Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering, © JOT 2010
Online at http://www.jot.fm.

Inferring design patterns using the ReP graph

Tushar Sharma[a] Dharanipragada Janakiram[a]

a. Distributed and Object Systems Lab, Computer Science & Engineering department, Indian Institute of Technology-Madras, Chennai-36, India.
http://dos.iitm.ac.in/index.shtml

Abstract Periodic refactoring of a large source code often becomes a necessity especially for long-lived projects. In order to increase maintainability and extensibility of such projects, design pattern based refactoring can be seen as an emerging alternative. Manual inspection of source code to find candidate spots where patterns can be introduced is time consuming. Therefore automated tools can help in identifying candidate spots where patterns can be introduced. The level of source code abstraction plays an important role for building such tools. We propose a new abstraction for object oriented source code that is named as "Refactoring Pattern (ReP) Graph" to realize an effective design pattern based refactoring tool. The ReP graph abstracts the source code information thereby making the process of design pattern inference easier. The proposed tool identifies candidate spots in a given source code to introduce design patterns.

Keywords design pattern, design pattern inference

1 Introduction

Maintenance becomes the longest phase in the software development life cycle, especially for large and long-life projects. A software system becomes a candidate for refactoring when extensibility and maintainability issues are experienced during feature addition or bug fixing sessions. In such cases, the pattern based refactoring helps the developer to extend the software system with ease.

The process of finding candidate spots in a given source code where specific design patterns can be introduced to make the overall design better is known as design pattern *inference*. A related but different term is design pattern *detection*. In case of detection, the patterns are detected when they already exist in the underlying code. On the other hand, inference looks for the available intent-aspect to restructure the code into pattern based code. Software quality attributes such as maintainability, extensibility and understandability might be improved by introducing the inferred patterns into the code.

The abstraction level of the source code plays an important role in the process of inferring design patterns from a given source code. Class diagrams provide design level abstracted view, however they may lack the useful information required to identify refactoring opportunities. At the same time, full source code might be voluminous and may have information overload. Therefore, a proper abstraction can lead to an effective design pattern inferring technique.

There are attempts to achieve pattern based refactoring such as "Refactoring to Patterns" [K.05] which provide handful of tips on how to refactor code to design patterns based code. Nevertheless analyzing huge software systems manually is a tedious and time consuming job. An automated tool can analyze the code and point out the refactoring opportunities.

In essence, a technique for inferring patterns to refactor a given source code is required to improve the quality of a software system. Given a source code as input, the technique must be able to identify candidate spots where specific patterns can be introduced. The candidate spot should point out the class(es)/method(s) where the pattern can be applied. Additionally, the technique itself must be extensible in terms of pattern rule base, so that inclusion of addional rules for pattern identification is feasible at a later stage.

The rest of the paper is organized as follows. Section 2 introduces the pattern graph, ReP graph and respective notations. Section 3 describes the inferring process using the ReP graph. Section 4 presents summary of an analysis done on various open source projects. Section 5 discusses related work, section 6 summarizes contribution of this paper and finally we conclude in section 7.

2 ReP Graph

The work presented in this paper is based on the concepts of design structures and pattern graph proposed by Janakiram et.al. [JAG$^+$00]. The design structure concept is explained in brief here to make the rest of the paper understandable.

2.1 Design structures and pattern graph

Design patterns are community accepted solutions to recurring design problems [GHJV95]. Furthermore, a finer level fundamental abstraction is defined which is used to describe design patterns. These abstractions are known as design structures. Every design pattern is a composition of one or more design structures [MS05].

The new abstraction can be used to view the entire design of a software system. A design diagram built from the design structures is defined as a pattern graph. Pattern graph [JAG$^+$00] is derived from UML class diagram and interaction diagram. The pattern graph uses the following notation to represent the source code information. Classes are represented by rounded rectangles and these rectangles have 3 partitions. These partitions describe number of template methods (T), hook methods (H) and rigid methods (R) in the class(es). A hook method is declared in a class and defined in its subclass. A method which calls at least one hook method is known as a template method, while a rigid method is declared and defined in a same class. These methods are chosen to define the design structures and pattern graph because these methods form a minimal set by which structure, behavior, and rationale of design patterns can be captured and described.

A set of classes, which differs only in definition of hook methods is treated as a single node and is represented only once in a pattern graph. A client class is a class, which makes use of a pattern for getting its services. A rounded rectangle without partitions is used to represent a client node. One-to-one and one-to-many associations are depicted by a dashed line arrow and a solid line arrow respectively. A gate node is used to capture inheritance and polymorphism. Every gate node is associated with a set of classes called a *receive-set* (from objects it receives messages) and a set of classes called a *send-set* (to objects it sends messages).

An illustrative example is presented here to understand notations of the pattern graph. Figure 1 shows a class diagram and a corresponding pattern graph for the composite pattern [GHJV95]. The *Leaf* class with its base class *Component* is represented by a pattern graph node (T1, H1, R1). Similarly, the *Composite* class with its base class is represented by another pattern graph node. The *Client* class is calling either a *Composite* or a *Leaf* object based on an instantiated object. This polymorphic behavior is captured by a gate node. The *Composite* class has a one-to-many aggregation relationship with its base class which is shown by a solid line from the second pattern graph node to the gate node.

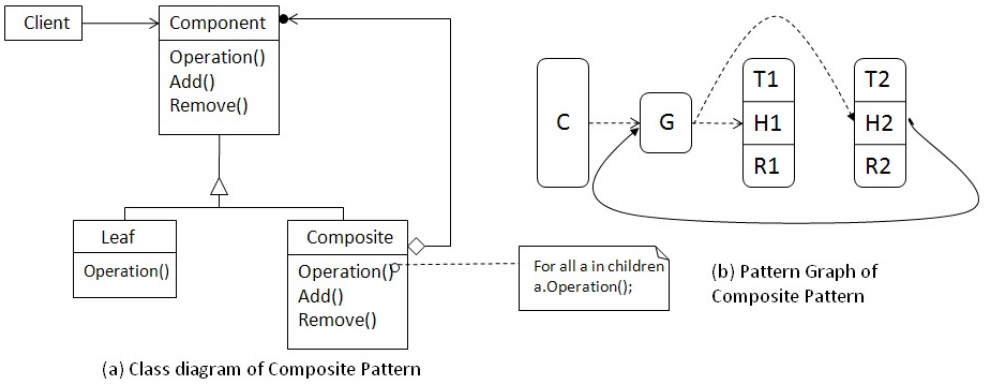

Figure 1 – Class diagram and pattern graph for the composite pattern

In essence, design structures and pattern graph can be defined as follows:

Design Structures: *A set of T, H and R methods with their interactions are defined as design structures.*

Pattern Graph: *A design diagram in which design entities are represented by design structures is known as a pattern graph.*

2.2 The ReP graph

The inferring process requires source code level information, which is not present in a pattern graph. Hence, the pattern graph by itself is not useful in the refactoring context. The proposed solution, Refactoring-Pattern (ReP) graph is an extension of the pattern graph (described in section 2.1). The ReP graph abstracts the source code information and exposes them in terms of fundamental constructs. These fundamental constructs are defined from program entities such that they capture intent-aspect of one or more design patterns. These constructs are used to form design pattern inference-rules. The information contained within these constructs can be accessed by a set of APIs provided with the ReP graph.

In essence, a ReP graph consists of:

ReP Graph Constructs	Description
`THR (T, H and R methods)`	Template, hook and rigid method information
`Containment, association and create lists`	List of various relationships among classes
`Condition, condition-within-condition, condition-within-function`	A condition, a condition within a condition, a condition within a function
`Create-within-condition, create-within-function`	A create list within a condition/function
`Condition-alt-path-list`	A list of alternate paths for a condition
`Updated-var-list-by-condition, updated-var-list-by-function`	Updated variable lists by condition/function
`Function-called, passed-params`	Called function list and passed parameters

Table 1 – A brief list of `ReP graph fundamental constructs`

1. **ReP nodes:** Each ReP node represents a set of classes of an inheritance tree. A new level of abstraction is achieved using these nodes. A super class and its sub classes are represented by one ReP node; provided hook method(s) are defined on these classes and there is no association relation among these classes. A gate node is used to manage inheritance and polymorphism where association relations exist among a super class and its sub-classes.

   (a) **Fundamental constructs:** Fundamental constructs are basic building blocks, which can be combined together to form a meaningful inference-rule. Each ReP node consists of a few fundamental constructs. These constructs capture source code information such as inheritance, aggregation, association, T/H/R method calls, conditions, and create instructions. Moreover, composite information such as *create-within-condition* and *condition-within-condition* is also supported by the provided constructs. The inference-rules are written using these constructs determine a candidate spot for a design pattern. A brief list of the ReP constructs is given in Table 1.

   (b) **Accessor APIs:** Inference-rules use information contained within ReP graph constructs. The ReP graph provides a set of APIs to access the ReP graph constructs.

2. **ReP edges:** ReP nodes are connected to each other either by solid lines (1:n relationship) or dotted lines (1:1 relationship).

A ReP graph can be defined as follows:

ReP graph: *ReP graph is an extended pattern graph, which consists of ReP nodes and edges. Each ReP node consists of a set of fundamental constructs, which can be accessed using provided accessor APIs. ReP edges connect ReP nodes by solid/dotted lines.*

Design Pattern	ReP constructs
`Abstract Factory`	Create-within-condition
`Bridge`	Inheritance-depth, THR
`Builder`	Create, Function-called, Passed-params
`Composite`	Containment, Function-called
`Decorator`	Function-called, THR
`Facade`	Association, Function-called
`Observer`	Association, Updated-var-list-by-function, Function-called, Passed-params
`Singleton`	THR, Member-variables
`State`	Condition, Condition-alt-path-list, Function-called, Updated-var-list-by-condition
`Strategy`	Condition, Condition-alt-path-list, Function-called

Table 2 – Design patterns and corresponding ReP fundamental constructs to infer them

3 Inferring patterns using the ReP graph

The motivation of the proposed work is to infer design patterns in a given source code. The work introduces design pattern at appropriate places (i.e. candidate spots) in the source code to make the source code more maintainable, extensible and understandable. The proposed work offers a new abstraction level for the source code, which makes the process of design pattern inference easier and flexible. The ReP graph captures the required source code information in terms of fundamental constructs. These constructs are used to detect the intent-aspect of design patterns within a given source code. Table 2 shows some of the GoF patterns [GHJV95] and the corresponding set of constructs used to infer these patterns.

A tool "Refactor-it" is developed to realize this idea. The tool accepts source code of a program as input, identifies fundamental constructs and stores them in a ReP graph. In order to identify the intent-aspect of design patterns; inference-rules are written for the patterns. The tool infers design patterns by executing inference-rules on the ReP graph. The output of the tool shows candidate spots with applicable patterns.

3.1 Architecture of the design pattern inferring tool

The method to generate the ReP graph and then using it for inferring design patterns comprises of three steps:

1. Generate Abstract Syntax Tree (AST)
2. Generate ReP graph
3. Infer design patterns

As shown in Figure 2, the first step uses an appropriate parser to generate Abstract Syntax Tree (AST) of a given software system. The tool employs Code-inspector [Cod] to generate an AST. The Code-inspector provides a set of APIs to access the generated AST. API Abstraction Layer (AAL) exposes a set of higher level useful

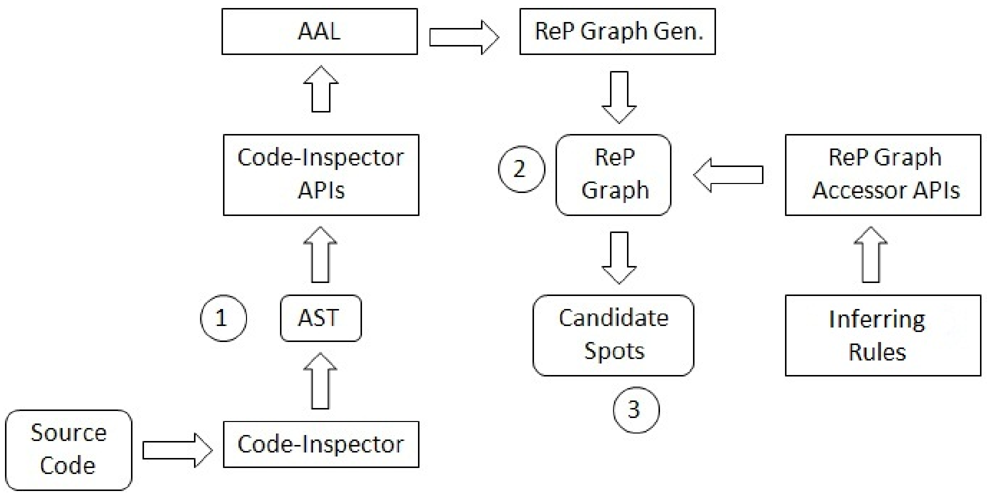

Figure 2 – System architecture of the design pattern inference tool

methods (such as *get-all-classes*, *get-all-methods*) to access the AST. The AAL uses APIs provided by the Code-inspector to extract the information from the AST. ReP graph generator collects required source code level information using the AAL and puts them into a ReP graph.

The ReP graph can be accessed by the accessor API set provided with the ReP graph. Inference-rules are written using these APIs for inferring design patterns. User can extend this rule base with his/her rules for additional patterns. These rules are evaluated on the generated ReP graph and an output consisting of a set of candidate spots is generated. A candidate spot is reported wherever the ReP graph satisfies an inference-rule of a design pattern. A candidate spot provides information about the inferred site (class(es) and/or method(s)), the inferred pattern and roles of the design pattern played by existing program entities.

3.2 ReP graph generation

As shown above in Figure 2, ReP graph generator (RGG) uses the AAL to extract the required source code information and generates ReP graph for a given software system. The RGG uses an algorithm to generate a ReP graph which is given as below in Listing 1.

A top-level class is a class, which do not have any super-class in a given source code. The RGG retrieves all top-level classes and maintains a corresponding top-level ReP node list. Classes belong to an inheritance tree form a working-class-list. The RGG derives an expression (i.e. ReP graph expression) from inter-class relationships among classes of the working-class-list. This expression indicates number of required ReP nodes, classes belong to each ReP node, and number of required gate nodes to represent the inheritance tree. A partial ReP graph is created using the ReP graph expression. Fundamental constructs of each ReP node are populated with information discovered from the source code. The process is repeated for each of the top-level class to create and populate the complete ReP graph.

```
//Function: generate-rep-graph
//i/p params: AST
//o/p type: list of ReP nodes
generate-rep-graph (AST)
  {
  //get all top-level classes
  top-level-class-list = AAL.get-top-level-class-list(AST)
  top-level-rep-nodes = null
  //create rep graph for each top level class
  do while (top-level-class-list.empty() != true)
    {
    cur-class = top-level-class-list.getItem()
    //get all derived classes from cur-class
    working-class-list = AAL.get-inheritance-tree(AST,cur-class)
    //get rep-graph expression based on inter-class relationship
    rep-graph-expression = get-rep-graph-expression(AST,
      working-class-list)
    //create rep-graph for the working-class-list
    partial-rep-graph = create-rep-graph(rep-graph-expression)
    //find the source code constructs and populate the rep-graph
    populate-construct-info(AST, partial-rep-graph)
    //remove the processed class from the list
    top-level-class-list.remove(cur-class)
    //add the created rep-graph to the top-level list
    top-level-rep-nodes = top-level-rep-nodes +
               get-top-rep-node(partial-rep-graph)
    }
  //connect nodes of one inheritance tree to other
  connect-rep-nodes(top-level-rep-nodes)
  return top-level-rep-nodes
}
```

Listing 1 – Algorithm to generate ReP graph.

3.3 Inferring using the ReP graph: An example

An illustrative example is presented to explain various steps of the inferring process. Here, Figure 3 shows a partial UML class diagram for the example program. Classes *Button* and *Window* along with their subclasses define product hierarchies and class *Display* is creating a family of these product-objects based on a condition.

The design pattern inferring tool accepts this program as an input and generates an AST with help of the Code-inspector. The generated AST is analyzed and fundamental constructs from the source code are identified. The tool then populates the ReP graph with these constructs.

The intent-aspect of a design pattern is translated into an inference-rule using the accessor APIs provided by the ReP graph. For example, the intent-aspect of the abstract factory design pattern [GHJV95] should satisfy the following clauses:

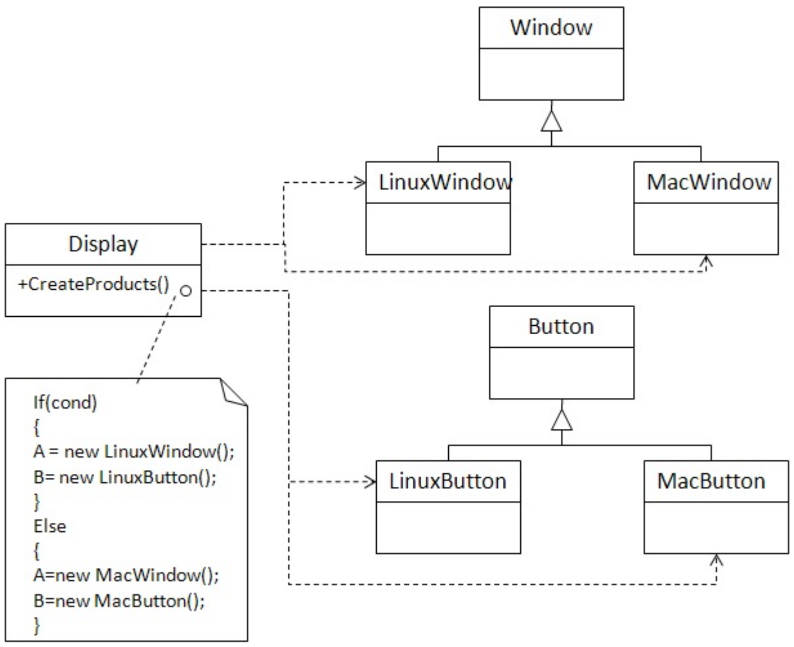

Figure 3 – UML class diagram for considered example

1. There is a condition which has at least two alternate paths.
2. Every alternate path should have at least two *Create* constructs and the number of *Create* constructs in each path must be equal.
3. The objects created in each alternate path must have siblings in all other alternate paths.

Listing 2, as given below, shows the algorithm to identify the above said intent-aspect of the abstract factory design pattern. The algorithm maps the intent-aspect of the pattern to *create-within-condition* construct.

```
//Function: rule-abstract-factory
//i/p params: rep-node
//o/p type: boolean
rule-abstract-factory(rep-node)
  {
  //get all function object list for the rep node
  func-obj-list = get-function-objects(rep-node)
  do(for all func-obj in func-obj-list)
    {
    //get all condition objects
    cond-obj-list = get-condition-nodes(func-obj)
    do(for all cond-obj in cond-obj-list)
      {
      //get all alternate path objects list
      alt-obj-list = get-alt-path-list(cond-obj)
      if(alt-obj-list.length() < 2)
      return false
    cur-top-list = null
    do(for all alt-obj in alt-obj-list)
      {
      //get create object list
      create-obj-list = get-create-list(alt-obj)
      if(create-obj-list.length() < 2)
              return false
      //check for the inheritance hierarchy
      if(cur-top-list == null)
        cur-top-list = get-top-list(create-obj-list)
      else
        if(not(match-inh-hierarchy(cur-top-list,create-obj-list)))
          return false
        }
       report-abstract-factory(func-obj, cond-obj)
       }
    }
}
```

Listing 2 – Algorithm for inference-rule to identify intent-aspect of the Abstract Factory pattern

Accessor APIs provide an interface to access a ReP graph. Inference-rules use these APIs to specify the intent-aspect of design patterns. The rules for the standard patterns are written using these APIs and made available with the tool. User may extend this rule base by writing rules for additional patterns. All aforementioned inference-rules are implemented in LISP using ReP graph constructs.

Once the ReP graph is generated, the tool fires inference-rules on the ReP graph. In the example considered, *create-within-condition* construct is identified which satisfies the intent-aspect of the abstract factory design pattern.

A partial ReP graph as shown in Figure 4.a corresponds to the illustrative considered in Figure 3. The figure is showing the intent-aspect for the abstract factory pattern. Figure 4.b shows the resultant ReP graph for the abstract factory pattern. This resultant diagram shows transformed ReP graph, which can be obtained by introducing the inferred abstract factory pattern into the inferred site.

The result of the inferring process is a set of candidate spots. Each candidate spot reports not only the inferred site (concerned class(es) and/or method(s)) and the inferred pattern but also various roles of the inferred design pattern assigned to various program entities. In this example, *client* and *product* roles are identified and reported in the candidate spot (as given in Listing 3). The information of the candidate spot is provided in XML format, therefore a transformation engine can take it as input and transform the code.

4 Summary of analysis

The tool "Refactor-it" is employed against various open source projects to check the usability of the tool. The projects considered are Jnet-lib [Jne], Astro [Ast], Chess [Che], Notepad [Nota], Free Framework [Fre] and Notepad++ [Notb]. Here, the chosen projects are as small as 10 classes (Astro) and as big as 120+ classes (Notepad++).

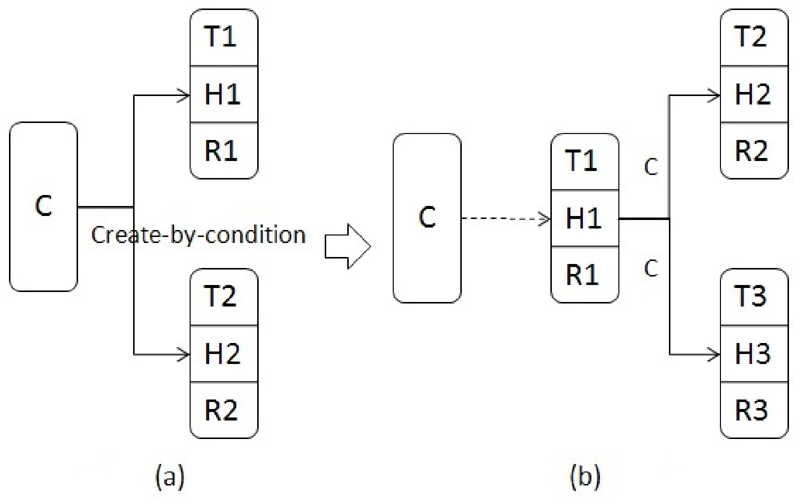

Figure 4 – ReP graph showing (a) intent-aspect of Abstract factory pattern and (b) its corresponding resultant ReP graph

Projects/ Design Pattern	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10
`Chess`			1	1						
`Astro`					2					
`Jnet-library`			4	1			1	1	2	
`Notepad`	1	1	2		1		1			
`Free Framework`	1	2				2	2			1
`Notepad++`		2	2	11	5		2	1	1	

Table 3 – Summary of analysis

The tool reported various applicable patterns at appropriate places. The summary of the analysis is presented in Table 3. Please note that patterns D1 to D10 in the table refer to abstract-factory, composite, state, strategy, observer, bridge, facade, builder, singleton and decorator design patterns respectively.

The tool may produce false positive/negative instances occasionally due to semantics of the source code. In order to observe false negative/positive instances; results reported by the tool are verified by manual inspection. Most of patterns inferred manually coincide with patterns inferred by the tool. A summary of inferred patterns and corresponding false negative/positive instances are listed in Table 4.

Table 4 shows that one instance each of state and singleton pattern are reported as false-negative. This can be justified as follows:

- The inference-rule for the state pattern conveys that a condition variable of a switch-case block (or if-elseif ladder) should be modified within every conditional case. In one specific instance, the condition variable changed within all but one conditional cases. Therefore, it is not reported by the tool. However, application semantics suggest applicability of the state pattern.

```
<CandidateSpot>
  <InferredPattern>Abstract Factory </InferredPattern>
  <InferredSite>
    <class>Display</class>
    <function>CreateProducts</function>
  </InferredSite>
  <Roles>
    <Role>
      <RoleName>Client</RoleName>
      <Class>Display</Class>
    </Role>
    <Role>
      <RoleName>Product</RoleName>
      <Class>Window</Class>
    </Role>
    <Role>
      <RoleName>Product</RoleName>
      <Class>Button</Class>
    </Role>
  </Roles>
</CandidateSpot>
```

Listing 3 – XML output of a candidate spot

- The singleton pattern intent states that if all methods of a class are static and the member variable count is not zero then the class is a candidate for a singleton pattern. During the manual analysis; a class was found with all but one static method. The tool did not report this class as a singleton candidate because of the non-static method. However, a deeper look revealed that the non-static method was an obsolete method and not called from the rest of the code. Therefore the class is a candidate for the singleton pattern.

Similarly, there are false-positive instances of the observer and strategy pattern. This can be reasoned as follows:

- The inference-rule of the observer pattern suggests that if two or more objects are interested in a change of one subject then the observer pattern can be implemented. In this case, there are two observers for a subject; hence, the tool inferred observer pattern. But, the source code semantics suggest that the number of observers is very unlikely to increase. Therefore, the instance can be considered as a false-positive inference.
- According to the strategy pattern intent, if every case of a conditional block has number of statements greater than a threshold value; it can be reported as a strategy pattern candidate. In one particular instance, the number of such conditional cases is greater than 15. The tool identified the instance as a candidate for strategy pattern. But introducing strategy pattern in such cases may lead to a subclass explosion (in this case more than 15 classes need to be introduced). Therefore the strategy pattern might not be the most suitable pattern in this case.

Design Pattern	Total number of inferred instances	False positive instances	False negative instances
`Abstract Factory`	2		
`Composite`	5		
`State`	9		1
`Strategy`	13	1	
`Observer`	8	1	
`Bridge`	2		
`Facade`	6		
`Builder`	2		
`Singleton`	3		1
`Decorator`	1		

Table 4 – Number of `false positive/negative instances` in analyzed projects

From the above discussion, we can deduce that false positive/ negative instances reported by the tool are well justified. In summary, the tool is able to infer design patterns with reasonable efficiency.

5 Related work

Many prior attempts have been made to refactor a given software system by introducing design patterns. Some of the closely related attempts are JIAD [RJ04], SQPR [SS03], [HHHL03], [GAA01], [JLB02] and [KJ09].

SPQR [SS03] defined *elemental design patterns* as a base concept on which more complex and large design patterns can be built. Their approach focuses on identification of design patterns and their isotopes, while our approach focuses on inference of design patterns. A similar attempt is made by Heuzeroth et.al. [HHHL03], they proposed a method to detect design pattern instances automatically using static and dynamic analysis.

An approach to automatically discover distorted forms of a design pattern is proposed by Yann et.al. [GAA01]. Their approach is applicable if patterns are employed in a distorted form, while our approach can infer patterns if intent-aspect of patterns exist within a given source code.

Sang-Uk et.al. [JLB02] proposed an automated approach to refactor a given source code in which an inference-rule and refactoring strategy is defined for each of the supported design patterns. Their strategy assumes that the need to introduce a design pattern arises when a design spot evolves frequently. Hence, their strategy compares two versions of a program to identify a candidate spot. This can result into a high number of false-negative instances, if some design deficiencies exist in the original design itself and most of the design is not evolving frequently. Our approach does not make such assumption; thus is free from such deficiencies.

An inference strategy is proposed by Vinay et.al. [KJ09] to introduce design pattern abstractions in C code. The strategy proposed inference-rules specific to a procedural language, therefore is not suitable for object oriented software systems.

The most similar work from literature is JIAD [RJ04]. The proposed work outperforms JIAD and related attempts, which can be argued as follows:

1. There are multiple ways to implement an intended logic. Therefore, a rule written to identify an intent-aspect should be able to adapt itself with the implementation. The ReP graph constructs abstract the implementation details so that a user can write rules without worrying about low level implementation logic. For example, the *condition* construct abstracts all type of supported condition statements (e.g., if and switch); the *create-within-condition* construct abstracts all create statements (such as new, malloc) within all condition statements. Further, provided abstractions work beyond a local scope (e.g., a variable can be set in a condition block directly by an assignment statement or by a setter method). These multiple ways to set a variable is abstracted by the *updated-var-list-by-condition* construct.

   Apart from facilitating a user to write rules without worrying about low-level implementation details; the abstraction increases the cover of inference-rules. This extended cover results in a less number of false-negatives. This abstraction is missing from JIAD and other similar attempts.

2. There might be multiple intent-aspects for a design pattern. Normally, a default inference-rule is provided, which covers the most usual case. But, while observing the source code a user may figure out other intent-aspects for the design pattern. The proposed work offers fundamental constructs and an API set to write additional inference-rules easily. It enables the user to write alternate inference-rules for a design pattern; thus increases the possibility to find more pattern candidates than JIAD.

3. Project specific settings (such as classes to exempt from analysis) can be specified using the proposed work. This feature not only avoids extra processing but also may avoid the possibility of false-positives.

6 Contributions

The contributions from the proposed work can be summarized as below:

1. The ReP graph introduces a new abstraction of source code. It captures relevant information in the form of fundamental constructs; leaving out the voluminous details of the source code, which eases the design pattern inferring process.

2. The proposed solution provides a framework for inferring design patterns. The framework maintains a rule-base of inference-rules. Each inference-rule maps a combination of fundamental constructs to a design pattern intent-aspect. A set of accessor APIs is provided to infer a pattern using the ReP graph. The inference-rules are written using these accessor APIs.

3. ReP graph captures the essence of patterns in terms of design structures and their interactions. Design structures can be considered as building blocks of design patterns. Therefore, they can be used not only to infer patterns but also to detect the existing patterns.

4. Extensibility is another important aspect; the proposed approach supports inclusion of inference-rules for additional patterns.

7 Conclusions

In summary, the ReP graph captures the relevant information in the form of fundamental constructs leaving out unnecessary voluminous details of a given source code. The information stored in these constructs is accessible by a set of accessor APIs provided with the ReP graph. Generation of ReP graph and detecting candidate spots for design patterns can be achieved using the proposed technique. The set of acessor APIs can be used to specify the intent-aspect of any design pattern. New rules for patterns can be written easily using the provided set of accessor APIs and can be added to the inference-rule base.

The ReP graph can be used to measure the quality attributes of an object oriented software system. The quality attributes before and after the code transformation can be compared to evaluate the effectiveness of the transformation. The proposed work can also be used to establish goal oriented quality measures.

References

[Ast] Astro. `http://mhuss.com/AstroLib/AstroCpp.zip`.

[Che] Chess. `https://www.pscode.com/vb/scripts/ShowCode.asp?txtCodeId=6337&lngWId=3`.

[Cod] Code-inspector. Siemens Corporate Research Princeton, USA.

[Fre] Free framework. `http://www.ebleda.com/opensource/ffw.php`.

[GAA01] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, page 296, Washington, DC, USA, 2001. IEEE Computer Society.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[HHHL03] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

[JAG+00] D. Janakiram, K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S. V. G. K. Raju, and A. Ananda Rao. An approach for pattern oriented software development based on a design handbook. *Ann. Softw. Eng.*, 10(1-4):329–358, 2000.

[JLB02] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in java programs. In *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, page 337, Washington, DC, USA, 2002. IEEE Computer Society.

[Jne] Jnet library. `http://www.nullsoft.com/free/jnetlib/`.

[K.05] Joshua K. *Refactoring to Patterns*. Addison-Wesley, 2005.

[KJ09] Viany Kumar Reddy K and D. Jankiram. Design pattern abstraction in c. Technical report, Technical report IITM-CSE-DOS-2006-09.

[MS05] Rajashree MS. Quality estimation model for software development. Technical report, Ph.D. thesis,2005.

[Nota] Notepad. `https://www.pscode.com/vb/scripts/ShowCode.asp?txtCodeId=840&lngWId=3`.

[Notb] Notepad++. `http://notepad-plus.sourceforge.net/`.

[ON99] M. O'Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 463, Washington, DC, USA, 1999. IEEE Computer Society.

[RJ04] J. Rajesh and D. Janakiram. Jiad: a tool to infer design patterns in refactoring. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 227–237, New York, NY, USA, 2004. ACM.

[SS03] Jason McC. Smith and David Stotts. Spqr: Flexible automated design pattern extraction from source code. *Automated Software Engineering, International Conference on*, 0:215, 2003.

About the authors

Tushar Sharma is currently pursuing Master of Science (MS) by research under Prof. D. Janakiram from Distributed and Object Systems Lab, Department of Computer Science and Engineering, Indian Institute of Technology (IIT), Madras.

His research interests include object oriented design, design patterns and refactoring. His MS thesis is focusing on design structures and their use in refactoring object oriented software systems. He can be reached at `000.tushar@gmail.com`.

Dharanipragada Janakiram is currently a professor in the Department of Computer Science and Engineering, Indian Institute of Technology (IIT), Madras, India.
He obtained his Ph.D degree from IIT, Delhi. He heads and coordinates the research activities of the Distributed and Object Systems Lab at IIT Madras. He has published over 30 international journal papers and 60 international conference papers and edited 5 books. His latest book on Grid Computing has been brought out by Tata Mcgraw Hill Publishers in 2005. He served as program chair for 8th International Conference on Management of Data (COMAD). He is the founder of the Forum for Promotion of Object Technology, which conducts the National Conference on Object Oriented Technology (NCOOT) and Software Design and Architecture (SoDA) workshop annually. He is the principal investigator for a number of projects which include the grid computing project from Department of Science and Technology, Linux redesign project from Department of Information Technology, Middleware Design for Wireless Sensor Networks from Honeywell Research Labs and A Mobile Data Grid Framework for Telemedicine from Intel Corporation, USA.
He has taught courses on distributed systems, software engineering, object-oriented software development, operating systems, and programming languages at graduate and undergraduate levels at IIT, Madras. He is a consulting engineer in the area of software architecture and design for various organizations. His research interests include distributed and grid computing, objects technology, software engineering, distributed mobile systems and wireless sensor networks, and distributed and object databases. He is a member of the IEEE, the IEEE Computer Society, the ACM, and a life member of the Computer Society of India.
He can be reached at `djram@iitm.ac.in`. See also `www.cs.iitm.ac.in/~djram`.