Object Oriented Analysis, Design, & Development(2013)

Styles HotKeys:

Heading 1-6:	C-A-[1,2,3,4,5,6]
code1:	C-A-C,C
code-Output:	C-A-C,V
Bullets:	C-S-L

Chapter 1:

Software Development Methodologies

Waterfall

Analysis > Design > Implementation > Installation > Maintenance

Agile / Iterative approach

Repeating cycles, getting closer to the goal
Just enough design to move forward successfully
"Good Enough"

Object Oriented Languages

Separate program into modular blocks

Procedural Languages

ProLog – Logic programming language
Haskell – Functional programming language

What is an Object?

Not always physical items
Not always visible items
Can you put 'THE' in front of it?

Example:

Bank Account

- Property: Number
- Property: Balance
- Method: Deposit
- Method: Withdrawl

What is a Class?

The Blueprint of an Object
NOT the Object itself

type
name: what is it?

Employee, BankAccount, Event, Player, Document, Album

Properties, data
Attributes: what describes it?

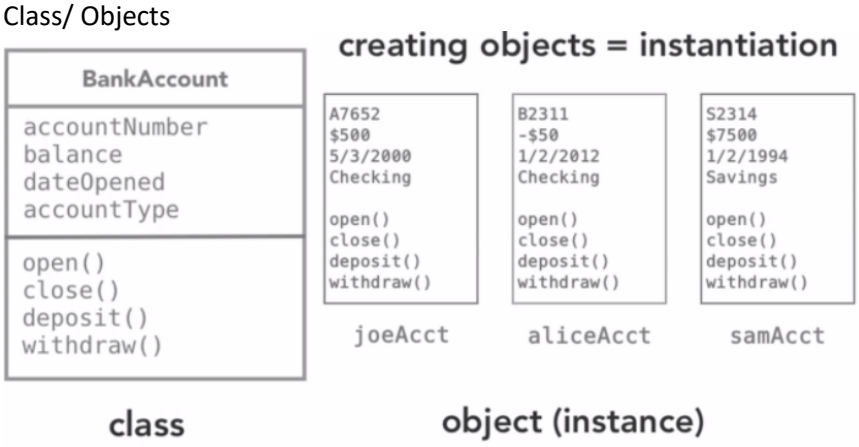
Width, Height, Color, Score, FileType, Length

Operations, methods
Behavior: what can it do?

Play, Open, Search, Save, Print, Create, Delete, Close

Example:
Name: BankAccount
Attributes: accountNumber, balance, dateOpened, accountType
Behavior: open(), close(), deposit(), withdraw()

Class/ Objects



Existing Classes in OO Languages

- Most OOP have predefined classes: strings, dates
- Java Class Library
- .NET Framework BCL
- C++ Standard Library
- Ruby Standard Library
- Python Standard Library

4 Fundamental Ideas when Creating Classes

- Abstraction
- Polymorphism
- Inheritance
- Encapsulation

Abstraction

- Focus on the essentials
- Ignore the irrelevant
- Ignore the unimportant

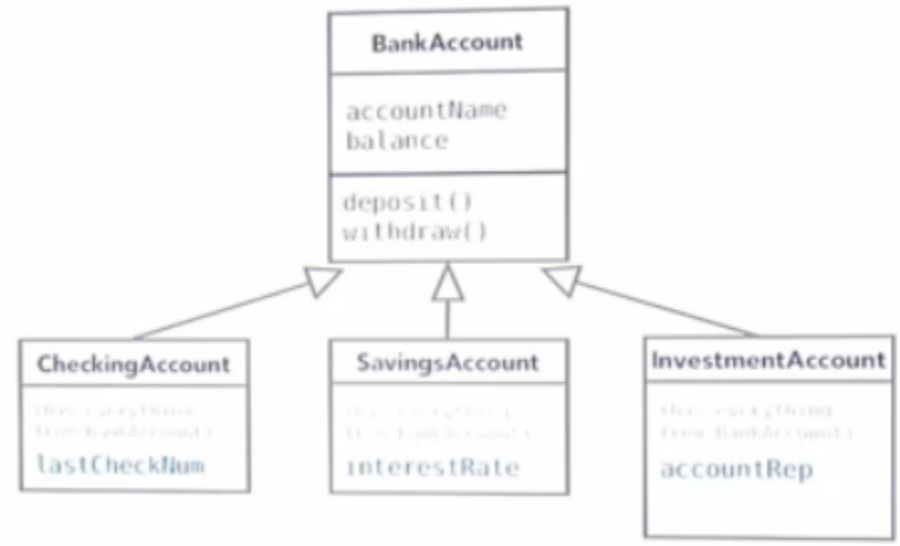
Example:

Polymorphism ("many forms")

- Automatically do correct behavior if class can do many things
- With inheritance…can change base class when inheriting.
- a + b
    - if string concatenation
    - if integer, arithmetic
- Overriding method of base class is one form
- Inheriting when useful, overriding when useful

Example:

Can call the withdrawl() method regardless of account type

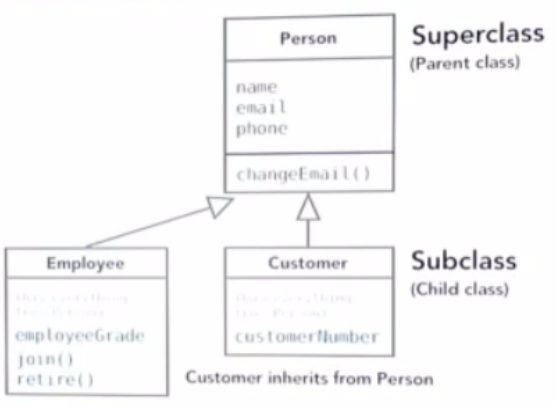


Inheritance

- Code Re-use
- Hand down attributes from one class to another – no need to re-write code

Person > Customer(+accountNumber)
Person > Employee(+companyName)

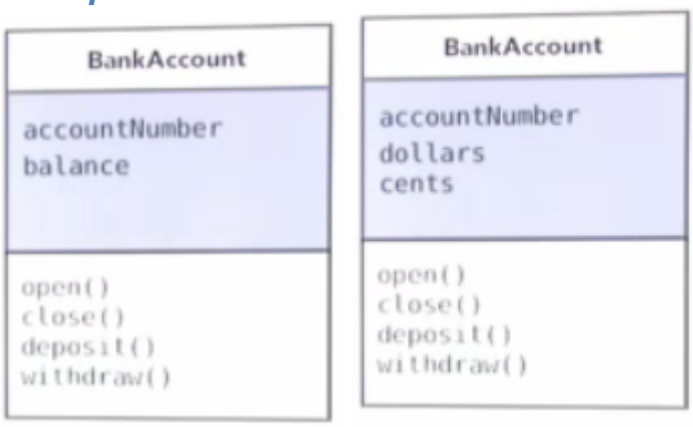
Example:



Encapsulation

- Bundle Attributes and behaviors in same class
- Restrict access to those attributes
- Reducing dependencies

Example:



Chapter 2: Object Oriented Design Process

1. Gather Requirements
    a. What does the app need to do?
2. Describe the app
    a. Build a simple narrative in common language
    b. Use cases and user stories
    c. Smallest number of stories to suffice
    d. Prototype of UI
3. Identify the main objects
    a. Use stories/descriptions to create objects
    b. Should be basis for classes
4. Describe the Interactions
    a. Work out how the objects work together
    b. Sequence diagram
5. Create a Class Diagram
    a. Visual representation of class

Gathering Requirements

Functional Requirements:

- What does it need to do
- Feature
- Capabilities

Example:

System must:

- Allow user to search by customer's last name.

Non-Functional Requirements:

- What else?
- Help
- Legal
- Performance
- Support

Example:

System must

- respond to searches within 2 seconds
- Helpdesk available by phone 24/7

FURPS / FURPS+

- Functional
- Usability
- Reliability
- Performance
- Supportability
- +Design
- +Implementation
- +Interface
- +Physical

Go for minimum set of requirements to satisfy needs
WHAT IS REQUIRED?

Status ok if:

- Not Applicable
- TBA

SOMETHING WRITTEN DOWN

UML is at Tool in this process

Chapter 03

Use cases

- Title – what is the goal?
- Actor – who desires it?

- Scenario – how is it accomplished?

Title

Short phrase, active verb

- Register new member
- Transfer funds
- Purchase items

Actor

Need to identify WHO is having the interaction

- User
- Customer
- Member
- Administrator

Scenario

Details of accomplishing this one goal

One paragraph:

Title: Purchase items
Actor: Customer
Scenario: Customer reviews items in basket, checks out and pays

As Steps:

Title: Purchase items
Actor: Customer
Scenario:

1. Customer reviews items in basket
2. checks out
3. pays

Additional Details

Extensions: Describe steps for out-of-stock situations
Extensions: Describe steps for order never finalized
Precondition: Customer has added at least one item to shopping cart

Fully Dressed Use Case

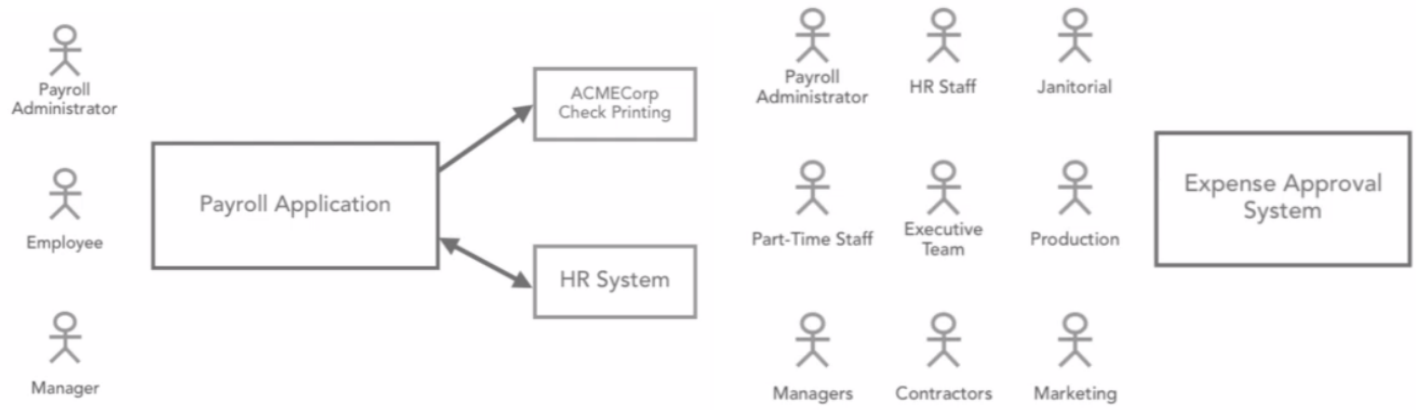
Templates useful here

Title: Purchase items
Actor: Customer
Secondary actor: ...
Scenario: ...
Description: ...
Scope: ...
Level: ...
Extensions: Describe steps for out-of-stock situations
Extensions: Describe steps for order never finalized
Precondition: Customer has added at least one item to shopping cart
Postcondition: ...
Stakeholders: ...
Technology list: ...

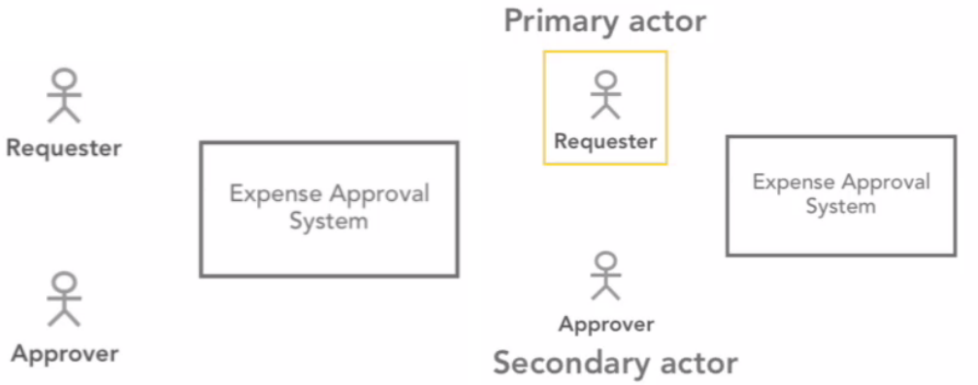
Other
Spend 1 or 2 days per iteration
Only written text, no diagrams

Identify Actors
Actor – anything with behavior outside who lives outside of app, but has goal to accomplish with it
Questions:

- Other computer systems/organizations
    - External data sources, web services, other corp apps, tax reporting, backup systems
- Distinguish between roles/security
    - Visitor, member, admin, owner
- Job titles / Departments
    - Manager, payroll admin, Production Staff, Exec, Account
- Focus not on ROLE, but GOAL actor wants to accomplish
    - Different roles may have SAME GOAL



Many different ROLES, but only two main GOALS



Identify Scenarios

Emphasize the goal of ONE encounter

GOOD:

- Purchase items
- Create new Document
- Balance accounts

BAD: Too Big

- Log in to application
- Write Book
- Merge Organizations

Multiple Scenarios

Use Case Writing Tips

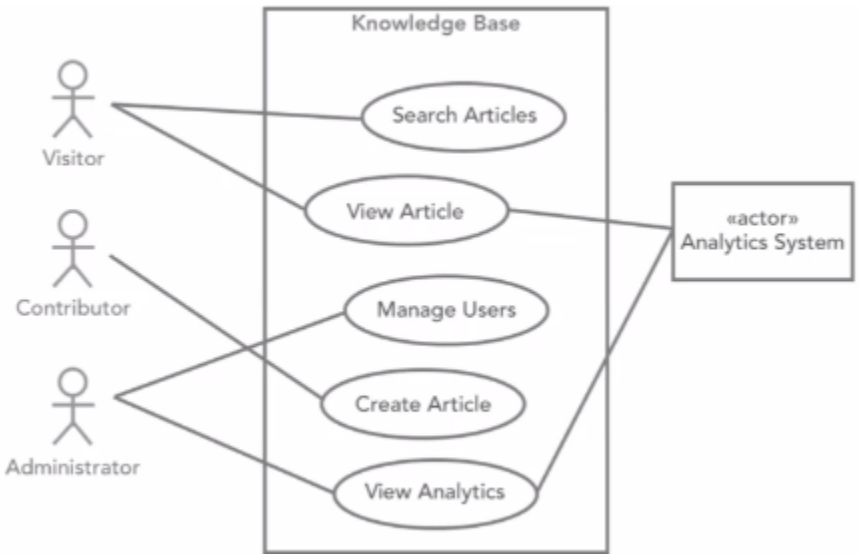
- Keep it simple
  - Use Active Voice.
  - Omit Needless words.
  - Shoot for one sentence
- Focus on intention, leave UI out of it
- Use Case prompts (helps ID actors)
  - Who performs sys admin tasks
  - Who managers users and security
  - What happens if the system fails
  - Is anyone looking at performance metrics or logs

Diagramming Use Cases

Knowledge Base Example

- Box Application name and Titles (elipses)
- Titles (elipse)
  - Search Articles
  - View Article manage users
  - Create Article

    - View Analytics
- Actors (stick figures)
    - Visitor
    - Contributor
    - Administrator
- Draw Lines from Actors to Titles
- <<external actor>>: Analytics System (<< >>)



User Stories

- Simpler and Shorter than Use Case
- Describes single small scenario from users perspective (where & why)
- One-two sentences on index cards (to force them to be short)
- Format
    - As a (type of user)
    - I want (goal)
    - So that (reason)
- Example 1
    - As a Bank Customer
    - I want to change my PIN online
    - So that I don’t have to go into a branch
- Example 2
    - As a User
    - I want to search by keyword
    - So that I can find and read relevant articles
- Can brainstorm many scenarios quickly
- Focused on intention
- Serve as placeholders for deeper conversation on a feature
- Both stories and cases serve different purposes
- Whatever we write stories/cases, they are input for the next step

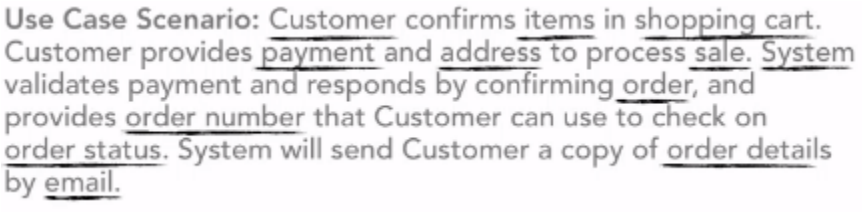
USER STORIES	USE CASES
short - one index card	long - a document
one goal, no details	multiple goals and details
informal	casual to (very) formal
"placeholder for conversation"	"record of conversation"

Chapter 4: Creating a Conceptual Model

- ID most important objects in the app
- Change from users/actors to wider scope
- Example: Product, shopping cart,
- Focus on the OO structure of the application
- Need requirements and user goals to consider structure
- A few hours per iteration
- Should be incomplete first time around

Identifying Objects

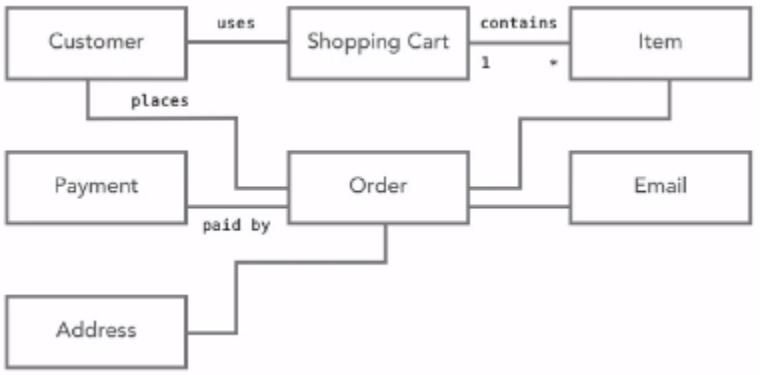
- Start picking out nouns
- Create noun list
- Consolidate list – things that belong to a more general class


Use Case Scenario: Customer confirms items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by email.

Customer	Order
Item	~~Order Number~~
Shopping Cart	~~Order Status~~
Payment	~~Order Details~~
Address	Email
~~Sale~~	~~System~~

- Box all the objects
- Conceptual Object Model
    - Just using names of objects
    - By Creating diagram easy to see responsibilities and relationships between objects

- Indicate main relationships/associations
- Add a short note for the association
    - Better to have specific terms
- Symbols to aid in visualization
    - One to many: 1-----*
-



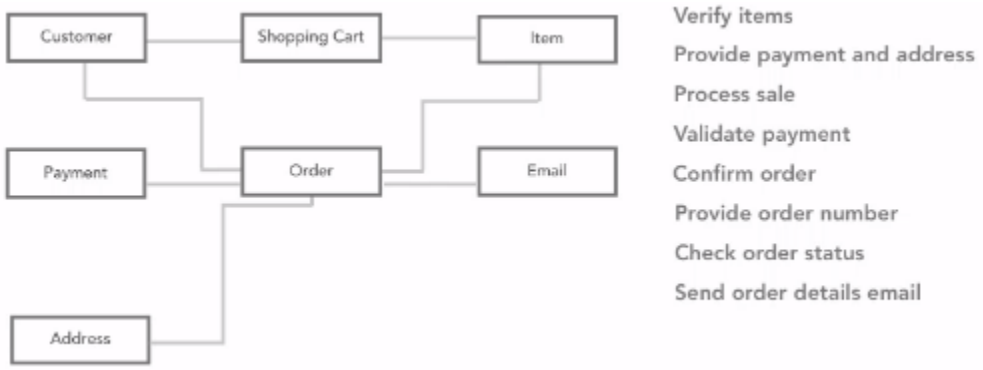
Identifying Responsibilities

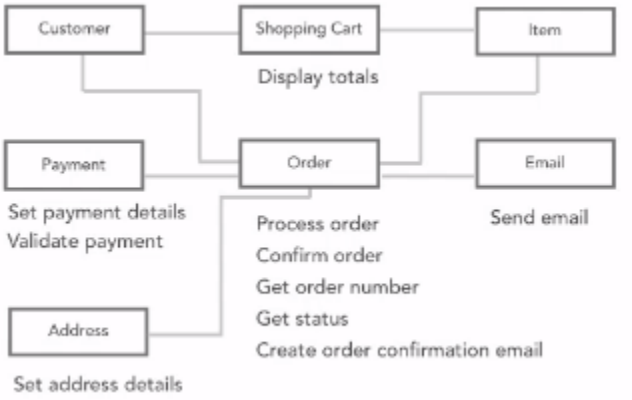
- Highlight Verbs
- Easy to find responsibilities
- Not obvious where the responsibilities belong
-

Use Case Scenario: Customer verifies items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by email.

Verify items	Confirm order
Provide payment and address	Provide order number
Process sale	Check order status
Validate payment	Send order details email

- Object should be responsible for itself
- Object has responsibility of an action, not what is requesting
- Customer is creating order, but is Order's responsibility
-

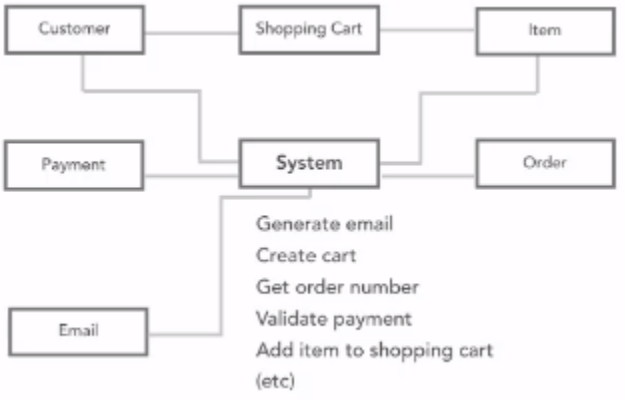




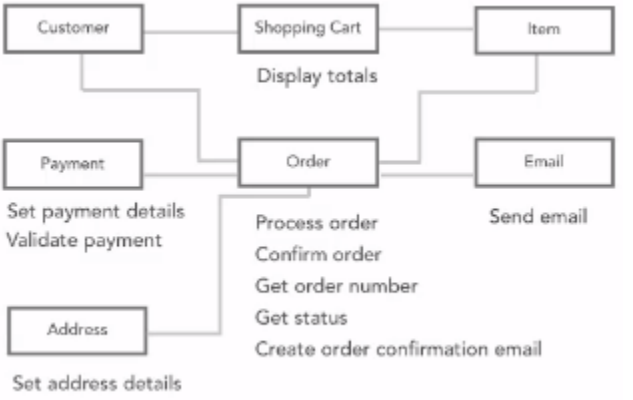
- Order has many responsibilities while Customer has none
- A common trap is to assign too much responsibility to an Object

Use Case Scenario: Customer verifies items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by email.

- Avoid Global Master Objects

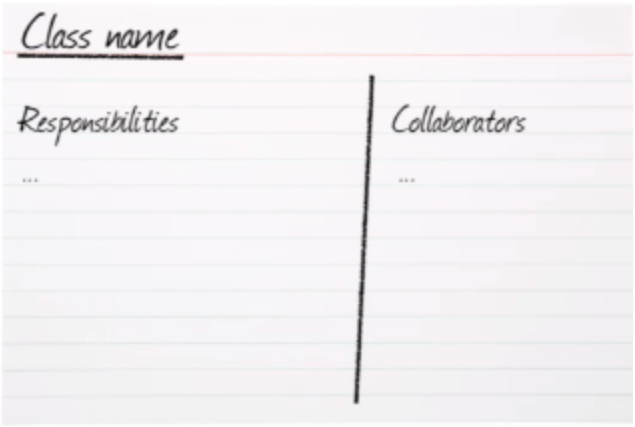


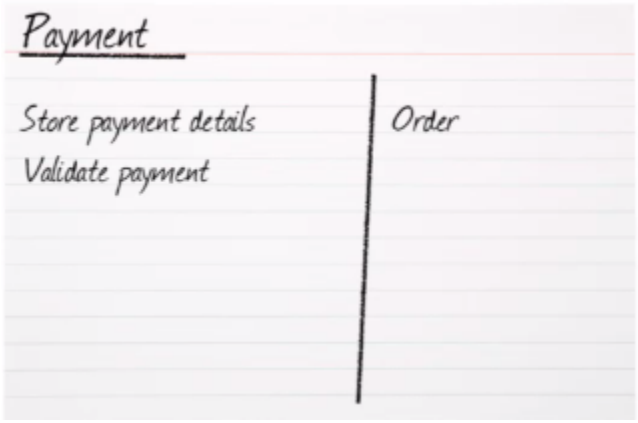
- Responsibilities should be well distributed



CRC – Another Format To Use

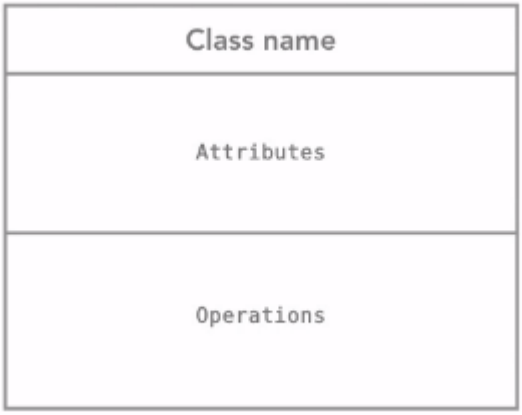
- Class
- Responsibility
- Collaboration
- Card (index card)





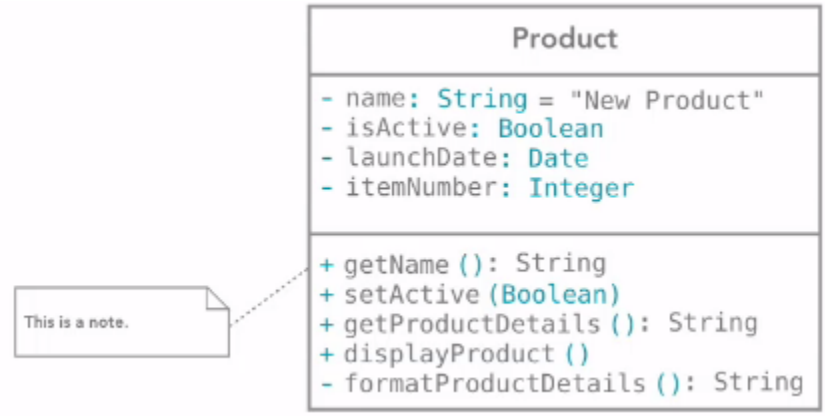
- Move related CRC Cards together
- Don’t go electronic yet
- Forced constraint only can have so many cards
- Too many cards can indicate another class is needed
- Fless out ideas next

Chapter 5: Creating Class Diagram



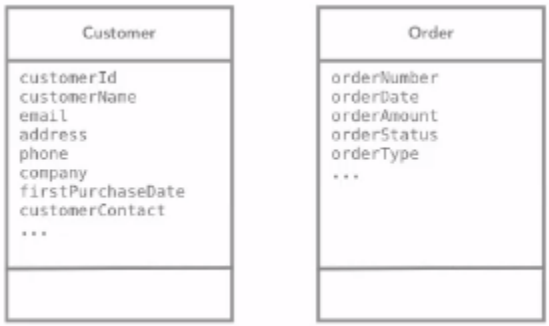
- Now closer to coding, official names encouraged
- Attributes (Pascal Case (thisVar”))
    - Suggested Data Type (: String)

    - Actual syntax not important, general idea ok
    - Default value ok too (= “new product”)
- Operations
    - get/set encouraged over change/read
    - Parameters ()
    - Return Type :()
- - denotes private (encapsulation)
- + denotes get/set
    - - name
        - Make private as much as possible
    - +getName
    - +setName
- Notes with line to item



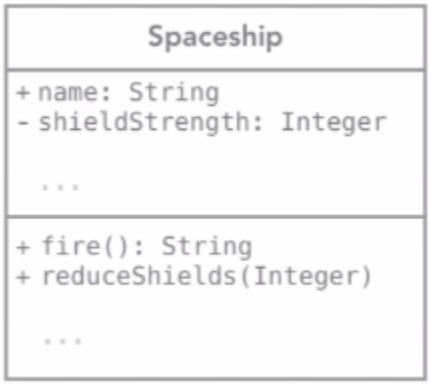
Avoid Building Plain Data Structures

- Often people focus on the data, creating data structures
- Focus on what they DO
- Class has no behavior – revisit responsibilities



Transforming Class Diagrams to Code

Class Diagram



Example: Java

```java
public class Spaceship {

   // instance variables
   public String name;
   private int shieldStrength;

   // methods
   public String fire() {
       return "Boom!";
   }

   public void reduceShields(int amount) {
       shieldStrength -= amount;
   }

}
```

Example: C#

```csharp
public class Spaceship {

  // instance variables
  public String name;
  private int shieldStrength;

  // methods
  public String fire() {
      return "Boom!";
  }

  public void reduceShields(int amount) {
      shieldStrength -= amount;
  }

}
```

Example: VB.NET

```vb
Public Class Spaceship

  ' instance variables
  Public Name As String
  Private ShieldStrength As Date

  ' methods
  Public Function Fire() As String
      Return "Boom!"
  End Function

  Public Function ReduceShields(Amount as Integer)
      ShieldStrength -= Amount
  End Function

End Class
```

Example: Ruby

```ruby
class Spaceship

  # instance variables
  @name
  @shield_strength

  # methods
  def fire
      return "Boom!"
  end

  def reduce_shields(amount)
      shield_strength -= amount
  end

end
```

Example: Objective-C

- Has two files that separate interface and implementation

```
@interface Spaceship : NSObject {
    @public
    NSString *name;
    @private
    int shieldStrength;
}

// method declarations
-(NSString *) fire;
-(void) reduceShields:(int)amount;

@end
```

interface

```
@implementation Spaceship

-(NSString *) fire {
    return @"Boom!";
}

-(void) reduceShields: (int)amount {
    shieldStrength -= amount;
}
@end
```

implementation

Exploring Object Lifetime

- How are objects created
- What happens when they are created
- What happens when we are finished with them

Instantiation

- Create a new object
- “new” keyword
- Do you take part in the instantiation?

Java
```
Customer fred = new Customer();
```

C#
```
Customer fred = new Customer();
```

VB.NET
```
Dim fred As New Customer
```

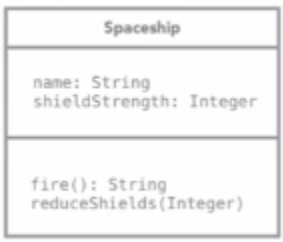
Ruby
```
fred = Customer.new
```

C++
```
Customer *fred = new Customer();
```

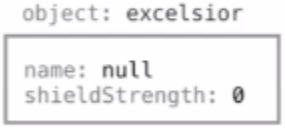
Objective-C
```
Customer *fred = [[Customer alloc] init];
```

Constructor

- Special method that is called when object is created




```
Spaceship excelsior = new Spaceship();
```



Constructor in UML

Example: Java/C#/C++

- Created Method in the Class with the same name as the Class

```
public class Spaceship {

  // instance variables
  public String name;
  private int shieldStrength;

  // constructor method
  public Spaceship() {
      name = "Unnamed ship";
      shieldStrength = 100;
  }




  // other methods omitted

}
```

```
Spaceship excelsior = new Spaceship();
```

object: excelsior

```
name: Unnamed ship
shieldStrength: 100
```

Syntax for several languages

Java/C#/C++

- className()

VB.NET/Python

- new()

Ruby

- initialize()

OC

- init()

Overloaded Constructors

- multiple constructors with same name
- each takes different number of parameters

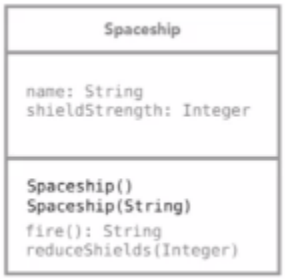
```
public class Spaceship {

  // instance variables
  public String name;
  private int shieldStrength;

  // constructor method
  public Spaceship() {
      name = "Unnamed ship";
      shieldStrength = 100;
  }
  // overloaded constructor
  public Spaceship(String n) {
      name = n;
      shieldStrength = 200;
  }
  // other methods omitted

}
```

```
Spaceship excelsior =
    new Spaceship("Excelsior 2");
```

object: excelsior

```
name: Excelsior 2
shieldStrength: 200
```

Overloaded Constructor in UML



Destructors / Finalizers

- Called when an object is destroyed/deallocated/released
- Use for releasing resources
    - FileIO/ Document Open

Static/Shared Members

- Method shared cross all members of a class
- Many accounts share one interest rate
    - 1000s of accounts/variables
- Don't create a global variable for everything
- Class Level variable vs. Instance Level variable
- Value can change, but there's only one of them

Creating Static Variables

```
public class SavingsAccount {

  // instance variables
  public String accountNumber;
  private Money balance;
  // static variables
  public static float interestRate;

  // other code omitted

}
```

```
' VB.NET - shared variables
Public Shared interestRate As Float

# Ruby - class level variables
@@interestRate
```

Accessing Static Variable

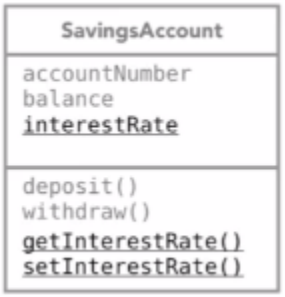
- Use the name of the class to access
- savingsAccount.interestRate =- 0.5;

Creating Static Methods

- Static Methods CANNOT work with Instance Variables
- Static Methods CAN work with Static Variables

```
SavingsAccount.setInterestRate(0.9);
public class SavingsAccount {

  // instance variables omitted

  // changed to private
  private static float interestRate;

  // public static methods
  public static setInterestRate(float r) {
      // add code to log any change
      interestRate = r;
  }
  public static getInterestRate() {
      return interestRate;
  }

  // other code omitted
}
```

Showing Static Members in UML



Chapter 6: Inheritance

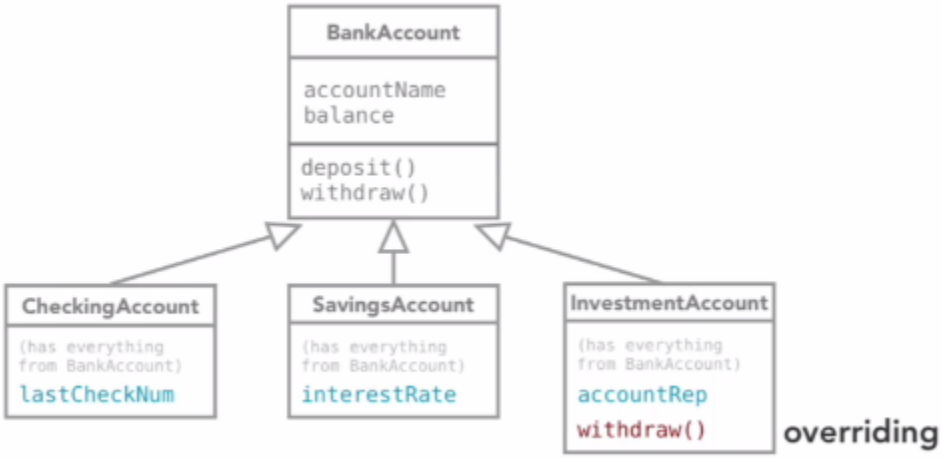
Inheritance

- Describes “is a” relationship
- Example 1
    - A Car is a Vehicle
    - Bus is a Vehicle
    - Car is a Bus
- Example 2
    - An Employee is a person
    - A customer is a person
- Example 3: multiple inheritance
    - A Corvette is a car is a vehicle
- Example 4: No relationship
    - Bank is not a Bank Account
    - Bank Account is not a Bank
    - A Checking Account is a Bank Account
    - A Savings Account is a Bank Account

UML

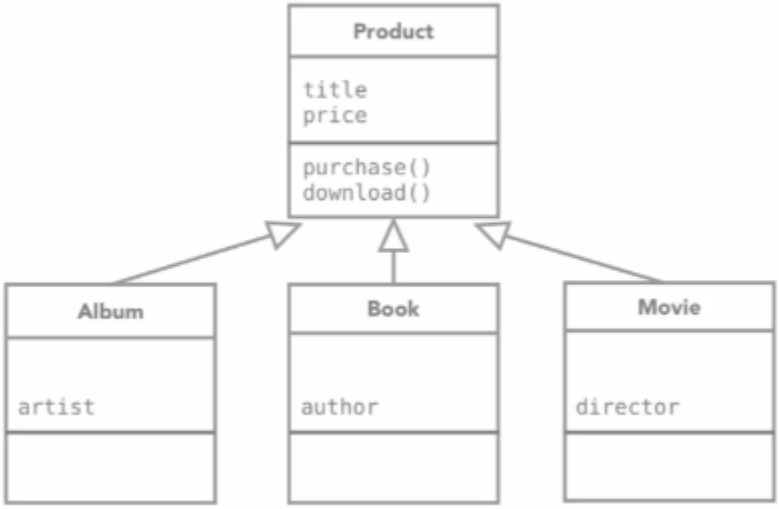
- Open Arrow
- Parent/Super Class
- Child Sub Class
    - Children can ‘override’ Parent Class
    - To REPLACE or ADD Parent Class implementation

- Don’t over-inherit or over-think



- Sub Classes will present themselves
    - Album
    - Book
    - Movie

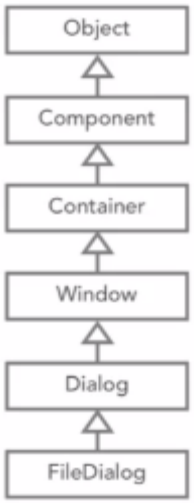
- THEN create Super Class and strip out similar behaviors and put into Super Class



- Should be obvious
- Can be identified from top-down or down-up
- Not all classes will have inheritance

Example: Java

- Object is Super Class of everything else

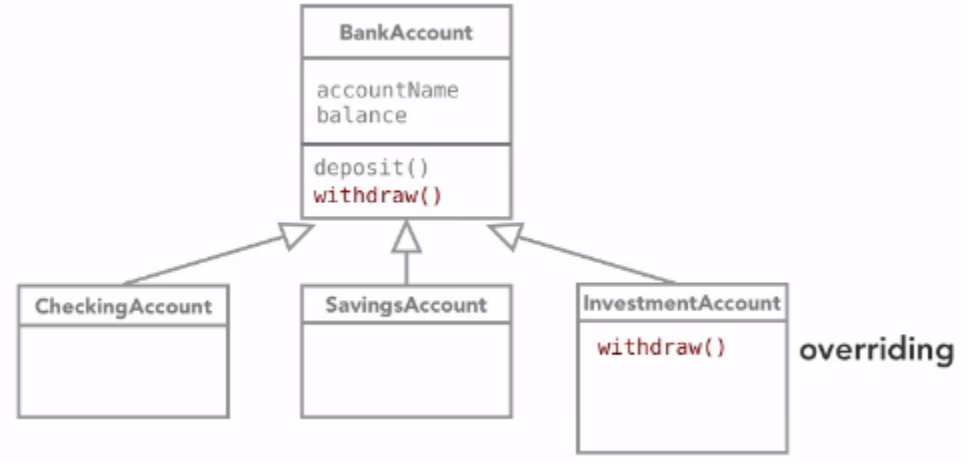


- Be weary of more than 2-3 levels of inheritance

Inheritance in Varying Languages

```
Java        public class Album extends Product {  ...
C#          public class Album : Product {  ...
VB.NET      Public Class Album
               Inherits Product ...
Ruby        class Album < Product   ...
C++         class Album : public Product {  ...
Objective-C @interface Album : Product {  ...
```

Overriding in Varying Languages – See Language Ref

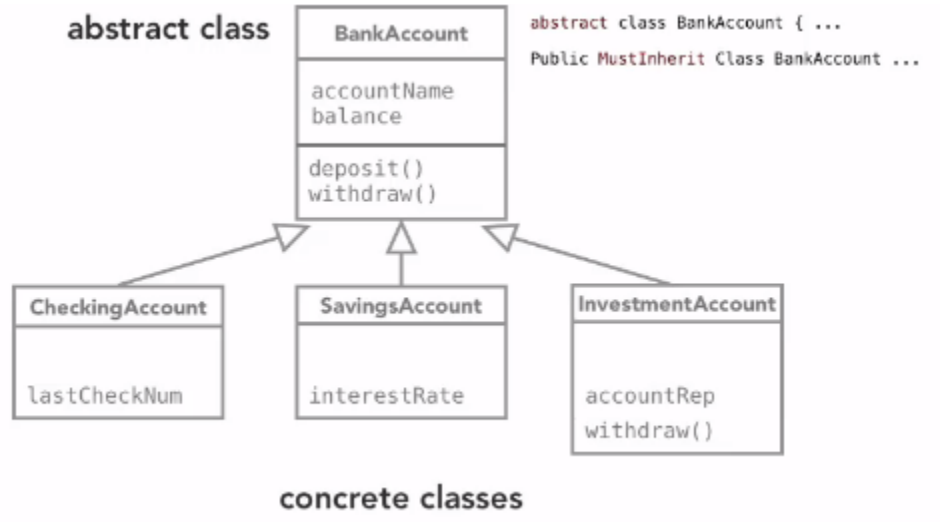


Calling a Method in the Super/Parent/Base Class in Varying Languages

Java	`super.doSomething();`
C#	`base.doSomething();`
VB.NET	`MyBase.doSomething()`
Ruby	`super do_something`
Objective-C	`[super someMethod];`
C++	`NamedBaseClass::doSomething();`

Composition

- Class that is never instantiated
- Abstract Class
    - Never instantiated
    - Used as blueprint only
    - Used for inheritance only
- Concrete Class
    - Instantiated



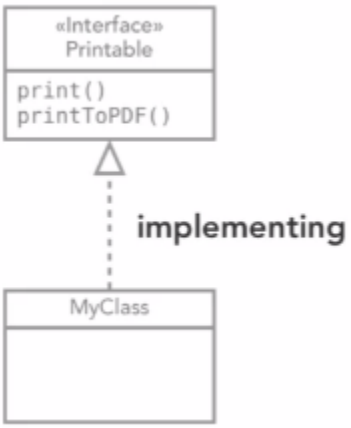
User Interfaces

- Defining Interface Contracts
    - Has no functionality and only defines structure
    - Create a new class and choose to use an interface
    - More classes that use the contract, the better
    - Easier to manage/maintain
    - Don’t need to know how the Object works, just adhere to the Interface
- Using Interfaces
    - Interface Title as <<Interface>>
    - Dotted Line to represent Interface- - - ->
    - “Implement an Interface”
    - “Program to an interface, not an implementation”
    - Then developer can choose how to implement the methods, not restricted
    - Interfaces preferred over inheritance, cleaner model
    - Objective-C uses “Protocol”: list of method signatures
        - “Conform to a protocol”

```
class MyClass implements Printable {


    // method bodies
    public void print() {
        // provide implementation
    }

    public void printToPDF(String filename) {
        // provide implementation
    }

   // additional functionality...

}
```

```
// sometime later
while (genericObject in listOfObjects)  {

     if ( genericObject instanceOf Printable ) {
          // if it implements the interface, we can use it
          genericObject.print();
     }

}
```

UML

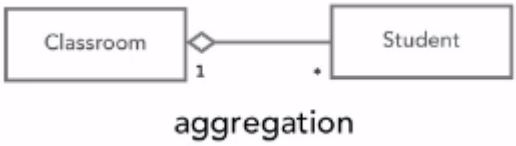


Aggregation

- Associations
- Lines between objects suggesting some sort of interaction
- Inheritance with Arrow
- Describe an obvious relationship between objects
- Object can be built off other objects
- “HAS A” vs. “IS A”
    - Customer has a address
    - Car has a engine
    - Bank has many bank accounts

UML

- Unfilled diamond <>

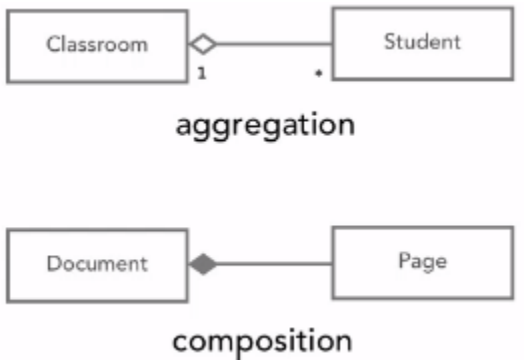


- A Classroom HAS MANY Students

Composition

- Composition implies Ownership
- Not just “HAS A”
- When Object is destroyed, associated Objects are destroyed also
- Might need to write a deconstructor to remove
- Difference
    - Removing Classroom, Students should still persist
    - Removing Document, removes Page

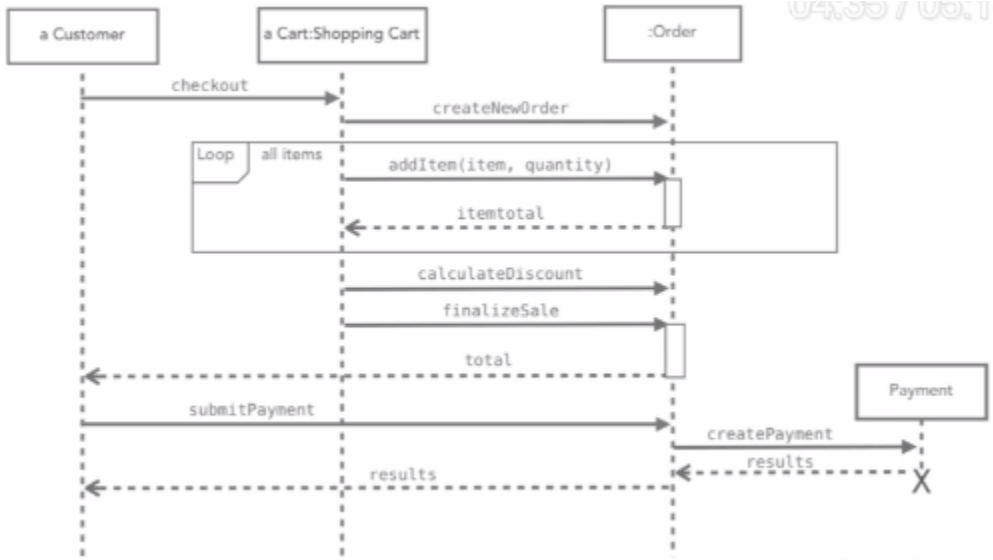
UML



Chapter 7: Sequence Diagrams

Sequence Diagrams

- Does not describe entire system
- Only one interaction
- Start with object boxes for participants in interaction
- If using class name, use colon
    - A Customer
    - a Cart: Shopping Cart
    - :Order
- Draw Lines for interactions
    - Method names with (parameters)
    - createNewOrder
    - AddItem(item, quantity)
    - Itemtotal
- Surround loops with a Frame
- Don’t get too detailed, just basic interactions
- For short-lived interactions, use X for the line
- Send ----------→ (solid line)
- Return - - - - - - → (dotted line)
- Should be able to work with BA (non-developer)
- No need to diagram EVERY part of system
- Only for situations that are not vary clear
- Could identify need for new Class



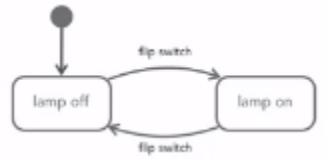
UML Diagrams

- 14 types

- Class Diagram
- Use Case Diagram
- Object Diagram
- Sequence Diagram
- State Machine Diagram
- Activity Diagram
- Deployment Diagram
- Package Diagram
- Component Diagram
- Profile Diagram
- Communication Diagram
- Timing Diagram
- Composite Structure Diagram
- Interaction Overview Diagram

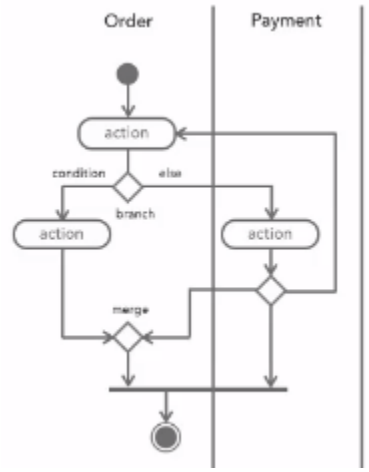
State Chart

- Focused on a single object
- How it changes state over time
- Rectangle with rounded corners



Activity Diagram

- Detail steps, transitions, and decisions that occur as flow through part of the system
- Partitions/swim lanes
- Represent different classes/business units that handle activity



UML Tools

Wikipedia.org has a list of UML tools

Diagramming Tools
Visio, OmniGraffle

Web-based Diagramming
gliffy.com, creately.com, lucidchart.com

Programming Tools: IDE-based
Visual Studio, Eclipse with UMLTools

Commercial Products
Altova UModel, Sparx Enterprise Architect, Visual Paradigm

Open-Source
ArgoUML, Dia

Chapter 8

Design Patterns

- Gang of Four / "GoF"

Creational Patterns	Structural Patterns	Behavioral Patterns
Abstract Factory	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template method
		Visitor

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype

- Singleton

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Singleton Design Pattern

- Want only ONE of a class
- NOT STATIC, since that is 0
- SPECIFICALLY ONE

Example: Java

- Create a static variable that acts as a placeholder for a Singleton (__me)
- Create new method
    - Lazy Singleton, create upon FIRST request
    - Call static method “getInstance()”
    - Do I exist? If no, return the object
    - If so, do not create one

```
public class MySingleton {
    // placeholder for current singleton object
    private static MySingleton __me = null;
    // private constructor - now no other object can instantiate
    private MySingleton()  { }
    // this is how you ask for the singleton
    public static MySingleton getInstance() {
        // do I exist?
        if ( __me == null ) {
                // if not, instantiate and store
                __me = new MySingleton();
        }
        return MySingleton;
    }

   // additional functionality
   public someMethod()  { //... }
}
```

- Calling

```
// ask for the singleton
MySingleton single = MySingleton.getInstance();

// use it
single.someMethod();


// or even just call directly
MySingleton.getInstance().someMethod();
```

Memento Design Pattern

- Managing change in a way that doesn't violate encapsulation
- Used for "UNDO" type functions
- Arose naturally in the programming world and will be encountered often

Originator

- original_state

Caretaker

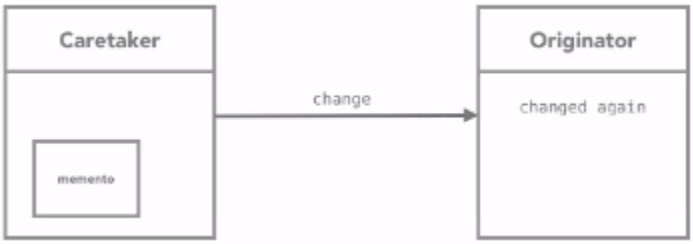
- When and why Originator needs to save or revert its state
- Request originator to save itself

Pattern

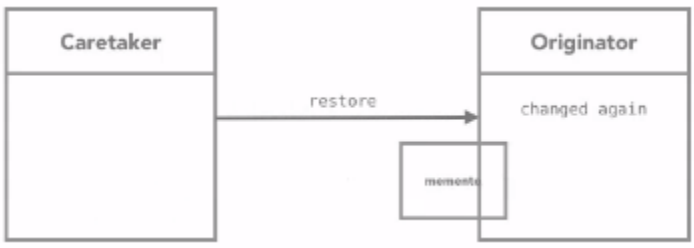
- Originator creates memento

- Sends memento to Caretaker for storage

- Request can come from anywhere, not just Caretaker, though caretaker always does the storing



- Upon "REVERT" request, the Caretaker sends memento back to the originator



- Originator back to original state

Chapter 9: Object Oriented Design Patterns

- Language will not prevent poor design
- Apply good OO practices
- Not design patterns, but are general rules to follow
- Did I design this well?

General Software Development Principles

- DRY: Don't Repeat Yourself (duplicated code)
    - Don't block copy code over and over, create a function/method
    - One place in the system that takes care of a problem
- YAGNI: You Ain't Gonna Need it (unnecessary code)
    - Solve today's problem
    - Don't code for tomorrow

Code Smells

- Long method (200-300 lines)
- Very short (or long) identifiers
- Pointless Comments

```
//this creates i and sets it to zero
int i = 0;
```

- God Objects
    - Objects that do everything
- Feature Envy
    - Class does very little, but use methods of another class

SOLID

- Checklist of things to consider and look out for

SOLID

- S: Single Responsibility Principle
- O: Open/Closed Principle
- L: Liskov Substitution Principle

- I: Interface Segregation Principle
- D: Dependency Inversion Principle

S: Single Responsibility Principle

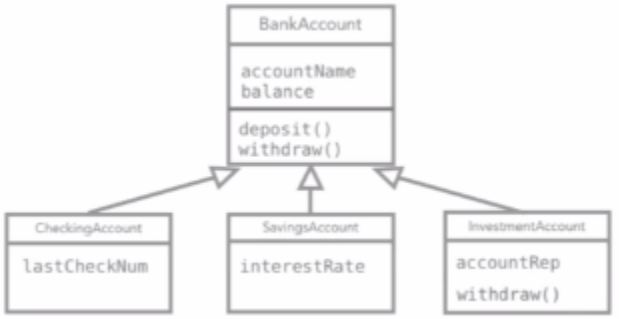
- An object should have one reason to exist, one reason to change – one primary responsibility

O: Open/Closed Principle

- Open for extension, but closed for modification
- Once written and working, don't change old code, create new code
- Inheritance
    - Class created, then new feature arises
    - Create new method/subclass

L: Liskov Substitution Principle

- Derived classes must be substitutable for their base classes without altering correctness of the program
- Extension of inheritance
- Derived classes should be able to pass subclasses around in lieu of base class
- Never should be able to use all BUT ONE subclass



I: Interface Segregation Principle

- Multiple specific interfaces are better than one general purpose interface
- Interfaces should be as small as possible
- If too large, split into smaller Interfaces
- No class should be forced to support huge interfaces it doesn't need

D: Dependency Inversion Principle

- Depend on abstractions not on concretions
- Don't tie concrete objects together, but deal with abstractions to reduce dependencies

- Disconnect two very concrete classes
- Insert new layer, abstract class
- Store class isn't dependent on the two AudioFile classes
- Now any low-level class that inherits from Reader/Writer can be used
    - MovieFile Reader/Writer
    - GameFile Reader/Writer
- Store object doesn't need to be altered for updates
- Flexibility

GRASP

GRASP

- GRASP: General Responsibility Assignment Software Patterns
- Focus on Responsibility
- Who creates this object
- Who takes care of receiving information from user interface
- Compatible with SOLID

9 Principles

- Creator
- Controller
- Pure Fabrication
- Information Expert
- High Cohesion
- Indirection
- Low Coupling
- Polymorphism
- Protected Variations

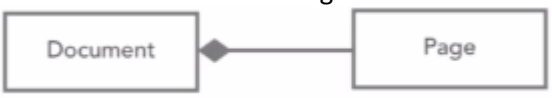
Expert/Information Expert

- Assign the responsibility to the class that has the information needed to fulfill it
- Object should take care of itself
- Example
    - Customer, Shopping Cart, Item
    - “Current Total” belongs to the Shopping Cart as it knows most about the total

Creator

- Who is responsible for creating an object
- How the objects are created
- Easy to tell interactions

- Not easy to tell how they come into being
- Sequence diagram can help
- Q's
    - Does one object contain another? (Composition)
    - One object closely use another
    - Know enough to make another object?



Low Coupling / High Cohesion

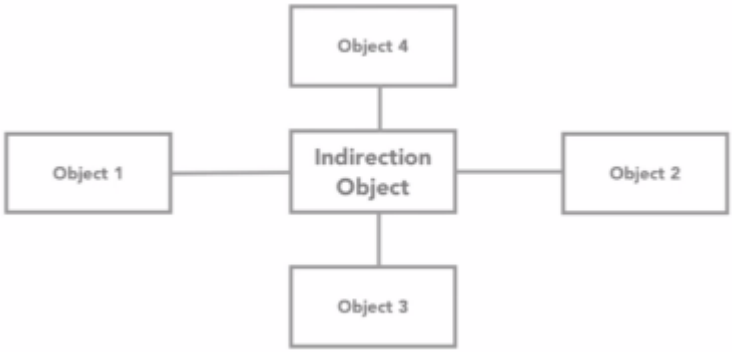
- Coupling: the level of dependencies between objects
- If one objects connects tightly to other objects
- Things can break easily
- Reduce dependencies to a minimum
- Cohesion: the level that a class contains focused, related behaviors
- AIM is LOW COUPLING, HIGH COHESION
- Don't connect UI elements directly to business objects
- Concept: Model U Controller
    - Well described idea within the app
- Create controller class for UI and Business Object

Pure Fabrication

- When the behavior does not belong anywhere else, create a new class
- Rather than force into an existing class (decreasing cohesion), create new class

Indirection

- Decrease coupling between objects
- Ro reduce coupling, introduce an intermediate object



Polymorphism

- Automatically correct behavior based on type
- As opposed to: conditional logic that checks for particular types

Protected Variations

- Protect the system from changes and variations
- Identify the most likely points of change
- Use multiple techniques: encapsulation, LSP, OCP
- Interfaces, Substitution, Overriding Classes, Open/Closed Principle

Chapter 10

Feature Support Across Languages

Language	Inheritance	Typing	Call to super	Private Methods	Abstract Classes	Interfaces
Java	Single	static	super	Yes	Yes	Yes
C#	Single	static	base	Yes	Yes	Yes
VB.NET	Single	static	MyBase	Yes	Yes	Yes
Objective-C	Single	static/ dynamic	super	No	No	Protocols
C++	Multiple	static	name of class::	Yes	Yes	Abstract Class
Ruby	Mix-ins	dynamic	super	Yes	n/a	n/a
JavaScript	Prototype	dynamic	n/a	Yes	n/a	n/a

- Inheritance
- Typing
    - Most are static typed
    - Dynamic languages have flexibility without the static typing
- Call to super
- Private Methods
    - Protecting classes
- Abstract Classes
- Interfaces

Resources

- Initial stage of determining requirements
    - “Software Requirements” by Carl Regas
- Use Cases
    - “Writing Effect Use Cases” by Alistair Cockbens
- “User Stories Applied for Agile Software Development” by Mike Koehn
- UML
    - “UML Distilled” & “Refactoring” by Martin Fallor
- Design Patterns
    - “Gang of Four” (C++)
    - “Head First Design Patterns” (Java)