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
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
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
ClassDiagram =

classes : Class^+, 
inters : Interface^*, 
assocs : (Classifier, Classifier)^*, 
geners : (Classifier, Classifier)^*, 
deps : (Classifier, Classifier)^*, 
calls : (Operation, Operation)^*
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
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

\[ \exists ys \subseteq \text{classes} \land \forall C \in ys \cdot \forall C' \in \text{classes} \cdot (C' \mapsto C) \in \text{deps} \Rightarrow C' \in ys \lor C' = \text{Facade} \]
Bridge Design Pattern

Formalising Design Patterns in Predicate Logic

Client

Abstraction
- Operation()

Implementor
- OperationImp()

RefinedAbstraction

ConcretImplementorA
- OperationImp()

ConcretImplementorB
- OperationImp()
Classes: Abstraction, Implementor ∈ classes

Associations: Abstraction ↦ Implementor ∈ assocs

Conditions:

1. Implementor is an interface:
   Implementor ∈ inters

2. Client dependencies are on Abstraction alone:
   access({Abstraction}, {Implementor} ∪ subs(Abstraction)
   ∪ subs(Implementor))

3. Every operation in the subclasses of Abstraction call an operation in Abstraction:
   ∀A ∈ subs(Abstraction) · ∀o ∈ opers(A) · ∃o′ ∈ opers(Abstraction) · o ↦ o′ ∈ calls
every operation in Abstraction calls an operation in Implementor:
\[ \forall o \in \text{opers}(\text{Abstraction}) \cdot \exists o' \in \text{opers}(\text{Implementor}) \cdot o \mapsto o' \in \text{calls} \]
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
Classes:  AbstractFactory ∈ classes,  
          AbstractProducts ⊆ classes  
Operations: creators ⊆ oper(AbstractFactory)  
Conditions:

1. AbstractFactory is an interface:  
   AbstractFactory ∈ inters  
2. every factory method is abstract:  
   ∀o ∈ creators · isAbstract(o)  
3. every class in AbstractProducts is abstract:  
   ∀C ∈ AbstractProducts · isAbstract(C)
For each abstract product, there is a unique factory method \textit{creator} of \textit{AbstractFactory} that returns the product:

$$\forall AP \in \text{AbstractProducts}. \exists! creator \in \text{creators} \cdot \text{returns}(creator, AP)$$

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

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

$$xs \mapsto ys \in \text{iso}(R) \equiv \forall x \in xs \cdot \exists! y \in ys \cdot x \mapsto y \in R \land \forall y \in ys \cdot \exists! x \in xs \cdot x \mapsto y \in R$$
Classes:  \( \text{AbstractClass} \in \text{classes} \)

Operations:  \( \text{templateMethod} \in \text{opers(AbstractClass)} \)

Conditions:

1. \( \text{templateMethod} \) calls an abstract operation of \( \text{AbstractClass} \).

\[
\exists o \in \text{opers(AbstractClass)} \cdot \\
(\text{templateMethod} \mapsto o) \in \text{calls} \land \\
\text{isAbstract}(o)
\]

- every abstract operation must be redefined in a subclass
- so abstract operations called by \( \text{templateMethod} \) 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

\[
\#\text{interpret.parameters} = 1 \land \\
\exists p \in \text{interpret.parameters} \cdot \\
\text{type}(p) = \text{Context}
\]
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