Class Diagrams

Class diagrams and models can be looked at from three different perspectives:

• conceptual – represent the concepts in the domain without regard to the software, related to the classes that implement them but there is often no direct mapping

• specification – look at the software but concentrate on the interfaces, not the implementation – the “Type”; o-o languages usually combine interface and implementation (Modula 2: DEFINITION and IMPLEMENTATION MODULE)

• implementation – this is the actual implementation level frequently very close to the programming language used
Associations

 Associations represent relationships between between instances of classes

 - conceptual – relationships between classes
   - each association has 2 ends – each can be labeled with a role name
   - an association end has a multiplicity indicating how many instances may participate in this relationship
 - specification – relationships represent responsibilities
   - realized through one or more methods (get, update, . . .)
   - can not infer structure from the interface description
 - implementation – here the details become visible
   - e.g. a doubly-linked pointer structure
   - a “ref” implementation (Java)

Vocabulary

 Classes are the most important building blocks

 A class represents a set of objects sharing the same properties (structure, behaviour)

 A class implements one or more interfaces

 Classes may represent:

 - Software things
 - Hardware things
 - Things that are purely conceptual

 Modeling involves identifying the things important to the particular view

 These things form the vocabulary of the system
Simple and Path Names

- Temperature Sensor
- BusinessRules::FraudAgent
- Customer
- Car

Things have names - identity

Simple and Path Names

- BusinessRules::FraudAgent
- java::awt::Rectangle

Things have names - identity

Attributes - 1

<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>address</td>
</tr>
<tr>
<td>phone</td>
</tr>
<tr>
<td>birthDate</td>
</tr>
<tr>
<td>. . .</td>
</tr>
</tbody>
</table>

An attribute is a named property of a class that describes a range of values that instances may hold. Usually a short noun or noun phrase that represents some property.

Things have state - attributes
Attributes - 2

<table>
<thead>
<tr>
<th>Wall</th>
</tr>
</thead>
<tbody>
<tr>
<td>height: Float</td>
</tr>
<tr>
<td>width: Float</td>
</tr>
<tr>
<td>thickness: Float = 10.0</td>
</tr>
<tr>
<td>isLoadBearing: Boolean</td>
</tr>
</tbody>
</table>

An attribute can be further specified by giving its class (type) and possibly an initial value.

Operations - 1

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>add()</td>
</tr>
<tr>
<td>grow()</td>
</tr>
<tr>
<td>move()</td>
</tr>
<tr>
<td>isEmpty()</td>
</tr>
<tr>
<td>. . .</td>
</tr>
</tbody>
</table>

An operation is the realization of a service that can be requested from any instance of the class; it is something one can do to an object. Usually a short verb or a verb phrase.

Things have behavior - operations
An operation can be further specified by stating its signature.

TemperatureSensor

reset()
setAlarm(t: Temperature)
value() : Temperature
...

Attributes and Operations

* When drawing a class it is not necessary to show every attribute and every operation.
* In most cases there are too many and only the ones relevant to a specific view are included.
* For these reasons it is common to elide a class and show only some or none of the attributes and operations.
* An empty compartment does not mean that there are no attributes or operations.
* To better organize long lists of attributes and operations one can prefix each group with a descriptive category by using “stereotypes”.
Organizing Attributes and Operations

FraudAgent

<<constructor>>
new()
new(p: Policy)
<<process>>
process(o: Order)
...
<<query>>
isSuspect(o: Order)
isFraudulent(o: Order)
<<helper>>
validateOrder(o: Order)

Responsibilities

FraudAgent

Responsibilities
-- determine the risk of a customer order
-- handle customer-specific criteria for fraud

A responsibility is a contract or an obligation of a class; responsibilities are derived from the UseCases. Usually given in free-form text, written as a phrase, a sentence or a short paragraph.

Things have responsibilities - services
Classes rarely stand alone; when building models one usually focuses on groups of classes that interact. These societies of classes form collaborations and are usually visualized in class diagrams.

Modeling the Vocabulary of a System

1. Identify those things that users or implementers use to describe the problem or solutions. UseCase-based analysis helps finding these abstractions.
2. For each abstraction identify the responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all classes.
3. Provide the attributes and operations that are needed to carry out the responsibilities for each class.

Modeling the Vocabulary

- **Customer**
  - name
  - address
  - phone
  - birthDate

- **Warehouse**
  - id
  - name
  - price
  - location

- **Order**
  - item
  - quantity

- **Invoice**

- **Transaction**
  - actions
  - commit()
  - rollBack()
  - wasSuccessful()

- **Shipment**
  - Responsibilities
    - maintain the information regarding products shipped against an order
    - track the status and location of the shipped products
Modeling the Vocabulary of a System

- As the models get larger, classes tend to cluster in groups that are conceptually and semantically related.
- In the UML packages can be used to model these clusters of classes.
- Most of the models will rarely be completely static but include interactions between the vocabulary.
- The UML provides a number of ways to model the dynamic behaviour.

Modeling the Distribution of Responsibilities in a System

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of the classes.
- Split classes that have too many responsibilities into smaller abstractions.
- Collapse tiny classes that have trivial responsibilities into larger ones.
- Reallocate responsibilities so that each abstraction reasonably stands on its own.
- Consider the ways in which those classes collaborate with one another and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.
Modeling the Vocabulary

<table>
<thead>
<tr>
<th>Model</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
<td>Responsibilities</td>
</tr>
<tr>
<td>-- manage the state of</td>
<td>-- render the model</td>
</tr>
<tr>
<td>the model</td>
<td>on the screen</td>
</tr>
<tr>
<td></td>
<td>-- manage movement</td>
</tr>
<tr>
<td></td>
<td>and resizing of the</td>
</tr>
<tr>
<td></td>
<td>view</td>
</tr>
<tr>
<td></td>
<td>-- intercept user</td>
</tr>
<tr>
<td></td>
<td>events</td>
</tr>
</tbody>
</table>

Controller

Responsibilities

-- synchronize changes in the model and its views

Modeling the distribution of responsibilities in a system

Modeling Primitive Types

Primitives data types such as integers, characters, floating point numbers, strings and enumerations available in most programming languages can be modeled in UML

To model primitive types

- model the thing to be abstracted as a type or an enumeration which is rendered using class notation with the appropriate stereotype
- If the range of values associated with this type needs to be specified, make use of constraints.

<table>
<thead>
<tr>
<th>&lt;&lt;datatype&gt;&gt;</th>
<th>&lt;&lt;enumeration&gt;&gt;</th>
<th>&lt;&lt;datatype&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Status</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>idle</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>working</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td></td>
</tr>
</tbody>
</table>

(values range from $-2^{31}$ to $+2^{31}-1$)
Hints and Tips

Every class should map to some tangible or conceptual abstraction in the user or implementer domain. A well-structured class:

- provides a crisp abstraction of something drawn from the vocabulary of the problem domain or the solution domain
- embodies a small well defined set of responsibilities and carries them out very well
- provides a clear separation of the abstractions specification and its implementation
- is understandable and simple yet extensible and adaptable.

Hints and Tips - cont

When drawing a class in UML:

- show only those properties of the class that are important to understand the abstraction in its context;

- organize long lists of attributes and operations by grouping them according to their category;

- show related classes in the same class diagrams.
Classes and Relationships

Very few classes stand alone
Model how these classes stand in relation to each other
In object-oriented modeling three types of relationships are especially important:
  • dependencies – using relationships, e.g. object1 (surgeon) depends on object2 (nurse)
  • generalizations – the class/subclass relationship (generalization/specialization) e.g. class1 (vehicle) generalizes car-objects
  • associations - structural relationships among instances, aggregation and composition e.g. a car consists of engine, body, chassis, . . .
Relationship Notation

The UML provides a graphical notation for each of these kinds of relationships:

- Window
  - open( )
  - close( )
  - move( )
  - display( )
  - handleEvent( )

- ConsoleWindow

- DialogBox

- Control

Event

dependency

generalization

association

Dependency

A dependency is a using relationship

"uses-a"

FilmClip

name

start( )
stop( )
pause( )
playOn( c: Channel)

Can have a name (rarely used)

Channel

most often one class uses another as an argument in the signature of an operation.
Generalization / Specialization

Graphical notation:

- **Shape**
  - origin
  - move()
  - display()
  - resize()

- **Square**
- **Rectangle**
  - corner: Point

- **Circle**
  - radius: Float
  - points: List

Superclass

Subclass

Leaf class

Association

A structural relationship

Graphical notation:

Basic form

- **Person**
- **Company**

Association

Four “adornments” can be used for additional specification of the association:
- name
- role
- multiplicity
- aggregation
 Associations

Associations represent relationships between between instances of classes

- conceptual – relationships between classes
  - each association has 2 ends – each can be labeled with a role name
  - an association end has a multiplicity indicating how many instances may participate in this relationship
- specification – relationships represent responsibilities
  - realized through one or more methods (get, update, . . .)
  - can not infer structure from the interface description
- implementation – here the details become visible
  - e.g. a doubly-linked pointer structure
  - a “ref” implementation (Java)

Association – Name / Role

An association can have a name to describe the nature of the relationship

- works_for
- employee
- employer
- role
- name
- name direction
State the multiplicity at one end of an association; this means that for each object of the class at the opposite end, there must be that many objects at the near end!

Association – Multiplicity / Aggregation

Multiplicity - how many instances may participate

Person

employee

1..*

Company

employer

multiplicity

Association

Company

1

Department

whole

part

multiplicity

association

aggregation

1

Company

*
## Composition and Aggregation

**Aggregation** – "has a" relation
- more loosely coupled components
- car: body, chassis, engine, transmission, . . .
- plane: cell, wings, engines, hydraulic_system, . . .

**Composition** – "consists of" relation
- partitions, components
- more tightly coupled components (engine – piston)
Example 4-1/ Fowler p.50

Relationships

- "is-a" - inheritance
- "has-a" - composition (aggregation)
- "is-aware-of" ("uses-a") - calls
Associations - Responsibilities

_static relationships are represented by associations_, e.g.

- pilot flies a plane
- customer rents a car
- customer buys n books
- buffer holds n packets (\([0 .. 127]\) or \([*]\) ?)

_subtypes - inheritance_, e.g.

- a nurse is a person
- a car is a vehicle
- a truck is a vehicle
- a circle is a GraphSymbol
- a triangle is a GraphSymbol

Associations – conceptual level

<table>
<thead>
<tr>
<th>Class A</th>
<th>1 association</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>role A</td>
<td>role B</td>
</tr>
<tr>
<td>person</td>
<td>car</td>
<td></td>
</tr>
<tr>
<td>customer</td>
<td>rental</td>
<td></td>
</tr>
</tbody>
</table>

An association represents a conceptual relationship between classes. Each association has 2 ends labeled with a “role” name.

Each association has a “multiplicity” – an indication of how many objects may participate in a given relationship.

\([\text{lowerbound} .. \text{upperbound}]\)  \([m .. n], [1 .. 4], [0 .. 1]\)

- 1 one instance only - \([1 .. 1]\)
- “many” – unspecified – \([0 .. \text{Inf}]\)
- \([0 .. 1]\) - optional
Example 4-2/ Fowler p.56

Associations – specification level

**Class A**

<table>
<thead>
<tr>
<th>Method_a()</th>
<th>Method_b()</th>
</tr>
</thead>
</table>

Methods are responsible for performing certain actions.

**Class B**

<table>
<thead>
<tr>
<th>Method_1()</th>
<th>Method_2()</th>
</tr>
</thead>
</table>

Example: class A – PriorityQueue, class B – Entry, the associations is named "contains"; clearly procedures for inserting and deleting an entry are needed.

Question: whose responsibility is this – where are the methods to be implemented? (member functions in C++)

Since this is the specification level, only the interface is specified, nothing is said about the actual implementation!
Associations – implementation level

- On the implementation level associations are represented by specific constructs in the respective programming language, e.g.
  - pointers, references, procedure calls

- This should only be done after the specifications have been frozen

- A suitable programming language permitting separate specification (interface) and implementation is desirable (Ada package)

Navigability

- Indicate which class is responsible
- Use directed associations – pointers
  - unidirectional – one side only ---->
  - bidirectional – both sides or undecided --------

- Naming associations
  - verbs for the associations
  - nouns for the roles
Examples

Possible implementations in a Java-like language:

```java
class Order {
    public Customer getCustomer( ); interface
    public Set getOrderLines( );
    . . .
}

class Order {
    private Customer _customer;
    private Set _orderLines;
}

class Customer {
    private Set _orders;
}
```

Attributes

- Attributes are the state variables i.e. their values determine the state of the object
  - on the conceptual level, - “names”
  - on the specification level – set/get operations
  - on the implementation level – variables, fields, structs

- Notation: visibility name: type = defaultValue;
  - visibility: + public
    # protected
    - private

- Visibility as a concept is different from the visibility and scoping rules available in programming languages; be aware of the differences between Smalltalk, C++, Java, C# etc.
Operations

- Operations specify a service and are implemented by methods (or member functions in C++)
- Operations are defined in the interface and may be implemented by many methods
- Syntax:
  - visibility name(parameter_list): return_type_expression
    {property_string}
  - parameters are described by:
    {direction} name: type {default_value}
    - direction: in / out / inout
  - properties are defined as:
    isQuery / sequential / guarded / concurrent
  - queries / modifiers – getting- / setting-methods

Constraint Rules

- “Much of what you are doing in drawing a class diagram is indicating constraints”
  e.g. the number of instances, the order, etc.
- Enclosed within braces “{“ and “}” or expressed in Object Constraint Language
- Constraint rules could be implemented as assertions (pre- and postconditions) in the programming language

Read the “Design by Contract” section p.62 .. 65
When to use Class Diagrams

Simple answer: always
They represent "the backbone of all o-o methods"
But they are so rich, they can be overwhelming

Caveats - KISS principle:
- during elaboration and analysis draw conceptual models
- start with the simple stuff, leave out the details
  - classes, associations, attributes
  - generalizations, constraints
- during design concentrate on specification models
- implementation models only when a particular implementation technique is used (e.g. circular buffer)
- don’t draw models for everything!

Modeling Simple Dependencies

The most common form of dependency is the connection between a class the uses another class as a parameter to an operation

CourseSchedule

```
add(c: Course)
remove(c: Course)
```

Course

iterator

Dependency points “from” the class with the operation “to” the class used as a parameter in the operation
Final Remark

"On the conceptual level there is really no significant difference between attribute and association".

On the specification and implementation levels:
• attributes have value rather than reference semantics
• associations typically have reference semantics

Attributes are usually single-valued
• reference = identity (pointer)
• value = content