). In Figure 13, *p.d* and *p.m* stand for *p.dist* and *p.mvx*, and A, B, C, D, E, F, G stand for the following triples respectively:

[7]Here is a subtle point indicated by the OTG object subtyping: if any of the coordinates $x_2, \ldots, x_n$, say x_i, does not occur in method *dist* (or in any other method included in type P), then we allow x_i be affected by the changes of x_1 while requiring that the type of p is a subtype of type P.

$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis \{p.m(-\frac{1}{2}p.x)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis \{a = sin^{\text{-}1}(p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis \perp \rrparenthesis \{p.m(\delta); a = sin^{\text{-}1}(1/p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis \{p.m(-\frac{1}{2}p.x); a = sin^{\text{-}1}(p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \wedge p.d > 1 \rrparenthesis \{p.m(\delta); a = sin^{\text{-}1}(1/p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \wedge p.d \leq 1 \rrparenthesis \{p.m(-\frac{1}{2}p.x); a = sin^{\text{-}1}(p.d)\} \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$,
$\llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis ms(p) \llparenthesis p.d \leq 1 \wedge p : P \rrparenthesis$.
The validity of triple A on the top of the proof tree is provided by Lemma 1. □

$$\dfrac{\dfrac{C}{E}\text{(implication)} \qquad \dfrac{\dfrac{\dfrac{}{A}\text{(Lemma 1)} \quad \dfrac{}{B}\text{(assignment)}}{D}\text{(composition)}}{F}\text{(implication)}}{G}\text{(if-statement)}$$

Figure 13: The Proof of ms's Property

7 Conclusion and Future Work

Typing is an efficient means in program verifications. Object component interdependency information is critical in determining and predicting object behaviors and in shaping object types. If this information is not captured in object typing, as is the case in conventional object type systems, then a statically well-typed program may go wrong at run-time causing run-time errors and program verification troubles. We proposed object type graphs (OTG) as an initial treatment for handling object component interdependencies in object typing and program verifications. We have seen that due to OTG's ability of revealing more information about object behaviors,

- Programs that go wrong at run-time in conventional object type systems can be effectively detected at compile-time under OTG typing/subtyping.
- Program verifications that cannot be done with conventional object type systems can be easily carried out with the support of OTG typing/subtyping.

This demonstrates that OTG is a safer typing scheme than conventional ones, and provides a valuable support for OOP program verifications. The following issues are of immediate interests for future work:

- Devise a link computation algorithm and assess its complexity.
- Prove/disprove that the standard properties of type systems, such as subject reduction and soundness, hold under OTG.
- As far as applying the idea of OTG to practical object-oriented languages is concerned, we believe that a direct approach would be to adapt OCaml [1] by modifying its type for classes. Influenced by OOP theory research, Ocaml, unlike other object-oriented languages (e.g. Java) where classes are the sole type of objects, gives a type

for each of its classes. In a sense, the type of a class in OCaml is the (more abstract) type of the object generated by that class. This is a typical case where practice benefits from theory, and it would be very interesting to keep extending OCaml along this line.

References

[1] *http://caml.inria.fr/ocaml/.* 2007.

[2] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1996.

[3] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping for Extensible, Incompete objects. *Fundamenta Informaticae*, 38(4):325–364, 1999.

[4] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell lambda calculus of objects. In *Proc. of International Conference of Computer Science Logic*, number 933 in LNCS, pages 16–30. 1995.

[5] K. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[6] K. Bruce. *Foundations of Object-Oriented Languages.* MIT Press, 2002.

[7] K. Bruce, A. Schuett, R. van Gent, and A. Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.

[8] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of the 17the Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[9] R. Deline and M. Fahndrich. Typestates for objects. In *ECOOP 2004*, 2004.

[10] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Applications to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, 1978.

[11] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, 1999.

[12] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3. World Scientific, 1999.

[13] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conference of Automata and Switching Theory*, pages 167–180, 1973.

[14] K. Fisher, F. Honsell, and J. Mitchell. A lambda calculus of objects and method specialization. *Nodic Journal of Computing*, 1:3–37, 1994.

[15] J. Hickey. *Introduction to OCaml, http://caml.inria.fr/tutorials-eng.html.* 2002.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[17] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge University Press, 2nd edition, 2004.

[18] L. Liquori. On object extension. In *ECOOP'98 Object-oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 498–522. Springer–Verlag, 1998.

[19] L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. Number 1179 in Lecture Notes in Computer Science, pages 129–141. Springer–Verlag, 1996.

[20] B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[21] O. L. Madsen. Towards Integration of State Machines and Object-Oriented Languages. In *Technology of Object-Oriented Languages and Systems (TOOLS Europe'99)*, 1999.

[22] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.

Informatics Europe

Informatics Europe (*http://www.informatics-europe.org*) is the association of computer science/IT/informatics departments of universities and research organizations, public and private, in Europe and neighboring areas.

The mission of Informatics Europe is to foster the development of quality research and teaching in information and computer sciences.

Informatics Europe was created as a result of the first two European Computer Science Summits (ECSS), held at ETH Zurich in October 2005 and October 2006, where heads of computer science departments from all over the European region joined forces for the first time to define and promote common policies and study common issues.

Informatics Europe is a nonprofit membership organization; members are organizations such as CS departments of universities as well as public or private research laboratories. Informatics Europe maintains close ties with other academic and professional organizations.

The association pursues its goals through meetings, working groups, newsletters and other activities. Currently five working groups are active, each with a mailing list and a Wiki page:

- Evaluation criteria for informatics research (what are the appropriate measures for evaluating the work of researchers in computer science and information technology?)
- Curriculum issues (what is an appropriate informatics curriculum, how do we assess equivalences for exchange students etc.?)
- Facts and figures (collecting the basic data about informatics in Europe, from the mere list of departments to bachelor/master/PhD graduation figures, faculty salaries etc.).
- Lobbying and strategy (making our voice heard by political authorities and others).

Informatics Europe also publishes the weekly "Tech Watch Digest", a concise summary of the latest development in the field, with a European accent. Subscription is free and open to anyone; see *http://www.informatics-europe.org/techwatch.html.*

The next annual meeting of Informatics Europe, the European Computer Science Summit 2007, will take place in Berlin on 8–9 October 2007.

Informatics Europe is currently building up its membership; major universities from across the region have already joined. Membership is for the calendar year and covers access to all activities of Informatics Europe; it is the opportunity to engage in contacts with many colleagues facing the same issues, learn from their experience, and help the recognition and progress of informatics in Europe.

This note was received from
prof. Bertrand Meyer, ETH Zurich, Switzerland

The Short Story of SDC Wrocław – Two Software Development Centers at the Oder River

Siemens Software Development Center was established in Wroclaw in the year 2000. As a part of Siemens' former Information and Communications Group, SDC was set up as one of many Siemens Research and Development centers in the world.

Back in 2000, SDC employed only 10 people working on customizing core mobile network software to customers' specific requirements. Since then, the task list has significantly expanded and SDC became a leading and one of the largest R&D centers in Poland as well as a major R&D unit within Siemens Communications branch. In 2005 SDC already employed over 700 people.

A year later, in October 2006 SDC was divided into two parts as a result of Siemens AG management's decision to merge Siemens' and Nokia's telecommunications businesses within Nokia Siemens Networks joint venture, what is planned for the first quarter of 2007. As a result of these decisions and as a preparatory step for the merger, a separate unit called Siemens Networks was established, comprising mobile networks, fixed networks and carrier services departments of the former Siemens Communications branch.

Thus Siemens Networks Software Development Center (Siemens Networks SDC) was carved out of SDC. Siemens Networks SDC employs currently 670 specialists and is a part of Siemens Networks Sp. z.o.o. It develops solutions and applications in the latest technologies: 3G Technology (HSDPA, HSUPA, UMTS), GSM, GPRS, IN Services, Wimax.

The part of Software Development Center which remained in Siemens Sp. z o.o. is now Siemens Development Center (SDC). SDC is a software house that provides complete software engineering services, application management and professional services for Siemens AG and Siemens Group companies. The Center continuously moves into new areas of activity and grows quickly. Siemens Development Center at the moment employs more than 250 people.

e-Informatica

ISBN 987-83-7493-305-6