Specifying Behavioural Features of Design Patterns in First Order Logic

Dr Ian Bayley and Prof Hong Zhu,
Oxford Brookes University

30th July 2008
COMPSAC ’08, Turku, Finland
Purpose is to “capture design experience in a form that people can use effectively”
- eg for reusability, testability, modifiability (non-functional)

23 patterns in GoF book eg Template Method
- informal English plus indicative UML diagrams
- class diagrams for structural features
- sequence diagrams for behavioural features

Formal model of UML specified in GEBNF
- BNF Graphically Extended for references
- predicates induced to inspect model
- pattern is a first-order predicate on models
Example of a Class Diagram (Visitor)

Dr Ian Bayley and Prof Hong Zhu, Oxford Brookes University

Specifying Behavioural Features of Design Patterns in First Order Logic
ClassDiagram ::= 
   classes : Class^+, 
   assocs : Rel^*, inherits : Rel^*, CompAg : Rel^*

Rel ::= 
   [name : String], source, end : End

Class ::= 
   name : String, [attrs : Property^*], 
   [opers : Operation^*]
Operation ::= 
\[\text{name} : \text{String}, [\text{params} : \text{Parameter}^*], [\text{isQuery} : \text{Boolean}], [\text{isLeaf} : \text{Boolean}], [\text{isNew} : \text{Boolean}], [\text{isStatic} : \text{Boolean}], [\text{isAbstract} : \text{Boolean}]\]
Parameter ::= 
  [direction : ParameterDirectionKind],
  [name : String], [type : Type],
  [mult : MultiplicityElement]

ParameterDirectionKind ::= 
  "in" | "inout" | "out" | "return"

MultiplicityElement ::= 
  [upperValue : Natural | "∗"],
  [lowerValue : Natural]
Property ::= 
name : String, type : Type, [isStatic : Boolean],
[mult : MultiplicityElement]

End ::= 
node : Class, [name : String], [mult : MultiplicityElement]
Example of a Sequence Diagram (Visitor)

Dr Ian Bayley and Prof Hong Zhu, Oxford Brookes University

Specifying Behavioural Features of Design Patterns in First Order Logic
**Formalisation of Sequence Diagrams**

\[
\text{SequenceDiagram ::= lifelines : Lifeline}^{*}, \text{messages : Message}^{*},
\text{ordering : (Message, Message)}^{*}
\]

\[
\text{Lifeline ::= activations : Activation}^{*},
\text{className : String, [objectName : String]},
\text{isStatic : Boolean}
\]
Activation ::= 
    \text{start} : \text{Event}, \text{finish} : \text{Event}, \text{others} : \text{Event}^*$

Message ::= 
    \text{send} : \text{Event}, \text{receive} : \text{Event}, \text{sig} : \text{Operation}
Defining Constraints on Diagrams

- Quantification over sets: classes, C.ops, msgs
- Symbols: →, →, ⊲ →
- Predicates and functions include:
  - subs(C), isAbstract(C)
  - $m < m'$, calls($m, m'$), isNew($o$), returns($m$)
  - fromAct($m$), fromLL($m$), fromClass($m$)

- Inter-diagram constraints include that every message to an activation must be for an operation of a concrete class

$$\forall m \in msgs. \ m.sig \in toClass(m).opers \land \neg isAbstract(toClass(m))$$

- Can’t be done in OCL and would be far more complex anyway
Components

- ObjectStructure, Visitor, Element ∈ classes
- visitops ⊆ Visitor.opers

Static Conditions

- allAbstract(visitops)
- For every kind of element, there’s a unique visit operation for that element and a unique operation defined only for that element subclass.

\[ \forall E \in \text{subs(Element)} . \exists! opv \in \text{Visitors.opers} . \exists! op \in E.\text{opers} . \neg \exists op' \in E.\text{opers} . op = E.\text{op}' \]

- furthermore, denoting the witnesses op and opv by f(E) and g(E), the functions f and g are total bijections
Dynamic Conditions - Antecedent

- For every kind of element, if that element is told to accept a visitor then

\[
\forall E \in \text{subs}(\text{Element}) . \exists ma \in \text{messages} . \\
ma\. \text{sig} = \text{accept} \land \text{toClass}(ma) = E \land \\
\exists l \in \text{lifelines} . \text{hasParam}(ma, l\. \text{name}) \land \\
l\. \text{class} \in \text{subs}(\text{Visitor}) \Rightarrow
\]

Dynamic Conditions - Consequent

- the message came from the object structure and

\[
\text{fromClass}(ma) = \text{ObjectStructure} \land
\]
the message will call the visit operation and

$$\exists mv, mo \in \text{messages} . \quad mv.\text{sig} = g(E) \land mo.\text{sig} = f(E) \land$$

that operation will then call the unique operation for the element

$$\text{toLL}(mv) = l \land \text{calls}(ma, mv) \land \text{calls}(mv, mo) \land \text{toLL}(mo) = \text{fromLL}(mv)$$
Class Diagram for Factory Method Pattern

- Product
- ConcreteProduct
- Creator
  - FactoryMethod()
  - AnOperation()
- ConcreteCreator
  - FactoryMethod()
- ... product = FactoryMethod()
  - ...
- return new ConcreteProduct
Formalisation of Factory Method Pattern

Components
- Creator, Product ∈ classes
- factoryMethod ∈ Creator.opers

Static Conditions
- factoryMethod.isAbstract
- for every creator subclass, there is a product subclass

∀C ∈ subs(Creator) . ∃!P ∈ subs(Product)

- furthermore, denoting witness P by f(C), then f is a total bijection.

Dynamic Conditions
for every creator subclass, the factory method creates a unique product subclass:

\[ \forall C \in \text{subs}(\text{Creator}). \]
\[ \text{isMakerFor}(C..\text{factoryMethod}, f(C)) \]

\[ \text{isMakerFor}(\text{op}, C) \equiv \]
\[ \exists m \in \text{messages} . \ m\.\text{sig} = \text{op} \Rightarrow \]
\[ \exists m' \in \text{messages} \land \text{isNew}(m'.\text{sig}) \land \]
\[ \text{calls}(m, m') \land \text{toClass}(m') = C \land \]
\[ \text{returns}(m) = \text{toLL}(m').\text{name} \]
## Results of Case study

Dr Ian Bayley and Prof Hong Zhu, Oxford Brookes University

### Specifying Behavioural Features of Design Patterns in First Order Logic

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Simple structural properties</th>
<th>Interface properties</th>
<th>Inheritance</th>
<th>Collaboration</th>
<th>Alternatives</th>
<th>Specified adequately</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Adapter</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bridge</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Builder</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chain of Respons.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Command</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Composite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decorator</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Facade</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Factory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flyweight</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Interpreter</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Iterator</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mediator</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Memento</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Observer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Prototype</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Proxy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Singleton</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>State</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Strategy</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Template</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Visitor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Further Work

- Tool support for detection of Design Patterns
  - translate any UML model into logical statements
  - use SPASS theorem prover to prove the predicate true
  - class diagrams are easier than sequence diagrams
- Define a composition operator
- Formalise the intent of Design Patterns