Announcements

- **Homework 1**
  - Due **Tomorrow 11:58pm**
- **Code repository, project name, and product backlog**
  - Due **Friday, September 8**
Project grade breakdown

- Project Charter – 5%
- Requirements Document – 14%
- Design Document – 14%
- Sprint 1 Planning Document – 5%
- Sprint 1 Review – 13%
- Sprint 1 Retrospective – 2%
- Sprint 2 Planning Document – 5%
- Sprint 2 Review – 13%
- Sprint 2 Retrospective – 2%
- Sprint 3 Planning Document – 5%
- Sprint 3 Review – 13%
- Sprint 3 Retrospective – 2%
- Final Project Presentation and Demo – 7%
Lecture 05

- Basing development on reusable technology
- Client-server architecture
- Unified Modeling Language (UML)
“If I have seen further it is by standing on ye sholders of giants.”
– Isaac Newton
Building on the experience of others

- Software engineers should avoid re-developing software
- ...and try to reuse:
  - Expertise
  - Standard designs and algorithms
  - Libraries
  - Frameworks
  - Complete applications
Reusability and reuse

- Reuse and design for reusability should be part of the culture of software development organizations.
- But
  - Why take extra time to develop something that will benefit other projects?
  - What if management primarily rewards ‘visibility’?
- Software is often created without enough attention to quality or reuse.
Vicious cycle

- Developers take short cuts to save time, sacrificing quality and reusability.
- Important to recognize that:
  - This cycle costs money.
  - Investing in reusable code is important.
  - Attention to quality is essential.
  - Employing reusable components often simplifies design.
Frameworks

- A **framework** is reusable software that implements a generic solution to a generalized problem
  - Provides common facilities applicable to different applications
- Based on the principle that applications that do related things tend to have similar designs
Frameworks promote reuse

Intrinsically incomplete

- **Slots**: certain classes or methods are missing
- **Hooks**: optional functionality, allowance made for developer to provide it

Developers use the services that the framework provides

- Application Program Interface (API)
Object-oriented frameworks

- Framework is composed of a library of classes
  - API is defined by the set of all public methods
  - Some classes intentionally abstract

- For example
  - Payroll management
  - Frequent buyer clubs
  - University registration
  - E-commerce web site
Product line

- **A product line** (or product family) is a set of products built on a common technology base.
  - Individual products have different features to satisfy different markets.
  - Common software technology included in a framework.
  - Each product produced by filling in desired hooks and slots.
Distributed system

- A distributed system is a system of discrete networked components
  - Have concurrency
  - Lack a global clock
  - Can encounter independent failure of components
  - Coordinate by passing messages
- Components cooperate to create a system
Client-Server Architecture

- One kind of a distributed application or system
- Server
  - Program that provides a service for other programs
- Client
  - Program that accesses one or more servers to obtain services
- Communication Channel
  - Generally a computer network
  - Client must initially know the server, but not vice versa
Examples

- Email
- DNS
- World Wide Web
- /etc/services
Sequence

- Server starts running
  - Creates a socket
  - Binds the socket to an address
  - Waits for clients (listening)
- Client requests something from the server
  - Creates a socket
  - Attempts to connect to server
- Server accepts the connection
- Send and receive data (read and write)
...and

- Initialization
- Disconnection
- Termination
Server

For the server as a whole:
- Initializing
- Waiting
  - start listening
  - stop listening
  - accept connection
- terminate

For each connection:
- Handling a Connection
  - do: react to messages
  - handle disconnection
Client

- Initialize
  - Initiate a connection to a server
  - Interact with the user, sending messages to the server as necessary
  - Respond to events triggered by the server
    - Do: respond to messages
    - Do: handle server disconnection
- Terminate
Advantages

- Work can be distributed among different machines
  - Load balancing
  - Failover
- Client can access server from a distance
- Client and server can be designed separately
  - Both can be simpler
Data can be kept centrally at the server
Or among many geographically distributed clients or servers
Server may be accessed simultaneously by multiple clients
Thin vs. fat

- **Thin-client**
  - Client is small as possible
  - Most work done on server
  - Client easy to download

- **Fat-client**
  - As much work as possible delegated to clients
  - Server can handle more clients

- Usually somewhere in between
Protocols

- Protocols are to communications what programming languages are to computation
- Server and client are programmed to understand the protocol
Developing client-server applications

- Design the primary work to be performed by client and server
- Design how the work will be distributed
- Design the protocol
Broadly

- Initializing
- Handling connections
- Sending and receiving messages
- Terminating
Object Client-Server Framework (OCSF)
Using OCSF

- More at http://lloseng.com/
- Avoid modifying the three classes
- Create subclasses
- Call public methods
- Supply slot methods and override
- Provide hook methods
What is UML?

- A “general purpose” modeling language
- Provides a standardized method for designing a system
- Adopted by OMG (Object Management Group) in 1997
- Latest standard, 2.0, published in 2005
UML

- ...has detailed semantics
- has extension mechanisms
- has an associated textual language
  - Object Constraint Language (OCL)
- Not a methodology
  - Assists in software development
What is a “good” model?

- A model that
  - Uses a standard notation
  - Is understandable by all stakeholders
  - Helps software engineers gain insight into the system
  - Provides abstraction

- When do you use it?
  - To help create designs
  - To analyze and review designs
  - As documentation for the system
Diagram essentials

- **Classes**: represent the types of data themselves
- **Associations**: linkages between class instances
- **Attributes**: simple data found in classes and instances
- **Operations**: represent the functions (methods) performed by classes
- **Generalizations**: grouping of classes into inheritance hierarchies
Diagrams

- Static (structural) view
  - Class diagrams
  - Composite structure diagrams

- Dynamic (behavioral) view
  - Sequence diagrams
  - Activity diagrams
  - State machine diagrams
Class diagrams

- Describes the system’s classes
  - Attributes
  - Operations (methods)
  - Relationship among objects

<table>
<thead>
<tr>
<th>Rectangle</th>
<th>Rectangle</th>
<th>Rectangle</th>
<th>Rectangle</th>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getArea()</td>
<td>height</td>
<td>height</td>
<td>- height: int</td>
</tr>
<tr>
<td></td>
<td>resize()</td>
<td>width</td>
<td>width</td>
<td>- width: int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+ getArea(): int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+ resize(int,int)</td>
</tr>
</tbody>
</table>
Composite structure diagrams

- Shows internal structure of a class
- Parts: runtime role of a classifier (e.g., class) or a collection of instances
- Port: connects classifiers with their parts and environment
- Connector: binds two or more entities together
Sequence diagrams

Diagram showing the interaction between a computer and a server for checking and sending emails.
Activity diagrams

- Graphical workflow showing stepwise activity and actions
  - Supports choice, iteration, and concurrency
- i.e., a flowchart
  - Petri-net (later)
State machine diagrams

- Recall: mealy/moore FSMs
Associations and multiplicity

- **An association** shows how two classes relate to each other.
- **Symbols indicating multiplicity** are shown at each end of the association.

![Diagram showing associations and multiplicity](image)
Labeling associations

```
Employee * worksFor 1 Company

AdministrativeAssistant * 1..* supervisor Manager

Company 1 1 BoardOfDirectors

Office 0..1 allocatedTo * Employee

Person 0,3..8 boardMember * BoardOfDirectors
```
Many-to-one

- A company has many employees
- An employee can work for only one company
- A company could have zero employees
  - “Shell” company
- Cannot be an employee unless you work for a company
Many-to-many

- An assistant can work for many managers
- A manager can have many assistants
- Assistants may work in pools
- Managers may have a group of assistants
- Some managers may have zero assistants
- Could an assistant have zero managers?
One-to-one

- For a given company, exactly one board of directors
- Board oversees only one company
- Company must always have a board
- Board must always be with a company
Analysis and validation

Avoid unnecessary one-to-one associations

Not this

This

Person
  name
  1

PersonInfo
  address
  email
  birthdate
  1

Person
  name
  address
  email
  birthdate
  1
More complex example

- Booking is always for exactly one passenger
  - Never zero passengers
  - Never more than one
- Passenger can have any number of bookings
  - None
  - More than one
- Frame is an optional feature of any UML diagram
Association classes

- Sometimes, an attribute concerning two classed cannot be placed in either
- These are equivalent:
Reflexive associations

- An association can connect a class to itself
Generalization

- A generalization set is a labeled group of generalizations with a common superclass.
- The label (discriminator) describes the criteria used for specializing.

![Diagram of Animal classification]

- Animal
  - habitat
    - AquaticAnimal
    - LandAnimal
  - typeOfFood
    - Carnivore
    - Herbivore
Multiple discriminators

![Class diagram showing the hierarchy of animals with their habitats and type of food]

1. **Animal**
   - **Habitat**: Aquatic, Land
   - **Type of Food**: Aquatic Carnivore, Aquatic Herbivore, Land Carnivore, Land Herbivore
Objects do not change class

For example, a student may be part-time some times or full time others.
Notes and descriptive text

- Descriptive text and other diagrams
  - Embed your diagrams in a larger document
  - Text can explain aspects of the system using any notation you like
  - Highlight and expand on important features
    - Give rationales

- Notes
  - Small block of text embedded in the UML diagram
  - Think comment
Modelio

- Open source modeling environment
  - https://www.modelio.org
- Demo
Developing class diagrams

- UML models can be created at different stages
  - Different purposes?
  - Different levels of detail

- Exploratory domain model
  - Domain analysis, represents aspects of the domain

- System domain model
  - Models aspects of domain represented by the system

- System model
  - Includes classes for UI and system architecture
System domain model vs. system model

- The system domain model omits many classes needed for a complete system
  - Could contain less than half of them
  - Developed and used independently of
    - UI classes
    - Architectural classes

- Complete system model includes
  - System domain model
  - UI Classes
  - Architectural classes
  - Utility classes
Suggested sequence

- Identify a first set of candidate classes
- Add associations and attributes
- Find generalizations
- List the main responsibilities of each class
- Decide on specific operations
- Iterate over the entire process until model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities, operations, etc
  - Identify interfaces
  - Apply design patterns (later)
Identifying classes

- Developing a domain model of leads to discovery of classes
- When you work on the UI or architecture, one tends to invent classes to solve a design problem
- Reuse should always be kept in mind
Discovering domain classes

- Look at source material such as description of requirements
- Extract nouns and noun phrases
- Eliminate nouns that
  - ...are redundant
  - ...represent instances
  - ...are vague or highly general
  - ...not needed in the application
- Pay attention to classes in a domain model that represents types of users or other actors
Identifying associations and attributes

- Start with classes you think are most central and important
- Decide on clear and obvious data each must contain in its relationships to other classes
- Work outwards towards classes that are less important
- Avoid adding many associations and attributes to a class
  - Systems are simpler if they manipulate less information
Attributes

- Look for information about each class that must be maintained
- Nouns rejected as classes may make good attributes
- Attribute should generally contain a simple value
  - String, number, etc
Tips

- Avoid duplicates if possible
- If a subset can be formed, create a distinct class

Bad, due to a plural attribute

Bad, due to too many attributes, and the inability to add more addresses

Good solution. The type indicates whether it is a home address, business address etc.
Example

The diagram illustrates the relationships between entities:

- **Passenger**:
  - name
  - number

- **Employee**:
  - name
  - employeeNumber
  - jobFunction

- **RegularFlight**:
  - time
  - flightNumber

- **Booking**:
  - seatNumber

1. Passenger is related to Employee with a * multiplicity.
2. Employee is related to SpecificFlight with a * multiplicity.
3. SpecificFlight is related to Passenger with a * multiplicity.
4. Passenger is related to RegularFlight with a 1 multiplicity.
5. Employee is related to RegularFlight with a 0..1 multiplicity and is labeled as "supervisor."
Allocating responsibilities

- A responsibility is something that the system is required to do
- Each functional requirement should be attributed to a class
  - All responsibilities for a given class should be clearly related
  - Too many responsibilities? Consider splitting into two or more classes
  - No responsibilities? Likely useless
  - Trouble attributing a responsibility to a class? Create a new class
Responsibilities

- Perform **use case analysis**
  - Look at user stories
- Look for **verbs and nouns describing actions in the system description**
Example

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking
Prototyping class

- Write names on small cards as you identify classes
- List attributes and responsibilities as they become known
  - If you cannot fit everything on one card, maybe it should be split
- Move cards around on a whiteboard to create a class diagram
- Draw lines among the cards for associations and generalizations
Identifying operations

- Operations are needed to realize the responsibilities of each class
  - May be several operations for one responsibility
  - Main operations are normally public
  - Other methods that collaborate to perform the responsibility should be private
Class diagrams in Java

- Attributes are instance variables
- Generalizations → \texttt{extends}
- Interfaces → \texttt{implements}
- Associations are normally implemented using instance variables
  - Divide two-way association into two one-way associations (each class has an instance variable)
  - Multiplicity one or optional, declare a variable of that class (a reference)
  - Multiplicity of many, use a collection class implementing List (e.g., \texttt{vector})
Questions?