Object-oriented Design

Software Engineering

2004-2005

Marco Scotto (Marco.Scotto@unibz.it)
Content

Introduction

- OO system architecture design
- From OOA to OOD object model
- Design patterns
- Principles of OOD
- Design of dynamic model
- Physical design
Introduction

➢ Before OOD, there is OOA…
   • OO model of key classes developed with CRC method
   • May be documented with UML diagrams
   • Only deals with the problem domain

➢ OOD deals with the specific implementation details, such as …
   • User interface
   • Database

➢ Class: unit of modular decomposition in OOD
Content

- Introduction
- **OO system architecture design**
  - From OOA to OOD object model
  - Design patterns
  - Principles of OOD
  - Design of dynamic model
  - Physical design
OO design models (1/2)

- Used to devise a detailed picture of all classes and objects → drive and ease coding afterward
- UML diagrams are used
  - Same concepts and notation behind both OOA & OOD
- 3 models: static, dynamic, functional

1. **Functional model**
   - Describing data value transformations within system
   - Essentially contains data flow diagrams
   - Least important of the three models
2. **Static** (object model)
   - Static structure of objects and their relationships
   - Info not affected by time (no temporal info included)
   - **UML diagrams**
     - Static structural diagrams
       - Class diagram
       - Object diagram
     - Implementation diagrams
       - Component diagram
       - Deployment diagram

3. **Dynamic model**
   - Aspects of system that change over time
   - Working scenarios, message passing, state changes, and activity flows
   - **UML diagrams**
     - Use case diagrams
     - Interaction diagrams
       - Sequential diagram
       - Collaboration diagram
     - Activity diagrams
     - State diagrams
OO design phases (1/2)

- High level design and low level design

1. High level design
   - Overall hard/soft-ware architecture devised
   - OO subsystems identified
     - Object model, data management component, user interface, system interaction component
   - Basic elements and relationships identified
     - Classes, data model, interface specification of external parts
2. **Low level design**

- Every class defined; attributes & methods specified
- Realizing possible reuse
- Physical components built
  - Files or entities containing code
- Interdependencies among components identified
- Detailed design of hard/soft-ware components
  - For system documentation and future maintenance
OOD main subsystems (1/2)

- **Object model**
  - Kernel of system: location of key processing
  - Derived from OOA model directly

- **Human interaction component (HIC)**
  - Objects of user interface
  - e.g. Windows and widgets of the GUI; applications to take user inputs, to show results, etc.

- **Data management component (DMC)**
  - Objects for permanently storing and retrieving data

- **System interaction component (SIC)**
  - Interface with external devices, network, & internet
OOD main subsystems (2/2)

GUI

HIC

GUI Applications

Model (from OOA)

DMC

Database

SIC

Internet

LAN

Software Engineering

Devices
Content

- Introduction
- OO system architecture design
- From OOA to OOD object model
- Design patterns
- Principles of OOD
- Design of dynamic model
- Physical design
From OOA to OOD object model

- Logic design: **how** to implement the system
  - Definition of data types and new classes not in OOA
  - Detailed definition of data structures of all classes
  - Assoc., aggreg., compos. relationships implemented
  - Detailed definition of interfaces
  - Use of design patterns to address OO issues
  - Creation of algorithms of relevant operations

- Two options
  - Original OOA model changed to OOD model
  - Different OOD model generated with OOA model maintained separately
    - Manual updating for both models required
**UML object diagrams**

- Snapshot of the detailed state of a class at a point in time
- Often used within collaboration diagrams
- One ideal full class diagram ➔ many different object diagrams
- Associations among object are shown as links
  - Association name and role name may be shown

```
object name

intercity32:Train

object class

instance variables

code = 32
maxSpeed = 160
length = 240
velocity = 124.0

Note: no operation compartment
```
Implementation of relationships

- Task of OOD static model
- Associations with cardinality 0..1 and/or 1..1 at both ends
  - Implemented using variable or pointer in data structure
- Associations with cardinality 1..* or 0..*
  - Modify OOD model (may include new method) with a 1..1 or 0..1 association as replacement
  - Use container to hold many instances of one of more classes
  - Examples on the following slides
Example a)

Class diagram for OOA model

Class diagram for modified OOD model

Cardinality changed

Named roles added

New method added

Software Engineering
Example b)

Class diagram for OOA model

A container class in library; Attributes and methods not shown (built-in)

Class diagram for OOD model

New variable of type LinkedList; Added to data structure of Track
## Main kinds of containers

<table>
<thead>
<tr>
<th>Name</th>
<th>Indexing</th>
<th>Java class(es)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector/Array</td>
<td>Integer index</td>
<td>Array</td>
<td>Fixed length container.</td>
</tr>
<tr>
<td>List/Ordered-Collection</td>
<td>Integer index</td>
<td>List, ArrayList, LinkedList</td>
<td>Variable length container. Objects can be inserted and deleted.</td>
</tr>
<tr>
<td>Set</td>
<td>No index</td>
<td>Set, HashSet</td>
<td>Variable length container with no index. Objects must be unique (no duplication).</td>
</tr>
<tr>
<td>SortedCollection/SortedSet</td>
<td>Internal order</td>
<td>TreeSet</td>
<td>Objects are maintained in a pre-defined order. E.g.) a list of strings in alphabetic order.</td>
</tr>
<tr>
<td>Map/Dictionary</td>
<td>Hash table</td>
<td>Map, HashMap</td>
<td>Variable length container where objects are inserted and accessed using a unique key.</td>
</tr>
</tbody>
</table>
Methods/operations (1/3)

- Its specification describes system behaviour
- Executed on specific instances (i.e. objects)
- **Class/static method**
  - Executed on a class, not on its instance
- **Abstract (virtual) method**
  - Shown in italics in UML class diagram
  - Abstract class: has at least one abstract method
    - No instances of the class can be created
Methods/operations (2/3)

- UML notation

Methods are specified by signature:
- Name
- Name and types of arguments
- Return type

Signature of a method
Methods/operations (3/3)

- Operations categories:
  - **Algorithmically simple**
    - All operations every object must have
    - e.g. constructors, destructors, getters, setters
  - **Algorithmically complex**
    - Directly related to class responsibilities
    - Already identified during analysis phase
Interface (1/2)

- In general, a set of public methods
- In Java, a set of method definitions (signature) without method bodies
  - “Pure” abstract class
- Does not have attributes usually
- A class may implement one or more interfaces
  - i.e. to provide implementation for every method of the interface
  - Allows classes of different hierarchies with different parents and behaviour to all be treated in same way
- Interfaces can inherit from other interfaces
UML notation

Interface (2/2)

String

isEqual(String):Boolean
isGreater(String):Boolean
hash():int

Comparable

isEqual(Comparable):Boolean
isGreater(Comparable):Boolean
hash():int

String

isEqual(String):Boolean
isGreater(String):Boolean
hash():int

StringHashTable

insert(String)
delete(String)
includes (String):Boolean

Comparable

No attributes

Software Engineering

“Implements” relationship

Shorthand notation for interface

Dependency arrow: a class using the methods of an interface
Content

- Introduction
- *OO system architecture design*
- *From OOA to OOD object model*

**Design patterns**

- Principles of OOD
- Design of dynamic model
- Physical design
Design pattern

- Well engineered and proven solution to common, recurring design problem
- Has the form of a set of interacting classes
- Helps developers understand object orientation
- Facilitates communication among developers
- Each is characterized by:
  - Name, solved problem, context and forces, goal, solution, usage (examples), and consequences
**Singleton**

- **Description:** How do I have a class, so that only one object of the class can be alive
- **Solution:** Preventing the creation of objects, still letting one object alive
- **Sample:**

```java
class Singleton {
    private Singleton() { }

    Singleton onlyObject = new Singleton();

    static Singleton get Singleton() {
        return onlyObject;
    }  
}
```

Software Engineering
Classification of patterns (1/2)

- **Behavioural patterns**
  - Describing complex control flow and communication
  - To maximize balance of responsibilities and interface uniformity
  - To decouple cooperating objects; easing reuse and software changes

- **Creational patterns**
  - Creating and initializing of objects using high-level interface
  - Implementation details hidden
  - To decouple object creation from class specification
  - To increase software modularity
Classification of patterns (2/2)

- **Structural patterns**
  - Creating and assembling many objects
  - To decouple access to objects from specific implementation
  - To be able to incrementally add new features
  - To change complex object composition at runtime
  - To increase software modularity
Content

- Introduction
- **OO system architecture design**
- **From OOA to OOD object model**
- **Design patterns**

**Principles of OOD**

- Design of dynamic model
- Physical design
Info hiding, low coupling, cohesion (1/3)

- **Principle 1**: *access instance variables of objects only through accessors*
- Instance variables should be private
- Getters – for security
  - Read instance variables; copies of data returned
  - e.g. `int getSize()`
- Setters – for data consistency
  - Write instance variables; only way of modifications
  - e.g. `void setSize()`
- Data access can be controlled and logged
Info hiding, low coupling, cohesion (2/3)

- Principle 2: minimize the number of public and protected methods
- Looser coupling among classes
- Protecting subclasses from changes in their superclasses
- Interfaces of public and protected methods of a class should never change (or be minimized)
Info hiding, low coupling, cohesion (3/3)

- **Principle 3: Law of Demeter**

  - An object $O$ in response to a message $M$ should only send messages to the following objects:
    - $O$ itself
    - Objects sent as arguments of message $M$
    - New objects created while executing method $M$
    - Directly accessible instance variables (objects) of $O$
    - Objects providing global services to $O$

- Methods should only operate on objects directly available

- “Kill the middle-man” principle
Style of reuse (1/3)

- **Design reuse through**
  - Patterns, schemata, composition, and inheritance

- **Composition**
  - Using existing objects as parts of new object
  - “has-a” relationship
  - Additive: new object is sum of its parts
  - Projective: new object wrapping an existing object
    - Hiding some behaviour and exposing others

- **Schemata**
  - Template: parameterized generation of new elements
  - Interface: many different implementations of the same abstraction
Design by composition

- **Wireless Microphone**
  - Aggregation of
    - Regular Microphone
    - Radio Transmitter

- **Ball pen**
  - Aggregation of
    - Ink
    - Stilo
    - Top
Design by schemata

- Stack of <Generic Type>
  - I can instantiate it as:
    - Stack of int
    - Stack of double
    - ...

- Glass of <Generic Beverage>
  - I could instantiate it as:
    - Glass of beer
    - Glass of coke
    - Glass of wine
  - But I could also have something a class Glass with an attribute Content, which could be any beverage
Style of reuse (2/3)

Inheritance
- Subclass reuses and extends features of superclass
- Inheritance by exception: subclass limiting features of superclass
  - Cannot be used when performing polymorphism

Principle 4: Liskov’s Principle of Substitution
- If class B is “just like” class A but with extensions…
- Possible to use a B object for an A object anywhere

Principle 5: use inheritance only to reuse and extend functionalities, also only if it reflects a real-world relationship
Style of reuse (3/3)

- Inheritance not for reusing implementation
  - Use composition if it is the case

- Roles: an exception to Principle 5
  - e.g. human roles; a person can play different roles
    - Class Person composed of a list of roles instead of being inherited from subclasses with high possibility of overlaps

- Principle 6: roles are acquired via attributes, not by subclassing

- Principle 7: use projective composition in place of inheritance by exception
Design by contract

- Provides a specification of each non-trivial method of the class and the class’ legal state

- Preconditions and postconditions
  - Assumptions about inputs and outputs of method, & object states just before and just after execution
  - Callers must guarantee preconditions are satisfied
  - Same for developer and postconditions

- Assertions and invariants
  - Specification of legal states and constraints on data values of classes
  - Usually boolean predicates; verification is embedded in code
Preconditions and postconditions at work

PRECONDITION: X > 0
POSTCONDITION: RETURN > X

Function Fact(int X)
BEGIN
    RETURN X*(Fact(X-1))
END
Content

- Introduction
- OO system architecture design
- From OOA to OOD object model
- Design patterns
- Principles of OOD
- Design of dynamic model
- Physical design
Design of dynamic model

Dynamic behaviour in UML
- Interactions between objects at run-time
  - Sequence diagrams and collaboration diagrams
- State changes of objects over time
  - State diagrams
- State changes and control flow inside methods
  - Activity diagrams

Steps
- Identify use cases; define scenarios of interactions
- Identify objects, events and exchange of messages
- Develop interaction diagrams & state diagrams
- Check for consistency and completeness

Remember: Diagrams are not a value by themselves
Be stingy!
UML interaction diagrams

- Snapshot of the system in a given state

- Ingredients
  - Interacting objects
  - Links between objects
  - Messages exchanged
  - Flow of time
  - Possible events from outside triggering the scenario
  - Data and info flow carried and returned by messages

- Sequence diagrams & collaboration diagrams
UML sequence diagram

- Highlights sequence of messages through time
- Does not show how objects are linked

Object line thickened during method execution

Software Engineering
UML collaboration diagram

- Highlights links between objects

1. restart()
2. getSignal()
3. ^false : isRed()
4. new RestartEvent()
5. insertEvent(event)

Link & stereotype

1234:Train

TK2:Track-Section

S3:Signal

:Simulator

RestartEvent

Message sending

Created during execution

New

Local event

Local signal

Parameter

Global

Associated
## Links in collaboration diagram

<table>
<thead>
<tr>
<th>Name</th>
<th>Stereotype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association</td>
<td>«association»</td>
<td>The receiver is accessible since it is contained in the data structure of the sender. It is the default case; stereotype can be omitted.</td>
</tr>
<tr>
<td>Global object</td>
<td>«global»</td>
<td>The receiver is globally accessible in the system or a sub-system. A global variable.</td>
</tr>
<tr>
<td>Parameter</td>
<td>«parameter»</td>
<td>The receiver is a parameter of the method executed by the sender.</td>
</tr>
<tr>
<td>Local object</td>
<td>«local»</td>
<td>The receiver is contained in a local variable of the method executed by the sender, or is an intermediate result of an expression. Two sub-cases:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- the receiver is obtained as the result of a message sent to another object;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- the receiver is created <em>ex-novo</em> during the execution of the method. The stereotype «new» may be shown in the rectangle representing the object created.</td>
</tr>
<tr>
<td>Self</td>
<td>«self»</td>
<td>The receiver and the sender are the same object. The link is a line connecting the object with itself.</td>
</tr>
</tbody>
</table>
Activity diagram

- Graphically documents complex operations of the system
- Expresses a decision with different possible transitions (outcomes)
  - Depending on Boolean conditions
  - Each possible outcome labelled by a distinctive guard condition
- Its usage should be limited
Content

- Introduction
- OO system architecture design
- From OOA to OOD object model
- Design patterns
- Principles of OOD
- Design of dynamic model
- Physical design
Component diagram

- Shows relationship between software components
  - Source code, binary code, executable, etc.
  - Some components exist at compile time, link time, and/or run time

Composition: Shown by physical containment or the usual symbol of...
Deployment diagram

- Shows configuration of run-time entities
- Depicts Nodes and their relationships
  - Nodes are connected by communication associations

Diagram:
- Node: WK2: Workstation
- Node: LP1: Printer
- Node: AP1: Internet Server
- Node: DBX: Database Server
- Communication link
- Component contained in Node: DBMS, VSAM, FileServices