CODE GENERATION IN JAVA
A modular approach for better cohesion

Bachelor Degree Project in Informatics

30 ECTS
Spring term 2015

Emil Forslund

Supervisor: Henrik Gustavsson
Assistant supervisor: Per-Åke Minborg
Examiner: Sanny Syberfeldt
Abstract

This project examines how the quality of a code generator used in an Object-Relational Mapping (ORM) framework can be improved in terms of maintainability, testability and reusability by changing the design from a top-down perspective to a bottom up. The resulting generator is tested in a case study to verify that the new design is more cohesive and less coupled than an existing code generator.

Keywords: Java, Code generation, Cohesion, Maintainability, Testability, Modularization, Automated Programming, Object Relational Mapping, ORM
Table of Contents

1 Introduction ......................................................................................................................... 4

2 Background .......................................................................................................................... 5
  2.1 Object-Relational Mapping (ORM) .................................................................................. 5
  2.1.1 Database connection and JDBC .................................................................................. 6
  2.1.2 Code Generation in ORM .......................................................................................... 6
  2.2 Measuring code generator quality ................................................................................... 6
  2.3 Chidamber & Kemerer (CK) quality metrics ................................................................... 7
    2.3.1 Weighted Methods per Class (WMC) ....................................................................... 8
    2.3.2 Depth of Inheritance Tree (DIT) .............................................................................. 8
    2.3.3 Number of Children (NOC) ..................................................................................... 8
    2.3.4 Coupling between object classes (CBO) ................................................................. 8
    2.3.5 Response For a Class (RFC) ..................................................................................... 9
    2.3.6 Lack of Cohesion Of Methods (LCOM) ................................................................... 9
  2.4 Beck & Diehl (BD) modularization metrics ..................................................................... 9
  2.5 Code Generation in an example ORM framework ........................................................... 10

3 Problem .................................................................................................................................. 12
  3.1 Method ................................................................................................................................ 12
    3.1.1 Method discussion ..................................................................................................... 13

4 Implementation ...................................................................................................................... 15
  4.1 Progression ....................................................................................................................... 16
    4.1.1 Interfaces model ........................................................................................................ 17
    4.1.2 Controls were added .................................................................................................. 19
    4.1.3 Keep track of dependencies .................................................................................... 19
    4.1.4 Traits in models ........................................................................................................... 19
    4.1.5 Transforms instead of views .................................................................................... 20
  4.2 Pilot Study .......................................................................................................................... 21

5 Evaluation ............................................................................................................................ 23
  5.1 The Study .......................................................................................................................... 23
    5.1.1 Verification ................................................................................................................. 26
  5.2 Analysis ............................................................................................................................... 27
  5.3 Conclusions ....................................................................................................................... 31

6 Concluding Remarks ............................................................................................................. 34
  6.1 Summary ............................................................................................................................. 34
  6.2 Discussion ........................................................................................................................... 34
  6.3 Future Work ....................................................................................................................... 35

References ................................................................................................................................... 36
  1 Old version (Proprietary) ................................................................................................... 40
  2 New version (Open Source) ............................................................................................... 43
1 Introduction

Some Object-Relational Mapping (ORM) frameworks use code generation to build custom components for a particular database. The code generator creates code-stubs and helper classes for each table to minimize the amount of manual work for the programmer. Some generators are designed in a way that result in bad cohesion and high coupling. Singh, Bhattacherje & Bhattacharjee (2012) show that these are properties that can make the system hard to maintain, test and reuse.

In this project a modular code generator was designed for high cohesion and low coupling. The goal was to create a tool that is more maintainable, testable and reusable than an existing code generator used in ORM today. The resulting code generator can be used to generate fully functional java code from a model centered description of the final program. This design also makes it possible to perform automated tasks on the model before the code is created.

The code generator performed well in comparison with other open source projects measured by Lincke, Lundberg & Löwe (2008), but when integrated into the existing ORM software, only testability and reusability seem to have improved. At the same time the measured maintainability appear to have decreased in the modular approach. The measurements was done using the metric suite defined by Chidamber & Kemerer (1994) and the quality of the proposed design was therefore only analyzed theoretically.

This report is divided into six chapters. This first (1 Introduction) describes what the project is about and how the document is structured. The second chapter (2 Background) summarizes earlier work in the area of code generation and quality measurement. It also describes how a code generator for a standard ORM could be designed. This description is then used in the next chapter (3 Problem) to formulate a hypothesis about how the quality can be improved. This is also where the scientific method used in this project is described. In (4 Implementation) the actual code generator is created, and in the two final chapters (5 Evaluation) and (6 Concluding remarks) the results and measurements of the project are presented and discussed.
2 Background

2.1 Object-Relational Mapping (ORM)

An Object-Relational Mapping (ORM) framework is a tool that makes it possible for the developer of a database application to work with the database in an object oriented manner. The framework instantiates objects based on the structure of the database so that tables become containers, rows become entities and columns become member fields. Foreign keys between tables are used to map objects to each other in a graph. The graph can then be traversed using graph search algorithms to access the data. Using this technique, the developer of an application can get most of the advantages of graph databases without changing the storage structure (Fahl & Risch, 1997).

```java
// Find all purchases on more than 100 $ made by a
// person that has shopped for more than 1000 $ in
// the same two week span.
static Transactions findSuspicious() {
    return Transactions.where(t ->
        t.getItem().getCost() > 100 &&
        t.getCustomer().findTransactions(t2 ->
            Period.between(
                t.getTimestamp(),
                t2.getTimestamp()
            ).getDays() <= 7
        ).sum(t2 -> t2.getItem().getCost() > 1000)
    );
}
```

**Figure 1:** A simple example how a method using an ORM to query a database could look.

In Figure 1, a simple query for suspicious transactions in a banking application is illustrated. Since the data is organized in a graph, parameters may involve not only local column data in the accessed table but also referenced data in other tables as well as complex operations on whole columns like the sum-operation.

There are many advantages with this kind of framework. An object oriented database API can blend seamlessly into the client implementation while any SQL transactions are managed in the background (Cheung, Solar-Lezama & Madden, 2013).

To prevent loss of efficiency, some frameworks use caching techniques to determine which objects to keep in the primary memory and which to keep on a secondary storage device. Other frameworks use lazy loading to not only load a required row into memory but also all rows that can be accessed from that row (Alhajj & Elnagar, 1999), (van Zyl et al. 2009). That makes it possible to perform chained queries in-memory once the first row has been loaded.
2.1.1 Database connection and JDBC

An ORM framework can do more than map the database relations to objects. It is also responsible for keeping the database updated on the state of the entities. If the data contained in an entity is updated in the graph, the database needs to be updated as well.

Java has a framework for working with databases called Java Database Connectivity (JDBC). It acts as a middle layer between an application and the Database Management System (DBMS). As long as the appropriate drivers for the specific DBMS is installed, the application can work with any kind of database. The JDBC API is demonstrated in a paper by Dietrich, Urban & Kyriakides (2002).

2.1.2 Code Generation in ORM

To make the API as tailored for the end user as possible and reduce the amount of manual labor for the programmer, a code generator is often used to generate code stubs for each table in the database as well as helper classes for traversing the database in an efficient way. When developing models to represent a specific domain it is often necessary to write code that is solely dependent on the structure of that domain. The framework analyzes the domain and then generates classes, methods and fields to represent the properties of that particular domain (Kulkarni & Reddy, 2008). Instead of writing that code manually, which is both time consuming and increases the risk of errors, the code can be generated. The time-saving aspect is seen in Zhang, Chung & Chang (2004) where a code generator is built to automate common tasks in web development. The ability to produce less bugs is shown by Mathiske, Simon & Ungar (2006) in a study where generated code is three-way compared to the system specification and manually written code.

2.2 Measuring code generator quality

To measure the quality of a code generator you must first decide which metrics you use. Cavano & McCall (1978) propose the following eleven quality factors when developing computer software: Correctness, Reliability, Efficiency, Integrity, Usability, Maintainability, Testability, Flexibility, Portability, Reusability and Interoperability. Each quality factor defines properties that are desirable in a system, but lacks quantifiable metrics.

Correctness: A code generator for an ORM framework must generate valid, compilable code to be correct. This can be tested by compiling the generated code and testing it on an example database.

Reliability: Independently of how the database is structured, the code generator must be able to generate a valid API. Using JDBC, the underlying architecture of the database can be concealed from the generator. To further increase reliability, different ways of using a database manager must be tested to ensure that the generator still creates valid code based on that structure.

Efficiency is less important in a code generator as it only executes when the structure of the database changes. It must still be able to finish in a reasonable amount of time and with a reasonable amount of memory at its disposal.

Integrity: A code generator can only generate code for a database that it has at least reading rights to. Apart from that, integrity can be assured since the system does not have to share the required data beyond storing the generated code on a local device.
Usability: Setting up the models used by the code generator could be a risk to usability if the process is too complex, but it is not as important as it would be in a user application.

Maintainability: There must be a close correlation between where the symptoms of a bug appears and where the error is located in the generator. For good maintainability, the generator should be structured in such a way that it resembles the final output rather than the expected input.

Testability is very important since it is required to verify other quality factors. A code generator with good testability is expressed in small modules that can be tested independently from each other.

Flexibility: If external API:s used by the generator are expanded or if new functionality becomes available, the system must be adaptable to the changes.

Portability: The generator should be designed in such a way that it can be implemented in different languages on different platforms.

Reusability: As much as possible of the code should be reusable in other applications. This requires low coupling between packages within the generator and high cohesion within components.

Interoperability: The generator should have a simple, well-documented API for it to be easily integrated into other applications.

In this project, maintainability and testability is in focus. Code generators differ from many other applications in one aspect. Since it is a program designed to write code, the code written by it can never be more sophisticated than the definition of the generator. Varadraj Gurupur and Urcun Tanik describe this phenomenon in the following way:

“Software development has been a process where lot of time and money has to be spent not only on the development process but also on a never-ending process of software maintenance. Sometimes this development process could face obstacles such as the creeping requirements problem where the requirements may change while the software development is still in process. Many software development processes may end in failure owing to the moving target problem due to the rapid change in the domain knowledge.”

Gurupur & Tanik, 2006, p. 786

This means that the generator will have to be updated as new design techniques, programming languages or API versions are made available. Otherwise, a ten year old generator will generate ten year old code, which might not be optimal.

Testability is also a priority aspect since a bug in the generator might result in erroneous code being created in many systems using it. The generator must therefore be built with testability in mind so to minimize the risk of errors.

2.3 Chidamber & Kemerer (CK) quality metrics

The eleven quality factors specified by Cavano & McCall only describe how quality in software can be categorized, not how to measure it. Chidamber & Kemerer (1994) propose a suite of six quality metrics to use for measuring the quality. The metrics are Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC),
Coupling between object classes (CBO), Response For a Class (UFC) and Lack Of Cohesion of Methods (LCOM).

2.3.1 Weighted Methods per Class (WMC)
This metric represents the sum of the static complexity of each method in a class. A high value indicates a high level of complexity within the class, which could indicate low maintainability. Which complexity measure to use is not defined by CK but left as an implementation decision.

A popular measurement of code complexity is the Cyclomatic Complexity Number (CCN) introduced by McCabe (1976). It describes how many paths through the code there are and can give a hint to how complex the system is, independent of the size. CCN is however designed for procedural systems and is not that suitable for object oriented ones, but could be used as a part of the WMC measure of method complexity as shown by Li & Henry (1993).

What exactly should be counted as a method is not covered by the original CK metric as remarked by Churcher et al. (1995). They propose that the definition should be expanded based on which language is measured to account for language-dependent concept such as operator-overloading or inherited super-methods.

Depth of Inheritance Tree (DIT)
This is the depth of the inheritance tree of a particular class, in other words, how many potential ancestors that exist. If the tree is tall, that could indicate a complex behavior of overridden methods and behavior which could lead to low maintainability (Poels & Dedene, 2001). A high value might not necessarily be a bad sign, however. Inheritance can also be a sign of a high degree of reusability in the ancestral component. In an industrial case study Ping, Systa & Muller (2002) shows that DIT is insignificant as a quality aspect for predicting fault-proneness, but Daly et al. (1996) get a different result in an experiment on multiple software developers. They found that an inheritance depth of 3 is optimal for maintainability, but that more than 5 can be worse than a single level. The experiment by Poels & Dedene suggested an inheritance depth of two (one parent and a number of children) as the optimal.

2.3.2 Number of Children (NOC)
NOC is the number of immediate subclasses to a particular component. A high value might both be a sign of good reusability as shown by Goel & Bhatia (2013) but also a sign of bad abstraction which leads to misuse of sub-classing. It can also be a sign of fault-proneness because of the high influence the class has on the overall design as indicated by Ping, Systa & Muller (2002).

2.3.3 Coupling between object classes (CBO)
There is a connection between high coupling between components and low reusability, maintainability and testability. CBO is the number of couplings a particular component has to other classes. This is only the static coupling, however. Arisholm, Briand & Foyen (2004) show that dynamic coupling is also an important factor for maintainability and that it is not redundant with static coupling measurement as covered in CBO. If you need to know how surrounding components work to understand a class, that will affect the maintainability of the class, as shown by Singh, Bhattacherjee & Bhattacharjee (2012).
2.3.4 Response For a Class (RFC)
The RFC metric is a value for how many methods in a class may respond to a message from outside the class. It can be seen as a measure of in how many ways the class can be communicated with. Only the first level of response is measured in the case of recursive methods. Bruntink & van Deursen (2004) indicates that a high RFC value might indicate low testability of the component.

2.3.5 Lack of Cohesion Of Methods (LCOM)
LCOM is a measure of how similar methods within a class are. This is done by comparing which member fields are used by the methods. Two methods that use the same exact fields are considered similar, which will increase the LCOM value. Two methods that have no common fields are not considered similar and will therefore not contribute to a higher LCOM value. As with all CK metrics, a low LCOM value is better than a greater.

The CK metrics are evaluated as a means of identifying early quality indicators in an empiric study by Basili & Melo (1996) that indicates that both structural complexity and coupling are useful for fault prediction. The study supports five of the six metrics, with an exception of LCOM. Dubey & Rana (2010) summarizes how the CK metrics can be used to improve maintainability in a system. They calculate a value of the overall maintainability by combining all the CK metrics and normalizing them to a value between 0 and 1 where 0 is bad maintainability and 1 if the best maintainability. In a study by Khalid, Zehra & Arif (2010), some of the CK metrics were evaluated as a means of measuring code testability. The study indicates that high values in DIT, NOC and CBO are in fact indicators of bad testability. The study did not compare WMC, RFC or LCOM.

2.4 Beck & Diehl (BD) modularization metrics
Another way to measure code complexity in object oriented systems is to focus on how they approach modularization. Beck & Diehl (2011) did a study on 16 open source projects to see how different coupling concepts affect the overall congruence of the system. They called their metrics compared structural dependencies, fan-out similarity, evolutionary coupling, code ownership, code clones, and semantic similarity. The results suggest that coupling can't be determined as a one-dimensional value but rather a vector of different properties.

Structural dependencies is very similar to CBO and RFC with the difference that BD distinguished between different kinds of dependencies by separating the metric into three values; Inheritance, Aggregations and Usages. The Fan-out similarity metric has also about the same purpose as CBO, but can also be compared with RFC. It measures if there are any other components with the same or a similar set of dependencies, which could indicate that the two components have an indirect coupling. This value is also separated into three categories. Evolutionary coupling is a measure of how often two components are changed at the same time during the evolution of the system. BD argue that this can be a sign of another indirect coupling. Code Ownership is a metric for any indirect coupling caused when two components share the same authors. Code Clones is a way to measure coupling by looking at how similar two components are without looking at the direct dependencies. Semantic

1 The paper calls the third metric CBC (Coupling Between Classes) instead of CBO (Coupling Between Objects) as in the original CK paper. The definition of the metrics is the same, however.
similarity is similar but it measures how often the same vocabulary is used in the components.

BD use a number of different tools for measuring the different aspects. Most of the aspects in BD are designed to measure coupling between components while CK have a broader, if yet not as detailed, perspective.

2.5 Code Generation in an example ORM framework

In the following example, a Top-Down Code Generator (TDCG) is described. TDCG is part of a large ORM that enables programmers to work with databases using object-oriented techniques. TDCG traverses all the tables in the database and creates Builders for each table. There are different Builders for the different types of components that TDCG can generate. All Builders inherit from the “Builder” super class where common code used in all Builders is located, as shown in Figure 2. This involves functions for writing method declarations, opening and closing code blocks, indenting and similar operations. Each builder also has a “build”-method that generates code from the information supplied. The build()-method of the EntityListBuilder class is displayed in Figure 3 as an example of this.

![Diagram](image)

**Figure 2**: A simple example of how a code generator for an ORM framework designed in a top-down fashion could be composed.

The class structure in Figure 2 is meant to illustrate how components are coupled. For most real applications, the class diagram contains many more Builders and a much deeper inheritance tree.
Figure 3: The build()-method of EntityListBuilder.
3 Problem

In a top-down approach used by some code generators today like the Speedment (2015) software, the system is designed so that most of the functionality is located in the top of the inheritance tree and details are implemented at a lower level. Some inheritance can increase the maintainability of a system as shown by Daly et al. (1996) and Poels & Dedene (2001), but as the depth of the inheritance tree increases, the quality worsens. This leads to bad scalability as all classes must be updated if the super class is changed. Maintainability is an important quality aspect of a code generator since it will have to be updated frequently to keep up with current research, new API:s and available tools (Gurupur & Tanik, 2006). You could assume that the super class will have to be changed from time to time if you want to add more functionality. Khomh, Di Penta & Guéhéneuc (2009) presents a case study that supports the theory that large classes with many responsibilities are changed more often.

Furthermore, the top-down design prevents reusability of components as almost every class must inherit from one that has a very broad purpose (Goel & Bhatia, 2013). This creates a dependency on coupling that has been shown to increase the complexity of a system (Beck & Diehl, 2011). It is also difficult to implement a broader range of functionality, like support for other programming languages, as the language specific structure is implemented in the ancestor. Bruntink & van Deursen (2004) and Khalid, Zehra & Arif (2010) show that components with broad responsibilities are also more difficult to test.

A third issue is that common procedures are hard to automate. The template of the class structure is traversed only one time, which makes it hard to attach any additional functionality. Each time new options like automatic final parameters, automatic getters and setters or automatic documentation stubs are to be implemented, for existing classes to make use of that functionality, they will have to be updated to add support for that option. This is a sign of low modifiability.

The top-down approach also affects other quality aspects like efficiency, interoperability and flexibility, but these are not covered in this project.

Kulkarni & Reddy (2008) propose an abstraction for reusable model-driven-development components. In this modular approach for designing a code generator, a higher grade of reusability was achieved. This project will take inspiration from that design to see if it can also result in higher maintainability and testability.

This project will focus on the following problem. A code generator designed with the top-down approach is tightly coupled and has bad cohesion, which leads to low modifiability, testability and reusability (Basili & Melo, 1996).

The hypothesis is that a modularized code generation framework with functionality separated into models and views can reach better cohesion and less coupling.

3.1 Method

Many studies on design practices and the effects on software quality in computer science use formal experiments, case studies and/or surveys as a scientific method. These are described by Wohlin et al. (2012).
In a formal experiment, a sample application is developed in a lab environment to test if the hypothesis can be confirmed. The controlled form of the research make it possible to be very concrete in the measurement, but there is always a risk that the environment differs from reality.

In a case study, an application is developed and then tested in a “real” environment. In the case of code generation, that could mean integrating the tool in a commercial ORM software to see if it performs as well as the existing tool.

In a survey, multiple users are asked what they think of the software through interviews or written forms. This is good for measuring soft qualities like usability, but requires that the participants have a chance to form their own opinion on the software.

In this project, a code generator is implemented and integrated into an industry ORM software available as a case study. The proposed code generator as well as the existing one is then tested using the CK metrics to see how the proposed changes affect the cohesion and coupling of the whole system. Minimal changes will be done to the surrounding modules to integrate the new component. This study is inspired by the study on quality improvement by Shrivastava & Shrivastava (2008). To make sure no functionality is lost, both solutions will also be used on a real database to verify that the code is generated correctly.

3.1.1 Method discussion

For a project about creating a code generator, any of the scientific methods mentioned could work. If the projects goal was oriented around creating a good API, usability tests would probably be necessary to make sure that it covers everything the users would want to use it for. In a similar project, Altiparmak et al. (2013) use a survey to verify the usability and functionality of a graphical code generator.

If the project aimed at a broader comparison between different design techniques and their effect on software quality, a formal experiment would probably be more suitable. In a controlled environment, test cases can be produced to provoke uncommon but still significant behaviors in the system. This can be useful when developing security critical systems where use cases are not possible for different reasons. Bank applications with stringent transactions is an example of an environment that you would probably want to test in a safe and controlled environment. Briand et al. (1999) describe how experiments and case studies can be performed in software projects.

This project has maintainability and testability as a focus and not usability. A survey is therefore not optimal in this case. A formal experiment could be used to test if a code generator developed using a modularized design performs better than one designed using top-down, but the risk is significant that the results of the project are irrelevant if neither of the compared designs match one used in the industry.

In a case study, however, a new code generator could be designed and then compared to a code generator used by a real company in a commercial ORM. This would require permission from the company in question to participate in the study. Measures must also be taken to assure that no sensitive information is exposed in the generated code. The positive side of using a case study instead of an experiment is that it shows that the solution works
not only in a lab environment under the best conditions but also in an uncontrolled environment.

A potential risk when choosing only a few quality parameters is that optimizations on a few might be at the expense of the others. It is possible that one solution might be more maintainable and reusable than another, but so resource demanding or time consuming to implement that it is not worth the final cost. Some trade offs will always have to be made and in those cases it is important to know what the options are. In a code generator, efficiency is one quality aspect that might be “cheaper” to sacrifice than others. Compared to other programs, a code generator is executed very scarcely. Both CPU and memory requirements must be within reasonable limits, but it is not paramount to have the most efficient code generator available. Usability is another aspect that is less important in this project. Since this code generator is a middle-layer component that is called from within a program, the user never has to interact directly with it. Even so, no quality aspect can be completely ignored since that might make the entire system unsuitable for the end user. Maintainability and testability are the focus of this study, but other quality aspects will have to be within reasonable limits.

Measuring cohesion and coupling in a software system is not easy. There are many different metric suites to choose from. In this project, the six Chidamber & Kemerer (1994) metrics will be used. CK is not the most advanced suite of metrics, but it has been around for several decades and has been used frequently during this time. There exists a lot of documentation as well as discussions on when the values are suitable and when they are not. They cover coupling (CBO and RFC), cohesion (LCOM) as well as complexity (DIT, NOC and WMC). The CK metrics can also be calculated automatically using a number of tools that are available as open source. There is always a risk however that the tool interprets the metrics differently and therefore returns different output for the same input. Lincke, Lundberg & Löwe (2008) compare different open source measurement tools for java and c++ on the CK metrics and their results show that the outputs can vary greatly depending on the presumptions made by the tool in question.

The CK metrics could be calculated manually using the method originally proposed by the authors. This would give a more certain result, but would also take far too much time in a case study on a system with hundreds of components.

Another suite of metrics that could be used are the Beck & Diehl (BD) modularization metrics proposed by Beck & Diehl (2011). BD is closer associated with coupling and cohesion than CK and could therefore be more suitable in this study. The methods of calculating BD is however quite complex and might not be possible in the time frame of this project. BD is also much younger than CK and there are not as many existing studies out there to compare the results with.
4 Implementation

In the abstraction proposed by Kulkarni & Reddy (2008), a code generator should consist of multiple reusable “building blocks” with very specific tasks. Some of these blocks are “Models” that hold the state of a particular language concept and others are “Transformations” that transform a model into another representation. Kulkarni & Reddy propose both model-to-model and model-to-code transformations. The latter is called “views” in this project.

For each language concept (class, interface, method, field, etc) there is a model class. The sole responsibility of the model is to hold the state of that concept. The models are designed to be as general as possible so that they do not limit future functionality requirements. Models are hierarchical. A “class” model contains multiple “method” models, and both “class” and “method” might contain “field” models. To keep the coupling of the system low, models are only dependent on their direct children in the hierarchy.

For each model block there is a corresponding view block. The view has a direct dependency on the model, but there are no dependencies between views or from a model to a view. In this way, the number of dependencies between components are maintained at a low level to keep the coupling under control.

The generation process is governed by a “Generator” block. The generator has methods for transforming models of an arbitrary type into code, in one step or in multiple. The generator does not have any dependencies to either models or transforms. Instead it uses “TransformFactory” blocks to instantiate an appropriate view for a given model. These factories can be plugged into the generator using the factory method pattern described by Freeman & Freeman (2004, p. 134). A reference to the generator is passed to each transform together with the model being transformed so that tasks can be broken up into sub-tasks that can be delegated back to the generator. This is an example of the divide-and-conquer tactic (Cormen et al. 2009, p. 65). The tree is going through this circular process (generator → installer → transform → generator) as shown in Figure 4 until the entire model has been transformed into text.

![Figure 4: A simplified UML-diagram of how the basic components of the code generator interact.](image)

Some transforms might not go directly from model to text. The code generator also supports model-to-model transformation. If so, multiple transforms might have to be linked into a
graph to create the rendering pipeline. This is done by the generator using a component called “TransformBridge”. It concatenates multiple transforms into one chain that can convert an inputted model into the final representation (text) by delegating each step to the correct transform.

4.1 Progression

The first draft of the code generator was designed to model the program from an expression level and up. Operators, statements and blocks of code made up the foundation and more complex concepts as methods and classes were built using these blocks. Every model inherited from the “Model” super class. This opened up many opportunities when it came to performing calculations like automated tasks on the model, but soon resulted in a very large amount of components. It became increasingly difficult to make changes to the generation process since the changes had to be implemented in several places. It was also difficult to describe a program flow to generate since the design had to cover everything from which classes should reside in which files to which order the operands of an if-statement should be in. Since one of the primary goals of the design was to achieve a high level of maintainability, that design was discarded.

In the second iteration, the design began at a component level with classes, interfaces, enumerations and annotations. Two major things made this design differ from the previous one. First of, different language concepts do not have a common ancestor. It does not matter if you put a “Method” or a “Field” class into the generator, as long as there is a corresponding view, it will be generated. This was really important for reducing coupling, as shown in the “Favor composition over inheritance”-philosophy (Freeman & Freeman, 2004, p. 23). Another major design change was that the models were changed into interfaces instead of classes. By using the factory-pattern explained by Freeman & Freeman, but modified to fit into the interface of the model, the implementation of a particular language concern can be changed without affecting other components. This makes it possible to change the way the model works without affecting the code generation. The implementation in Figure 5 was inspired by the enum singleton pattern described by Bloch (2008, p. 18).

```java
public interface File {
    enum Factory {
        INST;
        private Supplier<File> prototype = () -> new FileImpl(null);
    }
    static File of(String name) {
        return Factory.INST.prototype.get().setName(name);
    }
    static void setSupplier(Supplier<File> supplier) {
        Factory.INST.prototype = supplier;
    }
}
```

**Figure 5:** The private enum singleton pattern in use in the "File" model interface.

There are many advantages that come with this beyond reducing the coupling. It makes it easier to reuse existing components from older systems as models and also makes it possible to construct “patterns” for systems without knowing exactly how each building block is
working. The downside is that each interface will be dependent on one default implementation of that interface.

4.1.1 Interfaces model

One of the advantages of having a single super class for each model that was sacrificed was the ability to place similarities between different concepts in the super class. Interfaces, classes and enums all look pretty much alike in java so it felt natural that these concepts should share a common ancestor. For this purpose, the abstract component “ClassOrInterface” was implemented. The name is a bit misleading since it is also the super type of enums, but it was chosen as it is the same title that the java documentation by Gosling et al. (2015, p. 700) uses. This design, however, led to some challenges when creating the views for these concepts. Both classes and fields can have member fields and methods, but they look very different depending on the context. A method in a class should have a body if it does not have the “abstract” keyword, but in an interface, it should never have a body except if it has the “static” keyword, and “abstract” should not be written out at all.

The first approach to this problem was to create more model components. A class contained “Field” and “Method” instances exactly as before, but an interface would contain “InterfaceField” and “InterfaceMethod” instead. The classes were almost identical, but since they had different “View” components, they could be rendered differently. The advantage of this solution was that it was really easy to test the solution. It required more tests since you had more classes, but you could easily isolate the responsibilities of each component since you knew if it was an interface or a class that was being tested. The downside was that a large portion of the rendering code was identical between the “MethodView” and the “InterfaceMethodView”, which goes against the “Don’t Repeat Yourself”-principle by Hunt & Thomas, (2000, p. 27) and increases the risk of errors. An UML-diagram of the “Method” component in this first approach is shown in Figure 6.

![UML-diagram](image)

**Figure 6**: The first approach where interfaces and classes used separate models and views.

Two alternative strategies were tried to solve this problem. The first was to add a render stack to the generator interface. This idea was inspired by the context stack used in many low level processor architectures (Johnson 2009). Whenever a view is initialized, the corresponding model is first pushed to the stack. When the render is completed, the model is pulled from the stack. If a view delegated more rendering tasks to the generator, the stack will grow, making it possible for any child processed to know in which context they are being generated. Using this approach, no additional interface components were required. An
interface model could contain ordinary methods and fields, and it was up to the views for those components to render proper code depending on their context. The advantage of this approach was that it required a lot less classes, which decreased the complexity of the generator design. The problem was that it put extra responsibilities on the view components. The example with the “Method” component can be seen again in Figure 7 but with the RenderStack in place.

The second alternative strategy attempted was to separate the view into “traits”. The concept of traits has been explained by Black & Scharli (2004). Parts of the rendering process can be implemented in separate interfaces and then called from the views that required that part. Figure 8 shows how the “FieldView” component looks with traits.

```
public class FieldView implements View<Field>, HasNameView<Field>, HasJavadocView<Field>, HasModifierView<Field>, HasTypeView<Field>, HasValueView<Field>, HasAnnotationView<Field> {
    @Override
    public Optional<String> render(Generator cg, Field model) {
        return Optional.of(
            renderJavadoc(cg, model) +
            renderAnnotations(cg, model) +
            renderModifiers(cg, model) +
            renderType(cg, model) +
            renderName(cg, model) +
            renderValue(cg, model)
        );
    }
}
```

**Figure 8:** Using traits, each part of the rendering process is delegated to a separate component. These "traits" can be shared between components.

This strategy drastically decreases the average complexity of the views. It does however add a large amount of extra components to the design. This is not completely bad, however. If every piece of javadoc is rendered using the JavadocView, which is only called from one single “HasJavadocView”-trait, then you can assume that is where the bug is if the javadoc is rendered incorrectly. This design results in a large amount of extra components, but the responsibility of a single component is minimal.
In the final solution, a mix of these techniques was used. The interface methods and fields are still being generated by separate views, but using traits, the amount of redundant code can be held to a minimum. The render stack was kept as an additional feature as future transforms might need that kind of information about the current render.

4.1.2 Controls were added
When the model components were finished and a first program hierarchy could be expressed, a new concern was that the model initialization process contained a lot of repetition. If you wanted to generate a standard Java bean with three member fields, you not only needed to add each field, you also needed to add setters and getters for each field. A setter required a field as an argument, so you ended up appending two methods and two fields for each member field in the component. The concern of “repeatable tasks” is raised by Hunt & Thomas (2000, p. 27) and is addressed in this project by adding a concept called “Controls” to the design. A control is a class that operates on a model and changes it before it enters the generator. In this case, a control that automatically adds a setter and a getter method for each member field was implemented. Additional controls like automatic documentation stubs were also added later on to automate other tasks.

4.1.3 Keep track of dependencies
In java as well as many other languages, you can get around importing dependencies manually by specifying the absolute path to each component whenever you use it. The most trivial way to do this in a generator would be to always write out the entire class name, but it does not result in very readable code. The way this code generator is designed, a view should not have to know anything about models further up or down the hierarchy. So how can a FileView know which components it should import? This is a tricky problem with many different solutions. The most simple one is to add an “Import” model type and let the user specify exactly which components should be imported and assume that all components are. If the user does not manually add each dependency, the generated code will not compile. This might work in most scenarios, but not all. Sometimes you will have name collisions when multiple classes share the same name. In those cases, you must specify the absolute path so that the compiler can know which component you are referring to. Therefore differentiating between absolute paths and short names are necessary.

The solution chosen for this design is a combination of the render stack and the controls specified earlier. An “AutoImports” control was implemented that traverses the program model before generation and appends any non-colliding dependencies to the file. Whenever a “Type” is rendered, it can use the render stack to access the file model and check if it has been imported and thereby is eligible for a short name or if it should be rendered using the absolute path. A downside with this solution is that it is hard to verify the correctness of the method. Since the outcome depends on the state of the render stack, the same view and model might get different results in different situations. The solution was chosen anyway since it required very little change to surrounding components and did not require any additional render passes which other solutions might require.

4.1.4 Traits in models
When implementing more advanced controls for automating tasks, a problem was that controls became coupled to every model that should be able to use them. The “AutoJavadoc”-control for example required knowledge of not only the “File” and “Class” model but also “Interface”, “Field”, “Method”, “Constructor” and so on. Every language concept that could
have a Javadoc-tag attached to it would have to be expressed in the AutoJavadoc-component. This was not a very good design. To solve this, the trait-pattern (Black & Scharli, 2004) from the view problem earlier was used. Each model was redesigned to use a combination of different traits. A file might have the “HasImports”, “HasName” and “HasClasses” traits and a class might have the “HasName”, “HasClasses” and “HasMethods” traits. By adding the “HasJavadoc”-trait to a model, it will be marked to implement the “getJavadoc”-method that AutoJavadoc requires. This pattern proved to work even better with the models than with views since the responsibility of a model is to hold a state while a view also needs to operate on that state. This made it possible to write very powerful controls that can traverse the model hierarchy recursively without being dependent on the model implementations. Figure 9 shows how the “ClassOrInterface” model looks when the different traits have been isolated to their own files.

```java
public interface ClassOrInterface<T extends ClassOrInterface<T>> extends Copyable<T>, Callable<T>, HasName<T>, HasJavadoc<T>, HasGenerics<T>, HasImplements<T>, HasClasses<T>, HasMethods<T>, HasFields<T>, HasAnnotationUsage<T>, HasModifiers<T>, HasInitializers<T> {}
```

Figure 9: An example of a model using traits.

4.1.5 Transforms instead of views

When the code generator had reached the state that it could be used in a real ORM, an interesting user behavior was noted. Since you very rarely have all the information ready to construct your program model in one step, a middle layer was required to transform the original data into a model. This process looked very much like the rendering that the code generator was doing, except that it translated classes of one type into code generator models. Views in this project can be seen as model-to-text-transformations as described by Kulkarni & Reddy (2008), but it seemed that model-to-model-transformations were also required. But if the views were changed to output data of a general type instead of strings, how does the generator find a correct way from the original form to the desired one?

The available transforms can be seen as paths in a graph were the different models make up the nodes. By traversing the graph, a model can be transformed from and to whatever form you want, assuming there is a path between the nodes. The graph is directed, which means that a particular path can only be walked in one direction. An overview of the graph search problem can be found at Wikipedia (2015). When a path has been found, all the transforms are concatenated into a single “BridgeTransform” that can transform from the original form to text. That transform is then used by the generator as before.

Figure 10: An "Entity"-class is generated from a SQL table.
Figure 10 shows how an SQL table is generated by delegating each column to the generator. Three different transforms use the column input to generate fields, setters and getters respectively for that data. Everything is then collected back into the class before delegating that back to the standard views.

4.2 Pilot Study

To try out the code generator it was used to generate a small “Hello World”-program. The model consists of a single class in a single file. The class has a method and a static field. A utility class will also be used to automatically generate documentation for the class. The model can be seen in Figure 11 and the resulting output in Figure 12.

The CK metrics defined by Chidamber & Kemerer (1994) of the pilot project were calculated using the CKJM-tool by Spinellis (2013). The jar-file used as input was a compiled version of the code in Figure 11 with all dependencies from the code generator included. Class files not used by the example (or within the generator) were not included. The results are shown in Table 1:

<table>
<thead>
<tr>
<th>Measure</th>
<th>WMC</th>
<th>DIT</th>
<th>NOC</th>
<th>CBO</th>
<th>RFC</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Maximum value</td>
<td>34</td>
<td>2</td>
<td>7</td>
<td>25</td>
<td>58</td>
<td>477</td>
</tr>
<tr>
<td>Average</td>
<td>6.04</td>
<td>1.01</td>
<td>0.12</td>
<td>3.99</td>
<td>12.29</td>
<td>28.09</td>
</tr>
<tr>
<td>Median</td>
<td>3.0</td>
<td>1.0</td>
<td>0.0</td>
<td>3.0</td>
<td>7.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Components measured</td>
<td>161</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: The CKJM results for the pilot study.

The code generated can compile and the CK metrics can be calculated. The comparison was limited to min, max, average and median values of each metric, but to get an exact measure you would have to compare individual components. Shrivastava & Shrivastava (2008) did a study where the same method was used continuously throughout the project to see if the maintainability increased, but the metrics used were different than CK. Lincke, Lundberg & Löwe (2008) used the CKJM tool on two different open source projects and measure min, max and average values. The measurements of this pilot are within the boundaries of their results.
Figure 11: The model of a simple "Hello, World"-program.

```java
package org.example;

/**
 * Write some documentation here.
 */

@javax.annotation.Generated
public class BasicExample {
    public final static String BASIC_MESSAGE = "Hello, world!";

    /**
     * This is a vary basic example of
     * the capabilities of the Code Generator.
     * @param params
     */
    public static void main(String[] params) {
        System.out.println(BASIC_MESSAGE);
    }
}
```

Figure 12: The resulting code from the program in Figure 11.
5 Evaluation

5.1 The Study

To evaluate the code generator, it was integrated into the commercial version of the Speedment ORM framework (Speedment, 2015). Speedment is an Object-Relational Mapping tool that utilizes code generation to create models from a SQL database. The current generator, in this project called “ACE”, uses a top-down approach with various “builders” capable of generating different types of components. All builders share a common ancestor and functionality can be shared using inheritance. A complete list of the builders is shown in Figure 13. The code generator developed as part of this project is called “CodeGen”.

![Figure 13: A list of all the builder components of the Speedment ACE software version 1.0.8.](image)

For this study, one of the builders was completely replaced by one written using the code generator created in this project. The “ListBaseBuilder” component was chosen as the code it generates both depends on and is a dependency to other generated components. It was also neither the most complex nor the most trivial of the builders to replace.

Most of the generation is done in the file “ListBaseBuilder.java”. Both the old version of the file and the new can be seen in Appendix A. Except rewriting the “ListBaseBuilder.java” file, some minor changes also had to be made to the rest of the Speedment software. First of, a tailored transform factory had to be made so that the new code generator could find the correct transform to use. This new source file called “SpeedmentTransformFactory.java” is shown in Figure 14. Secondly, the “InitBuilder” file had to be modified so that it calls the new code generator instead of the old one when a “ListBase” is generated. These changes can be seen in Figure 15, 16 and 17. The old code is shown in the comments. Lastly, the “pom.xml”-file of the project had to be appended with the new CodeGen library. This is shown in Figure 18.
```java
package com.speedment.ace.plugin.java.codegen;

import com.speedment.ace.plugin.java.builders.ListBaseBuilder;
import com.speedment.ace.plugin.java.codegen.base.DefaultTransformFactory;
import com.speedment.codegen.lang.models.File;
import com.speedment.core.db.model.Table;

/**
 * @author Emil Forslund
 */
public class SpeedmentTransformFactory extends DefaultTransformFactory {
    public SpeedmentTransformFactory() {
        super (SpeedmentTransformFactory.class.getSimpleName());
        install(Table.class, File.class, ListBaseBuilder.class);
    }
}
```

**Figure 14:** SpeedmentTransformFactory.java. A transform factory is used by the new code generator to associate transform components with the type they can generate from and to. In this case, a component that can transform from a "Table" into a "File" is installed.

```java
if (table.isIncluded()) {
    openBlock("Table (" + table.getName(), ") ");
    if (table.isTriggerlogTable()) {
        println("Skipping this table because it is the " + TriggerlogMap.
        progress(WORK_ITEMS_PER_TABLE);
    } else {
        // Set up CodeGen.
        final Generator gen = new JavaGenerator(
            new JavaTransformFactory(),
            new SpeedmentTransformFactory());

        // Create a 'File' that can be written to disc.
        gen.metaOn(table, File.class)
            .map(Meta::getResult)
            .forEach(this::make);

        // Structs
        TableBuilder structBaseGenerator = new StructBaseBuilder(table,
            make(structBaseGenerator);
        TableBuilder structGenerator = new StructBuilder(table, getLang,
            make(structGenerator);

        // Managers
        TableBuilder mgrGenerator = new MgrBuilder(table, getLanguageVer,
            make(mgrGenerator);
```

**Figure 15:** CodeGen is setup in the "InitBuilder.java"-file. The metaOn-method traverse the transform graph and creates a “File”-class from every table in the database. That “File” is then passed to the “make()”-method in this which is shown in Figure 17.
Figure 16: The old generation code for the ListBase-component was removed from the "InitBuilder.java"-file. The old code is shown in the commented rows.

```java
// Lists
TableBuilder listGenerator = new ListBuilder(table, getLanguageVersion());
make(listGenerator);

// TableBuilder listBaseGenerator = new ListBaseBuilder(table, getLanguageVersion());
// make(listBaseGenerator);
```

Figure 17: The "make()"-method of the "InitBuilder.java"-file mentioned in Figure 15. This takes a generated "File" model and writes it to the disc. Since the file is still a complex object, it needs to be sent back to the generator one last time to transform it into java-code (line 301 - 306).

```java
@Override
protected void make(ClassBuilder classBuilder) {
    super.make(classBuilder);
    progress();
}

// protected void make(File model) {

    try {
        final String fullname = dbEntity.getProject().getBaseDir()
            .getAbsolutePath().replace("\\", "/") + "/" + model.getName();

        println("Generating file "+ fullname + " using CodeGen 1.0.");

        final Generator gen = new JavaGenerator();
        final String result = gen.on(model).orElseThrow(
            () -> new UnsupportedOperationException()
                .message("Could not generate code from file.")
        );

        final java.io.File file = new java.io.File(fullname);
        file.getParentFile().mkdirs();

        try (Writer out = new OutputStreamWriter(new FileOutputStream(file),
            UTF_8)) {
            out.write(result);
        }

    } catch (Exception e) {
        LOGGER.error(e.getMessage(), e);
    }

    progress();
// }
With these changes, the tool can generate equivalent code from the same inputted database tables. The two versions of the software was compiled into two different .jar-files. The first one contained the original code and was marked “ACE only” and the second contained the old code but with one of the builders replaced with CodeGen, marked “ACE + CodeGen”. These two modules was analyzed using both the CKJM (Spinellis, 2013) and CK4J (Github 2013) tools for measuring Chidamber & Kemerer (CK) metrics. The CodeGen library file was also analyzed to see how it measured without the integration code and the ORM framework present. This was marked “CodeGen only”.

5.1.1 Verification

To make sure the code generator can uphold the quality standards of the last one, each of the eleven quality aspects defined by Cavano & McCall (1978) was reviewed. To verify the correctness of the code generator, a MySQL database with an installment of WordPress 4.22 (WordPress, 2015) was set up. The database has 11 tables. Source code was then generated using first the existing Speedment generator and then again using the new one with CodeGen integrated. Both systems generated 133 production ready java-components with equivalent code. The generated code compiled well in Java JDK 1.8.40. This ends the verification process for this case study as the primary focus is the quality factors of the generator itself, not the generated code.

Reliability is difficult to verify, but since the database and the generator communicates through the JDBC API, it should be independent on how the data is structured. The efficiency of the application is considered enough as the generation was performed in less than a second. Integrity is upheld as the generator only requires reading rights and none of the data is accessed, only the table structure. Usability is complex to verify but a good sign is that the code length of the generator integration could be kept down. The generator uses no external libraries, which is an indication of high flexibility. It is also designed to be modular and tolerate custom implementations of all interfaces. The portability aspect is not so good since the design relies not only on the Java language but also on a specific version of the language. This is not entirely a bad thing as the java language makes it easy to run the generator on all platforms that support the Java virtual machine, but might limit the potential usage. Finally, the interoperability is ensured by documenting the most prominent features of the generator on a wiki page.

Since the testability, maintainability and reusability aspects are in focus in this project, these are covered in more detail in the Analysis and Conclusions chapters.
5.2 Analysis

The original compiled Speedment executable consists of 1679 components and the “CodeGen only” file has 184. The compiled version of the ORM with the new code generator integrated (ACE + CodeGen) has a total of 1864 components. The reason why the number of components does not add up completely is that the integrated version also contains the new “SpeedmentTransformFactory” component described earlier. These three resulting .jar-files were analyzed using the Chidamber & Kemerer (1994) metrics. Lincke, Lundberg & Löwe (2008) show that the CK-metrics can be measured differently by different tools. Therefore two different tools were used. The results are compared in Table 2.

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE_only.jar</td>
<td>11629</td>
<td>1679</td>
<td>10,13</td>
<td>9,83</td>
<td>0,38</td>
<td>0,42</td>
<td>23,59</td>
</tr>
<tr>
<td>ACE+CodeGen.jar</td>
<td>13112</td>
<td>1864</td>
<td>9,72</td>
<td>9,46</td>
<td>0,35</td>
<td>0,38</td>
<td>22,59</td>
</tr>
<tr>
<td>CodeGen_only.jar</td>
<td>184</td>
<td>184</td>
<td>6,08</td>
<td>6,08</td>
<td>0,1</td>
<td>0,1</td>
<td>13,59</td>
</tr>
</tbody>
</table>

Table 2: This table show a comparison between the results returned by the two CK-tools CKJM and CK4J. The first .jar-file contained the old 1.0.8 version of the Speedment code generator. The second row is a recompiled .jar-file where CodeGen has been integrated and the last row is the results of only measuring the CodeGen-library. The first column shows how many components were measured. For each measure, both the CKJM and the CK4J result is shown.

As can be seen in Table 2, CKJM did not measure all the files in the .jar-archive of the first two rows. In the last row every component is measured. Another problem is that some of the values appear to be incorrect. The Depth of Inheritance Tree (DIT) value of the “InitBuilder” component is valued to “1” by the CKJM-tool, but if the value is calculated manually, it should be “3”. In the following tables, only the CK4J tool was used since those values appeared to be more reliable.

<table>
<thead>
<tr>
<th>Summary “ACE only”</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>384,00</td>
<td>43,00</td>
<td>981,00</td>
<td>164,00</td>
<td>7,00</td>
<td>72260,00</td>
</tr>
<tr>
<td>Min</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>-1,00</td>
<td>0,00</td>
</tr>
<tr>
<td>Average</td>
<td>9,83</td>
<td>0,42</td>
<td>25,45</td>
<td>10,41</td>
<td>1,90</td>
<td>168,47</td>
</tr>
<tr>
<td>Median</td>
<td>4,00</td>
<td>0,00</td>
<td>11,00</td>
<td>7,00</td>
<td>1,00</td>
<td>4,00</td>
</tr>
<tr>
<td>Components measured</td>
<td>1679</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: The results of measuring the existing Speedment software version 1.0.8 with the original code generator.

<table>
<thead>
<tr>
<th>Summary “ACE+CodeGen”</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>384,00</td>
<td>43,00</td>
<td>981,00</td>
<td>164,00</td>
<td>7,00</td>
<td>72260,00</td>
</tr>
<tr>
<td>Min</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>0,00</td>
<td>-1,00</td>
<td>0,00</td>
</tr>
<tr>
<td>Average</td>
<td>9,46</td>
<td>0,38</td>
<td>24,33</td>
<td>10,45</td>
<td>1,83</td>
<td>154,33</td>
</tr>
<tr>
<td>Median</td>
<td>4,00</td>
<td>0,00</td>
<td>11,00</td>
<td>7,00</td>
<td>1,00</td>
<td>4,00</td>
</tr>
<tr>
<td>Components measured</td>
<td>1864</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: The results of measuring the Speedment software with the new code generator integrated to replace the “ListBaseBuilder” functionality.

27
### Table 5: The results of measuring only the new code generator library.

At a first look, we can conclude that the maximum and minimum values in Table 3 and 4 are the same. This means that the boundaries of the measurements have not changed by adding the 184 CodeGen components or the custom transform factory. One value is troublesome however. A Depth of Inheritance Tree (DIT) value of -1 is not possible if the calculation has been done properly. The definition by Chidamber & Kemerer (1994) of the DIT value requires the value to be at least 1 since it is a measurement of the number of levels in the inheritance hierarchy. This appears to be due to an error in the measurement software. Of the entire log with 1864 rows, only one component had a DIT value of less than 1 and that was “org.apache.log4j.net.SMTPAppender$1”. Since it is a component from an external library and it appears in both versions of the generator, it is likely to have a negligible affect on the results.

By comparing the average values of Table 3 with those of Table 4 we can see that not all the values have improved by integrating CodeGen. The Weighted Methods per Class (WMC), Number Of Children (NOC), Responsibility For a Class (RFC), Depth of Inheritance Tree (DIT) and Lack of Cohesion Of Methods (LCOM) have improved slightly, but Coupling Between Objects (CBO) has increased. It should be noted that the average and median values are dependent on the total number of components measured.

We can also conclude from the summaries above that the average values of the “CodeGen only” measurements in Table 3 are below the ones from the “ACE only” binary in Table 5 on all six of the CK metrics. Since a lower value is better in the CK suit (Basili & Melo, 1996), this is a good sign. To put these values into perspective, we can compare them to values measured by Lincke, Lundberg & Löwe (2008) on three different open source projects of various size and scope. These can be seen in Table 6.

### Table 6: The CK measurements of three different open source projects made by (Lincke, Lundberg & Löwe, 2008) using the CKJM tool.

It should be noted that Lincke, Lundberg & Löwe (2008) used the CKJM-tool and not CK4J for these tests and that the CKJM tool is designed to count the complexity of all methods as 1 and not use any complexity measurement algorithm as proposed by Li & Henry (1993). The Depth of Inheritance Tree (DIT) values produced by CKJM should be incremented by one before comparing the results to those of CK4J since CK4J counts inclusively of the original ancestor while CK4J counts non-inclusively.
The code generator measurements shown in Table 5 are compared to the three open source projects of Table 6 in the graphs shown in Figure 19.

Compared to the three open source projects, CodeGen performs well in most metrics except Coupling Between Objects (CBO). With almost no inheritance, both DIT and NOC values will be low. In responsibility (RFC), CodeGen has the lowest value of all the four projects. Coupling Between Objects is high, the highest of them all. Lack of Cohesion Of Methods is low, but not the lowest. The “Jaim” project has a lower score still.

In Figure 20, the six average values of the three binaries from Table 3, 4 and 5 are shown in diagrams.
Since the integration is only a replacement for one of the builders seen in Figure 13, there is a risk that the measurements are affected by the fact that the “ACE + CodeGen” contains two different code generators in the same executable. Therefore, a separate analysis was made as well. This was performed with the same tool and the same binaries as the one above, but only the components that differed between the two versions were considered. This means that it was narrowed down to five components in the “ACE only” version and six components in the “ACE + CodeGen” version. The results are shown in Table 7 and 8. Note that in this case we can display exact measures as the total number of rows are limited.

<table>
<thead>
<tr>
<th>Subset results “ACE only”</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitBuilder$1</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>InitBuilder</td>
<td>13</td>
<td>0</td>
<td>84</td>
<td>44</td>
<td>4</td>
<td>74</td>
</tr>
<tr>
<td>ClassOverrider$1</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ClassOverrider</td>
<td>14</td>
<td>3</td>
<td>66</td>
<td>25</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td>ListBaseBuilder</td>
<td>15</td>
<td>0</td>
<td>66</td>
<td>11</td>
<td>6</td>
<td>89</td>
</tr>
</tbody>
</table>

Table 7: The results of measuring the original files in the “ACE only” version later changed when integrating CodeGen.

**Figure 20:** The average values of the CK metrics measured on the entire compiled .jar file of each version of the system. These values are taken from the “Average” row in Table 3, 4 and 5.
Table 8: The results of measuring the changed files in the “ACE + CodeGen” version. The values that differ from Table 7 are written with gray background. In addition to the six components shown above, all the 184 components of the CodeGen library were of course added as well. A summary of the measures of those is shown in Table 5.

The first three component measurements did not differ at all between the two versions. This is mostly because the changes made in Figure 15, 16 and 17 add the same amount of dependencies as they took away. A slight improvement is noticed in the “ClassOverrider” component, but this is almost negligible. The greatest improvements can be noted in the ListBaseBuilder component. 3 of the measures have improved by the changes (WMC, DIT and LCOM), two have become worse (RFC and CBO) and one remains the same (NOC). The changes to the ListBaseBuilder component are responsible for a large part of the total integration effect on the measurement values. This can be seen if the two measurements of ListBaseBuilder are compared to the total difference of the integration changes seen in Table 9.

<table>
<thead>
<tr>
<th>Component</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
<th>LCOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitBuilder$1</td>
<td>2</td>
<td>0</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>InitBuilder</td>
<td>13</td>
<td>0</td>
<td>84</td>
<td>44</td>
<td>4</td>
<td>74</td>
</tr>
<tr>
<td>ClassOverrider$1</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ClassOverrider</td>
<td>14</td>
<td>2</td>
<td>66</td>
<td>24</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td>ListBaseBuilder</td>
<td>7</td>
<td>0</td>
<td>78</td>
<td>51</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>SpeedmentTransformFactory</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9. The sum of the values in Table 7 and 8, with the difference between the two calculated on the last row. A value below zero generally signifies an improvement and a value above zero signifies a reduction.

5.3 Conclusions

Low Chidamber & Kemerer (CK) values are a sign of good maintainability and testability in the software, as described by both Chidamber & Kemerer (1994) and Basili & Melo (1996). In the analysis we can see that the CodeGen library seen in Table 5 has very low values compared to the whole Speedment tool seen in Table 3. We can also compare five of the six metrics to the results of three other Open Source projects in Table 6 and see that CodeGen performs well here in some of the measured aspects, but not all of them.

It is a good sign that the average Response For a Class (RFC) value of 13.87 in CodeGen is the lowest of all the tested projects in Table 6. This can probably be attributed to the fact that CodeGen has many small components with distinct purposes. Bruntink & van Deursen (2004) show that low RFC values is a sign of good testability, which was one of the goals with the project. This can be put into perspective by looking at the average RFC of the whole Speedment generator before and after the changes (Table 3 and 4 respectively). Even though 1679 components were measured in the first run and only 185 were added, the average RFC
dropped from 25.45 to 24.33. This value would most likely continue to drop if more of the builders was replaced since the CodeGen average is only 13.87.

The Depth of Inheritance Tree (DIT) and Number Of Children (NOC) averages are also very low in CodeGen. This is good since Poels & Dedene (2001) show that a large inheritance tree makes it more difficult to maintain and test the system. The reason for this improvement is a combination of the trait system that allows components to share functionality through functional interfaces and the design that favors composition above inheritance. This effect has been described before by Black & Scharli (2004) and Hunt & Thomas (2000) respectively. Low values of DIT and NOC might not be entirely desirable, however. Both Poels & Dedene (2001) and Daly et al. (1996) suggest an optimal DIT value of 2 respectively 3, which is above the average of CodeGen.

A big issue with the design shown in both Figure 19 and 20 as well as the summary in Table 9 is that the Coupling Between Objects (CBO) has increased. In Figure 19 specifically, CodeGen stands out with an average coupling of 10,212 compared to the second highest (ProGuard) with only 8,712. There are many reasons for this. First of, every “Transform” component is dependent on the model it is transforming. Since there are 26 different models included in the CodeGen library and every one of them has a corresponding transform component, this will add to the total amount of couplings within the library. Another factor is that all the models are expressed as interfaces to make it easier to create custom implementations, but that also add an additional standard implementation for each of the 26 models. The factory pattern used to determine which implementation should be used when instantiating the models is realized using an inner class in the interface components. The CK4J tool does not differentiate between inner classes or ordinary components and it therefore adds an additional component for each of the 26 models. Conclusively, the chosen design requires multiple tightly coupled components to be implemented for every language concern. This leads to the high average CBO value of 10,21 which is more than any of the projects measured in Figure 19. Sahraoui, Godin & Miceli (2000) propose a maximum CBO of 14. Singh, Bhattacherjee & Bhattacharjee (2012) note that high CBO values are bad if they appear in the interaction between modules, which the integration results in Table 8 is an example of, but not necessarily within a module. It is possible that the effects of the high CBO average is countered by the low Lack of Cohesion of Methods (LCOM) and Response For a Class (RFC) values, but there is too little data in this study to verify if that assumption is correct.

The Weighted Methods per Class (WMC) metric is difficult to conclude anything from. The 6.08 average in CodeGen is below the measured 9.83 average of “ACE only”, and the integration did indeed lower it to 9.46 as seen in Figure 20. These values do not say much, however, since the definition of WMC is somewhat unclear (Churcher et al. 1995). Different measurement tools interpret the metric differently (Lincke, Lundberg & Löwe, 2008). Li & Henry (1993) propose that the Cyclomatic Complexity Number by McCabe (1976) could be used when calculating the WMC of a component to improve the metric, but no automatic tool for measuring in that way was used in this project.

The method chosen to integrate CodeGen into the Speedment tool requires minimal changes to the existing software. The file that was changed the most was “ListBaseBuilder.java”. The changes resulted in a significant decrease in Lack of Cohesion Of Methods (LCOM) as show in Table 9. The decrease came at a cost of higher values in Response For a Class (RFC) and Coupling Between Objects (CBO). The reason for the increase in RFC is probably that
CodeGen has a large number of components that are involved in almost every task. The large amount of coupled components is also the reason why CBO is high. Even though the integration worsened the RFC value of the ListBaseBuilder.java-file alone, the measurements in Table 4 indicate that the average RFC of the whole project was actually improved by the changes.

These results might indicate that the use of CodeGen will probably help improve the cohesion and complexity of a system as indicated by the low LCOM, NOC and DIT values both in the CodeGen library and the integration code, but that it might come at a cost of more coupling as indicated by the high values of CBO and RFC. Finally it should be mentioned that the integration measurements are probably affected by the fact that only one of the builders in Figure 13 was replaced. If all of the builders were to be replaced, a large portion of the components in the study could be removed and that could improve the measurements significantly as the “CodeGen only” measures is lower than all the “ACE only” stacks in Figure 20.

The low RFC, NOC and DIT values in the “CodeGen only” measurements indicate that the testability of CodeGen is good. This is similar to the conclusions made by Poels & Dedene (2001), Bruntink & van Deursen (2004) and Khalid, Zehra & Arif (2010). The high CBO averages in both the CodeGen library and the integration code can be seen as a sign of bad maintainability (Singh, Bhattacherjee & Bhattacharjee, 2012) as classes with many dependencies take more time to modify. At the same time, a low value in WMC is good for maintainability as small components are less frequently subject to change as described by Khomh, Di Penta & Guéhéneuc (2009). This is hard to verify in this case as the chosen measurement tools make a reliable WMC value difficult to establish (Churcher et al. 1995).
6 Concluding Remarks

6.1 Summary
The problem that motivated this project is that a code generator designed with the top-down approach is tightly coupled and has bad cohesion, which leads to low modifiability, testability and reusability (Basili & Melo, 1996). The hypothesis was that a modularized code generation framework with functionality separated into models and views can reach better cohesion and less coupling. In a case study on the Speedment tool, a code generator written with a top-down approach was partly replaced with a modular one. The results of the study indicates that the testability of the generation indeed seem to have improved with the modular approach, but does not indicate any improvements on maintainability.

6.2 Discussion
This case study was done on the Speedment Object-Relational Mapping tool. The source code of the Speedment tool is proprietary software and permission from the copyright holder was required to do the study as it involved modifying parts of the product. All the results produced as part of this project are released as Open Source under the Apache 2 license.

Since no data was collected from users through either interviews or other methods, there was no risk that the integrity of anyone was compromised due to this study. Neither the existing code generator nor the one produced in this project takes existing data into consideration when analyzing the SQL database, therefore there was no risk of exposing sensitive information when verifying the application on the existing database.

The metrics by Chidamber & Kemerer (1994) have been interpreted in many different ways during the years since the suit was first introduced. Lincke, Lundberg & Löwe (2008) illustrated in a study how much different measurement tools could differ when calculating the metrics on the same projects. In this project, two different tools were used to measure quality, but since the reliability of one of the tools was questionable with obviously erroneous output, it was discarded in the final analysis. This should be considered as it might affect the conclusion of the study. Until the results can be verified using other instruments, the results should be used very carefully.

In an ethical perspective, there is a potential risk that more automated programming might minimize the influence of the developer over the final program. Questions could be raised regarding who really holds the intellectual property over a software when a large part of the code base is suddenly being written by the computers themselves. Or, if a security critical program was to be generated and that program failed, who is really responsible? Those are concerns that need to be addressed in the near future.

In this case, this is less of a risk since no artificial “thinking” is involved in the generation process. The program is a tool that produces one correct result for every one set of inputted database tables. On the contrary, there are many social benefits with this kind of automated programming. By freeing up time for the developers to focus on the architecture of a system instead of implementing the same patterns over and over again, the productivity and creativity of the programmers can be put to better use. If a more automated way of working can help prevent bugs and increase the testability of a system as indicated by Mathiske,
Simon & Ungar (2006), it can lead to better and more secure programs running on everything from servers to dishwashers.

### 6.3 Future Work

In this study, some improvements and some reduction of the quality aspects could be measured when replacing parts of the generation pipeline with the CodeGen library. In a short perspective, it would be interesting to see if this effect continues when more builders shown in Figure 13 are replaced. If all the builders were to be replaced, it is likely that many components of the old system could be removed, which probably would put the average values closer to the “CodeGen only” measures. Another thing that could continue this project in a shorter perspective is to recreate the measures using other Chidamber & Kemerer (CK) tools than CK4J. As Lincke, Lundberg & Löwe (2008) illustrated, different tools can vary greatly in their results.

Building on the results of this case study, it would be interesting to see how the maintainability aspect of the modular design technique can be improved. Seeing as most of the CK metrics improved with the design, is there a way to avoid the increased coupling and still get the positive effects of reduced complexity and improved cohesion? Since many of the features in the java language used in this project like default implementations of interfaces were quite new, it will be interesting to see how the usage of these techniques evolves in the coming years.

In a more distant time frame, improved code generators might very well change the way software developers work with programs. Already, memory management, closing of resources and multi-threading can be done more or less in the background without bothering the developer. It is not impossible that writing code altogether might soon be something of the past. When the point is reached were a computer program can probably solve a problem in a better and more efficient way than an algorithm engineer, there will be no need to create anything other than even better code generators.
References


Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R. & Black, A. (2006). Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems. 28 (2), pp. 331 – 388.


performance of an object relational mapping tool. Proceedings of the 2009 Annual 
Research Conference of the South African Institute of Computer Scientists and 
Information Technologists. pp. 150 – 159.
Appendix A - ListBaseBuilder.java

Most of the integration of the case study was done by replacing the “ListBaseBuilder.java” file of the commercial Speedment framework. Both versions of the file is shown here. The old version is copyright Speedment AB and the new one created as part of this project is licensed under the Apache 2 license. The old file is cited here with permission from the original author.

1 Old version (Proprietary)

```java
package com.speedment.ace.plugin.java.builders;

import com.speedment.core.cache.list.OrderByType;
import com.speedment.core.cache.list.StructList;
import com.speedment.core.cache.struct.Struct;
import com.speedment.core.db.model.Column;
import com.speedment.core.db.model.Table;
import com.speedment.core.tableorder.TableOrder;
import java.lang.reflect.Method;

/**
 * Copyright (c), Speedment AB
 * @author peml
 */

public class ListBaseBuilder extends ClassOverrider {

    public static final String LIST_KEY_TYPE = "String";

    public ListBaseBuilder(final Table table, final String languageVersion) {
        super(table, languageVersion, StructList.class);
        MapBaseBuilder.addExcludeInstanceMethods(this);
    }

    @Override
    public String getClassName() {
        return super.getJavaName() + Table.LIST_BASE_POST_TAG;
    }

    @Override
    public boolean isInBasePackage() {
        return true;
    }

    @Override
    public void onClassStart() {
        super.onClassStart();
        doclines("@author Generated by www.speedment.com",
                "", "This file is generated automatically using the Speedment Java " + getLanguageVersion() + " plugin.",
                "Any manual changes may be overwritten.",
                "Please make changes by overriding methods in the " +
                docLink(getTable().getMapSimpleName()) + " class."
            );
        openBlock("public class ", getClassName(), " extends ",
                "StructArrayList<", getJavaName(), ">");
    }

40
@Override
public void onStaticMembers() {
    super.onStaticMembers();
    appendlnSerialVersionUID();
}

@Override
public void onConstructorsStart() {
    super.onConstructorsStart();
    doclines("Plain constructor");
    openBlock("public ", getClassName(), "() ");
    codeln("super()");
    closeBlock();
    doclines("Copies a new (shallow) instance of this <code>" +
    getJavaName() + "</code> object.");
    openBlock("public ", getClassName(), "(final ", getClassName(), " ", getJavaVariableName() + Table.LIST_POST_TAG, ") ");
    codeln("super(" + getJavaVariableName() + Table.LIST_POST_TAG, ")");
    closeBlock();
    addImport(com.speedment.core.cache.map.StructMap.class);
    addImport(com.speedment.core.cache.list.StructList.class);
    addImport(com.speedment.core.cache.list.StructArrayList.class);
    addImport(com.speedment.core.cache.list.OrderByType.class);
    addImport(getNamespaceForNonBase() + "." + getJavaName());
    addImport(getNamespaceForNonBase() + "." + getJavaName() +
    Table.MAP_POST_TAG);
    addImport(getNamespaceForNonBase() + "." +
    table.getMapOnHeapImplSimpleName());
    addImport(getNamespaceForNonBase() + "." + getJavaName() +
    Table.LIST_POST_TAG);
}

@Override
public void onMethodsStart() {
    commentln("Instance methods");
    overrideln();
    openBlock("public ", table.getListSimpleName(), " newInstance() ");
    returnln("new ", table.getListSimpleName(), "()");
    closeBlock();
    overrideln();
    openBlock("public StructMap<K, V> from a StructList<V>, K unknown");
    closeBlock();
    overrideln();
    openBlock("public ", table.getMapSimpleName(), " newStringMapInstance() ");
    returnln("new ", table.getMapOnHeapImplSimpleName(), "()");
    closeBlock();
    overrideln();
    openBlock("public ", table.getListSimpleName(), " newListInstance() ");
    returnln("new ", table.getListSimpleClassName(), "()");
}
closeBlock();

commitempty("Override methods");
super.onMethodsStart();
}

@override
public void onTableForeignKeyBackreferenceMethod(final Column column, final Table fkTable, final Column fkColumn) {
super.onTableForeignKeyBackreferenceMethod(column, fkTable, fkColumn);
if (!fkTableExistsElseAdd(fkTable)) {
addImport(TableOrder.class);
addImport(fkTable.getStructImport(getBaseNamespace()));
addImport(OrderByType.class);

fkOrderBy(column, fkTable, fkColumn, "", "OrderByType.ASC");
fkOrderBy(column, fkTable, fkColumn, ", final OrderByType type", ", "type");
}

private void fkOrderBy(final Column column, final Table fkTable, final Column fkColumn, final String extraParam, final String extraCallParam) {
openBlock("public ", getJavaName(), Table.LIST_POST_TAG, ", orderBy", fkTable.getJavaTypeName(), "(final TableOrder<", fkTable.getJavaTypeName(), ", > order", extraParam, ",")");
returnln(getJavaName() + Table.MAP_ON_HEAP_IMPL_BASE_POST_TAG, ", orderBy", fkTable.getJavaTypeName(), "(this, order", ", extraCallParam, ")");
closeBlock();
}

@override
public long getSerialVersionUIDDelta() {
return -5;
}

//ClassOverrider methods overrided...
@override
public void appendMethodAnnotations(final Method method) {
overrideln();
}

@override
public void appendMethodBody(final Method method) {
indent();
appendln("\n");
String returnTypeName = getMethodReturnTypeName(method);
print("return ", returnTypeName, ", super.", method.getName(), ",");

int counter = 0;
for (final Class<?> parameterClass : method.getParameterTypes()) {
if (counter > 0) {
append("", ");
}
append(getParameterName(method, counter, parameterClass));
counter++;
}
```java
appendln("\n", getDelimiter());
outdent();
println("\n");
}

@ Override
public String getStructContainerSubstitute() {
    return table.getListSimpleName();
}

@ Override
public boolean isIncluded(Method method) {
    if (method.getReturnType() == Struct.class) {
        // Do not override a method that returns the basic
        // Struct object since it becomes a redundant cast...
        return false;
    }
    return super.isIncluded(method);
}
}

2 New version (Open Source)

package com.speedment.ace.plugin.java.builders;

import static com.speedment.codegen.Formatting.classToJavaFileName;
import static com.speedment.codegen.Formatting.shortName;
import com.speedment.codegen.base.Generator;
import com.speedment.codegen.base.Transform;
import com.speedment.codegen.lang.controller.AutoImports;
import com.speedment.codegen.lang.controller.AutoJavadoc;
import com.speedment.codegen.lang.models.Class;
import com.speedment.codegen.lang.models.Constructor;
import com.speedment.codegen.lang.models.Field;
import com.speedment.codegen.lang.models.File;
import com.speedment.codegen.lang.models.Generic;
import com.speedment.codegen.lang.models.Import;
import com.speedment.codegen.lang.models.Method;
import com.speedment.codegen.lang.models.Type;
import static com.speedment.codegen.lang.models.constants.DefaultAnnotationUsage.OVERRIDE;
import static com.speedment.codegen.lang.models.constants.DefaultJavadocTag.AUTHOR;
import static com.speedment.codegen.lang.models.constants.DefaultType.LONG_PRIMITIVE;
import static com.speedment.codegen.lang.models.constants.DefaultType.STRING;
import com.speedment.codegen.lang.models.values.ReferenceValue;
import com.speedment.core.cache.common.StructContainer;
import com.speedment.core.cache.list.OrderByType;
import com.speedment.core.cache.list.StructArrayList;
import com.speedment.core.cache.map.StructMap;
import com.speedment.core.db.model.Column;
import com.speedment.core.db.model.Table;
import com.speedment.core.expression.WhereExpression;
import com.speedment.core.tableorder.TableOrder;
import java.util.Comparator;
```

import java.util.Optional;
import java.util.function.Supplier;
import static java.util.stream.Collectors.joining;
import java.util.stream.Stream;

/**
 * Licensed under the Apache 2 license.
 * @author Emil Forslund
 */

public class ListBaseBuilder implements Transform<Table, File> {

    @Override
    public Optional<File> transform(Generator gen, Table table) {
        // Prepare types.
        final Type entityType = createType(table, false, table::getJavaTypeName);
        final Type listType = createType(table, false, table::getListSimpleName);
        final Type mapType = createType(table, false, table::getMapSimpleName);
        final Type mapOnHeapImplType = createType(table, false, table::getMapOnHeapImplSimpleName);
        final Type baseType = createType(table, true, table::getJavaTypeName);
        final Type mapOnHeapImplBaseType = createType(table, true, table::getMapOnHeapImplName);
        final Type listBaseType = createType(table, true, table::getListSimpleName);

        final File file = File.of(classToJavaFileName(listBaseType.getName()));

        // Create class with supertype.
        final Class clazz = Class.of(shortName(listBaseType.getName())).public_()
                .set(createJavadoc(table))
                .setSupertype(Type.of(StructArrayList.class)
                        .add(Generic.of().add(entityType))
                );

        // Add member field.
        file.add(Import.of(baseType));
        clazz.addField(Import.of(baseType)).private_.static_.final_()
                .set(new ReferenceValue(shortName(baseType.getName()) + ".SERIAL_VERSION_UID - 5")
        );

        // Create constructors.
        clazz.add(Constructor.of().public_()
                .set(Javadoc.of("Plain constructor"))
        );

        clazz.add(Constructor.of().public_()
                .set(Javadoc.of("Copies a new (shallow) instance of this <code>" + table.getJavaTypeName() + "</code> object.")
                        .add(Field.of(table.getJavaVariableName(), listBaseType).final_())
                .add("super(" + table.getJavaVariableName() + ");")
        );
    }
}

44
// Create methods.
// newInstance
clazz.add(Method.of("newInstance", listType)
    .public_().add(OVERRIDE)
    .add("return new " + table.getListSimpleName() + "();")
);

// newInstance
file.add(Import.of(Type.of(UnsupportedOperationException.class)));
class.add(Method.of("newMapInstance", Type.of(StructMap.class)
    .add(Generic.of().add(Type.of(String.class)))
    .add(Generic.of().add(entityType))
    .public_().add(OVERRIDE)
    .add("throw new UnsupportedOperationException("Unable to create generic StructMap<K, V> from a StructList<V>, K unknown");")
);

// newStringMapInstance
file.add(Import.of(mapOnHeapImplType));
class.add(Method.of("newStringMapInstance", mapType)
    .public_().add(OVERRIDE)
    .add("return new " + table.getMapOnHeapImplSimpleName() + "();")
);

// newListInstance
class.add(Method.of("newListInstance", listType)
    .public_().add(OVERRIDE)
    .add("return new " + table.getListSimpleName() + "();")
);

// orderBy[FK table name]
table.getForeignKeyReferences().stream()
    .flatMap(fk -> {
        final Table foreign = fk.getTable();
        final Type foreignType = createType(foreign, false,
            foreign::getJavaTypeName);
        final Type orderType = Type.of(TableOrder.class)
            .add(Generic.of().add(foreignType));
        file.add(Import.of(foreignType));
        file.add(Import.of(mapOnHeapImplBaseType));
        file.add(Import.of(Type.of(OrderByType.class)));

        final Field orderField = Field.of("order",
            orderType).final_();
        final Field typeField = Field.of("type",
            Type.of(OrderByType.class)).final_();
        final String mapOnHeapImplBaseName =
            shortName(mapOnHeapImplBaseType.getName());

        final Method m1 = Method.of("orderBy" +
            foreign.getJavaTypeName(), listType)
            .public()
            .add(orderField)
            .add(
                "return " + mapOnHeapImplBaseName + 
                ".orderBy" + foreign.getJavaTypeName() + 
                "(this, order, OrderByType.ASC);"
            );
    });
```java
final Method m2 = Method.of("orderBy" +
foreign.getJavaTypeName(), listType)
  .public()
  .add(orderField)
  .add(typeField)
  .add(
    "return " + mapOnHeapImplBaseName +
    ".orderBy" + foreign.getJavaTypeName() +
    "(this, order, type);");

return Stream.of(m1, m2);
}).forEachOrdered(clazz::add);

// clone
clazz.add(Method.of("clone", listType)
  .public().add(OVERRIDE)
  .add(createExtending(table, "clone"))
);

// limit
clazz.add(Method.of("limit", listType)
  .public().add(OVERRIDE)
  .add(Field.of("limit", LONG_PRIMITIVE).final_())
  .add(createExtending(table, "limit", "limit"))
);

class.add(Method.of("limit", listType)
  .public().add(OVERRIDE)
  .add(Field.of("start", LONG_PRIMITIVE).final_())
  .add(Field.of("limit", LONG_PRIMITIVE).final_())
  .add(createExtending(table, "limit", "start", "limit"))
);

// orderBy
clazz.add(Method.of("orderBy", listType)
  .public().add(OVERRIDE)
  .add(Field.of("columns",
    Type.of(Column.class).setArrayDimension(1)).final_())
  .add(createExtending(table, "orderBy", "columns"))
);

class.add(Method.of("orderBy", listType)
  .public().add(OVERRIDE)
  .add(Field.of("expressions",
    STRING.setArrayDimension(1)).final_())
  .add(createExtending(table, "orderBy", "expressions"))
);

final Field comparatorField = Field.of("comparator",
Type.of(Comparator.class)
  .add(Generic.of().add(entityType))
).final_();

class.add(Method.of("orderBy", listType)
  .public().add(OVERRIDE)
  .add(comparatorField)
  .add(createExtending(table, "orderBy", "comparator"))
);
```
clazz.add(Method.of("orderBy", listType)
    .public_().add(OVERRIDE)
    .add(comparatorField)
    .add(Field.of("orderByType", Type.of(OrderByType.class)).final_())
    .add(createExtending(table, "orderBy", "comparator", "orderByType"));
}
clazz.add(Method.of("orderBy", listType)
    .public_().add(OVERRIDE)
    .add(comparatorField.setName("comparator0"))
    .add(comparatorField.setName("comparator1"))
    .add(createExtending(table, "orderBy", "comparator0", "comparator1"));
}
clazz.add(Method.of("orderBy", listType)
    .public_().add(OVERRIDE)
    .add(Field.of("orderByType", Type.of(OrderByType.class)).final_())
    .add(Field.of("columns", Type.of(Column.class).setArrayDimension(1)).final_())
    .add(createExtending(table, "orderBy", "orderByType", "columns"));
}
clazz.add(Method.of("orderBy", listType)
    .public_().add(OVERRIDE)
    .add(Field.of("column", Type.of(Column.class)).final_())
    .add(Field.of("orderByType", Type.of(OrderByType.class)).final_())
    .add(Field.of("expressions", STRING.setArrayDimension(1)).final_())
    .add(createExtending(table, "orderBy", "orderByType", "expressions"));
}
clazz.add(Method.of("orderBy", listType)
    .public_().add(OVERRIDE)
    .add(Field.of("column", Type.of(Column.class)).final_())
    .add(Field.of("orderByType", Type.of(OrderByType.class)).final_())
    .add(createExtending(table, "orderBy", "column", "orderByType"));
}
clazz.add(Method.of("orderBy", listType)
    .public_().add(OVERRIDE)
    .add(comparatorField.setName("comparator0"))
    .add(comparatorField.setName("comparator1"))
    .add(comparatorField.setName("comparator2"))
    .add(createExtending(table, "orderBy", "comparator0", "comparator1", "comparator2"));
}
clazz.add(Method.of("orderBy", listType)
    .public_().add(OVERRIDE)
    .add(Field.of("column0", Type.of(Column.class)).final_())
    .add(Field.of("orderByType0", Type.of(OrderByType.class)).final_())
    .add(Field.of("column1", Type.of(Column.class)).final_());
}
...
// union
class add(Method of("union", listType)
   .public_().add(OVERRIDE)
   .add(Field of("structContainer",
       Type.of(StructContainer.class)
       .add(Generic.of().add(STRING))
       .add(Generic.of().add(entityType))
       ).final_()
       .add(createExtending(table, "union", "structContainer")))
   );

// where
final Field whereField = Field.of("whereExpression",
       Type.of(WhereExpression.class)
       .add(Generic.of().add(entityType))
       ).final_();
class add(Method of("where", listType)
   .public_().add(OVERRIDE)
   .add(whereField)
   .add(createExtending(table, "where", "whereExpression"))
   );
class add(Method of("where", listType)
   .public_().add(OVERRIDE)
   .add(whereField.setName("whereExpression0"))
   .add(whereField.setName("whereExpression1"))
   .add(createExtending(table, "where", "whereExpression0",
      "whereExpression1"))
   );
class add(Method of("where", listType)
   .public_().add(OVERRIDE)
   .add(whereField.setName("whereExpression0"))
   .add(whereField.setName("whereExpression1"))
   .add(whereField.setName("whereExpression2"))
   .add(createExtending(table, "where", "whereExpression0",
      "whereExpression1", "whereExpression2"))
   );
class add(Method of("where", listType)
   .public_().add(OVERRIDE)
   .add(Field.of("expression", STRING).final_())
   .add(createExtending(table, "where", "expression"))
   );
file.add(clazz)
   .call(new AutoImports(gen.getDependencyMgr()))
   .call(new AutoJavadoc<>());

   return Optional.of(file);
}

private Type createType(Table table, boolean isBase, Supplier<String>
   nameGetter) {
   final String namespace =
table.getNameSpace(table.getProject().getNameSpaceName());
   return Type.of(
      namespace +
(isBase ? ".base." : ".") +
nameGetter.get() +
(isBase ? "Base" : "")
);

private Javadoc createJavadoc(Table table) {
    final String namespace =
table.getNamespace(table.getProject().getNamespaceName());
    final String mapName = table.getMapSimpleName();

    return Javadoc.of(
        "This file is generated automatically using CodeGen 1.0.",
        "Any manual changes may be overwritten.",
        "Please make changes by overriding methods in the " +
        "{@link " + namespace + "." + mapName + " " + mapName + "}" +
        "class."
    ).add(AUTHOR.setValue("Generated by CodeGen 1.0"));
}

private String createExtending(Table table, String methodName,
String... params) {
    return "return (" + table.getListSimpleName() + ") super." +
    methodName + "(" + Stream.of(params).collect(joining(", ")) + ");";
}