Design Patterns

Patterns are general, reusable solutions to commonly occurring problems

• Concept originated with Christopher Alexander’s work on architecture

• Popularised in software engineering by the “gang of four”

• Subject of study by the knowledge engineering community
Design Patterns for the Semantic Web

N-ary relations
- How can we say more about a relation instance?
- How do we represent an ordered sequence of relations?

Value partitions and value sets
- How do we represent a fixed list of values?

Part-whole hierarchies
- How do we represent hierarchies other than the subclass hierarchy?
N-ary Relations
Binary Relations

In RDF and OWL, binary relations link two individuals, or an individual and a value.

The properties `birthYear` and `fatherOf` are binary relations.
Relations with Additional Information

In some cases, we need to associate additional info with a binary relation
• e.g. certainty, strength, dates

For example, Holbein the Elder’s date of birth is unconfirmed
• He was born in either 1460 or 1465
• How can we represent this uncertainty?
N-ary Relations

N-ary relations link an individual to more than a one value

Possible use cases:
1. A relation needs additional info
   e.g. a relation with a rating value
2. Two binary relations are related to each other
   e.g. body_temp (high, normal, low), and trend (rising, falling)
3. A relation between several individuals
   e.g. someone buys a book from a bookstore
4. Linking from, or to, an ordered list of individuals
   e.g. an airline flight visiting a sequence of airports
Pattern 1: Reified Relation

To represent additional information about a relation:

• Create a new class to represent the relation
• Individuals of this class are instances of the relation
• Relation class can have additional properties to describe more information about the relation
Use case 1: additional information

Jack has given the film ‘I Am Legend’ a four-star rating
• We need to represent a quantitative value to describe the rating relation
Use case 1: additional information

Person $\subseteq \forall \text{issuedRating}. \text{RatingRelation}$

$\text{RatingRelation} \subseteq \exists \text{ratedObject}. \text{Film} \sqsubseteq 1 \text{ratedObject}$

$\text{RatingRelation} \subseteq \forall \text{ratingValue}. \text{Rating} \sqsubseteq 1 \text{ratingValue}$
Use case 2: different aspects of a relation

Steve has a temperature which is high, but falling
- We need to represent different aspects of the temperature that Steve has
Use case 3: no distinguished participant

John buys a “Lenny the Lion” book from orinoco.com for $15 as a birthday gift

- No distinguished subject for the relation
- i.e. no primary relation to convert into a Relation Class as in cases 1 and 2

Source: W3C
Use case 3: no distinguished participant

Purchase ⊑ ∃ hasBuyer. Person ∩ = 1 hasBuyer
Purchase ⊑ ∃ hasSeller. Company ∩ = 1 hasSeller
Purchase ⊑ ∃ hasObject. Object
Purchase ⊑ ∀ hasAmount. Quantity ∩ = 1 hasAmount
Purchase ⊑ ∀ hasPurpose. Purpose
Pattern 2: Sequence of arguments

United Airlines, flight 1377 visits the following airports: LAX, DFW, and JFK
- For such an example, we need to represent a sequence of arguments
Pattern 2: Sequence of arguments

\[ \top \equiv \forall \text{flightSequence}^- \cdot \text{Flight} \]  
\[ \top \equiv \forall \text{flightSequence} \cdot \text{FlightSegment} \]  
\[ \top \equiv \leq 1 \text{flightSequence} \]  
\[ \top \equiv \forall \text{nextSegment}^- \cdot \text{FlightSegment} \]  
\[ \top \equiv \forall \text{nextSegment} \cdot \text{FlightSegment} \]  
\[ \top \equiv \leq 1 \text{nextSegment} \]  
\[ \top \equiv \forall \text{destination}^- \cdot \text{FlightSegment} \]  
\[ \top \equiv \forall \text{destination} \cdot \text{Airport} \]  

\[ \text{flightSequence} \text{rdfs:domain Flight} \]  
\[ \text{flightSequence} \text{rdfs:range FlightSegment} \]  
\[ \text{flightSequence is functional} \]  
\[ \text{nextSegment} \text{rdfs:domain FlightSegment} \]  
\[ \text{nextSegment} \text{rdfs:range FlightSegment} \]  
\[ \text{nextSegment is functional} \]  
\[ \text{destination} \text{rdfs:domain FlightSegment} \]  
\[ \text{destination} \text{rdfs:range Airport} \]  

FlightSegment \( \equiv 1 \text{destination} \cap \leq 1 \text{nextSegment} \)
FinalFlightSegment \( \equiv \text{FlightSegment} \cap = 0 \text{nextSegment} \)
Value Partitions and Value Sets
Descriptive Features

Descriptive features are quite common in ontologies:

- **Size** = {small, medium, large}
- **Risk** = {dangerous, risky, safe}
- **Health status** = {good health, medium health, poor health}

Also called “qualities”, “modifiers” and “attributes”

- A property can have only one value for each feature to ensure consistency

Three main approaches:

- Enumerated individuals (a value set)
- Disjoint classes (a value partition)
- Datatype values (not considered in this lecture)
Value Sets

Values of descriptive feature are individuals
Value Sets

A health value can be either poor, medium or good:

\[ \text{HealthValue} \equiv \{ \text{poorHealth, mediumHealth, goodHealth} \} \]

Poor, medium and good are all different from each other:

\[ \text{poorHealth} \neq \text{mediumHealth} \]
\[ \text{poorHealth} \neq \text{goodHealth} \]
\[ \text{mediumHealth} \neq \text{goodHealth} \]

A healthy person is a person who has some health status which is the value good:

\[ \text{HealthyPerson} \equiv \text{Person} \cap \exists \text{hasHealthStatus}.\{ \text{goodHealth} \} \]
Notes on Value Sets

Need axioms to set the three health values to be different from each other
  • This way, a person cannot have more than one health value at a time

Values cannot be further partitioned
  • e.g. cannot have fairly_good_health as a subtype of good_health

Only one set of values is allowed for a feature
  • The class HealthValue cannot be equivalent to more than one set of distinct values
  • Doing so will cause inconsistencies
Value Partitions

Values of descriptive features are disjoint subclasses:
Value Partitions

Poor, medium and good are types of health value:

\[
\begin{align*}
\text{PoorHealth} & \sqsubseteq \text{HealthValue} \\
\text{MediumHealth} & \sqsubseteq \text{HealthValue} \\
\text{GoodHealth} & \sqsubseteq \text{HealthValue}
\end{align*}
\]

Covering axiom (the only types of health value are poor, medium and good):

\[
\text{HealthValue} \equiv \text{PoorHealth} \sqcup \text{MediumHealth} \sqcup \text{GoodHealth}
\]

Poor, medium and good are pairwise disjoint:

\[
\begin{align*}
\text{PoorHealth} \sqcap \text{MediumHealth} & \equiv \bot \\
\text{PoorHealth} \sqcap \text{GoodHealth} & \equiv \bot \\
\text{MediumHealth} \sqcap \text{GoodHealth} & \equiv \bot
\end{align*}
\]

A healthy person is a person who has some health status which is an instance of good

\[
\text{HealthyPerson} \equiv \text{Person} \sqcap \exists \text{hasHealthStatus}. \text{GoodHealth}
\]
Value Partitions

The instance JohnsHealth can be made anonymous
Notes on Value Partitions

Values can be further partitioned
  - Simply add subclasses to the value classes
Can have alternative partitions of the same feature

OWL 2 contains specific support for defining disjoint unions

\[ C \equiv C_1 \cup C_2 \cup \cdots \cup C_n \]
\[ C_1 \cap C_2 \equiv \bot \]
\[ C_1 \cap C_3 \equiv \bot \]
\[ \cdots \]
\[ C_{n-1} \cap C_n \equiv \bot \]
Part-Whole Hierarchies
Meronymies (part-whole relations)

Taxonomies are not the only hierarchical relation that we wish to model

• A spark plug isn’t a kind of engine (class-instance)
• A spark plug is a part of an engine
Simple Part-Whole Representation

We need two properties:

- partOf (a transitive property)
- directPartOf (a subproperty of partOf)

\[
\text{part of } \circ \text{ part of } \subseteq \text{ part of } \\
\text{directPartOf } \subseteq \text{ part of }
\]
Part-Whole Hierarchies

Represent part-whole relationships between classes using existential restrictions:

Every spark plug is a direct part of some engine:  \[ \text{SparkPlug} \sqsubseteq \exists \text{directPartOf}. \text{Engine} \]

Every engine is a direct part of some car:  \[ \text{Engine} \sqsubseteq \exists \text{directPartOf}. \text{Car} \]

Every wheel is a direct part of some car:  \[ \text{Wheel} \sqsubseteq \exists \text{directPartOf}. \text{Car} \]
Defining Classes of Parts

Extend the ontology with classes of parts for each level, so that the reasoner can automatically derive a class hierarchy:

A car part is a part of some car: \( \text{CarPart} \equiv \exists \text{partOf}. \text{Car} \)

A direct car part is a direct part of some car: \( \text{DirectCarPart} \equiv \exists \text{directPartOf}. \text{Car} \)

An engine part is a part of some engine: \( \text{EnginePart} \equiv \exists \text{partOf}. \text{Engine} \)

A reasoner will infer that \( \text{EnginePart} \sqsubseteq \text{CarPart} \) (but not \( \text{EnginePart} \sqsubseteq \text{DirectCarPart} \))
Fault Location

Once we have a meronymy, we can use it to inherit features within that hierarchy

For example, a reasoner could infer that a fault in a part is a fault in a whole

- Need a new property for the location of a fault: hasLocus
- Need a new class for faults: Fault

We can then define general types of located faults:

\[
\text{FaultInCar} \equiv \text{Fault} \sqcap \exists \text{hasLocus. CarPart} \\
\text{FaultInEngine} \equiv \text{Fault} \sqcap \exists \text{hasLocus. EnginePart}
\]
Fault Location

Now we can define specific types of located fault:

\[
\text{DirtySparkPlug} \sqsubseteq \text{Fault} \sqcap \exists\text{hasLocus}.\text{SparkPlug} \\
\text{FlatTyre} \sqsubseteq \text{Fault} \sqcap \exists\text{hasLocus}.\text{Wheel}
\]

The definition of the hierarchy allows a reasoner to infer that:

\[
\text{DirtySparkPlug} \sqsubseteq \text{FaultInCar} \\
\text{DirtySparkPlug} \sqsubseteq \text{FaultInEngine} \\
\text{FlatTyre} \sqsubseteq \text{FaultInCar}
\]

But not:

\[
\text{FlatTyre} \not\sqsubseteq \text{FaultInEngine}
\]
Further Reading
SWBP Notes

Defining N-ary Relations on the Semantic Web
http://www.w3.org/TR/swbp-n-aryRelations

Representing Specified Values in OWL
http://www.w3.org/TR/swbp-specified-values

Simple part-whole relations in OWL Ontologies
http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/