Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr
http://www.yildiz.edu.tr/~naydin

1

Class and Method Design

- The most important step of the design phase
  - designing the individual classes and methods.
- OO systems can be quite complex,
  - analysts need to create instructions and guidelines for programmers that clearly describe what the system must do.
- A set of criteria, activities, and techniques used to design classes and methods will be presented.
  - Together they are used to ensure the object-oriented design communicates how the system needs to be coded.

Objectives

- Become familiar with coupling, cohesion, and connascence.
- Be able to specify, restructure, and optimize object designs.
- Be able to identify the reuse of predefined classes, libraries, frameworks, and components.
- Be able to specify constraints and contracts.
- Be able to create a method specification.

3

- Class and Method design is where all of the work actually gets done during the design phase.
- No matter which layer you are focusing on, the classes, which will be used to create the system objects, must be designed.
- Some people believe that with reusable class libraries and off-the-shelf components, this type of low-level, or detailed, design is a waste of time and that we should jump immediately into the “real” work:
  - coding the system.

4

- However, low-level or detailed design is critical despite the use of libraries and components.
- Detailed design is still very important for two reasons.
  - First,
    - even preexisting classes and components needs to be understood, organized, and pieced together.
  - Second,
    - it is still common for the project team to have to write some code (if not all) and produce original classes that support the application logic of the system.

5

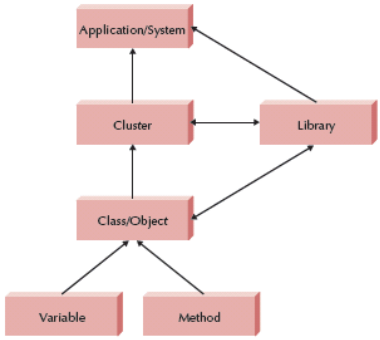
- Jumping right into coding will guarantee results that can be disastrous.
- For example,
  - even though the use of layers can simplify the individual classes, they can increase the complexity of the interactions between them.
  - As such,
    - if the classes are not designed carefully, the resulting system can be very inefficient
    - the instances of the classes (i.e., the objects) will not be capable of communicating with each other,
      - which, of course, will cause the system not to work properly.

6

- in an OO system, changes can take place at different levels of abstraction. These levels include
  - variable, Method, Class/Object, Cluster,Library, and Application/System levels.
- The changes that take place at one level can impact other levels
  - e.g., changes to a class can affect the cluster level, which can affect both the system level and the library level,
    - which in turn can cause changes back down at the class level.
- Finally, changes can be occurring at different levels at the same time.

7

Levels of Abstraction in OO Systems



8

- The detailed design of the individual classes and methods is fairly straightforward
- The interactions among the objects on the problem domain layer have been designed, in some detail, in the analysis phase
- As far as the other layers go (system architecture, human computer interaction, and data management), they will be highly dependent on the problem domain layer.
- Therefore, if we get the problem domain classes designed correctly, the design of the classes on the other layers will fall into place.

9

REVISITING THE BASIC CHARACTERISTICS OF OBJECT-ORIENTATION

Object-oriented systems can be traced back to the Simula and Smalltalk programming languages.
However,
until the increase in processor power and the
decrease in processor cost that occurred in the 1980s,
- object-oriented approaches were not practical.

Classes, Objects,Methods, and Messages

- The basic building block of the system:
  - the object.
- Objects
  - instances of classes
    - used as templates to define objects.
- A class
  - defines both the data and processes that each object contains.
- Each object has attributes
  - describing data about the object.

11

Classes, Objects,Methods, and Messages

- Objects have state
  - defined by the value of its attributes and its relationships with other objects at a particular point in time.
- each object has methods
- specifying what processes the object can perform.
- In order to get an object to perform a method,
  - a message is sent to the object.
- A message
  - a function or procedure call from one object to another object.

12

Encapsulation and Information Hiding

- Encapsulation
  - the mechanism that combines the processes and data into a single object.
- Information hiding
  - suggests only the information required to use an object be available outside the object.
  - Exactly how the object stores data or performs methods is not relevant, as long as the object functions correctly.
- All that is required to use an object is the set of methods and the messages needed to be sent to trigger them.
- The only communication between objects should be through an object's methods.

13

Polymorphism and Dynamic Binding

- Polymorphism
  - means having the ability to take several forms.
- By supporting polymorphism, object-oriented systems can send the same message to a set of objects, which can be interpreted differently by different classes of objects.
- Based on encapsulation and information hiding,
  - an object does not have to be concerned with *how* something is done when using other objects.
- It simply sends a message to an object and that object determines how to interpret the message.
  - This is accomplished through the use of dynamic binding.

14

Polymorphism and Dynamic Binding

- Dynamic binding
- refers to the ability of OO systems to defer the data typing of objects to run time.
- For example,
  - imagine that you have an array of type employee that contains instances of hourly employees and salaried employees.
  - Both of these types of employees implement a "compute pay" method.
  - An object can send the message to each instance contained in the array to compute the pay for that individual instance.

15

Polymorphism and Dynamic Binding

  - Depending on whether the instance is an hourly employee or a salaried employee, a different method would be executed.
  - The specific method is chosen at run time.
  - With this ability, individual classes are easier to understand.
- However, the specific level of support for polymorphism and dynamic binding is language specific.
- Most object-oriented programming languages support dynamic binding of methods, and some support dynamic binding of attributes.
  - As such, it is important to know what object-oriented programming language is going to be used.

16

Inheritance

- Inheritance
  - allows developers to define classes incrementally by reusing classes defined previously as the basis for new classes.
- Although we could define each class separately, it might be simpler to define one general superclass that contains the data and methods needed by the subclasses, and then have these classes inherit the properties of the superclass.
- Subclasses inherit the appropriate attributes and methods from the superclasses "above" them.
- Inheritance makes it simpler to define classes.

17

Inheritance

- There have been many different types of inheritance mechanisms associated with OO systems.
- The most common inheritance mechanisms include different forms of single and multiple inheritance.
- Single inheritance
  - allows a subclass to have only a single parent class.
- Currently, all OO methodologies, databases, and programming languages permit extending the definition of the superclass through single inheritance.

18

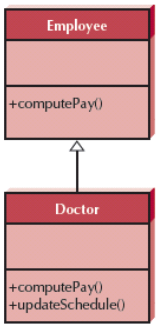
Inheritance

- Some OO methodologies, databases, and programming languages allow a subclass to redefine some or all of the attributes and/or methods of its superclass.
- With redefinition capabilities, it is possible to introduce an inheritance conflict
  - an attribute [or method] of a subclass with the same name as an attribute [or method] of a superclass.
- For example in next figure, Doctor is a subclass of Employee.
  - Both have methods named computePay().
- This causes an inheritance conflict.

19

Inheritance

- When the definition of a superclass is modified, all of its subclasses are affected.
- This may introduce additional inheritance conflicts in one (or more) of the superclass's subclasses.
- For example in the Figure,
  - Employee could be modified to include an additional method, updateSchedule().
  - This would add another inheritance conflict between Employee and Doctor.
  - Therefore, developers must be aware of the effects of the modification not only in the superclass, but also in each subclass that inherits the modification



20

Inheritance

- through redefinition capabilities, it is possible for a programmer to arbitrarily cancel the inheritance of methods
  - by placing stubs in the subclass that will override the definition of the inherited method.
    - a stub is the minimal definition of a method to prevent syntax errors occurring
- If the cancellation of methods is necessary for the correct definition of the subclass,
  - then it is likely that the subclass has been misclassified
    - i.e., it is inheriting from the wrong superclass.

21

Inheritance

- inheritance conflicts and redefinition can cause all kinds of problems with interpreting the final design and implementation.
- most inheritance conflicts are due to poor classification of the subclass in the inheritance hierarchy
  - i.e., the generalization A-Kind-Of semantics are violated,
- or the actual inheritance mechanism violates the encapsulation principle
  - i.e., subclasses are capable of directly addressing the attributes or methods of a superclass.

22

Inheritance

- To address these issues, Jim Rumbaugh, and his colleagues, suggested the following guidelines:
  - Do not redefine query operations.
  - Methods that redefine inherited ones should only restrict the semantics of the inherited ones.
  - The underlying semantics of the inherited method should never be changed.
  - The signature (argument list) of the inherited method should never be changed.

23

Inheritance

- Many existing OO programming languages violate these guidelines.
- When it comes to implementing the design, different OO programming languages address inheritance conflicts differently.
- Therefore, it is important at this point in the development of the system to know what the programming language that you are going to use supports.

24

Inheritance

- With multiple inheritance, a subclass may inherit from more than one superclass.
- In this situation, the types of inheritance conflicts are multiplied.
- In addition to the possibility of having an inheritance conflict between the subclass and one (or more) of its superclasses,
  - it is now possible to have conflicts between two (or more) superclasses.
- In this latter case, there are three different types of additional inheritance conflicts that can occur:

25

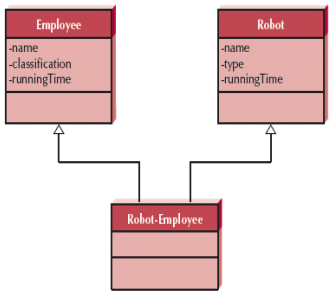
Inheritance

1. Two inherited attributes (or methods) have the same name and semantics.
2. Two inherited attributes (or methods) have different names but with identical semantics
   - i.e., they are synonyms.
3. Two inherited attributes (or methods) have the same name but with different semantics
   - i.e., they are homonyms.
   - This also violates the proper use of polymorphism.

26

Additional Inheritance Conflicts with Multiple Inheritance

- In the Figure,
  - Robot-Employee is a subclass of both Employee and Robot.
- Employee and Robot conflict with the attribute name.
- Which one should Robot-Employee inherit?
- It is also possible that Employee and Robot could have a semantic conflict on the classification and type attributes if they are synonyms.



27

Design Criteria

When considering the design of an OO system, there is a set of criteria that can be used to determine whether the design is a good one or a bad one.

These criteria include

- coupling, cohesion, and connascence.

Coupling

- refers to how interdependent or interrelated the modules (classes, objects, and methods) are in a system.
  - The higher the interdependency, the more likely changes in part of a design can cause changes to be required in other parts of the design.
- For OO systems, Coad and Yourdon* identified two types of coupling to consider:
  - interaction
  - inheritance.

* Peter Coad and Edward Yourdon, *Object-Oriented Design (Englewood Cliffs, NJ: Yourdon Press, 1991)*

29

Coupling

- Interaction coupling
  - deals with the coupling among methods and objects through message passing.
- Law of Demeter* states that
  - an object should only send messages to one of the following:
    - itself,
    - an object that is contained in an attribute of the object
      - or one of its superclasses,
    - an object that is passed as a parameter to the method,
    - an object that is created by the method,
    - an object that is stored in a global variable.

* Karl J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns (Boston, MA: PWS Publishing, 1996)*

30

Coupling

- In each case, interaction coupling is increased.
- For example,
  - the coupling increases between the objects
    - if the calling method passes attributes to the called method,
    - or if the calling method depends on the value being returned by the called method.
- There are six types of interaction coupling,
  - each falling on different parts of a good-to bad continuum.
  - They range from no direct coupling (Good) up to content coupling (Bad).
- Following figure presents the different types of interaction coupling

31

Types of Interaction Coupling

Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite, (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

32

Coupling

- In general, interaction coupling should be minimized.
  - one possible exception is that non–problem domain classes must be coupled to their corresponding problem domain classes.
  - For example,
    - a report object (on the HCI layer) that displays the contents of an employee object (on the problem domain layer) will be dependent on the employee object.
    - In this case, for optimization purposes, the report class may be even content or pathologically coupled to the employee class.
  - However, problem domain classes should never be coupled to non–problem domain classes.

33

Coupling

- Inheritance coupling,
  - deals with how tightly coupled the classes are in an inheritance hierarchy.
- Most authors tend to say simply that this type of coupling is desirable.
- However, depending on the issues raised previously with inheritance
  - inheritance conflicts, redefinition capabilities, and dynamic binding

  a high level of inheritance coupling may not be a good thing.

34

Coupling

- For example,
  - should a method defined in a subclass be allowed to call a method defined in one of its superclasses?
  - should a method defined in a subclass refer to an attribute defined in one of its superclasses?
  - can a method defined in an abstract superclass depend on its subclasses to define a method or attribute on which it is dependent?
- Depending on the object-oriented programming language being used, all of these options are possible.

35

Coupling

- Snyder* has pointed out that
  - most problems with inheritance is the ability within the OO programming languages to violate the encapsulation and information hiding principles.
- Therefore, knowledge of which OO programming language is to be used is crucial.
- From a design perspective,
  - the developer will need to optimize the trade-offs of violating the encapsulation and information hiding principles and increasing the desirable coupling between subclasses and its superclasses.

* Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," in: N.Meyrowitz, Ed., *OOPSLA '86 Conference Proceedings, ACM SigPlan Notices, 21 (11) (November 1986)*

36

Coupling

- The best way to solve this conundrum
  - to ensure that inheritance is used only to support
    - generalization/specialization (A-Kind-Of) semantics
    - the principle of substitutability.
- All other uses should be avoided.

37

Cohesion

- refers to how single-minded a module (class, object, or method) is within a system.
  - A class or object should only represent one thing,
  - A method should only solve a single task.
- Three general types of cohesion have been identified by Coad and Yourdon* for OO systems:
  - method,
  - class,
  - generalization/specialization,

* Coad and Yourdon, *Object-Oriented Design*

38

Cohesion

- Method cohesion
  - addresses the cohesion within an individual method.
    - i.e., how singleminded is a method
  - Methods should do one and only one thing.
- There are seven types of method cohesion that have been identified
  - illustrated in next slide
- They range from functional cohesion (Good) down to coincidental cohesion (Bad).
- In general, method cohesion should be maximized.

39

Types of Method Cohesion

Level	Type	Description
Good	Functional	A method performs a single problem-related task (e.g., Calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is send by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.

40

Cohesion

- Class cohesion
  - level of cohesion among the attributes and methods of a class
    - *i.e.*, how single-minded is a class.
  - A class should only represent one thing, such as an employee, a department, or an order.
- All attributes and methods contained in a class should be required for the class to represent the thing.
- For example,
  - an employee class should have attributes that deal with a
    - social security number, last name, first names, middle initial, addresses, and benefits,
  - but it should not have attributes like
    - door, engine, or hood.

41

Cohesion

- A class should have only the attributes and methods necessary to fully define instances for the problem at hand.
- In this case, we have ideal class cohesion.
  - Glenford Meyers* suggested that a cohesive class should:
    - contain multiple methods that are visible outside of the class and that each visible method only performs a single function,
    - have methods that only refer to attributes or other methods defined with the class or its superclass(es),
    - not have any control-flow couplings between its methods.

* Glenford J. Myers, Composite/Structured Design [New York, NY: Van Nostrand Reinhold, 1978]

42

Cohesion

- Page-Jones* has identified three less than desirable types of class cohesion:
  - mixed instance,
  - mixed-domain,
  - mixed-role

  which are summarised in next slide.
- An individual class can have a mixture of any of the three types.

* Meilir Page-Jones, *Fundamentals of Object-Oriented Design in UML (Reading, MA: Addison-Wesley, 2000)*

43

Types of Class Cohesion

Level	Type	Description
Good	Ideal	The class has none of the mixed cohesions.
↓	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) have nothing to do with the underlying semantics of the class.
↓	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.

44

Connascence

- generalizes the ideas of cohesion and coupling, and it combines them with the arguments for encapsulation.
- To accomplish this, three levels of encapsulation have been identified.
  - Level-0 encapsulation
    - refers to the amount of encapsulation realized in an individual line of code,
  - Level-1 encapsulation
    - the level of encapsulation attained by combining lines of code into a method,
  - Level-2 encapsulation
    - achieved by creating classes that contain both methods and attributes.

45

Connascence

- Method cohesion and interaction coupling primarily address level-1 encapsulation.
- Class cohesion, generalization/specialization cohesion, and inheritance coupling only address level-2 encapsulation.
- Connascence, as a generalization of cohesion and coupling, addresses both level-1 and level-2 encapsulation.
- Connascence means to be born together.
  - From an OO design perspective, it means that two modules (classes or methods) are so intertwined, that
    - if you make a change in one, it is likely that a change in the other will be required.

46

Connascence

- On the surface, this is very similar to coupling, and as such should be minimized.
- However,when you combine it with the encapsulation levels, it is not quite as simple as that.
- In this case, you want to
  - minimize overall connascence by eliminating any unnecessary connascence throughout the system,
  - minimize connascence across any encapsulation boundaries, such as method boundaries and class boundaries,
  - maximize connascence within any encapsulation boundary.

47

Connascence

- Based on these guidelines,
  - a subclass should never directly access any hidden attribute or method of a superclass .
  - If direct access to the non-visible attributes and methods of a superclass by its subclass is allowed and a modification to the superclass is made,
    - then due to the connascence between the subclass and its superclass, it is likely that a modification to the subclass also will be required.
  - Practically speaking, you should maximize the cohesion (connascence) within an encapsulation boundary and minimize the coupling (connascence) between the encapsulation boundaries.
- There are many possible types of connascence.
  - Following slide describes five of the types.

48

Types of Connascence

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.

Source: Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

49

Object Design Activities

The design activities for classes and methods are an extension of the analysis and evolution activities presented previously.

The expanded descriptions are created through the activities that take place during the detailed design of the classes and methods.

- The activities used to design classes and methods include
  - additional specification of the current model,
  - identifying opportunities for reuse,
  - restructuring the design,
  - optimizing the design,
  - mapping the problem domain classes to an implementation language.
- Any changes made to a class on one layer can cause the classes on the other layers that are coupled to it to be modified also.

51

Additional Specification

- At this point in the development of the system, it is crucial to review the current set of structural and behavioral models.
- Ensure the classes are both necessary and sufficient for the problem
  - To do this,we need to be sure that there are no missing attributes or methods and no extra or unused attributes or methods in each class.
- Finalize the visibility of the attributes and methods of each class
  - Depending on the object-oriented programming language used, this could be predetermined

52

Additional Specification

- Determine the signature of every method of each class
  - The signature of a method comprises three parts:
    - the name of the method,
    - the parameters or arguments that must be passed to the method,
    - the type of value that the method will return to the calling method.
- Define constraints to be preserved by objects
  - There are three different types of constraints:
    - pre-conditions,
    - post-conditions,
    - invariants.
- These are captured in the form of contracts and assertions added to CRC cards and class diagrams.

53

Additional Specification

- We also must decide how to handle a violation of a constraint.
  - Should the system simply abort?
  - Should the system automatically undo the change that caused the violation?
  - Should the system let the end user determine the approach to correct the violation?
- The designer must design the errors that the system is expected to handle.
  - It is best to not leave these types of problems for the programmer to solve.
- Violations of a constraint are known as exceptions in languages, such as C++ and Java.

54

Identifying Opportunities for Reuse

- In the design phase, in addition to using analysis patterns, there are opportunities for using
  - design patterns,
  - frameworks,
  - libraries,
  - components.
- The opportunities will vary depending on which layer is being reviewed.
  - For example,
    - it is doubtful that a class library will be of much help on the problem domain layer,
    - but an appropriate class library could be of great help on the foundation layer.

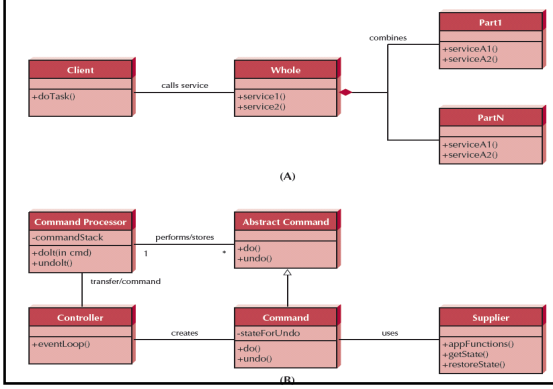
55

Identifying Opportunities for Reuse

- Design patterns
- useful grouping of collaborating classes that provide a solution to a commonly occurring problem.
- useful in solving "a general design problem in a particular context."
- For example,
  - a useful pattern is the Whole-Part pattern (see the next slide, Part A).
  - The Whole-Part pattern explicitly supports the Aggregation and Composition relationships within the UML.
  - Another useful design pattern is the Command Processor pattern (see the next slide, Part B).

56

Sample Design Patterns



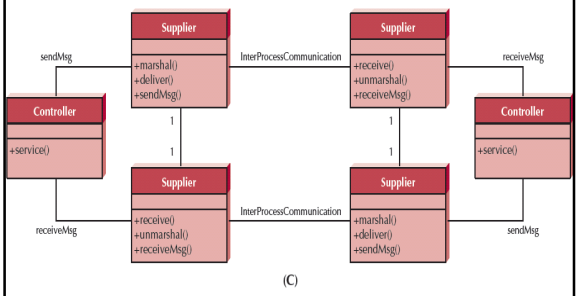
57

Identifying Opportunities for Reuse

- The primary purpose of the Command Processor pattern
  - to force the designer to explicitly separate the interface to an object (Command) from the actual execution of the operation (Supplier) behind the interface.
- Some of the design patterns support different physical architectures.
- For example,
  - the Forwarder-Receiver pattern (see the next slide, Part C) supports a peer-to-peer architecture.
- Many design patterns are available in C++ or Java source code.

58

Sample Design Patterns



59

Identifying Opportunities for Reuse

- A framework
  - composed of a set of implemented classes that can be can be used as a basis for implementing an application.
- For example,
  - there are frameworks available for CORBA and DCOM on which you could base the implementation of part of the physical architecture layer.
- Most frameworks allow you to create subclasses to inherit from classes in the framework.
  - There are object-persistence frameworks that can be purchased and used to add persistence to the problem domain classes, which would be helpful on the data management layer.

60

Identifying Opportunities for Reuse

- A class library
- similar to a framework in that you typically have a set of implemented classes that were designed for reuse.
- A typical class library could be purchased to support
  - numerical or statistical processing,
  - file management (data management layer),
  - user interface development (HCI layer).
- In some cases, you will create instances of classes contained in the class library and in other cases you will extend classes in the class library by creating subclasses based on them.

61

Identifying Opportunities for Reuse

- If you use inheritance to reuse the classes in the class library,
  - you will run into all of the issues dealing with inheritance coupling and connascence.
- If you directly instantiate classes in the class library,
  - you will create a dependency between your object and the library object based on the signatures of the methods in the library object.
- This will increase the interaction coupling between the class library object and your object.

62

Identifying Opportunities for Reuse

- A component
  - a self-contained, encapsulated piece of software that can be "plugged" into a system to provide a specific set of required functionality.
- Today, there are many components available for purchase that have been implemented using ActiveX or JavaBean technologies.
- A component has a well-defined Application Program Interface (API).
  - The API is essentially a set of method interfaces to the objects contained in the component.

63

Identifying Opportunities for Reuse

- The internal workings of the component are hidden behind the API.
- Components can be implemented using class libraries and frameworks.
- However, components also can be used to implement frameworks.
- Unless, the API changes between versions of the component, upgrading to a new version normally will require only linking the component back into the application.
  - As such, recompilation typically is not required.

64

Identifying Opportunities for Reuse

- Which of these approaches should you use?
  - It depends on what you are trying to build.
- frameworks are used mostly to aid in developing objects on the physical architecture, human computer interaction, or data management layers
- components are used primarily to simplify the development of objects on the problem domain and human computer interaction layers
- class libraries are used to develop frameworks and components and to support the foundation layer

65

Restructuring the Design

- Once the individual classes and methods have been specified, and the class libraries, frameworks, and components have been incorporated into the evolving design,
  - factoring should be used to restructure the design.
    - Factoring is the process of separating out aspects of a method or class into a new method or class to simplify the overall design.
  - For example,
    - when reviewing a set of classes on a particular layer, you might discover that a subset of them shares a similar definition.
    - In that case, it may be useful to factor out the similarities and create a new class.
    - Based on the issues related to cohesion, coupling, and connascence, the new class may be related to the old classes via inheritance (generalization) or through an aggregation or association relationship.

66

Restructuring the Design

- Another process that is useful to restructure the evolving design is normalization.
- Normalization can be useful at times to identify potential classes that are missing from the design.
- Also, related to normalization is the requirement to implement the actual association and aggregation relationships as attributes.
- Virtually no OO programming language differentiates between attributes and association and aggregation relationships.
  - Therefore, all association and aggregation relationships must be converted to attributes in the classes.

67

Optimizing the Design

- optimizations that can be used to create a more efficient design:

✓ review the access paths between objects.
- In some cases, a message from one object to another may have a long path to traverse
  - i.e., it goes through many objects.
- If the path is long, and the message is sent frequently, a redundant path should be considered.
- Adding an attribute to the calling object that will store a direct connection to the object at the end of the path can accomplish this.

68

Optimizing the Design

✓ review each attribute of each class.
- Which methods use the attributes and which objects use the methods should be determined.
- If the only methods that use an attribute are read and update methods, and only instances of a single class send messages to read and update the attribute,
  - then the attribute may belong with the calling class instead of the called class.
- Moving the attribute to the calling class will substantially speed up the system

69

Optimizing the Design

✓ review the direct and indirect fan-out of each method.
- Fan-out refers to the number of messages sent by a method.
- The direct fan-out is the number of messages sent by the method itself,
- The indirect fan-out also includes the number of messages sent by the methods called by the other methods in a message tree.
- If the fan-out of a method is high relative to the other methods in the system, the method should be optimized.
  - One way to do this is to consider adding an index to the attributes used to send the messages to the objects in the message tree.

70

Optimizing the Design

✓ look at the execution order of the statements in often-used methods.
- In some cases, it may be possible to rearrange some of the statements to be more efficient.
- For example,
- if it is known, based on the objects in the system, that a search routine can be narrowed by searching on one attribute before another one,
  - then the search algorithm should be optimized by forcing it to always search in a predefined order.

71

Optimizing the Design

✓ avoid recomputation by creating a derived attribute (active value), e.g., a total that will store the value of the computation.
- This is also known as caching computational results.
- This can be accomplished by adding a trigger to the attributes contained in the computation.
  - This would require a recomputation to take place only when one of the attributes that go into the computation is changed.
- Another approach would be to mark the derived attribute, and delay the recomputation until the next time the derived attribute is accessed.
  - This delays the recomputation as long as possible.
  - In this manner, a computation does not occur unless it has to occur.
    - Otherwise, every time a derived attribute needs to be accessed, a computation would be required.

72