17th Working Conference on Reverse Engineering, WCRE 2010

# Computer-Aided Extraction of Software Components

Andreas Marx, Fabian Beck, and Stephan Diehl
University of Trier, Germany
Email: {beckf,diehl}@uni-trier.de

***Abstract*—In a software project, outsourcing the development of a particular functionality, reusing a part in another software, or handing-over a part of the code to a new team member requires the extraction of an independent subset of the software—a component. This paper describes and analyzes the process of extracting such a component. We introduce an automated approach based on optimizing the cut between the new component and the remaining system. A visual development tool implements our approach and interactively supports the extraction. Finally, we look at the results of a thinking aloud user study and discuss the lessons learned about the extraction tool as well as the extraction process.**

## I. Introduction

Assume you are involved in the development of a software project. Your project grows and grows. And it finally reaches a point where driving the development forward overburdens you and your team. You are thinking about outsourcing a part of the project to another development team. If you have invested much effort in a highly modularized design or if you are just lucky, you will find an independent, well-defined component in your software, which you can simply hand over to the other team. If not, you will have to manually restructure your system in a time-consuming process until you have established such a component. Since we believe the latter case to be very common, we investigate in this paper how a tool might support you in the extraction of components.

**Identify Key Entities** This process starts with an existing project of a development team and the idea of extracting a part of the system that, for instance, another team is able to develop independently. The original development team knows the software in detail. The team probably has a clear idea which functionality could be candidate for this extraction. But this idea is more part of a semantic level than directly expressible in source code entities. Furthermore, this functionality might partly cross-cut over the dominating software architecture. Nevertheless, the developers would be able to name a few key source code entities to be extracted.

**Shape the Component** The main challenge is to shape the exact contour of the extracted part in the source code. The key source code entities could be the seeds for the final component. They are connected to other entities by dependencies like method calls, variable usage, inheritance, or aggregation. These links help to identify strongly coupled entities. A goal in the extraction process would be to avoid cutting such strong couplings.

**Generate a Contract** Once the contour is identified, the project can be split. Despite the optimization, some dependencies between the two parts will remain. To promote an independent development of both parts, a kind of contract is necessary. This contract should regulate the programming interface but not its implementation.

We already used the term *component* to denote the extracted part of the software. In the application at hand, such a component is not necessarily equivalent to a component in terms of component-based software engineering, but could be. In this paper we use the term according to an early definition of software components (1st ECOOP Workshop on Component-Oriented Programming, Summary [20]):

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

But we consider the second part of the definition—the independent deployment and composition—as an optional criterion. We want to focus on an independent development and hence do not insist on an independent deployment. Moreover, we do not impose strict constraints on the structure of the components, like conforming to a particular component model or targeting a particular component platform [19].

As the main contributions of this paper, we formally analyze and propose a solution to the problem of automatically choosing an appropriate component starting from a set of predefined key entities (Section II). Extracting this component requires a set of source code changes, mostly refactorings; the interface of the component becomes manifest in a contract (Section III). We implemented the process in a tool that enables developers to interactively extract a component using an intuitive visual model of the software project (Section IV). To evaluate our novel approach and the proposed tool, we conducted a thinking aloud study on three real-world systems (Section V).

## II. Extraction of Components

We intend to analyze the problem of extracting a component in general way, but without neglecting practical implications. We thus concentrate on object-oriented systems. Although we use Java terminology in the following to be clear and specific, similar concepts may also apply to other object-oriented languages.

### A. Code Dependencies

A class consists of a declaration, fields, constructors, and methods; an interface only of a declaration and methods. Inheritance relationships, method calls or field accesses represent different forms of dependencies among those source code entities. Such structural relationships will provide the underlying code dependencies that we have to consider when extracting a component. In particular, we regard the dependency types listed in Table I.

TABLE I
DIFFERENT DEPENDENCY TYPES AMONG SOURCE CODE ENTITIES.

| Dependency Type | Code entity $e$ depends ... | Allowed type of $e$ | | |
|---|---|---|---|---|
| | | Class/ Interf. | Field | Constr./ Method |
| **Inheritance** | ...on class/interface $A$ if $e$ inherits from $A$ or implements $A$. | × | | |
| **Type** | ...on class/interface $A$ if $A$ is used to specify a type in $e$. | × | × | × |
| **Constructor Call** | ...on constructor $c$ if $e$ calls $c$. | | | × |
| **Method Call** | ...on method $m$ if $e$ calls $m$. | | | × |
| **Field Access** | ...on field $x$ if $e$ accesses $x$. | | | × |

We allow multiple dependencies between two entities. For instance, three field accesses to the same field in one method lead to three dependencies. Later, these multiple dependencies may help to define the strength of an edge. Furthermore, please note that enumerations and reflections—two advanced concepts in Java—are ignored.

The code dependencies imply a graph structure on the code entities, which we formally define as follows.

**Definition 1** (Code Dependency Graph)**.** *Let $V$ be the set of source code entities consisting of all classes, interfaces, fields, constructors and methods of a program, let $E_D \subset V \times V$ represent all source code dependencies as introduced above, and let $E_I \subset V \times V$ be the set of inclusion relations according to the program structure, then the compound directed multi-graph $G = (V, E_D, E_I)$ represents the **code dependency graph** of the program.*

A compound graph denotes a graph with an additional hierarchy on the nodes [18]. This additional hierarchy is provided by the inclusion relations between the code entities: High-level code entities include low-level code entities. For instance, a class may include a set of fields and methods. Since the dependencies are directed, the graph at hand is directed, too. And since multiple edges between two nodes might exist, the graph is a multi-graph.

We consider classes and interfaces as elementary units for extracting a component. They are distinct units that can be moved independently—only minor adjustments are necessary. In contrast, moving methods would be much more complex. Furthermore, classes and interfaces represent a intermediate level of abstractions: They are fine-grained enough to cover the most important design decisions and they are granular enough to preserve an overview on the system.

For the component extraction, we define a simplified graph structure based on classes and interfaces.

**Definition 2** (Class Dependency Graph)**.** *Let $G = (V, E_D, E_I)$ be a code dependency graph, let $C$ be the set of classes and interfaces in $V$, and let $c \rightarrow^* c'$ denote that there is a path from $c$ to $c'$ in the inclusion relation $E_I$. Then, the directed graph $G_C = (C, E_C)$ with*

$$E_C = \{(c_1, c_2) | \exists v_1, v_2 \in V : c_1 \rightarrow^* v_1,\ c_2 \rightarrow^* v_2, \text{ and } (v_1, v_2) \in E_D\}$$

*is called **class dependency graph**.*

Classes and interfaces form the nodes of this aggregated graph. An edge between two classes (or interfaces) is formed by aggregating all dependencies that connected the one class or one of its descendants with the other class or one of its descendants in the original graph.

### B. General Component Extraction Problem

The user identifies key source code entities—a set of classes and interfaces—as the heart of the new component. The remaining nodes are free to be moved into the extracted component or to stay in the remaining part (also called *original component*). Nevertheless, some classes and interfaces might definitely belong to the core of the application and must not be extracted. We denote these classes and interfaces by *key entities of the original component*. Together, these three sets form the input for the component extraction process, the initial component partition.

**Definition 3** (Initial Component Partition)**.** *Let $C$ be the set of classes and interfaces, then a partition $s = (C_o, C_e, C_f)$ of $C$ where $C_o$ represents the key entities of the original component, $C_e$ the key entities of the extracted component, and $C_f$ the free nodes is called an **initial component partition** of $C$. The set of all possible initial component partitions is denoted by $S$.*

Based on such an initial component partition, the extracted component is to be formed by assigning the free nodes to the extracted or the original component. But the key entities are not allowed to be moved. A partition consisting of an extracted component and a remaining component created like this is called *valid*.

**Definition 4** (Valid Component Partition)**.** *Let $s = (C_o, C_e, C_f)$ be an initial component partition of $C$, then each partition $(C'_o, C'_e)$ of $C$ with $C_o \subset C'_o$ and $C_e \subset C'_e$ is called a **valid component partition** of $s$. The set of all valid component partitions is denoted by $\mathcal{V}(s)$.*

Extracting a component from a given software project consists of transforming an initial component partition into a valid component partition. Thus, finding an optimal extracted component is equivalent to finding an optimal valid component partition with respect to an objective function.

**Problem 1** (General Component Extraction)**.** *Given an initial component partition $s$ and an objective function $m : \mathcal{V}(s) \rightarrow$*

$\mathbb{R}$, *then a valid component partition* $s^* \in \mathcal{V}(s)$ *solves the* ***general component extraction problem*** *if it is optimal in terms of* $\forall s' \in \mathcal{V}(s) : m(s^*) \leq m(s')$.

The core of this optimization problem is the objective function. It defines the quality of a valid component partition.

As long as there are no further restrictions on the objective function, the general component extraction problem is difficult to solve. We have to exhaustively evaluate all possible $2^{|C_f|}$ valid partitions to find the best one.

### C. Simple Component Extraction Problem

An approach based on the dependencies between classes and interfaces, however, simplifies the complexity significantly. Assume a user-defined function $\mu$ states the dependency strength between two such code entities (e.g., by assigning weights to each edge type in the code dependency graph). Then a reasonable objective is to minimize the aggregated strengths of the dependencies that cross component borders (interdependencies).

**Problem 2** (Simple Component Extraction)**.** *Given an initial component partition $s$ and an objective function*

$$\overline{m} : \mathcal{V}(s) \to \mathbb{N}_0,\ (C'_o, C'_e) \mapsto \sum_{c'_o \in C'_o} \sum_{c'_e \in C'_e} \mu(c'_o, c'_e)$$

*where* $\mu : C \times C \to \mathbb{N}_0$. *The* ***simple component extraction problem*** *is a specialization of the general component extraction problem using objective function* $\overline{m}$ *instead of* $m$.

This specialized problem is equivalent to the min-cut problem in graphs [3] as we will show in the following. A cut in a graph splits the nodes of the graph into two sets. The min-cut problem is to find a minimal cut that separates a predefined source node from a predefined sink node. This problem is equivalent to find the maximal flow between source and sink (max-flow problem).

We start at an initial component partition on a class dependency graph (Definition 2). In contrast to the code dependency graph (Definition 1), information about the edge types and multiplicity is lost. We could derive edge strengths ($\in \mathbb{N}_0$) from the code dependency graph and assign them as edge weights in the aggregated code dependency graph. For instance, we may sum up edges according to their multiplicity and type. But it is not necessary to define a particular strategy to compute the edge weights—an arbitrary metric is applicable. The result of this first step is a graph like sketched in Figure 1 (a).

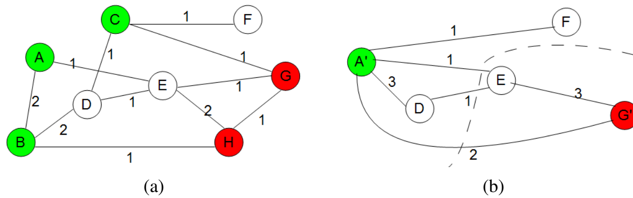


Fig. 1. Example for transforming the class dependency graph to apply the min-cut problem.

But we cannot directly apply the min-cut problem to the retrieved graph. Min-cut needs exactly one source node and exactly one sink node. But, in the terminology of network flows, we have a set of source nodes as well as a set of sink nodes—the sets of key entities. We thus aggregate the key entities $C_o$ to a single node $c_o$, and $C_e$ to $c_e$ analogously. This aggregation of nodes also requires to join edges and to sum up their weights like sketched in Figure 1 (b). Minimizing the cut between $c_o$ and $c_e$ in the resulting graph $\hat{G} = (\hat{C}, \hat{E}_C)$ with edge weights $\mu$ solves the simple component extraction problem because $\overline{m}$ is equal to the capacity of the cut, which sums up the weights of all cut edges.

The Ford-Fulkerson algorithm solves the min-cut problem in polynomial time, more precisely, in $O(|\hat{E}_C| f^*)$ where $f^*$ is the capacity of the minimal cut [3].

## III. A Contract on Dependencies

The component extraction process generates two sets: the set of extracted classes and the set of classes that remain in the original part. There are still dependencies between these two sets, though minimized. As a result, developing either of the parts might still interfere with the other part.

A first step to tackle the problem of the interdependencies is to state them. When these interdependencies are explicit, the developers know which parts they are allowed change independently and when they have to coordinate changes with the other development team. The recorded interdependencies form a kind of contract between both teams, a document that can only be changed in agreement and that guarantees certain services.

A second step is, not only to state the dependencies in a document, but to adapt the source code with respect to them. In particular, the goal is to generate a set of interfaces that defines and handles all interdependencies. Changes of the contract would be explicitly related to changing such an interface.

### A. Type, Method Call, and Field Access

The *Dependency Inversion Principle*, as introduced by Martin [15], postulates that “details should depend on abstractions”. In other words, concrete classes should not reference concrete classes but abstractions of these classes. To this end, an additional interface of the called class is introduced and calls are redirected through this interface.

Following the idea and technique of the *Dependency Inversion Principle*, we redirect the interdependencies between the extracted component and the remaining software through interfaces. For a class affected by interdependencies, an interface is automatically generated as follows (Figure 2):

- A blank interface is generated, the class implements this interface.
- Each interdependently called method is added to the interface, the call is redirected over the interface.
- For an interdependently accessed field, getter and setter methods are generated and also added to the interface and redirected.

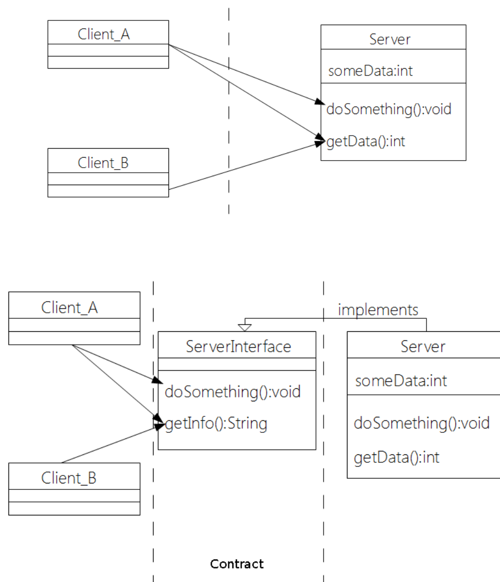


Fig. 2. Delegating interdependencies through a new interface in the contract.

Since two-way communication between the components is possible, we generate interfaces for the classes of the extracted component as well as of the original component.

Applying this approach to interfaces is straightforward: For interdependent method calls targeting an interface, another interface redirecting these calls is generated as part of the contract. At first, it may sound strange to add another layer of abstraction for interfaces. However, in many cases, only a part of the interface is intertwined with the other component via interdependencies. Adding the whole interface would unnecessarily swell the contract. In contrast, creating additional tailored interfaces keeps the contract reduced to the minimum.

These new interfaces cover interdependent type dependencies, method calls and field accesses—three of the five dependencies presented in Table I. All necessary code changes are simple because they can be implemented by refactorings—automatable code change patterns that preserve the semantics of the source code. *Encapsulate Field* and *Extract Interface* refactorings [5] are sufficient.

The *Interface Segregation Principle*, also put forward by Martin [15], suggests that “clients should not be forced to depend on methods that they do not use”. It thus propagates using a tailored interface for every client. If each class represents a client, this leads to a large set of interfaces with multiple redundancies—especially if applied automatically like necessary for our approach. But when extending the definition of *client* from classes to components, our approach already fulfills this principle: It generates tailored interfaces for each client (i.e., component).

### B. Constructor Call

Constructor calls are a special form of method calls. In contrast to method calls, they cannot be redirected over interfaces—interfaces do not allow constructors. We thus need a different mechanism to add constructor specifications to the contract.

Factory methods are a simple solution for this problem. For each interdependently called constructor, we create a static factory method that just redirects the constructor call. We gather all these factory methods in a single factory class called `ContractFactory` and add this class to the contract.

This approach is, however, not as clean as the redirection by interfaces: The implementing class has to be defined in the factory method. Replacing the implementation of the class always requires mutual agreement. A further indirection, however, would only shift this problem—some constructor must be explicitly called in the contract.

### C. Inheritance

Inheritance dependencies crossing component borders are much more difficult to indirect than method calls. While a calling class only has very limited access to the callee, an extending class could be heavily intertwined with the extended class: Dependencies may exist to fields, to methods, or to constructors of the superclass, or the class may overwrite methods and fields.

Nevertheless, with some effort, we are able to indirect an inheritance dependency over an interface. For instance, class `B` extends class `A` as depicted in Figure 3. The first step would be to extract an interface `IA` from class `A` that class `B` also implements. Additionally, as the second step, class `B` is aggregating class `A` through the interface `IA`. The aggregation allows to redirect calls of the superclass to class `A`.

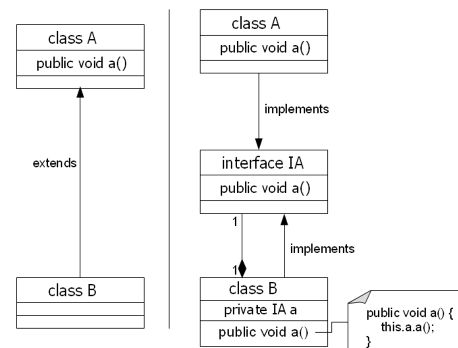


Fig. 3. Replacing inheritance by delegation.

This approach might work for some examples, but cannot be applied as a general method. We have to guarantee, for example, that an object of class `B` really aggregates an object of class `A` because otherwise it might aggregate another object of class `B` which would lead to infinite redirection (`IA a = new B();`). Other problems are the initial instantiation of A in class B or necessary extensions of visibility.

In conclusion, resolving an inheritance dependency should not be a default operation. The user has to be aware that inheritance is a strong kind of coupling between classes. If possible, such a dependency should not be parted by component extraction—superclasses that are supposed to be

very stable might form an exception, for instance, members of a software library.

### D. Contract Contents

With the help of the presented refactoring steps, we encapsulate interdependencies between the components in a contract. All dependency types but inheritance can be managed automatically in this way. Finally, the contract includes

- a set of interfaces that encapsulates classes and interface and
- a factory class that redirects constructor calls.

All newly built interfaces are part of the contract. The extracted component communicates with the remaining system only through these interfaces—as discussed, inheritance may form an exception. Since interfaces and factory methods are only generated as far as necessary to cover all interdependencies, this contract is minimal. It exactly represents the parts that are necessary to coordinate the development.

### E. Reconsidering Component Extraction

Some interdependencies might only slightly affect an independent development of the components, while others constitute a significant hurdle. As we have seen, for example, inheritance relationships between components are difficult to handle. The simplified component extraction problem allows to take such differences into account by assigning different edge weights to different dependency types.

But as the following example shows, this approach cannot directly take the necessary refactorings into account. Assume a method is called by a method from the other component. If we now add another method calling from the other component, in terms of edge weights, this would be twice as bad than just one method calling. But we do not need to add anything to the contract. Thus, the quality of a partition might depend more on the size of contract and the complexity of the necessary refactorings than on the capacity of the cut.

Just changing the edge weight metric does not solve the problem because the structure of the contract depends on the structure of the whole partition. We would need adaptive edge weights that change with the layout of the partition. This, however, violates the preconditions of the min-cut problem.

But it is possible to implement a more elaborate approach based on the general component extraction problem. An objective function may take the complexity of the contract into account. To this end, we propose to assign fixed costs $\gamma(r)$ to each refactoring $r$. This approach already looks ahead at the later on necessary refactorings.

Finally, we suggest the following objective function for the general component extraction problem:

$$m(s) = \lambda_1 \cdot \sum_{c'_o \in C'_o} \sum_{c'_e \in C'_e} \mu(c'_o, c'_e) + \lambda_2 \cdot \sum_{r \in R} \gamma(r)$$

where $s = (C'_o, C'_e)$ is a valid component partition, $\mu$ an edge weight, $R$ the set of all necessary refactorings for the particular partition, and $\lambda_1, \lambda_2 \in \mathbb{R}_0^+$ weight parameters. This function includes the complexity of the refactorings as well as the capacity of the cut. The weight parameters $\lambda_1, \lambda_2$ allow to balance these concepts individually.

## IV. Interactive Tool and Visualization

We developed a tool called *ComponentExtractor* as a prototype implementation of the component extraction approach introduced in Section II and Section III. The tool is a stand-alone Java application that processes Java software projects based on their byte code. It visualizes the source code entities and their dependencies in a high-level visual representation. The core of the tool is an implementation of the component extraction process in different variants. The visualization provides a user interface to interactively control the extraction process as well as to present the results of the automatic extraction. Finally, an export of the proposed source code refactorings to the Eclipse IDE integrates our tool into the development environment.

### A. Dependency Extraction

The first step toward a tool support for component extraction is the data acquisition—extracting the different kinds of source code dependencies as listed in Table I. We use the BCEL library (Byte Code Engineering Language) [4], which works on Java byte code, to extract all necessary information. Please note that, due to late binding in Java, the results are an overestimation of the actual dependencies. Nevertheless, since we are only interested in transitions of the static source code, they are sufficiently exact for our purposes. The result of this data acquisition phase is a code dependency graph as introduced in Definition 1.

### B. Component Extraction Algorithms

Our specification of the component extraction problem (Problem 1 and 2) does not exactly lay down the implementation of the component extraction algorithm. First of all, some parameters need to be defined.

To apply the simple component extraction problem (Problem 2), we have to specify edge weights $\mu$. We use the following algorithm to compute these weights: Each dependency type has a specific weight, which is defined by the user. For two entities $c_1, c_2$, the edge weight $\mu(c_1, c_2)$ is the sum of these type weights over all edges between $c_1$ and $c_2$ in the class dependency graph.

We implemented an instance of the general component extraction problem (Problem 1) according to the propositions in Section III-E. Additional to the edge weights, which also play a certain role in this variant, the user is able to set up the refactoring costs for each elementary refactoring. The balance between the different aspects of optimization, defined by the parameters $\lambda_1, \lambda_2$, is also customizable.

Furthermore, the specification allows to select an optimization algorithm. Our tool *ComponentExtractor* includes different implementations that solve the component extraction problem.

**Min-Cut Algorithm** The algorithm addresses the simple component extraction problem (Problem 2) by solving the

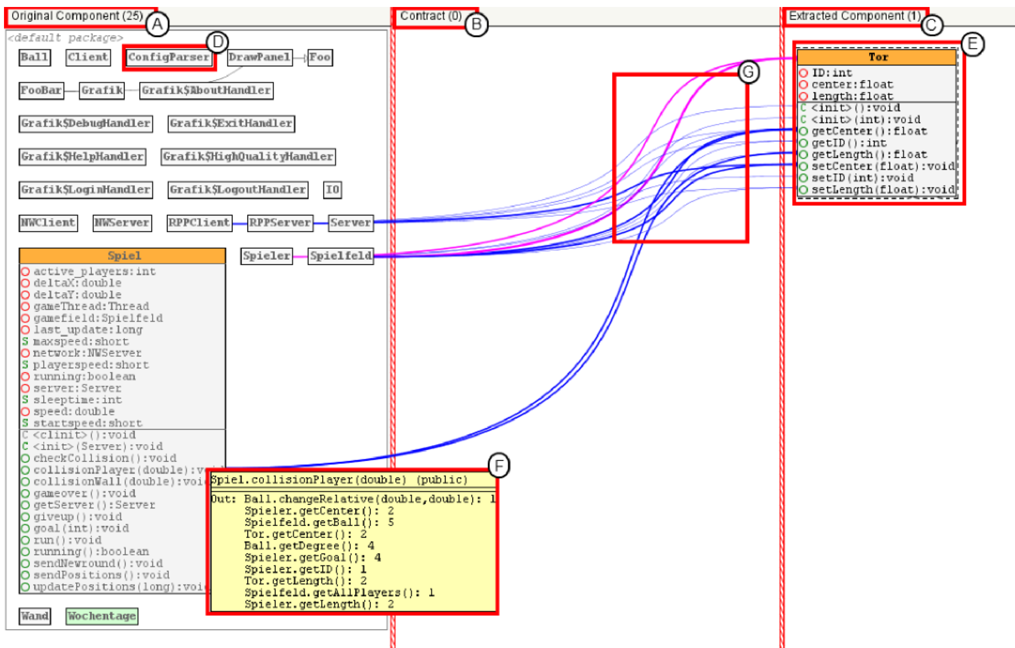


Fig. 4. Visualization of the code dependency graph in *ComponentExtractor*.

min-cut problem with the Ford-Fulkerson algorithm as described in Section II-C.

**Exhaustive Algorithm** The algorithm performs an exhaustive search to find a global optimum for the general component extraction problem (Problem 1). This naive implementation does only work for very small datasets.

**Hill Climbing Algorithm** This hill climbing optimization algorithm provides a more efficient approach to the general component extraction problem (Problem 1). The starting point is given by the current component partition as visualized in the tool. The algorithm, however, only yields approximations of the optimal solution.

**Library Extraction** This algorithm addresses the extraction of a library, a specialization of component extraction that we did not cover in Section II. The algorithm just computes the transitive hull on the class dependency graph starting from the selected key entities (i.e., the set of all reachable entities) and extracts this hull. This approach could be useful for extracting passive components—components that do not access the original component.

These algorithms are also applicable in combination. An efficient strategy might be to first run the fast min-cut algorithm and then use the result as the starting point of the hill climbing algorithm, which additionally takes the refactoring costs into account. The hill climbing algorithm might not yield as good results when starting from default.

### *C. Visualization*

The user interface of *ComponentExtractor* mainly consists of a visual representation of the code dependency graph. We use a node-link diagram for this visualization. In particular, it is a simplified variant of UML class diagrams. Figure 4 gives a first impression of the visualization.

The view is split into three main parts: the original component (A), the contract (B), and the extracted component (C). While in the initial state, all classes and interfaces belong to the original component, using the tool some of these entities will move to the extracted component or the contract—either by user's activity or by the automatic component extraction.

Like in UML, boxes represent the classes and interfaces. Since it is very important to use the screen space efficiently, these boxes are collapsed by default (D). Clicking on a node enlarges the box and provides attribute and method information (E). Tooltips show additional details on demand (F).

Links between the boxes depict the dependencies among the entities. But unlike UML, we use colors to discern dependency types instead of more space consuming markers and stroke types. Displaying all dependencies of the graph at once produces cluttered diagrams with many overlapping edges. Hence, the initial view just shows inheritance relationships—probably the most important dependency type. The dependencies of other types appear when they become the center of focus, for example, when they form interdependencies between extracted and original component (G).

### *D. Component Extraction Process*

In the following we describe how our tool implements component extraction as an interactive process. Figure 5 provides an overview.

First of all, the users are able to choose an initial component partition (Definition 3): They firmly assign classes and interfaces either to the original or the extracted component. The border color of the boxes indicates these assignments. All remaining entities are free to be moved by the algorithm.

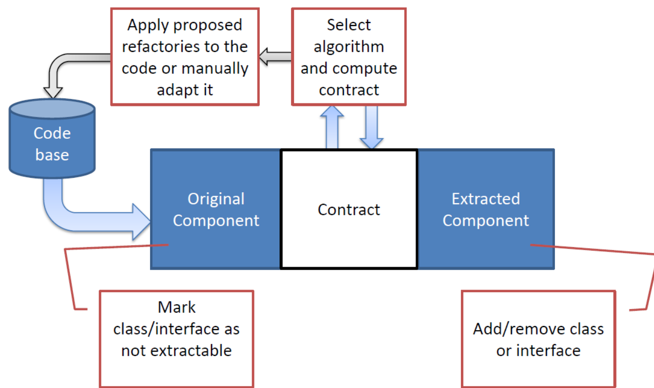


Fig. 5. Overview on the interactive component extraction process.

Before starting the component extraction, at least one entity has to be assigned as a key entity of the extracted component. Then, component extraction can be applied. The algorithm might move free entities from the original component to the extracted component, or vice versa. Moreover, if activated, it probably creates some new interface and factory methods, which form the contract.

The visualization presents these results to the user as illustrated in Figure 6. In this example, the user selected `Grafik` to be extracted while the `RPPClient`, `RPPServer`, `Server` class had to remain in the original component. The component extraction algorithm proposes to extract six further classes. The contract redirects all interdependencies. It consists of five interfaces and one factory class.

The user is able to interactively modify the components. For example, it might turn out, in contrast to the initial choice, that some entities should not be moved. Or the identified key entities of the extracted component were not enough to extract a meaningful component. After refining the preconditions, reapplying the algorithm probably provides more suitable results. This process of changing inputs and checking results would typically be repeated several times until the identified components are satisfying.

Finally, the users might adapt some details. For instance, they want to undo some of the automatically proposed refactorings, to manually add another interface to the contract, or to move a class from one component to the other. Our tool *ComponentExtractor* provides the necessary features to customize the partition proposed by the algorithm.

These interactions are very important for the practical applicability of our approach. We do not claim that our component extraction approach provides optimal results in one step. But it proposes component structures that help the users to design suitable components with respect to their individual needs. We think component extraction should be more an interactive and iterative process than a fully automatic method.

### E. Export Refactorings

Finally, we need to reintegrate our results into the development process: The source code has to be changed according to the proposed refactorings. A simple way to implement this task is to use the refactoring capabilities of an Integrated Development Environment (IDE). Eclipse[1], a popular IDE in the Java community, provides a refactoring description language based on the XML data format of the *'CatchUp!'* refactoring recording tool [8]. We use this format to import the refactorings into Eclipse and to apply the refactorings to the code.

At present, still some code changes have to be performed manually because, for instance, introducing new classes as required for the factory class cannot be directly described as an Eclipse refactoring.

## V. User Study

We conducted a small user study to test the general applicability and usefulness of the proposed component extraction process. In particular, we evaluated the presented approach in a thinking aloud user study [12] with three experienced developers. We prefer such an experimental design, where the users are asked to speak out their thoughts while interacting with the tool under examination, for several reasons: It is a light-weight design that provides a broad picture of the examined approach. Furthermore, the explorative character of such a study allows to work on real-world examples.

We are aware that a thinking aloud study is only a first step toward an exhaustive evaluation of our approach. Such a study does not provide any statistical evidence, but only subjective ratings. Nonetheless, due to its explorative character, it could be better suited for a first evaluation than a more narrow quantitative study.

### A. Experimental Design

We invited three postgraduate computer scientists to participate in the study. They were experienced software developers with programming experience of 9 to 12 years (4 to 8 years in Java). We introduced the task of component extraction and asked them to think about a software project where they want to apply this task. Each participant chose a project, resulting in the following list of three systems:

**P1** web-based visualization of large bibliographic networks, research project, 139 classes.

**P2** poker game, student project, 72 classes.

**P3** code clone detection tool, research project, 123 classes.

In each case the participant was the exclusive developer of the software system. While P1 wanted to extract the database connection and P2 the GUI and AI component, P3 preferred to check whether our tool confirms the already existing components.

The experiment started with a brief oral tutorial on component extraction. We varied the intensity of this tutorial: The experimenter explained the task and introduced the user interface of *ComponentExtractor* for participants P1 and P2. To test how intuitive our tool works, P3 only got an oral description of the task.

[1] http://www.eclipse.org

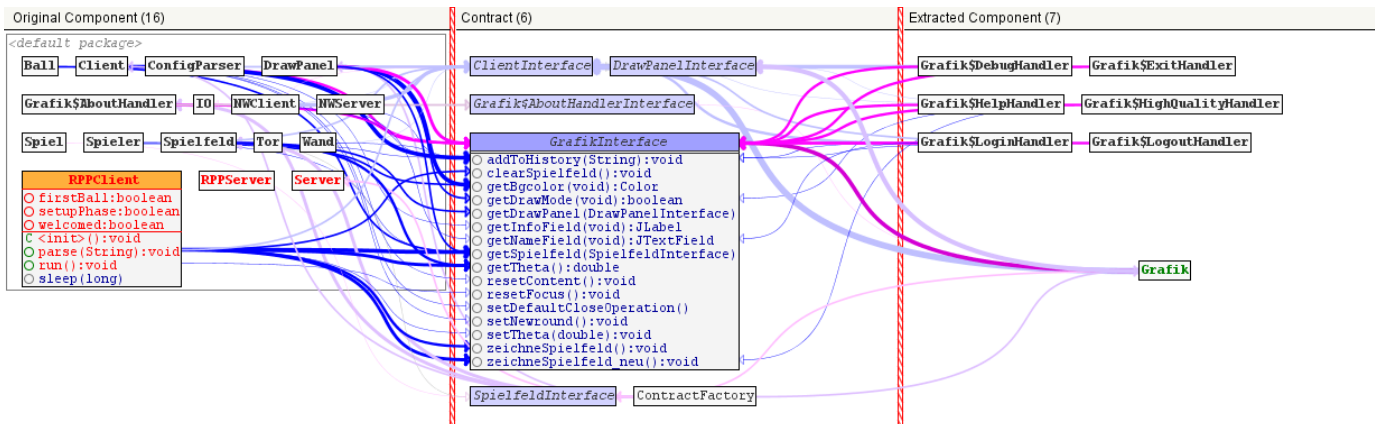


Fig. 6. An example of the visualization in *ComponentExtractor* after component extraction.

The thinking aloud test itself took 50 to 60 minutes. Only the participant and the experimenter were in the room. The experimenter briefly explained the thinking aloud method and, if necessary, again encouraged the participants to speak out their thoughts loudly during the test. The test was taped on video for later analysis.

In each case, the test began with a small introductory example, a student project consisting of 23 classes. The participants were asked to explore the software project and to try to initially extract a component they consider useful. This first stage intended to assess intuitiveness and usability of the visualization. Here, they were asked to extract the component they had chosen beforehand. This second stage aimed for checking the applicability of the proposed component extraction approach.

### B. Usability Results

A thinking aloud study often yields a great deal of detailed ideas on how to enhance usability. From our study, we retrieved concrete usability problems like particular ambiguous labels, missing help texts, mistakable and cumbersome interactions. But the study also indicated some general usability aspects of component extraction:

**Intuitiveness largely depends on familiarity.** As we used UML-like symbols, the participants immediately understood the rough outlines of the visualization. Even P3, who did not get any description of the tool, was able to use most of its features. Problems occurred when the visualization deviated from UML notation. For instance, we color coded visibility modifiers instead of using the symbols +, #, −, and ˜.

**The representation of dependencies significantly influences usability.** We invested some effort to create a readable layout, for instance, by only showing focused dependencies or by summarizing dependencies. Nevertheless, participants complain about missing overview, difficulties in following dependencies, or hard to perceive directions of dependencies.

### C. Component Extraction Results

The experiments also provided some insights into the process of component extraction. In general, the first run of extracting a component was not satisfying. All participants noticed dependencies that they had not expected. They expressed the wish to change their source code and to eliminate those dependencies. Since this was not possible in the study, they had to react otherwise. For example, they tried to adapt the parameters of the objective function to find another solution.

In detail, P1 was able to identify a component but was not completely satisfied because two obsolete classes interfered with the extraction. P2 realized that the classes of his program were too strongly connected to extract a meaningful component. Finally, P3 could mainly confirm the already existing components with the algorithm and furthermore noticed some unintended dependencies.

Our observations lead to the following conclusions:

**Component extraction is an iterative process.** The result of the component extraction process might reveal design weaknesses in the examined code. Iteratively revising the code and reapplying component extraction would be necessary.

**Component extraction supports program comprehension.** Every participant of the study gained new and unexpected insights into the source code of their system. The interplay between component extraction and the visual representation of the system could identify design problems like unintended dependencies or badly structured classes.

## VI. Related Work

Automatic identification and extraction of components in terms of component-based software engineering is closely related to our approach. Washizaki and Fukazawa [22] start at a key class and trace all dependencies to find the boundary of the new component—similar to our library extraction variant based on computing the transitive hull (Section IV-B).

A different application, however employing similar algorithms, is tailoring libraries for mobile systems (also called *application extraction*): The application should be reduced in size—which is crucial for the mobile devices—without changing its behavior. Tip et al. [21] present an approach that, among other things, looks at the call graph of the application and cuts all parts that will not be needed. This again is equivalent to computing the transitive hull.

On program statement level, extracting parts of reusable software like functions could be done by program slicing [11]. A program slice describes the set of program statements that affects a set of variables at a particular point of interest. Slicing was also applied on architecture description languages [23]. This allows extracting parts of the software architecture for reuse, but needs a formal description of the software architecture. Usually, much of the intelligence of a slicing approach is in constructing an appropriate graph structure, not in a graph algorithm. Our approach focuses, in contrast, more on the graph algorithm that cuts the graph into two parts. Program slicing might add to our approach when changes below class level would be necessary.

Some approaches focus more on the identification of components than on the extraction. Mockus and Weiss [16] try to find parts of the software as component candidates that were frequently changed together, but were developed at different sites. Extracting those parts may reduce the communication overhead. Baniassad and Murphy [1] present an approach where the user selects a set of lines of code that will be virtually transformed into a new component. Querying the changed structure provides a preview on the new component. Switching between selecting and querying might support the user to find appropriate candidates for component extraction.

In contrast to the previously described works, approaches based on clustering [14] address a more holistic perspective: They intend to classify every part of the system, not only a smaller part of particular interest. A recent study by Glorie et al. [7], however, raises the question how far these holistic approaches are really applicable in practice. They tried different software clustering and concept analysis techniques on splitting large software systems and came to the conclusion that the algorithm should support initial partitions and should not be allowed to tear these partitions apart. Our component extraction approach fulfills these two requirements—not on clustering the whole software, but on a local level. Other works also stress the importance of interactivity in similar scenarios: Koschke [10] presents an incremental and interactive approach to software clustering. Müller et al. [17] argue that subsystem identification should never be fully automatic.

From the algorithmic point of view, our approach is related to min-cut clustering [9]. This approach clusters the nodes of a graph into $K$ sets by minimizing the cut between these sets. As in our work, the sum of the weights on edges that go between the sets forms the cut. The difference is, first, that we only consider two clusters (i.e., components), and second, that we start with key entities for both clusters.

A similar graph clustering algorithm has already been applied to code dependency graphs [13]. The objective function applied there does not only consider the capacity of the cut but also the cohesion among the entities in the components. Chiricota et al. [2] present an elaborate edge weight metric for software dependencies that indicates whether an edge is in the center of a component. They used this metric to fade out less important dependencies. Perhaps this edge weight metric can be profitably applied in our approach.

Visualizing code dependencies of software projects is part of software visualization and related to graph drawing. UML diagrams are the industry standard to visualize code entities and their dependencies. But there exist lots of alternatives with different focuses [6].

All in all, we are not aware of any approach that is tackling exactly the same problem as we discuss in this paper. Other component extraction and slicing approaches focus on other applications and use less sophisticated or extensible extraction processes. Existing holistic software clustering approaches do not allow to incrementally and interactively improve a software decomposition and shape a customized component.

## VII. Discussion

**Applicability** We expect component extraction to be useful for various software development tasks:

- A part of a software system is to be outsourced to another development team.
- A new team member starts working in a development team; a part of the source code is to be assigned to his responsibility.
- A part of an old software project is to be reused in another project.
- An existing modularization of a project is to be checked against design shortcomings.
- A software project is to be re-modularized.

A well-organized software project already consists of component-like structures—parts of the software that have a high internal cohesion and a low external coupling. But like code refactorings are necessary to react to evolving code, component extraction—as a kind of high-level architectural refactoring—is necessary to react to evolving architectures. Moreover, considering all possible extensions of a software system in the initial design is known as the 'bad smell' of *speculative generality* [5].

Nevertheless, there are limiting factors for our approach. Concerns intertwined in the class structure are a major problem. Extracting these would require changes below class level. As we learned from the user study, this problem occurs, for instance, when there exist too many interdependencies between the classes. Similarly, the component extraction approach is not suitable for the application of aspect mining, where the concerns are distributed across classes per definition.

Different applications of component extraction might require different strategies. Since we described the component extraction problem as a general optimization problem, it is open for adaptations. The particular definition of the objective function expresses these adaptations. For instance, we take refactoring costs into account in our implementation. The simplified component extraction problem somewhat narrows the scope. But still the graph structure and edge weights may model different types of code dependencies.

**Scalability** The user study shows that our prototype tool is working for projects up to 139 classes—small projects.

On the one hand, the scalability of component extraction is defined by the scalability of the tool and its visualization.

We observed that the visualization nearly reached its capacity. We believe our visual representation could be enhanced by focusing, zooming, and filtering techniques to handle at least mid-size projects.

On the other hand, we have to discuss the scalability of the extraction approach itself. The Ford-Fulkerson algorithm solves the simple component extraction problem in $O(|\hat{E}|f^*)$ where $f^*$ is the capacity of the minimal cut. This algorithm is able to also handle large projects: We applied the algorithm to the Azureus project (now called Vuze) including over 3000 classes and got the results within a few seconds computation time. In contrast, the general component extraction problem is more difficult. We already had to employ a heuristic to solve the problem for smaller projects in feasible computation time. The current implementation of the hill climbing optimization is too slow for large projects like Azureus. But we assume that a more elaborate implementation or optimization approach could solve the problem much faster.

**Future Work** As we learned from the user study, it is very important to seamlessly integrate a component extraction tool into the development environment. With exporting the refactorings to Eclipse we have already done a first step toward this goal. But further steps would be necessary, like integrating a smooth and easy switch from source code to component extraction and vice versa.

From the viewpoint of the component extraction approach, we proposed two explicit optimization criteria, namely, min-cut and refactoring costs. These criteria were derived from theoretical considerations, but not directly assessed by an evaluation. The next major step would be to evaluate these and maybe further criteria in a thorough quantitative study.

## VIII. Conclusion

We introduced component extraction as an important problem in software maintenance. It applies when existing code needs to be partitioned, for instance, to integrate new team members, to facilitate software reuse, or to enable outsourcing. We formally described the extraction process as a general optimization problem. In a simplified version, the problem is equivalent to the min-cut problem in graphs. The Ford-Fulkerson algorithm thus solves this simplified problem efficiently.

After the extraction process, a set of interfaces and factory methods encapsulates the still existing interdependencies among the extracted component and the remaining part of the software. This set forms a kind of contract between the developers of the two parts. We derive the content of the contract automatically by applying code refactorings.

Our tool *ComponentExtractor* implements the component extraction approach and provides a visualization of the code dependencies. The tool facilitates to interactively control the component extraction process and visually represents its results. We evaluated the tool in a small thinking aloud user study. The study showed the applicability of the approach and its additional capabilities concerning program comprehension. Lessons learned were that component extraction should be an interactive and iterative process that needs to be seamlessly integrated into the development process.

## References


[1] E. L. A. Baniassad and G. C. Murphy, "Conceptual module querying for software reengineering," in *ICSE '98: Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, 1998, pp. 64–73.

[2] Y. Chiricota, F. Jourdan, and G. Melançon, "Software components capture using graph clustering," in *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*. IEEE Computer Society, 2003.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[4] M. Dahm, "Byte code engineering," in *Java-Informations-Tage*, 1999, pp. 267–277.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 1999.

[6] Y. Ghanam and S. Carpendale, "A survey paper on software architecture visualization," University of Calgary, Tech. Rep., 2008.

[7] M. Glorie, A. Zaidman, A. van Deursen, and L. Hofland, "Splitting a large software repository for easing future software evolution - an industrial experience report," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 113–141, 2009.

[8] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*. ACM, 2005, pp. 274–283.

[9] E. L. Johnson, A. Mehrotra, and G. L. Nemhauser, "Min-cut clustering," *Mathematical Programming*, vol. 62, no. 1, pp. 133–151, 1993.

[10] R. Koschke, "An incremental semi-automatic method for component recovery," in *WCRE '99: Proceedings Sixth Working Conference on Reverse Engineering*, 1999, pp. 256–267.

[11] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph-based program slicing," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 246–259, 1997.

[12] C. Lewis and J. Rieman, *Task-Centered User Interface Design: A Practical Introduction*. University of Colorado, Boulder, 1993.

[13] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*. IEEE Computer Society, 1998, pp. 45–52.

[14] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. on Software Engineering*, vol. 33, no. 11, pp. 759–780, 2007.

[15] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, 1st ed. Prentice Hall, 2002.

[16] A. Mockus and D. M. Weiss, "Globalization by chunking: A quantitative approach," *IEEE Software*, vol. 18, no. 2, pp. 30–37, 2001.

[17] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A reverse engineering approach to subsystem structure identification," *Journal of Software Maintenance: Research and Practice*, vol. 5(4), pp. 181–204, 1993.

[18] K. Sugiyama and K. Misue, "Visualization of Structural Information: Automatic Drawing of Compound Digraphs," *IEEE Trans. on Systems, Man and Cybernetics*, vol. 21, no. 4, pp. 876–892, 1991.

[19] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley Professional, 2002.

[20] C. Szyperski and C. Pfister, "Workshop on component-oriented programming, summary," in *Special Issues in Object-Oriented Programming - ECOOP '96 Workshop Reader*, M. Mühlhäuser, Ed. dpunkt Verlag, 1997.

[21] F. Tip, P. F. Sweeney, and C. Laffra, "Extracting library-based Java applications," *Commun. ACM*, vol. 46, no. 8, pp. 35–40, 2003.

[22] H. Washizaki and Y. Fukazawa, "Automated extract component refactoring," in *Extreme Programming and Agile Processes in Software Engineering*, 2003, p. 1016.

[23] J. Zhao, "A slicing-based approach to extracting reusable software architectures," in *CSMR '00: Proceedings of the Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2000.