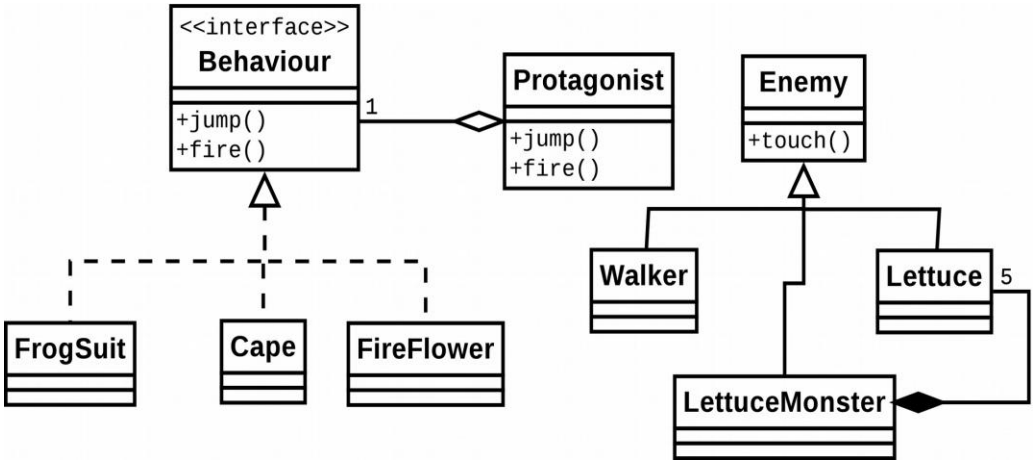
Phillip Hoang 1409768

■-Listing potential classes, actions/methods/relationships (verbs). And **DRAW** the **UML class diagram** based on them.
-Converting **Java code to UML class diagrams**

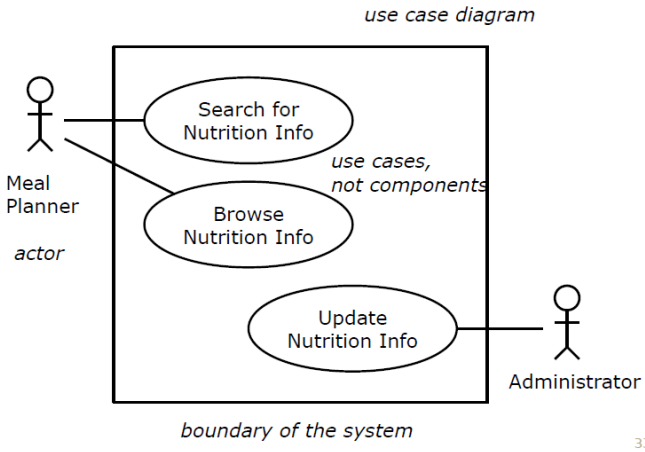


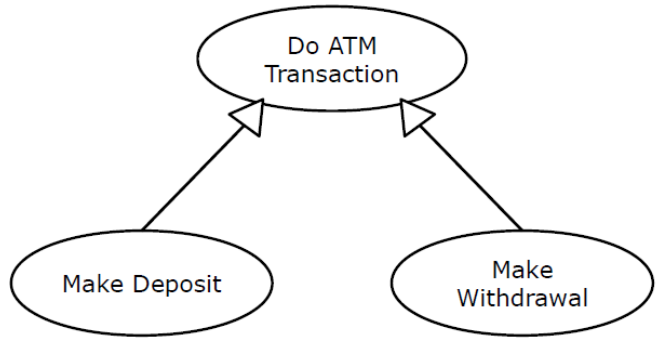
+ is public, - is private, triangle is ISA, dashed lines are for implementation of interface, diamonds for HASA, black diamonds for STRONG HASA

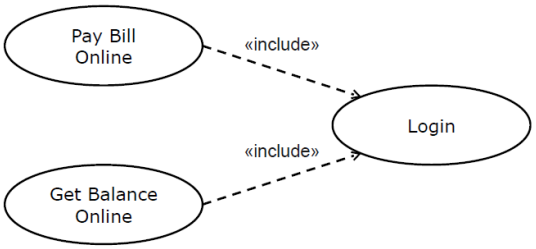
0..* is 0 or more

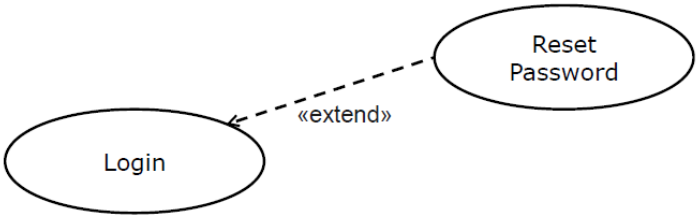
LABEL relationships

■-**Deriving Use Cases** from goals/description, make **UML Use Case diagram** from this including boundary, actors, use case bubbles, and relationships between actor and use case







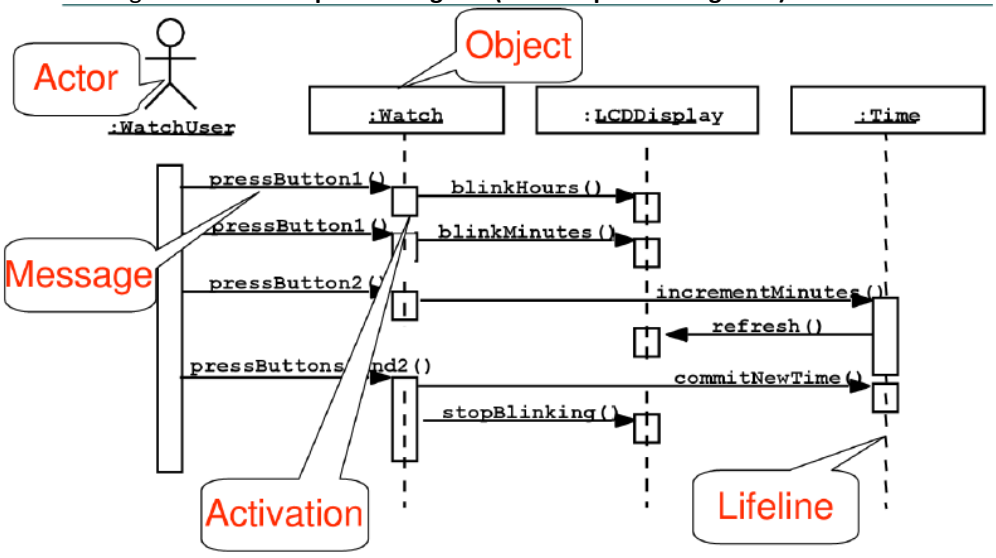


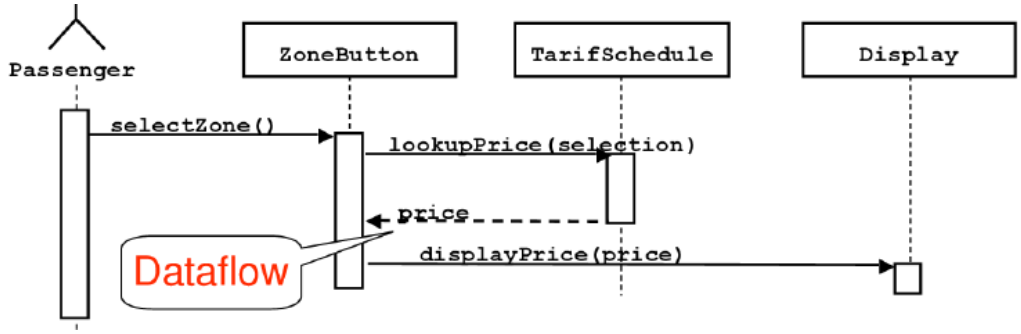
Use Case Name	**SearchForNutritionInfo**
Participating Actors	Meal Planner (primary)
Goal	Meal Planner finds nutrition information
Trigger	Meal Planner chooses the Search option
Precondition	Meal Planner knows food name and amount.
Postcondition	On success, nutrition information displayed.
	...

Basic Flow 1	System prompts Meal Planner to enter keywords.
2	Meal Planner submits keywords.
3	System lists matching foods, prompting for a selection.
4	Meal Planner browses and selects a food.
5	System prompts for food weight in units of 100 g.
6	Meal Planner enters food units.
7	System presents nutrition data for the amount of food.
	... *avoid implementation specifics*

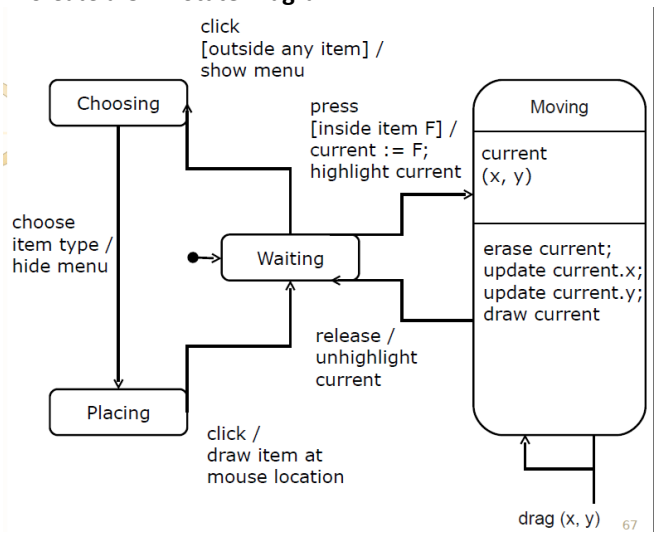
Exceptions 3	If there are no matching foods
3.1	System displays an error
3.2	System returns to step 1
7	If given food units is non-numeric, use 0 and proceed

■-Converting Use Case to **Sequence Diagram (UML Sequence Diagram?)**





■-Create a **UML State Diagram**



■-Git

*-How to track staged delivery process where clients might be using older (stabler) versions: *Check which version (commit) the client is using*

-In **daily scrum meeting** why would you use **git log**? *To show commit history to ...*
-Using git repos, how would you enable and track an iterative software dev process? *Check branches and master merges*

■-Human Error and Usability

-Name of the law that describes the **speed of choosing from a list of choices:** *Hick's law: T=a+b*log(n)*
-Why is the **difference in time of choosing 2 and 8** choices greater than the difference between 80 and 100

-Name of the law that defines **speed of clicking on a target:** *Fitt's Law*
-Why does it take longer to click on the other targets?**:** *Affected by: time to move to object, distance from starting point to object, and width of the object*

-Use **Fitt's law** and **Hick's law** to explain which array of buttons would be faster (ex list vs pie)

-What is **Saccadic Masking** and how does it affect software, affect use scrolling through a large and long webpage, how to design against it: *is a visual phenomenon where the brain blocks visual processing during eye movements in such a way that neither the motion of the eye nor gap in visual perception is noticeable to the viewer. It can cause users to miss important information that they did not acknowledge come on the screen. It can be countered by repeatedly alerting the user when there is important information and not stopping until the user acknowledges the information.*

-Why is there a difference in time between choosing 1 item from 80 unordered, or 1 item from 80 ordered. **How much** is the approximate difference in time? *Random, must read every entry consuming linear time, with ordered, can use subdivision so will take logarithmic time*

-[toggle between edit and delete] what kind of common error will the UI cause the user (name of error)? How to fix it
-What is a mode error? How does one prevent mode errors? *Use modeless interface which makes mode error impossible*

■-User Interfaces

-What is one UI method that aids usability but also reduces human error? *Using elemental widgets such as dropdown lists and radio buttons instead of textfields*

-Why must we be careful about **colours** we use in UI? *Colour blindness*

-Give 2 examples (or instances) of interface metaphors. *desktop, menus, rooms, shopping carts*

-Name of the law that estimates average time to make a simple decision, *n* choices vs *t* time: *Hick's law above*

-GOOD UI Design: *alignment, balance, symmetry, use labels, separators, proximity*

■-Software Process

-Explain what a software development process is: *life cycle or structure on the dev of software*

-Relationship between **iterative model** and **waterfall model**. What is their primary difference? Why would you choose waterfall over iterative? Give 1 example where you would use waterfall over iterative, and 1 example of iterative over waterfall: *iterative when you need to update, waterfall for example an embedded system*

-Give 1 reason why the **Unified Process** is similar to the **waterfall process,** 1 reason why different: *both have principle-bound stage such as requirements and implementation, unified iterative, waterfall goes sequentially*

-In what parts of waterfall model would you use refactoring? *Integration and Testing*

-In SCRUM, what is a daily standup meeting and what are the questions asked? *What did you do, what are you going to do, what is blocking you*

-How does version control like git relate to the notion of **courage** in agile software development? *Branching allows programmers to try new things with no fear of messing up the current product.*

-Test driven = test first
-How is **Test Driven Development** employed in the design of APIs?
-How does **test first development** work? How does it affect software design?
    -Write tests → run tests → write method interface and stub → run tests → write method body → run tests → adjust method and retest until tests pass

-How do agile processes provide feedback to stakeholders? *Stakeholders can track progress daily, or per iteration*

-How would one use continuous integration in staged delivery process? *No fucking idea*

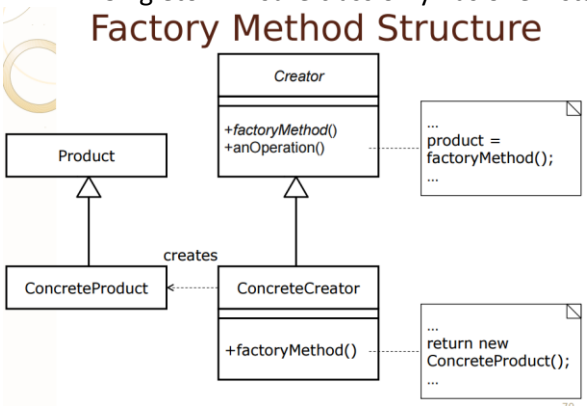
-List 2 tools that promote courage (agile). Explain why: *Version control because it allows trying new things without messing up the current software. Teamwork?*

■-Identify Design Patterns based on picture (draw design the design patterns)

■-Identify Design Patterns appropriate for (and explain why):

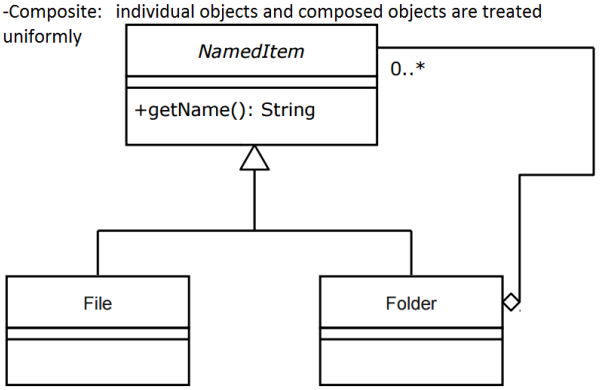
■-Creational Patterns (creating objects): abstract factory, builder, prototype

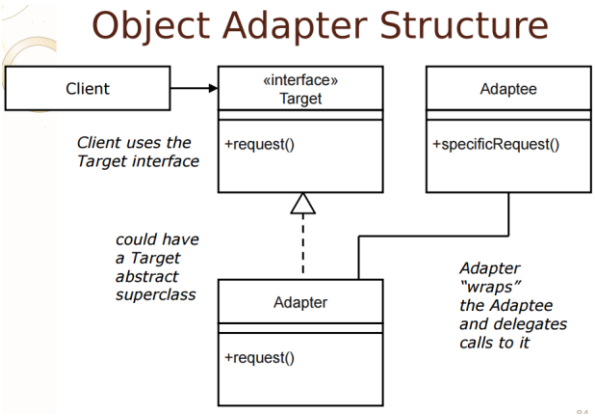
-Singleton: Ensure class only has one instance, provide global point of access to it.



■-Structural patterns (connecting objects): adapter, bridge, flyweight

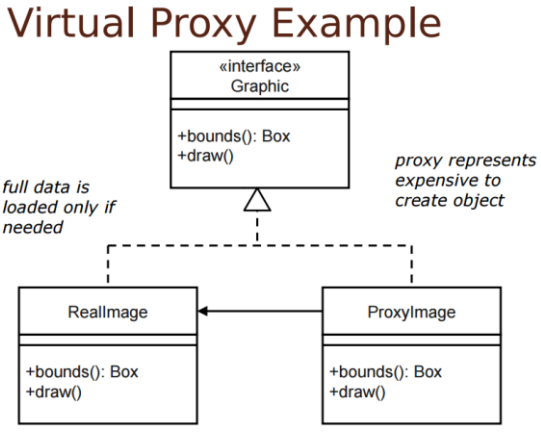
-Composite: individual objects and composed objects are treated uniformly

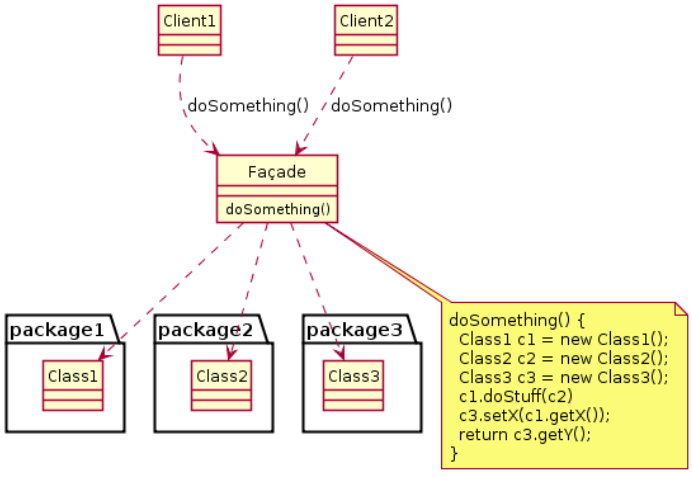




-Adapter/wrapper: convert the interface of a class into another interface that clients expect, lets classes work together that couldn't otherwise because of incompatible interfaces

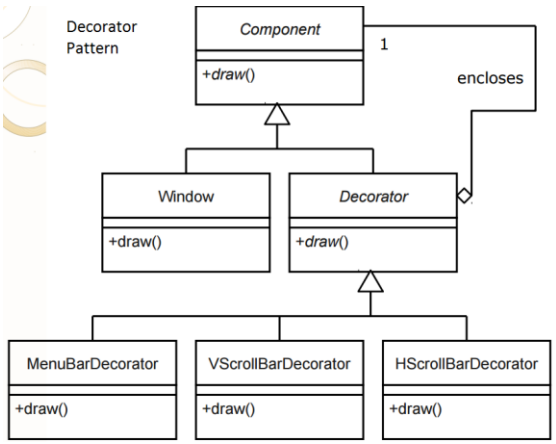
-Proxy: defer the full cost of creation and initialization of an object until we actually need it ex big jpg and thumbnail





Facade

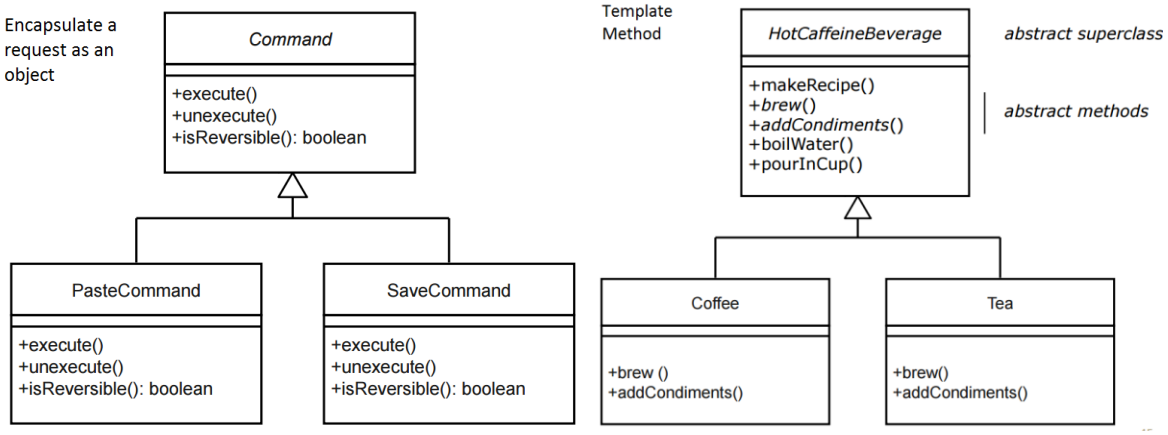
The facade class abstracts Packages 1, 2, and 3 from the rest of the application.



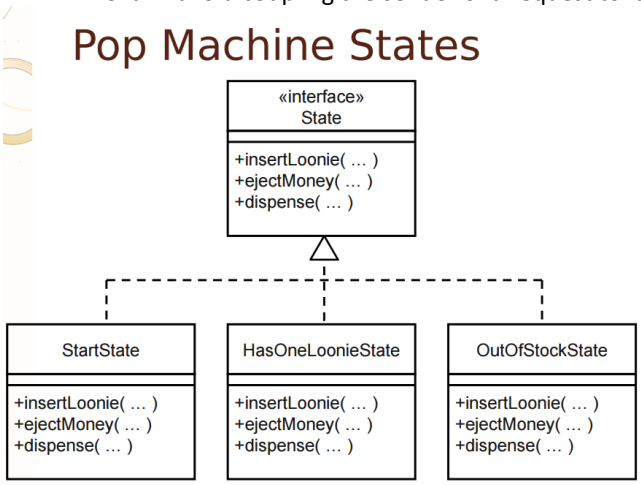

```
class MenuBarDecorator extends UIDecorator {
    MenuBarDecorator(UIComponent c) {
        component = c; }
    public void draw() {
        component.draw();
        /* draw menu bar here */
        System.err.println("Draw MenuBar");
    }
}
```

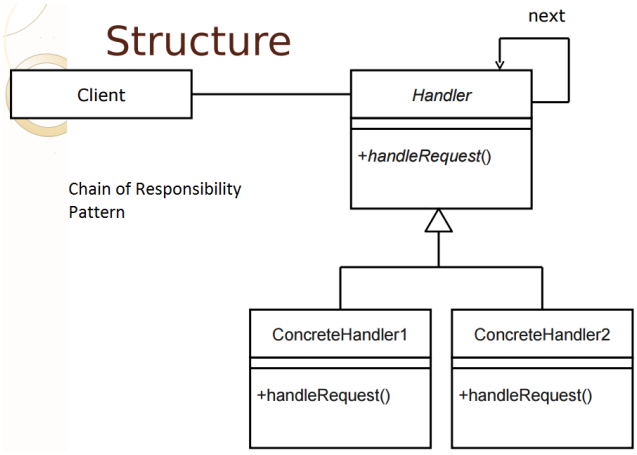
-Decorator: attach additional responsibilities to an object dynamically

■-Behavioral patterns (distributing duties): interpreter, iterator, mediator, memento, strategy



-Command: class may want to issue request without knowing anything about the operation or receiver obj
-Template to remove duplicate code, keeps algorithms in one place so easier to change
-State Pattern: allows an object to alter its behavior when its internal state changes (simplify ops with long conditionals)
-Chain: avoid coupling the sender of a request to its receiver by giving more than 1 object chance to handle the request





-Observer Pattern (picture in MVC and Observer Pattern): Data object that notifies all view objects when its data changes. Views can be added or removed dynamically, data does not need to know details of each type of view

■-OO Principles

-Explain how the **hide delegate** refactoring applied to the **message chains** bad smell increases or decreases **coupling:** *decreases coupling as the client does not need to know about the relationships between objects which makes making changes and testing easier (p22 of Lec 08 Refactoring)*

*-Explain how coding to the **specification** rather than the **implementation** increases or decreases coupling: *Increases coupling as coding for specification to the implementation allows for refactoring to make less coupled code.*

-Explain how Java's **dot operator** and **switch statement** increase or decrease coupling: *Increases coupling as the dot operator requires an object. Switch statements are highly coupled because one change in the class requires the switch statement to be changed as well*

-Explain why using ArrayList as a Stack violates the **Liskov substitution princple** eg class Stack extends ArrayList<Object>: *Arraylist can be substituted anywhere a stack is used, but stack cannot be substituted for arraylists*

-Explain how to **replace conditional with polymorphism** refactoring applied to the **switch statement** bad smell increases or decreases coupling: *Decreases coupling as the new code will not need to be edited when a new class is called because it will be polymorphed.*

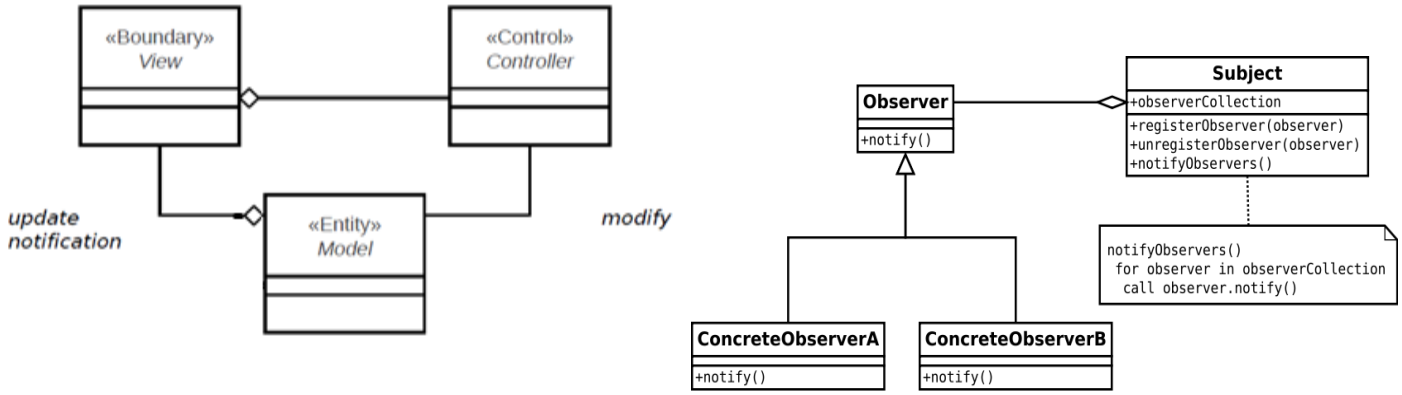
-How are cohesion and coupling affected by: high dependence on abstractions, low dependence on concrete classes: *Coupling decreases, as abstractions allow for more types of concrete classes to be able to be used*

-How are cohesion and coupling affected by: favor composing objects (delegation) over implementation inheritance: *Coupling increases as the calling class must be changed when there is a change to/addition of a concrete class.*

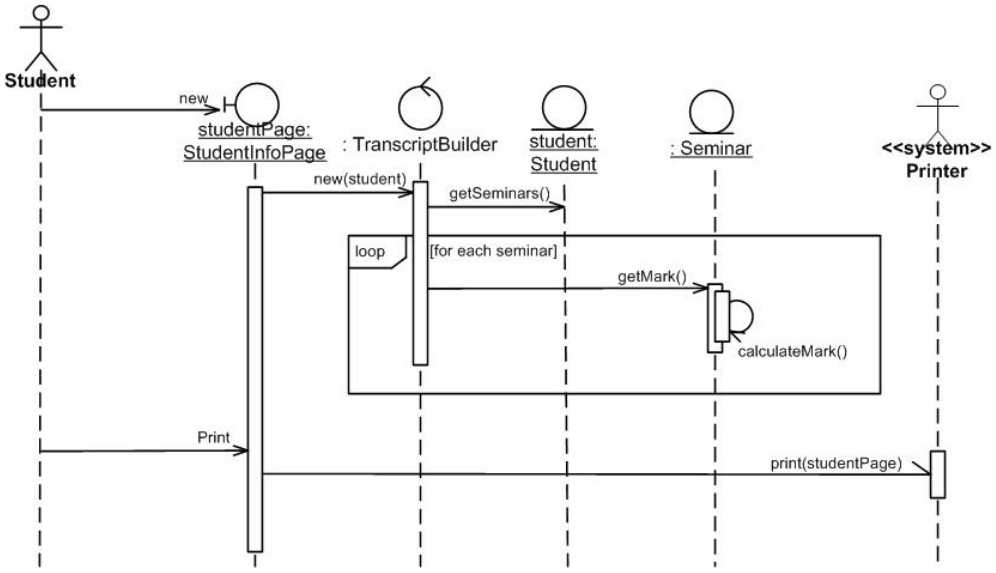
■-MVC and Observer Pattern

*-How does observer pattern **decouple** a model from views? *The model does not need to know about the views*

-Draw **UML Class Diagram of MVC, of observer pattern**



■-Draw **UML Sequence diagram** for update of model in observer pattern



--Model: entity layer, complete, self-contained representation of data (back end)
--View: boundary layer, what the user sees (front end)
--Controller: control layer, handles events and uses elements from the UI to modify the model

-Refactoring (and Decorator) [and Template Method] {and Factory Method}

-Find at least 3 **bad smells**, and at least 2 refactoring that could be applied to this code snippet. Then **DRAW** the UML class diagram of the code after refactoring

-Duplicate code [extract or pullup method], Long method [extract method], Large/blob/god class [extract class], shotgun surgery [move method], long param list [replace param with method, introduce param object] , feature envy [move or extract method], data class [encapsulate field, extract or move method], primitive obsession [replace data value with object], switch statements [extract or move method, replace type code, replace conditional with polymorphism], message chain (lots of ().) [hide delegate], inappropriate intimacy [move method, extract class], comments [rename method]

[replace temp with query]

-Provide UML class diagram for xClass after refactoring xMethod() using specified design pattern: *refer to design patterns*

-Testing

-Write a class for a **mock object** that will allow for testing of line x of xClass

```
public class DummyPrinterService  implements PrinterService{
  boolean anInvoiceWasPrinted = false;
  @Override
  public boolean isPrinterConfigured() {
    return true;
  }
  @Override
  public void printInvoice(Invoice invoice) {
    anInvoiceWasPrinted = true;
  }
  public boolean anInvoiceWasPrinted() {
    return anInvoiceWasPrinted;
  }
}
```

-Provide 5 good test cases for a function. (Max of 1 test per equivalence)

	Addends	Sum	Description (also check commutative)
0	0	0	all zero special case
0	23	23	zero plus positive
-78	0	-78	negative plus zero
127	127	254	max signed bytes
-128	127	-1	min and max signed bytes
-128	-128	-256	min signed bytes
2147483647	2147483647		max signed integers
-2147483648	2147483647	-1	min and max signed integers
-2147483648	-2147483648		min signed integers

	Addends	Sum	Description (also check commutative)
4/3	2		expression
$2	$2		currency symbols
+5	3		plus sign
(9)	9		parentheses around negatives
l	1		lower case letter l
O	0		upper case letter O
<tab>	<tab>		no input
1.2	5		decimal
A	b		invalid characters

-Write test-cases using jUnit style unit tests for xMethod()

```
// check that value-based equality works
public void testEquals() {
    Assert.assertTrue(!aNumber.equals(null));
    Assert.assertEquals(aNumber, aNumber);
    Assert.assertEquals(aNumber, new Number(2));
    Assert.assertTrue(!aNumber.equals(anotherNumber));
}
```

-Theory

-Waterfall: Requirements Specification → Architectural Design → Detailed Design → Integration and Testing → Delivery and Operation → Maintenance and Support
    -verification at every phase, sequential, easily understood, enforces discipline, manufacturing view of software

-Prototyping: iterative design cycling through more than 1 design, improving with each iteration
    -throwaway: build and test prototype to be thrown away to build the real product
    -incremental: have "must do", "should do", and "could do"
    -evolutionary: features are refined over time ex: key for cut → undoable cut → drag and drop cut & paste
    -hand drawn sketches, storyboards,

-XP: 40 hour work week, metaphor, simple design, collective ownership, coding standards, small releases, continuous integration, refactoring, planning game, testing, on-site customer, pair programming

-Decomposition: dividing whole things into parts

-Dynamic Binding: selection of method to be run is made at run time depending on input parameters

-Requirements
    -User requirements: what tasks the user can do with the system
    -Functional requirements (features): what behaviors does system support
    -Non-functional requirements (qualities): how well the system should do what it does (fast? mem usage)

-Dependency Inversion Principle: Depend on abstractions, not concrete classes