CS 619 Introduction to OO Design and Development

Fall 2014

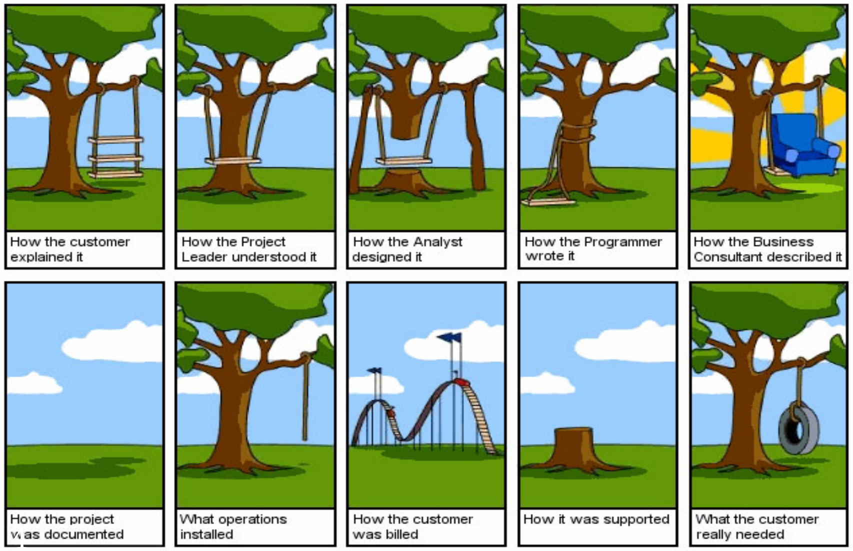
Course Information

- Instructor:
  - Karen (Hong) JIN
  - jin at cs.unh.edu
- TA:
  - Simon Ogbamichael
  - smt725@wildcats.unh.edu
- Course Piazza Site:
  - http://www.cs.unh.edu/~cs619

Topics:

- Software engineering / Iterative software development process
- OOA/D
- Design Patterns
- Android programming

Software Engineering - Motivation


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

The Task of Software Engineers?

Software engineers should

- adopt a **systematic** and **organised** approach to their work
- use appropriate tools and techniques depending on
  - the problem to be solved,
  - the development constraints and the resources available

Why Software Engineering is Needed?

- **Software development is hard**! How to build high-quality software systems?
- **Important to distinguish**

  "easy" systems (*one developer, one user, experimental use only*) **from "hard" systems** (*multiple developers, multiple users, products*).

- **One person techniques do not scale up.**

Why Software Engineering Is Hard?

- The main problem is ***complexity***

  larger software projects (greater than 25, 000 SLOC)

- Many difficulties sources, but ***size*** is key:
  - UNIX contains 4 million lines of code
  - Windows 2000 contains 108 lines of code

- Also ***changeability*** **Change is constant!**

What Will (not) Make a Big Difference:

Minor difference:

- Ada or Java or C# or Python or ...
- Object-Oriented Programming
- Automatic Programming
- Graphical programming
- Environments and tools

Major difference:

- Buy v.s. build your own
- **Requirements refinement and prototyping**
- **Incremental (iterative) development**
- **Great designers** (Unix, Pascal,Smalltalk vs Cobol,PL/I.Ada,MS-DOS)

Software Engineering Activities

- **Problem statement**
  - needs analysis
  - requirements specification: functional, non-functional
- **Design**
  - architectural
  - detailed
  - (communication, database)
- **Implementation**
  - coding
  - testing: modular and integration
  - documentation
- **Maintenance**
  - corrective
  - adaptive
  - enhancement

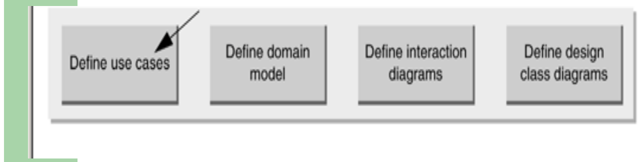
A short Example of OOAD

- Let’s put them into the perspective of OOAD using this small example.

A Dice Game:

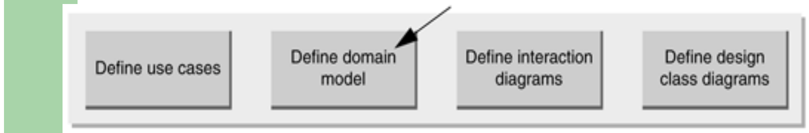
software simulates a player rolling two dice. If the total is seven, they win; otherwise, they lose.

OOAD



- **Requirements analysis** may include stories or scenarios of how people use the application; these can be written as **use cases**.
- Use cases are not an object-oriented artifact, they are simply written stories. However, they are a popular tool in requirements analysis

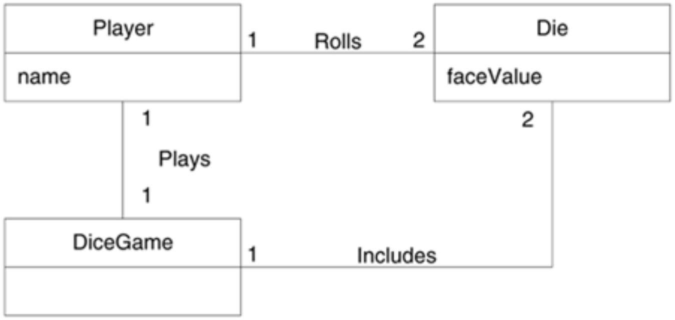
OOAD



- **Object-oriented analysis** is concerned with creating a description of the domain from the perspective of objects.
  - Identification of the concepts, attributes, and associations that are considered noteworthy.
- The result can be expressed in a **domain model** that shows the noteworthy domain concepts or objects

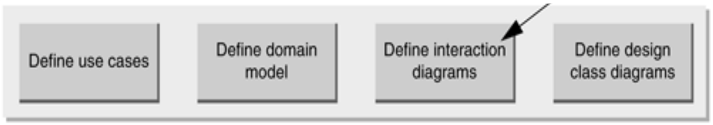
OOAD

Figure 1.3. Partial domain model of the dice game.



Note that a domain model is NOT a description of software objects

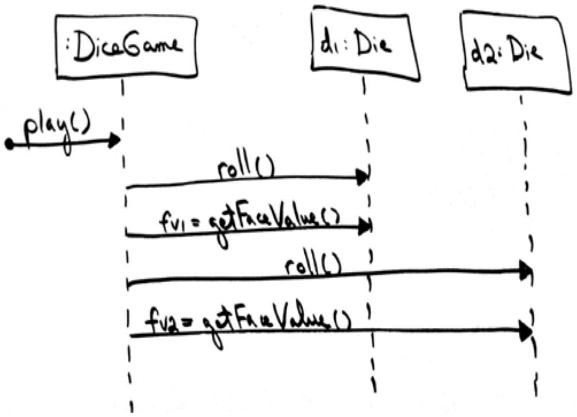
OOAD



- **Object-oriented design** is concerned with defining software objects - their responsibilities and collaborations.
- Common notations to illustrate these collaborations are
  - UML **sequence diagram** shows the flow of messages between software objects, and thus the invocation of methods.
  - UML **class diagram** illustrates the static relationship between objects.

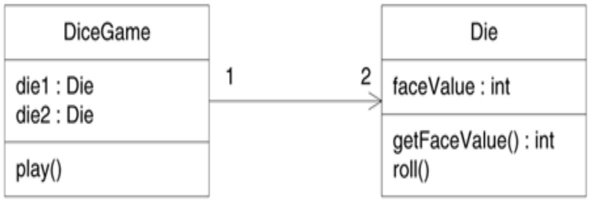
A Sequence Diagram Example

Figure 1.4. Sequence diagram illustrating messages between software objects.



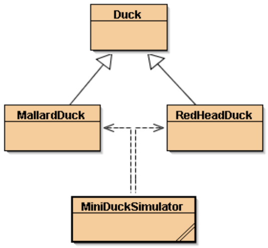
A Class Diagram example

Figure 1.5. Partial design class diagram.

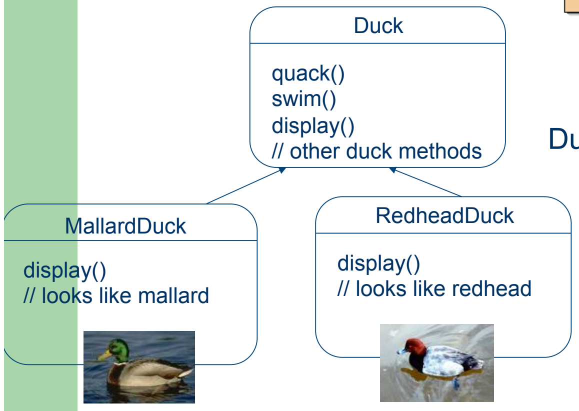


Class diagram illustrates the attributes and methods of the classes.

A short Example of Des Patterns



Duck Behavior Simulator



Duck

quack()
swim()
display()
// other duck methods

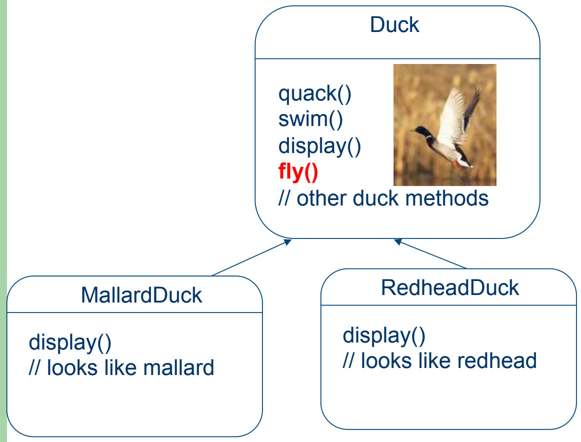
MallardDuck

display()
// looks like mallard

RedheadDuck

display()
// looks like redhead

What if we want to simulate flying ducks?



Duck

quack()
swim()
display()
fly()
// other duck methods

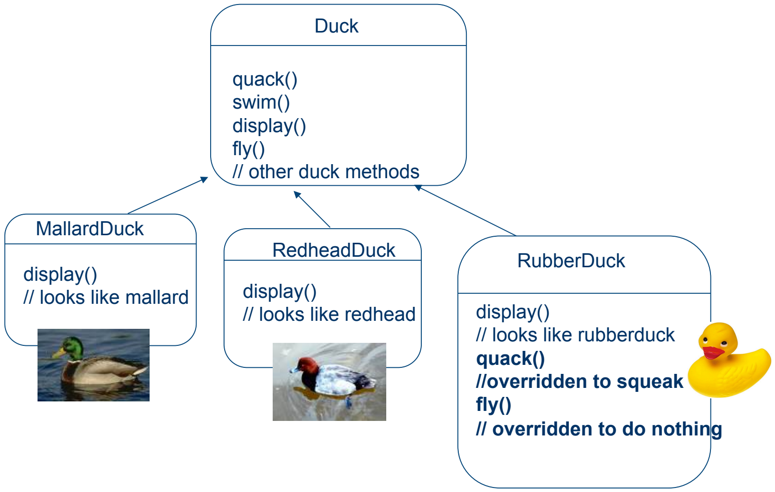
MallardDuck

display()
// looks like mallard

RedheadDuck

display()
// looks like redhead

Other ducks?



Duck

quack()
swim()
display()
fly()
// other duck methods

MallardDuck

display()
// looks like mallard

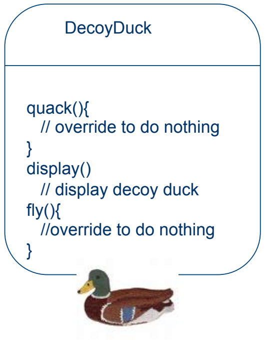
RedheadDuck

display()
// looks like redhead

RubberDuck

display()
// looks like rubberduck
quack()
//overridden to squeak
fly()
// overridden to do nothing

More ducks: add a wooden decoy ducks to the mix

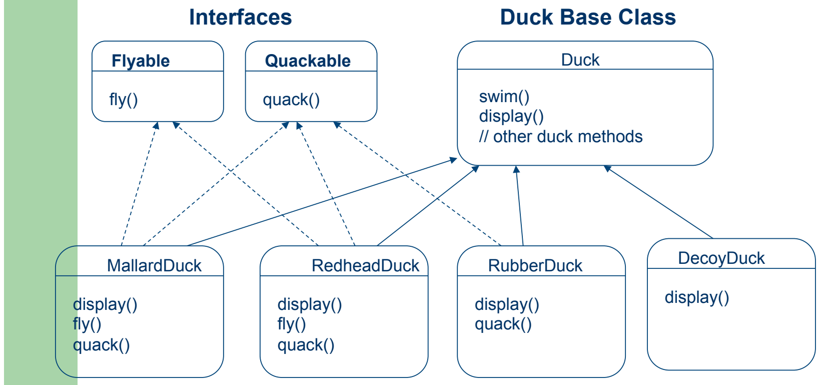


DecoyDuck

```
quack(){
   // override to do nothing
}
display()
   // display decoy duck
fly(){
   //override to do nothing
}
```

- Applying inheritance to achieve re-use
- Poor solution for maintenance

Another solution:



Interfaces

Duck Base Class

Flyable

fly()

Quackable

quack()

Duck

swim()
display()
// other duck methods

MallardDuck

display()
fly()
quack()

RedheadDuck

display()
fly()
quack()

RubberDuck

display()
quack()

DecoyDuck

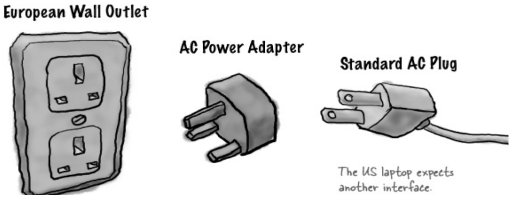
display()

Design Patterns

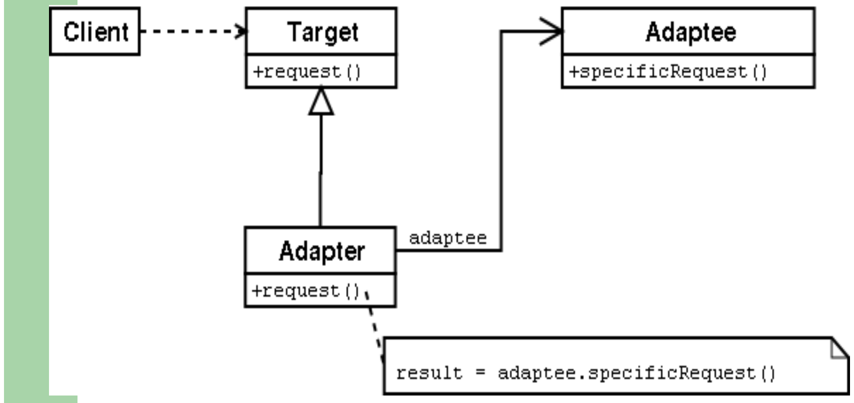
- A **design pattern** is not a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.
- OO design patterns shows relationships and interactions between classes or objects, but without specifying the final application classes or objects that are involved.

Adapter Design Pattern

- The intent of Adapter is to
  - *Convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.*
- Use it when you need a way to *create a new interface for an object that does the right stuff but has the wrong interface*



GoF Adapter Pattern Structure



Client

Target

+request()

Adaptee

+specificRequest()

Adapter

+request()

adaptee

```
result = adaptee.specificRequest()
```

More on Adapter Pattern

Pattern Participants:

- **Client**. The client class is that which requires the use of an incompatible type.
- **Target**. This is the expected interface for the client class.
- **Adaptee**. This class contains the functionality that is required by the client. However, its interface is not compatible with that which is expected.
- **Adapter**. This class provides the link between the incompatible Client and Adaptee classes.

Consequences: Allows for preexisting objects to fit into new class structures.