# Hypermodelling Live
# OLAP for Code Clone Recommendation

Tim FREY and Veit KÖPPEN
*Otto-von-Guericke-University Magdeburg, Germany*

**Abstract.** Code bases contain often millions lines of code. Code recommendation systems ease programming by proposing developers mined and extracted use cases of a code base. Currently, recommender systems are based on hardcoded sets what makes it complicate to adapt them. Another research area is adaptable live detection of code clones. We advance clone detection and code recommender systems by presenting utilization of our Hypermodelling approach to realize an alternative technique. This approach uses Data Warehousing technology that scales for big data and allows for flexible and adaptable queries of source code. We present the generic idea to advance recommendation and clone detection based on queries and to evaluate our application with industry source code. Consequently, recommender systems and clone detection can be customized with flexible queries via Hypermodelling. This enables further research about more complex clone detection and context sensitive code recommendation.

**Keywords.** Data warehousing, software engineering, hypermodelling

## Introduction

Code recommender systems advance integrated development environments. They are based on the idea to extract and mine information from code bases to generate recommendations. Recommendation data contains information, which method calls occur commonly together or which methods of a super class get overwritten [1]. This data is compared to current coding of a developer and proposals are offered. The extraction and mining process limits recommendation to be easily adjusted to specific requirements. For instance, it is desirable to have recommendation information available for diverse APIs and also for an own project. Furthermore, project requirements often differ. In one project, it is required that the recommendation code base just comes out of a specific project and in another setting all available code should be used for recommendation. However, this type of flexible recommendation is currently not available. One main reason for this may be the immense size of modern code bases, resulting in difficulties to adjust the extraction process: It is necessary to scan the code base for different recommendation information every time.

Another challenge is the detection of code clones [2, 3]. Thereby, code bases are scanned for duplicates. One main challenge is the different type of code bases and clone detection methods. Sometimes, a certain package should be excluded because replicas are allowed. Furthermore, clones may be exact duplicates of a code fragment or similar pieces of code. Hence, clone detection faces the challenge to provide an easy adjustable infrastructure that allows detecting different kinds of clones and different code base configurations.[1]

Altogether, recommender systems and clone detection face the challenge to provide an adaptable infrastructure for large code bases that allow by flexible means to detect code clones and recommendations. In order to overcome these current limitations, we propose to use our Hypermodelling approach [4, 5, 6] for flexible code recommendation and clone detection. Through this approach both techniques can be covered with Data Warehouse (DW) and Online Analytical processing (OLAP) technology [7] that scales well for large data sets and is easily adjustable for multi-dimensional queries. Therefore, our contribution is to describe, how the Hypermodelling approach can be used to advance recommender systems and clone detection at the same time. We also provide an evaluation, in which we demonstrate the approach on a real world source code excerpt.

The remainder of the paper is structured as follows: First, we recap important facts about our Hypermodelling approach. Next, we describe the general approach, how Hypermodelling can be used to detect clones and recommend code fragments. Afterwards, we describe the approach with a concrete example. In Section 4, we refer to related work and point out differences.

## 1. Hypermodelling

Hypermodelling is the idea to combine program analysis and DW. A more detailed description of the Hypermodelling approach is available in [4].[2] In [5, 6] different reporting possibilities of this approach are presented.

DW systems are an integrative component in business computing [7].[3] They are used to assemble data of different sources together. The integrated data are arranged into multi-dimensional data structures, i.e., data cubes, which serve as base for queries [4]. Queries can be used to aggregate different measures and dimensions (within their hierarchies) that occur in the data. For instance, sales for an employee can be computed for a given time period. Thereby, this query aggregates the region, the products (sales) and the time in relation to financial indicators. Likewise, hierarchies can be abstracted. For instance, the region can be split into continents, countries, counties, as well as cities and the aggregates are associated with the distinct sales for those. This can be done for other hierarchies, e.g., customer group, year, or department. Generally, the idea is that different aggregations enable detailed investigations. With Hypermodelling, we introduce the idea that programming elements, like annotations or classes, are similar to data that are used within a DW. For instance, classes are defined within a package hierarchy. Annotations are associated with classes and their members. They are also defined in their own package. This is like the association of a salesman to a region, time period, and revenues. Hierarchies in code are similar to hierarchies of region or time. All together, we load source code into a DW and realize associations of classes, their inheritance, packages, and annotations as a DW cube and execute queries on it. For this paper, we combine cubes that are used in [4, 5, 6]. We present the results of exemplarily queries on our aggregated cube as reports in this paper, see Section 3.

[1]Note, clone detection methods can be configured and live detection is possible [3]. However, it still is a challenge. Thus, we present alternative approach as complementary subsidiary to code recommendation.
[2]http://hypermodelling.com
[3]Note, there are also open source Data Warehouse solutions available. For instance: http://pentaho.com.

## 2. Clone Recommendation with Hypermodelling

First, we describe the overall idea to use OLAP queries for code analysis. Afterwards, we refer to recommendation and clone detection.

### *2.1. Overall Idea*

We propose a query based approach to advance recommendation and clone detection. We use OLAP queries and our data warehouse approach, out of the size of modern code bases. Additionally, DW technology has many best practices and tuning methods at hand. Queries allow flexibility, adaptability, and DW technology scales for big data. Thus, our live query approach should help to avoid clones and give recommendations at the same time.

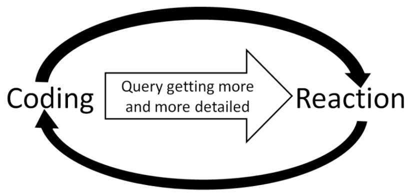


**Figure 1.** Query refinement throughout the coding process

A developer encodes functionality (coding) and in the background, her written code is used to execute an OLAP query against the Code-DW that contains coded structure and elements. For instance, such code can be parameters of methods. Then, the query result gets presented to the developer. This may be in the form of recommendation or in a notification of a code clone. Hence, this query result influences ongoing coding process. From the result, the programmer goes on and encodes more functionality. With this additional functionality, again a query can be executed that contains more information than the first. Every time, as more and more code exists, the query is more and more detailed. If the programmer finishes her current coding, the query is quite detailed with all code fragments that are belonging to a method. If another method shares enough similarities, the corresponding code is presented to the developer and she has to decide if a code clone exists.

Currently, recommendation is not done live and neither is based on a query approach. Even more, code recommenders are inflexible and do not support slicing and dicing data for specific needs. This is often likewise the same for clone detection. So, with our Hypermodelling approach to load code into a DW, every query could be customized to meet specific requirements or just be sliced by a specific viewpoint.

### *2.2. Recommendation*

Code recommendation systems mine facts that plenty of programmers have done out of a code base. Imagine, a developer overrides a method. The written method is compared with the recommendation data and she gets proposed which methods were called by others that did an override of this method. Exactly the same information can be revealed with an OLAP query. This information can be presented to a developer to recommend her what others did.

What we describe is a typical application of a code recommender system. Therefore, we propose a query process for recommender systems in Figure 1. A developer encodes functionality and queries containing this encoded functionality are done. These queries reveal similar code artifacts and most things that others have done in those are proposed to a developer. The developer continues to encode functionality. This encoded functionality can be added as refinement to the query to get more detailed information on what most others did in a similar situation. With these queries, the recommendation data can be generated live and adapted easily. The Hypermodelling approach ensures that different code bases can be loaded into a DW. This allows us to reuse infrastructure that is designed for big data.

### *2.3. Clone Detection*

Similarly to recommendation, we also realize live clone detection. Imagine a developer encodes a method, sharing a high similarity or equality to another method. Today, she continues her work. However, in this moment, she has knowledge at hand what she has programmed and can easily compare it with potential clones. This step fits perfectly into workflow and potential clones can be avoided easily. We propose to use our Hypermodelling approach to detect code clones live though a query based approach as described in Section 2.2. Through queries the detection process can be easily adjusted to project specific requirements.

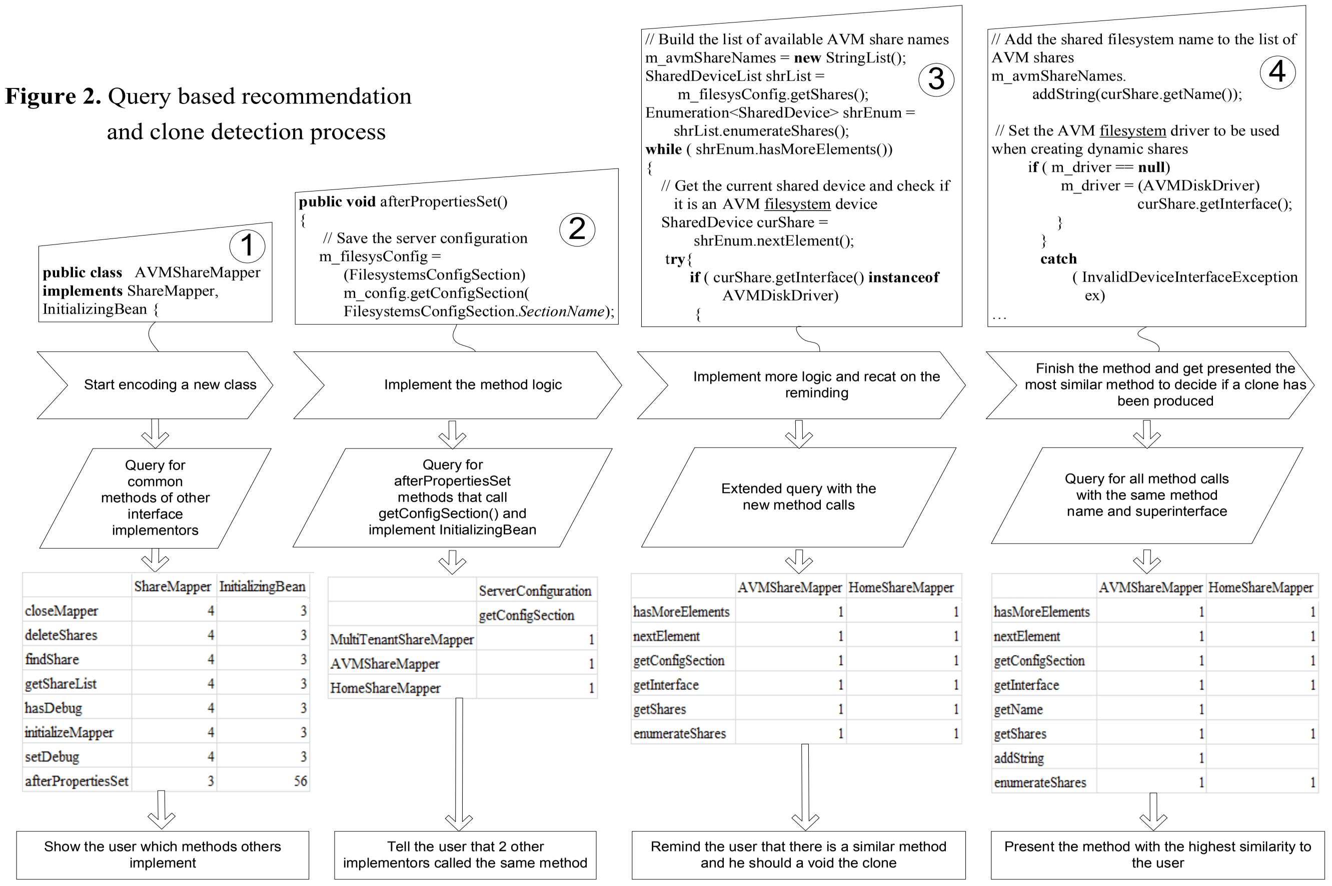
For clone detection, imagine a developer is encoding functionality in a method. She creates the method declaration with all method parameters and then she encodes logic into the method. Methods of objects are called and other constructs are realized. Like for recommenders, regularly queries are executed to determine duplicates. If similarity is too high, code is presented to the developer so that she can decide if she has produced a clone. We show this process in Figure 1, that also describes a clone detection cycle.

## 3. Evaluation

In order to evaluate the application ability of recommendation and clone detection based on queries, we select a method of a program that we describe in [6]. Data in our DW and queries on a real project demonstrate that our approach enables recommendation not only with prepared data. We depict a class (*AVMShareMapper*) that implements two interfaces that are implemented by other classes of the application. Thereby, we query if others also implement these interfaces to ensure a valid example that shares similarities with other classes. We select the *afterpropertiesSet* method for investigation and divide the method in four different parts. For every part, we execute queries to simulate how a developer would encode this method and queries would be executed in the background. Our scenario is mainly based on live clone detection. However, the same approach of queries can be used for recommendation.

Figure 2 shows the extended coding process that can be supported by queries. We imagine the coding process there as follows: In the first step a developer encodes the class body. Then, she goes on and encodes step by step a method. We split the *afterpropertiesSet* method and arranged it above, corresponding to the process of a developer. We describe exemplary queries in natural language beneath the process. Those queries can be executed with the DW query languages from the development environment in the background, while the developer is coding. Behind queries is their

**Figure 2.** Query based recommendation and clone detection process


1
2
3
4
Start encoding a new class
Implement the method logic
Implement more logic and recat on the reminding
Finish the method and get presented the most similar method to decide if a clone has been produced
Query for common methods of other interface implementors
Query for afterPropertiesSet methods that call getConfigSection() and implement InitializingBean
Extended query with the new method calls
Query for all method calls with the same method name and superinterface
Show the user which methods others implement
Tell the user that 2 other implementors called the same method
Remind the user that there is a similar method and he should a void the clone
Present the method with the highest similarity to the user

```
public class  AVMShareMapper
implements ShareMapper,
InitializingBean {
```

```
public void afterPropertiesSet()
{
    // Save the server configuration
    m_filesysConfig =
        (FilesystemsConfigSection)
        m_config.getConfigSection(
        FilesystemsConfigSection.SectionName);
```

```
// Build the list of available AVM share names
m_avmShareNames = new StringList();
SharedDeviceList shrList =
        m_filesysConfig.getShares();
Enumeration<SharedDevice> shrEnum =
        shrList.enumerateShares();
while ( shrEnum.hasMoreElements())
{
    // Get the current shared device and check if
    it is an AVM filesystem device
    SharedDevice curShare =
        shrEnum.nextElement();
    try{
        if ( curShare.getInterface() instanceof
            AVMDiskDriver)
        {
```

```
// Add the shared filesystem name to the list of
AVM shares
m_avmShareNames.
        addString(curShare.getName());

 // Set the AVM filesystem driver to be used
when creating dynamic shares
        if ( m_driver == null)
            m_driver = (AVMDiskDriver)
                    curShare.getInterface();
        }
    }
    catch
        ( InvalidDeviceInterfaceException
        ex)
…
```

| | ShareMapper | InitializingBean |
|---|---|---|
| closeMapper | 4 | 3 |
| deleteShares | 4 | 3 |
| findShare | 4 | 3 |
| getShareList | 4 | 3 |
| hasDebug | 4 | 3 |
| initializeMapper | 4 | 3 |
| setDebug | 4 | 3 |
| afterPropertiesSet | 3 | 56 |

| | ServerConfiguration |
|---|---|
| | getConfigSection |
| MultiTenantShareMapper | 1 |
| AVMShareMapper | 1 |
| HomeShareMapper | 1 |

| | AVMShareMapper | HomeShareMapper |
|---|---|---|
| hasMoreElements | 1 | 1 |
| nextElement | 1 | 1 |
| getConfigSection | 1 | 1 |
| getInterface | 1 | 1 |
| getShares | 1 | 1 |
| enumerateShares | 1 | 1 |

| | AVMShareMapper | HomeShareMapper |
|---|---|---|
| hasMoreElements | 1 | 1 |
| nextElement | 1 | 1 |
| getConfigSection | 1 | 1 |
| getInterface | 1 | 1 |
| getName | 1 | |
| getShares | 1 | 1 |
| addString | 1 | |
| enumerateShares | 1 | 1 |

result or at least an excerpt. At the bottom, possible ensuing actions are described. In the following, we go through the process of Figure 2:

First (1), a developer starts encoding the class and implements the *ShareMapper* and the *InitializingBean* interface. These two interfaces are used to create a query for the most common method names of the classes that implement one of the interfaces.[4] The result shows the amount of children types with the same method name. The result is sorted following the occurrence of method names of *ShareMapper* implementers. It would be possible to sort the result after the occurrence in *InitializingBean* or to merge most common method names in both interface implementers. Anyway, the result information can be used to show developers which methods other developers implemented by extending a certain interface. For our scenario, we imagine that developers see plenty of times the *afterPropertiesSet* method is implemented and start encoding this. Thereby, it can be recognized that a developer uses a method name based on the *InitializingBean* and the following queries can be specialized on this interface. To give a better impression about the technique, we present the query in Listing 1. The parent class or interface is named *ParentType* and the CodeStructure is the OLAP cube. The query is based on multi-dimensional expressions standard[5] and shows that the amount of methods for children is computed for extenders of the *ShareMapper* and *InitializingBean*.

```
SELECT { [ParentType].[Name].&[ShareMapper], [Parent].[Name].&[InitializingBean] } ON COLUMNS ,
{ [Method].[Name].[All].CHILDREN } ON ROWS
FROM [Code-Structure]  WHERE ( [Measures].[Method-Count] )
```

**Listing 1.** Query for method names of interfaces children

In the second step (2), the developer starts encoding logic of the *afterPropertiesSet* method. She calls a method of what is used to refine the former query. It is enriched with information which methods are called to reveal which types also obtain same methods, implement the interface and have the same method name. Such kind of similarity can be an indicator that the developer produces a clone. Furthermore, maybe the developer is putting effort and thoughts into implementing a method that is already implemented. Therefore, a developer gets presented other methods that share a high similarity. If a user rejects the proposals and wants to encode further functionality, the information about similar methods can also be used to present methods that are called in the similars. We show the corresponding query in Listing 2. The result types of this query that share a similarity (*MultiTenantShareMapper*, *HomeShareMapper*) can also be used in another query to generate recommendation information. Exemplarily, such a recommender query is shown in Listing 3. The called methods of two similar types (*MultiTenantShareMapper*, *HomeShareMapper*) are computed based on a query. This can be used to propose a user which other methods are called by other developers in a similar situation.

```
SELECT { [CalledMethod].[Name].&[getConfigSection]} ON COLUMNS ,
{[Type].[Name].[All].CHILDREN} ON ROWS   FROM [Code-Structure]
WHERE ( [Measures].[Method Calls],[ParentType].[Name].&[InitializingBean],
[Method].[Name].&[afterPropertiesSet] )
```

**Listing 2.** Determining similar code

```
SELECT NON EMPTY { [Measures].[Method Calls  Anzahl] } ON COLUMNS ,
{ [Called Method].[Name].[All].CHILDREN } ON ROWS
FROM [Code-Structure]
WHERE ( [Type].[Name].&[HomeShareMapper], [Parent - Type].[Name].&[InitializingBean],
[Method].[Name].&[afterPropertiesSet] )
```

**Listing 3.** Determining what similar code did for recommendation

[4]Note, a query can be sliced by a project or a specific package. Plenty of customizations are thinkable.

[5]We show a few queries to give an impression of the query language. For further information, see the MDX language reference: http://msdn.microsoft.com/en-us/library/ms145595.aspx / http://xmlforanalysis.com

Succeeding (3), a developer encodes more functionality. In the background the query is extended to compare method calls of the current method with the ones of other extenders. It is like a fail save detector that tries to uncover if there is a clone produced. With every new method call, the query is extended and executed again. After six shared method calls and the super interface, enough indicators are collected to remind the user with a traffic light or a pop up about similarity.[6]

Finally, in the forth step (4), the user finishes the method. This is recognized in the background and a complete query for all method calls is executed. Out of the high similarity of the method calls, it is assumed that the method logic is maybe "externalizable" and customizable through parameters. However, the easiest way is to present methods with a high similarity to the user and let him decide if this is a clone.

## 4. Related Work

Our prior works describes a general approach how code structure can be processed with a DW and queries [4, 5, 6]. In this paper, we focused on a concrete application.

Others describe different clone detection methods [2]. One clone detection method [3] proposes live clone detection with a client-server architecture. In comparison to our approach, the described detection [3] is not customizable. Another one, CloneDetective [8], offers an advanced framework and tool chain for clone detection, which is especially geared towards flexibility of clone detection research. Hypermodelling, on the contrary, targets to utilize DW technology. In general, we see the use of DWs as an addition to known clone detection mechanisms and not as competitor. DWs often already exist in enterprises where they are used for business applications. Therefore, our approach makes a reuse of DW technology for clone detection possible. Hence, further research should determine in detail if and which of those other clone detection approaches can be realized with DW technology.

Other related work can be found in the area of code recommender systems [1]. They provide methods to mine data out of code bases and generate recommendations out of it. The whole recommendation is hard wired and fixed. Further, no scaling technology for live recommendations with a client server infrastructure is proposed. With our approach, through DW query based recommendations a more flexible and adjustable infrastructure is at hand. Therefore, we see the emerging need to investigate DW technology further to advance the recommender systems.

## 5. Conclusion and Future Work

We described the problem of large code bases and the inflexibility of current code recommendation and challenges of clone detection systems. We proposed to overcome these limitations through a query based approach that uses DW technology. Our approach is evaluated and its application ability is shown by queries to a real code base. So, the next generation recommendation and clone detection techniques can be based on DW technology. Related work reveals possible synergies with other research and indicates that DW technology is a promising area to advance software engineering.

[6]Note, at this moment such kind of queries can also be used to propose method calls like in step 2.

In general, we see the need to describe our method in more details. Additionally, the use of DW technology enables further research on DW based clone detection. We showed the capability to find “full method clones”. However, clones can also be copied code fragments within different methods. Therefore, further investigations can focus on identification of queries for these clones. Thereby, we see ways to compute similarity indicators based on queries and the clone granularity level (method or fragment based) as important questions. The precision of our technique is fixed to method calls, discarding method parameters. Hence, the same called methods that take different parameters and are considered as clones. For that reason, the possibility to enhance or adjust the precision of our queries through additional facts should be considered. Therefore, we see the need to work together with industry developers to evaluate which level of precision and granularity is desired in practice.

Furthermore, we connect code recommendation with clone detection. Therefore, our work makes it possible to regard both areas together and investigate possible synergies in the future. This connection is an additional difference to previous research. Generally, our approach shows a rudimentary and first scenario with primitive queries. More complex queries, scenarios, and areas are of interest for further investigations. For instance, the area of refactoring is also near clone detection and relations to it can be investigated. We also see an advanced trail by integrating the context (e.g., the package, the prior studied code, or the task) wherein a developer encodes functionality to advance recommendation systems. Currently, code recommenders are dull and based on the same rule set. Our dynamic query based approach enables further research, how recommendations need to be altered and adjusted to different contexts of a developer to respect his current programming tasks within the recommendation.

## References

[1] M. Bruch, M. Mezini, and M. Monperrus, Mining subclassing directives to improve framework reuse, In: *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, IEEE, 2010, 141-150.

[2] C. K. Roy, J. R. Cordy, and R. Koschke, Comparison and evaluation of code clone detection techniques and tools: a qualitative approach, *Science of Computer Programming* **74**(7) (2009), 470-495.

[3] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, and H. Iada, *Shinobi: a Real-Time Code Clone Detection Tool for Software Maintenance*, Technical Report NAIST-IS-TR2007011. Graduate School of Information Science, Nara Institute of Science and Technology, 2008.

[4] T. Frey, V. Köppen, and G. Saake, Hypermodelling – introducing multi-dimensional concern reverse engineering**.** In: *2nd International ACM/GI Workshop on Digital Engineering (IWDE),* Germany, 2011, 58-66.

[5] T. Frey, Hypermodelling for drag and drop concern queries. In: *Proceedings of Software Engineering 2012 (SE2012)*, Gesellschaft für Informatik (GI), Berlin, Germany, 2012, 107-118.

[6] T. Frey and V. Köppen, Exploring software variance with hypermodelling – an exemplary approach. In: S. Jähnichen, A. Küpper, S. Albayrak, editors, *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik,* Berlin, Germany, 2012, 121-140.

[7] W. H. Inmon, *Building the Data Warehouse*. 4th ed. J.Wiley & Sons, New York, USA, 2005.

[8] E. Juergens, F. Deissenboeck, and B. Hummel, CloneDetective – a workbench for clone detection research. In: *Proceedings of the 30th International Conference on Software Engineering*, IEEE, 2009, 603-606.