SOFTWARE RELEASE PLANNING FOR EVOLVING SYSTEMS

PhD Research Proposal

by

SALIU, Moshood Omolade

Department of Computer Science
University of Calgary, Calgary, Alberta
CANADA

November 2005

Abstract

Large-scale software systems continually evolve because of the needs to extend functionality and correct errors discovered during operation. A major problem faced by companies developing or maintaining such large and complex systems is determining what features should be in the next sequence of releases. Release planning involves decision making about what new features and changes to implement in which releases of a software system. Release planning for evolving systems is challenging, because characteristics of the components making up the existing systems and design decisions implemented in them restrict the type of features they can accommodate during evolution. These characteristics also affect the amount of resources required for their implementation.

The claim of this proposed research is that, features scheduled during software evolution by factoring the influence of target components of existing system where the features will be integrated into release planning decision process are most likely to be successfully implemented within available resources.

This research presents a release planning methodology that characterizes and evaluates the Difficulty of Modification (DoM) of components of existing systems, and combines this evaluation with features-driven impact analysis (FDIA) to assess the impact of the components on features selection and scheduling decisions. The methodology would form part of a hybrid planning approach that aims to combine the strength of computational intelligence with the knowledge and experience of human experts in building the core of a decision support engine. Concepts from fuzzy theory will be employed to model vagueness in the assessment of interdependencies among the features.

This approach to release is fundamentally different from existing approaches in its detailed incorporation of information from the target operational environment of proposed features, flexibility in tailoring the methodology to different project contexts, and mitigating the weaknesses of using either pure computational algorithms or pure ad hoc planning. To evaluate this research, we would perform empirical studies to assess the utility (or lack thereof) of incorporating existing system information into the decision process.

Table of Contents

1. Introduction

1.1 Software Release Planning

A program only stops to evolve when it is no longer used [22]. According to "Lehman's laws of software evolution" [29], software systems must be continually adapted, or they become progressively less satisfactory to use in their environment. Therefore, irrespective of the degree of success of an operational system, it has a tendency to evolve because of the need to extend functionality of the system by adding new features or correcting errors that are discovered during operation of the software. Most of the features originate from diverse stakeholders that require their needs to be met despite resource and risk constraints. These features (or requirements) could represent customer wishes derived from perceived market need, or product requirements that the company developing the product consider worthwhile to pursue [36]. In such large systems, there are schedule constraints and the number of features requested typically exceeds the available resources. The very essence of incremental software development is to address this problem by allowing compromises of providing different features at different release points.

Incremental software development offers sequential releases of software systems with additive functionalities in each increment. Thus, each increment is a collection of features that form a complete system that would be of value to the customer. This approach allows customers to receive parts of a system early – a situation that allows for creating early value and to provide early feedback. New releases also serve to fix defects detected in former product variants. Meanwhile, a major problem faced by companies developing or maintaining large and complex systems is deciding which features should go into which releases of the software [4], considering that only a subset of these features can be implemented in the next release [36].

Release planning (RP) for incremental software development involves decision making about which new features or changes should be implemented during which release of the software. RP is a key determining factor of the success of a software product, and it also determines which customer gets what feature at what particular time [8]. Because feature selection and

assignment to releases must generally be preceded by feature prioritization [8], RP generalizes the mere prioritization of features [49].

Several reasons contribute to the need for RP:

- **Mismatch Customer Satisfaction:** Since the Standish Group [57] began its CHAOS research in 1994, they have concentrated on tracking project success and failure. In the 2002 report of the Standish Group, it was observed that mismatch customer satisfaction with the functionality of delivered software is still one of the main reasons that many software projects fail to achieve their stated objectives. We have a firm believe that mismatch would definitely occur if the stakeholders are not sufficiently involved in selection of features for implementation. In order to develop a system that matches the customers need, clear requirements must be defined and customers must be given the chance to specify their release preferences for desired features.
- **Budget and Schedule Overrun:** In 1995, The Standish Group report showed that the average US software project overran its scheduled time by 190%, its budgeted costs by 222%, and delivered only 60% of the planned functionality. According to their later reports, improving trends have been noticed. Even at that, the 2003 Standish Group Chaos still shows a relatively low success rate with an estimated 70% projects either failed completely or challenged (i.e. they were completed with exceeded budget and schedule estimates, and had less functionality than originally requested). Even though planning releases may not "always" guarantee success, refusing to plan is definitely a recipe for failure (If you fail to plan, you plan to fail!). A good release plan is necessary to ensure the feasibility of projects in terms of the existing resource capacities available. With sound RP, less important features are left until later and so, if the time or budget is not sufficient, the least important features are the ones most likely to be omitted
- **State of the Practice:** RP is typically done ad hoc and not based on sound models and methodologies. This is even the case when planning for several hundreds of features. Karlsson and Ryan [27] and Lehtola *et al.* [31] observed during their industrial studies that most organizations select requirements informally. The project

management process area as defined in CMMI [11] covers activities related to planning, monitoring, and controlling projects. According to the CMMI, the purpose of project planning is to establish and maintain plans that define project activities. These include developing the project plan, involving stakeholders appropriately, obtaining commitment to the plan, and maintaining the plan. For all these processes, there are existing guidelines and standards (e.g. IEEE/EIA 12207.0) on how "in principle" planning should be performed. However, not so much is known how to actually perform this process in an effective and efficient manner. The standards and guidelines do not give the proper answer on how to operationally assign features to releases to ensure maximal business value.

Now, talking about evolving systems, software development typically proceeds as a series of changes to a base set of software. Such evolution is achieved by modifying parts of the existing components of the software system to implement the new features and requested changes. This implies that questions would be asked pertaining to what components or modules that a developer need to check through to implement a feature. In this case, RP would be more challenging because the design decisions already implemented in existing architecture of the system restricts its ability to accommodate future sequence of releases easily. RP in this context must take into consideration the operating environment and quality of existing components of the software system. This aspect of planning is extremely important since we cannot add new functionalities or change existing functionalities without proper knowledge of the impact of these enhancements and changes.

1.2 Motivation for the Proposed Research

This research is motivated by the need to assist software engineers in the process of assessing the effects of proposed features on components of existing systems in order to determine the set of features to schedule for implementation. Industrial study by Carlshamre *et al.* [9] concluded that the dependencies between requirements and code base of already implemented requirements needs to be investigated. Although the authors have not discussed how to formally achieve this, their observation is in line with reality, because knowledge about the existing software product and the environment within which the product operates

add another dimension to the complexities involved in making release planning decisions for evolving systems. To the best of our knowledge, no release planning technique incorporating the effects of feature implementation on existing system has been published yet. We intend to extend and adapt the existing RP techniques to fit the peculiarities of planning the releases of evolving software systems. The need for RP that we have established above further motivates this research, with the aim of exploring all impacting factors for a more informed selection and scheduling of the most promising set of features that satisfy a broad range of stakeholders.

1.3 Scope of Research/Research focus

RP for evolving systems is a broad topic. Several issues are involved in the characterization and assessment of relevant quality attributes of the existing system, impact analysis of the components impacted by each feature to be implemented, resource estimation tasks, and also in the formal modeling of the RP problem itself. Obviously, we would not be able to address every single aspect of these issues in details.

Even for the assessment of the system components where features would be integrated, there are several possibilities ranging from quantitative metric-based approach and qualitative expert judgment-based approach. Based on experiences from our preliminary work so far, meaningful data collected over previous releases of a system are hard to come by. Our focus has so far shifted to exploring the most efficient and effective ways to employ expert judgment in this regard. We will concentrate on the specific issues that make RP for evolving software systems more complex and challenging than traditional RP for new projects. Ultimately, we are interested in developing a methodology that could be tailored to both new project contexts and evolving software projects alike.

1.4 Research Contributions

The expected contribution of the proposed research to the problem of RP is a methodology that helps in characterizing system components, assessment of the difficulty involved in modifying the components to accommodate new features, and integration of the impact into RP modeling and decision process. Specifically, the proposed research aims on helping with:

- *Evaluating existing software systems:* New and changed features for subsequent releases of evolving systems must be integrated into the existing architecture of the system. Developing a scheme consisting of attributes for characterizing system components and implementing a systematic assessment technique on these attributes enable software engineers to understand the current state of the system. This would help in determining the type of features the system would be flexible to accommodate, just as it would be helpful in reasoning about existing architectures for reengineering, refactoring, and other software evolution decisions.
- *Making informed release planning decisions:* A disciplined approach that formally examines impact of existing system, stakeholders' preferences, interdependencies among features and resources constraints, guarantees that such detailed analysis of all relevant contributing factors would result in RP decisions that are based on informed knowledge and experienced sources which help in rationalization and justification of such decisions.
- *Incorporating divergent concerns of different stakeholders:* By involving stakeholders and allowing them to specify their priorities for different features and aggregating these priorities in a systematic manner, it is possible to strike a balanced and acceptable compromise in features selection and scheduling. This would also results in developing products that satisfy a broad range of stakeholders, and avoid mismatched customer satisfaction. The approach for collecting priorities values from stakeholders would be suitable for both local and global software development.
- *Managing complexities of release planning:* Through formal modeling of RP problem and augmentation of the model results with human knowledge and experiences, we can manage the cognitive limitations of formal models and also manage the computational limitations of human coping with several hundreds of features that are impacted by different constraints. This would also help to alleviate the uncertainties in the input given to formal models, as human decision makers have a chance to further explore the alternatives in the solution space.
- *Making systematic resource estimation:* Through the information made available to software engineers on the difficulty of modifying system components to integrate

new or changed features, they would be able to make more systematic estimate of resources. This could help alleviate problem of budget and schedule overruns.

- *Modeling vagueness in the assessment of interdependencies:* Through fuzzy modeling of the existence of interdependencies between features, it would be possible to bring soft reasoning to bear on rather uncertain assessment situations.
- *Guiding the direction of software evolution:* Because of the visibility into the relative manner in which each modifiability factor affects the components of a system, a property-based classification would be useful in ranking components per modifiability factor. This information could guide software evolution direction. For example, if the difficulty of modifying a component stems more from "complexity" than "size", it gives a hint as to what aspect requires attention during evolution.

1.5 Organization of the Proposal

This research proposal is further organized as follows:

- Chapter 2 (Section 2) presents a survey of topics that are considered to be essentially related to the proposed research, and also an overview of important concepts required towards realizing the goals of the research.
- Chapter 3 presents the proposed research.
- Chapter 4 presents some of the preliminary results based on the proposed methodology, and presents our next direction.

2. Related Work and Enabling Concepts

In this section, we will discuss a background survey of existing RP approaches. We also give an overview of supporting concepts that are considered fundamentally related to the realization of the proposed research. Before the survey is presented, we deem it important to present some basic terminologies relating to RP.

2.1 Basic Release Planning and Evolution Terminologies

Below, we define basic terminologies that would facilitate the understanding of this research proposal:

- **Requirements:** requirements pertains to all the important behavioral characteristics expected of a system that are captured in a requirements document [58].
- **Features:** Several definitions exist in the literature on what constitutes a feature. Some of the candidate definitions and their interpretations have been discussed in [58]. According to [58], an abstract look at the concept reveals that feature represents a cohesive set of system functionality. In the context of this research, we follow the definition given in [59] which identifies a feature to be "a logical unit of behavior that is specified by a set of functional and quality requirements". In other words, features are abstractions or groupings from requirements that both customers and developers understand [49]. We will focus on features as main characteristics of a release plan in this research.
- **Prioritization:** prioritization methods guide decisions about analyzing features to discover their order of importance for implementation. Selection of the most appropriate set of features for product releases is highly dependent on how well we succeed in prioritizing the candidate features. Software release planning extends requirements prioritization.
- **Stakeholders:** stakeholders are defined to be anyone that influences or is influenced by the project plan [16], and this includes customers, various types of users, developers, testers, project managers (decision-makers), and so on.
- **Software configuration:** the current state of a software system and the interrelationship between the constituent components – include source code, data files and documentation [22].

- **Version:** a version represents a small change to a software configuration [22]. These minor changes are typically implemented while a system is in operation.
- **Release:** a release is a collection of new and/or changed features that form new versions of an evolving product [49]; it is a software milestone that is set for distribution. A new release of a software contains major enhancements that are usually planned and incorporated together with other minor changes [22].
- **Software evolution:** the tendency of a software to change overtime [22].
- **Modifiability**: modifiability of a component or software system refers to the ease with which it can be modified to changes in the environment, requirements or functional specification [5].
- **Decision making**: refers to the entire process of making a choice among a number of alternatives.
- **Decision support**: a broad and generic term that covers all aspects related to supporting people in making decisions [6]. It constitutes part of the decision making process.
- **Framework**: a set of ideas, conditions, or assumptions that determine how something will be approached, perceived, or understood [22].
- **Release planning decision framework**: the process, context, and environment in which release planning decisions are carried out.

2.2 Existing Release Planning Techniques

RP can be approached from different perspectives. We have identified two fundamental approaches which we will call: (i) the art, and (ii) the science-based planning of releases. The art of RP involves mainly relying on human intuition, communication and capabilities to negotiate between conflicting objectives and constraints. The science of RP involves emphasizing formalization of the problem and applying computational algorithms to generate best solutions. Both art and science are important to achieve meaningful RP results.

A number of techniques for RP have been proposed in the literature - some specifically targeted at the RP problem while others are simply requirements prioritization and selection techniques. We have discussed some of the notable RP approaches that we considered to be

representative of the current state of research and practice in [54]. These approaches include: Estimation-Based Management Framework for Enhancive Maintenance (EMFEM)[1] [41], Incremental Funding Method (IFM) [13], Cost-Value Approach for Prioritizing Requirements (COVAP)[1] [27], Optimizing Value and Cost in Requirements Analysis (OVAC)[1] [26], Provotype by Carlshamre [8], The Next Release Problem (NRP)[1] [4], Planning Software Evolution with Risk Management (PSERM)[1] [21], Hybrid Intelligence in Software Release Planning (EVOLVE*) [50], and the recently proposed Flexible Release Planning using Integer Linear Programming (FRP)[1] [2]. We will succinctly discuss some of these approaches in the sequel.

ENFEM by Penny [41] is a high-level framework for planning enhancive software maintenance. Penny's approach focuses on release monitoring by ensuring balance between required effort and available effort, as project implementation progresses.

Denne and Cleland-Huang proposed the IFM in [13] as a data driven and financially-informed approach for software release planning. IFM focuses release planning decisions on revenue projections. That is, it concentrates on the rate at which value is returned to the customer. IFM decomposes the system into units of functionality called minimum marketable features (MMF), which is defined as small self-contained features that can be delivered quickly and that can provide market value to the customer. The main thrust of IFM is the sequencing of features by their values, which could be measured in terms of revenue generation, cost savings, and so on. Once the MMFs have been identified, the costs and projected revenues are analyzed over the number of periods established by the business. Developers estimate cost and effort for developing each MMF while business stakeholders estimate each MMF's revenue. The authors equally identified the presence of dependencies between elements to be developed in what they term "precursor" relationship – this is what we generally refer to as precedence relationships that require some elements to be developed before another.

[1] The abbreviation here is merely used for convenience; the technique was not originally named as such.

Karlsson and Ryan's COVAP [27] is essentially a requirements prioritization approach that is discussed in the context of RP. This approach ranks requirements in two dimensions: according to their relative value to the customer and their estimated cost of implementation Relative value and relative cost of requirements to the customers are determined by using the pair-wise comparison of the Analytic Hierarchy Process (AHP) [52], this implies that the method only dealt with comparisons between requirements and not the actual value or cost. These relative measures form the basis for prioritization of the requirements for an onward selection from the pool based on ranking.

Jung's OVAC [26] is an extension of the cost-value approach discussed by Karlsson and Ryan. Jung argued that ordinary inspection to perform pair-wise comparison of requirements using AHP can be very complex as number of requirements grow larger, thus the need to deal with such complexity using optimization techniques. The resulting cost-value model based on AHP was also formulated as a variant of the 0/1 knapsack problem with multiple objective. Just like COVAP, the goal is to select features that give maximum value for minimum cost within allowable cost limits of a software system.

Par Carlshmare [8] developed an RP tool that was aimed at understanding the nature of RP from industrial planner's perspectives. The tool known as "Provotype" was based on the formulation of RP problem as a binary knapsack problem, since requirements cannot be partially selected. The objective function and constraints were based on value of requirements and estimated resource demands respectively, while making sure interdependencies are accounted for. A preliminary empirical study on Provotype was conducted with release planners in the industry as a means of learning about RP. According to Par Carlshamre, the preliminary study revealed that "…release planners would not be interested in one *optimal* solution…by some magical algorithms", but a number of good suggested solutions would suffice. The study also discovered that cost and value are not sufficient in determining the goodness of a solution.

The NRP [4] formulated the RP problem as a variant of the 0/1 knapsack problem. As a variant of the 0/1 knapsack problem, the release planning problem was classified as an NP-

hard problem. By its NP-hardness, the release planning problem would not be amenable to solutions by polynomial time algorithms. At the core of their method is the assignment of weights to customers based on their importance to the software company; the objective is to find a subset of customers whose features are to be satisfied within available cost. Bagnall *et al.* [4] investigated the applicability of a variety of solution algorithms to the resulting optimization model. These include integer programming, greedy algorithms, and local search-based algorithms (specifically, hill climbing and simulated annealing). While experimenting with the algorithms, Bagnall and his colleagues found that integer programming proved sufficient for finding optimal solutions to smaller size problems but could fail to yield such optimal solutions in reasonable time for larger size problems, while simulated annealing found good quality solutions (possibly sub-optimal) on larger size problems. The authors also considered precedence dependencies among features which require that certain features must be implemented before another.

EVOLVE* proposed by Ruhe and Ngo-The [50] facilitates the involvement of stakeholders in RP, so as to achieve releases that result in the highest degree of satisfaction among different stakeholders. It also accounts for the dependencies between features as well as other constraints that restrict the amount of features that can be selected for implementation. The approach here also formulates RP as an optimization problem; the solution algorithm for the formalized problem is based on Integer Linear Programming combined with some heuristics. In agreement with Carlshamre's submission that planning for only one release is not enough and could lead to dissatisfaction [8], EVOLVE* focuses on offering higher flexibility in the number of releases to be planned ahead. Despite EVOLVE* formulation of RP as an optimization problem that can be solved using computational intelligence, the need for human decision maker to further analyze resulting alternative solution plans forms the bedrock of the technique. We have further built on this foundation to establish the need for this strong marriage between computational intelligence and human intelligence in the context of releases planning in a latter work by Ruhe and Saliu [51].

FRP [2] formulates RP as an optimization problem based on integer linear programming. The focus is to select features with maximal project revenues against available resources.

Flexibility in the RP process was introduced to the RP problem by permitting deadlines extension, hiring external resources when needed, and allowing team transfers during implementation of selected features.

In the simple case where we do not need to consider many impacting factors than cost and value, as well as not planning more than one release ahead, pure prioritization of features would suffice. We need only select the top-most subset of features for implementation. Several prioritization techniques exist; with a comparative analysis of the most important and representative techniques by Karlsson *et al.* [28] finding AHP to be the most promising prioritization technique.

2.3 System Components Characterization

In order to account for the contribution of an evolving system environment into RP decisions, it is necessary to examine the current state of the components that constitute the software system. Thus, system components characterization is one of RP supporting concepts in the context of software evolution. In this section, we take a very brief overview of relevant research that deals with characterization of evolving software systems.

Several researchers have explored product release histories in order to derive quality measures that could be relevant in characterizing evolving software systems. Notable in this regard include the modeling of code decay by Eick *et al.* [15], the work of Gall *et al.* [18],[19] on identifying change patterns in evolving products, Mockus and Weiss [35] risk prediction, and Porter and Selby [43] classification scheme. The findings from these works show the relationships they share with the topic of this research. For example, Eick *et al.* [15] and Lehman [30] observed that, as the code base of a system evolves through releases it begins to decay, which implies that current and/or future changes to the code are difficult to make. The difficulty in changing decayed code has also been noted as affecting three aspects of product evolution [20] [60]: increased cost of implementing a change, increased time to complete a change, and reduced quality of the changed product. The results of the work by Ohlsson *et al.* [40] concluded that the ability to identify parts of a system that need improvement makes planning and managing for the next release easier and more predictable.

Based on the results of some of these studies, it is imperative that RP for evolving systems would need to build on ideas from system component characterization, such as to enable analysis of current state of the existing system.

2.4 Software Change Impact Analysis

The purpose of the change analysis is to identify the entities or elements of an existing software system that will be affected by a change [56], before the actual change is made [44]. In the real world of software evolution, changes affect both the components to be changed and their interactions with other components. Change impact analysis provides techniques to solve this problem by identifying the likely ripple effect of software changes [63]. Change impact analysis constitutes part of change planning, and the output from such impact analysis has been identified to be a useful input to many project-planning activities [33]. For impact analysis to be done there must be a starting point, from where all other dependencies can be traced to discover what other components to change together with the starting element.

Early works on impact analysis began in the 1980's and concentrated on ripple effects of changes [56]. The literature on software change impact analysis is rich with several techniques already proposed. For a detailed overview of the research results in the area of change impact analysis, Bohner and Arnold [7] provides a good collection of papers that offer a great starting point. Most of these impact analysis methods concentrate on analyzing the source code of a system.

An emerging research direction on automated change impact analysis pertains to mining the repositories of software projects to determine dependencies among source codes when carrying out changes. One of the first works in this direction was that of Gall *et al.* [18] that studied change patterns among different modules of a system in order to determine logical dependencies using historical co-change; the authors reported that modules that changed together in the past have tendency to change together in the future. Hassan *et al.* [23],[24] studied change propagation and complexity of software development process using data from software repositories. Ying *et al.* [61] and Zimmermann *et al.* [64] applied data mining techniques to explore version histories to discover change patterns, so as to be able to predict

and suggest likely further changes to developers. According to Gall *et al.* [18] mining the repositories of software projects would help in discovering couplings that static analysis of source codes could not reveal.

In order for change impact analysis results to be useful in the context of RP for evolving software systems, it is highly necessary to have traceability from the features to the initial components of the existing system that would directly be impacted. From this starting point, change impact analysis could then analyze ripple effects to identify other components that are indirectly impacted but would need to be changed. Unfortunately, the change impact analysis literature discussed above presupposes that a particular set of primary starting-point component is already known in the existing system, and we are only left with the task of identifying other components that would change with the starting-point. Pfleeger and Bohner [42] identified the importance of traceability from requirements to code for application in impact analysis. Lindvall and Sandahl [33] carried out empirical study to determine the effectiveness of software developers in identifying both starting-point and other co-changed components based on pure experience. Lock and Kotonya [34] classified existing impact analysis techniques for identifying impact propagation paths between requirements level entities as: pre-recorded traceability analysis, dependency analysis, plain experience analysis, and extrapolation analysis.

2.5 Software Engineering Decision Support (SEDS)

RP decisions are not always straightforward as they exhibit difficulties that characterize decision-making in natural settings. The inability of human to cope well with complex decisions involving competing and conflicting goals in software engineering suggests the need for supplementary decision support [47]. Considering problems involving several hundreds of features and large number of widely distributed stakeholders, it becomes very hard to find appropriate solutions without intelligent decision support systems.

Experience with RP reveals that difficulty of the problem results from two aspects: (1) *cognitive complexity* owing to the ill-defined nature of the problem which limits the ability to completely formalize of the problem, and (2) *computational complexity* relating to the

concern for near-optimal selection and scheduling of features, which limits the capability of relying on pure experience and intuition. Thus, any attempt to model the RP problem would essentially be an approximation of the reality, because tacit judgment and knowledge will always influence the actual decision. The nature of the problem, therefore, has several implications for the type of support needed [8]. As it would not be realistic to fully formalize the problem and knowing it is hard to give a final word on what is right and what is wrong, what we proposed in [51] is to adopt a synergy between computational intelligence to address formal modeling of the problem, and human intelligence and experience to address the non-formalizable aspects. Such synergism follows the paradigm of software engineering decision support (SEDS) as instituted by Ruhe in [48]. Since it is hard to have a final world on what release plan is right and what is wrong in every situation, what we need is a decision support methodology that helps in identifying what is right from what is wrong for a specific organization and project.

The idea of offering decision support always arises when timely decisions must be made in unstructured or semi-structured problem domains, where multiple stakeholders are involved, and when the information available is uncertain. The inherent uncertainties present in most of the resource estimates and feature definitions make it even harder to simply apply a fixed and formal closed-world type approach to solve the RP problem [50].

2.6 Summary

To support RP decisions for evolving systems, we need to consider impact analysis to assist in estimation of the amount of work required to realize a feature and also assist decisions to delay or schedule a feature depending on the impact level on existing components. Components characterizations are necessary in order to quantitatively rank the components identified during impact analysis. This chapter provides a summary of the techniques that could be useful in characterizing and assessing the quality of software components, and also an overview of impact analysis techniques. The chapter also introduces the paradigm of software engineering decision support as one of the supporting concepts for the topic of this research.

3. Research Proposal

3.1 Problem Definition

Our analysis of RP literature in Chapter 2 reveals there is lack of support for evolving software systems when selecting and assigning features to future releases. From our discussion in Chapter 1, it is clear that RP for evolving systems need to factor the contributions of system constraints imposed by the current architecture of the existing system into the decision process. If we have to address this issue at all and offer appropriate decision support, we cannot find an isolated solution to fill this gap, as only a holistic view of problem would be desirable. In this context, we first summarize the challenges of the RP problem by identifying which aspects contribute to what dimension of the problem, as given in Figure 1:

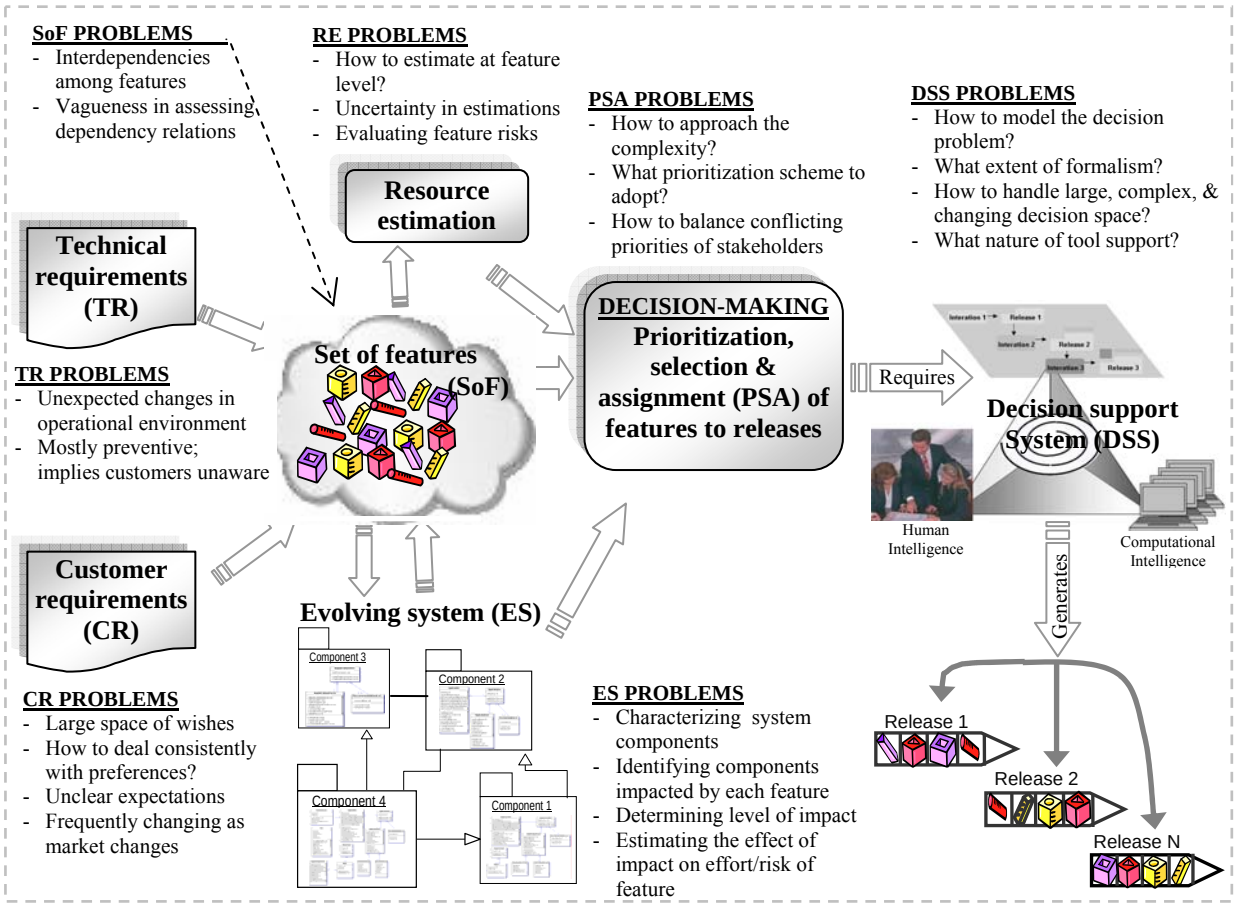

Figure 1: Instantiation of Release Planning as a Decision Problem

While we do not claim that Figure 1 is all-inclusive, it gives an overview of the different problems that characterize RP within the realm of SEDS. The problem dimensions relate to: customer and technical requirements, extracting features from requirements, estimation of resource required to realize each feature, evaluating how the features impact existing systems, and the actual prioritization and assignment of features to releases. This last dimension represents the core of the decision-making problem that calls for decision support system as discussed in Chapter 1. While prioritization, selection and assignment of features remain the actual goal, we cannot under-estimate the contributions of the other problem dimensions if release plans must be realistic.

The visualization of the problem dimensions in Figure 1, as well as the identification of difficulties plaguing the problem makes it easy to formally define the RP problem for evolving software system as follows:

> Given an existing software *system* S composed of components[2] c(1), c(2), …, c(m), a set of desired *features* f(1), f(2), … , f(n) with values v(1), v(2), … , v(n) respectively and requested by *stakeholders* s(1), s(2), …,s(q), a set of *resources* r(1), r(2), …, r(R) required for implementation of the features, find a sequence of assignment of the features to *releases* k(1), k(2), …, k(K) such that the implementation of f(i) in C(m) is *feasible* and the value of the releases are maximized while making sure most of the important stakeholders are satisfied.

The term "feasible" in the problem definition pertains to realization of the objectives of maximizing value while making sure that the imposed constraints are not violated, and selected features are deliverable. These constraints include efforts, schedule, budget, risk, dependencies between features, and system constraints resulting from dependence of features on existing system, that could impact feature scheduling and selection decisions. Current RP approaches discussed in Chapter 2 have addressed, at least partially, some aspects of the problems and constraints. Unfortunately, none has so far attempted to consider the critical

[2] The granularity of components is not pre-defined here; it could mean subsystems, modules, classes, packages, etc. It depends on the system and it is ultimately at the discretion of software engineers familiar with the system.

constraints imposed by the evolving system environment where the requested features would be implemented. Thus, these solution approaches were based on the assumptions that every time we collect features we are in the process of planning releases of new software products that has no prior evolution history. The reason for not considering this vital dimension may be connected to the non-availability of an RP process model. To be able to even attempt a solution to this problem, we must be able to provide answers to these two broad research questions:

- How do we account for the influence of existing systems characteristics on RP decisions?
- How can we model the RP problem and proffer solution by combining knowledge of the impact of each feature on existing system, feature values, stakeholder preferences, resources and risk constraints, to generate realistic plans?

3.2 Research Objectives

The goal of this research is to address the questions above by providing a flexible decision support methodology for RP that can easily be tailored to different project contexts – new and evolving software systems alike. In order to achieve this goal and address the research questions, we need to resolve the following key issues that constitute research challenges in this work:

1. First, we need to develop an RP process decision framework that encompasses all the vital activities and also act as a guide for performing RP. It should be well-structured, and also flexible for easy adaptation to different organizations and project contexts. That is, the process model should be valid for planning releases of both evolving software systems and new software systems alike. An instantiation of the generic framework could then form the basis of any specific RP methodology.
2. Second, we need to develop a scheme that will be useful for characterizing the components that make up a software system. Such characterization scheme may be based on specific quality attributes. In this research, the quality attribute of interest is the modifiability of components of a system. The scheme would form the basis for developing an assessment metric (the terms *metrics* and *measures* are typically used interchangeably in the software engineering literature[46]; and same applies to our

usage here) to evaluate modifiability and quantitatively rank the components according to the difficulty of modification.

3. Third, we need to develop a technique to determine how the implementation of each feature would affect each component of the system. This would involve impact analysis to identify components that would be impacted by the implementation of proposed features, and also analysis of the relative amount of modification required on such components identified. The results of the impact analysis together with the assessment method for difficulty of modification would be useful for assessment of impact in quantitative terms.
4. Fourth, we need to be able to evaluate how characteristics of impacted components influence the resources required to implement a feature. Also, how such knowledge about the components determines the way the set of features can be accommodated in existing system.
5. Fifth, we need to develop a decision support environment that models the RP problem by considering all the relevant factors, improve on existing feature prioritization and scheduling mechanism which form the core of the problem modeling. Considering the fact that the environments in which software projects operate are plagued with uncertainties and vagueness, it would be desirable to model such vagueness in the context of RP parameters. The desired decision support environment must be able to assist project managers in making good RP decisions.

The goal of this chapter is to define a research proposal that aims to address these issues itemized. The following five sections discuss our proposed approach, with each subsection corresponding to the five previously listed key issues.

3.3 Release Planning Process Decision Framework: The S-EVOLVE* Approach

The RP process decision framework that we are developing in this research was inspired by an earlier hybrid intelligence-based RP approach (i.e. EVOLVE*) proposed by Ruhe and Ngo-The [50]. Our proposed decision framework depicted in Figure 2 first establishes RP in the context of software engineering decision support by providing a flexible process model consisting process elements that could be customized to different project environments. The

framework is called S-EVOLVE* (System-EVOLVE*) to reflect the consideration the system constraints imposed by evolving systems on RP.

We have identified nine(9) key process elements that should constitute an RP process decision framework based on our analysis of the RP literature in [54], industrial experience reports on state-of-the-practice by Carlshamre [8], Lehtola *et al.* [31], Regnell *et al.* [45], and analysis of the problem dimensions in Figure 1. The process elements of S-EVOLVE* shown in Figure 2 represent RP activities indicated by the rounded-rectangles, and intermediate output from each activity indicated by the ovals. Therein, three roles that contribute to the process and products of RP are identified: project manager (PM), other-stakeholders apart from the PM, and the support environment. Activities occur directly under the roles that are involved. For example, the roles indicated as project manager and other-stakeholders are involved in feature elicitation, while support environment maintains group of features elicited.

In the sequel, we succinctly describe the nine (9) process elements:

- ***P1: feature elicitation*** to collect the set of features requested. The most important issue here is that features extracted from customer and technical requirements should be described so they are understandable, consistent, complete, and correct. Collecting meaningful features is a complex problem in itself, and it is not studied in this research. Christel and Chang [10] gave an overview of the issues involved.
- ***P2: problem specification*** to identify and model the objectives and constraints based on stakeholders propositions. Such propositions may relate to expectations in terms of value or satisfaction, depending on the project and stakeholders involved. Constraints impacting the features must be identified while executing this process element.
- ***P3: stakeholders voting*** to allow the stakeholders identify and assign priorities to features based on their preferences. Giving stakeholders the opportunity to vote on features according to their preferences is important because they decide the success criteria for products.
- ***P4: components modifiability assessment*** to evaluate the Difficulty of Modification (DoM) of each component of the target evolving system. The assessment required

here is a generic measure that could be run on components of a system at any time, so we could compare the DoM of one component to another.

- ***P5: features-driven impact analysis*** to identify the components that would need to be modified when implementing a given feature. We do not, however, expect a very thorough impact analysis at this stage, as they would still be done during design and implementation, a preliminary impact analysis is essential for product planning. This process element also requires an assessment of the eXtent of Modification (XoM) that the implementation of each feature would require on the target components that would host the feature. The XoM should be a local measure that is specific to each feature in relation to the target components.
- ***P6: impact quantification*** to quantitatively assign to every feature a measure that reflects an aggregation of both the DoM and XoM over all impacted components.
- ***P7: resource estimation*** to predict the likely amount of effort, schedule, and cost to implement the features. This process element is crucial to the success of RP, since it matches the resource requirements of the problem with the available resources.
- ***P8: release plan generation*** to aggregate the results of all the preceding process elements, and assign the features to releases making sure the constraints are satisfied.
- ***P9: evaluation of release plan alternatives*** to analyze the release plans generated whether the defined objectives are realized, and the plans meet the expectations of the decision-maker. An unsatisfactory set of plans could lead to re-planning through reformulation of the problem based on current understanding.

We have merely presented an overview of the process elements here, a more detailed discussion on these process elements is contained in [53]. For the purpose of this proposal, we will expatiate further on process elements (P4, P5, and P6) that are captured in the shaded region of Figure 2. These three process elements constitute the evolving systems extension, which form one of the core contributions of this work.

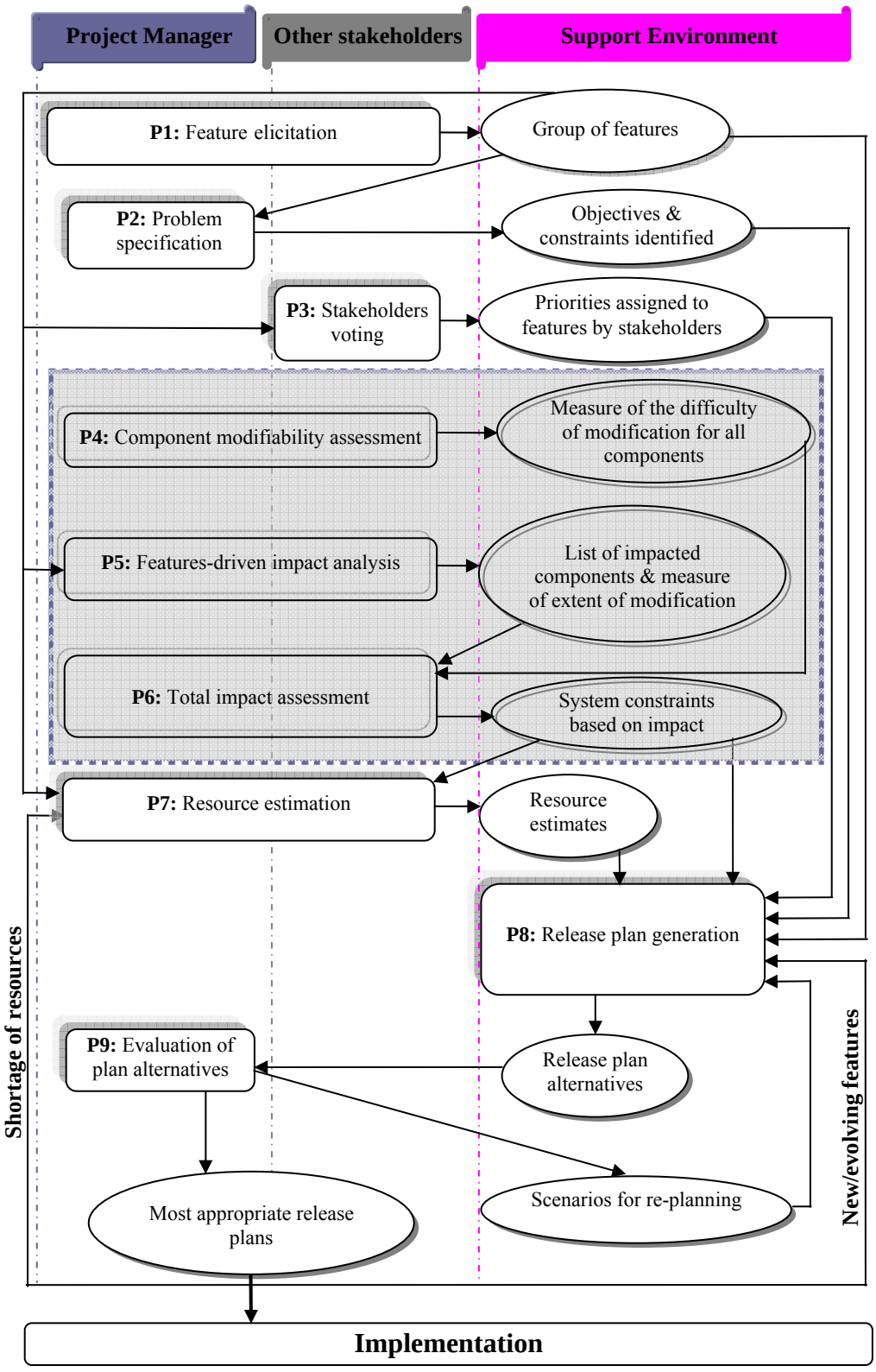

Figure 2: Release Planning Process Decision Framework (S-EVOLVE*)

One of the most important aspect of S-EVOLVE* is the assessment and quantification of the impact that each feature exerts on existing system components. The process of realizing the activities of the process elements (P4-P6) constituting this aspect is detailed in Figure 6, which is further expatiated in subsections 3.5 and 3.6. Meanwhile, there is no dependency in the order of execution of P4 and P5; both process elements can be performed in parallel. In fact, P4 and P5 are not tied to release planning, and their results can serve as input to other software evolution activities. For example, DoM assessment could aid decisions on which components of the system should be refactored to improve structure or reengineered to improve quality.

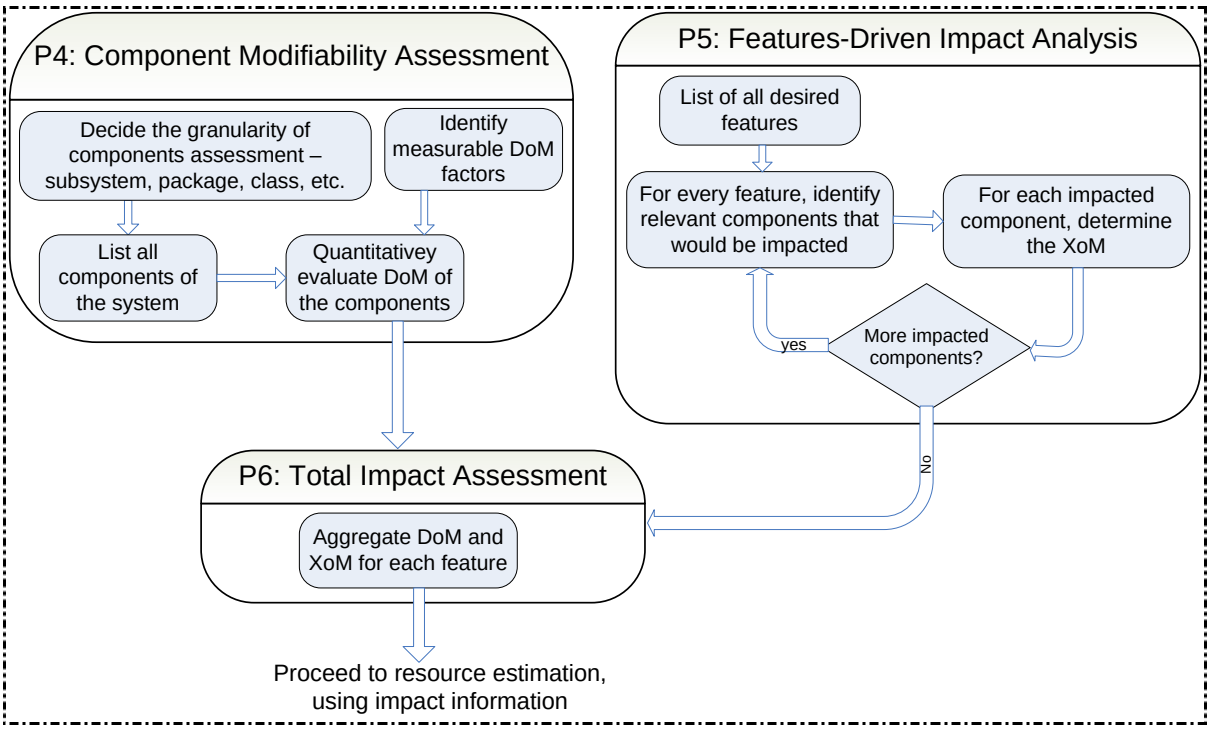

Figure 3: Quantitative Process for Evaluating Features Impact on Existing System

3.4 Components Characterization Scheme and DoM Assessment

3.4.1 Overview of the Characterization Scheme

Performing an assessment of the components of a system based on the attribute of interest (i.e. DoM) requires that we characterize this attribute based on influencing factors. In addition, any assessment procedure to determine DoM must be based on an abstract model

for it to be universal. Hence, a major task is to identify influencing factors that would apply to all components and for which possible quantitative or qualitative metrics can be defined, at least within the same environment or project. The high level abstraction desired in the assessment procedure should guarantee the possibility to localize measures in order for such measures to be realistic.

Several researchers have explored product release histories in order to derive quality measures of a system. These studies include Eick *et al.* [15] modeling of code decay, Gall *et al.* [18],[19] work on identifying change patterns in evolving products, Mockus and Weiss [35] risk prediction, and Porter and Selby [43] classification scheme. In the sequel, we present an initial high-level characterization scheme that could guide data collection and metrics derivation, or expert-based assessment of difficulty of modification. We have initially identified several attributes of the characterization scheme in [54], and further developed the attributes into the current scheme in [53]. This scheme is expected to evolve as we gain more understanding through empirical results. One of the main goals involves developing a method that guides the identification of important attributes that characterize system components in each particular organization.

Any set of factors identified as contributing to DoM must be an instance of the hierarchical characterization scheme shown in Figure 4. The hierarchy is necessary because most of these factors can be assessed based on further refinement into lower level criteria that are directly measurable and for which data can be collected. However, we do not in anyway presume that formal metrics collection would be the most plausible approach for assessing these factors. When applicable, the definition of appropriate metrics depends on the level of historical data available in an organization, as well as experiences of software engineers that are familiar with the system under consideration.

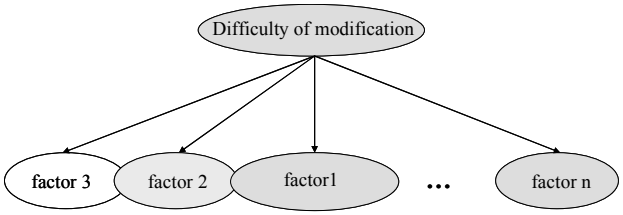

Figure 4: Characterization scheme for component DoM

For the purpose of this work, and based on analysis of existing works in this area that we discussed above we have identified six major factors that contribute to DoM of a component – size, complexity, understandability, health, criticality, and functionalities. It should be noted that the six factors is an instantiation of characterization scheme in Figure 4, the methodology that we present here can accommodate any number of factors. These factors are by no means exhaustive since several local factors could also contribute to the assessment of DoM. The factors are, however, representative for classification of components according to the degree to which each factor affects each component. It is also important to stress that the factors are not assumed to be necessarily orthogonal – for example, complexity may impact understandability of a component. We briefly expatiate on these factors:

- **Size:** most components of a software product exhibit varying sizes depending on the functionalities implemented in them and other product factors. When designing and implementing larger components, it is harder to keep track of all details and interactions on mind and vice-versa for components with smaller sizes. Different types of metrics are available for determining the size – the most popular is the thousand lines of code (KLOC).
- **Complexity:** the complexity of a code base is determined by the structure of the code, which is also largely affected by the interactions among the components. For instance, as the coupling value of a component increases, so does the number of other components that need to be modified whenever a change affects one of them. According to Lehman's laws of software evolution [29] software complexity tends to increase over time while the quality of the product will tend to decrease. A comparison of several complexity measures are discussed by Schneidewind and Hoffman in [55].
- **Understandability:** refers to the ease with which components can be understood by the developers that are modifying it. That is the ability to determine what a component does and how it works. Empirical studies have revealed that this factor is a function of the expertise of whoever is making the changes [15] [22], the growth rate of the component from one release to the other [29], how long the component has been part of the system [15], and quality of documentation

available [22]. All these information put together could guide software engineers in assessing the understandability of a component. Program understanding and correspondingly component understanding is said to consume a significant proportion of maintenance effort and resources [22], which further justifies its inclusion as part of the factors that could contribute to DoM.

- **Health:** the health of a component refers to the extent it can be trusted to exhibit consistency in performance without failures. If components are unhealthy, they will be fault-prone, and changing any small part of the code can be very risky and time consuming. Empirical studies [20] [15] reveal that correlations exist between the health (i.e., ability to use or reuse existing code base [12]) of program code, the quality of the resulting product, and the functionality that can be added to the system. There are empirical results revealing that it becomes more and more to add new features to a code base as it evolves [20] [30], and similar trend is expected of the components constituting the code base. In his dissertation, Homayoun [12] observed that assessing the health of existing products has received little attention in the literature. In fact the assessment of components for healthiness may be related to the number of operational faults that have been reported against the components during field operation.
- **Criticality:** In any system, some parts are mission critical while others are not, and normal operation could still be carried out without the non-critical parts. Some components add only minor functionality, or functionality that is rarely, if ever, used. Clearly, deficiencies in these kinds of components, although unpleasant, are not service impacting. The criticality of a component must be taken into consideration when evaluating DoM because extra effort and resources may be needed to ensure that mission-critical components achieve their goals. If a bug manifests itself as a failure in a more critical system component, consequences on the system will likely be far greater and more profound, and vice-versa for low criticality components.
- **Functionalities:** The number of functionalities implemented in a component could contribute to the difficulty of modification of the component. Some components may implement single functionality while others may implement

several functionalities. In the case of components implementing several functionalities, modifying such to incorporate new features would be arduous. Without a careful analysis, some of the functionalities in use could be affected negatively.

3.4.2 The DoM Assessment Framework

Developing a characterization scheme is silent about how to actually quantitatively or qualitatively measure DoM; it is just a means to an end. Two fundamental approaches can be employed in developing this assessment framework. First, we can adopt a metric-based based approach that seeks to explore the historical data available about a software product to collect relevant metrics. Secondly, we can adopt an approach that is entirely based on expert judgment derived from the opinions of those that hold the domain knowledge and are familiar with the evolving system under consideration. However, our major focus would centre on the expert judgment-based approach, because of the limitations of the metric-based approach that we discuss below.

1. The Metric-based Approach

Our initial work that explores the automated quantitative metric-based approach is discussed in [54]. We evaluated the risk of integrating new features into existing system components using a metric-based approach that exploits historical defect data about a system to characterize the health of system components. In reality, the ability to establish and adopt this quantitative metric-based approach depends on the maturity of the organization in collecting relevant historical data over several releases. In spite of the fact that we considered only one of the factors, there were several challenges regarding data collection. The challenges are expected to be more pronounced when the measurement is to be done using our entire characterization scheme comprising all the six factors. In general, this approach appears to be fraught with the following difficulties:

(i.) it requires data to be kept for most previous releases of the software, a situation that could be expensive and unpractical.

(ii.) software engineering data are known to be generally difficult to collect, as we have experienced during the initial the study mentioned above.

(iii.) even when the data are available, it is always difficult to guarantee their quality.

(iv.) metrics defined on datasets from a specific software product cannot always be easily generalized to other products and even more so if the products are from different domains.

(v.) raw measurements taken on different attributes would all be on different scales, which complicates the comparison of different components within a software system using raw measurement data [39].

(vi.) considering the many metrics available, and that many are widely criticized, it is always not clear how to determine the appropriate ones for a project?

2. The Expert Judgment-based Approach

Based on these difficulties, we have developed an alternative but complementary DoM assessment approach in [53] which aims on capturing the knowledge of experienced software engineers (i.e. experts) that are familiar with the system under consideration, and who would be participating in the software evolution project. This approach emphasizes the necessity to tailor the assessment method to the organization based on availability of individuals that would play the roles of experts. Expert opinion plays a vital role in any assessment problems for which there are problem of data availability. Expert-judgment has been found extremely useful in software engineering, especially in the area of effort prediction [1]. While discussing refactoring of codes, Fowler and Beck [17] also submitted that "no set of metrics rivals informed human intuition".

The approach we proposed in [53] is a DoM assessment framework that employs a multiple-attribute decision making (MADM) [62] technique - the analytic hierarchy process (AHP) [52], which is an acceptable systematic method to elicit expert opinion in a formalized way. The fundamental philosophy of AHP is to use judgments from experts to derive a set of ratio-scaled measures for decision alternatives. It uses mathematical algorithms to transform qualitative subjective judgments elicited from experts into quantitative data. AHP is an established multi-criteria decision method that has been applied in various fields of human endeavor - medicine, politics, human resource management, agriculture, and so on. AHP provides consistency check capability; a very useful feature since human judgment is far from perfect.

Our AHP-based assessment method emphasizes that, expert opinion about how each factor determines DoM of the components must be done from the perspective of participating knowledgeable experts (e.g. developers, designers, etc.) that are familiar with the architecture of the system under consideration. The assessment method for DoM is performed in six steps:

Step 1: Definition of Criteria and Alternatives

The first step is to structure the problem in a hierarchical form as shown in Figure 5. At the top level is the overall goal - "Difficulty of Modification". In the second level are the factors that contribute to the goal; we have identified six for illustration of the method here. The third level represents the components that are to be evaluated in terms of the factors in the second level.

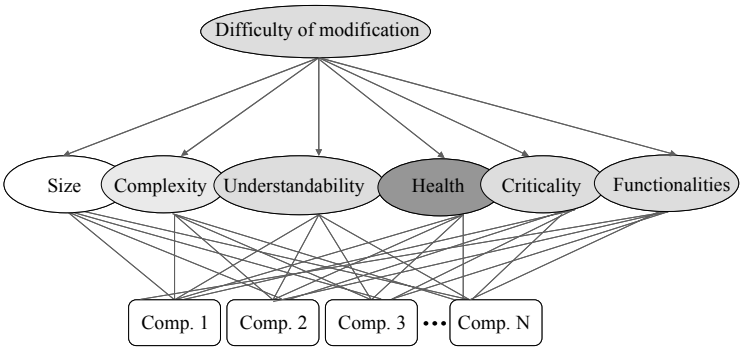

Figure 5: Hierarchical Decomposition of DoM Assessement

Step 2: Computing Priorities of Experts Based on Familiarity with Components

The second step is the assignment of importance values to the experts that would participate in the assessment of all the components for DoM. In our earlier work [53] the importance of participating experts was assessed by the project manager (or the Lead Architect), given that he understands the role that each expert plays in the current project, independence of each individual component. With the experience gained during our preliminary case study, we discovered that familiarity with a component plays a significant role in determining the confidence and ease with which each expert assesses each component. To bring this to bear with reality, the importance of expert should be component-dependent.

The consequence of this premise is that, the more an expert is familiar with the component in question, the more we should trust his judgment in correctly assessing the component. One way to assign priorities to experts relative to the components is to ask each expert to fill a questionnaire stating how many times he had worked on each component of the system since last release of the system. Given *E* experts and *M* components, the priority *W(e,m)* $(1 \le e \le E, 1 \le m \le M)$ assigned to each expert relative to each component depends on the number of times *N(e,m)* that the expert has modified the corresponding component. The priority is computed as:

$$W(e,m) = \frac{N(e,m)}{\sum_{e=1\ldots E} N(e,m)} \quad (1)$$

W(e,m) $(0 \le W(e,m) \le 1)$ is a priority matrix, with each column consisting normalized priority vector of the relative importance of experts on each component. Thus, for any unique component *m*, the priorities of all the experts satisfy $\sum_{e=1\ldots E}$ *W(e,m)* = 1. During aggregation of the final result, the importance of each expert with regard to each component would be a multiplicative weighting factor of the vector of relative DoM of components, as assessed by the corresponding expert.

Step 3: Prioritization of DoM Factors

The third step is for each expert to perform pair-wise comparison of the factors impacting DoM. This pair-wise comparison is carried out using AHP. According to Saaty [52], the fundamental weighting scale used for the purpose of this pair-wise comparison is shown in Table 1. For each pair of factors (starting with Size and Complexity, for example) their relative importance value describes the "intensity of importance". The question to be asked at this stage when comparing two factors to get the intensity of importance value is that, "of the two factors being compared, which of them is more important in determining the DoM of components?"

After completing the pair-wise comparison, the aggregated eigenvalues computation establishes a priority vector, *V(z)* $(1 \le z \le Z)$, $(0 \le V(z) \le 1)$, which represents the relative importance of all the Z factors from the perspective of this expert. It is guaranteed that the entries in the vector V(z) satisfy $\sum_{z=1\ldots Z} V(z) = 1$.

Table 1: AHP weighting scale for pair-wise comparison

Intensity of importance	Definition	Explanation
1	Equal importance	The two activities or variables (i and j) are of equal importance
3	Moderate importance of one over another	Experience slightly favor one activity over another
5	Strong importance	Experience strongly favors one activity over another
7	Very strong importance	An activity is strongly favored and its dominance demonstrated in practice
9	Extreme importance	The evidence favoring one over another is of highest possible order of affirmation
2, 4, 6, 8	Intermediate values between the two adjacent judgments	When compromise is needed
Reciprocals	If activity *i* has one of the above numbers assigned to it when compared with activity *j*, then *j* has the reciprocal value when compared with *i*	

Step 4: Prioritization of Components for a Fixed Factor

The fourth step is for each expert to carry out a relative pair-wise comparison of the components with respect to each of the DoM factors. The objective is to determine how DoM of each component is relatively determined by the relevant DoM factor. The guiding question to ask when comparing two components based on the factors is of the following forms: "How much more difficult is it to modify component A than component B with respect to Size?" The same question applies to all other factors. That is, complexity, understandability, health, criticality, and functionality in our case.

After the end of this phase, each expert would have developed an M x Z matrix, which shows for every entry (m,z) the relative manner in which each factor in column *z* affects the DoM of the component in the row *m*. Higher value in the row of the matrix translates to higher complexity (or any other factor under consideration) of the component relative to other components, and consequently the more difficult it is to modify the component. The matrix is given as:

$$\phi(m,z); 1 \le m \le M, 1 \le z \le Z \qquad (2)$$

where each column entry (representing a factor) in the matrix satisfies the inequality $0 \le \phi(z) \le 1$.

Step 5: Relative Priorities of Components from the Perspective of Each Expert

The fifth step is to establish the overall relative priorities of the components from the perspective of each expert. We compute the priority vector showing the relative DoM of each component from the perspective of expert *e* using the formula:

$$\theta(e,m) = \sum_{z=1\ldots Z} V(z) \cdot \phi(m,z) \quad (3)$$

If only one expert were responsible for evaluating the DoM of the components, then we could as well assume $\theta(e,m) \equiv \theta(m)$, which gives the desired overall priority vector representing the relative DoM(C(m)) for all components. In fact, it is highly unlikely that only one software engineer would be working on evolving a large software projects. This implies that the vectors by each expert *e*, is combined to form the M by Z matrix $\theta(e,m)$.

Step 6: Aggregation of the Judgments of Expert

The sixth step is to establish the final overall relative priorities of the components based on their DoM from the perspective of all participating experts. That is, the computed component prioritization vectors from each expert are linearly combined according to the importance of the experts relative to each component. This establishes the final aggregated priority vector that represents the DoM(C(m)). The higher the priority value assigned to a component, the more difficult it is to modify the component, and consequently the higher the DoM(C(m)) assigned to the component. We represent this concept by the formula:

$$DoM(C(m)) = \sum_{e=1\ldots E} \theta(e,m) \cdot W(e,m) \quad (4)$$

where $\theta(e,m)$ is the M by N matrix computed in step 6, and $W(e,m)$ is another M by N matrix computed in step 2. Each entry in the DoM vector satisfies the inequality $0 \le DoM(C(m)) \le 1$. Since the importance priorities assigned to the experts are dependent on the component they are assessing, it is not guaranteed that, $\sum_{m=1\ldots M} DoM(C(m)) = 1$, but this could be easily achieved.

3.5 Feature-Driven Impact Analysis and Total Impact Assessment

In most cases, implementation of a feature would require modification of one or more components. Even when we identify only one component as the impacted component, such components might interact with other components (both structural and logical interactions) that would need to be modified as well. Features-driven impact analysis (FDIA) refers to the process of identifying the component(s) or other entities of the existing system that would need to be modified directly or indirectly in order to implement each new or changed feature. The input to the FDIA process is a set of features to be implemented together with the existing system to be modified [33].

The process of performing FDIA correctly is a hard task that has been found to largely depend on experience with the system and the change tasks [33]. Empirical studies by Lindvall and Sandahl [33] have indicated that software engineers are fairly able to identify components that would be impacted by the features to be implemented. Although their empirical results reveal that software engineers are only able to identify or predict about 50% of such impacted components, but their observations show that all impacted components that software engineers identify are correct. Based on this study on impact analysis, Jonsson and Lindvall also concluded in [25] that "interviewing knowledgeable developers is probably the most common way to acquire information about likely effects of new or changed requirements…"

While empirical studies support the use of human judgment in performing FDIA, we have found the emerging research in mining version histories for automated change impact analysis [18],[23],[24],[61],[64] appealing, as they appear to be very promising towards supporting human experts in identifying indirectly impacted components, especially for large systems that may not be easily tractable. The automated analysis could be used to suggest more components for experts to reevaluate whether such components would actually be impacted. This automated aspect of FDIA cannot replace human experts, but could only complement their ability to discover more indirectly impacted components. This is because, logical co-changes may be amenable to automated detection, but automated traceability from new features to directly impacted components of existing system would be difficult since

these features would not have been envisaged when naming program segments and architectures. Thus, traceability of features to primary components impacted would largely require manual intervention. Of these automated co-change detection, we find the idea proposed by Zimmermann *et al.* [64] to develop rules on co-changing components is the most promising approach that could be modified for early querying of components. In the current strategy of Zimmermann *et al.* [64], rules are fired to suggest other changes to developers when they are ready to commit the changes they have made to the version control system. However, with the difficulties we have outlined for automated metrics collection, the use of automatic co-change analysis depends on the availability of version history data, and would only be useful in environments where such data are kept.

In addition to identification of components to be modified, this process elements also requires the assessment of the eXtent of Modification (XoM) required in order to implement and integrate each feature into components C(m). Since this assessment is feature- and component-specific, we denote the measure as XoM(C(m), f(i)). We define XoM(C(m),f(i)) to be the extent to which an existing component is to be changed by a proposed feature, and measure it as a percentage of expected code modification relative to the original size of the component. This is essentially expected to be an approximation, and would suffice for estimation at this early stage in the project. In [53], we have carried out such assessments using opinions of developers and designers who are familiar with the existing architecture of the system and who would be performing the implementation. Measuring XoM could also be done at higher granularity than estimating code percentages – for example, it could be measured as percentage of objects or class modification to be carried out on a package in case we are dealing with an object-oriented system.

3.6 Total Impact Assessment

The two concepts, DoM and XoM, are illustrated in Figure 6. A system S is assumed to consist of six components, S = C(1), …,C(6), although the number of components in a system is expected to grow with releases. For illustration purposes, four components are assumed to be affected by the implementation of feature f(i), e.g., Ψ(i) = C(1), C(2), C(4), C(6). The different grey levels of the components C(m) refer to the different levels in the

difficulty of modification. The hatched areas within affected components describe the extent of modification XoM(C(m),f(i)). The larger the hatched areas, the larger the extent of modification will be.

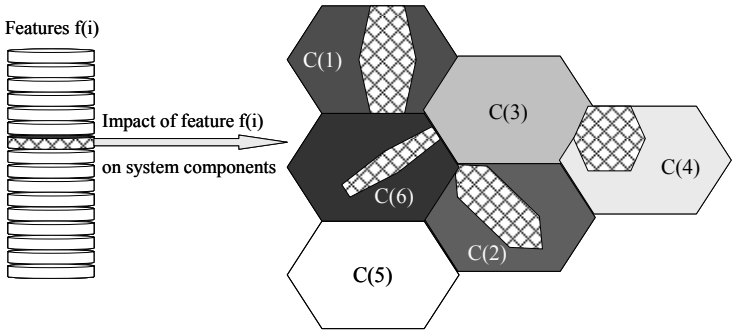

Figure 6: Impact of the implementing and integrating feature f(i) into system components C(m)

Assessing the total impact of implementing any feature in existing components requires that the values of DoM(C(m)) and XoM(C(m), f(i)) be aggregated for all such identified components C(m) ∈ Ψ(i) that are impacted by the feature f(i).

Since the results for DoM's derived using AHP are normalized measures (i.e. $0 \leq$ DoM(C(m)) ≤ 1), while the XoM's measured as percentages and not necessarily normalized, we need to consistently aggregate both measures over all impacted components. We express the aggregation function as a multiplicative function, such that for any feature f(i), we define the impact as follows:

$$\text{Impact (f(i))} = \sum_{m=1\ldots M} \left[\left(1 + XoM\left(C(m), f(i)\right) \Big/ 100 \right) \cdot DoM\left(C(m)\right) \right] \cdot \delta_{im} \qquad (5)$$

Where,

$$\delta_{im} = \begin{cases} 1 & \text{if C(m)} \in \Psi\text{(i)} \\ 0 & \text{otherwise} \end{cases}$$

The expression given in Equation (5) represents our initial attempt at formally assessing the impact of features on existing systems. However, the entire assessment framework for DoM, XoM, and impact quantification is still under investigation.

3.7 Effect of System Impact on Resources

Estimation or prediction of resource required to implement a feature depends on nature of feature and parts of the existing system components that the feature would be integrated. In this context, we initiated work in [54] that focuses on risk of integrating features. Reflection of the effect of impact assessment on extra integration effort was also discussed in the paper. Formally accounting for this and the relevance of impact quantification on technical risk assessment would be further investigated in this work.

3.8 Decision Support for Planning Releases of Evolving Systems

The decision support (DS) envisaged in this research would essentially extend the EVOLVE* model proposed by Ruhe and Ngo-The [50] and implemented in the ReleasePlanner® system (http://www.releaseplanner.com). Our solution approach, S-EVOLVE*, would benefit from these initial results and enhance it to account for the impact of system characteristics on feature selection and scheduling decisions. We would implement a variant of the optimization-based model developed for EVOLVE*. The model would also include our system constraint dimension for evolving systems, as well as capturing the vagueness in interdependencies assessment using concepts from fuzzy graphs. The problem modeling is composed of definition of key decision variables, interdependencies, objective functions and constraints.

3.8.1 Features and related decision variables

We assume a set of features $F = \{f(1), f(2), \ldots, f(n)\}$. The goal of RP is to assign the features to a finite number K of release options, with the option of postponing the assignment of some features. A release plan is characterized by a vector of decision variables $x = (x(1), x(2), \ldots, x(n))$ with $x(i) = k$ if feature $f(i)$ is assigned to release option $k \in \{1,2,\ldots K\}$, and $x(i) = K+1$ if the feature $f(i)$ is unassigned.

3.8.2 Stakeholders

Stakeholders are extremely important for performing realistic and customer-oriented release planning. We assume a set of stakeholders $S = \{S(1),\ldots,S(q)\}$. Each stakeholder $S(p)$ can be assigned a relative importance $\lambda(p) \in \{1..,9\}$. Larger values of $\lambda(p)$ indicate greater

importance. The assignment of relative importance is applicable in the same way for other scales; and it is independent of the number of stakeholders involved.

3.8.3 Prioritization of features by stakeholders

In order to select and schedule features, there must be an agreed upon statement of priorities for features. In our model, prioritization by each stakeholder S(p) is done with respect to two different types of criteria, each defined on a nine-point ordinal scale: Value (denoted value(i,p) $\in$ {1…9}) assigned by stakeholder p to feature f(i), and Satisfaction (denoted sat(i,p) $\in$ {1…9}) assigned by stakeholder p to feature f(i). An increasing order of value/satisfaction corresponds to an increasing value-based/satisfaction-based priority.

The value-based priority measure is used to express the expected value that the implementation of this feature will add to the stakeholder. The satisfaction-based priority measure is used to express the extent of satisfaction with the situation that f(i) is assigned to the next release. A measure of satisfaction is different from that of value because satisfaction expresses the urgency with which this stakeholder desires the corresponding feature.

3.8.4 Feature interdependency constraints

Various interdependencies between features are possible, and most features are typically involved in some sort of interdependency relationship [9], hence increasing the complexity of release planning. In our formulation, we consider two types of interdependency constraints: a coupling relation called C and a precedence relation called P. Coupled features should be released jointly because they depend on each other – this dependency could be due to implementation or usage issues. Precedence features should be released in a certain order because offering a particular feature might not make sense if it depends on a related feature that won't appear until a later release. Both relations are subsets of the product set $F \times F$. Two coupled features f(i) and f(j) are written as (i,j) $\in$ C, while feature f(i) is in a precedence relation to feature f(j) (written as $(i,j) \in P$) if feature f(j) is not allowed to be implemented in a release earlier than f(i). In terms of the introduced decision variables, this means that:

$x(i) = x(j)$ for all $(i,j) \in C \subset F \times F$ (Coupling)

$x(i) \leq x(j)$ for all $(i,j) \in P \subset F \times F$ (Precedence)

3.8.5 Resource consumptions

Resources required for the implementation of features are essential part of release planning. The resources refer to any input to the software production process, which includes personnel, materials, software, hardware, and so on. Usually, these resources relate to either budget or effort consumptions, and there are bounds on the maximum capacities available for each resource type in each release cycle. In the general model, we have considered R type of resources tentatively involved in the implementation of the features. Thus, we define resource consumption resource(i,r) for each feature i = 1,…,n and resource type r = 1,…,R. Correspondingly, we define resource capacities Cap(r,k) for each resource type r = 1,…,R and all releases k = 1,…,K. To become a feasible plan, decision variables must satisfy

$$\sum_{x(i)=k} \text{resource}(i,r) \leq \text{Cap}(r,k) \quad (6)$$

for all resource types r =1,…,R and all releases k=1,…,K.

3.8.6 System constraints

In order to implement a feature, one or more of existing components of system S must be modified to integrate the feature. Suppose we have a mapping Ψ from the set of features to the power set of all components, then each feature f(i) is assigned a set of impacted components $\Psi(i) \subset S$. We assume that, for each release k, the impact of implementing new features (which includes updating existing ones) in existing system S is restricted by a user-defined threshold β(k). The actual value of the threshold is problem-specific and can be varied to study the sensitivity of proposed solutions with respect to this value.

$$\sum_{x(i)=k} \text{Impact}(f(i)) \leq \beta(k) \quad (7)$$

for all releases k = 1,…,K.

3.8.7 Objective function

The objective of release planning is typically a mixture of different aspects such as value, urgency, risk, satisfaction, and return on investment. The goal is to define a function that tries to bring these different aspects together in a balanced way. We assume an additive function in which the total value of the objective function is determined as the sum of the weighted average priority WAP(i, p) defined on the values and satisfactions that stakeholders derive from all features f(i) when assigned to release option k. According to this assumption, the resulting objective function F(x) is defined as:

$$F(x) = \sum_{k=1\ldots K} \xi(k) \left[\Sigma_{x(i)=k} \mathrm{WAP}(i,p)\right] \quad (8)$$

where the *weighted average priority* WAP(i,p) is given as:

$$\mathrm{WAP}(i,p)=\sum_{p=1\ldots q} \lambda(p)\cdot \mathrm{sat}(i,p)\cdot \mathrm{value}(i,p) \quad (9)$$

for all features f(i) and releases k = 1…K. For each release option k, parameter ξ(k) describes the relative importance of the release option and its relative impact on the objective function.

3.8.8 Formal Problem Statement

Having followed through the problem modeling process, we need to represent the formulation in the format of a typical optimization problem, following the formulation discussed by Denzinger and Ruhe [14]. Now, suppose we represent the computed objective function in the right-hand-side of Equation (8) simply as v[i,k] (i.e. the objective function score when feature f(i) is assigned to release k), and x[i,k] ∈ {0,1} (i.e. binary decision variables, with x[i,k] = 1 if feature f(i) is assigned to release k, and 0 otherwise). Then the formal problem statement becomes:

Maximize:

$$\sum_{k=1\ldots K} \sum_{i=1\ldots n} v[i,k]\cdot x[i,k] \quad (10)$$

Subject to:

$$\sum_{i=1\ldots n} resource[i,r]\cdot x[i,k] \le Cap(r,k), \qquad r = r\ldots R;\ k = 1\ldots K \quad (11)$$

$$\sum_{k=1\ldots K} x[i,k] \le 1 \quad (12)$$

$$x[i_1,k] = x[i_2,k], \qquad k = 1\ldots K, \qquad \forall\ (i_1,i_2) \in C \quad (13)$$

$$\sum_{k=1\ldots K} \left(K+1-k\right)\left(x[i_1,k]-x[i_2,k]\right) \ge 0, \qquad \forall (i_1,i_2) \in P \quad (14)$$

$$\sum_{i=1\ldots n} impact[i]\cdot x[i,k] \le \beta(k), \qquad k = 1\ldots K \quad (15)$$

$$x[i,k] \in \{0,1\} \quad (16)$$

The given formulation of the RP problem in Equations (10)-(16) constitutes an integer linear programming (ILP) problem. Equation (10) maximizes the cumulative value and satisfaction scores by selecting the best possible blend of features. Equation (11) guarantees that any

release plan alternatives generated do not violate the resource constraints. Equation (12) ensures that no one feature can be assigned to more than one release. Equation (13) ensures that coupled features are release together, while Equation (14) ensures that feature i_1 is never released later than feature i_2. Finally, Equation (15) makes sure that features are selected within the tolerable impact level for an evolving system, while Equation (16) states that each feature can either be assigned to a release or unassigned to any release.

Although the formalized description we present in Equations (11) - (16) allows us apply efficient optimization algorithms that offer at least near-optimal solutions, it is highly unlikely that formalization of the problem would completely model the reality. As discussed in Section 2.5, decision support for release planning need to augment formal models with the experience of human decision maker. Our approach to providing decision support would follow this direction, as we have discussed in [51]. So, our approach would formulate a series of problems as variants of the original formal model. Then we solve these problem variants to generate a set of qualified alternative solutions. A human decision maker - such as the project manager - evaluates the solutions based on experience and familiarity with the problem context.

3.8.9 Fuzzy modeling of uncertainty in feature interdependencies

The need to model the uncertainty associated with determining the existence of interdependency between features is based on the fact that judgment about whether two features are dependent may not be easy to make. This is further complicated by the fact that different experts might have different opinion about whether a particular dependency exists or it does not exist. In other words, we need to capture the subtlety of the situation by allowing knowledge elicitation from experts based on their degree of belief in the existence of feature dependency. Carlshamre *et al.* [9] observe the need to base management of interdependencies on the strength of such interdependencies rather than type of interdependencies, but did not go further on how to formally achieve this. We first initiated an approach to model this vagueness with the relaxation of feature dependency constraints evaluation using fuzzy theory in [37]. We discussed an extension to this initial model in [38], where we employed fuzzy graphs to structurally represent the whole set of dependency constraints present in the entire features of the project.

The entire set of dependency constraints specified in the release planning problem are materialized as a fuzzy graph G = (F, C ∪ P), where F is the vertex of the graph representing the features and C is the non-directed edge representing coupling and P is a directed edge representing precedence constraints between any two vertices. The edges are labeled with the "degree of belief" ($\mu_p(i,j)$ for precedence and $\mu_c(i,j)$ for coupling) given by the PM based on his knowledge of the existence of the corresponding dependencies (i.e. C(i,j) or P(i,j)). Each solution plan generated would also correspond to a fuzzy graph. The fuzzy graph must be acyclic, except when all the edges involved in the subgraph with cycle represent coupling dependencies whose edges are not directed. An example fuzzy graph is shown in Figure 7.

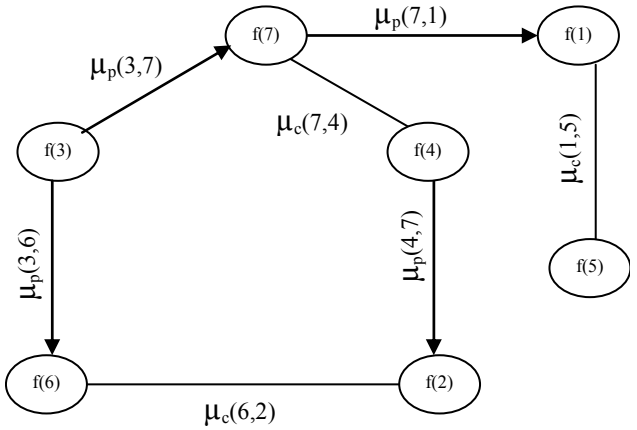

Figure 7: Sample fuzzy graph of dependency constraints

The project manager (PM) could give a more subtle judgment for degree of belief, such as "likely", "almost sure", etc. In this case, we code each such judgment by a value (sure = 1, almost sure = 0.8, likely = 0.6…). Using this scheme, the relations C and P become "fuzzy relations", that is the fuzzy subsets of F x F.

$$C = \{ ((i,j), \mu_C(i,j)) \mid (i,j) \in F \times F, \mu_C(i,j) \in [0,1] \}$$
$$P = \{ ((i,j), \mu_P(i,j)) \mid (i,j) \in F \times F, \mu_P(i,j) \in [0,1] \}$$

Once the uncertainty concerning the degree of "coupling" or "precedence" is represented by fuzzy relations C and P, the next question is how to exploit these relations. We have two possibilities to explore: an individual approach that consists of deciding whether to include each dependency constraint, and a holistic approach considering a whole set of dependency constraints. Following the first approach is rather simple: the PM can adjust a "fuzzy

dependency threshold" $\alpha \in [0,1]$ in order to eliminate "weaker" constraints. In our case, we explore the second approach representing dependency constraints by a graph, and defining level of satisfaction by distance between graphs.

We can define the two fuzzy relations

$$C_x(i,j) = \begin{cases} C(i,j) & if\ (C(i,j) > 0\ and\ y[i] = y[j]) \\ 0 & Otherwise \end{cases}$$

$$P_x(i,j) = \begin{cases} P(i,j) & if\ (P(i,j) > 0\ and\ y[i] = y[j]) \\ 0 & Otherwise \end{cases}$$

A graph of the solution does not necessarily have to be a perfect match or replica of the problem fuzzy graph. There could be violation of some dependencies. Once a release plan x is generated, the graph constructed from x is G(x) = (R, $C_x \cup P_x$) and the fuzzy graph representing the problem constraints is G(p) = (R,C $\cup$ P). The level of satisfaction of all the dependency constraints of solution x, denoted by $\mu_D(x)$, can be defined as the "similarity" between these two graphs.

$$\mu_D(x) = 1 - \delta(G(x), G(p))$$

where δ is the measure of the "distance" (or dissimilarity) between two graphs. A measure of dissimilarity between intuitionistic fuzzy subgraph was proposed by Li [32]. The concept of intuitionistic fuzzy set introduced by Atanassov [3] is a generalization of the conventional fuzzy set; generalization includes degree of membership and degree of non-membership of every element of a set. If we specialize the intuitionistic dissimilarity measure proposed in [3], we will obtain the following dissimilarity measure for conventional fuzzy graphs:

$$D(G(x), G(p)) = \sum_{i,j} \left| C(i,j) + P(i,j) - C_x(i,j) - P_x(i,j) \right|$$

By definition, for all x, i, j we have $C(i,j) \geq C_x(i,j)$and $P(i,j) \geq P_x(i,j)$, therefore

$$D(G(x), G(p)) = C(i,j) + P(i,j) - C_x(i,j) - P_x(i,j)$$

The distance between two fuzzy graphs can be described as the normalized dissimilarity defined as:

$$\delta(G(x), G(p)) = \frac{D(G(x), G(p))}{\sum_{i,j} (C(i,j) + P(i,j))}$$

Recalling that,

$$\mu_D(x) = 1 - \delta\big(G(x), G(p)\big)$$

We have the following,

$$\mu_D(x) = 1 - \frac{\sum_{i,j}\big(C(i,j) + P(i,j) - C_x(i,j) - P_x(i,j)\big)}{\sum_{i,j}\big(C(i,j) + P(i,j)\big)}$$

$$\mu_D(x) = 1 - \left(1 - \frac{\big(\sum_{i,j}\big(C_x(i,j) + P_x(i,j)\big)\big)}{\sum_{i,j}\big(C(i,j) + P(i,j)\big)}\right)$$

Therefore,

$$\mu_D(x) = \frac{\sum_{i,j}\big(C_x(i,j) + P_x(i,j)\big)}{\sum_{i,j}\big(C(i,j) + P(i,j)\big)}$$

The gain in this dependency modeling approach is that, we would be able to relax the stringent demand that a specific interdependency constraint must be preserved, even as the software engineers are not exactly sure about the existence of such interdependencies. The fuzzy modeling extension constitutes another dimension of flexibility introduced to the interdependencies among features, which still lends itself to the (ILP) problem model discussed above. In the fuzzy-based version of the implementation, the interdependencies constraints in Equations (13) and (14) would be replaced by fuzzy interdependencies, and we can evaluate the degree to which the constraints are satisfied in different solutions.

4. Feasibility Study and Preliminary Results

In this section, we discuss initial experiences drawn from a case study investigating the feasibility of the decision framework proposed in this research. Details of the case study design and analysis is contained in [53].

4.1 The Target System – ReleasePlanner®

ReleasePlanner® is a decision support system able to perform web-based RP and prioritization based on comprehensive stakeholder involvement. It combines innovative ideas from computational intelligence, mathematical optimization, multi-criteria decision aid and intelligent decision support systems. The system currently comprises eight main components and is more than 80K source lines of code in size. The overall effort in research and development incorporated into the system is about 22 person-years. Among the companies who have performed trial or professional projects using the tool include: Corel iGrafx, Siemens Corporate Technology, Trema Laboratories, Nortel Networks, Solid Technology, City of Calgary, Autotech, and Ericsson Canada.

We have chosen this evolving system for our experiment because of the difficulty of getting assess to industrial projects at this stage. Efforts would be geared towards validating the results of this research in other industrial settings.

4.2 Applying S-EVOLVE* to Evolution of the ReleasePlanner®

Based on the releasing planning technique we have described in Chapter 3, we carried out a proof-of-concept case study experiment and collected results to evaluate the impact of implementing new features on existing components. The result was used in the current RP model to plan for releases 1.3 and 1.4 of the system. This pilot study did not include the use of fuzzy dependency constraints, as we hope to further evaluate that during the next course of this research. In the following we summarize the process and discuss the results:

4.2.1 Case Study Design

The presented case study involves performing RP for the ReleasePlanner® tool. There were 33 features desired for the next two releases of the system. Four system experts consisting of developers, designers and GUI specialists were involved in performing the DoM and XoM assessments.

In planning the study, a small workshop was held to present the challenges of RP for evolving systems. Our proposed method was also presented to the experts since they will be involved in the assessment of DoM and XoM for components of the system. After the presentation, the expert participants were asked to identify the components making up the system. The following 8 components were identified: *Reporting component, Validator component, IP component, Java Broker component, Import/Export component, Stakeholder Voting Analysis component, DB Connectivity component, and Alternative Analysis Wizard component*.

To enable us accommodate different perspectives about the difficulty of modifying each of the components, the participants were given instructions to carry out their assessment of DoM independently and with no interactions, but following the AHP-based scheme we have discussed.

4.2.2 Results

Our preliminary results indicate that factoring impact characterization into the planning process is highly essential, as strong correlation exists between the strictness of impact threshold and the value of the objective function. We also discovered that there are substantial changes in the structure of the release plans depending on the impact threshold adopted. These results are not overly surprising, since we believe that adding another dimension of constraints would restrict the value of the objective function attainable vis-à-vis the features that could be accommodated.

4.3 Related Publications

The following contains a listing of my publications that are directly related to the research proposal presented in this document:

Journal Publications:

[1] Saliu, O. and Ruhe, G., "Software Release Planning for Evolving Systems", *Innovations in Systems and Software Engineering: a NASA Journal*, 1(2), pp. 189-204, September 2005.

[2] Ruhe, G. and Saliu, M. O., "The Art and Science of Software Release Planning", *IEEE Software*, 26(6), pp. 47-53, Nov/Dec 2005.

Book Chapters:

[3] Maurice, S., Ruhe, G., Saliu, O., Ngo-The, A., and Brassard, R., "Decision Support for Value-Based Software Release Planning", in: S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, P. Grünbacher (Eds.), *Value-Based Software Engineering Management*, Springer, September 2005, ISBN: 3-540-25993-7.

Refereed Conferences:

[4] Ruhe, G. and Saliu, O., "How to Plan for Releases of Software Products?" (Tutorial) *28th International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, May 20-28, 2006 (submitted).

[5] Saliu, O. and Ruhe, G., "Supporting Software Release Planning Decisions for Evolving Systems", *29th IEEE/NASA Software Engineering Workshop (SEW-29)*, Greenbelt, MD, USA, April 6-7, 2005, pp. 14-26.

[6] Ngo-The, A. and Saliu, M. O., "Measuring Dependency Constraint Satisfaction in Software Release Planning using Dissimilarity of Fuzzy Graphs", *4th IEEE International Conference on Cognitive Informatics (ICCI'05)*, Irvine, California, USA, Aug. 8-10, 2005, pp. 301-307.

[7] Ngo-The, A. and Saliu, M. O., "Fuzzy Structural Dependency Constraints in Software Release Planning", *2005 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2005)*, Reno, Nevada, USA, May 22-25, 2005, pp. 442-447.

[8] Ruhe, G., Saliu, O., Bhawnani, P., Momoh, J. and Ngo-The, A. "Decision Support for Software Release Planning - Methods, Tools, and Practical Experience", (Tutorial) *29th IEEE/NASA Software Engineering Workshop (SEW-29)*, Greenbelt, MD, USA, April 3 & 8, 2005, pp. 217-250.

Posters:

[9] Saliu, O. "S-EVOLVE* - A Decision Support Method for Planning Releases of Evolving Software Systems", *Students Poster Competition*, 2005 Informatics Circle of Research Excellence (iCORE) Summit, Banff, Canada, Aug 31 - Sep 2, 2005.

4.4 Summary

In this research proposal, we have discussed a methodology for planning releases of evolving software systems. The proposed methodology involves a process decision framework that encourages characterization of system components based on selected quality attribute and lower level factors, in order to assess the impact of implementing one or more features. To date, we have created initial characterization scheme for assessing system components. We have also developed a systematic expert judgment-based approach for assessing the DoM of system components. Each of these parts will provide us with a starting point for the proposed work. The S-EVOLVE* solution approach we have discussed would form the basis of our work in this research. Preliminary results from a case study using S-EVOLVE* reveals the importance of accounting for all relevant factors before making feature selection and scheduling decisions. Essentially, we still need to focus on addressing the following:

1. *Modeling effect of new components due to feature implementation:* So far, we have concentrated efforts on accounting impact of features on existing systems due to modification required on existing components. In some cases, feature implementation would require both modifications on existing components as well as implementation of new components. We need to further investigate how this can be factored into our impact computation.
2. *Resource estimation:* We need to investigate the way that knowledge we derive from assessing DoM and XoM could help in making more accurate resource estimation for software evolution planning. In this direction, a promising route to explore would involve using productivity of programmers on combination of XoM which accounts for volume of changes to be made, and DoM as an impacting cost driver.
3. *Empirical studies:* There are needs to further test the validity of our assumptions. We have initially carried out a case study on ReleasePlanner® system to verify the added advantage of the proposed methodology, but further validations are necessary in this regard. Our future empirical evaluations would still involve this system as well as other systems. I am currently planning a 3-4 months internship in the second quarter of 2006 at the Fraunhofer Institute of Experimental Software Engineering, Maryland (FC-MD), USA. The purpose of the internship is to look for avenues to get exposed to other evolving software systems emanating from different domains, on which we

can further perform empirical validation of the proposed methodology. Some of the hypotheses that I would like to verify include:

(i.) Evaluating whether or not software engineers find the proposed component characterization and evaluation techniques for assessing evolving systems intuitive and easy to use.

(ii.) Our claim that factoring component characteristics into release planning process results in more realistic plans that are likely to be successfully implemented.

(iii.) Assumptions that systematic approach to eliciting experiences of human experts in order to evaluate software system components would provide meaningful results. Case studies would help verify both valid and invalid assumptions we have made about the components characterization and assessment schemes.

4. *Implementation of the fuzzy interdependencies:* We will implement the fuzzy interdependencies extension and evaluate its contribution to release planning decisions.
5. *Refining and improving component assessment scheme:* This would derive from the results and feedback received through our empirical studies. Experiences gained from empirical studies would guide the improvement in assessment of DoM and XoM.

In conclusion, our proposed research offers a fundamentally new approach that seeks to systematically utilize the experience of experts in assessing the impact of implementing proposed features in evolving systems. This research is expected to offer a more realistic guidance on assessing amount of work required for feature implementation, before release planning decisions are made.

5. Thesis Research Schedule

Based on the progress of the research work, these schedules may be adjusted in consultation with the thesis supervisor. I will present progress of the proposed research work at the Computer Science Departmental graduate student seminar in due course. I am currently preparing a submission to the ICSE-2006 Doctoral Symposium that would enable us present the research. We estimated that the final thesis for the PhD degree will be defended and submitted by the end of Winter 2007.

Literature survey, defining research agenda and proposal writing:	Winter 2004 – Fall 2005
PhD Candidacy examinations:	October/November 2005
Thesis Research:	Dec. 2005 – Dec. 2006
Internship at Fraunhofer-IESE, USA	April 2006 – July 2006
Expected Thesis Defense and Submission:	Fall 2006

6. References

[1] Ahmed, M., Saliu, M. O., and AlGhamdi, J., "Adaptive Fuzzy Logic-Based Framework for Software Development Effort Prediction," *Information and Software Technology*, vol. 47, no. 1, pp. 31-48, 2005.

[2] Akker, J. M., Brinkkemper, S., Diepen, G., and Versendaal, J. M., "Flexible Release Planning Using Integer Linear Programming," In Proceedings of 11th International Workshop on Requirements Engineering for Software Quality (REFSQ'05), Porto, Portugal, 2005.

[3] Atanassov, K. T., "Intuitionistic Fuzzy Sets," *Fuzzy Sets and Systems*, vol. 20, no. 1, pp. 87-96, 1986.

[4] Bagnall, A. J., Rayward-Smith, V. J., and Whittley, I. M., "The Next Release Problem," *Information and Software Technology*, vol. 43, no. 14, pp. 883-890, 2001.

[5] Bengtsson, P., Lassing, N., Bosch, J., and van-Vliet, H., "Architecture-Level Modifiability Analysis (ALMA)," *The Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129-147, 2004.

[6] Bohanec, M., "What is Decision Support?" In Proceedings of 4th International Multi-conference Information Society (IS-2001), Ljubljana, 2001, pp. 86-89.

[7] Bohner, S. A. and Arnold, R. S., "Software Change Impact Analysis." Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[8] Carlshamre, P., "Release Planning in Market-Driven Software Product Development: Provoking an Understanding," *Requirements Engineering*, vol. 7, no. 3, pp. 139-151, 2002.

[9] Carlshamre, P., Sandahk, K., Lindvall, M., Regnell, B., and Nattoch Dag, J., "An Industrial Survey of requirements interdependencies in Software Release Planning," In Proceedings of 5th IEEE International Symposium on Requirements Engineering, Toronto, Canada, 2001, pp. 84-91.

[10] Christel, M. G. and Kang, K. C., "Issues in Requirements Elicitation," SEI, Carnegie Melon University, Pittsburgh, CA CMU/SEI-92-TR-12, 1992.

[11] CMMI, "Capability Maturity Model Integration (CMMI®) Version 1.1 Staged Representation," SEI, Carnegie Mellon University, Pittsburgh, PA 2002.

[12] Dayani-Fard, H., "Quality-Based Software Release Management," PhD, School of Computing, Queen's University, Canada, Kingston, Ontario, 2003

[13] Denne, M. and Cleland-Huang, J., "The Incremental Funding Method: Data Driven Software Development," *IEEE Software*, vol. 21, no. 3, pp. 39-47, 2004.

[14] Denzinger, J. and Ruhe, G., "Decision Support for Software Release Planning using e-Assistants," *Journal of Decision Support Systems*, vol. 13, no. 4, pp. 399-421, 2004.

[15] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1-12, 2001.

[16] Farbey, B. and Finkelstein, A., "Exploiting Software Supply Chain Business Architecture: A Research Agenda," In Proceedings of 1st Workshop on Economics-Driven Software Engineering Research (EDSER-1), 21st International Conference on Software Engineering (ICSE), Los Angeles, CA, 1999.

[17] Fowler, M. and Beck, K., "Bad Smells in Code," in *Refactoring: Improving the Design of Code*1st ed, Boston: Addison-Wesley, 2000, pp. 75-88.

[18] Gall, H., Hajek, K., and Jazayeri, M., "Detection of Logical Coupling Based on Product Release History," In Proceedings of IEEE International Conference on Software Maintenance, Washington D.C., 1998, pp. 190-198.

[19] Gall, H., Jazayeri, M., Klosch, R. R., and Trausmuth, G., "Software Evolution Observations Based on Product Release History," In Proceedings of International Conference on Software Maintenance (ICSM'97), Bari, Italy, 1997, pp. 160-166.

[20] Graves, T. L., Karr, A. F., Marron, J. S., and Siy, H., "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, 2000.

[21] Greer, D., "Decision Support for Planning Software Evolution with Risk Management," in *16th International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*. Banff, Canada, 2004, pp. 503-508.

[22] Grubb, P. and Takang, A. A., "Software Maintenance: Concepts and Practice," 2nd ed: World Scientific, 2003.

[23] Hassan, A. E. and Holt, R. C., "Predicting Change Propagation in Software Systems," In Proceedings of International Conference on Software Maintenance, Chicago, Illinois, USA, 2004, pp. 284-293.

[24] Hassan, A. E. and Holt, R. C., "Studying the Chaos of Code Development," In Proceedings of Working Conference on Reverse Engineering (WCRE 2003), Victoria, Canada, 2003, pp. 123-133.

[25] Jönsson, P. and Lindvall, M., "Impact Analysis," in *Engineering and Managing Software Requirements*, Aurum, A. and Wohlin, C. (Eds.), Heidelberg, Germany: Springer, 2005, pp. 116-142.

[26] Jung, H.-W., "Optimizing Value and Cost in Requirements Analysis," *IEEE Software*, vol. 15, no. 4, pp. 74-78, 1998.

[27] Karlsson, J. and Ryan, K., "A Cost-Value Approach for Prioritizing Requirements," *IEEE Software*, vol. 14, no. 5, pp. 67-74, 1997.

[28] Karlsson, J., Wohlin, C., and Regnell, B., "An Evaluation of Methods for Prioritizing Software Requirements," *Information and Software Technology*, vol. 39, no. 14-15, pp. 939-947, 1998.

[29] Lehman, M. M., "Laws of Software Evolution Revisited," In Proceedings of 5th European Workshop on Software Process Technology (EWSPT'96), Nancy, France, 1996, pp. 108-124.

[30] Lehman, M. M., "On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle," *Journal of System and Software*, vol. 1, no. 3, pp. 213-221, 1980.

[31] Lehtola, L., Kauppinen, M., and Kujala, S., "Requirements Prioritization Challenges in Practice," In Proceedings of PROFES'2004, Kansai Science City, Japan, 2004, pp. 497-508.

[32] Li, D.-F., "Some Measures of Dissimilarity in Intuitionistic Fuzzy Structures," *Journal of Computer and System Sciences*, vol. 68, no. 1, pp. 115-122, 2004.

[33] Lindvall, M. and Sandahl, K., "How Well do Experienced Software Developers Predict Software Change?" *Journal of Systems and Software*, vol. 43, no. 1, pp. 19-27, 1998.

[34] Lock, S. and Kotonya, G., "An Integrated Framework for Requirement Change Impact Analysis," In Proceedings of 4th Australian Conference on Requirements Engineering, Sydney, Australia, 1999, pp. 29-42.

[35] Mockus, A. and Weiss, D. M., "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169-180, 2000.

[36] Natt-och-Dag, J., Regnell, B., Gervasi, V., and Brinkkemper, S., "A Linguistic-Engineering Approach to Large-Scale Requirements Management," *IEEE software*, vol. 22, no. 1, pp. 32-39, 2005.

[37] Ngo-The, A. and Saliu, M. O., "Fuzzy Structural Dependency Constraints in Software Release Planning," In Proceedings of 2005 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2005), Reno, Nevada, USA, 2005.

[38] Ngo-The, A. and Saliu, M. O., "Measuring Dependency Constraint Satisfaction in Software Release Planning using Dissimilarity of Fuzzy Graphs," In Proceedings of 4th IEEE International Conference on Cognitive Informatics (ICCI'05), Irvine, California, USA, 2005, pp. 301-307.

[39] Nikora, A. P. and Munson, J. C., "Dertemining Fault Insertion Rates for Evolving Software Systems," In Proceedings of 9th International Symposium on Software Reliability Engineering, 1998, pp. 306-315.

[40] Ohlsson, M. C., Andrews, A. A., and Wohlin, C., "Modelling Fault-Proneness Statistically over a Sequence of Releases: A Case Study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 3, pp. 167-199, 2001.

[41] Penny, D. A., "An Estimation-Based Management Framework for Enhancive Maintenance in Commercial Software Products," In Proceedings of International Conference on Software Maintenance (ICSM'02), Montreal, Canada, 2002, pp. 122-130.

[42] Pfleeger, S. L. and Bohner, S. A., "A framework for software maintenance metrics," In Proceedings of IEEE Conference on Software Maintenance, San Diego, CA, USA, 1990, pp. 320-327.

[43] Porter, A. A. and Selby, R. W., "Empirically Guided Software Development using Metric-Based Classification Trees," *IEEE Software*, vol. 7, no. 2, pp. 46-54, 1990.

[44] Rajlich, V., "Modeling Software Evolution by Evolving Interoperation Graphs," *Annals of Software Engineering*, vol. 9, no. 1-4, pp. 235-348, 2000.

[45] Regnell, B., Beremark, P., and Eklundh, O., "A Market-driven Requirements Engineering Process – Results from an Industrial Process Improvement Programme," *Requirements Engineering*, vol. 3, no. 20, pp. 121-129, 1998.

[46] Roche, J. M., "Software metrics and measurement principles," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 1, pp. 77-85, 1994.

[47] Ruhe, G., "Software Engineering Decision Support – Methodology and Applications," in *Innovations in Decision Support Systems*, Tonfoni and Jain (Eds.), 2003, pp. 143-174.

[48] Ruhe, G., "Software Engineering Decision Support and Empirical Investigations - A Proposed Marriage," In Proceedings of Empirical Studies in Software Engineering (ESEIW'03), Rome, Italy, 2003, pp. 25-34.

[49] Ruhe, G., "Software Release Planning," in *Handbook of Software Engineering and Knowledge Engineering*, vol. 3, Chang, S. K. (Ed.): World Scientific Publishing, 2005.

[50] Ruhe, G. and Ngo-The, A., "Hybrid Intelligence in Software Release Planning," *International Journal of Hybrid Intelligent Systems*, vol. 1, no. 2, pp. 99-110, 2004.

[51] Ruhe, G. and Saliu, M. O., "The Art and Science of Software Release Planning," *IEEE Software*, vol. 26, no. 6, pp. 47-53, 2005.

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

[53] Saliu, O. and Ruhe, G., "Software Release Planning in Evolving Systems," *Innovations in Systems and Software Engineering - A NASA Journal*, vol. 1, no. 2, pp. 189-204, 2005.

[54] Saliu, O. and Ruhe, G., "Supporting Software Release Planning Decisions for Evolving Systems," In Proceedings of 29th IEEE/NASA Software Engineering Workshop (SEW-29), Greenbelt, MD, USA, 2005, pp. 14-26.

[55] Schneidewind, N. F. and Hoffman, H. M., "An Experiment in Software Error Data Collection and Analysis," *IEEE Transactions of Software Engineering*, vol. 5, no. 3, pp. 276-286, 1979.

[56] Sneed, H. M., "Impact Analysis of Maintenance Tasks for a Distributed Object-Oriented System," In Proceedings of IEEE International Conference on Software Maintenance (ICSM'01), Florence, Italy, 2001, pp. 180-189.

[57] Standish, "CHAOS Report," Standish Group Research, 2002.

[58] Turner, C. R., Fuggetta, A., Lavazza, L., and Wolf, A. L., "A Conceptual Basis for Feature Engineering," *Journal of Systems and Software*, vol. 49, no. 1, pp. 3-15, 1999.

[59] van-Gurp, J., Bosch, J., and Svahnberg, M., "Managing Variability in Software Product Lines," In Proceedings of LAC2000, Amsterdam, 2000.

[60] Van Scoy, R. L., "Software Development Risk: Opportunity, Not Problem," Software Engineering Institute, Pittsburgh (CMU/SEI-92-TR-30, ESC-TR-92-030), 1992.

[61] Ying, A. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., "Predicting Source Code Changes by Mining Change History," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574-586, 2004.

[62] Yoon, P. K. and Hwang, C.-L., "Multiple Attribute Decision Making: An Introduction," Sage Publications, 1995.

[63] Zhao, J., Yang, H., Xiang, L., and Xu, B., "Change Impact Analysis to Support Architectural Evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 5, pp. 317-333, 2002.

[64] Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A., "Mining Version Histories to Guide Software Changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429-445, 2005.