Design and UML Class Diagrams

Suggested reading:
Practical UML: A hands on introduction for developers
http://dn.codegear.com/article/31863

UML Distilled Ch. 3, by M. Fowler
How do people draw / write down software architectures?
Example architectures

- person
  - UW student
    - CSE 403 student
  - sea agent
    - lake agent
      - amphibious agent
  - VerizonWireless
    - GPS satellite
      - Cell phone
Big questions

• What is UML?
  – Why should I bother? Do people really use UML?

• What is a UML class diagram?
  – What kind of information goes into it?
  – How do I create it?
  – When should I create it?
Design phase

- **design**: specifying the structure of how a software system will be written and function, without actually writing the complete implementation

- a transition from "what" the system must do, to "how" the system will do it
  - What classes will we need to implement a system that meets our requirements?
  - What fields and methods will each class have?
  - How will the classes interact with each other?
How do we design classes?

- **class identification from project spec / requirements**
  - nouns are potential classes, objects, fields
  - verbs are potential methods or responsibilities of a class

- **CRC card exercises**
  - write down classes' names on index cards
  - next to each class, list the following:
    - **responsibilities**: problems to be solved; short verb phrases
    - **collaborators**: other classes that are sent messages by this class (asymmetric)

- **UML diagrams**
  - class diagrams (today)
  - sequence diagrams
  - ...

![UML Diagram Example](image)
What is UML?

• UML: pictures of an OO system
  – programming languages are not abstract enough for OO design
  – UML is an open standard; lots of companies use it

• What is legal UML?
  – a descriptive language: rigid formal syntax (like programming)
  – a prescriptive language: shaped by usage and convention
  – it's okay to omit things from UML diagrams if they aren't needed by team/supervisor/instructor
Uses for UML

• as a sketch: to communicate aspects of system
  – forward design: doing UML before coding
  – backward design: doing UML after coding as documentation
  – often done on whiteboard or paper
  – used to get rough selective ideas

• as a blueprint: a complete design to be implemented
  – sometimes done with CASE (Computer-Aided Software Engineering) tools

• as a programming language: with the right tools, code can be auto-generated and executed from UML
  – only good if this is faster than coding in a "real" language
In an effort to promote Object Oriented designs, three leading object oriented programming researchers joined ranks to combine their languages:

- Grady Booch (BOOCH)
- Jim Rumbaugh (OML: object modeling technique)
- Ivar Jacobsen (OOSE: object oriented software eng)

and come up with an industry standard [mid 1990’s].
UML – Unified Modeling Language

• Union of all Modeling Languages
  – Use case diagrams
  – Class diagrams
  – Object diagrams
  – Sequence diagrams
  – Collaboration diagrams
  – Statechart diagrams
  – Activity diagrams
  – Component diagrams
  – Deployment diagrams
  – ....

• Very big, but a nice standard that has been embraced by the industry.
Object diagram (≠ class diagram)

• individual objects (heap layout)
  – objectName : type
  – attribute = value

• lines show field references

• Class diagram:
  – summary of all possible object diagrams
Object diagram example

**Object Diagram**

**Instructor**
- [DB]
  - homeCampus = Oxford
  - name = Vlad VARNICA

**[Class]**
- code = UML2Topgun
- semester = winter

**Course**
- [uml2]
  - credit = 10
  - title = DynamicMetamodel

**Instructor**
- [DS]
  - homeCampus = Tunis, Tunisia
  - name = Noura Kossentini

**Class**
- code = CS221
- semester = autumn

**Course**
- [reg]
  - credit = 10
  - title = EMF best practises
UML class diagrams

• **UML class diagram**: a picture of
  – the classes in an OO system
  – their fields and methods
  – connections between the classes
    • that interact or inherit from each other

• **Not** represented in a UML class diagram:
  – details of how the classes interact with each other
  – algorithmic details; how a particular behavior is implemented
Diagram of one class

- class name in top of box
  - write <<interface>> on top of interfaces' names
  - use *italics* for an *abstract class* name

- attributes (optional)
  - includes fields of the object

- operations / methods (optional)
  - may omit trivial (get/set) methods
    - but don't omit any methods from an interface!
  - should not include inherited methods
Class attributes (= fields)

- attributes (fields, instance variables)
  - visibility name : type [count] = default_value
  - visibility:
    - public
    - protected
    - private
    - package (default)
    - derived
  - underline static attributes
  - derived attribute: not stored, but can be computed from other attribute values
    - “specification fields “ from CSE 331

- attribute example:
  - balance : double = 0.00
Class operations / methods

- operations / methods
  - visibility name (parameters) : return_type

- visibility:
  - + public
  - # protected
  - - private
  - ~ package (default)

- underline static methods
- parameter types listed as (name: type)
- omit return_type on constructors and when return type is void

- method example:
  + distance(p1: Point, p2: Point): double
Comments

- represented as a folded note, attached to the appropriate class/method/etc by a dashed line
Relationships between classes

• **generalization**: an inheritance relationship
  – inheritance between classes
  – interface implementation

• **association**: a usage relationship
  – dependency
  – aggregation
  – composition
Generalization (inheritance) relationships

- hierarchies drawn top-down
- arrows point upward to parent
- line/arrow styles indicate whether parent is a(n):
  - class: solid line, black arrow
  - abstract class: solid line, white arrow
  - interface: dashed line, white arrow
- often omit trivial / obvious generalization relationships, such as drawing the Object class as a parent
Associational relationships

- associational (usage) relationships

1. multiplicity  (how many are used)
   - *  ⇒ 0, 1, or more
   - 1  ⇒ 1 exactly
   - 2..4  ⇒ between 2 and 4, inclusive
   - 3..*  ⇒ 3 or more (also written as “3..”)

2. name  (what relationship the objects have)

3. navigability  (direction)
Multiplicity of associations

- **one-to-one**
  - each student must carry exactly one ID card

- **one-to-many**
  - one rectangle list can contain many rectangles
Association types

- **aggregation**: “is part of”
  - symbolized by a clear white diamond

- **composition**: “is entirely made of”
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond

- **dependency**: “uses temporarily”
  - symbolized by dotted line
  - often is an implementation detail, not an intrinsic part of that object's state
Composition/aggregation example

If the movie theater goes away
so does the box office => composition
but movies may still exist => aggregation
Class diagram example

No arrows; info flows in both directions; each knows about the other

Aggregation – Order class contains OrderDetail classes. Could be composition?
UML example: people

Let’s add the visibility attributes
Class diagram: voters

- **VoterAuthentication**
  - voterPersonalInfo: VoterPersonallInformation
  - voterID: String
  - voterPassword: securePW

- **VoterPersonalIdentification**
  - voterLastName: String
  - voterFirstName: String
  - voterMiddleName: String
  - voterSSN: String
  - voterAddress1: String
  - voterAddress2: String
  - voterCity: String
  - voterState: String
  - voterZIP: String
  + validateZipCode(voterZIP: String): String
  + validateState(parameter0: VoterState: String): String

- **BallotCreation**
  - ballotName: String
  - candidates: String[]
  - displayBallot(): void
  - createBallot(): void

- **securePW**
  - PWEntered: JPasswordField
  - securePW(PW: securePW): securePW

This is only a small subset of the actual package...
Class diagram example: video store

- **Class**: Customer
- **Abstract Class**: Rental Item
- **Generalization**: DVD Movie, VHS Movie, Video Game
- **Composition**: Rental Invoice
- **Simple Association**: Checkout Screen
- **Multiplicity**: 1..* (DVD Movie, VHS Movie, Video Game), 1 (Rental Invoice), 0..1 (Checkout Screen)
Class diagram example: student

- **StudentBody**
  - + main (args : String[])

- **Student**
  - firstName : String
  - lastName : String
  - homeAddress : Address
  - schoolAddress : Address
  - + toString() : String

- **Address**
  - streetAddress : String
  - city : String
  - state : String
  - zipCode : long
  - + toString() : String
Tools for creating UML diagrams

• Violet (free)
  – http://horstmann.com/violet/

• Rational Rose
  – http://www.rational.com/

• Visual Paradigm UML Suite (trial)
  – http://www.visual-paradigm.com/
  – (nearly) direct download link:

(there are many others, but most are commercial)
Design exercise: Texas Hold ‘em poker game

- 2 to 8 human or computer players
- Each player has a name and stack of chips
- Computer players have a difficulty setting: easy, medium, hard
- Summary of each hand:
  - Dealer collects ante from appropriate players, shuffles the deck, and deals each player a hand of 2 cards from the deck.
  - A betting round occurs, followed by dealing 3 shared cards from the deck.
  - As shared cards are dealt, more betting rounds occur, where each player can fold, check, or raise.
  - At the end of a round, if more than one player is remaining, players' hands are compared, and the best hand wins the pot of all chips bet so far.

- What classes are in this system? What are their responsibilities? Which classes collaborate?
- Draw a class diagram for this system. Include relationships between classes (generalization and associational).
Class diagram pros/cons

• Class diagrams are great for:
  – discovering related data and attributes
  – getting a quick picture of the important entities in a system
  – seeing whether you have too few/many classes
  – seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
  – spotting dependencies between one class/object and another

• Not so great for:
  – discovering algorithmic (not data-driven) behavior
  – finding the flow of steps for objects to solve a given problem
  – understanding the app's overall control flow (event-driven? web-based? sequential? etc.)