Object Oriented Design

Kenneth M. Anderson
Lecture 20
CSCI 5828: Foundations of Software Engineering
Object-Oriented Design

- Traditional procedural systems separate data and procedures, and model these separately.
- Object orientation combines data and methods together into a cohesive whole.
  - data abstraction
- The purpose of Object-Oriented (OO) design is to define the classes (and their relationships) that are needed to build a system that meets the requirements contained in the SRS.
OO A&D

- OO techniques can be used in analysis (requirements) as well as design
  - The methods and notations are similar
- In OO analysis we model the problem domain, while in OO design we model the solution domain
- Often structures created during OO analysis are subsumed (reused, extended) in the structures produced by OO design
  - The line between OO analysis and OO design is blurry, as analysis structures will transition into model elements of the target system
Relationship of OO A&D
OO Concepts

- Encapsulation
  - grouping of related ideas into one unit which we can refer to by a single name
  - For example, methods, classes, packages
- Provides information hiding by restricting the external visibility of a unit’s information
- In OO A&D, the object is the unit of encapsulation
  - An object’s data is hidden behind the public interface (methods) of the object
OO Concepts...

- **State Retention**
  - the functions of function-oriented design do not retain state; an object, on the other hand, is aware of its past and maintains state across method invocations

- **Identity** – each object can be identified and treated as a distinct entity
  - very important issue, see lecture 10

- **Behavior** – state and methods together define the behavior of an object, or how an object responds to the messages passed to it
Classes – a class is a stencil from which objects are created; defines the structure and services of a “class” of objects. A class has
- An interface which defines which parts of an object can be accessed from outside
- A body that implements the operations
- Instance variables to hold object state

Objects and classes are different; a class is a type, an object is an instance
- State and identity is associated with objects
OO Concepts – access

- Operations in a class can be
  - Public: accessible from outside
  - Private: accessible only from within the class
  - Protected: accessible from within the class and from within subclasses
Inheritance

- Inheritance is unique to OO and not available in function-oriented languages/models.
- If class B inherits information from class A, it implicitly acquires the attributes and methods of A.
  - Attributes and methods of A are reused by B.
- When B inherits from A, B is the subclass or derived class and A is the base class or superclass.
Inheritance..

- A subclass B generally has a derived part (inherited from A) as well as new attributes (new instance variables or methods)
  - B’s specification only defines the new attributes
- This creates an “is-a” relationship
  - objects of type B are also objects of type A
Inheritance...

X
Base Class

"is–a"

Derived Part
(from X)

Incremental
Part
(new)

Y – Derived class
Inheritance…

- The inheritance relationship between classes forms a class hierarchy
  - In models, hierarchy should represent the natural relationships present in the problem domain
  - In a hierarchy, all the common features of a set of objects can be accumulated in a superclass

- This relationship is also known as a generalization-specialization relationship
  - since subclasses specialize (or extend) the more generic information contained in the superclass
Inheritance…

- There are several types of inheritance
  - Strict inheritance: a subclass uses all of the features of its parent class without modification
    - The subclass only adds new attributes or methods
  - Non-strict inheritance: a subclass may redefine features of the superclass or ignore features of the superclass

- Strict inheritance supports “is-a” cleanly and has fewer side effects
  - If a subclass redefines a method of the parent, it can potentially break the contract that the superclass offers its users
Inheritance…

- Single inheritance – a subclass inherits from only one superclass
  - Class hierarchy is a tree

- Multiple inheritance – a class inherits from more than one class
  - Can cause runtime conflicts
  - Repeated inheritance - a class inherits from a class but from two separate paths
Inheritance and Polymorphism

- Inheritance enables polymorphism, i.e. an object can be of different types
  - An object of type B is also an object of type A
- Hence an object has a static type and a dynamic type
  - Implications on type checking
  - Also brings dynamic binding of operations which allows writing of general code where operations do different things depending on the type
Module Level Concepts

- Basic modules are classes
- During OO design, a key activity is to specify the classes in the system being built
- In creating our design, we want it to be “correct” (i.e. cover its requirements)
  - But a design should also be “good” – efficient, modifiable, stable, …
- We can evaluate an OO design using three concepts
  - coupling, cohesion, and open-closed principle
Coupling

- In OO design, three types of coupling exists
  - interaction
  - component
  - inheritance
Coupling...

- Interaction coupling occurs when the methods of a class invoke methods of another class
  - this can’t be avoided, obviously…
  - but we want to ensure that an object’s public interface is used
    - a method of class A should NOT directly manipulate the attributes of another class B
  - Why?
Coupling...

- Component coupling – when a class A has variables of another class C
  - A has instance variables of type C
  - A has a method with a parameter of type C
  - A has a method with a local variable of type C

- When A is coupled with C, it is coupled with all subclasses of C as well
  - Component coupling will generally imply the presence of interaction coupling also
Coupling...

- Inheritance coupling – two classes are coupled if one is a subclass of the other
  - again, can’t be avoided, inheritance is a useful and desirable feature of OO approaches
  - however, a subclass should strive to only add features (attributes, methods) to its superclass
    - as opposed to modifying the features it inherits from its superclass
Cohesion

- Cohesion is an intramodule concept
- Focuses on why elements are together
  - Only elements tightly related should exist together in a module (class)
  - This gives a module a clear abstraction and makes it easier to understand
- Higher cohesion leads to lower coupling as many otherwise interacting elements are already contained in the module
- Goal is to have high cohesion in modules
- Three types of cohesion in OO design
  - method, class, and inheritance
Cohesion…

● Method cohesion
  ● A class should attempt to have highly cohesive methods, in which all of the elements within a method body help to implement a clearly specified function

● Class cohesion
  ● A class itself should be cohesive with each of its methods (and attributes) contributing to implement the class’s clearly specified role
Cohesion…

- Inheritance cohesion – focuses on why classes are together in a hierarchy
  - Two reasons for subclassing
    - generalization-specialization and reuse
    - The former occurs when the classes in the hierarchy are modeling true semantic (“is-a”) relationships found in the problem domain
    - The latter sometimes occurs when a pre-existing class does most of what you need but for a different part of the semantic space; the subclass may not participate in an “is-a” relationship; this should be avoided!
Open-closed Principle

- Principle: Classes should be open for extension but closed for modification
  - Behavior can be extended to accommodate new requirements, but existing code is not modified
    - allows addition of code, but not modification of existing code
  - Minimizes risk of having existing functionality stop working due to changes – a very important consideration while changing code
Open-closed Principle…

- In OO design, this principle is satisfied by using inheritance and polymorphism
  - Inheritance allows creating a new class to extend behavior without changing the original class
  - This can be used to support the open-closed principle
  - Consider example of a client object which interacts with a printer object for printing
Example

Client \(0..n\) \(0..n\) Printer1
Example..

- Client directly calls methods on Printer1
- If another printer is required
  - A new class Printer2 will be created
  - But the client will have to be modified if it wants to use this new class
- Alternative approach
  - Have Printer1 be a subclass of an abstract base class called Printer
  - Client is coded to access a variable of type Printer, which is instantiated to be an instance of the Printer1 class
  - When Printer2 comes along, it is made a subclass of Printer as well, and the client can use it without modification
Example…

Client 0..n Printer

Printer 1

Printer 2
Liskov’s Substitution Principle

- Principle: A program using an object o1 of base class C should remain unchanged if o1 is replaced by an object of a subclass of C
  - The open-closed principle allows the creation of hierarchies that intrinsically support this principle
Unified Modeling Language (UML) and Modeling

- UML is a graphical design notation useful for OO analysis and design
  - Provides nine types of diagrams to model both static and dynamic aspects of a software system
- UML is used by various OO design methodologies to capture decisions about the structure of a system under design
Modeling

- Modeling is used in many disciplines
- A model is a simplification of reality
  - “All models are wrong, some are useful”
- A good model includes those elements that have broad effect and omits minor elements
  - A model of a system is not the system!

- We’ve covered models at the beginning of the semester in the concurrency textbook
Modeling

- UML is used to create models of OO systems
- It contains notations to model both structural and behavioral aspects of these systems
  - Structure-related notations
    - class, object, package, use case, component, and deployment diagrams
  - Behavior-related notations
    - structure, collaboration, state, and activity diagrams
Class Diagrams

- The class diagram is a central piece of the design specification of an OO design. It specifies the
  - classes in a system
  - the associations between classes
    - including aggregation and composition relationships
  - the inheritance hierarchy
- We covered class diagrams back in lecture 10
Interaction Diagrams

- Class diagrams represent static structures
  - They do not model the behavior of a system
- Interaction diagrams are used to provide insight into a system’s dynamic behavior
  - Useful for showing, e.g., how the objects of a use case interact to achieve its functionality
  - Interaction is between objects, not classes
    - An object look likes a class, except its name is underlined
- Interaction diagrams come in two (mostly equivalent) styles
  - Collaboration diagram
  - Sequence diagram
Sequence Diagram

- Objects participating in an interaction are shown at the top
  - For each object a vertical bar represents its lifeline
  - A message from one object to another is represented as a labeled arrow
  - Messages can be guarded (similar to boolean guards in FSP)
- The ordering of messages is captured along a sequence diagram’s vertical axis
Example – sequence diag.
Collaboration diagram

- Also shows how objects interact
- Instead of a timeline, the diagram shows the instantiation of associations between classes at run-time
  - The ordering of a set of messages is captured by numbering them
Example – collaboration diag

1: getStudentDetails()
2: getGrades()
3: getCourseDetails()
4: ReturnDetails()
5: ReturnCourseDetails()
6: StudentDetails
Other Diagrams

- State diagrams (Labeled Transition Systems)
- Activity diagrams (from Scott Ambler’s website)
OO Design Methodologies

- Many OO A&D methodologies have been proposed
- Basic goal is to identify classes, understand their behavior, and relationships
  - Different UML models are used for this
**Basic steps (note: different from text book)**

- Step 1: Analyze use cases
- Step 2: Create activity diagrams for each use case
- Step 3: Create class diagram based on 1 and 2
- Step 4: Create interaction diagrams for activities contained in diagrams created in step 2
- Step 5: Create state diagrams for classes created in step 3
- Step 6: Iterate; each step above will reveal information about the other models that will need to be updated
  - for instance, services specified on objects in a sequence diagram, have to be added to those objects’ classes in the class diagram
Restaurant example: Initial classes

Supply Handling

SupplyOrder

Supplies

Restaurant

Menu

Order

Bill
Note: this is not pure UML notation; see Lecture 10 for Additional details
Restaurant example: sequence diagram
Metrics

- OO metrics focus on identifying the complexity of classes in an OO design
  - Weighted Methods per Class
  - Depth of Inheritance Tree
  - Number of Children
  - Coupling Between Classes
  - Response for a Class
  - Lack of Cohesion in Methods
Weighted Methods Per Class

- The complexity of a class depends on the number of methods it has and the complexity of those methods
  - For a class with methods $M_1$, $M_2$, ..., $M_n$, determine a complexity value for each method, $c_1$, $c_2$, ..., $c_n$
    - using any metric that estimates complexity for functions (estimated size, interface complexity, data flow complexity, etc.)
  - $\text{WMC} = \sum c_i$; this metric has been shown to have a reasonable correlation with fault proneness
Metrics...

- Depth of Inheritance Tree
  - DIT of class C is depth from the root class
  - DIT is significant in predicting fault proneness
    - basic idea: the deeper in the tree, the more methods a particular class has, making it harder to change

- Number of Children
  - Immediate number of subclasses of C
  - Gives a sense of reuse of C’s features
  - Most classes have a NOC of 0; one study showed, however, that classes with a high NOC had a tendency to be less fault prone than others
Metrics…

- Coupling between classes
  - Number of classes to which this class is coupled
  - Two classes are coupled if methods of one use methods or attributes of another
  - A study has shown that the CBC metric is significant in predicting the fault proneness of classes
Metrics...

- **Response for a Class**
  - CBC metric does not quantify the strength of the connections its class has with other classes (it only counts them)
  - The response for a class metric attempts to quantify this by capturing the total number of methods that can be invoked from an object of this class
  - Thus even if a class has a CBC of “1”, its RFC value may be much higher
  - A study has shown that the higher a class’s RFC value is, the larger the probability that class will contain defects
Metrics…

- Lack of cohesion in methods
  - Two methods form a cohesive pair if they access common variables (they form a non-cohesive pair if they have no common variables)
  - LCOM is the number of method pairs that are non-cohesive minus the number of cohesive pairs
- Highly cohesive classes have small LCOM values
  - A high LCOM value indicates that the class is trying to do too many things and its features should be partitioned into different classes
- However, a study found that this metric is NOT useful in predicting the fault proneness of a class
Metrics

- Note: the study referenced in the previous slides was published in the following paper
Summary

- OO design is a newer paradigm that is replacing function-oriented design techniques
- OO A&D combines both data and methods into cohesive units (classes)
- UML is a notation that is often used to model OO systems
  - It provides various diagrams for modeling a system’s structure, dynamic behavior, states, architecture, etc.
- Creating an OO design is an iterative process based on applying the knowledge stored in a system’s use cases
- Several OO metrics exist that are useful in predicting the fault proneness of a class