Key Concepts

- **General**
  - Cohesion
  - Coupling
  - Information hiding
    - Encapsulation
    - Secrets
  - Binding time
- **OO Specific**
  - Class vs. Interface Inheritance
    - Class = implementation
    - Interface = type
  - Inheritance / composition / delegation

Key design activity

- Assigning responsibilities to classes
  - A *responsibility* is something that the class is required to do (contract or obligation of a class)
    - All the responsibilities of a given class should be *clearly related.*
    - If a class has too many responsibilities, consider *splitting* it into distinct classes
    - If a class has no responsibilities attached to it, then it is probably *useless*
    - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created
- Determine which objects need to know of other objects (determine class navigability)

General Responsibility Assignment Software Patterns or Principles (GRASP)

Information Expert: A general principle of object design and responsibility assignment?

Assign a responsibility to the information expert – the class that has the information necessary to fulfill the responsibility

Creator: Who creates?

Assign class B the responsibility to creat an instance of class A if one of these is true:

(1) B contains A

(2) B aggregates A

(3) B has the initializing data for A

(4) B records A

(5) B closely uses A

Controller: What first object beyond the UI layer receives and coordinates ("controls") a system operation?

Assign the responsibility to an object representing one of these choices:

(1) Represents the overall "system", a "root object", a device that the software is running within, or a major subsystem (these are all variations of a façade controller)

(2) Represents a use case scenario

GRASP

Low Coupling: How to reduce the impact of change?

Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives

High Cohesion: How to keep objects focused, understandable, and manageable, and as a side-effect, support low coupling?

Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives

Polymorphism: Who is responsible when the behavior varies by type?

When related alternatives or behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies

Pure fabrication: Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent the problem domain concept – something made up, in order to support high cohesion, low coupling, and reuse.

GRASP

Indirection: How to assign responsibilities to avoid direct coupling?

Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

Protected Variations: How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?

Identify points of predicted variations or instability; assign responsibilities to create a stable "interface" around them

Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

Why Patterns?

- Design for re-use is difficult
- Experienced designers:
  - Rarely start from first principles
  - Apply a working "handbook" of approaches
- Patterns make this ephemeral knowledge available to all
- Support evaluation of alternatives at higher level of abstraction

Discussion question:

“New Pattern” is an Oxymoron

Example of a Software Development Pattern - The Model-View-Controller (MVC) Pattern

- The *Model* component: encapsulates core functionality; independent of input/output representations and behavior.
- The *View* components: displays data from the model component; there can be multiple views for a single model component.
- The *Controller* components: each view is associated with a controller that handles inputs; the user interacts with the system via the controller components.

Pattern Types

- **Requirements Patterns**: Characterize families of requirements for a family of applications
  - The checkin-checkout pattern can be used to obtain requirements for library systems, car rental systems, video systems, etc.

- **Architectural Patterns**: Characterize families of architectures
  - The Broker pattern can be used to create distributed systems in which location of resources and services is transparent (e.g., the WWW)
  - Other examples: MVC, Pipe-and-Filter, Multi-Tiered

- **Design Patterns**: Characterize families of low-level design solutions
  - Examples are the popular Gang of Four (GoF) patterns

- **Programming idioms**: Characterize programming language specific solutions

The Gang of Four Catalog Method

- Pattern name and classification
  - Purpose classification: creational, structural, behavioral
  - Scope classification: class (compile time) or object (run-time)
- Creational
  - class => defer creation to subclasses
  - object => defer creation to another object
- Structural
  - class => structure via inheritance
  - object => structure via composition
- Behavioral
  - class => algorithms/control via inheritance
  - object => algorithms/control via object groups

Pattern Description - 1

- Intent
  - What does pattern do?
  - What is its rationale?
  - What issue/problem does it address?
- Also Known As = other names for pattern
- Motivation
  - Scenario illustrating problem and solution
  - A concrete exemplar
- Applicability
  - When to apply the pattern
  - Poor designs addressed by pattern

Pattern Description - 2

- Structure
  - Graphical representation – OMT/UML
- Participants
  - Classes, objects, and their responsibilities
- Collaborations
  - Class/object interactions

Pattern Description - 3

- Consequences
  - How does pattern meet its objectives
  - Tradeoffs and results
  - Flexibility: what parts can vary independently?
- Implementation & Sample Code
- Known Uses
- Related Patterns

Design problem

- We start you off immediately thinking about the design of a software system
  - This class
    - Individually create a design for the stated problem
    - Ask questions/Collaborate with others
  - Next class
    - Submit individual design at start of class
    - Groups of students create a consensus design
    - Designs will be presented
    - Designs will be compared and contrasted