Outi Räihä, Hadaytullah, Kai Koskimies and Erkki Mäkinen

Synthesizing Architecture from Requirements: A Genetic Approach

DEPARTMENT OF COMPUTER SCIENCES
UNIVERSITY OF TAMPERE

D-2010-10

TAMPERE 2010

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCES
SERIES OF PUBLICATIONS D – NET PUBLICATIONS
D-2010-10, AUGUST 2010

Outi Räihä, Hadaytullah , Kai Koskimies and Erkki Mäkinen

Synthesizing Architecture from Requirements: A Genetic Approach

DEPARTMENT OF COMPUTER SCIENCES
FIN-33014 UNIVERSITY OF TAMPERE

ISBN 978-951-44-8218-2
ISSN 1795-4274

Synthesizing Architecture from Requirements: A Genetic Approach

Outi Räihä, Hadaytullah, Kai Koskimies

Department of Software Systems

Tampere University of Technology

Tampere, Finland

outi.raiha@tut.fi, hadaytullah@tut.fi, kai.koskimies@tut.fi

Erkki Mäkinen

Department of Computer Sciences

University of Tampere

Tampere, Finland

em@cs.uta.fi

Abstract

The generation of software architecture using genetic algorithms is studied with architectural styles and patterns as mutations. The main input for the genetic algorithm is a rudimentary architecture representing the functional decomposition of the system, obtained as a refinement of use cases. Using a fitness function tuned for desired weights of simplicity, efficiency and modifiability, the technique produces a proposal for the software architecture of the target system, with applications of architectural styles and patterns. The quality of the produced architectures is studied empirically by comparing these architectures with the ones produced by undergraduate students.

1. Introduction

The design of the architecture of a software system is traditionally regarded as the most crucial and creative task of software development. If the software is implemented on the basis of an inappropriate architecture, the consequences can be dramatic and cost dearly. To facilitate software architecture design, various methods have been proposed for choosing architectural solutions on the basis of their known (or assumed) effects on the quality attributes (e.g. (Matinlassi and Niemelä, 2002; Diaz-Pace et al., 2008; Kim et al., 2009)) and assessing informally the "goodness" of the architecture on the basis of a small set of prioritized test cases, scenarios (Clements et al., 2002). Still, in practice software architecture design is often based on unarticulated and even esthetic arguments. For example, a well-known limitation of human design is the Golden Hammer syndrome (Brown, 1998): once the designer has found a solution that works in one context, she tends to rely on the same solution in other, perhaps less appropriate contexts as well. Is software architecture design inherently a human activity, sensitive to all human weaknesses, or could it be automated to a certain degree? Given functional and quality requirements for a particular system, could it be possible to generate a reasonable software architecture design for the system automatically, thus avoiding human pitfalls? Besides being interesting from the viewpoint of understanding the character of software architecture, we see answers to these questions relevant from a pragmatic viewpoint as well. In particular, if it turns out that systems can successfully design systems, various kinds of software generators can optimize the architecture according to the application description, and self-sustaining systems (S3 2008) can dynamically improve their own architecture in changing environments.

To facilitate this study, let us make certain simplifying assumptions. First, we assume that the architecture synthesizer need not know anything about the semantics of the system functionality, but it can rely on a "null architecture" that gives the basic decomposition of the functionalities into components. We will later show how the null architecture is derived from use cases. Second, we assume that the architecture is obtained by augmenting the null architecture with applications of general architectural solutions available in a knowledge base. Such solutions are typically architectural styles and design patterns (Buschmann et al., 1996). Third, we assume that the goodness of an architecture can be inferred by evaluating a representation of the architecture mechanically against the quality requirements of the system. Each application of a general solution enhances certain quality attributes of the system, at the expense of others.

With these assumptions, software architecture design becomes essentially a search problem: find a combination of applications of the general solutions that satisfies the quality requirements in an optimal way. However, given multiple quality attributes and a large number of general solutions, the search space becomes huge for a system with realistic size. This leads us to the more refined research

problem discussed in this paper: to what extent we could use heuristic search methods, like genetic algorithms (GA) (Mitchell, 1996), to produce a reasonable software architecture automatically for certain functional and quality requirements?

The third assumption above is perhaps the most controversial. Since there is no exact definition of a good software architecture, and different persons would probably in many cases disagree on what is a good architecture, this assumption means that we can only approximate the evaluation of the goodness. Obviously, the success of a search method depends on how well we can capture the intuitive architecture quality in a formula that can be mechanically evaluated. In this paper, we consider three basic quality attributes, modifiability, efficiency, and simplicity. For modifiability and simplicity, we rely on the use of existing software metrics (Chidamber and Kemerer, 1994); for efficiency, we exploit knowledge about the effect of the solutions on efficiency, and possible (optional) information given by the designer about the assumed resource consumption of certain functionalities. In addition, the designer can give more precise modifiability requirements as change scenarios (Clements et al., 2002), taken into account in the evaluation of modifiability as well.

Although a number of heuristic search methods could be used here (Clarke et al., 2003), we are particularly interested in GA for two main reasons. First, the structural solutions visible in the living species in nature provide an indisputable evidence of the power of evolution in finding optimal system architectures. Second, crossover can be naturally interpreted for software architecture, as long as certain consistency rules are followed. Crossover can be viewed as a situation where two architects provide alternate designs for a system, and decide to merge their solutions, (hopefully) taking the best parts of both designs.

This chapter studies the generation of software architecture based on genetic algorithms. The proposed evolutionary software architecture generation process and GA realization are discussed in Sections 2 and 3, concretized with an example system. An account of an empirical experiment evaluating the current level of the quality of the genetically produced software architecture is given in Section 4. Related work is discussed in Section 5. Finally, we conclude with some remarks about the implications of the results and future directions of our work.

2. Overview of Evolutionary Software Architecture Generation

Software architecture can be understood in different ways. The definitions of software architecture usually cover the high-level structure of the system, but in addition to that, often also more process-related aspects like design rules and rationale of design decisions are included (IEEE, 2000). To facilitate our research, we adopt a narrow view of software architecture, considering only the static structural aspect, expressible as a UML (stereotyped) class diagram. In terms of the

4+1 views of software systems (Kruchten, 1995), this corresponds to a (partial) logical view. While a similar approach could be applied to generate other views of software architectures as well, there are some fundamental limitations in using heuristic methods. For example, it is very difficult to produce the rationale for the design decisions proposed by a heuristic method.

A central issue in our approach is the representation of the functional and quality requirements of the system, to be given as input for the genetic synthesis of the architecture. For expressing functional requirements we need to identify and express the primary use cases of the system, and refine them into sequence diagrams depicting the interaction between major components required to accomplish the use cases. This is a manual task, as the major components have to be decided, typically based on domain analysis.

In our approach, a so-called null architecture represents a basic functional decomposition of the system, given as a UML class diagram. No quality requirements are yet taken into account in the null architecture, although it does fulfill the functional requirements. The null architecture can be mechanically derived from the use case sequence diagrams: the (classes of the) participants in the sequence diagram become the classes, the operations of a class are the incoming call messages of the participants of that class, and the dependency relationships between the classes are inferred from the call relationships of the participants. This kind of generation of a class diagram can be easily automated (Selonen et al. 2005), but in the experiments discussed here we have done this manually.

Depending on the quality attributes considered, various kinds of information may need to be associated with the operations of the null architecture. In our study we consider three quality attributes: simplicity, modifiability, and efficiency. Simplicity is an operation-neutral property in the sense that the characteristics of the operations have no effect on the evaluation of simplicity. In contrast, modifiability and efficiency are partially operation-sensitive. For evaluating the modifiability of a system, it is useful to know which operations are more likely to be affected by changes than others. Similarly, for evaluating efficiency it is often useful to know something about the frequency and resource consumption of the operations. For example, if an operation that is frequently needed is activated via a message dispatcher, there is a performance cost because of the increased message traffic. To allow the evaluation of modifiability and efficiency, the operations can be annotated with this kind of optional information. If this information is insufficient, the method may produce less satisfactory results than with the additional information.

The specific quality requirements of a system are represented in two ways. First, the fitness function used in the GA is basically a weighted sum of the values of individual quality attributes. By changing the weights the user can emphasize or downplay some quality attributes, or remove completely certain quality attributes as requirements. Second, the user can optionally provide more specific quality requirements using so-called scenarios. The scenario concept is inspired by the ATAM architecture evaluation method (Clements et al., 2002), where scenarios are imaginary situations or sequences of events serving as test cases for the fulfil-

ling of a certain quality requirement. In principle, scenarios could be used for any quality attribute, but their formalization is a major research issue outside the scope of this work. Here we have used only modifiability scenarios, which are fairly easy to formalize. For example, in our case a scenario could be: “With 50% probability operation T needs to be realized in different versions that can be changed dynamically”. This is expressed for the GA tool using a simple formal convention covering most usual types of change scenario contents.

Figure 1 depicts the overall synthesis process. The functional requirements are expressed as use cases, which are refined into sequence diagrams. This is done manually by exploiting knowledge of the major logical domain entities having functional responsibilities. The null architecture, a class diagram, is derived mechanically from the sequence diagrams. The null architecture is used by the GA to first create an initial population of architectures and then, after generations of evolution, the final architecture proposal is presented as the best individual of the last generation. New generations are produced by applying a fixed library of standard architectural solutions (styles, patterns, etc.) as mutations, and crossover operations to combine architectures. During the evolution, the GA makes use of the parameters set by the user (concerning, e.g., the weights of the fitness function) and the scenarios in the evaluation of the individuals. The probabilities of mutations and crossover can be set by parameters as well. The GA part is discussed in more detail in the next section.

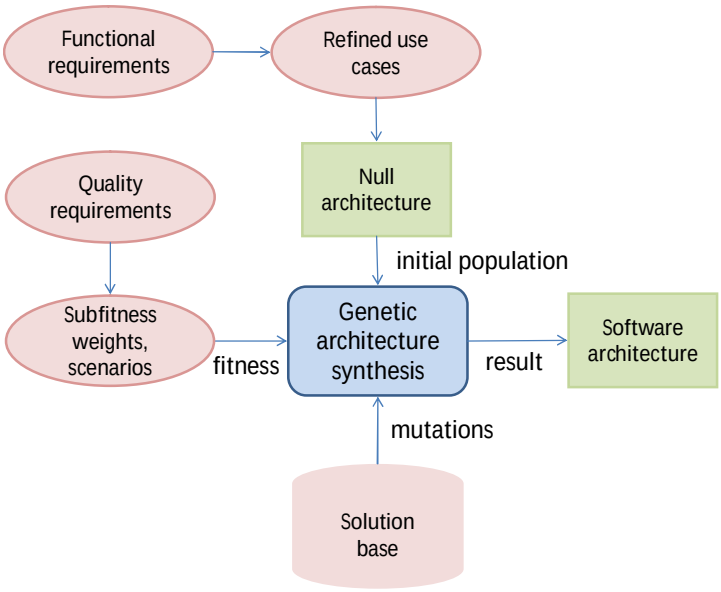

Fig. 1. Evolutionary architecture generation

3. Realizing Genetic Algorithms for Software Architecture Generation

Genetic algorithms (Mitchell, 1996) are generally used to find a good solution from a very large search space, the goal obviously being that the found solution is as good as possible. Each solution is encoded as a *chromosome*, which can be further divided into *genes*. When reproducing, *crossover* occurs: genes are exchanged between the pair of parent chromosomes. The offspring is subject to *mutation*, where gene values are changed. The *fitness* represents the quality of a solution. The set of chromosomes at hand at a given time is called a *population*.

3.1 Representing architecture

The genetic algorithm makes use of two kinds of information regarding each operation appearing in the null architecture. First, the basic input contains the call relationships of the operations taken from the sequence diagrams, other attributes like estimated parameter size, frequency and variability sensitiveness, and the null architecture class it is initially placed in. Second, the information gives the position of the operation with respect to other structures: the interface it implements and the design patterns (Gamma et al., 1995) and styles (Shaw and Garlan, 1996) it is a part of. The latter data is produced by the genetic algorithm. All data regarding an operation is encoded as a supergene. The chromosome handled by the genetic algorithm is gained by collecting the supergenes, i.e., all data regarding all operations, thus representing a whole view of the architecture. A more detailed specification of the architecture representation is given by Räihä et al. (2008a; 2008b).

The initial population is made by first encoding the null architecture into a chromosome form and creating the desired number of individuals. A random pattern is then inserted into each individual (in a randomly selected place). In addition, a special individual is left in the population where no pattern is initially inserted; this ensures versatility in the population.

3.2 Mutations and crossover

As discussed above, the actual design is made by adding patterns to the system. The patterns have been chosen so that there are very high-level architectural styles (dispatcher and client-server), medium-level design patterns (Façade and Mediator), and low-level design patterns (Strategy, Adapter and Template Method). The mutations are implemented in pairs of introducing a specific pattern or removing it. The dispatcher architecture style makes a small exception to this rule: the actual

dispatcher must first be introduced to the system, after which the operations can add new communicating pairs to the dispatcher.

The crossover is implemented as a traditional one-point crossover with a corrective function. This function ensures that the architecture stays coherent, as patterns may be broken by overlapping mutations. In addition to ensuring that the patterns present in the system stay coherent and "legal", the corrective function also checks that no anomalies are brought to the design, such as interfaces without any users.

The mutation (and crossover) points are selected randomly. However, we have taken advantage of the variability property of operations with the Strategy, Adapter and dispatcher communication mutations. The chances of a gene being subjected to these mutations increase with respect to the variability value of the corresponding operation. This should favor highly variable operations.

The actual mutation probabilities are given as input. Selecting the mutation is made with a "roulette wheel" selection (Michalewicz, 1992), where the size of each slice of the wheel is in proportion to the given probability of the respective mutation. Null mutation and crossover are also included in the wheel. The crossover probability increases linearly in relation to the fitness rank of an individual, which causes the probabilities of mutations to decrease in order to fit the larger crossover slice to the wheel. Also, after crossover, the parents are kept in the population for selection. These actions favor strong individuals to be kept intact through generations. Each individual has a chance of reproducing in each generation: if the first roulette selection lands on a mutation, another selection is performed after the mutation has been administered. If the second selection lands on the crossover slice, the individual may produce offspring. In any other case, the second selection is not taken into account, i.e., the individual is not mutated twice.

3.3 Fitness function

The quality function is based on software product metrics, most of which are from the metrics suite introduced by Chidamber and Kemerer (1994). These metrics have been used as a starting point for the quality function, and have been further developed and grouped to achieve clear "sub-functions" for modifiability and performance, both of which are measured with a positive and negative metric. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. A simplicity metric is added to penalize having many classes and interfaces.

Dividing the evaluation function into sub-functions gives the possibility emphasize certain quality attributes and downplay others. Denoting the weight for the

respective sub-function sf_i with w_i, the core evaluation function $f_c(x)$ for architecture x can be expressed as

$f_c(x) = w_1*sf_1 - w_2*sf_2 + w_3* sf_3 - w_4* sf_4 - w_5* sf_5$.

Here, sf_1 measures positive modifiability, which takes into account how well interfaces are used (number of implementing operations, number of calls through interfaces, and the sensitiveness to variation of operations called through an interface). Also the interfaces provided by design patterns are considered here. In addition, calls between operations that are handled via message dispatcher are rewarded, and the sensitiveness to variation of the operations is used to enhance the reward. Negative modifiability is calculated in sf_2 by penalizing calls between classes where the required operation is not placed behind a design pattern or called via message dispatcher or server.

As for efficiency, sf_3 measures positive efficiency by checking how well operations are placed within classes. We reward structures which lead to minimal amount of calls between different classes as well as calls between operations within the same class. The operation's required amount of data is also considered and used to increase the reward. Negative efficiency (sf_4) in turn counts the relation of calls between classes and within classes, and the amount of calls to the message dispatcher and through servers. The effect of using a dispatcher or server to call an operation is further emphasized by taking into account the frequency of calls to the operations involved.

Finally, complexity (or negative simplicity) is penalized in sf_5 by calculating the amount of classes and interfaces.

Additionally, scenarios can be used for more detailed fitness calculations. Basically, a scenario describes an interaction between a stakeholder and the system (Bass et al., 1998). In our approach we have concentrated only on change scenarios. We have categorized each scenario in three ways: is the system changed or is something added; if changed, does the change concern semantics or implementation of the operation, and whether the modification should be done dynamically or statically. This categorization is the basis for encoding the scenarios. In addition, each encoding of a scenario contains information of the operation it affects, and the probability of the scenario occurrence. Räihä et al. (2009) explain the scenario encoding in more detail.

Each scenario type is given a list of preferences according to the general guidelines of what is a preferable way to deal with that particular type of modification. These preferences are general, and do not in any way consider the specific needs or properties of the given system.

When scenarios are encoded, the algorithm processes the list of given scenarios, and compares the solution for each scenario to the list of preferences. Each solution is then awarded points according to how well it supports the scenarios, i.e., how similar the partial solutions regarding individual operations are to the given preferences.

Formally, the scenario sub-quality function sf_s can be expressed as

$sf_s = \sum$scenarioProbability*100/scenarioPreference.

Adding the scenario sub-quality function to the core quality function results in the overall quality, $f(x) = f_c(x) + w_s * sf_s$.

4. Application

4.1 Creating input

As an example system, we will use the control system for a computerized home, called ehome. Use cases for this system are assumed to consist of logging in, changing the room temperature, changing the unit of temperature, making coffee, moving drapes, and playing music. In Fig. 2, the coffee making use case has been refined into a sequence diagram. Since we are here focusing on the architecture of the actual control system, we ignore user interface issues and follow a simple convention that the user interface is represented by a single (subsystem) participant that can receive use case requests. Similarly, in the null architecture the user interface is in this example represented by a single component that has the use cases as operations.

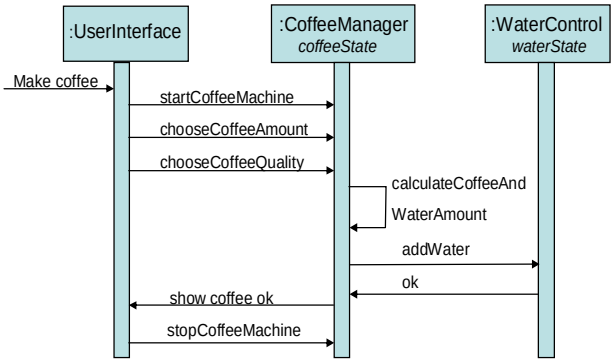

Fig. 2. Make coffee use case refined

To refine this use case, we observe that we need further components. The main unit for controlling the coffee machine is introduced as CoffeeManager; additionally there is a separate component for managing water, this is named as WaterControl. If a component has a significant state or it manages a significant data ent-

ity (like, say, a data base), this is added to the participant box. In this case, CoffeeManager and WaterControl are assumed to have significant state information.

In this case, the null architecture in Figure 3 can be mechanically derived from the use case sequence diagrams.

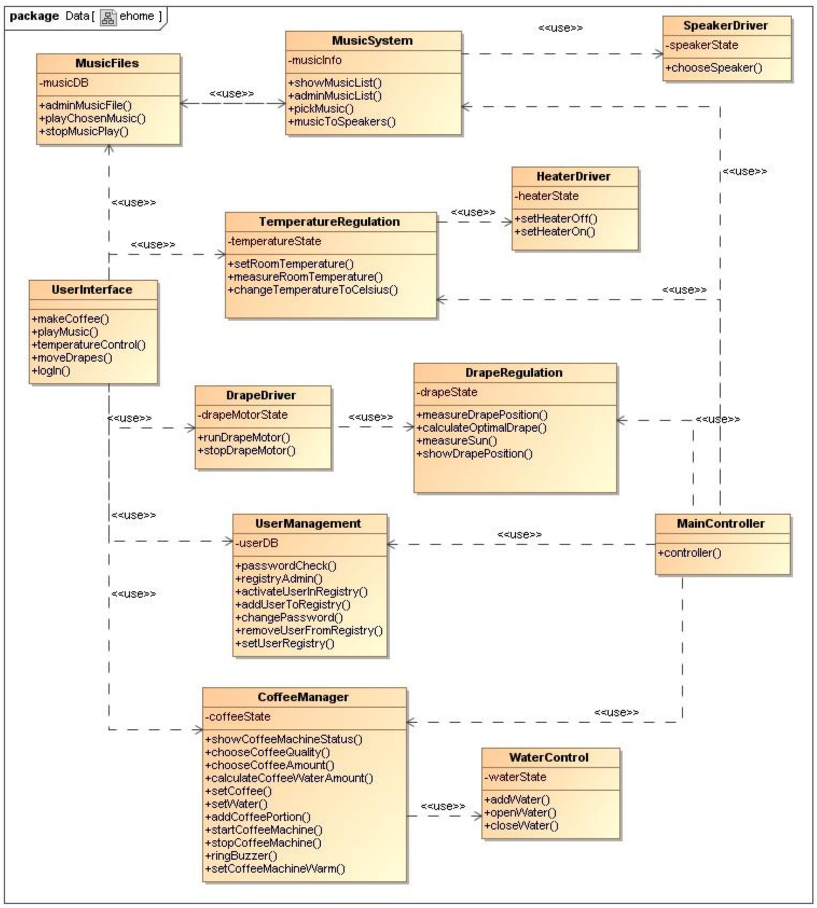

Fig. 3. Null architecture for ehome

After the operations are derived from the use cases, some properties of the operations can be estimated to support the genetic synthesis, regarding the amount of data an operation needs, frequency of calls, and sensitiveness for variation. For example, it is likely that the coffee machine status can be shown in several different ways, and thus it is more sensitive to variation than ringing the buzzer when

the coffee is done. Measuring the position of drapes requires more information than running the drape motor, and playing music quite likely has a higher frequency than changing the password for the system. Relative values for the chosen properties can similarly be estimated for all operations. This information, together with operation call dependencies, is included in the null architecture (not visible in Figure 3),

Finally, different stakeholders' viewpoints are considered regarding how the system might evolve in the future, and modifiability scenarios are formulated accordingly. For example, change scenarios for the ehome system include:

- the user should be able to change the way the music list is showed (90%)
- the developer should be able to change the way water is connected to the coffee machine (50%)
- the developer should be able to add another way of showing the coffee machine status (60%).

A total of 15 scenarios were given for the ehome system.

4.2 Experiment

In our experiment, we used a population of 100 and 250 generations. The fitness curves presented are averages of 10 test runs, where the actual y-value is the average of 10 best individuals in a given population.

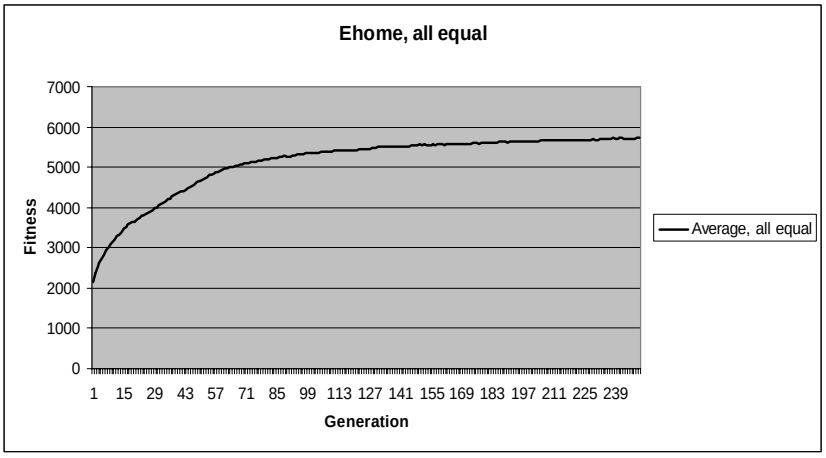

Fig. 4. Fitness development, all quality factors equal

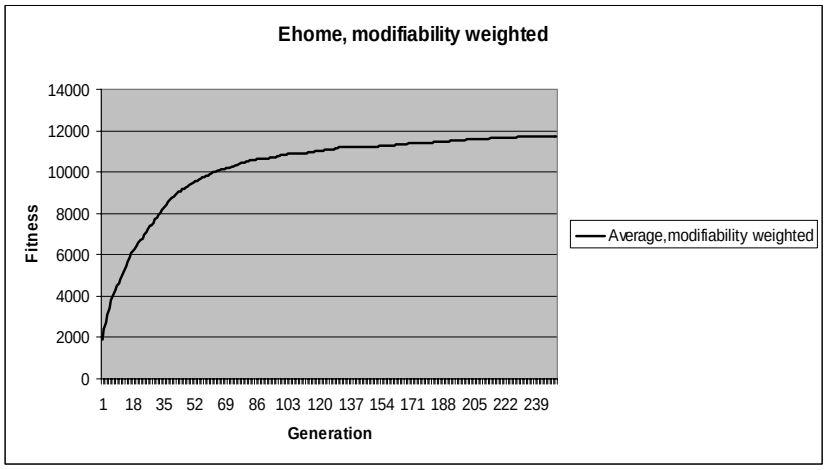

Fig. 5. Fitness development, modifiability weighted

We first set all the weights to 1, i.e., did not favor any quality factor over another. The fitness curve for this is given in Figure 4, and an example architecture is presented in Figure 6. To study the effect of the weighting we then assumed that modifiability should be emphasized, weighting positive modifiability over other quality attributes. Simultaneously negative efficiency was given a smaller than usual weight, to indicate that possible performance penalty of solutions increasing modifiability is not crucial. The fitness curve for this experiment is given in Figure 5 and an example solution is depicted in Figure 7. As can be seen, both fitness curves develop steadily. The higher values in Fig. 5 can be quite straightforwardly explained by the larger weight for efficiency, which yields in larger values. However, the fitness improvement between 1 and 100 generations (where most of the development happens) is also significantly larger (9000) when modifiability is weighted when compared to the standard test run in Figure 5 (fitness improvement 3500).

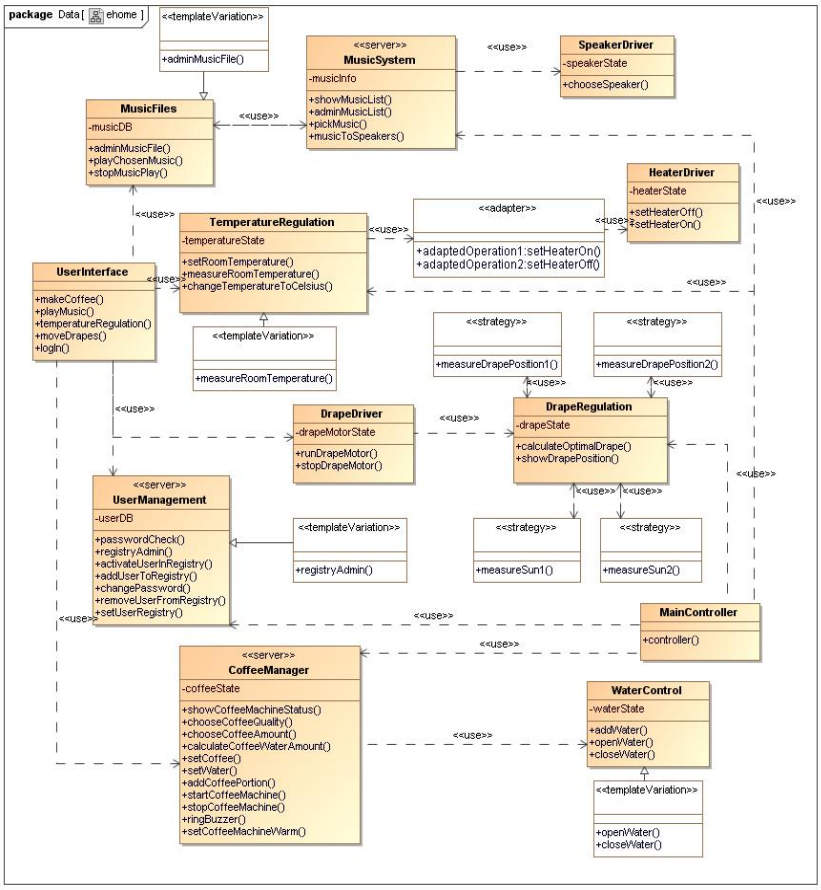

Fig. 6. Example architecture for ehome when all quality factors are equally weighted

The architecture in Figure 6, achieved with equal weights for all quality attributes, is quite simple. There are instances of all low-level patterns (Adapter, Template Method and Strategy), and the client-server architecture style is also applied. The patterns seem quite well placed, for example in the case of DrapeRegulation, the operations that are likely to contain much calculation (measureSun and measureDrapePosition) have been placed behind a Strategy pattern. In Figure 7 the solution with overweighted modifiability is quite different. The biggest difference is the presence of the dispatcher architecture style, and there are also many more Strategy patterns than in the solution where all quality factors are equally weighted. This is a natural consequence of the weighting: the dispatcher has a significant positive effect on modifiability, and since it is not punished too much for

inefficiency, it is fairly heavily used as a communication pattern. The same applies to Strategy, although in smaller scale.

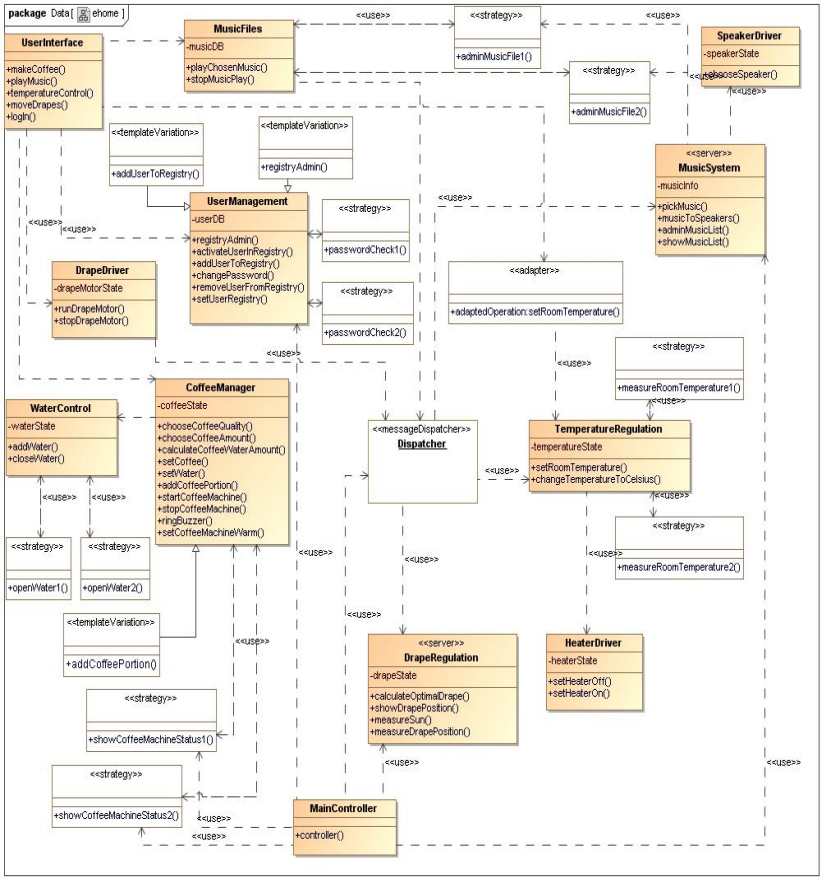

Fig. 7. Example architecture for ehome when modifiability is weighted over other quality factors

5. Empirical study on the quality of synthesized architectures

As shown in the previous section, genetic software architecture synthesis appears to be able to produce reasonable architecture proposals, although obviously they still need some human polishing. However, since the method is not deterministic,

it is essential to understand what is the goodness distribution of the proposals, that is, to what extent the architect can rely on the quality of the generated architecture. To study this, we carried out an experiment where we wanted to relate the quality of the generated architectures to the quality of the architectures produced by students. The setup and results of this experiment are discussed in the sequel.

5.1 Setup

5.1.1. Producing architectures

First, a group of 10 students from an undergraduate software engineering class was asked to produce an architecture for the ehome system. Most of the students were third year Software Systems majors from Tampere University of Technology, having participated in a course on software architectures. The synthesized solutions were achieved in 10 runs, resulting in 10 architecture proposals. The purpose of the study was to relate the quality of these proposals to the quality of student solutions. The setup for the synthesized architectures was the same as given in Section 4, where modifiability was weighted over other quality attributes.

The students were given essentially the same information that is used as input for the GA, that is, the null architecture and the scenarios. They were asked to design the architecture for the system, using only the same architecture styles (message dispatcher and client-server) and design patterns (façade, mediator, strategy, adapter, template method) that were available for GA. The students were instructed to consider performance, modifiability and simplicity in their designs, with an emphasis on modifiability.

5.1.2. Evaluating architectures

After the students had returned their designs, the assistant teacher for the course (impartial to the GA research) was asked to grade the designs as test answers on a scale of 1 to 5, 5 being the highest. The solutions were then categorized according to the points they achieved. From the categories of 1, 3 and 5, one solution for each category was randomly selected. These architectures were presented as grading examples to four software engineering experts.

The experts were researchers and teachers at the Department of Software Systems at Tampere University of Technology. They all had a M.Sc. or a Ph.D. degree in Software Systems or in a closely related discipline and several years of ex-

pertise from software architectures, gained by research or teaching. The experts were given 10 pairs of architectures. One solution in each pair was a student solution and one a synthesized solution. The solutions were edited in such a way that it was not possible for the experts to know which solution was synthesized. The experts were then asked to grade each solution with points 1, 3 or 5. They were given the same information as the students regarding the requirements.

5.2 Results

All points given by the experts to all the synthesized architectures are shown in Table 1, along with calculated averages for each synthesized solution (g1-g10), each expert (e1-e4), and all solutions. We excluded the cases (g2, g6 and g8) where the evaluation was considered inconclusive (both 1 and 5 appeared in the points). Solution g7 is presented in Figure 7 (Section 4.2) as an example of the synthesized solutions.

Table 1. Points for synthesized solutions

		Experts				
		e1	e2	e3	e4	Average
Solutions	g1	3	5	3	3	3.5
	g3	5	5	3	3	4
	g4	1	1	1	3	1.5
	g5	3	3	3	5	3.5
	g7	3	1	1	3	2
	g9	3	1	3	3	2.5
	g10	3	5	3	5	4
	Average	3	3	2.4	3.6	3

The synthesized solutions have a total average of 3 points, while the student solutions have a total average of 2.4 points. Two of the experts valued the synthesized solutions on average higher than the student solutions, for one (e3) they were just as good, and one (e1) valued the student solutions higher. There are some synthesized solutions that are clearly worse than average while some others are better than average.

The best synthesized solutions appear to be g3 and g10, with an average of 4 points. In solution g3 the message dispatcher was used, and there were quite few patterns, so the design seemed easily understandable while still being modifiable. However, g10 was quite the opposite: the message dispatcher was not used, and there were especially as many as eight instances of the Strategy pattern, when g3 had only two. There were also several Template and Adapter pattern instances. In

this case the solution was highly modifiable, but not nearly as good in terms of simplicity. This demonstrates how very different solutions can be highly valued with the same evaluation criteria, when the criteria are conflicting (it is quite impossible to achieve a solution that is at the same time optimally efficient, modifiable and still understandable).

The worst solution was considered to be g4, with an average of 1.5 points. This solution used the message dispatcher, but also the server style was eagerly applied. There were not very many patterns, and the ones that existed were quite poorly applied.

To summarize, the experiments suggest that, using this kind of application of GA, genetic software architecture synthesis works roughly at the level of an undergraduate student.

6. Related work

Search-based software engineering applies meta-heuristic search techniques to software engineering issues that can be modeled as optimization problems. A comprehensive survey of applications in search-based software engineering has been made by Harman et al. (2009). Recently, there has been increasing interest in software design in the field of search-based software engineering. A survey on this subfield has been conducted by Räihä (2009). We will now briefly discuss the most prominent studies in the field of search-based software design.

Bowman et al. (2008) study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes within given constraints. So far they do not demonstrate assigning methods and attributes "from scratch" (based on, e.g., use cases), but try to find out whether the presented MOGA can fix the structure if it has been modified.

Simons and Parmee (2007a; 2007b) take use cases as the starting point for system specification. Data is assigned to attributes and actions to methods, and a set of uses is defined between the two sets. The notion of class is used to group methods and attributes. This approach starts with pure requirements and leaves all designing to the genetic algorithm. The genetic algorithm works by changing the allocation of attributes and methods. However, no design choices beyond class structure are made, leaving the end result simpler than what is the goal with our approach.

Amoui et al. (2006) use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. The authors' goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. (1995). From the software design perspective, the transformed design of the best chromosomes are evolved so that abstract packages become more abstract and concrete packages in

turn become more concrete. This approach only uses one quality factor (reusability), and also a more refined starting point than what is used in our approach.

Seng et al. (2005) describe a methodology that computes a subsystem decomposition that can be used as a basis for maintenance tasks by optimizing metrics and heuristics of good subsystem design. GA is used for automatic decomposition. If a desired architecture is given, and there are several violations, this approach attempts to determine another decomposition that complies with the given architecture by moving classes around. Seng et al. (2006) have continued their work by searching for a list of refactorings, which deal with the placement of methods and attributes and inheritance hierarchy.

O'Keeffe and Ó Cinnéide (2004) have developed a tool for improving a design with respect to a conflicting set of goals. The tool restructures a class hierarchy and moves methods within it in order to minimize method rejection, eliminate code duplication and ensure superclasses are abstract when appropriate. Contrary to most other approaches, this tool uses simulated annealing. O'Keeffe and Ó Cinnéide (2006; 2008) have continued their research by constructing a tool for refactoring object-oriented programs to conform more closely to a given design quality model. This tool can be configured to operate using various subsets of its available automated refactorings, various search techniques, and various evaluation functions based on combinations of established metrics.

Mancoridis et al. (1998) have created the Bunch tool for automatic modularization. Bunch uses HC and GA to aid its clustering algorithms. A hierarchical view of the system organization is created based on the components and relationships that exist in the source code. The system modules and the module-level relationships are represented as a module dependency graph (MDG). The goal of the software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity while maximizing intra-connectivity. In our work we have

Di Penta et al. (2005) build on these results and present a software renovation framework (SRF) which covers several aspects of software renovation, such as removing unused objects and code clones, and refactoring existing libraries into smaller ones. Refactoring has been implemented in the SRF using a hybrid approach based on hierarchical clustering, GAs and hill climbing, also taking into account the developer's feedback. Most of the SRF activities deal with analyzing dependencies among software artifacts, which can be represented with a dependency graph.

Most of the approaches discussed above are different from ours in terms of the level of detail: we are especially interested to shape the overall architecture genetically, while the works discussed above consider the problem of improving an existing architecture in terms of fairly fine-grained mechanisms.

7. Conclusions

We have presented a method for using genetic algorithms for producing software architectures, given a certain representation of functional and a quality requirements. We have focused on three basic quality attributes: modifiability, efficiency and simplicity. The approach is evaluated with an empirical study, where the produced architectures were given for evaluation to experts alongside with student solutions for the same design problem.

The empirical study suggests that the current technique is at the level of an undergraduate student. In addition to the automation aspect, major strengths of the presented approach are the versatility and options for expansion. Theoretically, an unlimited amount of patterns can be used in the solution library, while a human designer typically considers only a fairly limited set of standard solutions. The genetic synthesis is also not tied to prejudices, and is able to produce fresh, unbiased solutions that a human architect might not think of.

The main challenge in this approach is the specification of the fitness function. As it turned out in the experiment, even experts can disagree on what is a good architecture. Obviously, the fitness function can only approximate the idea of architectural quality. Also, tuning the parameters (fitness weights and mutation probabilities) is nontrivial and may require calibration for a particular type of a system.

Our future research topics focus on the boosting of the simulated evolution e.g. by using more specialized crossover where parents are selected in a particular way, on the development of a genetic architecting tool environment, and on the studies of possible alternative ways to measure architectural quality.

References

Amoui M, Mirarab S, Ansari S, Lucas C (2006) A GA approach to design evolution using design pattern transformation, International Journal of Information Technology and Intelligent Computing 1: 235-245.

Bass L, Clements P, Kazman R (1998) Software Architecture in Practice, Addison-Wesley.

Bowman M, Brian, LC, Labiche Y (2007) Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms, Technical report SCE-07-02, Carleton University.

Brown WJ, Malveau C, McCormick HW, Mowbray TJ (1998) Antipatterns – Refactoring Software, Architectures, and Projects in Crisis. Wiley.

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

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6): 476-492.

Clarke J, Dolado JJ, Harman M, Hierons R, Jones MB, Lumkin M, Mitchell B, Mancoridis S, Rees K, Roper M, Shepperd M (2003) Reformulating software engineering as a search problem, IEE Proceedings – Software 150 (3): 161-175.

Clements P, Kazman R, Klein M (2002) Evaluating Software Architectures, Addison-Wesley.

Di Penta M, Neteler M, Antoniol G, Merlo E (2005) A language-independent software renovation framework, The Journal of Systems and Software 77: 225-240.

Diaz-Pace A, Kim H, Bass L, Bianco P, Bachmann F (2008) Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In: S. Becker, F. Plasil, and R. Reussner, (eds.) Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures, Karlsruhe, Germany, LNCS 5281, Springer: 171-188.

Gamma E, Helm R, Johnson R, Vlissides, J (1995) Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley.

Harman M, Mansouri SA, Zhang Y (2009) Search based software engineering: a comprehensive review of trends, techniques and applications. Technical report TR-09-03, Kings College, London.

IEEE (2000) IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Standard 1471-2000.

Kim S, Kim DK, Lua L, Park S (2009) Quality-driven architecture development using architectural tactics, Journal of Systems and Software 82 (8): 1211-1231.

Kruchten P (1995) Architectural blueprints – the "4+1" view model of software architecture, IEEE Software 12 (6): 42-50.

Mancoridis S, Mitchell BS, Rorres C, Chen YF, Gansner ER (1998) Using automatic clustering to produce high-level system organizations of source code. In: Proc. of the International Workshop on Program Comprehension (IWPC'98): 45-53.

Matinlassi M, Niemelä E (2002) Quality-driven architecture design method. In: Proc. International Conference of Software and Systems Engineering and their Applications (ICSSEA 2002): 8 p.

Michalewicz Z (1992) Genetic Algorithms + Data Structures = Evolutionary Programs, Springer.

Mitchell M (1996) An Introduction to Genetic Algorithms, MIT Press.

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

O'Keeffe M, Ó Cinnéide M (2006) Search-based software maintenance, In: Proceedings of CSMR 2006: 249-260.

O'Keeffe M, Ó Cinnéide M (2008) Search-based refactoring for software maintenance, Journal of Systems and Software 81 (4): 502-516.

Räihä O (2009) An updated survey on search-based software design, University of Tampere, Department of Computer Sciences, Report D-2009-5.

Räihä O, Koskimies K, Mäkinen E (2008a) Genetic synthesis of software architecture, In: Proc. of the 7th International Conference on Simulated Evolution and Learning (SEAL'08), Springer LNCS 5361: 565-574.

Räihä O, Koskimies K, Mäkinen E, Systä T (2008b) Pattern-based genetic model refinements in MDA, Nordic Journal of Computing 14(4): 338-355.

Räihä O, Koskimies K, Mäkinen E (2009) Scenario-based genetic synthesis of software architecture, In: Proc. of the 4th International Conference on Software Engineering Advances (ICSEA'09): 437-445.

Selonen P, Koskimies K, Systä T: Generating structured implementation schemes from UML sequence diagrams. Proc. TOOLS USA, IEEE CS, Santa Barbara, July 2001, 317-328.

Seng O, Bauyer M, Biehl M, Pache G (2005) Search-based improvement of subsystem decomposition, In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO'05) : 1045-1051.

Seng O, Stammel J, Burkhart D (2006) Search-based determination of refactorings for improving the class structure of object-oriented systems, In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO'06): 1909-1916.

Shaw M, Garlan D (1996) Software Architecture – Perspectives on an Emerging Discipline. Prentice Hall.

Simons CL, Parmee IC (2007a) Single and multi-objective genetic operators in object-oriented conceptual software design In: Proc. of the Genetic and Evolutionary Computation Conference (GECCO'07): 1957-1958.

Simons CL, Parmee IC (2007b) A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design, Engineering Optimization 39 (5): 631-648.

S3 (2008) Proceedings of the 2008 Workshop on Self-Sustaining Systems (S3'2008, Potsdam, Germany, May 15-16, 2008), Lecture Notes in Computer Science LNCS5146, Springer-Verlag.