Automatic Change Recommendation using Decision Trees

Bachelor Thesis Proposal

Valentin Reyes Häusler

April 8, 2018

1 Introduction and Motivation

- Design phase is a core step in any software development process
- Objective of design phase: creation of design for architecture, software components, interfaces and data (IEEE-Standard "design phase")
- This is accomplished by creating models and meta-models describing the solution precisely and completely while maintaining a necessary level of abstraction. (Skript 07.1)
- For example: In object-oriented modeling class diagrams are indispensable. **??** shows classes and their relationships such as dependencies and associations.
- Importance of models: Recognition of patterns and redundancies allowing simplification (creation of a super class) ... (TODO: more examples)
- It is clear that a precise and complete design serves as a stable base for an efficient implementation.
- Tools for the latter phase are vastly available. They range from code completion and action recommendation to generation of complete blocks of code.
- This allows the engineer to concentrate on the core tasks of the implementation phase.
- Although the design phase is equally important, comparable tools are often lacking or non existent.
- A possible improvement in this aspect are automatic action proposals for the creation of models.
- Basic to semi-complex actions such as the creation of a super class, separation of packages and so on, could be proposed to the designer.
- Such a tool would again allow the engineer to concentrate on the core tasks of a consistent and complete design.
- This thesis aims to create and evaluate such a tool.
- We intend to do this by analyzing the versionized histories of models
- Using machine learning algorithms and data structures we look for reoccurring patterns
- The patterns are then matched to a model which is being worked on.
- If a fit is found a recommendation can be done.
- Example for class diagrams:

Pattern: State: Two classes X and Y share some attributes and functions
Actions:

- Create superclass Z for X and Y
- Move shared attributes and functions from Y to Z

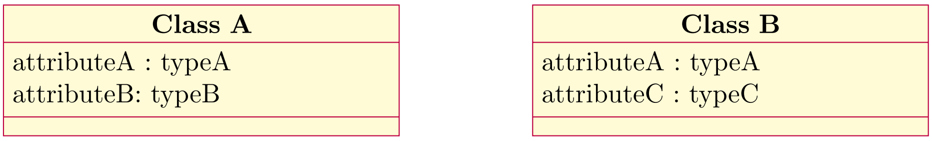


Figure 1: Class diagram. Class A and class B share attributeA.

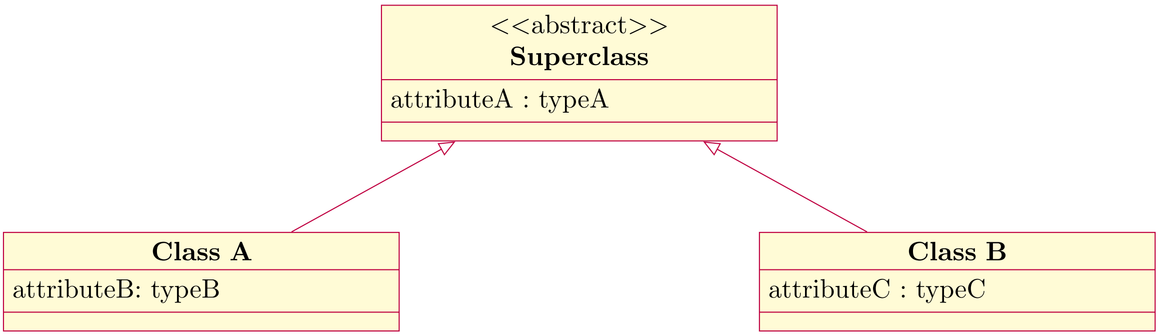


Figure 2: attributeA is "pulled up" from class A and B to Superclass.

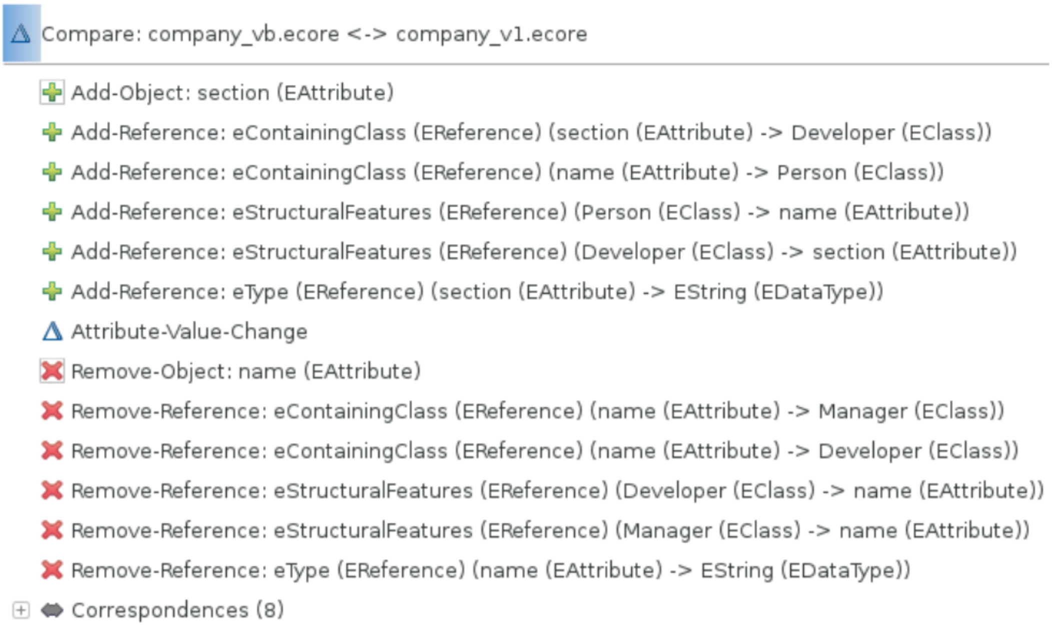
- Delete the same attributes and functions from X

Abstraction: Pull up shared attributes/functions to superclass

2 Problem Statement

- SiLift is able to compare different versions of a model
- It recognizes *low-level* actions and dependencies between those actions.
- Low-level actions are simple internal operations for the deletion and creation of objects and their references.
- Figure 3 shows the low-level differences between two versions of the model *company*.
- User-level actions consist of various low-level actions.
- For example: Creating a new string-attribute *section* in the class *Developer* consists of the low-level actions:

1. Create attribute: section
2. Add containingClass-reference from section to Developer
3. Add structuralFeatures-reference from Developer to section
4. Add type-reference from section to String
5. Change the value of section.



Compare: company_vb.ecore <-> company_v1.ecore

- ✚ Add-Object: section (EAttribute)
- ✚ Add-Reference: eContainingClass (EReference) (section (EAttribute) -> Developer (EClass))
- ✚ Add-Reference: eContainingClass (EReference) (name (EAttribute) -> Person (EClass))
- ✚ Add-Reference: eStructuralFeatures (EReference) (Person (EClass) -> name (EAttribute))
- ✚ Add-Reference: eStructuralFeatures (EReference) (Developer (EClass) -> section (EAttribute))
- ✚ Add-Reference: eType (EReference) (section (EAttribute) -> EString (EDataType))
- △ Attribute-Value-Change
- ✖ Remove-Object: name (EAttribute)
- ✖ Remove-Reference: eContainingClass (EReference) (name (EAttribute) -> Manager (EClass))
- ✖ Remove-Reference: eContainingClass (EReference) (name (EAttribute) -> Developer (EClass))
- ✖ Remove-Reference: eStructuralFeatures (EReference) (Developer (EClass) -> name (EAttribute))
- ✖ Remove-Reference: eStructuralFeatures (EReference) (Manager (EClass) -> name (EAttribute))
- ✖ Remove-Reference: eType (EReference) (name (EAttribute) -> EString (EDataType))

⊞ Correspondences (8)

Figure 3: Low-level differences between two models as recognized by SiLift

- This example can be seen in 3
- Abstracting the individual actions we can create a *recognition rule* for the user-level action INSERT <type><attr>=<val>IN <class>:

1. Create attribute: <attr>
2. Add containingClass-reference from <attr>to <class>
3. Add structuralFeatures-reference from <class>to <attr>
4. Add type-reference from <attr>to <type>
5. Change value of <attr>to <val>

- SiLift uses these recognition rules and *lifts* matching sets of low-level actions to user-level actions.

- We aim to automatically generate recognition rules from a model's history.
- The initial data is given in from of a dependency graph.
  - Nodes represent actions; edges dependencies.
- Figure 4 represents a dependency graph.
- Note that since not every action has a dependency there can be multiple separated subgraphs and nodes without parents.
- We call this nodes orphan-nodes.
- Generating the recognition rule for *INSERT <type><attr>=<val>IN <class>* we applied an abstraction to the low-level actions.

-The abstraction consists of mapping the arguments of an action to variables.
- For example: Create attribute: section becomes Create attribute: <attr0>with <attr0>= section.

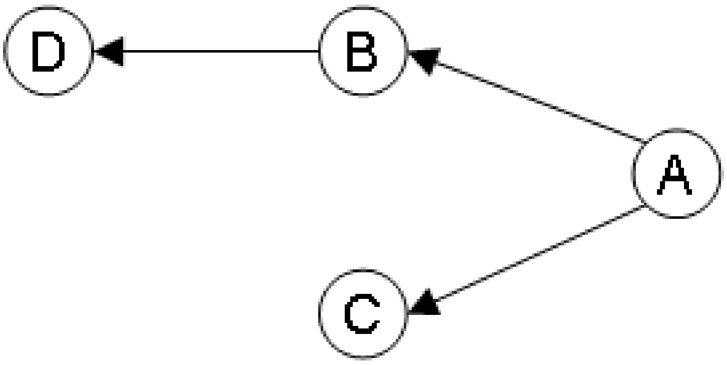


Figure 4: Dependency graph. A has to happen before B and C; B has to happen before D.

- With this abstraction two actions can be compared by simply comparing their abstracted version with one another.
- When comparing action sequences checking whether the mapping and using of variables are consistent with each other.
- If a variable is mapped to an object then all actions using that object must use the variable in the abstracted action sequence.
- Now we are able to find recurring implementations of the same abstract user-level action.
- We do this with following algorithm: - The construction of the data-structure automatically calculates

```
Input: Dependency graph G
Output: Decision trees for recommendation
foreach Orphan Node o do
    Create abstract version of action a;
    Find tree t with a at it's root;
    if No such tree exists then
        Create one
    end
    With Breadth-First-Search:
    foreach Path p reachable from o in G do
        if p does not exist in t starting at the root then
            Extend the longest matching path in t starting at the root to p;
        end
        Reinforce the probability of the path in t;
    end
end
```

the absolute number of occurrences of any path and with it the relative probability of it's occurrence. This allows us to match the last actions done by the user to the decision trees.
If a match is found the next nodes in the tree can be recommended according to their probability of occurrence.

TODO: Example of bigger dependency graph and transformation to decision trees.

3 Purpose of the study

The purpose of this study is to create a tool for change proposal during the creation of models and meta models. Such a tool, if efficient, could provide support for engineers during the design phase of software comparable to autocompletion tools used during the implementation phase.

4 Review of the literature

Programming:
Learning haskell data analysis
Author: James Church
Summary: The book gives an overview over the steps for data analysis. Practical examples with Haskell are given for every step. Significance: The methods for parsing and manipulating data taught in this book are applicable to the challenges of this thesis.

Learn You a Haskell for Great Good!: A Beginner's Guide
Author: Mirian Lipovaca
Summary: Ample overview over Haskell's functionality.
Significance: This book builds a stable base for learning Haskell.

Foundation:
Understanding Model Evolution through Semantically Lifting Model Differences with SiLift
Authors: Timo Kehrer, Udo Kelter, Manuel Ohrndorf, Tim Sollbach Summary: This paper presents SiLift. SiLift is able to lift low-level differences of models into user-level actions.

Automatic Change Recommendation of Models and Meta Models Based on Change Histories
Authors: Stefan Kögel, Raffaela Groner, and Matthias Tichy
Summary: Alternative aproach ... TODO
Significance: The results presented in this paper are comparable to our results. It would be interesting to compare effectiveness of both tools.

5 Research questions and/or Hypotheses

RQ1: How much data is necessesary for the tool to perform relatively well?

RQ2: Are there cases in which the tool performs better or worse?

RQ3: How does this tool compare to other implementations?

RQ4: Is it possible to use this aproach for other applications?

RQ5: If so, how has the data to be processed to the aproach to be applicable?

6 The Design - Methods and Procedures

We will follow Hevner's guidelines for design science research:

Design as an artifact: The aim of this paper is to produce an autorecommendation tool.

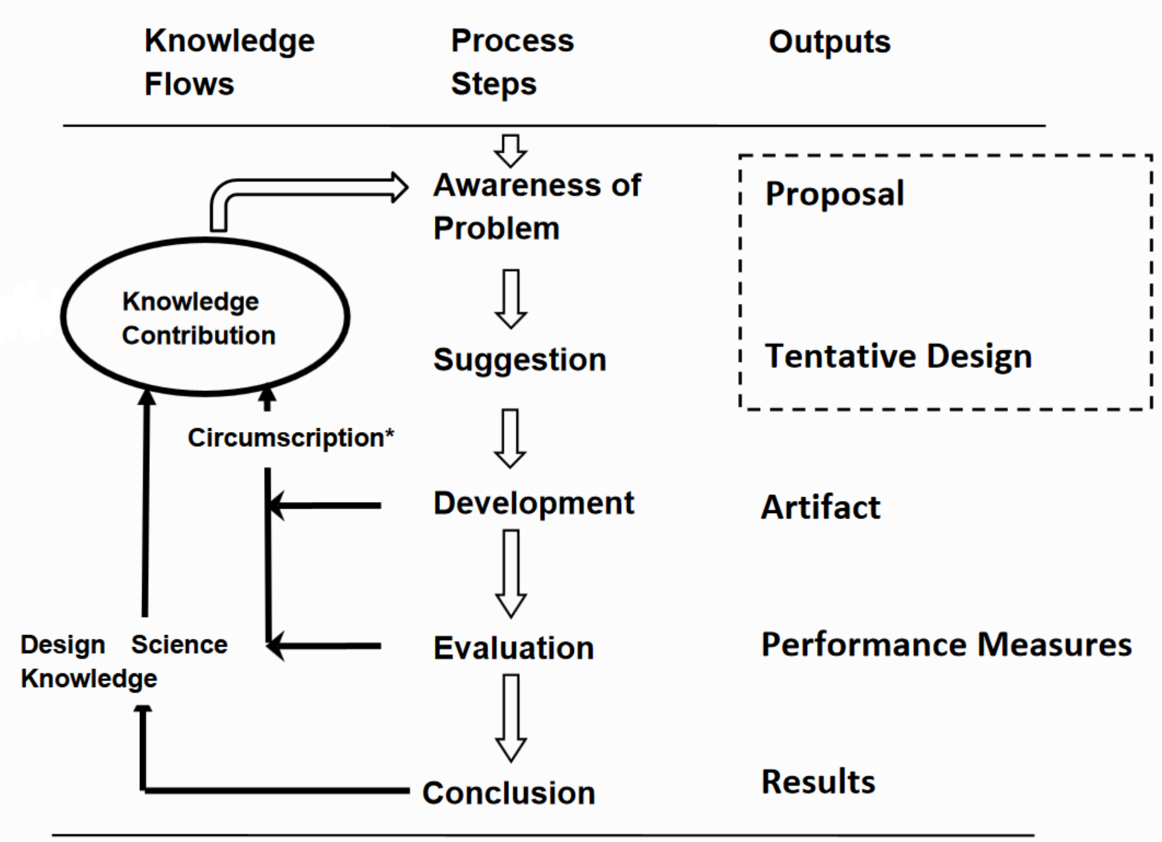


Figure 5: Design Science Research Process Model

Problem relevance: Comparable tools are often lacking or non-existent.

Design evaluation: This tool will be tested in it's performance and correctness. The results will be compared with comparable tools.

Research contribution: We strongly believe that this tool will provide considerable support to engineers.

Research rigor: The methods learned in various lectures will be applied to the construction and evaluation of this tool.

Design as a search process: The resources and methods made available through the university, such as the library, professors and lectures, will be used for the search of an effective solution.

Communication of research: Unknown concepts will be explained and sources for further explanations will be provided.

7 Limitations and Delimitations

The goal of this study is to create an effective and helpful change recommendation tool to aid during the creation of models. We will still aim to keep the tool as broad as possible so that it can be used for other application in which a change history is available.

Although a user interface and plugin for different model creation software are certainly imaginable, we won't try to archive those goals. Instead a command line interface will be implemented.

8 Significance of the study

The significance of this study lies within its usability and helpfulness. Streamlining the process of model creation aids to increase the quality of software and its design. The ability to bypass mindless repetition during the creation of a model helps the engineer to focus on the important aspects of that model.

The creation of this study will also help in deepening my knowledge in functional programming and machine learning. Both of which serve as useful tools for any programmer. With the former being an additional paradigm with which to program and the latter being a relevant topic in almost every aspect of computer science.

9 Planning

9.1 Own Background

My knowledge in functional programming, while lean, serves as a good start for the creation of this study. Especially when supported by my background with lectures such as "Programmierung von Systemen" and "Algorithmen und Datenstrukturen" as a student as well as a tutor. Both of which are highly applicable in this study. My participation in "Sopra 16/17" provides me with experience for the designing, implementation and testing of the tool which we aim to create.

Having said that, extensive practice in practical functional programming will be necessary as well as attaining knowledge in machine learning, especially regression algorithms. My successful completion of the lecture "Einführung in die künstliche Intelligenz" and the proseminar "Algorithmen" will hopefully provide a strong base for that.

9.2 Required Resources

No especial resources will be necessary. Only access to data in form of model histories and the tool SiLift should suffice for the creation of this study.

9.3 Work packages

M1 Extensive practice in Haskell, especially data-analysis. Research into decision trees and other applicable data-structures.

M2 Analysis and first design of a solution.

M3-4 Multiple iterations of deign - implementation - review.

M5 Final implementation. Testing of tool, comparison with similar tools.

M6 Collection of results on paper, answering all research questions.

9.4 Contingency plan

1. Failing to create such a tool with Haskell due to lacking knowledge. This eventuality will be discussed with the creators of the study.

References

[1] (). Eclipse modeling framework (emf) - tutorial, [Online]. Available: `http://www.vogella.com/tutorials/EclipseEMF/article.html`.

[2] J. Church, *Learning Haskell Data*, ser. Community experience distilled. Packt Publishing, 2015, ISBN: 9781784394707. [Online]. Available: `https://books.google.de/books?id=9vPXsgEACAAJ`.

[3] M. O.T. S. Timo Kehrer Udo Kelter, "Understanding model evolution through semanticallylifting model differences with silift," Software Maintenance (ICSM).

[4] M. T. Stefan Kögel Raffaela Groner, "Automatic change recommendation of models and meta models based on change histories," Institute of Software Engineering and Programming Languages, Ulm University.