

# Lecture 6: Design Evaluation and Intro to OO Design Patterns

# Principles

- Domain Simulation
- Information Hiding
- Cohesion and Coupling
- Design Patterns

# Domain Simulation

- Structure of system reflects structure of application domain
- Concepts and relationships derived from domain analysis
- Concept classes -> system classes
- Concept associations -> static class relationships

# Domain Simulation Benefits

- Domain changes less likely than functionality changes
- Functionality changes less likely to upset basic design structure

# Information Hiding

- Design decisions “hidden in modules”
  - Changes to implementation details can be localized
- Access to information via public methods/routines rather than direct access
- An original motivation for OO approach

# Information Hiding Examples

- DS Subsystems
  - GUI, Domain Logic, DataBase
  - DL and DB are components with subsystem proxy interfaces
    - All requests for DL and DB services are made to those proxies
    - Proxies hide how these responsibilities are designed and implemented

# Cohesion and Coupling

- Design expressed in terms of inter-related components
- Cohesion
  - Things in components belong together
  - Cohesion should be strong
- Coupling
  - Connections between components
  - Coupling should be weak

# Extreme Cases

- Single component
  - Bad cohesion
  - Good coupling
- Components contain a “single thing”
  - Good cohesion
  - Bad coupling
- Design is a balance between cohesion and coupling

# General Levels of Cohesion

- Coincidental
- Logical
- Temporal
- Procedural
- Data Flow
- Communicational
- Functional

# Coincidental

- Accidental, no functional rationale
- Division of code into modules determined by size consideration
- Isolation of commonly used but not functionally related pieces of code

# Logical

- Group things together that are the same kind of thing
- E.g. Module containing all the error handling routines
- E.g. Initialization module that opens files, initializes variables, etc.

# Temporal

- Group things together than can be done at the same time
- E.g. Initialization code
- E.g. Termination/clean up

# Procedural

- Group things together that will be part of the same computational structure
  - E.g. things that will be repeated inside a loop
- Control flow related items
  - E.g. if it is the first of the month, while looping through member records to find a date, check for account overdue members

# Data Flow

- Group A,B,C together if A prepares data used by B which prepares data used by C
- Idea is that A,B, and C must be part of the same function

# Communicational

- Group things together than use the same data
  - E.g. routines for accessing DB in different ways
  - E.g. methods for an object

# Functional

- Everything in component works together to support a common, well defined function
  - E.g. function for computing “goodness” of a match in a complex DS

# Coupling Measures

- Kinds of coupling
- Amount of coupling

# Kinds of Coupling

- One module changes the code in another
- One module directly accesses data in another
- Passing of control data, such as flags
- Passing of pointers
- Passing of functional computational data

# Amounts of Coupling

- Simple scalar variables
  - Number of
- Collections
  - Arrays, Vectors
  - But not simple count of size
- Record structures
  - Simple, nested

# Functional vs OO

## Cohesion and Coupling

- General levels of cohesion – created for functional/procedural programming
  - E.g. C programming
  - Best level of cohesion = functional
- OO orientation
  - Modules are OO components and classes
  - Class coupling/cohesion - different emphasis

# Class Cohesion

- Very low: class responsible for many things in very different functional areas
- Low: class responsible for many or complex things in one functional area
- Moderate: class has light or moderate responsibilities for a few different areas that are related to class but not each other
- High: class has moderate responsibilities in one functional area, cooperates with others to fulfill more complex responsibilities

# Class Coupling

- Class A is a subclass of class B (Strong)
- Class A implements interface B (Weaker?)
- A has an attribute (instance variable) for objects of class B (Weaker still)
- A class A method calls a class B method
- A has a method that
  - Returns items of class B
  - Has a local variable of class B
  - Has a parameter of class B (often associated with local variable case)

# Design Patterns

- Best Practices
- Described in general or abstract terms
  - Name, problem it solves, solution to problem
  - Notation: collaboration diagrams showing examples
- Guidelines for the creation of and allocation of responsibilities to design units

# System/Component Design Patterns – Discussed Earlier

- Layers
- Model-view, model-view-controller
- Subsystem as component plus interfaces
- Subsystem proxies
- Observer-observable

# Basic Patterns

- Expert
  - Who gets the responsibilities
- Creator
  - Who creates objects
- Controller
  - Who detects events, creates class instances if necessary, and calls on appropriate class methods

# Expert Pattern

- Assign the responsibility to the class that has the information necessary to fulfill those responsibilities
- Related principle
  - Object animation: Take the things that you would normally do to an object and have it do it itself
- Eg. DatingRequest object animation

**dL : DomainLogic**

```
dateRequest : DateRequest
```

|  |
|--|
|  |
|--|

**dL : DomainLogic**

|                   |
|-------------------|
| <b>: DataBase</b> |
|-------------------|

**dL : DomainLogic**

**: DataBase**

# Creator Pattern

- Give class B the responsibility of creating instances of class A if:
  - B aggregates object instances of A
  - B contains objects of class A
  - B records instances of objects of class A
  - B “closely uses” objects of class A
  - B has the initialization data that will be passed to an object of class A when it is created

# Creator Pattern Terminology

- B “aggregates” A objects: B has an instance variable of type A. Whole/part relationship.
- B “contains” A objects: B is a multi-object such as a vector
- B “records” instances of A objects: A objects are used to stand for events of type A

# Creator Pattern Example

- DatingRequest in DS
- Creator pattern criteria
  - B aggregates object instances of A
  - B contains objects of class A
  - *B records instances of objects of class A*
  - B “closely uses” objects of class a
  - *B has the initialization data that will be passed to an object of class A when it is created*

```

graph LR
    subgraph "Number Data"
        direction TB
        N1[ ]
        N2[ ]
        N3[ ]
        N4[ ]
        N5[ ]
        N6[ ]
        N7[ ]
        N8[ ]
        N9[ ]
        N10[ ]
        N11[ ]
        N12[ ]
        N13[ ]
        N14[ ]
        N15[ ]
        N16[ ]
        N17[ ]
        N18[ ]
        N19[ ]
        N20[ ]
        N21[ ]
        N22[ ]
        N23[ ]
        N24[ ]
        N25[ ]
        N26[ ]
        N27[ ]
        N28[ ]
        N29[ ]
        N30[ ]
        N31[ ]
        N32[ ]
        N33[ ]
        N34[ ]
        N35[ ]
        N36[ ]
        N37[ ]
        N38[ ]
        N39[ ]
        N40[ ]
        N41[ ]
        N42[ ]
        N43[ ]
        N44[ ]
        N45[ ]
        N46[ ]
        N47[ ]
        N48[ ]
        N49[ ]
        N50[ ]
        N51[ ]
        N52[ ]
        N53[ ]
        N54[ ]
        N55[ ]
        N56[ ]
        N57[ ]
        N58[ ]
        N59[ ]
        N60[ ]
        N61[ ]
        N62[ ]
        N63[ ]
        N64[ ]
        N65[ ]
        N66[ ]
        N67[ ]
        N68[ ]
        N69[ ]
        N70[ ]
        N71[ ]
        N72[ ]
        N73[ ]
        N74[ ]
        N75[ ]
        N76[ ]
        N77[ ]
        N78[ ]
        N79[ ]
        N80[ ]
        N81[ ]
        N82[ ]
        N83[ ]
        N84[ ]
        N85[ ]
        N86[ ]
        N87[ ]
        N88[ ]
        N89[ ]
        N90[ ]
        N91[ ]
        N92[ ]
        N93[ ]
        N94[ ]
        N95[ ]
        N96[ ]
        N97[ ]
        N98[ ]
        N99[ ]
        N100[ ]
    end
    N1 --> DL[dL : DomainLogic]
    DL --> DR[dateRequest : DateRequest]

```

**2: create(userName, daterPrefs)**  
**3: memberData = execute()**

```
graph LR
    subgraph "Domain Logic"
        direction TB
        DL[dL : DomainLogic]
    end
    DL --> UI[UI : UserInterface]
```

```

class Logic
class DataBase

```

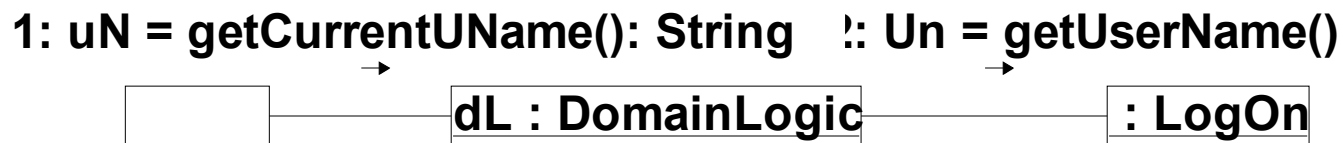
```
graph LR
    A[ ] --> B[dL : DomainLogic]
```

```
mD = getMemberData(userName)
```

```
Logic ————— : DataBase
```

# Expert & Creator Pattern - LogOn

- DL creates LogOn because it has the initialization data and records instances of the LogOn event
- LogOn will have the information necessary to determine user type, so it will be given this expert responsibility



# Controller Pattern

- Controllers
  - System
  - Business
  - Role
  - Use Case
- System may have more than one controller

# System Controller

- Object representing system or subsystem
- Could be the subsystem proxy/interface
- E.g. DS subsystem controllers
  - GUI, Domain Logic, and DB controllers
- E.g. sub-subsystem controller:  
OptionSelectionDialog,  
SetMemberDataDialog in GUI subsystem

# Business Controller

- Business is whole enterprise
- E.g. Dating System is part of a personal services corporation
  - Controller is personal service corporation object/class

# Role Controller

- Role is one of the actors using the system
- E.g. DS: Member Controller, Administrator Controller

# Use Case Controller

- Good cohesion
  - can keep track of all related events
  - Enforce sequence and out-of-sequence issues for use case
- E.g. DS
  - GetA Date
  - AddA Member

# Which to Use?

- System Controllers
  - If bloated and poorly cohesive, decompose to use case controllers
- Role Controllers
  - May violate object animation since tends to put responsibilities in an object that is the function rather than the data (the thing that acts on the data rather than the data that is acted on)