Design to Implementation

Jesper Andersson

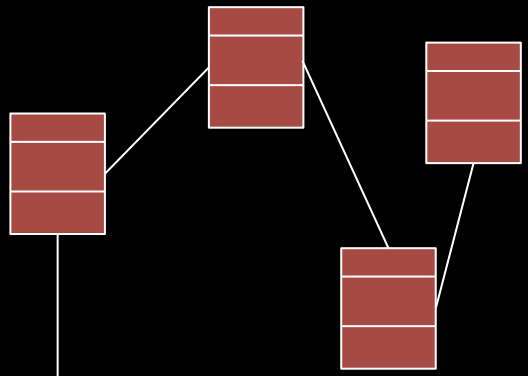
Linnæus University

From Design to Implementation

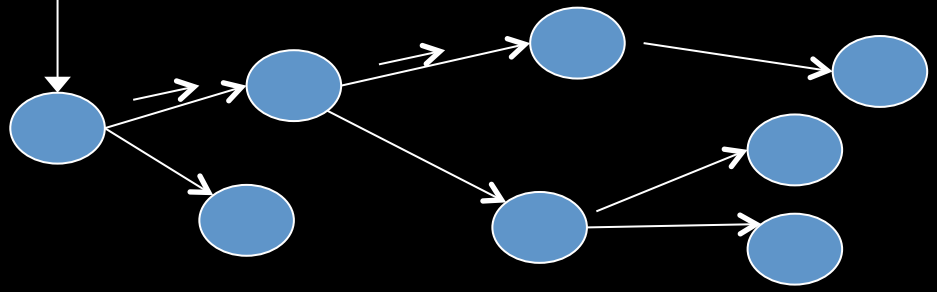
- ✓ Design Decisions in the small
- ✓ How do you express your design
  - Collaboration of Objects
  - Model these collaborations
- ✓ Best Practices for object collaborations
  - Abstraction
  - Hierarchy
  - Encapsulation
  - Design Patterns

Classes (Types) and Objects

- ✓ What are we trying to model?
  - Static – **Blue print** of participating entities
  - Dynamic – **Instances** and the protocol they follow in their communication

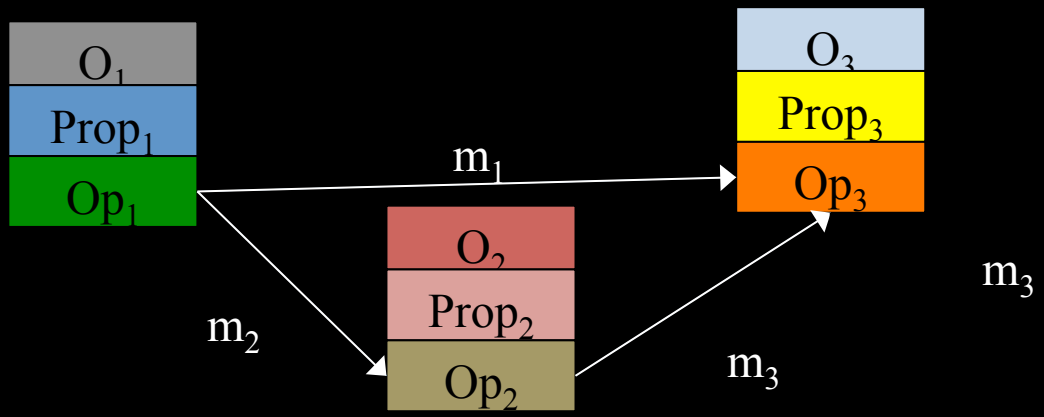


instantiate



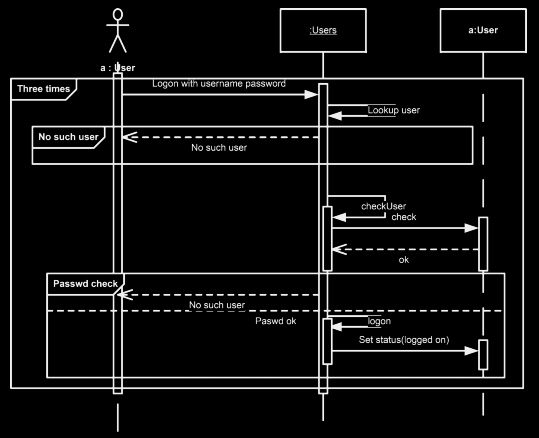
Object-Oriented Applications

- ✓ Collection of objects that interact with each other
- ✓ Objects have property and behavior (state transition)
- ✓ A sender object sends a request(message) to another object (receiver) to invoke a method of the receiving object



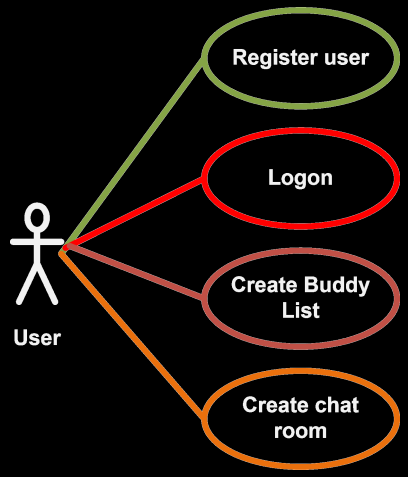
UML – Object Diagrams

- ✓ Describe Object Interactions
  - – Messaging
  - – Structure
    - • Ifs
    - • Loops
- ✓ Dynamic diagrams



Collaborations

- ✓ Object Models – “implement a use case (or part of)
  - Static
  - Dynamic
- ✓ Demonstrate high-level functionality (details)
- ✓ Model verification
  - Do you have the right classes?
  - Have you assigned the right responsibilities?
  - Have you specified all details?



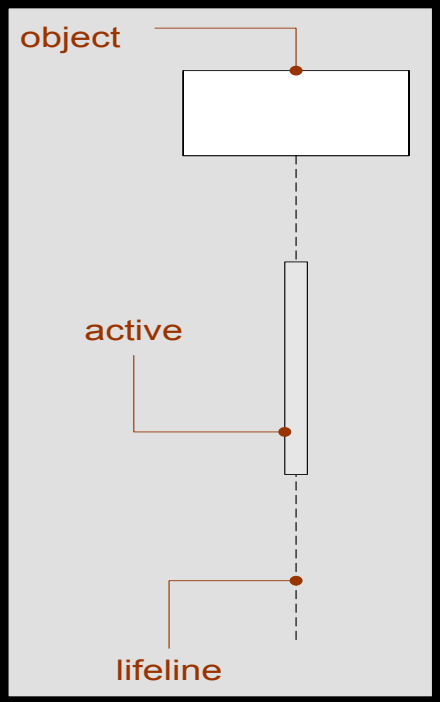
Interaction Overview Diagrams

- ✓ Top-level description of where the main nodes are individually documented using separate sequence (or other) diagrams.
- ✓ Interaction Frames – ‘encapsulates interactions’

Sequence Diagrams

- ✓ Displays a collaboration between
  - Objects
  - Actors
- ✓ Time passes as we move from the top to the bottom.
- ✓ Depicts communications
  - Synchronous and Asynchronous
  - Self
  - Return values
- ✓ Creation and deletion of objects

SD: Building blocks

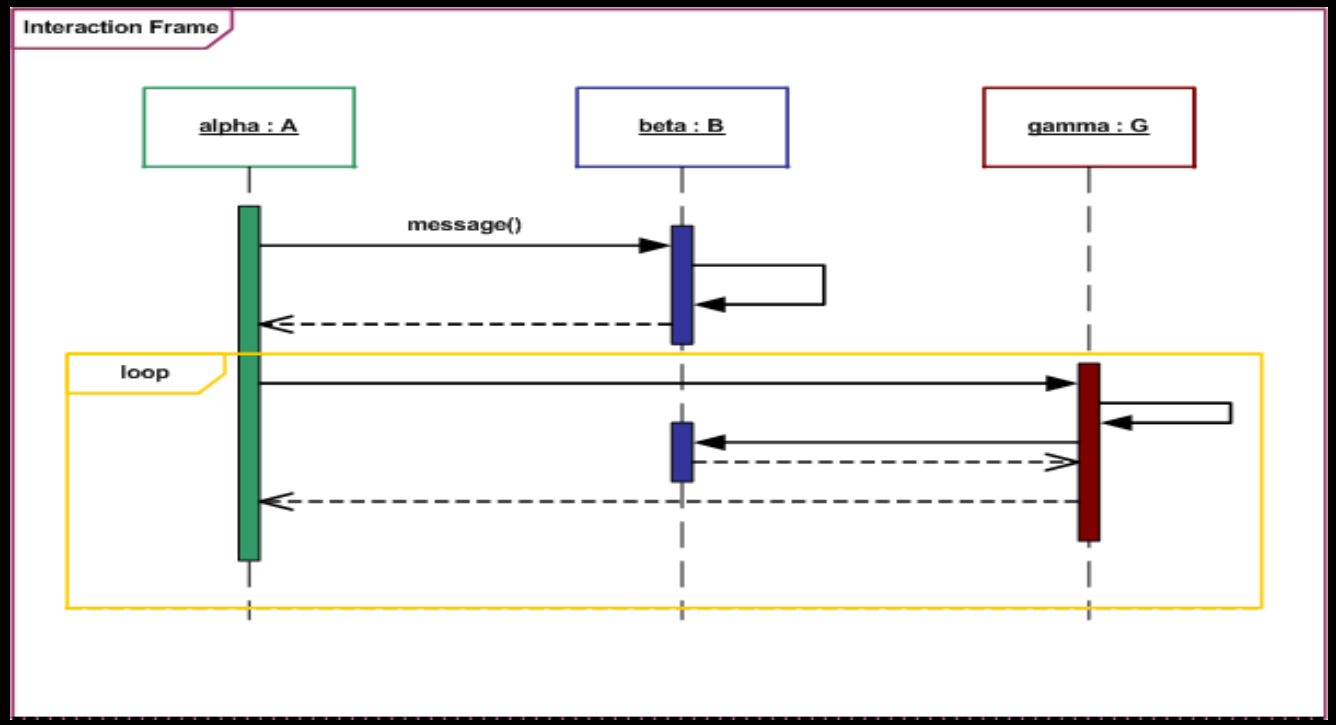


Object referencing

`A` – name only
`A : ClassName` – name and Class specified
`: ClassName` – some object of ClassName

Message types

Sequence Diagram Example

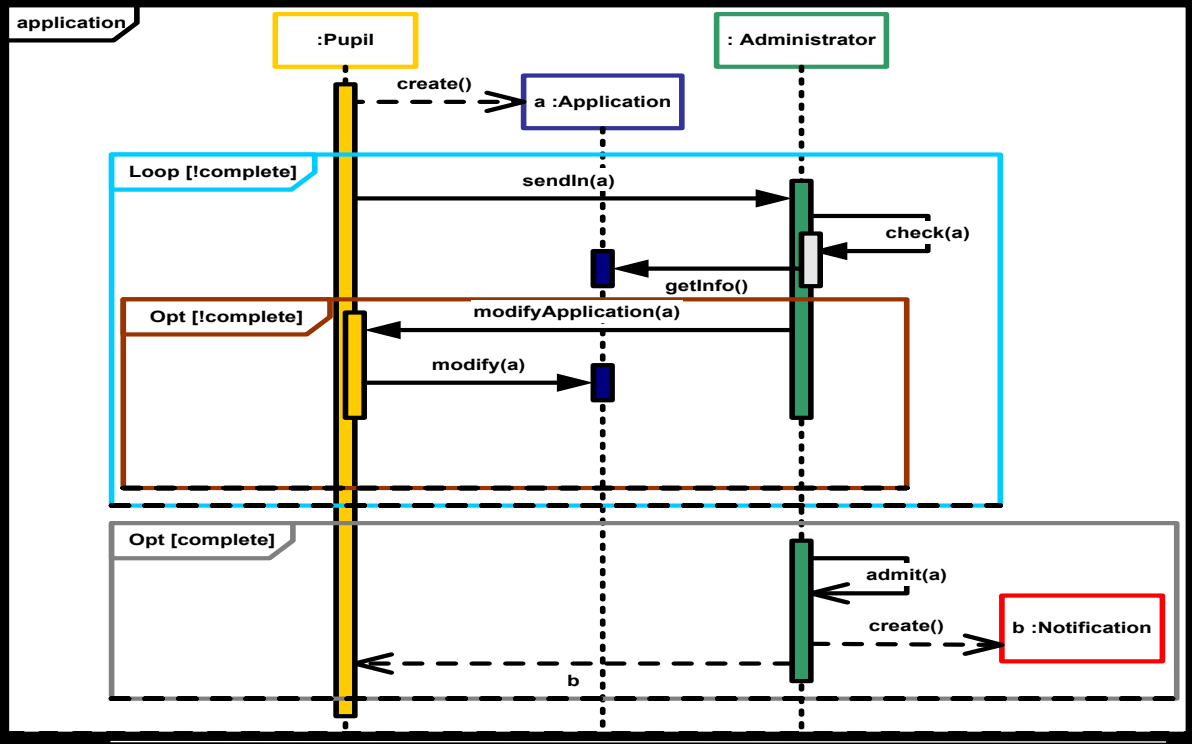


Advanced Sequence Diagrams

- ✓ Interaction frames
  - **Conditional**
    - Optional - Opt
    - Alternative - Alt
  - **Iterations**
    - Loop - Loop

alt – Alternative multiple fragments; only the one whose condition is true will execute
opt – Optional; the fragment executes only if the supplied condition is true. Equivalent to an alt with only one trace
sd – Sequence diagram; used to surround an entire sequence diagram, if you wish.

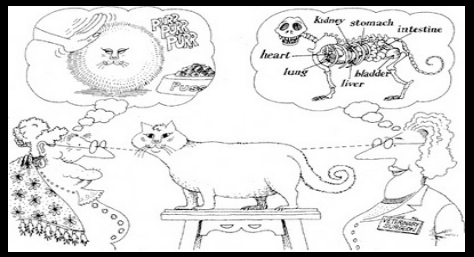
Example from a Design Model



Properties of Good Object Models

- ✓ When you design your Object Model
  - Abstraction – Reduce information
  - Modularity – Divide models up
  - Hierarchy – Structure models
  - Encapsulation - Interface for interactions

Abstractions



- ✓ Examples
  - Object
  - Class
  - Interfaces
  - Operation

"a ***simplified*** description, or specification, of a system that ***emphasizes some*** of the system's ***details*** or properties ***while suppressing others***.

A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary." -- Shaw, M. 1984

Modularity

Decomposing a system in to its parts

- ✓ Logical or Physical modules

- ✓ Examples
  - Classes (Logical)
  - Packages (Logical)
  - Files (Physical)

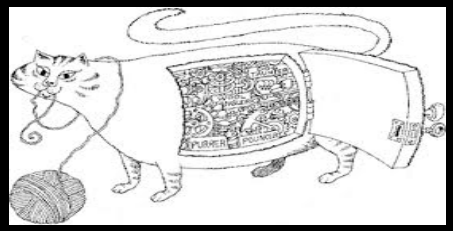
Hierarchy

- ✓ Compose subsystems into larger systems

Hierarchy is the **ranking** or **ordering** of **abstractions**

- ✓ Inheritance (prototype chain)

Encapsulation – Information hiding



"the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; ***encapsulation serves to separate the contractual interface of an abstraction and its implementation.***"

- ✓ Examples
  - – Class interface in Java
    - Attributes
    - Operations
  - – Access modifiers (Java)

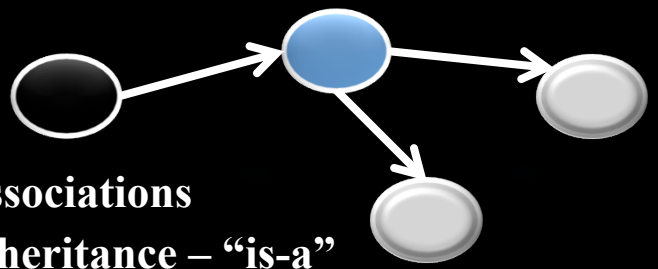
Class diagram

- ✓ Models the static structure of a system
  - Classes,
  - Packages, and
  - how they are related

Classification – Derive models

- ✓ The process of identifying a class model which reflects the required objects.
- ✓ Elements – Classifiers
  - – The Class – Describes properties of several similar objects ~ concepts.

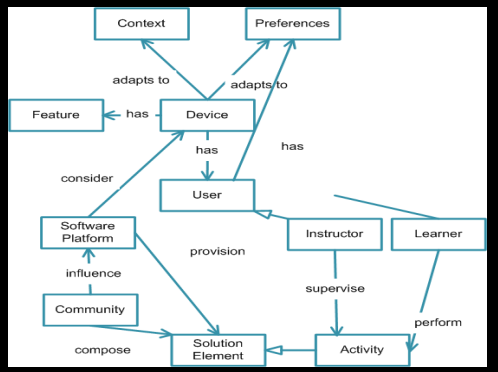
- ✓ Connect them in models
  - – Aggregate objects – **”part-whole”**



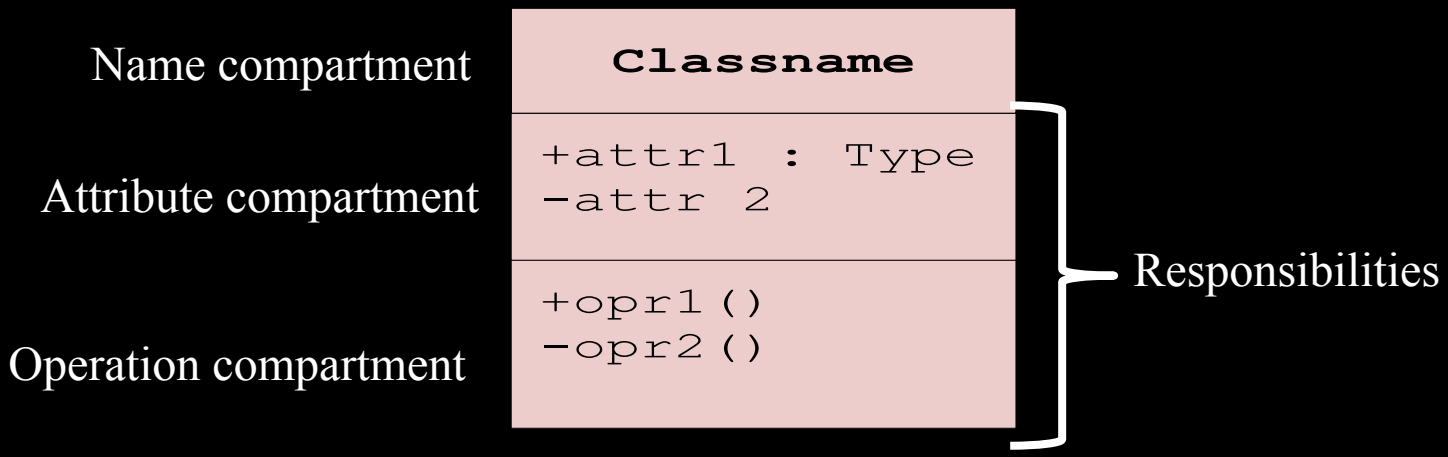
  - – **Associations**
  - – **Inheritance – “is-a”**

UML – Class Diagram

- ✓ Main diagram in OO modeling
  - **Conceptual modeling**
  - **Detailed modeling** for transformation into code
- ✓ Class diagrams represent main
  - objects and
  - main object interactions



Class Icon


Name compartment
Classname
Attribute compartment
+attr1 : Type
-attr 2
Operation compartment
+opr1()
-opr2()
Responsibilities

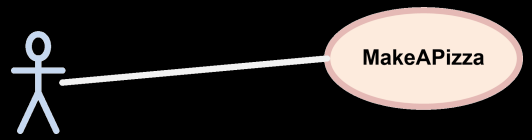
Object Oriented Design – a.k.a. OO Problem solving

- ✓ Two levels
  - Describe the objects (and object's **responsibilities** in the solution)
  - Describe how objects work together to solve a problem
- ✓ Describe objects of a class
  - The class is an abstraction (describes many objects)
  - Which data must the object have? *State - Attributes*
  - Which services shall it provide? *Behavior – Operations*

Problem: Hunger Solution: Pizza!

- ✓ You need a recipe!

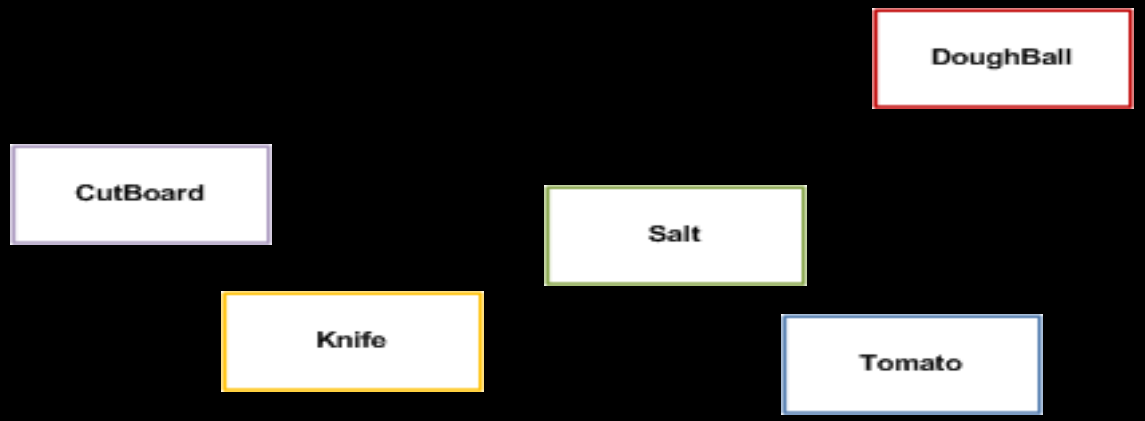
- ✓ First question – Which recipe?
  - Requirements
    - Ingredients (must be garlic)
    - Complexity (less than 2 hours)
    - Cost (less than 15 RNB)
  - Where to find such a recipe?
    - Cookbook
    - You create your own new recipe



The objects – Our Ingredients and “Equipment”

4 Individual Pizza Dough Balls - Defrosted
2 Cups Chopped, Ripe Plum Tomatoes
4 Tablespoons Fresh Chopped Basil
Salt And Pepper
Red Pepper Flakes
4 Tablespoons Olive Oil
2 Cloves of Garlic, Minced
3 Cups Shredded Mozzarella Cheese
Oven, Knife, Cut board
Rolling pin

Classification of the the Pizza example



Classes - Responsibilities

CutBoard

DoughBall

Tomato

Whole

Attributes - State

Methods - Behavior

Color

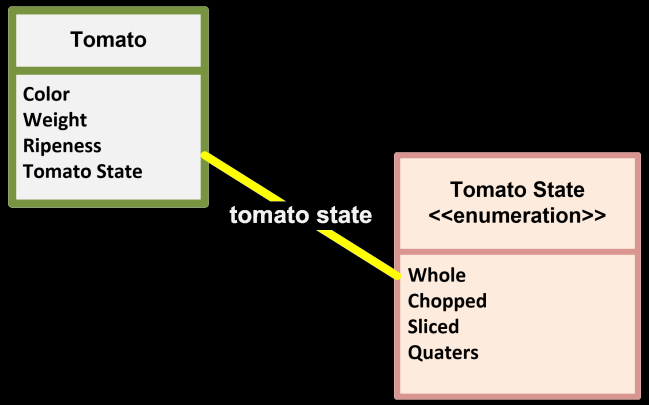
Weight

Ripeness

Sliced

Chopped

Quarters



Classes and their Responsibilities

Encapsulation

Modularity

Abstraction

Hierarchy

Change Color

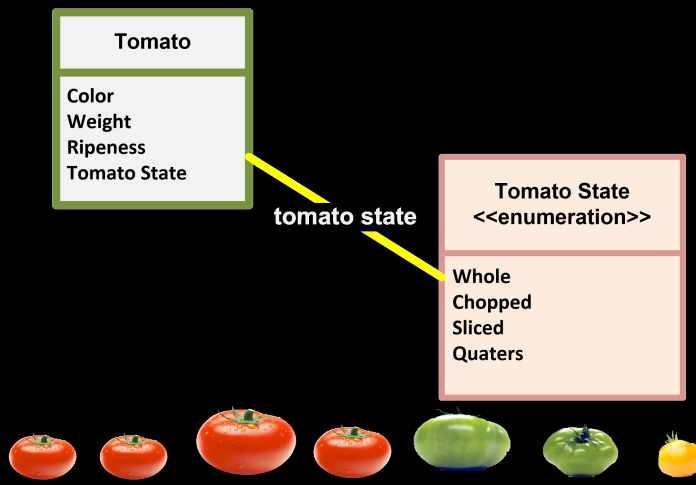
State

Change tomato's state

Behavior

Tomato's ripeness

Tomato's weight



Now we have our Objects described!

4 Individual Pizza Dough Balls - Defrosted
2 Cups Chopped, Ripe Plum Tomatoes
4 Tablespoons Fresh Chopped Basil
Salt And Pepper
Red Pepper Flakes
4 Tablespoons Olive Oil
2 Cloves of Garlic, Minced
3 Cups Shredded Mozzarella Cheese
Oven, Knife, Cut board, Rolling pin

What's missing?

Collaboration!

```
theCutBoard.put(theTomato);
theKnife.chop(theCutBoard.getItems());
```

- ✓ No collaboration – Nothing happens!
- ✓ Real world
  - Atoms interact (subatomic particles, e.g. electron)
  - No interaction – nothing happens
- ✓ Pizza example
  - Our ingredients and the “equipment” must interact
  - No interaction – nothing happens!

Object-Oriented Applications – Pizza example

- ✓ Collection of objects interacting with each other
- ✓ Objects have property and behavior (state transition)
- ✓ A sender object sends a request(message) to another object (receiver) to invoke a method of the receiving object

```
theCutBoard.put(theTomato);
theKnife.chop(theCutBoard.getItems());
```

Example – Continued

```
class Knife {
    // class descriptions describe all objects of this class
    // object's state and possible behavior
    // think of it as a state machine

    private float length;
    private Manufaturer make;
    …

    public void chop(Collection<IChoppable> someObjects)
    public void stab(IStabbable anObject)
    public void slice(ISliceable anObject)
}
```

How objects interact - Message passing

- ✓ Objects in specific state may
  - **receive messages,**
  - **process messages,**
  - **and return to ”callee”**
- ✓ Sender **must** have a path along which the message can be passed
- ✓ Receiving objects **must** “*understand*” the message – have a matching operation with a method that *matches* the message.

Example – Continued

```
class CutBoard{
    private Collection <ICuttable> objects;
    private Manufacturer make;
    …
    public void put() {ICuttable object }
    public Collection getItems() {…}
    …
}
```

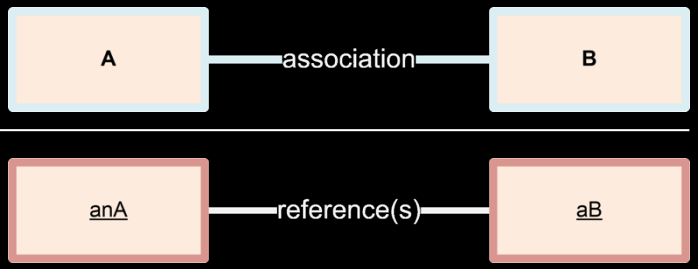
```
theCutBoard.put(theTomato);
theKnife.chop(theCutBoard.getItems());
```

Goal – Dynamic Structure

Preheat the oven to 425 degrees F. Place the chopped tomatoes in a sieve, and sit it over a bowl for 30 minutes. Once drained, place the tomatoes in a bowl, and add the basil, olive oil, salt, pepper, red pepper flakes and the garlic. Roll out the pizza into 4 thin discs. Spoon the tomato mixture evenly over each of the four pizzas. Divide the cheese and place it on top of the tomatoes. Bake the pizzas for about 15 minutes, or until the top is golden and bubbly, and the crust has begun to brown. Serve immediately.

```
theOven.start(425);
theCutBoard.put(theTomatoes);
theKnife.chop(theCutBoard.getItems());
...
```

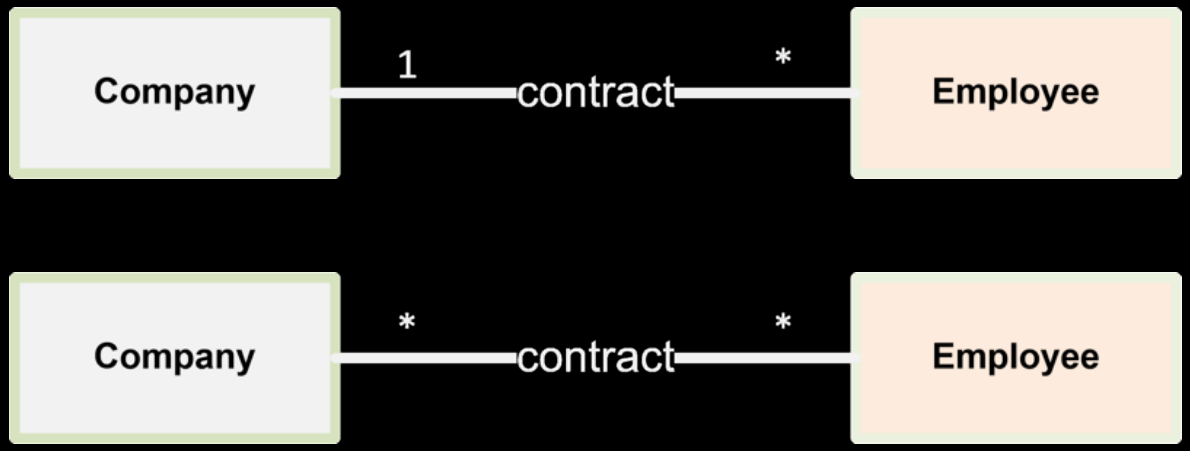
Associations



- ✓ Connects one class to another
  - Links objects of one class to objects of another or the same
- ✓ An alternative way to specify an attribute
- ✓ Hmm, alternative…
  - Small things as attributes such as value types
  - Significant information (context dependent) as classes -> associations

Class – Association, Semantics

- ✓ What are the semantics of this model?

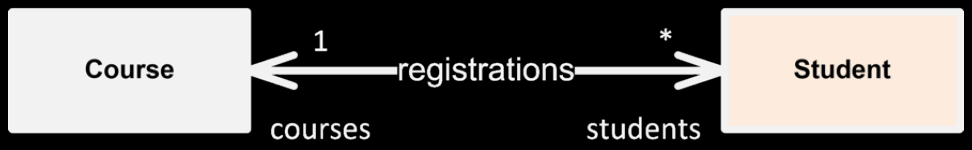


Properties of an Association

- Name
- Role names
- Multiplicity
- Navigability

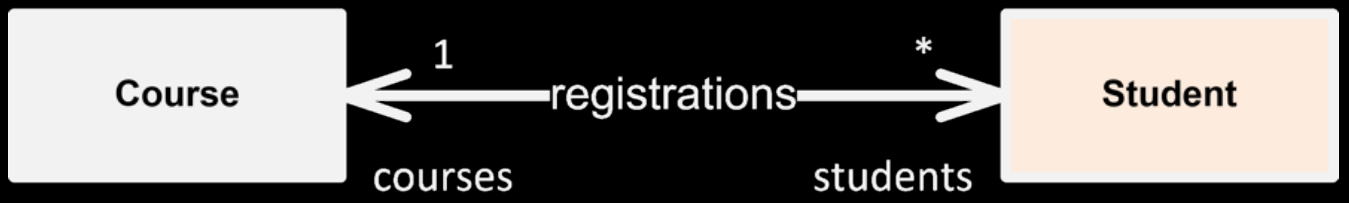
UML Association names

- ✓ Association name
  - "Explains" the association
  - "Read out"
- ✓ Attribute names
  - An association may play different roles at different ends. Simplifies reading!
  - Use only when it provides useful information!



UML Navigability

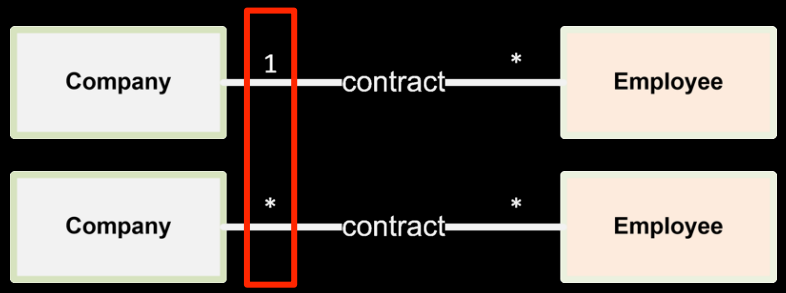
- ✓ Unidirectional
- ✓ Bidirectional
  - Two properties in different classes that are linked together as inverses
  - Decide which route a message may take!



UML Multiplicities

Attributes on associations that specify more details about the relationship

Multiplicities	Meaning
0..1	zero or one instance. The notation ***n . . m*** indicates ***n*** to ***m*** instances.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance



Principles of Good Design

- ✓ High Cohesion – Low Coupling
- ✓ Avoid
  - Rigidity, It is hard to change because every change affects too many other parts of the system ➔**Ripple effects!**
  - Fragility, When you make a change, unexpected parts of the system break ➔ **Implicit dependencies!**
  - Immobility, Difficult to reuse in another application because it cannot be “factored out” from its current application.
- ✓ Design principles are the cornerstones in most Design Patterns.

Software Patterns, a Classification

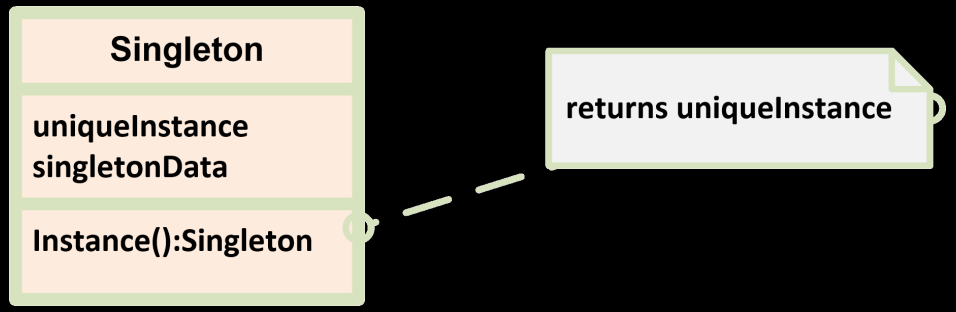
- ✓ Architectural patterns
- ✓ Analysis Patterns
- ✓ Design Patterns
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- ✓ Idioms

Creational patterns

- ✓ Deal with object creation mechanisms
- ✓ Trying to create objects in a manner suitable to the situation.
- ✓ Creational design patterns solve this problem by somehow controlling this object creation.

- ✓ The Singleton - Ensure a class only has one instance, and provide a global point of access to it.

Singleton

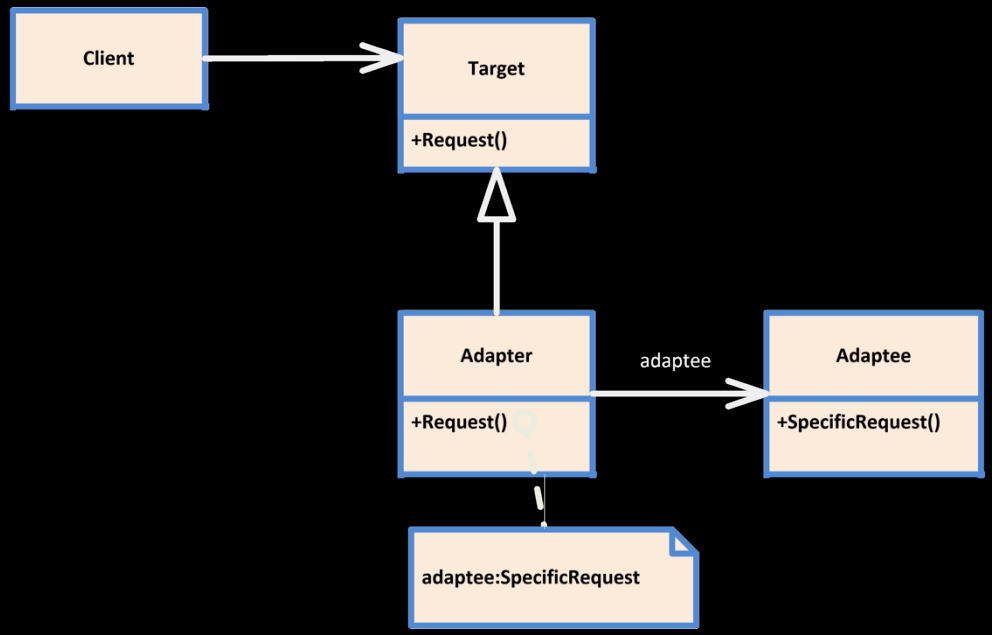



```
public class Singleton {

/** * The constructor could be made private to prevent others from
instantiating this class. But this would also make it impossible to *
create instances of Singleton subclasses. */
protected Singleton() { // ... }
/** * A handle to the unique Singleton instance. */
static private Singleton _instance = null;
/** * @return The unique instance of this class. */
static public Singleton instance() {
   if(null == _instance) {
      _instance = new Singleton();
   }
   return _instance;
}
 /. ...additional methods omitted...
}
```

Structural Patterns

- ✓ Ease design by identifying a simple way to realize relationships between entities
- ✓ The Adapter - Convert the interface of a class into another interface clients expect.
- ✓ Adapter lets classes work together that couldn't otherwise because of incompatible interfaces

Adapter Pattern



Adapter Example

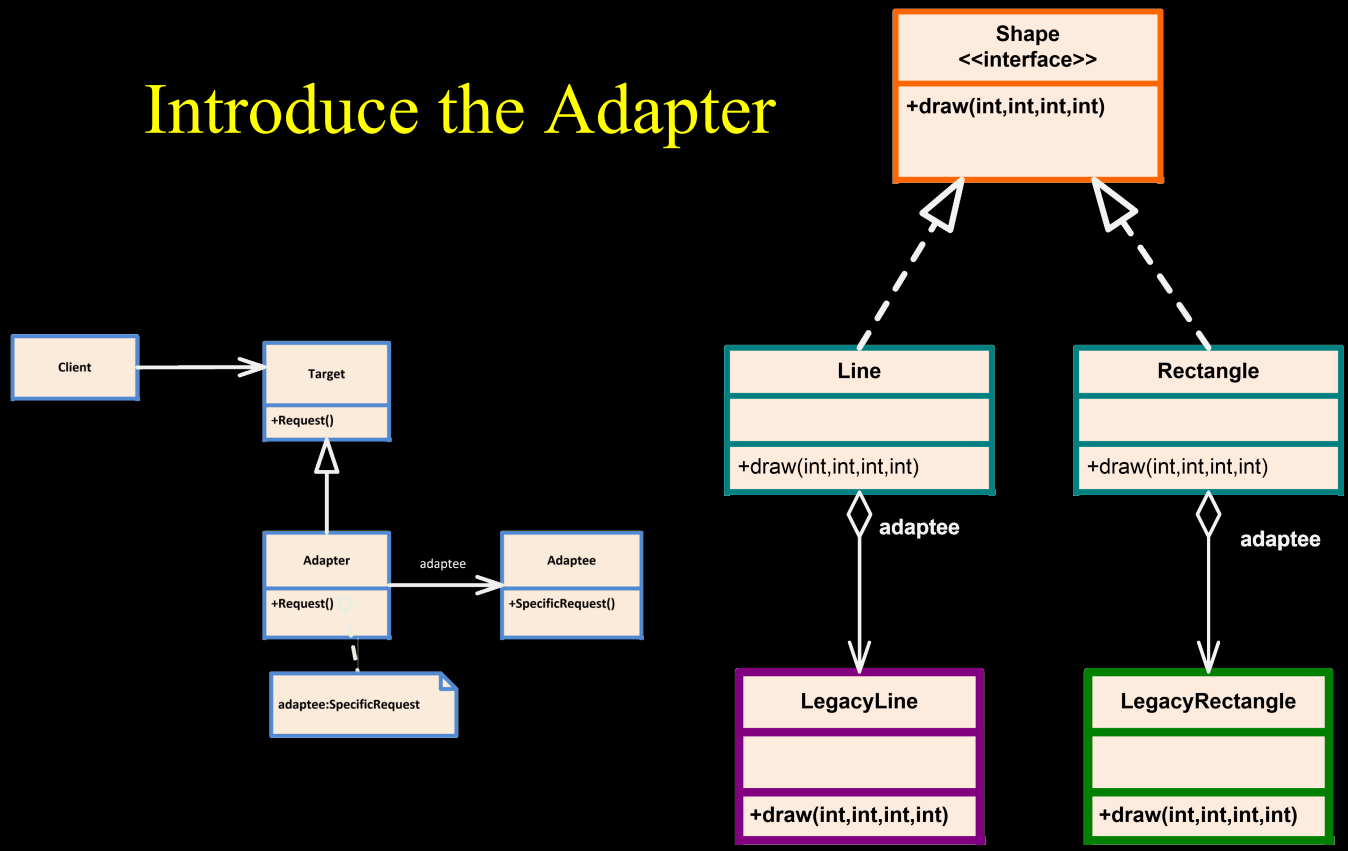

```
class LegacyLine {

   public void draw(int x1, int y1, int x2, int y2)

   {

to

}
```

```
class LegacyRectangle{

   public void draw(int x, int y, int w, int h)
   {
       System.out.println("rectangle at (" + x + ',' + y + ") with
width " + w

         + " and height " + h);

   }

}
```

AdapterDemo

```
public class AdapterDemo {
   public static void main(String[] args) {
       Object[] shapes = {new LegacyLine(), new LegacyRectangle()};
       // A begin and end point from a graphical editor
       int x1 = 10, y1 = 20;
       int x2 = 30, y2 = 60;
       for (int i = 0; i < shapes.length; ++i)
         if (shapes[i].getClass().getName().equals("LegacyLine"))
           ((LegacyLine)shapes[i]).draw(x1, y1, x2, y2);
         else if (shapes[i].getClass().getName().equals("LegacyRectangle"))
           ((LegacyRectangle)shapes[i]).draw(Math.min(x1, x2), Math.min(y1, y2)
             , Math.abs(x2 - x1), Math.abs(y2 - y1));
   }
}
```

Introduce the Adapter



Line and Rectangle

```
class Line implements Shape {
   private LegacyLine adaptee = new LegacyLine();
   public void draw(int x1, int y1, int x2, int y2)
   {
       adaptee.draw(x1, y1, x2, y2);
   }
}
```

```
class Rectangle implements Shape {
   private LegacyRectangle adaptee = new LegacyRectangle();
   public void draw(int x1, int y1, int x2, int y2) {
      adaptee.draw(Math.min(x1, x2), Math.min(y1, y2),
                   Math.abs(x2 x1),Math.abs(y2 - y1));
   }
}
```

New AdapterDemo

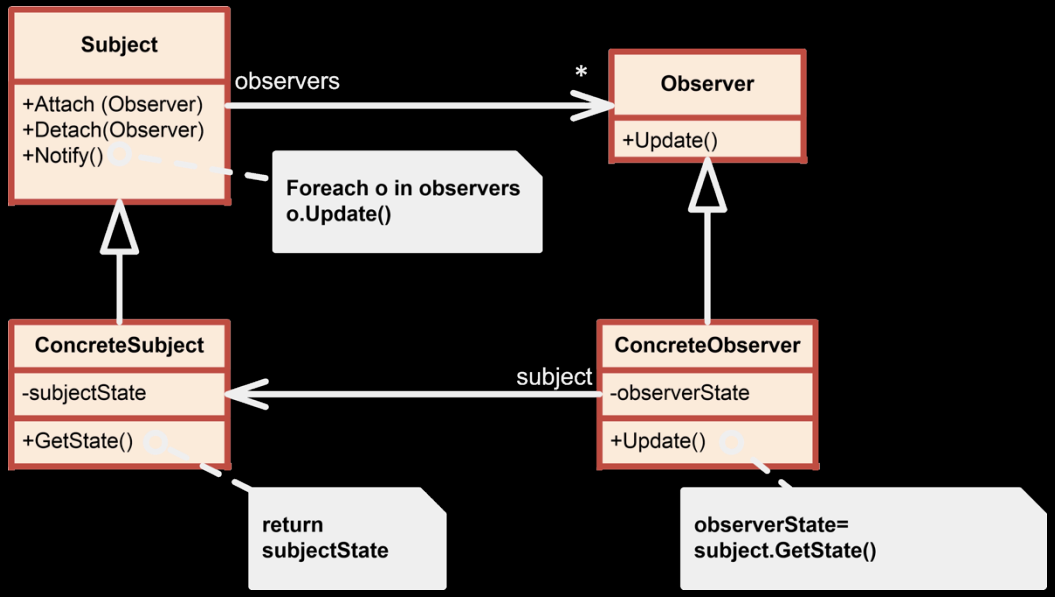
```
public class AdapterDemo
{
    public static void main(String[] args)
    {
        Shape[] shapes =
        {
            new Line(), new Rectangle()
        };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
          shapes[i].draw(x1, y1, x2, y2);
    }
}
```

Behavioral patterns

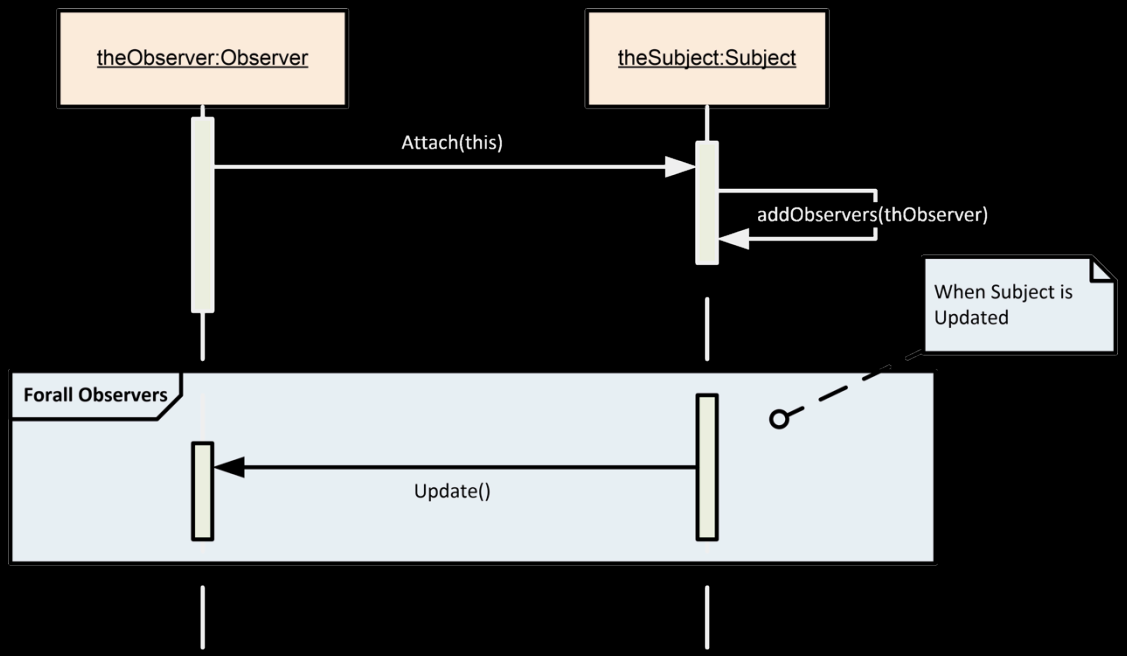
- ✓ Identify common communication patterns between objects
- ✓ Realize these patterns.
- ✓ These patterns increase flexibility in carrying out this communication.

- ✓ The Observer Pattern - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Observer Pattern



Observer – Sequence Diagram



Today's takeaways

- ✓ Design Decisions at all levels
- ✓ Design you use cases with interacting objects
- ✓ UML Object Diagrams
- ✓ UML Class Diagrams

- ✓ Design Principles
- ✓ Design Patterns