The Evolutionary Model In The World Of Object-Oriented Languages and Version Control Systems

René Boschma
University of Twente
The Netherlands
r.boschma@student.utwente.nl

ABSTRACT
Modern day software systems go through a lot of change during development and creation. Bug fixes and requirement changes are two examples of reasons that give rise to change in a system. All these changes combined can be seen as the evolution of a system. This process resembles the process of animals evolving and eventually giving rise to new species. When looking to the evolution of animals it may be hard to pinpoint the exact time of each change. When looking back to the evolution of software systems, this problem is obsolete when a system is developed using a version control system (VCS). A VCS keeps records of each change made to a system. These records contain the change made to the system, but also when and by who the change was made. A more commonly used name for such a record is a commit. In other words, a VCS maintains the entire history of a system. Analyzing this history may give interesting insights in the development of systems. An example of such a question is what characteristics give rise to frequent change in part of a system. Answering this question may lead to new programming strategies which aim to reduce code change and aid in code reuse. Answering this question requires a researcher to write code that extracts the data from a VCS. This code most often limits to a single VCS and maybe a single programming language. An abstract model between the research and the combination of VCSs and programming languages might make it easier for a researcher to gather the data needed while also broadening the possible dataset. No such model could be found. That is why this paper proposes such an abstract model. The model in this paper will be referred to as the Evolutionary Model (EM). The goal of the EM is to create an abstract layer between the programming language and VCS in order to make evolution analysis easier. We believe that directed graphs are a great way of creating such an abstract layer. Therefore this paper will use graphs to describe the model. This paper, due to time constraints, limits its scope to the object-oriented programming paradigm. Nevertheless, we believe that the model, with some changes, can be applied to other language paradigms.

Before the evolution of a system can be modeled, first the system itself needs to be modeled as a graph. Various graph types are known to model a software system. Examples are call graphs, control flow graphs, abstract syntax trees, inheritance graphs and class collaboration graphs. This paper identifies some graph types used in available evolution analysis tools. The EM will be based on a class collaboration graph to model a single instance of a system. This graph type uses the class definitions in a system. Class collaboration graphs show the architecture of an object-oriented system, while still maintaining a reasonable size. A consequence of this decision is that most of the data, that can be extracted from the EM, is on a class-based level.

In the process of evolution, it can be asked which version originated from which version. For example, adding a feature X to version Y might have a different impact on the system then adding X to a version Z. The same functionality is added, but different approaches might be used which have a different result on the evolution of the system. This paper will identify some possible version dependencies and describe how these are represented in the EM.

The data describing the evolution of a system is stored in a VCS. Different VCSs might store the changes in different ways. One system could for example only keep track
of the changes made (e.g. the additions and removal of files and lines), while another VCS could store each individual version. This paper abstracts away from individual VCSs and assumes concepts present in modern VCSs. The concepts are explained in Section 3.1.

The goal of the EM is to form an abstract layer between a research and a combination of programming languages and VCSs to make repository analysis easier. From this four requirements arise:

1. The EM must be language and VCS independent.
2. The EM must be able to answer typical system evolution related questions.
3. The EM must be generated in reasonable\(^1\) time.
4. The EM must support bigger projects.

To test the capabilities of the EM three example questions, that we regard as typical questions that one might ask about a system, are defined. These questions are:

1. What is the number of classes edited per commit?
2. What is the number of lines of code (LOC) changed per commit per person?
3. Which class, contained in the latest version, was mostly under change?

How the answers of each of the above questions can be extracted from the EM is discussed in Section 3.5.

A proof-of-concept (POC) will be made to prove that it is possible to extract the EM from a VCS and programming language. This POC will focus on the VCS Git\(^2\) and programming language Java\(^3\). The POC will be generated from five smaller systems to answer the above-stated questions. The EM will be generated from a bigger system to test the scalability.

In this paper, the words snapshot and version are used. When using the word snapshot, an actual instance of the source code is intended. When using the word version, a node representing a snapshot is intended.

### 1.1 Structure

In Section 2 existing work will be covered. In Section 3 the definition of the EM will be given. Furthermore, how the answers of each of the above questions can be extracted from the EM is discussed. In Section 4 a possible design for a program that builds the EM and details about the POC are given. In Section 5 the results of the questions are given. This section also elaborates on the results of the stress test. Section 6 gives the conclusion and Section 7 mentions possible future work.

### 2. RELATED WORK

The only related work we found addresses visualizing the evolution of a system. As visualization needs an underlying model, these are to some degree a source of inspiration for the work reported in this paper. Different kinds of tools are available to visualize system evolution based on different techniques. The most common technique used is graphs \([9]\). All tools identified focus on the visualization rather than the modeling.\(^4\)

\(^1\)We believe that 10 seconds is a reasonable time for the identified systems.
\(^2\)https://git-scm.com/
\(^3\)https://www.java.com

#### 2.1 TypeV

Feist et al. \([4]\) presents the tool TypeV. TypeV uses the abstract syntax tree as underlying model. TypeV uses commit-by-commit abstract syntax tree to capture changes in the system. Feist et al. suggests that line changes also give rise to non-trivial code changes and therefore may not be a good indicator for system changes. Feist et al. conclude that although line changes do not represent non-trivial system changes, they do correlate strongly. This tells us that the lines of code changes are a reasonable measure of the actual change of a system.

#### 2.2 Gevol

Gevol \([2]\) uses multiple types of graphs to visualize the evolution of a system. The graphs used are inheritance graphs, call graphs and control-flow graphs. As noted in the paper of Collberg et al. the graphs generated are huge. Their test system Sandmark \([3]\) has a total of 760,201 nodes and 2,216,034 edges generated for the call graphs, a total of 100,722 nodes and 123,145 edges generated for the inheritance graphs and a total of 3,091,105 nodes and 3,294,038 edges generated for the control-flow graphs. From this, we learn that we need to find a type of graph that describes the architecture of a system while not exploding in size. This even gets worse since our model will take every individual commit into consideration while Gevol uses a temporal granularity, by default set to one day.

#### 2.3 Other

Some papers about VCSs have been studied. Rama Rao et al \([8]\) introduce some base concepts about modern day VCSs. Ball et al. \([1]\) discusses information stored in VCSs. We also discuss some papers with less significant impact on our work:

- Gource.io\(^4\) displays systems as animated trees. In this tree, directories appear as branches and files as leaves. Gource.io supports Git, Mercurial, Bazaar and SVN. No paper is written about Gource.io. It is more a ‘fun’ tool to see how systems evolve than an analytic tool.
- Yarn \([5]\) uses an analogy to a yarn ball to visualize changes of a system over time.
- SoftwareNaut \([6]\) uses a module based approach to visualize the evolution of a system.
- SHriMp \([10]\) is a tool used to visualize a single version of a system. This tool is not designed to display the evolution of a system.

### 3. DEFINITION

The EM proposed in this paper consist of two types of graphs and an additional set of edges. The first graph type used describes how each version of the system depends on other versions. This graph will be referred to as the evolution graph. Each node in the evolution graph contains another graph of a different type, which describes the system architecture at a given version. This graph will be referred to as the snapshot graph. Last but not least there are the additional edges. These edges describe the evolution path of the nodes of the snapshot graphs by connecting nodes of different snapshot graphs. These edges will be referred to as the transition edges. These concepts are discussed below in more detail. Before we discuss the concepts of the EM, we first expand some concepts and notations described below.

\(^4\)http://gource.io/
3.1 Evolution Graph

The evolution graph describes the evolution path of a system. When developing a system using a VCS, every change made to the system is logged. Rather than sending each bit changed, most VCSs group changes in a commit. Each commit gives rise to a new version of the system. It should be possible to extract the static code in each version from the different VCSs. From this, the evolution graph can be generated.

Two concepts used by most of the VCSs are branching and merging. Branching creates a new evolution path. Merging is the act of combining two or more evolution paths into a single new one. Note that merging results in a new snapshot, regardless if any of the static code changes.

The evolution model, as they have not (yet) contributed. When analyzing a system using an EM, a branch is identified to extract the data from. A branch forms a linear path with a unique initial version and a unique final version. The versions between these two forms the evolution graph. This has the additional result that it may happen that a system has multiple starting points. However, it can not happen that the system has multiple final versions. It may be the case that one or more branches, from a certain point onwards, are still unmerged and may remain that way. We disregard these partial branches in the evolution model, as they have not (yet) contributed.

From the branches present in the evolution graph, we can define a Branch Path. A Branch Path is a path of consecutive versions with the same label:

- There exist exactly one initial node: \( \exists n \in V^b_p : (n, a, m) \in E^b_p \)
- There exist exactly one final node: \( \exists n \in V^b_p : (n, m, a) \in E^b_p \)
- The total number of edges is equal to the number of nodes minus one: \( |E^b_p| = |V^b_p| - 1 \)

![Evolution Graph](image)

**Figure 1:** A figure showing the evolution graph (right) of the system described (left). As can be seen is that branch D does not contribute to the final version and therefore is not present in the evolution graph.
As consequence from disregarding non-merged branches, it holds that the evolution model has only one final version:

$$\forall v \in V_v^5 : 1 \leq |(a, m, c) \in E_p^5 | n = a \land n = c | \leq 2$$

A consequence that can be derived from the Branch Path is that a version has at most one child version that originated from the same branch. Given \( m \in M \), the following holds:

$$\forall v \in V_s : |\{(a, m, x) \in E_m | x \in V_v\}| \leq 1$$

A special version type can be identified in the evolution graph, namely a merge version. A version is a merge version if the version has at least two incoming evolution edges:

**Concepts and notation 4.** (Merge Version) \( v \) is a merge version if and only if:

$$|(x, m, v) \in E_m | x \in V_v| \geq 2$$

The set of all non-merged versions is defined as follows:

**Concepts and notation 5.** The set of non-merged versions is given by:

$$V_{non} = \{ v | v \in V_v \land |\{(x, m, v) | (x, m, v) \in E_m\}| < 2\}$$

### 3.2 Snapshot Graph

A snapshot graph describes the architecture of single version of the system. The way the EM is defined, the snapshot graph can be any arbitrary type of graph that describes the architecture of a single snapshot. An example is to use the directory structure to create a tree, as is done by Gource.io. Another approach would be to use inheritance or call graphs both used by Gevol.

In this paper we have chosen to use class collaboration graphs (CCG) as described by Myers [7] as snapshot graphs. CCGs model the basis of OO systems. The basis of OO systems evolves around the idea of objects. Objects are instantiated from class definitions. Objects work together to achieve a greater goal. Classes describe how these objects look like and depend on each other. These dependencies can be displayed using a CCG. Two types of dependencies are considered: inheritance and association. Inheritance occurs when a class \( A \) is defined as a subclass of class \( B \). Association occurs when a class \( A \) has a field of class \( B \).

**Definition 2.** (Snapshot Graph) A snapshot graph \( G_s \) is a tuple \((V_s, E_s)\) where

- \( V_s \subseteq N \times L \) is a set of nodes.
- \( E_s \subseteq V_s \times V_s \) is a set of directed edges.

**Concepts and notation 6.**

- \( G_s \) is the snapshot graph of snapshot \( v \).
- The label of a node indicates which class it represents. This can be any arbitrary identifier, as long as it uniquely distinguishes all classes.
- A directed edge from a node representing class \( A \) to a node representing class \( B \) is present if and only if \( A \) is a subclass of \( B \) or \( A \) has a field of type \( B \).

---

**class A { }**

**class B extends A { }**

**class C {**

- A a;
- B b;

**}**

Figure 2: A figure showing the class definitions in Java syntax of \( A \), \( B \) and \( C \) (left) and the corresponding class collaboration graph (right).

### 3.3 Transition Edges

The transition edges describe the evolution path of the nodes of the snapshot graphs by connecting two nodes from different versions with each other. Where the edges of the evolution graph describe how versions of the system depend on one another, the transition edges describe how entities within the different snapshot graphs depend on one another. The definition of the transition edges is given below and in Figure 2 an example of edges is shown. In this Figure 3 an example of edges are given.

**Definition 3.** (Transition Edges) Let \( G_n \) be an evolution graph and for all \( v \in V_v \), let \( G_n^v \) be the snapshot graph of \( v \). The set of transition edges \( E_t \) is a subset of \((\bigcup_{v \in V_v} V_v^s) \times N \times (\bigcup_{v \in V_v} V_v^s)\).

**Concepts and notation 7.**

- If a class is present in two consecutive snapshots, a transition edge is present between the nodes representing this class in the different versions.
- If a class is added in a commit, the node representing this class in the new version has no incoming edge from the originating version.
- If a class is removed in a commit, the node representing this class has no outgoing transition edge towards the new version.
- A transition edge never connects two nodes in the same snapshot.

**Definition 4.** \( P^n \) is the set of consecutive transition edges from \( n \in G_n^v \) then \( P^n \) is defined as the smallest set such that for all \( e \in E_t \):

- if \( e = (v, i, n) \) then \( e \in P^n \).
- if \( e = (v, i, x) \) and \( \exists (x, j, y) \in P^n \) then \( e \in P^n \).

### 3.4 Evolutionary Model

The evolution graph, snapshot graph and transition edges are combined to form the EM. Below the definition of the EM is given. In Figure 3 an example of two snapshots and corresponding EM are shown.

**Definition 5.** (Evolutionary Model) An Evolutionary Model \( G_e \) is a tuple \((G_n, \{G_n^v | v \in V_v\}, E_t)\) where

- \( G_n \) is an evolution graph.
- \( G_n^v \) is a snapshot graph.
- \( E_t \) is a set of transition edges.

The following properties need to hold for a valid EM:

- If a transition edge exist between two versions, then also a evolution edge exist:

$$\forall(a, i, b) \in E_t, a \in V_v^v_1 , b \in V_v^v_2 : \exists(n_1, m, n_2) \in E_e$$
Every version with no parent version, has an empty snapshot graph:
\[
\forall v \in V_c: \exists (a, m, v) \in E_v \rightarrow V_v^* = E_v^* = \emptyset
\]
This constraint is met by adding an artificial version to the versions with no parent versions, as discussed before.

**Concepts and notation 8.** (Incoming transition edges) \( T_{in}^v \) is the set of all incoming transition edges of version \( v \in V_c \):

\[
T_{in}^v = \{(a, i, b) \mid b \in G_v^* \land (a, i, b) \in E_v\}
\]

**Concepts and notation 9.** (Outgoing transition edges) \( T_{out}^v \) is the set of all outgoing transition edges of version \( v \in V_c \):

\[
T_{out}^v = \{(a, i, b) \mid a \in G_v^* \land (a, i, b) \in E_v\}
\]

**Definition 6.** (Question 1) Let \( V_{non} \) be the set containing all non-merge versions. The collection of the number of changed classes per commit is defined as:

\[
\{(a, i, b) \in T_{in}^v \mid i > 0\} \mid v \in V_{non}
\]

**Definition 7.** (Question 2) Given a developer \( m \in M \) the set of the number of lines code changed per commit is defined as:

\[
\{ \sum_{(a, i, b) \in T_{in}^v} i \mid v \in V_{non} \land (f, m, v) \in E_v \}
\]

Which class, contained in the latest version, was mostly under change?

This question is based on the definition of what one regards as most under change. We define a score that can be calculated. The class with the highest score is regarded as the class most under change. The definition of the score used is:

\[
\frac{\text{numberOfChangedVersions}^2}{\text{numberOfPresentVersions}}
\]

This results in a weighted percentual change score. Classes which are changed more often with a lower percentual change get a higher score then classes less often changed with a higher percentual change. The reason for this definition is that we find it more important to find the class which changed 19 out of 100 instances then a class changed 1 out of 5 times.

**Definition 8.** (Question 3) Let \( v \in V_c \) be the final version then the class most under change in the final version is defined as:

\[
n \in V_v^* : \max\left(\frac{\{(a, i, b) \in P^n \mid i > 0\}^2}{|P^n|} \mid |P^n| > 0\right)
\]

**4. IMPLEMENTATION**

This section will discuss the design we have in mind for a system that generates EMs. From this paper, a POC was made.

**4.1 Design**

We believe that a good design enables the real power of the EM. Just like the EM, a certain degree of abstraction will be present. This will result in not having to build things multiple times. For example, one builds an implementation of the EM that supports the VCS Git and programming language Java. Then the implementation of the VCS Mercurial and programming language C++ is added. A good design allows that now the implementation of Mercurial will also work with Java and the implementation of Git with C++.

It is chosen to define two parts of the system: the Miner and the Builder. The Miner is responsible for extracting the data from the VCS. The Builder is responsible for generating the snapshot graphs. The Miner will feed the necessary data from the VCS to the Builder. This data includes file contents, metadata of commits and a list of changed files between two consecutive commits and corresponding LOC changes. The Builder generates from the information retrieved from the Miner a snapshot graph. Furthermore, a Commit is assumed which holds references to its parents.

In the EM proposed, a class collaboration graph is used. The classes that fill this graph are constructed from the files present in a commit. When the name of the class in the definition is not altered, it implies that in both graphs the nodes representing the class have the same identifier. When the name is changed, it is needed that different identifiers are used. These identifiers are used to create the transition edges. This means that transition edges can only exist between two snapshot nodes with the same identifier.
In Algorithm 1 the pseudo-code of the main loop of a program generating an EM is given. The main loop builds the evolution model top-down. This means the last commit made (the final commit) is touched first. The Commits are retrieved from the Miner. Generating a snapshot graph uses both the Miner and the Builder. The Miner is needed at rule numbers 3, 6 and 16. The Builder is used at rule numbers 6 and 16.

Algorithm 1: Algorithm showing how to build the EM. The EM is generated top-down, the latest commit is dealt with first:

```
1 begin
2   EM := empty EM;
3   for Commit in reversed chronological order Commits do
4     if Commit not yet in EM then
5       add Commit to EM;
6       build snapshot graph of Commit;
7       add to EM;
8     end
9     if Commit has no parents then
10    add artificial commit to EM;
11   end
12 else
13    for PCommit in Commit.parents() do
14      add PCommit to EM;
15      build snapshot graph of PCommit;
16      add to EM;
17      add transition edges;
18    end
19 end
20 end
```

Adding a single instance (i.e. a node or edge) to the EM, theoretically, can be done in $O(1)$. Building a snapshot graph of a commit depends on how one implements the Builder. Extracting the data from the contents of the file can be done in $O(s)$, where $s$ is the size of the file. From this, generating the snapshot graph can be done in $O(xd)$, where $x$ is the number of classes and $d$ the average amount of dependencies of the system. Adding the transition edges takes $O(x)$, assuming the number of LOC changes is available and can be extracted in $O(1)$. The main loop ensures that every commit is only calculated once, resulting in an overall time complexity of $O(n \times (s + (s + d) + x))$, where $n$ is the number of commits, $x$ the average amount of dependencies per commit, $s$ the average file size of every source file present in each commit, $d$ the average amount of dependencies per class per commit. This can be simplified to $O(n \times (x^2 + xs))$.

We believe that it is possible to make the generation of the EM more efficient. This can be achieved by only generating the snapshot graphs of the changed classes and copying the non-changed classes from the snapshot graph of the previous version.

### 4.2 Proof-of-Concept

The POC was implemented to show that it is possible to extract the EM from systems. The POC did not implement the design proposed in Section 4.1, due to the fact that this design was thought of after implementing the POC. It did follow the main loop described in Algorithm 1. The POC was written in Java.

The problem the POC faced, is how to extract dependency data from class definitions. Two possible solutions were identified: parsing the class files or use regular expressions. Parsing the files has the benefit that the solution is 100% accurate, whereas regular expressions only give an approximation. The benefit of regular expressions is that it is way faster than parsing the files. We believe that regular expressions give a good enough approximation.

In order to extract the needed data first, the string declarations and comments are removed to ensure no false dependencies arise from them. From what is left, the dependencies are extracted. From this extraction, an issue arises. It can not always be discovered where a certain class dependency originates from. For example, when a class uses a class located in the same package, it is not possible using only string matching in one class to find this out. Another thing is star imports (imports containing a `*`). To overcome these problems, after extracting all the necessary data, all the data is put together to figure out where certain classes originated from. It might happen that sometimes a class location can not be derived. An example of such a case is when a class uses multiple star imports and no other class in the system uses any of those classes. As far as we are aware, no instances of this occurred in analyzed projects.

### 5. RESULTS

In this section, the results of the stated question in the introduction will be answered. The results of question "What is the number of classes edited per commit?" will be discussed in Section 5.1, the results of "What is the number of LOC changed per commit per person?" will be discussed in Section 5.2 and the results of "Which class, contained in the latest version, was mostly under change?" will be discussed in Section 5.3. The results of the generation of the EM from the bigger system are discussed in Section 5.4.

The answers to the above-stated questions are extracted from EMs. The EMs are generated from five relative small scale systems. These systems are from the ING honors program. The ING honors program is an initiative of the Dutch bank ING. In this program, students are given the task to implement a basic banking system. Over time requirements changes follow to which the students need to adapt their system. The goal of this program is to create a testbed for research. The main research for which these systems are made is for predicting the amount of work a requirement change will bring. Each system is made in a team of two students. Each team is denoted by a letter. Teams analyzed are: A\textsuperscript{0}, B\textsuperscript{0}, C\textsuperscript{0}, D\textsuperscript{0} and E\textsuperscript{0}. Some metrics of each system’s final version are displayed in Table 1 to give in indication of the size of each system.

5.4. Some metrics of each system’s final version are displayed in Table 1 to give in indication of the size of each system.

<table>
<thead>
<tr>
<th>Team</th>
<th>Size of File</th>
<th>LOC Changed</th>
<th>Classes Edited</th>
</tr>
</thead>
<tbody>
<tr>
<td>A\textsuperscript{0}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B\textsuperscript{0}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C\textsuperscript{0}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D\textsuperscript{0}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E\textsuperscript{0}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://github.com/FHast/INGhonours_GereonFritz
https://github.com/tristandb/springbank-spring
Table 1: Table showing metrics of the system's final version. The metrics are: the number of written classes (written), the number of used classes (used), the number of incoming dependencies of written classes (in), the number of outgoing dependencies of written classes (out) and the average LOC (lines). By written we mean something that is made by the developers of the system.

<table>
<thead>
<tr>
<th>Team</th>
<th>written</th>
<th>used</th>
<th>in</th>
<th>out</th>
<th>lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>44</td>
<td>68</td>
<td>83</td>
<td>208</td>
<td>181</td>
</tr>
<tr>
<td>B</td>
<td>18</td>
<td>25</td>
<td>28</td>
<td>49</td>
<td>70</td>
</tr>
<tr>
<td>C</td>
<td>22</td>
<td>45</td>
<td>31</td>
<td>82</td>
<td>131</td>
</tr>
<tr>
<td>D</td>
<td>32</td>
<td>52</td>
<td>3</td>
<td>51</td>
<td>105</td>
</tr>
<tr>
<td>E</td>
<td>34</td>
<td>48</td>
<td>54</td>
<td>103</td>
<td>88</td>
</tr>
</tbody>
</table>

The EMs were generated on a laptop with an i7-4700MQ 2.4GHz and a 100Mbps WIFI connection. The times it took to download the projects and generate the EMs from are displayed in Table 2. From the generated EMs, the answers of the questions were extracted. This was done by implementing three scripts that adapted the definition of each question. The scripts were written in the same system as the POC and took as parameter the generated EM.

It is chosen to only use the main Java source folder to answer the questions. Any folder outside this one is disregarded. The goal of this is to disregard the effects of testing. One could argue that test classes should also be included in the research, but we believe that tests only affect the evolution of a system indirectly. Another thing that we took into consideration was that some teams were further in the development of the banking system than others. It is chosen to only look at the history of the tagged version marked as version 1. Furthermore, it is known that every system only had two developers. During analysis of the projects, it came to notice that some teams had used multiple Git accounts. This results in different developer labels in the evolution graph. It is assumed that if large parts of the name corresponded, it belonged to the same developer. Adding this data to a single set was done manually.

Table 2: Table showing the time needed to clone the project and the time needed to generate the EM from the project.

<table>
<thead>
<tr>
<th>Team</th>
<th>cloning</th>
<th>generating</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6.95s</td>
<td>10.46s</td>
</tr>
<tr>
<td>B</td>
<td>2.36s</td>
<td>0.30s</td>
</tr>
<tr>
<td>C</td>
<td>4.56s</td>
<td>3.67s</td>
</tr>
<tr>
<td>D</td>
<td>8.53s</td>
<td>0.81s</td>
</tr>
<tr>
<td>E</td>
<td>2.44s</td>
<td>1.67s</td>
</tr>
</tbody>
</table>

5.1 Changed Classes

The results of the first questions are displayed in Figure 4. The figure displays graphs which indicate how often a commit changing a certain number of classes occurred. It can be seen that the number of commits changing 0 classes is most common at most teams. A commit changing zero classes appears when one does a commit where the source code inspected is not affected. This can be either files used by the system (i.e. HTML and XML) or a commit changing test code. It can also be that the source code is altered. This can be in the form of adding or deleting a class. It is noticeable that team A and E have a high number of commits where only a single class is edited.

When looking to the distribution of the quantity of the number of classes, the distribution looks like an inversely proportional one. The dataset is too small to confirm this.

Figure 4: These figures show on the x-axis the number of classes changed and on the y-axis the number of occurrences. Note that every dot in the graph has an occurrence of at least one.

5.2 Changed LOC

The results of the second question are displayed in Figure 5. The figure shows the graphs displaying the number of commits changing a certain number of lines of code. When considering only the number of LOC one has changed to the system, one can conclude that in every team the work is mostly done by a single developer.

When looking to distribution, especially developers 2 from team A and C, it looks like an inversely proportional distribution. Again the dataset is too small to confirm this hypothesis.

5.3 Most Changed Class

The results of the third question are displayed in Table 3. The scores and corresponding data to calculate the scores are shown in Table 4. The scores are calculated using the formula described in the introduction. An interesting case is the class ui.UIService from team A. This class was present from the beginning of the system. In the latest version present, the class consists of 931 LOC (including whitespaces and comments), 1 incoming dependency and 13 outgoing dependencies. This is a big class compared to the averages of the system. An average class of team B has 4.7 outgoing dependencies and 181 lines of code. This may be an indication that the class should be decomposed (if possible) into smaller classes. An interesting question following from this small research if there is a correlation between one of the metrics and the change score.

After studying the actual code of the class, we would indeed suggest, out of own programming experience, to decompose the class into smaller classes. This is mainly based on the fact that the class handles things which are in our opinion not that closely related to one another.
5.4 Stress Test

The bigger system chosen to generate the EM from is MyCollab\footnote{https://github.com/MyCollab/mycollab}. MyCollab is an open-source project management tool. It consists out of 721 commits and in its latest version 2238 written classes are present.

It took around 1 minute to clone the project and after around half an hour the POC ran out of memory. A small size investigation took place to find out more details. We found out that roughly every 4 minutes 50 commits were finished. This means that if the POC continued this rate, the entire generation of the EM would take around an hour. At roughly 400 commits the POC ran out of memory.

After analyzing the POC with the Java profiler VisualVM\footnote{https://visualvm.github.io/} we found out that by far most of the memory was used by HashMaps. We used HashMaps to guarantee an $O(1)$ complexity in requesting data from the EM. We also used a lot of HashSets, which internally uses HashMaps. We believe that this problem can be overcome by using something else then HashSets and HashMaps or limit their usage. Another option is to calculate the EM in parts.

6. CONCLUSION

This paper has introduced a model, called the Evolutionary Model (EM). This model acts as an abstract layer between a system evolution research and the combination of programming languages and VCSs. A proof-of-concept (POC) was made to verify the requirements stated in the introduction. The stated requirements were: The EM must be language and VCS independent, the EM must be able to answer typical system evolution related questions, the EM must be generated in reasonable time and the EM must support bigger projects.

The first requirement states that the EM must be language and VCS independent. Due to the usage of a class collaboration graph to describe the architecture of a single instance of a system, the EM supports every object-oriented language. This is because a class collaboration graph relies on the base principles of object-oriented languages. It still needs to be proven if it is possible to extract the EM from any modern day VCS. Base concepts used by most VCSs were used when designing the EM. This is why we...
are quite certain that this model can be generated from other modern VCSs, but proof is needed.

The POC could answer the typical evolution related questions mentioned in the introduction. The results of these questions are discussed in Section 5.

The POC shows that it is indeed possible to extract the EM for smaller systems from a VCS within a reasonable time. All EMs for the smaller systems but one were generated within the aimed goal of 10 seconds. The one that was not generated in 10 seconds was generated in 10.46 seconds. We believe that the efficiency of generating the EM can be greatly increased. An idea we have is that most of the classes will be left unchanged in a commit. This idea is confirmed by the results discussed in Section 5.1. This means that not every class has to be generated again, but only the ones changed.

The final requirement for the EM was that it is possible to generate it from bigger systems. The EM could not be generated from the bigger system identified. This result may sound as that the EM is not viable for bigger systems but we believe that the problem lies in our implementation and does not mean that generation is not possible. The reason why the EM could not be generated is explained in Section 5.4.

The last note we want to make is that the POC did not implement the fully defined EM. Branch labels on evolution nodes were not implemented. This was done due to time constraints and the fact that it was not needed for answering the questions stated in the introduction. We believe however that it is possible to extract this data from a VCS, but proof is needed.

7. FUTURE WORK

A lot of future work can arise from this paper. The first big step needed to take is to actually implement the EM in a proper manner. This includes implementing multiple VCSs and multiple programming languages. From this, the EM can be a base of various other evolution-related research. A big step that would aid in the adaptation of the EM is a domain specific language (DSL). A DSL will allow researchers to easily define new research questions that can be directly applied to the EM.

A research that seems very interesting is to find a correlation between the number of times a class changed and various characteristics of a class. From this research, it would be an interesting question if one could predict if certain classes will change in the future or propose new programming strategies. This research can be conducted using the EM.

The EM also gives the opportunity to analyze different programming behaviors. Is it, for example, common if one commits a lot of lines at once, or only a few? Does this differ per person? It can be also analyzed if a programmer uses allot of decomposition or few. It would be interesting to do a study among multiple groups and compare the results. Is there a notable difference in programming style between high school students and undergraduate?

Another thing that can be researched is that the division of classes in folders correlates to strongly and weakly connected components in the snapshot graph. If this does not correlate, one might discuss which of the two divisions is better.

Besides the directions of future research discussed above, one can also extend the EM. For example, add a label indicating inheritance or association to the edges of the snapshot graph. Even the entire snapshot graph as it is defined in this paper can be swapped out by another type of graph. Research can be done to make the EM also available to other programming paradigms.

8. ACKNOWLEDGEMENT

We would like to thank Arend Rensink for useful conversations and for giving feedback on this paper, Sven Konings and Rogier Monshouwer for reviewing this paper and supplying constructive feedback.

9. REFERENCES