Formalising Design Patterns in Predicate Logic

Software Engineering and Formal Methods '07

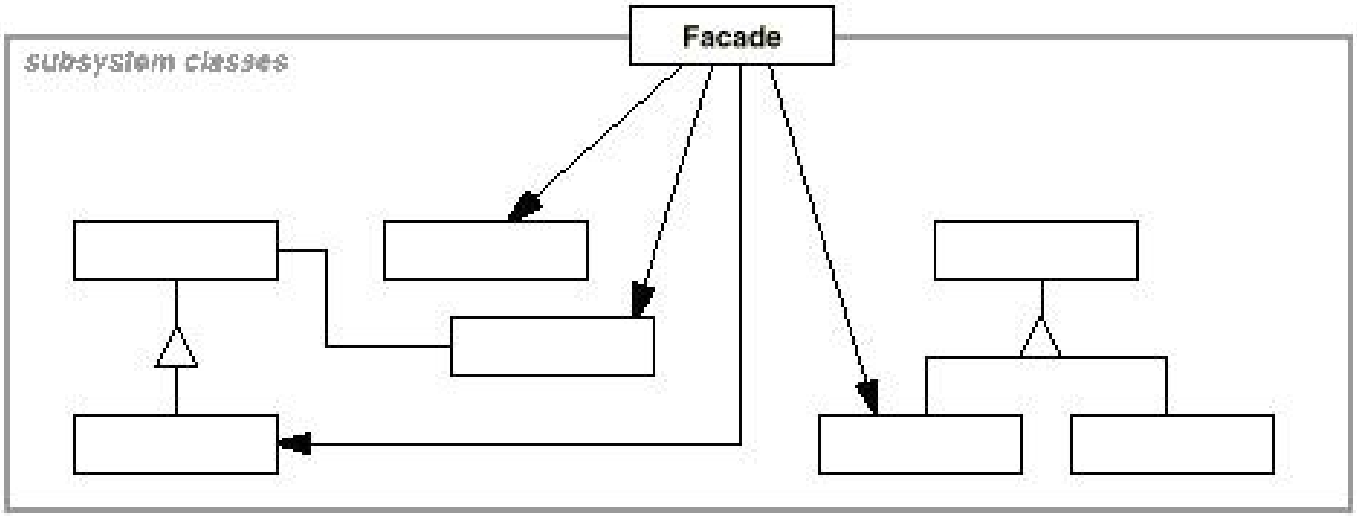
Dr Ian Bayley and Prof Hong Zhu
Oxford Brookes University

12th September 2007

Design Patterns

- What is the purpose of Design Patterns?
  - "to capture design experience in a form that people can use effectively"
  - from G4 book (6th most cited)
- How are Design Patterns specified?
  - Name
  - Intent
  - Motivating example
  - Class Diagram
  - C++ code

Facade Design Pattern



- what do the arrows and boxes actually mean?

Related Work

- Shi & Olson (PINOT) - 2006
- Lano et al (VDM++) - 1996
- Lauder and Kent (three layer approach) - 1998
- Mapelsden et al (DPML) - 2002
- Eden (LePUS) - 2002
- Taibi (pre/post conds and temporal logic) - 2006
- Mikkonen (temporal logic of actions) - 1998
- Le Guennec (extend UML meta-model) - 2000
- Mak et al (action semantics) - 2004

- open problems include expressiveness and support for formal reasoning

Our Approach

- formalise structure of class diagrams
  - using language GEBNF
  - G=Graphical
- specify extraction functions
- pattern is a sentence of predicate logic
- classes ... exist such that ... and ... and ...
- OCL can only be used either to augment class diagrams or at meta-level to define the notion of class diagrams themselves

Graphical EBNF

- EBNF: repetitions are separate entities
- Graphical models have several occurrences of same entity
  - eg nodes and edges (set of pairs of nodes)
  - eg classes and associations/generalisations
- GEBNF is EBNF extended with references

Definition in Graphical Extended BNF

$$
\begin{aligned}
&\mathit{ClassDiagram} = \\
&\quad \mathit{classes} : \mathit{Class}^{+}, \\
&\quad \mathit{inters} : \mathit{Interface}^{*}, \\
&\quad \mathit{assocs} : (\underline{\mathit{Classifier}}, \underline{\mathit{Classifier}})^{*}, \\
&\quad \mathit{geners} : (\underline{\mathit{Classifier}}, \underline{\mathit{Classifier}})^{*}, \\
&\quad \mathit{deps} : (\underline{\mathit{Classifier}}, \underline{\mathit{Classifier}})^{*}, \\
&\quad \mathit{calls} : (\underline{\mathit{Operation}}, \underline{\mathit{Operation}})^{*}
\end{aligned}
$$

First Order Predicate Logic On Diagrams

- domain of quantifiers are variables from graphical model
  - *classes* and *inters* for the nodes
  - *assocs*, *geners*, *deps*, *calls*
- extraction functions
  - eg isAbstract(C) tells whether a class C is abstract
  - defined as part of the GEBNF

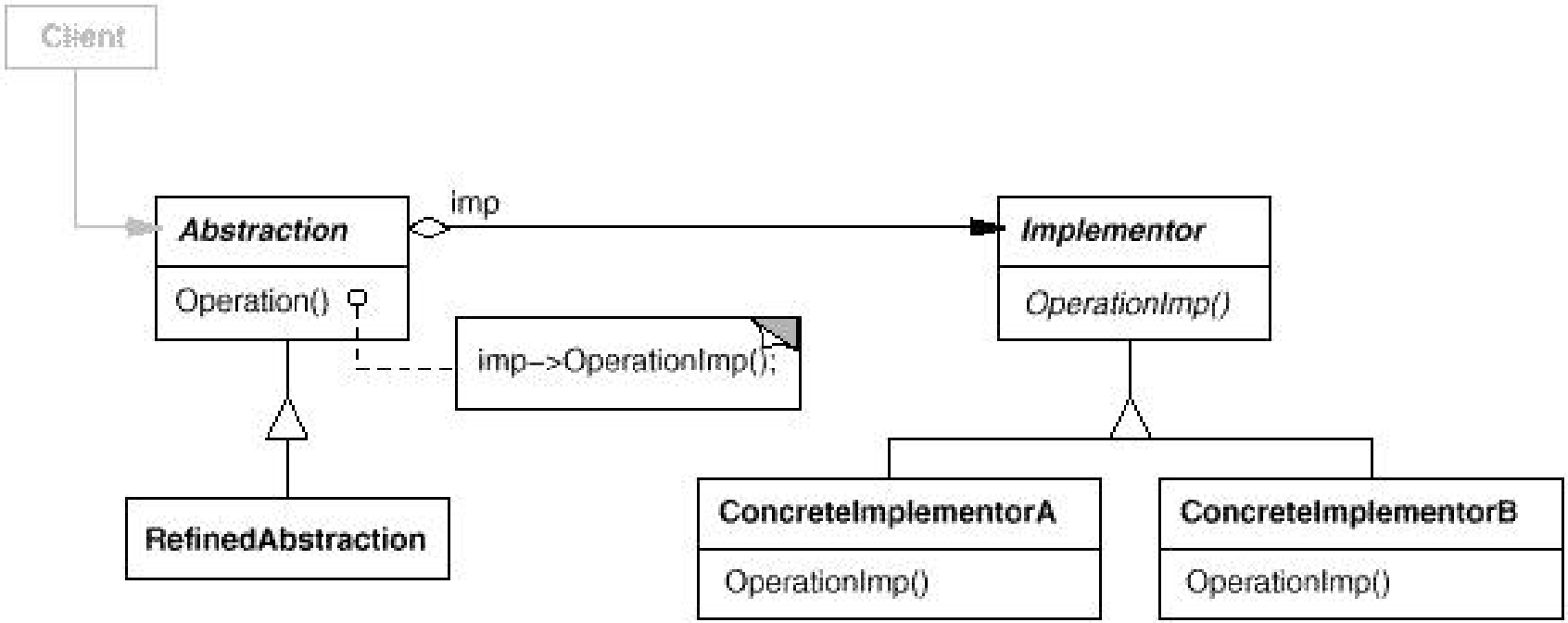
Specification of Facade Design Pattern

- domain of quantifiers are variables from graphical model
  - *classes* and *inters* for the nodes
  - *assocs*, *geners*, *deps*, *calls*
- extraction functions
  - eg isAbstract(C) tells whether a class C is abstract
  - defined as part of the GEBNF
- there's a subset of the classes *ys* such that any dependency arrow to *ys* must either be from *ys* or *Facade*

$$\exists ys \subseteq classes \land \forall C \in ys \cdot \forall C' \in classes \cdot$$

$$(C' \mapsto C) \in deps \Rightarrow C' \in ys \lor C' = Facade$$

Bridge Design Pattern



Specification of Bridge Design Pattern I

Classes: $Abstraction, Implementor \in classes$

Associations: $Abstraction \mapsto Implementor \in assocs$

Conditions:

1. *Implementor* is an interface:
   $Implementor \in inters$
2. client dependencies are on *Abstraction* alone:
   $access(\{Abstraction\}, \{Implementor\} \cup subs(Abstraction) \cup subs(Implementor))$
3. every operation in the subclasses of *Abstraction* call an operation in *Abstraction*:
   $\forall A \in subs(Abstraction) \cdot \forall o \in opers(A) \cdot \exists o' \in opers(Abstraction) \cdot o \mapsto o' \in calls$

Specification of Bridge Design Pattern II

4. every operation in *Abstraction* calls an operation in *Implementor*:
   $\forall o \in opers(Abstraction) \cdot \exists o' \in opers(Implementor) \cdot o \mapsto o' \in calls$

Uses of Specification in Software Engineering

- support software design
  - recognise design patterns at design stage
  - transformation of designs
  - understanding of design patterns
  - relationships between design patterns
    - specialisation
    - compatibility
- deducing properties of design patterns

Formal Recognition of Design Patterns I

Classes: $AbstractFactory \in classes$,
$AbstractProducts \subseteq classes$

Operations: $creators \subseteq opers(AbstractFactory)$

Conditions:

1. *AbstractFactory* is an interface:
   $AbstractFactory \in inters$
2. every factory method is abstract:
   $\forall o \in creators \cdot isAbstract(o)$
3. every class in *AbstractProducts* is abstract:
   $\forall C \in AbstractProducts \cdot isAbstract(C)$

Formal Recognition of Design Patterns II

4. For each abstract product, there is a unique factory method *creator* of *AbstractFactory* that returns the product:

$$\forall AP \in AbstractProducts \cdot$$
$$\exists! creator \in creators \cdot returns(creator, AP)$$

5. The different creation operations and the concrete products are connected by a special one-one correspondence.

$$\{o \in opers(AbstractFactory) \cdot$$
$$\{s \in subs(AbstractFactory) \cdot red(o, s)\}\} \mapsto$$
$$\{p \in AbstractProducts \cdot subs(p)\} \in iso(iso(returns))$$

$$xs \mapsto ys \in iso(R) \equiv$$
$$\forall x \in xs \cdot \exists! y \in ys \cdot x \mapsto y \in R \wedge \forall y \in ys \cdot \exists! x \in xs \cdot x \mapsto y \in R$$

Inferring Properties of Design Patterns I

Classes: $AbstractClass \in classes$

Operations: $templateMethod \in opers(AbstractClass)$

Conditions:

1. *templateMethod* calls an abstract operation of *AbstractClass*.

$$\exists o \in opers(AbstractClass) \cdot (templateMethod \mapsto o) \in calls \land isAbstract(o)$$

- every abstract operation must be redefined in a subclass
- so abstract operations called by *templateMethod* are redefined in concrete subclasses.

Specialisations of Design Patterns

- modulo renaming, Interpreter can be seen to be a specialisation of Composite
- six conditions for both plus the following for Interpreter alone

$$\begin{aligned} &\#interpret.parameters = 1 \wedge \\ &\quad \exists p \in interpret.parameters \cdot \\ &\quad type(p) = Context \end{aligned}$$

Conclusion

- Advantages
  - Easy to understand
  - Helps clarify concepts
  - Can explore alternative definitions
  - Facilitate reasoning about design patterns
- Open problems and future work
  - Behavioural characteristics
  - Tool support