# Specifying Behavioural Features of Design Patterns in First Order Logic

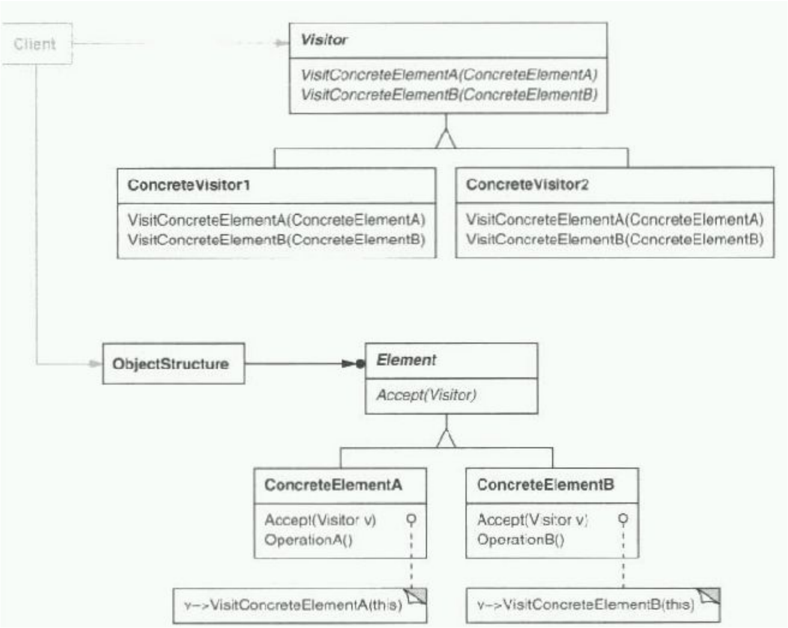
Dr Ian Bayley and Prof Hong Zhu,
Oxford Brookes University

30th July 2008
COMPSAC '08, Turku, Finland

# Introduction to Design Patterns

- 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)



# Formalisation of Class Diagrams I

$$ClassDiagram ::= \\ \quad classes : Class^{+}, \\ \quad assocs : Rel^{*}, inherits : Rel^{*}, CompAg : Rel^{*}$$

$$Rel ::= \\ \quad [name : String], source, end : End$$

$$Class ::= \\ \quad name : String, [attrs : Property^{*}], \\ \quad [opers : Operation^{*}]$$

# Formalisation of Class Diagrams II

$$
\begin{aligned}
&Operation ::= \\
&\quad name : String, [params : Parameter^*], \\
&\quad [isQuery : Boolean], [isLeaf : Boolean], \\
&\quad [isNew : Boolean], [isStatic : Boolean], \\
&\quad [isAbstract : Boolean]
\end{aligned}
$$

# Formalisation of Class Diagrams III

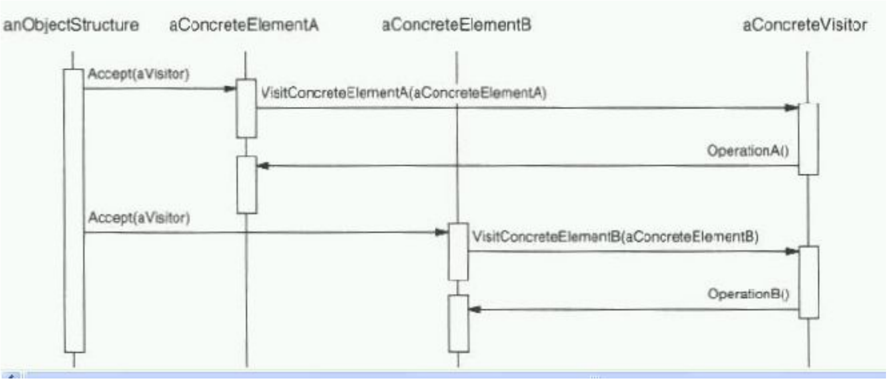
$$
\begin{aligned}
&\mathit{Parameter} ::= \\
&\qquad [\mathit{direction} : \mathit{ParameterDirectionKind}], \\
&\qquad [\mathit{name} : \mathit{String}], [\mathit{type} : \mathit{Type}], \\
&\qquad [\mathit{mult} : \mathit{MultiplicityElement}] \\
&\mathit{ParameterDirectionKind} ::= \\
&\qquad \text{``}\mathit{in}\text{''} \mid \text{``}\mathit{inout}\text{''} \mid \text{``}\mathit{out}\text{''} \mid \text{``}\mathit{return}\text{''} \\
&\mathit{MultiplicityElement} ::= \\
&\qquad [\mathit{upperValue} : \mathit{Natural} \mid \text{``} * \text{''}], \\
&\qquad [\mathit{lowerValue} : \mathit{Natural}]
\end{aligned}
$$

# Formalisation of Class Diagrams IV

$$Property ::=$$
$$name : String, type : Type, [isStatic : Boolean],$$
$$[mult : MultiplicityElement]$$

$$End ::=$$
$$node : \underline{Class}, [name : String], [mult : MultiplicityElement]$$

# Example of a Sequence Diagram (Visitor)



# Formalisation of Sequence Diagrams I

$$SequenceDiagram ::= lifelines : Lifeline^*, messages : Message^*, ordering : (\underline{Message}, \underline{Message})^*$$

$$Lifeline ::= activations : Activation^*, className : String, [objectName : String], isStatic : Boolean$$

# Formalisation of Sequence Diagrams II

$$Activation ::= \\ \quad start : Event, finish : Event, others : Event^*$$

$$Message ::= \\ \quad send : \underline{Event}, receive : \underline{Event}, sig : \underline{Operation}$$

# Defining Constraints on Diagrams

- quantification over sets: *classes*, *C.opers*, *msgs*
- symbols $\longrightarrow\!\!\triangleright$, $\longrightarrow$, $\diamond\!\!\longrightarrow$
- 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

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

- can't be done in OCL and would be far more complex anyway

# Formalisation of Visitor Pattern I

**Components**

- $ObjectStructure, Visitor, Element \in classes$
- $visitops \subseteq Visitor.opers$

**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.

$$\forall E \in subs(Element) \;.\; \exists! opv \in Visitors.opers \;.\; \exists! op \in E.opers \;.\; \neg\exists op' \in Element.opers \;.\; op = E.op'$$

- furthermore, denoting the witnesses *op* and *opv* by $f(E)$ and $g(E)$, the functions $f$ and $g$ are total bijections

# Formalisation of Visitor Pattern II

**Dynamic Conditions - Antecedent**

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

$$\forall E \in subs(Element) \;.\; \exists ma \in messages \;.$$
$$ma.sig = accept \wedge toClass(ma) = E \wedge$$
$$\exists l \in lifelines \;.\; hasParam(ma, l.name) \wedge$$
$$l.class \in subs(Visitor) \Rightarrow$$

**Dynamic Conditions - Consequent**

- the message came from the object structure and

$$fromClass(ma) = ObjectStructure \wedge$$

# Formalisation of Visitor Pattern III

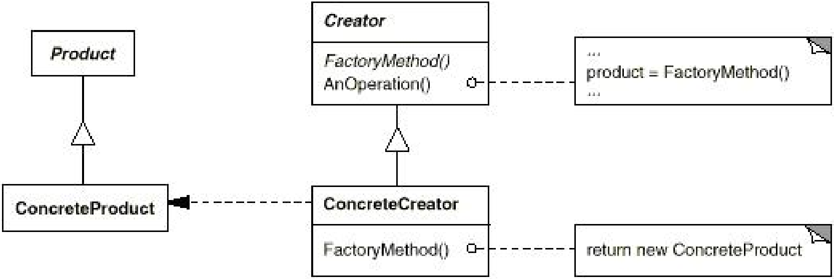
- the message will call the visit operation and

$$\exists mv, mo \in messages \ . \\ mv.sig = g(E) \wedge mo.sig = f(E) \wedge$$

- that operation will then call the unique operation for the element

$$toLL(mv) = l \wedge calls(ma, mv) \\ \wedge calls(mv, mo) \wedge toLL(mo) = fromLL(mv)$$

# Class Diagram for Factory Method Pattern



**Components**

- $Creator, Product \in classes$
- $factoryMethod \in Creator.opers$

**Static Conditions**

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

$$\forall C \in subs(Creator) \;.\; \exists! P \in subs(Product)$$

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

**Dynamic Conditions**

# Formalisation of Factory Method Pattern II

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

$$\forall C \in subs(Creator).\quad isMakerFor(C..factoryMethod, f(C))$$

$$isMakerFor(op, C) \equiv \exists m \in messages\ .\ m.sig = op \Rightarrow \exists m' \in messages \wedge isNew(m'.sig) \wedge calls(m, m') \wedge toClass(m') = C \wedge returns(m) = toLL(m').name$$

# Results of Case study

| Pattern | Simpler structural properties | Improved behavioral properties | Many alternatives | Specified adequately |
|---|---|---|---|---|
| Abstract Factory | ✓ | ✓ | ✓ | ✓ |
| Adaptor | ✓ | ✓ | ✗ | ✓ |
| Bridge | ✓ | ✗ | ✓ | ✗ |
| Builder | ✓ | ✓✓ | ✓ | ✗ |
| Chain of Respons. | ✓ | ✗ | ✓ | ✓ |
| Command | ✓ | ✓✓ | ✓ | ✓ |
| Composite | ✓ | ✓ | ✓ | ✓ |
| Decorator | ✓ | ✓ | ✓ | ✗ |
| Facade | ✗ | ✓ | ✗ | ✓ |
| Factory | ✓ | ✓ | ✓ | ✓ |
| Flyweight | ✗ | ✗ | ✗ | ✗ |
| Interpreter | ✓ | ✓ | ✗ | ✓ |
| Iterator | ✓ | ✓ | ✗ | ✓ |
| Mediator | ✓ | ✓ | ✗ | ✓ |
| Memento | ✗ | ✓✓ | ✗ | ✓ |
| Observer | ✓ | ✓✓ | ✓ | ✗ |
| Prototype | ✓ | ✓ | ✓ | ✓ |
| Proxy | ✓ | ✓ | ✓ | ✗ |
| Singleton | ✗ | ✓✓ | ✗ | ✓ |
| State | ✓ | ✓ | ✗ | ✓ |
| Strategy | ✓ | ✗ | ✗ | ✓ |
| Template | ✓ | ✗ | ✗ | ✓ |
| Visitor | ✓ | ✓✓ | ✗ | ✓ |

# 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