JOHANNES KEPLER
UNIVERSITY LINZ

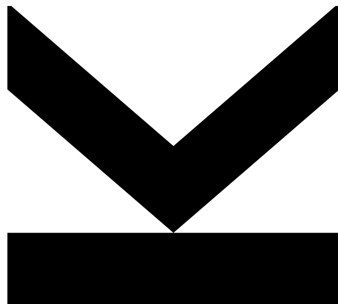
Submitted by
Hannes Thaller

Submitted at
Institute for Software Systems Engineering

Supervisor
Univ.-Prof. Dr. Alexander Egyed M. Sc.

10.2016

Towards Deep Learning Driven Design Pattern Detection



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Towards Deep Learning Driven Design Pattern Detection

Detecting Design Patterns by Convolving Object Oriented Properties

Hannes Thaller

Abstract

Design patterns are elegant and well tested solutions to recurrent software development problems. Their extensive use in every day programming weaves valuable architectural information into software systems. Despite the wide usage of design patterns, system documentations seldom contain information about their existence. This work presents a fully fledged approach to extract design patterns such that the lost information can be of value for architects, developers and maintainers. It includes the common design pattern detection steps that extract features, sample candidates from the system under inspection and infer whether the candidates are of a certain pattern or not. The approach incorporates the usage of object oriented properties in form of micro-structures that are projected onto feature maps. These feature maps are then analyzed by a convolutional neural network that extracts high-level features from which robust prediction results can be drawn. Results indicate that deep learning methods bare great potential for the design pattern community as reliable inference procedure.

Towards Deep Learning Driven Design Pattern Detection

Detecting Design Patterns by Convolving Object Oriented Properties

Hannes Thaller

Kurzfassung

Entwurfsmuster sind elegante und vielfach erprobte Lösungen für wiederkehrende Softwareentwicklungsprobleme. Ihr häufiger Gebrauch in der täglichen Programmierung bettet wertvolle Informationen bezüglich der Applikationsarchitektur in den Sourcecode. Trotz der breiten Nutzung von Entwurfsmustern ist ihre Existenz und Implementierung nur selten dokumentiert. Diese Diplomarbeit stellt einen vollwertigen Ansatz zur extrahierung von Entwurfsmuster vor, sodass Architekten, Entwickler und Maintainer von den eingebetteten Informationen profitieren können. Es werden Lösungen für die typischen Phasen der Entwurfsmustererkunnung vorgestellt wie das extrahieren von Features, das finden von potentiellen Entwurfsmusterinstanzen und der Prozess zur Entwurfsmusterklassifizierung. Der präsentierte Ansatz verwendet typische Eigenschaften der objektorientierten Programming in Form von micro-structures welche auf feature maps projeziert werden. Diese werden dann mittels eines Convolutional Neural Networks analysiert sodass eine robuste Klassifizierung erfolgen kann. Die Ergebnisse zeigen das Deep Learning ein große Potenzial für die Entwurfsmustererkennung hat und sich somit als zuverlässige Klassifizierungsmethod eignet.

Acknowledgements

The research reported in this thesis has been partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

Contents

List of Figures 3

List of Tables 5

1 Introduction 9

1.1 DPD Basics . . . 10

1.1.1 Research Question . . . 12

1.2 System Overview . . . 13

1.2.1 Detected Design Patterns . . . 14

2 Feature Extraction 19

2.1 Related Work . . . 19

2.1.1 Micro-Architectures . . . 20

2.2 MS Features . . . 22

2.2.1 Technical Overview . . . 23

2.2.2 Micro-Structure Catalog . . . 24

3 Candidate Sampling 43

3.1 Heuristic Search . . . 45

3.1.1 Adapter Sampling . . . 46

3.1.2 Composite Sampling . . . 46

3.1.3 Decorator Sampling . . . 47

3.1.4 Factory Method Sampling . . . 47

3.1.5 Singleton Sampling . . . 48

3.1.6 Template Method Sampling . . . 49

3.1.7 Technical Overview . . . 49

3.2 Evaluation . . . 51

3.2.1 Discussion of Noteworthy Instances . . . 52

4 Feature Normalization 57

4.1 FRN . . . 58

4.1.1 Issues . . . 59

4.1.2 Properties of the Feature-Role Normalization . . . 61

4.1.3 Alternative Mapping Schemes . . . 65

5 Design Pattern Detection 71

5.1 Machine Learning . . . 71

5.1.1 Data . . . 73
5.1.2 Preprocessing . . . 74
5.1.3 Feature Selection . . . 75
5.1.4 Model Class Selection . . . 76
5.1.5 Model Training . . . 83
5.1.6 Model Evaluation . . . 85
5.2 DPD via CNN . . . 89
5.2.1 Data . . . 90
5.2.2 Preprocessing . . . 93
5.2.3 Feature Selection . . . 94
5.2.4 Model Selection . . . 95
5.2.5 Model Training . . . 100
5.2.6 Model Evaluation . . . 102

6 Conclusion **113**

Bibliography **115**

A Training Parameters **121**

List of Figures

Figure 1.1 Basic detection process. 12
Figure 1.2 System overview 14
Figure 1.3 Adapter Pattern 15
Figure 1.4 Composite Pattern 16
Figure 1.5 Decorator Pattern 17
Figure 1.6 Factory Method Pattern 17
Figure 1.7 Template Method Pattern 18

Figure 2.1 Feature extraction process 19
Figure 2.2 Micro-structures example 22
Figure 2.3 Feature extraction module 24

Figure 3.1 Candidate sampling process 43
Figure 3.2 Adapter Sampler 46
Figure 3.3 Composite Sampler 47
Figure 3.4 Decorator Sampler 48
Figure 3.5 Factory Method Sampler 49
Figure 3.6 Template Method Sampler 50
Figure 3.7 Candidate sampling technical overivew 50
Figure 3.8 Search space accross projects 53
Figure 3.9 Search space accross patterns 54

Figure 4.1 Feature normalization process 57
Figure 4.2 Feature-role normalization mapping sheme. 58
Figure 4.3 Feature Role Normalization: Issue 1 60
Figure 4.4 Feature-Role Normalization: Issue 2 61
Figure 4.5 Virtual roles prevalence analysis 62
Figure 4.6 Heatmap of a feature map 63
Figure 4.7 Normalization data structure comparison 69

Figure 5.1 Design pattern detection process 71
Figure 5.2 Data analysis workflow 72
Figure 5.3 Example for linear model 78
Figure 5.4 Example for multiple linear models 79
Figure 5.5 Neural network 80
Figure 5.6 Convolution example 82
Figure 5.7 Max pooling 83

Figure 5.8 Training Process 84
Figure 5.9 Feature Importance for the Adapter and Composite roles 96
Figure 5.10 Feature Importance for the Decorator and Singleton roles 97
Figure 5.11 Feature Importance for the Factory Method and Template Method roles . 98
Figure 5.12 Cross-validated bagged learning 102
Figure 5.13 ROC/PR - Adapter 104
Figure 5.14 Adapter - Decision Boundary 105
Figure 5.15 ROC/PR - Composite 106
Figure 5.16 Composite - Decision Boundary 107
Figure 5.17 ROC/PR - Decorator 108
Figure 5.18 Decorator - Decision Boundary 109
Figure 5.19 ROC/PR - Factory Method 109
Figure 5.20 Factory Method - Decision Boundary 110
Figure 5.21 ROC/PR - Singleton 110
Figure 5.22 Singleton - Decision Boundary 111
Figure 5.23 ROC/PR - Template Method 111
Figure 5.24 Template Method - Decision Boundary 112

List of Tables

3.1 Naive Space and H.Space comparison 51

4.1 Sparness Statistics of Feature Maps 64
4.2 Role-Role Normalization Example 66
4.3 Feature-Channel Normalization Example 68

5.1 Dog example dataset 73
5.2 Confusion matrix 86
5.3 MMC interpretation 88
5.4 P-MARt Project Overview 92
5.5 P-MARt Pattern Overview 92
5.6 Class label proportion of the patterns 93
5.7 Basic network topology 100
5.8 Detection performance 103

A.1 Adapter parameters 122
A.2 Composite parameters 123
A.3 Decorator parameters 124
A.4 Factory Method parameters 125
A.5 Singleton parameters 126
A.6 Template Method parameters 127

List of Listings

Listing 3.1 Node figure fragment 55
Listing 3.2 West handle 56

1. Introduction

Design patterns (DPs) are elegant and well tested solutions to recurrent software development problems. *Design Patterns – Elements of Reusable Object Oriented Software*, written by Gamma et. al [15], is the most well known collection of patterns and inspiration for many follow ups. They are the result of software developers dealing with problems that occur frequently, solving them in the same or a slightly adapted way. Design Patterns are the generalized version of the different adapted implementations such that they can be reused and applied over and over again in different situations. Patterns have a name that identify them and help during the communication needed in the development process. Usually they solve some higher level Object Oriented (OO) architectural problem dealing with creation, structure or behavior of a small set of classes or objects. Some of the problems are related to deficiencies in OO languages and thus only occur on certain technologies, but most solve the problem of inflexible design during development of non trivial software systems. The solution that the pattern provides is described in an abstract fashion such that it can act like a template solution in many different situations. Class arrangements are defined rather loosely with a detailed description and examples of implementations to reduce the learning curve and to provide aid during the actual realization. Despite their obvious advantage of solving and providing a well known and applied architectural solution to recurrent problems, design patterns also impair consequences related to their specific solution that need to be considered. These trade-offs are usually associated with space or time demands in exchange for more flexibility in the design, such that the evolution of the system can be done more fluently. Additionally certain patterns add a rather high entrance level for junior developers, nonetheless because of their good documentation, unfamiliar developers may easily study them during their development activities.

Pattern descriptions are very detailed and contain their name, intent, motivation, where they are applicable, structures, participants, collaborations, and so forth. The patterns semantic is given by the intent, motivation and applicability, which describes what the pattern does, why the pattern is needed, and where it is useful. Participants reflect the template nature of patterns as they are roles that classes can adopt and, structure and collaborations describes how these roles interact. A well of information is encoded in design patterns and their descriptions, and developers weave this information into their system during the implementation activities. The usage of a pattern is related to some specific design decision during development and often these decisions and their rationals are not documented throughout the process. Also the usage of a pattern is usually not documented, thus decision, rational and their materialization in form of the pattern's implementation are lost in the system's source code. Retrieving this encoded information such that development,

redevelopment and maintenance can profit from it is the main motivation of Design Pattern Detection (DPD).

Software engineers can profit from DPD by analyzing existing similar projects by looking for performance bottlenecks, evolutionary dead ends and frequently extended modules such that they can make an informed decision during their development activities. They may learn from past architectural failures or successes enabling them to sharpen their skills and tailor their decisions for new systems. In running systems pattern information helps the evolution of submodules by reusing existing structures and by controlled and complementary extensions to the architecture. In addition to the analysis aspects, DPD might enable several automation technologies once it reaches a mature state. For instance automatic test generation for pattern participants and their collaboration would yield more robust software systems as a set of well known and through thought test suits could be generated for each situation. By defining smaller and more general patterns, DPD could enhance static and dynamic code analysis by providing guidelines and heuristics to the developers during programming directly into their Integrated Development Environment (IDE). These heuristics could point out possible failures, bad practices and how to resolve them. Additionally, extending DPD to the domain of software models would enable the software engineers to avoid typical architectural missteps and to simulate the subsystems behavior and performance based on empirical data retrieved by previously analyzed systems. The ultimate goal of DPD is to define and detect a well defined catalog of micro-architectures on which analysis and automation tasks can be executed.

1.1. Basics Notions of Design Pattern Detection

A design pattern is a set of roles (participants) to which classes are mapped, that communicate (structure and collaboration) in an organized fashion with respect to the pattern, and that have some specific semantics (intent, motivation and applicability) attached to them. Design pattern detection reconstructs the original mappings between classes and roles with respect to their communication such that the attached semantics provide information about the system under inspection. Figure 1.1 (based on an example in [15]) illustrates this process in which the input is a set of classes and the output are the annotated version of them. The classes within the example represents a subsystem of a text processor that handles line breaks within a text. Composition contains a Compositor that is responsible for handling the line breaks. Different implementations are given by the subclasses of Compositor, and the Composition uses one of them on demand. The subsystem is an instance of the Strategy pattern where the left side in Figure 1.1 represents the initial state before, and the right side the annotated version after the detection process. Mapping 1 for instance contains the atomic mapping Context $\leftarrow$ Composition, Strategy $\leftarrow$ Compositor and ConcreteStrategy $\leftarrow$ ArrayCompositor, hence each role mapping assigns at least one class to one role resulting into multiple role mappings for the same subsystem (situation). Mapping 1 to 3 differ only by one role which is a common scheme in design patterns. Primary roles define the communication scheme within the pattern and drive the communication through the patterns class structure. Secondary roles provide the implementation for the abstractions and inherit the protocol from the primary roles,

thus are commonly fluctuating in their class assignment. A system usually provides multiple versions of the secondary but only a handful of different implementations for the primary roles. All mappings that share the same primary role belong to the same *unique role mapping* representing one specific implementation of a pattern within a sub-system. More formally, a mapping is a k-fold relation between a set of classes $\boldsymbol{C}$ and a set of roles $\boldsymbol{R}$ with

$$m_{P^k} = \{(\boldsymbol{a}, \boldsymbol{b}) \in \boldsymbol{C}^k \times \boldsymbol{R}^k : \boldsymbol{a} \text{ complies with } \boldsymbol{b}\}, \tag{1.1}$$

in which P^k is a specific pattern with k roles. Each unique role mapping reflects an equivalence class in which the *primary roles* are compared. Given a set of mappings $\boldsymbol{M}_{P^k}$ with the equivalence relation $\sim_{P^k}: M_{p^k} \times M_{p^k}$ in which the classes mapped to the primary roles are compared, then

$$\langle m \rangle_{\sim_{P^k}} = \{x \in \boldsymbol{M}_{P^k} : x \sim_{P^k} m\} \tag{1.2}$$

represents the P^k equivalence class from m. That is, all role mappings of pattern P with k roles that have the same primary roles as mapping m. Figure 1.1 illustrates an example where Composition and Compositor are mapped to Context and Strategy representing the primary roles. Array, Simple and Tex-Compositor are mapped to the secondary role each representing Mapping 1-3 that all belong to the Unique Mapping 1.

The question mark in Figure 1.1 represents the algorithm that makes the inference step and decides whether a proposed role mapping is a valid instance of a predefined pattern. The inference method takes in the simplest case k classes and returns a boolean decision whether they are valid or not with respect to the pattern P. One of the classical problems of DPD is the amount of candidate mappings that need to be processed by the inference method. A candidate mapping is a set of classes that are tentatively assigned to a set of roles and tested by the inference method. The amount of classes combined with the amount of rBuilding a system that takes plain source code and returns a list of grouped classes that make up a certain pattern needs several steps to function. Not only is source code a rather complex domain, but also the multivariate output, in form of role mappings, adds to the complexity of the problem. That is, not only one artifact needs to be classified (e.g., ClassA is an abstraction), but a group of classes where each class on its own and the relationships between them need to be considered. Additionally the system needs to cope with the combinatorial explosion caused by the number of classes in the system and number of participating classes in a pattern instance.oles that need to be tested in a non-trivial system results in a intractable big search space, hence high potential *candidate mappings* need to be sampled from the system. Sampling ideally collects all true mappings along with little to no false mappings such that a minimal set of candidates needs to be processed by the inference method. Given a set of potential candidates, features need to be extracted that capture the crucial information stored in the classes. The extraction can be done either directly from the text source or from an intermediate representation like the Abstract Syntax Tree (AST) or Abstract Semantic Graph (ASG). Features range from simple class metrics (e.g., number of functions) to OO relationships (e.g., aggregation) up to predefined subtrees of the ASG called Micro-Structures (MS) (e.g., delegation). Features are then optionally

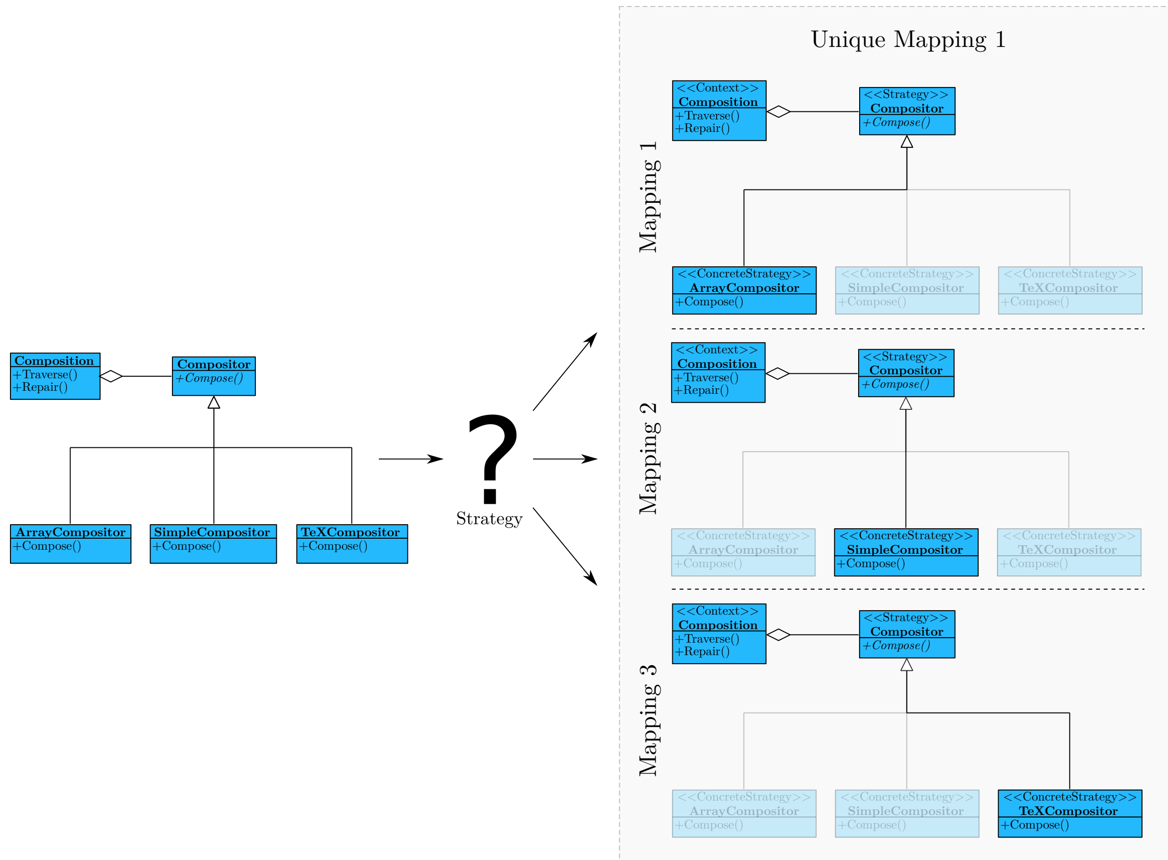


Figure 1.1.: The goal is to find a process that can reconstruct the original role mappings of a given pattern. Inputs are classes of a system, outputs are mappings between pattern roles and the input classes. A unique mapping defines a set of mappings that share the same communication scheme, e.g., Composition and Compositor are the primary roles and define the communication scheme, xCompositor the implementation of the service.

normalized such that they can be processed via the inference method. The features and their normalization mitigate the problem associated with graph based algorithms that are usually computationally intensive up to the point where they become infeasible. After the extraction, sampling and normalization comes the inference step done by a method that can cope with uncertainties, where the uncertainties are caused by implementation details that differ from pattern instance to pattern instance.

1.1.1. Research Question

This work focuses on Convolutional Neural Networks (CNNs) to make the inference, in which subtrees of the ASG are represented by a compressed version of their adjacency matrix. Given the basic notions the concretized research questions are:

1. Are design patterns detectable via modern machine learning methods, i.e., Convolutional Neural Networks?
   a) Are micro-structures appropriate features for the design pattern detection task?

b) How can the amount of candidate mappings within a system be reduced to manageable size?

c) How can role-mappings be efficiently represented such that CNNs can learn from them?

Essentially this work is concerned with building a design pattern detection tool using a modern machine learning approach in the inference step. It does this by providing a system in its entirety, i.e., a system that takes classes as input and provides role mappings of certain patterns as output (see Section 1.2). The preexisting micro-architectures (micro-structures) are revised and reformulated in a formal and unambiguous way to support further research in the area of DPD (see Chapter 2). Also an approach to reduce the amount of candidates is given, which is a crucial step in making the entire pipeline feasible in the first place (see Chapter 3). This thesis discusses multiple ways to normalize micro-structures (subgraphs) into a fixed homogeneous form such that it can be used with most machine learning approaches that rely on fixed sized data structures (see Chapter 4). Furthermore it provides two additional normalization strategies and discusses their differences and applicabilities. At last, training and architecture of the CNNs are provided along with their results and possible implications for 6 design patterns (see Chapter 5). The conclusion elaborates the results of the system and discusses the system in its entirety (see Chapter 6).

1.2. System Overview

Building a system that takes plain source code and returns a list of grouped classes that make up a certain pattern needs several steps to function. Not only is source code a rather complex domain, but also the multivariate output, in form of role mappings, adds to the complexity of the problem. That is, not only one artifact needs to be classified (e.g., *ClassA* is an abstraction), but a group of classes where each class on its own and the relationships between them need to be considered. Additionally the system needs to cope with the combinatorial explosion caused by the number of classes in the system and number of participating classes in a pattern instance. To cope with all the these issues, design pattern detection systems usually have multiple phases each handling one issue at a time. Phase 1 optionally transforms the source code of the system into a different representation (AST, ASG, . . .). After that, phase 2 extracts features that abstract the system into a set of characteristics. These characteristics are the digest of the classes and their relationship and help to pre-filter the huge search space for candidate mappings in phase 3. In phase 4, the candidate mappings are classified whether they are an instance of a certain pattern or not. Specifically steps 2, 3, 4 are very common design decisions and reflect the main problems of DPD. Example systems that have this general structure are presented by Alhusain et al. [1], Antoniol et al. [4], Uchiyama et al. [52], Zanoni et al. [57] and De Lucia et al. [31]. Note that this list may not be complete but is rather a glimpse onto multiphase detection systems.

The system presented in this work is build up similar to other multiphase detection systems. Figure 1.2 illustrates the basic components and their outputs where rounded rectangles are processes and edged rectangles are artifacts. The question mark within Figure 1.1 represents

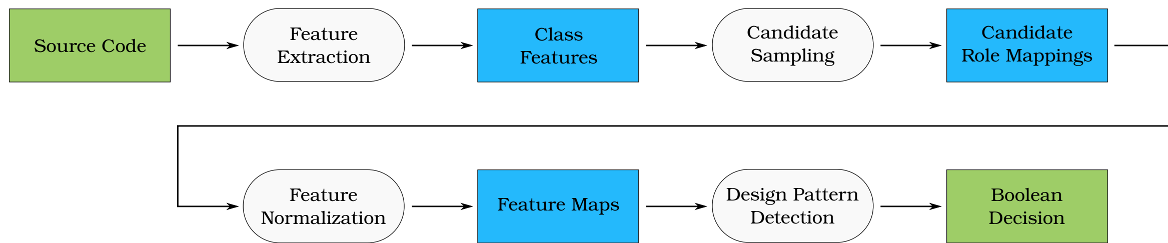


Figure 1.2.: Overview of the detection pipeline. Rectangles are processes; ellipses are artifacts. First class features are extracted from the source code then, based on the features, are candidate role mappings sampled. Features of the sampled role mappings are normalized into feature maps and ultimately used as input for the detection process that provides a decision, whether a given candidate mapping is of a certain pattern.

these processes an each step helps to produce the annotations on the right side. Input is the systems source code (Java) and output are the candidate mappings with their decision. In fact the attached decisions are probabilities or the degree of belief the system thinks that the presented candidate mapping is an instance of a certain pattern or not. All steps in between either transform or filter the input and are discussed in detail in their respective chapters. The only process different from the typical multiphase architecture is the feature normalization that transforms the features of the candidate mapping into a fixed sized matrix such that it can be handled by most machine learning models. Each design pattern parameterizes the pipeline that extracts, samples and detects the pattern according to its peculiarities. Obviously this is needed as each pattern has different communication schemes and participants that do not allow for one general pipeline.

The system detects 6 different design patterns: Adapter, Composite, Decorator, Factory Method, Singleton and Template Method. These were subjects in many research scenarios throughout the DPD community [1, 4, 19, 31, 51, 57] hence offer a way to produce somewhat comparable results. Furthermore the selected patterns are very popular and broadly used in projects such that they have a high relevance in the context of software engineering. Additionally there exists a benchmark platform for design pattern detection [6] on which results from DPD tools are evaluated. This happens on a semi-automatic basis where platform users can vote on the found results. The benchmark was not used throughout this work because of time limitations. The peer reviewed instances to build this tool came from the Pattern-like Micro-Architecture Repository (P-MARt 04/10/19) [56] where 8 of the 9 projects were used (see Chapter 5 for details).

1.2.1. Detected Design Patterns

Despite the fact that there exist 23 classical design patterns only 6 are evaluated, two from each DP category: Singleton and Factory Method from Creational Patterns, Composite and Decorator from Structural Patterns, and Strategy and Template Method from Behavioral Patterns.

Adapter

Adapter is a pattern often used in the context of legacy code or frameworks. It is useful to bridge the legacy interfaces to new modules via an adapter allowing the new modules to be defined in a clean way. Frameworks use adapters to offer an entrance point for external components that are implemented by the framework users. An example for this would be the Eclipse Project that uses this concept extensively to allow loosely coupled but dynamic plug-ins for the entire Eclipse ecosystem [54]. Figure 1.3 shows the patterns structure in which the client code calls the interface defined by Target. Adapter provides an implementation for Target's interface, adapts the message and redirects it to Adaptee. In the context of DPD, Client roles offer little to no additional information as some code must call the pattern anyway or the entire implementation would be superfluous. Hence clients are not considered a valid role during the detection itself despite their usefulness in general, as they point out the start of the patterns communication sequence. This has little to now influence on the software engineer that uses the detection tool as the Client is rather inconsistently part of a pattern or not within [15].

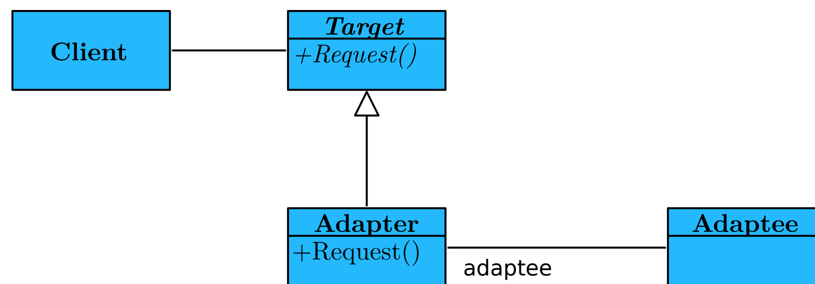


Figure 1.3.: An Adapter converts the interface from the Adaptee to a Target such that Clients can interact with them.

Composite

Figure 1.4 illustrates the class structure of the Composite pattern, which is useful to build hierarchies and interact with them in a uniform way. Client calls an operation on Components which is forwarded by Composites in the hierarchy to Leafs that implement the actual behavior. New Leafs and Composites can be added dynamically such that the hierarchy structure is flexible and adaptable during runtime. Again, Client is ignored in the context of DPD resulting in a total of 3 roles. The Composite pattern is often used in drawing tools and Graphical User Interface (GUI) frameworks like Swing [37] for the Java platform. In Swing, all UI components comply to the Component interface and offer a *draw* method that is either implemented with a specific drawing algorithm for the component, or delegates the call if the current component is a container.

Decorator

Decorators allow to dynamically attach new responsibilities to existing objects thus may be used as flexible alternative to inheritance. Sometimes inheritance hierarchies force developers

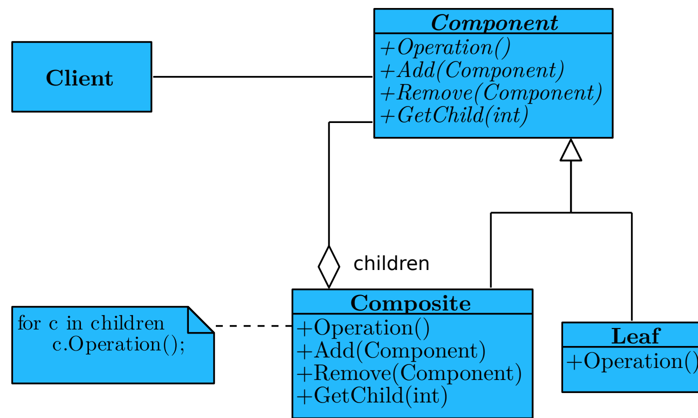


Figure 1.4.: Composite offers a way to build up dynamic hierarchies where the elements can be used uniformly through a shared interface. Component offers this shared interface and Composite and Leaf implement the hierarchy classes. Each hierarchy class contains the Operation method that execute the designated algorithm on the hierarchy parts.

to enumerate many possible combinations of subclasses along with their responsibilities in order to cover all use-cases. This becomes very quickly unpractical and unfeasible from a maintenance perspective. Decorators circumvent this limitation by using object composition in which Decorator delegates the original call, made to the shared Component interface, to Concrete Component and add new states and responsibilities along the way. The InputStream and OutputStream classes from the Java IO package are a very prominent instance of the Decorator pattern. InputStream represents the Component and Concrete Component role, and may be extended by decorators that specialize the stream to certain sources, e.g., audio, bytes, files, objects, string and so forth. These decorators are then applied in a nested fashion such that the original InputStream may become a ByteArrayInputStream and then a AudioInputStream such that it may function as a stream for audio signals. The basic structure is given in Figure 1.5 which contains two distinct Concrete Decorator representing the ByteArrayInputStream and AudioInputStream from the previous example.

Factory Method

A broadly used pattern to create objects is the Factory Method pattern. The Creator defines interfaces to create Products that are implemented by the sub-classing Concrete Creator. This is useful if the Concrete Products stand in a relationship with its creators. A typical use-case for this pattern is if a class hierarchy represents the elements within a system and a second class hierarchy the handlers that work with them. The base class of the elements may now define a factory method that all the subclasses implement such that each element has a chance to return a handler specialized for its needs. This is reflected by Figure 1.6 in which the Creator is the element that creates the handlers.

Template Method

The Template Method pattern enables an algorithm and its steps to be dynamic by providing a skeleton of the algorithm which is completed by its subclasses. This allows a framework to

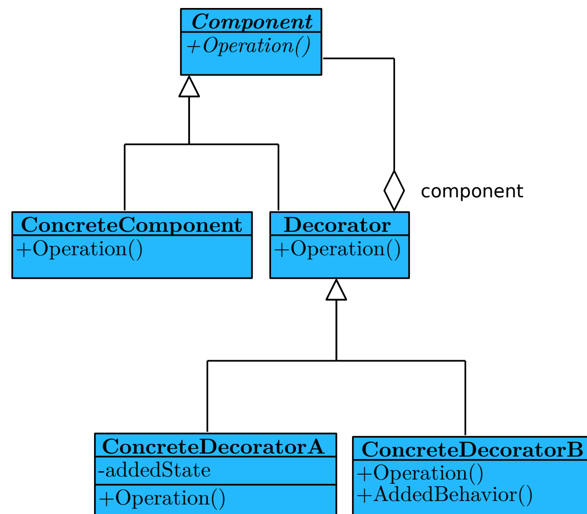


Figure 1.5.: Decorators allow to dynamically attach new responsibilities to existing objects. Concrete Components are basic classes that are enriched by the Decorators. Both share the same interface via the Component and each Concrete Decorator offers a specific additional implementation detail.

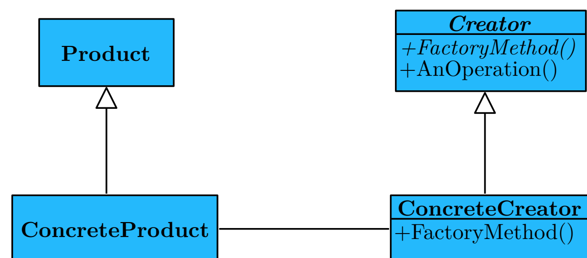


Figure 1.6.: Factory Method defines an interface to create objects but deferrers the actual instantiation to the subclasses. Creator defines the factory method which is implemented by the Concrete Creator.

outline the necessary steps to complete a task by defining an Abstract Class, and relying on the users to provide the missing steps in terms of a Concrete Class. Figure 1.7 illustrates the basic structure where the Abstract Class relies on the subclass implementation within the *TemplateMethod()* method.

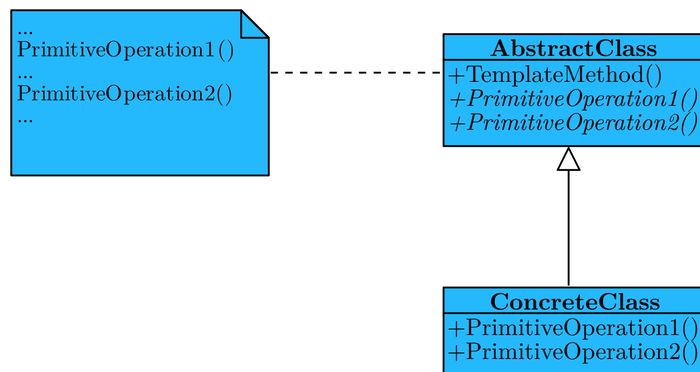


Figure 1.7.: Template Method allows to provide different implementations for certain steps of an algorithm. These steps are defined and called in the Abstract Class in the order the algorithm needs them. Concrete Classes only provide the implementation of the steps and but cannot control their call order.

2. Feature Extraction

Source code is usually presented in text form along with a specific syntax that allows the developer to capture the algorithms and structure them in an appropriate way. Thou simpler than natural language, source code still is a challenging representation for analysis and automation tasks as modern general purpose languages provide a wide variety of grammar constructs. Furthermore DPD does not limit itself to the mere syntax of languages, but tries to reason and interpret the content of the code such that it can make informed decisions. Feature extraction helps to reduce the complexity of syntax and semantics by abstracting the source code into high level constructs that can be easier processed. The abstract process is depicted in Figure 2.1 where the source code (given as ASG) is the input and class features are the output of the feature extraction.

Figure 2.1.: The feature extraction process takes the source code transforms it into an ASG and extracts the class feature from it. Class features are subtrees within the ASG, so called micro-structures.

2.1. Related Work

There are numerous ways to represent source code such that models and algorithms are able to detect patterns in it, and in fact, before explicit features are extracted, the text is usually transformed into an internal representation. For instance, the source code might be transformed into an Abstract Syntax Tree (AST) [30], Abstract Semantic Graph (ASG) [12], Control Flow Graph [2] or into a specific bit representation from which features are extracted or which directly represent the features. Don et al. [11] reports up to 18 different internal representations in 23 tools, which is not a complete list of all available tools but emphasizes the importance of the internal representation and features.

Guéhéneuc et al. [19] uses an AST/ASG to extract object-oriented attributes from classes like: size (i.e., number of methods, fields. . .), filiation (i.e., number of parents, childrens), cohesion (i.e., number of internal class calls), coupling (i.e., number of external calls), which are then used to compute fingerprints. The rational behind these fingerprints is actually related to the process of candidate space reduction but Alhusain et al. [1] incorporated the original metrics along with the results of CKJM [46], Dependency Finder [49] and JMT [24] to build a set of features. These tools work on the AST or bytecode of a system and extract mainly unary OO metrics that capture a variety of aspects similar to the metrics for

fingerprinting. Although containing features that capture information about multiple classes and their relationships, most features only provide an abstract one-dimensional view onto these. For instance CKJM measures metrics like *Depth of Inheritance Tree (DIT)* or *Number of Public Methods (NPM)*, similar JMT contains the *NPEC - Number of Public Extendable Classes* mearsure, which all abstract the original multivariate relationship between classes.

A more exotic approach is given by De Lucia et. al [31] in which the class diagrams of a system are transformed into a visual language representation that describe the diagram structure in form of visual sentences. This tool uses the intermediate representation, the diagram description given in the visual language, directly as feature instead of transforming it into an AST. In contrast Antoniol et al. [4] transforms the AST one step further into an Abstract Object Language (AOL) representation to extract language independent OO metrics similar to the features extracted by Alhusian et. al. Both tools have the ability to work with multiple platforms as they either abstract the language dependent representation or directly work with an abstraction of it (diagrams).

In contrast to the previously presented approaches, Uchiyama et al. [52] used the Goal-Question-Metric (GQM) [53] approach to produce metrics that are closely related to the design patterns and their roles. Following the scheme of GQM, they defined goals and questions tightly related to the design pattern and their roles, resulting in metrics that highly correlated with them. An example would be *Number of other classes with field of own type - NCOF* metric which is a detailed description of the aggregation relationships to others classes and useful in outlining the Adaptee in the Adapter pattern.

Tsanatalis et al. [51] uses similarity scoring on the system's and pattern's adjacency matrices. It uses the fact that the system class diagram and the pattern class diagram is essential a directed graph that can be perfectly mapped onto square matrices. Each feature is then represented in its own square matrix capturing relational information between classes and within classes. Used features are for example *association*, *generalization*, *method invocation*, but also unary features like *abstract class* (is the class an abstract class). The features themselves are in comparison to GQM designed features rather fundamental aspects of OO programming, however they maintain the entire relationship information. Instead of providing a single scalar for a relationship (e.g., *NCOF*) the matrix itself can relate the two classes along with the number of fields that connect the two classes. This information is obviously essential for DPD an mimics how humans detect design patterns in code.

2.1.1. Micro-Architectures

Micro-structures (MSs) are a catalog of very small design patterns that capture characteristic between a very limited set of roles (usually 1-2 roles). They represent the best of both worlds, features that maintain actively the relationship information between classes, but also are tailored and designed for design pattern detection and analysis. They provide means to describe software on an abstract level beyond simple counting, by building up relationships between elements and describing them in form of patterns. To most outstanding difference between MSs and DPs is that MSs can be detected by means of logic (first-order logic). The entire catalog of Micro-Structures is made up by three sub-catalogs: Elemental Design

Patterns (EPDs), Design Pattern Clues (DPCs) and Micro-Patterns (MPs). The sub-catalogs where defined independently, each having their own motivations and goals, but all of them prove valuable in the process of detecting design patterns as Fontana et al. [5] concluded in a series of experiments. Not all MSs are important for every DP role but each role has its own small set of MSs that describe its main aspects. The most complete catalog is given by Maggioni [34] in which not only all catalogs are unified but also each pattern is defined by means of logic. The redefinition by means of logic is a crucial step to a unified catalog as not all sub-catalogs are presented in an unambiguous way.

Elemental Design Patterns (EDPs), defined by Smith [44], capture solutions to common problems in everyday programming and basic concepts of OO design. In fact the 16 EDPs structured in 5 categories are so basic that they are used without even noticing them as pattern. For example the Abstract Interface pattern states that a common interface for a family of types is defined where the implementation is deferred to the subtypes. The concept behind the pattern is so natural that it is usually materialized and integrated in the programming language directly (e.g., in Java: interfaces and abstract classes). A more sophisticated pattern would be the Redirected Recursion that defines a recursion involving multiple objects of the same type. From these examples it should be clear that EDPs are defined to break down complex abstract (sub-)systems into smaller easy recognizable and fundamental bits of code that allow automatic reasoning and analysis.

In contrast to EDPs that capture also relationships between types, Micro-Patterns (MPs) limit themselves to recurring implementation practices on a single class. The 27 MPs, defined by Gil and Maman [17], are structured in 8 categories with the common purpose to define and name everyday practices (on a class level) such that easy communication is possible, much like Design Patterns. The patterns were found by searching for so called pre-patterns in a source code corpus and identifying commonalities that were then restricted in a sensible manner. The catalog itself is defined in a rather ambiguous way thus slight variations between the intention of the original authors and the logical definition by Maggioni might exist. Differences between the definitions might even increase as most of the MPs are defined too restrictive to be useful in the context of design pattern detection. For example the Implementor pattern is a concrete class where all methods override inherited abstract methods. Though being useful in the context of communication, the restriction that all methods must be inherited might be to strong thus reduces the patterns applicability. Possible changes involve that only public methods are inherited which allow classes to implement a private behavior providing a more realistic implementation style.

With the direct intention to find DPs, Maggioni [32] defined 41 Design Pattern Clues (DPCs) divided in 8 categories. DPCs are possible hints that can be used to find a certain DP and were constructed by manually analyzing DP implementations. For instance, the Concrete Product Getter pattern helps to find Creational Patterns, as it defines a class that declares one or more methods that return objects belonging to some other class. Although Smith mentions the usefulness of EDP in the context of DPD, design pattern clues are the only micro-structures that where design exclusively with the purpose of detecting design patterns, thus their amount and also their semantics differ from the others. DPCs are very

specific and describe detailed information of classes, for instance the Proxy Method Invoked is applicable to classes that invoke a referred subject by using the Redirect in Limited Family EDP. Not only does the DPC make a statement about the caller by referring it as proxy (which again referrers to a MS) but also makes use of an element design pattern to describe how the communication should look like.

2.2. Micro-Structures as Features

The approach used within this work extracts micro-structures from an ASG that are than normalized to a matrix form as described in Chapter 4. Figure 2.2 illustrates the running Strategy example of the text compositor and extends it by a factory from which the Compositor can be retrieved. Again the roles of the Strategy pattern are annotated to the Composition, Compositor, ArrayCompositor, TeXCompositor and SimpleCompositor class, but in addition there are also micro-structures roles attached. Obviously the Compositor functions as Superclass in the Inheritance MS and ArrayCompositor, TeXCompositor and SimpleCompositor as Subclass. These classes make up a named subgraph related to the inheritance of this particular subsystem and enables analysis methods to directly interpret the classes and their relationships. Additionally the Retrieve MS is given which states that a Sink uses a Retrieved object within its local scope, by retrieving it from a Source. In this case Compositor is the Sink and retrieves the Composition from the CompositeFactory. Again the intent of the subgraph is inherently given by the MS, i.e., it is clear how these classes interact with each other. Extracting named subgraphs has the advantage that the amount

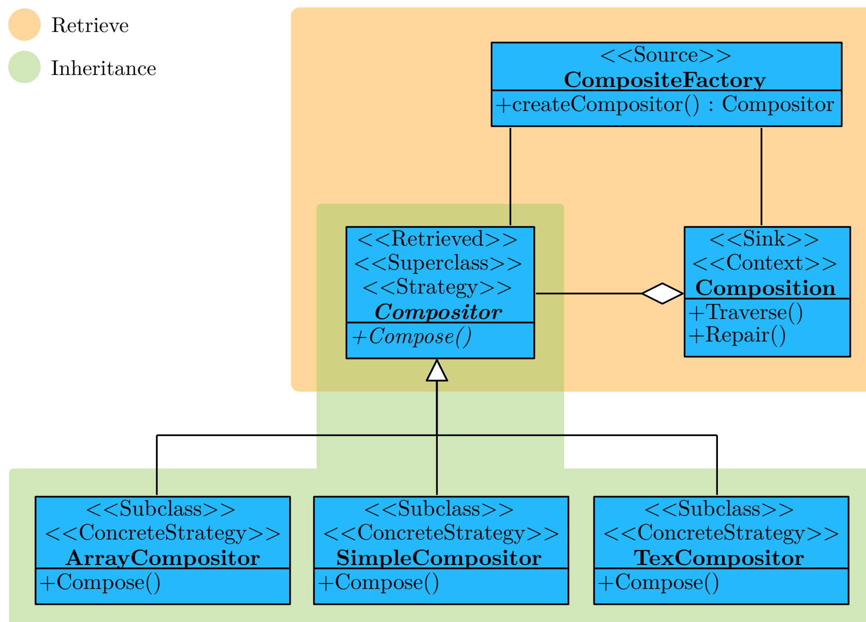


Figure 2.2.: The text processor sub-system handles line breaks within a document. Strategy pattern roles are attached as given in the original version of the example (Strategy ← {Compositor}, ConcreteStrategy ← {Array-, Tex-, SimpleCompositor}, Context ← Composition). Additionally Retrieve and Inheritance MS roles are given and represent a certain proportion of the ASG.

of subgraphs that need to be processed is effectively reduced to the bare minimum but still retains most of the crucial information stored within the entire graph. These subgraphs may (partially) overlap since they describe different aspects of the same elements thus do not interfere with each other. In fact they even complement themselves and help building complex semantics by combining multiple MSs where each pattern contributes a piece of information. For example the combination of Inheritance and Retrieve enables the reasoning that the Composition uses an object of a different class, to retrieve its aggregate that participates in a inheritance structure. The basic structure of the Strategy pattern becomes clear if the example is enriched by a MS related to method invocation (e.g., Template Method) such that the Composition calls a Template Method on the Compositor. Each additional MS contribute to the entire semantic of the structure and demonstrates the advantages of using micro-structures in the analysis process. The obvious disadvantage is that features that might not appear important from a human perspective, but would work well from a machine learning point of view, may be excluded because of the "human" biased view onto it.

2.2.1. Technical Overview

Initially the source code, given in form of Java files, is parsed, analyzed and abstracted into an AST/ASG by the Spoon [38] library, which is an open-source library for Java source code analysis and transformations. It provides an abstract view onto an AST, much like an ASG with convenient methods for faster analysis, and modification of the original code. A very important feature of the library is that it works with systems that do not provide the complete classpath, i.e., not all dependencies must be provided in order to analyze the system. This situation can be considered the normal case in DPD as not always all transitive dependency might be accessible. The basic class structure of the system is the same for detecting DPs and MSs as both represent the same concept of role mappings.

Figure 2.3 shows the subsystems class diagram responsible for detecting patterns. Roles and Patterns are Describables that have a name and description, where patterns additionally store their associated Roles along with a Detector that can be used to detect itself. A detector is a strategy implementing the detection of a specific pattern and returns a set of role mappings. A RoleMapping maps multiple qualified names of (inner) classes and interfaces to specific roles. Despite the fact that multiple qualified names can be stored on the same role within a role mapping, the detector still needs to return a set of role mappings as multiple independent instances of the same pattern may occur within a class, making them indistinguishable. The example in Figure 2.3 depicts the implementation of the Inheritance MS with its Subclass and Superclass roles, and its detector. The remaining micro-structures are implemented similar to this example where the concrete detector strategies return the role mappings that represent the class features.

The system detects 16 out of 16 EDPs, 27 out of 27 MPs and 18 out of 41 DPCs resulting in a total of 61 MSs, some of them are slightly adapted to the purpose of design pattern detection (MPs). Additionally the system detects 6 basic concepts of OO programming, that are partly covered by some MSs in some situations. The reason for not implementing even

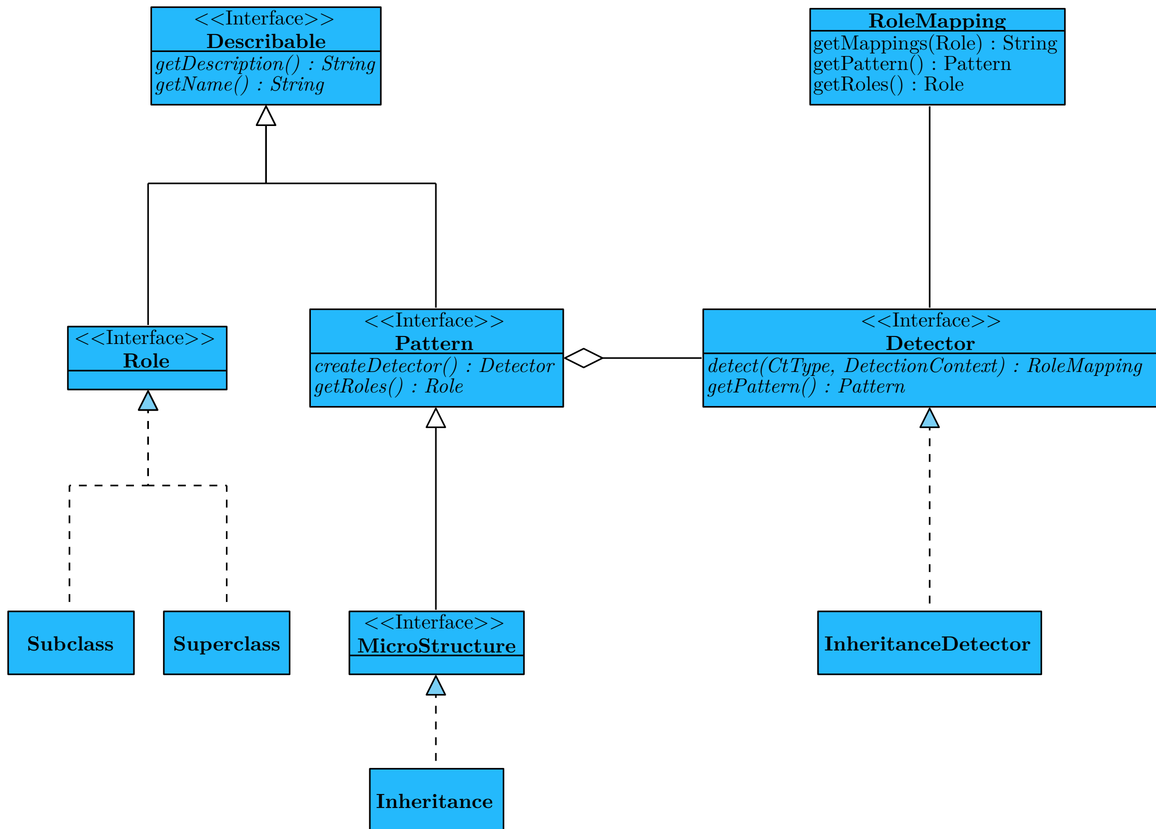


Figure 2.3.: Describable is the base type of roles and patterns. Each pattern defines a Factory Method for its detector. Detectors return (candidate) RoleMappings which contain the mapping between class (qualified name) and the role.

half of the DPCs is that DPCs are very specific in their definition thus being not helpful for detecting design patterns they are not design for.

2.2.2. Micro-Structure Catalog

The following catalog provides the definitions of the MSs used in this work and is based on Maggioni's catalog [32]. It is reworked version in which MSs reflect the implementation that was used to detect pattern within this approach. Furthermore the redefinitions operate on basic OO constructs to remove previous ambiguities and to build a proper context in which the predicates live.

Basic Elements

Basic elements define the abstraction for platform specific constructs. They provide basic structures on which all MSs operate on and convenient functions that make the access to the elements more readable.

Let π be the projection function that returns the i'th element of a tuple or a set such that $\pi_3((1, 2, 3)) = 3$ holds. Furthermore let the following structures be defined where the set definitions are provided in the next section:

Statement Element A statement is a tuple with its components $\boldsymbol{C} \in \mathbb{T} \cup \mathbb{M} \cup \mathbb{F}$ and its type $T \in \mathbb{T}$ to which it resolves. Statements represent the formalism boundary and might not be fully specified, i.e., if needed special statements (e.g., return statement) are defined informally on the fly to retain readability. Statements usually invoke build in operations or operations defined on types.

$$St \coloneqq (\boldsymbol{C}, T) \tag{2.1}$$

Field Element A field is a triple consisting of its name $N \in \mathbb{ID}$, type $T \in \mathbb{T}$ and its modifiers $\boldsymbol{Mod} \subseteq \mathbb{MOD}$. Fields store information of a specific types T.

$$F \coloneqq (N, T, \boldsymbol{Mod}) \tag{2.2}$$

Method Element A method is a quintet consisting of its name $N \in \mathbb{ID}$, (return) type $T \in \mathbb{T}$, its modifiers $\boldsymbol{Mod} \subseteq \mathbb{MOD}$, its parameters $\boldsymbol{P} \subseteq \mathbb{T}^n$ and its statements $\boldsymbol{St} \subseteq \mathbb{ST}$. Methods execute an algorithm given by its statements and return some information in form of T (which might by a NULL/VOID type).

$$M \coloneqq (N, T, \boldsymbol{Mod}, \boldsymbol{P}, \boldsymbol{St}) \tag{2.3}$$

Constructor Element A constructor is a quartet consisting of its name $N \in \mathbb{ID}$, its modifiers $\boldsymbol{Mod} \subseteq \mathbb{MOD}$, its parameters $\boldsymbol{P} \subseteq \mathbb{T}^n$ and its statements $\boldsymbol{St} \subseteq \mathbb{ST}$. Similar to methods, constructors execute an algorithm but focus on the initialization of the enclosing object.

$$Co \coloneqq (N, \boldsymbol{Mod}, \boldsymbol{P}, \boldsymbol{St}) \tag{2.4}$$

Object Element An object is a tuple consisting of its placeholder name $N \in \mathbb{ID}$ and its type $T \in \mathbb{T}$. Objects are instances of a specific type.

$$O \coloneqq (N, T) \tag{2.5}$$

Type Element A type is a quintet consisting of its name $N \in \mathbb{ID}$, fields $\boldsymbol{F} \subseteq \mathbb{F}$, methods $\boldsymbol{M} \subseteq \mathbb{M}$, ancestors $\boldsymbol{A} \subseteq \mathbb{T}$ and constructors $\boldsymbol{C} \subseteq \mathbb{CO}$. Types declare methods and fields, i.e., data and operations, to encapsulate a specific idea.

$$T \coloneqq (N, \boldsymbol{F}, \boldsymbol{M}, \boldsymbol{A}, \boldsymbol{C}) \tag{2.6}$$

Basic Collections

All reasoning is constrained on a System Under Inspection (SUI) $\boldsymbol{S} \coloneqq \{T_1, T_2, \dots T_x\}$ where T defines a type that is part of the type system, e.g., Interfaces or Classes within the Java language. Function and other special types are not considered as currently no DP or MS makes use of them within this work. Collections of objects are, if not stated otherwise, always relative to the system such that $\mathbb{F} = \mathbb{F}_S$.

Types Types declared within the system.

$$\mathbb{T}_S := \boldsymbol{S} \tag{2.7}$$

Interfaces Interfaces declared within the system that do not provide any implementation detail.

$$\mathbb{I} := \{t \in \mathbb{T} : \forall m \in \pi_3(t) : abstract \in modifiers(m)\} \tag{2.8}$$

Classes Classes declared within the system that provide implementation details.

$$\mathbb{C} := \boldsymbol{S} \setminus \mathbb{I} \tag{2.9}$$

Fields The set of fields within a system, a type, declared within a type or within a type's ancestors.

$$\begin{aligned}
\mathbb{F} &:= \bigcup \{t \in \mathbb{T} : \pi_2(t)\} \\
\mathbb{F}^*_T &:= \{f \in \mathbb{F} : f \in \pi_2(T) \vee \exists a \in \pi_4(T) : f \in \pi_2(a)\} \\
\mathbb{F}_T &:= \{f \in \mathbb{F} : f \in \pi_2(T)\} \\
\mathbb{F}^+_T &:= \mathbb{F}^*_T \setminus \mathbb{F}_T
\end{aligned} \tag{2.10}$$

Methods The set of methods within a system, a type, declared within a type or within a type's ancestors.

$$\begin{aligned}
\mathbb{M} &:= \bigcup \{t \in \mathbb{T} : \pi_3(t)\} \\
\mathbb{M}^*_T &:= \{m \in \mathbb{M} : m \in \pi_3(T) \vee \exists a \in \pi_4(T) : m \in \pi_3(a)\} \\
\mathbb{M}_T &:= \{m \in \mathbb{M} : m \in \pi_3(T)\} \\
\mathbb{M}^+_T &:= \mathbb{M}^*_T \setminus \mathbb{M}_T
\end{aligned} \tag{2.11}$$

Constuctors The set of constructors within a system or (declared) in a type.

$$\begin{aligned}
\mathbb{CO} &:= \bigcup \{t \in \mathbb{T} : \pi_5(t)\} \\
\mathbb{CO}^*_T &:= \{c \in \mathbb{CO} : c \in \pi_5(T) \vee \exists a \in \pi_4(T) : c \in \pi_5(a)\} \\
\mathbb{CO}_T &:= \{c \in \mathbb{CO} : c \in \pi_5(T)\} \\
\mathbb{CO}^+_T &:= \mathbb{CO}^*_T \setminus \mathbb{CO}_T
\end{aligned} \tag{2.12}$$

Attributes The set of attributes within a system or (declared) in a type.

$$\begin{aligned}
\mathbb{A} &:= \mathbb{F} \cup \mathbb{M} \\
\mathbb{A}^*_T &:= \mathbb{M}^*_T \cup \mathbb{F}^*_T \\
\mathbb{A}_T &:= \mathbb{M}_T \cup \mathbb{F}_T \\
\mathbb{A}^+_T &:= \mathbb{A}^*_T \setminus \mathbb{A}_T
\end{aligned} \tag{2.13}$$

Statements The set of statements within a system, type or method.

$$
\begin{aligned}
\mathbb{ST} &:= (\bigcup\{m \in \mathbb{M} : \pi_5(m)\}) \cup (\bigcup\{c \in \mathbb{CO} : \pi_4(c)\}) \\
\mathbb{ST}_T^* &:= \bigcup\{m \in \mathbb{M}_T^* : \pi_5(m)\} \\
\mathbb{ST}_T &:= \bigcup\{m \in \mathbb{M}_T : \pi_5(m)\} \\
\mathbb{ST}_T^+ &:= \mathbb{ST}_T^* \setminus \mathbb{ST}_T \\
\mathbb{ST}_M &:= \pi_5(M)
\end{aligned}
\tag{2.14}
$$

Objects The set of objects that a system or type can span.

$$
\begin{aligned}
\mathbb{O}_S &:= \{\text{all possible objects in } \boldsymbol{S}\} \\
\mathbb{O}_T &:= \{\text{all possible object of } T\}
\end{aligned}
\tag{2.15}
$$

Identifier All valid identifiers on a specific platform.

$$
\mathbb{ID} := \{\text{all valid identifiers}\} \tag{2.16}
$$

Modifier A modifier represents a property of an element that restricts its access or makes statements about the completeness of its definition. Abstract defines an element that does not provide any implementation details thus applicable to types and methods. Final states that a variable is not modifiable outside of its creation context. Private states than the element is not accessible from outside its definition context, e.g., method M is only visible within its declaring type T. Static states that the attribute belongs to the class and not to a specific instance, i.e., the attribute value is shared across all instances.

$$
\mathbb{MOD} := \{abstract, final, private, static\} \tag{2.17}
$$

Basic Expressions

Basic expressions are OO properties that are usually part of the language. They are the basic building blocks from which micro-structures are defined thus non-optional.

Name Returns the name of the artifact.

$$
name(x) : \mathbb{T} \cup \mathbb{A} \cup \mathbb{O} \to \mathbb{ID},\ x \mapsto \pi_1(x) \tag{2.18}
$$

Type Returns the type of an object, field, statement, or method.

$$
type(x) : \mathbb{A} \cup \mathbb{ST} \cup \mathbb{O} \to \mathbb{T},\ x \mapsto \pi_2(x) \tag{2.19}
$$

Declaring Type Returns the declaring type of an element.

$$\begin{aligned}
&declaringType(x) : \mathbb{F} \to \mathbb{T} : \pi_1(\{t \in \mathbb{T} : x \in \pi_2(t)\}) \\
&declaringType(x) : \mathbb{M} \to \mathbb{T} : \pi_1(\{t \in \mathbb{T} : x \in \pi_3(t)\}) \\
&declaringType(x) : \mathbb{C} \to \mathbb{T} : \pi_1(\{t \in \mathbb{T} : x \in \pi_5(t)\})
\end{aligned} \tag{2.20}$$

Modifiers Returns the modifiers of an element.

$$modifiers(x) : \mathbb{A} \to \mathbb{MOD},\ x \mapsto \pi_3(x) \tag{2.21}$$

Parameters Returns the parameters types of a method.

$$parameters(x) : \mathbb{M} \to \mathbb{T}^n,\ x \mapsto \pi_4(x) \tag{2.22}$$

Signature Returns the signature of a method.

$$signature(x) : \mathbb{M} \to \mathbb{ID} \times \mathbb{T} \times \mathbb{T}^n,\ x \mapsto (name(x), type(x), parameters(x)) \tag{2.23}$$

Invocation Whether a statement invokes an attribute. Note that the invocation of a field returns the field as a statement and allows further invocations on attributes defined by the type of the field, e.g., $Invokes(Invokes(m1, f1), m2)$ means that method $m1$ invokes the field $f1$ which invokes the method $m2$ defined by $f1$. Note that the method $\mathbb{S} \times \mathbb{F} \to \mathbb{S}$ is written in title case for interchangeability reasons between the predicates.

$$\begin{aligned}
&Invokes(a, b) : \mathbb{S} \times \mathbb{F} \to \mathbb{S},\ (a, b) \mapsto \text{returns } b \text{ as statement} \\
&Invokes(a, b) : \mathbb{S} \times \mathbb{A},\ (a, b) \mapsto a \text{ invokes } b \\
&Invokes(a, b) : \mathbb{F} \times \mathbb{A},\ (a, b) \mapsto b \in \mathbb{A}_{type(a)} \wedge Invokes(a, b) \\
&Invokes(a, b) : \mathbb{M} \times \mathbb{A},\ (a, b) \mapsto \exists s \in \mathbb{ST}_a : Invokes(s, b)
\end{aligned} \tag{2.24}$$

Abstract Whether a class defers some implementation details to subclasses, or whether a method only declares its signature but deferrers its implementation to subclasses.

$$Abstract(x) : \mathbb{T} \cup \mathbb{M},\ x \mapsto abstract \in modifiers(x) \tag{2.25}$$

Final Whether the element can be modified outside of its creation context.

$$Final(x) : \mathbb{F},\ x \mapsto final \in modifiers(x) \tag{2.26}$$

Private Whether the element can be accessed outside its declaration context.

$$Private(x) : \mathbb{A},\ x \mapsto private \in modifiers(x) \tag{2.27}$$

Static Whether the attribute is shared across all instances.

$$Static(x) : \mathbb{A},\ x \mapsto static \in modifiers(x) \tag{2.28}$$

Ancestor Whether type a is the ancestor of type b (includes generalizations and realizations).

$$Ancestor(a,b) : \mathbb{T}^2,\ (a,b) \mapsto a \in \pi_4(b) \tag{2.29}$$

FamilyHead Whether type a is a family head of type b, i.e., participates in the same inheritance hierarchy but has no ancestor.

$$\begin{aligned} FamilyHead(a,b) : \mathbb{T}^2,\ (a,b) \mapsto Ancestor(a,b) \\ \wedge \nexists T \in \mathbb{T} : Ancestor(T,a) \end{aligned} \tag{2.30}$$

Sibling Whether type a belongs to the same family as type b but are not in a Ancestor relationship.

$$\begin{aligned} Sibling(a,b) : \mathbb{T}^2,\ (a,b) \mapsto \\ \neg(Ancestor(a,b) \vee Ancestor(b,a)) \\ \wedge \exists t \in \mathbb{T} : Ancestor(t,a) \wedge Ancestor(t,b) \end{aligned} \tag{2.31}$$

Primitive Whether the type is a primitive, e.g., integer, float, double, etc.

$$Primitive(r) : \mathbb{T},\ r \mapsto r \text{ is primitive} \tag{2.32}$$

Overrides Whether a method overrides an inherited method.

$$\begin{aligned} Overrides(m1,m2) : \mathbb{M}^2,\ (m1,m2) \mapsto \\ signature(m1) = signature(m2) \\ \wedge Ancestors(declaringType(m2), declaringType(m1)) \end{aligned} \tag{2.33}$$

Replaces Whether a method overrides and replaces the implementation inherited.

$$\begin{aligned} Replaces(m1,m2) : \mathbb{M}^2,\ (m1,m2) \mapsto \\ signature(m1) = signature(m2) \\ \wedge Ancestors(declaringType(m2), declaringType(m1)) \\ \wedge \neg Invokes(m1,m2) \end{aligned} \tag{2.34}$$

Refines Whether a method overloads and refines the implementation inherited.

$$\begin{aligned} &Refines(m1, m2) : \mathbb{M}^2,\ (m1, m2) \mapsto \\ &\quad signature(m1) = signature(m2) \\ &\quad \wedge Ancestors(declaringType(m2), declaringType(m1)) \\ &\quad \wedge Invokes(m1, m2) \\ &\quad \wedge \exists s \in \mathbb{ST}_{m1} : \neg Invokes(s, m2) \end{aligned} \tag{2.35}$$

Getter Whether a method is a Getter used to access a field.

$$Getter(x) : \mathbb{M},\ x \mapsto \text{ is a getter} \tag{2.36}$$

Setter Whether a method Setter used to modify a field.

$$Setter(x) : \mathbb{M},\ x \mapsto \text{ is a setter} \tag{2.37}$$

Micro-Structures

This is an alphabetical list of most micro-structures described by Smith [44], Maggioni et al. [33], and Gil and Maman [17]. The definitions are based on the basic concepts from the previous section and provide a new formal redefinition that removes almost all ambiguities from their original definitions while still retaining their readability. The textual descriptions are either reformulated, if it improves the understandability, or kept as their original authors formulated them. The actual pattern definitions diverge from the original if the redefinition is more appropriate in the context of DPD. The parameters of the predicates represent the detected MS roles, i.e., the nodes within the ASG, and are always on a type level.

Abstract Interface Provides a common interface for a type family but deferrers the implementation details to subtypes.

EDP - Object Elements

$$\begin{aligned} &AbstractInterface(r) : \mathbb{T}, \\ &\quad r \mapsto r \in \mathbb{I} \\ &\qquad \vee Abstract(r) \\ &\qquad \vee \exists m \in \mathbb{M}_r : Abstract(m) \end{aligned} \tag{2.38}$$

Abstract Products Returned A method returns a reference to an abstract class.

DPC - Return Information

$$\begin{aligned} &AbstractProductsReturned(c, p) : \mathbb{T}^2,\ (c, p) \mapsto \\ &\quad \exists m \in \mathbb{M}_c : type(m) = p \wedge Abstract(p) \end{aligned} \tag{2.39}$$

Augmented Type A class with only abstract methods and three or more static final fields of the same type.

MP - Data Managers

$$\begin{aligned} &AugmentedType(r) : \mathbb{T},\ r \mapsto \\ &\quad |\{f \in \mathbb{F}_r : Final(f) \land Static(f)\}| > 2 \\ &\quad \land \forall m \in \mathbb{M}_r : Abstract(m) \end{aligned} \tag{2.40}$$

Box A class which has exactly one, mutable, instance field.

MP - Wrappers

$$Box(r) : \mathbb{T},\ r \mapsto |\mathbb{F}_r| = 1 \land \exists f \in \mathbb{F}_r : \neg Static(f) \land \neg Final(f) \tag{2.41}$$

Canopy A class with exactly one instance field that it assigns exactly once, during instance construction.

MP - Wrappers

$$Canopy(r) : \mathbb{T},\ r \mapsto |\mathbb{F}_r| = 1 \land \exists f \in \mathbb{F}_r : \neg Static(f) \land Final(f) \tag{2.42}$$

Cobol Like A class with a single static method and at least one static field but no instance fields.

MP - Degenerate Behavior

$$\begin{aligned} &CobolLike(r) : \mathbb{T},\ r \mapsto \\ &\quad |\mathbb{M}_r| = 1 \land |\mathbb{F}_r| > 0 \\ &\quad \land \exists m \in \mathbb{M}_r : Static(m) \\ &\quad \land \forall f \in \mathbb{F}_r : Static(f) \end{aligned} \tag{2.43}$$

Common State A class in which all fields are static.

MP - Degenerate State

$$CommonState(r) : \mathbb{T},\ r \mapsto |\mathbb{F}_r| > 0 \land \forall f \in \mathbb{F}_r : Static(f) \tag{2.44}$$

Component Method A class declares a method that takes an object of the same class as its single parameter.

DPC - Method Signature Information

$$\begin{aligned} &ComponentMethod(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists m \in \mathbb{M}_r : \\ &\qquad |parameters(m)| = 1 \\ &\qquad \land \pi_1(parameters(m)) = r \end{aligned} \tag{2.45}$$

Compound Box A class with exactly one non primitive instance field.

MP - Wrappers

$$CompoundBox(r) : \mathbb{T},\ r \mapsto |\mathbb{F}_r| = 1 \wedge \exists f \in \mathbb{F}_r : \neg Static(f) \wedge \neg Primitive(f) \quad (2.46)$$

Concrete Products Returned A method returns objects that belong to subclasses extending the class that represents the declared method return type.

DPC - Return Information

$$\begin{aligned} &ConcreteProductReturned(c,p) : \mathbb{T}^2,\ (c,p) \mapsto \\ &\quad \exists m \in \mathbb{M}_c, \exists s \in \mathbb{ST}_m : \\ &\qquad Ancestor(type(m), p) \\ &\qquad \wedge\, type(s) = p \\ &\qquad \wedge\, s \text{ is the return point} \end{aligned} \quad (2.47)$$

Conglomeration A type contains a method that calls another method within the same type.

EDP - Method Call

$$\begin{aligned} &Conglomeration(r) : \mathbb{T},\ r \mapsto \\ &\qquad \exists m1, m2 \in \mathbb{M}_r^* : m1 \neq m2 \wedge Invokes(m1, m2) \end{aligned} \quad (2.48)$$

Data Manager A class where all methods are either setters or getters.

MP - Data Managers

$$DataManager(r) : \mathbb{T},\ r \mapsto \forall m \in \mathbb{M}_r : Setter(m) \vee Getter(m) \quad (2.49)$$

Delegate A type contains a method that delegates a proportion of its task to a method in a different type.

EDP - Method Call

$$\begin{aligned} &Delegate(so, ta) : \mathbb{T}^2,\ (so, ta) \mapsto \\ &\qquad so \neq ta \\ &\qquad \wedge\, \exists m1 \in \mathbb{M}_{so}, \exists m2 \in \mathbb{M}_{ta} : \\ &\qquad\quad Invokes(m1, m2) \\ &\qquad\quad \wedge\, name(m1) \neq name(m2) \end{aligned} \quad (2.50)$$

Delegated Conglomeration A type contains a method that calls another method within the same type via a utility object of the same type.

EDP - Method Call

$$\begin{aligned} &DelegatedConglomeration(r) : \mathbb{T}, r \mapsto \\ &\qquad \exists f \in \mathbb{F}_r : type(f) = r \\ &\qquad\quad \wedge\, \exists m1, m2 \in \mathbb{M}_r : m1 \neq m2 \wedge Invokes(Invokes(m1, f), m2) \end{aligned} \quad (2.51)$$

Deputized Delegation A type contains a method that delegates a proportion of its task to a class within the same hierarchy.

EDP - Method Call

$$\begin{aligned} TrustedDelegation(so, ta) : \mathbb{T}^2,\ (so, ta) \mapsto \\ Sibling(so, ta) \\ \wedge \exists f \in \mathbb{F}_{so} : type(f) = ta \\ \wedge \exists m1 \in \mathbb{M}_{so}, \exists m2 \in \mathbb{M}_{ta} : \\ Invokes(Invokes(m1, f), m2) \\ \wedge\, name(m1) \neq name(m2) \end{aligned} \tag{2.52}$$

Deputized Redirection A type contains a method that redirects a proportion of its task to a class within the same hierarchy.

EDP - Method Call

$$\begin{aligned} DeputizedRedirection(so, ta) : \mathbb{T}^2,\ (so, ta) \mapsto \\ Sibling(so, ta) \\ \wedge \exists f \in \mathbb{F}_{so} : type(f) = ta \\ \wedge \exists m1 \in \mathbb{M}_{so}, \exists m2 \in \mathbb{M}_{ta} : \\ Invokes(Invokes(m1, f), m2) \\ \wedge\, name(m1) = name(m2) \end{aligned} \tag{2.53}$$

Designator An interface with absolutely no members.

MP - Degenerate State and Behavior

$$Designator(r) : \mathbb{I},\ r \mapsto \mathbb{A}_r = \emptyset \tag{2.54}$$

Extender A class which extends the inherited protocol, without overriding any methods.

MP - Inheritors

$$\begin{aligned} Extender(r) : \mathbb{C},\ r \mapsto \\ \mathbb{M}_r \neq \emptyset \\ \wedge \forall m1 \in \mathbb{M}_r, \exists m2 \in \mathbb{M}_r^+ : \\ \neg Public(m1) \vee \neg Overrides(m1, m2) \end{aligned} \tag{2.55}$$

Extend Method A type contains a method that overloads an existing parent method.

EDP - Method Call

$$\begin{aligned} ExtendMethod(sup, sub) : \mathbb{T}^2,\ (sup, sub) \mapsto \\ Ancestor(sup, sub) \\ \wedge \exists m1 \in \mathbb{M}_{sub}, \exists m2 \in \mathbb{M}_{sup} : Refines(m1, m2) \end{aligned} \tag{2.56}$$

Function Object A class with a single public instance method and at least one instance field.

MP - Degenerate Behavior

$$\begin{aligned}&FunctionObject(r) : \mathbb{T},\ r \mapsto\\&\quad |\mathbb{F}_r| > 0 \wedge |\mathbb{M}_r| = 1\\&\quad \wedge \exists m \in \mathbb{M}_r : \neg Static(m) \wedge Public(m)\end{aligned} \tag{2.57}$$

Function Pointer A class with a single public instance method, but with no fields.

MP - Degenerate Behavior

$$\begin{aligned}&FunctionPointer(r) : \mathbb{T},\ r \mapsto\\&\quad \mathbb{F}_r = \emptyset \wedge |\mathbb{M}_r| = 1\\&\quad \wedge \exists m \in \mathbb{M}_r : \neg Static(m) \wedge Public(m)\end{aligned} \tag{2.58}$$

Inheritance A type reuses the interface and implementation of its ancestors.

EDP - Type Relation

$$Inheritance(sup, sub) : \mathbb{T}^2,\ (sup, sub) \mapsto Ancestor(sup, sub) \tag{2.59}$$

Instance In Abstract Class An abstract class has a reference to another class.

DPC - Instance Information

$$\begin{aligned}&InstanceInAbstractClass(a, r) : \mathbb{T}^2,\ (a, r) \mapsto\\&\quad Abstract(a)\\&\quad \wedge a \neq r\\&\quad \wedge \exists f \in \mathbb{F}_r : type(f) = r\end{aligned} \tag{2.60}$$

Joiner An empty interface joining two or more super-interfaces.

MP - Degenerate State and Behavior

$$\begin{aligned}&Joiner(r) : \mathbb{I},\ r \mapsto\\&\quad \mathbb{A}_r = \emptyset\\&\quad \wedge \exists i1, i2 \in \mathbb{I} :\\&\qquad \wedge i1 \neq i2\\&\qquad \wedge \neg(Ancestor(i1, i2) \vee Ancestor(i2, i1))\\&\qquad \wedge Ancestor(i1, r) \wedge Ancestor(i2, r)\end{aligned} \tag{2.61}$$

Immutable A class with several instance fields, which are assigned exactly once, during instance construction.

MP - Degenerate State

$$\begin{aligned}&Immutable(r) : \mathbb{T},\ r \mapsto \\ &\quad |\{f \in \mathbb{F}_r : \neg Static(f) \wedge Final(f)\}| > 1\end{aligned} \tag{2.62}$$

Implementor A class that has at least one method and in which all public methods override inherited abstract methods.

MP - Inheritors

$$\begin{aligned}&Implementor(r) : \mathbb{C},\ r \mapsto \\ &\quad \mathbb{M}_r \neq \emptyset \\ &\quad \wedge \forall m1 \in \mathbb{M}_r, \exists m2 \in \mathbb{M}_r^+ : \\ &\quad\quad \neg Public(m1) \vee (Overrides(m1, m2) \wedge Abstract(m2))\end{aligned} \tag{2.63}$$

Instance In Abstract Referred A method of a class implementing Instance In Abstract Class invokes a method on the declared instance.

DPC - Method Body Information

$$\begin{aligned}&InstanceInAbstractReferred(a, r) : \mathbb{T}^2,\ (a, r) \mapsto \\ &\quad InstanceInAbstractClass(a, r) \\ &\quad \wedge \exists s \in \mathbb{ST}_a, m \in \mathbb{M}_r : Invokes(a, m)\end{aligned} \tag{2.64}$$

Leaf Class A subclass extends a superclass that contains a Component Method but does not override it.

DPC - Method Set Information

$$\begin{aligned}&LeafClass(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists p \in \mathbb{T} : Ancestor(p, r) \\ &\quad\quad \wedge m1 \in \mathbb{M}_p : \\ &\quad\quad\quad |parameters(m1)| = 1 \\ &\quad\quad\quad \wedge \pi_1(parameters(m1)) = p \\ &\quad\quad\quad \wedge \nexists m2 \in \mathbb{M}_r : Overrides(m2, m1)\end{aligned} \tag{2.65}$$

Multipile Trusted Redirection A class containing Trusted Redirection EDP within a cycle.

DPC - Method Body Information

$$MultipleTrustedRedirection(so, ta) : \mathbb{T}^2,\ (so, ta) \mapsto TrustedRedirection^*(so, ta), \tag{2.66}$$

where $_^*$ states that the pattern is executed within in a loop on a collection.

Node Class A subclass extends a superclass that contains a Component Method and which it redefines.

DPC - Method Set Information

$$\begin{aligned} &NodeClass(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists p \in \mathbb{T} : Ancestor(p, r) \\ &\quad\quad \wedge\ m1 \in \mathbb{M}_p : \\ &\quad\quad\quad |parameters(m1)| = 1 \\ &\quad\quad\quad \wedge\ \pi_1(parameters(m1)) = p \\ &\quad\quad\quad \wedge\ \exists m2 \in \mathbb{M}_r : Overrides(m2, m1) \end{aligned} \tag{2.67}$$

Outline A class where at least a method invokes an abstract method belonging to the same class.

MP - Data Managers

$$\begin{aligned} &Outline(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists m1, m2 \in \mathbb{M}_r : m1 \neq m2 \\ &\quad\quad \wedge\ Invocation(m1, m2) \\ &\quad\quad \wedge\ Abstract(m2) \end{aligned} \tag{2.68}$$

Overrider A class that has at least one method and in which all public methods override inherited non-abstract methods.

MP - Inheritors

$$\begin{aligned} &Overrider(r) : \mathbb{C},\ r \mapsto \\ &\quad \mathbb{M}_r \neq \emptyset \\ &\quad \wedge\ \forall m1 \in \mathbb{M}_r, \exists m2 \in \mathbb{M}_r^+ : \\ &\quad\quad \neg Public(m1) \vee (Overrides(m1, m2) \wedge \neg Abstract(m2)) \end{aligned} \tag{2.69}$$

Parent Product Returned A method returns a reference to the parent class of its declaring class.

DPC - Return Information

$$\begin{aligned} &ParentProductReturned(c, p) : \mathbb{T}^2,\ (c, p) \mapsto \\ &\quad \exists m \in \mathbb{M}_c : type(m) = p \wedge Ancestor(p, c) \end{aligned} \tag{2.70}$$

Pool A class which declares only static final fields, but no methods.

MP - Degenerate State and Behavior

$$Pool(r) : \mathbb{T},\ r \mapsto \mathbb{M}_r = \emptyset \wedge \forall f \in \mathbb{F}_r : Static(f) \tag{2.71}$$

Private Self Instance A class has a private instance of itself. Access to this instance can occur only from within the same class.

DPC - Instance Information

$$PrivateSelfInstance(r) : \mathbb{T},\ r \mapsto \exists f \in \mathbb{F}_r : type(f) = r \wedge \neg Public(f) \tag{2.72}$$

Protected Instantiation All the constructors within a given class are declared private.

DPC - Method Signature Information

$$ProtectedInstantiation(r) : \mathbb{T},\ r \mapsto \forall c \in \mathbb{CO}_r : public \notin \pi_2(c) \tag{2.73}$$

Pseudo Class A class which can be rewritten as an interface: no concrete methods, only static fields.

MP - Data Managers

$$PseudoClass(r) : \mathbb{T},\ r \mapsto \forall m \in \mathbb{M}_r : Abstract(m) \wedge \forall f \in \mathbb{F}_r : Static(f) \tag{2.74}$$

Pure Type A class with only abstract methods, and no static members, and no fields.

MP - Data Managers

$$\begin{aligned} &PureType(r) :,\ r \mapsto \\ &\quad \mathbb{F}_r = \emptyset \wedge \mathbb{M}_r \neq \emptyset \\ &\quad \wedge \forall m \in \mathbb{M}_r : Abstract(m) \wedge \neg Static(m) \end{aligned} \tag{2.75}$$

Record A class in which all fields are public and methods are declared.

MP - Data Managers

$$Record(r) : \mathbb{T},\ r \mapsto |\mathbb{M}_r| = 0 \wedge \forall f \in \mathbb{F}_r : Public(f) \tag{2.76}$$

Recursion A type contains a method that solves a problem by recursively calling it self.

EDP - Method Call

$$\begin{aligned} &Recursion(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists m \in \mathbb{M}_r : Invokes(m, m) \end{aligned} \tag{2.77}$$

Redirect A type contains a method that redirects a tightly related task to a similar method in a different type.

EDP - Method Call

$$\begin{aligned} &Redirect(so,ta) : \mathbb{T}^2,\ (so,ta) \mapsto \\ &\quad so \neq ta \\ &\quad \wedge \exists m1 \in \mathbb{M}_{so}, \exists m2 \in \mathbb{M}_{ta} : \\ &\quad\quad Invokes(m1,m2) \\ &\quad\quad \wedge name(m1) = name(m2) \end{aligned} \tag{2.78}$$

Redirected Recursion A type contains a method that solves a problem by recursively calling itself via a utility object of the same type.

EDP - Method Call

$$\begin{aligned} &RedirectedRecursion(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists f \in \mathbb{F}_r : type(f) = r \\ &\quad\quad \wedge \exists m \in \mathbb{M}_r : Invokes(Invokes(m,f),m) \end{aligned} \tag{2.79}$$

Reference To Abstract Class A class attribute is a reference to an abstract class.

DPC - Instance Information

$$\begin{aligned} &ReferenceToAbstractClass(r,a) : \mathbb{T}^2,\ (r,a) \mapsto \\ &\quad Abstract(a) \\ &\quad \wedge \exists f \in \mathbb{F}_r : type(f) = a \end{aligned} \tag{2.80}$$

Restricted Creation A class with no public constructors, and at least one static field of the same type as the class.

MP - Controlled Creation

$$\begin{aligned} &RestrictedCreation(r) : \mathbb{T},\ r \mapsto \\ &\quad \forall c \in \mathbb{CO}_r : \neg Public(c) \\ &\quad\quad \wedge \exists f \in \mathbb{F}_r : Static(f) \wedge type(f) = r \end{aligned} \tag{2.81}$$

Retrieve A class uses a Source to obtain an instance for a non local object (Retrieved) hence acts as Sink.

EDP - Object Elements

$$\begin{aligned} &Retrieve(si,so,re) : \mathbb{T}^3,\ (si,so,re) \mapsto \\ &\quad \exists f \in \mathbb{F}_{si} : type(f) = so \\ &\quad\quad \wedge \exists s \in \mathbb{ST}_{si}, \exists a \in \mathbb{A}_{so} : Invokes(s,a) \wedge type(a) = re \end{aligned} \tag{2.82}$$

Revert Method A type contains a method that provides no implementation details but reuses the parents implementation.

EDP - Method Call

$$\begin{aligned} &RevertMethod(sup, sub) : \mathbb{T}^2,\ (sup, sub) \mapsto \\ &\quad Ancestor(sup, sub) \\ &\quad \land \exists m1 \in \mathbb{M}_{sub}, \exists m2 \in \mathbb{M}_{sup} : \\ &\quad\ \ signature(m1) = signature(m2) \\ &\quad\ \ \land \exists s1 \in \mathbb{M}_{m1} : Invokes(s1, m2) \land \nexists s2 \in \mathbb{M}_{m1} : s1 \neq s2 \end{aligned} \tag{2.83}$$

Same Interface Container A class contains a set or a list of elements that are compatible with the same interface of the declaring class.

DPC - Instance Information

$$\begin{aligned} &SameInterfaceContainer(h, r) : \mathbb{T}^2,\ (h, r) \mapsto \\ &\quad h \neq r \\ &\quad \land f^* \in \mathbb{F}_h : type^*(f^*) = r \\ &\quad \land \exists p \in \mathbb{T} : Ancestor(p, h) \land Ancestor(p, r), \end{aligned} \tag{2.84}$$

where $_^*$ describes a collection variable or the element type of a collection.

Same Interface Instance A class contains a reference to an object whose type is compatible with the same interface of the declaring class.

DPC - Instance Information

$$\begin{aligned} &SameInterfaceInstance(h, r) : \mathbb{T}^2,\ (h, r) \mapsto \\ &\quad h \neq r \\ &\quad \land \exists f \in \mathbb{F}_h : type(f) = r \\ &\quad \land \exists p \in \mathbb{T} : Ancestor(p, h) \land Ancestor(p, r) \end{aligned} \tag{2.85}$$

Sampler A class with one or more public constructors, and at least one static field of the same type as the class.

MP - Controlled Creation

$$\begin{aligned} &RestrictedCreation(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists c \in \mathbb{CO}_r : Public(c) \\ &\quad \land \exists f \in \mathbb{F}_r : Static(f) \land type(f) = r \end{aligned} \tag{2.86}$$

Single Self Instance A class maintains a unique instance of itself, no matter if it is static or not.

MP - Instance Information

$$SingleSelfInstance(r) : \mathbb{T},\ r \mapsto \exists f \in \mathbb{F}_r : type(f) = r \tag{2.87}$$

Sink A class whose methods do not propagate calls to any other class.

MP - Data Managers

$$Sink(r) : \mathbb{T},\ r \mapsto \nexists m1 \in \mathbb{M}_r, m2 \in \mathbb{M} \setminus \mathbb{M}_r : Invokes(m1, m2) \tag{2.88}$$

Stateless A class with no fields other than static final ones.

MP - Degenerate State

$$Stateless(r) : \mathbb{T},\ r \mapsto \forall f \in \mathbb{F}_r : Static(f) \wedge Final(f) \tag{2.89}$$

State Machine An interface whose methods accept no parameters.

MP - Data Managers

$$StateMachine(r) : \mathbb{T}.\ r \mapsto \mathbb{M}_r \neq \emptyset \wedge \forall m \in \mathbb{M}_r : parameters(m) = \emptyset \tag{2.90}$$

Static Self Instance A class has a static instance of itself. Therefore this instance is unique inside the system.

DPC - Instance Information

$$StaticSelfInstance(r) : \mathbb{T},\ r \mapsto \exists f \in \mathbb{F}_r : type(f) = r \wedge Static(f) \tag{2.91}$$

Taxonomy An empty interface extending another interface.

MP - Degenerate State and Behavior

$$\begin{aligned} &Taxonomy(r) : \mathbb{I},\ r \mapsto \\ &\quad \mathbb{A}_r = \emptyset \\ &\quad \wedge \exists i \in \mathbb{I} : Ancestor(i, r) \end{aligned} \tag{2.92}$$

Template Implementor A method calls at least an abstract method within its body.

DPC - Class Declaration Information

$$\begin{aligned} &TemplateImplementor(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists t \in \mathbb{T} : Ancestor(t, r) \wedge TemplateMethod(t) \end{aligned} \tag{2.93}$$

Template Method A method calls at least an abstract method within its body.

DPC - Method Body Information

$$\begin{aligned} &TemplateMethod(r) : \mathbb{T},\ r \mapsto \\ &\quad \exists am \in \mathbb{M}_r, s \in \mathbb{M}_r : Abstract(am) \wedge Invokes(s, am) \end{aligned} \tag{2.94}$$

Trait An abstract class which has no state and defines at least one template method.

MP - Data Managers

$$Trait(r) : \mathbb{T},\ r \mapsto \mathbb{F}_r = \emptyset \wedge \exists m \in \mathbb{M}_r : Abstract(m) \tag{2.95}$$

Trusted Delegation A type contains a method that delegates a proportion of its task to a method within the same class hierarchy.

EDP - Method Call

$$\begin{aligned} TrustedDelegation(so, ta) : \mathbb{T}^2,\ (so, ta) \mapsto \\ FamilyHead(ta, so) \\ \wedge \exists f \in \mathbb{F}_{so} : type(f) = ta \\ \wedge \exists m1 \in \mathbb{M}_{so}, \exists m2 \in \mathbb{M}_{ta} : \\ Invokes(Invokes(m1, f), m2) \\ \wedge\, name(m1) \neq name(m2) \end{aligned} \tag{2.96}$$

Trusted Redirection A type contains a method that redirects a tightly related task to a similar method within the same class hierarchy.

EDP - Method Call

$$\begin{aligned} TrustedRedirection(so, ta) : \mathbb{T}^2,\ (so, ta) \mapsto \\ FamilyHead(ta, so) \\ \wedge \exists f \in \mathbb{F}_{so} : type(f) = ta \\ \wedge \exists m1 \in \mathbb{M}_{so}, \exists m2 \in \mathbb{M}_{ta} : \\ Invokes(Invokes(m1, f), m2) \\ \wedge\, name(m1) = name(m2) \end{aligned} \tag{2.97}$$

3. Candidate Sampling

The input of the candidate sampling are the extracted class features, which are the role mappings of the micro-structures. Output of this stage are design pattern candidate role mappings, i.e., potential instances of a predefined design pattern along with the features extraction in the previous stage. Figure 3.1 depicts this process which is an important step to enable the detection of design patterns. As discussed, role mappings link concrete classes to specific pattern roles, and these links spans a search space of

$$C = \binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{3.1}$$

potential mappings where n is the amount of available classes, and k the number of roles that need to be mapped. Obviously the search space is beyond computational viability for real world applications and the problem gets worse if partial role-mappings are considered i.e., not all roles have class assignments. In this situation the search space increases to

$$C^* = \sum_{k=0}^{n} \binom{n}{k} = 2^n \tag{3.2}$$

candidate mappings, representing the power set of n elements. This huge search space motivates the process of candidate sampling which searches for potential mappings. In contrast to filtering, sampling does not inspect the entirety of the search space but uses domain knowledge to retrieve only the useful proportion of it.

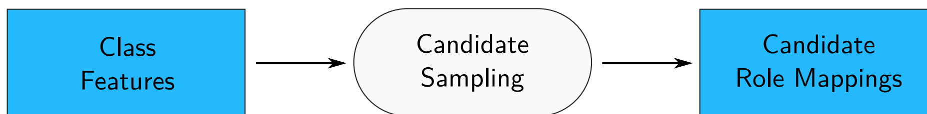


Figure 3.1.: The candidate sampling process takes the class features of all available classes and returns candidate mappings. Candidate mappings fulfill the basic structural properties a pattern has hence qualifies for the detailed analysis via the inference method. Each pattern has its own sampler that traverses the ASG and collects classes (nodes) that make up a candidate mapping.

As mentioned in Chapter 1.2 many DPD systems feature this sampling phase, nevertheless implementations vary as different inference approaches impose different requirements. Niere et al. [36] frame the 2-step filtering and inference step into a bottom-up and top-down strategy using a rule based approach. The detection process is tailored around rules that are applied to an annotated abstract syntax graph. These rules have different levels that are dictated by their dependencies to other rules. Rules without any dependencies are level 1 rules and can be applied to a graph node immediately. Level 2 rules or greater must resolve

their dependencies first in order to be applied. First the bottom-up strategy applies level 1 rules that are stored within a priority queue. Processed rules are removed from the queue but may add new rules that depend on it, e.g., after applying the Inheritance rule the higher-level Composite rule is added to the queue and executed next. The Composite rule represents the detection of the Composite pattern and thus kickstarts the top-down phase. The top-down phase resolves now all remaining rule dependencies for the Composite rule until the additional top-down priority queue is empty or the resolution fails. This effectively handles the size of the search space by iteratively applying lower level rules to add new top-level rules and resolving these in a greedy fashion. The phases are interleaved such that certain condition pre-filter the space and trigger the actual inference step as certain requirements are fulfilled.

Anotoniol et al. [4] uses a multistage filter/extraction process in which class level metrics and the shortest path in a graph are reformulated into constraints. Each design pattern usually inhibits a set of class level metrics that need to be fulfilled in order for the pattern to work. For example an instance of the Chain of Responsibility pattern must delegate a message to classes of the same hierarchy at some point, otherwise the semantics of the pattern would be contradicted. Thus a class level metric might be the number of delegations and the constraint formulation would be $x > 0$ with x being the number of delegations. This pre-filters all the class combinations for the shortest-path constraint evaluation in which patterns are interpreted as graphs with a shortest path. The remaining system is now analyzed for minimal sets that fulfill the shortest path criterion imposed by the pattern. Although effectively reducing the search space, classes that participate in multiple roles and structural differences in the pattern instance can not be found by the extraction affecting the recall of the procedure.

Gueheneuc et. al [19] proposed the fingerprinting method mentioned in Chapter 2 as means to find candidate mappings. The advantage of fingerprinting is that it is computationally efficient and easy to implement nevertheless lacks a high recall for all roles. Alhusain et al. [1] used similar metrics as Gueheneuc and applied feature selection methods to find the most import among them. After that a rule learner was used to identify conditions a class must meet in order to be an instance of a certain role. Another possibility they tried out was to train a neural network that can identify whether a class matches a designated role or not. The result of these methods where role mappings but without the actual pattern context. This enables preliminary filtering of classes and reducing the effective search space.

MARPLE-DPD, implemented by Fontana and Zanoni et al. [5], incorporates a Joiner module with the purpose of extracting candidate instances of a design pattern. An early version of the tool used micro structures in form of joiner rules that need to be fulfilled to find candidate mappings. The system under inspection is transformed into a graph and the possible classes are sampled from the graph according to the rules. This approach is very similar to the approach used within this work as it can be easily understood, implemented and is very intuitive. Later versions [57] use the resource description framework to represent the system under inspection and the pattern as graph that are then matched via graph matching algorithms.

Uchiyama et al. [52] trained neural networks to detect whether a class is of a certain role or not. The input are metrics and the output of the network is the probability that the given class (represented by its features) is an instance of a certain role. Detected roles are not grouped by their pattern but instead all available roles (12 roles) are judged at the same time. Depending on these results the user may enter the patterns that should be detected. The algorithm than follows the ASG of system to collect the remaining classes that comply with the patterns structure.

3.1. Heuristic Search

From Equation 3.1 it can be easily concluded that the naive approach is not feasible even for small systems. One problem with the naive search is that design patterns are made up of multiple roles leading to many different combinations that need to be checked. Another problem is that the naive search inspects combinations of classes that are illogical from a software engineering perspective, i.e., class combinations in which the classes have no relationship what so ever. Heuristic search is a way to mitigate this issues by only selecting sensible combinations of classes and ignoring the remainder. Similar to the methods presented above, heuristic search uses structural and behavioral information about classes to decide whether a certain combination is a candidate for a pattern or not.

Heuristic search follows certain paths within the ASG and collects nodes along the way. Instead of searching a combinatorial space, heuristic search only operates on the already very limited space an ASG can span. This mitigates the problem that unrelated classes are grouped together. Each pattern defines an **anchor role** that serves as entry point for the search. Anchor roles are usually related to the Inheritance MS as most of the DPs are defined over class inheritance. The sampling process inspects all classes starting with the property the anchor role imposes on the classes (e.g., superclass). If a matching class is found, it will continue to traverse the ASG such that all roles of the target pattern have a sensible mapping or not. For some patterns it makes sense to allow partial mappings, i.e., not all standard roles have a class assign because two ore more roles have been merged together. This is a very typical implementation detail that software developers use thus a common situation. For instance the Creator and Concrete Creator from the Factory Method pattern are often merged together as it allows the developers to add a default behavior to the factories. The samplers are derived by declaring an anchor role for a given pattern and moving along the direct connected nodes within the ASG. Constraints are added to the traversal such that the patterns communication protocol is considered, but all other possible combinations are ignored. This is done in a recursive fashion from role to role within a pattern until all roles (or a subset for partial mappings) have a class mapping. This very easy method is best explained by example which are given in the next sections. Each section explains the heuristic search for a specific pattern where the examples are based on the work of Gamma and his colleagues [15]. *Descendants* will be used to refer to all children in a type hierarchy, i.e., all subclasses if object A is a class, or all sub interfaces or realizations and their subclasses if A is an interface.

3.1.1. Adapter Sampling

Figure 3.2 depicts the workflow of the sampler for the adapter pattern. There are three roles that need to be found, Target, Adapter and Adaptee. Client was removed from the original set of roles as it is a universal role applicable to all patterns that does not provide any useful information in the context of DPD.

1. **Anchor role:** Is every class or type that functions as Target, i.e., has descendants.
2. Extracts all descendants from Target that function as Adapters.
3. Extracts all associations from Adapters that function as Adaptees and groups them to their respective Adapters.
4. Additionally extracts all descendants of the Adaptees and groups them to their respective Adaptee.

The sampler uses the Inheritance and the Association MSs and does not allow partial mappings as all roles are crucial for the pattern's semantics. Grouping the endpoint of relations to the respective sources is often not needed in order to build candidate mappings but greatly reduces the amount of candidates. For instances it would be possible to build candidates by simply collecting all subclasses from Target and all associations (plus subclasses) from the Adapters, without respecting their source. Nevertheless this again creates a huge amount of candidates that are (i) illogical from a ASG view point, (ii) unnecessarily increasing the amount of candidates leading to more false positives in the detection.

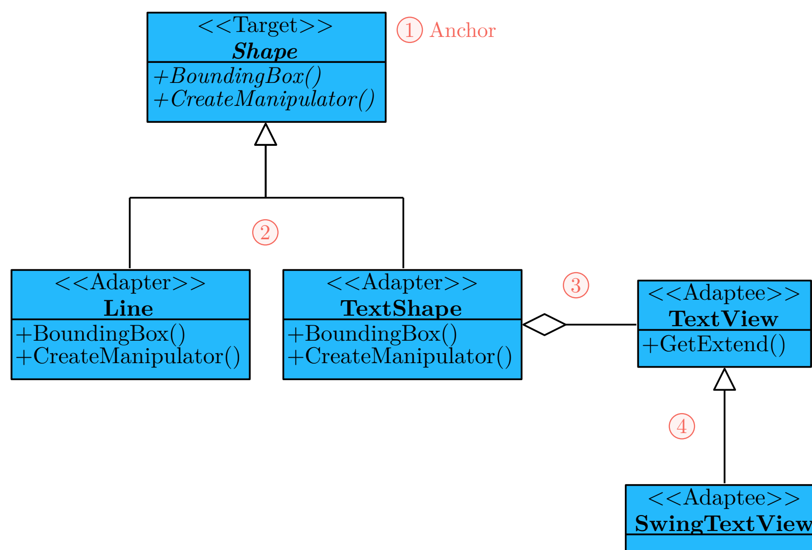


Figure 3.2.: The adapter sampler (i) starts form the anchor which is a class that has descendants, (ii) collects all descendants, (iii) collects all associations of the descendants and groups them to their respective Adapter and (iv) collects all descendants of the Adaptees and groups them to their respective Adaptee.

3.1.2. Composite Sampling

The workflow of the composite sampler is rather simple and presented in Figure 3.3. Again, the Client role has been removed which permits the exclusive use of the Inheritance MS to

find all candidate mappings. In fact the removal of the client role is essential as too many classes would fit this role causing the set of candidate mappings to explode. The sampler does not allow partial mappings, thus merging of Component and Composite is not considered as a candidate mapping.

1. **Anchor role:** Is every class or type that functions as Component, i.e., has descendants.
2. Extracts all descendants from Component that function as Leafs or Composites.

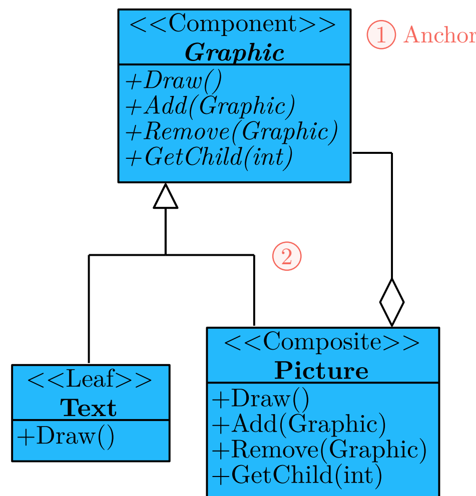


Figure 3.3.: The composite sampler (i) starts form the anchor which is a class that has descendants and (ii) collects all descendants which are used as Leaf or Composite.

3.1.3. Decorator Sampling

Component functions as an anchor role for the decorator sampler as given in Figure 3.4. Again, it would be possible to simply collect the descendants form the anchor to compute all combinations which definitely covers the patterns communication structures. Nevertheless this increases the amount of resulting candidate mapping by orders of magnitudes thus more restrictive constraints are applied. The constraints enable the sampler to distinguish between Concrete Component and Decorator via a self association. Merging the Decorator and Concrete Decorator is allowed as this is a very typical variant of the pattern.

1. **Anchor role:** Is every class or type that functions as Component, i.e., has descendants.
2. Extracts Concrete Components and Decorator from the descendants grouped by the self association constraint.
3. Extracts all descendants of the Decorator as Concrete Decorator grouped by the their respective Decorator.

3.1.4. Factory Method Sampling

Many projects implement some sort of factory mechanism, either simple static methods in the class, builder classes with a fluent Application Programming Interface (API), dedicated

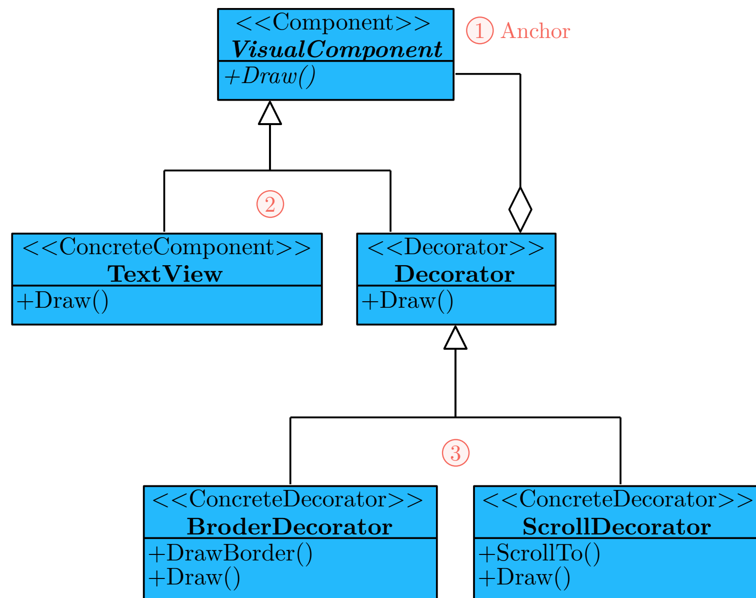


Figure 3.4.: The decorator sampler (i) starts form the anchor which is a class that has descendants, (ii) collects all descendants and groups the Decorators and Concrete Components depending whether they have a self association or not, (iii) collects all descendants of the Decorators and groups them to their respective Decorator.

factory classes or via dependency injection containers. The variety is even further enriched if all implementation variants are considered making the detection process harder and even human oracles might not conclude to the same decision. The role Creator and Product are the abstractions and Concrete Creator and Concrete Product their respective implementations. Being the most complicated sampler of all, the Factory Method sampler needs 4 dedicated steps (shown in Figure 3.5) to find most of the candidate mappings and allows for partial mappings in which Creator and Concrete Creator are merged into one role. It uses the Inheritance MS and a modified Association in which constructor calls, method invocations and method return types are considered. The Products are grouped by their Creator to reduce the overall result set but may cause the removal of true candidate mappings.

1. **Anchor role:** Is every class or type that functions as Creator, i.e., has descendants.
2. Extracts all descendants from Creator that function as Concrete Creator.
3. Extracts all associations from the Creators (including the abstract Creator) that function as Product and groups them to their respective Creator.
4. Extracts parents and descendants from Concrete Product and reorganizes them into a proper hierarchy.

3.1.5. Singleton Sampling

The singleton is the smallest design pattern as it hosts only one role, thus does not suffer from any combinatorial explosion caused by multiple role mappings. All classes are candidates for the Singleton pattern except for pure interfaces that cannot be initialized. This leads to a very small set of candidates nevertheless still smaller than with the naive approach because

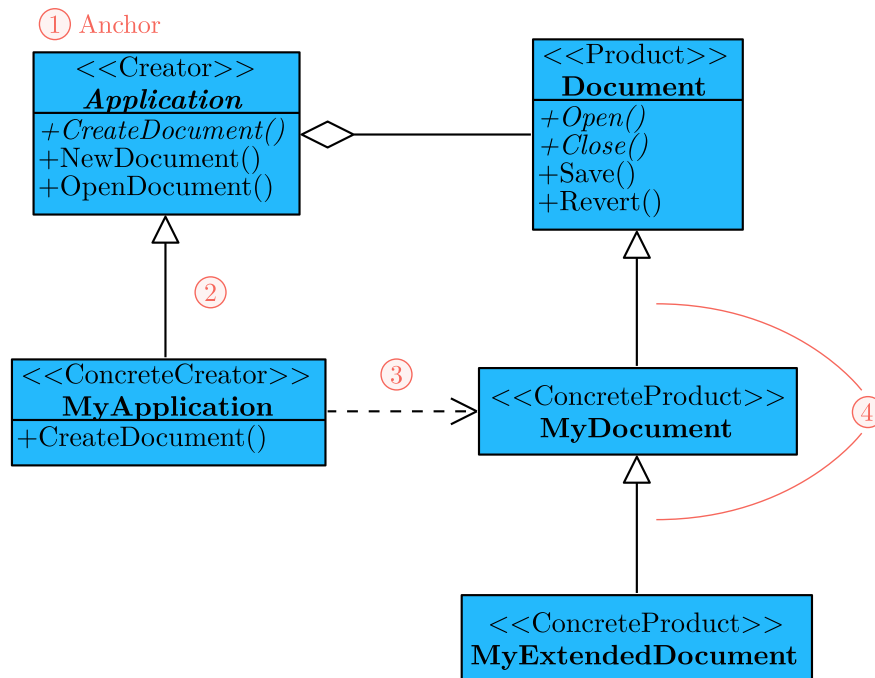


Figure 3.5.: The factory method sampler (i) starts form the anchor which is a class that has descendants, (ii) collects all descendants as Concrete Creators, (iii) collects all associations of the Creators as Products and, (iv) collects all related class from the products as Products.

the removal of interfaces. The anchor is of course the one Singleton role and is every class that can have a constructor.

3.1.6. Template Method Sampling

Template Method has the roles Abstract Class and Concrete Class that are connected via inheritance. This simple structure allows the sampler the exclusive use of the Inheritance MS which allows efficient candidate sampling as Figure 3.6 shows. No partial mappings are allowed as it would break the purpose of the pattern itself.

1. **Anchor role:** Is every class or type that functions as Abstract Class, i.e., has descendants.
2. Extracts all descendants from Abstract Class that function as Concrete Class.

3.1.7. Technical Overview

Candidate sampler are again an instance of the Strategy pattern where each sampler described in Section 3.1.1 - 3.1.6 takes the role of a Concrete Strategy illustrated in Figure 3.7. A Detector has a factory method to create its sampler which is then used in the BaseDesignPatternDetector that implements the basic detection process. The interface hierarchy of the DesignPatternDetector can be seen as an instance of the Template Method as getFeatures and createSampler are steps needed in the detect algorithm. In sum there are 6 candidate sampler implementations since each design pattern detector provides one implementation. Samplers

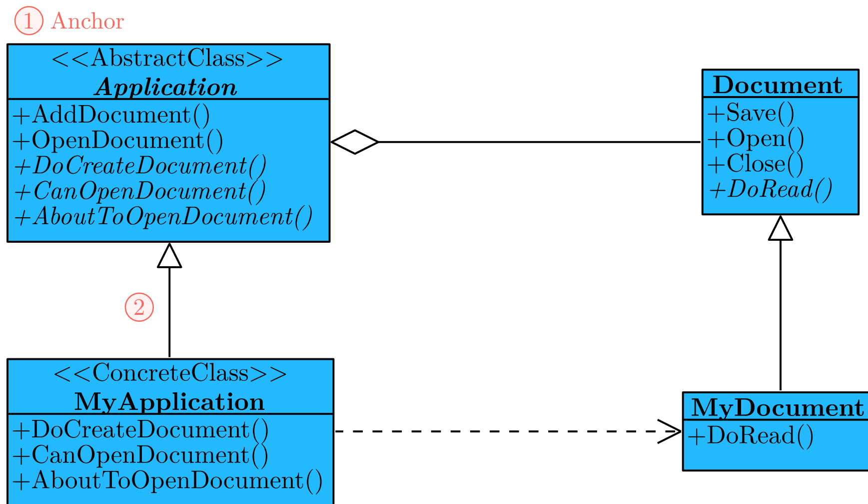


Figure 3.6.: The template method sampler (i) starts form the anchor which is a class that has descendants, (ii) collects all descendants that function as Concrete Class

themselves implement the sample method which takes a type (a type from the system under inspection) and the detection context which is a collection of all previously detected patterns along with the ASG. It then checks whether the given type is an instance of the predefined anchor role and proceeds the sampling process accordingly if the conditions are met. This is executed for each type within the system thus the complexity is linear with the system size. Result of the algorithm is a list of RoleMappings that function as the candidates.

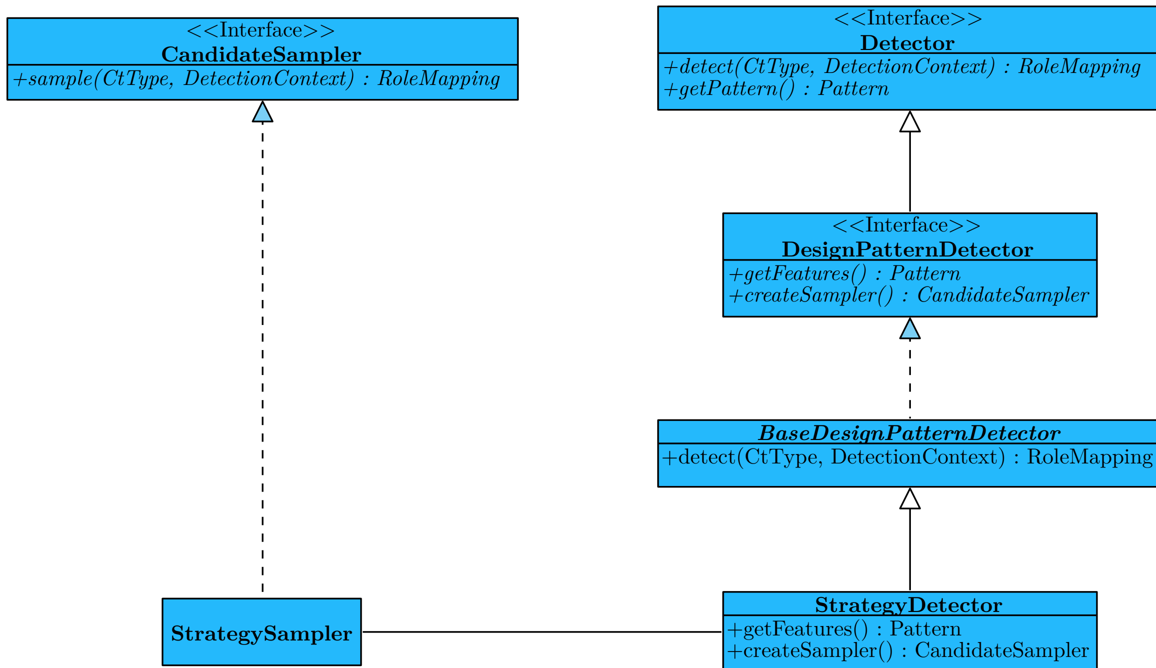


Figure 3.7.: Candidate samplers can be created via their detectors take a type which is checked whether it is an instance of the predefined anchor role.

Table 3.1.: Table of the cumulated naive space and heuristic space (H. Space) of all projects. Mapping Count are all known instance of the pattern in the projects. Naive Space are all candidate mappings within the naive space. H. Space (FP/TP) are all candidate mapping within the heuristic space. H. Proportion is the proportion of the heuristic space in the naive space. Recall is the coverage of TP to all known TP of the H. Space.

Pattern	Known TP	Naive Space	H. Space FP	H. Space TP	H. Space	H. Proportion	Recall
Adapter	172	46,192,862	106,937	164	107,101	$2.318 \cdot 10^{-1}$	.953
Composite	210	46,192,862	32,126	210	32,336	$6.968 \cdot 10^{-2}$	1.00
Decorator	170	5,432,630,268	688	160	848	$1.560 \cdot 10^{-5}$	.941
Factory Method	462	5,435,630,268	21,857	91	21,948	$4.040 \cdot 10^{-4}$	.196
Singleton	13	2,012	1858	13	1,871	$9.299 \cdot 10^{+1}$	1.00
Template Method	82	336,664	2,099	82	2,181	$6.478 \cdot 10^{-1}$	1.00

3.2. Evaluation

Table 3.1 contains the cumulated mapping count over all projects for each pattern, i.e., all known instances of the patterns (peer reviewed instances), along with several metrics describing the size of the naive space and the heuristic space (H. Space) spanned by the algorithms above. Note that some instances were removed from the original data set because they are faulty or inaccessible (for more details see Section 3.2.1). The first column describes the amount of peer reviewed instances of each pattern where Factory Method is leading with 462 and Singleton is last with 13 instances. Of course these numbers do not reflect the popularity of the patterns as the Singleton pattern has 13 unique instances in contrast to Factory Method having only 7 unique instances. As discussed above, unique instances describe instances of the pattern that are independent form each other, i.e., do not share the same primary roles but provide a complete new variant for a pattern. The amount of true positive samples for each pattern is rather small in the context of machine learning, but respectable for the small amount of projects they were drawn from. All projects together comprise 2012 types, which lead, depending on the amount of roles, to a maximum space of $5,432,630,268$ candidate mappings and a minimum size, for the one-role pattern (Singleton), of 2013 candidates. The heuristic space spans at most 106.937 candidates which might be an alarming amount, nonetheless this captures the cumulative space spanned by 8 non-trivial projects. The actual size of the H. Space with respect to the naive space is given by the H. Proportion where the best reduction was reached by the Decorator sampler that reduces the initial space to a fraction of $1.56 \cdot 10^{-5}$ candidates. Interestingly the second best reduction was reached by the Factory Method sampler which is also impaired with the worst recall as the sampler only reaches a recall of 19.69% (discussion below). Despite this extreme, all samplers have virtually 100% recall thus fulfill their purpose of finding all true positives and still providing a manageable set of candidates.

Figure 3.8 provides information of the number of candidates per project in form of box plots on a logarithmic scale ordered by the number of roles per pattern. The green and orange boxes are results from the naive approach and the blue boxes from the heuristic search. Expectations

for the one role case (Singleton) are that naive and heuristic approach perform equally good, which is in general the case. The H. Space with $\mu = 233.87; \sigma = 151.91$ Candidates per Project (C/P) is on average 17.75 C/P smaller as the naive space ($\mu = 251.62; \sigma = 155.97$) since Java interfaces can never implement the Singleton pattern (as they are not instantiable), thus are not sampled. Surprisingly the Template Method has on average only 38.75 C/P more than the Singleton space, where in contrast the naive approach already reaches 2 magnitudes higher with $\mu = 42,083.0; \sigma = 51,693.39$ C/P. This indicates that the heuristic search does **not** produce candidates sets that grow exponential with the amount of roles and types within a system. This seems to be a premature conclusion as the amount of candidates increases by nearly a factor of 2 for the Adapter pattern ($\mu = 13,387.62; \sigma = 16,759.96$) which has 3 roles, nevertheless this is the biggest space spanned by heuristic search. Composite ($\mu = 4042.0; \sigma = 4907.78$), Decorator ($\mu = 106.000; \sigma = 129.2440$) and Factory Method ($\mu = 2743.500; \sigma = 2356.23$) span a smaller candidate space although having the same or more amount of roles. The big leap between Template Method and Adapter (and the remaining patterns) is caused by the association that need to be considered during sampling, and of course these are usually plentiful.

There is no reliable way to predict or analytically compute the true growth function from the plain number of types as the growth mostly depends on the systearchitecture. The high standard deviation of the results come from the fact that the projects are of different sizes and that the amount of candidates strongly rely on the internal inheritance hierarchies. Figure 3.9 provides more insight into the distribution of candidates per projects where again the heuristic approach is given in blue. The naive approach indicates an exponential growth as the means grow approximately linear with the size of the projects. This trend can not be seen with the heuristic approach, nevertheless big boxes indicates a rather high deviation per pattern within a project. The bottom whisker is nearly equal for all project caused by the singleton pattern. An exception to this trend is the Lexi project as it is a rather small project (96 types) with many inner classes and only implementing one of the 6 patterns - the Singleton pattern. Top whisker reach from 10^2 to 10^4 which is an acceptable amount of candidates to process but also implies a possible large amount of false positives in the inference step. This is an issue that need to be addressed in order to get acceptable prediction results and may be handled by removing secondary roles from the sampling processes as several other research groups already do.

3.2.1. Discussion of Noteworthy Instances

All samplers are developed with the goal to provide maximum coverage in terms of true candidate mappings with the smallest possible amount of mappings. This is done by walking the ASG in such a way that most true mappings are found. Tailoring the sampling process to the available dataset might introduce a bias towards the presented implementation variants, nevertheless the used relationships are very basic and usually inherent in the patterns rational. An example for this situation would be the inheritance in the template method which could be rewritten into a composition in which the abstract methods are delegated to an unrelated class

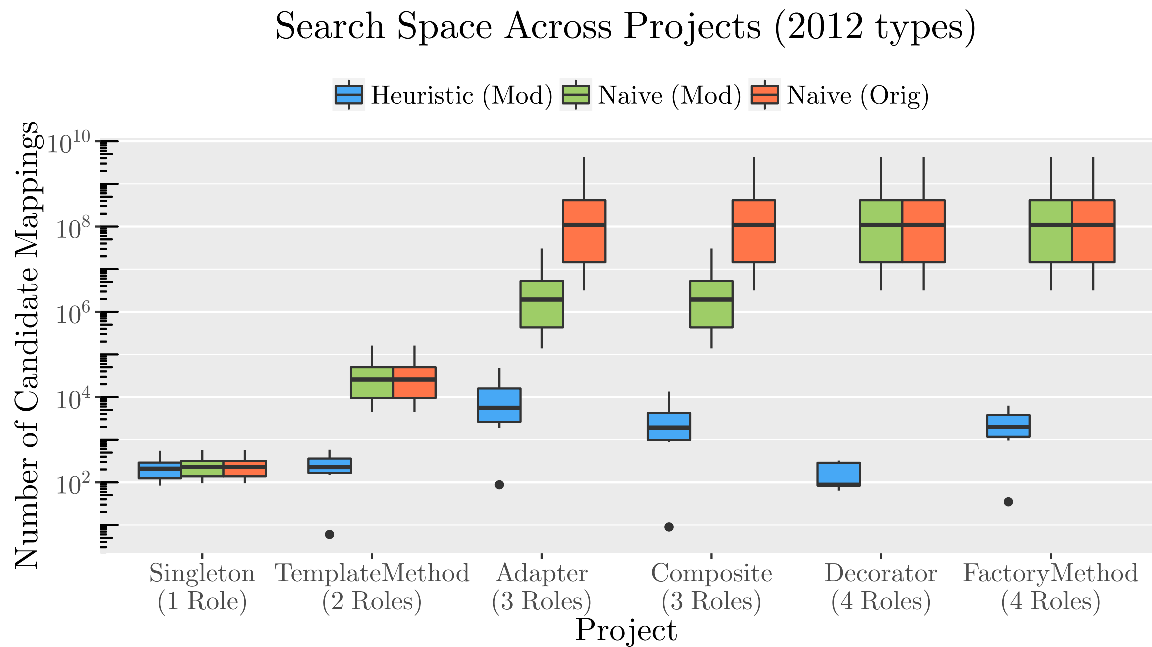


Figure 3.8.: The box plots show the difference between the number of candidate mappings across projects for each pattern on a logarithmic scale. The naive approach produces constantly larger amounts of candidate mappings with a maximum over 10^9 candidates. Difference between Mod and Orig is that Mod does not contain the Client role, whereas Orig describes the pattern in its original definition. The amount of candidates produced by heuristic search is in the region of $10^2 - 10^4$ and depends on the the projects ASG density and the pattern structure.

that provides the desired functionality. Though technically possible, this specific rewriting nonetheless stands in the face of the patterns intent to provide a skeleton for a closed algorithm in which only a few steps are deferred to a subclass. The abstract class is the skeleton, promising one specific algorithm that is only completed with its subclasses. This fact does not hold for the composition implementation as the abstract class would potentially be complete even without the deferring class. Furthermore the semantic coupling between deferring class and abstract class is more loose such that they do not represent one specific algorithm. Obviously the added semantics may be defined ambiguous or developers simply interpret it in a different way such that certain variants are seen as an instance of a pattern or not, depending on the viewer. Another situation is where the pattern implementation is "polluted" with anti-patterns or bad practices that make the detection even for experts difficult. This of course is the main problem of design pattern detection, but the open question on "where to draw the line" between variants of a pattern and the fact that a role mappings is not proper instance of a pattern remains.

For instance one peer reviewed Adapter instance is not included into the result set of its sampler because the example does not include a class for the Adaptee role. Many software engineers would not classify this mapping as a proper instance of the Adapter pattern simply because of the missing Adaptee that the Adapter is trying to adapt. The absence of this key role is illogical in the context of the pattern and the entire instance does not fulfill the patterns purpose, thus should not have been included into the peer reviewed repository at all.

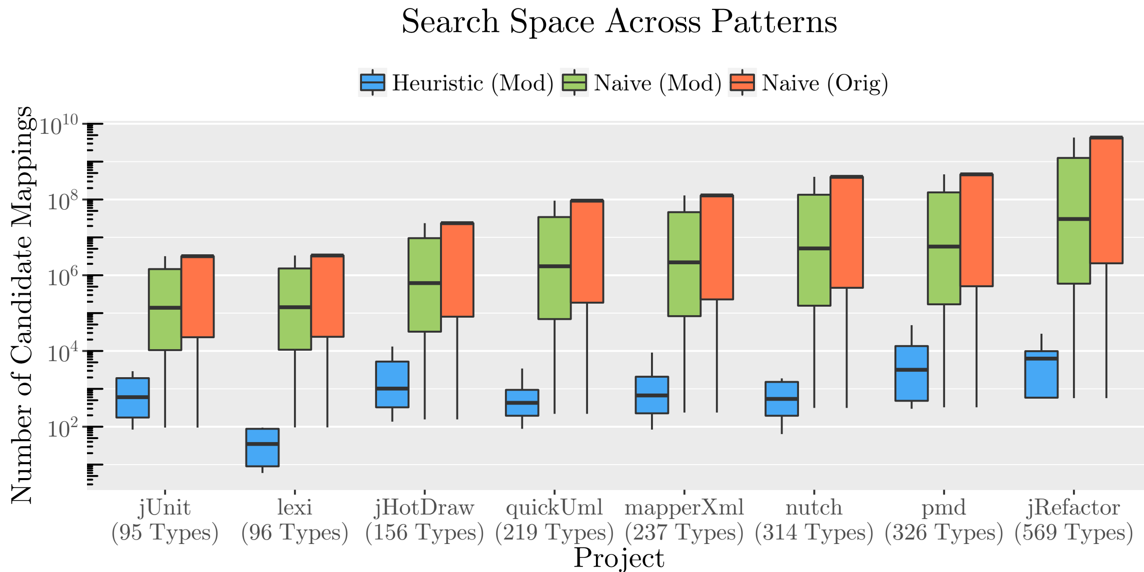


Figure 3.9.: The box plots show the difference between the number of candidate mappings across patterns for each project on a logarithmic scale. Heuristic search does not react exponential to the number of types within a system as the naive approach. The seemingly equal lower whisker is caused by the singleton pattern which always has constant candidate size.

Another debatable example would be given by a not sampled Decorator instance in which the Concrete Component and Decorator are mapped to the same class. The advantage of the Decorator pattern is that new responsibilities can be attached to a Component during runtime but attaching the same responsibilities to an object only adds a level of indirection without any additional effects. Obviously this instance can be ignored which would increase the sampler recall to nearly 1.

The worst performance is given by the Factory Method sampler which only reaches a recall of .196, but why is that so? As already mentioned, the way objects are constructed is manifold and of course there are also various implementation variants of the Factory Method. One interesting example for a variant of the pattern is given by Listing 3.1 in which the "xHandle" is the Product. The peer reviewed mappings contain instances that map ConnectionHandle and NullHandle to the Concrete Product, and NodeFigure to Concrete Creator role. There is nothing wrong with this relationship at the first glance but obviously the implementation is a rather bad example of the pattern. Not only obscures the return type the actual constructed product, but also the fact that the method creates multiple products (Product ← Handle) degrades the quality of the interfaces as it is not clear what the factory creates in this examples (from an API user perspective). This problem is of course partially related with the fact that the code base is rather dated thus no generics could be used in order to clarify the actual product. The sampler is nonetheless able to correctly find these instances. More interesting is the fact that there exist mappings for each direction (EastHandle, WestHandle, . . .) for the same creator despite the fact that RelativeLocator.west() returns an instance of the class RelativeLocator. These classes are siblings and do not stand in a child parent relationship. Listing 3.2 contains the WestHandle implementation which is also parameterized via the

Listing 3.1.: Node Figure implements a Factory Method for the ConnectionHandle and NullHandle but also for WestHandle. WestHandle is not explicitly defined as dependency but the ConnectionHandle that is equally parameterized hence behaves similar.

```
public class NodeFigure extends TextFigure {
  ...
  public Vector handles() {
    ConnectionFigure prototype = new LineConnection();
    Vector handles = new Vector();
    handles.addElement(new ConnectionHandle(this, RelativeLocator.east(), prototype));
    handles.addElement(new ConnectionHandle(this, RelativeLocator.west(), prototype));
    handles.addElement(new ConnectionHandle(this, RelativeLocator.south(), prototype));
    handles.addElement(new ConnectionHandle(this, RelativeLocator.north(), prototype));

    handles.addElement(new NullHandle(this, RelativeLocator.southEast()));
    handles.addElement(new NullHandle(this, RelativeLocator.southWest()));
    handles.addElement(new NullHandle(this, RelativeLocator.northEast()));
    handles.addElement(new NullHandle(this, RelativeLocator.northWest()));
    return handles;
  }
  ...
}
```

relative locator indicating that the products do not use inheritance but rather composition to share implementation details. The fact that there is a mapping between Concrete Product and WestHandle (as the other directions) that are merely based on a composition seems to be very debatable and points out the problem that human experts do not have a shared agreement of valid pattern instances. Truly, an example in which the Concrete Products type is defined via a composition of a third type (RelativeLocator), wrapped into a fourth type (ConnectionHandle) stored in a fifth type (Vector) should not be included and considered as a valid instance of the pattern as only the top level type should be mapped as product. Only the pattern's semantic should be considered during detection and not the software developers intension as the implementation might diverge too far to be considered as instance. This must not mean that the implementation is flawed by some way, but only that a more conservative human classification of the instances is desirable. There are many closely related example (> 300) that share the similar issues and excluding them would improve the recall to over 0.9.

In sum it can be concluded that most patterns can be reliable sampled via heuristic search in such a way that most or all true role mappings are within the result set of the sampler, and that the result set has a manageable size. "Manageable size" might be an ambiguous phrase but appropriate in this situation as the amount of candidates that can be handled via the detection module (see Chapter 5) largely depends on the computational efficiency of the detection module itself. The detection module presented in this work handles $50,000$ candidates in $2s$ on a Graphical Processing Unit (GPU) (in this case, Nvidia GeForce GTX 970), additionally it can also be extended to run on an entire cluster with multiple GPUs. Truly, the comparison between the combinatorial space is not challenging but most studies focus on the detection phase thus detailed reports that could be used as comparison are missing. Furthermore it should be noted that the benchmark with the companion web tool

Listing 3.2.: Locator handle take a relative position which allows

```
class WestHandle extends LocatorHandle {
  WestHandle(Figure owner) {
    super(owner, RelativeLocator.west());
  }
  public void invokeStep (int x, int y, int anchorX, int anchorY, DrawingView view) {
    Rectangle r = owner().displayBox();
    owner().displayBox(
      new Point(Math.min(r.x + r.width, x), r.y),
      new Point(r.x + r.width, r.y + r.height)
    );
  }
}
```

by Arcelli [6] addresses the problem of human expert agreement by introducing pattern instance scores. Each expert can comment and provide a relevance score for the found pattern instances, such that a more reliable corpus of peer-reviewed patterns can be build over the course of time.

4. Feature Normalization

Source code is transformed into an ASG, features in form of micro-structures are extracted and possible candidates from the system sampled such that the detection module can give a prediction on whether the given candidate is an instance of a pattern or not. The goal of the feature normalization is, based on the features of the candidate role mappings, to normalize the inhomogeneous feature vectors/matrices into a matrix of fixed size. This is needed because micro-structures are not plain scalar values but actually subgraph of a certain aspect from the system each having their own amount of roles. This process is shown in Figure 4.1 in which the candidate role mappings (and their respective features) are normalized into feature maps.

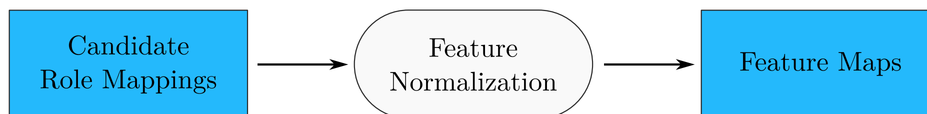


Figure 4.1.: The feature normalization process receives the candidate role mappings and their features and returns normalized features maps. Feature maps represent the different sub-graphs of the ASG in a homogeneous form such that inference methods that need fixed sized input tensors can handle them.

Most machine learning models are designed to take fixed sized vectors as input and return a fixed sized vector as output (flat methods). To learn from a software system in an end-to-end fashion, i.e., ASG as input and fixed sized vector as output, models need to be extended by means that enable them to understand what structure is. Kernel methods, e.g., Support Vector Machines [9], can be extended by an appropriate kernel that extracts the structural information of sub-trees in a directed tree [16]. Neural Networks can be extended to recursive neural networks that process trees by unfolding the graph structure to an encoding network where each node in the graph is computed by a feedforward network [18]. Despite the possibility of these extensions, and unmentioned methods, most direct approaches lack scalability as computations on generic subgraphs are often computationally intractable.

As mentioned in Chapter 2, micro-structures are named subgraphs that capture a specific amount of information from the entire graph and help to reduce the amount of subgraphs. Despite that they solve the problem of scalability, they do not solve the problem that most models are optimized for fixed sized vector inputs as they still represent graphs. As the example given in Figure 2.2 shows, the Inheritance and Retrieve MS have different amounts of roles (2 and 3), and in fact features beyond MS might even have more roles. The goal is now to find a representation that transforms all these subgraphs into a representation such that the resulting structure is independent of the amount of roles the features have, without loosing the original relational information the graphs contain.

4.1. Feature-Role Normalization

The goal of the normalization, more detailed, is to take n features each having m roles and map them to k design pattern roles such that the result is a matrix or a vector. Feature-Role Normalization (FRN) provides one approach to this problem by creating a $n \times k$ matrix out of the features of a candidate mapping. Figure 4.2 gives an overview of the mapping scheme in which columns are design pattern roles and rows represent micro-structures (features). The values of the matrix are role ids ($id \in \mathbb{N}$) that are globally unique within the entire system, i.e., no two roles of any pattern have the same values associated with. Zeros represent absent features meaning that the class mapped to the role does not participate in the micro-structure or does not have a certain characteristic. Recapitulating the running Strategy example in combination with Figure 4.2 would result in Compositor (R1 $\leftarrow$ Compositor) being an Abstract Interface (MS1), that provides a Template Method (MS2) and being a Superclass (MS3). The ConcreteStrategy (e.g., R2 $\leftarrow$ TeXCompositor) is a Subclass (MS3) of the Compositor but does not provide a Template Method or is an Abstract Interface. Context (R3 $\leftarrow$ Composition) does not participate in any of the given features thus all values for its column are zero. The resulting matrix is called feature map as it maps the sub-graphs of the system (micro-structures), which represent a certain characteristic (feature), onto the roles of the design pattern. The structure of the feature maps is fixed as their columns are dependent on the amount of roles the design pattern has, and the rows depend on the features, both being static at runtime. Sub-graph nodes (MS roles) are represented as values within the matrix, thus are theoretically infinite, and effectively map the uncertain aspect of the features, the amount of roles, onto the dynamic axis of the matrix.

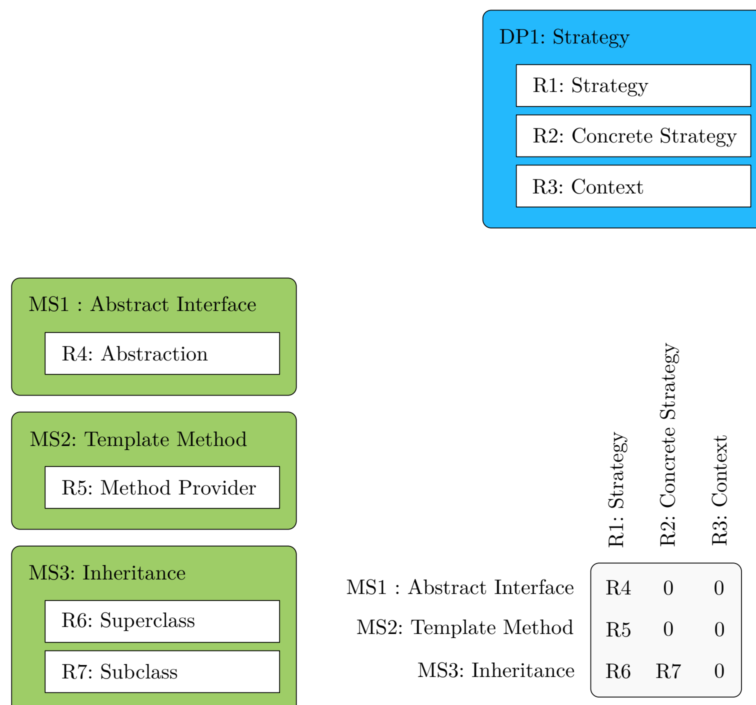


Figure 4.2.: The micro-structures are mapped on one dimension and the design pattern roles are mapped onto the second dimension. The domain of the matrix are the identifiers of the roles.

4.1.1. Issues

Normalizing subgraphs of different shape onto the graph of a pattern might impose certain restrictions onto the values or the feature map. These restrictions might be mitigated by changing the mapping scheme, and in fact, some of the presented schemes in Section 4.1.3 do not suffer of Issue I and II but may introduce their own trade-offs.

Issue 1: Feature Roles Associated to Unrelated Classes

Issue I is related with the noise Feature-Role Normalization might introduce into the feature maps. The values of the features are very precise in the sense that they are not produced by measuring a natural phenomenon, but by extracting deterministic properties from an artificial source (source code). This leads to very high Signal-to-Noise Ratio (SNR)

$$SNR = \frac{P_{signal}}{P_{noise}}, \tag{4.1}$$

in fact, the SNR should be infinity if the MS detectors are implemented without any defects. Off course this assumes that all features are perfectly uncorrelated **and** provide useful information in the context of the detection. An additional problem with sub-graphs as features is given by the fact that they usually have connections with classes that do not participate in a given candidate mapping. This causes ghost roles within the matrix, i.e., role values within the feature map that might be true on the global system scale, but are irrelevant for the given candidate mapping. Thus despite the precision of the data, feature maps can still contain information that is considered to be noise in the context of the role mapping. Figure 4.3 outlines the problem by removing the subclass relationship between the classes assigned to the Strategy and the ConcreteStrategy role. Strategy still has a superclass relationship but to a class that does not participate in the current role mapping (e.g., SimpleCompositor). The information is correct but is considered to be noise in the current context as feature maps should focus on the current role mapping, and not on the environment the entire system provides. Only features that represent a relationship between multiple DP roles are affected by this problem. One simple solution to the problem is to force the values of a feature to zero, if only one class of the current mapping participates in it. Figure 4.3 illustrates this solution in which a multivariate feature (Inheritance) does not provide context related information (left side) thus gets removed from the feature map (right side). In hindsight this means that instead of knowing that the Strategy class participates in a inheritance structure in that the others candidate classes do not participate, only the information is kept that none of the mapped classes are in the same inheritance structure. This of course comes along with a certain information loss, which must not necessarily mean that valuable information is thrown away. This is founded on the fact that the additional ghost values increase the amount of role collisions.

Issue 2: Role Collision

Issue 2 describes the problem that arises if a class participates in multiple instances of the same feature. Again the problem only is present if the feature relates multiple classes with

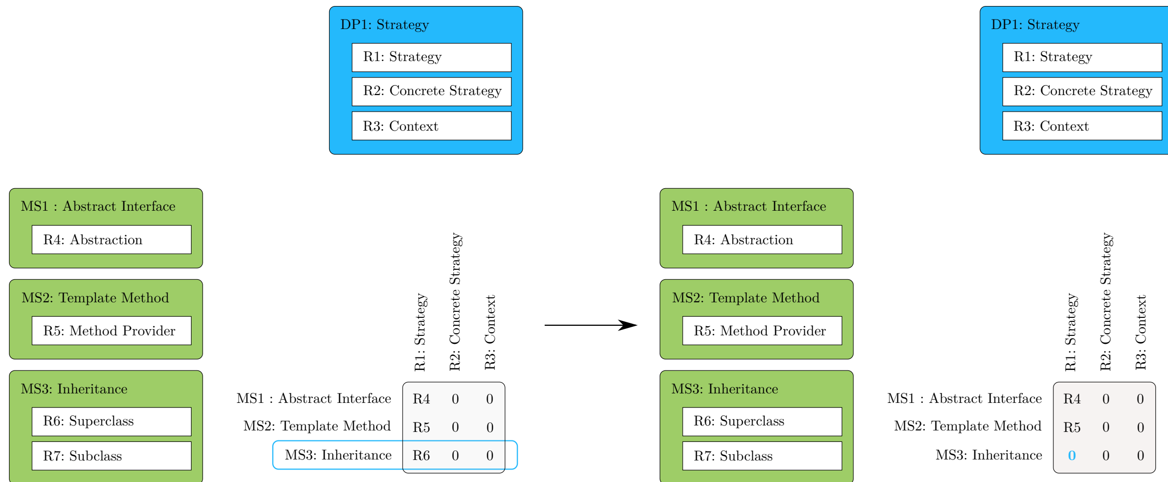


Figure 4.3.: Micro-structures with multiple roles are only considered if their roles trace at least to two roles of the current mapping.

each other. Figure 4.4 demonstrates this problem, where the Concrete Strategy is a subclass of Strategy but also a superclass of the Context. Though maybe not particular meaningful in the context of the Strategy Pattern, Issue 2 arise quite often and needs to be handled appropriately. A naive solution would be to duplicate the feature (rows) for each possible combination of DP role (in this case $2^3 = 8$ row per features), but this inefficiently increases the height of the feature map with respect to the amount of DP roles. Another way would be to consider only one relationship as valid and ignoring the others. Of course this would result in a huge information loss because of the removals.

The implemented solution uses a concept called virtual roles that represent multiple roles simultaneously. A virtual role is a composition of multiple real roles thus do not really exist in the context of their respective feature. A composition may only occur within the set of roles a MS defines, e.g., Superclass/Subclass is a valid virtual role in contrast Superclass/Abstraction is not. Superclass/Abstraction combines roles of different features thus would be nonsensical as Superclass cannot occur in the Abstract Interface MS and vice versa. Virtual roles are means to aggregate multiple instances of the same feature with the cost of losing the directional information of a relationship. This comes clear if the Inheritance is replaced by Association in Figure 4.4. For the inheritance case it clear that there is a linear inheritance structure between the classes mapped to R1, R2 and R3 as no circles are possible. This is not the case with an association as it would be possible that all three mapped classes relate to each other. The feature map would then contain the value for the virtual role, depicting the source and target of an association, in all columns. The directional information is shadowed by the coarse information a virtual role can represent, thus each role collision causes an information loss and should be avoided. Ghost values increase the amount of virtual roles as they sneak in information which is not essential in the current context but might shadow important directional information.

It is clear that an overwhelming amount of virtual roles would result in a non-negligible amount of information loss with respect to the relationships between columns. Figure 4.5 shows the prevalence of the top 5 virtual roles among different design patterns. For instance,

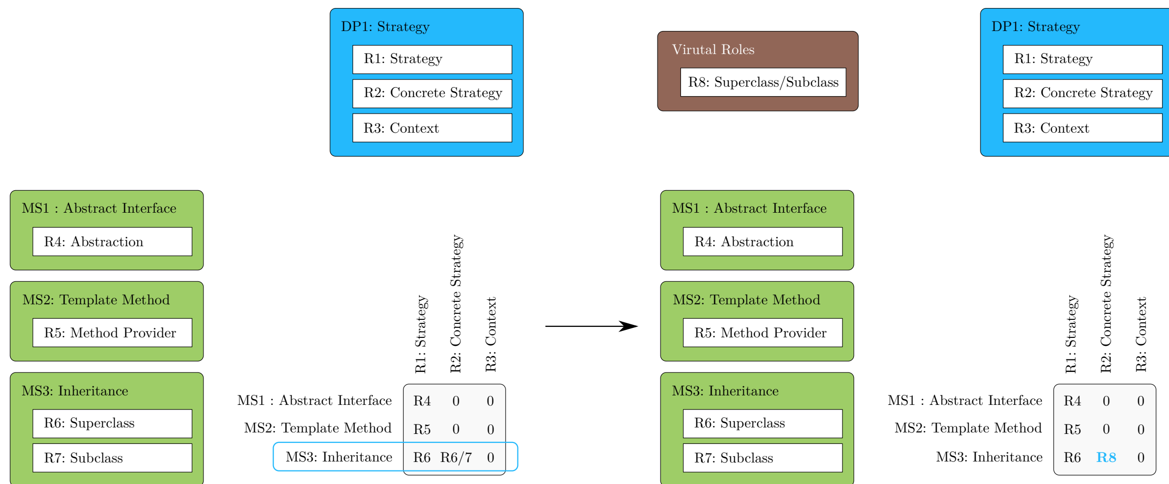


Figure 4.4.: Virtual roles are introduced for micro-structure roles that intersect. They represent multiple roles at the same time such that only the directional information is lost.

AssociationSource/AssociationTarget of the Association MS occurs in over 75% percent of the Singleton instances. This matches the implementation in which a class associates (static or dynamically) itself and offers a way to access its instance. A special case, that also reveals the main problem with virtual roles is the Compatible MS which nearly has 100% prevalence on all patterns. Of course, a class is assignment compatible to itself thus it will always be source and target at the same time. Not only will be the coefficients in the Compatible row be the same in every observation (indicating a dead feature), but also compatibility information between mapped classes is shadowed. Obviously, all prevalence values should hit 100% as classes are compatible to themselves, nevertheless not every instance of a DP must map every role. Often abstract roles are merged with their concrete roles e.g., the Decorator is often merged with the Concrete Decorator when no explicit interface is needed. These missing mappings cause the entries within the feature map to be 0, thus resulting in a prevalence of $\rho_{Decorator} = .812$ for the Compatible MS within the Decorator DP. It is not surprising that the unary nature of the Singleton pattern forces a lot of virtual roles. In the case of the Singleton pattern, virtual roles do even indicate whether a class is a Singleton or not, simply because of the high correlation between these. Features that capture the messaging behavior of classes and their respective objects also have a higher prevalence of virtual roles. For instance an implementation of a graph with nodes that store their connected nodes within themselves would cause virtual roles in the Association, Delegation and Deputized Delegation features. These virtual roles occlude valuable direction information within the feature map as in contrast to the Singleton case because of the multivariate nature of other pattern. Generally the amount of virtual roles is acceptable and noisy features like Compatible need to be removed from the feature set.

4.1.2. Properties of the Feature-Role Normalization

Normalization schemes enforce specific properties on the feature maps each influencing the learnability of the data. Figure 4.6 visualizes one Decorator feature map in which the column

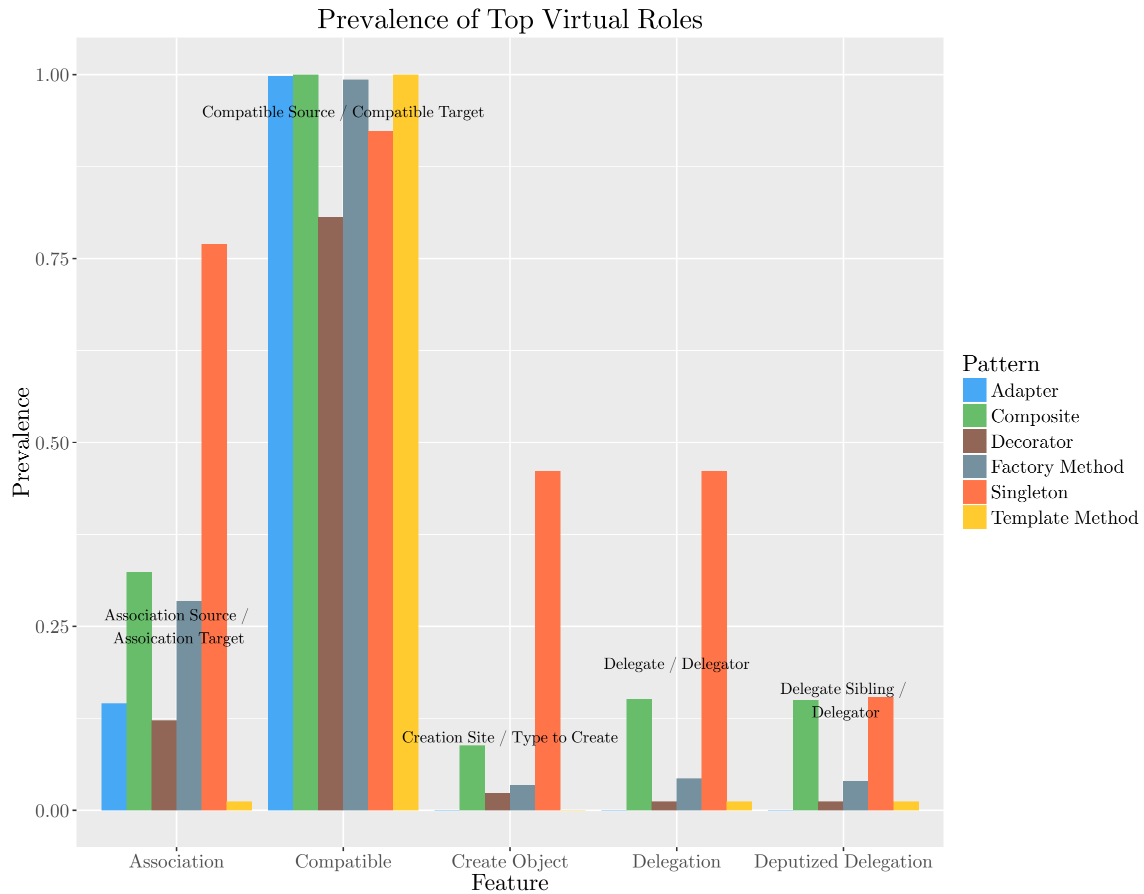


Figure 4.5.: The Compatible MS causes the most virtual roles for obvious reasons - every class is compatible to itself except if it is missing. Self associations, creations and delegations do also frequently occur indicated by the Singleton patterns virtual roles.

row structure becomes clear. Grey spots within the map reflect absent roles and the different shades of blue the specific role ids. The different intensities of the tiles come from their respective role ids thus form a gradient along the height. It is possible that two different micro-structures share a role id, which implies that they have the same roles (with similar semantics). The order of the columns is fixed but it is possible to train models that are order independent. The example in the feature map show that all classes mapped to their respective decorator roles participate in the same inheritance structure. Also there is a Aggregation relationship between Component and Decorator and a Delegation scheme between the classes. It is not possible to distinguish the different values via their color intensities because of the small differences between them within a row. This of course is only a visualization issue.

Sparsity

Sparsity is an important concept used in various different areas, e.g., signal processing, medical visualization, image processing, machine learning, etc. It represents data in which a small number of entries contribute the main proportion of the energy. Figure 4.6 illustrates this concept via a feature map produced by FRN, i.e., only real information causes a response

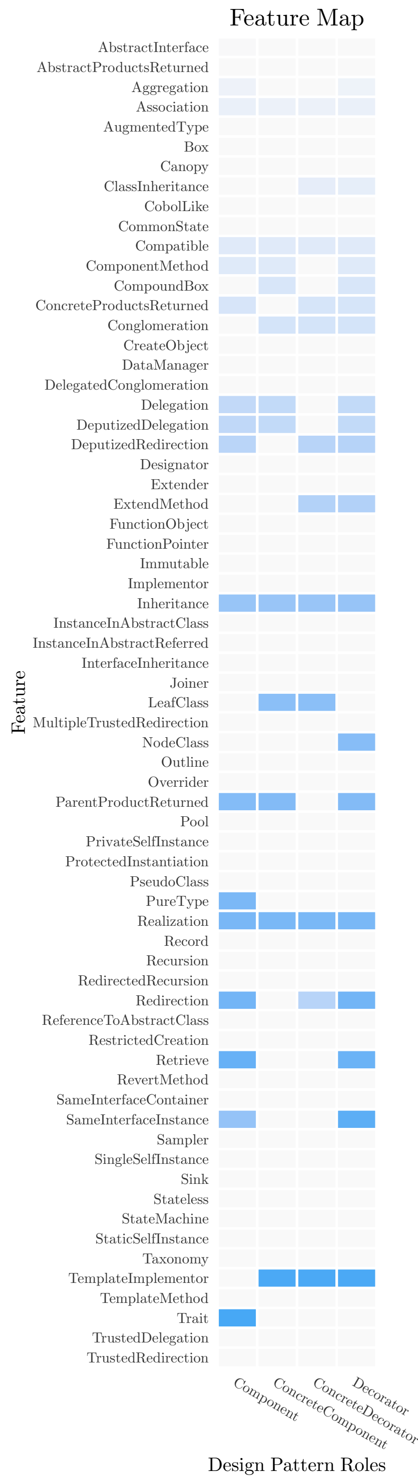


Figure 4.6.: The heatmap of a feature map produced by the feature-role normalization. Features are alphabetically sorted and role ids are integers in increasing order. Some roles might be reused in multiple patterns if they represent the same concept.

within the map all other values are zero. The implication of sparse data is often a high SNR as information is clearly presented. This is usually not the case in natural phenomenons as there is always a small amount of background noise caused by the measuring procedure and the environment. For instance background noise would cause in feature maps gray tiles to be blue depending on the amount and the intensity of the noise blurring the true information. Sparsity is enforced by the feature-role normalization, micro-structures and by the fact that DPD works on artificial data (source code). Table 4.1 presents the sparsity statistics in which the overall amount of feature maps is given in n, the mean for zero-values and non-zero values μ_x, standard deviation in σ, the proportion of non-zero values in the entire dataset and the Gini Index. The Gini Index, a function $Gini : \mathbb{R} \to [0, 1]$, is a measure of sparsity complying to many desirable properties (i.e., Robin Hood, Scaling, Rising Tide, Cloning, Bill Gates, Babies [23]), expressing perfect equality via a 0 and inequality by 1. First means that all values within the vector have the same coefficient and the second, for example, means that there is one infinite value with infinite zero values.

Gini Index and the Proportion of non-zero values (normalized to 1 as they represent the abstract concept of an identification number) indicate very sparse feature maps with a mean of $\mu_{Proportion_{\neq 0}} = 7.42\%$, $\mu_{Gini} = .914$. The table shows that the sparsity of the feature maps is rather independent of the pattern, i.e., the participating classes within the different patterns do not influence the sparsity of the feature maps. The composite pattern is an interesting case as its standard deviation differs from other the patterns. This is caused by the Decorator and Concrete Decorator role which are very often merged thus cause a higher deviation. Except for this peculiarity the sparsity of the features is rather constant among the patterns.

Table 4.1.: Mean and standard deviation of zero values and non-zero values within the feature maps along with the proportion and Gini index.

Pattern	n	$\mu_{=0}$	$\mu_{\neq 0}$	$\sigma_{=0,\neq 0}$	Proportion$_{\neq 0}$	Gini Index
Adapter	172	18.94	182.05	2.67	10.60	.940
Composite	216	34.51	166.48	6.45	5.82	.888
Decorator	170	35.27	232.72	15.19	7.59	.914
Factory Method	462	36.44	231.55	5.25	7.35	.915
Singleton	13	10.46	56.53	4.34	6.40	.902
Template Method	82	15.84	118.15	4.37	8.45	.924

Simplicity

The simplicity of the data directly correlates with its learnability. Intuitively, it is easier to train a model that learns a direct bijective mapping between inputs and outputs, than to learn a model that needs to perform a prime factorization before it can access the bijective mapped prime numbers the inputs are composed out. This example might by exaggerated but points out the importance of keeping the data simple as possible. When working with natural signals,

e.g., audio or visual signals, the data is usually transformed into some representation that capture the relationship of the signals characteristics. For instance audio signals are captured in terms of amplitude over time, but are converted via the Fast-Fourier Transform [10] into the spectrum that simplifies the analysis of frequencies. The decision to use micro-structures already imposes simplicity onto the data as it abstracts not only text but also the ASG in form of named subgraphs. Feature-Role Normalization transforms these subgraphs into a feature role independent matrix.

One form of simplicity is given by the sparseness discussed above as only a handful of subgraphs describe the relationship between the mapped roles. Secondly, the mapping uses the MS roles as values thus limits the effective range of the coefficients that can occur within the feature map. The coefficients range form 0 to 161 including all the virtual roles. This represents a very limited domain on which the models have to be trained compared to signals drawn from a biological/natural source. Another characteristic is that each row can draw only from a very limited set of values. For instance the row associated with the Abstract Interface feature can only contain one specific value. In the case of Inheritance, only two values within the row are possible namely the identifier for the role Superclass or Subclass. This is visible in Figure 4.6 where the patterns are sorted and the tiles get darker the further down they occur but stay constant along the width. These characteristics make feature maps very simple and predictable.

Flexibility

Flexibility states that the normalization can cope with an increasing amount of features and breaks up relationships that might impose computational limits. Feature-Role Normalization enforces that the number of facets or channels needed to encode one observation is independent of the amount of features. Each observation can be represented with a single feature map thus only one channel is needed. Different normalizations could result in multiple channels per observation, just like RGB images are encoded with three channels, and thus limit its computational tractability.

Furthermore, feature maps growth linear (along one dimension) with the number of features used in the detection process. Thus, additional features do not compromise the time and space effort needed to process one feature map. The last outstanding property of flexibility is that rows and columns of feature maps are order agnostic. This is important as candidate role mappings do not need to provide perfectly mapped roles but may make mistakes in their initial role mapping assumption. It does not matter whether the columns are in a specific order to identify the source and target relationship of a feature because each role has its dedicate value. This enables a wider variety of inference methods that work on feature maps.

4.1.3. Alternative Mapping Schemes

Several possible mappings schemes (normalization) were considered and the most promising (FRN) was implemented. Whether a normalization is more appropriate to learn from or not, can not be determined by theoretical means, especially when using black box methods as presented in Chapter 5. The situation gets even worse if the machine learning algorithm is

gradient based and may run into local minimum increasing the effort of finding a good model. That said, properties as described in Section 4.1.2 may help finding models that perform well thus should be perused. Due to time limitations, only one possible normalization scheme was implemented and tested, and others are subjected for future investigation.

Role-Role Normalization

Very similar to Feature-Role Normalization is the Role-Role Normalization (RRN) in which the m roles of the n features are mapped onto the k roles of the design pattern. The domain of the matrix is of binary or categorical nature, depicting whether a certain feature role holds for a DP role or not. Table 4.2 illustrates the feature map that is produced by RRN. Role ids are projected onto the rows such that only boolean values are needed in the map. This avoids role collisions (Issue 2) as each feature role has its own row, however source and target information is still occluded by intersecting mappings similar to FRN. In this example Strategy and Context is a Superclass but it is not clear which of the both the Concrete Strategy's parent is. This could be encoded into the values, for example each combination could be represented by an integer starting from 2 to allow booleans: $\{R1\} = 2$, $\{R2\} = 3$, $\{R3\} = 4$, $\{R1, R2\} = 5$, etc. Nevertheless this would enforce a column ordering and thus reduce the flexibility of the resulting feature map.

Table 4.2.: Example of the Role-Role Normalization feature map.

Micro-Structure	Role	Strategy	Strategy Concrete Strategy	Context
Abstract Interface	Abstractor	1	0	0
Inheritance	Superclass	1	0	1
	Subclass	0	1	1
Retrieve	Sink	0	0	1
	Source	1	0	0
	Retrieved	1	0	0

Another consideration is the resulting size of the feature map as the height of the map is now dependent on the number of roles the features have. This results in a $m \times k$ matrix with $m \geq n$, as opposed to FRN's $n \times k$ matrix. In the case of the currently implement version of the detection tools this would result in 161 rows per map if all 67 features are included. Figure 4.7 illustrates the data structural differences between FRN and RRN. Obviously both are very similar in their basic form and only differ by their actual content and height. The left data structure contains multitudes of values where the middle only has boolean or a very limited set of categorical coefficients. The middle data structure has roles along its height where the left has features. Additionally they differ in their actual height not visualized within this figure.

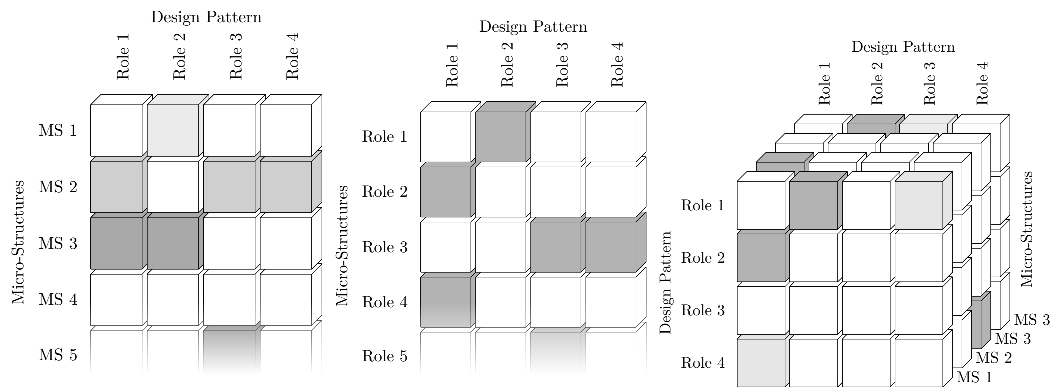
Feature-Channel Normalization

Feature-Channel Normalization (FCN) creates for each of the n features a $k \times k$ matrix, where k is the number of DP roles. The idea is similar to the work of Tsantalis et al. [51] in which sub-graphs of the ASG are represented via their adjacency matrix. Each feature (e.g., aggregation, inheritance, ...) forms its own $k \times k$ adjacency matrix and functions as channel within a feature map (similar to red, green, blue within an rgb-image). Thus, feature maps produced by FCN are volumes as in contrast to the feature maps produced by FRN and RRN. The data structure is given in Figure 4.7c in which the basic difference between the normalizations can be seen. Columns and rows represent the design pattern roles thus values within the matrix are always mirrored. The depth of the volume represents the features, i.e., the micro-structures that are used. Coefficients within the channels represent relational directions between DP roles, i.e, the edges within the ASG. A positive value is a *Source to Target*, and negative the inverse *Target to Source* relationship.

Table 4.3 illustrates an example of the Feature-Channel Normalization in which three channels are given: Aggregation, Inheritance and the Retrieve channel. A channel is read from row to column such that Concrete Strategy aggregates Strategy and Strategy is the Superclass of Concrete Strategy (conversely: Concrete Strategy is the Subclass of Strategy). The main advantage of this normalization is that the source/target information is preserved even if there are intersections between MS roles. This is paid with an increased space demand as FRN needs $n \times k$ coefficients, as in contrast to FCN with $k \times k \times n$ coefficients. Furthermore, FCN can not freely reorder columns independent from rows as it would break the encoded target source information. The fact that FCN produces volumes means that the used inference method must be able to process these, which limits the normalization's applicability. Simple down-projection of the volume, such that a greater variety of inference methods can handle the data, might destroy the information within the feature map. Note that this needs to be verified in a series of experiments, but the expectation is that the information represented by the volumes structure and its values cannot be preserved by the projection. FCN can be seen as the generalized version of FRN and RRN as it preserves all of the information extracted by micro-structures. FRN and RRN squashes the data into a matrix which causes the limitations with respect to role intersections within the data. FCN spans a volume such that intersections can be captured on an additional dimension.

Table 4.3.: Example of the Feature-Channel Normalization. Aggregation, Inheritance and Abstract Interface are the features that build up the tree channels of the feature map. Channels are square matrices that contain source/target information of the MS roles. For example Context aggregates a strategy or Strategy is the super class of Concrete Strategy. The value magnitude is the edge identification where 1 within Channel 1 would describe the Aggregator/Aggregate edge, or the Abstract Interface within Channel 3.

				Strategy	
Observation	Channel 1	**Aggregation**	Strategy	Concrete Strategy	Context
		Strategy	0	0	-1
		Concrete Strategy	0	0	0
		Context	1	0	0
				Strategy	
	Channel 2	**Inheritance**	Strategy	Concrete Strategy	Context
		Strategy	0	1	0
		Concrete Strategy	-1	0	0
		Context	0	0	0
				Strategy	
	Channel 3	**Abstract Interface**	Strategy	Concrete Strategy	Context
		Strategy	1	0	0
		Concrete Strategy	0	0	0
		Context	0	0	0



(a) Feature-Role Normalization feature map structure. The structure is flat and MS roles are expressed by the cell value indicated by the different shaded color. Height represents feature and width roles.

(b) Role-Role Normalization feature map structure. The structure is flat and directional information can be encoded into the values resulting in a very small value range.

(c) Feature-Channel Normalization feature map structure. Width and height represent both design pattern roles forming a square matrix. Features are stored in the depth where the values encode directional information that is mirror along the diagonal.

Figure 4.7.: Each feature normalization results in a specific structure and content of the feature map. FRN and RRN encode information in a flat structure incorporating information loss. FCN uses a third dimension to preserve the data but has higher space demands.

5. Design Pattern Detection

Figure 5.1.: The feature extraction process uses the feature maps and convolves over them. This results into a likelihood that the given map is of a certain pattern and ultimately to the boolean decision whether it is accepted or not.

The detection step takes the feature maps of the candidate role mappings and decides whether they are a valid instance of a specific pattern or not (Figure 5.1). This is done via a machine learning algorithm that trains a model which understands the inherent notions of the respective feature maps.

5.1. Machine Learning in a Nutshell

There are many problems that are closely related in their basic nature and some of these problems may be solved by exact deductive approaches. A deductive approach uses a system of formulas, axioms and rules of inference to find the **exact** solution to a class of problems (e.g., simple maximization problem). This implies that the system contains all the information such that the rules can be applied in a mechanical fashion. Nonetheless most real world problems cannot be fully specified or their computation is too expensive such that inductive approximative approaches are the only remaining option. Machine learning is an inductive approach that tries to build models/knowledge from previously observed data with the two main goals of getting insight into the data, and being able to make predictions about it. Simply speaking, machine learning is about learning from data.

Machine learning is split into to two categories: supervised and unsupervised methods. Supervised methods try to find the relationship between the input $\boldsymbol{X}$ and the output Y, where both are known beforehand. Unsupervised methods try to identify structural properties in $\boldsymbol{X}$ since no output Y is known beforehand. Classical tasks of supervised methods are classification and regression, where the first outputs class labels and the second numerical values. The predominate goal is to predict an output value for a given input value, thus DPD is a typical example for a classification task. Clustering, projection, and density estimation are the main tasks of unsupervised methods. The goals of these methods are usually to transform data or being able to reproduce it.

Tasks are executed by so called *models* which represent the specific relationship/representation with respect to the problem. For example, a classification problem has inputs and desires

a certain output. Models return these outputs based on their learned representations. The learned representations can be of various forms, e.g., graphical, mathematical, conditional, or simply a set of coefficients. *Parameters* are an actual instance of a representation thus a concrete model. A *learning algorithm* computes the set of parameters that provide the best results for the current problem, hence it builds the model. Learning algorithms posses their own parameters called *hyper-parameters* which enable the developer to tweak the algorithm to the specific task. The resulting models may differ with respect to their actual parameters even if multiple runs of the learning algorithm have the same hyper-parameters. This is caused by deliberately introducing randomness within the learning process or non-unique optimal solutions for the parameters. The set of all possible models a learning algorithm can generate forms a *model class*. *Model selection/training* is the process of finding the model within a model class that fits the data best.

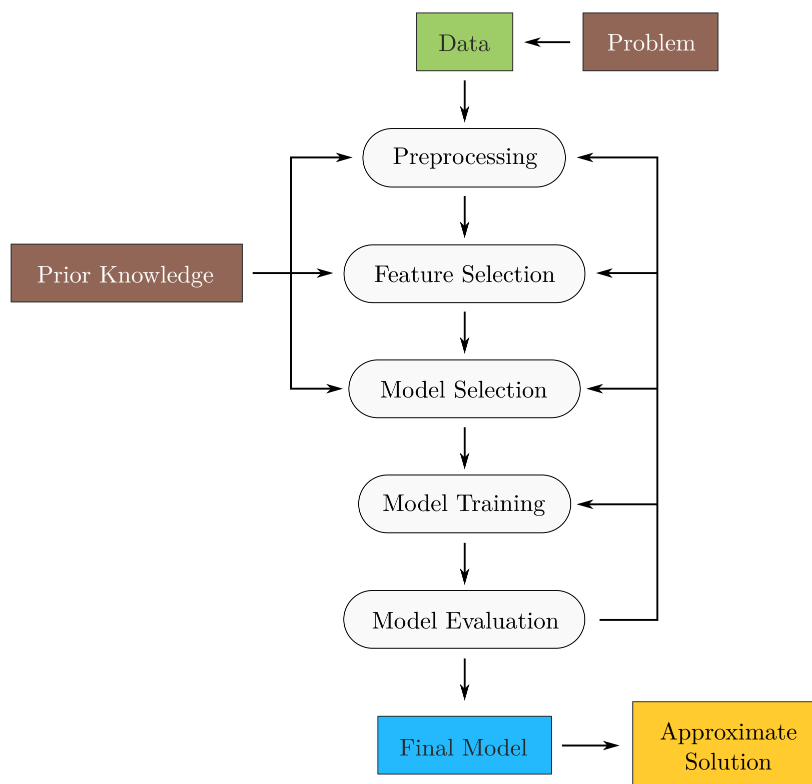


Figure 5.2.: The data analysis workflow is an iterative approach that is executed until a model is accepted (final model). The final model provides an approximate solution to the problem represented by its data. First three steps prepare the data or the model for the actual training and incorporate usually some sort of prior knowledge. Last two steps train and evaluate the models performance with respect to the previously trained models.

Figure 5.2 shows the process of building a model for a specific problem. Given is a classical machine learning task/problem as mentioned above and the data that is related to it. Result is the final model that is capable of providing the approximate solution. Steps in between prepare the data or the model and sometimes have prior (domain) knowledge as prerequisite in order to be executed efficiently. It is very common that there are many iterations of these

steps until an appropriate model is found. The final model rarely concludes the problem and ongoing repetition of these steps help to improve the overall performance of the models.

5.1.1. Data

The most important entity in the entire data analysis workflow is the data itself. It contains the needed ingredients to solve a specific problem thus the quality of it is crucial. This means data should correlate with the task and be free of noise and redundancies. The data approximates a process of the real world hence the amount is essential if the model should reflect it in an appropriate way. It is hard to tell how much data is needed as this depends on the complexity of the real world process that is modeled. A set 4 observations is enough to build a model that mimics the logical XOR gate, but how much data is needed to detect handwritten digits from images? No concrete answer can be given for this question as machine learning is of inductive nature but a rule of thumb may be "more is better". Another important point is that real world data is often imbalanced meaning that the class labels do not describe a uniform distribution. Balancing the data bares the risk of building models that do not learn relationships of the real world but an augmented view of it. This might be ok during the develop phase but becomes a problem as soon as it is used in a real-world scenario. Therefore additional care has to taken with imbalanced datasets.

Table 5.1.: Simple "dog" dataset with three features (Color, Weight, Length) and the targets Breed (classification) and Price (regression) of 5 observations.

Nr.	Color	Weight[kg]	Length[cm]	Breed	Price
0	Black	45	88	Labrador	845.45
1	Brown	40	80	Labrador	933.90
2	Brown	60	100	German Shepherd	999.45
3	Black	20	28	Miniature Poodle	500.75
4	White	15	33	Terrier	600.21

In the simplest case the dataset is a table where each row depicts an observation and each column represents a feature (or variable). Table 5.1 illustrates a sample dataset for dogs in which Color, Weight and Length are the features, and Breed (classification) and Price (regression) are the targets. A row without its labels is called feature vector and is usually the input to a model, e.g., $x_1 = (Black, 45, 88)$. The output is one value representing the target/label of the feature vector, e.g., $y_1 = Labrador$ (classification) or $y_1 = 845.45$ (regression). Labrador, German Shepherd, Miniature Poodle and Terrier are called class labels in the context of classification or simply class. Usually only one target column is given as it is either a classification or regression task. The example in Table 5.1 contains two targets for illustrative purposes. The labels are obtained through human experts or are product of some automated process. This simple example is a two dimensional dataset but for many problems multidimensional data is more appropriate. Images and time series are usually represented in higher dimensional datasets as the "flattened" version may destroy

essential information stored in the local correlations within the dimensions. For example an apple within a image is easy recognizable by humans as the different pixels "look" like an apple in real world from one specific perspective. This definitely gets harder if the image (which is nothing but a matrix) is flattened to a vector. It is most likely that humans would not be possible to identify the object within the image represented as vector because of the missing local correlations between the pixels. This information loss also affects the learning algorithm, thus it is important to keep the data in its "natural" form if possible.

5.1.2. Preprocessing

The first step is usually to preprocess the data which includes reshaping, normalizing, projecting, sampling etc, and often multiple preprocessing steps are applied in conjunction. This is needed because the data might combine multiple different datasets and features with incomparable values ranges, or it is given in a hierarchical form that needs to be flattened and so forth. Reshaping transforms the shape of a dataset into a new one that is more appropriate for a given model class. Normalization adjusts the feature values such that they have a common scale. Most prominent methods are the standard score (Equation 5.1) and feature scaling (Equation 5.2)

$$X^* = \frac{\boldsymbol{X} - \mu}{\sigma}, \qquad \text{z-score} \qquad (5.1)$$

$$X^* = a + \frac{(\boldsymbol{X} - \boldsymbol{X}_{min})(b - a)}{\boldsymbol{X}^{max} - \boldsymbol{X}^{min}}, \qquad \text{feature scaling} \qquad (5.2)$$

nonetheless there are also other methods.

An example application for normalization is given in Table 5.1 in which Weight and Length have different units thus are not comparable. Bringing these onto the same scale enables models to reason about them equally, i.e., models often interpret the magnitude of features thus may get biased towards dominating values. Projections transform the data from one representation into a target representation. These transformations must not be perfect in the sense that all data is retained during the process, thus projections that compress the data are possible and needed. An application for a projection is given in Table 5.1 in which the labels (Black, Brown, White) must be mapped to numbers in order to be processed by most learning algorithms. Another application of projections is dimensionality reduction for visualization or compression purposes. A specific algorithm to down-project high dimensional data into a space of smaller dimension is the Principle Component Analysis (PCA) [43]. The projection uses the variance within the data to build up a smaller dimensional space thus retaining most of the information. Sampling is the processes of retrieving a small subset of the original dataset such that the size becomes more manageable. Additionally sampling is used to balance the distribution of the labels as unbalanced datasets are very common in many real-world applications. For example the distribution of dog breed in the example is not balanced as Labrador is more frequent than order breeds. This of course is part of the problem the model tries to learn, nevertheless extreme imbalance may cause the model to neglect all other classes. Imagine a dataset of 100 dog observations in which exclusively Labradors are present except for the three German Shepherd, Miniature Poodle and Terrier

observations given in Table 5.1. Building a model with this data would cause it probably to neglected the three minority classes therefore always predicting majority class Labrador. The model would reach 97% accuracy by doing so thus decisively has a good performance. Sampling helps to mitigate the problem by modulating the label distribution. This must be done with caution as the model learns a different distribution and not the possible "real world" case.

5.1.3. Feature Selection

Retrieving the data involves probing a certain environment and often this is done in a redundant fashion because of external limitation or other non-controllable reasons. For example the signals from two microphones, used to record a band, will highly correlate as no perfect isolation of the different audio sources (singer, guitarist, drummer, ...) can be guaranteed. Feature selection only retains features within a dataset that contain viable information, i.e., is unique and contributes to the entire information content. Thus the process selects variables with the most information content, removes redundant ones and ranks them according to some importance metric. This can be done manually if domain knowledge is available or via algorithms that use statistical methods or some auxiliary predictor. The main problem with feature selection is that it is seldom known in advance which features are relevant, irrelevant or even misleading. It is a hard problem to pick a handful of essential variables from hundreds or even thousands of features without perfect knowledge about their relationships and interactions. Removing too many features might exclude important information, adding to many increases the overall noise and makes it hard to build a stable model because of the large amount of information. Additionally, linear and non linear dependencies between features need to be considered as they might be the key to solve the initial problem. For example the task is to build a model that predicts the area of a rectangle. Features might be: width, height, rotation, absolute position, etc. Obviously the only features needed for the task are width and height and all others can be considered as noise in the task's context. Both features are meaningless if they are inspected isolated but by considering the multiplicative relationship between them, it becomes clear that they are the only thing needed to solve the task.

There are various types of feature selection methods which can be categorized into: filter methods, wrapper methods, feature selection during learning and feature selection after learning. Filter methods select or rank features according to a statistical criteria without making use of any prediction method. Statistical criteria include the Preason's correlation coefficient, the Fisher criterion or the t-statistics, which reason about the correlation between features and output, and use a threshold to decide whether a feature is important or not [20]. Feature selection on its own can be viewed as machine learning task in which we try to find the best possible combination of features for the given labels. Wrapper methods make use of this by using any auxiliary machine learning algorithm to train models on different subsets of features. The model with the best performance defines the subset of features that is selected. Main problem is the combinatorial explosion of feature subsets that need to be considered in an exhaustive search. Forward selection and backward selection are methods that iteratively

add or remove features form the current used feature set with respect to the performance of the previously trained models. The last two types of feature selection methods use either rule-based predictors that inherently choose the best set of features to make their prediction, or methods that can be augmented with some sort of regularization that gears the predictor to use as few relevant features as possible.

5.1.4. Model Class Selection

There are many different machine learning algorithms each having their own unique properties, advantages and disadvantages. These generic learning algorithms are then used to build models tailored to specific tasks. Hyper-parameters help these generic models to adapt to different situation and thus improve (if correctly used) the resulting model's performance. In general there are two types of models, parametric and ndifferenton-parametric models. Parametric models store the learned information in form of parameters that are used, depending on the model class, during prediction. Non-parametric models solely uses the provided training dataset (thus the observation within the dataset are the parameters) to make their predictions. The advantage of dedicated parameters is that the model size is usually magnitudes smaller than in non-parametric models that always need the full training set in order to operate correctly. Model class selection limits the possible search space of models tremendously as each model class usually only has a small set of hyper-parameter that need to be figured out.

Nearest Neighbor

The most prominent example for a non-parametric classification model is the k-Nearest Neighbor algorithm (kNN) [35]. It uses the simple concept of counting and the spacial locality of the observations to make predictions. The hyper-parameter k defines how many neighbors should be considered to evaluate the prediction label. Consider the dog example with only two features Weight and Length. These can be represented as a point on a plane where observations with the same label would form clusters (because of their similar values), and some clusters may even overlap (e.g., big dog breed). Given now a new observation (new point) for which no label exists, kNN simply places the observation into this plane and uses the most prominent class of the k nearest neighbors as prediction result. A distance function (e.g., euclidean distance) is used to evaluate what the "nearest" neighbors are but despite that no additional information is needed except for the initial training set. This principle can be extended to the multidimensional case, nevertheless the more dimensions are used, the more observations are needed to fill the space appropriately. The obvious disadvantage of non-parametric models is that they need the training set at prediction time which increases space and time effort for this model class.

Decision Trees

A Decision tree [29] is a very intuitive parametric model where the parameters can actually be decoded by humans as opposed to many other models. The model is a tree where each non-leaf node is a "questions" directed to a specific feature of the observation, and each

leaf node corresponds to a final prediction. Thus decision trees partition the training data in a hierarchal fashion cutting the space (in the 2D dog example the plain) into segments that correspond to a certain class. The learning algorithm searches for the feature that provides the maximum gain of information and uses the feature's domain as subtrees. Gain of information is typically measured via Information Gain [35] or Gini Impurity (Gain) [50] and they provide a measure on how much information is encoded into a feature with respect to the labels. Consider now the dog example with only two features Weight and Color. Weight could be the root node with the decision $Weight < 30$ resulting in two subtrees each splitting the remaining feature Color. Color is a categorical feature with four different values thus resulting in four leaf nodes each having a target label associated. The most frequent label is chosen if multiple labels match the the criteria imposed by the trees path. For example the feature vector $(Brown, 35)$ would result into an ambiguous leaf node in which Labrador and German Shepherd would fit. In this case the model would return Labrador since it is the most probable class in this situation (with a probability of .66). Real world applications usually use multiple decision trees at the same time which is called Random Forest.

Support Vector Machines

Support Vector Machines (SVMs) [9] are based on the idea of finding a linear classification border that maximizes the margin between positive and negative samples. Inherently this means the SVMs are basically restricted to binary classification which can be circumvented by training multiple SVMs. Let the labels in Table 5.1 be *big* and *small* depending whether the dog reaches the threshold $Weight < 30kg$ instead of actual breed names. Again let the observations be points on a plane (consider only Weight and Length), SVMs then compute the line between the points (decision boundary), labeled as *big* and *small*, such that the distance between the nearest points (support vectors) of the classes big and small is maximized. Simply speaking, SVMs compute the line located in the middle between the nearest points of the two classes *big* and *small* (they are not allowed to overlap). This is solved by transforming the original margin maximization problem into a convex quadratic optimization problem which then can be solved by using Lagrange multipliers. To avoid overlapping the entire computation is executed in a higher dimensional space in which overlaps are very unlikely. SVMs are based on complex mathematical concepts as in contrast to the previous two models. Nonetheless they offer very good performance on rather small datasets making them an interesting choice for many applications.

Neural Networks

Neural networks are build up by many small units called perceptrons that are closely related to logistic regressors. The idea of a unit is that it gets numeric values as input and returns a specific output value that has some relationship to the inputs. Figure 5.3 illustrates such a unit in which observation #0 from Table 5.1 is the input, and some arbitrary value is the output representing the price estimate. Each input feature from $\boldsymbol{x}$ has a weight $\boldsymbol{w}$ assigned to it which can be seen as an importance measure. Length is the most important feature in this example as it has the highest weight, thus the longer the dog the higher its price estimate. f

is a specific function of the weighted sum of inputs and enables the unit to learn nonlinear relationships. These functions are called activation or transfer functions where for example the Sigmoid function is defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{5.3}$$

The unit would be a simple linear combination without the activation function, thus only able to learn linear relationships which is very limit. The full specified unit is given by

$$f(x) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}), \tag{5.4}$$

where the $(\cdot)$ is the inner product between the input vector and the weight vector.

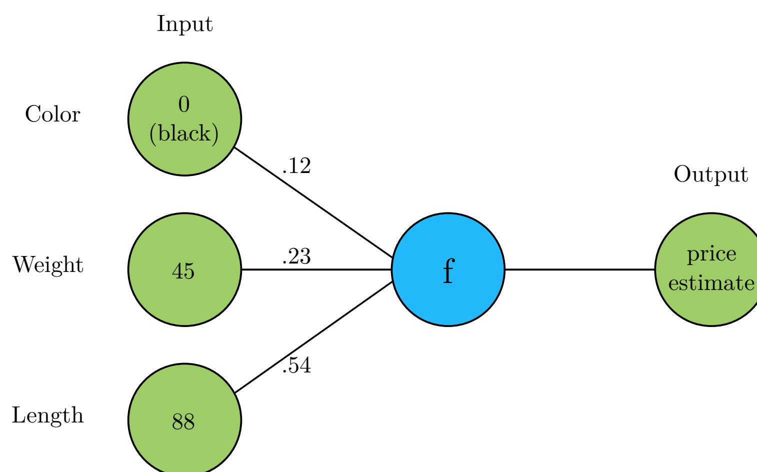


Figure 5.3.: Color, Weight and Length are the input features each weighted by a certain values and combined by f which returns the predicted price estimation.

Having only one model that provides an estimate price is probably not be enough as it is biased towards longer specimens. The next step would be to define multiple models (weights sets) each biased towards a certain feature combination, i.e., different estimation scenarios. Figure 5.4 shows 4 different weight sets each having different preferences of dog features. Model 4 will consider heavy dogs as valuable, where model 2 is rather balanced in its assumption. Off course the models are only valid in their specific scenario and will utterly fail in others.

The solution is to combine all specialized models into one unified model that captures the knowledge of all four. Figure 5.5 illustrates the unified regressors in which the previous models are present in the hidden layer. This is called a dense network as every unit form one layer is connected to the units of the adjacent layers. All of the outputs provided by $f_0(\boldsymbol{x}), f_1(\boldsymbol{x}), \dots, f_3(\boldsymbol{x})$, are combined by the output layer which again applies f to each of the intermediate estimations. Weights for the output layer might reflect how frequent each situation, thus each model was needed and weighting these frequencies with the intermediate estimates of the hidden layers results into the final price estimate.

The current model is capable of predicting the estimate price for a given dog based on its weight, length and color. It does this by repeatedly multiplying the inputs of a unit $\boldsymbol{u}_a$

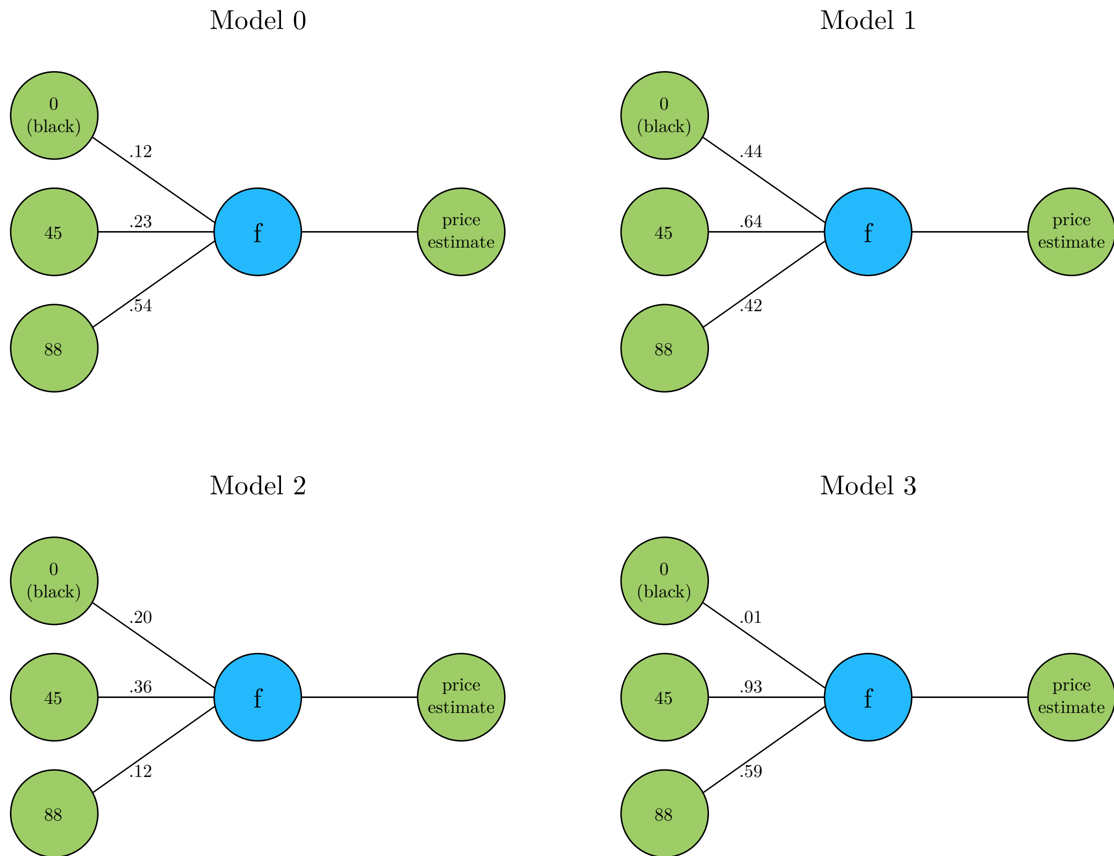


Figure 5.4.: Four linear models each having its own weight set producing different price estimation for the same input.

with its weights $\boldsymbol{w}_a$ and subjects the result to an activation function f. The result of f is again the input of the follow up unit. This structure (Figure 5.5) is already a proper neural network and these multiplication steps can be written in a more compact way. Let $\boldsymbol{W}_k$ be the weights of all units on layer k combined into a matrix and $\boldsymbol{x}_k$ its input vector, then $\boldsymbol{l}_k = f(\boldsymbol{W}_k \cdot \boldsymbol{x}_k)$ is the output of layer k. Now the question remains how these weights can be found automatically as defining them manually is of course not feasible. For now imagine that the weights are handled manually and that the model provides price predictions each time a specimen is given. The client that needs these predictions provides on a regular basis feedback to the values of the model. For example, whether they are too high or too low with respect to the market (Price in Table 5.1) or whether the market follows a certain trend (market needs Terriers). The most natural action would be to adapt the weights based on the feedback in a reverse fashion. First it would be advantageous to change the last layer that reflects how often the different scenarios occur as this reflects the current mood of the market. In the next step all the feature considerations and their price association would be changed. This basic concept is captured in the *back-propagation* algorithm [27] in which a *loss* or *cost*-function provides the feedback which is then applied in a backward fashion to the network. The cost function $L(\boldsymbol{x}^{(i)}, y^{(i)})$ measures the difference between the label and prediction of observation i (e.g., mean squared error, cross-entropy, Kullback-Leibler

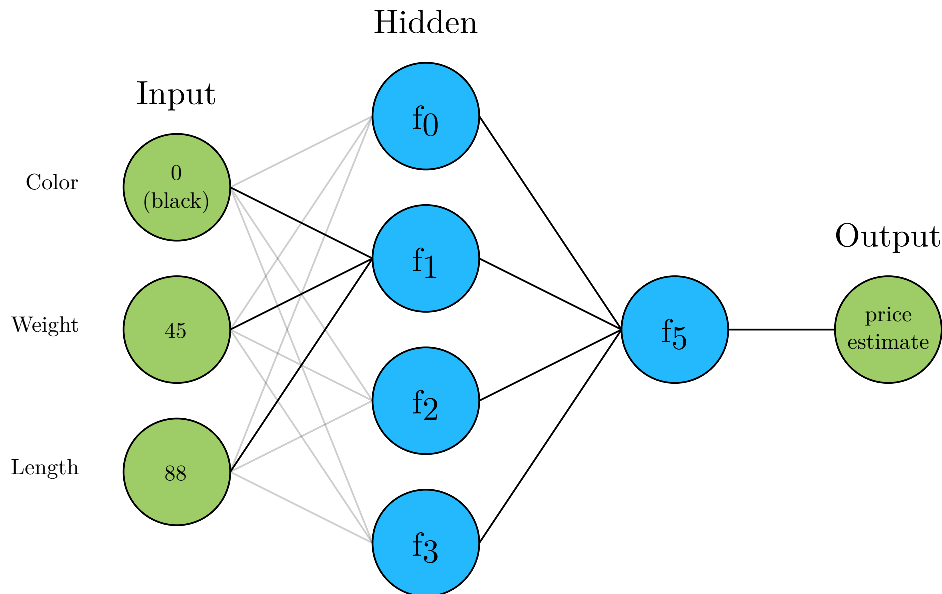


Figure 5.5.: The combination of all four model into a neural network. The hidden layer contains the weight sets of the different four models which are than combined by another weight set aggregated by f_5.

divergence, etc). This is done by using cost functions along with activation functions that are differentiable such that a minimization problem can be applied. Weights are then updated by

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial L}{\partial W}, \tag{5.5}$$

in which the previous weight is modulated by the partial derivative of the loss with respect to the weights multiplied by the learning rate α (hyper-parameter).

This iterative approach is an instance of the gradient descent method in which a local minima provides a sub-optimal solution. The learning algorithm might converge to "good" or "bad" local minima for the model thus weights are usually initialized randomly in conjunction with multiple experiments. The example from above can be generalized to arbitrary problems that have a fixed sized vector as input and return either a label or a value. Neural networks are very versatile as their expressive power can be "engineered" via its architecture which resulted in numerous extensions to the basic concepts.

Two very simple extensions to neural networks that contributed to the advent of deep learning are the Dropout regularization and the Relu activation. The biggest problem with the sigmoid (or tanh) activation is that the more layers a network has, the less learning progress is done in layers closer to the input. This problem is called "vanishing gradients" capturing the issue that successive gradient computation and propagation during the learning phase reduces the actual weight adaption values close to 0. In other words, the value with which the weights are modulate in Equation 5.5 is getting very small (close to 0) for layers far away from the output. Sigmoid (S shape between 0 and 1) functions have at maximum a derivative of $\max_x \sigma'(x) = .25$ thus propagating these values will reduce the feedback at least to a quarter of its initial value. The relu function defined by $relu(x) = max(0, x)$ circumvents this problem as its gradient is either 0 or 1 (for $relu'(0) = 0$).

Dropout [48] on the other hand helps the network to learn general relationships between the input and the output instead of learning the input and its output by "heart". This problem is called overfitting and states that a given model learned the notions of the training set, instead of the relationships between the input and the output. Overfitting models are very good in predicting values on the training set but perform poorly with new data. Neural networks are very prone for overfitting and dropout helps to regularize them. Dropout "removes" randomly nodes during training from the network, i.e., for each sample a different set of nodes are deactivated. This has the effect that the different nodes cannot rely on each other during training thus need to learn something general of the data. After training, all of these nodes work in conjunction together where each node represents something unique.

Convolutional Neural Networks

A Convolutional Neural Network (CNN) [27] extends the basic notions from neural networks and introduces two new layer types. Again CNNs incorporate units with learnable weights (and biases) that are combined via the dot product with the inputs and optionally apply a non-linearity as activation function. Still the entire network is trained by computing the gradients thus represents a single differentiable function that ranges from the input layer to the output layer. And still training is done via gradient descent. The main difference is in their structure as CNNs where developed to handle volumes, e.g., images (with color channels). Image processing is not scalable with normal dense layers as the number of parameters (weights) needed to fully connect even one unit is fairly high. An image with $32 \times 32 \times 3$ $(width \times height \times channels)$ pixels would need $32 \cdot 32 \cdot 3 = 3072$ weights on **one** input unit. CNNs solve the problem by exposing the fact that the data is organized on multiple dimensions thus efficiently make use of the spatial correlations in it. Instead of destroying the volumetric information by feeding the data as one big vector, CNN units extract the information from the volume directly in form of small patches.

Convolution layers consist of learnable filters where every filter captures a small proportion of the width and height of the image and the entire range of its depth (all channels). An example filter is given in Figure 5.6 where the filter has a $3 \times 3 \times 1$ shape covering 3 pixels along the width and height, and 1 pixels in the depth. These filters are slided (convolved) across the width and height of the image during the forward path (prediction) such that they cover any position on the image. Dot products are computed at every position during the convolution between the entries of the filter and the input. The result of the convolution is, instead of a single activation value, a 2-dimensional activation map containing the responses of the filter at any position. Usually a convolutional layer will consist of multiple filters and the resulting activation maps are stacked together to form the new volume for the next layer. Each filter is randomly initialized and it learns to detect one specific object (colors, shapes, faces, etc.) over the course of training. This object can be detected anywhere in the image because of the convolutions. Figure 5.6 shows a 3×3 filter that is applied to a patch of the input. Basically it computes the dot product between filter and patch and divides the result by the amount of pixels that are involved.

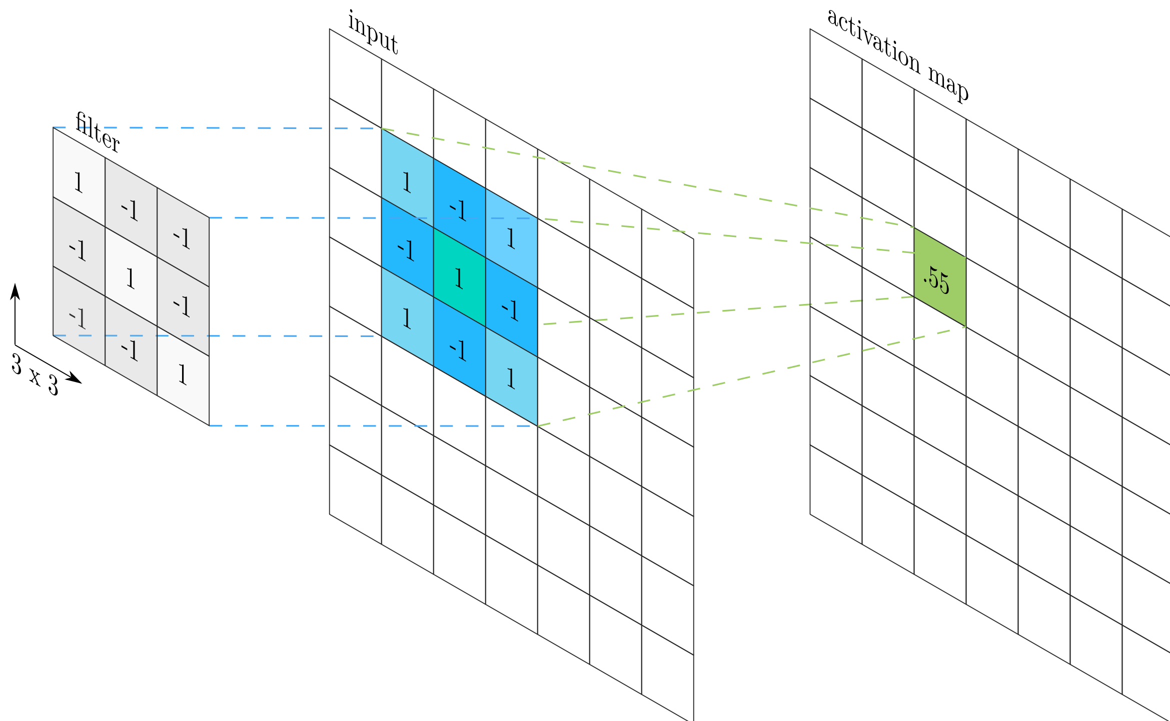


Figure 5.6.: The filter is slided across the input map producing one value on the activation map.

A hyper-parameter of the this layer is the *receptive field* that measures the extend of pixels it "sees" during the convolution. It defines the width and height shape of the filter but not the depth as the depth is always fully exposed. The main reason for this is weight sharing, which states that all neurons of an image slices (e.g., channels of an rgb image) share the same weights. This is a sensible assumption because an object located at position (x, y) is usually also important at position (x1, y1). Not only does weight sharing help to save parameters but also forces the network to learn sensible representations thus mitigates overfitting (discussed in the next section).

Additional to the convolutional layer CNNs also introduce pooling layers that reduce the spatial size of the representation therefore the overall amount of parameters. Most used pooling layer is the MAX layer that applies the maximum function, but other operations are possible (e.g., mean, ...). Pooling layers convolute over the input and execute the pooling operation on the receptive field similar to the normal convolution filters. A max layer therefore retains only the biggest value in its current receptive field, thus a 2×2 max layer will discard 75% of the activates (moving in strides of 2). Figure 5.7 illustrates the max pooling operation where the left side illustrates the input. Each color represents one receptive field of the pooling layer which moves in strides of 2 hence does not overlap. The result is given on the right side in which only one value is retain for each pooling operation.

Despite their wide use of application in CNNs, current trends [47] suggest that pooling layers should be replaced by normal convolution layers using different strides to reduce the amount of parameters. As for now, pooling layers are a useful ingredient in many CNNs and help to keep the amount of parameters under control.

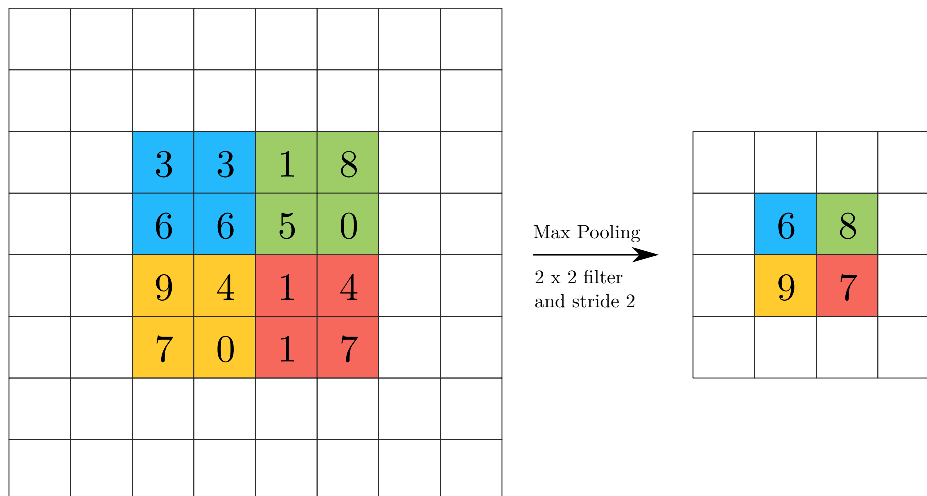


Figure 5.7.: Max pooling subsamples the input by retaining only the maximum activated value.

Note that this was only a very brief overview of CNNs and the fast paced research in this field adds frequently notable details to it. CNNs are the main reason for the new machine learning and Artificial Intelligence (AI) trends as they are applicable in many situation yielding outperforming results in nearly every category. They are the core component in many computer vision, natural language processing, video analysis, and medical tasks.

5.1.5. Model Training

Model training is the process of executing the learning algorithm on the data. This phase is usually tightly interleaved with the model evaluation, as finding the model that performs the task best involves generally hundreds of models. This is due to the hyper-parameters a learning algorithm has, the different preprocess operations that can be applied and model classes that can be selected. Its seldom clear which combination of these yield the best performance thus training and evaluation is usually automated. This is called hyper-parameter optimization in which potential combinations of settings are executed and ranked based on some objective function.

There are different ways to train models and a naive approach would be to use all the available data for training, and also use it to evaluate it. Obviously this approach lacks one important ingredient - generalization performance. The motivation for training a predictive model is that it can be used in situations where we have data that has no label assign to it. For the dog example this would mean that we want to use the model in future to automatically evaluate the dog breed and price based on the features and without human intervention. Off course using the data to train and evaluate the model does provide any information on how well it will perform in situations with new unseen data. There are several different training schemes that help assessing the generalization performance, each having their own advantages and disadvantages.

Re-substitution Validation

The re-substitution validation uses the training set as test set and provides too optimistic results. Despite that it is not evaluating the generalization performance, re-substitution can still carry some value. Computing the re-substitution performance is useful for selecting model classes, features and data preprocessing steps. The idea is that the model should perform extreme good on the test set which is the same as the training set. This provides information about the complexity of the data relative to the expressiveness (complexity) of the model. If the model does not reach an almost perfect score during re-substitution validation then it is unlikely that it can generalize the provided data good. This means that either features, preprocessing step or model class must be changed.

Holdout Validation

A more robust way to evaluate the performance is given via the holdout validation or sample estimation [25]. Figure 5.8 illustrates the process of the holdout validation in which the data is partitioned into two mutually exclusive subsets called training set and test set. The test set is usually comprised of 1/3 of the data and the training set with the remaining 2/3. The learning algorithm uses the training data along with its hyper-parameters to build a model. This model is then evaluated by using the test data and comparing the prediction labels with the test labels.

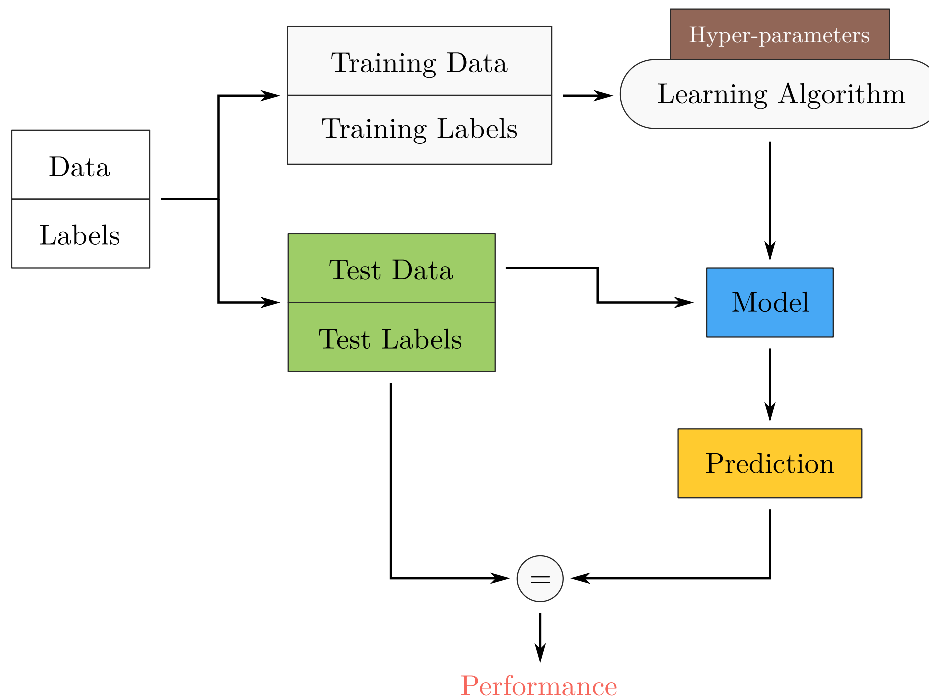


Figure 5.8.: The entire data is split into training and test data with their respective labels. Training data is used to derive the parameters of the model via a learning algorithms that has its own (hyper) parameters. After training the test data is used to produce predictions that are compared with the true labels resulting the model's performance.

The approach uses unseen data to evaluate the performance of the model thus reconstructs the situation in which the model will be later used. Nonetheless the approach leads to models that are biased towards the test data set. This is because the model developer will adapt the hyper-parameters according to the test set performance thus tailors the model exclusively to it. This off course leads to a bad generalization performance if the test data is not a good approximate for the real data distribution.

Cross Validation

Cross-validation takes the idea of the holdout validation one step further by partitioning the data randomly into k mutually exclusive subsets (folds). It then applies k holdout validations in which the test set is one fold, and the remaining folds comprise the training data. This results into k models each validated via a different proportion of the entire data. The performance is then the average of all models giving an estimate on how well the model would generalize on the data. The estimate is better than the simple holdout-validation as it mitigates the bias towards a small fraction of the data, nonetheless it still inherits some issues. One issue is that the folds are randomly partitioned which results into a different label distribution than in the original dataset. Stratified cross-validation mitigates this problem by arranging the folds such that the original distribution is preserved. Another issues is the pessimistic bias for small k's which is caused by the big test-set sizes. Models do not reach their full potential if data is withheld for test purposes. The pessimistic bias captures this concept by stating that the model performance is estimate too low because of the withheld data. This causes a dilemma in which the entire dataset is used to reduce the pessimistic bias but in hindsight does not represent the generalization performance. The dilemma cannot be avoided in real-world applications but increased awareness of it helps making better model selection decisions.

5.1.6. Model Evaluation

The only thing left after shaping the data, selecting the features and the model class, training concrete models and measuring their performance, is evaluating their performance. This process looks deceivingly easy at the beginning but model evaluation inherits many pitfalls invalidating the assessed results. Model evaluation compares and provides means to decide whether given models perform good or bad in future scenarios. These scenarios will most likely include observations that the models have not seen during training thus brings the learned relationships to test. Simply speaking, model evaluation assess the predictive performance of models on unseen data.

As already mention evaluation and training goes hand in hand and some of the issues with the performance assessment can be avoided by the training procedure (e.g., cross-validation). Nonetheless there are some issues related with the data itself and the performance measures used during training. One assumption to the data that need to hold is: data is independent and identically distributed (i.i.d) [22]. This assumption states that the samples are drawn from the same probability distribution and are statistically independent from each other. Coming back to the dog example, **identically distributed** means that the procedure on

how the length of dogs is measured must be the same for all specimens, otherwise the samples could represent a different distribution. Mixing up samples that measure the dogs length by either including its tail or not obviously breaks this assumption. **Independent** implies that changing one sample does not effect any other sample. This assumption does not hold for time series as objects over time are depended. In the case of design pattern detection this means that all unique mappings are independent but all mappings within the same equivalence class are not. Changing some properties on the abstraction definitely influences other samples in the same unique mapping. This is an important point to consider during the evaluation as it alters the predictive performance of models in a way that they are either too optimistic or pessimistic (usually too optimistic). Imaging a design pattern detector that is trained via cross-validation on one unique mapping that contains 1000 role mappings. The results will probably be extreme good but way to optimistic as each of the samples in the training or validation set share the same abstraction. It is easy to overlook such dependencies without any prior knowledge of the domain.

Measures of Performance

A performance measure is a metric capturing how different the predictions are from the true labels. There are measures for regression and measures for classification where this work will focus on the last. Additionally only binary classification measures are considered as multiclass scenarios are not relevant in the context of this work.

Every measure is build on the concept of counting the amount of labels the classifier was able to predict or not. Therefore we define L as positive labels and $\tilde{L}$ as negative labels, and additionally

True Positive (TP) as all positives that were predicted as positive,

False Negative (FN) as all positives that were predicted as negative,

False Positive (FP) as all negatives that were predicted as positive and

True Negative (TN) as all negatives that were predicted as negative.

The confusion matrix captures these notions and help to evaluate the models performance on a class level. Values within the matrix count how many samples fall into the different categories, thus might not be as intuitive as a single performance measure. Nonetheless confusion matrices are robust against metric issues i.e., too optimistic performance evaluation for imbalanced datasets as the prediction behavior of the model can be directly evaluated.

Table 5.2.: confusion matrix

		Predictions	
		L	$\tilde{L}$
Labels	L	TP	FN
	$\tilde{L}$	FP	TN

Accuracy is the number of correctly classified items, i.e.

$$ACC = \frac{TP + TN}{TP + FN + FP + TN}. \tag{5.6}$$

Precision is the proportion of predicted positive example that were correct, i.e.,

$$PREC = \frac{TP}{TP + FP}. \tag{5.7}$$

True Positive Rate (Recall) is the proportion of correctly identified positives, i.e.,

$$TPR = \frac{TP}{TP + FN}. \tag{5.8}$$

True Negative Rate is the proportion of correctly identified negatives, i.e.,

$$TNR = \frac{TN}{FP + TN}. \tag{5.9}$$

False Positive Rate is the proportion of negative examples that were incorrectly classified as positives, i.e.,

$$FPR = \frac{FP}{FP + TN}. \tag{5.10}$$

False Negative Rate is the proportion of positive examples that were incorrectly classified as negatives, i.e.,

$$FNR = \frac{FN}{TP + FN}. \tag{5.11}$$

Matthews Correlation Coefficient is the measure of non-randomness of classification defined as normalized determinant of the confusion matrix, i.e.,

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}. \tag{5.12}$$

F-score is the harmonic mean of precision and recall, i.e.,

$$F_1 = \frac{PREC \cdot TPR}{PREC + TPR}. \tag{5.13}$$

The accuracy is the most common measure of performance and reflects the number of correctly classified items. It is a percentage value where 0 states that the predictions were not accurate and 1 states that the predictions were perfectly accurate. The biggest problem with accuracy is that it returns a wrong performance value for classifiers trained on imbalanced datasets. If the dataset has 99% negative examples then the classifier will probably also have .99 accuracy as it will always predict the majority class (most frequent class label). The main problem is that the model will not have any understanding of the positive class as it is best to simply always predict the negative class. This can become a serious problem, for example if a self driving car needs to perform an emergency stop. Of course this situation is not as frequent than all other driving situations nonetheless because of its importance it should

be definitely be considered by the model. DPD also work on highly skewed datasets as the amount of classes matched by the search space reduction algorithm is usually much larger than the amount of true pattern instances. A more informative measure for these situations is the precision and recall.

Precision measures the proportion of correctly predicted positive examples and again reflects a percentage value. The precision is low if either no positive samples were correctly classified, or if many negative samples were classified as positive samples. Recall on the other hand measures how many of the positive samples were predicted. Both are somehow inverse related i.e., high precision often comes with low recall and vice verse. The measures are centered around the positive labels and thus do only provide indirectly information about the negative samples. This transitively applies also to the f-score as it is the harmonic mean of both.

Matthews Correlation Coefficient (MCC) ranges from -1 to 1 where 0 states that the classifier has the same performance as random guesses, -1 that it is a completely wrong classifier and 1 that it is a complete correct binary classifier. It uses all four frequencies (TP, TN, FP, FN) and is often a more balanced evaluation of the performance than the others. In contrast to the previous measures, MCC does not provide deceivingly good estimates for classifiers if the dataset is imbalanced. For example a majority classifier that always predicts the negative class because the class represents 99% of the class distribution would still result to a coefficient of 0. The majority classifier would force the numerator to 0 which is similar in the inverse case. MCC is the Pearson Correlation Coefficient computed out of the confusion matrix thus the interpretations are interchangeable. This includes the guidelines with respect to coefficient interpretation given in Table 5.3 proposed by Evans [13]. Evans splits the absolute range between $0-1$ into 5 levels of strength associations ranging from very weak to very strong. Although these interpretations are related to the linear correlation between two variables they may still prove valuable in terms of intuitive judgment. That is, a classifier with a coefficient $> .6$ can been seen as a strong classifier giving "good" results. Off course the total appropriateness of a classifier needs to be evaluated in the context of its application but the strength association provides a way to quickly and intuitively estimate the performance. This is important because in contrast to percentages like accuracy, precision, recall, etc., MCC is not as straight forward interpretable as it represents a correlation.

Table 5.3.: Pearson Correlation Coefficient interpretation guide by Evans [13].

	Coefficient	
Strength of Association	Positive	Negative
very weak	0.00 to 0.19	-0.00 to -0.19
weak	0.20 to 0.39	-0.20 to -0.39
moderate	0.40 to 0.59	-0.40 to -0.59
strong	0.60 to 0.79	-0.60 to -0.79
very strong	0.80 to 1.00	-0.80 to -1.00

A total framework for evaluation of classifiers is given by the Receiver Operator Characteristics (ROC) [14]. It uses visualization techniques to assess the performance of classifiers called ROC graphs. ROC graphs are two-dimensional graphs in which the TPR is plotted over FPR and every point in the graph corresponds to the performance of a single classifier. The graph visualizes the trade-off of classifiers with respect to these measures such that classifiers can be selected depending on the preferred misclassification type. Precision/Recall graphs (PR) on the other hand are more appropriate for the case of imbalanced data. Again imaging a imbalanced dataset with $\tilde{L} >> L$, than it can be easily seen that changes within the FPs are not captured well as the FPR will have a large denominator because of the negative examples. PR graphs on the other hand are sensitive to the FPs within a imbalanced set because of the precision metric and its ratio given in Equation 5.7.

This was just a brief introduction into measures for binary prediction performance and in depth discussions are given by Haibo and Garcia [21], Baldi et al.[7], and Powers and Ward [40]. Not every measure is appropriate in every situation and choosing the right one is important to not over or under estimate the models performance.

5.2. Detecting Design Pattern via Convolution Neural Networks

Detecting design patterns is a non trivial task as languages allow the developers to implement them in various different ways. Additionally the rather loose class arrangements given by the pattern descriptions provide developers with some space of interpretation that contributes to the divergence between the different implementations. This creates the perfect context for machine learning and its algorithms to do its work as many publications within the DPD community show.

Uchiyama et al. [52] uses neural networks to detect different roles that were provided by humans specialists. Input of these networks are the features evaluated via the GQM procedure as mentioned above. These metrics are also hand selected via a human specialist which are than used to train a neural network. After training developers input a sample system plus the semi-automatic selected candidate mappings that should be considered during the detection process. The output of the system are confidence values for each role (globally for all pattern that have been predefined) and based on these values it computes the most likely pattern. For example inputing Template Method and Adapter would result into a list of their roles along with their confidence values which add up to 1. The confidence value for a pattern is then computed by aggregating the role values and comparing the total to a predefined threshold. Sadly the work did not provide any details on the network structure or how the hyper-parameters for the different networks are selected. Simiar Alhusain et al [1] used neural networks on class relationships to evaluate candidate mappings. As with the work of Uchiyama no details of the network topology or the used hyper-parameters is given.

Zanoni et al. [57] evaluated 7 different learning algorithm along with different hyper-parameters to find the most appropriate model class for DPD. Model classes range form Naive Bayes, Decision Trees to Random Forest and Support Vector Machines where each was trained with different regularization methods or different kernels (SVMs). The report provides accuracy and f1-score along with the AUC thus only a skew view on the true performance

of the classifiers. This is not a real problem for the results of Factory Method and Adapter as the class distribution inferred from the ZeroR classifier is nearly balanced. For the other patterns ZeroR is not returning 0.5 accuracy implying imbalanced datasets thus hard to evaluate results. Furthermore the results of the detectors were manually evaluated therefore the dataset size was limited to 1000 candidates. It is unclear whether positive mappings were added to a higher priority than negative samples to avoid datasets that contain only a handful of positives. Nonetheless this needs to be factored in into all results as the learning algorithms were probably exposed to a different class distribution than in real world examples. Performances (best results) for the balanced cases are .86 via C-SVM with radial kernel for the Adapter, and .82 via Random Forest for the Factory Method indicating a relative good fit by the classifiers. All in all no preferred learning algorithm can be concluded. This is also because the evaluation of DPD tools is currently a weak spot in the entire DPD community despite the attempts with the benchmark platform.

Tsantalis et al. used a method beyond typical machine learning algorithms called similarity scoring. The method captures the pattern class diagram in form of adjacency matrices where each OO aspect is reflected in its own matrix (adjacency matrix for: generalization, method invocation etc.). These matrices are then compared with the matrices that the candidate mappings produce. The comparison is done via an iterative approach given by Blondel et al. [8] that calculates the similarity between vertices of two graphs. The method was applied on three open source projects (included in this work) and results in 100% recall and precision for nearly every pattern. One reason for these good results is that only the defining roles of a pattern have been retained in the patterns to avoid false positives.

The approach present by this work uses convolutional neural networks on the feature maps described in Chapter 4. It is similar to the work of Tsantalis in which sub-trees in form of adjacency matrices are compared. It also shares commonalities with the work of Zanoni in which micro-structures are used to detect design patterns, but diverges strongly in the actual detection process and in the data representation. This work uses feature maps to represent the same sub-tree from multiple perspectives (features) and CNNs to convolve over these. This enables the classifier to reason about different aspects of a sub-graph at the same time and to find correlations between them. Humans analyze systems in a similar way by finding a class as entry point, and inspecting the different edges between the classes. The difference is that humans usually work sequence and "store" the previously found relationship by remembering it. The stored information is then compared on the fly with the new observations which is off course error prone and tedious, and the main reason why there exists no big dataset.

The following sections captures the experimental setup used to derive the provided models. Each subsection provides detailed information about one step within the data analysis workflow such that experiments may be repeated.

5.2.1. Data

Supervised machine learning techniques inspect labeled data (input and output is know) and build models from it. These models are then able to predict the output from newly given

inputs for which no output is know beforehand. So in order to train a design pattern detector a big set of pre-classified role mapping instances is needed from which the learning algorithm can build the model. Labeled data is in some situations hard to come by because it is either expensive or difficult to collect. In the case of design patterns both problems occur, not only is it time consuming to inspect systems for patterns but also the agreement among the human experts that classify the examples might diverge because of the different interpretation a pattern implementation may provide. This work uses the Pattern-like Micro-Architecture Repository (P-MARt 04/10/19) [56] which is a peer-reviewed repository that contains 9 open source projects along with their design patterns. The repository was used in many research projects and contains various instances of patterns that can be used for detection purposes. Applications within the repository range from modeling and drawing tools to static analysis and refactory frameworks. The diversity of the projects is good representation of the real world but their implementation is a little dated. For instance QuickUml is already 15 years old by the time this work was written thus lacks state of the art advancements in language and software system design.

Table 5.4 contains the basic statistics of the projects within P-MARt. The biggest project is Netbeans, which could not be used because of multiple error during the analysis with the Spoon framework. Uncommenting of the problematic code sections turned out to be futile as there were to many of them. Furthermore, removing too many code fragments could have compromised and biased the detector because of incomplete data. At the end the project was excluded from the set to avoid any bias. This is problematic because Netbeans contains more than 50% of all Adapter instances within P-MARt (8 unique instances). The second biggest contributor of pattern instances is the JHotDraw project (155 types) that implements 22 unique instances over 12 different design patterns. This is quite impressive as Netbeans with 2558 types only implements 4 different DPs (mainly Adapter). Table 5.5 provides a pattern centric perspective of P-MARt where the singleton pattern is the most popular out of these. It occurs 13 times in 7 different projects and indicates that many projects need the concept of a single instance within one runtime. Least often implemented is the decorator pattern that is present in 2 projects (JHotDraw and jUnit). Most instances are given by the Adapter pattern which indicates its usefulness in the context of frameworks as Netbeans contributes a total of 2612 instances. No additional source of pattern instances was used within this work leading to a rather small sample size that ranges from 2 to 13 unique instances per pattern. Truly, the amount of independent instances is not high enough for a classical machine learning problem and this gets even worse if noisy samples are removed.

Two different datasets were used to train and evaluate the hyper-parameters and models. The first dataset are the observations provided directly by the sampler without any modifications. This dataset represents the real world situation the inference method is operating in, as the data follows the exact same pipeline (source to AST to MS to FRN maps) as in production. Table 5.6 contains the class distribution for each pattern without the Netbeans project. Observations that map classes from third party libraries were removed from the dataset. This includes samples that map to classes within the packages `java.awt`, `javax.swing`, `com.borland.primetime`, `org.w3c` and `org.apache`. Some

Table 5.4.: Pattern-like Micro-Architecture Repository (P-MARt) [56] includes 9 open-source projects. Pattern instances where peer reviewed nevertheless the authors do not claim that all possible patterns are declared.

Project	Version	Number of Types	Unique Mappings	Mappings
QuickUml	2001	155	7	103
Lexi	0.1.1	24	5	13
JRefactory	2.6.24	569	11	2147
Netbeans	1.0.x	2558	26	4146
JUnit	3.7	79	8	218
JHotDraw	5.1	155	22	3201
MapperXML	1.9.7	217	13	208
Nutch	0.4	172	12	104
PMD	1.8	447	10	43

Table 5.5.: Distribution of the detected pattern within P-MARt. Project Distribution describes in how many projects the given pattern was found.

Pattern	Project Distribution	Unique Mappings	Mappings
Adapter	4	13	2952
Composite	4	5	1149
Decorator	2	2	176
Factory Method	3	7	522
Singleton	7	13	13
Template Method	3	8	85

samples within the project's namespace did not exist, e.g., `com.taursys.xml.CheckBox` or `net.nutch.db.LinkMD5Extractor`, thus were also removed. It is not possible to detect patterns with these observations as the source is not given hence no features can be extracted. Sadly this removes quite a lot of Adapter instances within the jRefactory project (~ 24 instances). The class distribution for each pattern is extremely skewed towards negative samples with a mean positive instance proportion of $1.175 \cdot 10^{-2}$. This off course influences the training of the model therefore extra precautions need to be taken. The Adapter pattern with its extreme high amount of negative instances has the smallest proportion of positives samples. This immediately indicates that the adapter sampler needs to be revised in order to avoid the huge amount of negative instances. On the other side of the scale lies the Decorator with only 688 negative samples resulting in a acceptable proportion of .2325. It should be noted that P-MARt does not claim to be complete, i.e., all possible instances within the projects are given. Thus it may be that the detector finds observations among the negative instances that are actually a true instance.

Table 5.6.: Negative instances describe all FPs and positive instances all TPs that are returned by the samplers. Positive Instance Proportion describes the amount of positive instances within all instances returned by the respective sampler.

Pattern	Negative Instances	Positive Instances	Unique Instances	Positive Instance Proportion
Adapter	106,937	164	13	$1.533 \cdot 10^{-3}$
Composite	32,126	210	5	$6.536 \cdot 10^{-3}$
Decorator	688	160	2	$2.325 \cdot 10^{-1}$
Factory Method	21,857	91	7	$4.163 \cdot 10^{-3}$
Singleton	1,858	13	13	$6.996 \cdot 10^{-3}$
Template Method	2,099	82	8	$3.906 \cdot 10^{-2}$

The second dataset is a synthetically enriched set of positive observations. Positive samples are taken without any further modification. Negative samples though, are synthetically created such that the class distribution can be freely set. This is useful for initial model and parameter evaluations as the direct dataset often causes sub-optimal models that always predict the majority class (negative samples). The synthetic dataset circumvents this problem by providing a balanced label distribution such that the model converges and generalizes properly. Obviously the distributions between direct and synthetic dataset is different, not only with respect to their class distribution, but also with respect to the features within observations. Models trained with this dataset cannot make proper prediction on the real data created by the sampler, nevertheless they can function as initial model from which the true training may start (pre-training). Negative samples are created by mutating the role mappings of positives samples. The mutation uses n of the k mappings of an observation and replaces the true classes, with classes that do not map to the same unique mapping (the equivalence class) thus are unrelated. This happens on a project basis and the mutations are random among the roles. Furthermore the amount of roles that are modified within an observation is evenly distributed over the total amount of DP roles and observations. For example a dataset with 100 positive examples for a DP with 4 roles will contain 25 negative observations in which only 1 role is modified, 25 negative observations in which 2 roles were modified and so forth. Again, the rational of the dataset is not to represent the true distribution but to create means by which appropriate model architectures can be found.

5.2.2. Preprocessing

The first preprocessing step is the feature normalization mentioned in Chapter 4. This leads to feature maps that contain positive integers ranging from 0 to 161. The second step normalizes these integer such that they are contained within a tighter range. Many machine learning algorithms including neural networks work best if the data has a mean of 0 and a variance of 1. The intuition is the algorithms interpret the magnitude of the values thus get biased towards certain features. A very prominent scaling method is the z-score in Equation 5.1 which subtracts the mean from the data and normalizes it by its standard deviation. First operation centers the data around 0 and the second normalizes dimensions such that they are of approximately the same scale. This is not useful in the case of feature maps as it

destroys the identification properties of the values. Experiments with the z-score confirmed this behavior resulting in models that either converged extremely slow or not at all. Feature scaling given in Equation 5.2 is more appropriate in this situation as it preserves the identity property of the coefficients. Best convergence was reached by either scaling the values around 0, e.g., between -50 and 50, or by applying no scaling at all.

The third preprocessing step uses a slightly modified version of the Easy ensemble method [28] that helps balancing the class distribution. Easy ensemble is an undersampling method used in imbalanced settings, in which k randomly selected observations from the majority class are used to level with all observation from the minority class. Usually k is set to $|\mathcal{C}_{minorty}|$ thus perfectly balances the data set but other ratios are sensible in order to preserve the true distribution. One big drawback of plain undersampling is that it throws away a lot of valuable information of the majority class. Easy ensemble circumvents this issue by subsampling T independent datasets and uses them to train T classifiers. Each classifier is exposed to a different subset of the majority class but will always see the entirety of the minority class. The final model is constructed by combining the different predictors into one strong model. First advantage is that it simplifies the training as each weak model converges faster because of the balance between the classes. The second advantage is that the ensemble classifier is still able to distinguish the majority class and the minority class given their real (skewed) distribution. The obvious drawback is that training T classifiers can be time consuming and technically demanding as the space demand for T models might exceed the resource limit of one machine.

5.2.3. Feature Selection

The project extracts micro-structures that represent sub-graphs within the ASG and uses them as features in the inference step (Chapter 2). In total 67 different micro-structures are extracted from the source code, which combines the majority of the Elemental Design Patterns, Micro-Patterns and Design Pattern Clues, as basic characteristics of OO programming. Many features are only relevant in the context of a specific pattern, especially the DPCs. As mentioned in Section 5.1.3, selecting the most relevant features is a hard task as it is not always clear which features are truly useful or not. The approach followed in this work uses wrapper methods to select appropriate features, more specifically it used boosted trees from the gbm package [41] and random forest from the randomForest package [3] as the underlying models. Both classifiers are used to compute the feature importance by repeatedly computing the fit of the models with a subset of features. This is done automatically via the Classification And Regression Training (CARET) package implemented by Kuhn [26]. Caret offers a wide variety of training algorithms, parameter tuning, model and feature selection procedures and acts as facade for many prominent R-packages.

Given a dataset for a specific pattern, the feature selection method computes for each role the feature importance resulting into k feature rankings where k is the amount of roles a DP has. The final feature selection is then determined by the union of all features that have non-zero importance. Figure 5.9, 5.10 and 5.11 visualizes the results of the importance computation for each pattern. The color intensity of each tile reflects its importance for

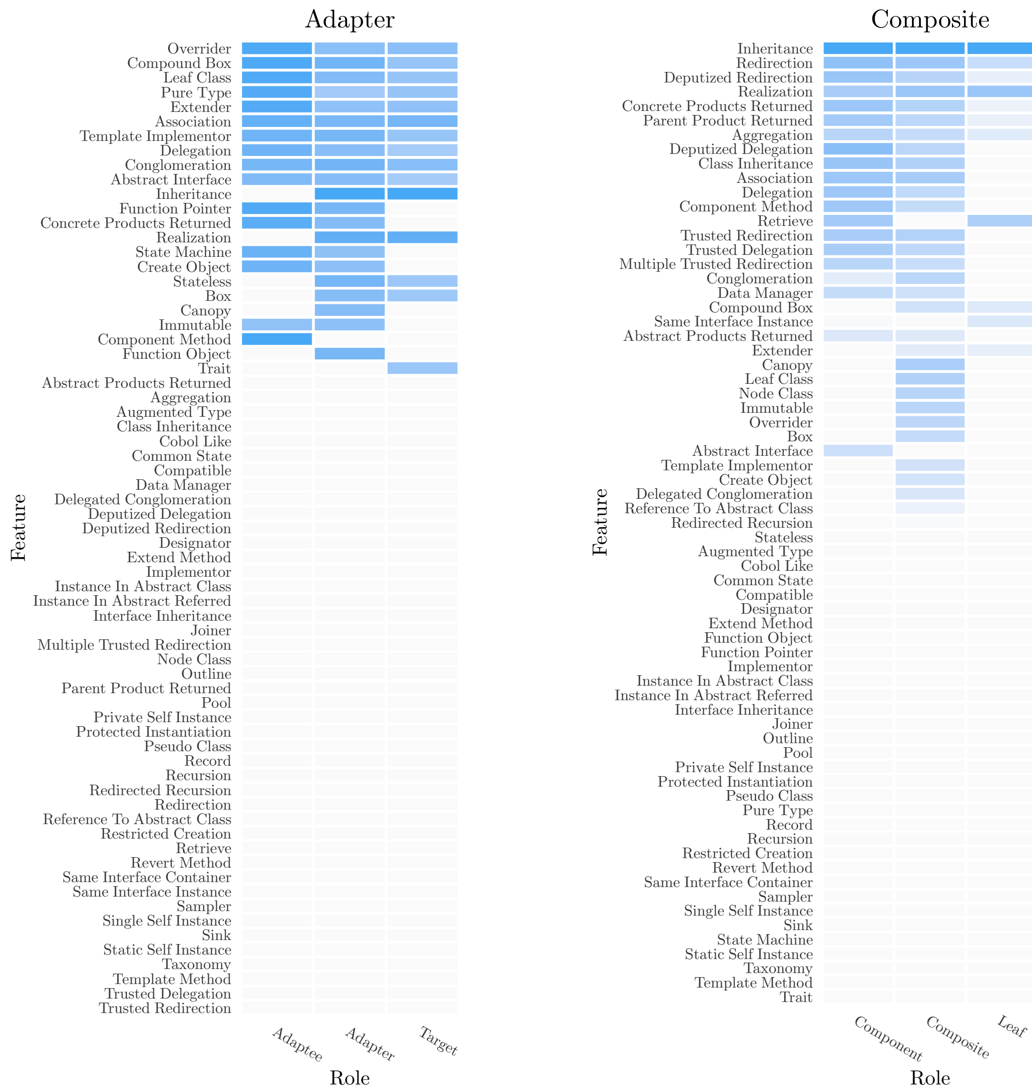
the specific DP role. The features are sorted by their coverage, i.e., for how many roles it is important, and by its overall importance, i.e., the mean importance along all DP roles. Noteworthy features that are contained in almost all multivariate patterns are off course the basic OO concepts like Inheritance, Association, Realization, Aggregation etc. Also the EDPs (Deputized) Redirection, (Deputized) Delegation, as the Concrete/Parent Product Returned occur quiet often within the feature sets as they also capture basic notions of programming. The feature set for the Singleton pattern contains exactly the feature someone would expect with the most important feature being the Protected Instantiation. Additional all features related to self instances within a class are also part of the set reflecting the gist of the Singleton pattern. The visualization illustrate the features that are important for certain communication schemes or structural configuration within the patterns. For instances the Inheritance MS is important for the Adapter and the Target of the Adapter pattern and reflects its structural requirements. Another example would be the Aggregation within the Decorator pattern. The Concrete Component's aggregation importance is quite low as in contrast to all other roles. This captures the structure of the Decorator pattern as Concrete Component is usually the aggregatee, not the aggregator.

5.2.4. Model Selection

There exists a wide variety of model classes as mention in Section 5.1.4 each having specific strengths and weaknesses. The model class used within this work are Neural Networks, more specifically Convolutional Neural Networks. Their advantage is that they are specialized in detecting local correlation within a matrix, i.e., they are capable of learning how several small patterns within a matrix interact. It does this by capturing important objects/patterns as it convolves over the matrix with a small window (receptive field). Objects in the case of feature maps are OO properties and interactions (height) between a limited set of classes (width). Thus CNNs on feature maps inspect via their receptive field different combination of these properties and interactions on multiple levels of abstraction. Each convolutional layer sees a certain level of abstraction where the first layer inspects the plain micro-structures given in their feature maps. As always with neural networks the specific network topology needs to be developed by a series of experiments each shedding light on how many layers, units, which activation, etc are needed to handle the complexity of the data. Deriving the network topology can be a delicate endeavor as CNNs are black box methods, i.e., the exact way the model computes its results can not be decoded by normal means. Furthermore CNNs are trained via gradient descent that may run into sub-optimal local minima preventing the models to learn anything. That said, models presented within this work were derived by using the synthetic dataset such that a coarse network topology could be derived. Directly using the imbalanced datasets provided by the samplers was not possible because it resulted mainly in majority class predictors (except for the Singleton classifier).

Network Topology

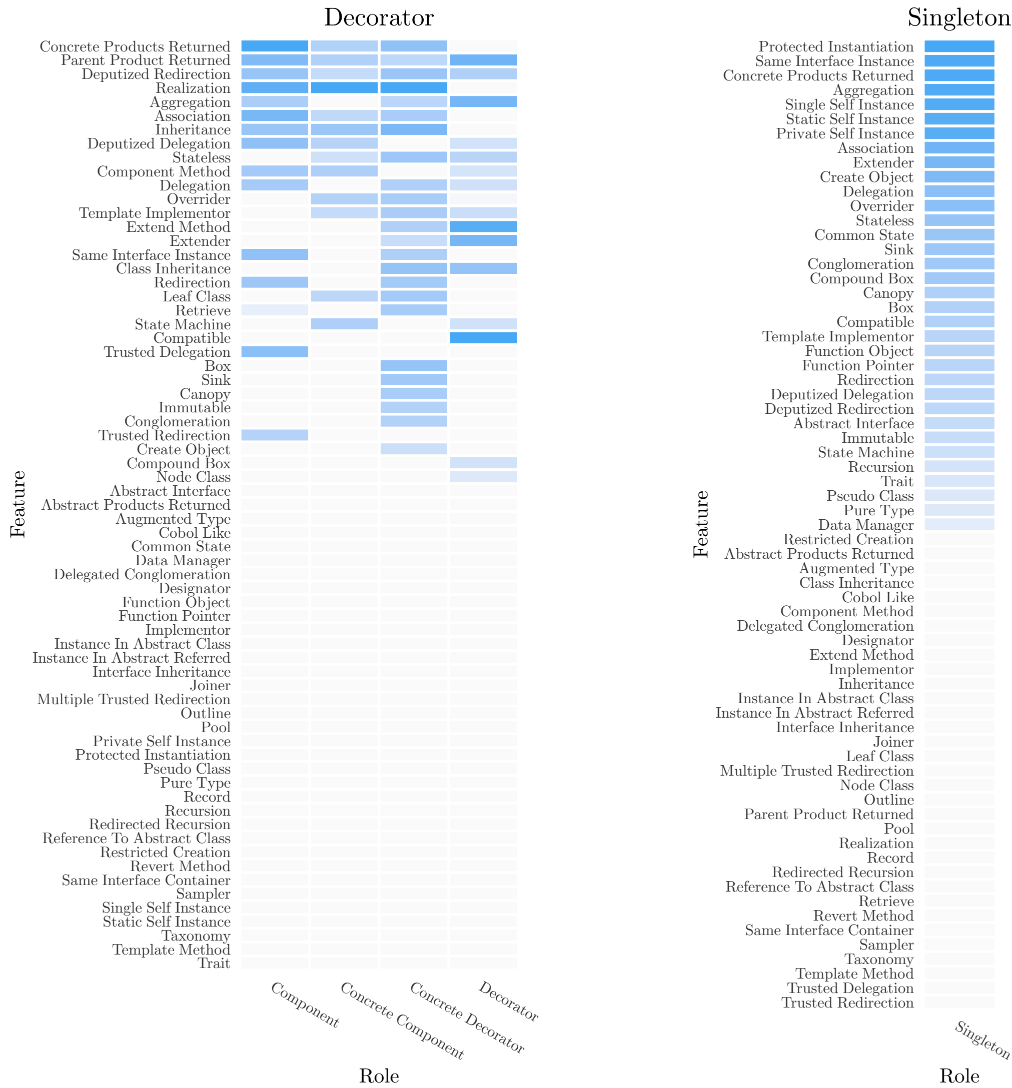
Table 5.6 contains the network topology derived by the initial experiments on the synthetic dataset. The network topology is similar to the well known CNN LeNet [55] and contains in



(a) Importance measures for all micro-structures with respect to the Adapter pattern (darker is better).

(b) Importance measures for all micro-structures with respect to the Composite pattern (darker is better).

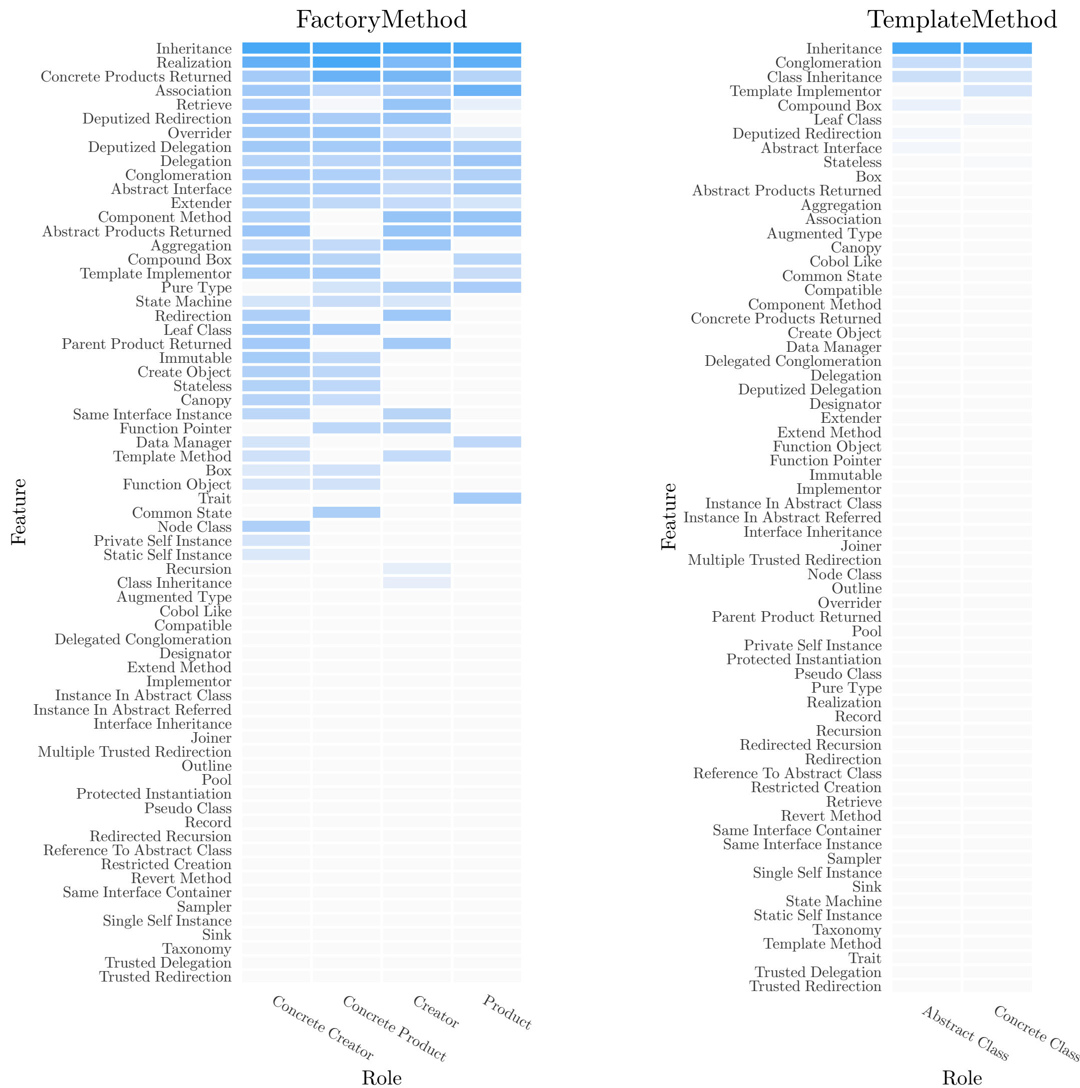
Figure 5.9.: Feature Importance for the Adapter and Composite roles. The darker the tile the more important the feature for the specific role. Tiles are sorted by their coverage of DP roles and total importance.



(a) Importance measures for all micro-structures with respect to the Decorator pattern (darker is better).

(b) Importance measures for all micro-structures with respect to the Singleton pattern (darker is better).

Figure 5.10.: Feature Importance for the Decorator and Singleton roles. The darker the tile the more important the feature for the specific role. Tiles are sorted by their coverage of DP roles and total importance.



(a) Importance measures for all micro-structures with respect to the Factory Method pattern (darker is better).

(b) Importance measures for all micro-structures with respect to the Template Method pattern (darker is better).

Figure 5.11.: Feature Importance for the Factory Method and Template Method roles. The darker the tile the more important the feature for the specific role. Tiles are sorted by their coverage of DP roles and total importance.

total 21 layers. There are two convolution layers at the beginning of the network followed by three dense layers. The shape of the input tensor has one channel, a height of 67 (features) and a width of 3 (e.g, roles for the composite pattern). The output shape is one scalar representing the probability that the input is of a certain pattern. b within the output shapes of Table 5.6 is the amount of observations within one batch of samples. Each convolutional layer is followed by a relu activation and a max pooling layer. Dense layers use relu or tanh as activation and are preceded by dropout layers that help to regularizing the model. The filter of the convolutional layers have a width equal to the amount of roles and a height of 3. Max pooling reduces the size of the feature maps by a half along the height (features) but does not pool along the width (roles). Thus the model learns abstract concepts among the features but always considers the entirety of the DP roles. Resulting models have a total of nearly 1.5 million parameters that are learned over the course of the training. Batch normalization [42], a normalization layer that normalizes the activations after each layer, could not provide any advantages during training. Assumptions are that it behaves similar to the z-score damaging the identification properties of the coefficients of the feature maps. No advanced network topologies and layers (e.g., RNN, LSTM, ResNet) were used in the experiments as the goal was to assess the basic applicability of deep learning in the context of DPD.

Training the network given in Table 5.6 directly on the real dataset results in majority predictors because of the extreme imbalance. Easy ensemble helps to improve the learnability of the data by balancing the class distribution for each ensemble participant (weak learner). This means that instead of one model with 21 layers and 1.5 million parameters, k copies are trained each exposed to a different subset of the majority class in the datasets. This approach enables each weak learner to properly converge while still letting the ensemble model the true distribution. The initial network from above is used as blueprint for the ensemble participants in which the network topology is fixed. This limits the set of possible parameters that need to be searched in order to find the optimal configuration for the bootstrapped sampled dataset. What remains are the layer parameters that need to be selected such that the model performs best. Maximizing the model performance by the correct selection of model parameters is often a "black art" and requires expert experience or brute-force methods. Snoek et al. [45] circumvent this problem by constructing a Bayesian optimization problem that maximizes the generalization performance of a given model over a set of configurations. Bayesian optimization builds a model of the parameters and their performance such that the next best configuration can be predicted with respect to the uncertainty. The uncertainty is off course the classifiers performance under a certain parameter configuration, i.e., the DPD detector and its performance with a set of hyper-parameters. A Gaussian process, a function of the form $f : \chi \to \mathbb{R}$, is used within the optimization as prior distribution and functions as assumption how the parameters are distributed. An acquisition function, e.g., Probability of Improvement, Gaussian Process Upper Confidence Bound, etc, is than used to make the actual prediction of the next best configuration. So instead of searching parameters on a fixed grid, randomly or via a brute-force method, Bayesian hyper-parameter optimization builds a proper model of the parameter configurations and their performance implications

Table 5.7.: The basic network topology derived via the synthetic dataset. Output Shape describes the shape of the output tensor of the respective layer and parameter count the number of trainable parameters. The network contains 2 convolutional layers that drive 3 dense layers plus the output layer.

Layer	Type	Output Shape	Parameter Count	Connected to
1	Input Layer	(b, 1, 67, 3)	0	-
2	Convolution2D	(b, 8, 67, 3)	80	1
3	Activation (relu)	(b, 8, 67, 3)	0	2
4	MaxPooling2D	(b, 8, 33, 3)	0	3
5	Convoluation2D	(b, 16, 33, 3)	1168	4
7	Activation (relu)	(b, 16, 33, 3)	0	5
8	MaxPooling2D	(b, 16, 16, 3)	0	6
9	Dropout	(b, 16, 16, 3)	0	7
10	Flatten	(b, 768)	0	8
11	Dense	(b, 1024)	787,456	9
12	Activation (relu)	(b, 1024)	0	10
13	Dropout	(b, 1024)	0	11
14	Dense	(b, 512)	524,800	12
15	Activation (tanh)	(b, 512)	0	13
16	Dropout	(b, 512)	0	14
17	Dense	(b, 256)	131,328	15
18	Activation (tanh)	(b, 256)	0	16
19	Dropout	(b, 256)	0	17
20	Dense	(b, 1)	257	18
21	Activation (sigmoid)	(b, 1)	0	19
Total Parameter Count: 1,445,089				

such that an informed decision can be made. The final model parameters can be found in the Appendix A and are product of this optimization technique.

5.2.5. Model Training

Models within this work are trained via project-fold cross-validation, i.e., each project represents a fold that is used once for validation. This is necessary to fulfill the i.i.d. assumption and to make the performance evaluation between models reliable. Cross validation will result a deceivingly good performance for datasets where project samples are mixed between training and validation set because of the dependencies between pattern instances. As mentioned in Section 5.1.1, i.i.d. implies that changing one specific sample does not influence any other sample. This is to a large amount enforced by unique mappings as usually changing the classes assigned to the roles of one unique mapping does not influence a second mapping (of the same pattern type). Off course there might be the case where a class

participates in multiple unique mappings for the same pattern, e.g., a class might be the adapter in one, and adaptee in a second unique role mapping. The uniqueness property, that the defining roles are different, would still hold, nonetheless the case in which two observations do not influence each other can not be ruled out. Hence making folds on project boundaries is the safest approach to guarantee an appropriate level of independence. An implication of this design choice is that the cross-validation might not be as stable as expected since it can happen that the validation fold contains only very few example of the positive class. This may lead to a higher variance between the folds since it very likely that one of fold performs significantly worse than the others.

Figure 5.12 illustrates the training processes in which the dataset contains three projects $P1, P2$ and $P3$. Cross-validated bagged learning is used in contrast to bagged cross-validated learning as suggested by Petersen et al. [39]. First applies cross-validation among the ensembles, second applies cross-validation on the level of weak learners. As pointed out by Petersen et al., bagged cross-validated learning does not evaluate the performance of the ensemble properly because of the increased bias caused by double sampling. The intuition is that each weak-learner is evaluated correctly (via cross validation) hence would allow for proper model selection on a specific bootstrap set (sampled dataset for the weak learner). However this does not mean that the selected model parameters perform equally good if an ensemble of them is used, thus an evaluation of the ensemble rather of the weak learners should be used. Cross-validated bagged learning, as used in this work, splits the data into folds illustrated in the first level in Figure 5.12. Each fold consists of one project used as validation set (green) and the remaining projects as training set (blue). Easy ensemble is than applied to each of the folds in which the datasets are sampled multiple times illustrated by the second level in the Figure. The sampling process balances the class distribution such that each bootstrap dataset (e.g., dataset used to train Model 1.1) contains all minority samples that are available within the fold dataset, along with the same amount of majority samples. The majority samples between bootstrap datasets are independent such that the ensemble of models has knowledge about the entirety of the majority class, but each weak learner focuses only on a small proportion of it. Model 1.1. and Model 1.2. build up Ensemble 1, which is then validated on the entire validation set ($P1$) assessing the generalization performance of the ensemble. Evaluation results from the folds are than combined by averaging the performances, leading to the overall generalization performance of the model on the dataset. Despite the fact that cross-validated bagged learning is more reliable in its evaluation, bagged cross-validated learning might still prove value in setting where the cross-validation is unstable because of the class distributions (as it is the case in this work). The assumption is that the pessimistic bias introduced by the unstable cross-validation folds caused by validation sets in which only a handful of positive samples are present, is worse than the higher bias introduced by the validation of weak learners. It trades the extreme variance of the folds for a higher bias stabilizing the overall performance evaluation. Nevertheless this assumption needs to be evaluated in an isolated context thus exceeds the scope of this work.

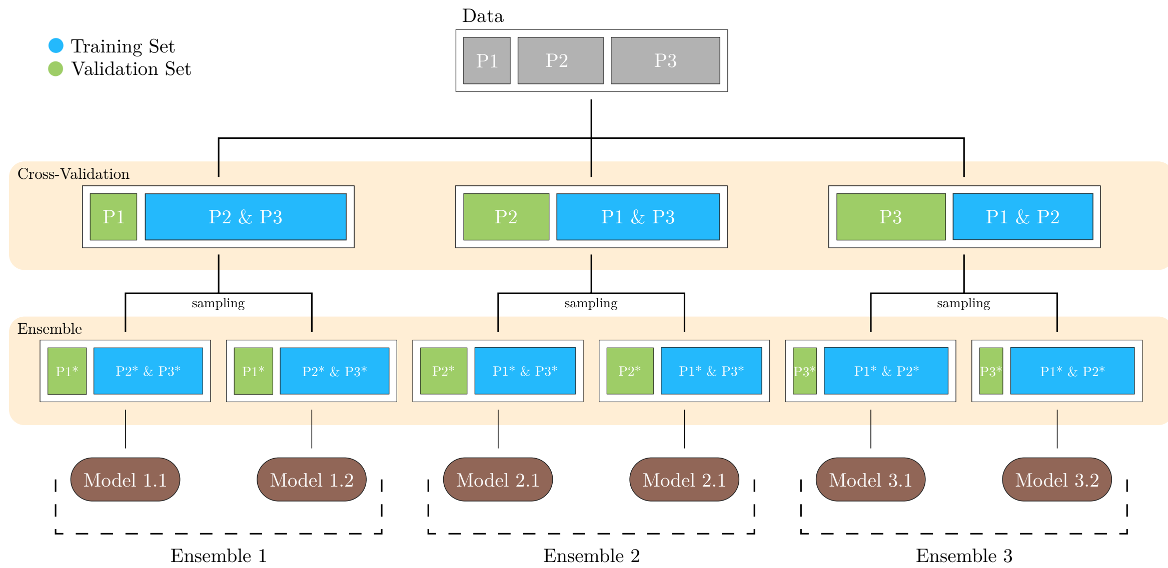


Figure 5.12.: The data is split on project boundaries into multiple folds (Cross-Validation). These folds train an entire ensemble by sampling different bootstrap dataset from it (Ensemble). One model per bootstrap set is trained and the average performance among the folds represents the generalization performance of the detector.

5.2.6. Model Evaluation

Several models are trained in order to validate one parameter configuration. The exact amount depends on the number of folds (projects) and the number of bootstrap sets that are used. On top of this, hundreds of parameter configurations are evaluated within a limited parameter space via Bayesian model optimization. This multiplied by the 6 patterns leads to thousands of models that are trained and evaluated thus makes the model selection a hard task. Models are evaluated via cross-validated bagged learning with the Matthews Correlation Coefficient as performance metric. The performance of the cross-validated parameter configuration is then the average performance of the folds (and their respective ensembles). The optimization expects a loss metric that is minimized as different model configurations are trained and evaluated. A simple approach is to define the loss as

$$\mathcal{L} = 1 - \frac{MCC + 1}{2}, \tag{5.14}$$

which normalizes the MCC to a range of $[0, 1]$ and converts it into a loss function with a maximum of 1 and a minimum of 0. $\mathcal{L}$ shares similar properties as MCC, i.e., it is less prone to imbalanced class distributions and provides a scalar measure of the performance representing the correlation between predictions and labels. Obviously $\mathcal{L}$ is not optimal in every situation as it does not allow to trade the performance between negative and positive classes. This is useful in situation where one class is more valuable than another class. Nonetheless $\mathcal{L}$ is appropriate in this setup up as it is merely a proof of concept and further optimizations with respect to the cost functions are postpone for future work.

Table 5.8 contains the performance measures of the final models for each pattern. The accuracy is very high for nearly all models (except for Adapter), nevertheless the values

should be handled with caution. Accuracy measures the performance of the classifier at one "optimal" threshold point. The threshold point is the cutoff probability at which a sample is classified as positive or negative and is based on the threshold that results the maximum MCC. Thus different cutoff points result into different accuracies which are sensitive to the imbalance of classes. A more informative measure is the Area Under Curve (AUC) that measures the discrimination performance of the classifiers given a pair of randomly chosen positive and negative examples. It is the area under a roc curve, e.g., area under the CV line in figure 5.13a and measures the performance across all cutoff points. All of detectors perform very good ($\mu = .932; \sigma = .066$) in terms of classifying the positive samples. The AUC is visualized in their respective ROC curve in the Figures 5.13a, 5.15a, 5.17a, 5.19a, 5.21a, 5.23a. AUC provides a reliable view on the classification performance with respect to the positive class, i.e., how many positive samples where correctly classified and how many negative samples where falsely labeled as positive class. The recall among pattern detectors is $\mu = .833; \sigma = .133$ thus has with a sightly higher standard deviation. All instances of the Decorator pattern are detected, and almost all instance of the Template Method (.913). Factory method is the worst pattern with a recall of .603 along with the best precision of .844. The worst precision is given by the Adapter with only .123 caused by the instable cross-validation, i.e., one validation fold contains only one example introducing a rather high variance between folds. The Composite detector has the second worst result of .328 impaired with a good recall of .809. The average recall is $\mu = .833; \sigma = .133$ and speaks for the usability of the method in terms of finding design patterns. An aggregated view on to all measures is given by MCC where the fit of the Singleton detector is very strong and above average with .813. The average MCC is $\mu = .581; \sigma = .244$ which is mostly pulled down by the Adapter pattern with a very weak fit of .139 caused by the low precision. All other detectors have either a moderate or strong fit hence confirms the benefit of the presented methods.

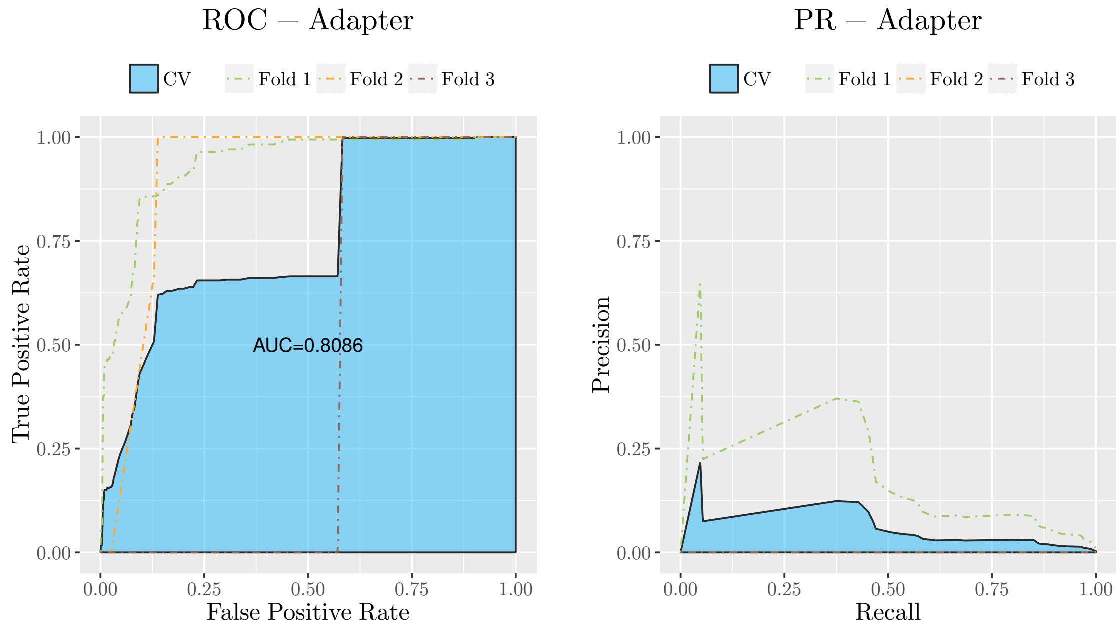
Table 5.8.: Performance metrics of the cross-validated ensembles for each of the 6 detected patterns.

Pattern	Accuracy	Precision	Recall	F-Score	AUC	Matthews Cor.Coeff.
Adapter	.754	.123	.809	.066	.808	.139
Composite	.961	.328	.818	.224	.928	.492
Decorator	.911	.689	1.00	.399	.950	.772
Factory Method	.985	.844	.603	.305	.984	.649
Singleton	.996	.809	.857	.398	.994	.813
Template Method	.871	.513	.913	.319	.927	.620

Adapter

Figure 5.13 shows the ROC and PR curve of the Adapter pattern. Folds are given as dash dotted lines and provides detailed insight into the evaluation. Fold 3 of the Adapter pattern contains $57,435$ negative examples along with only one positive sample. Similar

Fold 2 contains only 3 positive samples that come along with 36,617 negative samples. These extremes are the main source of the detector's weak performance and may be easily circumvented with more positive samples and a revised sampler that narrow the search space even further. The precision is rather instable as Figure 5.13b illustrates and drops rather quickly below .25 with increased recall.



(a) ROC curve and AUC for the Adapter classifier along with its fold performance.

(b) Recall/Precision curve for the Adapter pattern.

Figure 5.13.: ROC analysis of the Adapter model.

Figure 5.14 gives insight into the classes with respect to the cutoff. Each dot represents an observation within any fold of the cross-validation. This results into horizontal clusters and provides a detailed view on the threshold requirements of the final model. Violins illustrate the observation distribution on a specific thresholds abstracting the dot cloud for better readability. Negative samples are limited to 2000 observations to avoid over-plotting. The figure shows FN, FP, TN, TP with respect to the "optimal" cutoff point given at 0.63. The cutoff is the weighted-mean of all optimal cutoffs with respect to the fold performance. This may not be the optimal choice as the actual threshold depends on the value of the classes in a certain use-case. In some situation it might be better to predict conservative in some it is best to relax the threshold. Negative samples have two distinct clusters which is similar to the positive case. The positive case is good enclosed where the negative reaches up into the positive distribution. A detailed inspection of these overlapping negative samples would give a valuable insight into the sampled adapter candidates and where the samples, detector or the reviewed dataset needs to be improved. It can be concluded that the adapter patterns performance is not significantly worse than the other detectors, in fact the ROC curve indicates a good discrimitve performance. Nevertheless the amount of provided candidate samples need to be reduced plus additional positive samples within the folds is needed in order to leverage the full potential of the detector.

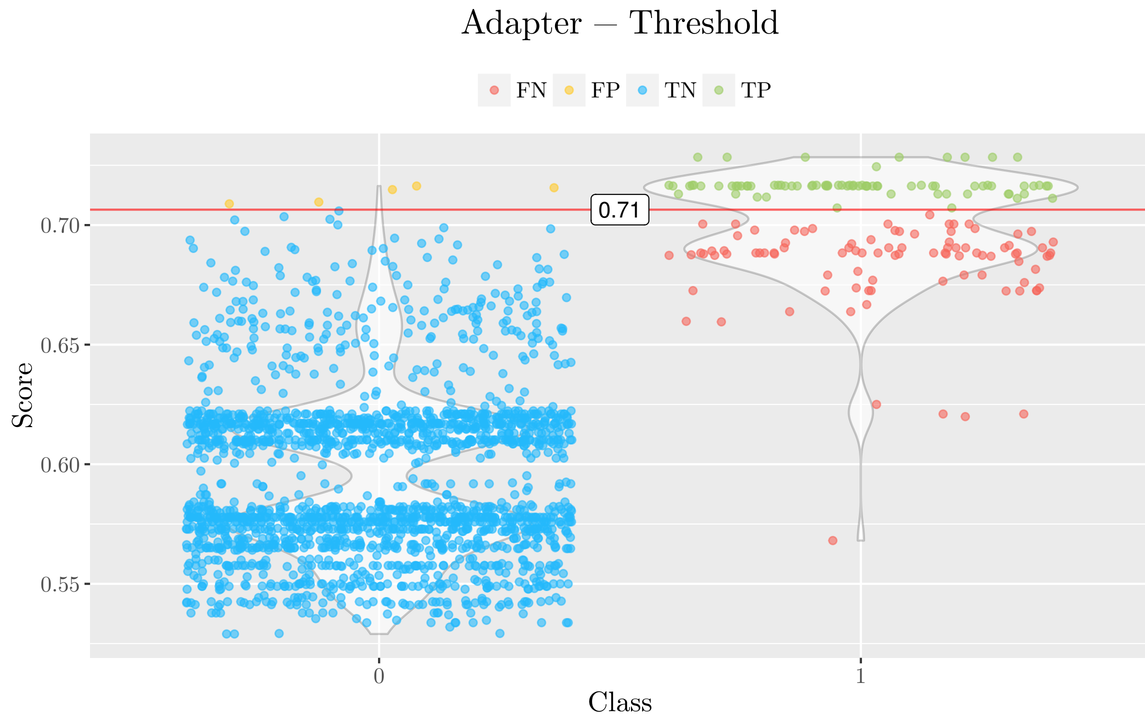


Figure 5.14.: The weighted average of thresholds among folds resulted into a proposed decision boundary of .71. It reduces the overall FP but causes an entire cluster of FN. Depending on the cost-function of the use-case a threshold that favors FP might be more appropriate.

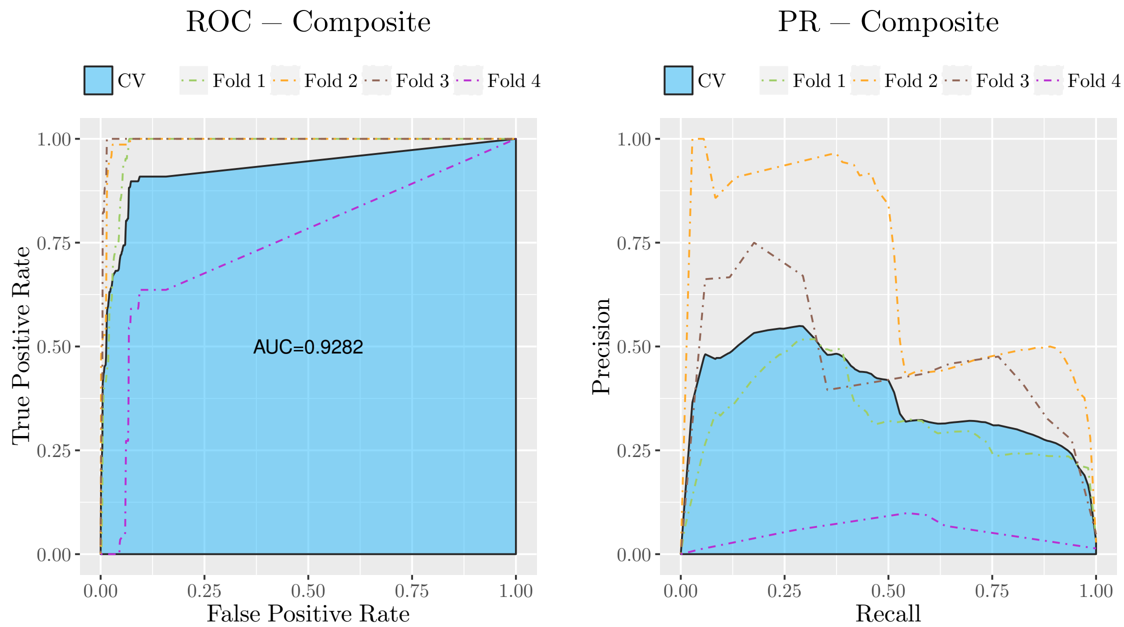
Composite

The ROC curve in Figure 5.15a for the composite pattern shows a very good fit for all folds except for fold 4. This indicates that the model learned a good representation of the positive examples. The PR curve in Figure 5.15b shows a rather high variance in terms of the precision undoubtedly caused by the imbalance. Fold 4 has an overall bad precision thus is an interesting point for further research. All other folds behave similar with a drop of precision at about .5 recall.

The optimal threshold is given at .61 in Figure 5.16. The negative samples produce three very distinct clusters that occur at the end of the positive extend. A rather harsh cutoff point is located at $\sim$ .44 at which the classifier abruptly sets the boundary for positive examples. The positive extend reaches far into the area of negatives samples and may be the combination of two distinct projects. All in all it can be concluded that the composite detector is good in detecting positive examples. Again the precision mostly dependents on the amount of sampled data but the optimization potential is rather low in the section. An additional analysis of Fold 4 and Fold 1 might prove valuable insight such that the overall fit can be solidified.

Decorator

The Decorator has only two folds thus is more stable in the the ROC (Figure 5.17a) and PR (Figure 5.17b) curve. Further more the Decorator datasets has the highest proportion



(a) ROC curve and AUC for the Composite classifier along with its fold performance.

(b) Recall/Precision curve for the Composite pattern.

Figure 5.15.: ROC analysis of the Composite model.

of positive samples thus provide valuable insight of the approach in a less extreme class distribution. The fit indicates a slightly higher proportion of FPR in order to reach the maximum TPR compared to the previous models. This is caused by the FPR denominator being magnitudes lower than in the above example resulting in a higher sensitivity within the graphs. The average precision is good and stable even with at higher true positive rates (recall).

Figure 5.18 illustrates a very interesting distribution of samples. Each class has two clusters belong to each other resulting in a rather questionable "optimal" threshold of .45. This leads to a rather deceivingly bad illustration of the facts as the image combines all datasets and fits into one graph. The most interesting part is that the clusters of positive and negative samples above the threshold overlap perfectly indicating instances in the dataset that are probably unlabeled decorator instances. This off course needs a throughout investigation but the cluster highly incentivise this assumption. The decorator is one of the strongest detectors among the others and the distribution within Figure 5.18 helps to understand why (despite illustrating it worse than it is). No FN are produced thus solidifying the assumption that the detectors learn a good representation about the true instances and mostly suffer from the extreme imbalance.

Factory Method

Factory Method with its wide variety of implementation styles provides a very good AUC and good precision with respect to the recall. This can be seen in Figure 5.19a in which the AUC of .984 indicates a very good fit for positive samples. The PR curve, which is more

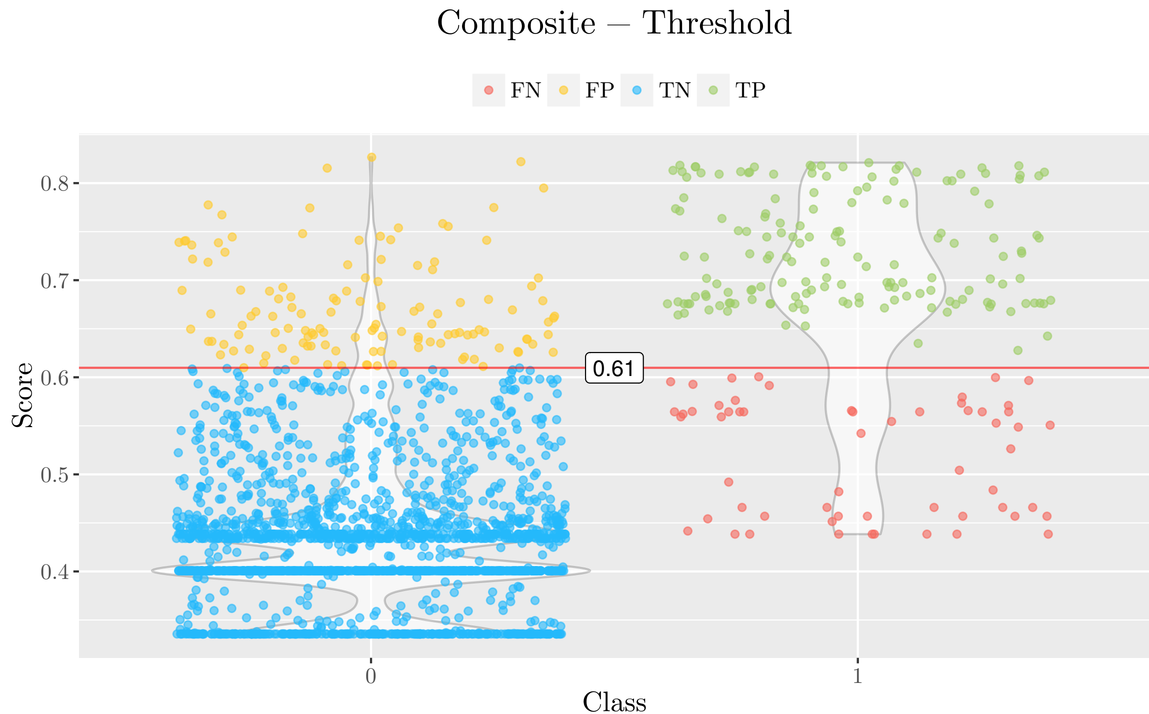


Figure 5.16.: The weighted average of thresholds among folds resulted into a proposed decision boundary of .61. The threshold balances the FP and FN quite well and might be appropriate in many use-cases.

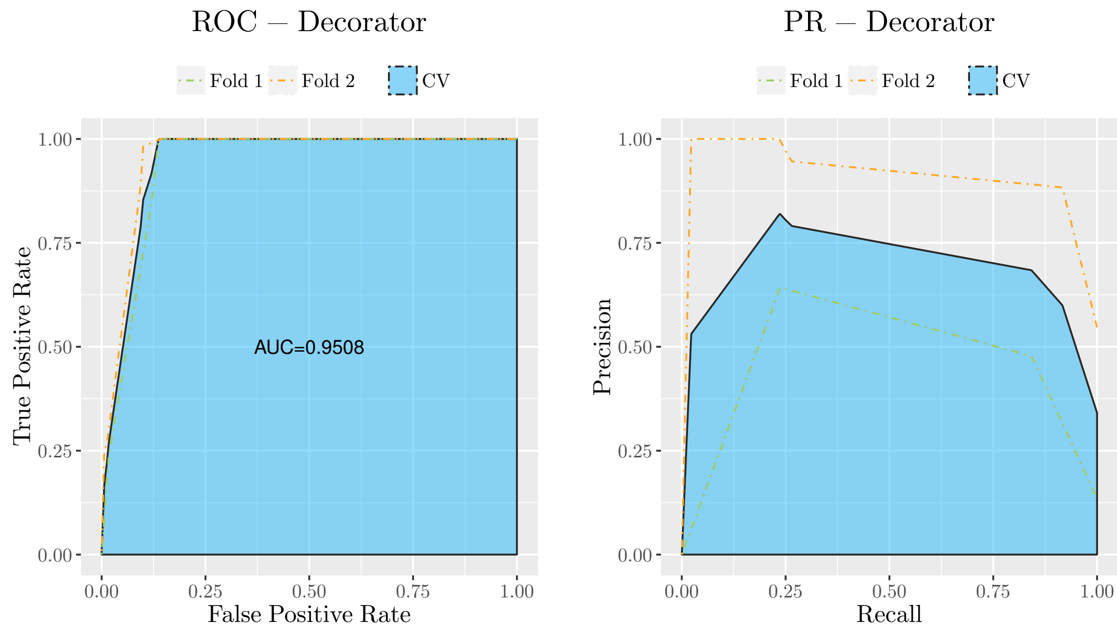
sensitive to the imbalance, shows a similar good behavior for Fold 1 and Fold 2, but an extreme decrease of precision for Fold 3.

Figure 5.19a helps understanding the problem a bit better in which two big clusters of negative examples exist. One reaches rather far into the positive distribution and represent probably samples that have the same structural properties as Factory Method instances. Factory Method has a rather generic structure in which a method (probably inherited) creates and returns a different type. This scheme matches on a lot of implementation situations but is seldom considered as the implementation of a certain pattern. The cutoff point within the Figure are extremely skewed because of the high variance between the fold thresholds. A more appropriate threshold would probably be at ~ 55 excluding a big batch of FP. All in all in can be concluded that the general performance is good with respect to the precision but moderate with respect to the recall.

Singleton

The singleton detector is the best with respect to precision and recall. Of course this is caused by the fact that Singleton has the second highest proportion of positive instances within the dataset ($6.999 \cdot 10^{-3}$). Both, ROC and PR curve, given in Figure 5.21a and 5.21b, show very good average performances among the folds.

Figure 5.22 illustrates the observation distribution in which only a handful of examples (13 in total) are present on the positive side. The negative samples indicate a rather uniform distribution over the entire threshold space where the positive samples are mostly located



(a) ROC curve and AUC for the Decorator classifier along with its fold performance.

(b) Recall/Precision curve for the Decorator pattern.

Figure 5.17.: ROC analysis of the Decorator model.

the the end of the spectrum. The expectations for the Singleton pattern detector are that it (nearly) perfectly detects the pattern because of its simplicity. Results indicate that this expectation is fulfilled to the biggest parts but still has open potential for improvements.

Template Method

The template method detector has a moderate fit in terms of recall with respect to the FPR as Figure 5.23a shows. Perfect recall is reached at a FPR of .4 implying a rather high cost for full coverage. The Precision drops at .5 recall below .5 similar to the other detectors but again increases at a recall of 1.

The distribution shows two distinct blobs in which a second cluster extends within the range of the positive samples. Again it can be expected that these samples are unintentional instances of the pattern caused by the generic structure of the Template Method pattern. The amount of FNs is low similar to the other detectors, where the main problem is given by the FPs.

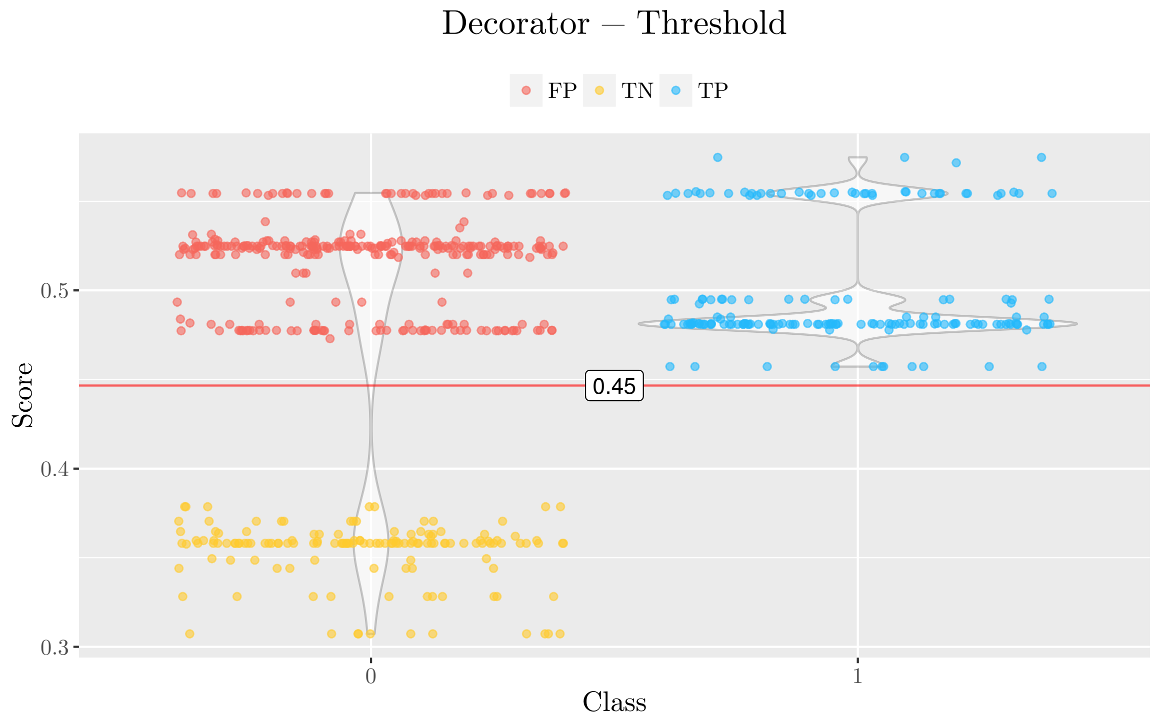
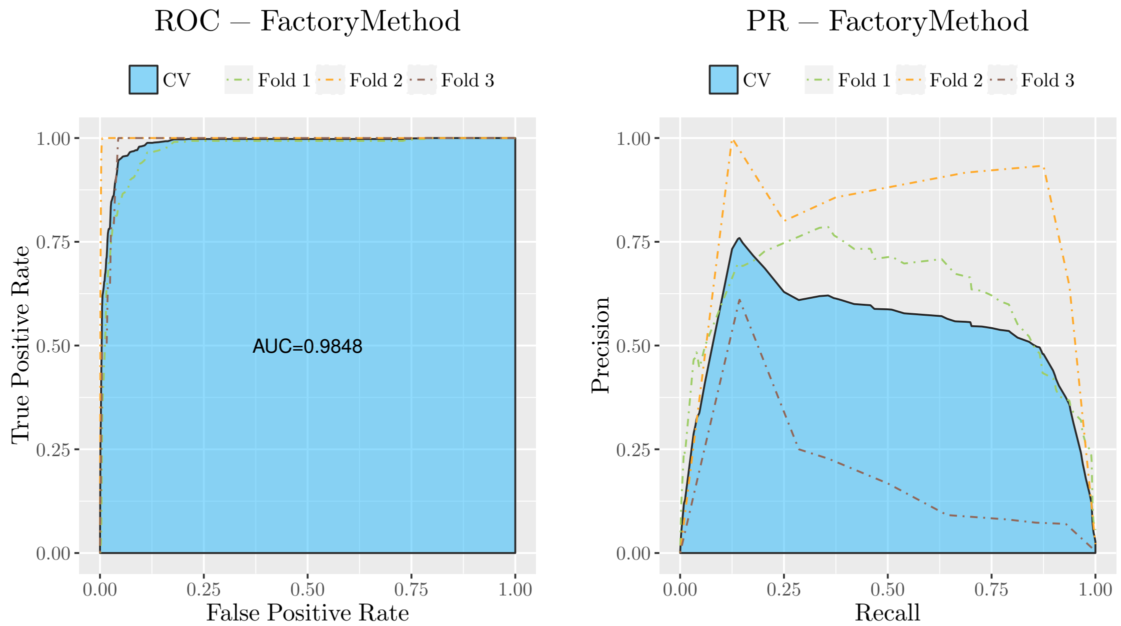


Figure 5.18.: The weighted average of thresholds among folds resulted into a proposed decision boundary of .45. It produces quite a lot of FP that may be unintentional Decorator instances.



(a) ROC curve and AUC for the Factory Method classifier along with its fold performance.

(b) Recall/Precision curve for the Factory Method pattern.

Figure 5.19.: ROC analysis of the Factory Method model.

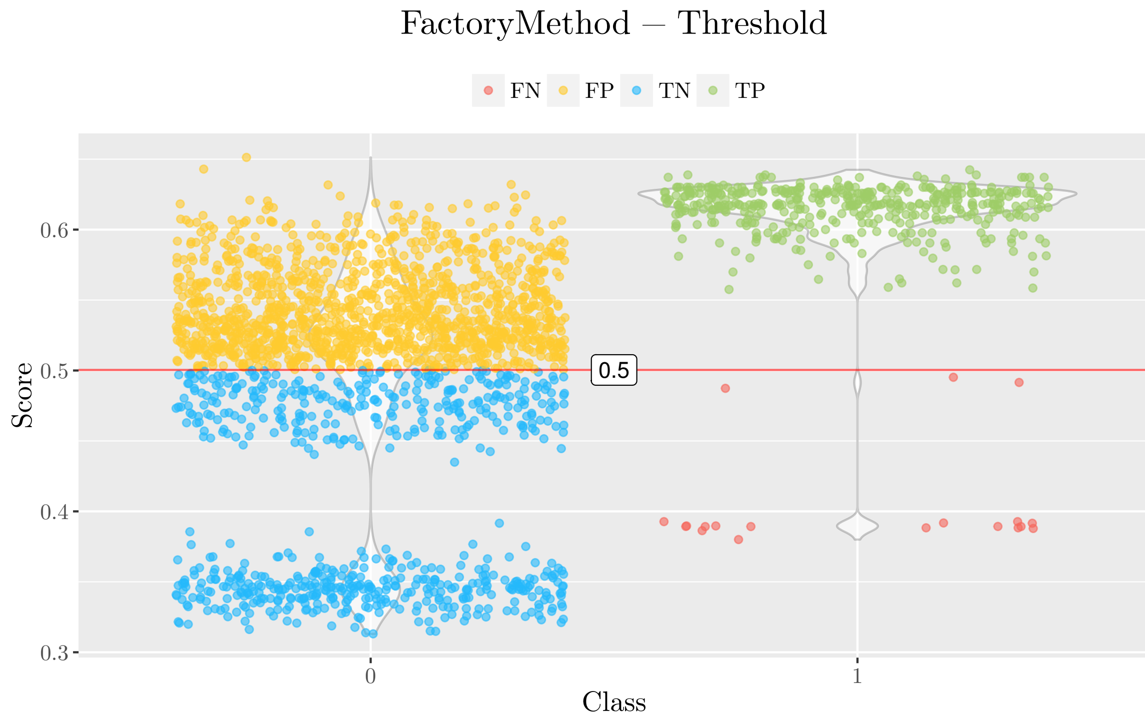
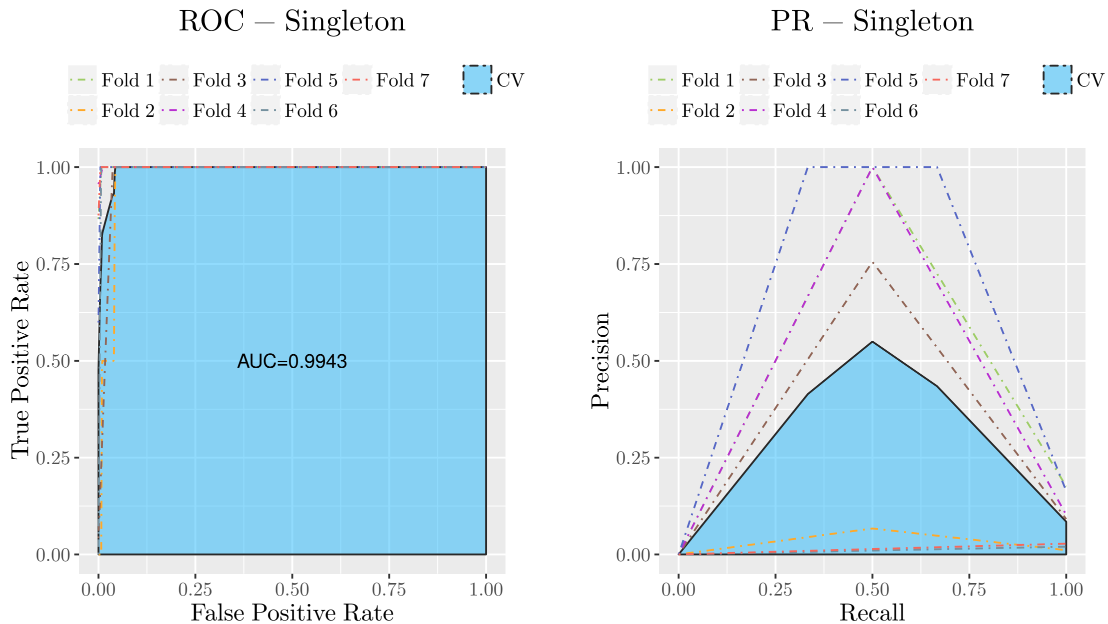


Figure 5.20.: The weighted average of thresholds among folds resulted into a proposed decision boundary of .5. The proposed threshold is sub-optimal caused by the difference of the folds. More appropriate would be a threshold of 0.55.



(a) ROC curve and AUC for the Singleton classifier along with its fold performance.

(b) Recall/Precision curve for the Singleton pattern.

Figure 5.21.: ROC analysis of the Singleton model.

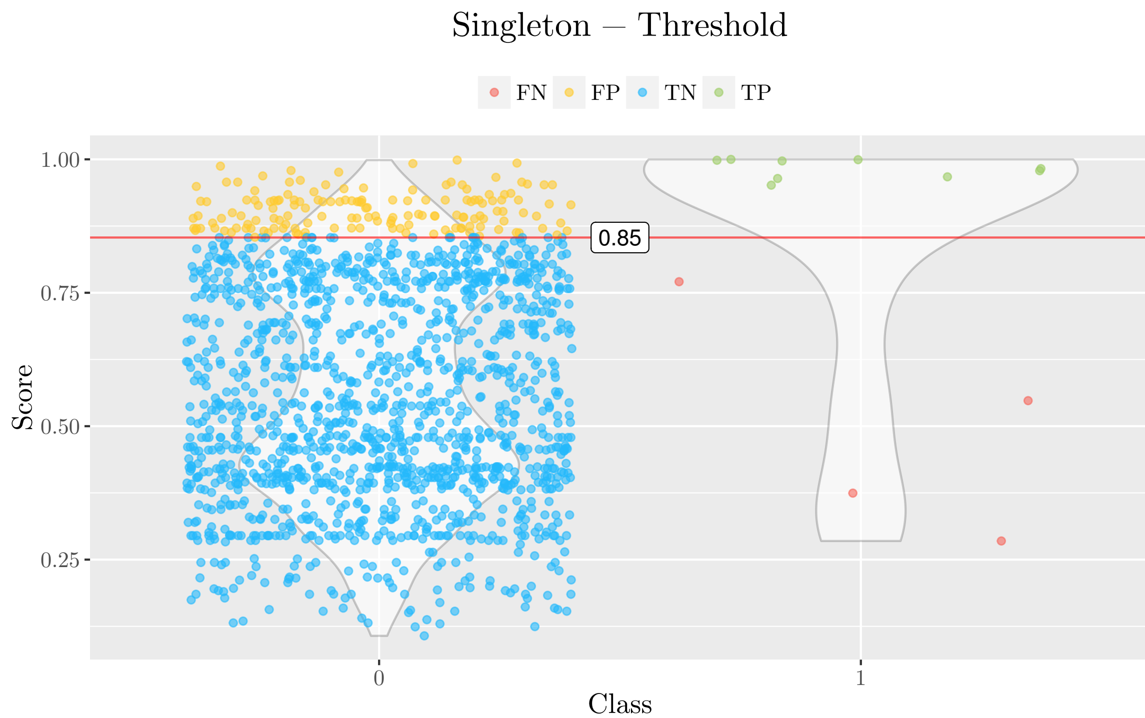
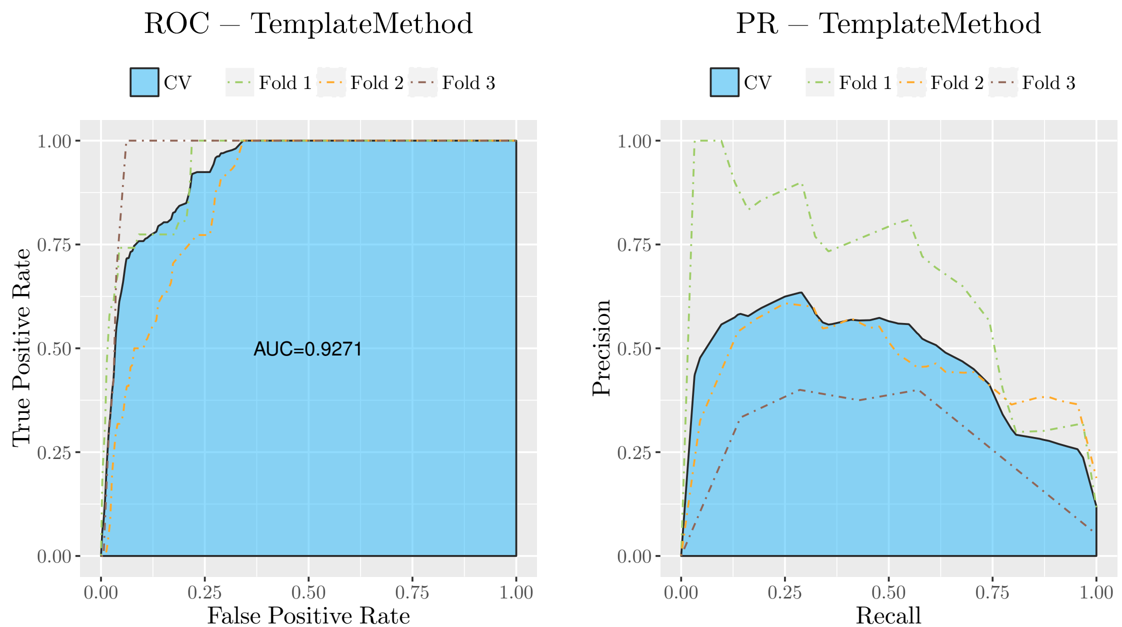


Figure 5.22.: The weighted average of thresholds among folds resulted into a proposed decision boundary of .85. The proposed threshold is sub-optimal as a higher value might produces better results.



(a) ROC curve and AUC for the Template Method classifier along with its fold performance.

(b) Recall/Precision curve for the Template Method pattern.

Figure 5.23.: ROC analysis of the Template Method model.

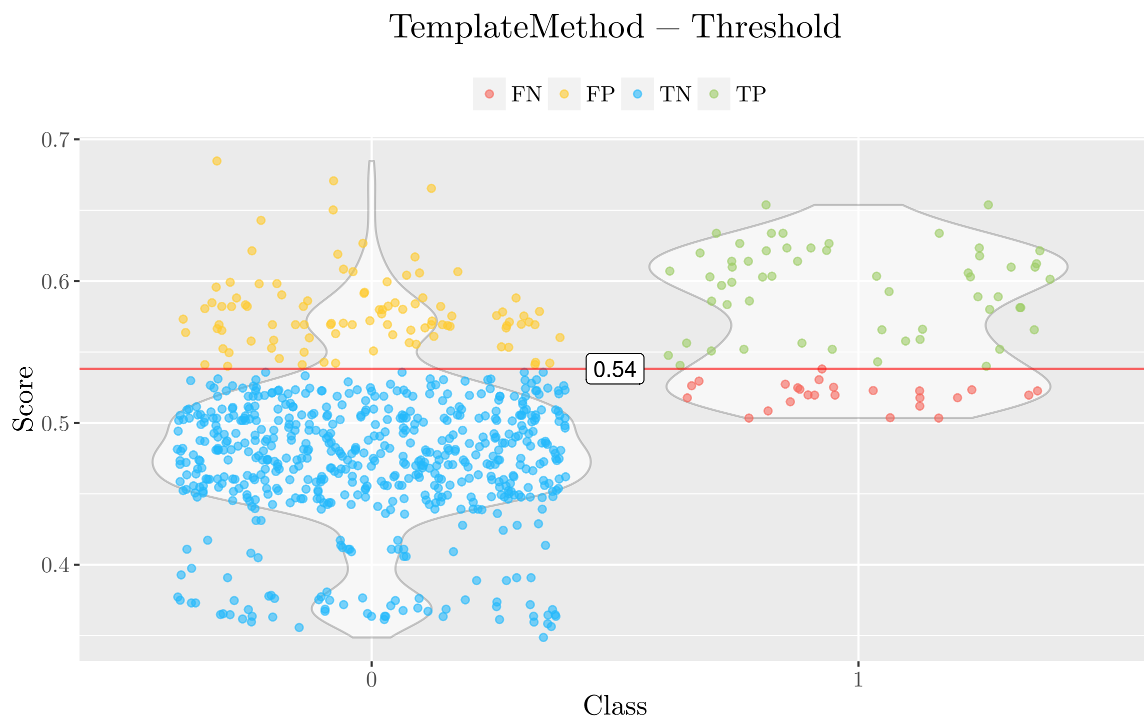


Figure 5.24.: The weighted average of thresholds among folds resulted into a proposed decision boundary of .54. The proposed threshold produces a good balance between FN and FP thus might be appropriate in many situations.

6. Conclusion

This work presented a fully fledged design pattern detection approach ranging from source code to the actual classification decision. The initial code was transformed into an ASG from which micro-structures where extracted. These micro-structures represent sub-graphs within the ASG and express predefined, named and well-understood OO concepts. Based on these concepts, an intuitive sampling method was presented that extract candidate classes for a specific pattern from the system. This effectively reduces the search space the source system classes and design pattern roles span. Furthermore it pointed out problems in the available peer reviewed data and confirms the need of a platform, on which researches and developers are able to create comprehensive and manifold datasets. Feature maps were generated by the feature role normalization and provides means to represent the actual sub-graphs of the candidate mappings in form of a tensor. Additionally role-role and feature-channel normalization were proposed as potential alternatives to FRN. Future work can benefit from the detailed discussion of the normalization approaches and should open a dialog about the various implications. The basic notions of machine learning where discussed on an intuitive level by providing an introduction to the main topics of the data analysis workflow. Standard methods are elaborated such that a common understanding of methods, pitfalls, and responsibilities is established with pointers to their respective related work. This helps researchers that are interested in DPD but are unfamiliar with machine learning techniques to grasp the basics and ease the entrance level difficulties. Feature maps are used as target medium for convolutional neural networks and is presented in Chapter 5. These convolve over a multitude of different aspects of a given candidate mapping represented by the micro-structures. The deep network structure helps to build robust classifiers that reach moderate to very good results even in the extreme imbalance that some datasets contain. This was achieved by using an augmented dataset with synthetically generated negative examples such that a basic network topology could be found. The found topology was then trained using the Easy ensemble approach such that, despite the hindering imbalance within the datasets, a good learning progress could be achieved.

We have shown that design patterns are detectable by modern machine learning methods, i.e., convolutional neural networks. This was done by using state-of-the-art network topologies, training and evaluation strategies that lead to very promising results. Furthermore, by using micro-structures to represent the ASG, we have further strengthened the importance of micro-structures as means of features. They provide a valuable abstraction of the core concepts while still being interpretable by humans such that advanced reasoning in the detection process is possible. We have presented yet another alternative that reduces the search space of the possible design patterns. The detailed analysis of the different sampling approaches provides, to the best of our knowledge, the first comparable results of the sampling step. It

showed that the combinatorial explosion can be overcome with simple techniques that traverse the ASG in a focused manner, collecting the valuable classes along the way. Nonetheless the provided approach still holds potential that may yield to an improved detection performance by avoiding excessive oversampling of the search space. This is mainly related to the Adapter sampler and further optimization may benefit the entire sampling phase of all sampler. However each sampler reduces the amount of candidate mappings to a manageable size while still retaining most of the true pattern instances thus fulfilling their main purpose. At last, this work answers the question whether role-mappings can be efficiently represented such that machine learning algorithms can handle the data, while still maintaining its inherent properties.

We are confident that this work provides a valuable first step for deep learning techniques within the domain of design patterns. There are many open issues and interesting opportunities for future research. Extending the feature catalog with the remaining MS and new features carries a lot of potential. More detailed information about the classes within a candidate mapping may enrich the detectors view onto the inherent properties of the graph and boost the confidence in the predictions. Optimizing the current sampling techniques may be, despite the usual focus on the detection step, one of the most important issues that can improve the overall results. This must include a throughout inspection of the misclassified examples to avoid over engineering of the samplers while still reducing the amount of proposed mappings. One of the most interesting open issue is the effect of the different normalization techniques with respect to the learnability of the data. In depth experiments might suggest one of the three normalization technique or even inspire a new one. Furthermore, the shear amount of machine learning algorithms and techniques, especially deep learning architectures, hold great potential for low hanging fruits in terms of detection performance.

Bibliography

[1] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, editors. *Towards machine learning based design pattern recognition: Computational Intelligence (UKCI), 2013 13th UK Workshop on*, 2013.

[2] Frances E. Allen. Control flow analysis. In Robert S. Northcote, editor, *a symposium*, pages 1–19.

[3] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002.

[4] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.

[5] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, 2011.

[6] Francesca Arcelli Fontana, Marco Zanoni, and Andrea Caracciolo. A benchmark platform for design pattern detection. In *PATTERNS 2010, The Second International Conferences on Pervasive Patterns and Applications*, pages 42–47, 2010.

[7] P. Baldi, S. Brunak, Y. Chauvin, C. A. F. Andersen, and H. Nielsen. Assessing the accuracy of prediction algorithms for classification: An overview. *Bioinformatics*, 16(5):412–424, 2000.

[8] Vincent D. Blondel, Anah\'ı Gajardo, Maureen Heymans, Pierre Senellart, and Paul van Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Rev*, 46(4):647–666, 2004.

[9] Christopher J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[10] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297, 1965.

[11] JING DONG, YAJING ZHAO, and T. U. PENG. A review of design pattern mining techniques: International journal of software engineering and knowledge engineering. *Int. J. Soft. Eng. Knowl. Eng.*, 19(06):823–855, 2009.

[12] Edward B. Duffy. *The Design & Implementation of an Abstract Semantic Graph for Statement-level Dynamic Analysis of C++ Applications.* PhD thesis, Clemson, SC, USA, 01/01/2011.

[13] James D. Evans. *Straightforward statistics for the behavioral sciences*. Brooks/Cole Pub. Co, Pacific Grove, 1996.

[14] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[15] Erich Gamma. *Design patterns: Elements of reusable object-oriented software / Erich Gamma ... [et al.]*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass. and Wokingham, 1995.

[16] Thomas Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49, 2003.

[17] Joseph Gil and Itay Maman. Micro patterns in java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 97–116, New York, NY, USA, 2005. ACM.

[18] M. Gori, M. Maggini, and L. Sarti, editors. *A recursive neural network model for processing directed acyclic graphs with labeled edges: Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 2, 2003.

[19] Gueheneuc Y-G, H. Sahraoui, and F. Zaidi, editors. *Fingerprinting design patterns: Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, 2004.

[20] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[21] Haibo He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[22] Karin Anna Hummel, Helmut Hlavacs, and Wilfried Gansterer. *Performance evaluation of computer and communication systems: Milestones and future challenges : IFIP WG 6.3/7.3 International Workshop, PERFORM 2010, in honor of Günter Haring on the occasion of his emeritus celebration, Vienna, Austria, October 14-16, 2010 : revised selected papers*, volume 6821 of *LNCS sublibrary. SL 1, Theoretical computer science and general issues*. Springer, Berlin and New York, 2011.

[23] Niall Hurley and Scott Rickard. Comparing measures of sparsity. *IEEE Transactions on Information Theory*, 55(10):4723–4741, 2009.

[24] Kersten Martin. Javameasurementtool: For measuring of java source code, 2002.

[25] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[26] Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software*, 28(5), 2008.

[27] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[28] Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. Exploratory undersampling for class-imbalance learning. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, 39(2):539–550, 2009.

[29] Wei-Yin Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[30] Kenneth C. Louden. *Compiler construction: Principles and practice / Kenneth C. Louden.* PWS Publishing, Boston, Mass. and London, 1997.

[31] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery through visual language parsing and source code analysis. *Journal of Systems and Software*, 82(7):1177–1193, 2009.

[32] S. Maggioni, editor. *Design pattern clues for creational design patterns*, 2006.

[33] S. Maggioni, F. Arcelli, C. Tosi, and M. Zanoni. Refining design pattern detection through design pattern clues. *submitted to the Journal of Systems and Software*, 2009.

[34] Stefano Maggioni. *Design pattern detection and software architecture reconstruction: an integrated approach based on software micro-structures.* PhD thesis, Università degli Studi di Milano-Bicocca, 01/01/2010.

[35] Tom M. Mitchell. *Machine Learning.* McGraw-Hill series in computer science. McGraw-Hill, New York and London, 1997.

[36] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In Will Tracz, Jeff Magee, and Michal Young, editors, *the 24th international conference*, page 338.

[37] Oracle. Swing, 2016.

[38] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, pages n/a–n/a, 2015.

[39] Maya L. Petersen, Annette M. Molinaro, Sandra E. Sinisi, and van der Laan, Mark J. Cross-validated bagged learning. *Journal of multivariate analysis*, 25(2):260–266, 2008.

[40] David Martin Ward Powers. Evaluation: from precision, recall and f-factor to roc, informedness, markedness and correlation. *School of Informatics and Engineering Technical Reports (24p ver.of ECAI'2008 Evaluation Evaluation)*, (SIE-07-001), 2007.

[41] Greg Ridgeway. gbm: Generalized boosted regression models. *R package version*, 1(3), 2006.

[42] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[43] Jonathon Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014.

[44] Jason McC Smith. *Elemental design patterns.* Addison-Wesley, 2012.

[45] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc, 2012.

[46] D. Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22(4):9–11, 2005.

[47] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[48] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014.

[49] Tessier Jean. Dependency finder. Internet download, 2001.

[50] Roman Timofeev. *Classification and Regression Trees(CART) Theory and Applications.* PhD thesis, Wirtschaftswissenschaftliche Fakultät, 20/12/2004.

[51] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring: Software engineering, ieee transactions on. *Software Engineering, IEEE Transactions on*, 32(11):896–909, 2006.

[52] Satoru Uchiyama, Hironori Washizaki, Yoshiaki Fukazawa, and Atsuto Kubo, editors. *Design pattern detection using software metrics and machine learning*, 2011.

[53] Rini van Solingen, Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal question metric (gqm) approach. In John J. Marciniak, editor, *Encyclopedia of Software Engineering.* John Wiley & Sons, Inc, Hoboken, NJ, USA, 2002.

[54] Wayne Beaton. Eclipse corner article: Adapters: Adapters, 2008.

[55] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[56] Yann-gaël Guéhéneuc. P-mart: Pattern-like micro architecture repository.

[57] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103:102–117, 2015.

A. Training Parameters

Table A.1, A.2, A.3, A.4, A.5, A.6 contains the actual parameters for Adapter, Composite, Decorator, Factory Method, Singleton and Template Method. The amount of evaluations (testing different configurations) depends on the performance of the best model an reaches from 50 to 200 evaluations. Presented parameters produce the results presented within Chapter 5 but may not be optimal, i.e., it is very likely that there is a parameter configuration that yields better results.

Table A.1.: Parameters for the adapter model.

Adapter	Parameter	Value
Data	feature selection	true
	preprocess	true
	bootstrap k	7
Training	optimizer	adam
	nb epoch	10
	learning rate	$7.24 \cdot 10^{-3}$
	batch size	31
	pos cls weight	1.30
	neg cls weight	0.79
Network	l1 nb row	4
	l1 nb col	4
	l1 nb filter	14
	l1 init	lecun uniform
	l2 activation	elu
	l3 nb row	1
	l3 nb col	1
	l4 nb row	2
	l4 nb col	2
	l4 nb filter	45
	l4 init	uniform
	l5 activation	elu
	l6 nb row	1
	l6 nb col	1
	l7 p	0.444
	l9 output dim	916
	l10 activation	elu
	l11 p	0.528
	l12 output dim	263
	l13 activation	elu
	l14 p	0.744
	l15 output dim	34
	l16 activation	tanh
	l17 p	0.651

Table A.2.: Parameters for the composite model.

Composite	Parameter	Value
Data	feature selection	false
	preprocess	false
	bootstrap k	6
Training	optimizer	adagrad
	nb epoch	10
	learning rate	$4.17 \cdot 10^{-3}$
	batch size	270
	pos cls weight	0.34
	neg cls weight	1.56
Network	l1 nb row	2
	l1 nb col	2
	l1 nb filter	1
	l1 init	lecun uniform
	l2 activation	tanh
	l3 nb row	2
	l3 nb col	1
	l4 nb row	4
	l4 nb col	3
	l4 nb filter	58
	l4 init	glorot uniform
	l5 activation	relu
	l6 nb row	1
	l6 nb col	1
	l7 p	0.090
	l9 output dim	674
	l10 activation	elu
	l11 p	0.310
	l12 output dim	103
	l13 activation	relu
	l14 p	0.322
	l15 output dim	972
	l16 activation	relu
	l17 p	0.895

Table A.3.: Parameters for the decorator model.

Composite	Parameter	Value
Data	feature selection	false
	preprocess	true
	bootstrap k	10
Training	optimizer	rmsprop
	nb epoch	10
	learning rate	$9.99 \cdot 10^{-3}$
	batch size	198
	pos cls weight	1.99
	neg cls weight	1.48
Network	l1 nb row	2
	l1 nb col	4
	l1 nb filter	59
	l1 init	lecun uniform
	l2 activation	tanh
	l3 nb row	2
	l3 nb col	1
	l4 nb row	2
	l4 nb col	4
	l4 nb filter	1
	l4 init	lecun uniform
	l5 activation	relu
	l6 nb row	2
	l6 nb col	1
	l7 p	0.959
	l9 output dim	449
	l10 activation	elu
	l11 p	0.99
	l12 output dim	470
	l13 activation	elu
	l14 p	0.0
	l15 output dim	125
	l16 activation	relu
	l17 p	0.585

Table A.4.: Parameters for the factory method model.

Factory Method	Parameter	Value
Data	feature selection	true
	preprocess	true
	bootstrap k	10
Training	optimizer	rmsprop
	nb epoch	49
	learning rate	$9.47 \cdot 10^{-3}$
	batch size	17
	pos cls weight	1.39
	neg cls weight	1.46
Network	l1 nb row	2
	l1 nb col	3
	l1 nb filter	30
	l1 init	uniform
	l2 activation	elu
	l3 nb row	2
	l3 nb col	1
	l4 nb row	2
	l4 nb col	3
	l4 nb filter	51
	l4 init	lecun uniform
	l5 activation	relu
	l6 nb row	2
	l6 nb col	1
	l7 p	0.998
	l9 output dim	774
	l10 activation	tanh
	l11 p	0.58
	l12 output dim	971
	l13 activation	elu
	l14 p	0.192
	l15 output dim	202
	l16 activation	tanh
	l17 p	0.100

Table A.5.: Parameters for the singleton model.

Singleton	Parameter	Value
Data	feature selection	false
	preprocess	false
	bootstrap k	7
Training	optimizer	adagrad
	nb epoch	10
	learning rate	$8.17 \cdot 10^{-3}$
	batch size	435
	pos cls weight	1.790
	neg cls weight	1.721
Network	l1 nb row	2
	l1 nb col	2
	l1 nb filter	63
	l1 init	uniform
	l2 activation	relu
	l3 nb row	2
	l3 nb col	1
	l4 nb row	3
	l4 nb col	2
	l4 nb filter	10
	l4 init	uniform
	l5 activation	tanh
	l6 nb row	1
	l6 nb col	1
	l7 p	0.99
	l9 output dim	100
	l10 activation	relu
	l11 p	0.0
	l12 output dim	993
	l13 activation	relu
	l14 p	0.457
	l15 output dim	1024
	l16 activation	relu
	l17 p	0.0

Table A.6.: Parameters for the template method model.

Singleton	Parameter	Value
Data	feature selection	false
	preprocess	false
	bootstrap k	10
Training	optimizer	adagrad
	nb epoch	10
	learning rate	$9.14 \cdot 10^{-3}$
	batch size	382
	pos cls weight	1.003
	neg cls weight	0.639
Network	l1 nb row	2
	l1 nb col	4
	l1 nb filter	51
	l1 init	uniform
	l2 activation	tanh
	l3 nb row	1
	l3 nb col	1
	l4 nb row	4
	l4 nb col	4
	l4 nb filter	4
	l4 init	uniform
	l5 activation	tanh
	l6 nb row	1
	l6 nb col	1
	l7 p	0.59
	l9 output dim	931
	l10 activation	relu
	l11 p	0.371
	l12 output dim	1022
	l13 activation	relu
	l14 p	0.103
	l15 output dim	321
	l16 activation	relu
	l17 p	0.993

Hannes Thaller

Curriculum Vitae

Education

2014–2016	**Master of Science**, *Johannes Kepler University*, Linz. Specialized in Computer Science
2011–2014	**Bachelor of Science**, *Johannes Kepler University*, Linz. Specialized in Computer Science
2010–2011	**Bachelor of Science**, *Johannes Kepler University*, Linz. Specialized in Information Electronics
2009–2010	**Bachelor of Science**, *Johannes Kepler University*, Linz. Specialized in Computer Science
2008–2009	**Vocational Matriculation Examination**, *BFI*, Wels.
2007–2008	**Military Duty**, *Panzerbataillon 14*, Wels.
2002–2007	**Technical School for Electrical Engineering**, *Polytechnic*, Wels.

Master Thesis

Title	*Towards Deep Learning Driven Design Pattern Detection*
Supervisors	Univ.-Prof. Dr. Alexander Egyed M.Sc.
Description	The thesis explored the capabilities of deep learning methods in the context of software design pattern detection.

Bachelor Thesis

Title	*Driver Performance Manipulation via Visual Subliminal Cues*
Supervisors	Dipl.-Ing. Dr. Andreas Riener
Description	The thesis explored the capabilities of visual subliminal messages in a driving setting to mitigate rear-end accidents.

Experience

Vocational

2015–Present	**Student Researcher**, Johannes Kepler University, Linz. Supportive position in a project that extracts features from a given set of derivatives to build up a product line.

Lindenstr 22 – Wels, Austria
+43 660 5225409 • *hannes.thaller.at@gmail.com*

2013–2016 **Tutor**, Johannes Kepler University, Linz.
Supporting the practical teaching program by correcting the exercise and answering questions.

2014 **Summer Intern**, ecx.io, Wels.
Developing, integrating and testing of a notification service.

2013 **Summer Intern**, ecx.io, Wels.
CRM System integration.

2012 **Summer Intern**, ecx.io, Wels.
Configuring a JBoss server and a Hibernate L2 Cache. Implementing a geo data compressor.

Publications

2014 **Subliminal Visual Information to Enhance Driver Awareness and Induce Behavior Change**, *6th International Conference on Automotive User Interfaces and Interactive Vehicular Applications (AutomotiveUI '14)*, Seattle.
The paper elaborates the results of the thesis 'Subliminal Visual Information to Enhance Driver Awareness and Induce Behavior Change'

Computer skills

Basic C, Haskell, Scala,
Intermediate LaTeX, Lua
Advanced Java, C#, python, R

Languages

German **Mothertongue**
Engish **Intermediate** *Conversationally fluent*

Interests

- Playing Guitar
- Running
- Machine Learning
- Traveling

Declaration

I hereby declare under oath that the submitted Master degree thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited. The submitted document here present is identical to the electronically submitted text document.