4. Design Extraction

- **Why Extract Design? Why UML?**
- Interpreting UML
- Tracks For Extraction
- Extraction of Intention
- Extraction For The Reusers

**Design Extraction**

*Design is not code with boxes and arrows*

- Design extraction is not trivial
  - If you are serious about it, not a low level task!
- Design extraction should scale up
- Design extraction can be supported by computers but not fully automated
- A critical view on hype: “we read your code and generate design documents”
- Fertilize you with some basic techniques that may help you
- Show that UML is not that simple and clear but still useful

**UML (Unified Modelling Language)**

- Successor of OOAD&D methods of late 80 & early 90
- Unifies Booch, Rumbaugh (OMT) and Jacobson [Booc98a] [Rumb99a]. Currently standardized by OMG.
- UML is a modelling language and not a methodology (no process)
- UML defines
  - a notation (the syntax of the modelling language)
  - a meta-model (eMof in UML 2.0) — a model that defines the “semantics” of a model
  - what is well-formed, defined in itself but weakly!

**Why is Design Extraction Needed?**

- Documentation inexistent, obsolete, or too verbose
- Abstraction needed to understand applications
- Original programmers left
- Only the code available
- Why UML?
  - **Standard**
  - Communication based on a common language
  - Can support documentation if we are precise about its interpretation
  - **Extensible**

**Roadmap**

- Why Extract Design? Why UML?
- Interpreting UML
- Tracks For Extraction
- Extraction of Intention
- Extraction For The Reusers

**Levels of Interpretations: Perspectives**

M. Fowler proposed 3 levels of interpretations called **perspectives** [Fow97a]:

- **Conceptual**: we draw a diagram that represents the concepts that are somehow related to the classes but there is often no direct mapping.
- **Specification**: we are looking at interfaces of object not implementation, types rather than classes. Types represent interfaces that may have many implementations
- **Implementation**: implementation classes
Attributes in Perspectives

- Syntax:
  - visibility attributeName: attributeType = defaultValue
  - E.g.: name: String
- Conceptual:
  - Customer name \mapsto Customer has a name
- Specification:
  - Customer class should provide a way to set and query the name
- Implementation:
  - Customer has an attribute that represents its name
- Possible Refinements: Attribute Qualification
  - Immutable: Value never change
  - Read-only: Client cannot change it

Operations in Perspectives

- Syntax:
  - visibility name (parameter-list) return-type
  - E.g.: public, if protected, \rightarrow private
- Conceptual:
  - principal functionality of the object. It is often described as a sentence
- Specification:
  - public methods on a type
- Implementation: methods
  - Operations approximate methods but are more like abstract methods
- Possible Refinements: Method qualification:
  - Query (does not change the state of an object)
  - Cache (does cache the result of a computation). Derived Value (depends on the value of other values). Getter, Setter

Associations: Specification Perspective

- Associations represent responsibilities
- Implications:
  - One or more methods of Customer should tell what Orders a given Customer has made.
  - Methods within Order will let me know which Customer placed a given Order and what Line items compose an Order.
- Associations also imply responsibilities for updating the relationship, such as:
  - specifying the Customer in the constructor for the Order
  - add/removeOrder methods associated with Customer

Arrows: Navigability

- No arrow \equiv navigability in both directions or unknown

Private you said!! Which one?

Is it [class-based (C++) or instance-based (Smalltalk)]?
- In C++:
  - any public member is visible anywhere in the program
  - a private member may be used only by the class that defines it
  - a protected member may be used by the class that defines it or its subclasses
  - Class-based private
- In Smalltalk:
  - instance variables C++ protected, methods are public
- In Java:
  - a protected member may be accessed by subclasses but also by any other classes in the same package as the owning class
  - protected is more public than package

Need for a Clear Mapping

- UML
  - language independent even if influenced by C++
  - fuzzy (navigability, package...)
    - We should define how we interpret it
      - Define some conventions
- Some C++ examples:

<table>
<thead>
<tr>
<th>Board game()</th>
<th>Board game operator (const Board game other)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piece() myMap;</td>
<td>myMap: Piece</td>
</tr>
<tr>
<td>class Booklet: public Board game</td>
<td>public inherits</td>
</tr>
<tr>
<td>static</td>
<td>int width();</td>
</tr>
</tbody>
</table>

Generalization

UML semantics only supports generalization and not inheritance.

- Conceptual:
  - What is true for an instance of a superclass is true for a subclass (associations, attributes, operations).
- Corporate Customer is a Customer
- Specifications:
  - Interface of a subtype must include all elements from the interface of a superclass.
- Implementation:
  - Generalization semantics is not inheritance. But we should interpret it this way for representing extracted code.
Stereotypes: To Represent Conventions!

- Mechanism to specialize the semantics of the UML elements
- New properties are added to an element
- When a concept is missing or does not fit your needs select a close element and extend it
- 40 predefined stereotypes (c = class, r = relation, o = operation, a = attribute, d = dependency, g = generalization): metaclass (c), instance (r), implementation class (c) constructor (o), destructor (o), friend (d), inherits (g), interface (c), private (g), query (o), subclass (g), subtype (g).
- Do not push stereotypes to the limit or you will lose standards

Roadmap

- Why Extract Design? Why UML?
- Interpreting UML
- Tracks For Extraction
- Extraction of Intention
- Extraction For The Reusers

Design is not code with boxes

Association Extractions (i)

**Goal: Explicit references to domain classes**

- **Domain Objects**
  - Quality as attributes only implementation attributes that are not related to domain objects.
  - Value objects + attributes and not associations.
  - Object by references + associations
  
  E.g.: Street name + an attribute
  
  Order order + an association
  
  Piece myPiece (in C++) + composition
  
- Define your own conventions
  
  E.g.: Integer x integer + point attribute
  
- Two classes possessing attributes on each other
  
  + an association with navigability at both ends

Convention Based Association Extraction

- Filtering based coding conventions or visibility
- In Java, C++ filter out private attributes
- In Smalltalk depending on coding practices you may filter out attributes
- that have accessors and are not accessed into subclasses
- with name: "Cache.
- attributes that are only used by private methods.
- If there are some coding conventions
  
  class Order {  
  
  public (Customer customer);  
  
  // single value  
  
  public (Customer orderLine);  
  
  // multi-value  
  
  }

Operation Extraction

- You may not extract
  
  + accessors
  
  + operators, non-public methods,
  
  + simple instance creation methods (new in Smalltalk, constructor with no parameters in Java)
  
  + methods already defined in superclass,
  
  + methods already defined in superclass that are not abstract
  
  + methods that are responsible for the initialization, printing of the objects

- Use company conventions to filter
  
  + Access to database, Calls for the UI, Naming patterns

Operation Extraction (ii)

**If there are several methods with more or less the same intent**

- If you want to know that the functionality exists not all the details
  
  select the method with the smallest prefix

**If you want to know all the possibilities but not all the ways you can invoke them**

- select the method with the most parameters

**If you want to focus on important methods**

- categorize methods according to the number of times they are referenced by clients
  
  + a hook method is not often called but is still important

- What is important to show: the creation interface
  
  Smalltalk class methods in ‘instance creation’ category,
  
  Non default constructors in Java or C++

Roadmap

- Why Extract Design? Why UML?
- Interpreting UML
- Tracks For Extraction
- Extraction of Intention
- Extraction For The Reusers

Operation Extraction

Design Patterns reveal the intent so they are definitely appealing for supporting documentation [John92a] [Oden97a]

But

- **Difficult** to identify design patterns from the code [Brow96, Wuyt98a, Pec98a]
- What is the difference between a State and a Strategy from the code point of view?
- Need somebody who knows
- Read the Code in one Hour
- Lack of support for code annotation so difficult to keep the use of patterns and the code evolution [Hof97a]
### DPs are NOT about Structure

- **Adapter Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.
- This code structure IS NOT an Adapter; it may if the relationship between B and C is about protocol adaptation!

### DPs are about Intent and Pros/Cons

- DPs are not carved in stone
- They are vocabulary and intention
- They are tradeoffs
- Read the class names
- Read the comments
- Watch out for “DPs Magic Extracting tools”

### Evolution Impact Analysis: Reuse Contract

- How to identify the impact of changes?
- How to document for reusers/extenders?
- How to document framework?

### Example

- **Domain** (e.g., ordered collection)
  - `add(element)`
  - `addAll(elements)`

- **Application** (e.g., increment)
  - `increment(element)`

- **Change Propagation**
  - New version
  - Order of collection

### Reuse Contracts: General Idea

- **Reuse Contracts** [Stey96a] propose a methodology to:
  - Specify and qualify extensions
  - Specify evolution
  - Detect conflicts

### Lessons Learned

- **You should be clear about:**
  - Your goal (detailed or architectural design)
  - Conventions, like navigability,
  - Language mapping based on stereotypes
  - Level of interpretations
- **For Future Development**
  - Emphasize literate programming approach
  - XSLT-like approaches
  - Extract design to keep it synchronized
- **UML as Support for Design Extraction**
  - Often fuzzy
  - Do not support well dynamic/reflective languages
  - But UML is extensible, so define your own stereotype!