Flexible Design Pattern Detection Based on Feature Types

Ghulam Rasool
COMSATS Institute of Information Technology
Lahore, Pakistan
e-mail: grasool@ciitlahore.edu.pk

Patrick Mäder
Institute for Systems Engineering and Automation (SEA)
Johannes Kepler University, Linz, Austria
e-mail: patrick.maeder@jku.at

***Abstract*—Accurately recovered design patterns support development related tasks like program comprehension and re-engineering. Researchers proposed a variety of recognition approaches already. Though, much progress was made, there is still a lack of accuracy and flexibility in recognition. A major problem is the large variety of variants for implementing the same pattern. Furthermore, the integration of multiple search techniques is required to provide more accurate and effective pattern detection. In this paper, we propose variable pattern definitions composed of reusable feature types. Each feature type is assigned to one of multiple search techniques that is best fitting for its detection. A prototype implementation was applied to three open source applications. For each system a baseline was determined and used for comparison with the results of previous techniques. We reached very good results with an improved pattern catalog, but also demonstrated the necessity for customizations on new inspected systems. These results demonstrate the importance of customizable pattern definitions and multiple search techniques in order to overcome accuracy and flexibility issues of previous approaches.**

***Keywords*-Design Pattern Recognition; Pattern Detection; Pattern Definition; Feature-Based Pattern Recognition; Program Comprehension; Code Analysis; Regular Expressions;**

I. Introduction

Design patterns are meant to solve recurring problems in software system design and so to improve reusability, maintainability, comprehensibility, evolvability, and robustness of applications [1]. Design patterns are not per se beneficial, but do have advantages if chosen wisely [2]. The recovery of design patterns from applications started 14 years ago with the publication of Gamma et al.'s [1] prominent book and the first attempt for the recovery of these patterns by Krämer et al. [3]. The fact that the later publication is cited more than 235 times highlights the relevance of the topic. Though, much work has been done in the area of pattern recognition since the early days, some key questions still remain unanswered. In the following paragraphs, we will emphasize on four of the main issues and how our approach contributes to their solution.

First, there is still no consensus about which information realizes the existence of patterns in source code. A fundamental demand for the pattern research community is the development of consistent definitions of design patterns across the community [4]. Until that problem is solved, comparing the results of pattern recognition techniques is a challenging problem as we will demonstrate in that paper. Authors claim correctness of their results on the same problem, though a deep analysis shows differing recovered patterns. The cause of disparity in most cases are different interpretations of patterns and their differing implementation variants. Though, a consistent definition of patterns is required and will help to mitigate the problem to some extend, such a definition cannot solve the problem completely. Programming languages will offer new features, programmers will find new pattern variances and even new patterns. A solution to this problem should be an extensible and customizable set of pattern definitions containing all the "standard" patterns while leaving room for new developments.

Second, the comparison of approaches is extremely difficult due to the unavailability of benchmarks and trusted baselines for them. Similar to the creation of consistent pattern definitions, the pattern research community is required to provide agreed benchmark examples with known baselines to cope with that problem. Creating a baseline for a project requires the manual inspection of the source code, which is especially for bigger systems a very challenging and expensive task. To cope with that problem we introduce trusted baselines by comparing and analyzing the results of previous approaches. These baselines and the results of our approach will be published as an attempt to overcome the problem of missing benchmarks.

Third, pattern detection approaches are usually language dependent and support in the best case only a small number of programming languages. The underlying problem here, apart from the different syntax of different programming languages is that the features for the implementation of patterns vary across languages and so require a successful approach to handle language dependent variants of patterns. Our approach is currently capable to detect patterns within C, C++, C#, and Java source code and it can easily be extended for other languages. Furthermore, we provide a customizable and extensible catalog of patterns and their variants, which facilitates the recognition in all current and in future programming languages.

Fourth, many previous approaches have only been demonstrated in extracting a subset of the 23 so-called "Gang of Four" (GoF) design patterns [1]. The issue here is that the recognition of these patterns varies in complexity. While some patterns are easily identifiable due to their unique structure, the task becomes more challenging for others. Our approach to that problem is the separation of a pattern definition into recurring features. These features are structural, relational, and behavioral parts that make up a pattern. A feature type will be detected in the source code with a search technique that is most fitting for it's characteristics. That means that we apply different technologies for detecting different parts of a pattern in order to overcome accuracy problems. We evaluated our approach with three applications and all 23 GoF patterns.

In conclusion, the contribution of our work is a thoroughly evaluated pattern detection technique based on flexible and extensible pattern definitions and search techniques.

The paper is organized as follows, Section II discusses the current state of art of work that has resemblance with our approach. Section III refers to the specification of patterns and their variants as a basis for our approach. Section IV discusses the actual approach used for the detection of design patterns. The prototyping tool used to realize the concept of our approach is presented in Section V. Section VI describes the experimental set-up chosen to evaluate our approach, shows results extracted by the approach and discusses them. Finally, Section VII concludes the whole approach.

II. Related Work

As discussed in the introduction, early research on design pattern detection dates back almost 15 years. A number of approaches, applying different recognition techniques to the problem has been developed and studied. We cannot introduce all of these approaches here and it is not necessary as there are good reviews about the characteristics of different techniques used in the field of design pattern recovery [5], [6]. The authors of the first paper [5] presented a deep insight on work starting from the infancy of design pattern recovery to the publication of this article.

The approach presented in [7] uses static program analysis for the detection of patterns from source code of different applications. The authors represent examined software and design patterns as graphs and use matrices to represent relationships between source code artifacts. The proposed methodology uses similarity algorithms to cluster hierarchies, which reduce the search space for pattern detection. The authors report 100% precision and recall on the examined examples. Though, precision and recall become suspicious, as upon comparison of our results with the reported ones, a number of reported true positives seem to be false positives. Other researchers [8], [9] also refer to disparities in the results of the discussed approach.

Guéhéneuc and Antoniol [8] present a multilayered semiautomatic approach for design pattern detection from Java source code. The approach uses static analysis to detect relationships. Dynamic analysis based on trace analysis techniques is used to compute exclusivity and life time relationships for aggregation and composition relationships. The authors performed experiments on open source systems and achieved an average precision of 34% for the 12 supported design motifs and report 100% recall. Authors of the same group presented an improved approach [10] using constraints programming, supplemented with numerical analysis in order to improve the performance of their previous approach. They performed experiments on 18 GoF patterns applying the previously used examples and additional ones. While performance was improved, accuracy did not change.

Costagliola et al. [11] present a visual language based pattern recovery approach along with different case studies. The approach extracted Adapter, Bridge, Composite, Decorator, and Proxy with 100% precision and recall from small systems. The same group proposed a two-phase approach [12]. In phase 1, design pattern instances are recovered at a coarse grained level by parsing the design structure of an inspected system. Identified patterns are then, in phase 2, validated by a fine-grained source code analysis. The authors performed experiments on open source systems extracting structural design patterns. They improved and extended their approach in [6] and used the same case studies. The extended approach first constructs a UML class diagram represented in SVG format. The class diagram is then mapped with a visual language grammar to detect different patterns. Comparing the results of both approaches, we noticed that the extracted pattern are the same in both approaches, so it is hard to conclude about an improvement. We will refer to the results of that approach, when discussing the evaluation of our approach.

Dong et al. [9] present a design pattern recovery approach based on the use of matrices and weights. The DP-Miner toolkit is used to build a matrix of inspected source code. All classes in the system correspond to rows and columns, and relationships between a pair of classes to a value of the corresponding cell in the matrix. Information on design patterns is encoded as matrix and weights. The discovery of design patterns is performed by matching matrices and weights with arithmetic computations. The authors performed experiments on different case studies and aim to recover Adapter, Bridge, Strategy and Composite design patterns. Precision values are only provided for JHotDraw and recall has not been computed.

Shi and Olsson [13] extract program intent in order to recognize patterns from Java source code. The technique has reclassified all GoF patterns in the context of reverse engineering. Their tool recovers all GoF patterns from source code with the exception of Prototype, Iterator, and Builder. The authors do not provide precision and recall for their extracted results. While comparing our results, we found a large number of false positives and false negatives among the results provided by their tool.

Concluding from the discussion of related work, we found that most approaches are based on a single recognition technique. Furthermore, the approaches perform experiments either on Java or C++ projects and most of them not on all GoF patterns. Achieved results are debatable for some approaches. The various implementation variants for each pattern, hinder approaches with only a fixed set of pattern definitions from reaching high accuracy of pattern recovery. Within that paper we are introducing an approach that seeks to overcome these problems.

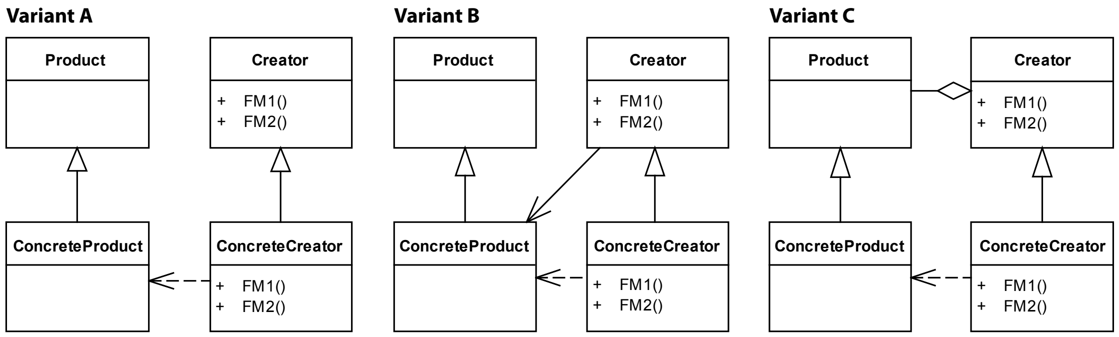

Fig. 1. Three implementation variants of the Factory Method pattern

III. Design Patterns as Collection of Features

Design patterns are typically described in terms of several aspects, such as intent, structure, behavior, and sample code. A design pattern usually consists of multiple elements forming its structure, of relationships between these elements and of a described behavior of the elements. A clear specification and description of a design pattern not only helps with its application in forward engineering, i.e. for its successful implementation, but it also plays a key role in the recovery of a pattern from existing source code.

The aim of our work was to develop a pattern specification technique that is precise enough to be automatically applicable by a tool, but also understandable and customizable for humans. While analyzing all the GoF patterns and their known variances, we found that they were composed of recurring sub-structures among different variants of the same pattern, but also among different patterns. We call these sub-structures: features of a pattern and were able to specify all GoF patterns from a collection of 44 different feature types. Examples of these feature types are:

- a class that is part of a pattern;
- a generalization relationship between two classes;
- an aggregation between two classes; and
- a method return type, which is equal to a certain class.

A. Example: Factory Method

This subsection introduces the factory method pattern in the original version proposed by the GoF and also shows two alternative ways for implementing the same pattern. We will use that example throughout the paper to discuss and illustrate our approach. The Factory Method pattern is used for creating a class of products without specifying the class creating those products. Recognizing the factory method pattern in source code is a challenging task due to its many different implementation variants. Radonjic and Corriveau [14] report about nine structural variants to implement that pattern.

Figure 1 shows three of these variants, all implementing the Factory Method pattern. On the left hand side of the figure (Variant A), the original version as described by the GoF is depicted. In that original version, the Factory Method pattern is a composition of the following features:

F1 A *ConcreteCreator* class
F2 A *Creator* class generalizing the *ConcreteCreator* class
F3 A *Creator* class and a *ConcreteCreator* class that have at least one common *FactoryMethod* method
F4 Every *FactoryMethod* method is creating a *ConcreteProduct* class and returns a *ConcreteProduct* of type *Product*
F5 A *ConcreteProduct* class realizing the *Product* interface
F6 A *ConcreteCreator* class not generalizing the *ConcreteProduct* class
F7 A *ConcreteCreator* class not generalizing the *Product* class

A second and a third variant of the pattern are depicted as Variants B and C in Figure 1. Both differ slightly from the original version and their specification consists of other and additional features.

IV. Pattern Recognition Approach

Based on an analysis of open problems in the field of pattern recognition (see Section I) and based on an intensive study of related work in the same field (see Section II), we propose an open and extensible detection approach that consists of two stages:

Stage 1 Creating semi-formal definitions for each pattern of interest and its variants based on common extensible feature types.
Stage 2 Detecting patterns by identifying its features, with different search technologies best fitting to the respective feature.

Within the first stage of the approach, a catalog of pattern specifications to be recognized is being created. The fundamental concept applied in this stage is the use of recurring feature types for the specification of patterns and their variants. These feature types have iteratively been derived while creating a catalog of all GoF patterns and their commonly known variants. Both collections, the set of feature types as well as the set of pattern definitions are extensible.

Within the second stage of the approach, the specified definitions are used to detect pattern instances in the source code. Figure 2 depicts the recognition process. We assume that the source code as well as an automatically created model of the source code are available for analysis (see right hand side of Figure 2). A recognition controller triggers all required analysis steps. It iterates through each feature defined within the definition of a pattern under inspection. Depending on the type of the current feature, it selects a search technology for detecting the feature either within the source model or the source code itself. Depending on whether a feature is matched or not, existing candidate patterns are extended or pruned. Both stages of the approach are described in depth in the following two subsections.

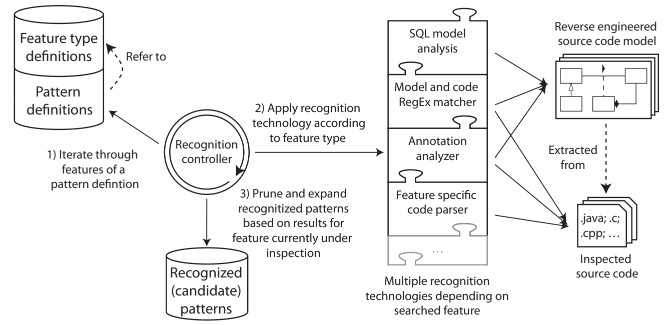
Feature type definitions
Refer to
Pattern definitions
1) Iterate through features of a pattern defintion
Recognition controller
2) Apply recognition technology according to feature type
3) Prune and expand recognitized patterns based on results for feature currently under inspection
Recognized (candidate) patterns
SQL model analysis
Model and code RegEx matcher
Annotation analyzer
Feature specific code parser
...
Multiple recognition technologies depending on searched feature
Reverse engineered source code model
Extracted from
.java; .c; .cpp; ...
Inspected source code

Fig. 2. Overview of the proposed recognition approach

A. Stage 1: Defining Patterns

As discussed before, a standard pattern catalog, defining all patterns and their variants is not available, but required for accurate pattern definition. A solution to that dilemma is a variable and customizable pattern catalog, collecting commonly agreed pattern definitions, but being also open for additions and improvements. Before referring to the concepts of that pattern catalog, we are introducing the concept of feature types that forms the basis of pattern definitions.

Feature types: The backbone of our pattern definitions are feature types. These feature types are reusable across all pattern definitions and can be imagined as being elementary and recurring across various design patterns (see Section III). For the detection of each feature type, we use a search technology that is most efficient while still being precise enough for the feature's detection. The current prototype uses the following search technologies: SQL for analyzing the source code model, regular expressions for analyzing texts (e.g., identifiers and comments) in the source code model and the source code, and source code parsers for identifying specific features not certainly identifiable through the other techniques.

A feature type is characterized by a search technology, a query, parameters, and a return type. The query is specific to the applied search technology and connects all these concepts. The purpose of a query is to retrieve all instances of a feature, meeting certain criteria. These criteria are either fixed within the defined query or they are parametric. Parameters receive their value upon the use of a feature type within a concrete pattern definition. Parameter can be defined as a static value or as a reference to the result of a related feature. The purpose of static parameters is keeping the number of required feature types at a convenient level. For example, a feature type for detecting a common attribute and that for detecting a common method between two classes, distinguish only in the type of the searched elements. Static parameters allow to create a feature type that detects common members of two classes and to set the type of the actually searched member as a parameter within the concrete pattern definition. Dynamic parameters allow to relate the detection results of multiple features. That is to pass the instances retrieved for one feature of a pattern definition can be used in a following query. Referring back to the example of a common method among classes, the intention usually is not to detect all common methods between all classes within an inspected source code, but to identify common methods between classes that have previously been detected as having other features.That means, dynamic parameters allow to relate features to each other.

TABLE I
EXAMPLES OF FEATURE TYPES USED FOR THE DEFINITION OF THE FACTORY METHOD PATTERN

FT1	Has class() returns: list of classes technology: SQL query on source model
FT2	Has super class(class) returns: list of classes technology: SQL query on source model
FT3	Have common method(class1, class2) returns: list of methods technology: SQL query on source model
FT4	Is returning class(method1) returns: class technology: specific parser inspecting source code
FT5	Is realizing interface(class1) returns: list of interfaces technology: SQL query on source model
FT6	Are generalized(class1, class2) returns: boolean value technology: SQL query on source model
FT7	Are aggregated(class1, class2) returns: boolean value technology: specific parser inspecting source code

Our approach currently uses 44 feature types that allowed us to define all the GoF patterns and their variations. Table I shows a subset of these feature types. The table also shows the applied detection technology, parameters and the results type for each feature type. We show these feature types as they contribute to the factory method example used throughout that paper. Required feature types, not currently available, but required for a new pattern definition, can be added to the XML feature type catalog.

All feature types can be separated into two categories: exploring feature types and checking feature types. An exploring feature type returns all artifacts that form a searched feature, usually in combination with another artifact, provided as parameter. For example, FT2 (see Table I) returns all super classes of a given class. A checking feature type consumes one or more artifacts as parameters, checks for the existence of a feature involving the provided artifacts and returns a boolean result. For example, FT6 (see Table I) checks whether or not a generalization relationship is existing between two provided classes.

Pattern definitions: After introducing the concept of feature types, we are now discussing the actual pattern definitions. These are a hierarchical structure comprised of the following concepts:

- Pattern catalog – collection of all pattern definitions
- Pattern definition – section of the catalog collecting all those variant definitions that capture implementation variants of the same pattern and are focussed on the detection of the same type of pattern
- Variant definition – section of a pattern definition that defines one implementation variant of a pattern as a collection of features
- Feature – section of a variant definition that has a type, defined in the feature type catalog, and defines all the types' parameters as static values or dependencies to other features

The feature is the basic concept of our pattern definitions, the other three concepts group features into a pattern variant, multiple variants into a pattern definition, and finally all patterns into one pattern catalog.

A variant definition is a combination of all relevant features, which in turn may use different recognition techniques. A feature must be of a type that is part of the feature type catalog. Furthermore, each feature has to define the parameters of its feature type (see previous discussion of feature types). Parameters are either defined as fixed values (static parameter) or as reference to results of a previous feature within the current variant definition (dynamic parameter). In the current form of the approach, features are detected sequentially as they appear in a variant definition. That means that references defined within one feature always have to refer to features that have been defined previously. The limitation of following a fixed, predefined order kept the development of our prototype simple. If desired, it would be a future exercise to allow random orders of feature types and to sort these automatically into an executable order before the detection process.

Figure 3 shows a variance definition that recognizes the GoF version of the Factory Method pattern. It consists of seven features, which have been identified for that implementation variant of the Factory pattern (see Subsection III-A). A feature is defined with a <feature> tag within a variant definition. The first defined feature is of type *FT1*. This feature type retrieves all classes of the inspected system (see Table I). The XML attribute *result* allows to name the results of an evaluated feature and to pass these results as parameter for another feature. The second feature is defined of type *FT2* and returns all super classes of a provided class. The XML attribute *params* allows to define these provided classes. Here, we use the class(es) retrieved by the first feature. The result of this feature is named *Creator*. The following three features are defined in the same manner.

For the last two features the XML attribute *negative* is defined as *true*. The approach allows to define each feature as a negative feature. Negative features define those features that a valid instance of a pattern shall not consist of. The detection of negative and ordinary features works similar, but their results are treated differently. While for a candidate pattern, a positive feature has to provide a result to be further considered as a candidate pattern, a negative feature must not provide a result for a candidate pattern to be further evaluated. In Figure 3 the last two features are negative, they check that no generalization exists between *ConcreteCreator* and *ConcreteProduct*, and also between *ConcreteCreator* and *Product*.

Definition process: The pattern definition process maps different feature types to the features of patterns (see Section III). An experienced user can select from all existing feature types or, if required, can define new feature types. In order to create a new variant definition, one evaluates the structure of a pattern and selects a central coarse grained element of the pattern's structure. The idea behind selecting a central element is to keep references between defined features simple. It is further important to make sure that features that a defined feature is referencing, have previously been detected. That means, for example, that it is not possible to check for common methods without having detected two sets of classes first, between which common methods are searched.

After selecting a central element, the definition process iteratively identifies related, relevant features and adds them to the variance definition. For each feature, the user selects the appropriate feature type from the list of available feature types. If the desired type is not available, the user can define a new feature type. Please note, that we were able to define all GoF patterns and their variances with a list of 44 feature types, suggesting that additional feature types will only rarely be required. The definition process proceeds until a pattern definition is reached that can precisely match the pattern.

B. Stage 2: Recognizing Defined Patterns

The detection process starts by creating a source code model of the examined application with the help of a modeling tool. Once that model is available, the actual detection process start. The approach iterates through each feature of each variant definition, selects the appropriate search technology for each feature, and executes the defined query with that search technology. Except for the first feature of a variant definition, that step is performed separately for each candidate patterns identified during previous feature searches. Figure 2 depicts the recognition controller that performs these actions. The obtained results are used to prune or extend the candidate patterns identified to the present search. If the search for the current feature returns exactly one result, the evaluated candidate pattern stays a candidate pattern for the evaluation of the next defined feature. If the search for the current feature

```
<pattern name="Factory Method" variance="GoF">
  <feature type=FT1 result="ConcreteCreator" />
  <feature type=FT2 params="ConcreteCreator" result="Creator" />
  <feature type=FT3 params="ConcreteCreator, Creator" result="FactoryMethod" />
  <feature type=FT4 params="FactoryMethod" result="ConcreteProduct" />
  <feature type=FT5 params="ConcreteProduct" result="Product" />
  <feature type=FT6 params="ConcreteCreator, ConcreteProduct" negative="true" />
  <feature type=FT6 params="ConcreteCreator, Product" negative="true" />
</pattern>
```

Fig. 3. Example of a pattern definition matching the Factory Method pattern as defined by the GoF

returns more than one result, the candidate pattern is split into multiple candidate instances for the next feature search. If the search for one feature returns no result for a candidate pattern, then this candidate pattern will be removed and not further evaluated. For negative features, the candidate is only kept, if the query returns no result. This procedure is performed across all features within a variant definition and eventually returns the list of detected patterns for that pattern variant.

Currently, the following techniques for the detection of features are used: SQL queries on the source model, regular expressions on the source code model and the source code itself, and specific source code parsers (e.g., to detect compositions and aggregations).

Source code model: The created source code model provides an abstraction of the source code, representing its structural artifacts and generic relationships between these artifacts. The major benefit of creating an intermediate model is that the generated model contains the structure of a given source code in a query-able repository (in the case of EA, a SQL database). That repository allows the efficient search within the structure of the code and to match key structural concepts. Today, modeling tools provide very mature and efficient reverse engineering capabilities for common programming constructs. We decided to use that functionality for identifying the main features of a pattern and to develop more sophisticated techniques for remaining specific features that are required for a precise detection. This cascaded approach is focusing on a trade-off between detection efficiency and precision.

Repository queries: The application of repository queries relies on the prior step of reverse engineering the code into a source code model. That step consumes some time (two minutes for JRefactory example used during evaluation), but the investment into this transformation is paying off. The transformation is necessary only once for a given state of the source code and modeling tools are able to incrementally update the model upon changes to the source code. Furthermore, SQL queries are more performant than analyzing the source code directly and these queries are independent from the programming language of the underlying code, as general constructs are queried. Finally, an available source code model facilitates a graphical representation of detected patterns. In fact, our prototype is able to highlight patterns within class diagrams of the source code. A limitation of using queries is that a user creating new feature types must have knowledge of the model repository's internal data structure. Though, new feature types will only rarely be required and we found the database used by EA to be understandable after a short time, also by novice users.

Specific parser modules: Though, the analysis of the source code model is very powerful, it does not allow to match all the searched concepts. For example, the distinction between an aggregation and a composition of classes requires a more thorough analysis of the source code. For such purposes, we introduce the concept of source code parser modules that extract specific features directly from the source code. This detection method is less performant than model queries, but allows very specific and detailed analyses. Our parser modules are based on the Coco/R [15] parser generator. Coco/R provides grammars for all common programming languages. We created a reference implementation for parser modules, providing a standardized interface and allowing it to be used almost without configuration as a query of a feature type.

Our current prototype implementation is using five parser modules for detecting delegation between classes, aggregation between classes, method invocation, method invocation through reference, and true method return type. Parser are not language independent as the searched features can be implemented in very different ways with different programming languages. The four modules used by our prototype, detect the searched features in Java and C# code. The support of C/C++ code is under development.

Parser modules require additional effort for their implementation, they are not language independent as repository queries, and they are less performant than model queries. Nonetheless, their application is required in order to return precise detection results. Furthermore, we were able to define all GoF patterns and their variances with five parser modules and expect very rare demand for additional ones.

Regular expressions: In addition to repository queries and parser modules, our approach is using regular expressions to extract information either from the source code model or directly from the source code. We apply regular expressions especially to extract constructs from expressions like comments, classifier names and annotations. Regular expressions are simple to write and easy understandable. We avoid using source code parser modules for extracting information that is accessible by using regular expressions as their application is more performant than the the analysis with a parser. A general problem of regular expressions is that they are not able to certainly extract nested information within source code elements. For more sophisticated analyses, parser modules are used.

C. Pattern Detection Example

That subsection illustrates a full recognition procedure by an example. We assume that a modeling tool has been used to reverse engineer a model of the source code (see Figure 4). The model contains seven elements and seven relationships among them. The goal is to detect the Factory Method pattern within the source code based on the definition shown in Figure 3 and discussed in Subsection IV-A.

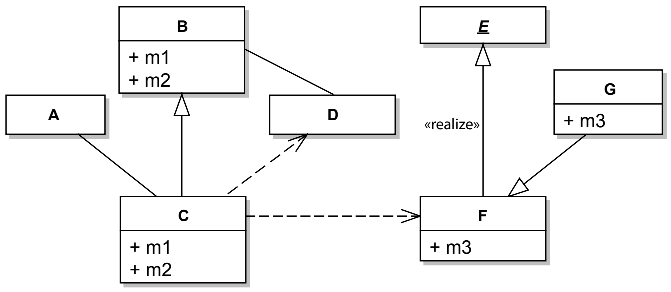

Fig. 4. Illustrative detection example

The detection process iteratively searches for each defined feature and returns the following results per feature:

F1 HasClass() $\rightarrow$ A, B, C, D, F, G
F2 HasSuperClass(A, B, C, D, F, G)
$\rightarrow$ {C, B}, {G, F}
F3 HaveCommonMethod({C, B}, {G, F})
$\rightarrow$ {C, B, m1}, {C, B, m2}, {G, F, m3}
F4 IsReturningClass(m1, m2, m3)
$\rightarrow$ {C, B, m1, F}, {C, B, m2, D}
F5 IsRealizingInterface(F, D) $\rightarrow$ {C, B, m1, F, E}
F6 NOT(AreGeneralized(C, F)) $\rightarrow$ {C, B, m1, F, E}
F7 NOT(AreGeneralized(C, E)) $\rightarrow$ {C, B, m1, F, E}

Searching for the first feature returns all classes of the model as possible candidate patterns. Please note that classifier E is an interface and the relation to class F an interface realization. After searching the second feature, only three classes prove to have a super class and accordingly, only three combinations ({C, B} and {G, F}) remain as candidate patterns. Searching for the third feature, a common method between the classes, returns two methods for the first class combination {C, B, m1}, {C, B, m2} and one method for the second combination {G, F, m3}. If the detection of one feature returns more than one result, the associated candidate pattern is duplicated and each combination will be separately evaluated from now on.

The fourth feature requires the previously detected common method to return a classifier. Only methods m1 and m2 are returning a class and accordingly only the candidate patterns {C, B, m1, F} and {C, B, m2, D} are further evaluated. The fifth required feature is present in a candidate pattern, if the class returned by the common method is realizing an interface. That is the case only for class F and accordingly the combination {C, B, m1, F, E} remains the only candidate pattern. The sixth and the seventh feature require that no generalization is present between ConcreteCreator and ConcreteProduct, C and F for the candidate pattern, and between ConcreteCreator and Product, C and E for the candidate pattern. Both feature are negative features and return no result. That means that {C, B, m1, F, E} combination of artifacts is a valid factory method pattern according to our definition.

V. PROTOTYPE

We have developed a prototype implementation of the proposed approach, implemented with C#. That prototype is in large parts independent of a specific modeling tool and can easily be adopted to a variety of tools like IBM Rational Software Modeler^{TM} and IBM Rhapsody^{TM}. For our current prototype implementation we selected Sparx Enterprise Architect^{TM} (EA) as modeling tool, due to its excellent capability for reverse engineering source code from more than ten programming languages and due to prior experience in creating extensions for that tool [16].

The tool integrates with the modeling tool. Upon starting the tool, the current pattern and feature type catalogs are loaded and validated. An additional menu is added to the modeling tool. That menu allows the user to detect either a specific type of pattern or all patterns defined within the current pattern catalog. Behind the scenes, a recognition controller loads variant definitions and iteratively executes the defined query on the search technology that is defined for a feature (see Subsection IV-A). After each executed query, the list of candidate patterns is being pruned, updated or extended. The final pattern list contains exact information about all detected patterns and their structural parts. The user can browse that information as a textual list and also highlight detected patterns within class diagrams. The pattern catalog and the catalog of feature types are stored in separate XML files and can be edited with any XML or text editor. In a future version of the tool, we plan to provide a customized editor with instant validation of changes.

VI. EVALUATION

We performed a series of experiments in order to evaluate our approach. After an initial validation with small C++ examples, we decided to set-up a larger experiment involving open-source projects, implemented in JAVA, ranging from 43 to 562 code classes.

A. Project selection

The goal of our experiment was to validate the recognition quality of our approach. For that purpose, we selected three software systems with differing size and complexity, based on the following considerations: 1) the source code of the systems is freely available, especially for other researchers following up on our results; 2) size and complexity of systems are increasing; 3) the system is known to incorporate a rich variety of patterns; and 4) the system has been used for pattern detection before and allows comparison of results.

Table II lists the selected systems and reports metrics about each in order to provide an impression of size and complexity. All of them have previously been used for the evaluation of pattern detection methods.

TABLE II
STATISTICS OF EXAMINED APPLICATIONS

	JUnit	JHotDraw	JRefactory
Version	3.7	5.1	2.6.24
Size on disk	1.46 MB	4.85 MB	12.5 MB
Code files	78	144	1167
KLOC	9.7	30.9	216.2
Classes	43	136	562
Methods	425	1314	4881
Attributes	114	331	1367

B. Variables and Measures

We controlled the independent, factorial variables project and type of pattern. Project had three levels according to the employed examples, while type of pattern had 23 levels representing the GoF patterns. For each combination of project and type of pattern, we captured the number of recognized pattern instances n_{rP} and classified each match and each missing match with respect to the baselines.

C. Determining Baselines

In order to compare results with previous works, it was necessary to evaluate each match and to also identify missing matches. While, it is relatively easy to evaluate the correctness of retrieved matches by comparing them with the definition of a pattern, the evaluation of completeness is extremely challenging as no documentation about existing patterns exists. We decided to use the results of previous approaches as a candidate baseline and to gradually improve that baseline with the results of further experiments.

Per project, we studied experimental results of previous approaches using that project. We evaluated published results and tried to run prototype implementations of the approach, where available. Furthermore, we used the results of our approach to enrich previously known instances. In result, we created a baseline for each of the three evaluated systems. These baselines are not equally trustworthy. For the smaller systems and for the easier recognizable patterns, more and stronger previous results are available and form a more trustworthy baseline. Appreciating that fact, we decided to tag each pattern type/project combination with the number of other approaches that found the same pattern instances. The BL columns of the results Table III shows the determined baseline for each pattern type/project combination. The number of stars attached to the figure reflects the number of other approaches agreeing on the same number of existing instances.

D. Results

Our main experiment was performed on the software systems listed in Table II. Table III reports the results of the evaluated systems. Per project and per pattern type, the table provides information about three facts. The first column (BL) reports the created baseline and its trustworthiness (see Subsection VI-C). The following three columns show the number of detected pattern instances by previous approaches. The figure in parentheses always shows the number of detected instances, shared with the baseline. For example, 4(3) means that four patterns have been found, but only three are shared with the baseline and considered true positives. A question mark (?) within the parentheses means that a detailed comparison with the baseline was not possible. The last column n_{rP} shows the number of patterns recognized by our approach and the number of instances shared with the baseline.

We applied a two stage strategy during evaluation. The first two projects (JUnit and JHotDraw) were not only used to evaluate the approach, but also to improve our pattern catalog. We extended and improved the specifications of several patterns after the first pattern detection. The JRefactory project was only used to evaluate the approach. The idea of that second stage was to initially validate the quality of the pattern catalog and how much customizations would be required for new projects under inspection.

Stage one started with JUnit, which is a standard example for pattern recognition. After improving the pattern catalog, we detected all patterns as expected, without false positives and negatives. Especially, for the patterns in the upper part of the table the baseline is strong. Multiple researchers agree on the correctness of these results. For several of the remaining patterns, we provide a first result, open for debate. In continuation, the approach was tested on JHotDraw. For JHotDraw, all recognized patterns were correct and complete as defined by the baseline, except for two instances of State and Strategy. The reason for these missing instances is an open issue in the delegation parser. We actually found a problem in the grammar of the Coco/R parser generator, which we hope will be fixed within the next months.

In stage two, we evaluated JRefactory. For that project we report results as obtained at the first detection run with the pattern catalog improved in stage one. For JRefactory, we miss one instance of Adapter, three instances of State and Strategy, and we recognize three false instances of Builder.

E. Discussion

During the first stage of the evaluation we were able to obtain very good results. The two missing instances in JHotDraw occured due to a technical problem in the delegation parser, which we hope to fix soon. These results show that a customized pattern catalog facilitates high quality detection results. During the second stage of the evaluation we experienced a larger number of false and missing recognitions. These happen due to three reasons: 1) the grammar issue with the delegation parser, 2) incomplete and incorrect pattern defini-

TABLE III
EVALUATION RESULTS n_{rP} OF OUR APPROACH IN COMPARISON TO THE ESTABLISHED BASELINE (BL) AND RELATED WORK. THE FIGURE IN BRACKETS BEHIND EACH RESULT SHOWS THE NUMBER OF SHARED INSTANCES WITH THE BASELINE. A QUESTION MARK MEANS THAT A COMPARISON WAS NOT POSSIBLE. THE COLUMN BL SHOWS THE NUMBER OF TRUE POSITIVE PATTERN INSTANCES. THE NUMBER OF STARS (*) REFERS TO HOW MANY RESEARCHERS AGREE WITH US ON THAT BASELINE AND IS A MEASURE OF TRUSTWORTHYNESS.

	JUnit 3.7					JHotDraw 5.1					JRefactory 2.6.24				
	BL	[7]^1	[8]^1	[17]	n_{rP}	BL	[7]^1	[8]^1	[6]	n_{rP}	BL	[7]^1	[8]^1	[18]	n_{rP}
Singleton	0**	0	0	–	0	2**	2(2)	2(2)	–	2(2)	12*	12(12)	2(2)	1(1)	12(12)
Adapter	6*	6(6)	0	–	6(6)	22	18(18)	1(1)	41(?)	22(22)	17*	26(16)	17(16)	16(16)	16(16)
Composite	1**	1(1)	1(1)	1(?)	1(1)	1**	1(1)	1(1)	0	1(1)	0**	0	0	–	0
Decorator	1**	1(1)	1(1)	–	1(1)	3*	3(3)	1(1)	0	3(3)	1*	1(1)	0	–	1(1)
Factory M.	0**	0	0	–	0	3**	3(3)	3(3)	–	3(3)	1**	1(1)	1(1)	0	1(1)
Template M.	1*	1(1)	0	1(?)	1(1)	5*	5(5)	2(2)	–	5(5)	17*	17(17)	0	–	17(17)
Prototype	0**	0	0	–	0	2*	1(1)	2(2)	–	2(2)	1	0	0	–	1(1)
Command	0**	0	0	–	0	8*	8(8)	1(1)	–	8(8)	0***	0	0	0	0
Observer	3*	4(3)	3(3)	4(?)	3(3)	2*	5(2)	2(2)	–	2(2)	0**	0	0	–	0
Visitor	0***	0	0	0	0	0**	0	0	–	0	2***	2(2)	2(2)	2(2)	2(2)
State/Strat.	3*	3(3)	0	–	3(3)	22*	22(20)	6(6)	–	20(20)	11*	11(8)	2(2)	3(2)	8(8)
Proxy	0**	0	0	–	0	0***	0	0	0	0	0***	0	0	0	0
Bridge	0	–	–	–	0	5	–	–	75(?)	5(5)	0	–	–	–	0
Interpreter	0*	–	0	–	0	8	–	0	–	8(8)	1	–	–	–	1(1)
Builder	0*	–	0	–	0	2	–	0	–	2(2)	2*	–	2(2)	–	5(2)
Iterator	0	–	–	–	0	0	–	–	–	0	2	–	–	–	2(2)
Memento	5	–	–	–	5(5)	10	–	–	–	10(10)	30	–	–	–	30(30)
COR	0	–	–	–	0	0	–	–	–	0	1	–	–	–	1(1)
Abst. Fact.	0*	–	0	–	0	0*	–	0	–	0	0*	–	0	–	0
Flyweight	3	–	–	–	3(3)	15	–	–	–	15(15)	15	–	–	–	15(15)
Facade	6	–	–	–	6(6)	30	–	–	9(7)	30(30)	24	–	–	–	24(24)
Mediator	0*	–	0	–	0	0*	–	0	–	0	0*	–	0	–	0

^{1}) Please note that we refer to results updated by the authors after the initial publication.
The latest results can be found on: java.uom.gr/˜nikos/pattern-detection.html and www.ptidej.net/downloads/pmart/ respectively.

tions, and 3) missing detection capabilities of our currently adopted search techniques.

Regarding issue 1) is a purely technical one and should be fixed soon. Issue 2) occurs, because the same pattern can be implemented in a large variety of different ways. With each additionally evaluated project for, which baseline results are available, the pattern catalog can be improved. There will probably never be the final catalog, but our technology facilitates an easy customization by the user. Regarding issue 3), we are currently working on an extension of the approach which will provide feature types based on an analysis of execution traces captured from the application at runtime.

a) Disparity of results: An analysis of patterns detected by our approach and by the approaches we used for creating baselines (see Tables III) shows for some patterns a wide disparity of results. We communicated these disparities to the authors and hope that this will ultimately lead to agreed benchmark results for the evaluated projects.

F. Threats to Validity

Validity is a key challenge for researchers and practitioners in conducting empirical research work. Regarding external validity, the threat is, whether our results are generalizable for a larger population, based on the performed experiment or not. In order to mitigate that threat, we decided to use three different examples for the evaluation of our technique. All three systems are open-source developments and might not be fully representative for industrial systems, but we decided for those to allow comparison of results with previous approaches. Our experiment focussed on the detection of all GoF patterns and their variations. Previous work was often restricted to a subset of patterns, with possibly reduced difficulty in recognition. Industrial systems may contain additional variations and additional patterns, but that is a scenario explicitly supported by our approach.

Regarding internal validity, we identified two possible threats. First, a lack of standard definitions for design patterns. Our approach proposes customizable pattern definitions that could help to eventually overcome that problem. For this experiment we created initial definitions for all GoF patterns and variances that we found in literature and by comparing our results with the results of other research groups. Second, no commonly agreed benchmarks are available for pattern recognition approaches. We tried to mitigate that threat by establishing combined baselines for all evaluated projects and by providing a rating for each pattern/project combination showing the number of researchers agreeing on that value.

VII. CONCLUSIONS AND FUTURE WORK

The detection of design patterns within source code of legacy applications supports reverse engineering, program comprehension, and maintenance activities. In this paper, we propose variable pattern definitions composed of reusable feature types. Each feature type is assigned to one of multiple search techniques that is best fitting for its detection. A prototype implementation was applied to three open source applications. For each system a baseline was determined and used for comparison with the results of previous techniques. We achieved high detection accuracy in cases where the pattern catalog was improved for the example and also demonstrated that customization might be required in order to detect pattern variations specific to other examples. Our results demonstrate the importance of customizable pattern definitions and multiple search techniques in order to overcome accuracy and flexibility issues of previous approaches.

Future work, will mainly focus on two topics. First, improving the pattern definition for the user by instant validations of changes and by a possibly graphical representation of defined patterns. Second, improving the detection of some patterns by integrating feature types based on run-time execution traces of a system.

Acknowledgments

The authors would like to thank Sebastian Klehm for implementing parts of the prototype. This work is partly funded by FWF grant M1268-N23.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[2] C. Zhang, "An empirical assessment of the software design pattern concept," Ph.D. dissertation, Durham University, 2011.

[3] C. Krämer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *WCRE'96*, 1996, pp. 208–215.

[4] N. Pettersson, W. Löwe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection," *IEEE TSE*, vol. 36, no. 4, pp. 575–590, 2010.

[5] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 6, pp. 823–855, 2009.

[6] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *JSS*, vol. 82, no. 7, pp. 1177–1193, Jul. 2009.

[7] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE TSE*, vol. 32, no. 11, pp. 896–909, 2006.

[8] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multilayered approach for design pattern identification," *IEEE TSE*, vol. 34, no. 5, pp. 667–684, 2008.

[9] J. Dong, D. S. Lad, and Y. Zhao, "DP-miner: Design pattern discovery using matrix," in *ECBS'07*, 2007, pp. 371–380.

[10] Y.-G. Guéhéneuc, J.-Y. Guyomarc'h, and H. A. Sahraoui, "Improving design-pattern identification: a new approach and an exploratory study," *Software Quality Journal*, vol. 18, no. 1, pp. 145–174, 2010.

[11] G. Costagliola, A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Case studies of visual language based design patterns recovery," in *CSMR'06*, 2006, pp. 165–174.

[12] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "A two phase approach to design pattern recovery," in *CSMR '07*, Mar. 2007, pp. 297–306.

[13] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *ASE'06*, 2006, pp. 123–134.

[14] V. D. Radonjic and J.-P. Corriveau, "Making patterns better design tools: Requirements analysis for a family of navigators for design pattern catalogs," in *Int'l Conf. Software Engineering*, 2005, pp. 349–354.

[15] H. Mössenböck, A. Wöß, and M. Löberbauer, "Der Compilergenerator Coco/R," in *Peter Rechenberg – Festschrift zum 70. Geburtstag*. Trauner-Verlag, 2003.

[16] P. Mäder, O. Gotel, T. Kuschke, and I. Philippow, "traceMaintainer – automated traceability maintenance," in *RE'08*, 2008, pp. 329–330.

[17] A. Alnusair and T. Zhao, "Towards a model-driven approach for reverse engineering design patterns," in *TWOMDE*, 2009.

[18] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design patterns and change proneness: An examination of five evolving systems," in *METRICS'03*, 2003, pp. 40–49.