Case Studies: A Metamodeling Approach for Pattern Specification and Management

Liang Dou, Qiang Liu and Zongyuan Yang

Department of Computer Science and Technology, East China Normal University, Shanghai, China,
ldou@cs.ecnu.edu.cn, lqiangecnu@gmail.com, yzyuan@cs.ecnu.edu.cn

This paper presents case studies that apply our metamodeling approach for pattern specification and management to some classic GOF patterns. Generally speaking, GOF patterns are divided into three categories: creational pattern, structural pattern and behavioral pattern. One representative pattern is chosen from each category and studied in detail. Specifically, we formalize the Factory method pattern, the Composite pattern and the Mediator pattern. For the Mediator pattern, due to its significant behavioral features, a set of dynamic relations are also defined to capture its behavior.

1 Factory method pattern

1.1 Pattern specification

The Factory method pattern is a typical creational pattern. An abstract class Creator defines an interface for creating products. However, which concrete product to create is determined by its concrete creators. The formalization of this pattern in Kermeta is straightforward. Firstly, we define an EClass `FactoryMethodPattern` and identify its participants and model them as EReferences. The invariant is defined by reusing primitive and complex relations. Fig.1 shows its EReferences and invariant.

1.2 Pattern management

Secondly, based on the pattern specification, `CreateInitStruct` is also defined to instantiate the pattern, in which singular participants are assigned with newly created UML elements(see Fig.2).

For pattern evolution, two evolving operations are also defined. `AddConCreator` adds a concrete creator and `AddConProduct` adds an concrete product. Due to space restriction, only the definition of `AddConCreator` is shown in Fig.3. The final parameter `ThePdt` determines the concrete product that concrete creator `conCrt` depends on.

For pattern implementation, since it is a creational pattern with no significant behavioral features, no dynamic relation is defined. However, the implementation of factory method of each concrete creator can be generated by a single template. Intuitively, it should return an instance of corresponding concrete product. Therefore, the template is parameterized by the name of the concrete product

```
class FactoryMethodPattern inherits DesignPattern{
    reference Creator:uml::Class[1..1]
    reference ConCrts:uml::Class[0..*]
    reference Product:uml::Class[1..1]
    reference ConPdts:uml::Class[0..*]
    reference FactoryMethod:uml::Operation[1..1]
    reference ConFatMtds:uml::Operation[0..*]
    reference ConCrtsDepConPdts:uml::Dependency[0..*]
inv spec is do
    IsAbstract(Creator) and
    IsAbstract(Product)  and
    Hierarchy(ConCrts, Creator) and
    Hierarchy(ConPdts, Product) and
    HasOperation(Creator, FactoryMethod) and
    IsAbstractOp(FactoryMethod) and
    HasOpOtO(ConCrts, ConFatMtds) and
    HasDepOtOtO(ConCrts, ConCrtsDepConPdts, ConPdts) and
    ReturnType(FactoryMethod, Product)
end  }
```

Fig. 1. Specification of the Factory method pattern

```
method CreateInitStruct():Void is
do
    super()
    Creator:=createClass(umlModel,"Creator",true)
    Product:=createClass(umlModel,"Product",true)
    FactoryMethod:=createOperation(Creator,"FactoryMethod",true)
    setRetType(FactoryMethod,Product)
end
```

Fig. 2. Instantiation of the Factory method pattern

```
operation AddConCreator(conCrt:Class,confm:Operation,ThePdt:Class):Void
do
  umlModel.packagedElement.add(conCrt)
  ConCrts.add(conCrt)
  conCrt.ownedOperation.add(confm)
  setRetType(confm, Product)
  ConFatMtds.add(confm)
  createGeneralization(conCrt, Creator)
  var newdep:uml::Dependency init uml::Dependency.new
  newdep:=createDependency(umlModel,conCrt,ThePdt)
  ConCrtsDepConPdts.add(newdep)
end
```

Fig. 3. Evolution of the Factory method pattern: add a concrete creator

`ConProName`, whose value is obtained by two helper functions: `GetClass` and `GetDependsOn`. Since we have documented every type of elements involved in a pattern, these helper functions can efficiently explore a pattern structure. For example, the function `GetDependsOn` iterates all the `Dependency` elements in the set `ConCrtsDepConPdts` and finds the depending class of the given class . The Kermeta codes of generating the implementation is shown in Fig.4.

```
method CreatePatternSpecification() : Void is
do
  // Iterate all factory methods in set ConFatMtds
  ConFatMtds.each { conFM |
      var owningclass:uml::Class init uml::Class.new
      // Returns the class that owns the given operation
      owningclass := GetClass(conFM, ConCrts)
      var depclass:uml::Class init uml::Class.new
      // Get the depending class
      depclass := GetDependsOn(owningclass)
      var ConProName:String
      ConProName := depclass.name
      //  The template
      theImp := { return new ConProName(); }
      //  Set the implementation as conFM's body
      CreateSpec(conFM, "Body", theImp)  }
end
```

Fig. 4. Implementation of the Factory method pattern

As an example, by executing the code in Fig.5, an instance of FactoryMethodPattern is generated as in Fig.6, with two concrete products and two concrete factories. The interesting Java code snippets for this instance is shown in Fig.7.

```
var AFMPat:FactoryMethodPattern init FactoryMethodPattern.new
AFMPat.CreateInitStruct()
var newpdt1:uml::Class init uml::Class.new
 newpdt1:=AFMPat.AddConProductE("MyProd1")
var newpdt2:uml::Class init uml::Class.new
 newpdt2:=AFMPat.AddConProductE("MyProd2")
AFMPat.AddConCreatorE("MyCreator1",void,newpdt1)
AFMPat.AddConCreatorE("MyCreator2",void,newpdt2)
AFMPat.CreatePatternSpecification()
AFMPat.checkInvariants
AFMPat.save("FactoryMethod")
```

Fig. 5. Example codes for usage of the Factory method pattern

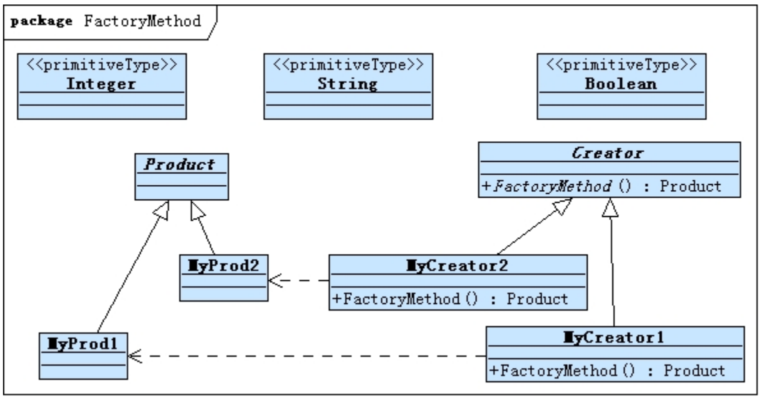

Fig. 6. A generated instance of the Factory method pattern

```
public class MyProd1 extends  Product {}
public class MyProd2 extends  Product {}
public class MyCreator1 extends  Creator {
  // MyCreator1's factory method creates an MyProd1 instance
  public  Product FactoryMethod()
  { return new MyProd1();}
}
public class MyCreator2 extends  Creator {
  // MyCreator2's factory method creates an MyProd2 instance
  public  Product FactoryMethod()
  { return new MyProd2();}
}
```

Fig. 7. Factory method pattern: generated Java code snippets

2 Composite pattern

2.1 Pattern specification

The Composite pattern is a frequently used structural pattern, which could "compose objects into tree structures". The most significant feature of this pattern is that a composite component inherits from an abstract component and also maintains a set of abstract component as its children, which again could be composite components. Our approach is able to handle this kind of structure. The Composite pattern is modeled as an EClass in Fig.8, in which only the most interesting EReference are shown. EReference `Component` defines an abstract interface for all components of the pattern. EReference `Composites` is a set of composite components. The tree structure is specified by two relations: (1) all the classes in `Composites` inherit from Component; (2) each class in `Composites` has an association with `Component`. The two relations are both formalized in the invariant.

```
class CompositePattern inherits DesignPattern
{
// The Component Part
reference Component:uml::Class[1..1]
// Operations of Component
......
// The Composite Part
reference Composites:uml::Class[0..*]
// Operations of Composites
......
// The set of Associations from Composites to Component
reference CpsAssCpn:uml::Association[0..*]
// The Leaves Part
reference Leaves:uml::Class[0..*]
inv spec is
do
  Hierarchy(Composites, Component) and
  Composites.forAll{com|CpsAssCpn.exists{asso|
    HasAssociation(com,asso, Component)}} and
  ......
end
}
```

Fig. 8. Specification of the Composite pattern

2.2 Pattern management

To instantiate an instance of the Composite pattern, `CreateInitStruct` is also defined in `CompositePattern`. Furthermore, evolving operation `AddComposite` adds a composite component and `AddLeaf` adds a leaf component. Their implementations are omitted here.

As a structural pattern, it is enough to generate standard implementations for pattern-level operations. The generation is also based on templates parameterized by variables. Values of variables are also obtained by exploring the pattern structure. `CreatePatternSpecification` iterates three operation sets `ComAdds`, `ComRemoves`, and `ComGetChilds` to generate implementations for `Add`, `Remove` and `GetChild` operation of each composite component. Fig.10 only shows templates for each set of operations and the Kermeta code is similar to the implementation of the Factory method pattern.

```
// Template for each operation in set ComAdds
this.ChildrenName.add(ParaName);
// Template for each operation in set ComRemoves
this.ChildrenName.remove(ParaName);
// Template for each operation in set ComGetChilds
return this.ChildrenName.get((ParaName);
```

Fig. 9. Implementation of the Composite pattern

By executing the codes in Fig.10, an instance of CompositePattern is generated and is shown in Fig.11, in which two composite components and two leaves are created. The corresponding Java code snippets are listed in Fig.12. When creating the pattern instance `AComPat`, we use different names for children set to demonstrate that values of variables in a template are dynamically acquired from the pattern structure. For example, in the generated Java code in Fig.12, value of variable `ChildrenName` is `children1` in class `MyComposite1` and `children2` in class `MyComposite2`. The process of getting the value of `ChildrenName` is convenient and efficient by using a set of predefined and pattern-independent helper functions. Again, these definitions are exploiting the fact that all the participants involved in a pattern, regardless of their types, are precisely documented as EReferences in our approach.

```
var AComPat:CompositePattern init CompositePattern.new
AcomPat.CreateInitStruct()
AComPat.AddCompositeE("MyComposite1","children1")
AComPat.AddCompositeE("MyComposite2","children2")
AComPat.AddALeafE("MyLeaf1")
AComPat.AddALeafE("MyLeaf2")
AComPat.CreatePatternSpecification()
AComPat.checkInvariants
AComPat.save("Composite1")
```

Fig. 10. Example codes for usage of the Composite pattern

3 Mediator pattern

The Mediator pattern is a behavioral pattern which "encapsulates how a set of objects interact" and "promotes loose coupling by keeping objects from referring to each other explicitly". Structurally speaking, the pattern is simple as in Fig.13. The behavioral intention of this pattern is significant. In [12], it claims that class `Mediator` should define an abstract interface for the communication of Colleagues. The communication protocol varies with the modeling domain, so

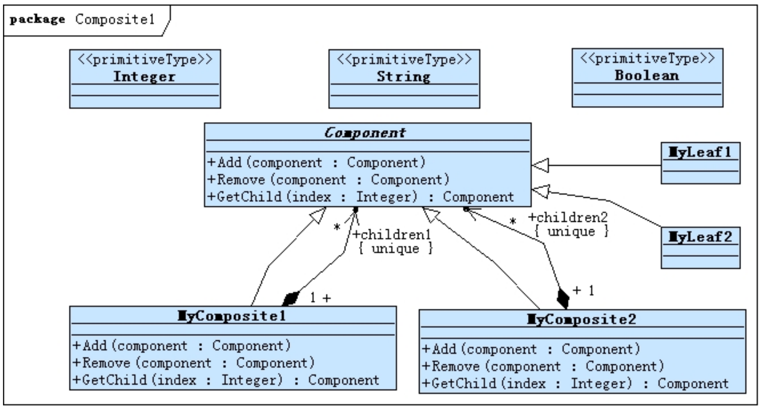

Fig. 11. A generated instance of the Composite pattern

```
// Code for MyComposite1
public class MyComposite1 extends  Component {
  public List<Component> children1;
  public  Component GetChild(Integer index)
      { return this.children1.get(index);}
  public  void Add(Component component)
      { this.children1.add(component);}
  public  void Remove(Component component)
      { this.children1.remove(component);}   }
// Code for MyComposite2
public class MyComposite2 extends  Component {
  public List<Component> children2;
  public  void Remove(Component component)
      { this.children2.remove(component);}
  public  void Add(Component component)
      { this.children2.add(component);}
  public  Component GetChild(Integer index)
      { return this.children2.get(index);}   }
```

Fig. 12. Composite pattern: generated Java code snippets

it is totally left undefined in the structure. However, as in previous work [3], one can define a particular (yet with great genericity) communication protocol to formalize the pattern behavior in a more precise way. This paper follows the communication interface defined in [3] and incorporates dynamic relations into the pattern specification. This case study demonstrates that it is possible to achieve the same formality in a more practical and usable environment.

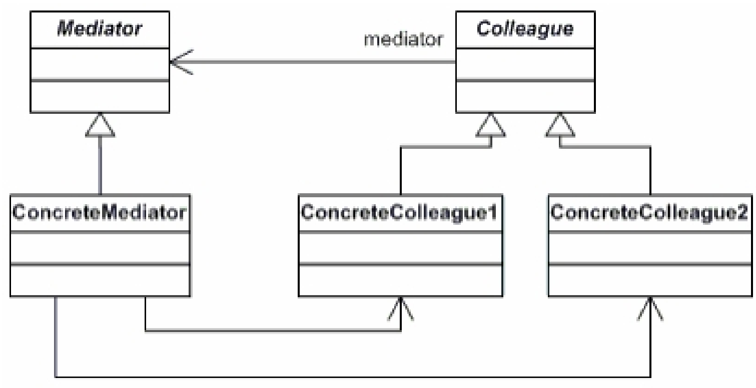

Fig. 13. The Mediator pattern in UML

To provide a general way of mediating the communication of concrete colleagues, a set of mailboxes is defined in as communication mediums. Each mailbox has a set of colleagues as its senders and receivers. If a colleague is a sender of a mailbox, it can put a message in the box. If it is a receiver of the mailbox, it can then read the message. In this sense, colleagues are not communicating directly, but in a mediated fashion. Furthermore, the structure of the message is undefined, which means one can communicate arbitrary complex messages through mailboxes. Fig.14 shows a generated instance of the Mediator pattern. Like previous cases, the instance is generated by the EOperations of instantiation and evolution. Since they are relatively simple, we omit their definitions. Here we only focus on the pattern Implementation.

In order to precisely capture the pattern behavior, three dynamic relations are defined. The first one is `Connected`. It records whether a colleague is under the mediation of a mediator. The abstract `Mediator` maintains a set of colleagues and the meaning of Connected relation could be interpreted as the membership of the set. Actually, `Connected` is similar to `Attached` relation of the Observer pattern. The second relation is `Sender` which records whether a colleague is able to put message into a mailbox. The meaning of this relation can be interpreted by the association between the class `MailBox` and `Colleague`, with one member end named as `senders`. Similarly, relation `Receiver` records whether a colleague can read message from a mailbox and its meaning is also interpreted by the association with one member end named as `receiver`.

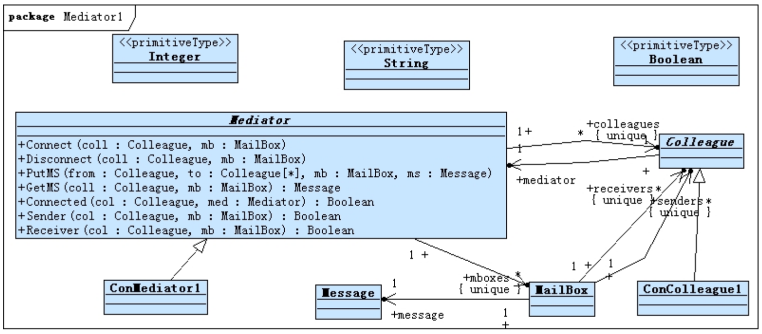

Fig. 14. A Mediator pattern instance

To change the truth value of these dynamic relations, four pattern-level operations are also defined. Operation `Connect` takes two parameters: `coll:Colleague` and `mb:MailBox`, which adds `coll` as a mediated colleague and set it as a sender of `mb`. Operation `Disconnect` does the opposite by removing `coll` from the mediation and canceling its identity as a sender of `mb`. Additionally, it deletes `coll` from any receiver set of mailbox. This makes sure that once a colleague is not connected to a mediator, it also can not read message from any mailbox. Operation `PutMS` takes four parameters: `from:Colleauge`, `to:Colleague[*]`, `mb:MailBox` and `ms:Message`, which means putting message `ms` from colleague `from` to a set of colleagues `to` in mailbox `mb`. The main task of the operation is to set `from` as a sender of `mb` and set all the element's in `to` as receivers of `mb`. Operation `GetMS` simply read message sent to `coll` from mailbox `mb`.

Similarly, by exploring corresponding EReferences of the Mediator pattern, it is possible to define templates of specifications and implementations for dynamic relations and pattern-level operations. Fig.15 shows templates of selected operations, in which variables are also italicized. Based on these templates, the generated JML specification and Java implementation of operation `Connect` and `PutMS` is presented in Fig.16, in which variables are replaced by corresponding values of the pattern instance in Fig.14.

By conducting these case studies, we believe our approach is capable of specifying design patterns in a precise and abstract way. Furthermore, based on our solutions for pattern-related problems, designers can instantiate, evolve and implement a pattern by writing only a few lines of codes.

References

1. Crane, M.L.: Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification. PhD thesis, Queen's University, January 2009
2. Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and System 15 (1), pp.73–132. (1993)

```
// Precondition for Connect operation
//@ requires !ConnectedName(ConPara1Name,this) &&
  !SenderName(ConPara1Name,ConPara2Name)&&
  this.mboxesEndName.contains(ConPara2Name);
// Post condition for Connect operation
//@ ensures ConnectedName(ConPara1Name,this) &&
   SenderName(ConPara1Name,ConPara2Name);
// Implementation for Connect operation
{ this.mboxesEndName.add(ConPara1Name);
  ConPara2Name.mboxesEndName.add(ConPara1Name); }
// Precondition for PutMs operation
//@ requires ConnectedName(PutPara1Name,this) &&
  (
    forall PutPara2TypeName tovar; PutPara1Name.contains(tovar) ==>
                           ConnectedName(tovar,this)) &&
  SenderName(PutPara1Name,PutPara3Name) &&
  this.mboxesEndName.contains(PutPara3Name)
// Postcondition for PutMs operation
//@ ensures (
  forall PutPara2TypeName tovar;
  PutPara1Name.contains(tovar)==>ReceiverName(tovar,PutPara3Name))&&
  PutPara3Name.messEndName==PutPara4Name;
        CreateSpec(PutMs, "Post", putmsPostSymbol)
// Implementation for PutMs operation
{ PutPara3Name.messEndName=PutPara4Name;
  for(Iterator<ColleagueName> itcol=PutPara2Name.iterator();itcol.hasNext();)
 { ColleagueName tempcol=itcol.next();
    PutPara3Name.receiverEndName.add(tempcol);}
}
```

Fig. 15. Implementation of the Mediator pattern

```
//JML Specifications for Connect
//@ requires !Connected(coll,this) && !Sender(coll,mb) && this.mboxes.contains(mb);
//@ ensures Connected(coll,this) && Sender(coll,mb);
//Java Implementation for Connect
public void Connect(Colleague coll,MailBox mb)
{ this.colleagues.add(coll);
  mb.senders.add(coll);
}
//JML Specifications for PutMS
//@ requires Connected(from,this) &&
 (
   forall Colleague tovar; to.contains(tovar)==>Connected(tovar,this)) &&
 Sender(from,mb) && this.mboxes.contains(mb) && mb.message==null;
//@ ensures (
              forall Colleague tovar; to.contains(tovar)==>Receiver(tovar,mb)) &&
  mb.message==ms;
// Java Implementation for PutMS
public void PutMS(Colleague from,List<Colleague> to,MailBox mb,Message ms)
{ mb.message=ms;
  for(Iterator<Colleague> itcol=to.iterator(); itcol.hasNext();)
 { Colleague tempcol= itcol.next();
    mb.receivers.add(tempcol);}
}
```

Fig. 16. Mediator pattern: generated Java code snippets

3. Mikkonen T.: Formalizing design pattern. In: Proceedings of the 20th International Conference on Software Engineering, pp.115–124. (1998)
4. Moriconi, M., Qian, X., Riemenschneider, R.A.: Correct architecture refinement. IEEE Transactions on Software Engineering 21 (4), pp.356–372. (1995)
5. Taibi, T., David, C.L.: Formal specification of design pattern combination using BPSL. International Journal of Information and Software Technology (IST) 45 (3), pp.157–170. (2003)
6. Eden, A.H., Hirshfeld, Y.: Principles in formal specification of object-oriented architectures. In: Proceedings of the 11th CASCON,Toronto,Canada (2001)
7. Eden,A.H., Hirshfeld,Y., Kazman,R.: Abstraction classes in software design. IEE Software, Vol. 153(4), pp. 163–182, (2006)
8. Jing, D., Alencar, P., Cowan, D.: Ensuring structure and behavior correctness in design composition. In: Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS), Edinburgh UK, pp.279–287. (2000)
9. Jing, D., Paulo, S.C., Alencar, P., Cowan, D., Sheng Y.: Composing pattern-based components and verifying correctness. The Journal of Systems and Software , pp. 1755–1769. (2007)
10. Jing, D.: Design component contracts: model and analysis of pattern-based composition. PhD Thesis, Computer Science Department, University of Waterloo. (2002)
11. Keller, R.K., Schauer R.: Design components: towards software composition at the design level. In: Proceedings of the 20th International Conference on Software Engineering, pp.302–311. (1998)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns:Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company. (1995)
13. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. Cambridge, MA: MIT Press. (2001)
14. Leavens,G.T., Baker,A.L., Ruby,C.: JML:a Java modeling language.In: Formal Underpinnings of Java Workshop at OOPSLA. (1998)
15. Mapelsden, D., Hosking, J., Grundy, J.: Design Pattern Modeling and Instantiation using DPML. In: Proceeding of TOOLS Pacific, Sydney, Australia. (2002)
16. Albin-Amiot, H., Gueheneuc, Y.G.: Metamodeling Design Patterns: Application to Pattern Detection and Code Synthesis. In: Proceedings of the ECOOP 2001 Workshop on Adaptative Object-Models and MetaModeling Techniques. (2001)
17. Mak,J.K.H., Choy,C.S.T., Lun,D.P.K.: Precise modeling of design patterns in UML. In Proceedings of the 26th International Conference on Software Engineering, pp.252C261. (2004)
18. The website of the Acceleo project: www.acceleo.org/pages/home/en
19. The website of Topcased: www.topcased.org
20. The website of EMF: www.eclipse.org/emf/
21. Zhao,C., Kong,J., Zhang,K.: Design pattern evolution and verification using graph transformation.In Proceedings of the 40th Annual Hawaii International Conferenceon System Sciences, Big Island, Hawaii, pp.290C297. (2007)
22. Dong,J., Yang,S., Zhang,K.: Visualizing design patterns in their applications and compositions. IEEE Transactions on Software Engineering, 33(7), pp.433C453.(2007)
23. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proceedings of OOPSLA, pp.161C173. (2002).
24. Soundarajan,N., Hallstrom,J.O.: Responsibilities and rewards: Specifying design patterns. In Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, 666C675(2004).