

Practical pattern-based design recovery approach

Huang Heyuan Zhang Shensheng Cao Jian Duan Yonghong

(Department of Computer Science and Engineering, Shanghai Jiaotong University, Shanghai 200030, China)

Abstract: A practical approach is presented to enlarge the recoverable scope and improve precision ratio of pattern recovery. To specify both structural aspects and behavioral aspects of design patterns, we introduce traditional predicate logic combined with Allen’s interval-based temporal logic as our theoretical foundation. Moreover, we take the Visitor pattern as an example to illustrate how to specify design patterns to support recovery of design patterns besides structural category. The experimental results show that the approach presented is practical for recovering design information of real world systems.

Key words: design pattern; pattern recovery; reverse engineering

While the merit of using design patterns for forward engineering is clear, we could also benefit from design pattern recovery in program understanding and reverse engineering. Some literature has addressed the recovery of design patterns. However, the recoverable patterns of these approaches are very limited, and most of these patterns belong to the structural category^[1-3]. Typically, their approaches would produce many false positives. Some researchers have tried to detect more instances of a pattern than approaches strictly relying on pattern structures^[4-6]. Their works are based on static analyses of source code files; however, detecting and deciphering interactions of objects in the source code is not easy: polymorphism makes it difficult to determine which method is actually executed at run time, and inheritance means that each object in a running system exhibits behavior which is defined not only in its class, but also in each of its superclasses.

In this paper, we present an approach based on both structural and behavioral analysis to enlarge the recoverable scope and improve the precision ratio of pattern recovery. In section 1, we propose the structural and behavioral constructs, which are the foundations of our approach. To characterize run-time behaviors, we introduce Allen’s interval-based temporal logic as our theoretical basis^[7,8]. In section 2, we use the Visitor pattern as an example to illustrate how to specify design patterns to support the

recovery of design patterns besides structural category^[9]. Finally, we summarize the contributions and limitations of our work and outline ideas for future work.

1 Fundamental Constructs

Basically, we can describe two aspects of a software system: structure and behavior. Similarly, we can also depict patterns from these two aspects.

Because the structural aspect is well understood, we do not give detailed definitions of structural constructs. Tab.1 gives the formal expressions of structural constructs.

Tab.1 Formal expressions of structural constructs

Construct	Formal expression
Class	Class (modifiers, name)
Interface	Interface (name)
Attribute	Attribute (modifiers, owner, name, type, decClass)
Operation	Operation (modifiers,owner,name,paraTypeSet,retType,decClass)
Constructor	Constructor (modifiers, owner, paraTypeSet, decClass)
Dependency	Dependency (client, supplier, op, depType)
Generalization	Generalization (sup, sub)
Association	Association (source, target, attr, multi)
Realization	Realization (inter, reali)

We classify behavioral constructs into three categories: basic behavioral constructs, event constructs, and message constructs. Behaviors are dynamic and time-dependent. To address the temporal aspect, we introduce Allen’s interval-based temporal logic^[7,8]. The typical predicates form in Allen’s temporal logic is as follows:

HOLDS (p , t), which means the property p holds during the time interval t .

OCCUR (e , t), which means the event e occurred over the time interval t .

There are 13 distinct elementary relations between two intervals. However, in this paper, we only mention one elementary interval relation: $<$ (before).

Received 2003-06-27.

Foundation items: The National High Technology Research and Development Program of China (863 Program) (No.2001AA415310 and 2002AA411420), and the National Natural Science Foundation of China (No.60073035).

Biographies: Huang Heyuan (1977—), male, graduate, heyuanhuang@yahoo.com.cn; Zhang Shensheng (corresponding author), male, doctor, professor, sszhang@sjtu.edu.cn.

For example, $t < s$ means interval t is before s .

1.1 Basic behavioral constructs

Basic behavioral constructs include Object and OAttribute. The definitions of these constructs are as follows.

Definition 1 Object

An object can be defined as a 2-tuple: Object (cl, obj), where cl denotes the class that the object belongs to, obj denotes the identification of the object.

Definition 2 OAttribute

An instantiated attribute can be defined as 3-tuple: OAttribute (owner, name, val), where owner denotes the object that the instantiated attribute belongs to, name denotes the name of the original attribute, val denotes the value that has been set.

1.2 Event constructs

There are five kinds of event constructs: Enter, Exit, Signal, Allocate, and Free. The following are the definitions of these constructs.

Definition 3 Enter event

An enter event can be defined as follows: OCCUR (Enter (enObj, enteredOp, paraSet), t), where enObj denotes the object to which the entered operation belongs, enteredOp denotes the operation which is entered, paraSet denotes the parameters which are set, and t denotes the time interval over which the event occurred.

Definition 4 Exit event

An exit event can be defined as follows: OCCUR (Exit (exObj, exitOp, retValue), t), where exObj denotes the object to which the exited operation belongs, exitOp denotes the operation which is exited, retValue denotes the exit value, and t denotes the time interval over which the event occurred.

Definition 5 Signal event

A signal event can be defined as follows: OCCUR (Signal (sig), t), where sig denotes the signal which is sent, t denotes the time interval over which the event occurred.

Definition 6 Allocate event

An allocate event can be defined as follows: OCCUR (Allocate(createdCl, paraSet, retObj), t), where createdCl denotes the class to which the created object belongs, paraSet denotes the parameters which are set, retObj denotes the object which is allocated, t denotes the time interval over which the event occurred.

Definition 7 Free event

A free event can be defined as follows: OCCUR (Free (freeObj), t), where freeObj denotes the object

which is free, t denotes the time interval over which the event occurred.

1.3 Message constructs

As defined in UML^[10], there are five kinds of message: Call, Return, Send, Create, and Destroy. While these messages cannot be recovered directly from run-time analysis, they can be constructed based on the aforementioned event constructs. Because of limited space, we only give a formal definition of the Call message in this section.

Definition 8 Call message

A call message can be defined as follows: OCCUR (Call (caller, callOp, callee, calleeOp, paraSet, enterT), msgT), where caller denotes the object which makes the call, callOp denotes the location where the calleeOp is called, callee denotes the object to which the calleeOp belongs, calleeOp denotes the operation which is called, paraSet denotes the parameters which are set, enterT denotes the time interval over which the enter event of callOp occurred, msgT denotes the time interval over which the enter event of calleeOp occurred, i.e., the time interval over which the message is sent.

Actually, the Call message can be defined according to the aforementioned event constructs. The following is its formal definition:

$$\begin{aligned} \text{OCCUR (Call(caller, callOp, callee, calleeOp, paraSet, enterT), msgT)} = & \\ \text{OCCUR (Enter(caller, callerOp, _, _), enterT)} \wedge & \\ \text{OCCUR (Enter(callee, calleeOp, paraSet), msgT)} \wedge & \\ \text{EventPair (enterT, exitT)} \wedge \text{enterT} < \text{msgT} \wedge \text{msgT} < \text{exitT} \wedge & \\ \neg \exists \text{enterT}' \cdot (\text{OCCUR(Enter(_, _, _), enterT')} \wedge & \\ \text{EventPair (enterT}', \text{exitT}')} \wedge \text{enterT}' < \text{enterT}' \wedge & \\ \text{the enterT}' < \text{msgT} \wedge \text{msgT} < \text{exitT}') & \end{aligned}$$

In the predicated EventPair (enterT, exitT), enterT is the time interval over which an enter event occurred, exitT is the time interval over which an exit event occurred. “EventPair (enterT, exitT) is true” means that these two events correspond to the enter event and exit event of the same operation invocation. Similarly, enterT' and exitT' represent time intervals over which an enter event and exit event of any operation invocation occurred.

A single underscore refers to an anonymous variable, which appears only once in a clause and from an implementation viewpoint need not be named.

Definition 9 Return message

A return message can be defined as follows: OCCUR (Return(returner, returnOp, acceptor, acceptOp, retVal, enterT), msgT), where returner denotes the object to which the returnOp belongs, returnOp denotes the operation which is returned, acceptor denotes the object to which the message returns,

acceptOp denotes the location where the message returns to, retVal denotes the value which is returned, enterT denotes the time interval over which the enter event of acceptOp occurred, msgT denotes the time interval over which the exit event of returnOp occurred, i.e., the time interval over which the message is sent.

Definition 10 Send message

A send message can be defined as follows: OCCUR (Send (sender, sendOp, sig, enterT), msgT), where sender denotes the object which sends the message, sendOp denotes the location where the signal is sent, sig denotes the signal which is sent, enterT denotes the time interval over which the enter event of sendOp occurred, msgT denotes the time interval over which the signal event occurred, i.e., the time interval over which the message is sent.

Definition 11 Create message

A create message can be defined as follows: OCCUR (Create (creator, createOp, createdCl, paraSet, retObj, enterT), msgT), where creator denotes the object to which the createOp belongs, createOp denotes the location where the retObj is created, createdCl denotes the class to which the created object belongs, paraSet denotes the parameters which are set, retObj denotes the object which is created, enterT

denotes the time interval over which the enter event of createOp occurred, msgT denotes the time interval over which the allocation event of retObj occurred, i.e., the **time interval over which the message is sent**.

Definition 12 Destroy message

A destroy message can be defined as follows: OCCUR (Destroy (destroyer, destroyOp, destroyedObj, enterT), msgT), where destroyer denotes the object to which the destroyOp belongs, destroyOp denotes the location where destroyedObj is destroyed, destroyedObj denotes the object which is destroyed, enterT denotes the time interval over which the enter event of destroyOp occurred, msgT denotes the time interval over which the destruction event of destroyedObj occurred, i.e., the time interval over which the message is sent.

2 Pattern-Based Design Recovery

In the following, we take the Visitor pattern as an example to illustrate how to specify design patterns to support the recovery of design patterns besides structural category^[10].

The Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called double-dispatch. Fig.1 is the structure of the Visitor pattern.

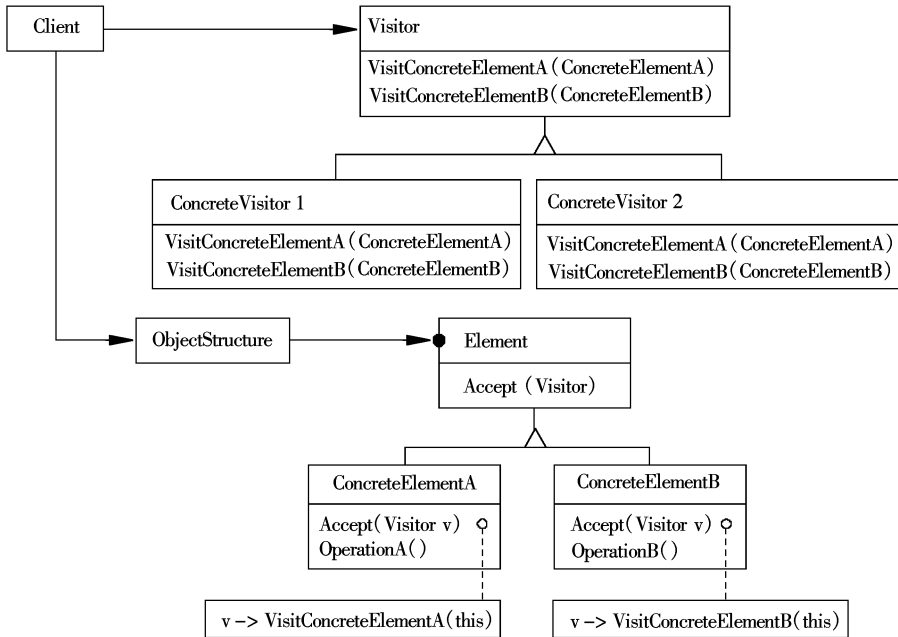


Fig.1 Structure of the Visitor pattern

Double-dispatch simply means the operation that gets executed depends on the kind of request and the types of two receivers. Accept is a double-dispatch operation. Its meaning depends on two types: the Visitor's and the Element's. Double-dispatching lets visitors request different operations on each class of

an element. Instead of binding operations statically into the Element interface, you can consolidate the operations in a Visitor and use Accept to do the binding at run-time.

The following is the formal specification of the Visitor pattern in our approach.

```

Visitor (visitor, concreteVisitor, element, concreteElement) ←
  Ancestor (visitor, concreteVisitor) ∧
  Ancestor (element, concreteElement) ∧
  Dependency (visitor, element, visitElement, 'depParameter') ∧
  Dependency (element, visitor, acceptVisitor, 'depParameter') ∧
  ∀ enterT · (Object(concreteElement, concreteElementObj) ∧
    OCCUR (Enter (concreteElementObj, acceptVisitor,
      {concreteVisitorObj}), enterT) →
    Object (concreteVisitor, concreteVisitorObj) ∧
    OCCUR (Call (concreteElementObj, acceptVisitor,
      concreteVisitorObj, visitElement, -, enterT), -))

```

From the above example, we can see that, in our approach, we can capture not only structural patterns but also creational patterns and behavioral patterns. Moreover, the specifications are easy to be implemented by logical programming languages such as Prolog to facilitate pattern recovery.

Some researchers have also tried to detect more instances of patterns by analyzing the system's behaviors. For example, Seemann has used a compiler to collect information about inheritance hierarchies and method call relations^[6]. SPOOL is an environment for the reverse engineering of design components based on the structural descriptions of design patterns^[4]. They all rely on the analysis of source code. Thus, their approaches cannot detect run-time interactions of objects. However, double-dispatches can only be captured at run time. So, we can see that their approaches would produce many false positives.

We have developed a tool named PRAssistor and taken two well-known open source frameworks as examples to evaluate our approach. One framework is JUnit, which is a unit testing framework written by Erich Gamma and Kent Beck. Another framework is JHotDraw, which is a GUI framework for building graphical drawing editor applications. Both frameworks are very mature and have high pattern density. Both recall ratio and precision of JUnit are 100%. The recall ratio of JHotDraw is 78.57%, and the precision of JHotDraw is 91.67%. The experimental results show that our approach is practical and useful for recovering design information of real world systems.

3 Conclusion

This paper presents a practical approach to support pattern recovery. While most other approaches are only based on the analysis of source code, our approach also analyzes dynamic interaction in the system at run time. By this method, we can identify more instances of patterns besides structural patterns and improve the precision of pattern recovery.

However, there still exist some limitations to our approach. Firstly, we have not provided an effective

mechanism to manage the implementation variants of pattern instances. Sometimes developers will choose implementation variants in their systems. Secondly, we have not exploited domain and context knowledge. By now, we have only considered the solution part of design patterns because of the limited information provided by source code and run-time behaviors.

In our future work, we will try to overcome the aforementioned shortcomings in our approach. For example, to identify implementation variants of pattern instances, we will enrich our rule library and provide a general approach to depict variants. Moreover, we will try to introduce domain and context knowledge in a formal or semi-formal way.

References

- [1] Antoniol G, Casazza G, Di Penta M, et al. Object-oriented design patterns recovery [J]. *Journal of Systems and Software*, **2001**, *59*(2): 181 – 196.
- [2] Brown K G. Design reverse-engineering and automated design pattern detection in small talk [D]. North Carolina: North Carolina State University, 1996.
- [3] Kramer C, Prechelt L. Design recovery by automated search for structural design patterns in object-oriented software [A]. In: *Proceedings of the Third Working Conference on Reverse Engineering* [C]. Monterey, CA: IEEE Computer Society Press, 1996. 208 – 215.
- [4] Keller R K, Schauer R, Robitaille S, et al. Pattern-based reverse-engineering of design components [A]. In: *Proceedings of the 21st International Conference on Software Engineering* [C]. Los Angeles, CA: IEEE Computer Society Press, 1999. 226 – 235.
- [5] Niere J, Schafer W, Wadsack J P, et al. Towards pattern-based design recovery [A]. In: *Proceedings of the 24th International Conference on Software Engineering* [C]. Orlando, Florida: IEEE Computer Society Press, 2002. 338 – 348.
- [6] Seemann J, von Gudenberg J W. Pattern-based design recovery of Java software [A]. In: *Proc of 6th International Symposium on the Foundation of Software Engineering, ACM SIGSOFT 6* [C]. Lake Buena Vista, Florida: ACM Press, 1998. 10 – 16.
- [7] Allen J F. Maintaining knowledge about temporal intervals [J]. *Communications of the ACM*, **1983**, *26*(11): 832 – 843.
- [8] Allen J F. Towards a general theory of action and time [J]. *Artificial Intelligence*, **1984**, *23*(2): 123 – 154.
- [9] Gamma E, Helm R, Johnson R, et al. *Design patterns: elements of reusable object-oriented software* [M]. Massachusetts: Addison Wesley Publishing Company, 1994. 416.
- [10] Booch G, Rumbaugh J, Jacobson I. *The unified modeling language user guide* [M]. MA: Addison Wesley Longman, 1999. 482.

一种实用的基于模式的设计重现方法

黄鹤远 张申生 曹 健 段永红

(上海交通大学计算机科学与工程系, 上海 200030)

摘要: 提出了一种实用的方法来扩大可重现模式的范围和提高模式重现的精度. 为了对设计模式的结构及行为 2 方面进行刻画, 本文将传统的谓词逻辑与 Allen 的基于时段的时态逻辑相结合作为理论基础, 并以 Visitor 模式为例说明了如何采用该方法刻画和重现结构型之外的其他设计模式. 实验结果表明该方法和工具可以有效地重现真实环境中软件系统的设计信息.

关键词: 设计模式; 模式重现; 反向工程

中图分类号: TP311.5