32 On Analyzing the Interfaces of Components

Jens Knodel
Fraunhofer Institute for Experimental Software Engineering (IESE), Sauerwiesen 6, D-67661 Kaiserslautern, Germany
knodel@iese.fraunhofer.de

Abstract

Reusing existing software components can significantly reduce the effort needed for the development of new products. In order to enable reuse, existing conceptual components have to be identified, well documented and in some cases migrated into physical component. This paper presents an approach that helps when migrating parts of existing software systems into components to be reused in other projects.

Keywords: component, interfaces, request-driven reverse architecting, reverse engineering, software architecture

32.1 Introduction

Reuse is a promising solution for challenges for software-developing organizations and their need for reducing cost, effort and time-to-market, the increasing complexity and size of the software systems, and increasing requests for high-quality software and individually customized products for each customer.

Documented interfaces are one of the prerequisites for effective reuse of components. Reuse works when the developers know which functionality is provided by a component and how to access the functionality implemented in such a component. Components in the context of interface analysis are collections of source code entities (e.g., files, groups of logically related routines, single or groups of classes or packages, or even whole subsystems). The interface analysis technique can be applied for one of the following purposes:

- Reduction of the complexity of given components with respect to the number of offered routines by minimizing the provided interfaces to only the actually used interfaces when to facilitate reuse
- Documentation of source code spots in usage lists where to change accesses to a component when migrating the software system towards component-based development.
- Extension of architectural descriptions (e.g., the module and/or the code view, see [1]) by explicit notation of the provided functionality of a component.
- Migration of a group of entities towards an encapsulated component with explicit boundaries.

Our interface analysis technique reveals the connections of the subject component to the rest of the software system, or if it should be migrated into a separate component in future, it documents the spots to be changed and how the future component is embodied in the system. To achieve these goals we apply reverse engineering techniques in form of the interface analysis as described in the next sections.

32.2 Interfaces Analysis

The interface analysis technique is part of an analysis catalogue developed at Fraunhofer IESE. All analyses of this catalogue can be considered as request-driven reverse architecting analyses. Each analysis in this catalogue operates on a fact base produced by fact extraction from existing artifacts (e.g., architecture descriptions, the source code). An analysis is always initiated by a concrete request that delivers the results on demand. The results of an analysis are (architectural) views or subset of views. Such a view contains only selected aspects of the systems that are of interest in the context of the given request.

A prerequisite for the interface analysis is that the fact base contains interface related information (e.g., calls or method invocation of class methods, access to variables or class members). Figure 32.1 shows the inputs to the interface analysis, a component or a collection of entities that should be migrated towards a component.

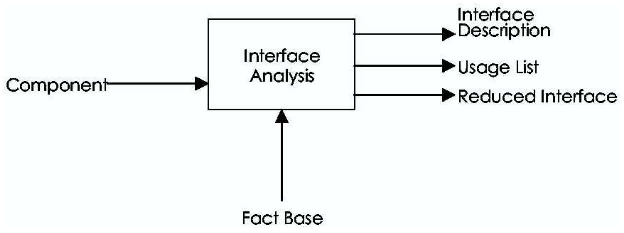

Abbildung 32.1: Interface Analysis

The inputs are then analyzed with respect to dependencies of the component (or the collection of entities) to the rest of software system whereby dependencies means for instance data dependencies, import relations, inheritance, or caller-callee dependencies. The resulting interface descriptions will be twofold:

- Required dependencies: the required dependencies are entities that the component needs in order to work properly. Usually a component communicates with other components to accomplish its tasks. In the migration context, it has to be decided whether those required source code entities become part of the to-be build component or they constitute a separate component. For reusing of a component it is important to know about such dependencies beforehand because they may influence the decision whether to reuse the component or not.
- Provided dependencies: the provided dependencies (or usage list) are the part of the given component

where the system accesses the component that is where it needs some functionality implemented in the component in order to work properly. There are two options, on the hand we can only document the currently used entities from the component (i.e., when changing the signature of a routine, it is beneficial to locate each place where a call has to be adjusted) and, on the other hand, we are able to document the complete interface offered to the outside (i.e., when reusing the component in another context, it is beneficial to know about the complete interface). To document both is required when planning to achieve both goals.

An interface description usually contains a list of routines, global variables, and class members that can be accessed from the outside. Variables and class members may lead to additional implementation effort for get and set access that should be scheduled in the migration. The interface description we produce includes the signature and the return parameter of the specific routines as well as the locations of different usages and in which file they are implemented. The description basically resolves the kind of dependencies which is present for each instance (e.g., calls, inheritance, data dependencies, imports, etc.), but it is possible to focus only on specific kinds of relations. Figure 32.2 shows an example for such an interface description.

```
Required Routines of class1.method1:
   Class2.method2
     Located in: c:/code/file2.java
        Type: method
        Signature: void method2
            ( value: int )
        Returns: void
```

Abbildung 32.2: Interface Description Example

Tools automate the task of analyzing the interfaces by querying the fact base. The queries can be parameterized by the list of components (or the collections of entities) for which the interfaces should be described. Single entities of a component will usually overlap in their interface description or their usage list. For this reason, we can create a single description for the whole component that contains no redundancies. These descriptions can be sorted by different criteria (e.g., places of usages, name of routines, classes, etc). The results of such an interface analysis can now be utilized to achieve the different purposes mentioned in the beginning.

32.3 Conclusion

The interface analysis documents the provided interfaces and the actually used interfaces of source code entities. This information can be used for the following purposes:

- Migrating collections of entities into an encapsulated component.
- Reducing the broadness of provided interfaces to only the actual needed and therefore reducing the complexity in terms of the size.
- Documenting the description and usage of entities in an automated way.

The presented interface analysis technique helps to counteract the degeneration of the interfaces. A more detailed description of the approach and how it is embedded in architectural design in a product family context can be found at [2]. We applied the interface analysis successfully in the context of two industrial case studies. Future work will involve the application in further case studies and the integration of the interface analysis together with other analysis we provide in our catalogue of request-driven reverse architecting analyses.

32.4 Acknowledgement

This work is partially funded by the European Commission under EUREKA 2023/ITEA-ip00009 „FAct based Maturity through Institutionalization Lessons-learned an Involved Exploitation of System-family engineering“ (FAMILIES).

32.5 References

[1] C. Hofmeister, R. Nord, D.: Applied Software Architecture, Addison-Wesley, 2000.

[2] J. Bayer, T. Forster, D. Ganesan, J.-F. Girard, I. John, J. Knodel, R. Kolb, D. Muthig: Definition of Reference Architectures based on Existing Systems, Technical Report, Fraunhofer IESE, March 2004