TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Hadaytullah

Applying Genetic Architectural Synthesis in Software Development and Run-time Maintenance

Julkaisu 1247 • Publication 1247

Tampere 2014

Tampereen teknillinen yliopisto. Julkaisu 1247
Tampere University of Technology. Publication 1247

Hadaytullah

Applying Genetic Architectural Synthesis in Software Development and Run-time Maintenance

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB219, at Tampere University of Technology, on the 10th of October 2014, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2014

ISBN 978-952-15-3365-5 (printed)
ISBN 978-952-15-3376-1 (PDF)
ISSN 1459-2045

Abstract

Software systems are becoming complex entities with an increasing diffusion into many new domains. A complex software system requires more resources to develop and maintain. Some domains demand continuous operation like security or control systems, web services and communication systems etc. The trend will lead software industry to a situation where it will be difficult to develop software systems through traditional manual software engineering practices in a feasible budget. Any level of automation can relieve the pressure on the cost. This thesis work explores the potential of genetic architectural synthesis to introduce automation in software development and maintenance.

The genetic algorithm operates at the architectural level. The fitness functions envelop the expert knowledge needed to gauge the quality (modifiability, efficiency and complexity) of architectures. The algorithm uses solutions which can be design patterns, architectural styles, best practices or application specific solutions to maintain the quality attributes. Each solution has a positive or negative impact on one or more quality attributes. Once calibrated, the genetic algorithm has been able to suggest good quality architectures. An empirical study has also been performed that suggests that the genetic algorithm's proposals are comparatively better than the under-graduate level students' designs. Tool support has been provided in the form of the Darwin environment. It facilitates a human architect to initiate, modify, monitor and analyze the results of a genetic architectural synthesis. Moreover, the genetic algorithm has been used to evolve software architectures to be easily distributable among the teams involved in its development. The algorithm takes into account the organizational information and proposes an initial work distribution plan along with the improved architecture.

The SAGA (Self-Architecting using Genetic Algorithms) infrastructure has been developed to enable self-adaptive and manual run-time maintenance in Java-based applications. SAGA allows Java-based distributed systems to self-maintain reliability and efficiency. Furthermore, non-self-maintainable properties of a system can be maintained manually at run-time. The decision making engine is the genetic algorithm. The unit of run-time modification is an architectural solution which in its entirety enters

of leaves the running instance of a system therefore affecting the system's run-time quality. A solution is composed of roles which are bound to real artifacts in the system. Multiple attributes concerning reliability and efficiency of the running system are monitored by SAGA. In the case of poor system quality in a changed environment, SAGA triggers the genetic algorithm to propose improvements in the architecture taking into account the monitoring data. The proposal is then reflected to the run-time and the cycle continues. In the experiments, an example distributed system used in changing environment has been implemented with self-maintaining capability. A significant improvement in both reliability and efficiency of the running system has been observed.

Preface

This work has been carried out from 2009 till 2013 under the Darwin project in the Department of Pervasive Computing at Tampere University of Technology, Finland. The work done under the thesis was funded by the Academy of Finland and Graduate School of Tampere University of Technology.

I like to express my deepest gratitude to Prof. Kai Koskimies for his valuable support and guidance during the thesis work. I also want to thank him for maintaining a positive atmosphere in the research group where newbie like me can excel and produce scientific results. I am also grateful to Prof. Kari Systä for the valuable discussions and feedbacks during the thesis work. I also like to thank Prof. Tarja Systä for showing confidence in me and recommending me for this Ph.D. work in 2009.

I am thankful to all my colleagues for their contributions and support throughout the thesis work. Special thanks to Outi Sievi-Korte and Sriharsha Vathsavayi for contributing to the techniques and tools developed during this thesis work. I also wish to thank Imed Hammouda and Anna Ruokonen for reviewing some of my work.

A big thank to Allan Gregersen and Bo Jørgensen of Maersk McKinney Moller Institute at the University of Southern Denmark for collaborating and sharing their technologies with us. I am deeply thankful for the swift updates to the shared technology in response to my requests.

Finally, I am grateful to my parents and the whole family for their love, prayers and care throughout my entire life. I am also thankful to all my friends for their support. You all are wonderful people and I am blessed to have you guys in my life. Thanks a lot!

Contents

Terms and Definitions

AST	Abstract Syntax Tree
CASE	Computer Aided Software Engineering
CO	Communication Overhead
DCEVM	Dynamic Code Evolution Virtual Machine
DM	Decision Maker
DNA	Deoxyribonucleic Acid
GA	Genetic Algorithm
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
J2EE	Java Enterprise Edition
JITA	Just in Time Architecture
JMS	Java Messaging Service
JVM	Java Virtual Machine
MDG	Module Dependency Graph
MOF	Meta-Object Facility
MOFM2T	MOF Model to Text Transformation
MOGA	Multi-Objective Genetic Algorithm
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MVC	Model View Controller
NGA	Nested Genetic Algorithm

NP-Hard	Non-deterministic Polynomial-time Hard
OA	Operation Allocation
OMG	Object Management Group
OOP	Object Oriented Programming
QoS	Quality of Service
SAGA	Self-Architecting using Genetic Algorithms
SASSY	Self-Architecting Software Systems
SOA	Service Oriented Architectures
SRF	Software Renovation Framework
STF	State Transfer Function
UI	User Interface
UML	Unified Modeling Language

List of Figures

List of Tables

List of Included Publications

[P1] Outi Räihä, Hadaytullah, Kai Koskimies and Erkki Mäkinen. Synthesizing Architecture from Requirements: A Genetic Approach. Relating Software Requirements and Architecture (eds. P. Avgeriou, J. Grundy, J.G. Hall, P. Lago, I. Mistrik), Chapter 18, Springer 2011, pp. 307-331.

[P2] Hadaytullah, Sriharsha Vathsavayi, Outi Räihä and Kai Koskimies. Tool Support for Software Architecture Design with Genetic Algorithms. In Proceedings of International Conference on Software Engineering Advances, Nice, France, August 2010, IEEE Computer Society Press, pp. 359-366.

[P3] Hadaytullah, Outi Räihä and Kai Koskimies. Genetic Approach to Software Architecture Synthesis with Work Allocation Scheme. In Proceedings of Asia Pacific Software Engineering Conference, Sydney, Australia, December 2010, IEEE Computer Society Press, pp. 70-79.

[P4] Hadaytullah, Allan Gregersen and Kai Koskimies. Pattern-Based Dynamic Maintenance of Software Systems. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), Hong Kong, December 2012, IEEE Computer Society Press, pp. 537-546.

[P5] Hadaytullah, Sriharsha Vathsavayi, Outi Räihä, Allan Gregersen and Kai Koskimies. Applying Genetic Self-Architecting for Distributed Systems. In Proceedings of 4th World Congress on Nature and Biologically Inspired Computing (NaBIC'12), Mexico City, Mexico, November 2012, IEEE, pp. 44-52.

PART I – Overview and Foundation

This part builds the motivation and presents the research questions and contributions made by this thesis work. This part also covers the fundamental concepts essential for understanding of this thesis work.

1 Introduction

1.1 Motivation

Software development life cycle is composed of many stages like planning, resource management, development, testing, maintenance etc. [1]. Due to the increasing complexity of modern software systems, the cost of each of the stages is growing rapidly. If the trend continues, we are facing a dilemma where the fruits of software systems will be overshadowed by the amount of resources required to develop and maintain them. One factor contributing to the cost is the nature of software development practices where majority of the tasks is performed manually. Certainly, more efficient methods can be developed resulting in cost effective execution of the software life cycle phases even in manual settings. However, manually performed tasks are typically slow with inherent threat of human errors that cannot be avoided. Furthermore, humans usually suffer from the Golden Hammer syndrome [2], one is inclined towards employing methods and solutions he or she is familiar with or successfully applied in the past. It is quite possible that potential solutions may be overseen by managers, architects, developers or test engineers.

In order to make a big cut in the cost, automation is the way to move forward. Many studies (e.g., [3][4][5][6][7][8][9] and [10]) have been performed to fully or partially automate various stages of software development. Automation promises reduced cost and faster software development. Furthermore, machines will objectively select a method for software development without suffering from the Golden Hammer syndrome. The full automation of all the stages is still a distant dream; however, any level of automation regardless of its scale is worth achieving.

Among the critical activities of software development is designing the architecture of a system. There are plenty of domains, each having different constraints, environment and variables which must be taken into account when laying foundation for a software system. Thus, software architecture designing requires experience,

ingenuity and creativity. At the same time, during the past few decades, practitioners have been able to capture the recurring practices in the form of design patterns [11], architectural styles [12], best practices, reference architectures etc. A sensible use of such available solutions can help reduce the load on the architect in the design phase.

The work distribution planning phase is equally resource consuming like software design, in fact both phases are inter-related. An architect should keep in mind work distribution issues while designing software architecture. An architecture designed in absence of the knowledge of the involved teams may lead to a poor dissemination of tasks and an increased inter-teams communication overhead [30]. Surely, it is not easy and becomes even more challenging in presence of teams located in different regions with different languages and cultural values.

Software maintenance is of utter importance when it comes to the cost of a software system as more than half of the budget of a software system is consumed by this phase [31]. Thus, automation of maintenance is a particularly attractive goal. One of the major types of maintenance is adaptive maintenance performed in response to changing circumstances [31][32][33]. Self-adaptive [34] maintenance is one step in the right direction where a system is expected to maintain itself according to the changing requirements or environment. Self-maintenance not only brings cost related advantages but it is highly desirable for systems with 24/7 operation demand. For software systems like electric grid controllers, web services, security systems, traffic lights, video control systems, and banking systems, maintenance breaks would cause business losses or other unwanted consequences.

Certainly, not everything can be self-maintained as maintenance can be triggered by a multitude of unpredictable situations. Typical maintenance activities like fixing bugs or introducing new features are still beyond the scope of the self-adaptive paradigm. In case of quality maintenance, there are many properties of software systems that machines cannot measure currently and therefore cannot be self-maintained. For example, subjective properties (e.g., usability, learnability etc.) are connected to human experiences which cannot be quantized automatically using currently available technologies. Unless new methods or technologies are developed, such properties will remain beyond the scope of the self-adaptive paradigm. Thus, a hybrid approach supporting both self-adaptive and manual run-time update mechanisms is a natural first step towards run-time maintenance support. The hybrid approach will allow for maintenance of a wider range of properties rather than only strictly self-adaptable features of a system.

1.2 Thesis Approach and Research Questions

This thesis work uses genetic synthesis of software architectures to introduce varying level of automation into design, planning and maintenance phases. The conducted studies use UML [36] based representation of software architecture. The work is based on the view that a software architecture is a result of a series of decisions [60] made to incorporate the concerns of different stakeholders in the system. The decisions translate into solutions inside software architectures where some solutions are specific to the system while others can be reused. In this work, all architectural changes resulting from architectural decisions are called architectural solutions or simply *solutions*.

Software architecture can be considered as a collection of architectural solutions. A solution in its entirety enters or leaves software architecture thereby affecting some property of the system. The literature contains a plethora of solutions (e.g., design patterns [11] [63], architectural styles [12]) to solve problems at the architectural level. In this thesis, the term solution will be interchangeably used for all such solutions including design patterns and architectural styles.

If we consider software architecture as a combination of solutions, then, designing a system can be understood as finding a feasible configuration of the solutions. Each solution not only resolves a functional requirement but also has an impact on one or more quality attributes [59]. The quality related implications of the well understood solutions are usually well documented and are known in advance. Thus, a careful selection of the solutions can be expected to lead to an acceptable architecture. This can be understood as a search problem: in principle, an algorithm can be designed which will find an optimal configuration of the solutions available in a solution database, given the architecturally significant requirements of the system.

In the design of the software architecture of a nontrivial system, the search space is huge. In this kind of problem, meta-heuristic search methods usually outperform deterministic approaches. Genetic algorithms [28] belong to the meta-heuristic search methods family and have been employed in many studies to address software engineering problems (e.g., [13], [14] and [15]).

In this thesis work, genetic algorithms have been employed to synthesize software architectures [P1]. The genetic algorithm synthesizes software architectures and applies solutions from a solution base thus producing improved designs. Each of the solutions is introduced through mutations employed in the genetic algorithm. The genetic algorithm is provided with an objective or fitness function to gauge the modifiability, efficiency and complexity of the architectures. Furthermore, tool (named Darwin [P2])

support for the genetic algorithm has also been provided. The tool enables an architect to fine tune the input parameters before and during the synthesis process as well as analyze the proposals from the genetic algorithm.

The genetic architectural synthesis has also been applied to work planning. The algorithm takes an input initial design and properties of the developing organization to produce initial work distribution plans. The solutions are introduced in the architecture to ease its distribution among the teams as well as to reduce the inter-team communication during the architecture development. The difficulties in communication among the involved teams are therefore taken into account in the fitness function. The difficulties usually have their origin in the cultural, lingual or social dissimilarities among the teams. Consequently, the genetic algorithm favors low coupling among the components to be assigned to teams with significant overhead in communication and vice versa. Furthermore, extreme over or under-loading of the teams are also discouraged by the fitness function.

To apply genetic synthesis for self-maintenance, a proof of concept infrastructure, named SAGA (Self-Architecting using Genetic Algorithms), has been developed exploiting genetic algorithms. The infrastructure enables a Java-based system to self-maintain its quality, more specifically, reliability and efficiency at run-time in response to changing environment or usage profiles. The unit of modification is an architectural solution. Each solution is composed of roles that are played by real artifacts in the implementation of a system. For properties of a system that cannot be self-maintained, the infrastructure provides tools to manually maintain such properties through injection and removal of solutions at run-time.

Current approaches rely on pre-planned strategies to address future modification needs (e.g., [7], [8], [37] and [38]). The strategies are embedded in the system's architecture and invoked as needed. It requires rigorous brainstorming to find out all possible future modification needs. Moreover, it will be hard to address a new unpredictable maintenance needs using the embedded strategies. Our genetic algorithms based maintenance is not planned around some specific modification needs; instead, one or more properties (e.g., efficiency, reliability etc.) are targeted for maintenance. The major pre-planning activities involve designing of the fitness function and selection of the solutions that have an influence on the maintained properties. Once equipped with the solutions and the fitness function, our genetic algorithm can address a wide range of future, possibly unpredictable, modification needs associated with the maintained properties.

The developed genetic algorithm includes a set of solutions with an effect on the performance and reliability in distributed systems settings. The fitness function measures efficiency and reliability of architectures. The infrastructure has been built on top of Javeleon [35] which allows run-time updating of Java classes. All the modifications, manual or automatic, are performed within the UML class diagram [36] based architectural representation of a running system.

The research questions studied in this thesis work are;

I. How to genetically synthesize software designs to automatically generate good quality architectures? Are the generated designs comparable to man-made designs? What kind of tool support is needed?

II. How to optimize work distribution plans along with software architectures using genetic algorithms for efficient distributed software development?

III. How to enable run-time maintenance using architectural solutions? What kind of infrastructure is required?

IV. How to apply genetic algorithms for enabling self-maintenance of non-functional properties associated with efficiency and reliability for software systems? What kind of infrastructure is required?

1.3 Research Overview

The research road map of this thesis work is shown in Figure 1.1. The horizontal axis shows the time line of the thesis work from fall 2009 till summer 2013. The vertical axis lists the major goals of the thesis work. The horizontal bars indicate the time period in which research has been conducted in the direction of the corresponding goal. The text on the bars shows the research questions explored and publications produced during that period.

The realization of the genetic algorithm [P1] addresses the first research question. The tool support for genetic synthesis of software architectures has been realized in the form of the Darwin environment [P2]. The research in the direction of work planning automation addresses the second research question, as reported in [P3]. As shown in Figure 1.1, a large fraction of the thesis work has been invested in automating maintenance. The work on the SAGA infrastructure explored the research questions III and IV. The findings were published in [P4] and [P5].

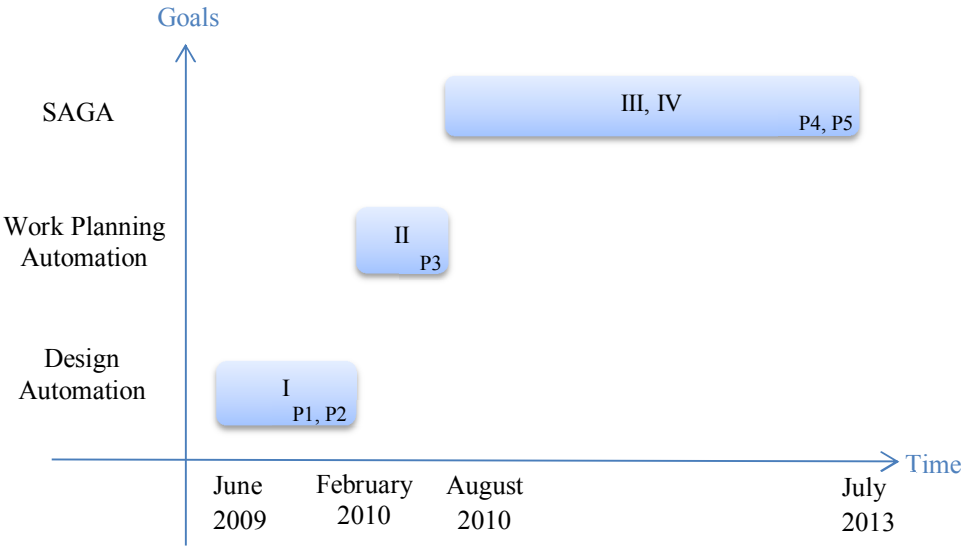


Figure 1.1 Research time line with goals and artifacts

1.4 Research Method

Design science [39] is about building and evaluating new artifacts or innovations that bring along some value to a community or users [40]. Hevner et al. [41] describes design science as "*It seeks to create innovations that define the ideas, practices, technical capabilities, and products through which the analysis, design, implementation, management, and use of information system can be effectively and efficiently accomplished*". It is a problem solving process, constructing artifacts providing solutions to new problems or more effective methods for already solved problems. The practitioners can leverage upon the capabilities of the artifact to build or extend systems efficiently. Some of the innovations are built upon well tested theories while others are commissioned to verify untested theories, models or concepts. Sometimes, artifacts are developed first for new areas lacking any established principles. It is the use of the artifact that then leads to new theories and concepts. These two aspects, theory and artifact, are linked to each other in most cases. Hevner et al. [41] have proposed the following guidelines for design science;

1. Design and develop an artifact to solve a problem. Identify the artifact's relevance to solving or re-solving a problem.
2. Use rigorous methods to evaluate the alternative solutions to choose the best one to be realized in the artifact.

3. Evaluate the artifact through mathematical, computational, empirical or qualitative methods.
4. Identify the major contribution (e.g., novelty, efficiency etc.) in solving the targeted problem.
5. Share the research with the world.

This thesis work studies the application of genetic architectural synthesis in software design, work planning, and run-time maintenance to enable automation. The artifacts realized were a genetic algorithm, Darwin tool environment and SAGA infrastructure. The author and the involved researchers had good experience with the Java-based technologies and therefore Java was selected as the main implementation technology. A widely used IDE for Java-based technologies called Eclipse [76] has also been employed.

The selection of genetic algorithms was made after a survey on the search methods [49]. Genetic algorithms avoid sticking to local optima which is the case with some other heuristic search methods. For the most part, the developed genetic algorithm has been created following the guidelines of Michalewicz [28]. Since Eclipse IDE's plugin based architecture allows the extension and customization of the environment, it has been used as the basis of the Darwin tool. The tool enables genetic synthesis of software architectures.

The genetic synthesis of software architectures has been evaluated through experiments as reported in [P1], [P2] and [P5]. Furthermore, the work (e.g., [24], [25], [26] and [27]) of fellow researchers has also evaluated the approach. Given the applicability of the genetic algorithm in producing good designs, the algorithm has been applied in work planning automation with integrated architectural design. The genetic synthesis of software designs along with work distribution plans has also been assessed through an experiment in [P3]. The work was published at international conferences and journals.

The application of genetic architectural synthesis in run-time maintenance has resulted in the SAGA infrastructure. Use of genetic algorithms was motivated by the fact that they do not require any pre-planning. SAGA was built upon the genetic algorithm and Darwin environment, therefore, the Eclipse IDE has served as the bedrock for the infrastructure as well. Furthermore, available open source third party components have been re-used in the process. The selection of these components was straightforward: the components are open source and widely used or officially developed or supported by the Eclipse's community.

The complete infrastructure and some of its parts have been individually evaluated through a set of experiments reported in [P4] and [P5]. The run-time maintenance abilities have been demonstrated through adaptive evolution scenarios of an example system in [P4]. An assessment of the self-maintenance side of the infrastructure has been presented using automated maintenance of efficiency and reliability of an example distributed system in [P5]. The results were shared with the scientific community at international conferences.

1.5 Thesis Contributions

The main contributions are

1. A genetic algorithm to synthesize software designs.
2. Tool support for genetic architectural synthesis.
3. A technique to work planning automation using genetic algorithms.
4. A technique for solution-based run-time maintenance.
5. An infrastructure enabling solution-based run-time maintenance.
6. A technique for solution-based self-maintenance using genetic algorithms.
7. An infrastructure for enabling solutions-based manual and self-adaptive run-time maintenance for Java applications.

These will be explained in more detail below.

(1) The genetic algorithm [P1] transforms a basic functional decomposition of a system, named as the null architecture, into a good quality architecture through application of a set of mutations and a crossover operation. The basic functional decomposition contains components and methods without any consideration for quality. Each mutation injects or removes a solution (here design pattern [11] or architectural style [12] only) to/from an architecture to improve its modifiability and efficiency or to reduce the complexity. A set of sub-fitness or objective functions evaluates the quality of the individual architectures during a simulated evolution. The best architecture of the last generation is the proposed architecture.

(2) The tool support has been realized as the Darwin [P2] environment. Darwin uses our genetic algorithm [P1] to enable genetic synthesis of software architectures. Darwin incorporates CASE tools [77] which can be used to realize the null architecture using UML [36] diagrams. Furthermore, the results of a simulated evolution can be studied in detail using various views in the Darwin environment.

(3) The work plan automation technique takes into account teams' configuration and inter-team differences for devising software architectures leading to efficient software development. Our genetic algorithm [P1] has been extended to include organizational structure and cultural and lingual differences among the teams in the fitness function. Consequently, the fitness function favors architectures with solutions conforming to the inter-team differences. Therefore, for any two teams with many differences, the genetic algorithm will favor architectures where fewer dependencies exist between them. Furthermore, coupling reducing solutions will be motivated between the components assigned to the teams with many differences. The reduced coupling between the components indirectly promises reduced communication among the developing teams. In addition, the genetic algorithm makes sure that the teams are not extremely over or under-loaded. This kind of multi-objective scenario involving consideration for architectural properties along with planning is truly challenging for a human architect.

(4) In the developed run-time maintenance technique [P4] a solution is viewed as a collection of roles. When a solution is used in a software system, the roles are played by the real artifacts within the system. The role playing artifacts need adaptation in order to behave in the way the applied solution dictates. Moreover, some roles partially depend on application logic which is truly challenging to automate. Thus, each role and therefore solution is unique requiring different level of adaptation efforts when introduced or removed at run-time. Consequently, the level of difficulty in making different solutions dynamic varies significantly from one solution to another.

(5) An infrastructure, named JITA (Just in Time Architecture)-plugin [P4], has been implemented to enable solution-based maintenance for Java applications. The solutions in focus were a small set of design patterns [11] including Adapter, Singleton and Observer. The infrastructure relies on Javeleon [35], a dynamic Java class updating facility. The infrastructure exploits the CASE tools in Darwin [P2] to allow for manual introduction and removal of the included solutions to/from the UML class diagram based representation of software architectures. The application independent parts of the solutions are generated by the infrastructure. However, the parts depending on application logic have to be manually written by the architect or developer of the system. The amount of manual work varies from one solution to another.

(6) At the heart of the self-maintenance technique is a decision making engine (a genetic algorithm) which incorporates architectural expertise. The decision making logic is driven by the maintenance goals. The goals that are

computationally measurable at the run-time can be self-maintained. The goals are continuously monitored at the run-time. A major deviation from the goals can be reduced through architectural changes by the decision making engine. Furthermore, architectural reflection is needed to reflect the changes in the architecture to the run-time.

(7) The implemented proof of concept infrastructure [P5], named SAGA (Self Architecting using Genetic Algorithms), enables self-maintenance for Java based distributed applications. The infrastructure supports manual maintenance of non self-maintainable properties, too. The infrastructure adapts a system to remain efficient and reliable in response to changing usage profiles through run-time injection and removal of architectural solutions. The infrastructure is composed of the Monitoring layer, Reflection layer, Architect (genetic) algorithm and Javeleon [35]. The Monitoring layer monitors the operations' execution times and usage frequencies as well as failures of the components. It reports the findings to the genetic algorithm to re-design the architecture, represented using a UML[36] class diagram. The genetic algorithm evaluates the architectures using efficiency and reliability sub-fitness functions formulated for distributed systems. The algorithm applies the Heartbeat [63] solution on failing and/or critical components to improve the reliability. The efficiency sub-fitness function motivates collection of highly inter-dependent components into the same node to reduce the number of slow remote interactions. The Reflection layer (or JITA plugin [P4]) is responsible for architectural reflection. The layer converts the proposals from the genetic algorithm into executable Java code. For the code reflection to the running system, the infrastructure uses Javeleon. As the new code is compiled, it becomes part of the running instance of the system and the cycle goes on.

1.6 Author Contributions

The idea that the genetic architectural synthesis can be integrated into standard UML based software design method was originated by the author. The Darwin tool's architecture and user interface have been designed by the author. The author and Sriharsha Vathsavayi have implemented the tool. Outi Räihä has integrated the genetic algorithm into the tool based on the author's design.

The author is the main contributor to the genetic algorithm application in the work planning study. The contributions include design and implementation of the new sub-fitness functions as well as mutations. The experiments were designed and carried out by the author. Also, the evaluation of the results was performed by the author himself.

The author is the main contributor to the technique and infrastructure enabling solution-based self-adaptive and manual run-time maintenance. The SAGA infrastructure was designed by the author. The author has formulated the concept of "solution" as a unit of modification in maintenance. The implications of solution-based maintenance were also studied and presented by the author. Furthermore, the author alone implemented the architectural reflection and run-time monitoring in the SAGA infrastructure. Moreover, the author has formulated and implemented the reliability sub-fitness function in the employed genetic algorithm. The author and Outi Räihä together designed and implemented the efficiency sub-fitness function. Outi Räihä has introduced the Heartbeat solution and related mutations to the genetic algorithm. Other mutations were realized by the author. The experiments were designed and conducted by the author, too. The results analysis and evaluations of SAGA are also work of the author.

1.7 Structure of the Thesis

The thesis is divided into four parts including this first introductory part. This part includes chapters 1 and 2. After this introductory chapter, Chapter 2 covers the background topics fundamental to this thesis work.

The second part is dedicated to the software development automation related studies performed under this thesis work. Chapter 3 explains the genetic synthesis of software architectures and details the empirical study on comparative analysis of manually and automatically designed architectures. Chapter 4 focuses on the developed Darwin tool. Chapter 5 explains the application of the genetic architectural synthesis for work planning automation.

The third part is focused on the thesis work related to run-time maintenance. The SAGA infrastructure is covered in details in Chapters 6. The last part concludes the thesis. Related work is presented in Chapter 7. An introduction to the included publications and author's contribution to each of the publication is provided in Chapter 8. Chapter 9 revisits the research questions. It also includes the limitations and future dimensions of this thesis work. The chapter finally ends with some concluding remarks. The included five publications [P1], [P2], [P3], [P4] and [P5] are provided in the appendix to the thesis.

2 Background

2.1 Software Architecture

The ISO/IEC/IEEE 42010 [58] standard defines a software architecture as "conception of a system- i.e., it is in the human mind". So, software architecture can be an imaginary concept nurturing in a human mind without any physical existence. However, it is useful to translate the concept into an artifact that can be presented to the people who have an interest in the system, like architects, developers, maintainers or users etc. That is an ***architectural description*** which describes a system in the real world. An architectural description not only defines the environment, system's elements and their relationships but also the rules and constraints the system and its ingredients must respect.

Software architectural description is not a mere consequence of the functionality expected of a system. Instead, many demands or ***concerns*** of the stakeholders as well as political or organizational constraints greatly contribute to shaping a system [58]. Software quality [59] is one such concern, encompassing many attributes like performance, complexity, maintainability, reliability, portability, security, usability etc. For example, a completely different architecture will emerge, if the system is to be portable across multiple platforms. Therefore, software architecture is enabler of system's quality.

Political and social concerns within a developing organization also have an influence on different aspects of software architecture. One such influence is reported in the form of Conway's law [57]. According to the Conway's law, software architectures usually resemble their developing organizations' structures. An organization with four teams will be biased towards dividing every system into four sub-systems or components. If one of the teams is experienced in databases, the organization may be inclined to add database into the systems it develops.

Software architecture is an entangled nest of concerns. In order to manage or extend such concerns in a controllable manner, different ***views*** are employed where each view contains one or more concerns. This makes it easier for us to focus on the selected concerns without getting lost in the concerns jungle. Furthermore, with each view is associated a ***viewpoint*** which provides the artifacts to realize the view [58].

2.2 Architectural Solutions

Software architecture can also be viewed as a result of ***decisions*** [60]. All the elements in software architecture are originated from decisions made during the designing phase. The decisions translate into ***architectural solutions*** that may appear as classes, components, sub-systems, relationships, events, or flows in the different views. Other decisions may just alter some property of a component or system. For example, in distributed systems, assignment of a component to a node is a solution, too. These different solutions collectively contribute to the goal(s) set for the target system.

Designing architectural solutions requires experience and creativity. Therefore, over the last few decades recurring architectural solutions have been identified and documented in the form of ***design patterns*** or ***architectural styles***. Gamma et al. in [11] created a comprehensive catalogue of design patterns for object oriented systems. Shaw et al. [12] reported a set of reusable solutions as architectural styles. Other catalogues of patterns/solutions have been presented, for example for enterprise [62] and fault tolerant [63] systems.

A pattern offers solutions to a problem as well as influences the quality of the system. From the patterns documented in the literature, Observer, Strategy, Template Method, Adapter, Façade, Mediator, Singleton [11], Message Dispatcher, Client-Server [12] and Heartbeat [63] have been used in this thesis work. All these patterns except Heartbeat are designed for improved modifiability.

The ***Observer*** pattern suggests an extendable approach to events sharing problem. An Observer can observe a Subject for a particular event. It is responsibility of a Subject to inform its observers about the occurrence of the event. The pattern introduces flexibility into the design as any number of observers can be added or removed at any time for a Subject.

The ***Strategy*** pattern enables a system to employ a set of implementations (strategies) for an algorithm, suitable for different situations. Any number of strategies can be included or excluded to/from the system in the future thus resulting in a flexible architecture. The ***Template Method*** pattern is used when some fractions of an algorithm

is left for sub classes to specialize. A Template Method holds the fixed part while leaving the variant part to the classes extending the Template.

Adapter resolves issue of interface incompatibility. When a component's (Adaptee) interface changes, the clients of the component will not be able use it. The Adapter pattern introduces a new Adapter component between the clients and the component that provides a compatible interface to the clients. It improves modifiability as any future variations in the interface of the component (Adaptee) will only require updating of the Adapter component without affecting any other part of the architecture.

The ***Façade*** pattern provides a single interface to access the functionality offered by a group of components. A client may use multiple components from the group to perform a single task. Such tasks can be moved to the Façade component thus relieving the client from dependence on many components inside the group. This will result in reduced number of dependencies or connections among the components thus reducing the coupling and improving the modifiability. When components are not be able to interact directly, the ***Mediator*** pattern is used to resolve such situation. A mediator component implements mediation logic required to enable the interaction among the components.

The ***Singleton*** pattern guarantees creation of a single instance for a Singleton component. The pattern allows more control over the instantiation of a component. Thus instantiation strategy (singleton or non-singleton) can be altered at any time without disturbing the clients of the affected component.

The ***Client-Server*** pattern eases resource sharing over the network. Servers host shared resources required by the Clients. Client and Server components usually exist in separate environments, even sometime on different machines. A Server waits for requests from its Clients, processes them and replies with the results or data. As long as the interface is respected, any Client built using any technology can access the resources hosted at the Server.

The ***Message Dispatcher*** pattern enables different components, built using different technologies to communicate through messages. A unified messaging protocol is to be respected by all participating components. The pattern hides information as internals of a component are not known to another component. The components are only aware of the messaging interface to request actions from other components. The flexibility in the architecture comes at the cost of increased complexity and reduced performance. Messaging is slow compared to direct calls. The performance may suffer due to the messages passing through the network, and to the need of composing and interpreting messages. The higher the number of messages, the more performance will suffer. The

messages may cause network congestion thus slowing down the whole system. The complexity of the architecture also increases as messaging infrastructure and interfaces have to be incorporated.

The ***Heartbeat*** pattern targets systems reliability. In a system, components with Heartbeat will periodically send beat messages to their client components. When a component with Heartbeat fails, its beating will stop as well and the clients will immediately notice its departure from the system. The result is an increased reliability as the whole system will not collapse due to the failure of one or more components. The failed component may be restarted or other complex strategies can be installed in the system to handle such situations. The solution is also useful in distributed systems. In a distributed system, components are located on different nodes communicating through a protocol (e.g., by messaging). If a remote component dies, the clients of the dead component on other nodes may keep on waiting for the dead component's response. This results in useless waiting time and undesirable system behavior which can be avoided with the Heartbeat solution. However, at the same time, the solution consumes some computational resources as the beats have to be created, sent, received, and processed. In addition, it increases the number of messages that might cause congestion in the network. The network congestion can lead to slowing of the whole system [64].

2.3 Unified Modeling Language (UML)

UML [36] is a system modeling language containing various views with own notations, techniques or viewpoints. The views are referred to as diagrams and divided into two groups, behavioral and structural diagrams. The behavioral diagrams show what the system actually does or how it reacts to external stimuli. The activity, use case, interaction and state diagrams are regarded as behavioral diagrams. The structural diagrams on the other hand show the structural elements of a system like classes, packages, components, sub-systems etc. and their relationships. Examples of structural diagrams are class, component and composite structure diagrams. In this thesis work, the use case diagrams have been used to specify the requirements of a system. The requirements are then transformed into rudimentary (null) architecture presented as a class diagram (details later). The class diagram has also been used to present the architectures produced by our genetic algorithm. This section details the diagrams and their design elements important for understanding of the remainder of the thesis. For the full specification of UML, see [36].

A use case diagram captures all possible interactions between the system and its external ***actors***, as shown in Figure 2.1. A ***use case*** defines one possible interaction with the system, represented using an elliptical shape. An actor, represented using a

stick figure, can be a person (e.g., user), an organization or another system interacting with the system. As shown in Figure 2.1, each actor is connected to the use cases that are available for it, him or her. For example, the Shutdown use case is only available to the Admin of the system. However, the Authenticate use case is available to both actors (Admin and User). Moreover, a use case may depend on another use case. Since authentication will require information about the registered users, e.g., their user ids and passwords, the Authenticate use case has a dependency on the Users Database use case.

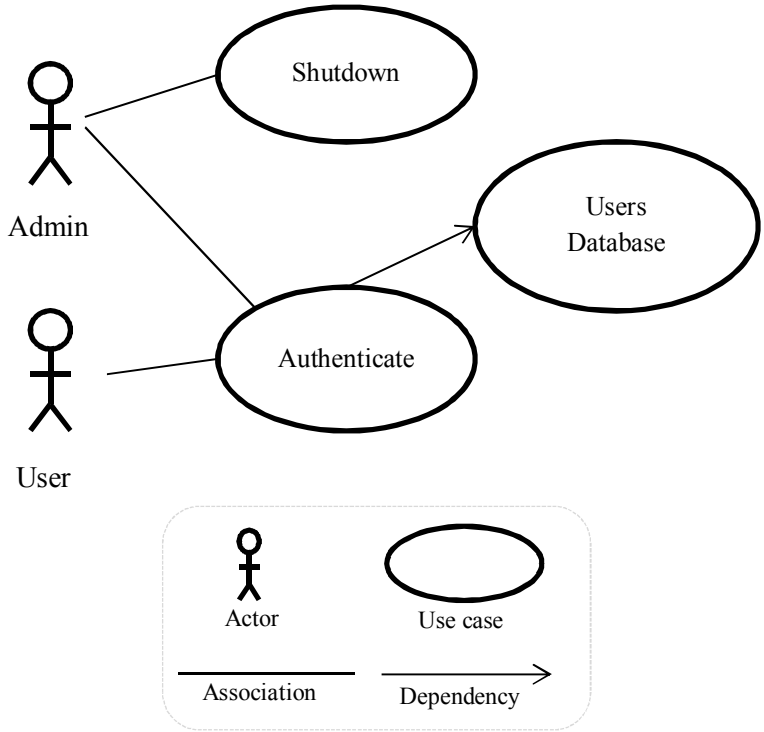


Figure 2.1 UML use case diagram notations

The ***class diagram*** view is widely used to model systems developed in a programming language belonging to the Object Oriented Programming (OOP) paradigm. Due to the strong correspondence between the class diagram notations and OOP paradigm, a class diagram of a system can be comparatively easily transformed into structural code in languages like C++, Smalltalk or Java. As shown in Figure 2.2, a class diagram visualizes static object oriented view of a system in the form of classes and their relationships**.** A ***class*** serves as a specification of objects by defining their features, constraints and semantics [36]. Its members are ***attributes*** and ***operations.*** The class members can have different ***visibility*** like ***public (+)***, ***private (-)*** or ***protected (#)***.

A public member is accessible to other classes while private is not. A protected member is accessible only to the child classes through an ***inheritance*** relationship (explained in the next paragraph).

A set of ***relationships*** is also available to present relationships among the classes of a system. A class can inherit properties of another class through the inheritance relationship. As shown in Figure 2.2, the Security class has access to encryption and decryption features provided by the Encryption class through the inheritance relationship. A class may need public features of another class for its functionality. This kind of relationship can be represented using a ***dependency***. For example, in Figure 2.2, the Security class needs to access the database of users and therefore depends on the Database Access class.

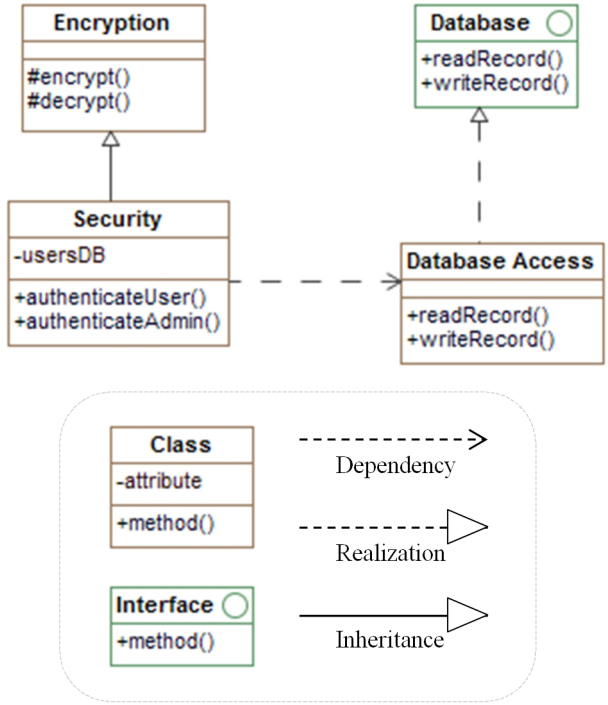


Figure 2.2 UML class diagram notations

In context of this thesis work, another important element of UML class diagram is the ***interface***. An interface specifies a set of features (methods or operations) that the class(es) realizing the interface must provide. For example, in Figure 2.2, the Database Access class is realizing the Database interface. The Database interface has two

methods (readRecord and writeRecord) which the Database Access is realizing. Note that interfaces are non-instantiable entities, only the classes that are realizing them can be instantiated.

UML profiling offers a mechanism through which standard UML elements can be extended for different domains. A profile contains stereotypes that can be applied to classes, interfaces, methods, dependencies etc. Typically, graphical representation of a stereotype application is denoted as <<stereotype>>. Also, it is possible to associate a set of attributes with a stereotype. The stereotype attributes can be used to annotate standard UML design elements with additional information. For example, in this thesis work, each operation in the null architecture is annotated with its call frequency, parameters size and variability (details in Chapter 3) using a stereotype.

The solutions in the proposed architectures are also represented using stereotypes. For each employed solution, a stereotype has been included in the UML profile. In SAGA, class diagrams of newly proposed architectures are transformed into Java. SAGA identifies the solutions through their stereotypes and generates the behavioral code necessary for their implementation. However, complex application specific behaviors are too challenging to generate automatically and therefore will require manual input. For example, in SAGA, a Heartbeat stereotype has been implemented to represent the Heartbeat solution [63] in the class diagrams. When the Heartbeat stereotype is applied to a class, it implies that the class holds all the behaviors associated with the Heartbeat solution. That is, it must transmit beats and allow other classes to listen to the beats. The code generation utility in SAGA generates such recurring Heartbeat behaviors for the classes with the Heartbeat stereotype applications.

2.4 Genetic Algorithms

Genetic algorithms [65][28] are inspired from the Darwinian or natural evolutionary process. In the natural evolutionary process, one creature evolves from another and develops traits that help it live in the changing environment. The focal point of such changes is the DNA (or ***chromosome***) of the living beings. The processes that are the driving forces behind the modifications are ***mutations*** and ***reproduction*** (or ***crossover***). A mutation alters a characteristic (or ***gene***) located at specific location or ***locus***. All possible variations for a gene are called ***alleles***.

The reproduction helps in growing the ***population*** which is essential for survival of a species. Furthermore, during reproduction, better individuals may be produced combining the good qualities of their parents. It is natural that healthy individuals will survive longer and may contribute to the new population through reproduction compared to the poor individuals. This process is termed as the ***natural selection*** which

will eventually lead to healthier future ***generations***. In other words, from one generation to another, it is very likely that the individuals will possess DNAs suitable for the changing environment.

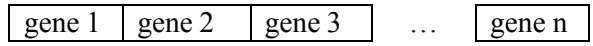


Figure 2.3 A chromosome as a stream of genes

In computer science, genetic algorithms come under the category of ***meta-heuristic search methods*** which are targeted for solving the problems with large search spaces that will take a considerably long time to resolved using deterministic methods. Furthermore, in the case of NP-hard problems which lack exact algorithms, genetic algorithms can be helpful in finding good enough solutions. From the natural evolution, the ideas of chromosome, mutation, crossover, population, generation, selection and fitness have been adopted in genetic algorithms [28]. To explore a solution space, the solutions have to be brought into genetic algorithms based representation. The solutions have to be encoded in the form of a chromosome composed of genes, as shown in Figure 2.3. A gene represents a characteristic of the enclosing solution. The encoding usually differs from one problem to another.

Components →	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Chromosome →	2	1	3	4	3	2	1	4	1	3

Figure 2.4 A solution as chromosome

Consider for example the problem of clustering or organizing components into sub-systems. Let say there are four sub-systems (1, 2, 3, 4) and ten components (C1-C10). A solution to the problem will propose sub-systems for the components. There can be multiple such solutions with different organizations of the components based on their properties or inter-dependencies. For example, it is sensible to place two interdependent components in the same sub-system for better organization. A genetic representation of one such solution is shown in Figure 2.4. In the chromosome, each gene is associated with a component and contains information on its host sub-system. The allele in this case is the set of sub-systems (1, 2, 3, 4). For example, the component C1 has been assigned to the sub-system 2 while component C10 has been allocated to the sub-system 3.

The flow of a typical genetic algorithm is shown in Figure 2.5. A genetic algorithm forms the initial population using the seed solution by applying random mutations on

the solution. Other methods may also be exercised like manually creating the initial population or using an intelligent algorithm to do the job.

In every generation, to produce presumably healthy individuals, the member chromosomes undergo a series of mutations and crossover operations. The fitness of each of the chromosomes in a generation is calculated. Each subsequent generation is formed by selecting the individuals from the previous generation according to the chosen selection strategy. In some implementations, a genetic algorithm is left for execution for a pre-defined time period. At the end, the best solution in the last generation is assumed to be the proposed or candidate solution. The number of generations can also be defined in a genetic algorithm.

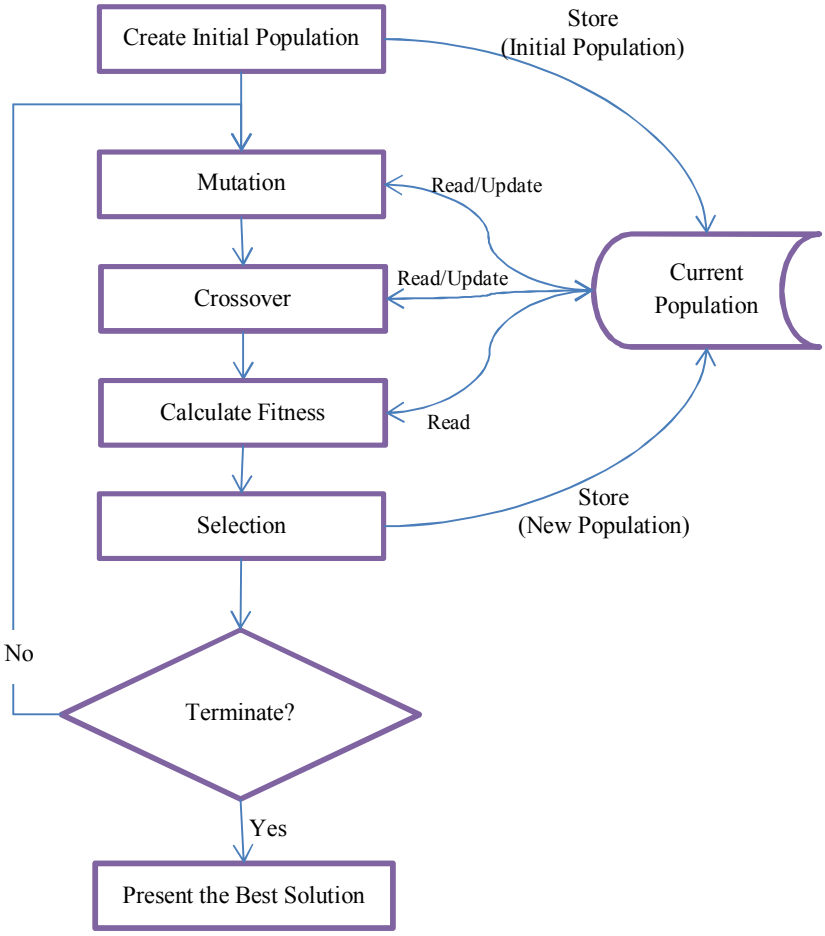


Figure 2.5 Genetic algorithm flow chart

A mutation alters the characteristic (or gene) by randomly shifting its value to another variation available in the alleles. The gene is usually selected randomly. In complex problems, a set of mutations is employed where each mutation is targeted for a particular gene type. The mutations are selected randomly; however, ***mutation rate or probabilities*** can be used to influence the selection. The higher is the mutation rate for a mutation, the greater is chance of its application during an evolution. For the discussed clustering problem, a mutation which will randomly assign a sub-system to a component will be required. When the mutation is applied to a randomly selected gene, it will randomly choose another sub-system from the set of sub-systems (alleles) and write its number to the gene. As a consequence, the component associated with the mutated gene has now been allocated to a different sub-system. As shown in Figure 2.6, the gene associated with C4 has been mutated and it has been assigned to the sub-system 1 now.

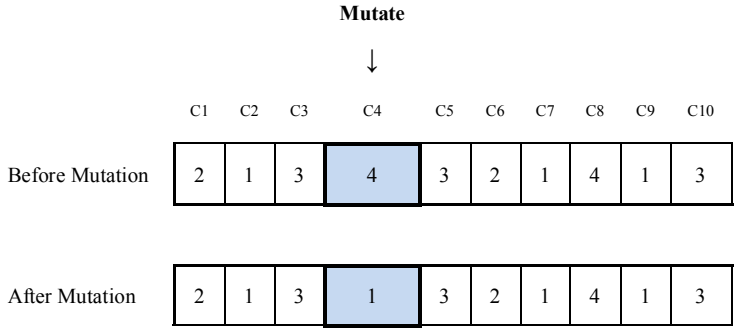


Figure 2.6 Mutation operation

During a crossover operation, typically two random individuals are chosen from the population. In a single point crossover, both parents are broken into two halves at the ***crossover point*** which is also randomly selected. The initial half comes from the first parent while the rear part from the second parent. Both are merged to create the offspring thus inheriting qualities from its parent chromosomes. In multi-point crossover, parent chromosomes are fragmented in more than two halves. The offspring is formed using multiple fragments from both of the parent chromosomes. An example of the single point crossover is shown in Figure 2.7 for the example clustering problem. The crossover point has landed between the genes associated with the components C5 and C6. Therefore, the child chromosome has been formed using genes associated with C1 to C5 from the first parent while the remaining genes, C6 to C10, came from the second parent.

The health of each individual is measured using an objective or fitness function. A fitness function is actually a mathematical formula which translates properties of a

solution into a numerical number. A fitness function can be a collection of further sub-fitness functions where each sub-fitness function measures a particular sub-property of the solutions. For the clustering example, the health or suitability of a solution could be related to coupling and cohesion. One way to indirectly approximate these qualities is to calculate the number of local inter-component dependencies. A local dependency is the one whose client and supplier both exist on the same sub-system. Therefore, a solution with a high number of localized inter-component dependencies could imply more cohesive sub-systems and less coupling among the sub-systems. The fitness function is shown in equation (2.1) which counts the local dependencies. As can be seen, the fitness function just estimates the quality of solution. This is the trickiest and the most important part in genetic algorithm design. A poorly designed fitness function will overshadow the promised benefits of genetic algorithms.

$$Maximize\ f(solution) = The\ number\ of\ local\ dependencies \qquad 2.1$$

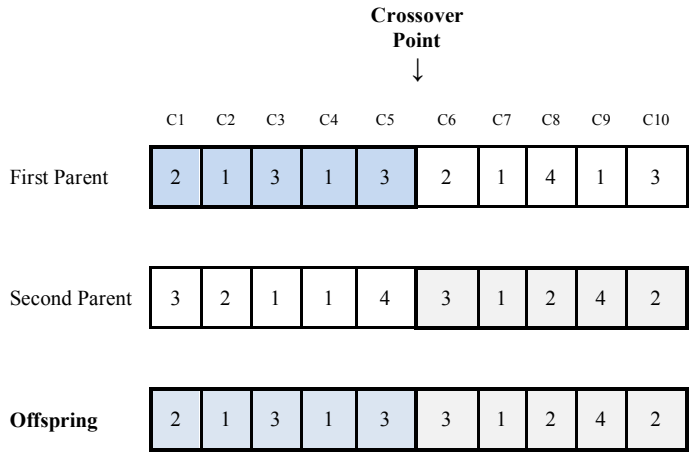


Figure 2.7 Single point crossover operation

The selection process will select individuals to form new generations during an evolution. The selection can be done in many ways. The simplest method is ***elitist selection***, i.e., to choose the best individuals of the previous generation, however, there is a threat associated with this method. It might converge the evolution to the local optima. Another option is to use the ***roulette wheel*** method where each individual is assigned a portion of the wheel based on its fitness value. The higher is the fitness of an individual, the larger will be its portion and therefore greater will be its chance to move on to the next generation. The method lends greater survival opportunity to the healthy individuals while at the same time keeps the door open for the poor individuals to

revive in the upcoming generations. This results in more diverse population and reduces the probability of ending up at the local optima.

2.5 Distributed Software Development

In distributed software development, teams located in different sites collaborate to develop a software system. The aim is to exploit the intellectual and materialistic resources in various parts of the world to develop high quality software systems with low cost. The practice has been extensively in use in other established industries and thus adopted by the software industry [42]. The physical distance among the teams can range from few to many thousands of kilometers. The developing organization can exercise effective access to the comparative inexpensive expertise available in different locations. When teams are located in different countries, round the clock development can be executed due to the differences in time zones. Also, the product can be more effectively localized for the regions where the teams are located [43].

Distributed software development has an influence on software architecture as recognized by the ***Conway's law*** [57]. The act of organizing people into teams is in fact an architectural decision in itself which will be directly reflected in the system's architecture. It is likely that the system will contain as many sub-systems as the number of teams. Furthermore, software architecture is also influenced by the time, budget, politics and personal motivations of the involved people. Moreover, the distribution may be based on the quality of communication channels available among the teams. High dependency among components typically leads to high communication among their developing teams. Therefore, it is sensible to assign such components to teams with low communication resistance. Furthermore, the very act of distribution limits the freedom of the teams in implementing their own share of the system. Each team has to make sure that their design or implementation choices do not jeopardize the system as a whole. Moreover, due to the different cultures and norms of the involved teams, there is always a risk that they might end up developing components that may not be able to interact [29].

Increasing the number of teams or sizes of the individual teams does not imply increased productivity [61]. The productivity may improve if there is no communication involved, for example, adding work force for picking fruits. However, in software development the increased communication usually overshadows the benefits of increased work force. The communication overhead is in fact recognized as the most troublesome aspect in distributed software development [30][43][44]. In direct interaction situations, lots of information is shared through informal means (gestures,

facial expression or talk over coffee table etc.) [43]. Thus, everything need not to be explicitly written down or mentioned.

Unfortunately, the communication enabling tools always lack capability to transmit such information effectively within the messages. Even if all the information is explicitly shared with all the teams there is always a threat that each team might interpret a message in different ways in their cultural context [43]. This can lead to issues in knowledge sharing, source code management and tasks distribution, coordination and synchronization [44]. The outcome could be a delayed and/or low quality product.

The way out of the situation is to adopt practices leading to reduced but effective inter-teams communication. Furthermore, the employment of more effective communication technologies, tasks and issues tracking tools and proper training of the personnel can dramatically improve the situation. Moreover, frequent internal deliveries can help identify and fix the inconsistencies at early stages during software development [44].

2.6 Run-time Software Maintenance

Software does not wear out, however, the changes in underlying hardware, environment or requirements may indirectly reduce its value [45]. Therefore, software requires maintenance to adapt to the changed context or requirements and to prevent or fix faults [33][46]. A typical maintenance cycle is shown in Figure 2.8. The developing organizations or users monitor the software for errors, quality degradation or missing features. Then, a maintenance plan is drawn, identifying the required personnel, methods, tools and financial resources. Next, the software is re-designed, coded and tested. The changes are reviewed and after that the product is deployed and monitored again.

The manual maintenance practices are expensive. It is estimated that maintenance can consume between 40 to 75% of the whole cost of a software system [31]. Furthermore, during maintenance, a software system can be in shutdown state or may offer limited functionality which is costly for systems with continuous operation demand. Run-time maintenance provides some relief by enabling run-time update of software systems. However, the involved manual planning, design and development can still be costly unless there are tools which can assist in those stages, too.

The ***Self-adaptive*** [47] paradigm goes one step further by eliminating manual maintenance completely. A self-adaptive system has been described as "*Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation*

indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible." [47]. The literature contains research works dealing with dynamic reconfiguration [3], self-adaptive [7], self-managing [68], self-tuning [100], self-healing [68] and self-architecting systems [48]. As obvious from their names, these systems are designed to perform reconfiguration, management, tuning, healing or re-architecting without or with minimum human intervention. They are collectively known as Self-* systems.

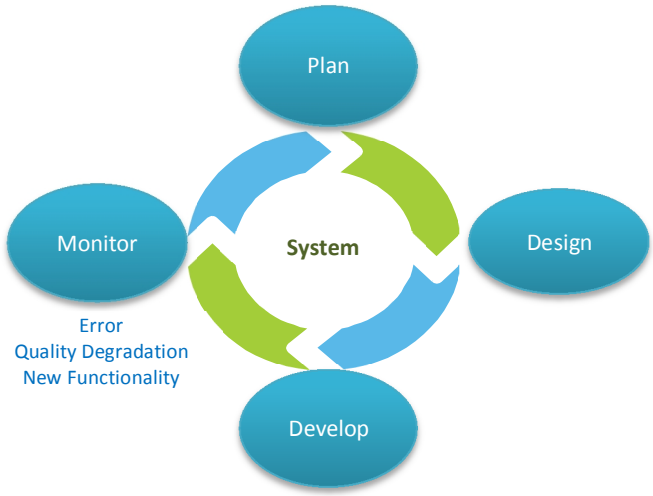


Figure 2.8 Software Maintenance

The self-adaptive paradigm requires automated monitoring mechanisms that can evaluate the targeted properties at run-time without consuming too much resources. Furthermore, automated decision making agents are required to plan, design and develop the system in response to the monitored deviations or needs. Also, the software itself has to be dynamically modifiable so that components can be moved in and out of the system or their configuration can be changed at run-time. A self-adaptive system can be open or close [37]. A close self-adaptive system relies on a pre-defined set of strategies to fulfil the maintenance needs. In contrast, an open self-adaptive system needs no pre-planning. The system can understand the situation on hand and then can act accordingly through a decision making mechanism.

PART II – Applying Genetic Software Architectural Synthesis in Software Development

The studies in this part use genetic algorithms to generate software designs and work distribution plans.

3 Genetic Synthesis of Software Architectures

3.1 Introduction

The brain behind all of the automation attempts in this work is a genetic algorithm [P1] adjusted for software architectural synthesis. Our genetic algorithm was originally envisioned by Räihä et al. in [24]. Assuming software architecture as a collection of solutions (e.g., design patterns [11], architectural styles [12] and other architectural solutions), the problem of software design can be seen as a search problem. Note that our genetic algorithm operates at the design level and the fitness function is concerned about the quality of the architectures. The search space is composed of architectures with different architectural solutions and therefore different qualities. Our genetic algorithm is a multi-objective genetic algorithm that explores the search space in quest to find the architecture possessing balanced values for the targeted quality attributes like modifiability, complexity and efficiency as defined by the fitness function.

3.2 The Genetic Algorithm

As shown in Figure 3.1, the genetic algorithm proposes an improved architecture given a basic functional decomposition of a system called ***null architecture*** in the UML notations. The null architecture embodies the functional obligations put upon the system without any consideration for quality [59] like modifiability, efficiency, complexity etc. Note that the genetic algorithm is not taking care of the functional requirements and the null architecture; instead, the user or architect is expected to handle these steps manually.

The quality requirements used with the genetic algorithm are generic in nature. For example, high modifiability, efficiency and simplicity are desirable for almost all kinds of systems. The requirements are designed so that they can be satisfied through architectural features independent of the semantics of the system. The different

architectures in the search space will have a variety of architectural features or solutions and therefore some of them will satisfy the quality requirements better than others. Furthermore, the generic nature of the quality requirements makes it possible to reuse them with a wide range of systems. Therefore, once such quality requirements are embedded as the fitness function in the genetic algorithm, they can be reused for many systems with different functionalities. The formulation of the fitness function and therefore the results will vary with the changes in the quality requirements.

Once the inputs are given, the genetic algorithm evolves the null architecture using a set of mutations and a crossover operation. The mutations add or remove solutions (design patterns [11], architectural styles [12] and other architectural solutions) from the Solutions Base to/from the architectures during the evolutionary process. The best architecture of the last generation is the proposed architecture.

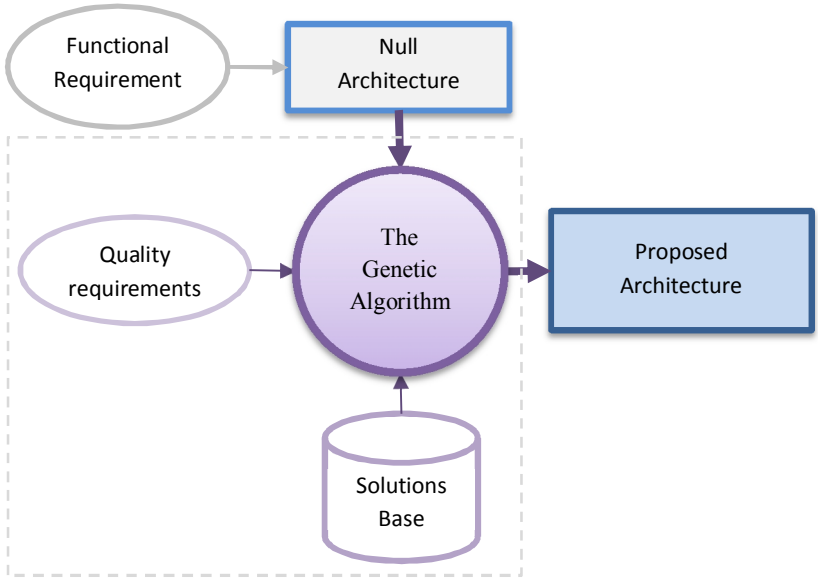


Figure 3.1 Genetic software architecture synthesis process

As discussed in the author's work in [P4], every solution is unique and has different kind and scale of impact on the system's structure and/or behavior. Some may require a complete rethinking of the system (e.g., Flyweight [11]) while others introduce simple call indirections (e.g., Proxy [11]). Thus, the impact of each solution on the system's structure and behavior has to be evaluated separately. In this thesis work, the employed modifiability improving solutions mostly changes the structure and involves behaviors that do not interfere with the functionality of the system. The work reported in this

chapter employs modifiability solutions at the design level where only their structural impact can be observed.

The author's work in [P4] has demonstrated the realization of behavioral impact (code) of three modifiability solutions Adapter, Observer and Singleton [11]. The results of the study were used in SAGA, as reported in Chapter 6. Moreover, the behavior of reliability and efficiency improving solutions employed in SAGA do not interfere with the system's functionality. In this thesis work, the behavior introduced by the quality improving solutions co-exists in harmony with the functional behavior. Therefore, the system's functionality remains the same regardless of what quality solutions have been introduced or removed to/from the system by the genetic algorithm.

3.2.1 The Null Architecture

The null architecture encompasses classes and operations without any other quality affecting solutions. Furthermore, the operations in the classes are annotated with additional support data like call frequency, parameters size and variability. Some of the support data can be precisely defined while others have to be estimated. The call frequency parameter approximates how frequently an operation is used. The parameters size provides a count of the number of parameters. The variation parameter indicates the possibility of variation in the behavior of an operation in future. In addition to the architecture, the fitness function will be relying on the support data for correct estimation of the quality of the architecture.

3.2.2 Encoding as Chromosome

The null architecture is encoded in a chromosome at the beginning of the genetic synthesis, as shown in Figure 3.2. The chromosome is a stream of supergenes where each supergene represents an operation in the input architecture. If there are n operations in an architecture there will be the same amount of supergenes in the chromosome. The support data is also stored in the genes along with the operations. Table 3.1 lists all the parameters (including the support data) that are stored in a supergene. The goal behind storing all the information is to be able to re-construct a UML class diagram of the architecture at later stages.

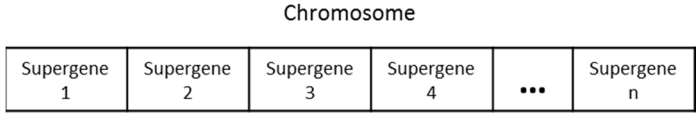


Figure 3.2 Chromosome as collection of supergenes

A supergene includes the position of the operation in the architecture by capturing information about its host class, interface and node. Moreover, relationships with other operations are stored in the form of a list of calling operations which may be located in the same or different classes. The information about the used message dispatcher and operations accessed through it is also recorded for all operations.

Table 3.1 List of parameters in a supergene

Parameter	Purpose
Name	Name of the operation.
Type	Operation or attribute.
Call Frequency	Estimated number of times the operation could be invoked.
Parameter Size	The number of parameters of the operation
Variations	A numeric estimate of the proneness of the operation to change (variability)
Class	The class hosting the operation.
Interface	The interface class (if any) the operation is part of.
Node	In distributed systems settings, the node this operation and its hosting class and interface is residing in.
Calling Operations	List of other operations that are invoking this gene's operation.
Dispatcher	The message dispatcher the operation is using for communication with other operations.
Dispatcher Communications	List of the operations accessed through the message dispatcher. In case of distributed systems, list of the remote operations the operation depends on.
Pattern/Solution	The solution(s) this operation is playing a part in.
Team	The team in the organization that will develop this operation.

A supergene also contains information on the solutions or patterns the associated operation is part of. A solution instance holds data on the existing or additionally required classes and interfaces, and operations participating in it. For the purpose of work planning automation, for every operation information about the developing team is also recorded in their supergenes. In order to create initial population the chromosome is copied multiple times and randomly mutated to introduce diversity into the first population.

3.2.3 Mutations and Crossover

The mutations are administered at the operation (or supergene) level; however, some may affect the hosting component as a whole (discussed later). A mutation only adds or removes some information to/from a supergene(s). First, a supergene is randomly selected. Then, a mutation or crossover is chosen through the roulette-wheel method. The mutations and the crossover operation are assigned a share of the wheel

based on their probabilities. The higher is the probability, the bigger will be the share of the wheel. A large section of the wheel implies significant likelihood of application of the respective mutation or crossover during a simulated evolution.

The list of mutations is given in Table 3.2. For the most part, the mutations in the genetic algorithm concerns introduction and removal of architectural solutions to/from software architectures. For example, the *add Adapter mutation* will introduce an adapter solution into the architecture while the *remove Adapter mutation* will do the opposite. The basic set of mutations add and remove the Strategy, Template, Façade, Mediator, Message Dispatcher and Client-Server patterns. Furthermore, introduction/removal of an interface to/from a component has also been realized as a mutation. A null-mutation, doing nothing, is also included so that the sum of mutation and crossover probabilities is 100.

Table 3.2 List of Mutations

Mutations	Description
Add/Remove Adapter	Adds or Removes the solution
Add/Remove Strategy	
Add/Remove Template	
Add/Remove Façade	
Add/Remove Mediator	
Add/Remove Message Dispatcher	
Add/Remove Client-Server	
Add/Remove Interface	
Null Mutation	Does nothing

The crossover is the single point crossover [28]. The point is selected randomly and a new offspring is formed by merging the halves from the parents. After a mutation or crossover, the resultant chromosome is checked for inconsistencies. Some of the inconsistencies may be repaired while for others the whole chromosome is to be thrown away.

3.2.4 Fitness Function

The fitness function measures the quality of the individual architectures. The size of population and the number of generations in an evolution can also be defined. A fitness graph can be drawn using the fitness values of the elite or the best architecture of the generations. At the end of an evolution, the best architecture of the last generation is the proposed architecture.

The fitness function (f) measures efficiency, modifiability and complexity quality of a software architecture (a), as shown in equation (3.1). A sub-fitness function can be either positive (award) or negative (penalty) depending on its implementation. A weight is associated with each sub-fitness function which acts as a balancing coefficient as well as can be used to emphasize importance of a quality attribute. Setting a sub-fitness weight to zero will result in its exclusion from the genetic synthesis process. Similarly, setting the weights to 1 will result in equal preference for all the quality attributes.

$$f(a) = w_e sf_e + w_m sf_m - w_c sf_c \quad 3.1$$

where sf_e, sf_m and sf_c are efficiency, modifiability and complexity sub-fitness functions, respectively. The multipliers (w) are weights of the sub-fitness functions.

In the included publication [P1], the sub-fitness functions were inspired from the work of Chidamber and Kremerer [66]. The sub-fitness functions efficiency is shown in the equations (3.2) which is further composed of positive (sf_{pe}) and negative (sf_{ne}) efficiency functions. The sf_{pe} formula rewards highly cohesive classes. Highly cohesive classes mean many invocations of operations will be within classes. These kinds of invocations are cheap from efficiency perspective. Moreover, the sub-fitness sf_{pe} promotes the situations where multiple operations of one class are invoked by another class or vice versa. Since remote calls are slow, the negative efficiency sub-fitness (sf_{ne}) penalizes the calls made through the message dispatcher (Message Dispatcher solution) and server (Client-Server solution). The frequency of the calls further amplifies the penalty. The negative efficiency sub-fitness (sf_{ne}) also includes class instabilities as the function $f_{instability}$ [52]. It promotes layered class structure and penalizes cyclic dependencies.

$$sf_e = sf_{pe} - sf_{ne} \quad 3.2$$

where $sf_{pe}$and sf_{ne} measures positive and negative efficiency values and are formulated as:

$$sf_{pe} = \sum(|\text{operations dependent of each other within same class}| * \text{parameterSize}) + \sum(|\text{usedOperations in same class}| * \text{parameterSize} + |\text{dependingOperations in same class}| * \text{parameterSize})$$

$$sf_{ne} = \sum(f_{instability}(k)) + (|\text{dispatcherCalls}| + |\text{serverCalls}|) * \sum\text{frequencies}$$

where $f_{instability}(k) = \frac{\text{the number of classes used by the class } k}{\text{the number of classes using the class } k + \text{the number of classes used by the class } k}$

The modifiability sub-fitness function (sf_m) is given in equation (3.3). It is further composed of positive (sf_{pm}) and negative (sf_{nm}) modifiability sub-fitness functions. Use of interfaces lends us flexibility to change the implementation without disturbing the rest of the system. Therefore, their use is rewarded by sf_{pm}. Furthermore, the sub-fitness sf_{pm} promotes the indirect invocation of operations through the message dispatcher due to the similar modifiability benefits. Components use messages to communicate with each other through the message dispatcher. They are loosely bound to each other and therefore their implementation can be changed independently.

Since high variability implies high tendency to change, the reward of using the message dispatcher is therefore multiplied by the variability parameter. Also, calls through servers are promoted by sf_{pm}. The multiplier α (α>1) in sf_{pm} heavily penalizes the architectures containing interfaces with unused operations. The value of α is found after some experimentation which may vary from one kind of systems to another. Since direct calls can create strong coupling among the operations and their hosting classes, the negative modifiability sub-fitness function (sf_{nm}) penalizes such calls.

$$sf_m = sf_{pm} - sf_{nm} \tag{3.3}$$

where $sf_{pm}$and sf_{nm} measures positive and negative modifiability values and are given as:

sf_{pm} = |interface implementors| + |calls to interfaces| + |calls to server| +|calls through dispatcher| $*$ $\prod$ (variabilities of operations called through dispatcher) – |unused operations in interfaces| $*$ α

sf_{nm} = |calls between operations in different classes, that do not happen through a pattern|

The complexity sub-fitness function (sf_c) is shown in equation (3.4). The function sf_c is a line of defense against heavy fragmentation of classes. In most cases, as a result of mutations, the number of classes and interfaces will grow. For example, when the Add Adapter mutation is applied on an operation (as supergene), it breaks away from its host class. Then, a new host class with an interface (or the Adapter solution) is added to the architecture that hosts the operation. The higher the fragmentation is, the higher the penalty will be. As shown in equation (3.4), the complexity sub-fitness function (sf_c) counts the number of classes and interfaces in an architecture.

$$sf_c = |\text{classes}| + |\text{interfaces}| \tag{3.4}$$

3.2.5 Selection

At the end of each generation, the selection process ranks the member chromosomes of the generation according to their fitness values. The topmost chromosome has the highest while the bottommost has the lowest fitness value. The topmost individual is directly moved to the next generation while remaining individuals have to go through the roulette wheel based selection process. Each individual is given a portion of the wheel in proportion to its rank. The higher the rank, the larger will be the portion. After every wheel spin and selection, the wheel is renewed based on the remaining individuals' ranks. Once 100 individuals are moved to the next generation the process is terminated.

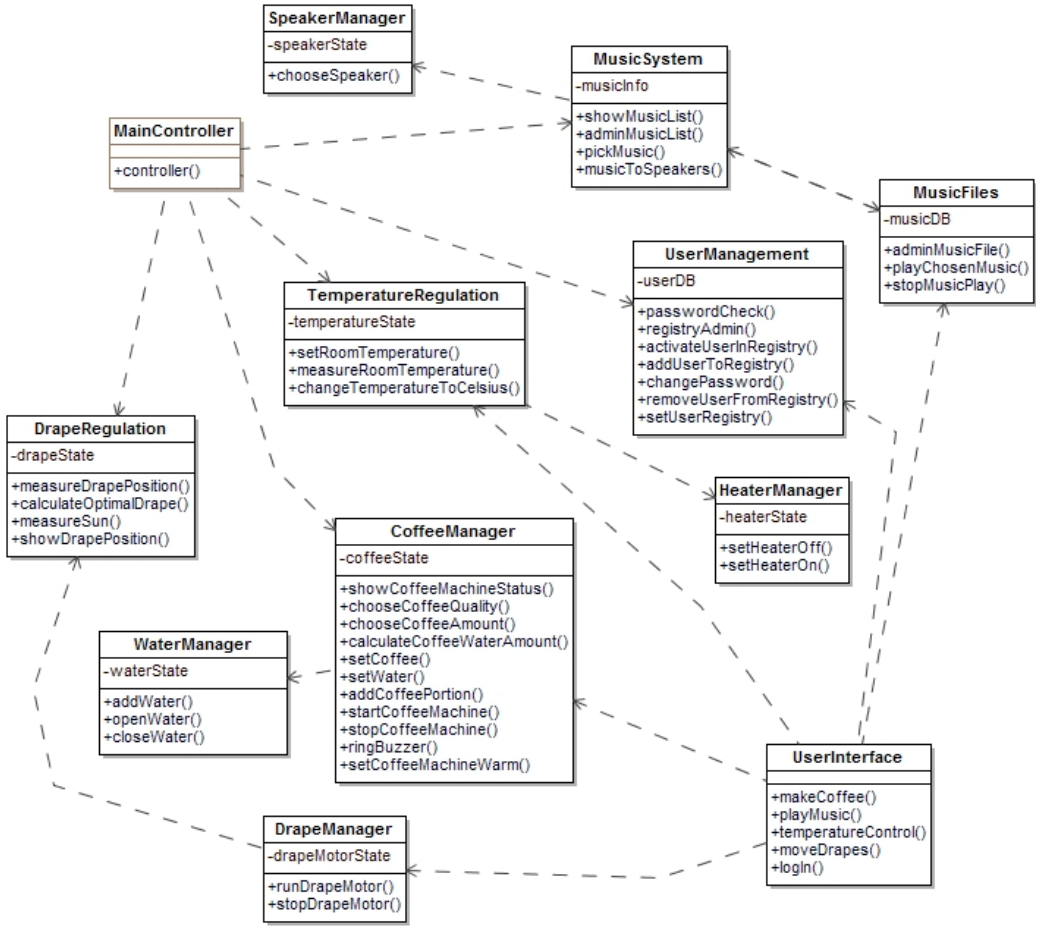


Figure 3.3 Null architecture for the e-home system

3.3 Experiments and Evaluation

The experiment concerns the architectural synthesis of an example system, named e-home or electronic home control system. The system enables its users to control temperature, music, coffee making and lighting in a home. The lighting is controlled through the movement of drapes over the windows. The example system is reasonably complex with 12 components and 56 operations as shown in Figure 3.3. The null architecture for the system is shown in Figure 3.3. The components TemperatureRegulation and HeaterManager handle the temperature. The MusicSystem, MusicFiles and SpeakerManager together implement the music management and control functionality. The DrapeManager and DrapeRegulation components handle the drapes while CoffeeManager and WaterManager control the coffee machine. The UserManagement component manages the security of the system by only allowing access to the registered users. The UserInterfacee component implements the user interface of the system.

An estimated set of support data for each operation is also provided. For example, there can be multiple ways to show the status of a coffee machine. Therefore, the operation "showCoffeeMachineStatus" in CoffeeManager has a high value for the "variations" (or variability) parameter in its support data. Music is played (playChosenMusic) more frequently than controlling the drapes (runDrapeMotor) or changing the password (changePasswor). To add a user to registry (addUserToRegistry) requires more parameters than turning on the heater (setHeaterOn).

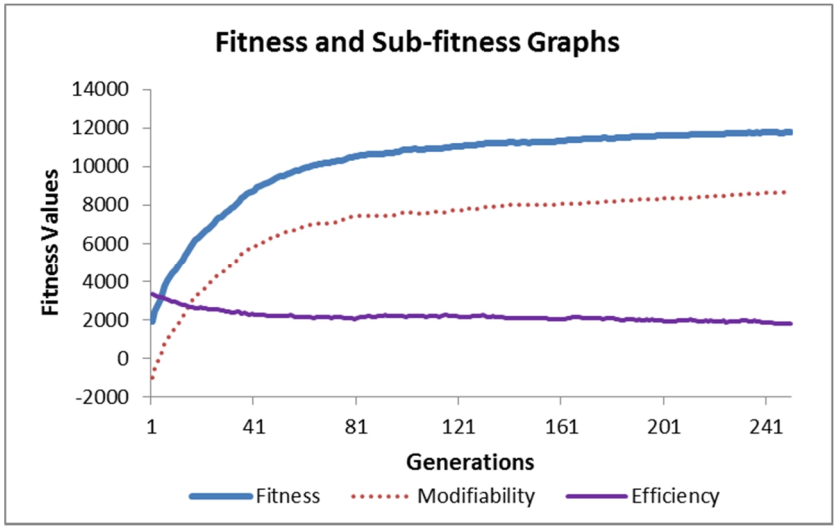


Figure 3.4 Fitness and sub-fitness graphs for the e-home system

Figure 3.5 Proposal for the e-home system

In the genetic algorithm population size is set to 100 while the number of generations is 250. Furthermore, the modifiability sub-fitness has been slightly over weighted compared to other sub-fitness functions. The relative comparison of using different weights is covered in the included publication [P1]. The average fitness and sub-fitness graphs for 10 runs are shown in Figure 3.4. The generations are on the horizontal axis while the fitness values are on the vertical axis. The overall fitness has

increased throughout the evolution. The modifiability sub-fitness grew more than efficiency, in fact, efficiency has suffered. It is because the solution choices made to increase modifiability have an adverse effect on efficiency.

In the proposed architecture, as shown in Figure 3.5, the Strategy solution has been used extensively. The operations with significant variability values are participating in the Strategy solutions. For example, showCoffeeMachineStatus can offer two implementation strategies in the proposed architecture. The presence of the Dispatcher solution is also due to the favoring of the modifiability sub-fitness. Furthermore, Template solutions are introduced to address the small possibility of variations in the registerAdmin, addUserToRegistery, addCoffeePortion and ringBuzzer operations. Similarly, possibility of changes in the setRoomTemperature operation's interface has been addressed with the Adapter solution. The application of the Server solution on the DrapeRegulation component is sensible as DrapeManager component is accessing its many operations through the message dispatcher. In the same way, the MusicFiles component needs many services from the MusicSystem (Server). The complexity has been increased as there are more classes and interfaces in the proposal than in the null architecture. Overall the solutions introduced by the genetic algorithm are reasonable and have led to improved quality compared to the null architecture.

3.4 Discussion

There are many factors that can influence the outcomes of the genetic algorithm and the shape and development of the fitness and sub-fitness graphs. Consider, for example, the population size parameter, using a large value for the parameter increases the opportunities for reproduction (crossover) and mutations. This could increases the likelihood of producing better individuals in early generations. Therefore, fewer generations would be required for the fitness and sub-fitness graphs to reach their peak values.

An increase or decrease in the weight of a sub-fitness function could shift its graph to higher or lower values. Note that the example null architecture is the most efficient architecture with no interfaces and other solutions. Therefore, an increase in the weight of efficiency may discourage the use of solutions with a negative impact on the speed of the system. It is likely that the produced architectures will be missing a message dispatcher and might contain fewer interfaces or modifiability improving solutions. Consequently, the efficiency will not suffer significantly but at the same time modifiability will not improve considerably. To balance the situation, the modifiability weight could be increased until equilibrium is achieved between both quality attributes.

The mutation rates also have an impact on the outcomes. For example, assuming equal weights for the sub-fitness functions, increasing the mutation rate of the Add Interface mutation may lead to more rapid introduction of interfaces in the architectures. This could result in a rapid growth of the modifiability fitness function and a steeper decline of the efficiency sub-fitness right from the beginning of the evolution. A probable consequence of the situation is a large number of interfaces in the produced architectures. Typically, the mutation rates should be tuned so that every solution gets a fair chance of participation, however, unless the architect him or herself wishes to do otherwise.

Another important factor is the size of the null architecture. It could influence the time it takes for the fitness/sub-fitness graphs to reach their highest values. The more classes there are in the null architecture, the larger will be the search space and therefore the genetic algorithm may take longer to find the best architecture. More generations and larger population sizes might be required for the fitness graphs to reach their highest possible values. Furthermore, the data associated with each of the operations can change the outcomes, too. Let's say a null architecture holds large values for the call frequency parameters of operations. During its genetic evolution, sub-fitness functions using the call frequency parameter as multiplier (e.g., negative efficiency) will have large values and therefore a greater impact on the evolutionary process. This kind of unfair advantage can be reduced through weights balancing.

It is obvious from the above discussion that every input parameter has potential to alter the fitness and sub-fitness graphs as well as the produced architectures. Therefore, a greater care has to be taken when setting theses parameters. The system dependent inputs like the null architecture, its size and attributes associated with the operations will vary only when the system itself changes. However, for one category of systems, values for adjustable parameters like weights, number of generations, population size and mutation rates can be calibrated through experimentation. Once calibrated, the values for the adjustable parameters can be reused for other systems belonging to the same category.

3.5 Empirical Study

In the empirical study [P1], experts have compared the automatically produced architectures with the architectures designed by undergraduate level students at Tampere University of Technology. The students were in their third year with major in Software Systems. The students were asked to design the architecture of an example system. In parallel, the genetic algorithm was used to produce 10 proposals for the same example system under the same setup discussed in this chapter. The same set of

information and solutions were available to both the genetic algorithm and students. The algorithm on average took one minute while the students on average needed 90 minutes to produce the architectures.

A set of architectures from the genetic algorithm and students was then presented to the experts. The experts were not aware of which architectures are from the students or from the genetic algorithm. The experts were then asked to rank the architectures based on their quality. The study revealed that the automatically generated architecture have quality comparable to the students' proposals. In fact, the experts have ranked the synthesized architectures slightly better than the designs of the students. The limitations of the empirical study are presented in Section 9.2.4.

Our work of genetic architectural synthesis is still in its infancy, yet the produced architectures scored well against the student made designs. The genetic algorithm was efficient as it took a fraction of the time the students have invested in the same problem. Certainly, even an experienced architect will find it difficult to beat the genetic algorithm in efficiency. It can be anticipated that with time understanding of the problem will increase and therefore future versions of the genetic algorithm could produce architectures comparable to designs of experienced architects.

4 Tool Support: Darwin

4.1 Introduction

The developed Darwin [P2] environment provides tools to enable the genetic synthesis of software designs. It has been built around the genetic algorithm introduced in Chapter 3. Using the embedded tools, the results from the genetic algorithm can be studied in more detail along different dimensions. UML [36] based CASE tools have also been integrated into Darwin. The tools can be used to design the null architecture as well to visualize and study the improved proposals from the genetic algorithm. The environment provides views to give values for the mutation probabilities, population size, number of generations, scenarios and weights of the sub-fitness functions. It also enables an architect to visualize the growth of the fitness and sub-fitness function on a graph. Note that the scenarios (a feature of the genetic algorithm) have not been used by the author in his work and therefore they will not be explored in details in this thesis. For readers interested in scenarios, see [26].

4.2 Darwin's Features

The user interface of the tool is shown in Figure 4.1. It contains Evolutions, Generations, Probabilities, Evolution Parameters, Graph, Family Tree, Weights and Scenarios views. In the Evolutions view a user can create, open, save, remove or alter an evolution. An evolution stores all the data related to an experiment, input architecture, mutation probabilities, population size, number of generations, weights, scenarios, proposed architectures etc. It can be stored on the disk and can be re-opened later at any time to resume an experiment.

Before the Darwin environment, the null architecture used to be provided in a text file to the genetic algorithm. With Darwin, it became possible to build the architecture in UML [36] notations using use case, sequence and class diagrams. The class diagram

based representation of the null architecture can be directly given as input to the genetic algorithm in Darwin.

An evolution can be divided into multiple periods. In an environment where mutation probabilities, population size, number of generations, weights and scenarios vary, multiple periods can be introduced and different parameters can be inserted for the periods.

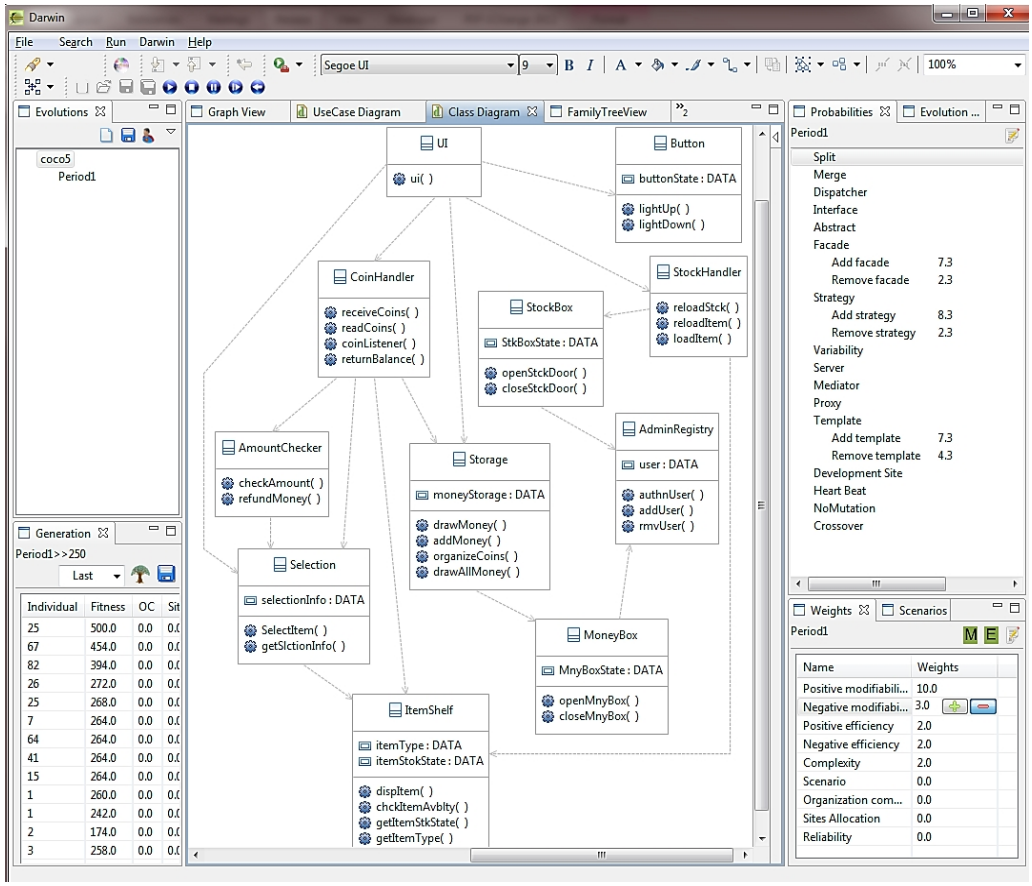


Figure 4.1. Darwin's User Interface

In the Generations view, individuals in a generation can be listed along with their fitness values. From the list, an individual architecture can be opened as a UML class diagram. Also, the view provides controls to visualize the origin of an individual in the Family Tree view. It shows all the parent and ancestor architectures that have contributed to the good or bad qualities of the individual. The tool also provides Evolution Controls to start, pause, resume and stop an evolution.

The Graph view displays fitness development during an evolution. Moreover, it can be used to visualize all the sub-fitness curves (modifiability, efficiency, complexity and reliability). On the horizontal axis are the generation numbers while on vertical axis are the fitness/sub-fitness values. The graphs are drawn using a representative fitness value for each of the generations in an evolution. The values can be either the fitness of the best architecture in the generations or an average of their elites. One of the two options can be used in an evolution which can be selected from the Evolution Parameters view. Moreover, total number of generations and the population size can also be specified in the Evolution Parameters view for a period. The Probabilities view lets a user specify probabilities of the mutations, null mutation and crossover. The weights for the involved sub-fitness functions can be entered in the Weights view. The Scenarios view can be used to add scenarios to an evolution.

4.3 Darwin's Internal Architecture

Darwin exploits plugin-based architecture of Eclipse [76], a widely used IDE (Integrated Development Environment). As shown in Figure 4.2, Darwin itself is a plugin extending Eclipse's user interface (UI) with features necessary for the genetic synthesis of software architectures. The genetic algorithm has also been wrapped inside a separate plugin.

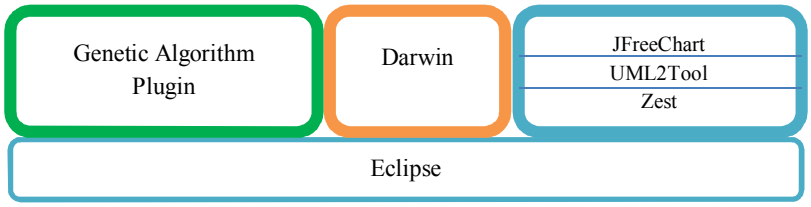


Figure 4.2 Darwin architecture

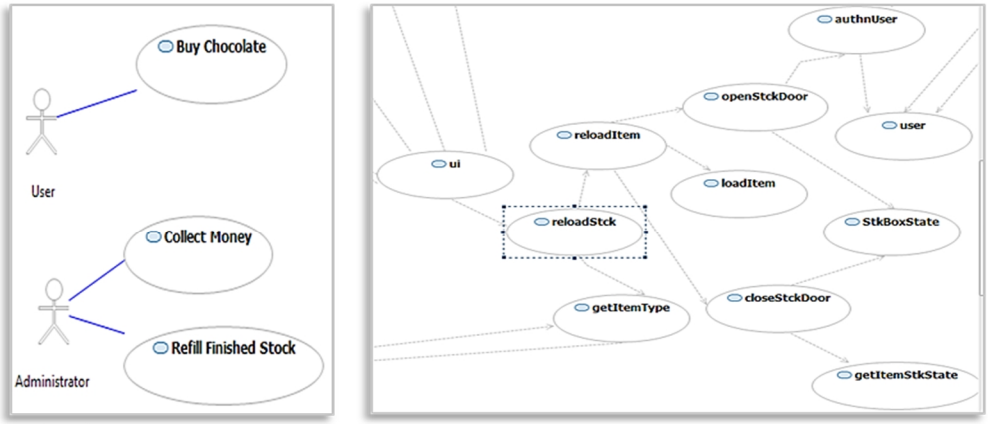
The Darwin plugin also relies on UML2Tool [77], JFreeChart [75] and Zest [56] plugins for its functionality, as shown in Figure 4.2. UML2Tool is a CASE tool providing graphical editors for UML use case, sequence and class diagrams. The editors can be used to refine the abstract requirements, represented as use cases, into a null architecture. The JFreeChart plugin provides tools for visualizing variety of graphs. It has been used to draw the fitness and sub-fitness graphs. The Zest plugin is a visualization toolkit that is used to draw the family trees in the Family Tree view.

Internally, Darwin's architecture follows Model, View and Controller (MVC) pattern [69]. An evolution is the model which holds all the information about one genetic evolution. It contains all input parameters for the genetic algorithm like the null

architecture, mutation probabilities, population size, number of generations, weights, scenarios and other settings. The outputs from the genetic algorithm are also stored in the model. The output holds all intermediate architectures and the proposed best architecture along with their overall and sub-fitness values. The Views are Evolutions, Graph (exploits JFreeChart), Family Tree (uses Zest), Generations, Probabilities, Evolution Settings, Weights and Scenarios views where each view shows a different aspect of the model. Within the views are controls that allow modification of the aspect the view is responsible for. Furthermore, Darwin plugin adds controls to the UI to start, pause, resume and stop an evolution.

4.4 Software Architecting using Darwin

The aim of this section is to provide a guided tour of the tool using an example Automated Chocolate Vending Machine (ACVM) system. First, using the CASE tools, a use case diagram containing three main use cases is drawn, as shown in Figure 4.3 (a). Using the ACVM machine, a user can only buy chocolates. The second user, administrator, can get the money out of the machine as well can initiate the refilling of the empty stock of chocolates.



(a) Abstract use cases (b) Refinement of the "Refill Finished Stock" use case

Figure 4.3 Chocolate vending machine's use cases and refinement

The second step is to refine the three use cases. A refinement of the "Refill Finished Stock" is presented in Figure 4.3 (b). From the user interface (UI) of the machine the administrator should be provided an option to reload the stock (reloadStck). The reload option will ask the administrator about what type of chocolate to reload (getItemType) and then execute series of actions opening the stock door (openStckDoor), loading the selected chocolate (loadItem) and finally closing the stock's door (closeStckDoor) to

accomplish the task. The security requirements require that the administrator has to be asked for authentication before opening the stock door (authnUser, user). Furthermore, the door's status (StkBoxState) and selected chocolate item status (getItemStckState) have to be kept updated during the whole process. The support data can be provided for each use case in the IDE's default properties view.

In the refinement process, the sequence diagrams provided by the UML2Tool [77] plugin can also be used. The refinement resulted in 37 operations which are then collected into classes to form the null architecture in the UML class diagram editor of the UML2Tool plugin. The null architecture for the ACVM system is shown in Figure 4.1. The next step is to set up the evolution and values for the population size, number of generations and mutation probabilities. For ACVM, only one period has been used with population size and number of generations set to 60 and 180, respectively. For the mutation probabilities, the default values provided by Darwin have been used.

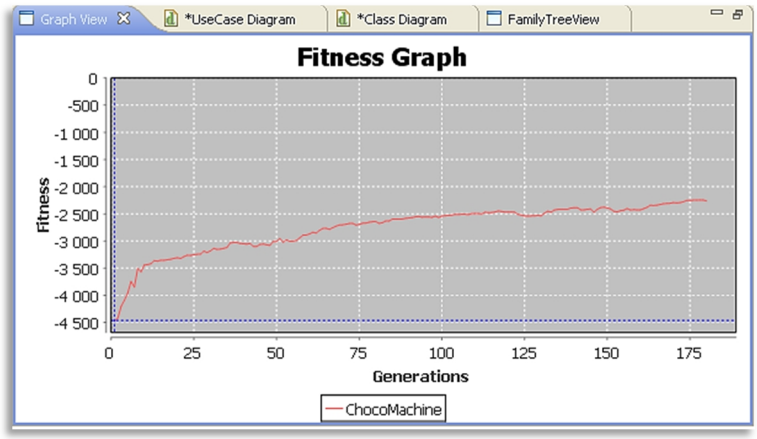


Figure 4.4 Overall Fitness graph

The simulated genetic evolution has been started using the Evolution Controls. The evolution graph develops as the evolution progress. Furthermore, the input parameters (mutation probabilities, number of generations, population size and weights) can be changed while the evolution is in progress. The changes are immediately effective. This feature provides a more interactive experience as the user can see right away how different values of the input parameters are affecting the growth of the fitness graph. Figure 4.4 shows the development of the fitness graph. The increasing fitness values indicate that the genetic algorithm has been able to improve the null architecture's quality over 180 generations. A user can select a generation from the graph.

The best architecture proposed by the genetic algorithm is shown in Figure 4.5. As visible in the figure, the algorithm has introduced interfaces as well as the Adapter, Template and Strategy patterns or solutions in the null architecture. The newly generated classes or interfaces for the solutions have been assigned automatically generated names, e.g., Adapter51, TemplateClass70 and IStrategy64. It is challenging to infer meaningful names automatically. In the future, the algorithm could be made more intelligent to assign meaningful names to the new classes and interfaces.

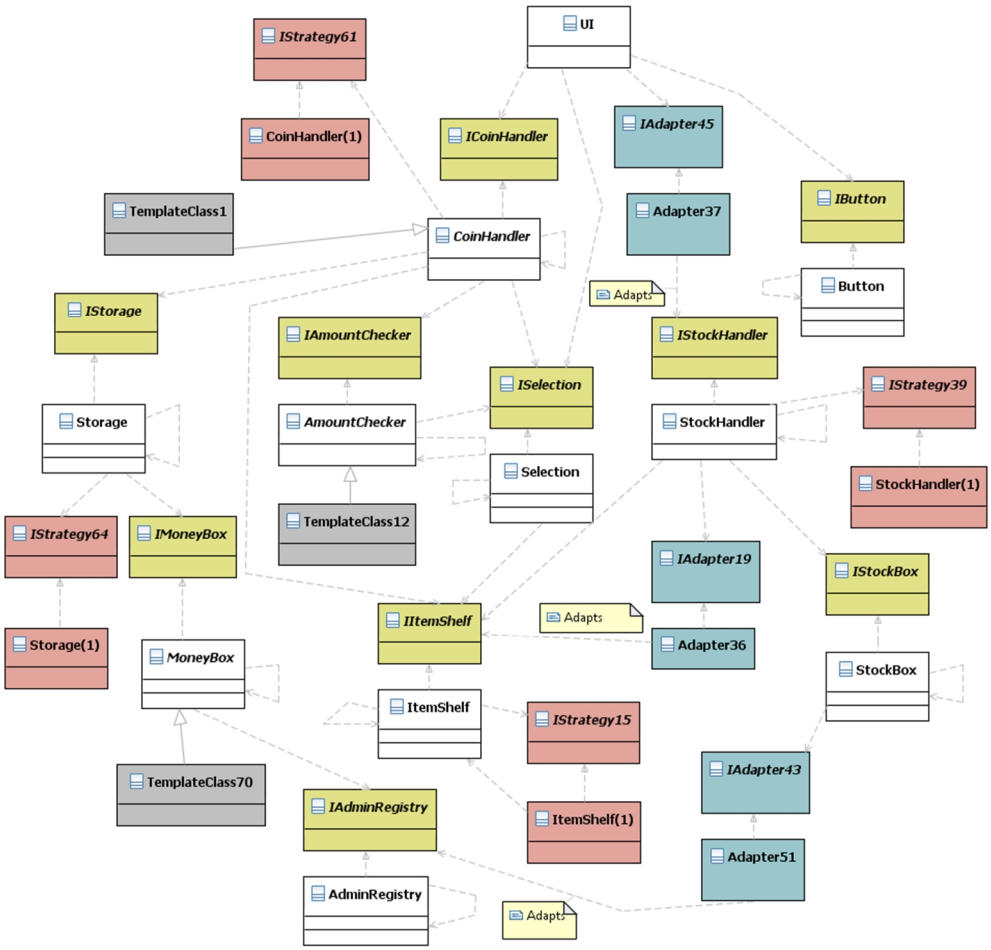


Figure 4.5 The proposed architecture for the ACVM system

4.5 Tool Efficiency

The Darwin tool enables the user to choose from two modes; fast and slow modes. In the fast mode most of the intermediate data is not stored. In the slow mode, however, everything is stored. The fast mode is more suitable for interactive experience while the

slow mode is good for deeper study of the simulated evolution. At the same time the efficiency is also dependent on the size of the system under study and its complexity as well as the input parameters to the genetic algorithm. Using a large population size or high number of generations will certainly increase the processing time as is evident from the graph of Figure 4.6. The graph shows the processing time of the genetic algorithm in relation to the increasing population sizes for the ACVM system under fast mode setting. The generations were 100 in all the runs. With the population size of 300, the genetic algorithm took 96 seconds to propose an architecture.

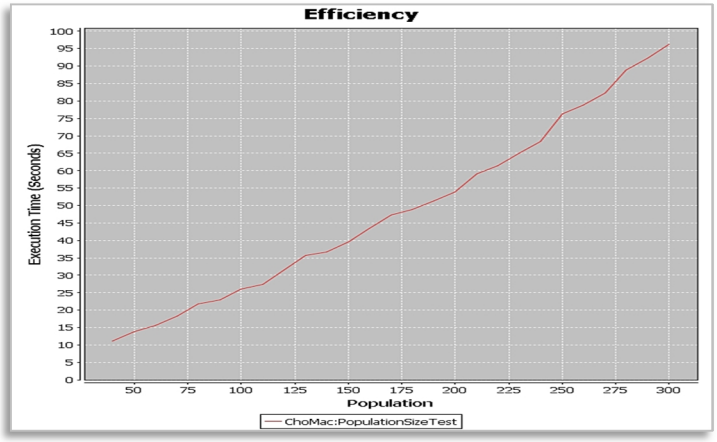


Figure 4.6 Population size vs genetic algorithm execution time

5 Work Planning Automation

5.1 Introduction

Software architecture is influenced by its development environment [57]. The factors like team structure, cultural and lingual differences among the teams as well as resources and skills of the people contribute to the final shape of the architecture. Two teams located in different regions with different languages and cultural values may develop components that may not be able to interact with each other [29]. Therefore, a correct translation of all the involved variables into software architecture is of high importance.

A poor dissemination of tasks among the teams can lead to an increased communication overhead [30]. The job of work planning becomes more challenging in presence of dynamic teams with varying traits and sizes. In this work, the differences among the teams are collectively denoted as *team distance*. A team should be interacting conservatively with another team at a high team distance. Two teams with a low team distance can communicate more frequently and clearly. The work planning must ensure that communication channels with significant resistance are less active, unless the resistance is removed. This will lead to faster development and less inconsistency among the components developed by the teams.

The type of dependencies among components also matters in work planning. The inter-component dependencies are originated from the dependencies among the operations they host. Two highly dependent components will raise the level of communication between the teams developing them. Conversely, two loosely connected components will need less interaction among the teams. Therefore, allocation of components to different teams is not only influenced by the team distances but also by the nature of the relationship among them.

In this study, the genetic algorithm introduced in Chapter 3 has been extended to address the work planning problem [P3]. The aim is to produce good quality architectures which are easy to distribute among teams. The algorithm now takes into account the organizational structure. The genetic algorithm introduces solutions and applies mutations to produce architecture suitable for the developing organization from the work distribution point of view. The outputs also include an initial work distribution plan.

5.2 Genetic Algorithm Extensions

5.2.1 Inputs

The input to the genetic algorithm contains the null architecture and the structure of the developing organization. The structure encompasses the teams, their sizes and distances among them. The size approximates a reasonable number of operations that can be assigned to a team without over-loading it. The team distance parameter is an estimated numeric value representing the level of difficulty in communication between any two teams. The higher is the value of team distance, the higher is the resistance in communication. The inputs also include an initial random work distribution within the null architecture. The developing team information is associated with each component in the null architecture.

5.2.2 Mutations

All the mutations introduced in Chapter 3 were used in this version of the genetic algorithm. In addition, a new mutation "Change team" has been introduced in the genetic algorithm for the purpose of this application. The mutation not only changes the node information for the operation in the target supergene but also for all the operations sharing the same component. As a result, the hosting component is moved from one team to another.

5.2.3 Fitness Function

The equation (5.1) shows the fitness function including modifiability (f_m), efficiency (f_e), complexity (f_c), communication overhead (f_{co}) and operation allocation (f_{oa}) sub-fitness functions. The multipliers (w) are the weights of the corresponding sub-fitness functions. The first three sub-fitness functions (modifiability, efficiency and complexity) are collectively referred to as core fitness in this work. The core fitness has been covered in details in Chapter 3. The communication overhead (CO) sub-fitness, as the name implies, measures the communication overhead associated with an

architecture. The operation allocation (OA) sub-fitness function measures the suitability of an allocation from the point of view of over/under loading of the involved teams.

$$f(a) = w_m sf_m + w_e sf_e - w_c sf_c - w_{co} sf_{co} - w_{oa} sf_{oa} \quad (5.1)$$

The nature of dependency among the components is directly correlated to the communication that will occur between the teams developing them. A strong dependency will require more communication compared to a weaker dependency. In this work, three classes of dependencies have been considered, direct, interfaced and through a message dispatcher. The direct dependency is the strongest one while the use of a message dispatcher implies a weaker dependency given that the messaging interface is well understood. The use of an explicit interface has been scaled as a moderate level dependency.

5.2.4 Communication Overhead

The communication overhead (CO) sub-fitness is shown in equation (5.2). A chromosome contains streams of operations as genes belonging to an architectural design. Let i and j be the pointers traversing all the genes sequentially. Also, assume that function $t(i)$ gives team information of the i^{th} gene. Furthermore, let $d(t_1, t_2)$ denote a team distance function which gives team distance between any two teams. Likewise, assume function $D(i, j)$ which provides the weight of the dependency between i^{th} and j^{th} operations derived from the relationship between their hosting components. A direct dependency has the highest weight whereas a dependency through the dispatcher has the lowest weight. A dependency passing through an interface has intermediate level weight. If there is no dependency between i^{th} and j^{th} operations, the weight is zero and therefore has no influence on the sub-fitness function.

$$Minimize\ sf_{co} = \sum_i \sum_j D(i, j)\ d(t(i), t(j)) \quad (5.2)$$

5.2.5 Operations Allocation

The operation allocation (OA) sub-fitness function is shown in equation (5.3). The function iterates over all the teams and for each team k it measures under or over-load using the load function $L(k)$. Every team has a size S_k which is the number of operations that can be allocated to the team. The number of allocated operations is represented by the variable A_k. For a team with operations more than its size, over-load is calculated. The under-load calculation is performed for a team with operations less than its size. An empty team with no allocation has no influence on the sub-fitness

function. Both over-load and under-load have weights w_{ol} and w_{ul}, respectively. Since an over-load situation is more undesirable than an under-load situation, $w_{ol} \gg w_{ul}$.

$$Minimize\ sf_{oa} = \sum_k L(k) \tag{5.3}$$

where

$$L(k) = \begin{cases} w_{ol}(A_k - S_k), if\ A_k \geq S_k \\ w_{ul}(S_k - A_k), if\ A_k < S_k \\ 0, \qquad\qquad if\ A_k = 0 \end{cases}$$

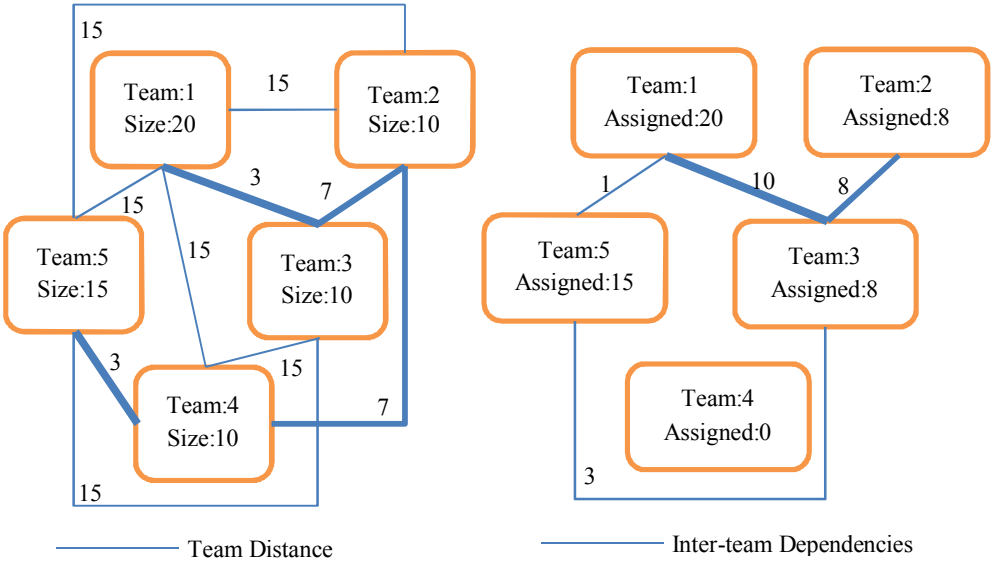
5.3 Experiments and Evaluation

In the experiment, the e-home system from Chapter 3 has been used again. The null architecture of the system is presented in Figure 3.3. The organizational structure of Figure 5.1 (a) has been used with the null architecture. The structure of the organization has five teams with different sizes. The team distances are shown on the connecting lines among the teams. A thin inter-team distance line expresses high resistance in the communication between the teams. The aspects of interest are:

1. How CO and OA sub-fitness functions develop during an evolution?
2. How the CO and OA sub-fitness functions influence the development of overall and core (modifiability, efficiency and complexity combined) fitness functions?
3. How well the work distribution suits the developing organization's structure?
4. What kind of dependencies are preferred by the genetic algorithm and why?

In the following the aspects will be addressed one by one. Since CO and OA have to be minimized, during the evolution, both sub-fitness functions start from a high point and reach a low value at the end of the synthesis as shown in the graphs of Figure 5.2 (a).

The overall and core fitness (modifiability, efficiency and complexity combined) are shown in Figure 5.2 (b) and (c), respectively. The overall fitness is lower with the new fitness functions, however, the difference between the lowest and peak point is comparatively high. The core fitness graph is also lower with CO and OA sub-fitness functions. It is natural that whenever a new aspect is brought in for consideration in software design process, some compromise has to be made on all other considered aspects. Therefore, the modifiability, performance and complexity had comparatively less room to evolve with inclusion of the CO and OA sub-fitness functions.



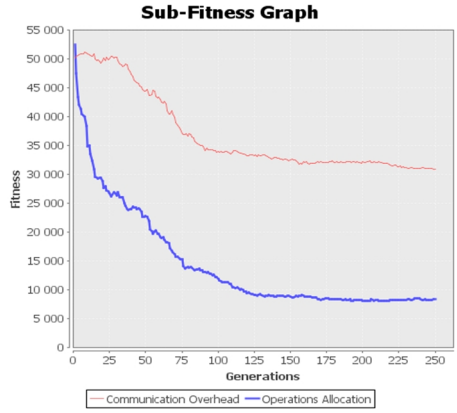
(a) Organization (b) Work allocation

Figure 5.1 Organization's structure and assignment

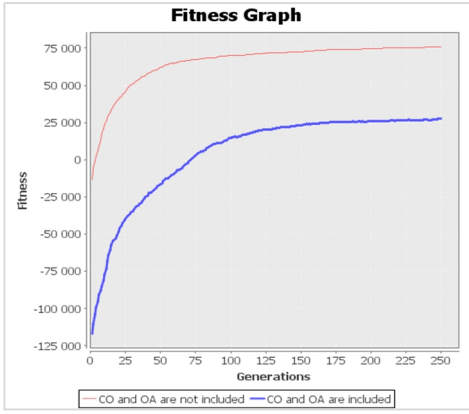
At the end of the genetic synthesis, the genetic algorithm proposed the work allocation as shown in Figure 5.1 (b). As far the over/under-load is concerned, none of the teams were over-loaded. Furthermore, one team Team4 has been spared from any work, thus using minimum number of teams to get the work done. The connections among the teams denote the number of inter-team communications or dependencies. A thick inter-team dependency line represents high number of interactions. From Figure 5.1 (b), it is visible that the genetic algorithm has reduced the amount of interactions among the teams with greater distances. For example, there is a high distance between Team3 and Team5 therefore only three dependencies exists between them. In contrast, Team3 has the highest number of interactions with Team1 because of the low team distance. Team2 and Team5 are not interacting at all because of the large team distance.

The types of the dependencies among the components on different teams are shown in Figure 5.3. Almost all the dependencies are going through the interfaces, except for the two between Team3 and Team2 which are through the message dispatcher. Intuitively, more connections through the message dispatcher were expected in order to reduce the communication over-load. However, an increase in the message dispatcher connections will have a negative effect on the performance and complexity of the architecture. Therefore, the genetic algorithm has to reach an architecture where

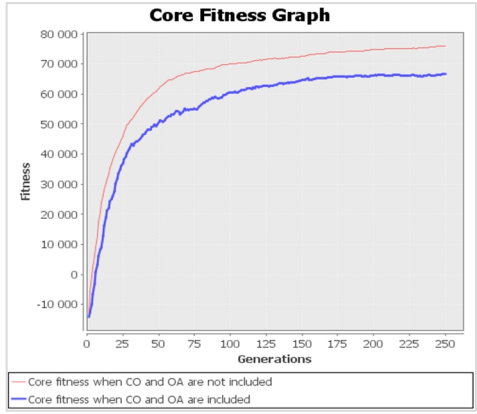
communication over-load is reduced with a reasonable compromise on other quality attributes. Use of interfaces is a fair choice as their use will increase modifiability, reduce CO as well as cause comparatively little damage to the efficiency and complexity of the system.



(a) CO and OA sub-fitness



(b) Overall fitness



(c) Core fitness

Figure 5.2 Fitness graphs

The part of the architecture assigned to Team2 is shown in shown in Figure 5.4. In addition to the message dispatcher, architectural solutions like Adapter, Strategy and Template [11] are injected into the null architecture to improve modifiability. The efficiency sub-fitness has been able to control the excessive use of not only the message dispatcher but interfaces as well. The complexity fitness function had penalized the creation of new classes and interfaces as a result of application of architectural solutions. It helps avoid the situation where each method would end up in its own class

at the end of the genetic synthesis thereby acting a counteracting force against intense fragmentation of the components.

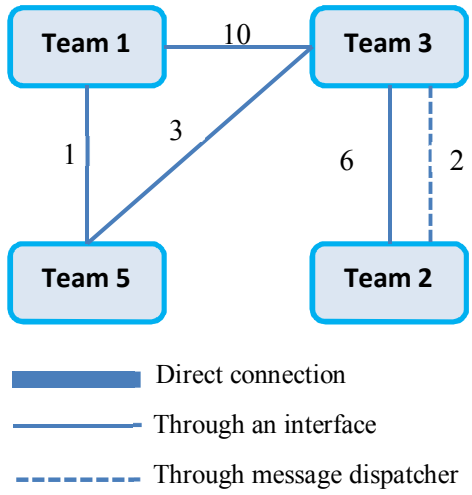


Figure 5.3 Connection types in the proposal

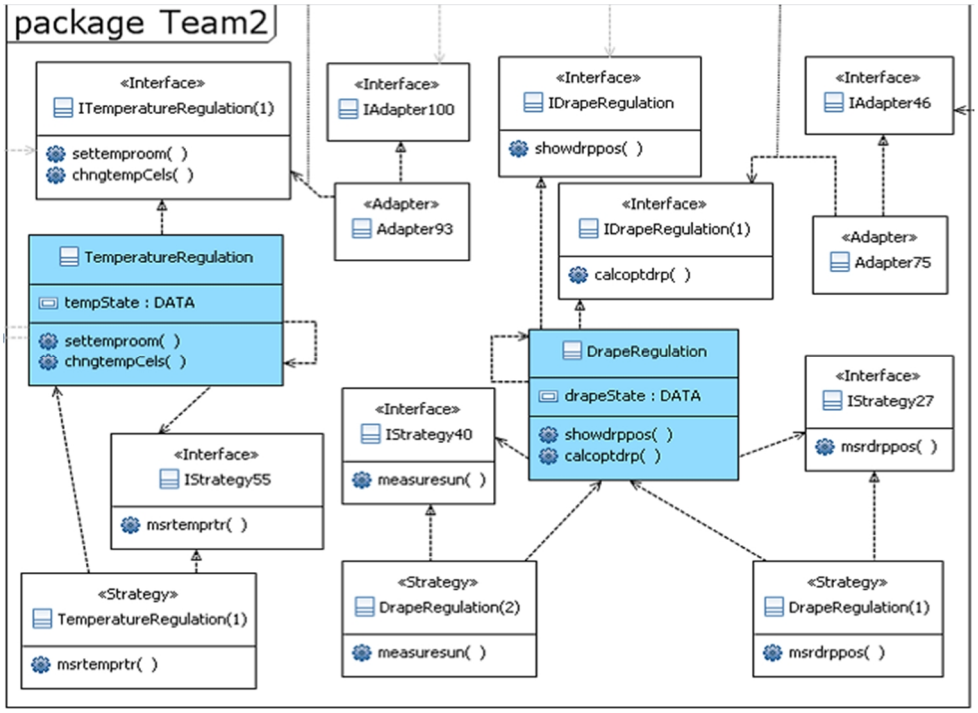


Figure 5.4 Fraction of the architecture assigned to Team2

To summarize, the genetic algorithm has made sensible solution choices and component allocations. The communication overhead reducing solutions were motivated among the components allocated to distant teams. Also, over-loads were avoided in the proposed initial work allocation. The core fitness had suffered, however, it is a natural price for including the new objectives into the design process.

PART III – Applying Genetic Architectural Synthesis in Software Maintenance

"At present, software is like clay: it is soft and malleable early in its lifetime, but eventually it hardens and becomes brittle. At that point, it is possible to add new bumps to it, but its fundamental shape is set, and it can no longer adapt adequately to the constant evolutionary pressures of our ever-changing world. We believe a critical goal of software engineering is to produce software more like gold- malleable and flexible for life."

Harold Ossher and Peri Tarr [67]

6 Solution-Based Self-Adaptive and Run-Time Maintenance

6.1 Introduction

During software development, all future needs cannot be foreseen. Therefore, a software system will always require maintenance in order to adapt to changing requirements or environment. According to [33] about 18% of maintenance is adaptive maintenance encompassing system updates introduced in response to unanticipated changes in the requirements or environment [32]. Given that the entire cost of maintenance is estimated to vary between 40% and 75% of the total cost of a software system during its lifetime [31], adaptive maintenance represents a notable portion of the software related costs. Thus, any techniques that could at least partly automate adaptive maintenance would be welcome. On the other hand, for many continuously running systems like embedded machine control systems, space systems, telecommunication systems or web systems traditional manual offline maintenance causes undesirable operational breaks and in some cases significant financial losses. For these kinds of systems, run-time adaptation of software systems would be desirable, even when performed by a human.

Ideally, a system should be capable of adapting itself according to the changes in its environment, without human involvement. So-called self-adaptive systems [34] are autonomous systems that are capable of observing their environment and modifying themselves to fit the new circumstances in the environment. Self-adaptive systems have been the topic of active research during the last few decades (e.g. [3], [37], [88]). On the other hand, if the required modification is triggered by phenomena that cannot be observed by a machine (e.g., changes in the company's policy, new software in the market etc.), there is a need to support manual but run-time adaptation as well, so that continuously running systems can be kept operational without long maintenance breaks.

In many existing studies ([7], [8], [38], [68], [85] and [100]), software architecture has been the focal point for self-adaptation. Indeed, according to Kramer et al. [101], software architecture can very well serve as a target for run-time modifications in the self-adaptive paradigm. This has clear advantages. Software architectures have been actively studied and many methods and techniques supporting the creation of software architecture have been developed, to be applied also in self-adaptive systems. Moreover, different kinds of software systems share standard fundamental concepts used in software architecture designs and therefore an approach based on software architecture can leverage on the standards to extend beyond limited number of applications. Furthermore, software architecture presents an abstract view of a software system thus making it relatively easy to introduce changes using components and connections rather than going into fairly complex lower level description of a software system. Various kinds of standard solutions, design [11] and architectural patterns [12], have been identified as quality-improving elements of software architecture, providing a natural basis for architectural improvement also in self-adaptation.

In this work, a hybrid run-time adaptive maintenance approach has been proposed. The approach combines both self-adaptive and manual maintenance of Java-based distributed systems at the architectural level. In the self-adaptive mode, the aim is to optimize the efficiency and reliability of a system in a changing environment by allowing the system to modify its own architecture. The optimization technique applies the genetic algorithm (introduced in Chapter 3) to produce a new improved architecture based on information captured from the running instance of the system. The manual modification mode, on the other hand, provides a high-level view, UML class diagram, of the software architecture of a running system and allows an architect to edit that view, so that the changes are automatically passed on to the running instance of the system. Since software architecture is the focus of adaptation in this work, the presented approach can be called a self-architecting approach.

In this work, *solutions* serve as the basic unit of modification. As mentioned earlier, a solution can be a design or an architectural pattern, but it can also be any non-standard architectural solution that has an identifiable effect on the representation of the software architecture. The presented concept of a solution is in line with the decision-centric view of software architecture [102]: a solution is essentially the effect of an architectural decision in the software architecture.

A significant advantage of a solution-based approach to self-adaptive systems is that the modified system remains understandable for humans. Since the solutions are usually well known and documented entities, the understandability of the system is not threatened even if they are introduced automatically by a machine. Anybody familiar

with the solutions can still easily understand the system, especially if the existence of the solutions is clearly commented in the code.

The employed genetic algorithm does not require pre-defined adaptation strategies. This, together with the generalized notion of solution, makes the developed approach an open self-adaptive [37] approach. In this work, the nature or scope of a solution is not restrained, in principle the whole architecture can be targeted for improvements, while existing approaches apply the changes to a certain part of the architecture. Although primarily well-known design patterns are used in this work, in principle any architectural solution can be incorporated in the developed approach, as long as it can be provided with automated insertion and removal operations.

As a proof of concept, the SAGA (Self-Architecting using Genetic Algorithms) platform [P5] has been built to support solution-based manual and self-adaptive architectural maintenance at run-time for Java based distributed systems. The infrastructure builds on Javeleon [35], which is a platform providing run-time updating facility of Java classes. The approach is applied to an example system, demonstrating both manual and self-adaptive maintenance through injection and removal of different solutions to/from the running instance of the system. In the self-adaption mode, optimization of the efficiency and reliability of the system has been focused in the context of changing usage profiles.

6.2 Solution-Based Software Architecture

Software architecture is defined as "*the fundamental conception of a system in its environment embodied in elements, their relationships to each other and to the environment, and the principles guiding its design and evolution.*" [58]. The elements (like components) and their relationships are parts of architectural solutions, resulting from the decisions made during the design process by the architect. Recently, decisions have been proposed as a fundamental concept of software architecture (e.g., [60]).

Basically, an architectural decision is made to resolve a design problem in the presence of various forces that influence the decision. A decision may lead to structural changes, applications of architectural styles [12] or design patterns [11], constraints or rules to be followed in the development, features, variation points etc. A decision is in turn influenced by the architect's experience, constraints, standards, guidelines, cost etc. [103]. The knowledge of the decisions is crucial to understand the context which has resulted in a particular architectural solution in a certain part of the architecture [60]. In this work, the effect of an architectural decision in some software architecture representation is called a *solution*. A solution can be the result of a single decision while others may be consequence of a set of architectural decisions.

The solutions resulting from the decisions can be grouped based on the nature of the problems they are solving. For example, one set of solutions may solve functionality related issues while others may address the quality requirements. For instance, the fault tolerant patterns [63] have a positive impact on the reliability of a system. Similarly, most of the standard design patterns [11] improve the modifiability of a system. Some decisions and therefore the solutions are imposed by the programming paradigm or underlying hardware infrastructure. For example, in a distributed environment, a communication interface (e.g., the message dispatcher solution [12]) has to be employed to let the components communicate over the network. Another example is the set of solutions inflicted by the organizational structure [57]. The structure of the organization may demand splitting or merging of components to efficiently distribute the work among the teams.

A solution can be generic or specific. A generic solution can be expressed in a way independent of any particular system or software element, while a specific solution is associated to certain fixed software elements. For instance, a design pattern is a generic solution, while the usage of a particular third party component is a specific solution. Generic solutions can be expressed as a set of roles, that is, placeholders for actual software elements. Here software elements can be any software units like classes, components, operations, nodes, sub-systems etc. When a solution is instantiated, certain actual software elements play certain roles in the solution, that is, the roles are bound to the elements.

A body of work (e.g., [70], [96], [97] and [98]) has attempted to represent a set of solutions like design patterns [11] with role-based models at different meta-levels. Riehle [96] has applied role-based definition for object oriented frameworks to address the issues originating from complex classes and object collaborations as well as requirements on using the frameworks. In Riehle's work, one or more roles can be associated with the classes in a framework while free roles are left for the users of the framework to provide. A class model contains all possible collaborations among the instances of the classes in a framework. Riehle also provides role-based definition for number of design patterns used in object-oriented frameworks. Riehle's view of roles stands at the class level. This work also employs similar role-based model for solutions of all kinds including the standard design patterns or architectural styles [11], however, the roles in this work are more fine-grained, as in [70], [97] and [98].

The developed role model for solutions is depicted as a concept diagram in Figure 6.1. A solution may place certain requirements for the legal binding of roles; these requirements can be expressed as (possibly informal) constraints associated with the roles. In addition, a role may be typed, implying that only certain types of system

elements can be bound to the role. System elements are the artifacts that can be used to realize solutions and therefore systems. A system element can participate in many bindings and therefore can be bound to many roles. Likewise, a role can be played by more than one system element. Moreover, roles can form hierarchical parent-child relations: a role can have child roles.

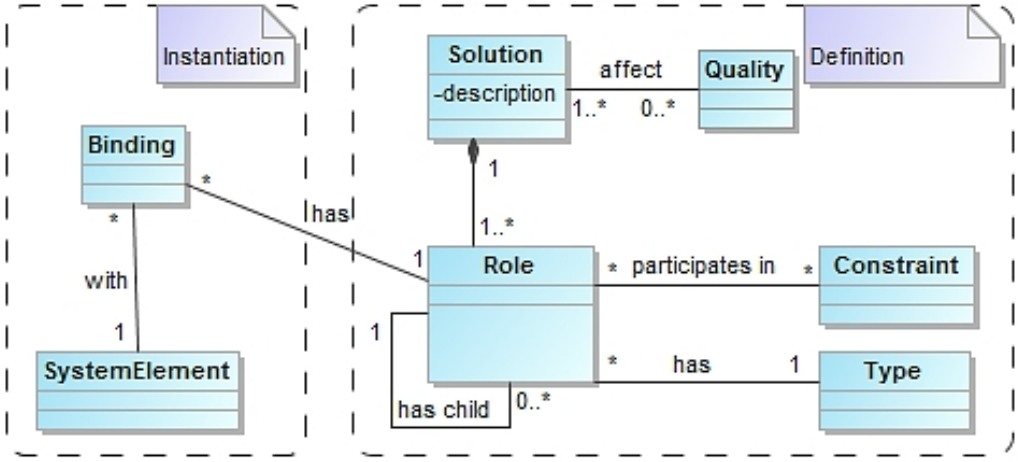


Figure 6.1 Meta-model of a solution

As shown in the model of Figure 6.1, each solution may or may not have an effect on quality attributes like modifiability, usability, efficiency, portability or reliability etc. A strictly functional solution will only introduce new features while a pure quality improving solution will only affect one or more quality attributes of the system. For example, when a Heartbeat [63] solution is introduced for a component in a system, it sends periodic messages (called beats) to other components to indicate that the component is alive. It adds nothing to the system's functionality but only improves its reliability by providing a monitoring apparatus. A missing beat message will indicate failure of the component and the system will be able to avoid a complete crash by invoking one of the employed recovery strategies. On the contrary, a third party component may just provide some functionality with no effect on quality. The solutions which lie in between the two extremes possess flavor of both functionality and quality. For example, the Observer [11] solution not only improves modifiability of a system but also extends its functionality at the same time.

In this work, a solution comes in and moves out of a system's design, thereby bringing in or taking out some properties of the system. In the developed manual and self-adaptive run-time maintenance approach, the unit of modification is therefore a solution, too. A solution in its entirety enters or leaves a system at run-time in order to

maintain a system's quality and functionality in response to new requirements, possibly imposed by new kinds of usage profiles.

6.3 Self-Maintaining Quality Using Solutions

Laddaga [47] characterizes self-adaptive software as follows: "*Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.*". The characterization highlights two essential stages, evaluation of the behavior and modification of the system in response to the evaluation. In this work, behavior evaluation concerns the software quality [59], i.e., measuring certain quality attributes at run-time. This places a fundamental requirement on the involved quality attributes: they must be observable at run-time by the system itself. Such quality attributes are *self-measurable* in nature. Once a quality attribute can be measured, only then can it be controlled.

Unfortunately, many quality attributes are not self-measurable. In particular, to measure subjective quality attributes like usability and learnability, some input from a human is required, which contradicts the goal of self-adaptive systems. Moreover, some quality properties are inherently difficult to measure even manually. For example, portability involves ease of re-design, coding, code transportation, re-testing and debugging, documentation, and deployment [99]. Providing automated measuring for these kinds of objectives appears to be next to impossible in the foreseeable future. Furthermore, some of the quality attributes don't even fit in the self-adaptive paradigm. For example, consider modifiability. Traditionally, modifiability is desired to keep the code easy for humans to change. However, when humans are out of the loop, modifiability in its original form becomes irrelevant. Instead, the ability of the system to change certain parts of itself is relevant.

In contrast, there are quality properties which can be measured or at least approximated automatically in a fairly straightforward way. For example, efficiency can be directly measured by recording the time it takes to execute certain tasks or operations. Similarly, reliability can be measured as mean time between failures (MTBF) or mean time to failure (MTTF) or by simply by counting the number of faults over a period of time.

In response to run-time evaluation, the system is subjected to a series of modifications. As mentioned earlier, SAGA employs solutions to adapt a system at run-time. The solutions introduction is a twofold process. First, the design or architectural representation of the system is modified. Then, the architecture of the run-time instance is updated according to changes in the architectural representation. The run-time

architectural representation is basically a UML class diagram [36] formed using a small set of the class diagram notations. When a human architect introduces a solution, she makes a rough qualitative estimate about its implications for the managed quality attribute based on his or her past experience or knowledge. However, the machine needs some metrics to perform an evaluation of the architecture in order to estimate the effect of the solutions. Obviously, the evaluation is possible only for self-measureable qualities.

After modifying the architecture representation in the desired manner, the changes have to be reflected to the run-time. A true self-adaptive system must make such reflection without any human intervention. For an approach to be truly self-adaptive, the chosen solutions should be insertable and removable at run-time without any manual intervention. However, in practice concrete implementations of a number of standard solutions depend on application logic. These solutions require application-specific code which is hard to generate automatically. For example, two example solutions which depend on application logic are Adapter and Observer [11]. An adapter can adapt an adaptee in a number of ways depending on the application type. Similarly, an observer needs to know what to do when an event occurs.

It can be argued that for practical self-adaptation it is natural to ask the human designer to provide some information and even manually written application-specific code, as long as the actual adaptation functionality (that is, inserting and removing the solution at run-time) is taken care of automatically. This allows a much wider set of solutions than a fully autonomous approach. On the downside, this means that parts of the code of certain solutions that are available for self-adaptation must be preprogrammed, which in turn implies that those solutions can be used by a self-adaptive system only in certain types of contexts. The efforts required to manually introduce the missing code will vary from one solution to another and also depend on the application logic.

This work focuses on two self-measureable quality attributes, efficiency and reliability in the context of Java-based distributed systems. For efficiency, execution times of usage scenarios are recorded while for the reliability measurement, the number of faults over a period of time is counted. A set of formulas realizing the evaluation of these quality attributes will be discussed in Section 6.5.5. To enable self-adaptation, solutions were selected which do not depend on the application logic. In distributed settings, improvements in efficiency are achieved through re-allocation of components to different nodes which do not require any manual code input but only components' addresses or locations are changed in the new architecture. The reliability has been

nurtured using the Heartbeat solution. A reusable realization of the Heartbeat solution has been employed requiring no manual input from humans.

The case of the solutions requiring manual input will be presented using Adapter, Observer and Singleton [11] solutions. The three classic design patterns primarily improve modifiability, which makes them less suitable for self-adaptation, but useful for manual run-time adaptation. Some parts of the solutions are realized by the infrastructure to assist in their manual introduction as well as their reflection to the run-time is handled by the infrastructure. By introducing such a solution in a system at run-time, a human architect can create variation points that in turn enable functional changes without stopping the system.

6.4 Role-based Solution Definition

Let's assume the example system of Figure 6.2. It is a distributed safety monitoring system. The main purpose of the system is to secure different locations against fire, burglary and weapon presence related threats. In response to the threats, the system must call the police, fire brigade or an ambulance. The version of the system shown in Figure 6.2 is monitoring two locations using two distributed nodes, Node1 and Node2. The employed solutions are Heartbeat, Observer, Adapter and Singleton. There are also solutions related to components allocation and remote communication not directly visible in Figure 6.2 but will be discussed in the sequel.

In Figure 6.2, each solution is presented inside a rectangle, with ellipses denoting the roles. The type of each role is also included written as <type>. The child relationships between the roles are denoted with broken arrows marked as <<include>> while the rest of the arrows are dependencies. The distributed nodes of the software architecture are presented as 3D boxes. The thick lines denote the bindings between the roles and the software elements.

In Figure 6.2, the location monitored by Node1 has a glass breakage detecting sensor which is interfacing with the system through the GlassBreakSensor driver component. The sensor will help to detect any burglary attempts in the location. In the location looked over by Node2, a temperature sensor is installed which is connected to the node through the TemperatureSensor driver component. The temperature sensor will be used to identify any fire breakouts in the location. Note that the driver components cannot be moved from one node to another while changing the configuration of components. They indirectly represent the fixed resources of the nodes.

The BurglarThreatHandler and FireThreatHandler components collect the information from the sensors. If they detect a signature of fire threat or burglary, an

alarm is raised and police is contacted using BellControl and Dialer, respectively. The AdminUI component in Node1 controls a panel and a display from where the maintenance staff can test and observe the system.

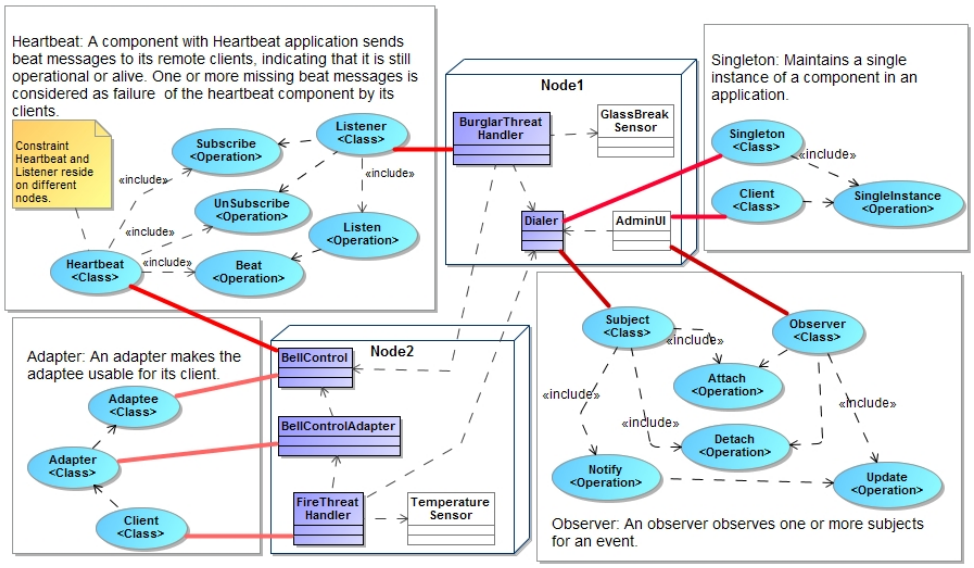


Figure 6.2. Architecture with solutions

The BellControlAdapter class plays the Adapter role as it adapts BellControl (Adaptee) for its client components. BellControl also has the Heartbeat role since BurglarThreatHandler (Listener) is listening to the heartbeat messages from BellControl. The Dialer class plays Singleton and Subject roles and the AdminUI class is acting as the observer of Dialer. The bindings for the <Operation> type roles are not shown in Figure 6.2 for clarity reasons. Chapter 2 (in Section 2.2) has introduced the Heartbeat, Adapter, Observer and Singleton solutions and their quality related implications.

As shown in Figure 6.2, the Heartbeat solution is composed of six roles. A component playing the Heartbeat role periodically sends out beat messages to the components playing the Listener (client) role. The Subscribe role is responsible for registering the listener for beats while the UnSubscribe role removes a listener from the register. The Beat role is responsible for generating the periodic beats while the Listen role is responsible for listening to the beats and to take appropriate actions when beats arrive or get lost. Only components with remote clients can take on the Heartbeat role. An application of the Heartbeat solution obviously has positive effect on the reliability of a distributed system. In Figure 6.2, BellControl plays the Heartbeat role and sends beats to its only remote client BurglarThreatHandler to assure a high level of reliability.

The high reliability demand on BellControl is due to the fact that the system is completely useless if it could not raise an alarm in an emergency situation.

The Adapter solution solves the incompatible interface problem effectively by three roles, Client, Adapter and Adaptee. The Adaptee role is played by the component with a new interface. A dedicated component, playing the Adapter role is to be inserted between the Client and the Adaptee role playing components. The Adapter component will provide the interface required by the clients, and will call the Adaptee component to get the work done. In Figure 6.2, it is assumed that the interface of BellControl changes and FireThreatHandler still requires the old interface.

The Observer solution has six roles with two top level roles Observer and Subject. The Subject role includes the Notify, Detach and Attach roles while the Observer role includes the Update role. The improvement in modifiability is achieved by decoupling the Observer and Subject role playing components. The solution imposes a standard engagement protocol between the subjects and observers. Any number of components can take on the Observer or Subject role as long as the protocol is respected, thus making the architecture more flexible for future changes. In the example system, it is required from AdminUI to report on the state of the telephone system by displaying its state on the installed display. As shown in Figure 6.2, AdminUI does so by observing the Dialer component, thereby realizing the Observer solution. AdminUI plays the Observer role while Dialer acts as a Subject. On updates from Dialer, AdminUI updates the display showing the state of the telephone system, i.e., calling police/fire brigade or being idle.

The Singleton solution has three roles, Client, Singleton and SingleInstance. All clients must request the single instance from the Singleton component before invoking any operation on the Singleton component. In Figure 6.2, Dialer has Singleton role as it represents and controls the single telephone system present in the system.

In Figure 6.2, FireThreatHandler and BurglarThreatHandler are communicating with Dialer and BellControl through the Message Dispatcher [12], respectively. As shown in Figure 6.3, the solution has five roles; Client Host, Client, Messaging, Supplier Host and Supplier. The Client Host and Supplier Host roles are played by the nodes hosting the Client and Supplier role playing components, respectively. Usually, the hosting nodes are different for Client and Supplier. The Client role playing component depends on the Messaging role playing service. The Messaging role playing element is expected to realize a messaging interface that can be used by Clients to communicate with Supplier role playing components over the network. In this work, a

realization of Java Messaging Service (JMS), OpenJMS [86] has been employed to serve this purpose.

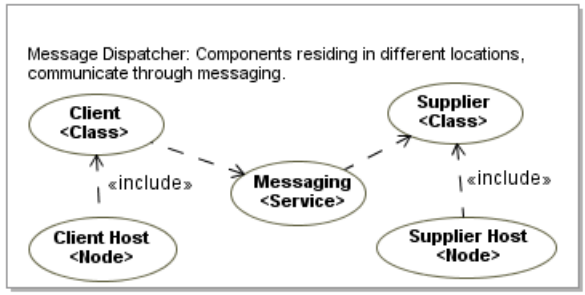


Figure 6.3. Message Dispatcher solution's roles

Two essential solutions providing guidelines for efficient allocation of components to nodes are not that apparent in Figure 6.2. The solutions and their roles are shown in Figure 6.4. The aim of both of the solutions is to reduce the number of remote communications which are usually costly, slow and error prone. A strong inter-dependency between two entities usually translates into increased communication. If the entities are located on different nodes, it will result in a high number of remote communications. Therefore, the solutions suggest localizing strongly inter-dependent entities (components and resources) thus localizing the highly active communications. A local communication is comparatively fast, cheap and secure which results in improved efficiency of the system. In addition, the solutions indirectly improve reliability due to the secure nature of local interactions.

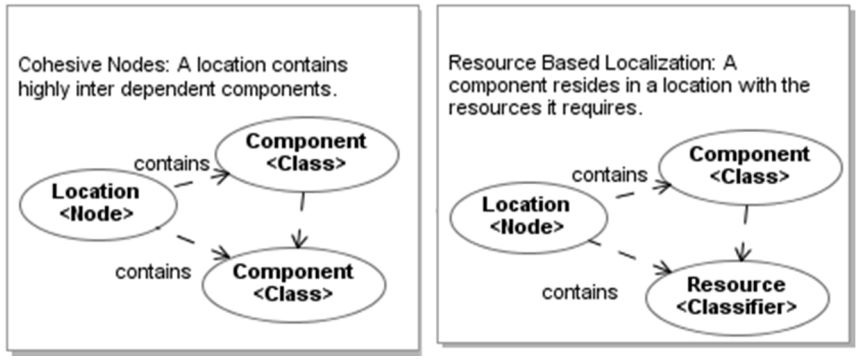


Figure 6.4. Component allocation solutions in distributed environment

In the solutions of Figure 6.4, there are in total three roles, Location, Component and Resource. The Resource-Based Localization solution in Figure 6.4 focuses on components and resources by allocating a component to a location where the resources

it requires are available. In Figure 6.2, FireThreatHandler resides in Node2 because it needs the temperature sensor (interfaced through the TemperatureSensor component) present in the location monitored by the node. Similarly, BurglarThreatHandler is placed in Node1 as it depends on the glass breakage detecting sensor located in the node. Similarly, the Cohesive Node solution motivates allocation of strongly inter-dependent components into the same node thereby reducing the number of highly active remote communications among the components. In Figure 6.2, BellControl resides on Node2 with FireThreatHandler while Dialer and BurglerThreatHandler are on Node1 because of the strong dependency between them.

6.5 SAGA Infrastructure

6.5.1 Overview

The SAGA (Self Architecting using Genetic Algorithms) infrastructure has four main components, the Architect Algorithm, Reflection and Monitoring layers and Javeleon, as shown in Figure 6.5. The Darwin environment presented in Chapter 4 has also been used in the implementation of SAGA. The Architect Algorithm is the genetic algorithm introduced in Chapter 3 embedded inside Darwin. Since Darwin was implemented in the Eclipse IDE [76], the SAGA infrastructure is also integrated with the IDE. The Eclipse IDE provides facilities to develop Java applications and SAGA enables the applications to perform self-adaptation at run-time. The code which is generated by the infrastructure can be directed to a source folder of a Java project in the Eclipse IDE. The code can also be edited manually by using the code editor provided by the IDE.

The Architect Algorithm synthesizes an input architecture, represented as a UML class diagram, and improves its quality by inserting and removing solutions from a predefined set of solutions. It does so to make the architecture more reliable and efficient for a given usage profile. In addition, a human architect can alter the proposal given by the algorithm using the UML class diagram editor [77], which is embedded inside the Darwin environment.

A UML profile has been employed for the representation of roles in UML class diagram based architecture. The profile provides stereotypes for the roles of the solutions. A role-based stereotype can be applied to a component which is playing the role. The Reflection layer depends on the stereotypes to correctly transform a UML class diagram with a set of roles belonging to different solutions into Java code. Any application-specific code for a role, if required, can also be manually added to the code.

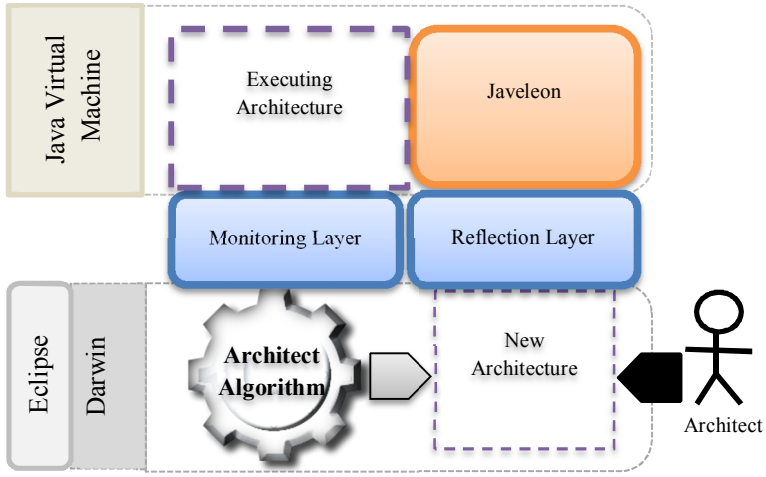


Figure 6.5. Self-Architecting enabling infrastructure

The code gets compiled and becomes part of the running instance of the application using Javeleon. The Monitoring layer watches the running system and logs the execution times, failure rates and usage frequencies of the operations. The next cycle of architectural improvements is triggered as the quality worsens beyond a preset level. In addition, a human architect can manually start the next improvement cycle whenever needed. The logged monitoring data is fed back to the Architect Algorithm to adapt the architecture accordingly.

6.5.2 Javeleon

In SAGA, Javeleon [35] has provided the flexibility of the run-time modifications for Java applications which is essential for run-time insertion and removal of solutions. At the time of development of SAGA (2011-2012), Javeleon (and its founders) was based at the Maersk McKinney Moller Institute of University of Southern Denmark. Thus, SAGA was developed with the early non-commercial versions of Javeleon obtained from its founders. In the beginning of year 2013, Javeleon was acquired by an established organization working in the same domain. Currently, SAGA's implementation is maintaining some elements of the distributed environment. The latest commercial version of Javeleon offers new improved features specifically included for distributed systems. These new features could free SAGA from taking care of the distributed environment.

The text in this section highlights the features of the early non-commercial version of Javeleon used in SAGA. The experiments reported in this chapter were performed

with Javeleon (version 3.1). Javeleon enables a Java application to be modified dynamically. New parent classes, attributes and methods can be introduced into Java classes while the application is running. Furthermore, components configuration can be altered at run-time without disturbing a running application. Likewise there is automatic support for some refactoring such as moving field in a hierarchy and moving method in a hierarchy.

Javeleon operates on top of the standard Java Virtual Machine without introducing new language constructs. At the basic level Javeleon is implemented as a Java agent intercepting class loading events in the JVM, transforming application classes to become update-enabled by manipulating the byte code of those classes.

An advance feature of Javeleon is the automatic initialization of newly added fields for already existing objects. The problem behind this feature is that an automatic and transparent dynamic updating system is not able to insert meaningful values to the newly added fields in general. The typical solution is to simply insert default values (like null), which often manifests as run-time error. For this purpose, Javeleon provides class-level assignments to newly added fields (instance or static). A class-level assignment basically means an assignment to the field made outside of a constructor.

There are other dynamic update facilities like DCEVM [71], JVolve [72] and JRebel [73], however, the kind of changes demanded by the architectural solutions are not fully supported by these tools. JVolve and JRebel lack capability to allow for run-time insertion or removal of parent classes and interfaces extended or realized by a Java class. That would fairly limit the scope of SAGA to a partial set of design solutions. DCEVM do support such features as well as addition of any new member to a class type. However, it does not allow removal of any elements from a class type. Javeleon is free of such issues and handles almost all modification scenarios with few exceptions.

Javeleon also provides a mechanism to circumvent the situations raised due to any obsolete assumptions [74]. For example, the updated code might have led to a slightly or in some cases entirely changed state for the system, if it were to be executed from the beginning. Javeleon does not re-execute the constructors of the updated classes because it will result in complete loss of the state. The state loss may jeopardize the overall stability and consistency of the system. Therefore, Javeleon offers State Transfer Functions (STF) to control such situations. For the updated classes, STFs can be written which will be executed for each object of the updated classes. The feature enables a developer to manually perform fine grained adaptations of the state in responses to new assumptions. If new assumptions demand re-initialization, that can also be accomplished through STFs when needed.

The insertion and removal of each solution presents a unique situation and therefore demands special means to handle the state of the affected objects. In this work, initialization/removal code for the roles taken/released by the system classes is hosted in the STFs thus enabling dynamic insertion and removal of the solutions.

The changes in the code can be reflected through automated or manual method. In the automated scheme, whenever a new version of a class file is available it is reflected to the run-time. In the manual method, reflection of the updates has to be invoked manually. In this work, automated method has been exercised.

6.5.3 Distributed Environment Setup

Javeleon was targeted for single machine environments at the time of development of SAGA. This has led us to use virtual nodes to simulate a distributed environment. Typically, an application should enter quiescent state after every reload. In that period the application is updated. The approach of Javeleon is fine grained; it updates each existing object when it is accessed for the first time after the update. In the current setup, all the nodes exist inside a single Java Virtual Machine (JVM) and therefore all the objects of the components reside on a single heap space. The components located inside a single node access each other directly while remote components use proxy components to communicate with each other. Another possible setup would be to use separate JVMs for each node. This setup will require migration of specifications of components from one JVM to another when a component is moved from one node to another. This capability was unfortunately not available in Javeleon at the time of development of SAGA.

A truly distributed environment like J2EE [81] lacks support for the run-time update of objects and therefore cannot be incorporated in SAGA at the moment. J2EE does provide hot deployment functionality, but it is intended mainly for development or testing phase. For traditional embedded system language technologies (say, C/C++), dynamic updating of components is even more challenging. In presence of actually distributed nodes, a new set of problems emerges, for example, components' objects need to be moved to their new locations. Additionally, definitions of the components have to be transferred to the new hosting nodes.

6.5.4 Monitoring Layer

The Monitoring layer is responsible for logging the number of failures for each component and execution times and usage frequencies of every operation in the system. In the experiment of Section 6.6, the failures are artificially introduced into selected

components. The monitoring requirement demanded insertion of logging related code into all components, specifically into their operations. For clean introduction and management of the logging code SAGA employs AspectJ [87]. Thanks to Javeleon, SAGA's monitoring aspect can be weaved with the existing or newly generated Java code on every reload at the run-time. The monitoring data is written into a text file and stored on the disk. The file is then consumed by the Architect Algorithm for next improvement cycle.

6.5.5 The Architect Algorithm

The Architect Algorithm introduces new architectural elements or modifies existing ones to maintain or improve system quality. Ideally, the algorithm should be a replacement for a human architect and should be able to make complex decisions. However, a complete automation of reasoning and decision making process is a truly challenging task.

During this work, the focus was on maintaining efficiency and reliability only. The Architect Algorithm is a variation of the genetic algorithm introduced in Chapter 3. The algorithm synthesizes and improves architecture based on the monitoring data. The monitoring data is used to update the support data (see Chapter 3) associated with each operation or hosting component in the input architecture. For this study, the support data has been extended with two new parameters criticality and vulnerability (explained in the sequel). The changes in support data will therefore affect the outcomes of a genetic synthesis.

Three mutations in the genetic algorithm have been used in this study. The first mutation changes the node of a component thereby enacting application or removal of Cohesive Nodes and Resource-Based Localization solutions. The mutation actually assigns a randomly selected node to the target component. The remaining two mutations are responsible for introducing and removing the Heartbeat solution to/from a component.

$$Maximize\ f(a) = f_e(a) + f_r(a) \tag{6.1}$$

The fitness function (f) is shown in equation (6.1). The fitness function (f) is composed of two sub-fitness functions concerning reliability (f_r) and efficiency (f_e). Among the parameters associated with each gene (operation), the call frequency and parameter size have a significant effect on the efficiency whereas the vulnerability and criticality values together play a vital role in the reliability assessment. A high call frequency value means the operation is frequently invoked by other client components.

The parameter size holds the information about the number of parameters that have to be passed to an operation.

The criticality and vulnerability of a component is derived from the individual criticality and vulnerability values of the operations it is holding. The criticality parameter indicates the importance of a component in a system: the higher it is, the more important is the component. The vulnerability parameter estimates the risk of failure of a component. It is derived from the failure rates of the components reported by the Monitoring layer. Therefore, an unsecured component with high criticality and vulnerability is considered unfavorable for reliability, and the genetic algorithm is expected to apply the Heartbeat solution on such components.

A Heartbeat application puts a strain on the system's efficiency. Using equation (6.2), the overall cost of all Heartbeat applications for each architecture can be estimated. In equation (6.2), *k* is the number of components with Heartbeat applications and *u* is the time required to prepare and send a single beat message. Furthermore, *n* represents the total number of clients listening to the beat messages while *d* is the time required to unpack and process a beat message. The frequency of heartbeat messages is represented by *i*. Note that *u*, *d* and *i* are constants, however, the values of *k* and *n* may change from one architecture to another.

$$Minimize\ f_{hb\_cost}(a) = (k\,u + n\,d)i \tag{6.2}$$

The efficiency sub-fitness with the help of equation (6.2) can be calculated as shown in equation (6.3). Let $x = 1 \ldots N$ and $y = 1 \ldots N$ where N is the total number of operations (or genes) in an architecture (a). The efficiency sub-fitness function punishes remote communications. The higher is the frequency (cf) of a communication the higher will be the penalty for it. The parameter size (p) also emphasizes the penalty as it will take relatively longer to deliver a large parameter data over the network. At the same time, the efficiency function rewards localized communications, i.e., intra-node communications. The reward is also emphasized by the frequencies (cf) of the communications and the parameter sizes (p) of the operations involved in the communications.

$$Maximize\ f_e(a) = \left[\sum_{x=1}^{N}\sum_{y=1}^{N} p_y cf_{xy} N_{xy}\right] - f_{hb\_cost}(a) \tag{6.3}$$

where p_y is the parameter size of operation *y* while cf_{xy} is the call frequency of *y* caused by operation *x*. Also, N_{xy} can be formally represented as

$$N_{xy} = \begin{cases} 1, & x.node = y.node \\ -1, & otherwise \end{cases}$$

The reliability measurement involves Heartbeat applications on components and the criticality and vulnerability values of their operations. A highly critical and/or vulnerable component with a Heartbeat application increases the reliability of the system. Equation (6.4) shows the reliability sub-fitness function. Let $s = 1 \dots M$ where M is the number of components with Heartbeat application. For each secured component, the reward is the product of its criticality (c) and vulnerability (v). A highly critical and/or vulnerable component will score high reward. On the contrary, a less critical and/or vulnerable component will contribute little to the overall reliability value. As a result, the evolutionary process will prefer architectures where highly critical and vulnerable components are secured with Heartbeat solutions.

$$Maximize\ f_r(a) = \sum_{s=1}^{M} c_s v_s \tag{6.4}$$

where c_s and v_s are the criticality and vulnerability of component s.

The genetic evolution continues until the last generation is reached. The best architecture of the last generation is then provided as the proposed architecture which is then transformed into UML class diagram notation. The Darwin environment provides a UML editor to view the diagram. The architecture is then passed over to the Reflection layer to generate and alter the existing code of the system to incorporate the modifications suggested in the new architecture.

6.5.6 Reflection Layer

The Reflection layer, also referred to as the JITA (Just in Time Architecture)-plugin [P4], is responsible for transforming a software architecture, represented as a UML class diagram, into an executable Java code. The new architecture proposed by the Architect Algorithm is taken as input by the layer, as shown in Figure 6.6. Furthermore, in order to generate the code, the layer needs the previous architecture as well as the code of the system. In addition, the layer is aware of a Solutions Repository which contains the reusable elements of different solutions. The previous architecture is required for the generation of the STFs. The STFs are essential for correct initialization of any newly inserted solutions and disposal of removed solutions. The code currently executing (will be referred as former code) is used to copy the application code to the new code.

The Reflection layer is an Eclipse [76] plugin. The plugin also exploits some of the features offered by other existing plugin of the Eclipse IDE. The UML class diagram editor of UML2Tool [77] plugin has been used to visualize and manipulate the architectures. Furthermore, Acceleo [78] plugin which is an implementation of Object Management Group's (OMG) MOFM2T [79] specification, was employed to convert

UML class diagram models into Java code. Abstract Syntax Tree (AST) [80] technology has also been utilized for code injection, removal and adaptation. The internals of the plugin are shown in Figure 6.6. The figure shows the flow of the conversion process as well as the components and underlying technologies involved.

The Reflection layer converts an architecture into code in three main steps. First elements of the new class diagram are converted into their Java based counterparts. For example, UML classes and their members are translated into Java classes, attributes and methods. For each dependency, a member variable of supplier type is inserted into the client class. The supplier type member variable is also annotated with the information about the publicly accessible methods the supplier is offering. The information will be later consumed by the AST parsers to make fine grained changes in the code. This all happens in the General Code Generator part of the Class Processor module, as shown in Figure 6.6. The Class Processor is further hosted inside the Model Processor. The Model Processor has been implemented using Acceleo.

The second step is realized by the Role Code Generator module which is aware of the semantics of the roles and solutions. The module has access to the former architecture, which is also a UML class diagram, and to the Solutions Repository. The repository contains reusable elements of the employed solutions. For example, the classes realizing the Heartbeat solution are reusable and can be added as parents (as `extends`) to application classes. Also, proxy classes are generated based on the new configuration. Moreover, a general realization of the Observer and Subject roles can be added to any component through inheritance relationship (as `extends`). Any compulsory interfaces can be brought in dynamically as `implements` relationships. Furthermore, operations are generated within the classes if they are required by the role (e.g., operation level roles) the hosting component is engaged in. The former architecture is needed to correctly generate the code for STFs that will initialize the new role or deactivate the lost role for a component.

At this point, the code has all the elements related to the solutions; however, application-specific code is yet to be moved in. Thus, in the final stage a set of AST parsers accomplishes this by moving in the class attributes and method bodies from the former code, as shown in Figure 6.6. In addition, the former code is adapted to work correctly under the new configuration by making fine-grained modifications using the annotations inserted during the first stage.

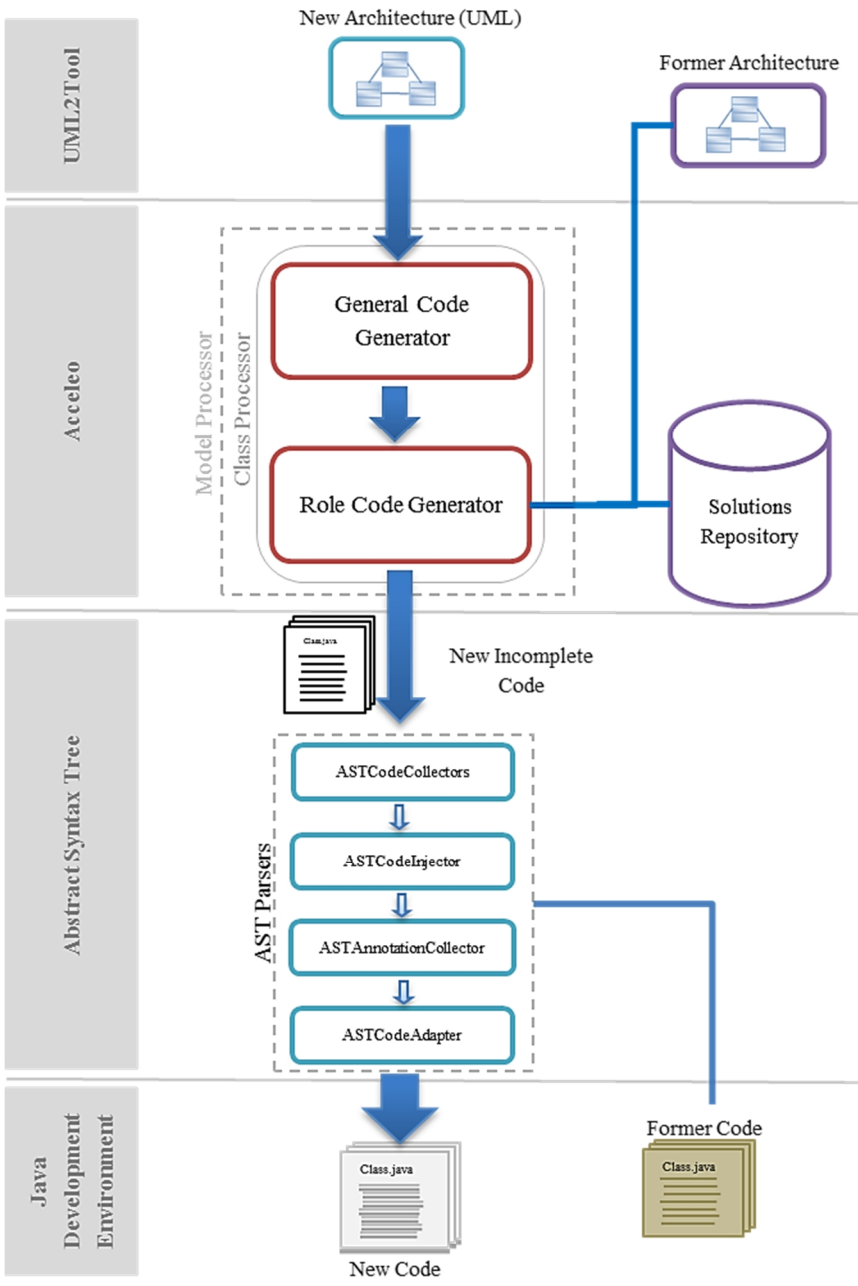


Figure 6.6 The Reflection Layer/JITA-plugin

The first parser, AST Code Collector, collects the application code from the former implementation of the system. The second AST Code Injector parser then injects the collected code into the newly generated code. The third parser, AST Annotation

Collector, sweeps through all the annotations in the new code that have been inserted in by the Class Processor module. The information is then used by the final AST Code Adapter parser to update the application code. The code is updated so that the newly generated suppliers' type members are used to accesses their public members (attributes and methods). At last, the generated code contains the new solutions and configuration of components.

The solutions that are independent of application logic are completely realized at this point. However, for the solutions which are dependent on application logic, the layer only generates the structural elements (operations, classes, etc.). The application specific codes for the elements have to be filled in manually. The code is moved to the Java application's source folder in its Eclipse project while the former code is transferred to a backup folder. The generated new code can be reviewed and updated manually (if needed) in the Java code editors provided by the Eclipse IDE. The code is reflected to the run-time by Javeleon as it is programmatically or manually compiled.

6.6 Experiments and Evaluation

6.6.1 Goals

In the experiment, the goal is to study the effectiveness of the presented approach in both manual and self-adaptive settings using an example system. The manual adaptation part studies how plausible modification scenarios can be carried out using SAGA for manual run-time maintenance. In the self-architecting part, the goal is to study how the changing usage environment affects the genetic evolution of the software architecture. The study also aims at exploring how coherently the modifications in the architecture fit the changing environment, and what effect the variations in the design have on the run-time efficiency and reliability of the system. Improvements in the quality are expected; however, the degree of enhancements is also a subject of the study. Furthermore, the time it takes to produce new improved architectures is also to be studied.

6.6.2 Experiment Setup

A controlled experiment [104] under laboratory settings involving both dynamic analysis and simulation using a representative example distributed system in circumstances that resemble real life will be performed. The example system will be under execution during the experiment and changes will be reflected at run-time to the system. The manual run-time maintenance part focuses on qualitative analysis of increased productivity during maintenance due to the tools and assistance offered by the

infrastructure. The system will be modified in response to two newly emerged requirements calling for introduction of solutions like Adapter, Singleton and Observer.

The self-adaptive part involves both quantitative and qualitative analysis. The running system will be subjected to two usage profiles (or scenarios) and collected data will be used to analyze the improvements in the system's quality. In order to test SAGA's capability of securing failing components with Heartbeat, faults will be artificially induce into one of the components.

The genetic algorithm will perform simulated evolution of software architectures to produce newly improved architectures. Due to the random nature of genetic algorithms, the null and all subsequent architectures will be evolved 100 times to establish the fact that the proposals are not accidental [82][83]. The proposals will be examined and will be reflected to the run-time to observe and measure the actual improvement in quality.

The changes in the efficiency will be presented quantitatively by measuring and comparing the execution times of the usage profiles before and after the updates. However, in case of reliability, it can never be known when a component might fail in real life. Therefore, once again the errors will be simulated in the secured components to see how the Heartbeat applications translate into the system behavior and observations will be reported.

6.6.3 Target System Selection

Consider a typical embedded system, consisting of computational nodes containing device drivers and other components of the system. The system is connected to various sensors and actuators. The components communicate with each other either using direct calls (if the components are in the same node) or using some messaging infrastructure (if the components are in different nodes). The components are allocated in different nodes according to the architecture. Obviously, since message-based communication is much less efficient than direct calls, it is generally preferable to allocate frequently communicating components in the same node, unless there are particular reasons not to do it. Similarly, some components are regarded vulnerable and associated with a Heartbeat, so that other components can observe the aliveness of that component. All these decisions are part of the architecture, and preserved during the operation of the system.

However, what if the system can be used in rather different environments, and the environment changes in such a way that the assumptions that were the basis of the architecture do not hold any more? For example, imagine that the communication frequencies between components change in a way that cannot be anticipated, and even

if it could, it would be difficult to find an allocation strategy that works optimally in all environments. Similarly, the optimal assignment of Heartbeat solutions, ensuring a reasonable balance between reliability and efficiency, may change in a new operation environment.

The target system used in the experiment is developed along these lines, as depicted in Figure 6.7. The system is similar to the one discussed in Section 6.4. In the figure, the lines connecting the components indicate communications or dependencies among the components. The system emulates a distributed safety monitoring system, capable of securing a house, shop, warehouse etc. against fire, burglary and presence of arms. The physical system consists of a certain number of nodes, here it is assumed that the system is delivered with four nodes. The system reacts to the threats by raising an alarm and calls the authorities (police, ambulance or fire brigade). The vibration, glass break and presence sensors are used to detect burglaries while temperature and smoke sensors are installed to detect fire breakouts. The cameras are capable of detecting fire arms.

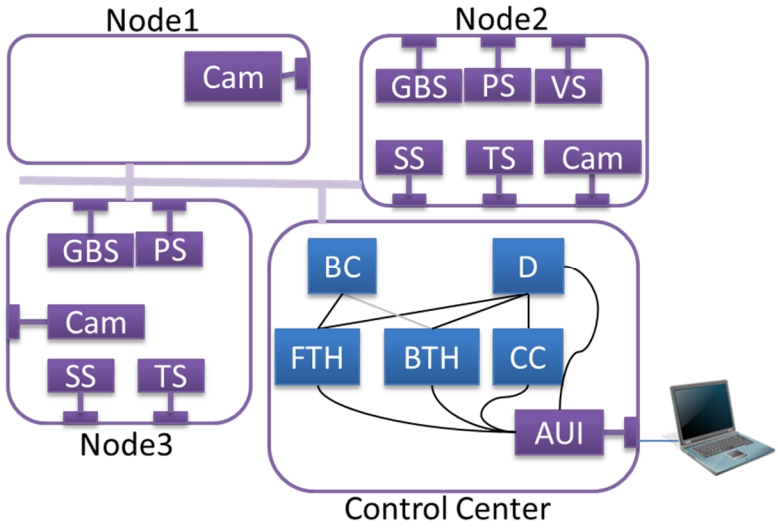


Figure 6.7. Generic monitoring system with default configuration

The driver components GlassBreakSensor(GBS), PresenseSensor (PS), VibrationSensor (VS), SmokeSensor (SS) and TemperatureSensor (TS) interface with glass break, presence, vibration, smoke and temperature sensors, respectively. Furthermore, the Camera (Cam) driver components are interfacing with the cameras installed on the nodes. According to monitoring requirements for a location, different nodes can be chosen to serve the purpose. For example, a location requiring only camera can be secured with Node1. Furthermore, different sensors and therefore the

driver components can be ignored if they are not required by a usage profile in a location. The system has been implemented as a Java application using virtual nodes where OpenJMS provides the messaging infrastructure.

The example system exhibits typical characteristics of an embedded control system with both free and fixed driver components on the nodes communicating through a messaging interface. A free component can be moved from one node to another. A fixed component on the other hand cannot be moved because of its dependence on a physical resource located in the hosting node. Moreover, like a typical safety system, it needs to be operational 24/7 securing a building or facility thus requiring non-disruptive maintenance. Moreover, the system has been designed and coded without any consideration for future changes. However, the flexibility offered by the free components is there which is essential for reducing the amount of expensive remote communications (to improve efficiency) through reconfiguration. The system also includes a critical component without which the system is useless (explained later). As far the presence of vulnerable components is concerned, any component can become vulnerable during the operation of the system.

As shown in Figure 6.7, Node1 supports a camera while the Control Center node can be used to test the system through the AdminUI (AUI) driver component. The AUI driver component allows connecting a display to the system for running the test sequences. The remaining two nodes, Node2 and Node3, have almost all the sensors installed. Since the driver components depend on physical sensors, they cannot be moved from one node to another when applying self-architecting to improve the system quality. Similarly, the AUI component is connected to a physical display and therefore it cannot be re-located. The rest of the components, however, can be re-located to different nodes.

The GBS, PS and VS driver components on the nodes report their findings to the BurglarThreatHandler (BTH) component which then takes appropriate actions. If the signature of a burglary is detected in the data, the BTH component uses the BellControl (BC) component to raise an alarm and calls the police using the Dialer (D) component. Similarly, the SS and TS driver components on the nodes send their information to the FireThreatHandler (FTH) component regarding the changes in the temperature values and smoke levels. Once a location's temperature or smoke level surpasses a safe point, the FTH component raises an alarm and contacts the authorities (police, fire brigade and ambulance) using the BC and D components, respectively. The data from the camera drivers (Cam) is handled by the CameraControl (CC) component which contacts the police if it detects presence of a weapon in the monitored location.

6.6.4 Manual Manipulation of Architecture

This section demonstrates manual introduction of the Adapter, Singleton and Observer solutions to maintain the system against changing requirements using the tools provided by the infrastructure. The tools are integrated into Eclipse IDE, a widely used development environment for Java applications. The system with the default configuration of Figure 6.7 has been executed. Then, a burglary through Node3 has been simulated in the system. The BurglarThreatHandler (BTH) detects the threat, rings the bell and calls the police as expected.

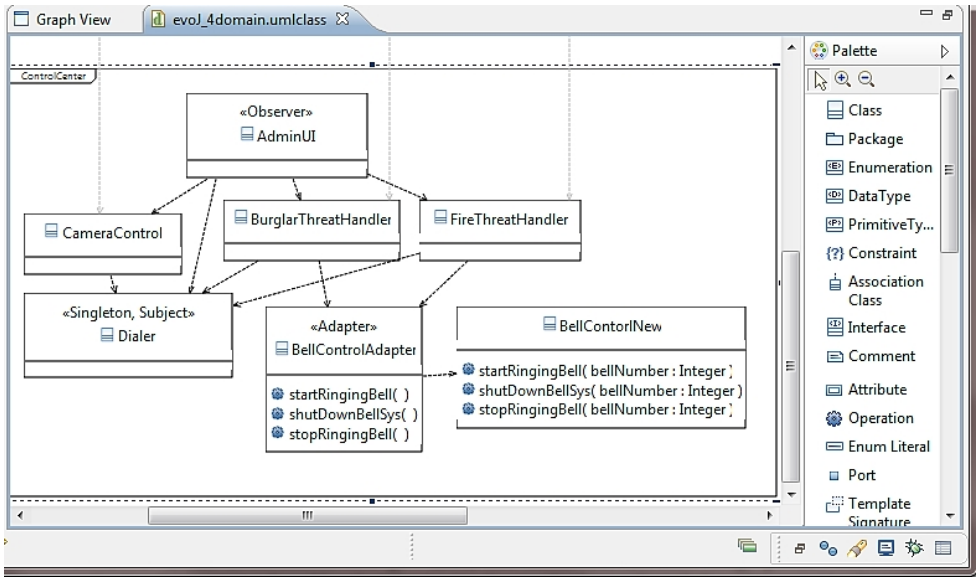


Figure 6.8. Application of Adapter, Singleton and Observer Solutions in the UML editor

Let us assume that two new requirements emerged. According to the first requirement, instead of one bell the system should be able to support arbitrary number of bells. Therefore, a new bell controller capable of controlling all the bells is needed. The new controller, BellControlNew, will now handle the bells; however, it has a slightly different interface compared to the previous controller BellControl. This makes the new controller incompatible with the rest of the system. The problem can be fixed using the Adapter solution.

In Figure 6.8, the UML class diagram editor of Darwin containing the architecture of the system is shown. In the figure, due to the space limitation, only the components

residing inside the Control Center node are shown. From the palette, an architect can add new elements to the architecture. Furthermore, different role-based stereotypes can be applied in the editor to introduce different solutions. Some roles have to be explicitly applied through the use of the stereotypes while others are automatically detected by the infrastructure.

The infrastructure will produce reusable parts of the used solutions while leaving the application-specific code for the architect or developer of the system. The UML class diagram editor assists the architect to introduce the elements of the solutions at the abstract level without going into code. Also, the partial generation of solutions saves significant time otherwise consumed in re-writing the reusable parts of the used solutions.

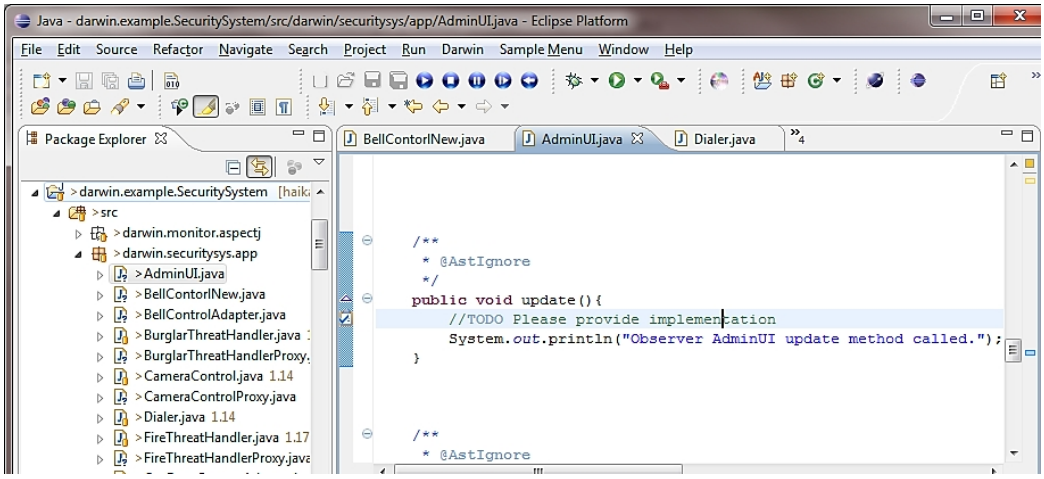


Figure 6.9. Editing generated code

For the Adapter solution, a new adapter class has been introduced from the palette and named BellControlAdapter. The methods in the adapter class are introduced so that the class can offer an acceptable interface to the rest of the system. The stereotype for the Adapter role has also been applied on the BellControlAdapter class, as shown in Figure 6.8. The other two roles, Adaptee and Client, are not required to be explicitly mentioned in the design. An Adapter role playing class will always be client of an Adaptee role playing class and therefore the infrastructure will treat the BellControlNew class as Adaptee. SAGA generates the skeleton for the new adapter and BellControlNew (Adaptee). The bodies of the methods have been filled in manually.

The second requirement demands displaying of phone calls on the display attached to the AdminUI driver. One way to realize the requirement is to notify the AdminUI component whenever Dialer makes a call. Thus, the Observer solution can be used along with the Singleton solution to implement the requirement. The Dialer component is playing the Subject and Singleton roles in the modified architecture. The Singleton role was essential, so that all components share a common Dialer thereby enabling Dialer to notify its observers whenever it is used by any component in the system. The typical realization of the Singleton solution is simple, reusable and does not involve any application-specific code. Therefore, the code realizing the Singleton solution is completely handled by SAGA and requires no manual input.

For the Observer solution, the infrastructure handles all the changes except the code for the Update role playing method as well as the notification triggering code in the Subject role playing class. The infrastructure by default produces an empty *update()* method in the Observer role playing class, however, any other method can also take on the Update role. An implementation for the Attach and Detach roles are incorporated through the parent classes, hosting the methods playing the roles. The parent classes are loaded from the Solutions Repository which contains the reusable parts of all the solutions considered in this work. As mentioned earlier, the infrastructure is integrated with the Eclipse IDE, therefore, generated Java code files can be opened for editing, as shown in Figure 6.9.

Note that the system is under execution with the architecture of Figure 6.7. After changing the architecture, the code has been generated and directed it to the source package (darwin.securitysys.app) of the example Java application's project in Eclipse, as shown in Figure 6.9. From the package, files were opened and the code was entered by hand for BellControlAdapter's operations. Furthermore, code is added to Dialer to trigger the Update role playing method in AdminUI. Also, the Update roles playing method in AdminUI were filled in to display a message when it receives an update from its subject Dialer. The project was then compiled to reflect the changes to the run-time. For the same burglary simulation, the system started to use the BellControlAdapter and BellControlNew components to ring all the bells. Also, the Dialer has notified its observers, here AdminUI, whenever it made a call to the police.

This part of the experiment has demonstrated that even manual adaptation with partial automation offers significant benefits. The solutions can be introduced and removed at the design level without losing the view of the big picture of the system. The executing system was never stopped during the whole experiment and the modifications were reflected right away to the run-time. Many areas with the requirement of continuously running systems can benefit from this kind of approach.

6.6.5 Usage Profiles based Self-Architecting

6.6.5.1 First Usage Profile

For the usage profiles, let us assume that the system has been employed in exhibition center settings. The exhibition hall has four locations, outdoor concert hall, gallery, conference hall and a control center. The requirements in the first usage profile require a camera in the outdoor location, fire detection in the conference hall while in the gallery protection against burglaries is needed. As shown in Figure 6.10 (left), Node1 is securing the outdoor concert hall, Node2 is looking over the gallery with the burglary sensors enabled. Furthermore, Node3 is monitoring the conference hall with fire detecting sensors enabled.

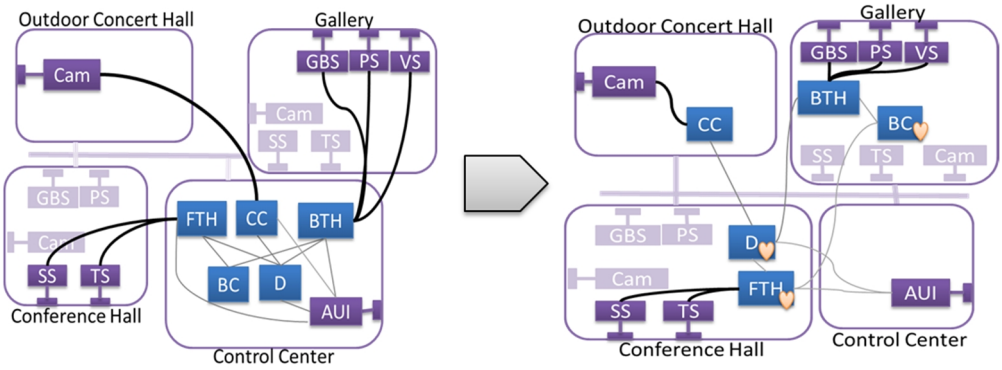


Figure 6.10. Proposal for the first usage profile

As a result of the configuration of the sensors, the burglary detecting sensors (GBS, PS and VS) of the Gallery node will actively use BTH which is in default configuration located on the Control Center node, as shown in Figure 6.10 (left). In the figure, the thick connecting lines indicate strongly active communications. Similarly, the fire detecting SS and TS sensors of the Conference Hall node are actively using FTH residing on the Control Center node. Furthermore, Cam driver on the Outdoor Concert Hall node is interacting with the CC component. To observe, how SAGA applies Heartbeat on vulnerable components, the FTH component will be artificially crashed to increase its vulnerability. The two highly critical components in the system are BC and D as without them the system will not be able to notify anybody or authorities to save the people or property in threatening situations. Therefore, high criticality values are associated with BC and D in the architecture.

The architecture of Figure 6.10 (left) in execution, the first usage profile has been simulated which took 558 ms. As a result, monitoring data was produced by the

Monitoring layer which was then fed to the genetic algorithm. For the first usage profile, the average fitness, reliability and efficiency graphs from 100 runs are shown in Figure 6.11. The graphs show a significant improvement in both of the quality attributes and the overall fitness during the evolution. The genetic algorithm consistently proposed the architecture shown in Figure 6.10 (right).

In the genetic algorithm, the runs took 24 seconds on average to complete. There were 50 generations in each run while each generation had population of 50 individuals. The number of generations has been chosen based on the development of the fitness curves. In the study, the fitness curves have reached their peak values before 50 generations and then stayed there. Therefore, evolving architecture beyond 50 generations brings no further benefit. Also note that the selection of the number of generations and population size will vary depending on the size of the system and search space.

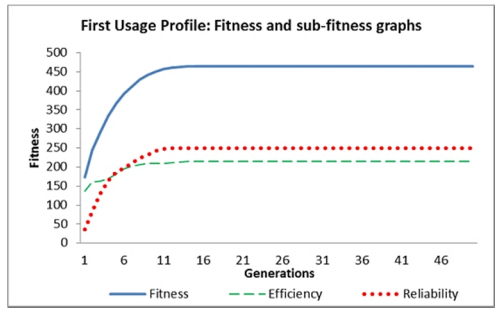


Figure 6.11. Fitness and sub-fitness graphs for the first usage profile

In the proposed architecture of Figure 6.10 (right), the vulnerable component FTH has been protected with a Heartbeat (♥). Furthermore, critical components D and BC are also secured with the Heartbeat applications. The Heartbeat applications on the failing and critical components have improved the overall reliability of the system. As far the allocation is concerned, the architecture has been fine tuned for the usage profile. The BTH component has been moved to the Gallery node as the GBS, PS and VS sensors have a strong dependence on BTH. Similarly, CC and FTH have been allocated to the Outdoor Concert Hall and Conference Hall nodes, respectively. The configuration will reduce the excessive remote communications between the sensors and the free components, thus improving the efficiency of the system. The proposed architecture was reflected to the run-time. The efficiency has been improved by 50 % (279 ms vs 558 ms) for the first usage profile. Even though the Heartbeat applications have introduced some efficiency overheads, however, it will help avoid system crashes if the same kind of failure behavior continues.

After the update, the broadcast of heartbeat messages from the components with Heartbeat applications has also been witnessed. On re-introduction of the artificial faults in the FTH component, its clients were able to notice the failure when three consecutive heartbeat messages were absent. In this work, no stand on the recovery strategies has been taken, the failed components can be re-started or an alert can be issued to the system administrator to take appropriate actions.

6.6.5.2 Second Usage Profile

In the second usage profile, let us assume that during the winter times the outdoor concert hall is rarely used. Also, it is very seldom that an international conference is held in the conference hall of the exhibition center during the winter time. The only actively used location is the gallery and during the winter period more security is required in that location. In addition to security against burglaries, fire alarm and camera are also needed for tight security of the gallery. As shown in Figure 6.12 (left), the fire detecting sensors (SS and TS) and the camera (Cam) in the Gallery node have been enabled now and they are communicating intensively with the FTH and CC components. The FTH component is residing on the Conference Hall node while the CC component is hosted by the Outdoor Concert Hall node. The sensors on the other nodes are comparatively less active.

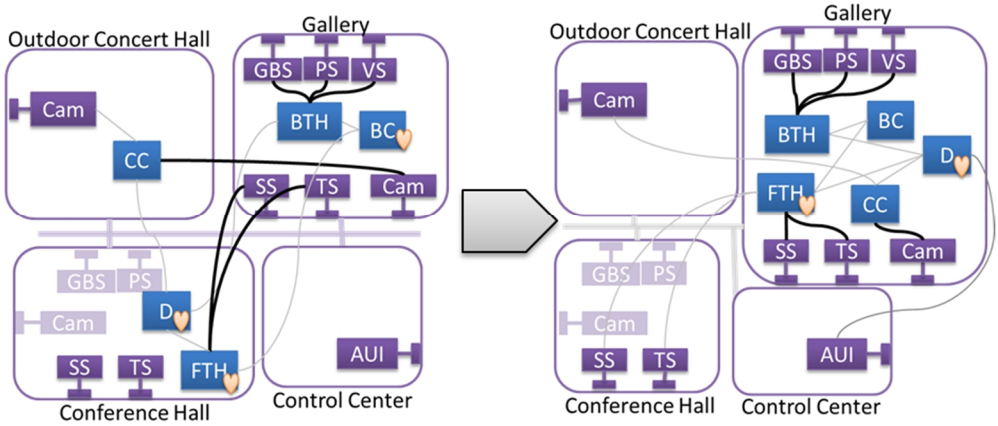


Figure 6.12. Proposal for the second usage profile

With the architecture of Figure 6.12 (left) in execution, the second usage profile has been simulated which took 280 ms. The collected monitoring data has been provided to the genetic algorithm. The genetic algorithm again significantly improved the quality attributes as depicted by the fitness and sub-fitness graphs shown in Figure 6.13. The graphs are again average graphs of 100 runs where each run on average took 23 seconds

to execute with the same number of generations and population size as used for the previous usage profile.

The proposal from the runs is shown in Figure 6.12 (right). The architecture has been tuned for the usage profile by moving all the components to the Gallery node. This resulted in improved efficiency due to the localization of highly inter-dependent components in the Gallery node. The FTH and CC components are moved to the Gallery node to avoid the time consuming remote communications with the sensors located on the node.

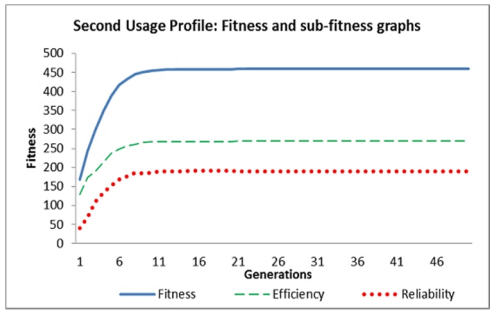


Figure 6.13 Fitness and sub-fitness graphs for the second usage profile

In the second usage profile, FTH was not crashing; however, it still has a Heartbeat application. On every reload, a non-crashing component's vulnerability is reduced by the infrastructure. FTH's vulnerability has also been reduced; however, it was nevertheless good enough to secure a Heartbeat for FTH. The highly critical component D has also been secured, however, BC was left out as it does not have any remote clients any more. Both of the clients FTH and BTH of BC reside in the Gallery node too.

Table 6.1 Execution times for the usage profiles

Architectures	**Execution Times for The Profiles (ms)**	
	First	**Second**
Initial	558.89	414.78
First Proposal	**279.87**	280.31
Second Proposal	286.49	**59.07**

The securing of FTH and D components with Heartbeats has a positive effect on the system reliability. After the reflection of the proposed architecture to the run-time, 79% (280 ms vs 59 ms) reduction in the execution time in comparison to the architecture of the first proposal has been noticed, as shown in Table 6.1. Furthermore, it is observed

that BC had stopped releasing the beats while FTH and D had kept on sending the beat messages.

In the experiment, note that the genetic algorithm has fine-tuned each input architecture for a specific usage profile. As shown in Table 6.1, both usage profiles performed relatively poor with the initial architecture and also with the architectures not intended for the usage profiles. An architecture intended for one usage profile could not guarantee improvement for any other. Moreover, a strong relationship between the demands put forward by the changing usage profiles and the modifications proposed by the genetic algorithm have been seen. The changes were logical in the sense that a human architect probably would suggest similar modifications in such situations.

PART IV – Closing

This part presents the related work and revisits the research questions. The limitations of this thesis work have also been discussed in this part. The thesis is concluded with final remarks from the author.

7 Related Research

7.1 Software Development Automation

Amoui et al. [14] use genetic algorithms to improve reusability of software architectures represented using UML model. They aim at finding sequence of patterns transformations leading to improved reusability. They used design patterns from Gamma et al. [11] in their genetic algorithm. The genetic algorithm presented in this thesis work improves multiple conflicting qualities of software architectures, instead of only one.

In the work of Simons and Parmee [19][20], the input to their genetic algorithm is not an architecture but requirements (use cases) where actions are associated with operations and data with attributes. The concept of class serves as the way to group the data and actions. O'Keeffe and Ó Cinnéide [6] have applied simulated annealing to restructure class hierarchy and developed a tool named Dearthóir. Their algorithm moves around operations in classes to satisfy a conflicting set of goals. The aim is to avoid code duplications and rejection of methods as well to ensure that the super classes are abstract where appropriate. They have continued their work by applying their approach to object oriented programs [9][18]. They have implemented another tool named CODe-Imp for this purpose. CODe-Imp transforms programs so they can be more aligned with the provided quality model. In this thesis work, however, the operations are already grouped into classes in the null architecture. Also, the developed genetic algorithm focuses on high level quality attributes and uses solutions to improve them instead of refactoring the architecture. Also, the metrics used in Dearthóir and CODe-Imp are comparatively simpler than used in this thesis work.

The work of Seng et al. [51][52] focuses on distribution of classes into sub-systems using genetic algorithms. Similarly, the Bunch tool by Mancoridis et al. [16] employs a genetic algorithm to group modules into clusters or sub-systems based on the coupling among them. The Bunch tool processes the source code of a system and constructs a

module dependency graph (MDG). Their genetic algorithm encourages the proposals leading to less inter-cluster and more intra-cluster connections. Mitchell et al. [17] have built their ARIS tool on top of the Bunch tool. The ARIS tool enables software developers to set rules on how the sub-systems should relate to each other. The ARIS tool provides assistance in identifying the violations of such rules. In comparison, the genetic architectural synthesis approach of this thesis work operates at the level of classes, operations and solutions instead of sub-systems, modules or source code. Furthermore, the developed genetic algorithm is concerned with engineering from scratch instead of re-engineering an existing matured system. At the same time, there is no apparent limitation in applying the presented genetic algorithm for re-engineering as well. The clustering into sub-systems is a higher level problem while the genetic algorithm of this thesis work is operating at the level of classes with aim of improving high level quality attributes.

Di Penta et al. [53] developed a toolkit, software renovation framework (SRF), to refactor systems' codes to get rid of unused objects and code clones as well as to reduce the size of libraries through refactoring. The SRF framework creates a dependency graph of the artifacts and then applies genetic and hill climbing algorithms as well as takes into account the feedback from the developers. The SRF framework focuses on re-engineering and operates at a detailed level than the genetic algorithm of this thesis work. Bowman et al. [50]'s multi-objective genetic algorithm (MOGA) optimizes an existing architecture when it violates pre-defined design constraints. In this thesis work, however, the null architecture is designed from scratch.

The work of Yu et al. [21] aims at reducing inter-DM (Decision Maker) communication which is similar to the approach presented in Chapter 5. They use a nested genetic algorithm (NGA) for tasks grouping and resource allocation. In one stage, their algorithm assigns resources (platforms: military equipment) to tasks while in another stage the tasks are grouped into DM (Decision Maker) cells. The aim is to minimize the platforms movements during tasks execution while at the same time the tasks allocation should lead to minimized inter-DM communication or coordination.

Like Darwin, the ArchE [54][55] tool enables an architect to specify the quality and functional requirements along with properties (support data) essential for good estimation of the quality. If a legacy design is available that can also be given as input to the tool. The tool identifies dependencies among the requirements and then presents the architect with initial architecture of the system and series of suggestions. If the architect accepts a suggestion, the architecture is revised by the tool. In contrast to Darwin, ArchE uses a deterministic approach and involves manual input from the architect during the architecting process.

7.2 Run-time and Self-Adaptive Maintenance

The SASSY (Self-Architecting Software Systems) of [48] is similar in nature to SAGA. The SASSY framework enables pattern-based self-adaption for SOA(Service Oriented Architecture)-based systems. In [85] QoS (Quality of Service) patterns and hill climbing algorithm have been used in SASSY to optimize availability and response time of SOA-based systems. The optimization process takes into account the collected run-time monitoring information. In contrast, SAGA's current implementation is for distributed systems, however, it can be extended to other kinds of systems. Furthermore, SAGA employs a genetic algorithm for generation of improved architectures whereas SASSY uses a hill climbing algorithm. Also, the flexibility to re-configure and move services to different nodes is inherently built into the SOA paradigm. The paradigm offers flexibility that is absent from the traditional static and strongly typed languages like Java which was focus of SAGA.

The approaches in [90] and [91] preserve reliability by replicating components on different nodes in distributed settings. Similarly, the MARS approach [92] takes into account the reliabilities of nodes when placing an object in the nodes to maintain overall reliability of a system. However, fault fixing requires a restart of the system and may also involve some delay in case of permanent faults. Another replication-based approach in [93] takes care of performance alongside with reliability. Their architect algorithm takes into account the reliabilities of nodes while finding suitable nodes for replicas. The genetic algorithm in SAGA, however, considers communication (usage frequencies) between the components that can influence their place in the system.

Aleti et al. [22] have developed ArcheOptrix tool which generates optimal allocation of software components to hardware platform in distributed systems. The tool employs a genetic algorithm and a Pareto optimal fitness function [23]. Their approach aims at optimizing the Data Transfer Reliability and Communication Overhead. ArcheOptrix is a design time tool while SAGA allows reflecting of a design to run-time.

Chameleon environment [88] provides an adaptable fault-tolerant platform for different applications. Chameleon only aims at improving the reliability by distributing the components. Furthermore, Chameleon itself is dynamically adaptable, however, the user applications are not and require a restart after fault detection. SAGA's restart mechanism on the other hand is fine grained as only the failed component can be re-started alone. Moreover, Chameleon relies on a set of strategies stored in a registry whereas SAGA does not require such pre planning, thanks to the genetic algorithm. An approach similar to Chameleon is the FRIENDS [89] system, which provides a meta-

level architecture containing libraries of meta-objects for fault-tolerance. In FRIENDS, however, the developer has to manually include the required libraries.

In manual reliability maintenance settings, Watchdog and Heartbeat solutions have been employed for reliability maintenance in [94]. The failure prediction through Palladio Component model is discussed by Brosch et al. [64]. They improve the reliability by reconfiguring and replicating the components. Their metrics are different than used in SAGA, however, the idea of reliability of a component as a sum of associated actions' failure probabilities, is similar to SAGA. Similarly, Martens et al. [95] have used Palladio model in their work to resolve deployment related issues. Their genetic algorithm considers performance, reliability and cost when improving a system design. Their approach still stays at the design level whereas SAGA generates executable code from the design to observe the actual effect of the architectural changes.

Kramer et al. [3] have presented an approach based on dynamically reconfigurable components. However, the reconfiguration has to be done manually. White et al. [8] have worked on self-organizing autonomic components. They designed special patterns which enable components to re-organize themselves to form functional systems. Gomma et al. [38] have introduced reconfiguration patterns for the run-time modification of software systems, however, a modification has to be initiated manually. In [37] a reusable Rainbow framework providing architectural styles has been developed. Each of the architectural styles allows a predefined set of modifications and strategies associated with each of the modifications. The strategies list actions to be carried out in order to successfully apply the corresponding modification at run-time.

In the patterns-based approaches, the pattern or solution oriented view on system evolution is in line with SAGA's approach, however, some of the methods involve humans in the loop while others are deterministic and confined to a pre-defined set of modifications and strategies. SAGA also has humans in the loop for maintenance of non-automatable properties. However, for self-maintainable properties, SAGA relies on architectural expertise, captured inside fitness functions. This allows for automatically addressing possibly un-anticipated future maintenance needs.

Many studies like [4], [70] and [84] in the early 90s have attempted to incorporate design patterns in the system without any human intervention, however, in an offline manner. The aim was to efficiently introduce patterns in the software designs and to automatically generate the code from the descriptions of the patterns. There was no consideration for quality attributes or for run-time maintenance.

8 Introduction to the Included Publications

The work presented in the included publications has been made possible with contributions from the author, supervisor and fellow researchers. An introduction and roles of the contributing authors for each included publication are given below.

P1. Outi Räihä, Hadaytullah, Kai Koskimies and Erkki Mäkinen. Synthesizing Architecture from Requirements: A Genetic Approach. Relating Software Requirements and Architecture (eds. P. Avgeriou, J. Grundy, J.G. Hall, P. Lago, I. Mistrik), Chapter 18, Springer 2011, pp. 307-331.

This study applies genetic algorithms to evolve software architectures into modifiable and efficient architectures with reduced architectural complexity. The genetic algorithm requires basic functional decomposition of a system as input. It breeds the architecture using mutations, crossover and the fitness function into a good quality proposal. A set of mutations apply or remove solutions obtained from the solutions repository containing design patterns [11] and architectural styles [12]. The architecture with the highest fitness value at the end of the evolution is the proposed architecture. An empirical study is also part of the contribution presenting comparative analysis of the algorithm and man-made architectures.

Outi Räihä was the main contributor of the work. The author of this thesis has originated the idea for integrating the genetic architectural synthesis approach with the typical UML based software design process. The process starts from abstract requirements in a use case diagram and then the use cases are refined into message sequences in a sequence diagram until the first basic architecture as a UML class diagram is achieved. Kai Koskimies and Erkki Mäkinen have supervised the work and contributed to the text.

P2. Hadaytullah, Sriharsha Vathsavayi, Outi Räihä and Kai Koskimies. Tool Support for Software Architecture Design with Genetic Algorithms. In Proceedings of International Conference on Software Engineering Advances, Nice, France, August 2010, IEEE Computer Society Press, pp. 359-366. © 2010 IEEE.

This study covers the development and usage of the Darwin tool. It provides an interface to give inputs and visualize the results produced by the genetic algorithm [P1]. A user can set and modify population size, number of generations and mutation and crossover probabilities before and during an evolution. The embedded CASE tools can be used to design the input null architecture. Also, intermediate software architectures and their fitness values can be explored using the built-in views in the Darwin environment. Moreover, the environment allows for seeking the origin of a property in the proposed architecture through a dedicated view. In the contribution, Darwin's features, architecture, and usage have been included.

The author of the thesis has designed the Darwin tool and contributed significantly to its implementation. Sriharsha Vathsavayi is the other major contributor to the tool's implementation. Outi Räihä has contributed to the integration of the genetic algorithm with Darwin. Kai Koskimies has supervised the whole design and implementation process. All authors have contributed to the text.

P3. Hadaytullah, Outi Räihä and Kai Koskimies. Genetic Approach to Software Architecture Synthesis with Work Allocation Scheme. In Proceedings of Asia Pacific Software Engineering Conference, Sydney, Australia, December 2010, IEEE Computer Society Press, pp. 70-79. © 2010 IEEE.

In this contribution, the genetic algorithm of [P1] has been extended to produce architectures with good quality and structures that are in line with the developing organization's structure, along with a work distribution plan. The study is motivated by the fact that software architecture is not a mere result of functional or quality requirements, but it is also influenced by the structure of the developing organization [57]. One of the critical factor is the communication overhead [30] involved among the teams which typically has its roots in the physical distance and cultural or linguistic differences present among the teams. The genetic algorithm therefore not only gauges the quality of architectures but also associated plans. It favors good architectures with plans leading to reduced inter-team communication. In the plans, extreme under/over-loading of the teams is also discouraged by the genetic algorithm.

The author of the thesis was the main contributor to the work. The author has extended the genetic algorithm with new mutations and fitness functions as well as the Darwin environment. The experiments, results analysis and reporting have also been done by the author. Outi Räihä has provided valuable insights of the genetic algorithm to the author which enabled him to realize the extensions needed for this work. She also contributed to the text of the paper. Kai Koskimies has supervised the whole process and contributed to the text.

P4. Hadaytullah, Allan Gregersen and Kai Koskimies. Pattern-Based Dynamic Maintenance of Software Systems. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), Hong Kong, December 2012, IEEE Computer Society Press, pp. 537-546. © 2012 IEEE.

In this study, an infrastructure containing JITA-plugin (later re-named as the Reflection layer) has been realized for run-time maintenance of Java-based systems. The unit of modification is a pattern [11]. The challenges in pattern-based maintenance have been highlighted in the study. JITA exploits Javeleon [35], a dynamic Java class updating facility. The benefits of the infrastructure are demonstrated in the study using an example system where Adapter, Singleton and Observer patterns are successfully injected into the system at the run-time in response to three maintenance scenarios.

The author of the thesis was the main contributor to the developed JITA-plugin and text of the paper. Allan Gregersen has helped the author to integrate Javeleon in JITA. He also contributed to the text of the paper. Kai Koskimies has supervised the work and took part in the writing and reviewing of the paper.

P5. Hadaytullah, Sriharsha Vathsavayi, Outi Räihä, Allan Gregersen and Kai Koskimies. Applying Genetic Self-Architecting for Distributed Systems. In Proceedings of 4th World Congress on Nature and Biologically Inspired Computing (NaBIC'12), Mexico City, Mexico, November 2012, IEEE, pp. 44-52. © 2012 IEEE.

In this study, a self-architecting enabling infrastructure (SAGA) has been realized for Java based distributed systems. The goal was to self-maintain efficiency and reliability of Java-based systems through automated architectural modifications. The genetic algorithm [P1] (and the Darwin environment [P2]) has been extended to handle distributed systems. The genetic algorithm was responsible for suggesting the improvements in the designs. The infrastructure monitors and records various properties of the running system that are given to the genetic

algorithm as feedback. The feedback data is taken into account during the next improvement cycle by the genetic algorithm thus producing an efficient and reliable architecture for the changed environment. The proposed changes are then reflected to the running instance of the system using the Reflection layer [P4]. In the experiment, self-maintenance of an example system has been demonstrated in response to two usage profiles. Improvements in efficiency and reliability have been observed after the architectural adaptations for the usage profiles.

The author of this thesis was the leading contributor to the work and text included in the paper. The author has designed the self-architecting infrastructure. The Reflection and Monitoring layers and integration with Javeleon have been realized by the author. The author has formulated and implemented the reliability fitness function in the genetic algorithm. Sriharsha Vathsavayi has extended the Darwin tool for this work and contributed to the text in the paper. Outi Räihä has assisted in the formulation and implementation of the efficiency sub-fitness function. She has also introduced the Heartbeat solution and related mutations in the genetic algorithm. She has reported her work in the text, too. Allan Gregersen has assisted the author in the integration of Javeleon in the infrastructure. Kai Koskimies has supervised the whole process of design and realization of the infrastructure as well as contributed to the text.

9 Conclusions

9.1 Research Questions Revisited

I. How to genetically synthesize software designs to automatically generate good quality architectures? Are the generated designs comparable to man-made designs? What kind of tool support is needed?

Software architecture is composed of architectural solutions (design patterns, architectural styles or application specific solutions) originating from architectural decisions. Given a basic functional decomposition of a system (the null architecture), the developed genetic algorithm finds a combination of such solutions leading to good quality as defined by the fitness function. The operations in the null architecture are annotated with their call frequencies, execution times, number of parameters and variability values. The solution set contains solutions having an influence on the targeted quality attributes. The genetic algorithm inserts and removes the solutions to improve the quality. The architectural expertise (or metrics) is formulated in the form of the fitness function in the genetic algorithm. Thus, the fitness function promotes good quality architecture during simulated evolution. At occasions, the genetic algorithm generated architectures with reasonable solutions one would not expect. According to the empirical study reported in Chapter 3, the generated architectures are comparable to manually constructed designs of under graduate students.

The tool support has been provided in the form of the Darwin environment, as presented in Chapter 4. Darwin envelops the genetic algorithm along with other supporting tools. Darwin makes it possible to save, open or edit an evolution. Darwin also includes UML-based CASE tools that enable an architect to design the null architecture. Moreover, the CASE tools can be used to review or edit automatically generated proposals afterword. Additionally, input parameters can be fine-tuned before and during a genetic synthesis and the impact can be immediately observed. Also, architectures in the intermediate generations can be viewed to understand the

evolutionary stages the null architecture has been through before reaching its final shape. Note that the Darwin environment and the genetic architectural synthesis have only been used in laboratory settings thus far. Their application in industrial settings is still an open direction which is yet to be explored.

II. How to optimize work distribution plans along with software architectures using genetic algorithms for efficient distributed software development?

The communication among the teams is one of the major problems in distributed software development. The resistance in communication has its roots in the cultural, lingual or social differences among the teams. A task distribution will result in a number of dependencies among the teams which need to be coordinated. The strength of the dependencies among the tasks dictates the amount of inter-team communications. Typically, the stronger a dependency, the more communication is required. The tasks assignment is to be done in a way that there exist fewer and weaker dependencies among the teams with high communication resistance and vice versa.

Assuming that the operations in software architecture represent high level tasks, the developed genetic algorithm, as presented in Chapter 5, evolves the architecture to be easily distributable among the teams. At the same time, the genetic algorithm suggests an initial distribution plan. A sub-fitness function measures the communication overhead involved in a plan's execution taking into account the differences among the teams. Consequently, the genetic algorithm favors architecture where fewer dependencies exist among the teams with greater differences. At the same time, the algorithm favors architectures where coupling reducing solutions are used between the tasks/operations (or hosting components) assigned to the distant teams. Another sub-fitness function takes care of the number of tasks assigned to a team. It discourages extreme over/under-loading of the teams. The approach has been evaluated using an example system and developing organization.

III. How to enable run-time maintenance using architectural solutions? What kind of infrastructure is required?

A solution is composed of roles that are played by real artifacts in the design and run-time. A role may have further child roles and may depend on other roles in the solution. The role playing artifacts act together to accomplish the solution's goal. Once such roles are assigned to artifacts in the design, run-time artifacts can be automatically updated/generated. The update may involve localized or system wide run-time adaptations. The wider the adaptations, the more difficult it is to automate the role and therefore the hosting solution. Run-time updating capability is precondition for this

kind of automated maintenance. The automated reflection of application dependent roles is too challenging.

As detailed in Chapter 6, the implemented Reflection layer (or JITA-plugin) supports solution-based run-time maintenance for Java-based systems. The layer is part of SAGA and embedded inside the Eclipse IDE. It can be used to manually introduce solutions in the designs, represented as UML class diagrams. The automatable roles in the solutions are reflected to the run-time automatically. For application dependent roles, the editors provided by the Eclipse IDE can be used to manually code the associated behaviors. The run-time Java class updating capability is provided by Javeleon.

IV. How to apply genetic algorithms for enabling self-maintenance of non-functional properties associated with efficiency and reliability for software systems? What kind of infrastructure is required?

In presence of monitoring and architectural reflection capabilities, the genetic architectural synthesis can serve as a decision making engine to realize self-maintainable systems. The monitoring mechanism will monitor the properties of the system to be self-maintained. It is mandatory that the properties are measureable at run-time. The genetic algorithm will take into account the monitoring data while improving the properties through introduction/removal of the solutions to/from the architecture. It is also obligatory that the solutions to improve the properties are automatically injectable and removable. The architectural reflection mechanism is responsible for injecting and removing the solutions to/from the running instance of the architecture based on the genetic algorithm's proposal.

As reported in Chapter 6, in this thesis work, quality attributes were focused for self-maintenance. Unfortunately, all quality attributes cannot be monitored at run-time. For example, subjective properties like usability and learnability are hard to monitor automatically. Some input from humans is required which is not in line with the primary objective of the self-maintainable systems. Other quality attributes involve so many stages that automatic monitoring of all the stages is beyond the scope of this thesis work. For example, the portability property involves re-design, coding, testing, debugging, deployment and documentation [99]. The maintainability quality attribute need to be redefined in the self-maintainable systems paradigm. Furthermore, the lack of automatable solutions for some quality attributes had hindered the path to make them self-maintainable. Many solutions (and roles within) influencing the quality attributes depends on application logic which is challenging to automate.

The developed SAGA infrastructure enables solution-based self-maintenance of efficiency and reliability in the context of changing usage profiles. Different usage profiles require different architectural solutions and configurations so that the system remains efficient and reliable. The infrastructure has been implemented for Java-based distributed systems. It is composed of the genetic algorithm and two layers Reflection and Monitoring layers. The Monitoring layer monitors the system for variations in its usage by recording usage frequencies and execution times of operations, and failures of the components. The data is then fed to the genetic algorithm. In response, the algorithm suggests changes in the architecture using a set of solutions having an impact on efficiency and reliability of the system. The Reflection layer then reflects the architecture to the run-time. It relies on Javeleon for dynamic update of Java classes at the run-time. The infrastructure also supports manual maintenance of non self-maintainable properties.

9.2 Limitations

9.2.1 Genetic Architectural Synthesis Limitations

The proposals generated by the genetic algorithm are of preliminary nature and could not be used on "as-it-is" basis. The genetic algorithm is not aware of the semantics of the architectural elements. The designs suggested by the genetic algorithm are constructed using a fairly limited set of solutions which may not address the wide range of matters typically associated with software systems. The selected set of solutions has an influence on the formulation of the fitness function and therefore on the generated proposals. One set or type of solutions is not enough to exercise the genetic architectural synthesis in different kinds of systems. The algorithm and solution set need to be extended for varying types of systems.

In the genetic algorithm (and therefore in SAGA), at the moment solutions, mutations and fitness functions are hard coded. This results in significant effort when applying the genetic synthesis to different kinds of systems involving diverse architectural solutions, metrics and heuristics. Integration of an extendable solutions database or a solution specification language could lift this constraint.

The null architecture and associated initial plan or node allocation has potential to influence the results. The support data in the null architecture is an estimation of the attributes associated with the operations. The estimation may vary from one architect to another and therefore lead to different proposals. The initial plan associated with the null architecture may have plenty of good work assignments and therefore the improvements in the plan or fitness graph may appear to be minuscule. Same applies to

the initial node allocation. For self-maintenance, the degree of improvement may vary depending on how much the starting node allocation favors or disfavors the current usage profile. However, the final proposal will be good as defined by the fitness function.

The capturing of true "goodness" into a fitness function is key to the genetic architectural synthesis. The formulation of the goodness may be biased towards the fitness function designer's know-how of software architectures. Furthermore, one set of heuristics may be true for one kind of systems, however, not so much for others. Therefore, the fitness functions need re-adjustment when producing architectures for different kinds of systems. However, the use of widely accepted metrics and well-known heuristics has potentially reduced this risk. Moreover, the incorporation of well documented solutions as base for the "goodness" of software architectures further decreases this threat as their quality related implications are usually well understood.

The single value fitness function sums up all the sub-fitness functions. The sub-fitness functions do not share a single unit and each may calculate an arbitrary range of values. A sub-fitness function with range of significantly large values may overshadow other sub-fitness functions with smaller range of values. The scaling of the values can be accomplished through weights, however, the scaling that works for one system may not work for another system. Therefore, the weights will require re-adjustment when moving from one system to another. This problem could be avoided by ranking each individual architecture for each sub-fitness separately (e.g., using Pareto optimality [23] method). This way an architect will have a choice to pick from a wide spectrum of proposals with varying traits.

9.2.2 Work Planning Automation Limitations

The work planning automation is based on a simplified work planning model, however, in reality there are many more forces contributing to the final software development plan. Inclusion of more attributes will indeed produce more practical work distribution plans. At the same time, it is challenging to program an algorithm to take care of all such complicated attributes related to the planning activity.

In the current setup, the numeric representation of differences among the teams in the organization has an influence on the outcomes. The estimation of the values is a challenging task and may vary from one manager/architect to another. However, unless the values are set so that they can clearly reflect the relative differences among the teams, the proposals would not be dramatically different.

9.2.3 Run-Time and Self-Architecting Maintenance Limitations

The run-time manual maintenance has been tested using a limited set of localized maintenance scenarios which were designed in the context of the available solutions. A maintenance cycle may require system-wide architectural adaptation instead of localized adaptations using localized solutions. Moreover, a maintenance situation can be anything and may require solutions not included in the available set of dynamic solutions. However, even if all solutions are available to an architect, it is challenging to identify the solutions that can address the maintenance situation on hand.

The maintenance scenarios were targeted to the software architecture level which might not be the case always. Some maintenance needs may require adjusting of few parameters or fine grained behavioral changes. Since the infrastructure is integrated into the Eclipse IDE, some of the complex maintenance needs, requiring fine grained behavioral or code changes, could be handled manually.

SAGA is a proof of concept, so far tried in laboratory settings. A real distributed environment will surely bring new challenges along. The SAGA infrastructure has been studied in the context of a limited set of solutions and run-time properties. The experiments were conducted in laboratory settings with controlled inputs or usage profiles. In reality, there can be complex and constantly changing usage patterns. Furthermore, the period for a usage profile may vary spanning over few minutes to weeks or months. Therefore, setting a period for adaptation cycle could be challenging.

The pre-defined threshold values for the self-maintained properties need to be re-adjusted for different systems. In addition, the threshold values may also vary from one system admin/architect to another. Moreover, the degradation in the self-maintained property may be due to un-monitored forces. For example, efficiency may be reduced due to the network congestion rather than the changed usage profile. Such situations have been avoided in SAGA's evaluation due to the controlled environment. The fitness function could be extended to include network congestion and other environmental parameters with considerable impact on the self-maintained properties.

In SAGA's experiments, efficiency related measurements were based on the execution times of the operations. In reality, an operation could spawn multiple threads. The threads durations should also be counted towards the execution time of the operation. Also, asynchronous network calls within an operation could lead to a reduced execution time. In the experiments, the example systems were designed to avoid these complicated situations.

There are usually many implied aspects of a system that are hardly reflected in its architecture or code. Even for a human developer, it is challenging to successfully maintain a code in absence of the knowledge of such tacit dimensions. Therefore, there is always a risk that the automatically generated or altered code contains errors. In other situations, the generated code can be executable but may not conform to the fundamental assumptions the system has been built upon. A systematic articulation of such conventions, aspects or assumptions is well beyond the scope of this thesis work.

Technically, Javeleon was intended for single machine applications. Therefore, the distributed setup was emulated using virtual nodes running inside a single machine. Currently, Java-based distributed technologies do not fully support dynamic update of running systems. Certainly, a real distributed setup will bring new challenges. For example, when a component is moved from one machine (node) to another, the definition and state has to be moved to the new host. In the current setup, the components on the virtual nodes communicate through a messaging interface. This arrangement will make it easier to adopt the infrastructure to a real distributed environment. The state and specification migration mechanism in the infrastructure has to be developed then.

The state maintenance is still partly an open issue in SAGA. Addition or removal of a solution transforms into fine grained additions/removals and alterations in the running system. The application or removal of the Heartbeat solution does not affect the state. However, when a component is moved out from a node, its clients in the node create a new remote instance of the moved component thus losing the state. Also, when a remote component is moved into a node, its clients in the node re-instantiate a local instance of the component. The automatic generation of STFs for such situations requires consideration of the current and previous roles of each of the alive objects in the system. Each role transition requires a unique set of state adaptation logic.

The number of possible role transitions will increase significantly as new solutions are included in the infrastructure. Since Javeleon was not designed for self-maintaining systems, the limited customization of state maintenance it offers is not enough to handle such complex role transitions based state maintenance. Hence, development of an appropriate infrastructure wide state maintenance mechanism seemed to be a more logical choice. Since a solution can be in principle anything, assessing the state maintenance related needs of all available or known solutions and translating them into a single mechanism requires a study of its own.

9.2.4 Evaluation Limitations

All the experiments were performed using a limited set of example systems. Inclusion of many more example systems, preferably industrial ones, could further confirm the feasibility of the presented techniques and artifacts. However, the example systems were realistic enough in regards to their size and complexity for evaluating the presented work. The consistency of the good outcomes from the genetic algorithm cannot be guaranteed due to the inherent random nature of genetic algorithms. To reduce this threat, for each automation scenario in all the experiments, the genetic algorithm was executed multiple times (10 to 100 times) [82][83]. The proposals were considered for further evaluation only once majority of the architectures were sensible enough with small variations. Certainly, a set of entirely different or weird proposals were also observed in the experiments.

The efficiency of the genetic algorithm depends on many parameters like population size, number of generations, number of mutation etc. Furthermore, the size of the example system can also influence the genetic algorithm's efficiency. Limited efficiency related data have been reported in Section 4.5, however, an extended study is still pending.

In the empirical study, the students' architectural decisions were confined by the limited set of choices and solutions. A broader set of choices could have resulted in better manually designed architectures. The selected architectures for expert review may not be a true representation of the whole collection. However, the experts were able to find good, worst and average architectures from the selected architectures. This indicates that the selection had a good variety of solutions. Furthermore, the evaluations are subjective depending on the experts' experiences. In the study, however, there was only one expert whose evaluations were significantly different than others. Therefore, it is fair to say that the opinions of the experts were reasonably consistent.

9.3 Future Work

The genetic algorithm could be extended to include new quality attributes of software architectures. The automatically generated plans would be more practical, if more variables related to software planning are taken into account. Similarly, the SAGA infrastructure could be improved to address many attributes currently not possible to self-maintain. More intelligent algorithms based on other methods like artificial intelligence could be introduced in SAGA to handle such complicated aspects.

Inclusion of a new solution or quality attribute to the genetic algorithm requires significant amount of efforts as everything is hard coded right now. However, a

specification language for solutions related quality attributes would reduce the efforts. A similar facility could be realized for the specification of the fitness functions, mutations and crossover.

One interesting area is to explore software planning automation in the context of global software development. A more concrete specification framework could be developed containing skills, expertise, experience, performance and availability of teams or personnel. Then, the produced plans would be more practical as well they could include the order of execution for the tasks. Some work has already been done in this direction [105].

An obvious extension of SAGA would be to automate other properties beside efficiency and reliability. For each new property, the driving forces need to be identified and formulated which could be a tough task in case of subjective properties. Furthermore, new dynamic solutions could be explored or designed for the new self-maintainable properties. Finally, the approach could be applied to different system categories like Service Oriented Architecture (SOA) based systems in future.

9.4 Final Remarks

Software design, work distribution and maintenance are challenging and resource consuming activities. The ever increasing complexity of software systems will only lead to an increased demand for resources. One solution is to develop and adopt automated approaches as presented in this thesis work. This thesis work has highlighted the potential of genetic algorithms in introducing varying levels of automation to design, planning and maintenance stages. The genetic algorithm was able to produce good architectures taking into account many cross cutting concerns which would be challenging for a human architect.

Manually cross referencing the attributes of teams with the properties of software architecture could be mind boggling for a manger or architect. The use of machines (like the presented genetic algorithm) makes more sense to resolve such complex multi-objective tasks. The genetic algorithm takes care of both architectural quality and work distribution at the same time. The work opens an interesting research direction with many more possibilities. It could be applied in work distribution in globally distributed organizations. The inter-tasks dependencies could be automatically processed to generate the efficient execution plans.

Software maintenance consumes a big portion of the budget. The self-adaptive approaches (like our SAGA approach) have potential to offer self-maintainable software systems aware of their changing environment. The development and

computational costs associated with self-adaptive technologies or infrastructures cannot be ignored. However, in the long run, the financial benefits will eventually outweigh the cost spent for the development of such approaches and related technologies. Furthermore, with ever increasing computing power the computational overheads may not be that big of a risk. The reduced cost would allow software systems to penetrate many more areas and small businesses. SAGA is the proof of concept that genetic architectural synthesis has its potential to relieve the load from the maintenance phase. SAGA has been able to self-maintain efficiency and reliability of software systems.

In the author's opinion, the software engineering concepts and current technologies are not yet ready for the self-adaptive systems. Self-adaptive systems need a paradigm shift on how software systems are viewed today. In near future the current concepts and technologies should be adapted to be more accepting to the self-adaptive systems. At the same time, new concepts, practices and technologies may also emerge as the software engineering community and industry embraces the self-adaptive paradigm.

The author is optimistic about the spread of automated approaches in all stages of software life cycle. The target is challenging but not impossible. A more realistic near future goal would be to develop approaches where man and machines can work side-by-side to design, develop, test and maintain software systems. A collaborative effort among the academic institutes and the industry is crucial to speed up the pace of research in this direction.

Bibliography

[1] International Standard on Systems and Software Engineering- Software Life Cycle Processes 2008, ISO/IEC Std 12207-2008.

[2] Brown W.J., Malveau R.C., McCormickIII H.W. and Mowbray T.J., "Antipatterns - Refactoring Software, Architectures, and Projects," In Crisis, Jhon Wiley and Sons, Inc., 1998.

[3] Kramer J. and Magee J., "Dynamic Configuration for Distributed Systems," In IEEE Transactions on Software Engineering, 11,4,1985, pp. 424–436.

[4] Budinsky F.J., Finnie M.A., Vlissides J.M. and Yu P.S., "Automatic code generation from design patterns," In IBM Systems Journal, 35, 2, IBM Corp. Riverton, NJ, USA, 1996, pp. 151-171.

[5] Selonen P., Koskimies K. and Systä T., "Generating structured implementation schemes from UML sequence diagrams," In Proc. TOOLS USA, IEEE CS, Santa Barbara, July 2001, pp. 317-328.

[6] Keeffe M.O. and Cinnéide M.O., "Towards automated design improvements through combinatorial optimization," In Proc. 4th International Workshop on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop - 26th International Conference on Software Engineering, 2004, pp. 75 – 82.

[7] Garlan D., Cheng S.W., Huang A.C., Schmerl B. and Steenkiste P., "Rainbow: architecture-based self-adaptation with reusable infrastructure," In IEEE Computer, 37, 10, 2004, pp. 46–54

[8] White S.R., Hanson J.E., Whalley I., Chess D.M. and Kephart J.O., "An architectural approach to autonomic computing," In Proc. The International Conference on Autonomic Computing, 2004, pp. 2-9.

[9] Keeffe M.O. and Cinnéide M.O., "Search-based software maintenance," In Proc. Conference on Software Maintenance and Reengineering (CSMR'06), 2006, pp. 249-260.

[10] Würthinger T., Wimmer C. and Stadler L., "Dynamic code evolution for Java," In Proc. 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10), ACM, New York, NY, USA, 2010, pp. 10-19.

[11] Gamma E., Helm R., Johnson R. and Vlissides J., Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[12] Shaw M. and Garlan D., Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[13] Clarke J. et al., "Reformulating software engineering as a search problem," In IEE Proceedings - Software, 150, 3, 2003, pp. 161-175.

[14] Amoui M., Mirarab S., Ansari S. and Lucas C., "A genetic algorithm approach to design evolution using design pattern transformation," In International Journal of Information Technology and Intelligent Computing, 1, 2006, pp. 235-245.

[15] Lutz R., "Evolving good hierarchical decompositions of complex systems," In Journal of Systems Architecture, 47, 2001, pp. 613-634.

[16] Mancoridis S., Mitchell B.S., Rorres C., Chen Y.F. and Gansner E.R., "Using automatic clustering to produce high-level system organizations of source code," In Proc. International Workshop on Program Comprehension (IWPC 98), USA, 1998, pp. 45-53.

[17] Mitchell B.S., Mancoridis S. and Traverso M., "Search based reverse engineering," In Proc. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), 2002, pp. 431-438.

[18] Keeffe M.O. and Cinnéide M.O, "Search-based refactoring for software maintenance," In Journal of Systems and Software, 81, 4, 2008, pp. 502-516.

[19] Simons C.L. and Parmee I.C., "Single and multi-objective genetic operators in object-oriented conceptual software design," In Proc. Genetic and Evolutionary Computation Conference (GECCO'07), 2007, pp. 1957-1958.

[20] Simons C.L. and Parmee I.C., "A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design," In Engineering Optimization, 39, 5, 2007, pp. 631-648.

[21] Yu F., Tu F. and Pattipati K.R., "A novel congruent organizational design methodology using group technology and a nested genetic algorithm," In IEEE Transaction on Systems, Man, and Cybernetics, part A: Systems and Humans, 36, 1, Jan. 2006, pp. 5-18.

[22] Aleti A., Björnander S., Grunske L. and Meedeniya I., "ArcheOptrix: an extendable tool for architecture optimization of AADL models," In Proc. ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software, 2009, pp. 61-71.

[23] Deb K., "Evolutionary algorithms for multi-criterion optimization in engineering design," In Proc. Evolutionary Algorithms in Engineering and Computer Sciences (EUROGEN'99), 1999, pp. 135-161.

[24] Räihä O., Koskimies K. and Mäkinen E., "Genetic Synthesis of Software Architecture," In Proc. 7th International Conference on Simulated Evolution and Learning (SEAL'08), Springer LNCS 5361, 2008, pp. 565-574.

[25] Räihä O., Koskimies K., Mäkinen E. and Systä T., "Pattern-Based Genetic Model Refinements in MDA," In Nordic Journal of Computing, 14, 4, 2008, pp. 322-339.

[26] Räihä O., Koskimies K. and Mäkinen E., "Scenario-based genetic synthesis of software architecture," In Proc. Fourth International Conference on Software Engineering Advances (ICSEA'09), 2009, pp. 437-445.

[27] Räihä O., Evolutionary Software Architecture Design, University of Tampere, Department of Computer Sciences, Report D-2008-11, 2008.

[28] Michalewicz Z. Genetic Algorithms + Data Structures = Evolution Programs. Springer, 1992.

[29] Herbsleb J.D. and Grinter R.E., "Splitting the Organization and Integrating the Code: Conway's Law Revisited," In Proc. 21st International Conference on Software Engineering, 1999, pp. 85-95.

[30] Ågerfalk P.J., Fitzgerald B., Holmstrom H, Lings B., Lundell B. and Conchuir E.O., "A framework for considering opportunities and threats in distributed software development," In Proc. International Workshop on Distributed Software Development, Paris, France: Austrian Computer Society, 2005, pp. 47-61.

[31] Pigoski T., Practical Software Maintenance, Wiley Computer Publishing, 1997.

[32] Sommerville I., Software Engineering; 8th edition. Addison-Wesley, 2007

[33] International Standard on Software Engineering-Software Life Cycle Processes-Maintenance 2006, 2, ISO/IEC 14764, IEEE Std 14764-2006.

[34] Cheng B., Lemos R. de, Giese H., Inverardi P. and Magee J., "Software Engineering for Self-Adaptive Systems: A Research Roadmap," In Software Engineering for Self-Adaptive Systems. Springer, 2009, pp. 1-26.

[35] Gregersen A.R. and Jørgensen B.N., "Dynamic update of Java applications - balancing change flexibility vs programming transparency," In Journal of Software Maintenance and Evolution: Research and Practice - Special Issue on the 12th Conference on Software Maintenance and Reengineering (CSMR), 21, 2, 2008, pp.81-112.

[36] UML (Unified Modeling Language). http://www.omg.org/spec/UML/, visited on 9 Dec. 2013.

[37] Oreizy P., Gorlick M.M., Taylor R.N., Heimhigner D., Johnson G., Medvidovic N., Quilici A., Rosenblum D.S. and Wolf A.L., "An architecture-based approach to self-adaptive software," In IEEE Intelligent Systems and their Applications,14, 3, 1999, pp. 54-62.

[38] Gomaa H. and Hussein M., "Software reconfiguration patterns for dynamic evolution of software architectures," In Proc. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA), 2004, pp. 79-88.

[39] March S.T. and Smith G.F., "Design and natural science research on information technology," In Journal of Decision Support Systems, 15, 4, 1995, pp. 251-266.

[40] Järvinen P., "Mapping Research Questions to Research Methods," In IFIP International Federation for Information Processing, 274; Advances in Information Systems Research, Education and Practice; David Avison, George M. Kasper, Barbara Pernici, Isabel Ramos, Dewald Roode; Boston, Springer, 2008, pp. 29–41.

[41] Hevner A.R., March S.T., Park J., and Ram S. "Design science in information systems research," In MIS Quarterly of March, 28, 1, 2004, pp. 75-105.

[42] Greenfield J., Short K., Cook S., Kent S. and Crupi J., Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. John Wiley & Sons, Chichester, 2004.

[43] Herbsleb J.D. and Moitra D., "Guest Editors' Introduction: Global Software Development," In IEEE Software, 18, 2, March 2001, pp. 16-20.

[44] Jimenez M. and Piattini M., Problems and Solutions in Distributed Software Development: A Systematic Review. Chapter in the Book on Software Engineering Approaches for Offshore and Outsourced Development, Lecture Notes in Business Information Processing (Berkling K. Joseph M. Meyer B. and Nordio M. edition), Springer Berlin Heidelberg, 16, 2009, pp. 107-125.

[45] Eick S., Graves T., Karr A., Marron J. and Mockus A., "Does Code Decay? Assessing Evidence from Change Management Data," In IEEE Transactions on Software Engineering, 27, 1, 2001, pp. 1-12.

[46] IEEE Standard 1219-1998, IEEE Standard for Software Maintenance, 1998.

[47] Laddaga R., Self-adaptive software SOL BAA 98-12, 1998. http://people.csail.mit.edu/rladdaga/BAA98-12excerpt.html, visited on 09 Dec. 2013.

[48] Malek S., Esfahani N., Menasce D.A., Sousa J.P. and Gomaa H., "Self-Architecting Software SYstems (SASSY) from QoS-annotated activity models," In Proc. ICSE Workshop on Principles of Engineering Service Oriented Systems 2009 (PESOS 2009), 2009 , pp. 62 – 69.

[49] Räihä O., "A survey on search-based software design," In Computer Science Review, 4, 4, Nov. 2010, pp. 203-249.

[50] Brown W.J., Malveau C., McCormick H.W. and Mowbray T.J., Antipatterns – refactoring software, architectures, and projects in crisis, 1998, Wiley.

[51] Seng O., Bauyer M., Biehl M. and Pache G., "Search-based improvement of subsystem decomposition," In Proc. The Genetic and Evolutionary Computation Conference (GECCO'05), Mannheim, Germany. ACM Press, 2005, pp. 1045-1051.

[52] Seng O., Stammel J. and Burkhart D., "Search-based determination of refactorings for improving the class structure of object-oriented systems," In Proc. The Genetic and Evolutionary Computation Conference (GECCO'06), Washington, USA. ACM Press, 2006, pp. 1909-1916.

[53] Di Penta M., Neteler M., Antoniol G. and Merlo E., "A language independent software renovation framework," In Journal of Systems and Software, 77, 3, 2005, pp. 225-240.

[54] McGregor J., Bachmann F., Bass L. and Bianco P., "Using ArchE in the classroom: one experience," Technical Note, CMU/SEI-2007-TN-001, 2007.

[55] ArchE, http://www.sei.cmu.edu/architecture/tools/arche/index.cfm, visited on 09 Dec. 2013.

[56] Zest, http://www.eclipse.org/gef/zest, visited on 09 Dec. 2013.

[57] Conway M., "How Do Committees Invent?," Datamation magazine, April 1968, http://www.melconway.com/research/ committees.html, visited on 09 Dec. 2013.

[58] ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description, http://www.iso-architecture.org/42010/index.html, visited on 09 Dec. 2013.

[59] ISO 2000 International Standard ISO/IEC FDIS 9126-1. Information technology – Software product quality. Part 1: Quality Model (2000).

[60] Heesch U.V., Architecture Decisions: The next step, Ph.D. Thesis at University of Groningen, Groningen, Netherlands, 2012.

[61] Frederick P. B., The Mythical Man-Month (1975), Addison-Wesley, 1995.

[62] Hohpe G. and Woolf B., Enterprise Integration Patterns, Addison-Wesley, 2003.

[63] Hanmer R., Patterns for Fault Tolerant Software, Wiley, 2007.

[64] Brosch F., Koziolek H., Buhnova B. and Reussner R., "Architecture-based reliability prediction with the Palladio component model," In IEEE Transactions on Software Engineering 2012, 38, 6, pp. 1319-1339.

[65] Holland J.H., Adaption in Natural and Artificial Systems, MIT Press, Ann Arbor, 1975.

[66] Chidamber S.R. and Kemerer C.F., "A metrics suite for object oriented design," In IEEE Transactions on Software Engineering, 20, 6, June 1994, pp. 476-493.

[67] Ossher H. and Tarr P., "Using multidimensional separation of concerns to (re)shape evolving software," In ACM Communications Magazine, 44, 10, 2011, pp. 43-50.

[68] Chess D.M., Segal A., Whalley I. and White S. R., " Unity: experiences with a prototype autonomic computing system," In Proc. International Conference on Autonomic Computing, 2004, pp. 140–147.

[69] Buschmann F., Meunier R., Rohnert H., Sommerland P. and Stal M., A System of Patterns – Pattern-Oriented Software Architecture. Wiley, 1996.

[70] Florijn G., Meijers M. and van Winsen P., "Tool Support for Object-Oriented Design Patterns," In Proc. 11th European Conference on Object Oriented Programming (ECOOP), Springer, 1997, LNCS 1241, pp. 472-495.

[71] Würthinger T., Wimmer C. and Stadler L., “Dynamic code evolution for Java,” In Proc. 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10), ACM, New York, NY, USA, 2010, pp. 10-19.

[72] Subramanian S., Hicks M. and McKinley K.S., “Dynamic software updates: a VM-centric approach,” In Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI '09), ACM, New York, NY, USA, 2009, pp. 1-12.

[73] Kabanov J., “JRebel Tool Demo,” Electronic Notes in Theoretical Computer Science (ENTCS), 264, 4, Elsevier Science Publishers B. V. Amsterdam, The Netherlands, The Netherlands, 2011, pp. 51-57.

[74] Gregersen A.R. and Jørgensen B.N., “Run-time Phenomena in Dynamic Software Updating: Causes and Effects,” In Proc. 12th International Workshop on Principles of Software Evolution and 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL '11), ACM, New York, NY, USA, 2011, pp. 6-15.

[75] JFreeChart, http://www.jfree.org/jfreechart/, visited on 09 Dec. 2013.

[76] Eclipse, http://www.eclipse.org, visited on 09 Dec. 2013.

[77] UML2Tool, http://www.eclipse.org/modeling/mdt/?project=uml2tools, visited on 09 Dec. 2013.

[78] Acceleo, http://eclipse.org/acceleo/, visited on 09 Dec. 2013.

[79] Model to Text, http://www.omg.org/spec/MOFM2T/, visited on 09 Dec. 2013.

[80] Abstract-Syntax-Tree, http://www.eclipse.org/resources/resource.php?id=260, visited on 09 Dec. 2013.

[81] J2EE. Available: http://java.sun.com/j2ee, visited on 09 Dec. 2013.

[82] Arcuri A. and Briand L., “A practical guide for using statistical tests to assess randomized algorithms in software engineering”, In Proc. 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 2011, pp. 1-10.

[83] Harman M., McMinn P., de Souza J.T. and Yoo S., “Search Based Software Engineering: Techniques, Taxonomy, Tutorial,” In Empirical software engineering and verification, Lecture notes in computer science, Springer-Verlag Berlin Heidelberg, 7007, 2012, pp.1-59.

[84] Dascalu S., Ning H. and Debnath N., “Design patterns automation with template library,” In Proc. Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005, pp. 699 – 705.

[85] Menascé D. A., Sousa J. P., Malek S. and Gomaa H.,“QoS architectural patterns for self-architecting software systems,” In Proc. of the 7th International Conference on Autonomic Computing, USA, 2010, pp. 195–204.

[86] OpenJMS. http://openjms.sourceforge.net/, visited on 09 Dec. 2013.

[87] AspectJ. Available: http://www.eclipse.org/aspectj/, visited on 09 Dec. 2013.

[88] Kalbarczyk Z.T., Iyer R.K., Bagchi S. and Whisnant K., “Chameleon: a software infrastructure for adaptive fault tolerance,” In IEEE Transactions on Parallel and Distributed Systems, 10, 6, 1999, pp. 560-579.

[89] Fabre J.C. and Pérennou T., “A meta-object architecture for fault-tolerant distributed systems: the FRIENDS approach,” In IEEE Transactions on Computers, 47, 1, 1998, pp. 78-95.

[90] Cristian F., “Automatic Reconfiguration in the Presence of Failures,” In International Workshop on Configurable Distributed Systems, IEE, March, 1992, pp. 4-17.

[91] Little M.C., Object Replication in a Distributed System, Ph.D. Thesis, University of Newcastle upon Tyne, 1991.

[92] Kopetz H., Kantz H., Griinsteidl G., Puschner P. and Reisinger J., "Tolerating Transient Faults in MARS," In 20th International Symposium on Fault-Tolerant Computing, 1990, pp. 466-473.

[93] McCue D. and Little M, "Computing Replica Placement in Distributed Systems," In Second IEEE Workshop on the Management of Replicated Data, Nov. 1992, pp. 58-61.

[94] Fliege I., Geraldy A., Gotzhein R., Kuhn T. and Webel C., "Developing safety-critical real-time systems with SDL design patterns and components," In Computer Networks, 49, 2005, pp. 689-706.

[95] Martens A., Koziolek H., Becker S. and Reussner R., "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," In WOSP/SIPEW'10, ACM Press, 2010, pp. 105-116.

[96] Riehle D., Framework design: A role modeling approach, Ph.D. Thesis, Swiss Federal Institute of Technology Zurich, Universitat Hamburg, 2000.

[97] Guennec A.L., Sunye G. and Jezequel J., "Precise modeling of design patterns," In UML'00, 2000, pp. 482-496.

[98] Kim D.K, France R., Ghosh S. and Song E., "A Role-Based Metamodeling Approach to Specifying Design Patterns," In 27th International Computer Software and Applications Conference (COMPSAC'03), IEEE Computer Society, Washington, DC, USA, 2003, pp. 452-457.

[99] Mooney J.D., "Issues in The Specification And Measurement of Software Portability," Technical Report TR 93-6, Dept. of Statistics and Computer Science, West Virginia University, Morgantown WV, 1993.

[100] Feitelson D.G. and Naaman M., "Self-tuning systems," In IEEE Journal on Software, 16, 2, 1999, pp. 52 – 60.

[101] Kramer J. and Magee J., "Self-Managed Systems: an Architectural Challenge," In Symposium on Future of Software Engineering, May 2007, pp. 259-268.

[102] Jansen A. and Bosch J., "Software Architecture as a Set of Architectural Design Decisions," In Working IEEE/IFIP Conference on Software Architecture (WICSA), 2005, pp. 109-120.

[103] Bosch J., "Software architecture: The next step," Lecture Notes in Computer Science, Software Architecture (Oquendo F., Warboys B. and Morrison R. edition), 3047, Springer, 2004, pp. 194–199.

[104] Zelkowitz M.V. and Wallace D., "Experimental Validation in Software Engineering," In Journal of Information and Software Technology, 39, 1997, pp. 735-743.

[105] Vathsavayi S., Sievi-Korte O., Koskimies K. and Systä K., "Planning Global Software Development Projects Using Genetic Algorithms," In Search Based Software Engineering, 8084, G. Ruhe and Y. Zhang Eds., Springer Berlin Heidelberg, 2013, pp. 269–274.

Appendices

Outi Räihä, Hadaytullah, Kai Koskimies and Erkki Mäkinen. Synthesizing Architecture from Requirements: A Genetic Approach. Relating Software Requirements and Architecture (eds. P. Avgeriou, J. Grundy, J.G. Hall, P. Lago, I. Mistrik), Chapter 18, 307-331, Springer 2011.

Hadaytullah, Sriharsha Vathsavayi, Outi Räihä and Kai Koskimies. Tool Support for Software Architecture Design with Genetic Algorithms. In Proceedings of International Conference on Software Engineering Advances, 359-366, IEEE Computer Society Press, Nice, France, August 2010.

Hadaytullah, Outi Räihä and Kai Koskimies. Genetic Approach to Software Architecture Synthesis with Work Allocation Scheme. In Proceedings of Asia Pacific Software Engineering Conference, 70-79, Australia, December 2010.

Hadaytullah, Allan Gregersen and Kai Koskimies. Pattern-Based Dynamic Maintenance of Software Systems. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), 537-546, Hong Kong, 2012.

Hadaytullah, Sriharsha Vathsavayi, Outi Räihä, Allan Gregersen and Kai Koskimies. Applying Genetic Self-Architecting for Distributed Systems. In Proceedings of 4th World Congress on Nature and Biologically Inspired Computing (NaBIC'12), 44-52, Mexico City, Mexico, 2012.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3365-5
ISSN 1459-2045