Outi Räihä

Evolutionary Software Architecture Design

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

D-2008-11

TAMPERE 2008

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCES
SERIES OF PUBLICATIONS D – NET PUBLICATIONS
D-2008-11, SEPTEMBER 2008

Outti Räihä

Evolutionary Software Architecture Design

DEPARTMENT OF COMPUTER SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 978-951-44-7505-4
ISSN 1795-4274

Abstract

This thesis experiments with a novel approach to applying genetic algorithms in software architecture design by giving the structure of an architecture at a highly abstract level. Previously in the literature, genetic algorithms are used only to improve existing architectures. The structure and evaluation of software architectures and the principles of meta-heuristic search algorithms are introduced to give a basis to understand the implementation. Current research in the field of search-based software engineering is explored to give a perspective to the implementation presented in this thesis. The chosen genetic construction of software architectures is based on a model which contains information of a set of responsibilities and dependencies between them. An implementation using this model is presented, as well as test results achieved from a case study made on a sketch of an electronic home control system. The test results show that quality results can be achieved using the selected approach and that the presented implementation is a good starting point for future research.

Key words and terms: search-based software engineering, genetic algorithms, software architecture, software design

Contents

1. Introduction 1
2. Software architectures 3
    2.1. The structure of an architecture 3
    2.2. Standard solutions 5
        2.2.1. Design patterns 5
        2.2.2. Architecture styles..7
    2.3. Evaluating an architecture 9
        2.3.1. Evaluation using metrics..9
        2.3.2. Evaluation using human expertise 14
3. Meta-heuristic search algorithms...16
    3.1. Genetic algorithms..16
        3.1.1. Encoding..17
        3.1.2. Mutations...18
        3.1.3. Crossover 18
        3.1.4. Fitness function...20
        3.1.5. Selection operator 20
        3.1.6. Executing a genetic algorithm 21
    3.2. Tabu search and simulated annealing 23
        3.2.1. Tabu search..23
        3.2.2. Simulated annealing 24
4. Search algorithms in software engineering 26
    4.1. Search algorithms in software design 26
        4.1.1. Software clustering 26
        4.1.2. Systems integration 30
        4.1.3. Systems refactoring...31
        4.1.4. Architecture development 34
    4.2. Search algorithms in software analysis and testing.............................37
5. Genetic construction of software architectures...40
    5.1. Architecture representation..40
    5.2. Mutations...41
    5.3. Crossover...44
    5.4. Architecture laws 47
6. Implementation...49
    6.1. Presenting the program..49
        6.1.1. Structure..49
        6.1.2. Algorithms 52

6.1.3. Parameters 59
6.2. Evaluation metrics 60
6.2.1. Efficiency 61
6.2.2. Modifiability 62
7. Experiments 64
7.1. Fitness development 64
7.2. Example solutions 69
7.3. Remarks on adjusting the parameters 76
8. Conclusions 78
8.1. Presenting the results 78
8.2. Success evaluation 79
8.3. Future work 80

References 82

Appendices
Appendix A: Case study data
Appendix B: Fitness study parameters
Appendix C: Example solutions' parameters

1. Introduction

The most constant thing in the field of software engineering today is that the field is changing. Software systems become larger and more complex, while at the same time the mobile industry is growing rapidly, calling for new techniques and intricate systems to be implemented with limited resources. As software enterprises become multinational, the need for shared systems also grows. As the systems grow in complexity, so does the need for highly talented software architects to keep the systems under control, which is not an easy task especially when thinking of dynamic systems with constantly changing architectures. Clearly, some kind of automated method is needed in order to aid the design of such dynamic architectures by giving the human architects suggestions and starting points which they can then fine-tune into quality software architectures.

What could such a method be? What can be used to evolve modifiable, reusable and efficient software architectures from complicated sets of requirements, especially if the architectures need to conform to changes in their environments? A precedent to this problem can be found in nature, where complex species have evolved from simple organisms, and are constantly able to adapt to changes in the environment. The evolution happens through generations with the idea of the survival of the fittest: the ones with the ability to survive will be able to produce new offspring who will then inherit the properties needed for survival. Changes in species also occur through mutations, which are the key to survival when the change in environment is so drastic that rapid adaptation is needed, but happen also constantly at a lower level. However, the adaptation "project" with species takes perhaps hundreds of generations and years, which is not acceptable in the field of software engineering. Fortunately, a simulation can be done quite fast to achieve similar results with the use of genetic algorithms.

Genetic algorithms operate with analogies to evolution in biology. As in biology a chromosome keeps the "solution" to the question as to how certain properties of a species work, a solution to a software engineering problem can be modeled as a "chromosome" in order for it to be operated by a genetic algorithm. This model is then altered by mutations, which change one specific feature, and crossovers which, as in nature, combine the characteristics of two individuals in their offspring.

Genetic algorithms are suitable for modifying software architectures as they too have certain constants which can be implemented in various ways. An architecture is based on the requirements as to what the software system is supposed to do. The basic architecture deals with the question of how the operations related to the requirements are divided into components. When further developing architectures, mechanisms such as interfaces and inheritance can also be added to the design. Thus, the set of requirements and their positioning in the system represents the basic individual, which then evolves as positions of requirements are changed and mechanisms are added. As

there is theoretically an exponential amount of possible designs for a system, the use of genetic algorithms to solve the problem is justified.

The common feature with all the current research activities on applying search algorithms to architecture design is that a reasonably good architecture is needed as a starting point, and the search algorithm merely attempts to improve this architecture with respective to some quality metrics. This means that considerable effort is needed before the algorithm can be executed, and as the base solution can be assumed as a standard one, this also somewhat limits the possible solutions the algorithm can reach. This restriction decreases the innovativeness of the method: if given the algorithm "free hands", it might be able to reach solutions that a human designer might not find at all, but still have a high quality. Thus, an approach that only needs the basic requirements (responsibilities) of the system would both save the initial work and give the algorithm a chance for a more thorough traverse through the possible solutions.

In my thesis, I have further developed the approach of starting only with a set of responsibilities: this approach was first introduced in my Master's thesis [Räihä, 2008]. I have derived a responsibility dependency graph which is then given as input to a genetic algorithm, which will produce a suggestion for the architecture of the given system as a UML class diagram. I begin my thesis by presenting the structure and current evaluation methods of software architectures in Chapter 2. In Chapter 3 I describe meta-heuristic search algorithms, and especially give a thorough presentation of genetic algorithms. The current research involved with the application of meta-heuristic search algorithms in software engineering is surveyed in Chapter 4. In Chapters 5 and 6 I present my implementation, first from a logical point of view, as to how an architecture can be modeled for a genetic algorithm, and then from a practical view by giving a detailed description of the implementation and the evaluation metrics used. Results from tests where the implemented algorithm was used on a model of an electronic home control system are presented in Chapter 7, and in Chapter 8 I present the outcome of this thesis and my concluding remarks.

2. Software architectures

Software architecture is defined by the IEEE Standard 1471-2000 [IEEE, 2000] as "the fundamental organization of a system embodied in its *components*, their *relationships* to each other and to the environment, and the principles guiding its design and evolution". Thus, software architecture defines the general structure of the software. An architecture should always be described or modeled somehow, otherwise it does not exist. In reverse engineering one tries to detect the architecture of software from the source code by looking at what kind of packages it has, and by generating class diagrams from the code. Normally, the architecture of software is designed before the actual implementation, as it is possible to efficiently evaluate the architecture, and thus point out possible weaknesses of the software before beginning the implementation.

The structure of architecture and the evaluation metrics presented in this chapter will be used in Chapter 5, where I present how architectures can be modeled in order to manipulate them with a genetic algorithm, and in Chapter 6, where I discuss the evaluation methods used in the implementation. The studies surveyed in Chapter 4 also use many of the metrics presented here as well as concepts concerning architectural quality.

2.1. The structure of an architecture

As stated, software architecture describes the components of a software and the relationships between these components. We must now consider what can be thought of as a component, and what as a relationship.

A software component is defined as an individual and independent software unit that offers its services through well-defined interfaces [Koskimies ja Mikkonen, 2005]. This definition requires that the topics of dependency, usage and size are also dealt with. Firstly, a component should never be completely dependent of another component. A component can, however, be dependent on services that are provided by some other components, thus requiring an interface to those components. Secondly, a component can be taken to use as a single unit with no regard to other software units if the component is still provided the services it needs. Thirdly, there are no general restrictions to the size of a component. A component can be extremely small, providing only a few simple services, or it can contain a whole application. If the component is very big and forms a significant sub-system within itself, it may be in order to describe the architecture of that single component, although normally an architecture description does not consider what the components entail [Koskimies ja Mikkonen, 2005].

When thinking of object-oriented design, the basic component provides some kind of functionality to the system and consists of classes. Classes can be defined as abstract and they can be inherited from each other. Classes interact with one another by either

straightforwardly calling operations from other classes or through interfaces. The simplest component may only include one class. Because of this close relationship between components and classes, architectures are often described with UML class diagrams. Other components that are often present in the system, but do not provide much functionality, are components such as databases, hardware drivers and message dispatchers.

One of the key points in software engineering is to separate what one wants to accomplish (the functionality provided by components) and how to accomplish it. This is applied to software components in such a way that the implementation of a service that a component provides should be separated from the abstraction of the service: components should not be directly dependent on one another, but on the abstraction of the service that the component provides [Koskimies ja Mikkonen, 2005]. The abstraction is presented as an interface that provides access to services to the components that require the services in question. This corresponds to the idea that interfaces may be either provided or required.

Interfaces include all the information about a service: the service's name, its parameters and their types and the type of the possible result [Koskimies ja Mikkonen, 2005]. Interfaces have developed from abstract classes into their own program units. Abstract classes and interfaces are still interlinked; by inheriting several concrete classes from an abstract class one can thus give several implementations to one interface [Koskimies ja Mikkonen, 2005]. One component or class can also implement several interfaces.

There are several ways for components to interact with one another. Most of these methods are fine-tuned ways of how interfaces are used in order to consider the needs of a specific type of application. I will briefly present these communication methods, as for the purpose of this thesis, it is more important to be aware that such methods exist and possibly recognize them from an architecture design than to know all the ins and outs of these communication methods and to able to actively implement them. I will describe the methods as they are presented by Koskimies and Mikkonen [2005].

Firstly, the interfaces a component provides may be divided into more detailed role-interfaces, each role-interface responding to the special need of the component requiring that interface, instead of keeping all the services of the providing component in one big interface. Secondly, when addressed with the problem of multiple components using each other and thus creating a complex net of dependencies, one can use a mediator to handle the interaction between the components. Thus, all the components only depend on this one mediator, which is often a specialized interface. Thirdly, an even more powerful method than the basic interface is forwarding. This means that the component receiving a request for a service does not provide that service itself, but forwards the request to another component, which then acts on it. Fourthly, the interaction between

components can be based on events. We can now think that asking for a service is the event itself, and providing a service is reacting to the event. The component creating the event is now the source and the component reacting to it is the observer. In this case both components are providing and requesting an interface to communicate with each other: the source component provides an interface through which the observer can register as a service provider, and the observer provides an interface through which its services can be provided.

I end this section with a brief summary. An architecture is based on the idea of components and the relationships between them. Components provide services that other components may need. This results in a dependency between components which is ideally handled with interfaces: the component needing a service requires an interface, which the component offering the service then provides by implementing that interface. How the interface is built, i.e. what kind of communication method is used, depends on the application and its requirements.

2.2. Standard solutions

When designing an architecture, there are some commonly used architecture styles and design patterns that can be used as general guidelines for the architecture. These styles and guidelines all have their positive and negative aspects, so one should think what the main problems in the system are, and then study the implementation of styles and design patterns that are generally known to solve those problems. One does not necessarily need to categorize an architecture as a set of the known styles or patterns, but if it can be categorized, it usually indicates good structure in the architecture.

2.2.1. Design patterns

A design pattern is used to solve a particular problem in the architecture. Design patterns often appear in several parts of an architecture, and one architecture can contain several different patterns. The list of design patterns made by Gamma et al. [1995] is recognized as the current standard in design pattern classification. This list contains over 20 patterns, which can be divided into creational patterns, structural patterns and behavioral patterns. For the purpose of this thesis it is not necessary to introduce them all, and thus only a few of the most common or relevant patterns are described in more detail.

Firstly, from the category of creational patterns, there are the factory method and the abstract factory method, which are common design patterns when one has a lot of components that work together or have a similar purpose. When applying the abstract factory method, an interface should be provided for creating families of related or dependent objects without actually specifying their concrete classes [Gamma et al., 1995]. This means that two or more concrete classes that are responsible for similar objects will implement the same interface, through which these families of objects can be

dealt with. In the factory method an interface is also used for creating an object, but deciding the class that the object represents is left to subclasses [Gamma et al., 1995]. This means that the objects of a certain family all inherit the "base-object" of that family in order to ensure that they contain the required properties.

These design methods are presented together as they are closely linked: abstract factory classes are commonly implemented with factory methods. Although the abstract factory method and the factory method are very commonly used in current architecture design, I can imagine that automatically producing an architecture where such a pattern could be found is a great challenge. These design patterns rely on the recognition of similarities between objects and the ability to group objects by some standards. However, similarities between objects are difficult to express formally, but are rather something that experts can simply see. Thus, to train an algorithm to find such abstract similarities will definitely need fine-tuned definitions of the objects and relations presented to the algorithm.

Secondly, there is the *composite method*, which is a structural pattern, in which objects are composed into tree structures to represent part-whole hierarchies. A composite also allows clients treat individual objects and compositions of objects uniformly [Gamma et al., 1995]. The composite pattern defines hierarchies consisting of primitive objects and composite objects. Primitive objects can form composite objects, which in turn can form more complex composite objects, and so on recursively [Gamma et al., 1995]. Vice versa, all composite objects can be broken down to primitive objects. The composite method goes well with the responsibility based approach used in this thesis, as all responsibilities can be thought of as primitive objects or services, which form composites that other composites use.

Thirdly, there is the *Façade pattern*, which has implemented in this work, and is therefore especially interesting. The Façade is a structural pattern, and its intention is to provide a unified interface to a set of interfaces in a subsystem [Gamma et al., 1995]. This will make the subsystem easier to use through the higher-level interface defined by the Façade. Structuring a system into subsystems reduces complexity, and a common design goal is to minimize dependencies between the defined subsystems [Gamma et al., 1995]. A façade can aid in achieving this goal, and its proper placement can be effectively measured by metrics dealing with complexity and the number of connections between classes.

Finally, I present the *Strategy pattern*, which has also been implemented. The Strategy pattern is a behavioral pattern, which encapsulates an algorithm, thus making it interchangeable and independent of the using client. A Strategy pattern can be applied, e.g., when different variants of an algorithm need to be defined or there are several related classes that only differ in their behavior [Gamma et al., 1995]. From the perspective of responsibilities, a strategy can be seen as well-placed if the responsibility

is highly variable. For example, measuring the position of drapes can be executed with different algorithms that have the same outcome. In this case, a Strategy pattern would very well fit in the system.

As automating the design of an architecture mainly deals with the structure of an architecture, structural patterns are logically the ones that are most likely to be found from the resulting architecture. Thus, structural patterns are the most interesting pattern group from the viewpoint of this thesis. Overall, structural patterns deal with how classes and objects are composed to form larger structures. Structural class patterns commonly solve problems with clever inheritance to achieve interfaces for implementations, and structural object patterns describe how objects can be composed to achieve new functionalities [Gamma et al., 1995]. Other structural design patterns besides the composite pattern are, for example, the *adapter* pattern. In this pattern, an incompatible interface is converted to let such classes work together that could not before because of the "wrong" type of the provided interface. Another example is the *bridge* pattern, which builds a "bridge" between an abstraction and its implementation, so they can vary independently [Gamma et al., 1995].

2.2.2. Architecture styles

Architecture styles have the same purpose as design patterns: they are used to solve a problem in the design of the architecture. It is often difficult to make a difference between design patterns and architectural styles, but the general guideline is that while design patterns are used at a particular subsystem in the architecture, architecture styles solve a problem regarding the whole architecture [Koskimies ja Mikkonen, 2005]. As with design patterns, it is not necessary to go through all possible architecture styles, so only the most interesting ones from this thesis' point of view are described with more detail.

Firstly, I present the *layered architecture*. A layered architecture is composed of levels that have been organized into an ascending order by some principle of abstraction [Koskimies ja Mikkonen, 2005]. This is usually done so that the parts of the system that are closer to the user have a lower level of abstraction than the parts that are closer to the application. Because the levels of abstraction can often be hard to identify, the levels or layers in the architecture are deduced by how different components use services from other components. A higher level in the architecture uses services from a lower level [Koskimies ja Mikkonen, 2005]. However, layered architectures are rarely so straightforward. It is quite common that a layer is simply passed in a service call, and, for example, a service is required at the fifth level that is provided at the third level. It is also possible that a lower layer needs to call a service from an upper layer. This is, however, a sign of a serious problem in the architecture. Layered architectures are very common, and can be used in almost any system [Koskimies ja Mikkonen, 2005]. The layered architecture model encourages a minimized design in terms of dependencies, for in the

ideal case, any layer only depends on layers below itself. This kind of architecture model is also very easy to understand, as it divides the system to subsections at a high level [Koskimies ja Mikkonen, 2005]. The layered architecture is something that is very interesting from my viewpoint and that of thinking through responsibilities. When having a network of responsibilities, we can quite simply begin forming layers by placing the responsibilities that do not depend from any other responsibilities at the bottom layer, and going on until at the top level are the responsibilities that have a very long dependency path behind them.

Secondly, there is the *pipes and filters* architectural style. It consists of processing units (filters) and the connections (pipes) between them that carry the information that needs to be processed. The role of pipes is to passively transport data which the filters will actively process. The pipes and filters architecture is good for the kind of system where the purpose is to mainly develop and process a common dataflow [Koskimies ja Mikkonen, 2005]. To implement the pipes and filters architecture it requires that each processing unit can be implemented independently: a unit can not depend on any of the other processing units, and must only be able to understand the data that is brought to it to process. The simplest form of a pipes and filters architecture is a pipeline architecture, where the data moves straightforwardly from one processing unit to another along a straight "conveyer belt". There are two ways in operating this "conveyer belt", to push or pull. If we choose to push, then the unit that first generates the data will push it to the second unit for processing, which will then continue to push to the next processing unit and so on, until the data reaches the final unit needing the "end product", i.e. the completely processed data unit. If we choose to pull the data, then the final unit needing the data will "pull" data from the processing unit preceding it, which will then call for the data from its preceding unit, and so on [Koskimies ja Mikkonen, 2005]. A pipes and filters architecture can be useful from this thesis's viewpoint if the responsibilities we work with all deal with the same kind of data, and merely have more fine-tuned responsibilities regarding that data, or if they can be arranged in quite a straightforward line, i.e., if the dependency graph does not have any cycles and a unique ending point can be identified.

Finally, an architecture style especially used in this thesis is the *message dispatcher architecture*, where a group of components communicate with each other through a centered message dispatcher. All the components have a common interface that contains all the operations that are needed in order to send and receive messages to and from the dispatcher [Koskimies ja Mikkonen, 2005]. It is important to notice that now the components only communicate with the dispatcher: although they send and receive messages to and from other components, no component can actually "see" the message's path past the dispatcher. Thus, no component actually knows where its messages will end up or where the messages it has received originate from. A message dispatcher

architecture suits well in a situation where the system has a large number of components that need to communicate with each other, but there is not much information of the quality or quantity of the messages sent between components [Koskimies ja Mikkonen, 2005]. A message dispatcher architecture is defined by the set of components communicating with each other, the messages with which the components communicate, the operations with which components react to messages, the rules with which the components and messages are registered to the system, the rules on how the dispatcher forwards messages to components and the model of concurrency [Koskimies ja Mikkonen, 2005].

Other common architecture styles are *service oriented architectures*, such as the *client-server* architecture, where client components ask for the services they need from the server components. Client-server architecture is often thought as a distributed system. Other, more specialized architecture styles are for example the *model-view-controller* architecture or the *interpreter* architecture.

2.3. Evaluating an architecture

When evaluating a software architecture we must keep in mind that the architecture under evaluation is, roughly stated, merely a picture of how the different components are placed in the system and how they depend from one another. Thus, there is no absolute method for evaluating an architecture; just as there is no absolute answer to the question how good exactly a particular architecture is. Currently there are two kinds of methods for software architecture evaluation. Firstly, there are metrics that can be used when one knows the software in detail. These metrics often calculate the cohesion and coupling between classes, so it must be known what kind of operations the classes include, and how they are linked to each other. Secondly, there are methods to evaluate the architecture by the means of using the expertise of software engineers, going through meetings and several iterations when the architecture is broken down to pieces and the analysts attempt to identify all the possible risks that can be related to the suggested solution.

Whatever method is used to evaluate architecture, one thing must be kept in mind: no architecture can be evaluated from an overall point of view. There are different viewpoints or *quality attributes* for an architecture, such as *efficiency* or *performance*, *maintainability*, *reliability*, *security*, *movability*, *usability*, *availability*, *reusability* and *modifiability* [Koskimies ja Mikkonen, 2005]. The actual evaluation of an architecture is the sum of evaluations of a combination of these viewpoints, and it is of course most preferred if as many relevant viewpoints as possible have been considered.

2.3.1. Evaluation using metrics

Evaluating a software architecture using some kind of metrics system is often based on the assumption that we are dealing with object-oriented design. Thus, metrics can be

used for different kinds of calculations of dependencies between and within classes, which can give guidelines on how good a structure the architecture in question has. Rosenberg and Hyatt [1997] define five different qualities that can be measured by metrics for object-oriented design: efficiency*, complexity, understandability*, reusability, and *testability/maintainability*. I will now introduce some metrics suites and definitions that can be used when evaluating object-oriented designs.

The metrics suite by Chidamber and Kemerer [1994] is based on four principles that rule object-oriented design process: identification of classes (and objects), identification of semantics of classes (and objects), identification of relationships between classes (and objects) and implementation of classes (and objects). Based on these principles, Chidamber and Kemerer [1994] present a metrics suite that consists of six different metrics: *weighted methods per class* (WMC), *depth of inheritance tree* (DIT), *number of children* (NOC), *coupling between object classes* (CBO), *response for a class* (RFC), and *lack of cohesion in methods* (LCOM).

The WMC metric is defined as the sum of complexities of the methods within a class. If all methods are equally complex, this is simply the amount of methods in a class. It predicts how much time and effort is required to develop and maintain the class, how much the children of the class are impacted by the class and how general the class is [Chidamber and Kemerer, 1994]. These aspects relate to quality attributes such as maintainability and reusability. Rosenberg and Hyatt [1997] point out that WMC also indicates understandability.

DIT is self-defined as it is the length from a class node to the root of the inheritance tree where the node is. If the class does not inherit any class, then DIT is zero. The deeper a class is in a hierarchy, the harder it is to predict its behavior, the more complex the design will most likely become, and the greater the potential reuse for inherited methods [Chidamber and Kemerer, 1994]. Thus, DIT predicts negative aspects of complexity and maintainability but a positive aspect of reusability. According to Rosenberg and Hyatt [1997], DIT primarily evaluates efficiency and reusability, but can also be used as an indicator for understandability and testability.

NOC is as clear as DIT as it calculates how many classes inherit the class in question. It also predicts good reusability, but a high value warns of improper abstractions of the parent class and indicates that a good deal of testing should be done to the methods of the class [Chidamber and Kemerer, 1994]. In addition to testability, NOC evaluates efficiency and reusability [Rosenberg and Hyatt, 1997].

CBO is defined as the number of classes to which the class in question is coupled, i.e., CBO for class A is |B| + |C|, where B is the set of classes that class A depends on, and C is the set of classes that depend on class A (where |X| stands for the cardinality of X). A high CBO value indicates poor reusability, modularity and maintainability, and is

usually a sign of need for excessive testing [Chidamber and Kemerer, 1994]. CBO can also be used as an evaluator for efficiency [Rosenberg and Hyatt, 1997].

RFC is defined as the size of the *response set* (RS) for the class, when the RS is the union of the set of all methods in the class and the set of methods called by the methods in the class. RFC contributes mainly in bringing out testing issues, but it also indicates complexity [Chidamber and Kemerer, 1994]. According to Rosenberg and Hyatt [1997], RFC evaluates understandability, maintainability and testability.

Finally, LCOM measures in what extend methods within the same class use the same instance variables. LCOM is a count of method pairs with a similarity of zero, i.e., they have no instance variables in common, minus the count of method pairs with a similarity that is not zero. Cohesiveness is very desirable, as it promotes encapsulation; classes with low cohesion should most probably be divided into two or more subclasses, and low cohesion also indicates high complexity [Chidamber and Kemerer, 1994]. In addition, LCOM evaluates efficiency and reusability [Rosenberg and Hyatt, 1997].

In addition to the metrics by Chidamber and Kemerer, Rosenberg and Hyatt [1997] present two additional metrics for evaluation at the method level, *cyclomatic complexity* (CC) and *size*. CC is used to evaluate the complexity of an algorithm in a method. Quite logically, CC measures mainly complexity, but is also related to all the other quality attributes. The size of a method can be measured by several ways, e.g., by lines of code or the number of statements. It evaluates mainly understandability, reusability and maintainability.

A popular metric when dealing with the software or module clustering problem is the *modularization quality* (MQ). There are several versions of this metric, but it is always some kind of a combination of coupling and cohesion metrics, calculating the *inter*- and *intra-connectivities* between and within clusters, respectively. A high MQ value indicates high cohesion and low coupling. One version of the MQ metric is presented by Doval et al. [1999], who begin by defining the intra-connectivity A_i of cluster i as

$$A_i = \frac{\mu_i}{N_i^2},$$

where N_i is the number of components and μ_i is the number of relationships to and from modules within the same cluster. A_i is 0 when no module is connected to another module within the cluster, and 1 when each module in the cluster is connected to every module in the same cluster. Inter-connectivity $E_{i,j}$ between clusters i and j, consisting of N_i and N_j components, respectively, with ε_{ij} relationships between the modules of both clusters, is defined as

$$E_{i,j} = 0 \quad \text{, if } i = j\text{, and}$$

$$E_{i,j} = \frac{\varepsilon_{ij}}{2N_i N_j} \quad \text{,if } i \neq j$$

[Doval et al., 1999]. MQ is now a combination of these connectivity measures: when a *module dependency graph* is partitioned into k clusters,

$$\mathrm{MQ} = A_i \quad \text{, if } k = 1 \text{, and}$$

$$\mathrm{MQ} = \frac{\sum_{i=1}^{k} A_i}{k} - \frac{\sum_{i,j=1}^{k} E_{i,j}}{\frac{k(k-1)}{2}} \quad \text{, if } k > 1.$$

The work by Doval et al. [1999] and the module clustering problem in which this metric is used, is presented in Chapter 4.

When defining what a software architecture is, the principles guiding its evolution were mentioned. Thus, it is natural that there should be metrics to evaluate the evolution and refactoring of an architecture. Mens and Demeyer [2001] present such evolution metrics, the main metric being the distance between classes. This metric is very flexible, as the distance it measures depends on what is needed, i.e., how far two classes are from each other when considering, e.g., the number of methods, number of children or depth of inheritance tree. The distance between classes is defined so that, when p(x) is the property that is measured from class x, the distance between classes x and y is

$$\mathrm{dist}(x; y) = 1 - \left| \frac{p(x) \cap p(y)}{p(x) \cup p(y)} \right|.$$

Large distances between classes can indicate a complex system. Mens and Demeyer [2001] also discuss the emphasis of abstract methods and abstract classes in a system, and point out that all abstract classes should be base classes.

Sahraoui et al. [2000] present a list of inheritance and coupling metrics, where the simplest metrics are NOC, CBO and number of methods (NOM), which is a simpler form of WMC, but the rest are more specialized extensions of the metrics presented earlier. These include metrics such as *class-to-leaf depth* (CLD), *number of methods overridden* (NMO), *number of methods inherited* (NMI), *number of methods added* (NMA), *specialization index* (SIX), *data abstraction coupling* (DAC'), *information-flow-based inheritance coupling* (IH-ICP), *other class-attribute import coupling* (OCAIC), *descendants method-method export coupling* (DMMEC) and *others method-method export coupling* (OMMEC). By analyzing the results given by these metrics, the following operations can be administered to the system: creating an abstract class, creating specialized subclasses and creating an aggregate class [Sahraoui et al., 2000].

Du Bois and Mens [2003] use a combination of the metrics defined above (number of methods, CC, NOC, CBO, RFC and LCOM) in order to administrate a selection of refactoring operations (extracting a method, encapsulating a field and pulling up a method) to a system. Thus, this suite of metrics can be used to both evaluate the existing system and to use those results to evolve a system. As can be seen, the metrics suite presented by Chidamber and Kemerer [1994] acts as a good base for evaluating architectures and evolving new metrics by using their six metrics as a starting point.

Another way of measuring is related to the stable/instable and abstract/concrete levels of the system, which is used by Amoui et al. [2006]; this is based on simply counting the number of certain types of classes and dependencies.

Losavio et al. [2004] present ISO quality standards for measuring architectures. This model is somewhere in between pure metrics and evaluation using human expertise, which is discussed further on. The ISO 9126-1 quality model's characteristics are *functionality*, reliability, usability, efficiency, maintainability and *portability* [Losavio et al., 2004] – a list quite similar to the one presented by Rosenberg and Hyatt [1997]. In the ISO model, the characteristics of quality are refined into sub-characteristics, which are again refined to attributes, which are measured by metrics. Thus, the model needs human expertise in making the refinements, but the end result is a measurable value related to the architecture. As the characteristics have from three to five separately measured sub-characteristics each, it is not practical to go through them all in the scope of this paper. The most interesting quality measures being efficiency and maintainability, I will now present some example metrics for measuring the sub-characteristics of these.

Efficiency is divided into time behavior, resource behavior and compliance. Let us now investigate how time behavior is measured. Time behavior means the capability of the software product to provide appropriate response time, processing time and throughput rates under stated conditions [Losavio et al., 2004]. To measure this, one must first identify all the components involved with functionality and the connections between them. The attribute is then computed as the sum of the time behaviors of the components and the time behaviors of the connections. The time behavior of a component or a connection depends on the stimulus/event/functionality and the path taken in the architecture to respond to a stimulus for a given functionality [Losavio et al., 2004].

Maintainability is sub-categorized into analyzability, changeability, stability, testability and compliance. Let us take changeability and stability as examples. Changeability is defined as the capability of the software to enable implementation of modifications, and stability is defined as the capability of the software to avoid unexpected effects from modifications of the software. In order to measure these (and testability), two additional sub-characteristics need to be added to the ISO model framework at architectural level: coupling and modularity [Losavio et al., 2004]. The computations for changeability and stability need to be made for each couple of connected components on the number of incoming/outgoing messages, and for each component on the number of components depending on that component.

The examples of time behavior, changeability and stability are still something that can be seen as metrics: the resulting values are something that can be computed, albeit that it might not be easy. However, there are many sub-characteristics in the ISO 9126-1 quality model when the "counting rule" does not contain any calculation and thus, the

result is not numeral. For example, functionality contains sub-characteristics such as interoperability and security, where the attribute that is to be "measured" is the presence of a certain mechanism. Thus, to "count" the attribute, one needs to identify whether the mechanism is present in the system [Losavio et al., 2004]. This is another point (in addition to the redefining steps) where the ISO quality model can be seen as relying more on human expertise than being a set of metrics that can be used for automated evaluation of an architecture.

2.3.2. Evaluation using human expertise

When evaluating an architecture there are three questions that should be answered in the evaluation. Firstly, is the designed architecture suitable for the system in question? Secondly, if there are several options to choose an architecture from, which is the best for the particular system and why? Thirdly, how good will different *quality attribute requirements* be? [Koskimies ja Mikkonen, 2005]

These questions alone demonstrate the difference between using metrics to give values to quality requirements and using human expertise: no metric can answer the question "why" when discussing the positive and negative points of different architectural options. Metrics may also give very good values to individual quality requirements, but as a whole the architecture may not be at all suitable for the system in question. Hence, although metrics can aid in architecture evaluation and are basically the only way of automated evaluation, they cannot replace the evaluation of experts.

The most widely used and known method for architecture evaluation is the Architecture Tradeoff Analysis Method (ATAM) by Kazman et al. [2000]. Other known architecture evaluation methods are the Maintenance Prediction Method by Jan Bosch, which concentrates in evaluating maintainability, and the Software Architecture Analysis Method developed in the Software Engineering Institute of Carnegie-Mellon University, which is mainly used for evaluating quality attributes that are related to modifiability [Koskimies ja Mikkonen, 2005]. As ATAM is the only method that can be used to evaluate all quality attributes, it is the one I will go into with more detail.

The main points of ATAM are to elicit and refine a precise statement of the key quality attribute requirements concerning the architecture, to elicit and refine precise designing decisions for the architecture, and based on the two previous goals, to evaluate the architectural design decisions to determine if they fulfill the quality attribute requirements satisfactorily [Kazman et al., 2000]. The ATAM uses scenarios in order to analyze whether the architecture fulfills all the necessary requirements and to see risks involved in the architecture. The ATAM proceeds in nine steps: presenting the method for the group of experts, presenting business drivers, presenting the architecture, identifying architecture approaches, generating quality attribute utility tree, analyzing architecture approaches, brainstorming and prioritizing scenarios, again analyzing architecture approaches, and finally presenting the results [Kazman et al., 2000]. The

steps where we can say that the architecture is evaluated as in how good it is in the ATAM are when the quality attribute utility tree is generated, architecture approaches are analyzed and scenarios are brainstormed, so I will now concentrate on these steps.

When the architecture has been presented and the architecture styles have been identified, a quality attribute utility tree is generated. This is done by eliciting the quality attributes that relate to the particular system and then breaking them down to the level of scenarios, which are shown with stimuli and responses and prioritized [Kazman et al., 2000]. For each quality approach, the quality factor is divided into sub-factors. For example, modifiability could be divided into GUI-modifications and algorithmic modifications. For each of these sub-factors, detailed scenarios are described in order to see how the sub-factor in question affects the architecture [Kazman et al., 2000]. For example, GUI-modifications may have a scenario that if a new feature is added to the application, the feature should be visible in the GUI within one day. These scenarios are then prioritized according to how relevant they are to the system, how likely they are to happen, and naturally, how critical they are for the quality attribute in question. Based on the utility tree, experts can now concentrate on the high priority scenarios and analyze architectural approaches that satisfy these scenarios.

While the utility tree is manufactured by a smaller group of specialized architecture experts, a scenario brainstorming session involves all the stakeholders involved in the project. The purpose of this session is to gather all the possible ideas and scenarios that relate to the system and should be considered in the architecture [Kazman et al., 2000].

After the brainstorming of scenarios, all possible scenarios should be documented either as a result of the utility tree or the brainstorming sessions. The architecture experts may now reanalyze the architecture styles that have been documented and discussed, and perhaps even suggest a completely different solution if the brainstorming session brought up many unexpected scenarios or the prioritizing of quality attributes was very different from the one in the utility tree.

After all the steps of the ATAM, the outcomes of this method will include the architectural approaches documented, the set of scenarios and their prioritization, the set of attribute-based questions, the utility tree, risks and sensitivity and tradeoff points in the architecture [Kazman et al., 2000].

As can be seen, the ATAM relies purely on human expertise, and the evaluation of architecture happens while the architecture is actually being developed. Some basic architectural approaches are first presented based on the known structure of the system, and as the quality attributes requirements of the system become clearer, the architecture undergoes several iterations of analysis, while the architecture is being refined and different approaches may be considered. The "goodness" of the architecture can be defined and measured by how well it satisfies the quality attribute requirements and how "easily" it responds to the scenarios related to the quality attributes.

3. Meta-heuristic search algorithms

Many sub-problems of several software engineering problems are known to be NP-hard. For example software clustering, which is a special case of the general graph partitioning problem, is NP-hard. In such cases, non-deterministic search algorithms are useful, as they are often capable of finding good enough solutions from a large search space. The characteristics that enable such good results are that they do not need to go through all the possible solutions; yet by being non-deterministic, it is possible to recover from a search path that seemed good in the beginning, but resulted in a bad solution.

There are certain terms that are common to most search algorithms; the *neighborhood* and *fitness* of a solution. Each solution can be regarded as a point in the search space that needs to be explored. The neighborhood of a solution is the set of all available solutions that can be reached with one technique-specific move from the current solution. The concept of neighborhood is especially used in local search algorithms, such as hill-climbing, tabu search and simulated annealing. The fitness of a solution indicates how good the solution is. In rare cases, when the optimum is known, one tries to get the fitness value as close to the optimum as possible. Since this is hardly ever the case, it is usually attempted to maximize or minimize a fitness function.

For the purpose of this thesis, it is necessary to understand how search algorithms operate in order to understand the underlying concepts of the research presented in Chapter 4, and the implementation presented in Chapters 5 and 6.

3.1. Genetic algorithms

Genetic algorithms were invented by John Holland in the 1960s. Holland's original goal was not to design application specific algorithms, but rather to formally study the ways of evolution and adaptation in nature and develop ways to import them into computer science. Holland's 1975 book *Adaptation in Natural and Artificial Systems* presents the genetic algorithm as an abstraction of biological evolution and gives the theoretical framework for adaptation under the genetic algorithm [Mitchell, 1994].

In order to explain genetic algorithms, some biological terminology needs to be clarified. All living organisms consist of cells, and every cell contains a set of *chromosomes*, which are strings of DNA and give the basic information of the particular organism. A chromosome can be further divided into *genes*, which in turn are functional blocks of DNA, each gene representing some particular property of the organism. The different possibilities for each property, e.g. different colors of the eye, are called *alleles.* Each gene is located at a particular *locus* of the chromosome. When reproducing, *crossover* occurs: genes are exchanged between the pair of parent chromosomes. The offspring is subject to *mutation*, where single bits of DNA are changed. The *fitness* of an organism is the probability that the organism will live to reproduce and carry on to the next generation [Mitchell, 1996]. The set of individuals at hand at a given time is called a *population*.

Genetic algorithms are a way of using the ideas of evolution in computer science. When thinking of the evolution and development of species in nature, in order for the species to survive, it needs to meet the demands of its surroundings. Such evolution is achieved with mutations and crossovers between different individuals, while the fittest survive and are able to participate in creating the next generation.

In computer science, genetic algorithms are used to find a good solution from a very large solution set, the goal obviously being that the found solution is as good as possible. To operate with a genetic algorithm, one needs an encoding of the solution, an initial population, mutation and crossover operators, a fitness function and a *selection operator* for choosing the survivors for the next generation.

3.1.1. Encoding

As stated, the basis of genetics in nature is a chromosome. When applying this thought to computer science and genetic algorithms, each individual in the search space, i.e. each solution to the problem at hand, needs to be encoded so that it can be thought of as a chromosome. The most common and traditional way of doing this is to use a bit vector, i.e., a string of ones and zeros [Mitchell, 1996]. Thus every bit in the chromosome represents a gene in that locus, the alleles being one and zero. This has the advantage of being very easy to interpret. Usually such encoding is used for combinatorial problems. For example, if we want to get as close to a value x by summing numbers from one to twenty, and using the minimal amount of numbers in the sum. We can now use a 20-bit chromosome, where each number is represented in its respective locus in the chromosome. If the allele in that locus is 1, the number is included in the sum, if 0, then not. Another way of using bits is when one is dealing with large scale numbers with tens or hundreds of decimals. The bits can thus be used to give a binary representation of such a number.

Another common way of forming a chromosome is to have a string of natural numbers. Such solutions are good for permutation problems, for example the traveling salesman problem (TSP) [Michalewicz, 1992]. The nodes in the graph are numbered and the travel route will be the order of the nodes in the chromosome. By mutations the places of the nodes can be switched, thus reforming the route.

Strings of bits are the most traditional way of encoding a chromosome, and some sources call only such solutions pure genetic algorithms. In fact, there can be as many ways to encode a chromosome, numeric and non-numeric, as there are algorithm developers, as long as the same developer can keep in hand the required mutations and crossovers so the solutions stay “legal”. Purists call genetic algorithms that use such advanced coding styles evolutionary programs, rather than pure genetic algorithms.

3.1.2. Mutations

Mutations are a way of creating new individuals from the population at hand by administering a minor change to one of the existing individuals by changing alleles in a random locus. When the chromosome is represented by a bit vector, a basic mutation is to change one bit from 0 to 1 or vice versa. For example, we could have a bit string 001100. By mutating this string in its third locus the result would be 000100. When the string contains natural numbers, a mutation could be to switch the places of two numbers. Whatever the mutations are, the result should always still be a legitimate individual, i.e., it should solve the defined problem. The more complex the encoding of the chromosome is, the more there usually are possible mutations that can be applied and the mutations may become more complex. It is also possible to have a separate "correction mutation" that will check the chromosome after a mutation to see that it still solves the problem that it is supposed to. If the mutation has caused the chromosome to become unnatural, i.e., it does not belong to the solution space anymore, corrective actions will take place. Such actions don't necessarily just revert the mutation that caused the problem, but might do even bigger changes to the chromosome.

There is always a defined probability how likely it is that the mutation in question would be applied to an individual. This is called the *mutation probability* or *mutation rate* [Mitchell, 1996]. As in nature, mutations are unwanted in most cases, thus the mutation probabilities are usually quite low. The mutation probabilities should be thought of carefully, as both too high and too low probabilities will result in problems. If the mutation probability is too high, one will end up wandering aimlessly in the solution space as the chromosomes mutate in high speed. If the mutation probability is too low, then the population stays very similar from one generation to the next, i.e., there are not enough of variation between individuals to ensure finding good enough solutions.

3.1.3. Crossover

The crossover operator is applied to two chromosomes, the parents, in order to create two new chromosomes, their offspring, which combine the properties of the parents. Like mutations, the crossover operator is applied to a certain randomly selected locus in the chromosome. The crossover operator will then exchange the subsequences before and after the selected locus to create the offspring [Mitchell, 1996; Michalewicz, 1992]. As an example, suppose we have chromosomes $c_1c_2c_3\ldots c_n$ and $b_1b_2b_3\ldots b_n$, and the selected locus is in position k, $k<n$. The offspring would then be $c_1c_2\ldots c_kb_{k+1}b_{k+2}\ldots b_n$ and $b_1b_2\ldots b_kc_{k+1}c_{k+2}\ldots c_n$. It is also possible to execute a multi-point crossover, where the crossover operator is applied to several loci in the parent chromosomes. Using the same parents as in the previous example and a three-point crossover to loci i, j and k, the resulting offspring would now be $c_1c_2\ldots c_ib_{i+1}\ldots b_{j-1}b_jc_{j+1}\ldots c_{k-1}c_kb_{k+1}b_{k+2}\ldots b_n$ and $b_1b_2\ldots b_ic_{i+1}\ldots c_{j-1}c_jb_{j+1}\ldots b_{k-1}\ b_kc_{k+1}c_{k+2}\ldots c_n$.

The crossover operator has a *crossover probability* or *crossover rate*, which determines how likely it is for the crossover operator to be applied to a chromosome, so that the probability of a crossover increases in some correlation with the fitness-value of the chromosome. For the crossover probability, there are two differences to the respective probability of the mutations. Firstly, the crossover probability is in relation to the fitness of the chromosome. The fitter the individual is, i.e., the more likely it will survive to the next population, the bigger the chance it should be that its offspring will also have a high fitness-value. Whether the offspring will actually have a higher fitness value depends on how well the crossover-operation is defined. The most desirable outcome is always that the crossover would generate chromosomes with higher fitness-values than their parents or at least have a big probability of doing so. Unfortunately, this can not always be guaranteed. Secondly, the term crossover rate is not always the same as crossover probability. In the case of a multi-point crossover operator, the crossover probability determines the likelihood of the operation while the crossover rate distinguishes the number of points at which the crossover takes place. [Mitchell 1996].

The *building block hypothesis* states that a genetic algorithm combines a set of sub-solutions, or building blocks, to obtain the final solution. The sub-solutions that are kept over the generations generally have an above-average fitness [Salomon, 1998]. The crossover operator is especially sensitive to this hypothesis, as an optimal crossover would thus combine two rather large building blocks in order to produce an offspring with a one-point crossover.

Where and how the crossover operator is used varies based on the application and developer. Mitchell [1996] and Reeves [1995] consider that the selection operator always selects parents, and thus all chromosomes selected to the next generation are subjected to the crossover operator. The crossover probability then determines whether a real crossover is performed, or whether the offspring are actually duplicate copies of the actual parents. Michalewicz [1992], on the other hand, applies the crossover probability when after selecting a new generation. The crossover probability of a chromosome is compared to the “limit” probability defining whether the crossover is performed. Chromosomes subjected to crossover are randomly paired, and offspring produced – in this approach the crossover does not produce any duplicates. Both approaches replace the parents with the resulting offspring.

For the rest of the thesis I have chosen to follow mostly Michalewicz’s views, i.e., the crossover probability is used purely to choose parents from the existing population. I have chosen a slightly different approach however, by not replacing the parent chromosomes with the offspring, but keeping both the parents and the offspring in the population. I justify this with keeping with the concept of biology; parents rarely die off because of producing offspring.

3.1.4. Fitness function

In order to evaluate how good the different individuals in the population are, a fitness function needs to be defined. A fitness function assigns each chromosome a value that indicates how well that chromosome solves the given problem.

Genetic algorithms are usually used in an attempt to optimize complex multivariable functions or non-numerical data [Mitchell, 1996]. Naturally, the more complex the problem, the more complex the fitness function usually becomes. When the algorithm is dealing with numerical data the fitness function can be detected from the actual optimizing problem, albeit that the problem is intricate. Thus, the most difficult fitness functions are the ones needed to evaluate non-numerical data, as the developer must find other metrics or ways to find a numerical evaluation of non-numerical data. An example of this is provided by Mitchell [1996], who describes the problem of finding the optimal sequence of amino acids that can be folded to a desired protein structure. The acids are represented by the alphabet {A, …, Z}, and thus no numerical value can be straightforwardly calculated. The used fitness function calculates the energy needed to bend the given sequence of amino acids to the desired protein.

3.1.5. Selection operator

Since the number of individuals in a population always increases with the result of crossovers, a selection operator is needed to manage the size of the population. The selection operator will determine the individuals that will survive to the next generation, and should thus be defined so that the ones with the best fitness are more likely to survive in order to increase the average fitness of the population.

The simplest way of defining a selection operator is to use a purely *elitist selection*. This selects only the "elites", i.e., the individuals with the highest fitness. Elitist selection is easy to understand and simple to implement; one can simply discard the weakest individuals in the population. However, elitist selection isn't the best choice, as it may very well result in getting stuck to a local optimum.

Another and a more common way of defining the selection operator is to use a *fitness-proportionate* selection, which can be implemented with a "roulette-wheel" sampling [Mitchell, 1996; Michalewicz, 1992; Reeves, 1995]. Here, each individual is given a slice of the "wheel" that is in proportion to the "area" that its fitness has in the overall fitness of the population. This way, the individuals with higher fitnesses have a larger area in the wheel, and thus have a higher probability of getting selected. The wheel is then spun for as many times as there are individuals needed for the population.

In general, a fitness-proportionate selection operator can be defined by assigning a probability of surviving, p_s, to each individual, with coefficient f_s to ensure that individuals with better fitness values are more likely to be selected. Comparing the actual values given by the fitness function is difficult, so these actual values should be used as coefficients with caution. However, by examining the order of fitnesses it is possible to

employ the idea of survival of the fittest by having a linear relation between the order of fitness and the coefficient.

A common selection operator is a crossing of the two methods presented above; the survival of the very fittest is guaranteed by choosing the best individual with elitist methods, while the rest of the population is selected with the probabilistic method in order to ensure variety within the population. Some approaches also use the *tournament technique* to select the next generation [Blickle, 1996; Seng et al., 2005].

As mentioned in the presentation of the crossover operator, there are different approaches to how to use the selection operator. Mitchell [1996] and Reeves [1995] consider that the selection operator selects the individuals that are most likely to reproduce, i.e., become parents. Michalewicz [1992] uses the selection operator in order to find the fittest individuals for the next generation. Both approaches keep the same selection probabilities for all individuals during the entire selection process, i.e., an individual with a high fitness value may be selected to the next population more than once.

For the rest of the thesis, as with the crossover operator, I follow mostly with Michalewicz's views. However, also with selection, I take a different path by not allowing multiple selections of the same chromosome. When applying this to the roulette-wheel, the wheel is adjusted after every spin by removing the area of the selected individual, and recalculating the areas for the remaining population so that they keep in proportion to itself.

3.1.6. Executing a genetic algorithm

The operation of a genetic algorithm can be examined through an example of the knapsack problem. Say we have five items, each with a weight w_i and a volume of v_i. The goal is to fit as much weight as possible to a backpack with a limited volume. The candidate solutions can now be represented by a vector of 5 bits, where 0 represents not picking the item represented by that gene, and 1 represents picking it. The items can be arranged by volume, weight, or any other way, as long as it is clear which weight and volume are connected to which index of the vector, i.e. which item is represented in which locus. Suppose that in this example the items are as follows

locus	*w*	*v*
1	5	1
2	6	3
3	10	7
4	4	9
5	9	12 .

Firstly, it must be agreed what the population size should be, and then initialize the first population. If possible, some kind of heuristic method should be used when

generating the initial chromosomes, so that some fitness is already ensured in the first population. If no heuristic can be applied to the problem in question, the chromosomes are randomly generated, while keeping in mind that they must be valid. We may now have a population of, say, five individuals, and the individuals can be the following:

A 00010
B 01100
C 10100
D 11100
E 10001.

By setting the target volume to 20, the fitness function $f(x)$ can now be defined as

$$f(x) = \sum w(\mathrm{x}),\ \sum v(\mathrm{x}) \leq 20.$$

Thus the fitnesses for the initial population would be: f(A) = 4, f(B) = 16, f(C) = 15, f(D) = 21 and f(E) = 14.

Secondly, the population is subjected to the crossover operator. The crossover probability for each chromosome is now pf_c, p being the “standard” probability of a crossover operation and f_c the fitness coefficient. Suppose that chromosomes B and E are subjected to crossover, with the crossover point being in locus 2. The resulting offspring would then be BE = 01001 and EB = 10100, with fitnesses f(BE) = 15 and f(EB) = 15.

Thirdly, the population is subjected to the mutation operator with the probability p_m. For this example, we define the mutation operator as the traditional one: changing the bit value from 0 to 1 or from 1 to 0. We now assume that chromosome A is subjected to mutation in locus 1, thus the result would be A’ = 10010, with f(A’) = 9. It is important to notice that in this example we have a risk of achieving an illegal chromosome as the result of a mutation. Since we have a volume limit of 20, no chromosome should represent a set of items if the sum of their volumes surpasses 20. We now have two options: either checking whether the mutation is possible before performing it or constructing a correcting operator which will go through the results of mutations. Let us assume that chromosome D is subjected to mutation in locus 5, producing the chromosome D’ (11101). The sum volume of items represented by chromosome D is 11 and since the item represented by locus 5 has a volume of 12, the total volume would now become 23, which isn’t allowed. If we choose to check each mutation beforehand, the mutation in chromosome D simply wouldn’t happen, as it would be considered unnatural.

Constructing a corrective operator is not straightforward. One example of a corrective operator would be the following. Say chromosome D has been subjected to mutation and the resulting chromosome D’ is now checked with the corrective operator.

First, the volume of the items represented by the chromosome is calculated, the sum of volumes being 23. After that, the operator begins correcting the chromosome by simply removing items in order to achieve a legal individual. The operator starts from the first locus and systematically changes ones to zeros until the sum of volumes is once again within acceptable limits. So, the operator would first achieve chromosome D'' (01101), the sum volume of which is 22. Since the limit is still surpassed, another iteration is needed. We now get D'''(00101), the sum volume of which is 19. The chromosome D''' is an acceptable individual and will replace the original chromosome D. The fitness of chromosome D''', *f*(D'''), is 19, which is lower than the fitness of the original chromosome, but still above the average fitness in the population.

Finally, the population is subjected to the selection operator, i.e., the individuals surviving to the next generation are chosen. The size of the population is now 7, with the individuals A', B, C, D''', E, BE, and EB. In this example we use a purely elitist selection operator, which simply drops two of the weakest individuals; they do not survive to the next generation. Thus the next population will be B, C, D''', BE and EB.

The population will go through as many generations of crossovers, mutations and selections as is needed to achieve a good enough fitness value, or it is decided that the generation number is high enough.

3.2. Tabu search and simulated annealing

While genetic algorithms use mutations and crossovers to constantly generate new solutions, other meta-heuristic search algorithms have their own methods of trying to get out of local optimums and reach the global optimum of the search space. I will now briefly describe the methods of tabu search and simulated annealing.

3.2.1. Tabu search

The word *tabu* or *taboo* is understood as something strictly forbidden and unacceptable. Tabu search is named such as it proceeds by setting barriers or restrictions to guide the search process. These restrictions operate, as Reeves [1995] describes, "in several forms, both by direct exclusion of certain search alternatives classed as 'forbidden', and also by translation into modified evaluations and probabilities of selection". Tabu search is seen as a sequence of moves from one possible solution to the best available alternative [Clarke et al., 2003]. The search technique relies on flexible memory and a set of intellectually chosen principles of problem solving. By remembering past search moves from several iterations and combining that information to the problem solving principles, the search algorithm is able to see what directions are indeed tabu in the present situation.

When administering the tabu search, one starts from a random point x in the search space. Next, the set of moves that are possible to perform at that point are determined, the resulting set being the neighborhood of the current solution, $N = \{x_1, x_2, \ldots, x_n\}$. The

tabu rules are then applied to N, which is now reformulated to the set of available moves, $A = N \setminus T$, T being the set of rules that are declared tabu. In some special cases, a move that is originally tabu, may become available if all the other available moves aren't satisfactory. The best available move x_k from set A is then chosen. [Clarke et al., 2003]

The tabu rules and ways of determining the neighborhood of a solution vary greatly between problems and applications. The common characteristics in tabu moves are recency and repetition, i.e., moves that have recently been done or have been repeated above the average amount are very likely to be declared tabu.

3.2.2. Simulated annealing

Simulated annealing is originally a concept in physics. It is used when the cooling of metal needs to be stopped at given points where the metal needs to be warmed a bit before it can resume the cooling process. The same idea can be used to construct a search algorithm. At a certain point of the search, when the fitness of the solution in question is approaching a set value, the algorithm will briefly stop the optimizing and revert to choosing a solution that is not the best in the current solution's neighborhood. This way getting stuck to a local optimum can effectively be avoided. Since the fitness function in simulated annealing algorithms should always be minimized, it is usually referred to as a *cost function* [Reeves, 1995].

Simulated annealing optimally begins with a point x in the search space that has been achieved through some heuristic method. If no heuristic can be used, the starting point will be chosen randomly. The cost value c, given by cost function E, of point x is then calculated. Next a neighboring value x_1 is searched and its cost value c_1 calculated. If $c_1 < c$, then the search moves onto x_1. However, even though $c \leq c_1$, there is still a small chance, given by probability p that the search is allowed to continue to a solution with a bigger cost [Clarke et al., 2003]. The probability p is a function of the change in cost function ΔE, and a parameter T:

$$p = e^{-\Delta E/T}.$$

This definition for the probability of acceptance is based on the law of thermodynamics that controls the simulated annealing process in physics. The original function is

$$p = e^{-\Delta E/kt},$$

where t is the temperature in the point of calculation and k is Boltzmann's constant [Reeves, 1995].

The parameter T that substitutes the value of temperature and the physical constant is controlled by a *cooling function C*, and it is very high in the beginning of simulated annealing and is slowly reduced while the search progresses [Clarke et al., 2003]. The actual cooling function is application specific.

If the probability p given by this function is above a set limit, then the solution is accepted even though the cost increases. The search continues by choosing neighbors and applying the probability function (which is always 1 if the cost decreases) until a cost value is achieved that is satisfactory low.

4. Search algorithms in software engineering

Search algorithms have been used widely in different fields of research, such as engineering, business and financial and economic modeling [Clarke et al., 2003], and recently there has been an increasing interest in implementing search algorithms in software engineering as well. This particular field of research is known as search-based software engineering. The areas where search algorithms are used can be divided into four categories [Rela, 2004]: analysis, design, implementation and testing. In this chapter I will explore how search algorithms are used in different areas of software engineering, with an emphasis on software design. The research is presented from the algorithmic viewpoint, accenting how fitness functions are defined and how the problem is modeled for the algorithm.

4.1. Search algorithms in software design

4.1.1. Software clustering

Software clustering or module clustering is a software engineering problem that is most related with software architectures. The goal is to find the best grouping of components to subsystems, i.e., the best clusters of an existing software system.

One way of representing a software system so that the representation is both language independent and "presentable" to a search algorithm, is to transform the structure of the system into a directed graph *G*. A partition of the graph *G* is a set of non-overlapping clusters that cover all the nodes in the graph, and the goal is to partition the graph so that the clusters represent meaningful subsystems. There are several viewpoint to defining the graph *G*, e.g. by considering modules and their relationships, object creation, runtime method invocation or generating a module dependency graph [Clarke et al., 2003].

When defining a fitness function for the clustering problem, the main question to be answered is what constitutes a good partition of the software structure graph. The goodness of a partition is usually measured with a combination of cohesion and coupling metrics, one of the most popular metric being the modularization quality MQ, introduced in Chapter 2, which combines these two metrics.

Clarke et al. [2003] present three different ways of dealing with the clustering problem: hill-climbing, hill-climbing with simulated annealing and genetic algorithms. The hill-climbing algorithm begins with a random partition m of the graph G, where nodes represent modules in the system. The neighboring partitions m_i (the neighborhood being as defined in Chapter 3) are then examined in order to find a better rearrangement of the original partition. If a better solution m_k is found, i.e. $MQ(m_k) > MQ(m)$, m_k is stored as the best partition found so far. The process is iterated until the neighborhood of the best found partition does not contain any partition with a better fitness value. The

hill-climbing solution can be varied by adjusting when it moves onto the next partition: does it select the first solution with a bigger MQ-value, does it go through all the neighboring solutions or does it search a set minimum amount of neighboring solutions. The hill-climbing search technique can be associated with a cooling function used with simulated annealing. Clarke et al. [2003] note that giving the algorithm this opportunity to momentarily accept worse solutions fitness-wise has shown an improvement in performance without jeopardizing the quality of the solutions.

Using a genetic algorithm for module clustering is quite straightforward: the main challenge is to find a suitable encoding, after which traditional mutation and crossover operators can be used. Defining these operations is, however, not so simple. Clarke et al. [2003] introduce several cases where the hill-climbing algorithm has outperformed genetic algorithms, and the blame is usually placed with the encoding and crossover used with the genetic algorithm.

Doval et al. [1999] have also studied the module clustering problem, and have used the module dependency graph (MDG) mentioned in Subsection 2.3.1. The module dependency class is defined as a directed graph that describes the modules (or classes) of a system and their static inter-relationships using nodes and directed edges, respectively. As with the more general software clustering problem presented by Clarke et al. [2003], the goal is to find a "good" partition of the MDG. A good partition features quite independent subsystems which contain modules that are highly inter-dependent [Doval et al., 1999]. This definition of a good partition justifies the use of the MQ metric for the fitness function: independent subsystems have low coupling, and high inter-dependency signifies high cohesion.

Doval et al. [1999] have used a genetic algorithm approach for the optimization of the module clustering problem. A numeral encoding is used, where each node N_i is assigned a unique number that specifies the locus with the information about that node's cluster, e.g. N_1 is in the first locus of the chromosome and N_2 is in the second locus. The actual alleles are the numbers of clusters where the nodes representing the components are assigned to. Formally, a chromosome is represented as a string S, which is defined as $S = s_1\ s_2\ s_3\ s_4\ \ldots\ s_N$, where N is the number of modules, i.e. the number of nodes in the MDG, and s_i, $(1 \leq i \leq N)$ identifies the cluster that contains the ith node of the graph. Doval et al. [1999] use a crossover rate of 80% for populations with 100 individuals or less, and 100% for populations of a thousand individuals or more. The rate varies linearly between those population values. The crossover function itself is the traditional one, i.e., it combines subsections of two parents from the left and right side of the crossover point. The mutation changes the value of one gene to a new, randomly generated value, thus moving the node represented by the locus in question to a new cluster represented by the new value. Doval et al. [1999] have used their algorithm on real systems, and stress the point of obtaining correct parameters (size of population,

number of generations and crossover and mutation rates) in order to achieve solutions with a higher quality and to also improve the algorithm execution performance. Tests on a real system with a documented MDG showed that Doval et al.'s [1999] algorithm produced a graph quite similar to the real one. The areas where the algorithm had the most problems with were interface and library modules.

Harman et al. [2002] make their contribution to the modularization problem by introducing a new representation for the modularization as well as a new crossover operator that attempts to preserve building blocks. They approach the clustering problem from a re-engineering point of view: after maintaining a system its modularization might not be as good as it was when it was taken to use. Thus, Harman et al. [2002] define their problem as searching the space of possible modularizations around the current granularity, i.e., the number of modules a modularization uses, to see if there exists a better allocation for the components.

Firstly, the new representation presented by Harman et al. [2002] ensures that each modularization has a unique representation. A look-up table is used in order to allocate components to numbered modules. It is also defined that component number one is always in module number one, as well as all components belonging to the same module. Then, component n with the smallest number that is in a different module as component number one is placed in module number two, and the process is repeated with components in the same module as component n, and again for all modules similarly [Harman et al., 2002].

Secondly, Harman et al. [2002] present a new crossover, which does not choose a random crossover point within the two parents, as crossover operators usually do, but a random parent, and a random module from that parent, which is then copied on to the child chromosome. The components in this module are then removed from the two parents in order to prevent clones of components, and the rest of the modules are copied to the child chromosome in a similar fashion from one or the other parent. This kind of crossover operator ensures that at least one of the modules from the parents is completely preserved in the child, and supports the building block hypothesis.

Di Penta et al. [2005] introduce the Software Renovation Framework (SRF) that attempts to remove unused objects and code clones and to refactor existing libraries into smaller, more cohesive clusters. Genetic algorithms have been used especially to help with refactoring. The SRF works in six steps [Di Penta et al., 2005]. Firstly, the software system's applications, libraries and dependencies among them are identified. Secondly, unused functions and objects are identified and removed. Thirdly, duplicated and cloned objects are identified and possibly factored out. Fourthly, circularly linked libraries are identified and either removed or reduced. Fifthly, large libraries are refactored into smaller ones. Finally, objects, that are used by many applications but are not yet

grouped, are grouped into new libraries. As the interest mainly lies with the use of genetic algorithm, I will concentrate now on the fifth step and the refactoring.

The library refactoring itself is done in three steps: determining the ideal number of clusters and an initial solution, determining the new candidate libraries with the use of a genetic algorithm, and after asking for feedback (as can be seen, this is a semi-automated form of using search algorithms, as human expertise is used in order to see how many iterations are needed), the second step may be repeated. The encoding used by Di Penta et al. [2005] is a bit matrix: each library is represented by a separate matrix, and the combination of matrices, *GM*, represents the system. The crossover operator is defined so that it changes the content of two matrices around the column defined as the crossover point. Mutations may either move an object by interchanging two bits in a randomly chosen column, or clone an object by taking a random position $gm_{x,y}$ in the matrix and changing its value to 1 if the bit in this position is zero, and the library represented by the matrix depends on the object y [Di Penta et al., 2005]. The probability of the moving mutation should always be bigger than the probability of the cloning mutation, as cloning is not recommended in general. The fitness function used by Di Penta et al. [2005] consists of four different factors: the number of inter-library dependencies in a given generation (*the dependency factor DF*), the total number of objects linked to each application (*the partitioning ratio PR*, which should be minimized), the size of new libraries (*the standard deviation factor SDF*), and the feedback given by developers (the *feedback factor FF*). The FF is calculated as the difference between the matrix *GM* developed by the algorithm and the *feedback matrix FM*, which contains information of the changes suggested by developers in matrix form. The overall fitness function F is defined as $F = DF(g)+w_1PR(g)+w_2SDF(g)+w_3FF(g)$, where w_1, w_2 and w_3 are real-valued positive weight-factors. Di Penta et al. [2005] report that tests with their SRF show very promising results especially with refactoring libraries and thus reducing dependencies.

Seng et al. [2005] represent the system as a graph, where the nodes are either subsystems or classes, and edges represent containment relations (between subsystems or a subsystem and a class) or dependencies (between classes). The encoding used for the genetic algorithm is to have each gene representing a subsystem, and each subsystem is an element of the power set of classes. Seng et al. [2005] use three kinds of mutations: the *split & join mutation*, the *elimination mutation* and the *adoption mutation*. The split & join mutation either divides a subsystem into two smaller subsystems or combines two existing subsystems into one. The subsystems are selected based on how strong their relationship is in the original dependency graph. The elimination mutation deletes a subsystem candidate and distributes its classes to other subsystems. The adoption mutation tries to find a new subsystem candidate for an orphan, that is, a subsystem with only one class. The crossover operator works in five steps and produces two children

from two parents. Firstly, a sequence of subsystem candidates, i.e., a sequence of genes, is selected from both parents. Secondly, the chosen sequences are integrated to the other parent. Thirdly, existing genes (subsystems) containing classes that are now present in the new, integrated sequence, are deleted. Fourthly, the classes that do not exist in the new sequence (and were parts of the deleted subsystem), are collected. Fifthly, the collected classes are distributed to other genes so that all classes will still stay present in the solution [Seng et al., 2005]. The fitness function is formed from a combination of metrics for cohesion, coupling, complexity, cycles and *bottlenecks*. Bottlenecks are subsystems that know about and are known by too many subsystems. A tournament selection is used for selecting the new generation [Seng et al., 2005].

Seng et al. [2005] also believe in the building block theorem, and construct their initial population accordingly. They bring solutions with high fitness values into the initial population in order to ensure the presence of good building blocks from the very beginning. As genetic algorithms demand diversity in order to get the best results, half of the initial population is constructed from the highly fit solutions, and half from randomly selected sets of connected components from the initial graph model.

Based on the tests by Seng et al. [2005] with large systems, e.g., the *javax.swing* that contains over 1500 classes, this method of subsystem decomposing was a highly successful one. The method was also fast, as the tournament technique used for selection is much more efficient than the roulette wheel – although the roulette wheel produces solutions with slightly better fitness values.

4.1.2. Systems integration

Systems integration resembles module clustering, only now the modules are known, and the order in which they are incorporated to the system is what needs to be decided. As the integration usually happens in an incremental way, and not all components are at use at the same time, a lot of *stubs,* i.e., components simulating the functionality of a missing component, often need to be created [Clarke et al., 2003]. A stub is needed when a component is integrated to the system and it uses another component that is still waiting for integration, and the more stubs are needed, the more the integration process will cost. Therefore, the usual solution is that components that are heavily used by other component are introduced early to the system, and components that need a lot of other components, are introduced last. Obviously some components are both heavily used and use a lot of other components, and timing the integration of these components is crucial when attempting to achieve the optimal integration sequence, i.e., the order of integrations which costs the least [Clarke et al., 2003].

The order of integration of components can be presented as a permutation of the set of components [Clarke et al., 2003], quite similarly to the TSP discussed in Chapter 3. However, one needs to be careful when defining the crossover operator to a permutation. A traditional crossover where parts of the chromosomes are interchanged

would very probably produce an illegal solution. Thus, Clarke et al. [2003] present the options of using *order crossover* or *cycle crossover*. Order crossover selects a random crossover point, and then copies the left substring of one parent directly to the child chromosome. The items that are not present in that substring are added in the order they appear in the other parent. Cycle crossover on the other hand merges two chromosomes. For mutations, Clarke et al. [2003] use the *swap* and *shuffle* operations. Swap changes two genes of the chromosome, and shuffle produces a new permutation. When the fitness function can be defined as the cost sum that would be associated with the solution represented by a specific permutation, systems integration can clearly be subjected to genetic algorithms. In order to apply hill-climbing and tabu search, a neighborhood must also be defined. This can be straightforwardly done as follows: two solutions *p* and *p'* are neighbors if and only if *p'* can be generated by swapping two adjacent genes in *p* [Clarke et al., 2003].

Le Hanh et al. [2001] present a very similar solution to the integration testing problem. They stress that the testing of components that are being integrated should be optimized. The chromosome representation is the same as defined by Clarke et al. [2003], as is the swap mutation. The crossover operation is very similar to the order crossover described by Clarke et al. [2003], only Le Hanh et al. [2001] have opted to directly copy the right side of the first parent instead of the left side. Le Hanh et al. [2001] use a simpler fitness function than Clarke et al. [2003], as they only calculate the amount of stubs needed for each solution. The selection function of Le Hanh et al. [2001] is quite unusual. The algorithm is run by first calculating the fitness of each individual of the population. Two individuals with the best fitness values are then chosen to produce the next generation by applying the crossover and mutation operators to these two elite solutions until there are enough individuals to form a population. One might wonder whether this kind of selection operator really gives the best results. The selection restricts the population to the neighborhoods of the two elite solutions, and thus greatly increases the chances of the algorithm getting stuck to a local optimum and not finding the global optimum. However, Le Hanh et al. [2001] report very promising results from their tests where the genetic algorithm was applied to real-world systems, such as *javax.swing*. They mention that the genetic algorithm is not very efficient, and perhaps some adjustments should be made to their fitness function (such as adding the cost of a stub – a metric used by Clarke et al. [2003]), but the quality of the solutions was good, and the genetic algorithm approach could be easily modified to take into account the complexity of the components.

4.1.3. Systems refactoring

Systems refactoring is a somewhat more delicate problem than module clustering. With module clustering, it is more a question of efficiency, while the contents of a system still stay the same. However, when refactoring a system, there is the risk of changing the

behavior of a system by, e.g., moving methods from a subclass to an upper class [Seng et al., 2006]. This risk should be duly addressed, and the refactoring operations should always be designed so that no illegal solutions will be generated or a corrective operation is used to check that the systems behavior stays the same.

O'Keeffe and Ó Cinneide [2004] define the refactoring problem as a combinatorial optimization problem: how to optimize the weighting of different software metrics in order to achieve refactorings that truly improve the system's quality. O'Keeffe and Ó Cinneide [2004] introduce four different kinds of refactoring mechanisms: moving a method up or down in the class hierarchy, extracting or collapsing a class, making a class abstract or concrete and changing the superclass link of a class. The metrics that are used are *rejected methods* (RM, should be minimized), *unused methods* (UM, should be minimized), *featureless classes* (FC, should be minimized), *duplicate methods* (DM, should be minimized) and *abstract superclasses* (AC, should be maximized). It is also pointed out that as metrics for object-oriented design often conflict, the priority of metrics should be made clear by a precedence graph and assign weights accordingly. With the metrics introduced by O'Keeffe and Ó Cinneide [2004], AC should have a lower priority than FC, RM and UM should have a higher priority than FC, and DM should have a higher priority than RM. Taking these priorities into account, some guidelines are achieved for assigning the weights, which together with the actual metrics form the fitness function $f(d) = \sum_{m=1}^{n} w_m metric_m(d)$, where d is the design to be evaluated, n is the number of metrics and w_m is the weight assigned to the mth metric. Initial tests show some promising results in using simulated annealing to improve the design of the system subjected to refactoring [O'Keeffe and Ó Cinneide, 2004]. The combinatorial optimization viewpoint should be noted as a general guideline for building any kind of genetic algorithm, as the fitness function often consists of several metrics that contradict each other.

Seng et al. [2006] have a similar approach as O'Keeffe and Ó Cinneide [2004], as they attempt to improve the class structure of a system by moving attributes and methods and creating and collapsing classes. Seng et al. [2006] begin by extracting a model of the system from its source code, the basic model elements being attributes, methods, classes, parameters and local variables. In addition, an *access chain* is presented in order to produce the best possible results. An access chain models the accesses inside a method body: this needs to be known in order to know the full effect of moving a method [Seng et al., 2006]. A genetic algorithm is used to find the optimal sequence of refactoring operations, thus the chromosome encoding is naturally the sequence of transformations, where each refactoring operation is located in one gene. The sequence can be extended by mutation, which adds another refactoring operation to the current sequence [Seng et al., 2006]. The crossover operator picks a subsequence,

from the first gene to gene *k*, from one parent and simply adds the whole sequence represented by the other parent to the selected subsequence. The transformations are then applied for the model. The firstly selected subsequence is always legal, but with the transformations specified after the crossover point it may be the case that the refactoring operations proposed cannot be performed, and such operations are simply discarded [Seng et al., 2006]. After the model has gone through the transformations specified by the genome, its fitness is calculated. Seng et al. [2006] use a combination of the following metrics for the fitness function: WMC, RFC, LCOM, *information-flow-based coupling* (ICP), *tight class cohesion* (TCC), *information-flow-based cohesion* (ICH), and *stability* (ST). Weights are also assigned to the metrics in order to focus on certain aspects of the fitness function. The fitness of a solution is calculated by adjusting the fitness achieved by metrics. The adjustments put the fitness value in perspective to the metric-fitness of the initial solution and the metric-fitness of the solution with the maximum metric values. Such a fitness function shows the relative improvement in fitness values, which is easier to evaluate than mere raw numerical values. Seng et al. [2006] have achieved some very promising results: the class structure was clearly improved, and there was low statistical spread and good convergence within the fitness values. The fitness values also settled to a standard after some 2000 generation runs. The metric values that improved the most in tests were ICH and ICP, both improving over 80% during refactoring.

O'Keeffe and Ó Cinneide [2007] have continued their research with the use of the representation and genetic operators introduced by Seng et al. [2006]. They introduce a wider list of refactorings that can be applied to the system and by introducing more fine-tuned fitness function metrics. The extended refactorings include operations that affect the security of attributes and methods, i.e. changing it from private to protected or vice versa, and changing a class from abstract to concrete or vice versa. O'Keeffe and Ó Cinneide [2007] use the following metrics: *data access metric,* which indicates cohesion within a class, NOM, *number of polymorphic methods*, CBO, *design size in classes*, i.e. the number of classes in the design, and *average number of ancestors.*

O'Keeffe and Ó Cinneide [2007] also compared the genetic algorithm to other search algorithms: simulated annealing, *multiple ascent hill climbing* (MHC) and *steepest ascent hill climbing* (SAHC). They used a standard geometric cooling schedule and a low starting temperature for the simulated annealing, and this technique proved to be the worst of the tested four methods. Reasons for the low success of SAHC were its very slowness and the facts that an effective cooling schedule is difficult to determine and that there was much variance between results. MHC begins similarly to the regular hill climbing algorithm discussed in Subsection 4.1.1. However, when the MHC algorithm reaches a local optimum, it does not stop, but performs a predefined number of random transformations to the solution. MHC then restarts the search from the

resulting solution; the number of restarts is given as a parameter. Both hill-climbing approaches produced high quality results, and MHC outperformed even the genetic algorithm approach by being extremely fast, while the SAHC technique was quite slow.

Harman and Tratt [2007] introduce a more user-centered method of applying refactoring. They offer the user the option to choose from several solutions produced by the search algorithm, and also point out that the user should be able to limit the kind of solutions he wants to see, as he may only have limited resources for the actual implementation of the suggested refactorings. The fitness functions of search-based algorithms are also problematized, as they often present a complex combinatorial problem, and Harman and Tratt [2007] attempt to achieve a solution where the search wouldn't rely so heavily on perfectly formulated fitness functions.

The refactoring methods are the same as presented by Seng et al. [2006], and two metrics are used to calculate the fitness of a solution: the well-known CBO and a new metric, *standard deviation of methods per class* (SDMPC) [Harman and Tratt, 2007]. Two combinations of these metrics, $f_1 = CBO*SDMPC$ and $f_2 = CBO+SDMPC$, are then considered as options for the final fitness function.

Harman and Tratt [2007] present *Pareto optimality* to aid the evaluation and selection of the results given by the fitness function. They define Pareto optimality as follows: "In economics the concept a Pareto optimal value is effectively a tuple of various metrics that can be made better or worse. A value is Pareto optimal if moving from it to any other value makes one of its constituent metrics worse; it is said to be a value which is not dominated by any other value. For any given set of values there will be one or more Pareto optimal values. The subset of values that are all Pareto optimal is termed the Pareto front of the set." Harman and Tratt [2007] point out that the "true" Pareto front for a search-based system is analytically impossible and impractical to search. Therefore, the front of Pareto optimal values that can be created through a series of runs is considered to be an approximation of the "true" Pareto front.

Pareto optimality is used when the user needs to choose the desired solution. It might be difficult to see what solutions have a truly good combination of the two metrics presented: by showing the solutions belonging to the Pareto front, the user can be sure that these are indeed "good" solutions.

4.1.4. Architecture development

Program transformations for architecture development apply bigger modifications to the system than simple refactoring operations. An example of program transformation is implementing software design patterns to an architecture representation. In general, program transformation is about changing the syntax of the program while keeping the same semantics [Clarke et al., 2003]. This can be achieved by applying a series of transformation steps. Thus, the solution that is searched for is the optimal sequence of transformations. The fitness function, on the other hand, is a combination of code level

software metrics, as introduced in Chapter 2, to measure the quality of the resulting architecture. Mutation operators feature e.g., replacing a transformation in the sequence, shifting transformations or rotating the sequence by swapping the places of two transformation steps. Program transformations can be used for maintenance and re-engineering purposes as well as developing an initial architecture [Clarke et al., 2003].

Amoui et al. [2006] have attempted to implement software design patterns with the help of genetic algorithms. Their goal is to use genetic algorithms to find the optimal sequence of high level design pattern transformations to increase the reusability of a software system. Amoui et al. [2006] introduce the concept of *supergenes* when defining the encoding for the chromosomes. Each chromosome representing a series of design transformation consists of a set of supergenes, each of which represents a single transformation. A supergene contains information of the pattern number implemented, the package number and the classes that the pattern is applied to. Because each supergene has different parameters, mutations and crossovers may result in invalid supergenes: these are found and discarded [Amoui et al., 2006]. The crossover operator has two different versions which can be used separately or together: one can either administrate a crossover at supergene level, swapping the places of the supergenes before and after the crossover point, or select two supergenes and apply a crossover at gene level to these supergenes. The mutation operator mutates a random number of genes inside a randomly chosen supergene. The fitness function used by Amoui et al. [2006] measures the *distance from the main sequence D*, and is defined as

$$D = \frac{|A + I - 1|}{\sqrt{2}},$$

where

$$A = \frac{AbstractClasses}{TotalClasses}$$

and

$$I = \frac{C_e}{C_e + C_a},$$

where C_e is defined as number of classes whose elements are used by other classes, and C_a is the number of classes using elements of the other classes [Seng et al., 2006]. Amoui et al.'s [2006] tests show that genetic algorithm finds better solutions in less time than a random search of design transformations. Similar results have also been achieved by Grunske [2006].

The performance of a software system comes down to how efficient the underlying architecture is. In addition to optimizing the efficiency of an architecture in terms of structure, there are still a set of parameters that can be optimized for any given architecture. These parameters are related to optimization methods such as loop tiling, loop distribution, loop unrolling and array padding optimization. Che et al. [2003]

present how these parameters can be optimized with a genetic algorithm by transforming the parameter selection into a combinatorial minimization problem. They give a vector containing the parameters to the application, and then execute the program in order to test the runtime achieved with the given parameters. The vectors containing the parameters are generated by a genetic algorithm, and their "goodness" is evaluated by the execution time, so that the less time it takes to run the program the better. The result should be a set of near optimal parameters for different architectures. In order to do the tests in reasonable time, Che et al. [2003] have done transformations to the initial code of the application the runtime of which is being tested. The encoding used for the genetic algorithm is a string of integers, the fitness function uses the knowledge on how high the execution time of the individual is on the list of execution times of the population, and selection is performed as a combination of elitist and roulette wheel selection. Preliminary results show that extreme improvements can be achieved in execution time using this approach for parameter optimization.

A different kind of approach is needed in developing service-oriented architectures (SOA). In this case, software offers its functionality as a service, and in order to realize more complex tasks, individual services need to be formed into compositions. When designing SOA one has to deal with two combinatorial problems: firstly, the combination of services must complete the required task and secondly, the quality-of-service (QoS) factors must be taken into account [Jaeger and Mühl, 2007]. Jaeger and Mühl [2007] have studied the optimization of the resulting QoS of the composition using a genetic algorithm. The algorithm identifies an assignment of service candidates to the tasks in the composition considering the selection criteria, for which the QoS is considered. Each individual handled by the algorithm represents an assignment of a candidate for each task, and is represented as a tuple. The fitness value for each individual is achieved by calculating the QoS resulting from the encoded task-assignments. Qualities such as availability, reputation, cost and time are used to calculate a fitness, as well as a penalty factor if the solution fails to meet the given constraints [Jaeger and Mühl, 2007]. A similar fitness function has been used in the work of Canfora et al. [2007], who map concrete services to abstract services, and who have refined the calculation of the total QoS by also examining workflow constructs, such as switches and loops, thus achieving a more accurate value for the total QoS. The distance from constraint satisfaction (D) is also calculated to determine whether the quality of a solution is acceptable. If $D = 0$, a solution has been found, but the iteration may be continued to find a solution with an even better fitness. The genetic algorithm outperformed Integer Programming in the tests performed by Canfora et al. [2007] when the number of concrete services is big, thus showing that meta-heuristic search algorithms can also be used in the case of service-oriented architectures.

4.2. Search algorithms in software analysis and testing

In addition to design related software engineering problems, there are several other fields of software engineering where search algorithms have successfully been implemented, e.g., testing, requirements engineering and project management. I will now present some examples as to demonstrate how widely search algorithms can indeed be used in the area of software engineering.

Search algorithms can be applied to the area of testing for they are convenient in producing optimal test cases. These test cases can be divided into categories depending on what kind of testing they are used for: structural testing, specification based testing or testing to determine the worst case execution time [Clarke et al., 2003].

Structural test techniques determine the adequacy of a test set by considering the structure of the code. Normally such techniques measure the *coverage* of source code, i.e., the proportion of different constructs that are executed during testing, and full coverage is usually expected. Coverage can be divided into three different categories: *statement coverage*, *branch coverage* and *path coverage* [Clarke et al., 2003]. Fitness functions may be defined according to what is measured: how many statements the test case covers, how close to the correct branch does the test case get to, or how many paths it covers and how close does it get to the paths it is supposed to cover. Specification-based testing can be done with the use of pre- and post-conditions P and Q, respectively, and forming a predicate $C(P, Q) = Q \vee \neg P$. A fault is detected if the predicate C is false, and it can be examined with, for example, simulated annealing [Clarke et al., 2003].

Genetic algorithms can quite straightforwardly be used in order to find minimal and maximal execution times as the fitness function is easy to define to be dependent on the execution time of the test case represented by the chromosome [Clarke et al., 2003].

In the cost estimation problem, the size of the application, usually measured in lines of code or in function points, is examined in relation to the effort, which is usually measured in person-months [Clarke et al., 2003]. Search algorithms, and especially genetic algorithms in this case, are used in order to find predictive functions for the relation. The operators of a solution function include +, -, *, /, *power, sqrt, square, log* and *exp*, which will allow approximation of almost any function likely to solve the problem. The initial population is formed of a set of well-formed equations, to which the normal operators of a genetic algorithm are applied [Clarke et al., 2003]. The fitness function used to evaluate the resulting equation is the *mean squared error*,

$$mse = \frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad .$$

The next generation is selected with the fitness-proportionate selection method [Clarke et al., 2003]. The main benefit of using a genetic algorithm in cost estimation is the achieved confidence in results; the algorithm explores solutions solely based on their

fitness values and does not constrain the form of the solution. Thus, even complex evaluation functions have the possibility of being found and the final set of equations provided by the genetic algorithm truly have the best predictive values [Clarke et al., 2003].

Clarke et al. [2003] present that search algorithms can also be used for requirements phasing. The development of a system consists of iterative cycles of selecting a set of requirements and implementing them, after which the system is presented to the customer. Problems arise when there are several customers with different interests: not all customers agree with what requirements should be implemented in the following iteration. To find out the most valued requirements, they need to be weighted or prioritized in some way by all customers. When the requirements have been scored in some way, the problem becomes about finding the optimal set of requirements to implement. However, this problem is an instance of the 0-1 knapsack problem, which is known to be NP-hard, and thus makes it appropriate for search algorithms [Clarke et al., 2003]. A solution to the problem can be represented as a bit vector, bits representing the presence or absence of a requirement, with the basic mutations and crossover operators as discussed in Chapter 3. Neighbor proximity can be represented by the Hamming distance. The fitness function will naturally be the sum of priorities, weights or votes assigned to the requirements represented by a solution. This kind of encoding enables the use of several different search algorithms [Clarke et al., 2003]. Unfortunately, requirements are rarely simple and independent of one another: usually requirements depend on other requirements, and implementing a requirement before the ones it depends on have been implemented will greatly increase its cost. Also, customers may not prioritize requirements using the same criteria, as others may value cost and others development time. Thus, each requirement needs to be represented with a vector that contains all the relevant information: cost, development time, dependencies, etc. This complicates the problem as the fitness function needs to be refined, and it may now be possible to generate illegal individuals, which need to be dealt with [Clarke et al., 2003].

A similar problem lies in the area of project management. When embarking on a project, there are several conflicting desires: costs and duration should be minimized while quality should be maximized, and human resources and the budget should be managed optimally. Alba and Chicano [2007] have approached the project scheduling problem with a genetic algorithm and they define the *project scheduling problem* as follows. Costs associated with the employees and the project should be minimized as well as the duration. The employee is regarded as a resource with several possible skills and a salary, which is the cost of the employee. The employee also has a maximum dedication to the project, which tells how much time the employee can use for the project. If the employee is presented with tasks requiring more time than his maximum

dedication, the employee is forced to work overtime, which results in a high risk of errors that in turn lower the quality of the project as well as an increase in duration.

Alba and Chicano [2007] model the possible solutions as a dedication matrix, encoded into a binary string, which is the representation used for the genetic algorithm. The fitness function is calculated from the weighted cost and duration of the suggested solution, and a substantial penalty is added if the solution is not feasible. The performance of the genetic algorithm is tested by varying the number of tasks and employees, the special skills of the employees and the number of skills an employee has. Results show that increasing the number of employees decreases the quality of the solution, as it becomes more difficult to effectively assign tasks to employees. The same result can be seen from the experiment with the number of tasks: the more tasks, the more complex the problem. Reversely, the more skills an employee has, the easier the problem becomes to solve.

Dick and Jha [1998] have applied a genetic algorithm to address the problem of co-synthesizing hardware-software embedded systems. A co-synthesis system determines the hardware and software processing elements (PE) that are needed and the links that are used for a given embedded system. A co-synthesis system must carry out four tasks: allocation, assignment, scheduling, and performance evaluation. The allocation/assignment and scheduling are known to be NP-complete for distributed systems, so the co-synthesis problem is an excellent candidate for search algorithms [Dick and Jha, 1998]. The implementation by Dick and Jha [1998] optimizes price and power consumption and heuristics are applied to allow multi-rate systems to be scheduled in reasonable time. The system is represented as a combination of the following data: cost, task graph, processing elements, communication links, constraints and a PE allocation string. Solutions are grouped into clusters so that systems with the same allocation string belong to the same cluster. Mutations and crossovers can be defined both at the cluster and the solution level. Dick and Jha's [1998] solution provides the user with the Pareto-optimal set of architectures instead of the single "best" solution, and has shown very promising results in solving co-synthesizing problems.

5. Genetic construction of software architectures

A software system is constructed to serve a specific purpose. In order to achieve the desired outcome, the software needs to complete several tasks leading to the final solution. The tasks can be grouped into *responsibilities*: a responsibility describes a logical function without giving specific details about the actual implementation. For example, a web application may have a responsibility "update user registry". This responsibility holds tasks such as processing the data to be updated, checking the validity of the user registry, and possible notifying of exceptions. The goal in this thesis is to apply genetic algorithms in order to build an architecture for a system when its responsibilities are given as a dependency graph. The basic architecture considers the class division of the responsibilities, and interfaces, design patterns and a message dispatcher are brought into the architecture as fine-tuning mechanisms. A set of "architecture laws" are also defined as to make sure that each produced solution could actually be accepted by a human expert, and that no anomalies are present in the system.

5.1. Architecture representation

When using a genetic algorithm, the first thing needed is the encoding of individuals. The encoding chosen for the implementation presented here follows the supergene idea given by Amoui et al. [2006]. A chromosome consists of supergenes, each of which represents one responsibility in the system. A supergene G_i contains two kinds of information. Firstly, there is the information given as input for the responsibility r_i: the responsibilities depending on it $\{r_{1i'}, r_{2i'}, \ldots, r_{mi'}\}$, its name n_i, execution time t_i, parameter size p_i, frequency of use $f_{i,}$, type d_i (functional or data), call cost c_i and variability v_i. Secondly, there is the information regarding the positioning of the responsibility r_i in the architecture, and for this, class and interface libraries need to be created in the initialization. For a system of *n* responsibilities, a class library is defined as CL = $\{(C_1, 1), (C_2, 2), \ldots, (C_n, n)\}$, so C_i can be identified by the integer value *i* of the tuple (C_i, i). The tuple notation is chosen so that the value *k* for class C_j, which represents the class for gene G_j can be mapped to the respective class (C_k, k) in the class library. An interface library is similarly defined as IL = $\{(I_1,1), (I_2,2), \ldots, (I_n,n)\}$, where I_i is identified by the integer value *i* of the tuple (I_i, i). As only one message dispatcher is allowed in the system, there is no need for a dispatcher library. These identifiers are used in G_i, as it contains information of the class(es) C_i that the respective responsibility r_i belongs to, the interface I_i it implements and the group of responsibilities, RD_i, it is communicating with through its dispatcher D_i, and the design pattern P_i it is a part of. The encoding is presented in Figure 1, which represents a chromosome with *n* responsibilities.

G_1													G_2	G_3	…	…	G_n
$r_{11'}$, $r_{21'}$, …, $r_{m1'}$	t_1	p_1	f_1	n_1	d_1	c_1	v_1	Class C_1	Interface I_1	Dispatcher D_1	$RD_1 \subset \{r_{11'}, r_{21'}, \ldots, r_{m1'}\}$	Pattern P_1					

Figure 1. Chromosome encoding.

This encoding ensures that the dependency graph given as input is never jeopardized, as there is no mutation that would alter the set of depending responsibilities. It is also a simple way to store all the necessary information. As the encoding is responsibility-centered, there is no need for separate encodings for, e.g., classes and interfaces. This also ensures that each responsibility is present in the system, as the class property must always have a value belonging to the given class library. The crossover operation is also safe regarding the basic structure and can be done as a traditional one-point crossover: there is no risk of any responsibility being removed from the system or the dependencies being broken as a result of crossover. A corrective operation is, however, needed to deal with overlapping design patterns.

The weakness of this kind of encoding becomes apparent when the solution needs to be visualized as a UML class diagram, and when class based quality metrics need to be calculated. As the information is now needed from the perspective of the classes and interfaces, extra effort is needed to extract it from the individual supergenes. However, the class diagram only needs to be drawn once to visualize the final solution, and operations calculating the different metrics also need information regarding each responsibility, so the cost of a responsibility-centered model is not that much greater in the end. As also the mutation operations truly benefit from the chosen encoding since the architecture is fairly easily kept legal, the benefits clearly overcome the shortcomings of the presented modeling method.

5.2. Mutations

Mutations transform the architecture in two ways: on system level, where the mutation affects the entire chromosome and on responsibility level, where the mutation affects one or more (in the case of design patterns) supergenes. Crossover is also considered as a mutation in this implementation, but will be discussed separately as it is still implemented as a traditional one-point crossover operation with a corrective function. The system level mutations are introducing or removing a dispatcher to or from the system. They are implemented by either adding or removing a “dummy” gene, and thus no actual mutation index is needed. When a dispatcher is added, the “dummy” gene only has information in the dispatcher locus – thus bringing the dispatcher to the system. When removing a

dispatcher from the system, it is first checked that no responsibility is using the dispatcher, after which the "dummy" gene carrying the dispatcher is simply discarded.

In the responsibility level, supergene G_i, representing responsibility r_i, can be subjected to the following mutations:

- split the class C_i in G_i into classes C_i and C_k
- merge two classes C_i and C_j where C_j is in G_j
- introduce interface I_k, $(I_k, k) \in IL$, to G_i
- remove interface I_i from G_i
- introduce a dispatcher connection to G_i
- remove a dispatcher connection from G_i
- introduce a Strategy pattern to G_i and G_j
- remove a Strategy pattern from G_i and G_j
- introduce a Façade pattern to a set of genes $G = \{G_i, G_j, \ldots, G_k\}$
- remove a Façade pattern from a set of genes $G = \{G_i, G_j, \ldots, G_k\}$
- use variability
- remove variability.

The mutation index (i.e., index i) for these mutations is selected randomly, with a slight alteration in the cases of introducing a strategy pattern and a dispatcher connection. As these particular mutations are especially favorable to responsibilities with a relatively high variability value, the probability of a responsibility being subjected to these mutations grows linearly according to the responsibility r_i's variability value v_i. After each mutation, the system is checked against the architecture laws, and if anomalies are found, they are corrected accordingly.

When splitting a class, the responsibilities located in C_i are divided into two classes, C_i and C_k. The split is done by checking good cutting points, i.e., if C_i contains responsibilities that depend on each other, they are kept together in the "old" class C_i while the other responsibilities are moved to C_k. Merging two classes is the counter-mutation for splitting classes: responsibilities from two different classes, C_k and C_i are placed in one class, C_i.

When introducing interface I_k to G_i, the interface I_k is first chosen randomly from the library, after which the interface value of G_i is set to *k,* thus implementing I_k through r_i. Removing an interface association is the counter-mutation, i.e., if r_i is implementing the interface I_i, the interface value of G_i thus being *i*, the implementation is removed by setting the interface value to 0.

Introducing a dispatcher communication to r_i will cause a depending responsibility r_j to communicate with r_i through the dispatcher D_i instead of being directly associated with the class C_i. Removing a dispatcher communication will cause a depending responsibility r_j to communicate with the responsibility r_i either directly or through an interface I_i, if the responsibility in question implements one.

Introducing a Strategy pattern to responsibilities r_i and r_j, when r_j depends on r_i and r_i and r_j are allocated to the same class C_j, will cause r_i to move to its own class C_i, which implements interface I_i, and to r_j to call r_i through the interface. A pattern instance is created where information regarding r_i, r_j, C_i, C_j and I_i is stored, and the pattern loci are updated in genes G_i and G_j. Removing a strategy pattern will again move the two responsibilities r_j and r_i to the same class and remove the interface I_i from the system. The pattern loci of the responsibilities are also set to null.

Introducing a Façade pattern will cause a group of responsibilities to communicate with a group of responsibilities forming several subsystems through a façade, i.e., a technical class and its interface. It is first checked that the responsibility where the façade should be implemented is indeed a part of a subsystem that could be called through a façade, or that it is a responsibility needing several subsystems, and could thus use a façade. When the legality of the mutation is checked, all responsibilities that are used through or using the façade are collected, as well as the classes they belong to and the interfaces they implement. In addition, a random unused class and interface is assigned to be the technical façade class and interface. All this information is collected into a pattern instance, and the pattern loci of the responsibilities involved with the façade are updated with the new pattern instance. Naturally, if a responsibility is already a part of some other pattern, it is not made a part of the Façade, and if there are not enough "pattern-free" responsibilities that satisfy the preconditions for implementing the façade pattern, the mutation cannot take place. Removing a Façade pattern will null the pattern loci of the responsibilities involved with the pattern, and remove the technical façade class and interface from the system.

Using variability in responsibility r_i will create another class $C_{i'}$ and the interface I_i, so that r_i is now in two classes, C_i and $C_{i'}$ that both implement the interface I_i. Removing variability will remove the class $C_{i'}$.

The presented mutations allow different ways of communications between responsibilities: direct associations between classes, communication through interfaces (the different varieties of which were introduced in Chapter 2), and communication through a dispatcher, which implicates that the message dispatcher architecture style would be an appropriate choice for the system in question. The class structure is modified by splitting and merging classes, and keeping sub-systems intact is encouraged by checking for split points. Each mutation also has a counter-mutation, so every move can be reversed in order to ensure the most flexible traverse through the search space.

The chosen mutations also conform to the idea of *unit operations* introduced by Bass et al. [1998]. These operations are used to achieve architecture styles and design patterns, and can be categorized to separation, abstraction, compression and resource sharing. Merging and splitting a class are clearly analogous to compression and separation, abstraction is achieved through abstract classes and interfaces, and resource

sharing can be done through a message dispatcher or an interface. It should be noted that this is a rough analogy of unit operations to these mutations, and at a more detailed level the unit operations are more complex, and, e.g., introducing an interface to a class is also a case of separation, as it separates that particular class by "hiding" it behind the interface. Bass et al. [1998] also discuss the actual resources to be shared and make an example of databases; this also justifies the incorporation of different types for responsibilities, and thus identifying data in the system.

5.3. Crossover

The purpose of a crossover is to combine good properties from two individuals. A crossover between two architectures can be implemented as a traditional single-point crossover. Figure 2 illustrates a crossover between chromosomes CR_1 and CR_2 at crossover index *k*, the result being chromosomes CR_{12} and CR_{21}.

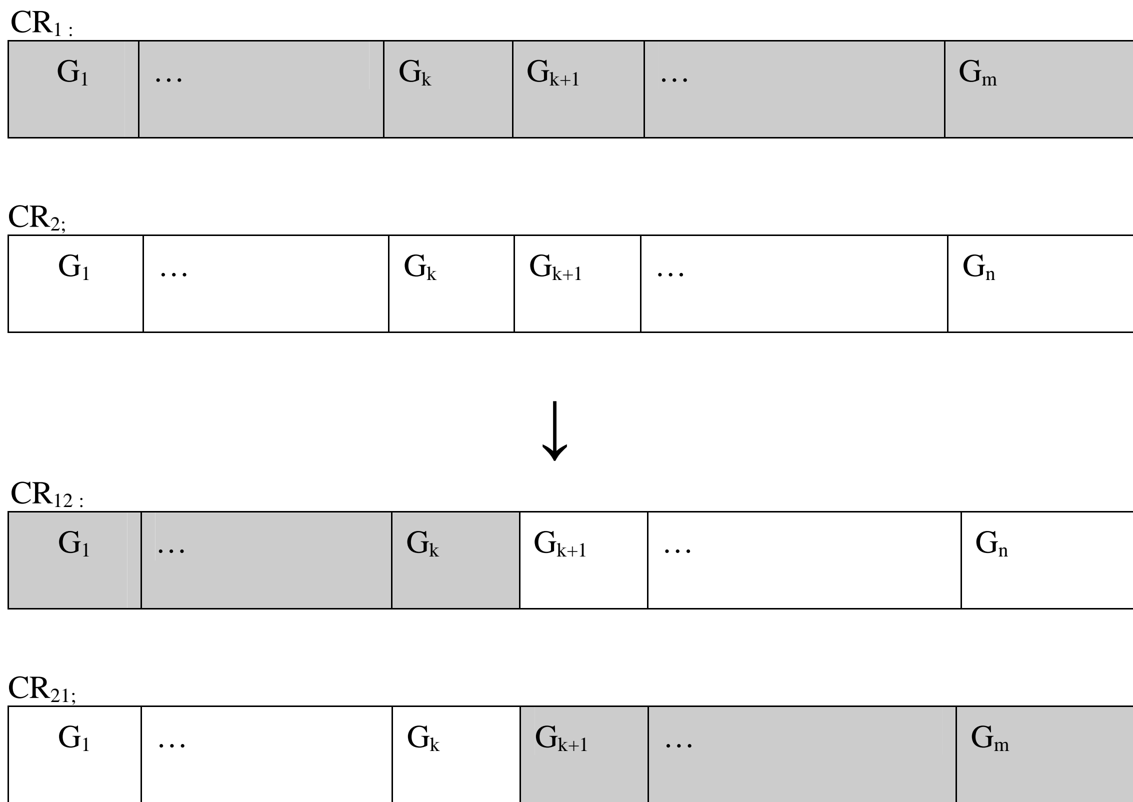


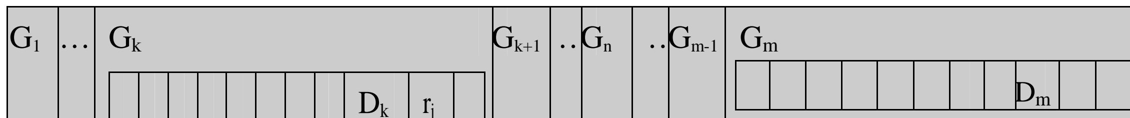
Figure 2. The crossover operation

The selected encoding and the way of performing a crossover operation ensure that the architectures stay legal, as the supergenes stay intact during the crossover operation, i.e., no responsibility can be dropped out of the system or be duplicated into two different classes, and no interface becomes "empty". The optimum outcome of a crossover operation at index *k* would be that CR_1 has found good solutions regarding interfaces and dispatchers and a clear structure for responsibilities from r_1 to r_k, and CR_2 contains good solutions for responsibilities from r_{k+1} to r_n in a system with *n*

responsibilities. Thus, the resulting chromosome CR_{12} would be a combination of these solutions, and contain a good solution of the entire system.

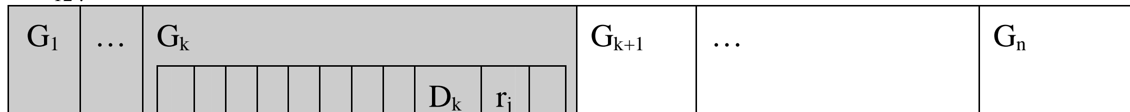
Decisions regarding architecture style and patterns are kept during a crossover operation, i.e., if a responsibility uses the message dispatcher for communication, this way of communication is maintained even after a crossover operation. Thus, a corrective operation is needed to deal with missing dispatchers and overlapping patterns. The former situation is presented in Figure 3, where the supergene G_m and the supergene G_k in chromosome CR_1 are separated during the crossover operation.

CR_1 :



CR_2;

CR_{12} :



CR_{21};

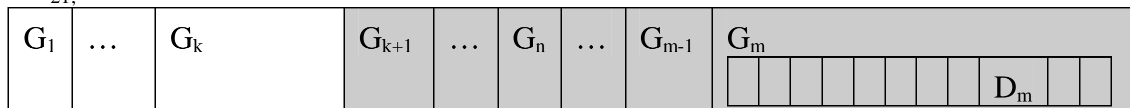


Figure 3. Chromosome CR_{12} contains a gene needing a dispatcher (G_k) but not the gene containing the dispatcher (G_m)

As a message dispatcher needs to be declared in the system so it can be used by the responsibilities, a corrective operation is now needed. The correction is done by adding the supergene G_m to chromosome CR_{12}, the end result $CR_{12'}$ shown in Figure 4.

$CR_{12'}$:

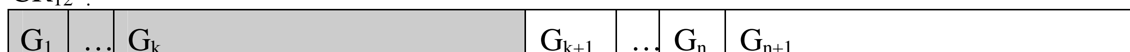
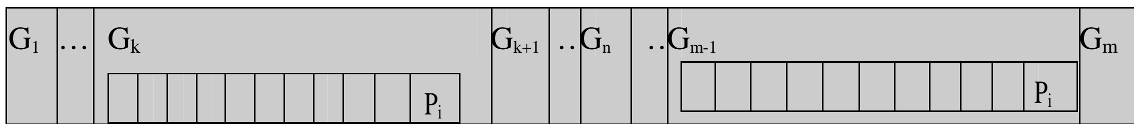


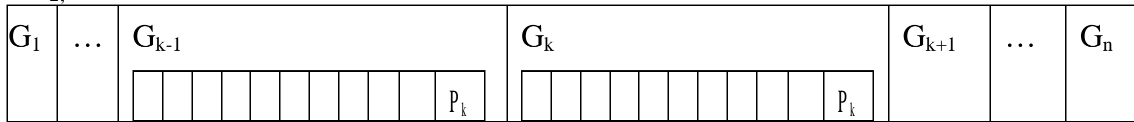
Figure 4. Corrected dispatcher usage.

The case of over-lapping patterns is described in Figure 5. In this example, the genes G_k and G_{m-1} in chromosome CR_1 are involved in the same pattern. After the crossover, however, these genes are separated. Moreover, the chromosome with which CR_1 produces offspring, CR_2, already has a different pattern involving gene G_k. For these cases, an order of priority must be issued so the offspring can be corrected rightly.

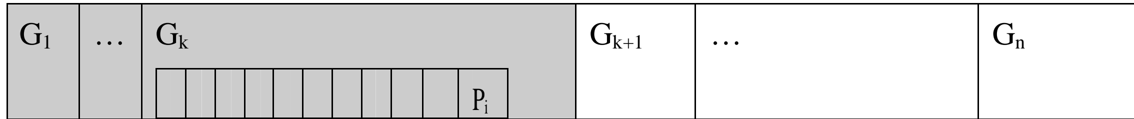
CR_1 :



CR_2;



CR_{12} :



CR_{21};

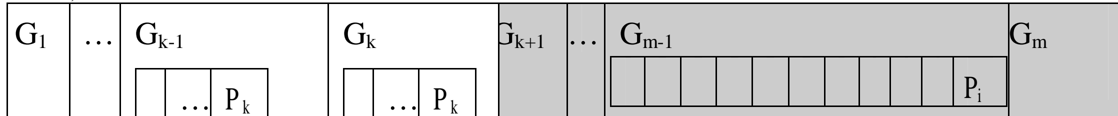


Figure 5. Overlapping patterns.

In order to deal with overlapping patterns, an order of importance must be decided, and the left side of the chromosome is chosen to be the correct one. The corrected offspring is depicted in Figure 6. Because of the chosen order of importance, the corrective operation ends up adding a pattern to $CR_{12'}$ and removing one from $CR_{21'}$. This is achieved as when first checking the left side of CR_{12}, a pattern is found in gene G_k. The chromosome is then altered so that the found pattern is made whole, i.e., all the genes that should be involved with the pattern, as listed in the pattern field, should contain this pattern. This results in adding the relevant information in the previously empty pattern field in gene G_{m-1}, thus achieving the new chromosome $CR_{12'}$. When CR_{21}

is taken under inspection, the left side of the chromosome does not give reason for any modifications. However, as we begin to inspect the right side of the chromosome, a pattern is found in gene G_{m-1}. In order to make this pattern legal, the left side would need an update. However, as the left side should remain untouched, the actual outcome of the correction is removing the pattern in G_{m-1}, as no pattern can only involve one gene.

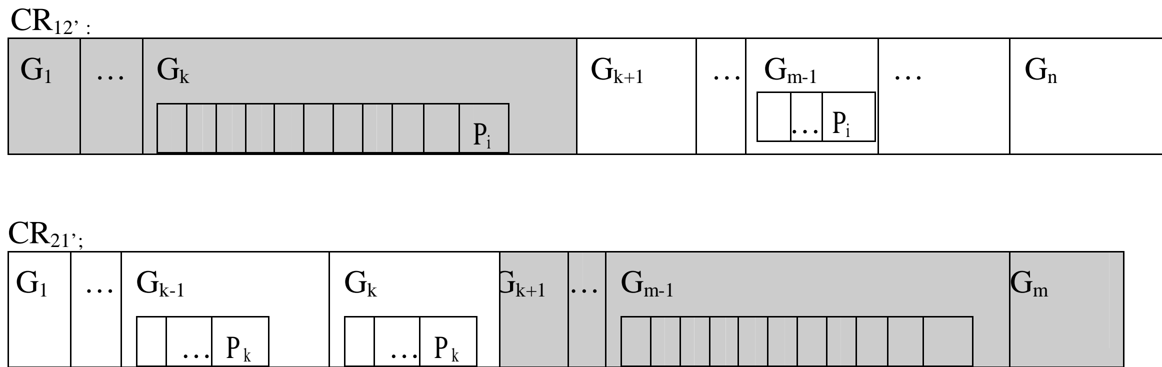


Figure 6. Corrected patterns.

To summarize, the crossover operation combines two subsets of responsibilities with their respective architectural structures by administering a one-point crossover. A checking and correcting operation is needed in the case where there is a message dispatcher present in the systems or there are overlapping patterns, but as this is a fairly simple procedure, the benefits of the defined crossover operation clearly overcome these minor disadvantages.

5.4. Architecture laws

There are a group of "laws" defined to make sure that there are no anomalies in the proposed system. These laws mainly state some rules on how responsibilities can be grouped to classes and how classes call each other through interfaces and the dispatcher.

Firstly, there are the rules regarding structure. By default, a responsibility is only present in one class. A responsibility can be placed in more than one class only if its variability is bigger than 1, and it has been subjected to the variability mutation. All functional responsibilities that use a data responsibility must be in the same class as the data responsibility. This rule is overridden in the case of patterns.

Secondly, there are rules regarding the interfaces. Each responsibility is only allowed to implement one interface. Interface implementations are also restricted in the way that only function-type responsibilities are allowed to implement an interface. Also, it is forbidden that a class would call an interface it implements itself. Furthermore, two classes cannot implement the same interface unless they both contain all the responsibilities implementing the particular interface. This is actually only possible in the case of varied responsibilities. An interface can also only be implemented if at least one

responsibility using the interface is in a different class than the responsibilities implementing the interface: no unused interfaces are allowed.

Finally, there are rules on how classes communicate. If responsibility r_i in class A calls responsibilities from class B, and one of the needed responsibilities implements interface C, then all the responsibilities needed by r_i that are located in class B must implement the interface C. Also, if responsibility r_i in class A calls responsibilities from class B, and one of the needed responsibilities is called through the message dispatcher, then all the responsibilities needed by r_i that are located in class B must be called through the dispatcher. Finally, if a responsibility that is called through an interface implements interface C, the message dispatcher must call the interface instead of communicating directly.

6. Implementation

6.1. Presenting the program

The implementation has been done with Java SE 1.5.0, and the core program implemented handles the given data, executes the genetic algorithm, stores data of fitness values and generates Java-files with javadoc-comments. These Java-files are then given to UMLGraph_4.8 [UMLGraph, 2007], which in turn generates a .dot –file containing information of the resulting class diagram. Finally, GraphViz_2.14 [GraphViz, 2007] is used to generate a GIF-picture from the .dot descriptive file.

6.1.1. Structure

The implementation is aimed to be as simple as possible, and it straightforwardly follows the execution of a genetic algorithm presented in Chapter 3. The modeling presented in Chapter 5 has been implemented with the Cromosome and SuperGene classes, presented in the class diagram of the implementation 'Frankenstein' in Figure 7. The SuperGene class is an inner class of the Cromosome, keeping a tight analogy between the implementation and the presented model. The Cromosome class holds all information of the system as a whole – the class and interface libraries, used classes, interfaces and the dispatcher, and the fitness value of the individual. It also has the crossover operation, the mutation operations that affect the entire chromosome, such as introducing a dispatcher to the system, and all the different fitness functions responding to the used quality metrics. The fitness function is implemented in the Cromosome class instead of the GeneticAlgorithm class in order to minimize calls between classes, as the fitness functions need to constantly access the information contained in both the chromosome as an entirety as well as its individual supergenes. The SuperGene class holds the information stored in a gene, as presented in Chapter 5. It also contains the mutation operations that affect an individual gene, such as introducing an interface, as well as operations for accessing all the information stored in the gene. The Pattern class is used as a base class to store all common information regarding design patterns, such as responsibilities and classes involved with the pattern. Each design pattern is given a special class that is inherited from the Pattern class, as has been done with StrategyPattern and FacadePattern classes here. These special pattern classes contain the information that is only relevant for the specific pattern, such as a technical pattern class in the case of a façade pattern. The GeneticAlgorithm class contains the basic operations of the genetic algorithm – creating a population, handling a population and selecting the next population. Other classes in the implementation are Frankenstein, which is the main class, OutputDataHandler, which takes care of storing the fitness data, InputDataHandler, which transforms the information given as input into a "base"

chromosome, and UMLGenerator, which transforms the information in the achieved best solution into Java-files.

Figure 7. Class diagram of ‘Frankenstein’

The process of ‘Frankenstein’ is described in the sequence diagram in Figure 8.

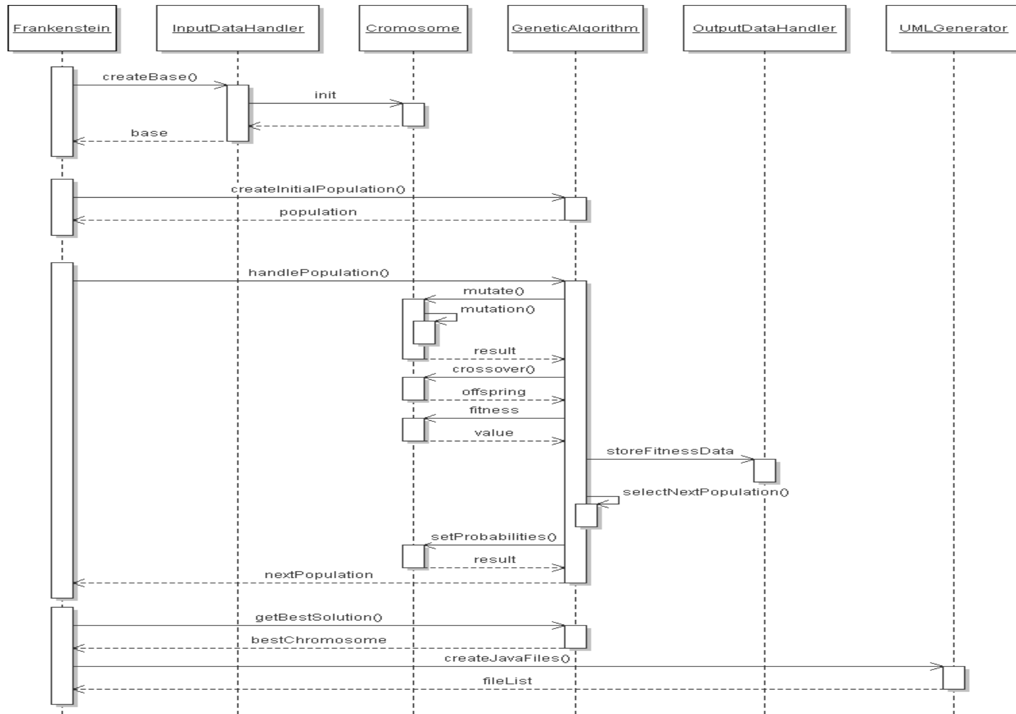


Figure 8. Sequence diagram for 'Frankenstein'

First, a "base" chromosome is created by the InputDataHandler, which returns a Cromosome type model of the given set of responsibilities. This Cromosome instance contains all the information concerning responsibilities given in the input. The chromosome representation is then given to GeneticAlgorithm so that an initial population can be created. The population is created in such a way that two special cases – all responsibilities in the same class and all responsibilities in different classes – are put in the population by default to ensure variability in the population. Other individuals are created randomly. In this stage, only the libraries described in Chapter 5 are set, and a random class is chosen for each responsibility. Interfaces, dispatchers and abstract classes are only incorporated through mutations; they are not present in the initial population.

After the initial population has been created, the actual algorithm can begin to process the chromosomes. The GeneticAlgorithm class communicates with the Cromosome class to mutate, crossover and calculate the fitness of each individual in turn. After the whole population is dealt with, fitness data is stored and the selection for the next population can begin. After an individual is selected, its mutation probabilities are adjusted in relation to its fitness value in the population (from now on, this will be referred to as an individual's fitness order). The selected next population is returned to the main class, which will again call the GeneticAlgorithm to handle it. This cycle continues until the termination condition is met. Finally, the best solution is picked up, and UMLGenerator is called to produce a representation of the solution.

6.1.2. Algorithms

So far I have presented the overall structure and flow of the implementation. I will now give more detailed descriptions of the most important algorithms within the implementation: the overall structure of the genetic algorithm, creating a population, crossover, mutate, selection and setting the probabilities. The mutate operation will select a mutation from the ones presented in Chapter 5. I will give examples of four specific mutations: splitting a class, introducing an interface, removing a dispatcher from the entire system and introducing a strategy pattern. The other mutations are quite similar, and the logic behind their implementation can be seen from the example algorithms.

Algorithm 1 presents the general genetic algorithm. Random mutation indexes and probabilities are set and the chromosome is subjected to mutation. The mutate operation returns the initial chromosome to *crossoverChromosome* if the chromosome should be subjected to crossover. If the chromosome is mutated, it still has a chance to be subjected to crossover: the second mutation is only effective, if the chosen mutation is the crossover operation (as discussed in Chapter 5, crossover is thought of as a mutation as well). The fitness value of the chromosome is calculated after the mutation. After all the chromosomes have been dealt with, it is known which chromosomes are subjected to crossovers, which are done in pairs. The fitness values of the offspring are then calculated, after which the fitness values of the entire population can be sorted. The next population can now be selected.

Algorithm 1 geneticAlgorithm

Input: base chromosome *b*, elitism integer *e*, population size *p*
Output: best chromosome after termination condition

```
chromosomes ← createPopulation(b)
do
    foreach chromosome in chromosomes
        p ← randomDouble
        i ← randomInteger
        crossoverChromosome ← mutate(chromosome, i, p)
        if crossoverChromosome == null then
            p ← randomDouble
            i ← randomInteger
            crossoverChromosome ← mutate(chromosome, i, p)
        end if
        if NOT crossoverChromosome == null then
            toCrossing.add(crossoverChromosome)
        end if
        fitness ← fitness(chromosome)
        fitnesses.add(fitness)
    end for
    while toCrossing.length > 1do
    offspring ←crossover(toCrossing[0], toCrossing[1])
    chromosomes.add(offspring)
```

```
    fitness ← fitness(offspring)
    fitnesses.add(fitness)
    remove processed chromosomes from toCrossing
    end while
    sort(fitnesses)
    fitnessBackUp ← fitnesses
    chromosomes ← selection(chromosomes, fitnesses, fitnessBackUp, p, e)
while NOT terminationCondition;
getBestSolution(chromosomes)
```

Algorithm 2 describes the creation of the initial population, already discussed in Subsection 6.1.1. The initial population is created by copying genes from the given base chromosome, and then giving each responsibility a class to which it is located. The special cases of having only one class, or having each responsibility in its own class, are created before any other individuals.

Algorithm 2 createPopulation

```
Input:   base chromosome b, population size s
Output:  linked list chromosomes containing the chromosomes that form the population
c ← copy(b)
set same class for all genes in c
setLibraries(c)
chromosomes.add(c)
d ← copy(b)
set a different class for all genes in d
setLibraries(d)
chromosomes.add(d)
for i ← 1 to s-2 do
    e ← copy(b)
    set a random class for all genes in e
    setLibraries(e)
    chromosomes.add(e)
end for
```

The crossover-operator is described in Algorithm 3. Both "children" first receive copies of genes from one parent, and at the crossover point locus, the parent from which the genes are copied is changed. However, mere copying is not enough, as discussed in Chapter 5; the use of a dispatchers must be checked. If the first part of the child needs a dispatcher, but it is not available in the parent providing the part after the crossover point, then the dispatcher is introduced to the system through a forced mutation.

Algorithm 3 crossover

```
Input:   chromosomes one and two
Output:  chromosomes onechild and twochild
i ← randomInteger
onechildDispatcher ← 0
twochildDispatcher ← 0
```

```
for j ← 0 to i do
    onechild.geneAt[j] ← one.geneAt[j]
    twochild.geneAt[j] ← two.geneAt[j]
    if one.geneAt[k] contains Dispatcher then
        onechildDispatcher ← 1
    end if
    if two.geneAt[k] contains Dispatcher then
        twochildDispatcher ← 1
    end if
end for
for k ← i +1 to two.length -1 do
    onechild.geneAt[k] ← two.geneAt[k]
end for
for m ← i +1 to one.length -1 do
    twochild.geneAt[m] ← one.geneAt[m]
end for
if onechildDispatcher == 1 AND NOT(two contains Dispatcher)  then
    mutationIntroduceMessageDispatcher(onechild)
end if
if twochildDispatcher == 1 AND NOT(one contains Dispatcher)  then
    mutationIntroduceMessageDispatcher(twochild)
end if
```

The general mutation is presented in Algorithm 4: this operation merely finds out the mutation that responds to the given probability, and passes along the chromosome.

Algorithm 4 mutate

```
Input:   integer i, the mutation index, double p, the mutation probability, chromosome c
Output:  c if subjected to crossover, else null
    mutationChoice ← selectMutation(i, p)
    mutation(c, mutationChoice)
```

As presented in Chapter 3, each mutation has a probability with which a chromosome is subjected to it. In the selection process, the crossover is also regarded as a mutation. As no chromosome can be subjected to more than one mutation during each generation, the sum of probabilities, given as percentages, should be 100%, as one mutation should indeed be chosen. As mutations should have fairly low probabilities in order to keep the evolving of solutions under control, a “null” mutation is used in order to bring the sum of percentages to 100. If the “null” mutation is chosen, the chromosome will remain as it was.

The selection of a mutation is presented in Algorithm 5. The principle is the same “roulette wheel” selection as in selecting chromosomes for the next population: the size of a “slot” in the “wheel” is determined by the probability of the respective mutation. The list of probabilities is gone through, and when the “slot” which includes the given probability value is found, the id-number of the corresponding mutation operation is returned.

Algorithm 5 selectMutation

Input: double value probability *p*
Output: integer id *mutationChoice*

```
for m ← 0 to probabilities.length do
    if p < probabilities [m] AND  m == 0 OR p > probabilities [m-1]
    mutationChoice ← m+1
    end if
end for
```

Splitting a class by a mutation is described in Algorithm 6. The class C_k of gene *g*, holding responsibility r_k, is found out, and a new class is selected randomly. The if-statement is for selecting the split points discussed in Chapter 5; responsibilities depending on one another are kept in the same class, and other responsibilities are moved to the randomly selected new class.

Algorithm 6 mutationSplitClass

Input: gene *g*

```
n ← g.getClass()
r ← randomInteger
foreach  gene sg in c do
    if sg.getClass() == n AND NOT(sg depends on g)  then
            sg.setClass(r)
    end if
end for
```

Introducing an interface to a responsibility is presented in Algorithm 7, and it is quite straightforward as well. The interface to be implemented is selected randomly. Then it is checked that the responsibility that should be implementing the interface is of the type ‘function’, and does not belong to a class that is already implementing the chosen interface.

Algorithm 7 introduceInterface

Input: gene *g*

```
n ← randomInteger
if g.type == ‘function’ then
    if NOT exists gene ge:: ge.class == g.class AND exists gene gn:: gn.Interface == n AND
    ge uses gn then
    g.setInterface(n)
    end if
end if
```

The mutation that removes a dispatcher, as presented in Algorithm 8, differs from the previous mutations in the way that its target is the chromosome, not an individual gene. It is first checked that the chromosome even contains a dispatcher. If a dispatcher is found, the default assumption is that no responsibility is using the dispatcher. The

genes are then iterated through, and if any of them uses the dispatcher for communication, removing of the dispatcher is not possible.

Algorithm 8 removeDispatcher

Input: chromosome *c*
  if *c* contains Dispatcher **then**
    usedDispatcher ← false
    foreach gene *g* in *c* **do**
      if *g* uses Dispatcher **then**
      usedDispatcher ← true
      end if
    end for
    if *usedDispatcher* == false **then**
      dg ← *g*.dispatcherGene
      g.remove(*dg*)
    end if
  end if

Algorithm 9 introduceStrategy

Input: gene *g*
  if NOT *g* used through a pattern **then**
    if exists *gd*:: *gd* uses *g* AND *gd*.class == *g*.class AND NOT *gd* uses Pattern **then**
      n ← random Integer
      g.setClass(*n*)
      p ← Strategy pattern instance
      g.setUsedPattern(*p*)
      gd.setUsingPattern(*p*)
    end if
  end if

The process of introducing a strategy pattern is given in Algorithm 9. It is first checked, whether the gene to be mutated still has “free” pattern fields: if the responsibility in question is already used through some other mutation, Strategy pattern cannot be implemented on top of the previous pattern. Also, if all responsibilities that use the selected responsibility already use some other needed responsibility through a pattern, the Strategy cannot be introduced. However, if suitable responsibilities are found, implementing the pattern is quite simple. The gene used through the Strategy pattern is assigned a new class, and a pattern instance is created, where information of the using and used genes are stored, as well as information of the classes and interfaces linked to the pattern. After the mutations, the fitness values of chromosomes are calculated, after which a new generation can be selected, as described in Algorithm 10. The method used for selection is the “roulette wheel”: each chromosome is given a slice of the “wheel” with respect to its fitness order. Before any other chromosome is selected, the best ones are automatically selected through elitism. After this, the slots are calculated for the “roulette wheel”, and a random probability is generated. Much like in

the mutation selection, the chromosome “owning” the slot responding to that probability is selected to the next generation. The selection process is repeated until the number of chromosomes selected for the next generation is equivalent to the given population size.

Algorithm 10 selection

```
Input:    list of chromosomes, cl, list of fitnesses, fl, sorted lists of fitnesses, sfl, population size
integer s, number of elites, integer eliteAmount
Output:  a list of chromosomes ncl
   for i ← 0 to eliteAmount do
      fitness ← sfl[i]
      nextFitnesses.add(fitness)
      j ← fl.indexOf(fitness)
      c ← cl[j]
      nextGeneration.add(c)
      cl.remove(c)
      fl.remove(fitness)
   end for
   for k ← 0 to s-eliteAmount  do
      wheelAreas  ← setWheelAreas(fitnesses)
      i ← randomDouble [0…1]
      chromosomeFound  ← false
      for m ← 0 to wheelAreas.length do
         if i < wheelAreas[m] AND  m == 0 OR i > wheelAreas [m-1]
         chromosomeFound  ← true
         goodness ← sfl.indexOf(fl[m])
         setProbabilities(chromosome, goodness, s)
         nextFitnesses.add(fl[m])
         nextGeneration.add(cl[m])
         sfl.remove(fl[m])
         fl.remove(m)
         cl.remove(m)
         end if
      end for
```

After a chromosome has been selected to the next generation, its mutation probabilities are set according to its fitness order, as described in Algorithm 11. The setting of the probabilities is done at this point to avoid calculating the fitness value twice during the process of handling a population. The probabilities are set so that if the chromosome’s fitness order is in the “better half” of fitnesses, the probability of crossover is increased in relation to the fitness order. If the fitness order of the chromosome belongs to the “lower half”, the probability of the crossover is halved. Since the sum of mutation probabilities should be 100%, the probabilities of the mutations must be decreased in relation to the increase in the crossover probability. The crossover probability is the last one in the list of probabilities in order to ease the execution of this algorithm. We also use dynamic mutations to take better advantage of the different levels of the implemented design patterns. In the early stages of

development, the dispatcher style is favored, as it is a high-level design decision. In the middle stages (after 1/3 of the generations), the façade pattern is more favorable, as it is a middle-level design pattern, and we assume that point the major decisions regarding the core architecture (including the decisions of using certain architectural styles) should have been made. At this point, the probability of introducing a message dispatcher to the system is decreased, and the probability of using a façade pattern is increased respectively. After the algorithm has proceeded through another 1/3 of the generations, another shift is made in favoring certain patterns. At this point the architecture should be at the last stages of development and the design decisions should be focused on low-level details. Thus, the probability of implementing a façade pattern is decreased and the probability of implementing a strategy pattern is increased respectively.

Algorithm 11 setProbabilities

Input: chromosome *c*, order of fitness *fo*, list of probabilities *pl*, population size integer *s*, generation number *g*, terminating generation *t,* Boolean first alteration *b*1, Boolean second alteration *b*2
Output: altered list of probabilities *pl*
if *g*>*t*/3 AND *g*<= 2*t*/3 AND NOT *b*1 **then**
dp ← *c*.newDispatcherProbability
fp ← *c*.introduceFacadeProbability
diff ← *dp*/3
ndp ← *dp* – *diff*
nfp ← *fp* + *diff*
c.setDispatcherProbability(*ndp*)
c.setFacadeProbability(*nfp*)
end if
if *g*>2*t*/3 AND NOT *b*2 **then**
fp ← *c*.introduceFacadeProbability
sp ← *c*.introduceStrategyProbability
diff ← *fp*/3
nfp ← *fp* – *diff*
nsp ← *sp* + *diff*
c.setFacadeProbability(*nfp*)
c.setStrategyProbability(*nsp*)
end if
if *fo* < *s/2* **then**
    multiplier ← 1/*fo*
    crossoverprobability ← *pl*.last – *pl* [*pl*.length-2]
    probabilityChange ← *crossoverprobability*multiplier*
    crossoverprobability ← *crossoverprobability* + *probabilityChange*
    mutations ← *pl*.length -1
    pl[*pl*.length-2] ← *pl*.last – *crossoverprobability*
    for *i* ← 0 to *mutations* **do**
        altering ← *probabilityChange/mutations*
        pl[*i*] ← *pl*[*i*] – *altering*
    end for
end if
if *fo* >= *s/*2 **then**

```
    crossoverprobability ← pl.last – pl [pl.length-2]
    probabilityChange ← crossoverprobability/2
    crossoverprobability ← crossoverprobability/2
    mutations ← pl.length -1
    pl[pl.length-2] ← pl.last – crossoverprobability
    for i ← 0 to mutations do
        altering ← probabilityChange/mutations
        pl[i] ← pl[i] + altering
    end for
end if
```

6.1.3. Parameters

The input for the implementation is the dependency graph of the responsibilities in the system as well as performance information of the responsibilities. The graph is given as an adjacency list, which makes it possible to present the information in a simple text-file, where each responsibility is represented by one line. The output is a UML class diagram, which is constructed of the best solution remaining in the final generation. Fitness data is also stored throughout the generations in a separate file so that the development of fitness values can be monitored.

For the genetic algorithm, there are two types of adjustable parameters: the common parameters for any genetic algorithm implementation and the parameters where the nature of the problem needs to be considered. The common parameters include the size of the population, the termination condition (often either tied to the fitness value or to the number of generations), the level of elitism and how the order of fitness affects the crossover rate. When choosing the level of elitism one should keep in mind that the level should be high enough to ensure that the solutions truly evolve by having the best material to develop from, but at the same time there should be enough room for selection through probability, in order to ensure a free enough traverse through the search space. As for the effect of the fitness order to the crossover probability, the current solution ensures a perfect relation between the order of fitness and the increase in the crossover probability. Other implementations are also possible, as long as the following requirements are met. Firstly, the probabilities should be kept under control, i.e., there must still be a possibility for at least some mutations after increasing the probability of the crossover. Secondly, increasing the probability of the crossover should have some logical relation to the fitness order. Thirdly, the probability of crossover should not be raised for the worst solutions, but rather deducted, as it can be assumed that they have poor material that should not be passed on to the next generation.

The problem specific parameters are the weights assigned to different fitness evaluators (quality metrics) and the probabilities given to different mutations. The fitness weights can be given freely, but in order to ensure that the relation between metrics is as intended, the ranges of the different quality metrics should be taken into account when

assigning the weights. When assigning weights, one should remember to think of what characteristics are most valued, as it is extremely difficult to optimize all quality aspects at the same time. In this implementation, I have used three different evaluation criteria, two of which have both a negative and a positive evaluator, thus giving a total of five evaluators. The ranges of these five evaluators have been taken into account so that the user does not have to consider the different ranges when assigning weights, as the "raw" values have been given multipliers to achieve same ranges for all evaluators. The restrictions to mutation probabilities have been discussed in Subsection 6.1.2, and adding the combinatorial problem of optimizing these probabilities alongside with the fitness weights results in a very complex task of parameter optimization.

6.2. Evaluation metrics

There are two different types of characteristics that can and should be evaluated in the produced architecture: the basic structure, i.e., how the responsibilities have been divided into classes and how many associations there are between classes, and the fine-tuning mechanisms, i.e., the use of interfaces and the message dispatcher. Since no particular pattern should be appreciated just for being a pattern, design patterns are valued based on the modifiability they bring to the system, which is largely credited to an increased usage of interfaces and a decrease in amount of connections between classes. As presented in Chapters 2 and 4, there are several structure evaluation metrics which have been successfully combined and used as a fitness function for genetic algorithms processing architectures. As for the evaluation of interfaces and using the dispatcher, there are no metrics found so far for pure numerical measurement. Thus, metrics for these fine-tuning mechanisms needed to be constructed based on the information at hand of software architectures.

For the literature based structure metrics, the analogy is used that each responsibility is equivalent to one operation in a class, and each class is a module or component, depending on what is used in the metric. As the concept of a responsibility is highly abstract, this most probably will not be the case if the system under construction would actually be implemented, but as there is no knowledge of what kind of operations each responsibility entails, this analogy seems justified enough.

I have chosen to measure the quality of a produced system in terms of modifiability and efficiency, with an added penalty for complexity. Modifiability and efficiency have both a positive and a negative sub-fitness. The sub-fitness functions are based on the metrics introduced in Chapter 2, although they have been combined and modified to achieve clear entities for measuring these selected quality values. The overall fitness is achieved by combining all sub-fitnesses, as described in Algorithm 12. As every sub-fitness has its own weight, the more desired quality can be weighted over the other, and thus achieve, e.g., a highly modifiable solution which may, however, lack in efficiency. Balancing the weights is especially important when measuring modifiability and

efficiency, as they are very much counter-qualities: highly modifiable architectures are rarely efficient, and efficient architectures are not especially modifiable.

In addition to modifiability and efficiency, a complexity sub-fitness has been constructed. It calculates the amount of classes and interfaces and penalizes especially large classes.

Algorithm 12 fitness
Input: chromosome *c*, list of weights *wl*
Output: double value *fitness*
fitness ← *wl*[0]*positiveModifiability(*c*) - *wl*[1]*negativeModifiability (*c*) + *wl*[2]*positiveEfficiency(*c*) - *wl*[3]*negativeEfficiency(*c*) - *wl*[4]*complexity (*c*)

The different ranges of the sub-fitness functions have been taken into account, and the values are adjusted so that the differences in end values of the fitnesses are solely caused by the weights given by user.

6.2.1. Efficiency

The efficiency of an architecture has much to do with structure, and how responsibilities are grouped to classes. Hence, common software metrics can very well be used to especially evaluate the positive efficiency of an architecture. Naturally, if the positive efficiency evaluator achieves very low values, it can be deduced that the architecture under evaluation is not very efficient. However, it is clearer to also construct a separate sub-fitness to evaluate those factors that have only a negative effect to the architecture, such as using the message dispatcher.

The positive efficiency sub-fitness is a combination of the cohesion metric [Chidamber and Kemerer, 1994], and evaluation of the grouping of responsibilities in classes. The grouping of responsibilities is good if there are many responsibilities in the same class that need a common responsibility, or many responsibilities that are needed by the same responsibilities are grouped in the same class. Furthermore, I have used the information-flow based approach [Seng et al., 2006] by multiplying the amount of connections with the parameter size relating to the called responsibility. Using the information-flow based version of cohesion serves two purposes: firstly, it is a standard quality metric, which increases the reliability of the results. Secondly, the evaluation of the structure is more detailed, and the information given of the responsibilities is better used, as the information-flow based metrics use the parameter size to evaluate the "heaviness" of a dependency between two responsibilities. The positive efficiency fitness can be expressed as ∑ (#(dependingResponsibilities within same class)∗ parameterSize + ∑ #(usedResponsibilities in same class) ∗ parameterSize + #(dependingResponsibilities in same class) ∗ parameterSize)).

The negative efficiency sub-fitness is a combination of the instability metric, as defined by Seng et al. [2006], and the amount of dispatcher connections. Instability is

well-suited for evaluating automatically generated architectures, as it is designed to measure the quality of the entire system. Amoui et al. [2006] have successfully used it as a part of their fitness function when evaluating architectures after the implementation of design patterns. Having the instability metric as an evaluator in an early stage will give a better base for further development. The negative effect of the dispatcher is further emphasized by multiplying the dispatcher connections with the call costs of those responsibilities that are called through the dispatcher. The negative efficiency sub-fitness can be expressed as $\sum$ ClassInstabilities + #(dispatcherCalls) $*$ $\sum$ callCosts.

As the re-grouping of individual metrics gives an even more powerful way to control the outcome as there was in my previous research [Räihä, 2008], the grouping of efficiency related metrics can be seen justified.

6.2.2. Modifiability

Although modifiability also deals with structure, and especially how much components depend on one another, an even bigger factor is the use of the message dispatcher and interfaces, as they effectively hide operations and thus highly increase modifiability. Thus, positive modifiability can be calculated as a result of using these fine-tuning mechanisms, and negative modifiability can be seen when components are highly dependable.

As mentioned, positive modifiability comes from the use of the dispatcher and interfaces. More specifically, the more connections between different responsibilities are handled through the dispatcher or interfaces, the more modifiable the architecture is. As there is no metric defined in the literature that would measure the effect of introducing interfaces to an architecture, such a metric had to be defined in order to prevent completely random incorporations of interfaces to the system. The logic behind the calculations is that an interface is most beneficial if there are many users for it. As there are no empty interfaces, i.e., an interface needs to be implemented by a responsibility belonging to the system, it can be concluded that an interface is well-placed if the responsibility implementing the interface in question is used by many other responsibilities. This increases reusability: changes to such a highly used responsibility have great impact on a system, and there is a big risk that the depending responsibilities may not get what they need from the changed responsibility. Thus, placing the needed responsibility behind an interface ensures that it will still service properly the responsibilities that need it even after it has been updated. The interface quality metric also considers how well the interface is implemented. A penalty is given for unused responsibilities in interfaces.

Thus, the positive modifiability sub-fitness can be calculated as a sum of responsibilities implementing interfaces, calls between responsibilities through interfaces and calls through the dispatcher multiplied by the variability factor of the called

responsibility, while taking in the account that unused responsibilities in interfaces are not appreciated. The positive modifiability sub-fitness can be expressed as #(interface implementors) + #(calls to interfaces) + (#(calls through dispatcher) $*$ $\sum$ (variabilities of responsibilities called through dispatcher)) – #(unused responsibilities in interfaces)$*$ 10. The multiplier 10 comes from unused responsibilities being nearly an architecture law, and thus the punishment should be grave.

The negative modifiability sub-fitness, as mentioned, comes from the amount of dependencies between different classes. As certain connections are already calculated in the efficiency sub-fitnesses, and there should be as little over-lapping between the different sub-fitnesses as possible, the actual sub-fitness for modifiability is quite simple, as it can expresse as #(calls between responsibilities in different classes). This actually captures the essence of both the coupling and RFC metrics [Chidamber and Kemerer, 1994]. These are both standard quality metrics, and using such highly recognized metrics increases the reliability of the results and confidence in the fitness function.

The contribution here is a new grouping of the underlying calculations, e.g., interface and dispatcher connections to achieve a clearer division of how different sub-fitnesses affect the fitness function as a whole. Since valuing highly the chosen modifiability sub-fitnesses produces solutions with a significantly increased amount of dispatcher connections and interfaces, as opposed to using a small weight to positive modifiability or appreciating efficiency, the chosen regrouping of different basic metrics can be seen as a success.

7. Experiments

In order to test how the fitness values of solutions develop and what kind of solutions are produced, the implementation should be tested with data that resembled a real system. For this purpose, example data for an electronic home control system was sketched.

The electronic home control system contains five main subsystems: logging in and user registry, temperature regulation, drape regulation, music system and coffee machine. The different subsystems may also contain smaller subsystems, e.g., the music system contains the speaker driver and the music file system. The main subsystems are connected to each other through a common user interface and main controller responsibilities, and the whole system has altogether 42 functional responsibilities and 10 data responsibilities with 90 dependencies between them. The detailed data set is given in Appendix A. I will now present both data from the development of the fitness curve in different cases, as well as some example solutions. The parameters for the fitness studies are given in Appendix B and the parameters for each example solution are given in Appendix C.

7.1. Fitness development

First, it was necessary to ensure that all the test runs have a similar fitness curve. That is, although architectures with the same fitness value may vary greatly, the development in overall architecture quality should be the same in every run to be able to somehow evaluate the end results. Figure 9 depicts the fitness curves of five test runs, all made with the same parameters, which are given in Tables 2 and 3 in Appendix B. The same mutation probabilities, given in Table 2, and weights, given in Table 3, were also used for tests, the results of which are shown in Figures 10 and 11. The actual curves are calculated as the average fitness values of the 10 best individuals in each generation. Thus, the curve shows how the fitness of the best individuals develops. As can be seen, the trends of the curves in Figure 9 are very similar, and thus averages of test runs can be confidently used to analyze the implementation. The following curves are calculated as the average of five tests.

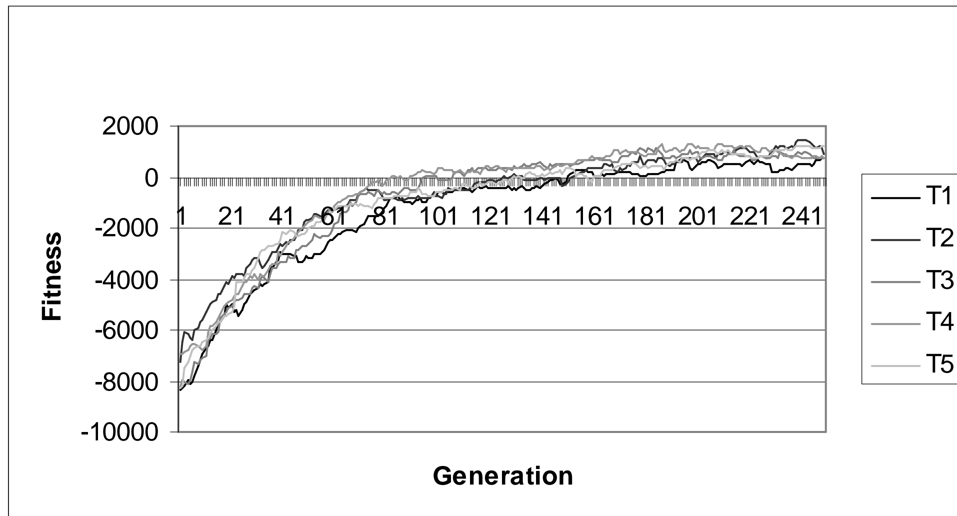


Figure 9. Fitness development of five test runs

Second, the effect of basic parameters, the size of population and number of generations, should be analyzed. In Figure 10, the fitness curves with different populations are shown. As can be seen, the fitness curve achieves higher values when the size of population is increased.

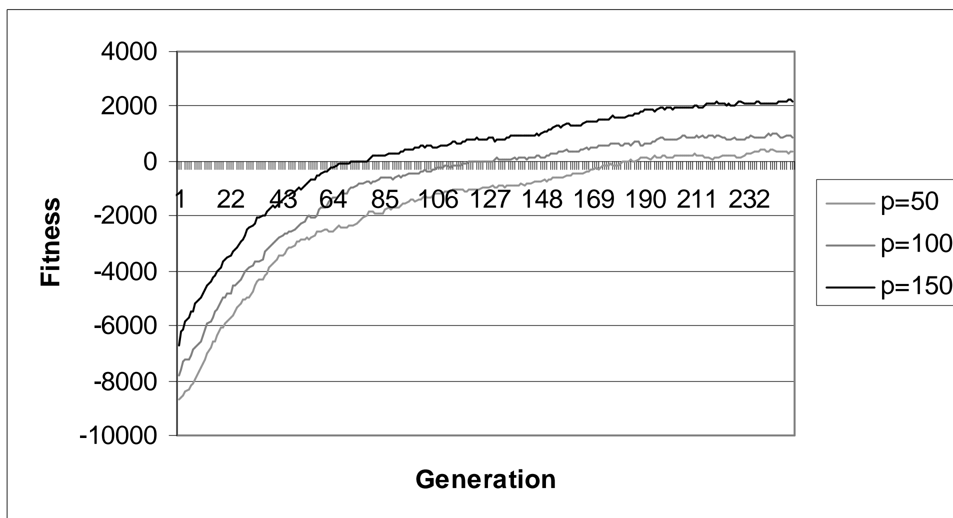


Figure 10. Different population sizes

The increase in fitness values is logical, as a bigger population makes it possible to have more variation within one generation of individuals. Thus, there is a bigger chance that a higher number of exceptionally good individuals are present in the population at the same time. This in turn reflects directly to the fitness curve in Figure 10, as it depicts the average development of the best individuals' fitness values.

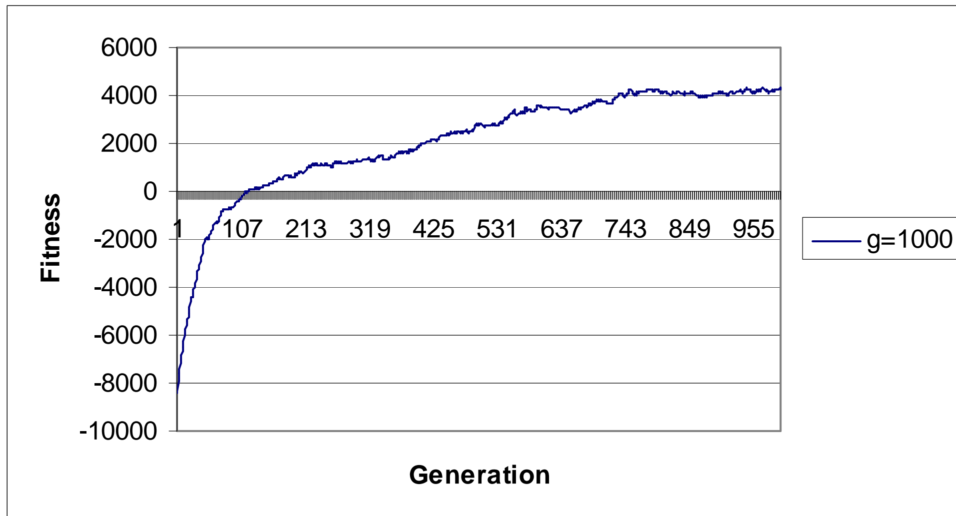


Figure 11. Fitness development over generations

The curve in Figure 11 shows the development of fitness values over 1000 generations. As can be seen, the fitness values develop very rapidly, and begin to achieve positive fitness values quite early, after only 100 generations. The development speed then slows down, but still stays significant up until around 750 generations. After this the development is quite stagnant, although still positive. This kind of development is customary for genetic algorithms when the fitness function is well defined. As this approach is novel in the way that architecture needs to be built completely 'from scratch', the rapid development during the first 100 generations is exceptionally noticeable: a legitimate architecture can be achieved after minimal genetic processing.

Third, it is interesting to examine the effect of weighting one of the chosen quality aspects over the other. The used weights are given in Tables 4 and 5 in Appendix B.

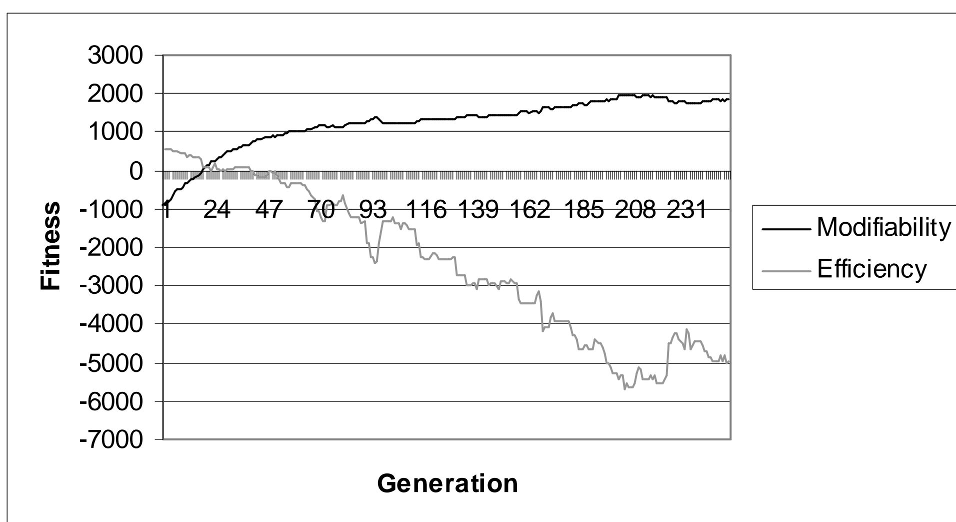


Figure 12. Modifiability weighted over efficiency

In Figure 12, modifiability is weighted over efficiency. The curves are 'normed', i.e., the values are adjusted so that they represent the true development of the sub-fitnesses, without the weights affecting the values. These curves represent the separated modifiability and efficiency curves of the individuals with the highest overall fitness in the population. When the modifiability and efficiency curves are separated, it can clearly be seen how difficult it is to achieve a solution that would be valued by both quality attributes. Weighting modifiability results in a 'normal' modifiability sub-fitness curve. However, the efficiency sub-fitness curve develops in a haphazard manner and actually worsens throughout the generations. As modifiability is valued, those solutions with a high modifiability sub-fitness survive for next generations, and in particular, make to the top of each generation. Such solutions have many dispatcher connections and a high usage of interfaces – these architectural decisions give a very negative efficiency value, and thus the development of the modifiability and efficiency curves is as can be expected.

In Figure 13, the opposite situation where efficiency is valued highly over modifiability is depicted. These curves differ drastically from the previous case, as there is hardly any development in either the efficiency or the modifiability curve, but they both remain quite stable. The biggest difference is that the efficiency curve achieves very high values from the very beginning and does develop a little, while the modifiability curve never achieves positive values and remains quite steadily around -500.

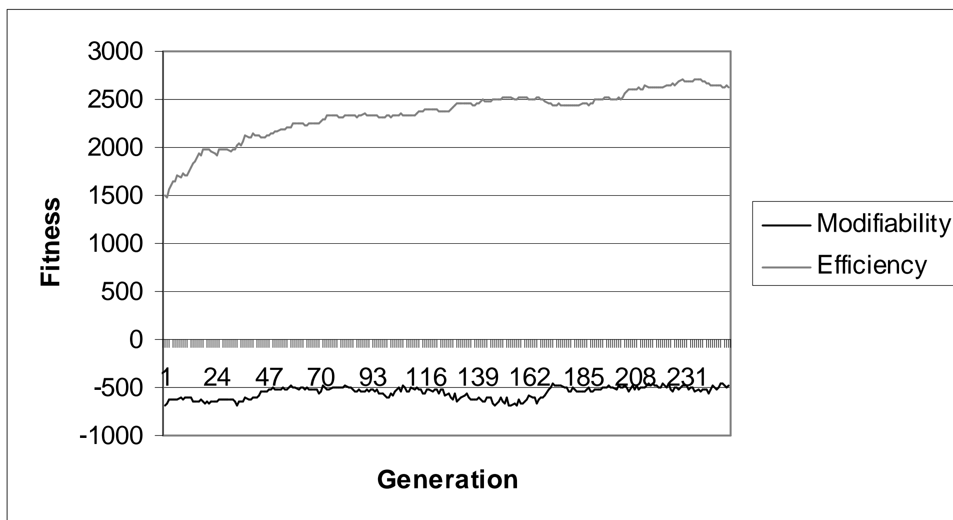


Figure 13. Efficiency weighted over modifiability

The reason for this kind of behavior on the fitness curves can be found within the initial population. The initial population has no instances of the message dispatcher, no interfaces and has a special individual where all responsibilities are in the same class. These individuals have an extremely high efficiency value, and thus are highly appreciated. As the crossover probability increases and mutation probabilities decrease with such fit individuals, highly efficient offspring are easy to produce early on in the

development. Also, with a population of 100, there are always so many highly efficient individuals that no individual with a high modifiability value could ever reach the top of the population. When the efficiency values are so high to begin with, it is also simply not even possible for the algorithm to develop the solutions in such a way that the fitness values would increase significantly.

As discussed in Chapter 6, I have used dynamic mutations to enhance the development of architecture by trying to take into account the level of detail of the applied patterns. I have compared the fitness curves achieved with these dynamic mutations to the traditional static mutations, where the mutation probabilities stay the same throughout the generations. The fitness curves with these different mutation probabilities are shown in Figure 14. The used parameters given in Tables 2 and 3 in Appendix B.

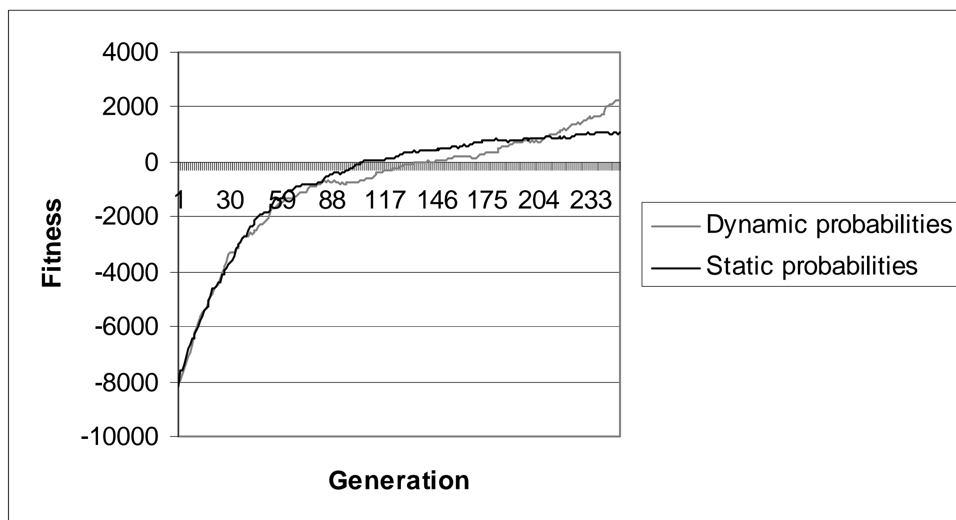


Figure 14. Different mutation rate applications

As can be seen, there is a slight difference in the behavior of the fitness curves. When dynamic mutations are used, the fitness values do not develop as steadily around 100-200 generations, but ascend in the end while. With static probabilities, however, the fitness curve is much more of the traditional kind: the values develop steadily and then seem to reach an optimum, as the values tend to stabilize around 1000. The period where the static mutation probabilities overcome the dynamic probabilities is the 'middle' section of generations, where the medium-level patterns, Façade in this implementation, is favored. When the change in mutation probabilities begin to favor the low-level design patterns, in this case Strategy, the development of fitness values turns is more again more significant. One explanation of why this is so, is that the selected middle-level pattern is not easy to apply in the example system. In fact, there are very few places where the application of a façade would be appropriate in the e-home example. Thus, when the Façade pattern is favored but there is nowhere to apply it,

these results in many unchanged individuals, which in turn would result in the slow development of fitness values. As the Strategy pattern is much easier to apply, favoring it and putting less emphasis on the implication of the Façade brings more variation to the populations and thus facilitates the more rapid development of the fitness curve.

7.2. Example solutions

Raw numerical fitness data does not reveal much of a produced solution: the fitness values merely benefit the analysis of how the values evolve, and ensuring that there is variance within the population. Statistics of the amount of classes or interfaces used does not reveal much of the structure either, apart from special cases when the number of classes is very small or very big, and likewise with the use of interfaces. In order to somehow evaluate the "goodness" of the produced solution, one needs a visualization to see how the responsibilities are distributed, how the patterns and dispatcher are used, and how clear the presented solution is overall. Thus, when evaluating the goodness of a solution from a class diagram, the evaluation is bound to be subjective, and people from different backgrounds may appreciate different things. I have evaluated the goodness of the solution based on three things: the structure, the use of patterns, and the use of dispatcher. The structure should be clear: there should not be a complex web of dependencies. Patterns should be used so that they clearly benefit the architecture and the structure should not suffer extensively. The dispatcher should also be used "with a thought"; the amount of connections to a dispatcher should remain reasonable and the connections should add a minimal amount of complexity to the system.

I will now present some example solutions achieved with the presented implementation. As these solutions demonstrate, it is difficult to achieve a solution, which could be appreciated from all viewpoints – normally a maximum of two of the three evaluation criteria (structure, patterns, dispatcher) have been satisfied. Thus, I have chosen examples which depict a good architecture from some viewpoint. Naturally, the goal is to achieve a combination of these solutions, so that the architecture would have a high overall quality that can be seen instantly with expert eyes.

In Figure 15, an example with quite a clear structure is shown. Here the different subsystems have been separated from each other, with the exception of the drape regulation system, and the responsibilities of individual subsystems have been well grouped. For instance, the coffee machine system has one big class with many responsibilities, from which several responsibilities have been extracted and put behind strategy-interfaces to support variability. The same design decisions have been made with the user registry subsystem – although in this case, some responsibilities belonging to the drape regulation system have been "falsely" combined in the same class with the user registry responsibilities. Also, the music system has been effectively combined with the speaker driver responsibilities. The user interface and main controller have been situated in their own classes, which call the other responsibilities. This is a very clear and

good design decision, as there is no need to combine these special responsibilities with anything else. This solution is fairly simplistic, as the only design pattern used is Strategy.

A similar solution is presented in Figure 16, which also has some big classes, quite a clear structure, and efficiently uses the Strategy pattern. Especially notable is the use of Strategy in the cases of measuring temperature and drape positions (Strategy1 at the very left side, and Strategy4 at the bottom right): these are operations that can quite naturally be varied depending on the circumstances and data at hand, and separating such operations behind an interface through Strategy, greatly increases the modifiability of the system.

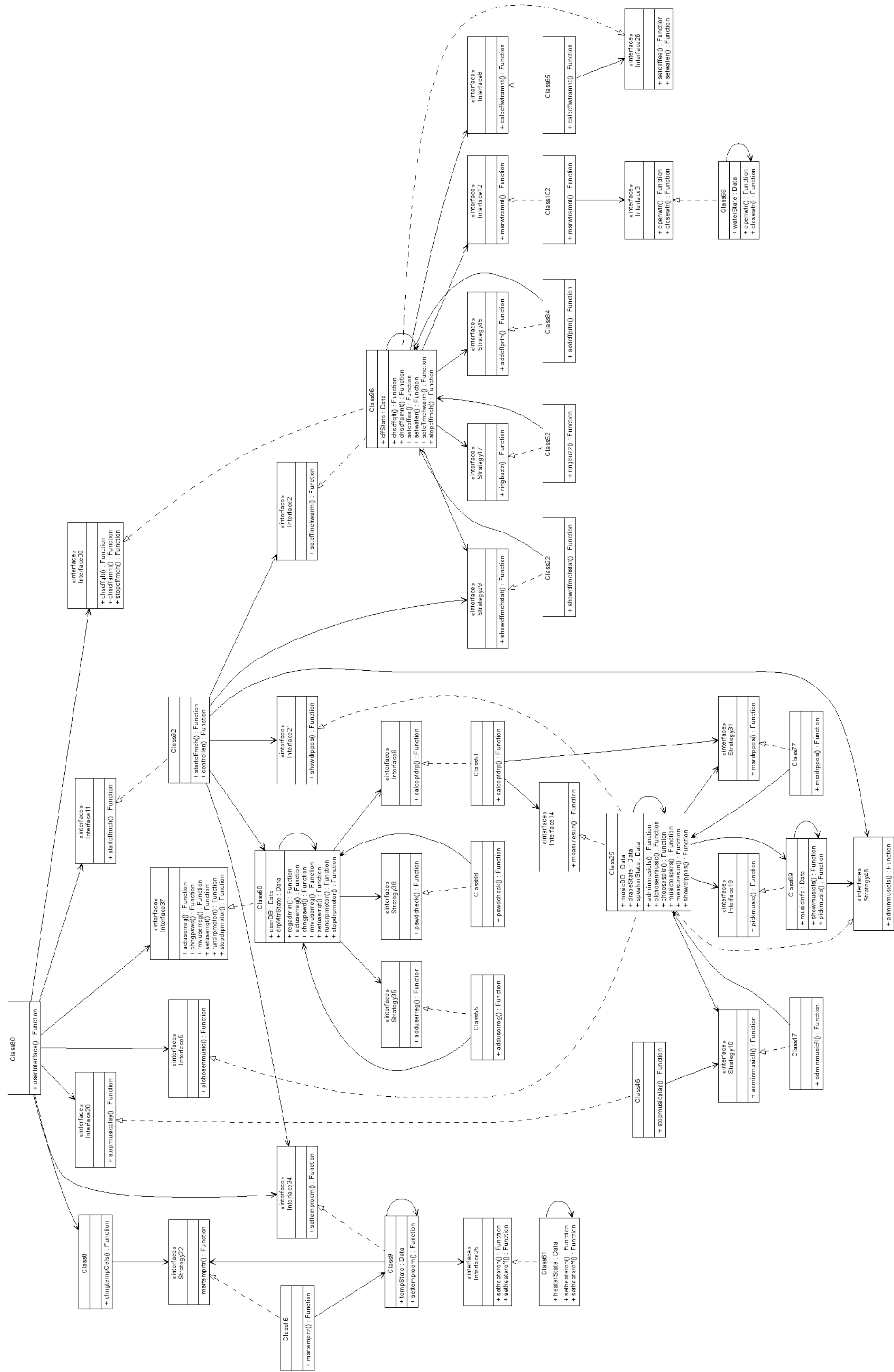


Figure 15. Example solution 1.

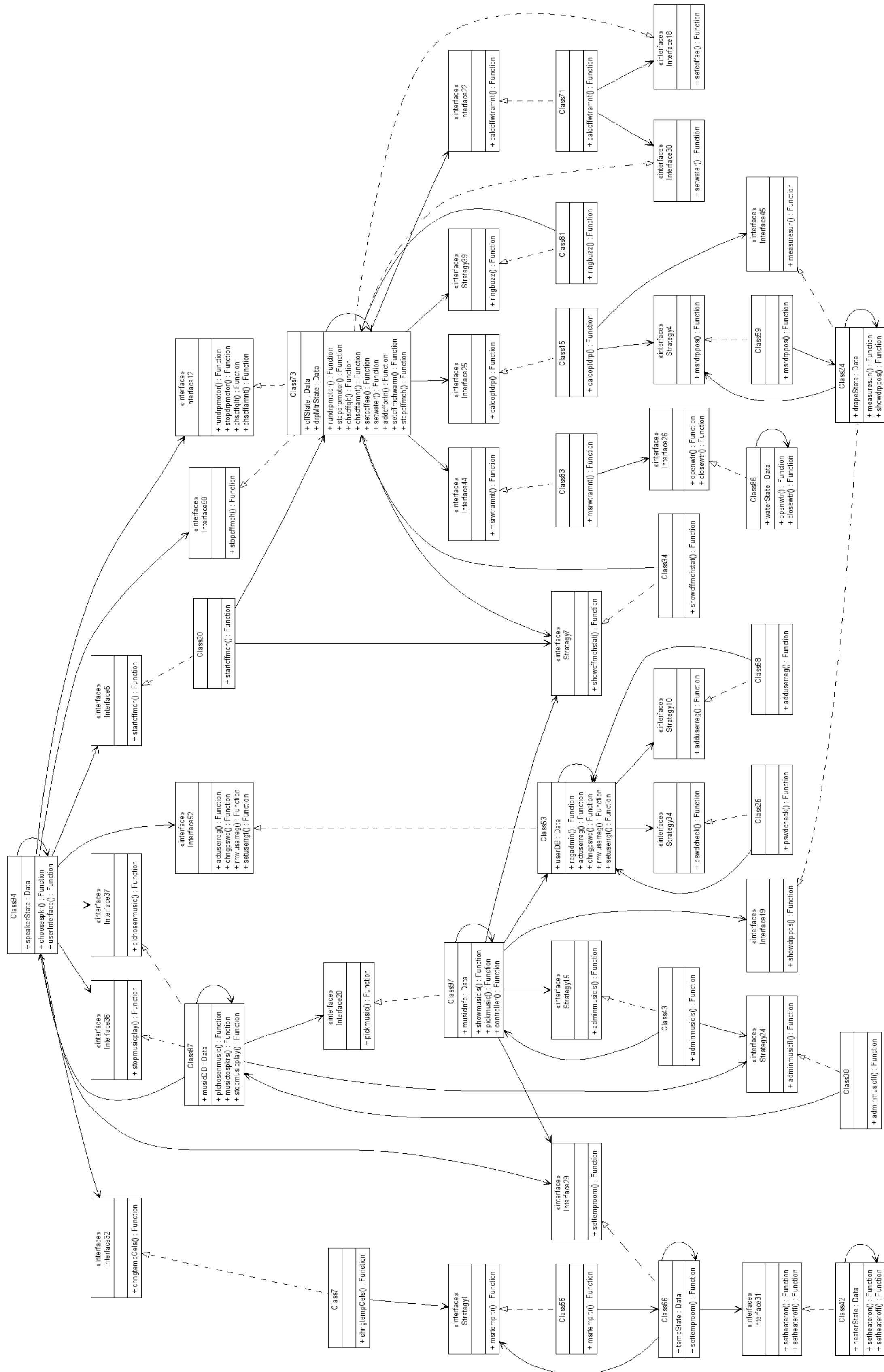


Figure 16. Example solution 2.

A solution that takes good advantage of the message dispatcher is depicted in Figure 17. Here the structure is not as clear as in Figures 15 and 16, and there are more dependencies between classes as in the solution in Figure 15. However, the solution is more modifiable, as many calls happen through the dispatcher, and thus the classes are not directly dependent on each other. For example, a good design decision in this case has been to put the water control subsystem behind the dispatcher, thus alleviating the dependency between the coffee machine and the water control. The benefits and shortcomings of using a message dispatcher architecture style are not straightforward. The positive effect in modifiability and the negative effect in efficiency are easy to see, but how the dispatcher affects the structure, the "beauty", of the solution, is more complicated. A well-placed message dispatcher may actually simplify the structure, and different components can easily be separated from the class diagram. A badly-placed dispatcher, on the other hand, can make the solution extremely messy, if there are many connections to and from the dispatcher and the class division of the responsibilities is not extremely successful. The solution shown in Figure 17 is somewhere between these extremes: the basic structure is successful enough so the use of message dispatcher does not make the solution completely incomprehensible, while some connections to the dispatcher clearly do more harm than good. In this case, the solution could benefit from either removing some connections to the dispatcher, or actually adding more connections, as the connections between classes would become more unified, and the dispatcher more central in the solution.

The Façade design pattern is not very often seen in the proposed solutions, as there are few places where it can be implemented in the example system. Also, as it adds complexity by bringing technical classes to the design, one might question the use of both a façade and a dispatcher in the same system. Figure 18 shows a solution where the Façade pattern has been used efficiently. Here, the user interface and the main controller call the façade interface instead of calling several responsibilities belonging to the many subsystems that these special responsibilities control. However, in this case the structure is not very good – there are many small classes and responsibilities from non-related subsystems have been grouped in the same class, e.g., class 24 contains responsibilities from the water control subsystem and the music system. The message dispatcher has also been brought to the system and used very inefficiently: only one responsibility uses the dispatcher to call two responsibilities through one interface.

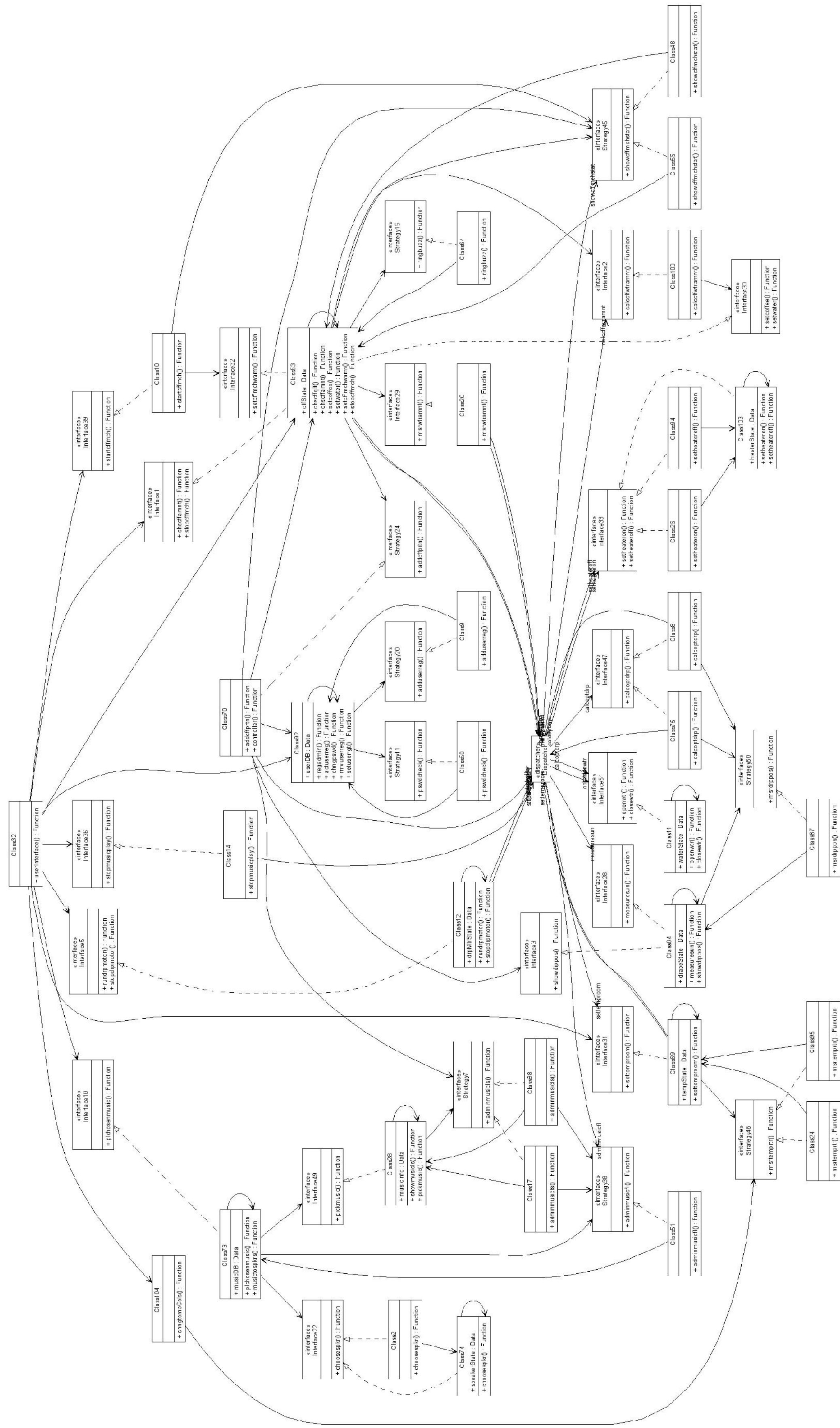


Figure 17. Example solution 3.

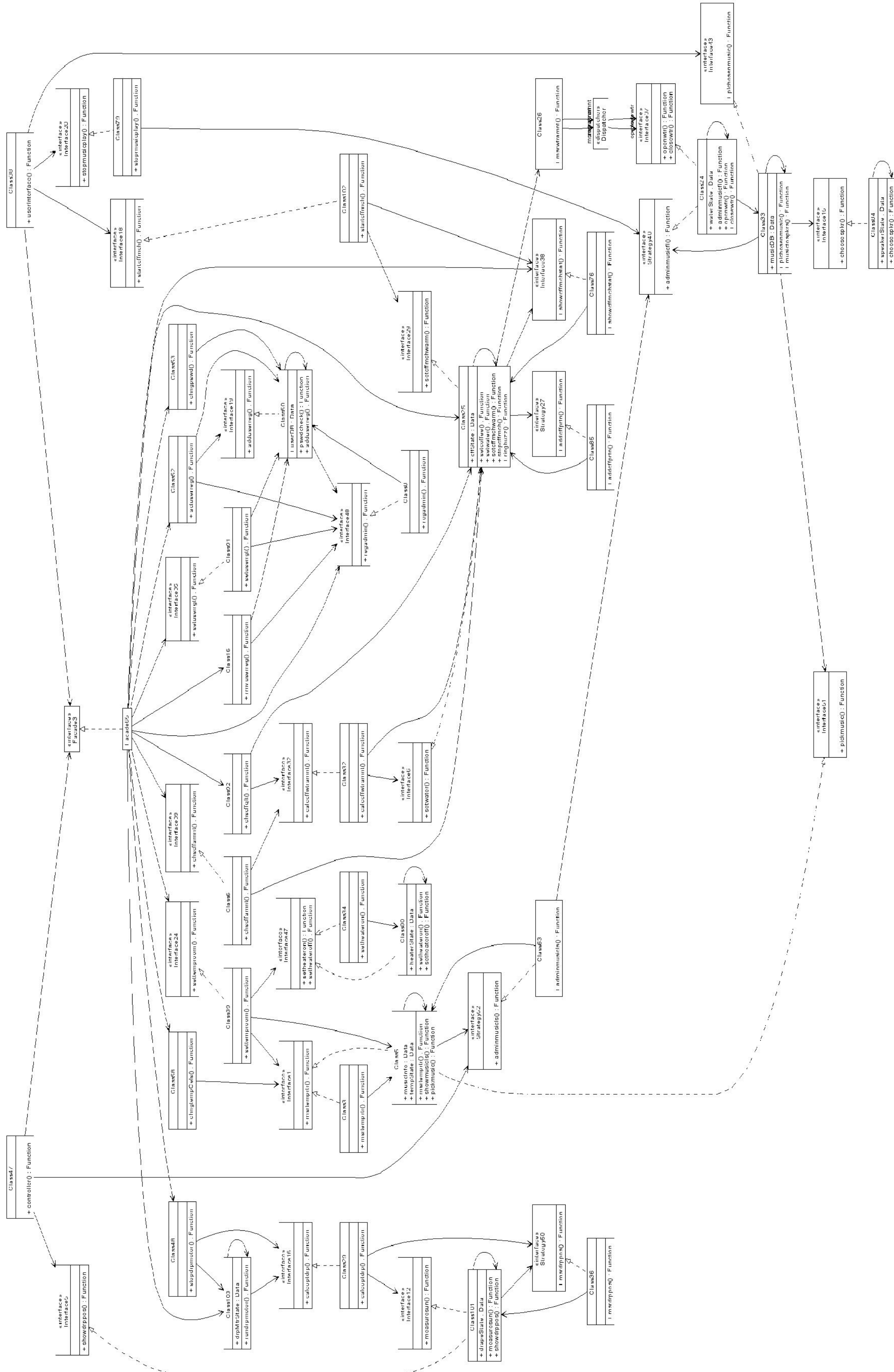


Figure 18. Example solution 4.

7.3. Remarks on adjusting the parameters

During the hundreds of test runs, some details about individual parameters and their relations to each other came apparent. I will now discuss the most remarkable findings of the tests.

Firstly, it could be seen that the fitness metrics have more influence to the final solution than the mutation probabilities. This came apparent when after nearly tripling the probability of using a dispatcher, the amount of dispatchers or their users did not significantly increase. Affect of the modifiability quality metric could, however, be seen instantly. Another point to make of mutation probabilities is that the probability of the crossover operation had a bigger effect, and after first starting with a crossover probability of 10%, in the end it was lowered down to 4%, which gave noticeably better results. The biggest probabilities are for splitting and merging classes, as they deal with the structure of the system. This has proven to be a good decision in optimizing the probabilities, as structured solutions are indeed achieved: if the probabilities of the fine-tuning mechanisms would be overpowering, the solutions would most likely be more randomly implementing the mechanisms, and the structure would rely heavily on randomized class division given in the creation of the initial population. The probabilities of bringing a dispatcher to the system and introducing a Façade pattern are also quite high. Thus giving a large probability for bringing the dispatcher to the system gives the algorithm a better chance to actually implement the mutations it chooses. The same applies for the façade mutation: implementing a façade requires quite a specific situation in the system, and thus often fails. Giving a larger probability will enhance the chances of actually achieving a Façade pattern in the system.

Secondly, there is the balancing problem. Naturally, the best solution would be efficient, modifiable, and simple. Thus, all the different fitnesses should be valued. However, finding real weights with which solutions that have an overall quality is not at all straightforward. Although the different ranges of the sub-fitnesses have been taken into account, some sub-fitnesses are still easier to affect, and thus achieve higher values with less work. For example, the positive modifiability and negative efficiency both measure the amount of dispatcher connections, and the overall values of these sub-fitnesses have been balanced so that the values are in the same range. However, when measuring the negative efficiency of a system, the number of dispatcher connections is multiplied with the call cost of the needed responsibility. This means that a single dispatcher connection has more effect in the efficiency than in positive modifiability. Thus, the weight for positive modifiability must be substantially higher than the weight for negative efficiency if one wants to achieve solutions that use the dispatcher effectively.

Balancing the weights is a difficult combinatorial optimization task also from the viewpoint of achieving solutions that would not have any big weaknesses. “Heavy” weights easily “cover” each other: a high positive value in some area may “cover” a high negative value, and leave the final value higher than that of other solutions which may have more balanced values, thus resulting in a solution with both very desirable and very undesirable qualities.

A particular balancing problem is in the structure, as the optimum solution would include medium-sized classes. However, the current metrics favor either very large (efficiency and complexity) classes or very small classes (modifiability). Large classes achieve high values from both positive efficiency and complexity, while small classes including only one or two responsibilities receive very small penalties from negative modifiability, hence keeping the overall fitness value quite high even though the positive metric value also stays small. This is actually a reverse situation of the “covering” discussed earlier: in this case both the positive and the negative values stay so small that the overall fitness value will still remain within the average and survive through many generations.

On a more detailed level, there are some restrictions regarding the fitness weights that should be considered. Firstly, the weights of complexity and positive efficiency should be quite similar, and quite low in relation to other metrics. Otherwise the number of classes will be very small, and the modifiability fitness will not be able to develop. At worst the solution only consists of a single class. Secondly, there is a delicate balance between negative efficiency and positive modifiability. The weight of positive modifiability should be at least five times higher than that of negative efficiency if one wants many users for the message dispatcher. Finally, the weight for negative modifiability should be kept within the same range as the weights for efficiency, but can be slightly higher. These observations are in the case when all quality metrics are used for evaluation.

To conclude: there are no absolute rules as to how the weights can be assigned, as they have a “see-saw” effect. By valuing some quality, another quality is hard to achieve. Rather, it should be attempted to balance the weights in such a way that no quality is completely overshadowed, thus making such a solution able that is somehow valued by all the quality aspects.

8. Conclusions

8.1. Presenting the results

I have presented a novel approach to software architecture design in the field of search-based software engineering. In this thesis I have taken a more abstract approach than the research done in the field so far as the structure of a software system is merely based on the concept of responsibilities, and no information of actual operations is known. Another contribution is to experiment with building a completely new architecture and not merely moving pieces in a ready-made system as done previously (see Chapter 4).

The results presented in Chapter 7 show that it is possible to design software architectures with this approach. Sensible solutions are achieved, and they can be controlled with the selection of fitness metrics – meaning that the construction of the architecture does indeed follow certain logic and is not completely random. The solutions mainly fell into two categories; either they had a good structure or they efficiently used design patterns and the dispatcher architecture style. This is natural, as when weighting efficiency, the structure was kept quite simple, and while weighting modifiability, design patterns and the dispatcher were vary much favored. Naturally, when neither quality aspect was weighted over the other, solutions were achieved that could be categorized as "somewhere in between", i.e., solutions with a good structure and some usage of interfaces and maybe one connection to the dispatcher, or good usage of the fine-tuning mechanisms and some structure. However, a truly good solution is one where the quality can be seen instantly – an average solution does not provide any new insights, as its biggest strength is actually its lack of weaknesses.

From the point of architecture design and architecture evaluation, the implementation presented here provides a strong starting point for further development where the common "laws" concerning architectural design can be taken more into account, thus ensuring quality solutions more consistently. In traditional architecture design, the software architect has the requirements for the system, and attempts to piece the respective operations together so the solution reaches high values when "measured" by some quality attributes. In this approach, the genetic algorithm actually evaluates a large number of architectures simultaneously, thus traversing through solutions a human architect would not have the time or the imagination to think of. Hence, as these initial tests already show that the implementation is able to find solutions greatly valued even after human analysis, this approach could affectively cut down the time used in architecture design as well as provide innovative solutions either as a starting point for further design or as ready architectures.

From the point of search-based software engineering and especially software design with the help of meta-heuristic search algorithms, this thesis clearly makes a contribution

to the field. As the starting point is raw data, and not a ready architecture, this approach gives the implementation a significantly freer traverse through the search space, thus resulting in more innovative solutions. When given a ready architecture, it can be assumed that the architecture is already of good quality. Thus, it might prove quite difficult to find such modifications to the architecture that would actually improve the fitness value. The higher level of abstractness in architecture representation also provides a better starting point for constructing architectures, as the logical entities in software systems do not always straightforwardly follow the structural or operational entities.

To conclude, the most important overall result of this thesis is that the approach taken here appears reasonably successful. An architecture can be designed with the help of a genetic algorithm with only abstract level knowledge of the architecture's contents and with no ready starting point.

8.2. Success evaluation

As discussed, the implementation was successful in proving that the selected approach was a good one. The produced solutions were also successful in either providing a good structure or good usage of communication mechanisms in the architecture. I will now discuss the main contributors to the success and also what could still be improved.

Firstly, the very basic elements needed for the algorithm to operate proved to be well chosen. That is, the modeling, mutations and crossover discussed in Chapter 5, provided the kind of basis for the algorithm to operate that it was possible to modify the architecture in such ways that would provide quality solutions.

Secondly, the fitness metrics proved to be very powerful, as weighting one fitness value over another showed instantly in the produced solutions. It should be emphasized that enhancing the fitness values by taking into account interfaces and dispatchers was not based on anything found in the literature, but was constructed by logically evaluating where these mechanisms would be best used. The grouping of different fitness metrics into clear sub-fitnesses proved to be powerful, thus making it a contribution of its own.

Finally, there is obviously much that can still be improved. The ultimate goal would be to consistently find solutions that are good from every quality aspect. Currently solutions with a high quality in any aspect are not consistent, and hardly any solutions are found with a good overall quality. Another improvement area lies within the "legality" of the produced architecture. The major reasons for not receiving even better results at this stage lies in the optimization problem with different fitness metrics. Naturally, the mutation probabilities also have impact in the solution, but as the implementation provided solutions from both "quality categories" by using the same set of probabilities, it is safe to assume that some sort of optimum combination of mutation probabilities has actually been found. As a result, further development lies more within the set of fitness weights in terms of parameter optimization.

Overall, it can be stated that the work was successful. The main research question was whether the selected approach would even be possible and sufficient enough to produce quality software architectures. As this was achieved, and there are clear views as to which direction the research could be taken, the implementation can indeed be viewed as successful, and the approach deemed possible.

8.3. Future work

The work presented in this thesis has been a follow-up to the work done in my Master's thesis, as the basic implementation has been taken further by introducing "architecture laws", design patterns, a more refined fitness function, and parameterizing mutation probabilities and other variables regarding the genetic algorithm. As the results have significantly improved from those introduced previously, there are even more grounds to further continue the work. There are still many ways with which the current implementation can be further developed and improved.

Firstly, the fine-tuning mechanisms can be put to better use by introducing more architecture styles and design patterns to the model. Naturally, the more mutations are introduced, the more the solution space grows. However, when the algorithm is not restricted to only a minimal amount of options in developing the architecture, the chances of finding a solution that is very close to the optimal increase significantly.

Secondly, more metrics should be considered, as evaluating an architecture from merely three quality viewpoints is hardly the case in a real world situation. As incorporating more "architecture laws" would also take care of some basic structural decisions, the existing metrics could also be adjusted to more effectively evaluate the structural decisions that are actually made by mutations and crossover.

Thirdly, the evaluation can be improved by making the fitness function dynamic. It could only evaluate the structure in the first generations, and then begin to evaluate the usage of fine-tuning mechanisms when they sufficiently exist in the architecture. As these mechanisms are not present in the initial population, this kind of adjusted fitness function could provide more quality structures.

Fourthly, another meta-heuristic search algorithm could be implemented in order to make a comparison between its results and the results provided by the genetic algorithm. If it is possible to model an architecture and achieve good results from, e.g., an implementation with simulated annealing, it could be researched whether the strengths of both the new and initial algorithm implementations could be combined.

Finally, the quality of the implementation can be improved by producing information of the solution in a format that can be easily modified, such as XMI.

To conclude: the work and results presented in this thesis are the beginning in a research approach with many possibilities. The basis has been made by producing a model and initial operations which can be further developed and combined to achieve significant results in the field of search-based software engineering.

References

[Alba and Chicano, 2007] E. Alba and J.F. Chicano, Software project management with GAs, *Information Sciences* **177**, 2007, 2380-2401.

[Amoui et al., 2006] M. Amoui, S. Mirarab, S. Ansari and C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation, *International Journal of Information Technology and Intelligent Computing* **1** (1, 2), June/ August, 2006, 235-245.

[Bass et al., 1998] L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[Blickle, 1996] T. Blickle, Evolving compact solutions in genetic programming: a case study In: H. Voigt, W. Ebeling, I. Rechenberg, and H. Schwefel (eds.), *Parallel Problem Solving from Nature IV*, *Proceedings of the International Conference on Evolutionary Computation,* LNCS **1141**, 1996, 564-573.

[Burgess, 1995] C.J. Burgess, A genetic algorithm for the optimisation of a multiprocessor computer architecture, In: *GALESIA'95, 1st IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, IEE Conference Publication* **414**, Sept. 1995, 39-44

[Canfora et al., 2005] G. Canfora, M. Di Penta, R. Esposito, M. L. Villani, An approach for qoS-aware service composition based on genetic algorithms, In: *GECCO 2005: Proceedings of the Genetic and Evolutionary Computation Conference*, June 2005, 1069–1075.

[Che et al., 2003] Y. Che, Z. Wang and X. Li, Optimization parameter selection by means of limited execution and genetic algorithms, In: X. Zhou et al. (Eds.): *APPT 2003*, LNCS **2834**, 2003, 226–235.

[Chidamber and Kemerer, 1994] S.R. Chidamber and C.F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* **20** (6), 1994, 476-492.

[Clarke et al., 2003] J. Clarke, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper and M. Shepperd, Reformulating Software Engineering as a Search Problem, *IEE Proceedings - Software*, **150** (3), 2003, 161-175.

[Di Penta et al., 2005] M. Di Penta, M. Neteler, G. Antoniol and E. Merlo, A language-independent software renovation framework, *The Journal of Systems and Software* **77**, 2005, 225-240.

[Dick and Jha, 1998] R.P. Dick and N.K. Jha, MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **17** (10),

Oct. 1998, 920-935.

[Doval et al., 1999] D. Doval, S. Mancoridis and B.S. Mitchell, Automatic clustering of software systems using a genetic algorithm, In: *Proceedings of the Software Technology and Engineering Practice*, 1999, 73-82.

[Du Bois and Mens, 2003] B. Du Bois and T. Mens, Describing the impact of refactoring on internal program quality. In: *Proceedings of the International Workshop on Evolution of Large-Scale Industrial Software Applications* 2003, 37-48.

[GraphViz, 2007] http://www.graphviz.org, checked 17.1.2008.

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

[Grunske, 2006] L. Grunske, Identifying "good" architectural design alternatives with multi-objective optimization strategies. In: *Proceeding of the 28th International Conference on Software Engineering,* Shanghai, China, 2006, 849 - 852.

[Harman et al., 2002] M. Harman, R. Hierons and M. Proctor, A new representation and crossover operator for search-based optimization of software modularization. In: *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, July 2002, 1351–1358.

[Harman and Tratt, 2007] M. Harman and L. Tratt, Pareto optimal search based refactoring at the design level, In: *GECCO 2007: Proceedings of the Genetic and Evolutionary Computation Conference,* 2007, 1106-1113.

[IEEE, 2000] *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE Standard 1471-2000, 2000.

[Jaeger and Mühl, 2007] M.C. Jaeger and G. Mühl, QoS-based selection of services: the implementation of a genetic algorithm, In T. Braun, G. Carle and B. Stiller (Eds.): *Kommunikation in Verteilten Systemen (KiVS) 2007 Workshop: Service-Oriented Architectures und Service-Oriented* Computing, VDE Verlag, March 2007, 359-371.

[Kazman et al., 2000] R. Kazman, M. Klein and P. Clements, ATAM: Method for architecture evaluation, Carnegie-Mellon University, Technical report CMU/SEI-2000-TR-004, August 2000.

[Koskimies ja Mikkonen, 2005] K. Koskimies ja T. Mikkonen, *Ohjelmistoarkkitehtuurit*. Talentum, 2005.

[Le Hanh et al., 2001] V. Le Hanh, K. Akif, Y. Le Traon and J-M. Jézéquel, Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies. In: J. Lindskov Knudsen (Ed.): *ECOOP 2001*, LNCS **2072**, 2001, 381-401.

[Losavio et al., 2004] F. Losavio, L. Chirinos, A. Matteo, N. Lévy and A. Ramdane-Cherif, ISO quality standards for measuring architectures. *The Journal of*

Systems and Software **72**, 2004, 209-223.

[Mens and Demeyer, 2001] T. Mens and S. Demeyer, Future trends in evolution metrics, In: *Proc. Int. Workshop on Principles of Software Evolution*, 2001, 83-86.

[Michalewicz, 1992] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolutionary Programs.* Springer-Verlag, 1992.

[de Miguel et al.,2000] M. de Miguel, T. Lambolais, S. Piekarec, S. Betgé-Brezetz and J, Péquery, Automatic generation of simulation models for the evaluation of performance and reliability of architectures specified in UML, In: *Revised Papers from the Second International Workshop on Engineering Distributed Objects*, LNCS **1999,** 2000, 83–101.

[Mitchell, 1996] M. Mitchell, *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[O'Keeffe and Ó Cinnéide, 2004] M. O'Keeffe and M. Ó Cinnéide, Towards automated design improvements through combinatorial optimization, In: *Workshop on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop -26th International Conference on Software Engineeering,* 2004, 75-82.

[O'Keeffe and Ó Cinnéide, 2007] M. O'Keeffe and M. Ó Cinnéide, Getting the most from search-based refactoring In: *GECCO 2007: Proceedings of the Genetic and Evolutionary Computation Conference,* 2007, 1114-1120.

[Räihä, 2008] O. Räihä, Applying Genetic Algorithms in Software Architecture Design, University of Tampere, Department of Computer Sciences, M.Sc. Thesis, February 2008.

[Reeves, 1995] C. R. Reeves, *Modern Heuristic Techniques for Combinatorial Problems.* McGraw-Hill Book Company, 1995.

[Rela, 2004] L. Rela, Evolutionary computing in search-based software engineering, Lappeenranta University of Technology, Department of Information Technology, M.Sc. Thesis, 2004.

[Rosenberg and Hyatt, 1997] L. Rosenberg and L. Hyatt, Software quality metrics for object-oriented design, available as http://satc.gsfc.nasa.gov/support/CROSS_APR97/oocross.PDF, checked 12.9.2007.

[Sahraoui et al., 2000] H.A. Sahraoui, R. Godin and T. Miceli, Can metrics help bridging the gap between the improvement of OO design quality and its automation? In: *Proc. of the International Conference on Software Maintenance (ICSM '00)*, 154-162, available as http://www.iro.umontreal.ca/~sahraouh/papers/ICSM00.pdf, checked 12.9.2007.

[Salomon, 1998] R. Salomon, Short notes on the schema theorem and the building block hypothesis in genetic algorithms, In: *Evolutionary Programming VII*, LNCS **1447**, 1998, 113-122.

[Seng et al., 2005] O. Seng, M. Bauyer, M. Biehl and G. Pache, Search-based improvement of subsystem decomposition, In: *GECCO 2005: Proceedings of the Genetic and Evolutionary Computation Conference*, 2005, 1045 – 1051.

[Seng et al., 2006] O. Seng, J. Stammel and D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, In: *GECCO 2006: Proceedings of the Genetic and Evolutionary Computation Conference*, 2006, 1909–1916.

[UMLGraph, 2007] http://www.umlgraph.org, checked 17.1.2008.

Case study data

The electronic home control system case study data. If the Depending responsibilities column has the value 0, then no responsibility uses the corresponding responsibility *i*. Type "f" stands for "functional" and "d" for "data". Groups of functional responsibilities forming a subsystem are separated by a bolded line.

Responsibility number	Depending responsibilities	Execution time (ms)	Parameter size	Frequency	Variability	Call cost	Name	Type
1	2,5	30	5.0	2	1	30	pswdcheck	f
2	3,4,6,7, 47	40	6.0	1	1	40	regadmin	f
3	48	30	6.0	1	1	30	actuserreg	f
4	3	30	6.0	1	1	30	adduserreg	f
5	48	50	8.0	1	1	50	chngpswd	f
6	48	60	2.0	1	1	60	rmvuserreg	f
7	48	70	5.0	1	1	70	setuserrgt	f
8	47,48	40	8.0	3	1	40	settemproom	f
9	8,10	60	4.0	3	2	60	msrtemprtr	f
10	48	20	4.0	3	1	20	chngtempCels	f
11	8	50	1.0	2	2	50	setheateron	f
12	8	50	1.0	1	2	50	setheateroff	f
13	14,47	70	9.0	5	2	70	adminmusicls	f
14	15	90	9.0	5	3	90	showmusicls	f
15	17	70	6.5	5	1	70	pickmusic	f
16	13,17,20	110	10.0	5	1	110	adminmusicfl	f
17	48	100	8.5	5	1	100	plchosenmusic	f
18	17	60	3.0	1	2	60	choosespkr	f
19	17	60	3.5	5	1	60	musictospkrs	f
20	48	50	3.0	1	1	50	stopmusicplay	f
21	24	80	7.0	3	1	80	measuresun	f
22	23,24	80	7.0	3	1	80	msrdrppos	f
23	47	70	5.0	3	1	70	showdrppos	f
24	25,26	90	6.5	3	3	90	calcoptdrp	f

25	48	60	2.0	2	1	60	rundrpmotor	f
26	48	50	1.0	2	1	50	stopdrpmotor	f
27	31,32,37, 47	110	10.5	2	2	110	showcffmchstat	f
28	48	40	5.0	2	3	40	chscffqlt	f
29	48	40	5.0	2	3	40	chscffamnt	f
30	28,29	50	6.0	2	1	50	calccffwtramnt	f
31	30	50	3.5	2	1	50	setcoffee	f
32	30	50	3.5	2	1	50	setwater	f
33	32	50	2.5	2	1	50	msrwtramnt	f
34	31	30	2.0	2	1	30	addcffprtn	f
35	33	30	1.0	2	2	30	openwtr	f
36	33	30	1.0	2	2	30	closewtr	f
37	48	70	2.0	2	2	70	startcffmch	f
38	37	70	3.5	2	1	70	setcffmchwarm	f
39	48	50	2.0	2	1	50	stopcffmch	f
40	39	20	2.0	2	1	20	ringbuzz	f
41	1,2,3,4,5, 6,7	10	2.0	7	1	10	userDB	d
42	16,17,19	10	2.0	3	1	10	musicDB	d
43	13,14,15	10	2.0	3	1	10	musicInfo	d
44	21,22,23	10	2.0	3	1	10	drapeState	d
45	27,28,29, 31, 32, 34, 38, 39, 40	10	2.0	5	1	10	cffState	d
46	8,9	10	2.0	3	1	10	tempState	d
47	0	10	2.0	3	2	10	controller	f
48	0	10	2.0	14	2	10	userInterface	f
49	11,12	10	2.0	2	1	10	heaterState	d
50	35,36	10	2.0	2	1	10	waterState	d
51	18	10	2.0	1	1	10	speakerState	d
52	25,26	10	2.0	2	1	10	drpMtrState	d

Table 1. Case study responsibility set.

Fitness study parameters

Mutation	Probability
Split class	0.08
Merge classes	0.06
Connect dispatcher	0.1
Remove dispatcher	0.04
Introduce interface	0.02
Remove interface	0.03
Introduce new dispatcher	0.09
Remove empty dispatcher	0.03
Introduce Façade	0.15
Remove Façade	0.02
Introduce Strategy	0.1
Remove Strategy	0.02
Use variability	0.08
Remove variability	0.06
Null	0.08
Crossover	0.04

Table 2. Mutation parameters

Sub-fitness	Weight
Positive modifiability	10
Negative modifiability	10
Positive efficiency	10
Negative efficiency	10
Complexity	10

Table 3. Fitness weights.

Sub-fitness	Weight
Positive modifiability	100
Negative modifiability	50
Positive efficiency	5
Negative efficiency	5
Complexity	5

Table 4. Fitness weights when modifiability overweighted.

Sub-fitness	Weight
Positive modifiability	5
Negative modifiability	5
Positive efficiency	100
Negative efficiency	50
Complexity	5

Table 5. Fitness weights when efficiency overweighted.

Example solutions' parameters

Example solution 1

Mutation	Probability
Split class	0.09
Merge classes	0.06
Connect dispatcher	0.13
Remove dispatcher	0.03
Introduce interface	0.02
Remove interface	0.03
Introduce new dispatcher	0.1
Remove empty dispatcher	0.03
Introduce Façade	0.13
Remove Façade	0.02
Introduce Strategy	0.08
Remove Strategy	0.02
Use variability	0.08
Remove variability	0.06
Null	0.08
Crossover	0.04

Table 7. Mutation probabilities for example solution 1.

Sub-fitness	Weight
Positive modifiability	45
Negative modifiability	15
Positive efficiency	10
Negative efficiency	5
Complexity	8

Table 8. Fitness weights for example solution 1.

Example solution 2

Mutation	Probability
Split class	0.08
Merge classes	0.06
Connect dispatcher	0.1
Remove dispatcher	0.04
Introduce interface	0.02
Remove interface	0.03
Introduce new dispatcher	0.09
Remove empty dispatcher	0.03
Introduce Façade	0.15
Remove Façade	0.02
Introduce Strategy	0.1
Remove Strategy	0.02
Use variability	0.08
Remove variability	0.06
Null	0.08
Crossover	0.04

Table 9. Mutation probabilities for example solution 2.

Sub-fitness	Weight
Positive modifiability	20
Negative modifiability	15
Positive efficiency	5
Negative efficiency	5
Complexity	10

Table 10. Fitness weights for example solution 2.

Example solution 3

Mutation	Probability
Split class	0.08
Merge classes	0.06
Connect dispatcher	0.1
Remove dispatcher	0.04
Introduce interface	0.02
Remove interface	0.03
Introduce new dispatcher	0.09
Remove empty dispatcher	0.03
Introduce Façade	0.15
Remove Façade	0.02
Introduce Strategy	0.1
Remove Strategy	0.02
Use variability	0.08
Remove variability	0.06
Null	0.08
Crossover	0.04

Table 11. Mutation probabilities for example solution 3.

Sub-fitness	Weight
Positive modifiability	45
Negative modifiability	15
Positive efficiency	15
Negative efficiency	2
Complexity	8

Table 12. Fitness weights for example solution 3.

Example solution 4

Mutation	Probability
Split class	0.08
Merge classes	0.06
Connect dispatcher	0.1
Remove dispatcher	0.04
Introduce interface	0.02
Remove interface	0.03
Introduce new dispatcher	0.09
Remove empty dispatcher	0.03
Introduce Façade	0.15
Remove Façade	0.02
Introduce Strategy	0.1
Remove Strategy	0.02
Use variability	0.08
Remove variability	0.06
Null	0.08
Crossover	0.04

Table 13. Mutation probabilities for example solution 4.

Sub-fitness	Weight
Positive modifiability	45
Negative modifiability	15
Positive efficiency	10
Negative efficiency	5
Complexity	8

Table 14. Fitness weights for example solution 4.