Politehnica University Timișoara
Faculty of Automation and Computers
Department of Computer and Information Technology

Towards the Detection of Correlated Type Usages in Generic Code

Dissertation Thesis

Alexandru Stana

Scientific coordinator
Conf. dr. eng. Petru Florin Mihancea

Timișoara
June, 2019

Table of contents

Chapter 1

Introduction

One of the main struggles a developer has to face when working on an object oriented system is trying to understand how different components interact with each other.

This increased difficulty in understanding a system is due to subtype polymorphism. Because of this feature a developer can understand that all usages of the parent of a class hierarchy can be substituted with any of its subtypes. This however might be a completely wrong assumption. In complex system there are certain situations which require that only some combination of types are valid, and only certain combinations of subclasses can be used by the client. This is a case where the so called *family polymorhism* is required [2]. This additional difficulty can be seen throughout multiple phases of the development life cycle:

- When trying to **understand** a software system, one must first understand the components in isolation and only afterwards the interactions between the components.
- **Maintaining** or **extending** can only be done when the interactions between components is known. Having this kind of knowledge will ensure that even though some modifications have been brought to existing components, or some additional components have been added the constraints between components have been kept.

In order to aid the developer in understanding the correlation between types and thus improve the development process the following chapters will present a tool which offers allows to analyze the codebase quickly and gain insights regarding the intended usages of certain types.

Chapter 2

State of the art

The following chapter presents some tools and methods which can be used in order to obtain a better understanding of different characteristics of a system(e.g. the intended usage of some hierarchies, the quality of the code, if some components are more error prone then others, etc.)

2.1 Analyzing the usage of types

A feature which introduces additional complexity in software systems and is present in all object-oriented languages is polymorphism. Polymorphism enables conceptual modeling [3], interface reuse and it also enables developers and architects to use well know coding conventions and design patterns.

Although the use of polymorphism greatly increases the flexibility and enables the developers to extend the existing software, it also increases the difficulty of understanding the system. This can even more be increased by the misuse of polymorphism. One such misuse is the wrong modeling of the class hierarchies. This can be seen when trying to modify or extend the system and although the modification respects the rules imposed by the language, it doesn't respect the additional rules embedded from within the structure.

Rules which are imposed from within the structure and are not visible from the structure itself(e.g. through the use of inheritance) can be observed by analyzing the clients which use the hierarchy. These additional rules may be used to limit the communication between components, and may have the scope to ensure the correctness of the software even when additional features are added.

Having these hidden constraints sometimes may shown a bad design of the class hierarchies. In some other cases the hierarchies have been designed in this way to simulate some additional features which the object oriented language doesn't support(e.g. family polymorphism [2]).

Proper usage of inheritance along with understanding and detecting design flaws in class hierarchies are just a few of the topics debated throughout the software engineering community [9, 1, 4, 13, 15].

2.1.1 Understanding class hierarchies

There are multiple ways proposed and implemented which offer developers ways to gain a quicker and more accurate understanding of a system(i.e. the interactions and constraints between components).

In [9] the author proposes a way of analyzing class hierarchies by looking on how they are used by their clients. This is done be defining a new set of metrics:

- PCR(Pure Code Reuse) is a metric which offer information regarding the intended usage of the class hierarchy
- TU(Total Uniformity), PU(Partial Uniformity), TNU(Total Non-Uniformity) are metrics which offer information which show if there are some constraints that are not obvious just by looking at the structure

Using the previously mentioned metrics the developer will be able to gain information regarding the intended usage of the class hierarchies much faster.

Another issue which must be tackled comes from the use of polymorphism and inheritance which may mislead developers during the software understanding activities. The authors in [13] identified a set of recurrent code patterns which mislead developers when trying to understand a system and offer some metrics which automatically detect such situations. Each metric targets a certain aspect might cause misunderstanding and it is constructed by the symptoms which describe the pitfall. Using this metrics(i.e. Partial Typing, Uneven Service Behavior and Premature Service) the developer can avoid falling victim to different misleads.

Graphical representation of a system can also provide meaningful insights which can be helpful when trying to understand a system. The authors in [14, 12] offer such representations, each of them tackling a different aspects.

The authors in [14] offer a 3D representation of the system in order to gain a quick overview. Since each class is represented using a 3D box, each characteristic of the box(e.g. height, width, color) can be used to display a metric. Using this type of representation the authors discover some patterns which appear in multiple projects:

- Large Root: this pattern reveals a root class with many children or many grandchildren
- Nested Families: this is represented by multiple levels of inheritance
- Exception Hierarchy: this pattern is represented by classes which don't define any instance methods. Usually this classes extends Throwable

Having the graphical representation of the system and the possible patterns which may exist, this overview of the system offers a quick and easy way to gain some valuable insights about the structure of the system.

In [10] and [12] the author presents a graphical representation which shows the polymorphic usage of class hierarchies. The views presented are named, "Level of Abstraction" and "Group Discrimination", both offering additional information to the developer regarding the usage of the polymorphic aspect in the client code. The "Level of Abstraction" view is based on a metric with the same name. The metric provides a value for each instruction which is proportional with the number of concrete classes a variable can be referred before executing the instruction. Using a color coding for the values produced by the metric, the "Level of Abstraction" view is created. This view offers some insight regarding the levels of abstractions used throughout a method. The second view, "Group Discrimination", shows which sub classes can be used for each instruction. Having this 2 views allow the developer to gain a much faster understanding of the class hierarchies and their usages.

When engaging in maintenance activities it is very useful for the maintainer/developer to have a good overview of the system and the implicit dependencies between entities. The aspect of detecting implicit dependencies between entities is tackled in [3, 4]. In order to discover the implicit relations between classes the authors use a technique known as Formal Concept Analysis which "identifies a set of recurring concepts among a set of elements with certain properties"[3](i.e. attributes accesses and method calls are known as elements and method calls which propagate through the class hierarchy are known as properties). With this additional information of the dependencies between classes enables the developer to modify existing features without affecting the integrity of the system.

The authors in [15] present a tool which helps developers understand the use of inheritance in their system. In order to offer this information about class hierarchies and their usages the authors offer answers for the following questions: "What are the interesting classes in an unknown program or framework?" and "What is the relationship between a subclass and its superclass?". The paper presents a set of metrics and rules which help identifying the different usages of inheritance. In order to answer the previous mentioned questions the authors focus on the following concepts:

- Large Classes (i.e. classes which have a large number of methods)
- Budding Classes (i.e. classes which provide more methods than their superclasses)
- Blooming Classes (i.e. classes which are a combination between Large Classes and Budding Classes)
- Subclassing Behaviors
- Overriding Behaviors

Grouping the classes according to these structural and behavioral categories offers the developer a quick overview of the system.

2.1.2 Detecting code smells

Besides understanding and visualizing class hierarchies another important topic is detecting design flaws, finding the source of these flaws and trying to find different correlations between them.

In [1] the authors propose a way of identifying the "Refused Parent Bequest" code smell through static code analysis and dynamic unit test execution. To identify if the code smell is present in a class hierarchy the authors propose to introduce errors in methods which are not overridden in the subclass and check if those errors appear at run-time. If the introduced errors cause the system to fail then the "Refused Parent Bequest" smell is not present because the subclass is not used only for its additional added methods, although if the system does not fail during run-time then this offers the developer a hint that the code smell could be present in the analyzed hierarchy. The proposed way offers the developers guidance for discovering design flaws.

The authors in [5] propose a study which aims to find if the "Interface Segragation Principle"(ISP) , "Program to an Interface not to an implementation"(PTIP) and cohesion can be joined or are conflicting design properties. In order to find the before mentioned correlations the authors analyzed a set of object-oriented applications. Using a set of metrics such as Service Interface Usage Cohesion, Sensitive Class Cohesion, Tight Class Cohesion and Loose Class Cohesion they identify that developers abide to ISP and PTIP principles but neglect the Cohesion property when designing an interface. Through this study, the authors have proven that the three design principles (i.e. PTIP, ISP and Cohesion) are not conflicting and developers only abide the first two thus neglecting the cohesion.

When detecting design flaws, design metrics are an important tool. In order to get a more accurate image of the code smell the developer can use a set of metrics which would provide some help. In [8] the author proposes a mechanism which enables developers to use metrics on a more abstract level. The mechanism proposed is called "Detection Strategy". The mentioned mechanism is composed of:

- "Filtering" which is used to reduce the initial data set
- "Composition" which is used to composes multiple metrics

The author offered a way for capturing design flaws through the use of a mechanism for formulating metrics-based rules.

2.1.3 Case studies

In order to find different correlations between code smells and other components such as polymorphism, the following authors conducted a set of case studies. The case studies gather different types of information from multiple projects and identify relations between components.

The authors in [11] conduct a case study in which they debate if there is a correlation between polymorphism, changes and defects. The study is aimed to answer if the following questions: "Is the intensity of the polymorphic calls from a class correlated with its change likelihood?" and "Is the intensity of the polymorphic calls from a class correlated with its defect likelihood?". In order to conduct this study they have used multiple versions of software systems and analyzed the polymorphic interactions between classes. To analyze the polymorphic interaction between classes the authors introduce the "Invocation Generality metric"(i.e. this being a particularization of the "Level of Abstraction metric" [10]). After conducting the case study the authors concluded that classes which use polymorphism are less likely to change and are also less error prone than other classes. Through their study the authors demonstrate that there is a correlation between classes using concrete types, polymorphism and their change and defect proneness[11].

Another case study which analyzes class hierarchies in order to find when are code smells introduced and when are they fixed is found in [7]. The authors wanted to answer the following questions: "When are code smells introduced?", "Why are code smells introduced?", "What is the survivability of code smells?" and "How do developers remove code smells?". The authors conducted their study on open source projects by investigating the commits made and picking out the ones which introduced code smells. Their case study focused on finding the following types of code smells: "God class", "Class data should be private", "Functional Decomposition" and "Spaghetti Code". After conducting the case study the following conclusions have emerged:

- a majority of code smells are introduced when the files are created, few result as a modification of existing files
- in general the people who introduce smells are the owners of the files and usually introduce them when they have higher workload
- about 80% of the code smells introduced remain in the system, the rest of them being removed in limited time period(100 days)

2.2 Problem

Traditional inheritance, polymorphism, and late binding interact nicely to provide both flexibility and safety when a method is invoked on an object via a polymorphic reference. Through the use of polymorphism we are granted the flexibility of using different kinds of objects and different method implementations, and we are guaranteed the safety of the combination. Nested classes, polymorphism, and late binding of nested classes interact similarly to provide both safety and flexibility at the level of multi-object systems. We are granted the flexibility of using different families of kinds of objects, and we are guaranteed the safety of the combination.

The problem arises when we use two or more independent hierarchies of classes together. In this case the collaborating "families" may consist of similar but not interchangeable classes. Because there can be subtype relationship between classes in the different groups, it is not obvious how to implement a constraint ensuring that only classes of the same family are used together. Traditional object-oriented languages like Java, don't offer out of the box support for this type of constraints between class hierarchies[6].

2.2.1 Handling graphs with traditional polymorphism

A common example used to present the issues which arise when trying to impose constraints on how elements of a hierarchy can interact with each other is by trying to model a graph.

A graph is defined by a set of nodes connected by a set of edges. Our following example will show two different types of graphs which should not be able to be communicate with each other.

Listing 2.1: Simple Graph implementation

```
abstract class AbstractNode {
  boolean touches(AbstractEdge e) {
        return (this==e.n1) ||
                (this==e.n2);
  }
}

abstract class AbstractEdge {
    AbstractNode n1, n2;
}

class Node extends AbstractNode {...}
class Edge extends AbstractEdge {...}
```

Listing 2.2: On Off Graph implementation

```
class OnOffNode extends AbstractNode{
  boolean touches(AbstractEdge e) {
        return ((OnOffEdge)e).enabled?
           super.touches(e) : false;
  }
}

class OnOffEdge extends AbstractEdge{
    boolean enabled;
}
```

In “Listing 2.1” and in “Listing 2.2” we have implemented 2 families of classes which cannot interact with each other due the exception which will arise if an `OnOffNode` receives an `Edge` instead of an `OnOffEdge`.

Another issue which may not be obvious at first is that the before mentioned constraint can be known only if the developer looks into the implementation. This can affect any future features which will be added later and depend on that specific part. The relation currently defined between the classes(i.e. inheritance) is not granular enough to avoid their misuse.

A way of avoiding the dynamic casts for forcing objects to have a certain type is by using parametric polymorphism. Although the parametric polymorphism solves the issue of receiving run-time exceptions due the invalid casts, it still doesn’t offer a way to tell the developer which combinations should be valid and which should be not.

Listing 2.3: Graph impementation using parametric polymorphism

```
class Node<T extends Edge>{
    boolean touches(T e) {
        return (this==e.n1) || (this==e.n2);
    }
}

class Edge<T extends Node>{
    T n1;
    T n2;
}

class OnOffNode extends Node<OnOffEdge>{
    boolean touches(OnOffEdge e) {
        return e.enabled? super.touches(e) : false;
    }
}
class OnOffEdge extends Edge<OnOffNode>{
    boolean enabled;
    OnOffEdge() {
        this.enabled=false;
    }
}
```

In “Listing 2.3” although the class cast exception has been avoided, there is no mechanism present which enforces the developer to associate an `OnOffNode` to an `OnOffEdge`.

2.2.2 Family polymorphism

Family polymorphism is a programming language feature that allows us to express and manage multi-object relations, thus ensuring both the flexibility of using any of an unbounded number of families, and the safety guarantee that families will not be mixed.

Traditional inheritance, polymorphism, and late binding of methods provide both flexibility and safety in the following sense. A polymorphic reference x may at run-time refer to an object which is an instance of some class C_i chosen from a set of classes C = {C_i ...C_k}. We may invoke a method m on x , typically using syntax such as x.m() , and each of the classes may provide its own method implementation for m or inherit an implementation defined elsewhere. Late binding ensures that the chosen implementation of m is the one associated with C_i (if any), the appropriate implementation for the actual object. Static type checking may be used to ensure that there is indeed an implementation for every invocation. In the example modeling a graph this can be seen both in the `Node`, `OnOfNode` classes and the method touches. The client code can call the method touches on a polymorphic reference of type `Node`. It doesn't really matter for the client the type of the object as long as it is a subtype of `Node`.

All in all, this provides the flexibility of using several classes and several method implementations, and the safety of ensuring that the chosen method implementation is always appropriate for the actual object. It is important to note that the same call-site, x.m() ,is reused with all those pairs consisting of a class and a method implementation; that it does not depend on the exact class of x or the exact choice of implementation of m; and moreover that the set of class/method pairs is open-ended.

Now consider the situation where two or more objects are involved, for instance where one object is given as an argument to a method on the other object, x.m(y) . In this case, traditional object-oriented languages such as the Java programming language will only allow us to associate two compile-time constant classes with this expression, namely the statically known class of x , C_x, and the statically known argument type of m ,the class C_m . At run-time, x may refer to an instance of any subclass of C_x and y may refer to an instance of any subclass of C_m . There is no way to ensure statically that a particular subclass C_x is always paired up with a particular subclass C_m. As in the previous example we were able of passing an `Edge` to an `OnOffNode` even though this would cause a run-time exception.

The fact is that the traditional notion of polymorphism is unable to capture relations between several objects and their methods, it only handles the case with one object and its methods.

The term family polymorphism is used to describe a generalized kind of polymorphism that will allow us to statically declare and manage relations between several classes polymorphically, in such a way that a given set of classes may be known to constitute a family – that family being characterized by having certain relations between its members. In [2] the author implements in `gbeta` a feature which ensures at compile-time that the relations between hierarchies are preserved.

Since Java doesn't offer this kind of polymorphism and some projects simulate this behaviour through the use of dynamic casts(i.e. although this may cause run-time errors) or parametric polymorphism(i.e. even though there can't be any static constraints imposed between the possible combinations) the tool described in the following chapter will offer Java developers some insights of possible family relations between objects(i.e. implemented using parametric polymorphism), which aren't obvious from the structure but can be spotted after analyzing the usage in the client code.

Chapter 3

Solution

Even though trying to simulate family polymorphism using parametric polymorphism is possible, it comes along with a significant downside. Trying to define static relations between "class families" is not possible, thus one cannot ensure at compile-time that all combinations of type arguments used throughout the code base are valid.

Because the Java programming language doesn't offer such a feature, developers simulate the relation between classes in different ways (e.g. using naming conventions, grouping classes in specific packages, creating subclasses, etc.). The issue is that this semantic coupling between classes might not be obvious, and it may require tedious work to discover the underlying logic.

The following tool offers an easy way to view all the combinations of types used throughout the code base, thus offering the developers a hint of how any future combination of types should be made. Additionally the tool can also show all the possible combinations which could be made using the subtypes of the type parameters. Besides the two lists the tool also offers a metric which shows the aperture of a type parameter(i.e. the ratio between all possible types and used types).

3.1 Design

Using the classes defined in the previous chapter, `Node`, `Edge` and `OnOffNode`, `OnOffEdge`, we created a client class which uses them.

Listing 3.1: Graph definition

```
class Graph<T extends Node, F extends Edge>{}
```

Based only on the information present in “Listing 3.1” (i.e. the subtype relation of the generic parameters) a developer cannot deduce that only some combinations of subtypes represent a valid usage of the class `Graph`.

In order to gain some insights regarding the valid combination of arguments which can be used as type arguments the developer would have to inspect all the usages of the class `Graph`. When looking after usages the developer would have to look in at least two places:

- **Hierachy usages**, the generic class `Graph` can be extended by multiple classes(e.g. “Listing 3.2”). Each class which extends it, defines a valid combination of type parameters(e.g. `Node` and `Edge` or `OnOfNode` and `OnOffEdge`). Seeing all these combinations can give the developer a hint which can direct him in the right way if he wants to further extend the hierarchy and mantain the implicit relations between classes

- **Variable usages**, in some cases developers might avoid using subtyping to offer new types, due to multiple reasons(e.g. additional functionality doesn’t need to be added to the hierarchy, to much boilerplate for just defining a type alias, etc.), in this case the workaround would be to define a variable and instantiate it with the appropriate types(e.g. “Listing 3.3”). Having this in mind, when looking after valid usages of a class one must consider looking at the variables as well.

Listing 3.2: Hierarchy usage

```
class SimpleGraph extends Graph<Node, Edge>();
class OnOffGraph extends Graph<OnOffNode, OnOffEdge>();
```

Listing 3.3: Variables usage

```
class Client {
    Graph<Node, Edge> simpleGraph = new Graph<>();
    void foo() {
        Graph<OnOffNode, OnOffEdge> onOffGraph = new Graph<>();
    }
}
```

Having all the usages of the class `Graph` the developer can now observe that the type arguments defined are always used in pairs and are

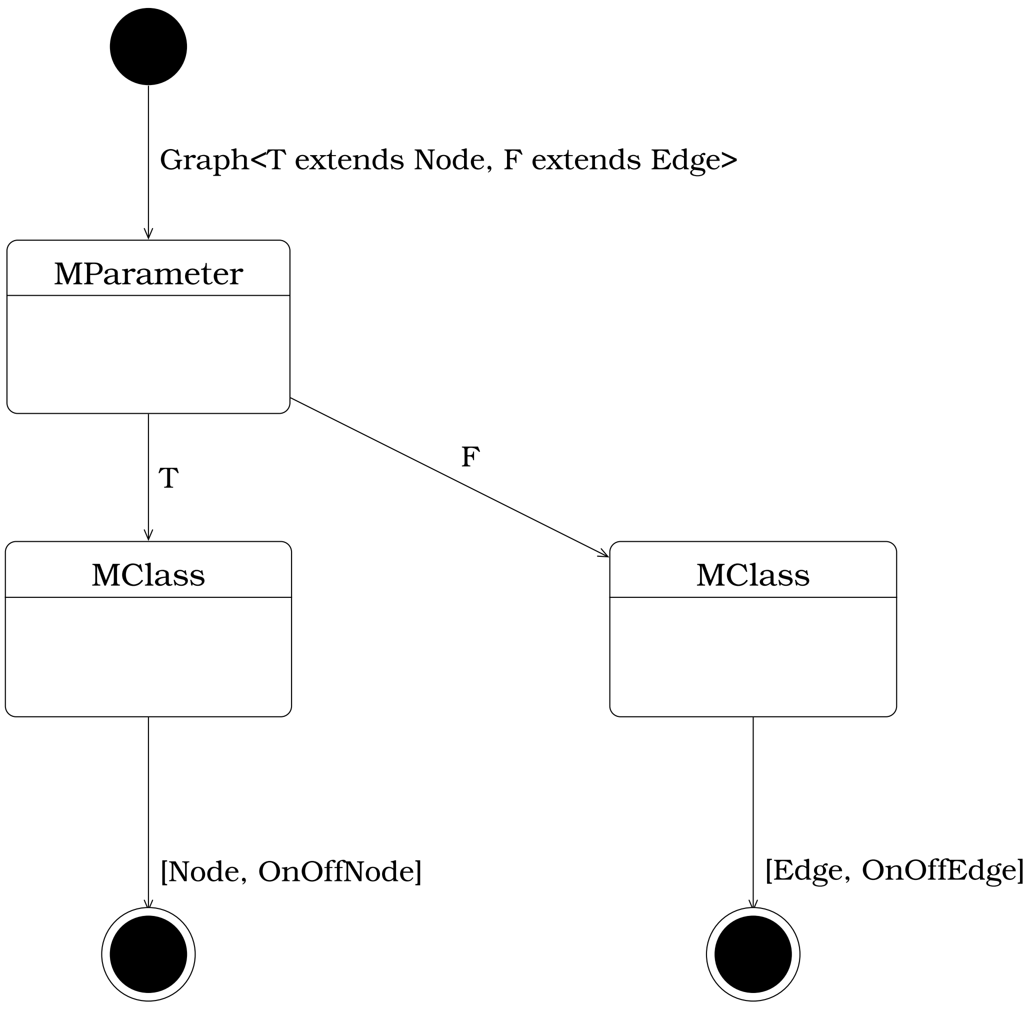

Figure 3.1: Data flow model

never combined. This might be a hint that the system relies upon this type of mixture, and it might fail is these conditions are not respected.

To model the information presented in “Listing 3.1”, “Listing 3.2” and “Listing 3.3” we decided to use the structure presented in “Figure 3.1”. Having each generic parameter mapped to a `MParameter` and each type argument mapped to a `MClass` allows us to group the information and offer the developers some additional information.

Because only showing individual parameters and their used types/all possible types might be to detailed and might not reveal implicit relations between the type parameters, in “Figure 3.2” is presented a model which groups these type parameters in pairs.

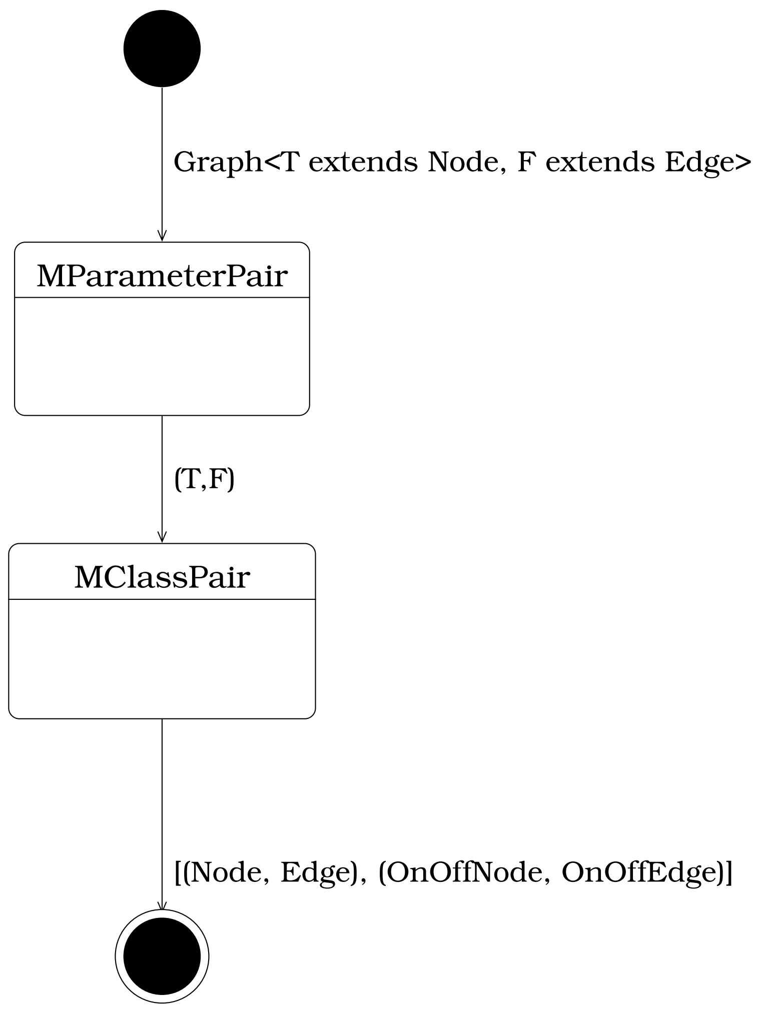
Graph<T extends Node, F extends Edge>
MParameterPair
(T,F)
MClassPair
[(Node, Edge), (OnOffNode, OnOffEdge)]

Figure 3.2: Pair parameters model

Having this representation a developer can observe that their might be some implicit relation relation between `Node` and `Edge` or `OnOffNode` and `OnOffEdge`.

Besides showing how the class is used by its clients the tool will also show all the possible combinations which could be made if the implicit relations between objects would be ignored.

In "Figure 3.1" there would be not difference when looking at all the possible arguments which can substitute the parameters `F` and `T` and showing all the types used in the client code(i.e. the client code uses all the possible subtypes). Instead when showing all the possible pairs which could be obtained by using the subtype of `T` and `F`, we would observe that an addtional pair of types, which is not used by the client, can be obtained. The client uses the pair `(Node, Edge)` and `(OnOffNode, OnOffEdge)`, but even though `(Node, OnOffEdge)` and `(OnOffNode, Edge)` are also valid substitutes for the type parameters, they are not used. As mentioned earlier this can even more show developers that there might be some additional coupling between the components involved.

3.2 Implementation

To be able to analyze the code base in order to detect how a client code uses certain hierarchies and if there are some implicit relations between the class hierarchies, we must first have access to the internal details of the analyzed code(e.g. types, subtypes, supertypes, hierarchies, etc.). For this we found that the easiest way to have access to the these details of the code base is to develop an eclipse plugin which uses Eclipse JDT for parsing and extracting information from the AST and XCore for modeling and storing this information. We have chosen the Eclipse JDT library for this because it provides a very powerful API which allows us to navigate the AST and extract relevant information from the different structures present throughout the code base.

Having the ability to chose the starting point, the developer can choose to gain informations regarding the entire project or only a specific class. Each of the elements used by the tool have an internal representation used by XCore. The project is persisted in an `MProject`, the generic classes are modeled as an `MClass` and the generic parameters for a specific generic class are modeled as an `MParameter`, or there are multiple generic parameters they can be modeled as an `MParameterPair`. Each of the previously mentioned entities offer a set of interaction the user can take.

3.2.1 Individual parameters

The `MParameter` offers a set of features such as:

- show types used by the client code.
- show all types which can substitute the generic parameter
- show aperture

When trying to obtain the **types used by the client code**, like mentioned in “Section 3.1”, the tool analyzes the code base and gathers all the type arguments specified in variables or in class hierarchies. All the gather types are then mapped again to an internal representation, `MClass`.

Obtaining all the possible substitutes for a type parameter doesn’t require us to analyze the client code, and instead only show all the child classes of analyzed type parameter.

When trying to obtain **all the possible type** we considered that there are two possible cases which must be considered.

Unbounded type parameter

When a generic class has an unbounded type parameter(e.g. “Listing 3.4), then this means it can’t make any assumptions on that type. Such a class can only act as a container for the type parameter. Retrieving all the possible parameters which can be used as a type parameter for such a class means retrieving all the classes which extend the class `Object`. Because in Java all classes extends `Object` this means that the result of returning all the types from the system and all the libraries used in the system.

Listing 3.4: Unbounded parameter

```
public class Foo<T>{}
```

Bounded type parameter

Having a bounded type parameter suggests that the declaring class might use some information from the super class or the super interfaces of the type parameter.

When getting all the subtypes of a bounded parameter, we must check the type of the bound. The generic parameter can be bounded by a superclass (e.g. “Listing 3.5), by multiple interfaces(e.g. “Listing 3.6) or by a combination between a superclass and multiple interfaces(e.g. “Listing 3.7).

Listing 3.5: Superclass bounded parameter

```
class Boo{}
class Boo1 extends Boo{}
public class Foo<T extends Boo>{}
```

Listing 3.6: Multiple Interfaces bounded parameter

```
interface Boo1{}
interface Boo2{}
interface Boo3 extends Boo1, Boo2{}
public class Foo<T extends Boo1 & Boo2>{}
```

Listing 3.7: Superclass and interface bounded parameter

```
class Boo1{}
interface Boo2{}
class Boo3 extends Boo1 implement Boo2{}
public class Foo<T extends Boo1 & Boo2>{}
```

When getting all the subtypes of a parameter which is only bounded by a class or an interface represents getting all the classes which extend or implement the parameters bound. In “Listing 3.5” all the subtypes which can be used are `[Boo, Boo1]`.

If the type parameter is bounded by multiple interfaces or a combination between a class and multiple interfaces, getting all the subtypes can prove to be a bit more difficult. In “Listing 3.6” and “Listing 3.7” only one type can fit the required constraints set by the type parameter. Finding it would require to search the hierarchies of both types and intersect the resulting lists, thus obtaining the types which can be used as arguments for the bounded types. In “Listing 3.6” only `Boo3` can be used as a type parameter and in “Listing 3.7” only `Boo3` can be used as a type parameter.

Having the types used in the client code as arguments for the type parameter and all the types which could be used as arguments for the type parameter enables us to show the developer a ratio between the two. The ratio shows how much of the type hierarchy is used throughout the client code, this giving the developer some insights on how the hierarchy is used and if there are some caveats which must be considered when using the hierarchy. This metric is named **Apperture**.

Using “Listing 3.2” the aperture for the type parameter `T` is:

$$aperture(TextendsNode) = \frac{|usedSubtypes(TextendsNode)|}{|allSubtypes(TextendsNode)|}$$

$$aperture(TextendsNode) = \frac{|(Node, OnOffNode)|}{|(Node, OnOffNode)|}$$

$$aperture(TextendsNode) = 1$$

Based on this value a developer would know that all the types of the `Node` hierarchy are used.

Listing 3.8: Additional node type

```
class FooNode extends Node{}
```

Adding an additional type to the `Node` hierarchy, like in “Listing 3.8”, would have the following changes.

$$aperture(TextendsNode) = \frac{|usedSubtypes(TextendsNode)|}{|allSubtypes(TextendsNode)|}$$

$$aperture(TextendsNode) = \frac{|(Node, OnOffNode)|}{|(Node, OnOffNode, FooNode)|}$$

$$aperture(TextendsNode) = \frac{2}{3}$$

$$aperture(TextendsNode) = 0.66$$

This suggests that some types of the hierarchy are not used throughout the code base. This can give the developer a hint that there may be some constraints which are not static and need to be considered, or the unused types can be removed from the system.

3.2.2 Pair parameters

Like `MParameter` the `MParameterPair` also offers a set of features such as:

- show pair of types used by the client code
- show all combination of types which can be used as arguments for the generic parameter pair
- show aperture

If the analyzed class has more then two generic parameters then the pairs are created using the formula for partial permutations and the total number of pairs is:

$$partPerm(analyzedClass) = \frac{numberOfGenericParameters(analyzedClass)!}{(numberOfGenericParameters(analyzedClass) - 2)!}$$

Having the example in “Listing 3.9”, the following pairs of parameters can be constructed: `(T,F); (F,H); (T,H)`

Listing 3.9: Multiple type parameters

```
class Foo<T,F,H> {}
```

When collecting all the **pair of types used in the client code**, the tool creates a list for each type parameter, and merge the two lists into a list of pairs which are afterwards mapped to an MParameterPair.

$$usedSubtypes(TextendsNode) = (Node, OnOffNode) = UT$$

$$usedSubtypes(FextendsEdge) = (Edge, OnOffEdge) = UF$$

$$usedSubtypes(TextendsNode, FextendsEdge) = zip(UT, UF)$$

$$usedSubtypes(TextendsNode, FextendsEdge) = ((Node, Edge), (OnOffNode, OnOffEdge))$$

Getting **all the possible types for a pair of types** involves getting all the types for both elements of the pair, and afterwards constructing all the permutations without repetition using the lists of both types.

$$allSubTypes(TextendsNode) = (Node, OnOffNode, FooNode) = AT$$

$$allSubtypes(FextendsEdge) = (Edge, OnOffEdge) = AF$$

$$allSubtypes(TextendsNode, FextendsEdge) = perm(AT, AF)$$

$$allSubtypes(TextendsNode, FextendsEdge) = ((Node, Edge), (Node, OnOffEdge), \ldots)$$

Having all the used pair of types and all the actual types which can be constructed using the type constraints (if any are present), the **Aperture** can be calculated.

Using "Listing 3.2" the aperture for the type parameters `(T,F)` is:

$$aperture(TextendsNode, FextendsEdge) = \frac{|usedSubtypes(TextendsNode, FextendsEdge)|}{|allSubtypes(TextendsNode, FextendsEdge)|}$$

$$aperture(TextendsNode, FextendsEdge) = \frac{|((Node, Edge), (OnOffNode, OnOffEdge))|}{|((Node, Edge), (Node, OnOffEdge), ...)|}$$

$$aperture(TextendsNode, FextendsEdge) = \frac{2}{6}$$

$$aperture(TextendsNode, FextendsEdge) = 0.33$$

Based on this value the developer can understand that not all possible combinations of types are valid and that there may be some implicit relation between the objects.

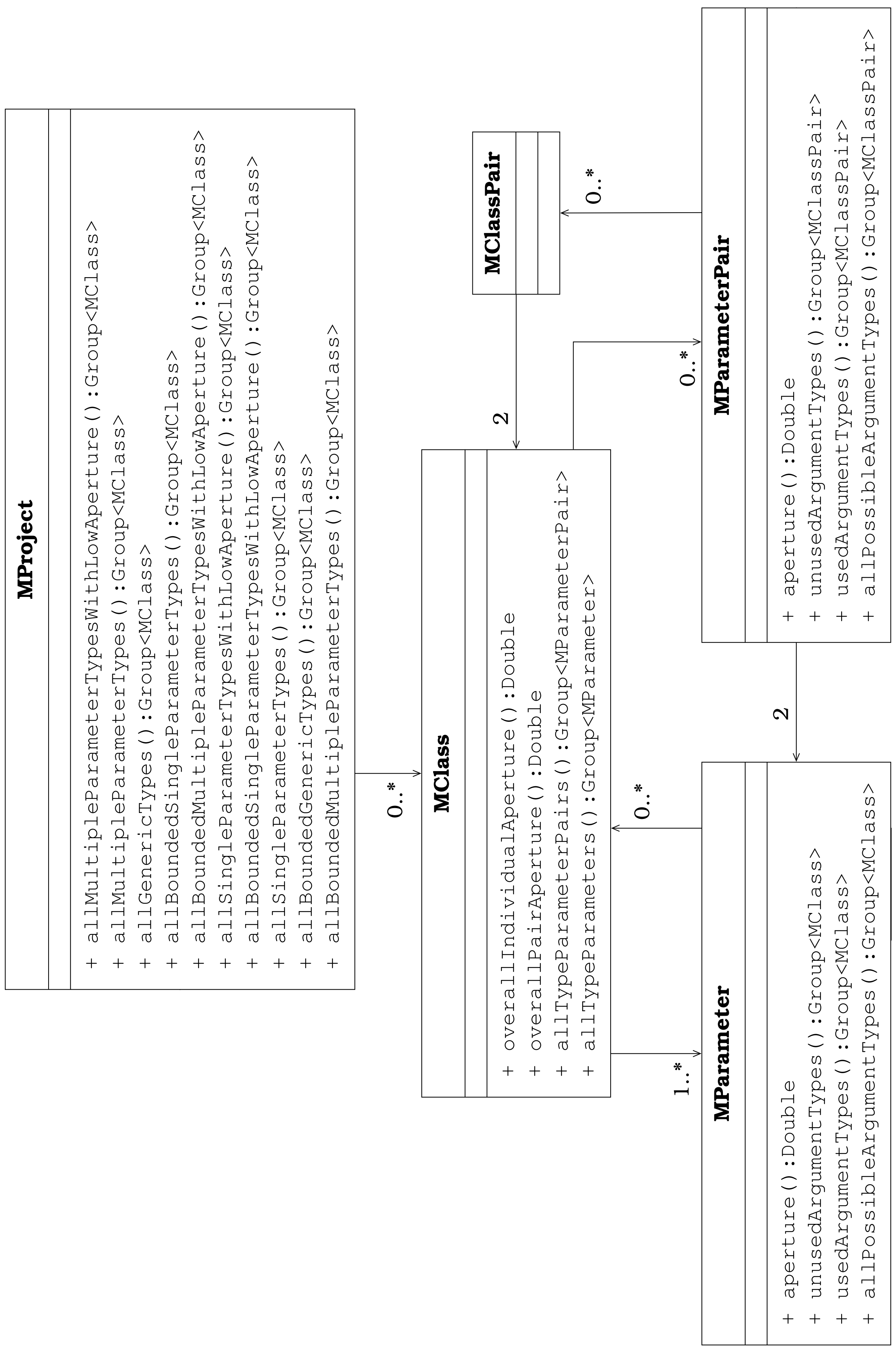
MProject
+ allMultipleParameterTypesWithLowAperture():Group<MClass>
+ allMultipleParameterTypes():Group<MClass>
+ allGenericTypes():Group<MClass>
+ allBoundedSingleParameterTypes():Group<MClass>
+ allBoundedMultipleParameterTypesWithLowAperture():Group<MClass>
+ allSingleParameterTypesWithLowAperture():Group<MClass>
+ allBoundedSingleParameterTypesWithLowAperture():Group<MClass>
+ allSingleParameterTypes():Group<MClass>
+ allBoundedGenericTypes():Group<MClass>
+ allBoundedMultipleParameterTypes():Group<MClass>
0..*
MClass
+ overallIndividualAperture():Double
+ overallPairAperture():Double
+ allTypeParameterPairs():Group<MParameterPair>
+ allTypeParameters():Group<MParameter>
MClassPair
2
0..*
0..*
MParameterPair
+ aperture():Double
+ unusedArgumentTypes():Group<MClassPair>
+ usedArgumentTypes():Group<MClassPair>
+ allPossibleArgumentTypes():Group<MClassPair>
2
0..*
1..*
MParameter
+ aperture():Double
+ unusedArgumentTypes():Group<MClass>
+ usedArgumentTypes():Group<MClass>
+ allPossibleArgumentTypes():Group<MClass>

Figure 3.3: Application architecture

Perspective	Parameters	All subtypes	Used types	Aperture
single parameters	[T]	[A, B, C, D]	[B, C]	0.5
pair parameters	[]	[]	[]	0

Table 3.1: Bounded one type parameter

3.2.3 Architecture

Based on the features presented in “Section 3.2.1”, “Figure 3.3” shows the relations between components and the actions each components allows the user to take. Since there are little differences if we are to draw a second diagram based on the features presented in “Section 3.2.2” we elided drawing it.

3.3 Usage

The following section presents a couple of examples and the results the tool will present for each of the use cases it provides.

One type parameter

“Table 3.1” presents all the results obtained after running the tool on “Listing 3.10”.

Listing 3.10: Bounded one type parameter

```
class A{}
class B extends A{}
class C extends B{}
class D extends A{}

class Foo<T extends A>{}

class FooH extends Foo<B>{}
class FooV{
    Foo<D> f = new Foo<>();
}
```

The aperture of 0.5 shows the developer that only half of all the possible types are used in the code base. From “Table 3.1” he can observe that the types used have in common that they are direct decedents of the class `A`. This can show the developer that additional implicit relations between entities may be used.

Perspective	Parameters	All subtypes	Used types	Aperture
single parameters	[T]	[C, D]	[C]	0.5
pair parameters	[]	[]	[]	0

Table 3.2: Synthetic bounded one type parameter

As mentioned in "Section 3.2.1" finding all possible types for a synthetic type parameter bound, can prove to be a difficult taks. "Listing 3.11" and "Table 3.2" presents the results obtained after running the tool on such a class.

Listing 3.11: Synthetic bounded one type parameter

```
class A{}
interface B{}
class C extends A implements B{}
class D extends A implements B{}
class E extends A{}
interface F extends B{}

class Foo<T extends A & B>{}

class FooH extends Foo<C>{}
```

Unbounded types although offer a lot of flexibility they don't offer any information regarding the context in which they should be used. Finding all the subtypes of unbounded type parameter means returning all the classes from all the libraries used in a project, including the Java standard library (i.e. since an unbounded parameter implicitly extends Object and all classes extend Object). "Listing 3.12" and "Table 3.3" presents the results obtained after running the tool on such a class.

Listing 3.12: Synthetic bounded one type parameter

```
class A{}
class B extends A{}
class C extends A{}
class D extends A{}

class Foo<T>{}

class FooH extends Foo<B>{}
class FooV{
    Foo<C> f = new Foo<>();
}
```

Perspective	Parameters	All subtypes	Used types	Aperture
single parameters	[T]	[* extends Object]	[B, C]	0.01
pair parameters	[]	[]	[]	0

Table 3.3: Synthetic bounded one type parameter

Even though the value showed by the aperture doesn't seem to have any meaning, there is still a couple of hints the developer can take away from it. Since the type is unbouded and the aperture is very small, this may mean that the declaring class of the type parameter just adds additional functionality to the type parameter, without making any assumptions regarding the type. This may lead the developer to thinking that this is a class which offers a set of utility methods. Another assumption a developer can make using this is that maybe this is an bug waiting to happen, since both usages of the class `Foo` use as arguments subtypes of `A`. Maybe the type should have been bounded.

Pair of type parameters

When analyzing a class which has multiple type parameters, the first step which must be done is to construct all the permutations without repetition of type parameters.

In "Listing 3.13" is presented a class which has two bounded type parameters. Running our tool on it will offer the results presented in "Table 3.4".

Listing 3.13: Bounded two type parameters

```
class A{}
class B extends A{}
class C extends A{}
class E{}
class F extends E{}
class G extends E{}
class H extends G{}

class Foo<T extends A, K extends E>{}
class FooH extends Foo<B,F>{}
class FooV{
    Foo<C, F> f = new Foo<>();
}
```

From the results presented in "Table 3.4", a developer can understand that parameter `T` takes as argument in the client code all sub-

Perspective	Parameters	All subtypes	Used types	Aperture
single parameters	[T, K]	T=[A,B,C] K=[E,F,G]	T=[B,C] K=[F]	T=0.66 K=0.33
pair parameters	[(T,K)]	[(A,E), (A, F),(A,G), (B,E), (B, F),(B,G), (C,E), (C, F),(C,G)]	[(B,F), (C,F)]	0.22

Table 3.4: Bounded two type parameters

classes of `A` thus having a pretty high aperture. Parameter `K` takes as argument only on parameter in the client code, thus having a lower aperture then parameter `F`. Looking at the type parameters as a pair however can suggest that even though there are 9 possible combinations which can be made based on the subtypes of each type parameter, only two combinations are actually used. Since both combinations use contain the type `F` and use all the subclasses of class `A`, this might suggest that there may exist a relation between the two parameters.

Another example which may appear is when a class has two type parameters which are not bounded. As in the case of a class which has a single unbounded type parameter, this may suggest that the class doesn't need any information about the type and just allows the user to wrap the type parameters in some additional structure which offers a set of features. The actual types which are used as arguments for the type parameters being described in the client code.

A final example which can be also be present is when there is a combination between bounded and unbounded type parameters. "Listing 3.14" and "Table 3.5" presents the results obtained after running the tool on such a class.

Listing 3.14: Two type parameters

```
class A{}
class B extends A{}
class C extends A{}

class Foo<T extends A, K>{}
class FooH extends Foo<B,Integer >{}
class FooV{
    Foo<C, Double> f = new Foo< >();
}
```

As in the previous example "Table 3.4", the developer can observe that `T` takes as arguments all sublclasses of class `A`, thus the 0.66 result

Perspective	Parameters	All subtypes	Used types	Aperture

single parameters	[T, K]	T=[A,B,C]
		K=[* extends Object]
			T=[B,C]
			K=[Integer, Double]
				T=0.66
				K=0.01

pair parameters	[(T,K)]	[(A, Object), ..., (B, Object), ..., (C, Object), ...]	[(B,Integer), (C,Double)]	0.01

Table 3.5: Two type parameters

for the aperture. Since parameter `K` doesn't have any bounds, the used types only suggest that the parameter takes as arguments in the client code `Numeric` classes. Inspecting the results of analyzing the pair of type parameters however suggests that there may be a relation hidden in the implementation of class `Foo` between the subclasses of `A` and the `Numeric` classes.

Chapter 4

Evaluation

In order to verify if family polymorphism is simulated in industry code through generic classes and parametric polymorphism we have decide to analyze a couple of open source projects. In order to detect if the type parameters are correlated, when analyzing the system we only selected the generic classes which have at least two type parameters and the type parameters are bounded.

4.1 Hibernate-ORM

The first analyzed project which which uses generic classes is the Hibernate framework. Even though the project defines a lot of modules, only two of them actually use generic classes.

4.1.1 hibernate-core

When analyzing this module, only a small amount of classes use parametric polymorphism.

One of these classes is `AbstractMultiTableBulkIdStrategyImpl` which defines two bounded type parameters:

1. `TT extends IdTableInfo`
2. `CT extends AbstractMultiTableBulkIdStrategyImpl.PreparationContext`

Because the tool offers two perspectives which can be used when analyzing the generic class (i.e. analyzing the type parameters individually or analyzing the type parameters in pairs), we analyze the type parameter individually first and afterwards as pairs.

Individual parameters

Getting all the types for **1** shows the following:

- `org.hibernate.hql.spi.id.persistent.IdTableInfoImpl`
- `org.hibernate.hql.spi.id.local.IdTableInfoImpl`
- `org.hibernate.hql.spi.id.global.IdTableInfoImpl`
- `org.hibernate.hql.spi.id.IdTableInfo`

Getting all the used types for **1** shows the following:

- `org.hibernate.hql.spi.id.persistent.IdTableInfoImpl`
- `org.hibernate.hql.spi.id.local.IdTableInfoImpl`
- `org.hibernate.hql.spi.id.global.IdTableInfoImpl`

Having these two lists, the aperture is *75.00%*, since all types, except `org.hibernate.hql.spi.id.IdTableInfo` are used. An important note is that the unused type is the parent of the whole hierarchy.

Getting all the types for **2** shows the following:

- `org.hibernate.hql.spi.id.AbstractMultiTableBulkIdStrategyImpl .PreparationContext`
- `org.hibernate.hql.spi.id.global.PreparationContextImpl`
- `org.hibernate.hql.spi.id.persistent.PreparationContextImpl`

Getting all the used types provides the same list as getting all the subtypes, meaning that all the types provided by the hierarchy are used in the client code.

Using the above information, the aperture for **2** is *100.00%*. Even though this result is obtained because all the types are used, this could still be obtained because the parent of the hierarchy is used by the client code.

Pair parameters

Analyzing the parameters individually offered a perspective of the types usages, however analyzing the parameters as pairs can highlight the implicit relations which might exist between the types.

Getting all the usages for the pair **(1,2)** shows the following:

- `(org.hibernate.hql.spi.id.global.IdTableInfoImpl, org.hibernate.hql.spi.id.global.PreparationContextImpl)`
- `(org.hibernate.hql.spi.id.local.IdTableInfoImpl, org.hibernate.hql.spi.id.AbstractMultiTableBulkIdStrategyImpl .PreparationContext)`
- `(org.hibernate.hql.spi.id.persistent.IdTableInfoImpl, org.hibernate.hql.spi.id.persistent.PreparationContextImpl)`

From this listing, it can be observed that only some combinations of all the possible combinations which can be obtained from permutations are used.

The aperture for these pair of types is *41.67%*. This low value might suggest that there is a relation between the types.

The implicit relation might be defined by the package which contains the types. The `global` package defines a relation between types, the `persistent` package defines a relation between types and a more odd association between types is the inner class `PreparationContext`, which is also the parent of the hierarchy, with the `IdTableInfoImpl` from the `local` package. This lack of continuity might suggest a design flaw, the class `PreparationContext` could have been moved to a separate file in the `local` package. This could also suggest that these pair was implemented by a different developer later on in the lifecycle of the project.

The next class which also uses a pair of type parameters is the class `AbstractLobTest`:

1. `B extends AbstractBook`
2. `C extends AbstractCompiledCode`

Individual parameters

Retrieving all the possible types for **1** provides the following list:

- `org.hibernate.test.annotations.lob.AbstractBook`
- `org.hibernate.test.annotations.lob.Book`
- `org.hibernate.test.annotations.lob.VersionedBook`

Retrieving all the used types for **1** returns the following list:

- `org.hibernate.test.annotations.lob.Book`
- `org.hibernate.test.annotations.lob.VersionedBook`

The aperture for **1** is equal to *66.00%*. Even this might not be such a good aperture, the developer can see that in order to extend the existing functionality he has to declare a subtype of `org.hibernate.test.annotations.lob.Book`.

Retrieving all the possible types for **2** provides the following list:

- `org.hibernate.test.annotations.lob.AbstractCompiledCode`
- `org.hibernate.test.annotations.lob.CompiledCode`
- `org.hibernate.test.annotations.lob.VersionedCompiledCode`

Retrieving all the used types for **2** returns the following list:

- `org.hibernate.test.annotations.lob.CompiledCode`
- `org.hibernate.test.annotations.lob.VersionedCompiledCode`

The aperture for **2** has the same value as the aperture for **1**, *66.00%*. The same hints can be observed for this parameter as for parameter **1**.

Pair parameters

Getting all the used types for the pair of parameters `(B extends AbstractBook, C extends AbstractCompiledCode)` return the following list:

- `(org.hibernate.test.annotations.lob.Book, org.hibernate.test.annotations.lob.CompiledCode)`
- `(org.hibernate.test.annotations.lob.VersionedBook, org.hibernate.test.annotations.lob.VersionedCompiledCode)`

From this list, a developer can observe that the parents of the hierarchies are never used throughout the client code. This might be an implicit restriction which must be considered when using the analyzed class. Based on the low aperture, *22.22%*, and the naming convention used when associating the types (e.g. `(VersionedBook, VersionedCompiledCode); (Book, CompiledCode)`), the developer can deduce that there are some guidelines which should be followed when extending the hierarchy.

The last class from this module which defines generic bounded parameters is `AbstractSchemaBasedMultiTenancyTest`, which defines the following type parameters:

1. `T extends MultiTenantConnectionProvider`
2. `C extends ConnectionProvider & Stoppable`

Individual parameters

Getting all the possible types for **1** returns the following list:

- `org.hibernate.engine.jdbc.connections.spi .AbstractDataSourceBasedMultiTenantConnectionProviderImpl`
- `org.hibernate.engine.jdbc.connections.spi .AbstractMultiTenantConnectionProvider`
- `org.hibernate.engine.jdbc.connections.spi .DataSourceBasedMultiTenantConnectionProviderImpl`
- `org.hibernate.engine.jdbc.connections.spi .MultiTenantConnectionProvider`
- `org.hibernate.test.multitenancy.TestingConnectionProvider`
- `org.hibernate.test.multitenancy.schema .SchemaBasedDataSourceMultiTenancyTest .AbstractDataSourceBasedMultiTenantConnectionProviderImpl`
- `org.hibernate.test.multitenancy.schema .SchemaBasedMultiTenancyTest.AbstractMultiTenantConnectionProvider`
- `org.hibernate.userguide.multitenancy .ConfigurableMultiTenantConnectionProvider`

Getting all the used types for **1** returns the following list:

- `org.hibernate.engine.jdbc.connections.spi .AbstractDataSourceBasedMultiTenantConnectionProviderImpl`
- `org.hibernate.engine.jdbc.connections.spi .AbstractMultiTenantConnectionProvider`

Looking at the usages and the type hierarchy, the developer can observe that only the direct children of `MultiTenantConnectionProvider` are used in the client code.

The aperture for **1** is equal to *87.5%*. Even though only two types from the hierarchy are directly used in the client code, they are extended by additional classes which can replace them at run-time.

Getting all the possible types for **2** returns the following list:

- `com.zaxxer.hikari.hibernate.HikariConnectionProvider`
- `org.hibernate.agroal.internal.AgroalConnectionProvider`
- `org.hibernate.c3p0.internal.C3P0ConnectionProvider`
- `org.hibernate.engine.jdbc.connections.internal .DatasourceConnectionProviderImpl`
- `org.hibernate.engine.jdbc.connections.internal .DriverManagerConnectionProviderImpl`
- `org.hibernate.engine.jdbc.connections.internal .UserSuppliedConnectionProviderImpl`
- `org.hibernate.engine.jdbc.connections.spi.ConnectionProvider`
- `...`

Type parameter **2** even though it's a synthetic type, it has a list of 83 possible types which can be used as type arguments.

Getting all the used types for **2** returns the following list:

- `org.hibernate.engine.jdbc.connections.internal .DatasourceConnectionProviderImpl`
- `org.hibernate.engine.jdbc.connections.internal .DriverManagerConnectionProviderImpl`

The aperture for **2** is equal to *7.23%*. Looking at all the possible types which can be used as arguments, it can be observed that most of them are specific implementations of a `ConnectionProvider`. The only actual types used being generic classes which don't define any specific details regarding the connection.

Pair parameters

Analyzing all the used combination of types for `(T extends MultiTenantConnectionProvider, C extends ConnectionProvider & Stoppable)`, only the following two elements are found:

- `(org.hibernate.engine.jdbc.connections.spi .AbstractDataSourceBasedMultiTenantConnectionProviderImpl, org.hibernate.engine.jdbc.connections.internal .DatasourceConnectionProviderImpl)`
- `(org.hibernate.engine.jdbc.connections.spi .AbstractMultiTenantConnectionProvider, org.hibernate.engine.jdbc.connections.internal .DriverManagerConnectionProviderImpl)`

Having only these two connections used and the low aperture, *3.35%*, can suggest that the project doesn't use the actual implementations of the database connections, and only uses the abstract ones for modeling the hierarchy. Another aspect which can be observed is that the classes from the package `org.hibernate.engine.jdbc.connections.spi` have an implicit relation with the classes from `org.hibernate.engine.jdbc .connections.internal`, this also being suggested by the termination of the class names `ConnectionProvider`.

4.1.2 hibernate-jpamodelgen

The class `AttachmentGroupPost` contains two bounded type parameters which can be used to do an analysis:

1. `UserRoleType extends UserRole`
2. `GroupType extends AttachmentGroup`

Individual parameters

Getting all the types for **1** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.UserRole`

Getting all the used types for **1** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.UserRole`

Having these two lists, the aperture is *100.0%*, since the single type defined is used in the client code.

Getting all the types for **2** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroup<GroupType, PostType, UserRoleType>`
- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupInTopic`

Getting all the used types returns the following list:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupInTopic`

Based on these two lists, the aperture for **2** is *50.0%*. Since only the concrete type is used in the client code.

Pair parameters

Even though there are not many possible permutations which can be obtained from all the possible subtypes of the generic parameters, only one combination is used.

Getting all the usages for the pair **(1,2)** shows the following:

- `(org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.UserRole, org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupInTopic)`

From this list, it can be observed that only the child classes are used in the client code. The aperture for these pair of types is *50.0%*.

The relation between the two used types can be observed even from analyzing the types individually, but having them grouped offers a clearer image. An implicit relation which might exists is suggested by the fact that the client code only uses fully defined classes, the generic class `AttachmentGroup` not being used. The initial suggestion that only child classes are used in the client code being invalidated since the class `UserRole` is the parent of the hierarchy and it is still used in the client code.

Another generic class from this module is the class `AttachmentGroup` which defines three bounded type parameters which can be used to do an analysis:

1. `GroupType extends AttachmentGroup`
2. `PostType extends AttachmentGroupPost<UserRoleType, GroupType>`
3. `UserRoleType extends UserRole`

Individual parameters

Getting all the types for **1** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroup<GroupType, PostType, UserRoleType>`
- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupInTopic`

Getting all the used types for **1** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupInTopic`

As in the previous example, **1** receives as argument only the child class, and not the parent, in the client code. The aperture for this parameter being *50.00%*.

Getting all the types for **2** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupPost<UserRoleType, GroupType>`
- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupPostInTopic`
- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupPostInTopic`

Same as in the case of the first parameter, only the child class is used as a type argument in the child code. The aperture for the second parameter having the same value as the aperture for the first, *50.0%*.

Getting all the types for **3** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.UserRole`

Getting all the used types for **3** shows the following:

- `org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.UserRole`

Having these two lists, the aperture is *100.0%*, since the single type defined is used in the client code.

Pair parameters

Because the class has more than 2 type parameters, we can group the parameters in pairs and analyze them individually. The pairs are the following:

1. `(GroupType, PostType)`
2. `(PostType, UserRoleType)`
3. `(GroupType, UserRoleType)`

Getting all the usages for the pair **1** shows the following:

- `(org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupInTopic, org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupPostInTopic)`

From this list, it can be observed that only the child classes are used in the client code. The aperture for these pair of types is *25.0%*, because there can be a total of 4 combinations between the types, and only one is used.

Getting all the usages for the pair **2** shows the following:

- `(org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupPostInTopic, org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.UserRole)`

From this list, it can be observed that only the classes which don't have any type parameters are used. The aperture for this being *50.0%* since there can be only two possible combinations of types, and one of them is actually used in the client code.

Getting all the usages for the pair **3** shows the following:

- `(org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.AttachmentGroupInTopic, org.hibernate.jpamodelgen.test.mappedsuperclass .typedmappedsuperclass.UserRole)`

The same observations can be made here as for the pair **2**. The aperture is the same, *50.0%*.

The implicit relation which can be observed is that the client code uses classes which don't define any additional type parameters. This being the case in all three pairs.

4.2 Elasticsearch

Like Hibernate the Elasticsearch framework declares multiple modules. Even though generic classes with multiple bounded type parameters are available in multiple modules, the following classes from *elasticsearch-core* present a obvious usage of family polymorphism.

The class `org.elasticsearch.action.GenericAction` contains two bounded type parameters which can be used to do an analysis:

1. `Request extends ActionRequest`
2. `Response extends ActionResponse`

Individual parameters

Getting all the types for **1** shows the following:

- `org.elasticsearch.action.ActionModuleTests .testPluginCanRegisterAction().FakeRequest`
- `org.elasticsearch.action.ActionRequest`
- `org.elasticsearch.action.admin.cluster .allocation.ClusterAllocationExplainRequest`
- `org.elasticsearch.action.admin.cluster .health.ClusterHealthRequest`
- `org.elasticsearch.action.admin.cluster .node.hotthreads.NodesHotThreadsRequest`

- `org.elasticsearch.action.admin.cluster`
  `.node.info.NodesInfoRequest`
- `...`

There are a total of 133 entities in the `ActionRequest` hierarchy.
Getting all the used types for **1** shows the following:

- `org.elasticsearch.action.ActionModuleTests`
  `.testPluginCanRegisterAction().FakeRequest`
- `org.elasticsearch.action.ActionRequest`
- `org.elasticsearch.action.admin.cluster`
  `.allocation.ClusterAllocationExplainRequest`
- `org.elasticsearch.action.admin.cluster`
  `.health.ClusterHealthRequest`
- `org.elasticsearch.action.admin.cluster`
  `.node.hotthreads.NodesHotThreadsRequest`
- `org.elasticsearch.action.admin.cluster`
  `.node.info.NodesInfoRequest`
- `...`

There are a total of 91 entities of the `ActionRequest` hierarchy which are used in the client code.

Although it is obvious that not all possible types are used in the client code, it might be quite hard to find which is not used. For this, the feature which returns all the unused types (i.e. which is a difference between the all possible types list and all used types list) comes in handy. Getting all the unused types returns the following list:

- `org.elasticsearch.action.admin.cluster.snapshots.status`
  `.TransportNodesSnapshotsStatus.Request`
- `org.elasticsearch.action.support.replication`
  `.TransportReplicationActionTests.Request`
- `org.elasticsearch.action.support.replication`
  `.ReplicationOperationTests.Request`
- `org.elasticsearch.action.support.replication`
  `.BasicReplicationRequest`

- `org.elasticsearch.action.support.replication .ReplicatedWriteRequest`
- `org.elasticsearch.action.fieldcaps .FieldCapabilitiesIndexRequest`
- `org.elasticsearch.index.reindex.AbstractBulkByScrollRequest`
- `org.elasticsearch.index.shard.PrimaryReplicaSyncer.ResyncRequest`
- `...`

Since this list represents the difference between all types and used types, there are a total of 42 types in this list.

A common pattern which can be observed by looking at all the unused types is that most of them are inner classes or are classes which themselves define additional generic parameters or are abstract classes. A quick analysis of the hierarchy points out the following information of the unused types:

- classes aren't completely defined (i.e. which are abstract or define additional parameters) are not used in the client code
- the types which are completely defined are not used in the client code because they are not direct descendants of **1** upper bound (`ActionRequest`). The only class which does not respect this convention is `org.elasticsearch.action.admin.cluster.node .liveness.LivenessRequest`. The interesting aspect about `LivenessRequest` it does not implement or define additional logic. Looking in the way it is used it reveals some strange behavior; even though a method must receive a `LivenessRequest` as a parameter, it completely ignores it in its implementation

Because the client code uses even the upperbound type, the aperture for **1** is *100.0%*.

Getting all the types for **2** shows the following:

- `org.elasticsearch.action.ActionResponse`
- `org.elasticsearch.action.DocWriteResponse`
- `org.elasticsearch.action.admin.cluster.allocation .ClusterAllocationExplainResponse`

- `org.elasticsearch.action.admin.cluster.health .ClusterHealthResponse`
- `org.elasticsearch.action.admin.cluster.node .hotthreads.NodesHotThreadsResponse`
- `...`

A total of 123 entities being present in the `ActionResponse` hierarchy.
Getting all the used types for **2** shows the following:

- `org.elasticsearch.action.ActionResponse`
- `org.elasticsearch.action.admin.cluster .allocation.ClusterAllocationExplainResponse`
- `org.elasticsearch.action.admin.cluster .health.ClusterHealthResponse`
- `org.elasticsearch.action.admin.cluster .node.hotthreads.NodesHotThreadsResponse`
- `org.elasticsearch.action.admin.cluster .node.info.NodesInfoResponse`
- `...`

The lists containing 86 entities from the hierarchy which are used in the client code.

Getting all the unused types provides the following list which contains the difference between all the possible types and all the used types, thus resulting in a list with 37 entities:

- `org.elasticsearch.action.DocWriteResponse`
- `org.elasticsearch.action.admin.cluster .node.liveness.LivenessResponse`
- `org.elasticsearch.action.admin.cluster .node.tasks.TaskManagerTestCase.NodesResponse`
- `org.elasticsearch.action.bulk.BulkShardResponse`
- `org.elasticsearch.action.get.MultiGetShardResponse`

Besides the deduction we came up after analyzing the unused types of `ActionRequest` there is an additional pattern which can be observed after analyzing the hierarchies and their usages in isolation. There are a couple of classes which even though they are not abstract they are still not used in the client code, however they are extend by other classes which are used in the client code. One such example is `org.elasticsearch.action.support.tasks.BaseTasksResponse` which isn't an abstract class, but still it is never actually used. However the classes which extend `BaseTasksResponse` are used in the client code. This might suggest that the class `BaseTasksResponse` is strictly used for sharing code among child classes and could have been made abstract, to respect the convention defined by the majority of other classes.

Because the parent of the hierarchy is used in the client code the aperture for **2** is *100.0%*.

Pair parameters

Getting all the used types for the pair of parameters `(Request extends ActionRequest, Response extends ActionResponse)` returns the following list:

- `(org.elasticsearch.action.get.MultiGetRequest, org.elasticsearch.action.get.MultiGetResponse)`
- `(org.elasticsearch.action.admin.cluster.node .info.NodesInfoRequest, org.elasticsearch.action.admin.cluster.node .info.NodesInfoResponse)`
- `(org.elasticsearch.action.admin.cluster.node .tasks.get.GetTaskRequest, org.elasticsearch.action.admin.cluster.node .tasks.get.GetTaskResponse)`
- `(org.elasticsearch.action.delete.DeleteRequest, org.elasticsearch.action.delete.DeleteResponse)`
- `(org.elasticsearch.action.admin.cluster .stats.ClusterStatsRequest, org.elasticsearch.action.admin.cluster .stats.ClusterStatsResponse)`
- `...`

As it can be observed by analyzing the usages all request type objects have an associated response type object. This suggests the presence of family polymorphism. When extending the `GenericAction` the developer must consider this implicit constraint before adding additional functionality. This correlated usage between types is even more obvious when seeing the very low value for the aperture, *2.13%*.

Even though the `class org.elasticsearch.action.Action<Request, Response, RequestBuilder>` is a subclass of `org.elasticsearch.action.GenericAction<Request, Response>`, thus it must preserve the correlation between the `Request` and the `Response`, it adds an additional type parameter.

1. `RequestBuilder extends ActionRequestBuilder<Request, Response, RequestBuilder>`

Because the first two generic parameters have been analyzed in the previous section, we will only focus in this section in analyzing the last parameter, and the pairs which use it.

Individual parameter

Getting all the types for **1** returns the following list:

- `org.elasticsearch.action.ActionRequestBuilder<Request extends ActionRequest, Response extends ActionResponse, RequestBuilder extends ActionRequestBuilder<Request, Response, RequestBuilder»`
- `org.elasticsearch.action.admin.cluster .allocation.ClusterAllocationExplainRequestBuilder`
- `org.elasticsearch.action.admin.cluster .health.ClusterHealthRequestBuilder`
- `org.elasticsearch.action.admin.cluster .node.tasks.get.GetTaskRequestBuilder`
- `...`

There are 106 classes which extend `RequestBuilder`.

From this list of 106 types only 93 types are used throughout the code base. Because the parent of the hierarchy is used directly in the client code, the aperture for this type parameter is *100.0%*.

Pair parameters

Because the generic class has 3 type parameters, there are three combinations of pairs which are obtained

1. `(Request, Response)`
2. `(Response, RequestBuilder)`
3. `(Request, RequestBuilder)`

Because the first pair was analyzed in the previous section, we will only analyze the last two pairs and see if the correlation between types is persisted.

Getting all the usages for **2** the following list is returned:

- `(org.elasticsearch.action.ActionResponse, org.elasticsearch.action.ActionRequestBuilder<Request extends ActionRequest, Response extends ActionResponse, RequestBuilder extends ActionRequestBuilder<Request, Response, RequestBuilder»)`
- `(org.elasticsearch.action.admin.cluster.health .ClusterHealthResponse, org.elasticsearch.action.admin.cluster.health .ClusterHealthRequestBuilder)`
- `(org.elasticsearch.action.admin.cluster.node .info.NodesInfoResponse, org.elasticsearch.action.admin.cluster. node.info.NodesInfoRequestBuilder)`
- `...`

Looking at the usages we can observe that there is a correlation between the `Response` and the `RequestBuilder`. Because the aperture considers all the subtypes of a type when getting all the used types, the aperture for this pair is *100%*. This high aperture is caused because the client code uses directly the upper bound of the hierarchy. But considering only the actual types used in the code base, there are 83 pairs of combination used and a 13038 which could be used. Even though some place allows all possible combinations to pass through, the actual client code doesn't break the correlation between types.

Getting all the usages for **3** the following list is returned:

- `(org.elasticsearch.action.ActionRequest, org.elasticsearch.action.ActionRequestBuilder<Request extends ActionRequest, Response extends ActionResponse, RequestBuilder extends ActionRequestBuilder<Request, Response, RequestBuilder»)`
- `(org.elasticsearch.action.admin.cluster.health .ClusterHealthRequest, org.elasticsearch.action.admin.cluster.health .ClusterHealthRequestBuilder)`
- `(org.elasticsearch.action.admin.cluster.node .info.NodesInfoRequest, org.elasticsearch.action.admin.cluster.node .info.NodesInfoRequestBuilder)`
- `...`

Observing the name of the types used in the client code, there is obviously a correlation between them. Because the upper bound is used directly in the client code, like in the previous case, the aperture is *100%*. But still, the client code preservers the rules when correlating types, even though there are no static constraints which enforce these.

Looking at the usages of the three pairs, there is an obvious correlation between the three type parameters. When extending the `Action` one must define an additional `Request`, `Response` and `RequestBuilder`. Doing this the developer keeps the implicit relations between the existing objects and also avoids introducing bugs when adding new features.

The class `org.elasticsearch.action.support.broadcast .BroadcastOperationRequestBuilder<Request, Response, RequestBuilder>` contains 3 type parameters:

- `Request extends BroadcastRequest<Request>`
- `Response extends BroadcastResponse`
- `RequestBuilder extends BroadcastOperationRequestBuilder<Request, Response, RequestBuilder>`

Because the following thesis focuses on finding correlations between types, in the following section we will only analyze the pairs which can be obtained from the type parameters.

Pair parameters

Based on the 3 type parameters, the following pairs can be obtained:

1. `(Request, Response)`
2. `(Response, RequestBuilder)`
3. `(Request, RequestBuilder)`

A first look at these three type parameters shows that these are the same type parameters names as declared by the `Action` class.

Retrieving the usages for **1** returns the following list:

-
  ```
  (org.elasticsearch.action.admin.indices.cache
  .clear.ClearIndicesCacheRequest,
  org.elasticsearch.action.admin.indices.cache
  .clear.ClearIndicesCacheResponse)
  ```

-
  ```
  (org.elasticsearch.action.admin.indices.flush.FlushRequest,
  org.elasticsearch.action.admin.indices.flush.FlushResponse)
  ```

-
  ```
  (org.elasticsearch.action.admin.indices.recovery.RecoveryRequest,
  org.elasticsearch.action.admin.indices.recovery.RecoveryResponse)
  ```
- `...`

There are a total of 10 combinations used throughout the code base. Looking at all the usages of **1** suggests that there is a correlation between the two type parameters. Another clue which again suggests a correlation between the type parameters is the low aperture, *5.95%*.

Analyzing the remaining pair of types **2** and **3**, again suggests that there is a correlation between types. This correlation being suggested by the naming conventions used (i.e. types used in correlation have the same prefix) and also by the low aperture, *7.58%* for **2** and *6.49%* for **3**.

Chapter 5

Conclusions

Analyzing the two projects it could be observed that the generic classes which define multiple bounded type parameters have a tendency in using family polymorphism. Even though family polymorphism is not available in Java, in both projects the developers concluded that the only way to model the relation between objects is to use parametric polymorphism. The work-around for "constraining" the developers into using valid combination of types was to use naming conventions.

Detecting the usage of family polymorphism can be a hard task since there is not built-in structure to mark this. The tool presented in the previous chapters enables the developers to detect this correlation between types and gain some quick insight about the way a generic class is used throughout the code base. Because this knowledge is available without any need for actually inspecting the implementation the time needed for understanding how a combination of types is used is greatly reduced.

The various features implemented in the tool allow the developers to chose the granularity of the analysis. The implemented metric also allows the developer to filter the classes which might implement family polymorphism through various work-arounds.

5.1 Future work

In order to have a more accurate depiction of the analyzed system a more in depth analysis of it might be required. At the moment when getting all the used types the tool only considers the types used in subtyping and the types used when declaring variables.

Another way of passing type arguments to type parameters is when declaring a method. The generic type can be used both as a return type or as an parameter type, and in both cases type arguments can be used. This is also a valid usage which could be considered when analyzing a generic class.

A second usage of generic classes is when creating anonymous classes. Because the tool only considers the types of attributes or variables, anonymous classes are completely ignored. Since some projects tend to use a more functional approach in the implementation, anonymous classes are heavily used. Considering these classes will increase the accuracy of the tool even more.

List of figures

List of tables

Listings

Bibliography

[1] T. Chaikalis N. Ygeionomakis E. Ligu, A. Chatzigeorgiou. *Identification of Refused Bequest Code Smells.* IEEE International Conference on Software Maintenance, 2013.

[2] E. Ernst. *Family polymorphism.* In Proceedings of the European Conference on Object-Oriented Programming, 2001.

[3] S. Ducasse G. Arevalo and O. Nierstrasz. *Discovering unanticipated dependency schemas in class hierarchies.* InProceedings of CSMR. IEEE Computer Society, 2005.

[4] S. Gordillo O. Nierstrasz G. Arvalo, S. Ducasse. *Generating a Catalog of Unanticipated Schemas in Class Hierarchies using Formal Concept Analysis.* 2013.

[5] S. Osama H. Abdeen, H. Sahraoui. *How We Design Interfaces, and How To Assess It.* 29th IEEE International Conference on Software Maintenance, 2013.

[6] Z. Porkolb I. Zlyomi. *A generative approach for family polymorphism in C++.* Proceedings of ICAI 2004, 2004.

[7] G. Bavota R. Oliveto M. Di Penta A. De Lucia D. Poshyvanyk M. Tufano, F. Palomba. *When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away).* IEEE Transactions on Software Engineering, 2017.

[8] R. Marinescu. *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws.* 20th IEEE International Conference on Software Maintenance, 2004.

[9] P. F. Mihancea. *Towards a Client Driven Characterization of Class Hierarchies.* Politehnica University of Timisoara, 2006.

[10] P. F. Mihancea. *A Novel Client-Driven Perspective on Class Hierarchy Understanding and Quality Assessment.* Politehnica University of Timisoara, 2009.

[11] P. F. Mihancea and C. Marinescu. *Defects and Polymorphism: Is there any Correlation?* 17th European Conference on Software Main- tenance and Reengineering, 2013.

[12] P.F. Mihancea. *Patrools: Visualizing the Polymorphic Usage of Class Hierarchies.* Politehnica University of Timisoara, 2010.

[13] R. Marinescu P.F. Mihancea. *Discovering Comprehension Pitfalls in Class Hierarchies.* European Conference on Software Maintenance and Reengineering, 2009.

[14] H. Sahraoui S. Denier. *Understanding the Use of Inheritance with Visual Patterns.* Empirical Software Engineering and Measurement, 2009.

[15] Y. Gueheneuc S. Denier. *Mendel: A Model, Metrics, and Rules to Understand Class Hierarchies.* IEEE International Conference on Program Comprehension, 2008.