Towards the Detection of Correlated Type Usages in Generic Code

Alexandru Stana
Faculty of Automation and Computing
Politehnica University Timisoara
Timisoara, Timis
alexandrustana2@gmail.com

***Abstract*—Understanding an object-oriented system can prove to be quite difficult. The increased difficulty of this task is not only introduced by the polymorphic interaction between classes, but also from the implicit relations which can exist between them. Family polymorphism represents a situation in which such implicit relation may appear. Because Java does not have a mechanism to statically enforce the correct correlated usage of types, understanding such cases is difficult. To address this issue, we present a tool that can be used to detect the intention of using family polymorphism in Java generic classes.**

***Index Terms*—polymorphism, static analysis, class hierarchies, parametric polymophism**

I. Introduction

Subtype polymorphism greatly increases the flexibility of a system and simplifies its extension. However, it also increases the difficulty of understanding that system. Because of this feature (i.e. polymorphism) 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 [1]. Let us see the example in the next listing.

Listing 1. Modeling Graphs

```
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 {...}

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

class OnOffEdge extends AbstractEdge{
    boolean enabled;
}

class Graph<N extends AbstractNode,
            E extends AbstractEdge> {
    List<N> nodes;
    List<E> edges;
}
```

While not very obvious, the intention in this code is to use OnOffNodes only in conjunction with OnOffEdges and Nodes only in combination with Edges. Combining, for instance, an OnOffNode with an Edge would result is a cast exception.

Unfortunately, Java does not offer a mechanism which statically enforces such relations between classes/types. However, the intention of using family polymorphism can be observed by analyzing the usages of types in the client code that can reveal such restrictions.

In this paper, we present a tool dedicated to detect the intention of using family polymorphism by observing the correlated usage of types in Java generic classes.

II. Solution

As it could be observed in the previous section, in order to gain a better understanding of how a certain class should be used an investigation of the actual usages of that class might provide some valuable insight.

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 generating Meta-Models used for modeling and storing this information.

A. Meta-Model

In order to allow the user to gain insight about the different generic classes available throughout the code base and their usages, we are using the Meta-Models generated with XCore and for each meta-model we provide a set of features which

offer a better understanding of the system. Each of these meta-models are mapped to an internal representation provided by Eclipse JDT (i.e. a class, a type parameter and a java project).

Because manually finding generic classes in a project can take quite a while, we have decided to map the Java project to an `MProject`. Having this mapping the developer will now be able to request the tool to return all the generic classes from a project. Besides returning all the generic classes, using the `MProject` the developer also has the possibility to filter the generic classes based on the number of generic parameters or filter the generic classes depending if the type parameters have upper bounds or not.

When getting all the generic classes from a Java project, before returning them to the developer we map them to a meta-model named `MClass`. Because we are interested in analyzing the correlations between generic parameters, `MClass` allows the developer to retrieve a list containing the generic parameters of the targeted class. Depending on the number of parameters, the developer can choose to retrieve the generic parameters individually, or as pairs.

Using the previous example in which we implemented the `Graph` hierarchy if we choose to get all the generic parameters of class `Graph` we obtain the following list, `[N,E]`. However if we want to get all the pair of generic parameters we obtain the following list, `[(N,E)]`.

As with the previous entities, before returning the generic parameters, depending on the option chosen by the developer, we map them to a `MParameter` or to an `MParameterPair`. Using these meta-models the developer can choose to see the all the type used as arguments for the generic parameters, he can retrieve all the types which can be used as arguments for the generic parameter (i.e. the list of all possible types is obtained by getting all the subclasses of the generic parameter upper bound) or he can request a ratio between the two.

B. Aperture & Aperture Coverage

When analyzing the type parameters, there are two possible aspects which must be considered.

Based on the hierarchy defined by the upper bound of a type parameter(e.g `AbstractNode` is the upper bound for `N`), if the type parameter is unbounded then it implicitly extends `Object`, what are all the possible types which could be used as arguments for the type parameter. We have decided to name this the **Aperture** of a type parameter, since it represents all the possible types which can be used as type arguments.

The apertures for the type parameters for class `Graph` are the following:

- `N = [AbstractNode, Node, OnOffNode]`
- `E = [AbstractEdge, Edge, OnOffEdge]`
- `(N,E) = [(AbstractNode, AbstractEdge), (AbstractNode, Edge), (AbstractNode, OnOffEdge), (Node, AbstractEdge), ...]`

As it can be observed, the aperture for the pair of type parameters is the permutation without repetition of all the possible types.

Because the aperture doesn't show the developer if there are any correlations between the type parameters, the developed tool offers the developer a way of retrieving all the usages of a certain class. By analyzing the usages of the generic class the developer can observe if there are any restrictions when using certain types as arguments. The list of all the used types is named **Aperture coverage**.

Listing 2. Graph usages

```
class ClassicGraph extends Graph<Node, Edge> {...}
class OnOffGraph extends Graph<OnOffNode,
                               OnOffEdge> {...}
```

Using "Listing 1" and "Listing 2" the aperture coverage for the type parameters is the following:

- `N = [Node, OnOffNode]`
- `E = [Edge, OnOffEdge]`
- `(N,E) = [(Node, Edge), (OnOffNode, OnOffEdge)]`

By looking at the aperture coverage for the pair `(N,E)` the developer can observe, without looking at the implementation, that there is a certain pattern which is followed when extending the class `Graph`.

Using these two apertures we can now calculate the overall aperture for a generic parameter. This is represented by the ratio between the cardinality of the aperture and the cardinality of the aperture coverage.

$$OverallAperture = \frac{CoverageAperture}{Aperture}$$

Using the previously obtained results we would obtain the following overall apertures:

- `N = 2/3 = 0.66`
- `E = 2/3 = 0.66`
- `(N,E) = 2/9 = 0.22`

C. Analyzing the results

Using the results obtained from analyzing the `Graph` class, a developer can observe that there are some interesting patterns which emerge.

The high overall aperture for the individual parameters, `N` and `E`, suggests that almost all the type of the hierarchy are used. Looking at the aperture and the aperture coverage reveals that the only type which is not used is the parent of the hierarchy. This can suggest that in order to extend the `Graph` class the developer should extend the upper bound of each type parameter, and not use the upper bound directly.

On the other side, the low overall aperture for the pair `(N,E)` suggests that there may be a correlation between the types. Looking at the aperture coverage, the developer can see that the client only pairs types which have the same name prefix (i.e. "OnOff" in the `OnOffGraph` and nothing in the `ClassicGraph`). Having this hint, the developer now knows that in order to extend the class `Graph` he must create 2 new additional types, which should have the same name prefix (in order to remain consistent with the rest of the hierarchy) and are part of the `AbstractNode` and `AbstractEdge` hierarchy.

D. Implementation

To get a more complete understanding of the methods available in each meta-model and the relations between meta-models, "Figure 1" shows the class diagram of the entire meta-model used for mapping the entities obtained from the AST.

III. Evaluation

In order to obtain some industry insights regarding the usages of generic types in a code base, we have decided to analyze the Hiberante-ORM repository. Analyzing the **hibernate-core** has shown that there is a set of classes which use generic parameter. In the following section we will only analyze one of them.

The class *AbstractMultiTableBulkIdStrategyImpl* contains two bounded type parameters which can be analyzed.

The first approach when analyzing the parameters was to inspect them in isolation. Getting all the possible types for the first parameter shows that there are 4 possible types which could be used:

- 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 the first type parameter 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 for the first parameter 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 upper bound of the type parameter.

Getting all the types for the second parameter 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 available in the hierarchy are used in the client code.

Using the above information, the aperture for the second parameter is **100.00%**.

Analyzing the parameters individually offered a perspective of the types usages, however analyzing the parameters as pairs can highlight if there are correlated usages between the types.

Getting all the usages for the pair the type parameters returns the following list.

- (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 the permutations are used.

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

The correlation is suggested by the way in which types are grouped in packages. The **global** package contains two types which are used in the same class, and also the **persistent** package. A odd correlation between types is between the inner class **PreparationContext**, which is also the parent of the hierarchy, and the **IdTableInfoImpl** class from the **local** package. This lack of continuity might suggest a couple of things. This is a design flaw, the class **PreparationContext** could have been moved to a separate file in the local package. The correlation between these two types was created by a developer which didn't knew about the existing two.

IV. Related Work

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 [5], [7], [9], [10], [12].

In [5] 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 such as PCR(Pure Code Reuse), which shows the developer the purpose of a class hierarchy(i.e. if it is intended for code reuse), and also some metrics which offer the developer an overview of the uniformity of the method calls from a class hierarchy(e.g. Total Uniformity, Partial Uniformity and Total Non-Uniformity). Using the previously mentioned concepts the developer will now be able to have 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 on during software understanding activities. The authors in [11] 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 of the misunderstanding and it is constructed by the symptoms which describe the pitfall together with a relational operator. Using these patterns (i.e. Partial Typing, Uneven Service Behavior and Premature Service) the developer can avoid falling victim to different misleads.

Using a graphical representation of a system can also help the developer when he is trying to understand a software system. The authors in [6], [8] offer such representations, each of them tackling a different aspect.

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

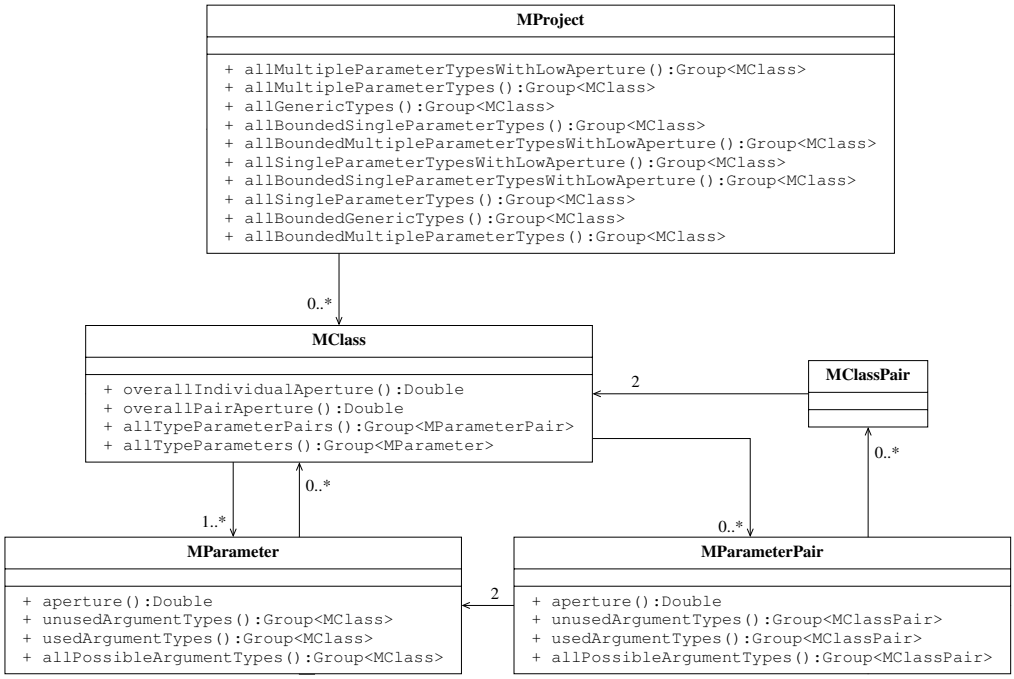

Fig. 1. Application architecture

tackled in [2], [9]. 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" [2] (i.e. attributes accesses and method calls are elements and method calls which propagate through the class hierarchy through as properties). Having this the developer has additional information of the dependencies between classes thus enabling him to extend, maintain and see if the system is well designed or if there are flaws which must be fixed.

To the best of our understanding, no of these previous approaches addresses the problem of detecting the simulation of family polymorphism in Java programs.

V. Conclusions

This paper presented a way of detecting the proper usage of certain classes. Even though there isn't a mechanism for defining families of classes there are a set of work-arounds which try to simulate this. This custom solution however comes at a cost. There isn't any way which enforces the client code to preserve the contracts which define the relations between classes. After analyzing the **Hibernate** repository it was observed that there were a couple of patterns which dictated the correct usage of types by the client code. The tool and metric described in this article aims in aiding the developers which are trying to understand the proper usage of classes. The tool providing all the information regarding the actual usages, the possible usages, and different layers of granularity when targeting an entity which is analyzed.

References

[1] E. Ernst, Family Polymorphism, Proceedings of the 15th European Conference on Object-Oriented Programming, 2001.
[2] G. Arevalo, S. Ducasse, and O. Nierstrasz, Discovering unanticipated dependency schemas in class hierarchies. InProceedings of CSMR. IEEE Computer Society, 2005.
[3] P. F. Mihancea and C. Marinescu, Changes, Defects and Polymorphism: is there any Correlation? 17th European Conference on Software Maintenance and Reengineering, 2013.
[4] P. F. Mihancea, A Novel Client-Driven Perspective on Class Hierarchy Understanding and Quality Assessment, Politehnica University of Timisoara, 2009.
[5] P. F. Mihancea, Towards a Client Driven Characterization of Class Hierarchies, Politehnica University of Timisoara, 2006.
[6] S. Denier, H. Sahraoui, Understanding the Use of Inheritance with Visual Patterns, Empirical Software Engineering and Measurement, 2009.
[7] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, N. Ygeionomakis, Identification of Refused Bequest Code Smells, IEEE International Conference on Software Maintenance, 2013
[8] P.F. Mihancea, Patrools: Visualizing the Polymorphic Usage of Class Hierarchies, Politehnica University of Timisoara, 2010
[9] G. Arvalo, S. Ducasse, S. Gordillo, O. Nierstrasz, Generating a Catalog of Unanticipated Schemas in Class Hierarchies using Formal Concept Analysis, 2010
[10] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away), IEEE Transactions on Software Engineering, 2017.
[11] P.F. Mihancea, R. Marinescu, Discovering Comprehension Pitfalls in Class Hierarchies, European Conference on Software Maintenance and Reengineering, 2009.
[12] S. Denier and Y. Gueheneuc, Mendel: A Model, Metrics, and Rules to Understand Class Hierarchies, IEEE International Conference on Program Comprehension, 2008.