I, Venkata Rama Chaitra Thota, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:
Risk as a Mechanism in Self-Organizing Agile Software Development Teams

Student's name: Venkata Rama Chaitra Thota

This work and its defense approved by:

Committee chair: Nan Niu, Ph.D.
Committee member: Raj Bhatnagar, Ph.D.
Committee member: Carla Purdy, Ph.D.
RISK AS A MECHANISM IN
SELF-ORGANIZING AGILE SOFTWARE
DEVELOPMENT TEAMS

A thesis submitted to the
graduate school
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

Master of Science

in the Department of Electrical Engineering and Computer Science
of the College of Engineering and Applied Sciences

by

VENKATA RAMA CHAITRA THOTA
M.S University of Cincinnati
July 2017

Committee Chair: Dr. Nan Niu, Ph.D.
Abstract

Risk—the possibility of an unsatisfactory outcome—is an essential vehicle for a software development project to progress. Risk in itself is neutral; it is essential to progress, and failure is often a key part of learning. However, we must learn to balance the possible negative consequences of risk against the potential benefits of its associated opportunity. Risk management has traditionally been an integral part of software development.

The change from traditional software development process models, such as the waterfall model, to agile methods has created new challenges in the field of risk management. This thesis discusses how risks act as a mechanism in self-organizing agile software development teams. We report our analysis from the students’ agile teams and have found that the risk management strategies adopted by our students were inherently collaborative. We further leveraged this collaborative activity to execute instructor interventions. In addition to this, our study also reports that besides being collaborative, the students’ teams are lean. Moreover, we reported our observation from Ganga software development team (A High Energy Physics Application) and have found that their risk management follows some of the similar trends that our students’ agile teams do.
Acknowledgement

This thesis came to completion after a long journey of encouraging, constructive feedback and generous guidance given to me by my advisor, Dr. Nan Niu, whom I would like to express my heartfelt gratitude. I also would like to thank the committee members Dr. Raj Bhatnagar and Dr. Carla Purdy for taking their time out to review my thesis and being part of my committee. I also give my gratitude to Wentao Wang for continued support, patience and invaluable suggestions.

Finally, I would like to thank my parents Mr. Venkata Chandra Sekhar Thota and Mrs. Bharatha Lakshmi Thota without whose constant support and encouragement, this accomplishment would not have been possible.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
<td>1</td>
</tr>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>4</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>5</td>
</tr>
<tr>
<td>Index</td>
<td>6</td>
</tr>
<tr>
<td>List of Figures</td>
<td>7</td>
</tr>
<tr>
<td>List of Tables</td>
<td>9</td>
</tr>
</tbody>
</table>
# Index

## 1 Introduction
1.1 Risk Management – Agile Software Development Teams 10
1.2 Lean Learning in Agile Software Development Teams 12

## 2 Related Work and Foundation
2.1 Agile Software Development 16
2.2 Self-Organizing Teams 19
2.3 Risk Management from Agile Point of View 21
2.4 RM in Agile Software Development 25
2.4.1 Scrum Risk Management 25
2.4.2 XP Risk Management 26
2.4.3 DSDM Risk Management 27

## 3 Student Perceptions of Software Risks
3.1 Research Methodology 28
3.2 Related Work 30
3.3 Course Information 34
3.4 Risks and Classification 37
3.5 Instructor Intervention 47

## 4 Lean Learning of Risks in Students’ Agile Teams
4.1 Research Methodology 53
4.1.1 Agile Software Development and its Limitations 54
4.1.2 Lean in Agile Software Development 54
4.2 Related Work 56
4.2.1 Definition of Lean 56
4.2.2 Principles of Lean Software Development 58
4.2.3 Agile and Lean Software Development 61
4.3 Risk Tree Analysis 63
4.3.1 Binary Decision Tree 67
4.3.2 Minimal Cut Sets 68
4.3.3 Fault Tree Minimal Cut Sets 70
4.4 Validation of Risk Tree 74
4.5 New Insights from linking Collaboration with Waste Elimination 76

## 5 Risk as a Self-Organizing Mechanism in Ganga Software Agile Team
5.1 Introduction to Ganga 79
5.1.1 Ganga User Interface 79
5.1.2 Development and Testing 82
5.2 Ganga – Agile Software Development Team 83
5.3 Ganga – Risk Management 88
5.4 Observations 96

## 6 Conclusion

References 102
List of Figures

Figure 1.1 : Phases of Risk Management in Agile Teams 12
Figure 1.2: Lean and Agile in Context of Ericsson R&D 14
Figure 2.1: Iterative Development Cycle 19
Figure 2.2: Risk Management-Agile Point of View 22
Figure 3.1: Eclipse Plug-ins built to process iTrust’s Requirements 37
Figure 3.2: Emulator view of Mapbox’s “Route Navigation” 37
Figure 3.3: Ranking Changes of Risk Categories (spring 2015) 47
Figure 3.4: Code Snippet introduced by the Instructor 51
Figure 3.5: Ranking Changes of Risk Categories (spring 2016) 52
Figure 4.1: Core Principles of Lean 59
Figure 4.2: Seven Principles of Lean applied to IT Stream 62
Figure 4.3: Diagrammatic Representation of a Fault Tree 66
Figure 4.4: RT Representation for Requirements 67
Figure 4.5: Risk Tree Representation for Scheduling 68
Figure 4.6: Risk Tree Representation for Proper Software 69
Figure 4.7: The BDD for Fault Tree in fig 4.3 70
Figure 4.8: Minimal Cut Sets for each Event 72
Figure 4.9: The Minimal Cut Sets for Combined Fault Trees 75
Figure 4.10: Validation of Risk Tree for Spring 2015 Class 77
Figure 4.11 Validation of Risk Tree for Spring 2016 Class 77
Figure 4.12 New Insights of linking Collaboration with Lean Principles 78

Figure 5.1 Three Different User Interfaces of Ganga: 84

Figure 5.2: Ganga Developers E-Group 86

Figure 5.3: Developer’s Mailing Lists 87

Figure 5.4: Ganga User-Developers’ Mailing Lists 88

Figure 5.5: Ganga LHCb FAQ Page 89

Figure 5.6: Ganga LHCb Questions and Answers 90

Figure 5.7: The SEI Risk Management Paradigm 91

Figure 5.8: TWiki Page of the CERN Community 94

Figure 5.9: Mapping RM Activities into the Sprint 95

Figure 5.10: An Example of an Action Plan 97

Figure 5.11: TWiki – Regarding the Pros and Cons 99
List of Tables

Table 3.1: Basic Course Information 35
Table 3.2: Top-10 Risk Items with Industrial Relevance 41
Table 3.3: Comparison of Students’ perceived Top-10 Risk Items 44
Table 3.4: Classification of Risk Items collected from Spring 2015 Semester 47
Table 3.5: Classification of Risk Items collected from Spring 2016 Semester 52
Chapter 1: Introduction

Risk—the possibility of an unsatisfactory outcome—is an essential vehicle for a software development project to progress. It is common to have risks in a software development project. Risk in itself is neutral; risk is essential to progress, and failure is often a key part of learning. But we must learn to balance between the possible negative consequences and the potential benefits associated with it [1]. Risk can come with opportunities for the risk takers to lead a project to a different course of action, or to successful development. Risk management has traditionally been an integral part of software development. It is a series of steps whose objectives are to identify, address, and eliminate software risk items before they become either threats to successful software operation or a major source of expensive rework. Risk identification, management, and mitigation are essential to the success of any software development projects [1].

The change from traditional models, such as the waterfall model, to agile methods has created new challenges in the field of risk management [2]. Figure 1.1 illustrates how risk management is done in agile software development teams. This work will discuss how risks act as a mechanism in self-organizing agile software development teams.

1.1 Risk Management – Agile Software Development Teams

It is common for a software project to face many risks in its lifecycle, from conception and construction to deployment and maintenance. Risk is any potential situation or event that negatively affects the project’s success. While the ultimate
success of software often hinges on the fulfillment of the stakeholders’ requirements, the project failure can be multi-faceted: frequent rework, architectural mismatch, implementation difficulty, integration delay, just to name a few [3]. A risk item, therefore, can refer to the task, process, or environment of software engineering. Risk itself is neutral, but if ignored, it can lead to, transform into, or otherwise correlate with the project failure.

Making use of risks, then, underpins a particular class of iterative and incremental process models. The most well-known of the class is probably the spiral model described first by Boehm [3]. The key characteristic of the spiral model is to reduce or eliminate the software engineering activities that would risk wasting effort by pursuing options unacceptable to a project’s success-critical stakeholders. Considering the win conditions of the critical stakeholders, evaluating the alternatives for satisfying the win conditions, managing the risks that stem from the selected alternative(s), and obtaining the stakeholder approvals and commitments to pursuing the next cycle are the four basic steps of the spiral model [3]. For example, if sensitive information is stored and accessed but the end user is not invited to participate in the trade-off analysis and prioritization of the requirements, then the software project is at risk of failing to satisfy the security needs of the user.

Having key stakeholders such as user and customer representatives involved is a hallmark of agile software development [4], which is considered to not only inherent the iterative and incremental natures embedded in the spiral model but also adopt the rapid
and evolutionary development styles. In this thesis, we integrate risk as a driver for the self-organizing agile software development teams to evaluate and advance their team-based projects. This is discussed in chapter 3 by examining students’ agile teams and the knowledge from it is significant because risk is fundamental to change and it is part of the agile manifesto to value continuous and quick responses to change [4].

1.2 Lean Learning in Agile Software Development Teams:

Lean Software Development is the application of the principles introduced by the Toyota Product Development System to software development. Toyota has been extremely successful in developing complex new vehicles that included vast amount of
embedded software in a very short time, and also delivered on time. When correctly applied, lean software development results in high quality software that is developed quickly and at the lowest possible cost. Moreover, the success of many of the practices of Agile Software Development can be explained by understanding the principles of Lean Software Development. For example, the agile principles such as motivated and self-organizing teams, stakeholder involvement, sustainable development map to the people focus principle of lean learning. The advantage of using agile methodology is to get benefits like flexible project management, minimize product cost, increase stakeholder involvement, face-to-face communication and finally that can lead to customer’s satisfaction [5]. On the other hand, lean is not too far away from software development. Different authors advocate that the idea behind lean manufacturing might be useful for software development.

Of the many methods that have arisen to improve software engineering processes, Lean is emerging as one that is grounded in decades of work understanding how to make processes better.
Lean thinking focuses on giving customers what they want, when and where they want it, without a wasted motion or wasted minute. Lean Software Development provides the theory behind Agile Software Development practices and gives organizations a set of principles from which to fashion software engineering processes that will work best in the context of their customers, their domain, their development capability, and their unique situation. Figure 1.2 illustrates the lean and agile concept in the context at Ericsson R&D [6]. Poppendieck et al. [7] have done an extensive work in explaining...
Lean Software Development. This work has led many organizations to reexamine and restructure software engineering practices, resulting in dramatic improvements in the quality, cost, and development speed of products and processes containing software. Although there exists substantial amount work on Lean Software Development, there is little knowledge about how lean is applied in risk management activities in agile software development teams. In Chapter 4, we present some of our observations and analyses from our students’ agile teams on how these self-organizing teams are applying lean principles in their risk management activities. In Chapter 5, we report our observations of how risk acts a mechanism in High Energy Physics’ (Ganga) agile software development team. Also, we report how these teams adopt lean principles into their software development. In Chapter 6 we conclude with summary and our ideas for future scope of this project.
Chapter 2: Related Work and Foundation

2.1 Agile Software Development:

Agile software development represents a new approach for planning and managing software projects. Agile development is different from the traditional methods such as waterfall as it focuses little on up-front plans and stringent plan-based control and emphasizes more on those mechanisms for change management during the project. Further, agile development relies less on processes but more on people and their creativity. Leadership and collaboration, informal communication and an organic (flexible and participative, encouraging cooperative social action) organizational form are other characteristics of agile software development [4]. Practitioners who have developed agile methods have formulated the agile manifesto, a statement that expresses a set of basic principles and rules: individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan. There is extensive work done which refers to a number of possible setbacks or errors that may occur when an organization is making the transition from plan-driven towards agile processes. The problems are mainly caused by resistance to, or over-enthusiasm for, agile practices within a software development team. When changing from plan-driven to change-driven, this may also impact several aspects of the organization including its structure, culture, and management practice [5]. Neither culture nor mind-sets of people can be easily changed, which makes the move to agile methodologies all the more formidable for many organizations. Both practitioners and academics have criticized agile development
methods, e.g., the lack of scientific support for many of the claims made by the agile community. Software development processes depend significantly on team performance, as does any process that involves human interaction. Two important factors that are considered for a significant amount of achievement of the team performance are feedback and communication. Agile development relies on teamwork, in contradictory to the individual role assignment around which the plan driven development revolves. A team following a plan driven model often consists of independently focused self-managing professionals. As discussed in many of the agile methodology works, the biggest challenge for introducing a change-driven development which is self-organized based, is the transition from high individual and low group autonomy to a high level of individual and group autonomy.

Agile software development comprises a number of practices and methods. Among the most known and adopted agile methods are Extreme Programming (XP) and Scrum. XP primary focus is on the implementation of software, while Scrum emphasizes on agile project management. Scrum focuses on project management in those situations where it is difficult to plan ahead, with mechanisms for “empirical process control”; where feedback loops are the core element. Software is developed by a self-organizing team in increments (called "sprints"), which starts off with a planning and has a review at the end. Features that are to be implemented in the system are registered in a backlog, and a product owner decides which backlog items should be developed, by prioritizing them, in the following sprint. Coordination and communication among the team members are
done in daily stand-up meetings and through the intra-communication such as e-mails. One team member, the scrum master, is responsible for solving problems that stops the team from being successful in delivering the increment. For studies of Scrum, the Scrum team is given significant authority and responsibility for many aspects of their work, such as planning, scheduling, assigning tasks to members, and making decisions: "The team is accorded full authority to do whatever it decides is necessary to achieve the goal". The name "Scrum" was inspired by an article characterizing new product development teams in Japan, by Takeuchi and Nonaka [8], where “self-organizing project teams” was one of six key characteristics. The Scrum master is often described as a coach or facilitator. He or she does not organize the team; the team organizes itself and makes the decisions concerning what to do. The Scrum master works to remove the impediments of the process, runs and makes decisions in the daily meetings and validates them with the management. Scrum and Agile development favor a leadership-and-collaboration style of management where the traditional project manager’s role is replaced with the Scrum master’s role of a facilitator or coordinator. Although agile development is a relatively new approach, this is not the case with self-management. Figure 2.1 illustrates the iterative development cycle. Research in this area has been around since Trist’s examination of self-regulated coal miners in the 1950s.
2.2. Self-Organizing Teams

We use the label "self-organizing" teams as a synonym for "autonomous teams" and for "empowered teams". Guzzo and Dickson [9] describe such teams as teams of employees who typically perform highly related or interdependent jobs, who are identified and identifiable as a social unit in an organization, and who are given significant authority and responsibility for many aspects of their work, such as planning, scheduling, assigning tasks to members, and making decisions with economic consequences. Autonomous teams stimulate participation and involvement, and as a result there is an increased emotional attachment to the organization. This leads to a
greater commitment, also the team members are highly motivated to perform and desire for responsibility. As an effect, employees care more about their work, which may lead to greater creativity and helping behavior, higher productivity and service quality. Self-management can also directly influence team effectiveness since it brings decision making authority to the level of operational problems, uncertainties and, thus, increase the speed and accuracy of problem solving [4].

Self-organizing teams are seen as one of the premises for succeeding with innovative projects. Kirkman and Rosen [10], for example, studied 111 teams in four organizations, and found that empowered teams (autonomy central in these teams) were more productive and proactive than less empowered teams. Also, they suggested that self-management is the “first design principle” for an innovative and collaborative organization. Takeuchi and Nonaka [8] found that although the members of a self-organized team are largely on their own, they are not uncontrolled. Management should establish enough checkpoints to prevent instability, ambiguity, and tension from turning into chaos. At the same time, managers should avoid the kind of rigid control that impairs creativity and spontaneity.

However, there is substantial variance in research findings regarding the consequences of such teams on such measures as productivity, turnover, and attitudes. It is important that the teams do not experience symbolic self-management. This variance may indicate that the effects of such teams are highly situationally dependent. Also there is evidence indicating that the type of autonomy may be just as important as the amount
Agile Software Development emphasizes if a team is self-organizing it will lead to a more motivated team.

Self-organizing teams possess their own knowledge, perspective, motivation and expertise to apply on the project they are working in. In this kind of environment “the team is treated as a partner with management and the customer, capable of providing insight, affecting decisions, and negotiating commitments”. Agile Software Development sees the concept of self-organizing team as a way to create motivated teams and most importantly to deliver technical excellence. Individuals in a team know about each other’s capabilities better than any other entity so when they are self-organized they can bring the best out of each other and deliver the best technical excellence.

2.3 Risk Management from Agile Point-of-View

Agile does not dictate a risk management approach but Agile is a risk mitigation strategy in itself, and several of the Agile practices make traditional risk management easier. The project manager is responsible for risk management. Figure 2.2 illustrates the overall view of risk management from agile point-of-view.
Figure 2.2 Risk Management – Agile Point of View

**Agile Risk Identification:** Risk identification is about thinking of what might occur and why it might occur. It also includes listing those risks that threaten the completion of the project implementation. There are several obstacles that the project face during its implementation. These risks or issues are highlighted in the agile daily meetings and most of them are handled by the team immediately, while the other are
processed outside the meeting. At the beginning it is expected to have more likelihood of identifying risky elements, due to the team collaborative nature of Agile project estimating. Threats to the scheduled are early highlighted because of the empirical nature of Agile Project Planning and Agile Project Control which ensures the velocity and continuously recalibrate.

**Agile Risk Analysis:** Risk analysis is the process of assessing the likelihood and impact of each risk. The project manager can do the ratings themselves or get a specialist to do it. Once these two dimensions are calculated, teams understand the threats of each risk through this analysis phase and get a better understanding on what would happen if they are not mitigated in time.

**Agile Risk Prioritization:** In this phase, the project manager identifies the significant risks and calculates the risk exposure by multiplying the likelihood by the impact. The risk prioritization in Agile aims to ensure that the work with the most significant risk is completed first. It is a continuous and dynamic process throughout a project by ranking of features and enables changing priorities and inserts new information.

**Agile Risk Mitigation Planning:** In risk management planning one decides how to deal with each significant risk. The approach taken will vary depending on the nature of the specific risk, but generally speaking there are four approaches to risk management planning. Risk retention – basically means accepting the loss when it occurs. One shall
adopt this only if the cost of addressing the risk is greater than the impact of the risk. Risk avoidance – avoid the risk by not doing the activity that carries the risk. Risk reduction – any method that reduces the impact or likelihood of the risk, hence the risk exposure. Risk transfer – get somebody else to accept the risk.

Agile offers some risk management techniques. The collaborative nature of an Agile team means that the team can share responsibility for resolving a particular risk. Agile advocates bringing risky requirements forward in the schedule. This means there is more time to assess the risk of the item and to identify feasible solutions. Agile also promotes the idea of investigating risky requirements.

**Agile Risk Resolution:** Risk resolution means executing the risk-management plan, i.e. dealing with each significant risk. If the Agile team has to do anything to resolve the risk, then it has to be factored into the plan. *Agile Project Planning* means all work for the Agile team appears either on the Release Plan as requirements and/or the Time box Plan as tasks.

**Agile Risk Monitoring:** The project manager must continue to monitor the risk-management plan dealing with each significant risk. Any work involving the Agile team will be in the Release Plan and/or Time box plan. The project manager must, however, also monitor risk-management plan for risks that are being dealt with outside the Agile team. Finally, the project manager also needs to continue risk identification which takes us back to the beginning of the risk management cycle.
2.4 RM IN AGILE SOFTWARE DEVELOPMENT

2.4.1 SCRUM Risk Management

Jean Claude [11] manages the risk as usual with Risks identification and prioritization in the first Sprint, then risks discussed with the team and associated with the actions to be done. Monitoring for risk is done with using Risk board, Task board and Burndown Chart. Carr et al. [12] modifies the previous process, risk identification applied to Risk board is in a form of a risk matrix, red sticky notes attached on the board for risks and yellow for solutions. Check list was used for high-risk identification with considerable to security of the case, then checklist added to the sprint backlog to be monitor. After response, note and risk will be moved to a new spot on the board. The accepted risk saved in a digital form for re-evaluation.

The literature gives an approach that specifies a five simple steps to Agile Risk Management which includes more details and specification than the previous approaches. The most complex step is Quantify risks according to two vectors, probability of the risk being realized, and Impact, and then multiply two vectors to end up with a Risk Matrix. After that an action is specified to mitigate the Risk. There was a recorded response strategy and treatment in a risk log. Communication, working practices and culture are the challenges that the paper addressed. One of the solutions to improve the agility of this company was to use video conferencing facilities which made the meeting as much as possible like actual meeting in one room, to share desktops GoToMeeting is used. Cristal et al. [13] described two experimented scrum in globally distributed companies cross
North America, South America and Asia. It is better to integrate use case diagram and user stories. Work as group is important. The researchers presented the challenges in Scrum implementation by a survey conducted in two companies and their experience in implementation of scrum. In order to have good code quality the paper suggested to do re-scoping for late tasks. These tasks can be divided into subtasks doing through sub sections in sprint moving and finished in the next sprint. The authors found that frequent informal meetings, debates, conferences, and policies reduced the social and cultural issues. Finally, Berczuk [14] reviewed Norway experience which is a company committed to using scrum as a development method. The paper recommended to use instruments like clocks to show the other time zones. This gave a connecting feeling: they used smaller sprint durations that gave the ability to maintain a stable backlog and frequently release.

2.4.2 XP Risk Management

Mathkour et al. [15] reviewed various tools for Risk Management. Risk tool was designed and developed using MS Excel, applied risk management on extreme programming practices, by inserting some information into the tool as project budget, risk management budget, cost of controls, the tool analyzes the priority practices such as pair programming, sustained pace, refactoring, Test-driven development, and decide which practice must be dealt with first. Dyba et.al [16], collected detailed requirements from user stories: then the user stories were prioritized by client, and given High index value for high priority. To ensure on time delivery and to facilitate managing stories planning
poker technique was implemented. The speed of the project is measured after the first phase of development is released, then consecutive measurements done at every release.

2.4.3 Dynamic Systems Development Method (DSDM) Risk Management

The studies also used suitability filter to check if a project is compatible for the use of DSDM. After that Risk log is created, updated throughout the life of the project. Requirements are prioritized and time boxing are used. Risk Estimation is done with help of both quantitative and qualitative techniques and finally, they took numerous stakeholders’ opinion to evaluate the risk. In his book, Moran [17] suggested an agile chart with appropriate placement of risk tasks, and mentioned that DSDM already contains some elements such as delivery plan, risk log, which address the risk and practices such as time box. In addition, scoring and tagging could easily be deployed in a DSDM to support identifying, analyzing and monitoring risks. DSDM provided many strategies for risk management such as, share strategy, in which there is a need to management of third parties to be implemented.
Chapter 3: Students’ Perceptions of Software Risks

In this chapter, we report the data collected and analyzed in two semesters of a junior-level software engineering course where undergraduate students were working in agile teams to deliver 4 major working increments per semester (62 students developed Eclipse plug-ins in one semester and 103 students developed Android apps in the other). In both semesters, we found that not only were our students’ perceived top-ranked risks remarkably different from what were previously published (including the industry-surveyed checklists in the 1990s and 2000s, as well as the ones collected from a graduate-level course), but the risk management strategies adopted by our students were inherently collaborative. We leveraged this collaborative nature to design and execute the instructor interventions. The results comparing the top risks between the two semesters show the effectiveness of the instructor interventions and suggest ways to further improve risk management in students’ agile software development teams.

3.1 Research Methodology

Risk—the possibility of an unsatisfactory outcome—is an essential vehicle for a software development project to progress. Iterative and incremental process models like spiral advocate the continuous identification of the items likely to compromise the project’s success and the early resolution of those top-ranked risk items. Although the concepts and principles, such as risk exposure and project top-10 risk-item monitoring, are commonly taught in undergraduate software engineering courses, little is known about
how students, especially those working in agile software teams, perceive, prioritize, and try to mitigate their risks over multiple development cycles.

Despite being confronted with challenges like scalability, agile software development methodologies are now ubiquitous within industry. It is therefore important for students to learn and practice agile project development before they enter the software industry. Many educators shared their experiences in teaching agile methodologies [2]; however, to the best of our knowledge, there is no prior work integrating risk as a driver for the students to evaluate and advance their team-based agile projects. This knowledge gap is significant because risk is fundamental to change and it is part of the agile manifesto to value continuous and quick responses to change [3].

To bridge the gap, we report in this chapter our recent experiences in teaching the junior-level software engineering classes where we instrumented risk as a first-class citizen in students’ agile development projects. In particular, we asked the student teams to self-identify a ranked list of 5 risk items that they perceived to hinder themselves from successfully completing the project’s next iteration. For the spring 2015 semester, 62 students worked in 15 teams to develop Eclipse plug-ins, and for the spring 2016 semester, 103 students developed Android apps in 25 teams. In both semesters, the students were tasked to deliver 4 major working increments where each increment was given 3 or 4 weeks to complete. These increments served as project milestones. Together with the working software, each team was required to submit their ranked risk items as
part of their agile reflections at each of the 4 project milestones. The risk items were the main sources for our data analysis and reporting.

Our contributions in this chapter lie in the analysis of students’ own perceptions of risks and their use of risks in managing the agile software development. The risks perceived by our students differed from those reported in prior studies [18] [19]. Furthermore, our classification of the results of the first offering of the course suggested collaborative ways of identifying and mitigating the risks. Based on the collaborative risk management, we designed several instructor interventions in our second course offering and showed the effectiveness of such interventions. The rest of this chapter is organized as follows. Section 3.2 describes related work. Section 3.3 presents the background information of our software engineering classes. Section 3.4 compares and classifies the risks perceived by our students. Section 3.5 presents the instructor interventions and evaluations.

3.2 Related Work

Risk management is one of the fundamental activities in industrial software development and therefore often taught in undergraduate software engineering classes. A risk item denotes a particular aspect, property, or characteristic of a development task, process, or environment which, if ignored, will increase the likelihood of a project failure [1]. Risk management typically involves identifying, analyzing, mitigating, and monitoring the risk items. Common techniques introduced in undergraduate software
engineering courses include risk exposure, which quantitatively analyzes the probability of an unsatisfactory outcome and the loss to the stakeholders affected if the outcome is unsatisfactory. Another method taught in the classroom, as well as used in industry, is the listing of project top-\(n\) risk items (e.g., \(n=10\)) \([18]\) \([19]\). Examples of top ranked risk items are personnel shortfalls, requirements volatility, architecture complexity, and quality tradeoffs. Although risks could be perceived from contractor or client perspectives \([12]\), our focus in this paper is on the students’ own learning and practice perspectives.

Koolmanojwong and Boehm \([20]\) analyzed the risks encountered by student teams in a graduate-level software engineering course sequence (i.e., two semesters of fall and spring). The students were grouped in a 5- or 6-people project team and their grades were used to correlate their risk management activities. Overall, Koolmanojwong and Boehm concluded that students who performed better risk management would receive better grades and that students who identified risks in the fall semester would collaboratively mitigate those risks in the subsequent spring semester. While the collaborative risk management was explored, our work differs from Koolmanojwong and Boehm’s study in two aspects: (1) we intentionally developed and executed instructor interventions in terms of assisting collaborative risk management, and (2) we taught junior-level undergraduate students with the same course in two different semesters, not graduate students in a sequence of courses in consecutive semesters.

One of the most comprehensive sources for potential risk items is the Software Engineering Institute’s risk management questionnaire consisting of 194 questions that a
software development team can use to identify risks in their project [21]. Clearly, not all projects need to address all 194 issues. A risk-tree structure was presented by Hoodat and Rashidi [18] where the lowest-level risk items were grouped and merged into a hierarchical representation. Using the different kinds of risk tree (e.g., requirements risk tree, cost risk tree, quality risk tree, or scheduling risk tree), the analysis and assessment could become less complex and more modular. Examining managers’ perceptions of software risks showed that project performance was influenced by at least two types: objective risks and resilience risks [19]. While objective risks had negative impacts on project performance, Han’s [22] study showed that either a risk-focused or a performance-actualized prioritization strategy would provide a gradual foundation for controlling risks without worrying about excess or deficient risk management investments.

While industrial projects customized risk management strategies, Collofello and Pinkerton [23] showed that undergraduate students adapted the taxonomy-based risk identification in their own ways. In a one-semester, project-based software engineering course in which students worked in teams of 5-6 members, risk management content was introduced about 5 weeks into the semester at a point where the teams had completed their requirements documentation and were ready to plan the remainder of the project. Even though the undergraduates were exposed to the 194 questions [21], the students perceived only 36 questions to be relevant and further grouped the identified risk items into 6 categories. The usefulness of the condensed list was shown in a waterfall development model in that the course reported by Collofello and Pinkerton [23] spanned the entire
semester starting with the teams defining their software projects’ requirements and ending with acceptance testing for the customer. In contrast, an increasing number of courses began to include the teaching of agile software development methodologies such as Scrum and XP (extreme programming). Schroeder et al. [24] presented their positive experiences of developing two software development labs using Scrum and found using fun challenges not only better motivated the students but also provided a skeleton and development environment for a quicker start to their projects. Rico and Sayani [25] reported their adaptation of final-year student courses to agile methods and found that proper tutoring and coaching of teams were key to project success.

Not only were positive experiences of teaching agile shared, but some cautions required attention. Anslow and Maurer [26] commented that an agile course should not have too many lectures and that a serious risk was scope creep, which caused unbalanced workloads for the students within the team. Devedzic and Milenkovic recommended eliminating major difficulties early and structuring short iterations [27]. It is important to realize that “agile works with agile students”. Therefore, mentoring is not always effective, and sometimes, it is best for the students to discover things on their own [27]. In this paper, we discuss our undergraduate software engineering classes where students working in agile development teams identified and mitigated risks in their projects.
3.3 Course Information

Table 3.1 presents the basic information from our junior-level software engineering course. To avoid the green-field, waterfall-style development, we chose an existing software system for our students to build upon. Both baseline systems are relatively mature: iTrust was established in 2007 and Mapbox started its development in 2010. Although written mainly in Java, iTrust is a Web application, whereas Mapbox is a mobile app. This leads to distinct integrated development environments for our students to use: Eclipse for iTrust and Android Studio for Mapbox.

<table>
<thead>
<tr>
<th>Semester</th>
<th>Spring 2015</th>
<th>Spring 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enrollment information</td>
<td>62</td>
<td>103</td>
</tr>
<tr>
<td># of students</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td># of teams</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version (release date)</td>
<td>v19 (January 2015)</td>
<td>v3.0.0 (December 2015)</td>
</tr>
<tr>
<td>Size of baseline version</td>
<td>24.6 kilo lines of code</td>
<td>6,214 lines of code</td>
</tr>
<tr>
<td>Written in</td>
<td>Java</td>
<td>Java</td>
</tr>
<tr>
<td>Development environment</td>
<td>Eclipse</td>
<td>Android Studio</td>
</tr>
<tr>
<td>Main deliverables</td>
<td>Eclipse plug-ins</td>
<td>New features in the mobile app</td>
</tr>
<tr>
<td>Milestone information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lab1</td>
<td>Requirements indexing</td>
<td>Route navigation</td>
</tr>
<tr>
<td>Lab2</td>
<td>Source code indexing</td>
<td>Making phone calls</td>
</tr>
<tr>
<td>Lab3</td>
<td>Refactoring</td>
<td>Refactoring</td>
</tr>
<tr>
<td>Lab4</td>
<td>Changing requirements</td>
<td>Changing requirements</td>
</tr>
</tbody>
</table>
Another noticeable difference shown in Table 3.1 is the size of the subject systems. This influenced our design of the overall themes of the student projects. Adding new and extended features was reasonable for the moderate Mapbox codebase, but doing so for iTrust with tens of thousands lines of code would require significant, upfront comprehension effort. Rather than directly extending iTrust, we asked the students enrolled in the spring 2015 semester to build Eclipse plug-ins to process the requirements and source code artifacts of iTrust.
Figure 3.1 Eclipse Plug-ins built to process iTrust’s Requirements and Source Code Artifacts

Figure 3.2 Emulator View of Mapbox’s “Route Navigation” and “Making Phone Calls” Feature
Figure 3.1 illustrates iTrust artifacts’ processing. Three Eclipse plug-ins are shown: Figure 3.1-A lists a set of to-be-indexed requirements, Figure 3.1-B outputs the indices of the input requirement, and Figure 3.1-D outputs the indices of a Java method that is selected as the to-be-indexed source code artifact (cf. Figure 3.1-C). The development of Eclipse plug-ins was allocated to the first two labs in the spring 2015 semester, as shown in Table 3.1. Similar development tasks were assigned in the spring 2016 semester, though the students worked on direct feature extensions of Mapbox (cf. Figure 3.2). In both semesters, the last two labs put more emphasis on nonfunctional requirements such as improving maintainability and extensibility via refactoring, enhancing scalability and accuracy of iTrust indexing, hardening Mapbox security, and so on.

3.4 Risks and Classification

We collected in total 160 lists of top 5 risk items identified by the 40 student teams themselves at each of the 4 milestone deliverables over the 2 semesters. Compared to the top-10 lists, we adopted the lightweight documentation of agile manifesto [4] to require only the top-5 risk items submitted by our students. Lightweight documentation does not mean no documentation. We explicitly instructed our students to submit a team-wide reflection document at each milestone delivery point, where they recorded experiences gained, lessons learned, and also risks encountered in the current iteration that they perceived to be serious and to have negative impacts on them moving forward as a team.
Early on in the semester, relevant materials like risk-driven process models were taught in the lectures. Selected materials like risk exposure were also practiced as in-class exercises. The team-based learning, including that involved the identification and mitigation of risks, was performed in the labs which were an integral component of the course. Following Devedzic and Milenkovic [27], we designed short iterations (with each iteration/lab spanning 3 to 4 weeks). This allowed the students to stay focused on their development activities while recognizing the risks. Note that in both semesters (cf. Table 3.1), the last iteration/lab was about addressing changing requirements. The risks submitted as part of lab 4’s reflection were different from those submitted in the other three milestones; however, we believe the differences should not be significant because the students were practicing active risk management throughout the semester.

To situate our results properly in the literature, we first list in Table 3.2 the top-10 risk items published in the 1990s and 2000s [19] [20]. The survey pool by Boehm [19] and Reifer [28] was a set of several experienced project managers and a set of consulted Internet/intranet projects. The lists of Table 3.2 contain items specific to industrial software projects, such as high turnover of skilled personnel and unproductive office space [21]. Common to the lists presented in Table 3.2 are personnel shortfalls and several requirements-related issues (mismatch, misalignment, and volatility). Koolmanojwong and Boehm [20] argued that those requirements-related issues could result in higher risks for the software teams working in an academic setting since
misinterpreting and/or not fully understanding the client’s requirements would lead to project failures and/or expensive revisit of requirements later on in the development phase.

The summary of the top-10 risks from Koolmanojwong and Boehm’s study [20] was shown in Table 3.3 where we also present the aggregated risk items from our spring 2015 offering of the software engineering course. The evolution of the students’ risk perceptions between our two semesters will be detailed in the next section, mainly due to the instructor interventions introduced in the spring 2016 semester. Note that although our student teams analyzed their top-5 risk items at each milestone, Table 3.3 shows the top-10 risk items from our spring 2015 dataset for the purpose of comparison. In our analysis, an item was ranked higher based on the frequency of occurrence, that is, if the item appeared more times (irrespective of being ranked 1st, 2nd, 3rd, etc.) when we considered all the student teams’ risk submissions in spring 2015.
Table 3.2 Top-10 Risk Items with Industrial Relevance

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Personnel shortfalls</td>
<td>Personnel shortfalls</td>
<td>Architecture complexity, quality tradeoffs</td>
</tr>
<tr>
<td>2</td>
<td>Unrealistic schedules and budgets</td>
<td>Misalignment with business goals</td>
<td>Requirements volatility</td>
</tr>
<tr>
<td>3</td>
<td>Requirements mismatch</td>
<td>Unrealistic customer and schedule expectations</td>
<td>Acquisition and contracting process mismatches</td>
</tr>
<tr>
<td>4</td>
<td>User interface (UI) mismatch</td>
<td>Volatile technology (e.g., .NET, Persistence, J2EE, etc.)</td>
<td>Budget and schedule constraints</td>
</tr>
<tr>
<td>5</td>
<td>Gold plating</td>
<td>Unstable software releases (especially poor performance and frequent crashes)</td>
<td>Customer-developer-user team cohesion</td>
</tr>
<tr>
<td>6</td>
<td>Requirements volatility</td>
<td>Constant changes in software functionality</td>
<td>Requirements mismatch</td>
</tr>
<tr>
<td>7</td>
<td>Shortfalls in externally furnished components</td>
<td>Even newer methods and more unstable tools</td>
<td>Personnel shortfalls</td>
</tr>
<tr>
<td>8</td>
<td>Shortfalls in externally performed tasks</td>
<td>High turnover (especially of those personnel skilled in the new technology)</td>
<td>COTS (commercial off-the-shelf) complexity and shortfalls</td>
</tr>
<tr>
<td>9</td>
<td>Real-time performance shortfalls</td>
<td>Friction within the team (lack of leadership, overwork, etc.)</td>
<td>Technology maturity</td>
</tr>
<tr>
<td>10</td>
<td>Straining computer science capabilities</td>
<td>Unproductive office space</td>
<td>Migration complexity</td>
</tr>
</tbody>
</table>
Compared to the results from Koolmanojwong and Boehm’s study [20], our students identified similar top-ranked risks, such as personnel shortfalls, requirements mismatch, and process quality assurance. Some items were refined, e.g., architecture complexity was mainly reflected in the lack of an overall view of the subject software system (iTrust or Mapbox) as it relates to the deliverables required for the specific development cycle. We note from Tables 3.2 and 3.3 that certain top-ranked risk items reported in prior studies [20] [21] [22] were no longer ranked high by our students; most noticeably, requirement volatility appeared only a few times, showing that new and changing requirements were not perceived as a high-risk item but rather something to be expected in an agile environment. Our results in Table 3.3 suggest a couple of new top-ranked risk items surrounding the socio-technical nature of software development.

**Software artifact dependencies** was the second highest ranked risk, indicating that a core aspect of software development is about engineering concrete work items that will make up the whole system. Although agile methods have an artifact focus (especially favoring the working software [3]), our students experienced challenges in managing the interdependencies among the various kinds of software artifacts.

**Task dependencies** was ranked as a top-5 risk in Table 3.3, implying the difficulty faced by our students in terms of assigning roles, distributing workload, and integrating individual contributions among themselves. While self-organizing
is considered a key attribute of an agile team, it is often easier said than done and the risk of not being able to self-organize is shown in our results.

It is important to note that only frequently occurred risks items are shown in Table 3.3. In our spring 2015 class—in fact, in both classes—certain risks were sensitive to only a particular phase. For example, technical background insufficiency appeared often in the beginning of the semester as the students became more familiar with the programming language (Java), the integrated development environment (Eclipse or Android Studio), and the subject system (iTrust or Mapbox). Similarly, the risk concerning unstable software releases began to emerge in the middle of the semester, echoing the results of Reifer’s industrial study [28]. Had the students’ projects included more development cycles, the unstable release risk might be ranked higher.
We classified the risk items collected from our spring 2015 semester in two dimensions: one was concerned about how the risk was identified and mitigated, and the other focused on who carried out the activities (i.e., done individually or collaboratively). While the first dimension touches upon core risk management activities, especially those in educational settings [20], the second is orthogonal yet important in team-based software engineering projects. Previous educational papers, surprisingly, paid little attention to the collaborative aspect of risk management, e.g., the undergraduates were asked to individually mitigate the risks and have their weekly top-ranked risk items

<table>
<thead>
<tr>
<th>Rank</th>
<th>Graduate-level course\textsuperscript{12}</th>
<th>Our junior-level course</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Architecture complexity, quality tradeoffs</td>
<td>Team cohesion and communication</td>
<td>Team members do not respond to each other when support is needed</td>
</tr>
<tr>
<td>2</td>
<td>Personnel shortfalls</td>
<td>Software artifact dependencies</td>
<td>Certain requirements are requisite for implementing others</td>
</tr>
<tr>
<td>3</td>
<td>Budget and schedule constraints</td>
<td>Schedule constraints</td>
<td>Different course schedules limit team members’ face-to-face meeting</td>
</tr>
<tr>
<td>4</td>
<td>COTS and other independently evolving systems</td>
<td>Task dependencies</td>
<td>Teams have difficulty in distributing workload and integrating individual contributions</td>
</tr>
<tr>
<td>5</td>
<td>Customer-developer-user team cohesion</td>
<td>Architecture complexity (especially lack of an architectural overview)</td>
<td>Team lacks an overall picture of the software as it relates to the lab deliverables</td>
</tr>
<tr>
<td>6</td>
<td>Requirements volatility</td>
<td>Personnel shortfalls</td>
<td>A single member needs to play multiple roles, sometimes with less competency</td>
</tr>
<tr>
<td>7</td>
<td>UI mismatch</td>
<td>Operational environment interoperability</td>
<td>Software debugged in one machine (e.g., Mac) has unexpected runtime behavior in another (e.g., Windows)</td>
</tr>
<tr>
<td>8</td>
<td>Process quality assurance</td>
<td>Requirements mismatch</td>
<td>Underspecifying requirements or making incorrect assumptions</td>
</tr>
<tr>
<td>9</td>
<td>Requirements mismatch</td>
<td>Process quality assurance</td>
<td>Team members cannot decide a sufficient unit testing level</td>
</tr>
<tr>
<td>10</td>
<td>Acquisition and contracting process mismatches</td>
<td>Unstable software releases</td>
<td>External components (e.g., libraries or APIs) become deprecated, causing unexpected crashes</td>
</tr>
</tbody>
</table>

\textsuperscript{12} We classified the risk items collected from our spring 2015 semester in two dimensions: one was concerned about how the risk was identified and mitigated, and the other focused on who carried out the activities (i.e., done individually or collaboratively). While the first dimension touches upon core risk management activities, especially those in educational settings [20], the second is orthogonal yet important in team-based software engineering projects. Previous educational papers, surprisingly, paid little attention to the collaborative aspect of risk management, e.g., the undergraduates were asked to individually mitigate the risks and have their weekly top-ranked risk items...
monitored and the graduate students jointly mitigated risks across different semesters once they identified similar risks in an earlier semester [20]. Here, we aim to examine more closely collaborative risk management within a single semester of our software engineering course. We illustrate the four categories of our classification as follows.

**C1 (individually-identified and individually-mitigated risks):** A majority of the risk items in this category related to lack of experience in technical areas such as the programming language (Java) and the development environment (Eclipse or Android Studio). Students recognized their own insufficiencies and expended individual effort in overcoming them.

**C2 (individually-identified and collaboratively-mitigated risks):** A representative risk item in this category was a merging conflict, which was impacting a specific team member (and hence tended to be identified individually), but in order to resolve the conflict, joint effort was typically required. As an example, one student found that some code that he committed to Git was overridden due to another team member’s more recent check-in of code files with the same name. Preventing risks of this kind from happening again would need team-wide coordination.

**C3 (collaboratively-identified and individually-mitigated risks):** A sample risk item was requirements mismatch, which the members of a project team...
discovered together (e.g., whether “stop word removal” should use the same or different lists of stop words for requirements and source code artifacts). The resolution of risks in this category was predominantly solo (e.g., reading more literature or discussing the issue with the customer representative).

**C4 (collaboratively-identified and collaboratively-mitigated risks):** Many instances of task dependencies (cf. Table 3.3) fell into this category. An example was the coordinated decomposition of development tasks with pre-defined interfaces to avoid potential problems in integration testing.

Table 3.4 presents the descriptive statistics of the risk items based on the above classification. In addition to the raw number of the risk items, the ranking of each of the four categories is provided in Table 3.4 in the form of mean and standard deviation (m±s.d). The temporal trends of each category’s mean rank are plotted in Figure 3.3, which we further analyze in the next section. It is important to note from Table 3.4 that a very few risk items did not fit well into our classification. These included: version control (two occurrences) as a risk associated with both inadequate experience and inappropriate process quality assurance, unexpected unavailability of a team member or team members (three occurrences), and crash of the GitHub server (one occurrence).
Table 3.4 Classification of Risk Items collected from the Spring 2015 Semester

| Category | Lab1 | | Lab2 | | Lab3 | | Lab4 |
|----------|------|----------|------|----------|------|----------|
|          | # of | Rank | items | $m \pm s. d$ | # of | Rank | items | $m \pm s. d$ | # of | Rank | items | $m \pm s. d$ |
| C1       | 29   | 1.89 \( \pm 1.55 \) | 20   | 2.43 \( \pm 1.52 \) | 22   | 4.07 \( \pm 1.74 \) | 19   | 4.96 \( \pm 0.62 \) |
| C2       | 23   | 3.38 \( \pm 1.08 \) | 22   | 2.47 \( \pm 1.63 \) | 24   | 2.82 \( \pm 1.44 \) | 24   | 2.07 \( \pm 1.28 \) |
| C3       | 12   | 1.54 \( \pm 1.16 \) | 16   | 1.68 \( \pm 1.33 \) | 13   | 1.93 \( \pm 1.05 \) | 11   | 4.03 \( \pm 0.74 \) |
| C4       | 9    | 4.38 \( \pm 0.74 \) | 14   | 3.79 \( \pm 0.92 \) | 15   | 3.57 \( \pm 1.83 \) | 21   | 1.77 \( \pm 1.38 \) |
| Others   | 2    | 3.00 \( \pm 2.83 \) | 3    | 4.00 \( \pm 1.73 \) | 1    | 4.00 \( \pm 0 \) | 0    | - |

Figure 3.3 Ranking changes of Risk Categories (spring 2015)
3.5 Instructor Interventions

Analyzing Figure 3.3, together with Table 3.4, allowed us to recognize C3 (i.e., collaboratively-identified and individually-mitigated risks) had the potential to be effected by the instructor in broad settings. The reason is two-fold: the raw number of C3 risk items was moderate and relatively stable as the spring 2015 semester proceeded (cf. Table 3.4), and the overall temporal pattern of C3 risk items’ severity was smoothly decreasing (cf. Figure 3.3). For the other risk categories, Figure 3.3 shows that C1 experienced an early drop in severity, C2 was fluctuating, and C4 roared in terms of ranking increase as the semester went on. In addition to the above observations, we regarded C3 risks as being suitable for the instructor to help identify, or at least, to raise the awareness of the class as a whole. Because C3 risks were mainly mitigated individually, we decided that our instructor interventions would touch very little on specific risk mitigation method, allowing different teams or different members to develop their own solutions. In our spring 2016 offering of the course, we designed and implemented the instructor inventions targeted at C3 risks at various stages.

**Before Lab1:** We identified in the lecture the risks related to UI mismatch between Mapbox as an implementation host and the “route navigation” as the to-be-implemented feature (cf. Table 3.1 and Figure 3.2). In particular, there already exists a DoubleClickZoomHandler in Mapbox ([https://www.mapbox.com/mapbox-gl-js/api/#DoubleClickZoomHandler](https://www.mapbox.com/mapbox-gl-js/api/#DoubleClickZoomHandler)) which zooms the map at a touch point by double clicking. Should the students apply double clicking to set up the to-be-navigated point,
the risk of UI mismatch would arise. Figure 3.4 shows the code snippet that we introduced in the lecture, as well as demoed in the Lab1 session, in order to point out the class of UI mismatch risks. Note that even though we showed only DoubleClickZoomHandler in the class, all the student teams were able to uncover other UI mismatch risk items such as LongPressHandler, SingleTapHandler, and TapAndDragHandler in Mapbox. Also note that, it was our intention not to teach any specific risk mitigation in the class. It turned out students had different ways to address the UI mismatch, e.g., by using single click to implement “route navigation” or by overriding DoubleClickZoomHandler in Mapbox.

**Before Lab2:** We emphasized the risks that might be caused by feature interactions, as the students were asked to deliver “making phone calls” on top of “route navigation” (cf. Table 3.1 and Figure 3.2). The feature interaction example that we used in the lecture was the security concern of accessing system resources like sending an e-mail or making a phone call. If the students only followed Google’s “development class interaction with other apps” guideline ([https://developer.android.com/training/basics/intents/index.html](https://developer.android.com/training/basics/intents/index.html)), then the risk would emerge in that Mapbox could be blocked from accessing the phone resources. Again, we did not endorse any specific solution in lecture. Students had diverse ways to handle the feature interaction: some engaged the Mapbox user in a security dialog, some re-directed the user to the security setting page, and yet others enforced the phone call without any user interaction.
Before Lab3: We highlighted in the lecture the risks associated with refactoring [21], especially the behavior-preserving property of refactoring as practiced in students’ projects. The code changes intended to improve software internal qualities such as extensibility and maintainability depended much on test cases (both at unit testing level and at integration testing level) to achieve a sense of behavior preservation. However, test cases in students’ projects varied not only in the quantity but also in the coverage. As in previous teaching inventions, the refactoring risks were only identified but not mitigated by the instructor. We found that some of our students revised their test cases, some employed automated refactoring tools to reduce code transformation errors, and yet others did not address the risks at all even after they were identified by the instructor.

Before Lab4: We lectured the interdependencies between changing requirements and existing ones [21], pointing out risks specifically related to nonfunctional tradeoffs. For example, we introduced in the lecture the change scenario where the user wanted to toggle some address on Mapbox as private. This could lead to risks of security breach if the phone call displays the phone number which should not have been revealed.

The instructor interventions that we adopted in the spring 2016 semester were of risk type “software artifact dependencies” (cf. Table 3.3) but in different manifestations. While Lab 1 and Lab2 were dependency-induced risks of artifacts of the same kind,
Lab3 exploited traceability between artifacts of different kinds (e.g., source code and test cases). The artifact dependencies not only existed in the current development cycle, but also impact the future software evolution. This point was illuminated in Lab4.

Figure 3.4 Code Snippet introduced by the Instructor to help identify UI Mismatch Risks (spring 2016)
Table 3.5 Classification of Risk Items collected from the Spring 2016 Semester

<table>
<thead>
<tr>
<th>Category</th>
<th>Lab1</th>
<th></th>
<th>Lab2</th>
<th></th>
<th>Lab3</th>
<th></th>
<th>Lab4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of</td>
<td>Rank</td>
<td>m ± s. d</td>
<td># of</td>
<td>Rank</td>
<td>m ± s. d</td>
<td># of</td>
</tr>
<tr>
<td>C1</td>
<td>35</td>
<td>1.27 ± 1.15</td>
<td>36</td>
<td>1.97 ± 1.69</td>
<td>40</td>
<td>3.85 ± 1.47</td>
<td>34</td>
</tr>
<tr>
<td>C2</td>
<td>48</td>
<td>1.80 ± 1.31</td>
<td>49</td>
<td>2.51 ± 1.27</td>
<td>50</td>
<td>2.03 ± 0.96</td>
<td>47</td>
</tr>
<tr>
<td>C3</td>
<td>18</td>
<td>2.83 ± 1.23</td>
<td>14</td>
<td>2.96 ± 1.02</td>
<td>9</td>
<td>4.79 ± 0.75</td>
<td>8</td>
</tr>
<tr>
<td>C4</td>
<td>23</td>
<td>4.13 ± 1.09</td>
<td>24</td>
<td>4.08 ± 0.92</td>
<td>26</td>
<td>4.04 ± 1.46</td>
<td>36</td>
</tr>
<tr>
<td>Others</td>
<td>2</td>
<td>3.50 ± 0.71</td>
<td>2</td>
<td>3.50 ± 2.12</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.5 Ranking changes of Risk Categories (spring 2016)
To assess the effectiveness of the instructor interventions, we present in Table 3.5 and Figure 3.5 the risks submitted by the student teams in spring 2016, which one could relate to Table 3.4 and Figure 3.3 respectively. Although the raw number of risk item in different categories was comparable between the spring 2015 and spring 2016 semester, the temporal patterns shown in Figure 3.5 varied from those shown in Figure 3.3. The most noticeable difference, in our opinion, was the students’ perceived severity of the risk items in C3 (i.e., collaboratively-identified and individually-mitigated risks). While in both Figure 3.3 and Figure 3.5, C3’s severity dropped as the semester went along, when the dropping occurred was much earlier in Figure 3.5. We attributed such an earlier drop to the intervention activities that we adopted in the spring 2016 course. Interestingly, the instructor interventions seemed to have some side effects on C4 risks. In Figure 3.5, the triggering point for the C4 curve to increase was delayed when compared with the point in Figure 3.3. In our intervention design and execution, we intentionally chose C3 over C4 because we believed that having the instructor endorse one and only one mitigation strategy might bias the student, or worse, limit the students’ creativity for devising innovative solutions. Nevertheless, we realized that C3 and C4 were not always distinct. Our emphasis of lecturing C3 risk items might have helped the identification of C4 items, and for that reason, might have even positively influenced the mitigation of those items. Further studies are needed to test the hypothesis and other subtle observations and correlations one might spot in our data.
Chapter 4: Lean Learning of Risks in Students’ Agile Teams

In this chapter, we report the data collected and analyzed in two semesters of a junior-level software engineering course. In this our undergraduate students were structured in agile teams and were asked to deliver 4 increments per semester (62 students developed Eclipse plug-ins in one semester and 103 students developed Android apps in the other). Our results show that the students’ risk management strategies were both collaborative and lean. For example, our students frequently leveraged social media features like collaborative bookmarking and tutorial co-creation to address technology-centric risks, and as learning progressed, the students wasted little effort on non-actionable risks. Linking collaboration and waste-elimination provided additional insights into teaching a wider range of lean principles in agile settings, e.g., students should deliver as fast as possible the non-collaborative risk mitigations, but should decide as late as possible when facing interdependent mitigations.

4.1 Research Methodology

Lean thinking emerged in Japanese automotive industry in the 1940’s. It was born out of studying the rise of Toyota Motor Company. Shortly, it focuses on maximizing value and minimizing waste in production process. Lean means creating more value for the customers with fewer resources. Lean learning is the Learning and Development process of getting the right learning to the right audience, at the right time, in the right quantity to achieve perfect work flow, while minimizing waste and being flexible to change. Basically lean is the practice to eliminate wastes in production cycle on a broader
perspective. Therefore, lean principle should be coupled with the unambiguous definition of wastes in the software development activities. In software engineering, wastes can be defined in many ways at each phase in development cycle.

4.1.1 Agile Software Development and its limitations

Agile cares more about speed and also adapting to changes. The individuals are self-organizing and cross-functional. Agile maximizes the velocity by exploiting the communication among such individuals. The individuals work iteratively until they get the quality right. This iterative action results in faster development than a traditional development process such as waterfall because the sequential method cannot reverse the plan in the event of rescheduling or rework [27]. Agile application is only limited to implementation and early testing. Many agile practices, such as daily standup meeting, pair programming and test-first coding are exercised during implementation phase, and few agile practices bring mature domain knowledge forward in a way that reduces the cost and effort of later decisions during implementation.

Agile teams often confuse following a plan with planning. Sticking to a plan discourages adaptive thinking. Therefore, a “plan” might be useless. However, many practices show that thoughtful planning ahead can avoid much rework.

4.1.2 Lean in Agile Software Development

Although universal applicability of Lean principles is still the subject of debate, especially in knowledge work such as software development, most studies suggest that
Lean principles could be applied to virtually any system. Recently, Poppendieck et al. [29] claimed that, if Lean is thought of as a set of principles rather than practices, then applying lean concepts to product development and software engineering makes more sense and can lead to process and quality improvements.

In recent years, progress toward Lean Software Development has been mainly driven by industry pioneers familiar to some extent with Agile Software Development. Consequently, Lean Software Development by itself is an area which is not heavily touched upon or researched. Therefore, there is a lack of a developed understanding on which of its elements are positively applied in practice. Also, there is a gap of understanding ways to combine principles of Lean with agile methodologies. Some of the Agile Manifesto principles resemble Lean thinking. For example, simplicity - the art of maximizing the amount of work that need not be done - is essential; at regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly; further, our highest priority is to satisfy the customer through early and continuous delivery of valuable software. However, welcoming changing requirements, even late in development seems to be antagonistic to Lean thinking.

In this chapter we report our recent experiences on how our student teams applied lean principles to the risk management of their development projects. Our contributions lie in the analysis of students’ lean risk management in the agile software development.

Furthermore, by making use of a risk tree analysis, we were able to differentiate
between actionable (those risks which could be easily, quickly or effectively acted upon) and non-actionable risks. Moreover, our analysis suggested that towards the end of the iterations, students wasted little efforts on non-actionable risks and in a way eliminated wastes. Section 4.2 presents the previous work on agile and lean software development along with background information of actionable and non-actionable risk classification. Section 4.3 provides validation of the risk tree we constructed in section 4.2. Section 4.4 presents additional insights into teaching a wider range of lean principles in agile settings when we link collaboration with lean principles.

4.2 Related Work

4.2.1 Definition of Lean

Lean is the product of an incremental development of techniques that have gradually evolved since the late 1940s from the Toyota Production System (TPS). Although difficult to specify, five core principles, first introduced by Womack and Jones in 1996, are considered to govern Lean thinking:

1. **Value** is defined as everything for which a customer is willing to pay. Value, understood from a customer viewpoint, is the core concept of any Lean company. Its counterpart, waste, is everything that consumes resources but produces no value to the customer. Lean seeks continuous identification and removal of waste. Figure 4.1 illustrates the diagrammatic representation of the core principles.
2. **Value stream** is the optimized, end-to-end collection of actions required to bring a product from customer order to customer care, ensuring that each activity provides value.

3. **Flow** means that activities are organized as a continuous 'flow', eliminating discontinuities in the value stream and enabling smooth deliveries.

4. **Pull** implies that everything is built only when it is needed, making customer order/market the main decision driver. Thus, unnecessary intermediate and unfinished product inventories must be eliminated.

5. **Perfection**, or Kaizen in Japanese, pursues continuous enterprise-level improvement based on the concept that there is no end to striving for perfection.

![Figure 4.1 Core Principles of Lean](image-url)
4.2.2 Principles of Lean Software Development:

In Implementing Lean Software Development, Poppendieck et al. [29] show how the seven principles of lean manufacturing can be applied to optimize the whole IT value stream. These principles are:

1. **Eliminate waste.** Lean thinking advocates regard any activity that does not directly add value to the finished product as waste. The three biggest sources of waste in software development are the addition of unrequired features, project churn and crossing organizational boundaries (particularly between stakeholders and development teams). To reduce waste, it is critical that development teams be allowed to self-organize and operate in a manner that reflects the work they are trying to accomplish.

2. **Build in quality.** One’s process should not allow defects to occur in the first place, but when this is not possible one should work in such a way that you do a bit of work, validate it, fix any issues that one find, and then iterate. Inspecting after the fact, and queuing up defects to be fixed at some time in the future, is not as effective. Agile practices which build quality into your process include test driven development and non-solo development practices such as pair programming and modeling with others.
3. **Create knowledge.** Planning is useful, but learning is essential. One wants to promote strategies, such as iterative development, that help teams discover what stakeholders really want and act on that knowledge. It is also important for a team to regularly reflect on what they are doing and then act to improve their approach.

4. **Defer commitment.** It is not necessary to start software development by defining a complete specification, and in fact that appears to be a questionable strategy at best. One can support the business effectively through flexible architectures that are change tolerant and by scheduling irreversible decisions to the last possible moment. Frequently, deferring commitment requires the ability to closely couple end-to-end business scenarios to capabilities developed in multiple applications by multiple projects.

5. **Deliver quickly.** It is possible to deliver high-quality systems quickly. By limiting the work of a team to its capacity, which is reflected by the team’s velocity (this is the number of “points” of functionality which a team delivers each iteration), you can establish a reliable and repeatable flow of work. An effective organization does not demand teams do more than they are capable of; but instead asks them to self-organize and determine what they can accomplish. Constraining these teams to delivering potentially shippable solutions on a regular basis motivates them to stay focused on continuously adding value.

6. **Respect people.** The Poppendiecks [29] also observe that sustainable advantage is gained from engaged, thinking people. The implication is that you need a lean
governance strategy that focuses on motivating and enabling IT teams—not on controlling them.

7. **Optimize the whole.** If one wants to be effective at a solution, one must look at the bigger picture. One needs to understand the high-level business processes that individual projects support—processes that often cross multiple systems. One needs to manage programs of interrelated systems so that one can deliver a complete product to stakeholders. Figure 4.2 illustrates the diagrammatic representation on the seven principles of Lean Software Development.

![Figure 4.2 Seven principles of lean applied to IT stream](image)

**4.2.3 Agile and Lean Software Development**

The potential exhibited by Lean in terms of productivity, time-to-market, product quality and customer satisfaction have aroused the interest of its application in software
industry. Although the application of Lean core principles to the knowledge work like software industry is still under discussion, most of the researchers agree that the practices and its tools can be adapted to make the processes better. Lean principles could be virtually applied to any domain. Lean thinking in software development started as early as the 1990s with concepts such as Lean software production. However, Lean Software Development is known today through its promotion by the Agile community, where it has progressively acquired an identity of its own. However, the particularities of software products, such as its value proposition and malleability, open new opportunities for combining Agile and Lean in a software domain. Thus, Poppendiecks [29] considered Lean thinking a “platform upon which to build agile software development practices.” Similarly, Brandao de Souza [30] said that “if we avoid nit-picking based on semantics, it is easy to come up with a many-to-many mapping [between the two approaches]”, and Ambler et al. [31] argue that Agile and Lean complement each other by addressing different components of systems development. Recent studies reviewing the body of knowledge of Lean in software development reflect the freshness of the topic. Based on 30 experience reports, Turk et al. [32] examined the purposes of applying Lean in Agile Software Development identifying six strategies: non-purposeful combination, using Lean to interact with other business areas while keeping Agile in software development, directly using Lean in software development processes to facilitate the adoption of Agile, using Lean in software development to improve Agile processes, transforming from Agile to Lean, and synchronizing Agile and Lean. Although Turk et al.’s [32] study offers significant insights into why Lean is applied with Agile Software Development, it
does not deeply explore how the combination is actually implemented. Most published knowledge in Lean Software Development are in form of books, which provide a diversity of interpretations of what Lean Software Development is and how it should be used. Perhaps, the most widely acknowledged generalization of Lean Software Development principles are the seven principles compiled by Poppendieck [29]: eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build in integrity, and see the whole. In this chapter, we discuss our undergraduate software engineering classes where students working in agile development teams applied elimination of waste—a lean principle to their collaborative risk management.

4.3 Risk Tree Analysis

Fault Tree Analysis (FTA) is a common probabilistic risk assessment technique that enables investigating safety and reliability of systems. FTA is a deductive top-down backward search method utilizing Boolean logic, which helps to identify the basic (causal) events that contribute to a hazardous top event.

Risk tree (constructed on the basis of fault trees) analysis and assessment can simply be described as an analytical technique. It is a graphical model of various combinations of risks that result in the occurrence of the predefined undesired event. To analyze using risk tree, it is necessary to specify the undesired state of the system. This state may be the failure of the system or of a subsystem. Then a list is made of all the
possible ways in which these events can occur. Each of the possible ways is then examined independently to find out how it can occur. Risk tree possesses many events. The lowest level events are called primary events. In the middle, intermediate events exist and the highest level event is called the top event. Also, all the events are connected in a tree by gates that show the relationship between successive levels of the tree.

Generally, the risk tree is represented by three types of graphic symbols: logic gates, events and transfer triangles. Risk tree depicts the logical interrelationships of the basic events that lead to the top events. We use fault tree analysis technique to represent our risk tree.
In the similar way, in this chapter we construct a risk trees (illustrated in Figure 4.4, 4.5, 4.6) that are based on our students’ risks – those that our students have come
across during their software development.

Figure 4.4: Risk Tree Representation for Unsatisfactory Outcomes in Requirements.
Figure 4.5 Risk Tree Representation for Unsatisfactory Outcomes in Scheduling
Figure 4.6 Risk Tree Representation for Unsatisfactory Outcome in Developing Proper Software

4.3.1 Binary Decision Diagrams

A Binary Decision Diagram is a directed acyclic graph, i.e. all paths through the Binary Decision Diagram are in one direction and no loops can exist. The Binary Decision Diagram is composed of terminal and non-terminal vertices (nodes) which are connected by branches. Terminal vertices correspond to the final state of the system, failure (1) or success (0), and non-terminal vertices correspond to the basic events of the fault tree. Each non-terminal vertex has a 1 branch, which represents basic event
occurrence, and a 0 branch, which represents basic event non-occurrence.

![Binary Decision Diagram](image)

Figure 4.7 The Binary Decision Diagram for the Fault Tree in Figure 4.3

4.3.2. Minimal Cut Sets

Minimal cut set analysis is a mathematical technique for manipulating the logic structure of a fault tree to identify all combinations of basic events that result in the occurrence of the top event. These basic event combinations, called cut sets, are then reduced to identify those “minimal” cut sets, which contain the minimum sets of events
necessary and sufficient to cause the top event. The logical structure of the original fault tree is mathematically transformed, using the rules of Boolean Algebra, into an equivalent minimal cut set fault tree. The transformed fault tree is mathematically and logically equivalent to the original fault tree, but the minimal cut set form is more amenable to quantification. The transformation process also ensures that any single event that appears repeatedly in various branches of the fault tree is properly accounted for. This methodology is applicable to all fault trees, regardless of size or complexity, that satisfy the following conditions.

All failures are binary in nature (components are either working or failed). Transition between working and failed states occurs instantaneously (no time delays). All component failures are statistically independent.
Figure 4.8 Minimal Cut Sets for each Event for the Fault Tree in Figure 4.3

4.3.3 Fault Tree Minimal Cut Sets

Minimal Cut Sets, as the core of Fault Tree Analysis, have been paid much attention in terms of their generation algorithm. Since the fault tree is often of large scale and great complexity, many methods are developed to analyze it for the purpose of efficiency and accuracy.

However, the traditional BDD method requires the basic event order to be
carefully chosen, which is an NP-complete problem. J. D. Andrews et al. [33] described a new method for the fault tree conversion to the Binary Decision Diagram which was called Component Connection Method. This method built the Binary Decision Diagram structure no matter what the order of basic events was. The method was improved by BDD weight comparison. Based on the component connection method, there is a new BDD structure of FTA with improved connection rules.

The qualitative ranking of importance would assume that events with more number of cut sets are more complex to be resolved than the ones with small number of cut sets. For example, the risk on the top of the tree is more complex to be acted upon as there will be many basic, intermediate events which contribute for the top risk to occur. Therefore, we divided our tree into three different categories based on the cut sets generated for each event. We illustrate our three categories as follows:

**D1 (Highly actionable risks):** A majority of risk items in this category are the basic events from the risk tree which are found to be easy or quick to act upon. Some of the examples would be collaborative book marking, tutorial co-creation. These are observed to be the doable mitigation strategies for the technology centric risks. Students’ response to these strategies was more positive and quick compared to just learning the technology all by themselves. Also, writing more number of test cases would fall into this category.
D2(Averagely actionable risks): A representative risk item in this category was a documentation risk, which could be because of lack of proper design document, improper guidelines and unclear documentation. Acting on these risks require a little more efforts than the previously mentioned D1 risks. However, it is found that our students were eventually successful in mitigating these risks at the end of the semester.

D3(Non-actionable risks): A sample risk item was team-wide communication and coordination. This is considered non-actionable not just because of more number of mitigation strategies are required but also involving more team-wide efforts to act upon these risks. A lot of team-wide efforts and commitment is required for a non-actionable risk. Figure 4.9 illustrates the combined fault trees (mentioned above) and the minimal cut set value for each node.
Figure 4.9 The minimal cut sets for the combined fault trees
4.4 Validation of the Risk Tree

Based on the new Minimal Cut Set generation algorithm [34], every node in the risk tree (Figure 4.9) is calculated with its minimal cut sets. We observed that from both spring 2015 and spring 2016 semesters, events with small number of cut sets (say 0,1,2) are easily acted upon. For example, improper training of technology centric risks is easily acted upon by the student teams (through collaborative book marking and tutorial co-creation). As the number of cut sets increased, the risk eventually became more non-actionable. For example, the risk of improper documentation. Based on the observations, the events with 0-2 cut sets are categorized as highly actionable. Those intermediate events with 3-5 cut sets are averagely actionable and the nearly top events with more than 5 minimal cut sets are categorized as non-actionable. Figures 4.10 and 4.11 show the risk tree validation of both spring 2015 and 2016 classes. We observe that most of the actionable risks are mitigated and maximum number of non-actionable risks are left un-mitigated. Also, there is a significant decrease of the non-actionable risks as the increments progress. This evidence reports that there is a reduction in efforts spent on wastes.
Figure 4.10 Validation of Risk Tree for Spring 2015 Class

Figure 4.11 Validation of Risk Tree for Spring 2016 Class
4.5 New insights on linking collaboration with waste elimination

On linking the collaboration with lean principles, we gain additional insights of lean learning. The Figure 4.12 illustrates our observations and insights from our students’ agile teams whose risk management strategies are not only inherently collaborative but also lean.

<table>
<thead>
<tr>
<th>Linking collaboration with waste elimination</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Team empowerment</td>
<td>Labor-intensive</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>Amplify learning</td>
<td>Mutual Recursion</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>Continuous Improvement</td>
<td>Moving towards flow</td>
<td>Time-consuming</td>
</tr>
<tr>
<td></td>
<td>Deliver as soon as possible</td>
<td></td>
<td>Decide as late as possible</td>
</tr>
<tr>
<td>C4</td>
<td>Amplify learning</td>
<td></td>
<td>Cyclic Dependency</td>
</tr>
</tbody>
</table>

Figure 4.12 New insights on linking collaboration with lean principles

For example, when students’ teams faced the technology centric risks (belonging to C1 category), lack of knowledge in Java or eclipse, the strategy most of them adopted initially was to learn the technology and improve the skills individually. This is highly
non-actionable as the scope of this action could be very large. Students made mistakes in understanding which areas to be touched upon in order to make their contribution significant in a particular increment. We observed that our students not only spent much efforts and time in working towards the strategy but also could not achieve mitigating the desired outcome (e.g., learning Java or becoming familiar with the IDE). On the other hand, we also found that student teams when faced lack of testing to be a severe risk, they were able to act upon it immediately (highly actionable) as the mitigation strategy of writing tests as and when the code is written is highly actionable. This spontaneity has not only seen the risk being mitigated but also empowered the team in terms of better risk management and in delivering a better error-prone increment.

Another observation from the student teams is that whenever there is a risk item that belongs to C3-D3 category, for example requirements mismatch such as UI related risks, we find that acting upon these category of risks is not only time consuming but also labor intensive. For example, in order to mitigate the requirements mismatch or ambiguity in requirements, few student teams opted for better learning of “Requirements Engineering”. This proved not only inefficient in time and efforts but also in mitigating the risk. Each one of them had their own perspective about the materials and there were many brainstorming arguments that went along among the student teams. However, the risk items in C3-D1 gave a good scope for continuous improvement of the student teams. For example, for the risk of ambiguity of requirements, acquiring better requirements from the beginning of the project is not only highly actionable and can reduce risk but
also enable continuous improvement that can perform vital task for proper development. In order to mitigate the requirements mismatch, students came up with a mitigation strategy of discussing with the instructor for further understanding. This is highly actionable and can be put in to action at the earliest. This quick action of the students gave them a scope of getting a feedback as quickly as possible. This immediate feedback gave them a chance to either improve their mitigation strategy if it did not work or continue mitigating such risks in the same way if they happen to see a positive response. Delivering as soon as possible helped students to get immediate feedback for continuous improvement. It is also observed that most of C4-D3 and C2-D3 category risks are categorized as cyclic dependent or mutually recursive. For example, most of the C4 risks are either teams having difficulty in distributing the work load or integrating the individual work. When non-actionable risks fall into these category, we see a lot of dependence not between the members of the team but also between the modules involved in the increment.
Chapter 5: Risk as a self-organizing mechanism in Ganga Software Agile Team – High Energy Physics’ Application

In this chapter, we report the data collected and analyzed from the Ganga software development team which follows agile practices to deliver its increments. The data is collected from the team’s TWiki(s), developers’ mailing lists, user-developers’ mailing lists and weekly vidyo calls and interactions with the team. Through the interactions and timely exposure to the data, we found that the risk management in Ganga not only follows some of the similar trends that our students’ agile teams do (though in a different form), but also observed that as increments progress, team became more lean in risk identification processes. For example: Team Ganga introduced Twiki(s) that made risks more visible to the team while leaving the actual mitigations to the individual’s capacity. Also, the Ganga team introduced the concept of “Threshold of Success” to help them understand and work on the immediate, for example high impact risks, thereby making efficient use of time and efforts in achieving a successful increment.

5.1 Introduction to Ganga:

Scientific users are driven by a requirement to get computing results quickly and with minimal effort. These users have a wide variety of computing resources at their disposal, namely workstations, batch systems, and computing grids. Each of these computing resources has a unique user interface, and users often face the burden of continually reconfiguring their applications in order to take the advantage of these diverse
systems. With this situation, it is so apparent that the need of the hour is an end-user tool that hides these complexities and enables users to concentrate on their science and not care much about the computation.

**Ganga** is one such easy-to-use frontend for the configuration, execution, and management of computational tasks. Ganga allows users to easily scale up their analyses from development on a local workstation or batch system to full scale analysis on the Grid. By providing a familiar and consistent user interface to all of the resource types, Ganga allows users to get their computing results quickly. It utilizes all of the available resources. Both ATLAS and LHCb (biggest experiments at Large Hadron Collider) collaborations that develop applications within a common framework called Gaudi/Athena aim to exploit potential of grid for a large-scale, data-intensive distributed computing. It started as a project that serves as a Grid user interface that helps in data analysis within the biggest experiments (for example ATLAS and LHCb) in High Energy Physics where large communities of physicists need access to Grid resources for data mining and simulation tasks. It also simplified the management of analysis and production jobs for end-user physicists by developing tools for accessing Grid services. Ganga provides a homogeneous environment for processing data on heterogeneous resources. We give examples from High Energy Physics, demonstrating how an analysis can be developed on a local system and then transparently moved to a Grid system for processing of all available data.
Ganga provides a simple but flexible and a consistent programming interface that can be used either interactively at the Python prompt, through a Graphical User Interface (GUI) or programmatically in scripts. The implementation uses an object-oriented design in Python [35]. The concept of a job component is vital as it not only contains the full description of a computational task which includes the code to execute; input data for processing; data produced by the application; the specification of the required processing environment; post-processing tasks; but also the metadata for bookkeeping. The purpose of Ganga can then be seen as making it easy for a user to create, submit and monitor the progress of jobs. Ganga keeps track of all jobs and their status. This is achieved through a repository that archives all information between all the independent Ganga sessions. It is also possible to switch between executing a job on a local computer and executing on the Grid by changing a single parameter of a job object. This simplifies the progression from rapid prototyping on a local computer, to small-scale tests on a local batch system, to the analysis of a large dataset using Grid resources [36]. In Ganga, the user has programmatic access through an Application Programming Interface (API), and has access to applications locally for quick turnaround during development.

5.1.1 Ganga User Interfaces

Ganga is a flexible tool with three different user interfaces. The graphical user interface (GUI) provides a simple point-and-click approach to configuring, submitting, monitoring, and otherwise managing jobs. Depicted in Figure 5.1, the GUI makes it simple for users to monitor the progress of many running jobs.
The command line interface (CLI) of Ganga is generally the most commonly used. The CLI provides a powerful interface to quickly manipulate job objects by providing users with an interactive Python prompt. Finally, Ganga provides an API to the data and functions available in the CLI. Using the API, users can write Python scripts to execute automated workflows.

![CLI, GUI, and Scripting Interfaces of Ganga](image)

Figure 5.1 Three Different User Interfaces of Ganga: CLIP, GUI, Scripting

5.1.2 Development and Testing

One of the main strengths of Ganga is its robustness, which is ensured by an organized release procedure and extensive testing. In order to create stable releases, Ganga developers rotate through six-week terms as release manager. In general, the job of the release manager is quite simple: a cut-off date for a pre-release is announced, the
release manager collects tag for the components, and finally the pre-release is built. Each pre-release undergoes extensive testing; with tests of both core and extension-specific functionalities, the testing framework includes approximately 500 test cases. In general, all core bugs submitted to the Savannah tracking system get a test case [37]; these validate the bug fixes and help prevent regressions. After approximately 24 hours of testing, the developers inspect the test results; if all developers are satisfied, the stable release is made. Generally, a new stable release is produced bi-weekly.

5.2 Ganga – Agile Software Development Team

GANGA has a large user base and is in active development. It consists of 9 developers who are also the physicists working on the various High Energy Physics’ experiments. The Team Ganga follows agile methodology for its development. Requirements are collected in the form of use-cases. This is done through following informal and un-facilitated brainstorming sessions and discussions with physicists who had the task of writing the simulation tasks, reconstruction and analysis algorithms. In a few cases there was only a vague and unclear knowledge of the requirements to start with, but this knowledge improved greatly with time and also helped in getting more insights as time progressed. This helps to underline the importance of continuously involving the users in the specification of design, to ensure producing a software closely matches their needs. The final software is approached via incremental release, adding functionality at each release according to the feedback and priorities given by the users. This can be done only by using an architecture-driven approach, i.e. to identify a series of
components with definite functionality and well defined interfaces. The team communicates with each other weekly through vidyo calling. They discuss about week’s work that each one of them had done and also discuss about the road blocks of the development. There is a scrum master who facilitates the meeting for an efficient management of time. The rest of the communication (for example, discussion about a hotfix issue, details about cleaning up the code files, bug fixes or updating with merge details etc.) among the developers is done through the developer’s mailing lists shown in Figure 5.3. A sample of this mailing list is shown in Figure 5.2 where the developers are informed about the bi-weekly release. Moreover, team Ganga also has an E-group for its developers where they post discussions for the other registered users to participate.

![E-group](image)

**Figure 5.2 Ganga Developers E-Group**
Figure 5.3 Developers’ Mailing Lists

In addition to this, these developers also attend the users’ problems (for example, unable to locate the IConversionSvc interface, backend issues in submitting a job, reduced grid availability, problems related to job submissions, etc.) and questions (for example, how to run ganga locally, how to access dirac-dms storage, etc.) through the user-developers’ mailing lists shown in Figure 5.4.
Same events with GaudiExec & GaussSplitter

Andrea Merli <andrea.merli@cern.ch>
Mon 10/24/2016, 4:19 AM
lhcb-distributed-analysis (User support for distributed analysis in LHCB) <lhcb-distributed-analysis@cern.ch>

This message has a digital signature, but it wasn't verified because the S/MIME control isn't currently supported for your browser or platform.

Dear experts,

I have 2 different jobs with 2 subjobs each. The former uses a private version of Gauss with GaudiExec, the latter a default Gauss version. Both use GaussSplitter:

```c
GaussSplitter();
  nnumber0fJobs = 2;
  eventsPerJob = 50,
  firstEventNumber = 0
```

as splitter. In the option file I setup the number of events to generate with

```
LHCbApp().EvtMax = 100
```

If I run the default version of Gauss then I get 50 events per subjob correctly uncorrelated. Don’t know why the NextNext event (spillover event) seems to have the same seeds between subjobs, but the main event have different seeds so it is fine.

If I run the same option file with GaudiExec then I get 100 events per subjob. 50 of each are totally correlated among the 2 subjobs since they have the same seeds.

I know I can easily avoid the problem if I change

```
LHCbApp().EvtMax = 100  -->  LHCbApp().EvtMax = 50
```

when I run with GaudiExec, but I’m wondering if this is a bug or a known behavior.

Thanks,

Andrea

Figure 5.4 Ganga User-Developer Mailing Lists

Team Ganga also has FAQ pages (shown in Figure 5.5), Question and Answers pages (shown in Figure 5.6), in order to attend the users and other new comers’ problems and questions. The registered user can search for the desired tag, make changes to the desired documentation, can create a new topic and many more. There is also a notification web service which makes it convenient for the users to not to get back to the docs each time there is a change but they get notified automatically by an e-mail when topics change in these FAQ webpages.
Figure 5.5 Ganga LHCb FAQs page

In these Questions and Answers pages, the registered user gets a chance to vote the optimal solution to the question raised. Also, they can provide various other solutions by making use of the comment section available on this page. The registered user can ask a question, can answer the raised questions. There are also tags available where every question is tagged with the significant words in that issue so that the users can search for the required issue easily.
5.3 Ganga – Risk Management

After deep examination of Team Ganga’s risk management methodologies, it is quite evident that it has borrowed tools and techniques from the SEI to aid in risk
management process. They are presented here, along with how they were accustomed to fit the needs of the project, to better understand the experiences that the team had with respect to managing the risks on their project. Since risk management was viewed as a continuous event, the team applied the SEI’s Risk Management Paradigm and the phases are presented in Figure 5.7. The steps of identification, analysis, mitigation, tracking and control occurred continuously and run concurrently - e.g., as new risks were identified and analyzed, other risks are being mitigated and tracked - and iteratively - e.g., one mitigation plan might yield new risks.

![Figure 5.7 The SEI Risk Management Paradigm](image)

Communication is central to this risk management paradigm. Therefore, team members were encouraged to communicate the status of the risk and its mitigated level to project (clients and team members) during status and review meetings.
The Ganga team, as mentioned earlier, followed Scrum for their software development process. As agile processes are inherently risk driven, the team started out managing risks implicitly. However, the physicists who are not purely the computer scientists kept themselves informed with literature and information about the software developments trends and practices through various ways (literature survey, mentors) and realized early on the fact that implicit risk management is just not sufficient. They realized the importance of explicitly addressing risks other than those prescribed by Scrum. The team started managing risks more explicitly by creating a risk manager role (for example the team lead). This person was responsible for helping the team identify the potential risks. He also has had the responsibility of documenting the identified risks. However, no tasks for mitigation strategies were defined at that point. This person also administered the high level risks such as gain or loss of a team member and the time contributed by them. The low level ones are dealt within in a more ad-hoc way among the team members. Risks were identified through informal, and un-facilitated, brainstorming sessions. The captured risks were placed in a spreadsheet and the risk manager was responsible for prioritizing the risks based on their expected impacts and timeframes. A very general risk mitigation task was included in iteration planning sessions for prioritization with other project tasks. This was however, later realized that it was not an effective approach in dealing with risks. It was difficult for team members to understand what the mitigation task involved how it supported the project and what impact it would make if not followed. As a result, the task either did not receive a high enough priority rating to make it into an iteration plan, or it was the most likely task to be dropped if
problems were encountered in other tasks. After a couple of iterations, the team realized a change in their risk management approach was required. This realization was fueled by several iterations that passed without proper mitigation of the identified risks. As a result, the team decided to facilitate a small team software risk evaluation. The goal of this evaluation was to identify and analyze those risks that might stop the team from reaching project success.

They used a technique called the Threshold of Success (ToS) where the team was able to define a set of minimum conditions (objectives) that needed to be met by projects’ end for it to be called a success and against which those risks were identified. If the team fails to meet even one of the objectives, then the particular increment may be under a threat.

The easiest way of creating a threshold of success is to create a picture of failure and then convert it into picture of success. For example, through the interactions with the development team it was known that losing man power is a high end risk. So the picture of failure for this can be “Team members are dissatisfied or bored with their jobs”. The threshold of success for this failure might look something like this: “All the employees gave a score above 5 on an average in the employee job satisfaction survey taken bi-weekly”. These risks were documented on the team’s TWiki to allow for easy access for viewing and frequent updates. TWiki is a web-based collaboration platform which is easy to use and a full-featured open communication environment. People anywhere on the Web or on an intranet can meet and rich Web text, images, and online multimedia are
easily shared. Moreover, documents and other files can be uploaded and downloaded all that is needed is a Web browser and any available network connection. The Twiki page of the CERN community is illustrated in Figure 5.8.

![TWiki Page of the CERN Community](image)

Figure 5.8 TWiki Page of the CERN Community

The final process the team followed and the way the risk management steps mapped to Scrum is depicted in figure 5.9.
The risk manager then documented the components of the risks – (source(s), condition, consequence, impact, timeframe, and probability) - to facilitate the creation of action plans and mitigation strategies. The risk manager then worked with other team members to identify mitigation strategies that might help in lowering the probability of its occurrence, reduce its impact, eliminate the source or consequences, or extend the time frame in which the risk needed to be addressed.
During iteration planning, the primary mitigation strategies were included in the sprint backlog for prioritization. This sometimes required multiple tasks per mitigation strategy. The reason for multiple tasks was to break up the amount of work required to mitigate a risk such that a portion of a mitigation strategy could be performed in one iteration and additional portions could be completed in the following iterations. This made it more likely for mitigation strategy tasks to be included in an iteration since they could be completed in small chunks. If a mitigation strategy was complicated and required a significant amount of time, it might be considered as too much effort for a single sprint because it could take too much time away from other aspects of the project that were required to show progress to the customers. At the end of each iteration, the team reviewed the triggers for risks that had timeframes associated with the end of the current sprint. If a trigger was triggered, then the team would take the appropriate next steps, such as moving to a backup strategy or closing the risk because the trigger indicated that the risk was mitigated. A sample of an Action Plan is illustrated in Figure 5.10.
Figure 5.10 An Example of an Action Plan

Once risks were mitigated, the risk list was updated. Also at the end of each iteration, the team would spend 10 to 15 minutes in the iteration review meeting discussing any changes in risks or any new risks that any team member had identified during the iteration. The risk list would be updated accordingly, and the risk manager would facilitate a quick re-prioritization of the list. For example; If the top 5 risks changed resulting in risks in the top 5 that did not have defined action plans, then the team would allocate time in the coming iteration to devise the mitigation strategies for the new risks and define the action plans. This was a slight drawback. With this method it would be possible to identify a new risk that was critical and should be mitigated immediately, but the mitigation of the risk would have to wait until the following iteration. The current iteration would be used to define the mitigation tasks that could then be prioritized during the planning of the following iteration.
5.4 Observations

Through the course of the Ganga team’s project, the team tried multiple techniques and constantly focused on process improvement through periodic reflections. Some of the trends that the Ganga Team followed are in line with our students’ agile teams. The risk champion was responsible for ensuring any necessary documentation is up to date, making sure that the mitigation strategies are being revisited at the end of each sprint and also that specific amount of time is spent in identifying new risks and the mitigation strategies are appropriately designed. In addition to this, to effectively manage the risks, the team had proper and unambiguous documentation also named as “Twiki(s)”.

These Twiki(s) are made available to all so that the developers are made aware of the risks in the first place and the mitigation strategies are left to them to solve. For example: The risk of inefficient and error prone output of a job is well addressed in the Twiki by providing information about the pros and cons of setting the output files using different methods (example: MassStorageFile or DiracFile). It is left to the individual developers to decide on which one to pick depending on their choice. This form of risk management is similar to the instructor intervention in the C3 category of our students’ agile teams. A sample Twiki page is illustrated in Figure 5.11.
Regarding the pros and cons of output files

Figure 5.11 TWiki – Regarding the pros and cons of output files (https://twiki.cern.ch)

Also we find that the risks were previously documented in excel spreadsheets. This is an evidence to show that the team is not bothered about the waste that is generated but all that they want is completeness in the work, but later threshold of success technique was introduced, which actually cared only about the immediate risks or those which have impact on the project. The attention to risks increased as the sprint meets included the analysis of the minimum number of conditions that has to be met for the project to be successful and the risks against them were also discussed. Because of this mapping of risk management into sprint made an efficient and effective usage of time and efforts of the team thereby reducing the waste generated. There are also multiple ways to prioritize and reprioritize risk items. Team Ganga found that the use of the multi-voting
technique was particularly effective in that the process was quick, easy to follow and included the opinions of all the team members within the prioritization effort. The swiftness allowed for the reprioritization of risks to be easily added as another activity into the regular Sprint planning meetings. The participation of all team members helped to ensure buy-in for the risk management activities and for the inclusion of mitigation strategy tasks in the team’s sprint backlogs.
Chapter 6: Conclusion

Risk is a fundamental vehicle for a software development project to progress and risk management is regarded as a critical skill in the software industry. In this thesis, we aim to observe how risk acts as a mechanism in self-organizing agile teams. We report in this work our two semesters’ of teaching of a junior-level software engineering course, where students were grouped in agile teams to use risks to drive their milestone deliverables. Our analysis of the students’ perceptions of risks uncovered some new items that the previous literature did not cover. These include software artifact dependencies and task dependencies, which if not properly managed, could lead to project failures. We further developed instructor interventions and implemented them in the spring 2016 offering of the course. In particular, we focused on making the collaboratively-identified risks more visible to the entire class while leaving the actual mitigations to individuals. Following an evidence-based approach [19], our examination of the temporal patterns of risk severity showed the effectiveness of our instructor interventions. Also, the observations showed that our student agile teams followed some of the lean principles during their iterations. Our work reported some new insights on how lean learning is applied to risk management activities. We also reported about the Ganga software development team where we have found that the risk management strategies followed by this High Energy Physics’ application are in line with some of the interesting trends as our student teams do. We also reported how waste is considered and treated by this agile team.
Future work will address the limitations of our current work. For example, we used only the frequency of occurrence to qualify the top-ranked risks in Table 3.3. Incorporating the temporal dimension may offer new insights. For example, many C1 risks in both the spring 2015 and spring 2016 semesters were centered on inadequate technical background. As can be seen in Figure 3.3 and Figure 3.5, these risks were addressed in a rather fast manner as the students became familiar with the subject system, the programming language, and the development environment. Calculating the frequency of occurrence of the entire semester would not allow this type of risks to surface, though doing so at specific phase of the course might lead to refined lists of top-ranked risks, thereby leading to more effective instructor interventions. Another limitation relates to our risk classification, where certain items were difficult to be cleanly categorized (cf. Table 3.4 and Table 3.5).

While having more classification categories may be counterproductive, it may be worth analyzing the dynamics of the risk categories and/or the items themselves. Finally, our interventions focused primarily on C3 risks. Our results suggest that C4 risks tended not to emerge in the beginning of the semester, but once they were identified, their likelihood of occurrence and potential negative impacts tended to become significantly greater if not managed well. C2 risks, on the other hand, maintained a fluctuating temporal pattern in both of our semesters. These results may indicate that future work can focus on recognizing those categories of risks more
effectively and developing innovative curriculum materials and activities to better handle them.

In context of Ganga risk management, we reported how the explicit risk management of this agile team is in line with our students’ agile teams. However, the future work will address various categories of risks that are exclusive for the Ganga team and also how the team adopts mitigation strategies to face them. In addition to this, future work will address how the lean principles are applied in risk mitigation strategies adopted by the team.
References


2. Ville Ylimannela - A model for risk management in agile software development - Tampere University of Technology.


36. A. Streit, et al., UNICORE – From Project Results to Production Grids, Grid Computing: