UML Design Modeling
## Analysis and Design

<table>
<thead>
<tr>
<th><strong>Analysis</strong> — “what (and under what constraints)”</th>
<th><strong>Design</strong> — “how”</th>
</tr>
</thead>
<tbody>
<tr>
<td>investigation and specification of the problem</td>
<td>specification of a software solution</td>
</tr>
</tbody>
</table>

### Requirements
- Use cases

### Structure of problem (domain) concepts
- Requirements class model

### Design constraints

### Structure of solution concepts
- logical/physical architecture
- design class model

### Behavioral concepts
- sequence diagrams
- statemachine diagrams
Requirements vs. Design

During requirements, we
- understand the problem,
- clarify and record the constraints and requirements,
- focus on essential behavior rather than solutions.

During design, we
- focus on developing “good” solutions,
- create a software (and hardware) solution that meets the wishes of the stakeholders.
What is Software Design?

- A software design expresses a solution to a problem in programming language independent terms.
- This permits a design to be implemented in any programming language.
- Designs can be expressed at varying level of abstraction
Controlling Complexity

Complexity handling mechanisms in software development are based on the following separation of concerns principles:

- Modularization - breaking up a solution into logical units.
  - In OO design solutions are expressed in terms of classes
  - In Structured Design, Service-Oriented Architectures, Component-Based Architectures, solutions are expressed in terms of modules that encapsulate a well-defined set of related functionality.

- Abstraction - use functional and data abstraction.
  - Functional abstraction: functions/services reveal only the data needed to use them (e.g., information on input and output data, pre- and post-conditions)
  - Data abstraction: data is understood in terms of how they can be manipulated; representation details are hidden from users of the data (e.g., abstract data types)
Design Principles

- Formality and rigor
- Separation of concerns
  - Abstraction
  - Modularity
  - Separation of views
- Design for change
- Reusability
- Incrementality
Primary trace relationships between basic requirements and design models

- **Requirements Models**
  - Requirements class model
  - Use cases

- **Design Models**
  - Design class model
  - Sequence model
  - Statemachine model
Design Models – Large System Modeling

- Architectural Models
  - Model solution as a network of subsystems
  - Subsystems treated as blackboxes (internal details not described)
  - Static Models: Subsystem structure models
  - Behavioral Models: Subsystem interaction models

- Detailed (Subsystem) Design Models
  - Describe how subsystems accomplish their goals (internal view of subsystems)
  - Static Models: Design class models
  - Behavioral Models: State models, sequence models
Basic Design Class Models

CS314 Review
A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

UP Domain Model
Stakeholder's view of the noteworthy concepts in the domain.

UP Design Model
The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.
Key design activities

- Assign responsibilities to classes
- Determine which objects need to know of other objects (determine class navigability)
Allocating responsibilities to classes

- A responsibility is something that the system is required to do.
  - Each functional requirement must be attributed to one of the classes
    - 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
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
Patterns of Responsibility Assignment

Principles

- Cohesion
- Expert
- Creator
- Low Coupling
- Controller
High Cohesion

- Cohesion is a measure of how diverse an entity’s features are.
  - A highly cohesive class has features that pertain to a single concept
  - A highly cohesive class has one general responsibility
    - Guideline: Should be able to describe responsibility of a highly cohesive class in one sentence
    - Use sentence as comment in code

- Guideline: Assign a responsibility so that parts of the class are strongly related and the class responsibility is tightly focused
  - Class easier to understand
  - Easier to maintain and reuse
When to ignore high cohesion guidelines

- A class that provides a single point of entry into a system may sometimes be desirable
  - Such a class is called a Façade and provides external clients with a single point of access to services offered by a system

- For efficiency reasons it may be more appropriate to place two diverse classes in the same class
  - Rather than an object delegating responsibility for a service to another object it may carry it out itself to avoid delegation performance overhead
Expert

- Assign responsibility to the class that has the information necessary to discharge the responsibility.
- Naïve use can lead to undesirable coupling and low cohesion.
  - Giving a class the responsibility for storing its objects in a database leads to low coupling and undesirable coupling
    - Low cohesion: class contains code related to database handling between the
    - Undesirable coupling: class is tightly coupled to database services provided by another system
  - Example: Elevator Control System
Class B can be responsible for creating objects of A in the following situations:

- A is a part class of B
- B is a container of A objects
- B records A objects
- B has the data needed to initialize A objects
Low Coupling

- Assign responsibilities to reduce high coupling to unstable classes (i.e., classes with high probability of significant changes)
  - Reduces impact of change
  - Classes can be understood in relative isolation

- Forms of coupling in OO designs
  - Class X contains a reference to Class Y objects
  - Class X operation includes calls to Class Y operations
  - Class X operation has a Class Y object as a parameter or declares a Class Y object as a local variable
  - Class X is a direct or indirect subclass of Class Y
  - Class X implements an interface Y

- Classes designed for reuse should have low coupling. Why?
Controller

- Assign responsibility for handling a system event to a class representing the system or a class that is responsible for handling the events in a group of related use cases.
  - A system event is an event generated by an actor. A system event results in the execution of a system operation.
  - A controller is a non-user interface class responsible for receiving and handling system events. A controller defines the method for the system operation.

- A good controller delegates the work needed to handle a system event to other objects.
  - A controller controls and coordinates the collaborating objects.
  - A controller does not do much of the actual work.
Controller Options

- Presentation objects (UI objects) should not be responsible for handling events
  - Decouple presentation layer from application processing layer. Why?

- System as controller
  - Referred to as a façade controller
  - Use when number of system events is not large
    - Large number of events can lead to a controller with low cohesion and high coupling

- Use case handlers
  - For each use case design a controller that handles the use case events
  - Use when number of system events is large
Bloated Controllers

- Signs of problematic design
  - Interface objects handle system events directly
  - Controller object handles many events
  - Controller object performs bulk of work needed to handle event.
  - Controller class has many attributes because of its many responsibilities.
General Guidelines

- Avoid dumb objects: objects that hold data and provide only get/set methods.
- Avoid “god” controllers: a “god’ controller is one that requests state information (e.g., using a get method) and uses the information to make decisions or perform calculations.
- Avoid coupling by having services above and beyond get/set services in interface of objects.
- A client should request an object to do something on its behalf, not request information about an object’s state.
Navigability

In a design model one can indicate that an object “knows about” another object it is linked to by using navigation arrows on associations.

- In UML 2.0 one can also explicitly show that one object does not know about the objects it is linked to.
The top pair AB shows a binary association with two navigable ends.
• The second pair CD shows a binary association with two non-navigable ends.
• The third pair EF shows a binary association with unspecified navigability.
• The fourth pair GH shows a binary association with one end navigable and the other non-navigable.
• The fifth pair IJ shows a binary association with one end navigable and the other having unspecified navigability.

The constructs in diagrams 1, 2, and 4 are new to UML 2.0 and thus are most likely not supported by UML tools as yet.
Interfaces

- An interface is a named collection of operations that represent the services provided by a class or component.
- Interfaces *do not* have attributes.
- Interfaces *do not* provide implementations for their operations.
Representing interfaces

TheftAlarm -- ISensor

activate()
read()

ProximitySensor

TheftAlarm -- ISensor

ProximitySensor
Design Class Model Example

**Customer**
- name
- address
+ findOrd()
  + getName()
  + getAddress()

**OrderManager**
+ findOrder()
  + sortOrders()
  + invoiceOrders()

**Order**
- status
  + invoiceOrder()
  + calcTotal()
  + getStatus()

**OrderItem**
- status
- qty
  + invoiceItem()
  + getDesc()
  + getStatus()

**ItemDesc**
- desc
  + getDesc()
  + getLocn()
Aggregation vs. inheritance

- Inheritance gives you fixed relationships between classes and objects.
- You *can’t* change the class of an object at runtime.
- There is a fundamental semantic error here. Is an Employee *just* their job or does an Employee *have* a job?

1. How can we promote john?
2. Can john have more than one job?
A better solution…

- Using aggregation we get the correct semantics:
  - An Employee has a Job

- With this more flexible model, Employees can have more than one Job

```
john:Employee
```

"just change this link at runtime to promote john!"
Multiple inheritance

- Sometimes a class may have more than one superclass
- The "is kind of" and substitutability principles must apply for all of the classifications
- Multiple inheritance is sometimes the most elegant way of modelling something. However:
  - Not all languages support it (e.g. Java)
  - It can always be replaced by single inheritance and delegation

In this example the AutoDialler sounds an alarm and rings the police when triggered - it is logically both a kind of Alarm and a kind of Dialler
Inheritance vs. interface realization

- With inheritance we get two things:
  - Interface – the public operations of the base classes
  - Implementation – the attributes, relationships, protected and private operations of the base classes

- With interface realization we get exactly one thing:
  - An interface – a set of public operations, attributes and relationships that have no implementation

Use inheritance when we want to *inherit implementation*. Use interface realization when we want to *define a contract*. 
Templates

Up to now, we have had to specify the types of all attributes, method returns and parameters. However, this can be a barrier to reuse.

Consider:

<table>
<thead>
<tr>
<th>BoundedIntArray</th>
<th>BoundedFloatArray</th>
<th>BoundedStringArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>size:int</td>
<td>size:int</td>
<td>size:int</td>
</tr>
<tr>
<td>elements[]:int</td>
<td>elements[]:float</td>
<td>elements[]:String</td>
</tr>
<tr>
<td>addElement( e:int ):void</td>
<td>addElement( e:float ):void</td>
<td>addElement( e:String ):void</td>
</tr>
<tr>
<td>getElement( i:int):int</td>
<td>getElement( i:int):float</td>
<td>getElement( i:int):String</td>
</tr>
</tbody>
</table>

spot the difference!

etc.
Template syntax

Template instantiation - the template parameters are bound to actual values to create new classes based on the template:

- If the type of a parameter is not specified then the parameter defaults to being a classifier
- Parameter names are local to the template – two templates do not have relationship to each other just because they use the same parameter names!
- Explicit binding is preferred as it allows named instantiations
Templates & multiple inheritance

Templates and multiple inheritance should only be used in design models where those features are available in the target language:

<table>
<thead>
<tr>
<th>Language</th>
<th>Templates</th>
<th>Multiple Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Java</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>C++</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Python</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**Nested classes**

- A nested class is a class defined inside another class
  - It is encapsulated inside the namespace of its containing class
  - Nested classes tend to be design artifacts
- Nested classes are only accessible by:
  - their containing class
  - objects of that their containing class
Modeling Behavior
Specifying behavior using the UML

- Class models describe objects and their relationships
  - Behavior can be specified in terms of operation pre and postconditions, but behavior is not the primary focus of a class model

- Behavioral models in the UML
  - State models: describe control aspects of a system – provides descriptions sequences of operations without regard for what the operation do.
  - Interaction models: describe interactions among objects
  - Activity models: description of a behavioral feature expressed in terms of sequences of steps.
How things happen in the UML

- An action is executed by an object
  - May change the contents of one or more variables or slots
  - If it is a communication ("messaging") action, it may:
    - Invoke an operation on another object
    - Send a signal to another object
    - Either one will eventually cause the execution of a procedure on the target object…
    - …which will cause other actions to be executed, etc.
  - Successor actions are executed
    - Determined either by control flow or data flow
Sequence Models
Overview

- Realizations of use cases can be expressed as interaction diagrams
  - Objects interact to accomplish use case goals.

- Object interactions are described in terms of
  - Collaborations: descriptions of object structures that support required behaviors
  - Interactions: descriptions of communication structures that support required behaviors

- Interaction diagrams allow one to view only the parts of a system involved in accomplishing use case goals
Key Definitions

- **Collaboration**: a collaboration defines the roles a set of objects play when performing a particular task, e.g. a task specified by a use case.

- **Interaction**: an interaction specifies a communication pattern to be performed by instances playing the roles of a collaboration.
Use case realization - design

- A collaboration of Design objects and classes that realise a use case

- A Design use case realization contains
  - Design object interaction diagrams
  - Links to class diagrams containing the participating Design classes
  - An explanatory text (flow)

- There is a trace between a requirements use case and a Design use case realization (sequence diagram)

- The Design use case realization specifies implementation decisions and implements the non-functional requirements

same as in Analysis, but now including implementation details
Sequence diagrams in design

sd AddCourse - design

Registrar

:Registrar

addCourse( "UML" )

:RegistrationUI

addCourse( "UML" )

:RegistrationManager

uml = Course("UML")

save(uml)

:DBManager

uml:Course

addCourse("UML")
Sequence Diagram: Basic Constructs

- **object symbol**
- **lifeline**
- **activation**
- **name** : Class
- **other**
- **stimulus**
  - name (…)
  - new (…)
  - delete
- **return**
- **create**
A More Complex Sequence Diagram

Frame and Name
Lifeline is an object

Plain asynchronous message
Interaction Occurrence

Combined Fragment
Combined Fragment Types

- Alternatives (alt)
  - choice of behaviors – at most one will execute
  - depends on the value of the guard ("else" guard supported)

- Option (opt)
  - Special case of alternative

- Break (break)
  - Represents an alternative that is executed instead of the remainder of the fragment (like a break in a loop)

- Parallel (par)
  - Concurrent (interleaved) sub-scenarios

- Negative (neg)
  - Identifies sequences that must not occur
Combined Fragment Types

- Critical Region *(region)*
  - Traces cannot be interleaved with events on any of the participating lifelines

- Assertion *(assert)*
  - Only valid continuation

- Loop *(loop)*
  - Optional guard: \([<\text{min}>, <\text{max}>, <\text{Boolean-expression}>]\)
  - No guard means no specified limit
Combined Fragments and Data

\[ sd \text{ GoHomeInvocation}(\text{Time invoc}) \]

**loop**

- **:ServiceUser**
- **:Clock**
- **:ServiceBase**
- **:ServiceTerminal**

**InvocationTime**

- **FindLocation**

**TransportSchedule**

- **ScheduleIntervalElapsed**
- **GetTransportSchedule**
- **TransportSchedule**

**loop**

**alt**

- \([\text{Now}>\text{invoc}+\text{last}]\)

- \([\text{pos}-\text{lastpos}>\text{dist}]\)

**FetchSchedule**

- **Guarding InteractionOperand with an InteractionConstraint**

**Choice**

**Operand Separator**
Different Kinds of Arrows

- Procedure call or other kind of nested flow of control
  - UML 1.4: Asynchronous
- Flat flow of control
- Explicit asynchronous flow of control
  - UML 1.4: Variant of async
- Return
Example: Different Arrows

Nested Flow

- teller
- : Order
- : Article

getValue

getName

price

Flat Flow

- caller
- exchange
- callee

lift receiver

dial tone

dial digit

dial digit

ringing tone

ringing signal

lift receiver
Referencing Interaction Diagrams

Interaction Frame

Lifeline is one object or a part

Interaction Occurrence

Interaction Frame

Interaction Occurrence

Interaction Frame

Interaction Occurrence

Interaction Frame
Decomposing Lifelines

Decomposition with global constructs corresponding to those on decomposed lifeline
Creating Interaction Diagrams: Basic Process

- Set interaction context
  - Use Case scenario
- Identify (controller) object responsible for handling the event initiating the interaction
- Identify objects that collaborate with the controller (collaborators).
- Specify message passing sequence that handles the initiating message.
Interaction Modeling Tips

- Set the context for the interaction.
- Include only those attributes of the objects that are relevant.
- Express the flow from left to right and from top to bottom.
- Put active objects to the left/top and passive ones to the right/bottom.
- Use sequence diagrams
  - to show the explicit ordering between the stimuli
  - when modeling real-time