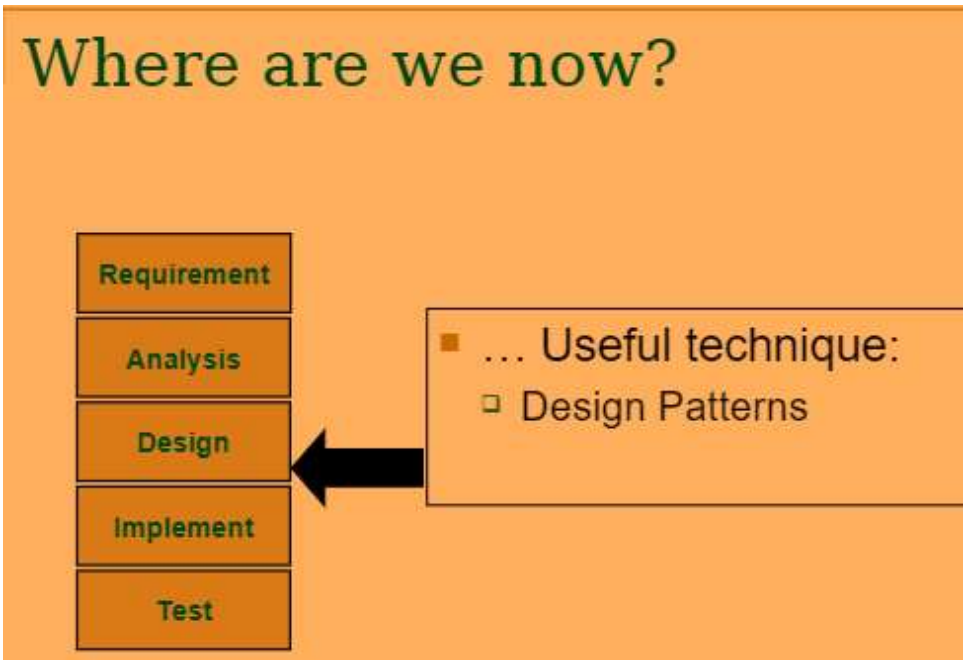
Lecture 9

Wednesday, March 15, 2023 5:40 PM

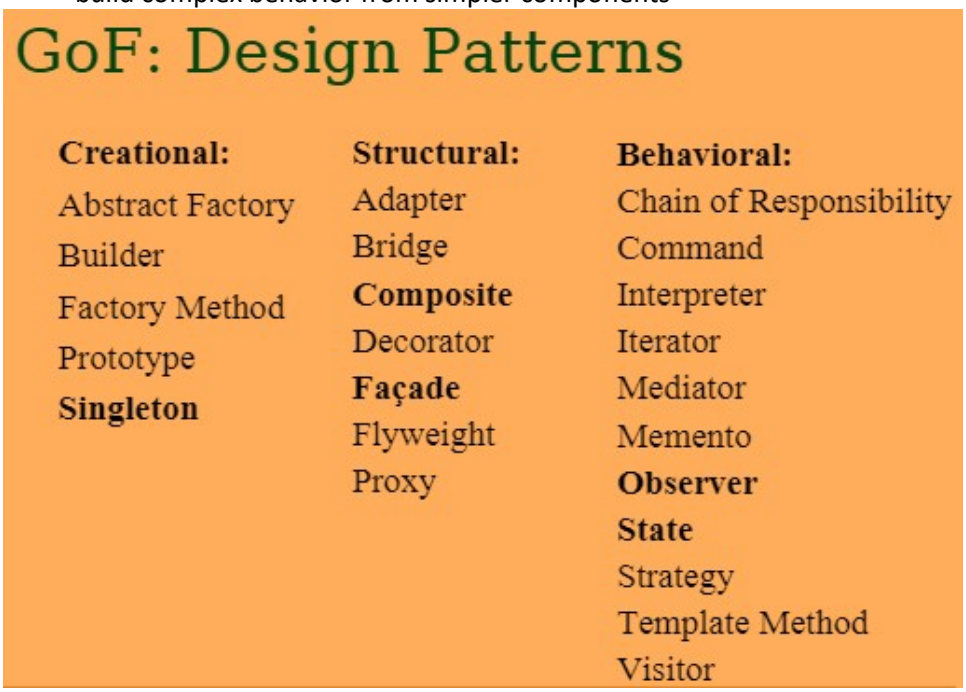
Overview of This Lecture

- Design Patterns:
  - Singleton
  - Abstraction-occurrence
  - General-Hierarchy
  - Composite
  - Façade
  - Player-role
  - State
  - Observer

-

- 

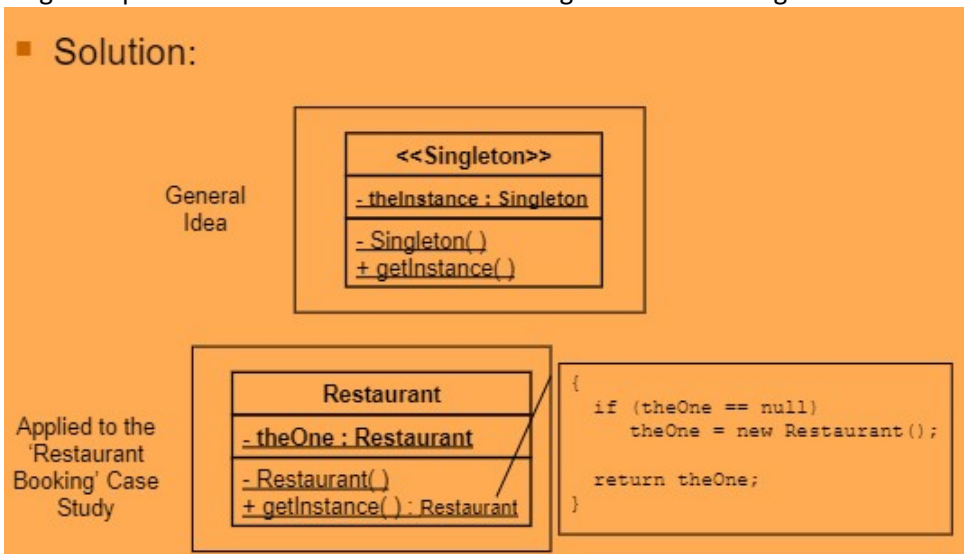
- Intro
  - Recurring/char problems during design activity
  - Designers come up w/ solutions
  - These are design patterns
- Design Patters:
  - Def: outline of a reusable solution to a general problem encountered in certain context
    - Describes recurring problem
    - Describes core of solution to that problem
  - These are named to facilitate ppl using & discussing them

- Usefulness:
  1. Reusing existing, high-quality solutions to design problems
  2. Improve future design: Solutions to new probs can be found w/ higher quality
  3. Shift lvl of thinking to higher perspective
  4. Use common terms to improve comm w/in team membs
  5. Improve documentation: smaller, simpler
  6. Use right design, not just one that works
  7. Studying patterns is effective way to learn from experience of others
- Pattern Categories
  1. Creational: creation of objects, separate the operations of an application from how its objects are created
  2. Structural: Concern abt composition of objects into bigger structures, provides possibility of future extension in structure
  3. Behavioral: define how objects interact and how responsibility is distributed among them. Use inheritance to spread behavior across the subclasses, or aggregation and composition to build complex behavior from simpler components
- 

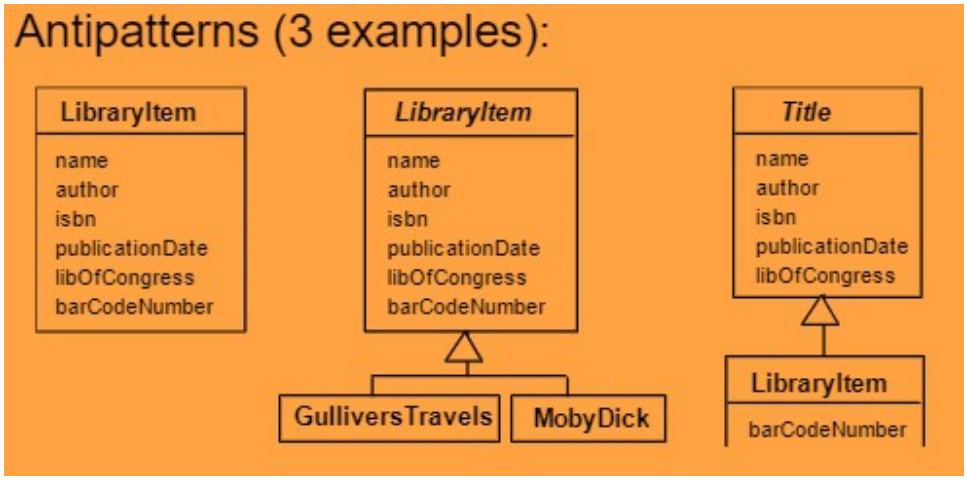
- 

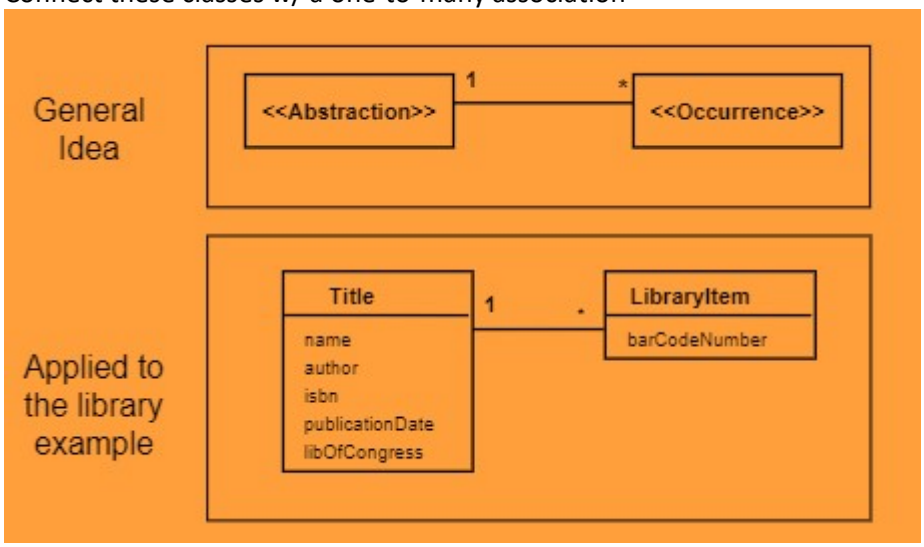
- Pattern Description Format
  - Context: general situation in which pattern applies

- Problem: main difficulty to be tackled, criteria for a good solution
- Solution: Recommended way to solve the problem
- Antipattern (optional): erroneous/inferior solution

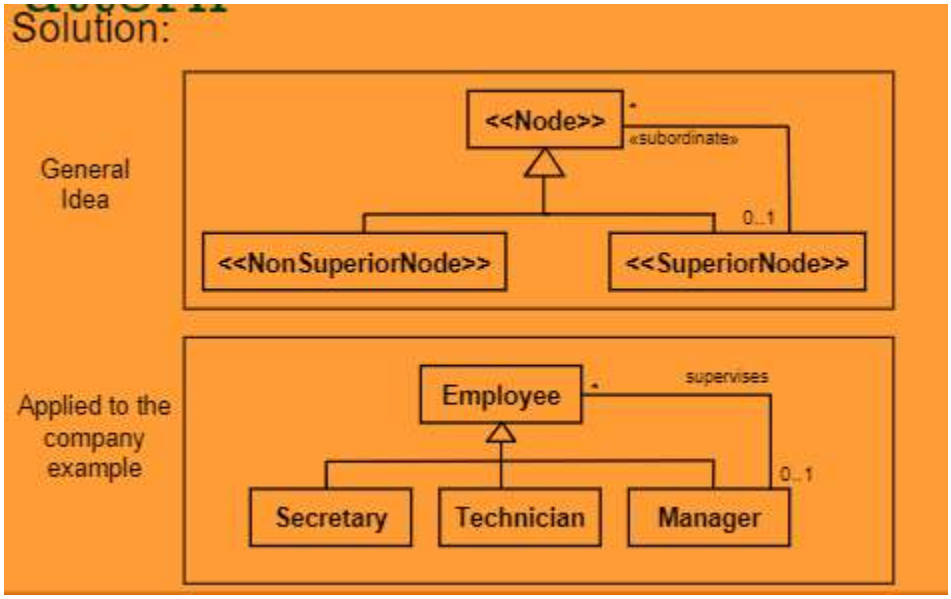
- Singleton Pattern
  - Context:
    - Common to have a class for which only 1 instance should exist
  - Problem:
    - Have to ensure that its impossible to make more than one instance
    - Having a public constructor means losing ctrl of the object creation process -> private constructor
    - But, singleton object itself must be accessible to other objects
  - Solution:
    - Use of public constructor can't guarantee that no more than one instance will be created
    - Singleton instance must also be accessible to all classes that require it
    - Solution:
      - A private (static) class variable, say theInstance
      - A public (static) class method, say getInstance()
      - A private constuctor
      - <<Singleton>> is abstract class icon, and Company is the concrete class icon
    - Singleton pattern shouldn't be overused bc Singleton instance is glbl var
    - 

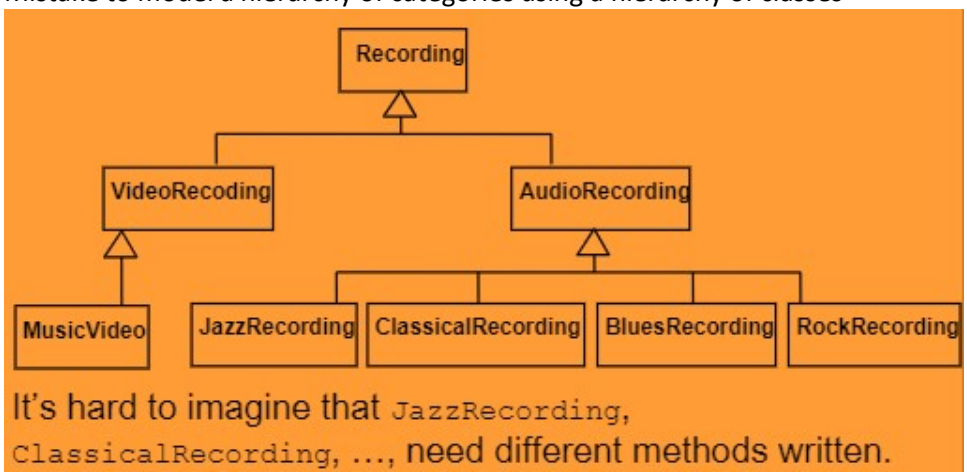
- Abstract-Occurrence Pattern
  - Context:
    - Often in domain model u find a set of related objects (occurrences), the members of such a set share common info but also differ from each other in important ways
  - Prob:
    - Find best way to rep such sets of occurrences in a class diagram
    - Rep objects w/out duplicating the common info
    - Avoid inconsistency when changing the common info
  - Antipatterns

- 

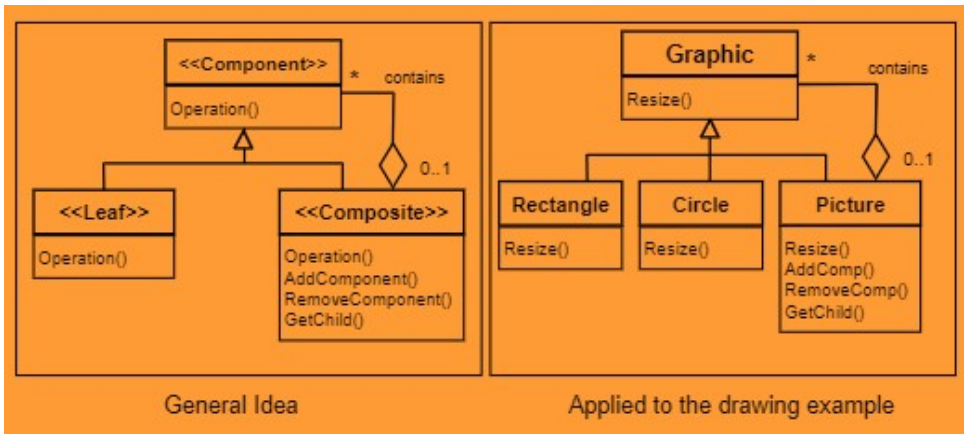
  - Solution:
    - Make an <<abstraction>> class that has the data that is common to all the members of a set of occurrences
    - Then make an <<occurrence>> class rep the occurrences of this abstraction
    - Connect these classes w/ a one-to-many association
    - 

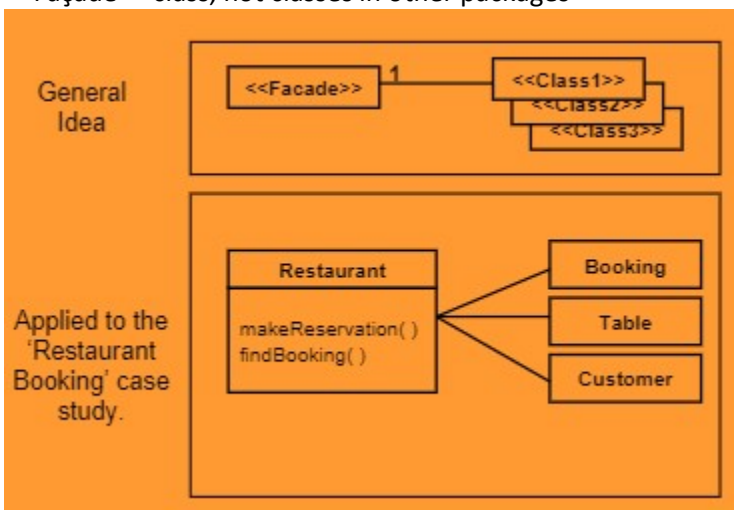
- General-hierarchy Pattern
  - Context:
    - Happens frequently in class diagram, there is a set of objects that have a naturally hierarchical relationship
  - Problem:
    - Draw a class diagram to rep a hierarchy of objects
    - Want a flexible way of representing the hierarchy
      - That prevents some objects from having subordinates
    - All objects share common features (properties and operations)
  - Solution:
    - Make an abstract <<Node>> class to rep the features possessed by each object in the hierarchy- one such feature is that each node can have superiors
    - Then make at least 2 subclasses of the <<Node>> class
      - <<SuperiorNode>> must be linked by a <<subordinate>> association to the superclass
      - <<NonSuperiorNode>> must be a leaf
      - Multiplicity of the <<subordinates>> association can be optional-to-many or many-to-many
    -

- 

  - Antipattern:
    - Mistake to model a hierarchy of categories using a hierarchy of classes
    - 

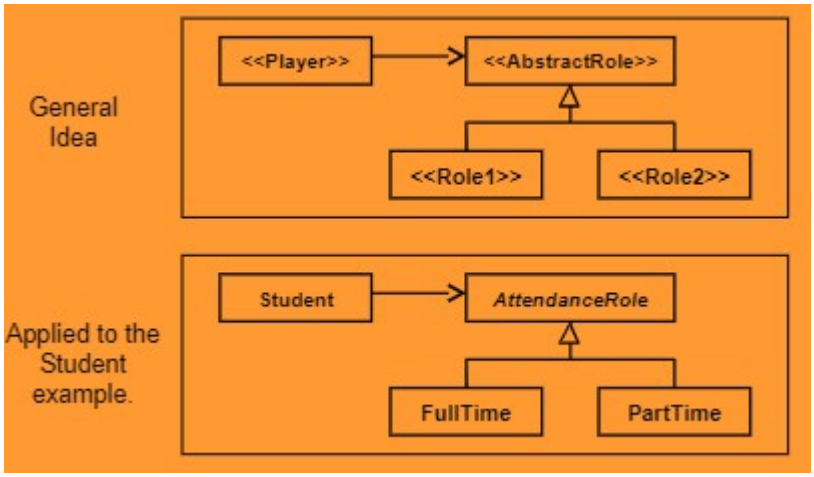
- Composite Pattern
  - Context:
    - U find that a set of objects shares similar functionality. More importantly, u can manipulate single object just like group of them
  - Problem:
    - Find best way to rep a single object as well as group of objects
    - Give same interface for all objects involved
  - Idea is to compose objects into tree structures ot rep part-whole hierarchies
  - Solution:
    - 'Leaf' and 'Composite' classes share a common interface, defined in 'Component'
    - 'Composite' implements this by iterating thru all its components
  - Composite pattern ('Gang of Four' book) is specialization of General-Hierarchy pattern
    - Association btwn <<SuperiorNode>> and <<Node>> is aggregation
  - Solution:

- 

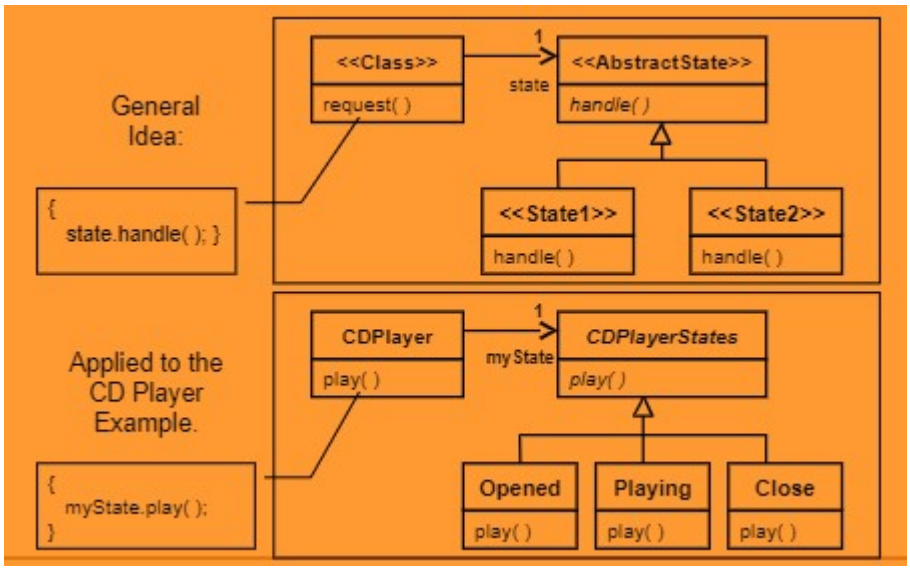
- Façade Pattern
  - Context:
    - Complex package/subsys can have many classes, pgrmers have to manipulate these classes in order to make use of the package/subsys
  - Problem:
    - Simplify the view for external users
      - Define a high lvl and simplified interface
    - Lower dependency of the external users on the internal working of a package
  - Solution
    - Make special class, called <<Façade>>, which will simplify the use of the package
    - The <<Façade>> will contain simplified set of public methods such that most other subsys don't need to access the other classes in the pckg
    - Net result is the pckg as whole is easier to use and has reduced numb of dependencies w/ other packages
    - Any change made to the package should only necessitate a redesign of the <<Façade>> class, not classes in other packages
    - 

- Player-Role Pattern
  - Context:
    - Role is certain set of properties associated w/ an object in a certain context, object may play diff roles in diff context
  - Problem:
    - Find best way to model players and roles so that a player can change roles/possess multiple roles
    - Want to avoid multiple inheritance
    - Cant allow instance to change class

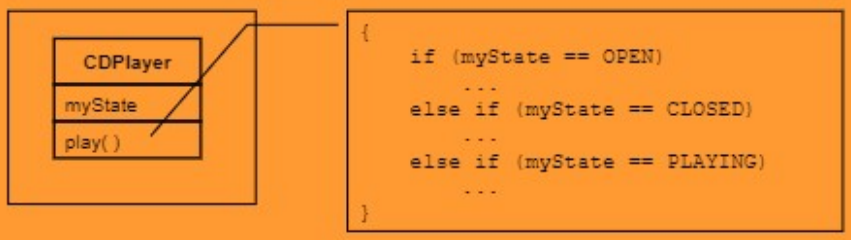
- Solution:
  - Make a class <<Player>> to rep objects that play roles
  - Make association from this class to an abstract <<Role>> class, a super-class of all possible roles
  - Subclass of <<Role>> encapsulate the properties and behaviors w/ diff roles
  - Multiplicity can be 1:1 or 1:many
  -
  -



- State Pattern
  - Context:
    - Object exhibits diff behavior, when internal state changes, object appears to have changed its class at run time
  - Problem:
    - Allow diff behaviors w/out actually changing the class of an object
    - State changing should be decided by the current behavior
    - Should allow only 1 behavior at any time
  - Solution:
    - Applications that request an object to alter its behavior when its internal state changes, like if a class is described by a state-chart
    - So, rep each state by a separate class
      - Each state class will implement the appropriate behavior for each op in that state only
    - Consequence of this pattern is that the state classes need access to the internal details of the context class
    -

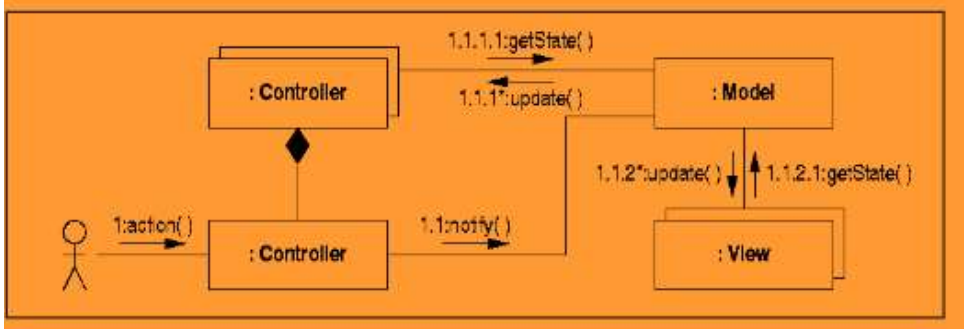


- Antipattern (like simple solutions):
  -
  - Code all the behaviors in one class, and make use of *if-then-else* or *switch* to decide what is the correct response.

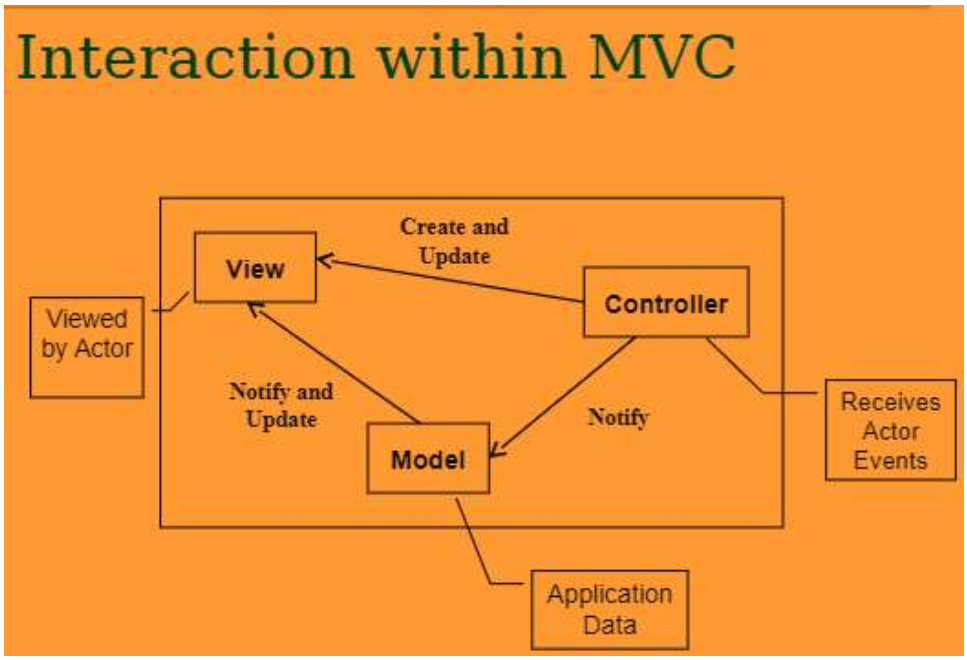


- Comparing player-role and state
  - Similarities:
    - <<Player>> is replaced by <<Class>>
    - <<AbstractRole>> is replaced by <<AbstractState>>
    - <<State1>>, <<state2>>, … are replaced by <<Role1>>, <<Role2>>, …
  - Diff:
    - Change of <<Role>> is decided by the <<Player>> instead of depending on the <,Role>>
    - Change of <<state>> is decided by the <<State>>
- Models, Views, and Ctrllers (MVC)
  - MVC: a design proposal put forward for the Smalltalk language:
    - To design pgrms w/ gui (- the user tho)
    - Sep manipulation and presentation of data
  - Now widely used in a variety of contexts:
    - Model stores and maintains data
    - Views display data in specific ways
    - Ctrllers detect and forward user input
  - Not a design pattern:
    - Background for understanding the observer pattern
  - 1 possible approach for conforming the Layered Architecture
  - EX:

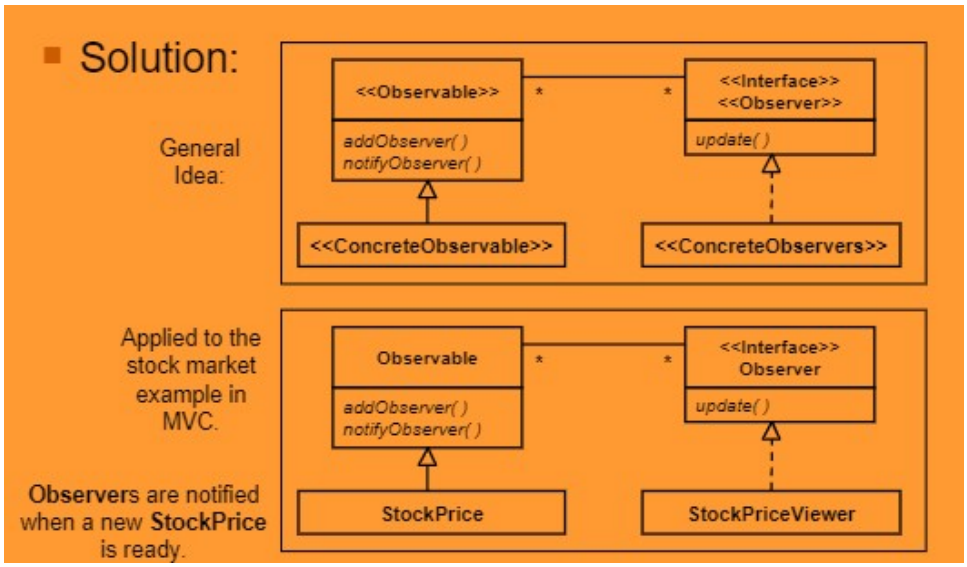
- The user input is detected by a controller.
- The model is notified.
- The views are updated.

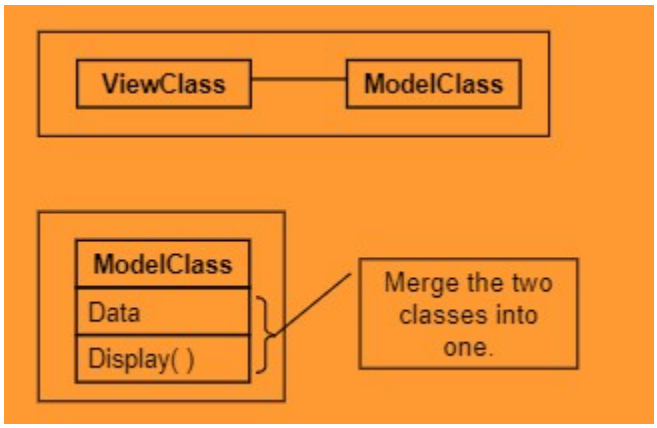


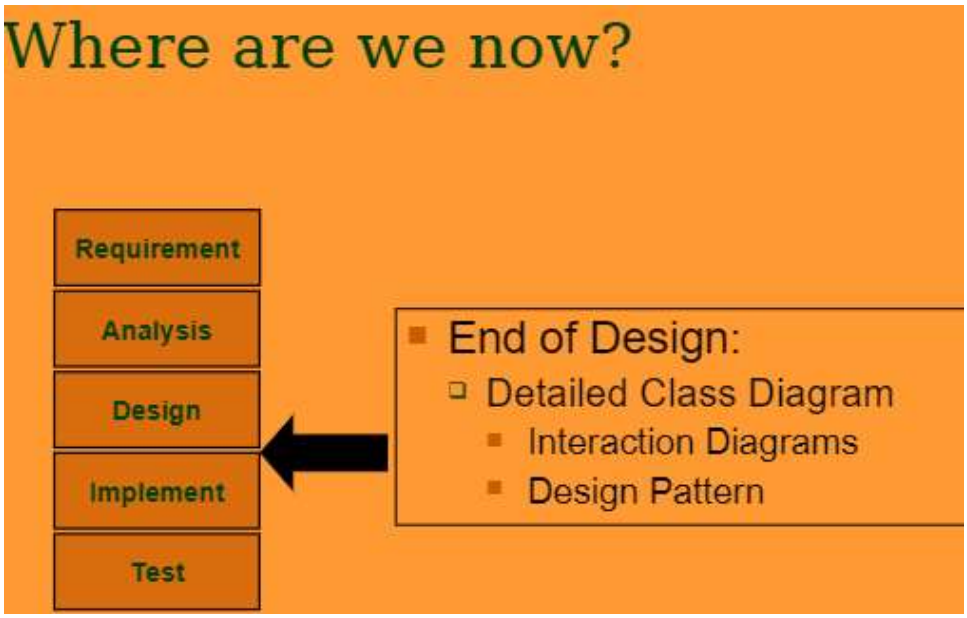
- Model:
  1. Manages behavior and data of the application domain
  2. Responds to instructions to change state (usually from the ctrller), and
  3. Responds to requests for info abt its state (usually from the views)
- View:
  1. Manages the graphical &/or text output of the application
  2. Gives presentation of the model
  3. Is look of the application that can access the model getters, but
  4. Has no knowledge of the setters
  5. Should be notified/updated when changes to the model happen
- Update of View
  - 2 simple models:
  - Push model
    - View registers itself w/ model for change notifications (observer pattern)
  - Pull Model:
    - View is responsible for calling the model when it needs to retrieve the most current data
  - EX:
  - Example: Stock monitoring program
    - Push Model: The view is notified whenever the stock price changes.
    - Pull Model: The view accesses the stock price from time to time automatically.
- Ctrller
  1. Reacts to user input, interprets mouse and keyboard inputs from the user, commanding the model &/or view to change as appropriate
  2. Translates interactions w/ the view into actions to be performed by the model
  3. Mods the model
- View and ctrller specifically designed to work together, tightly coupled

- 

- Observer Pattern
  - Context:
    - 2 way association makes tight coupling btwn 2 classes
    - Reusing/changing either of the classes will have to involve the other(when 1 object changes state, all its dependents are notified and updated auto)
    - On other hand, we want to max flex of the sys to best extent possible
  - Problem:
    - Lower the coupling btwn the classes, especially if they belong to diff subsys
    - Max the flexibility
  - Solution:
    - The <<Observable>> keeps list of <<Observer>>s
      - addObserver() method adds a new <<Observer>> to the list
    - When there is change in the <<Observable>>, all <<Observer>>s will be notified
      - notifyObserver() method notifies all <<Observer>>s in the list
    - <<Observer>> is an interface: any class that implements it can observe an <<Observable>>
    - Dependency btwn the <<Observable>> classes and <<Observer>> class is now transferred to <<Observable>> classes and the <<Observer>>s interface
    - As the interface is stable, changes to the classes that implemented the interface don't affect the <<Observable>> classes

- 

  - Antipatterns:
  - 

- Recommendations:
  - B4 applying a design pattern, make sure that:
    - Its relevant to ur prblm, not just superficial similarity
    - U have studied the tradeoff, sometimes a design pattern is not appropriate
  - Applying a design pattern implies a change in the overall design and coding
    - Make sure final sys follows thru
  - 


Summary
Design Patterns
Singleton
Abstraction-occurrence
General-Hierarchy
Composite
Façade
Player-role
State
Observer

Assessment L9

Wednesday, March 15, 2023 8:23 PM

Question 1
10 Points

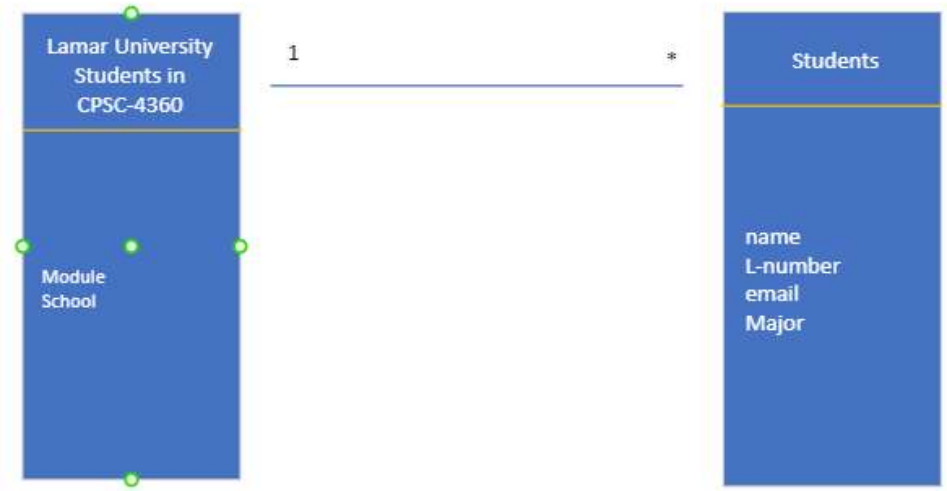
Suppose you have an application that needs to have only one object (instance) created for a given class (e.g., there is only one university called 'Lamar University'). Which is the design pattern proper to use in this case? Write a Java code so that your final program will respect this requirement. Also, show what happens when you try to create two objects of that class. Illustrate a way to access the private attributes of that class.

A: The appropriate pattern to use would be the Singleton Pattern.

Question 2
10 Points

Let us consider the set of Lamar University students that are registered to the CPSC-4360. Create a domain model where you group together the common information (e.g., university, module), but also differ from each other in important ways (e.g., student card number, name). Find the best way to represent such sets of occurrences in a class diagram. Try to represent the objects without duplicating the common information, by avoiding inconsistency when changing the common information. Which design pattern is appropriate for this domain model?

A: The most appropriate design pattern would be the Abstract-Occurrence Pattern.

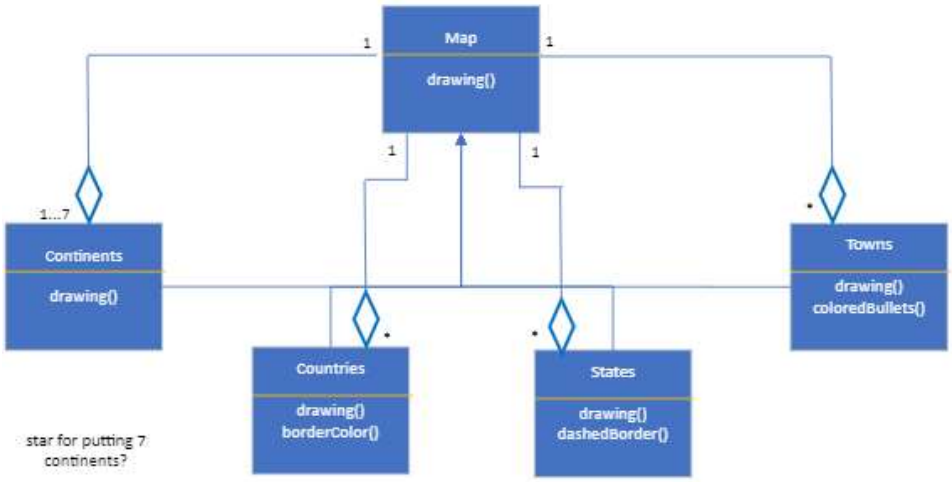


Question 3
10 Points

Suppose we want to draw continents, countries, states and towns. The 'drawing()' operation differs for each object: countries are shown with a specific colored border, states are shown with a specific colored dashed border, towns are pictured as different colored bullets, etc. For example, North America has countries like Canada, United States, and Mexico. Some of their states are Quebec, Ontario, Columbia,

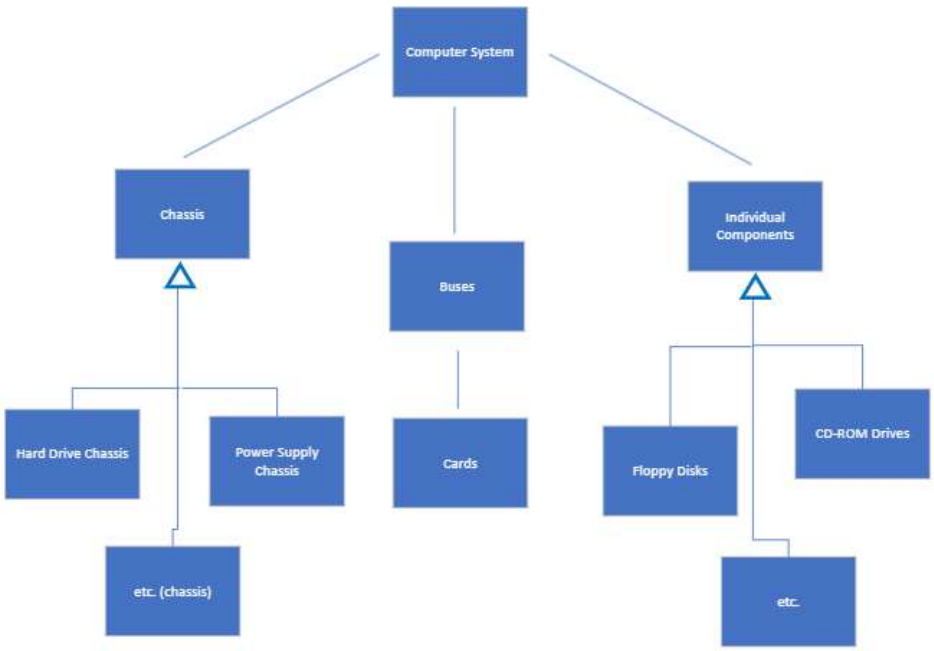
Texas, Louisiana, Arizona, etc. Some of their towns are Montreal, Toronto, Vancouver, Houston, Beaumont, New Orleans, Tucson, New Mexico, Guadalajara, etc. Write a design pattern that allows you to manipulate a single instance of the object just as you would do for a group of them when drawing them on a map.

A: An appropriate design would be the composite pattern.



Question 4
10 Points

A computer system can have various chassis that contain components (hard-drive chassis, power-supply chassis, etc) and buses that contain cards. The entire system is composed of individual components (floppy drives, cd-rom drives), buses and chassis. Design a UML class diagram to specify the 'is-a' relationships needed in this exercise.



Question 5
10 Points
What is true regarding the Observer pattern?

<<Observer>> must implement the method update();
Changes to the classes that implement <<Observer>> do not affect the observable classes;
There is a many-to-many dependency between the observable classes and the <<Observer>> abstract class;
Whenever there is a change in the observers, all observables will be notified;
None of the above.