An Approach for Extracting Modules from Monolithic Software Architectures

Ricardo Terra, Marco Túlio Valente, Roberto S. Bigonha

Universidade Federal de Minas Gerais, Brazil

{terra,mtov,bigonha}@dcc.ufmg.br

Abstract. *Software architectures commonly evolve into unmanageable monoliths, leading to systems that are difficult to understand, maintain, and evolve. In such common scenarios, developers usually have to invest considerable time in re-architecting the entire application, in order to restore its modular structure. However, re-architecting process are usually conducted in ad hoc way, without following any set of principled guidelines and methods. In order to tackle this problem, this paper describes an approach to segregate the code of a given system concern into modules with well-defined interfaces. We also present the first results of applying the proposed approach in an illustrative application.*

1. Introduction

Software architectures commonly evolve into unmanageable monoliths, leading to systems that are difficult to understand, maintain, and evolve. In such common scenarios, developers usually have to invest considerable time in re-architecting the entire application, in order to restore its modular structure [9, 10]. However, re-architecting processes are usually conducted in ad hoc ways, without following any set of principled guidelines and methods.

In order to tackle this problem, we propose in this paper an approach to extract modules from monolithic architectures. The proposed approach is based on a series of refactorings and aims to modularize concerns through the isolation of their code fragments. Figure 1a illustrates the approach goal. In this figure, letters `yyy` and `zzz` denote code related to a given concern. In the monolithic version of the system, the code suffers from tangling and scattering. Afterwards, such code is extracted and moved to new classes leaving only invocations (represented by the arrows) in the original class.

The remainder of this paper is organized as follows. Section 2 outlines the main phases of the remodularization approach we are currently investigating. Section 3 presents the first results of applying the proposed approach in an illustrative application. Section 4 presents related work. Finally, Section 5 discusses further and ongoing work.

2. Remodularization Approach

The proposed remodularization approach is summarized in Figure 1b. It is important to mention that the approach is able to extract modules from code fragments related to the manipulation of a well defined group of classes, such as classes provided by frameworks, abstract data types, etc.

To illustrate the process, we will first rely on a simple example based in the isolation of a hypothetical framework called `Zeta` that contains the classes `X`, `Y`, and `Z`. Basi-

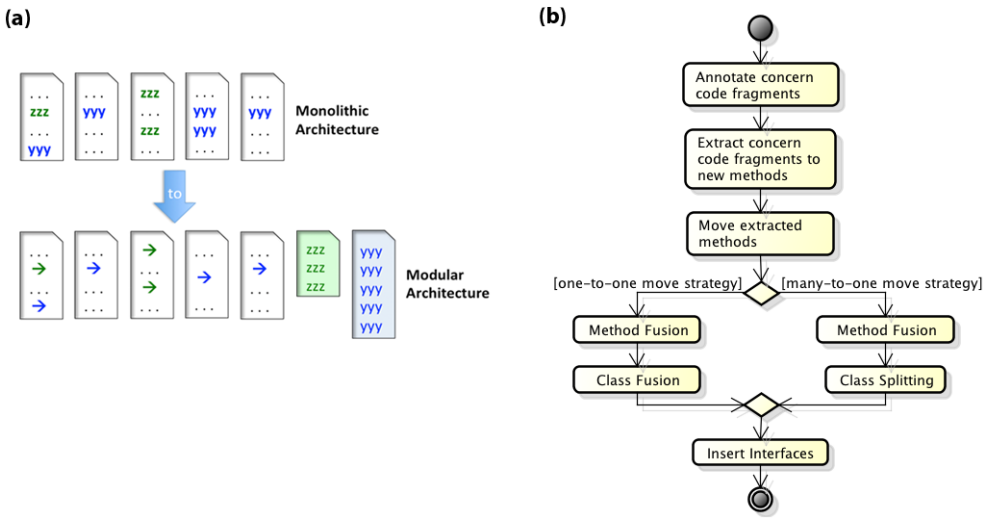

Figure 1. (a) Proposed approach goal and (b) their phases

cally, code associated to this framework is our target concern. The following subsections explain each phase of the approach.

2.1. Code Annotation

First, we must annotate the code fragments that implement the concern we intend to modularize. It is necessary to identify which code fragments are related to the concern. For instance, using the `Zeta` concern example, the code fragments that manipulate `Zeta` types are related to the concern. Hence, it is possible to annotate an entire class, an entire method, part of a method, or an attribute.

We use simple comments to annotate the code. These comments have the prefix "`c:`" followed by the concern name as can be observed in Listing 1. It shows a method called `exampleMethod` from the `ExampleClass` class whose main responsibility is not directly related to the `Zeta` framework. Meanwhile, this method uses objects from `Zeta` (lines 6-9).

```
1  public class ExampleClass {
2      private String via; ...
3
4      public void exampleMethod( String param ) { ...
5          int number = Integer.parseInt( param );
6          X x = new X( number );          //c:zeta
7          Object o = x.fx2( this.via );   //c:zeta
8          Y y = new Y();                  //c:zeta
9          y.fy2( o.toString() );          //c:zeta
10         ...
11     }
12 }
```

Listing 1. Annotation of `Zeta` code fragments

2.2. Method Extraction

This phase is responsible to extract the concern code fragments to new methods of the same class. For this purpose, we simply rely on the well-known *Extract Method* refactoring [3]. As can be observed in Listing 2, `Zeta` code fragments have been extracted to a new private method called `f1` (lines 11-16).

```
  public class ExampleClass {
2     private String via; ...

4     public void exampleMethod( String param ) { ...
          int number = Integer.parseInt( param );
6         this.f1( number );
          ...
8     }

10    //from:ExampleClass.exampleMethod(String)
      private void f1( int number ) {
12        X x = new X( number );          //c:zeta
          Object o = x.fx2( this.via ); //c:zeta
14        Y y = new Y();                  //c:zeta
          y.fy2( o.toString() );          //c:zeta
16    }
  }
```

Listing 2. `Zeta` **code fragments extracted to new methods**

2.3. Method Moving

It is the most crucial phase of the proposed approach. In this phase, architects should decide how to move the extracted methods to new modules. As illustrated in Figure 2, we propose the two following strategies for moving the extracted methods:

1) One-to-one move strategy: This strategy is indicated when the extracted methods reference attributes or call other extracted methods. As illustrated in Figure 2, suppose that class `A` has two extracted methods `f1` and `f2` and class `B` has an extracted method `g1`. This strategy moves methods `f1` and `f2` and the fields only used by such methods to a new class `A′` and similarly method `g1` to a new class `B′`. Furthermore, class `A` will have one attribute of the type `A′` and class `B` will have one attribute of the type `B′`. Finally, the original invocations of moved methods must be adjusted to invoke them by this new attribute.

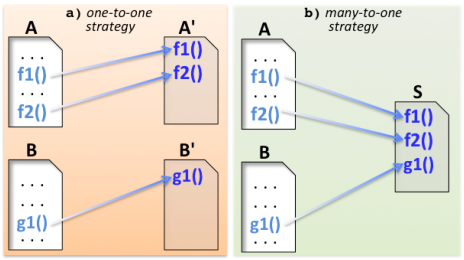

Figure 2. Move method strategies

Listing 3 illustrates the application of this strategy in the `Zeta` example. We have created the class `Zeta1` (new target class, lines 16-33) and an attribute of its type has been inserted in the origin class `ExampleClass` (line 2). Because the attribute `via` is used only by the extracted methods, it has also been moved to the target class (line 17) and constructor (line 20) and accessor methods (lines 22-24) have been created. Finally, we updated the constructor from the origin class (lines 4-7) and the invocations accordingly (line 11).

```
  public class ExampleClass {
2     Zeta1 zeta1; ...

4     public ExampleClass( ... , String via) {
          ...
6         this.zeta1 = new Zeta1( via );
      }
8
      public void exampleMethod( String param ) { ...
10        int number = Integer.parseInt( param );
          this.zeta1.f1( number );
12        ...
      }
14 }
```

```
16 public class Zeta1 {
       private String via; //from:ExampleClass
18     ...
20     public Zeta1(String via) { this.via = via; }
22     public String getVia() { return this.via; }
24     public void setVia( String via ) { this.via = via; }
26     //from:ExampleClass.exampleMethod(String)
       public void f1( int number ) {
28         X x = new X(number);          //c:zeta
           Object o = x.fx2( this.via ); //c:zeta
30         Y y = new Y();                //c:zeta
           y.fy2( o.toString() );        //c:zeta
32     }
   }
```

Listing 3. Application of *one-to-one* move strategy in `Zeta` example

2) Many-to-one move strategy

This strategy is indicated when the extracted methods rarely use attributes. Basically, the strategy prescribes moving all extracted methods—despite of their origin class—to a single new class. As illustrated in Figure 2b, suppose again that class `A` has two extracted methods `f1` and `f2` and class `B` has an extracted method `g1`. This strategy moves methods `f1`, `f2`, and `g1` as static methods of a single new class `S`. Thus, classes `A` and `B` will access—in a static way—the moved methods.

Listing 4 illustrates the application of this strategy in the `Zeta` example. As can be observed, we have created the class `Zeta` (target class, lines 11-20) and moved all extracted methods to this class with public visibility and a static modifier (line 13). It is important to highlight that because the attribute `via` is used by method `f1`, it is passed as formal parameter (line 13). Finally, we have adjusted the invocations of the moved method `f1` (line 6).

```
   public class ExampleClass {
2      private String via; ...
4      public void exampleMethod( String param ) { ...
           int number = Integer.parseInt( param );
6          Zeta.f1( number, this.via );
           ...
8      }
   }
10
   public class Zeta {
12     //from:ExampleClass.exampleMethod(String)
       public static void f1( int number, String via ) {
14         X x = new X( number );    //c:zeta
           Object o = x.fx2( via ); //c:zeta
16         Y y = new Y();            //c:zeta
           y.fy2( o.toString() );    //c:zeta
18     }
       ...
20 }
```

Listing 4. Application of *many-to-one* move strategy in `Zeta` example

2.4. Method Fusion

The classes created in the previous phase may have duplicated or similar methods. To determine these methods, we are currently investigating the following fusion criteria:

i. *By Equality*: identifies equal methods (i.e., methods that have exactly the same sequence of statements).

ii. *By Generalization*: identifies similar methods whose formal parameter types can be generalized in order to produce a single method. For example, suppose that method `f` receives a `Set` and method `g` receives a `List` parameter. Suppose also these methods are almost equivalent and they do not use specific methods from `Set`

or `List`, therefore it is possible to create a single method having as parameter the supertype `Collection`.

iii. *By Parameterization*: identifies similar methods that could be merged into a single method containing an expanded list of formal parameters. For example, suppose that methods `f` and `g` have almost the same body except by the fact that `g` has an extra behavior. Therefore, it is possible to create a single method including a new formal parameter to enable or disable this extra behavior.

2.5. Class Regrouping

Towards a better grouping of the moved methods, we are currently investigating means to *fusion* or *split* the created classes. When using the *many-to-one* strategy, the single created class may have several non-related methods, then it is reasonable to split such class into smaller and more cohesive classes—*Class Splitting*. On the other hand, when using the *one-to-one* strategy, some created classes may have no attributes, then it may also be reasonable to fusion them into more cohesive classes—*Class Fusion*. For this purpose, we are working in the following regrouping heuristics:

i. *By Signature*: regroups the extracted methods by their signature, i.e., based on their formal parameters and return types. For example, suppose method `f1` receives a `Y` parameter, method `f2` returns a list of `Y`'s, method `f3` receives a `Z`. Then, using this heuristic, `f1` and `f2` will become part of a group and `f3` of another.

ii. *By Known Types*: regroups the extracted methods according to types handled by them. For example, suppose that methods `f1` and `f4` handle the type `Y`, and that methods `f2` and `f3` handle the type `Z`. Then, using this heuristic, `f1` and `f4` will become part of a group, and `f2` and `f3` of another group.

iii. *By Name*: regroups the extracted methods taking into consideration their names. For example, suppose there are methods `saveY`, `saveZ`, `deleteY`, and `deleteZ`. Thus, using this heuristic, methods with the prefix `save` will become part of a group and those with the prefix `delete` of another group.

iv. *By Origin Class*: regroups the extracted methods based in the class from where they have been moved. For example, suppose methods `f1`, `f2` and `f4` have been extracted from class `A` and method `f3` from class `B`. Then, using this heuristic, `f1`, `f2` and `f4` will become part of a group and `f3` of another group.

2.6. Interface Insertion

Once the code fragments related to the concern are modularized, such modules need well-defined interfaces. Programming to interfaces is a design principle that advocates that developers should separate the interface of a component from its implementation. Essentially, it creates an abstract layer between client and server components [4].

Listing 5 illustrates the interface insertion in the `Zeta` example using *one-to-one* move strategy. As can be observed, we have created the interface `IZeta1` (lines 16-20) and made the `Zeta1` class implement it (line 22). We have also created the class `FactoryZeta` (lines 26-30) with respective factory method (lines 27-29). We have adjusted the attribute `zeta1` in the origin class to have the interface type (line 2). Finally, the constructor gets a reference to the concrete object by calling its factory method (line 6).

```
public class ExampleClass {
    IZeta1 zeta1; ...

    public ExampleClass( ... , String via) {
        ...
        this.zeta1 = FactoryZeta.getZeta1( via );
    }

    public void exampleMethod( String param ) { ...
        int number = Integer.parseInt( param );
        this.zeta1.f1( number );
        ...
    }
}

public interface IZeta1 {
    public String getVia();
    public void setVia( String via );
    public void f1( int number );
}

public class Zeta1 implements IZeta1 {
    /* same body */
}

public class FactoryZeta {
    public static IZeta1 getZeta1( String via ) {
        return new Zeta1( via );
    } ...
}
```

Listing 5. Interface insertion on `Zeta` after *one-to-one* move strategy

3. An Illustrative Application

To illustrate the approach, we have modularized some concerns of `MyWebMarket`, which is an illustrative web application we have developed[1]. This system handles common activities found in a simple e-commerce system, including managing customers and products, purchasing orders, generating reports, etc.

As illustrated in Figure 3, the system has been developed using Java EE and relies on the following frameworks: (a) Struts, responsible to handle HTTP requests and responses; (b) Hibernate, responsible for object/relational persistence; (c) DWR, responsible to provide Ajax communication; (d) Log4J for logging; (e) Quartz to schedule jobs; (f) JavaMail to send emails; (g) JasperReport to generate reports. More important, we have deliberately implemented the system as a monolithic block. Particularly, its classes are coupled in an anti-modular way to the aforementioned frameworks.

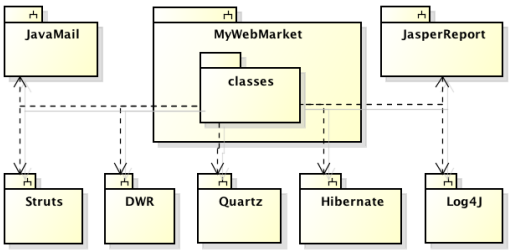

Figure 3. `MyWebMarket` and its frameworks

The next subsections describe the modularization of `MyWebMarket`'s concerns. We will assume that the system developers have annotated the code fragments related to each concern. To demonstrate the improvements on system quality achieved by the remodularization approach described in this paper, we will rely on conventional metrics

[1] All Java projects are publicly available at http://www.dcc.ufmg.br/~terra/wmswm2012.

for cohesion and coupling [2]. We will use LCOM (*Lack of COhesion of Methods*) for cohesion and CBO (*Coupling Between Object classes*) for coupling.

1) Modularization of Persistence Concern: Once the code fragments related to Hibernate have been annotated, they have been extracted to 22 new methods in 7 distinct classes. In the *Method Moving* phase, we used *one-to-one* strategy because the extracted methods have presented a high degree of similarity at the class level. No attribute could be moved to the new class because they were used by other methods. Therefore, 13 methods had to incorporate attributes as their formal parameters. With respect to *Interface Insertion*, we have created interfaces for each class and respective abstract factory methods.

2) Modularization of Scheduling Concern: Only one class in the system uses Quartz. Thus, we easily extracted two methods that did not use class attributes and therefore we used *many-to-one* move strategy. Finally, we have applied the *Interface Insertion* refactoring to a single class.

3) Modularization of Reporting Concern: Only one class in the system uses JasperReport. Thus, we have extracted two methods that also did not use class attributes and again we used the *many-to-one* move strategy. Last, we have applied the *Interface Insertion* refactoring and created the factory method.

4) Modularization of Mailing Concern: Only one class in the system uses JavaMail, more specifically there is one scheduling job responsible to send emails to administrators. Thus, we have extracted this unique method. Because it uses several class attributes, we used *one-to-one* move strategy, resulting in three attributes moved to the new class. Finally, we have performed the *Interface Insertion* and created one factory method.

Discussion: This case study, although simple, provides an indicator of the applicability of the proposed approach and an indicator about the gains that can be achieved by such approach. As presented in the Table 1, each modularization has contributed to improve cohesion (LCOM) and coupling (CBO). For instance, comparing the monolithic version with the last remodularized version, LCOM has decreased from 0.367 to 0.227 (an improvement of 38.14%) and CBO has decreased from 15 to 4 (an improvement of 60%).

Table 1. MyWebMarket quality improvement by Remodularization

	LCOM	CBO
Monolithic version	0.367	15
Version 1 (persistence)	0.282	9.333
Version 2 (scheduling)	0.261	6.2
Version 3 (reporting)	0.243	4.857
Version 4 (mailing) – last	0.227	4

It is important to mention that due to the small size of the application, we have decided to do not perform method fusion or class regrouping. Nevertheless, we are currently working in case studies of large monolithic applications.

4. Related Work

The concept of refactoring was first coined by Opdyke [7] and has been consolidated by the catalog proposed by Fowler [3]. Afterwards, Kerievsky [6] has written a catalog describing how to refactor to design patterns [5] and Bourquin and Keller [1] describe high-impact refactorings, which are refactorings with a strong impact on the system quality.

Tsantalis and Chatzigeorgiou [11] argue on the importance of moving state and behavior between classes to reduce coupling and to increase cohesion. They also propose a methodology to identify move method refactoring opportunities. Rama and Patel [9] have analyzed several large system modularization projects and identified recurring patterns. They define, formalize and demonstrate the applicability of six of such patterns (termed modularization operators), such as module decomposition and module union operators. Sarkar *et al.* [10] alert about the architectural erosion and highlight the importance on remodularizations that tackle such problem. They describe a modularization approach adopted by an IT company to reengineer a monolithic banking application, which grew from 2.5 to 25 MLOC. In a previous work we have compared and illustrated the use of three static architecture conformance techniques, namely reflection models, dependency structure matrices (DSM), and source-code query languages [8] Therefore, this research can be seen as a natural continuation of this previous work, but tackling not the detection but the correction of modular violations.

5. Conclusions and Further Work

This paper has outlined a set of principled guidelines to segregate the code of a given system concern into modules with well-defined interfaces. Our ultimate goal is to provide to architects means to extract modules from monolithic architectures through an organized process which contemplates since the code annotation until interface insertion, thereby preventing architects to conduct rearchitecturing processes in ad hoc way.

Moreover, we have presented the first results of applying the proposed approach in an illustrative application called `MyWebMarket`. After we have extracted four modules from the monolithic version of this system, cohesion and coupling have improved 38% and 60%, respectively.

Currently, we are working on: (i) the formalization of the proposed approach; (ii) new case studies in large monolithic systems to better demonstrate the applicability and scalability of our approach; (iii) automatization of the approach as an Eclipse plug in.

References

[1] F. Bourquin and R. K. Keller. High-impact refactoring based on architecture violations. In *11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 149–158, 2007.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, June 1994.

[3] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[4] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[5] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[6] J. Kerievsky. *Refactoring to Patterns*. Pearson, 2004.

[7] W. F. Opdyke. Refactoring object-oriented frameworks. Doctoral thesis, University of Illinois at Urbana-Champaign, June 1992.

[8] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonca. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27:82–89, 2010.

[9] G. Rama and N. Patel. Software modularization operators. In *26th International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[10] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE Software*, 26:28–35, 2009.

[11] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 99:347–367, 2009.