Module 1

Examining Object-Oriented Concepts and Terminology

Objectives

Upon completion of this module, you should be able to:

- Describe the important object-oriented (OO) concepts
- Describe the fundamental OO terminology

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Rumbaugh, James, Jacobson Ivor, Booch Grady. *The Unified Modeling Language Reference Manual (2nd ed).* Addison-Wesley, 2004.
- Larman, Craig. *Applying UML and Patterns (3rd ed).* Prentice Hall, 2005.
- Fowler, Martin. *UML Distilled (2nd ed).* Addison-Wesley, 2000.
- Booch, Grady. *Object-Oriented Analysis and Design with Applications (2nd ed).* The Benjamin/Cummins Publishing Company, Inc., Redwood City, 1994.
- Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, 2000.
- Meyer, Bertrand. *Object-Oriented Software Construction (2nd ed)*. Prentice Hall, Upper Saddle River, 1997.
- Pressman, Roger. *Software Engineering A Practitioner's Approach, Fifth edition*. McGraw-Hill, 2001.
- Knoernschild, Kirk. *Java Design (Objects, UML, and Process)*. Reading: Addison Wesley Longman, Inc., 2002.

Examining Object Orientation

OO concepts affect the whole development process starting with gathering requirements, domain modeling, design, and implementation. The following are considerations:

- Humans think in terms of nouns (objects) and verbs (behaviors of objects).

  Human languages and thought patterns are formed around objects in the world, such as people, places, and things, and around the actions that manipulate the world of objects. OO principles put our focus on objects and what actions these objects perform (their so called responsibilities). Objects can also collaborate with other objects to perform an action.

- With OOSD, both problem and solution domains are modeled using OO concepts.

  In OOSD, the development team is asked to create a software system that supports a *business process*. These processes are modeled with objects, responsibilities, and collaborations, because these notions fit our mental model of the *business process*.

Note – By business process, we mean a process that can have a wider scope than the software system.

- The *Unified Modeling Language* (UML) is a de facto standard for modeling OO software.

  Although not essential, it is useful to be able to visualize OO concepts in terms of views and models. As a consequence, a de facto graphical notation called the UML has evolved. The UML was designed to model many OO aspects, including *objects, responsibilities, and collaborations*. Therefore, the UML is a good tool for visualizing our mental models.

- OO languages bring the implementation closer to the language of mental models. The UML is a good bridge between mental models and implementation.

  Various OO languages provide language syntax (such as class definitions) and semantic structures (such as runtime objects) based on OO principles. The UML has mappings for many mainstream OO languages, such as the Java programming language and C++. Therefore, the UML is a good bridge to implementation language

and the implementation language enables you to implement the system in a manner closer to the mental models of the problem and solution space.

"Software systems perform certain actions on objects of certain types; to obtain flexible and reusable systems, it is better to base their structure on the object types than on the actions." (Meyer page vi)

OO concepts affect the following issues:

- Software complexity
- Software decomposition
- Software costs

Software Complexity

Complex systems have the following characteristics:

- They have a *hierarchical structure*.

  In procedural programming, a complex system is decomposed into a hierarchy of subroutines. This structure tends to be brittle.

  In OO programming, a complex system is decomposed into a hierarchy of collaborating objects. This structure tends to be more flexible and extensible because you are recomposing different collaborations between existing objects to solve new problems.

- The choice of *which components are primitive* in the system is arbitrary.

  The choice of what is primitive in a system depends on the perspective of the observer. This principle provides the designer with a handle on system complexity by shielding the complex details of one layer behind an abstraction (a method or object) on a higher level.

  This principle also affects modeling. You do not have to show all of the system complexity in one view; rather, you can create different views at different levels of abstraction.

- A system can be split by intra- and inter-component relationships. This *separation of concerns* enables you to study each part in relative isolation.

  Separation of concerns is an important principle in designing and building complex systems. Using this principle, an object should focus its activity on one or a few simple concerns. The client of that object does not need to know how the object does its work.

  For example, a UI component deals with a user's actions, such as processing a click to a button or verifying that a data field has a legitimate value. A business logic component does not need to know *how* the data is entered when it processes some operation.

- Complex systems are usually composed of only a *few types of components in various combinations*.

  Like organic systems, software systems tend to have few fundamental types of components that are used to build the system. In animal biology, these structures include skin, bones, muscle, neurons, and so on. In computer systems, these structures include user interface, control (workflow), and entity components.

- A successful, complex system invariably *evolves from a simple working system*.

  Successful complex systems tend to be grown rather than created. A complex system usually starts small with a few domain objects and adds complexity by extending these domain objects and rearranging them in new ways.

  For example, the web application for Amazon.com started by selling books, and then started selling CDs and videos, and providing reader reviews, promotions, and so on.

Software Decomposition

In procedural programming, tasks are decomposed into a hierarchy of procedures. Figure 1-1 shows an example hierarchy of procedures for managing human resources (HR) tasks, such as giving a promotion or a raise.

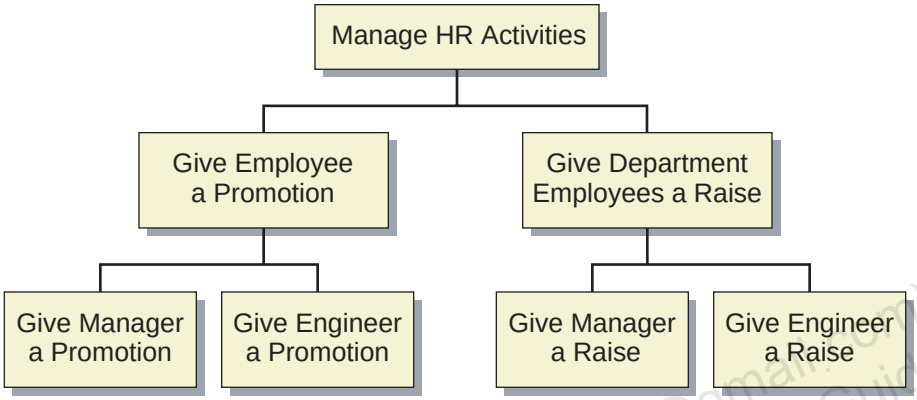


Figure 1-1 Functional Decomposition

This decomposition seems natural until you try to extend the program. For example, how would you support a new type of employee, such as a lab technician? If this type of employee requires unique actions for "give a raise" and "give a promotion," then the procedural hierarchy must be modified in several places. This is the definition of brittle.

Alternatively, using OO principles, a system is composed of a hierarchy of collaborating objects. Figure 1-2 shows an example object hierarchy. On the right, the department object delegates to the employee objects to perform a "give a raise" action. Each employee object has a type defined by the class hierarchy on the left.

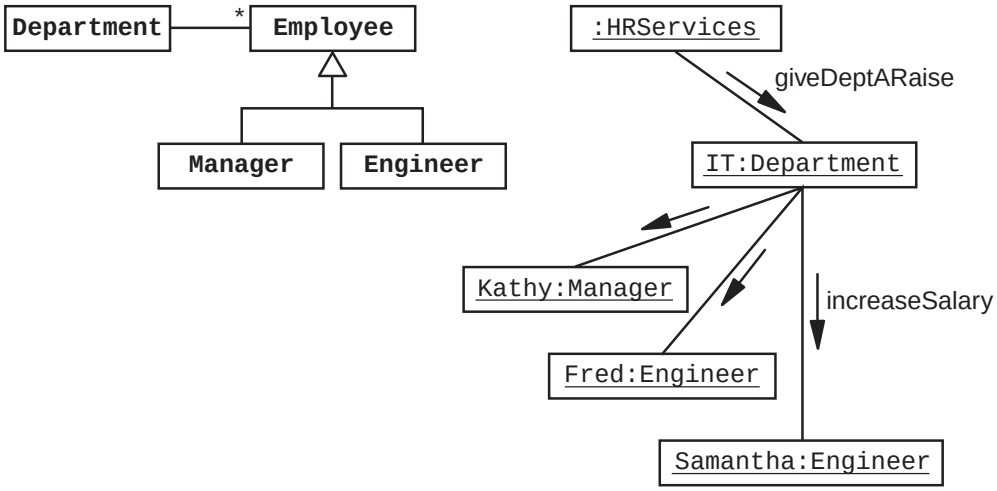


Figure 1-2 Object-Oriented Decomposition

To add a new type of employee, the system only needs to be extended by adding a new subclass of `Employee` with the code necessary to "give a raise." Adding this class does not require any changes to the `Department` class which executes the `increaseSalary` method on any type of employee. This makes OO designs more flexible and resilient to change.

Software Costs

OO principles can reduce development costs in the following ways:

- OO principles provide a natural technique for modeling business entities and processes from the early stages of a project.

  OO development is an improvement over procedural development because you can model software that maps closely to your mental models. This relationship between mental models and software models increases productivity.

- OO-modeled business entities and processes are easier to implement in an OO language.

  Building software in an OO language also increases productivity by enabling the programmers to write code that maps closely to the design models.

OO principles can reduce maintenance costs in the following ways:

- Changeability, flexibility, and adaptability of software is important to keep software running for a long time.

  Meyer (page 17) states that "it is widely estimated that 70 percent of the cost of software is devoted to maintenance," and that the bulk of the maintenance effort is in changes to user requirements and changes in data formats.

  OO designs tend to be more flexible and adaptable than procedural designs; thus reducing maintenance costs.

- OO-modeled business entities and processes can be adapted to new functional requirements.

  OO designs are easier to change in response to new business requirements. OO designs focus on identifying stable business objects that can be made to interact in new ways.

Comparing the Procedural and OO Paradigms

Many programming paradigms have risen over the past half-century. These include: assembler (machine language), procedural, OO, functional, and rule-based to name only a few. Procedural and OO are the leading paradigms in the software industry.

This course focuses on the OO paradigm because this paradigm (which subsumes the procedural paradigm) has significant benefits over a strict procedural paradigm. These paradigms are compared in Table 1-1.

Table 1-1 Comparing the Procedural and OO Paradigms

	Procedural Paradigm	**OO Paradigm**
Organizational structure	• Focuses on hierarchy of procedures and subprocedures • Data is separate from procedures	• Network of collaborating objects • Methods (processes) are often bound together with the state (data) of the object
Protection against inappropriate modification or access	Data is difficult to protect against inappropriate modifications or access when it is passed to or referenced by many different procedures.	The state (data) as well as methods of objects can be protected against inappropriate modifications or access by using encapsulation. Therefore, an object can protect its own state.
Ability to modify software	Can be expensive and difficult to make software that is easy to change, resulting in many "Brittle" systems with software that is difficult to change	Robust software that is easy to change, if written using good OO principles and patterns
Reuse	Reuse of methods is often achieved by copy-and-paste or 1001 parameters.	Reuse of code by using generic components (one or more objects) with well-defined interfaces. This is achieved by extension of classes (or interfaces) or by composition of objects.

Table 1-1 Comparing the Procedural and OO Paradigms

	Procedural Paradigm	**OO Paradigm**
Configuration of special cases	Often requires `if` or `switch` statements. Modification is risky because it often requires altering existing code. So, modifications must be done with extreme care apart from requiring extensive regression testing. These factors make even minor changes costly to implement.	Polymorphic behavior can facilitate the possibility of modifications being primarily additive, subtractive, or substitution of whole components (object or group of objects); thereby, reducing the associated risks and costs.

Surveying the Fundamental OO Concepts

In this section, you will survey the following fundamental OO concepts:

- *Objects*
- *Classes*
- *Abstraction*
- *Encapsulation*
- *Inheritance*
- *Abstract classes*
- *Interfaces*
- *Polymorphism*
- *Cohesion*
- *Coupling*
- *Class associations and object links*
- *Delegation*

There are many other concepts and terminologies that will be covered throughout the course, but these concepts and terminologies are essential for you to understand before this course describes the OOSD disciplines.

Objects

object = state + behavior

"An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class." (Booch Object Solutions page 305)

Objects are runtime features of an OO system. Objects:

- Have identity

  Every object at runtime has an identity that is unique and is independent of its attribute values. The internal representation of object identity is usually hidden by the OO language. However, in C++ an object's identity is its pointer (address in physical memory). In the Java programming language an object's identity is hidden from the programmer; however, program variables refer to objects. Therefore, object references can be passed between objects and methods.

- Are an instance of only one class

  A class defines a type of object. At runtime, an object is defined by one class. However, that class might be an extension of other classes as a result of using inheritance (which is covered later in this section).

- Have attribute values that are unique to that object

  Every object of the same class has the same set of attributes, *but* the values of these attributes are unique for every object. This means that you can change an attribute of one `Account` object without affecting any other `Account` object in memory.

- Have methods that are common to the class

  The operations (or actions) of an object are defined by the set of methods that are implemented in the object's class or in an ancestor class (inheritance is covered later in this section).

Figure 1-3 illustrates an example `Account` object. The outer boxes represent aspects of a program's runtime environment. The Stack box (lower rectangle) represents the current execution frame (a stack frame) with program variables for that frame. In this example, the `account` variable holds a reference to the `Account` object. The Objects box (upper right) represents the space in runtime memory that holds objects; this is usually called the "heap."

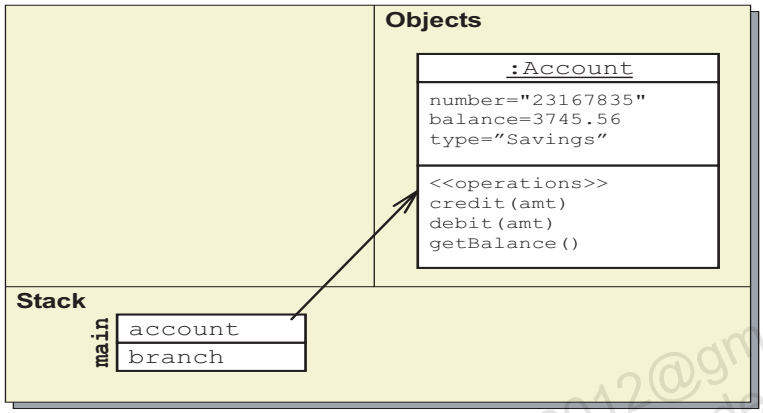


Figure 1-3 An Example Object at Runtime

Classes

> A class is a blueprint or prototype from which objects are created. (The Java™ Tutorials)

In OO programming, a class is usually a language construct that forms the common structure and behavior from which objects are instantiated.

Classes provide the following features:

- The metadata for attributes

  The class does not hold data; the object holds the data. However, the class maintains the metadata of each attribute. This metadata includes the data type and the initial value (if specified).

Note – Most OO languages also support attributes that are scoped on the class itself. Class-scoped attributes are shared among all instances of that class.

- The signature for methods

  The operations or actions of an object are usually called methods in OO terminology. Methods have two elements: signature and implementation. The signature includes the name of the method, the list of parameters (parameter name and type), and the return type.

- The implementation of the methods (usually)

  The implementation of a method is the set of programming statements that specify how the operation is to be performed.

  In some OO languages you can declare a method signature and not provide an implementation; this is called an *abstract method*. If a class has one or more abstract methods, then it is called an *abstract class*. Abstract classes are used in design to force a subclass to implement the abstract methods. Both the C++ and Java programming languages support abstract methods.

- The constructors to initialize attributes at creation time

  A constructor is a set of instructions that initializes an instance. Both the C++ and Java programming languages support constructors. Figure 1-4 illustrates an example `Account` class. Notice that the

operations have now been placed with the class rather than with the object. Classes hold methods and objects hold data. Objects execute methods by looking up the method implementation in the class.

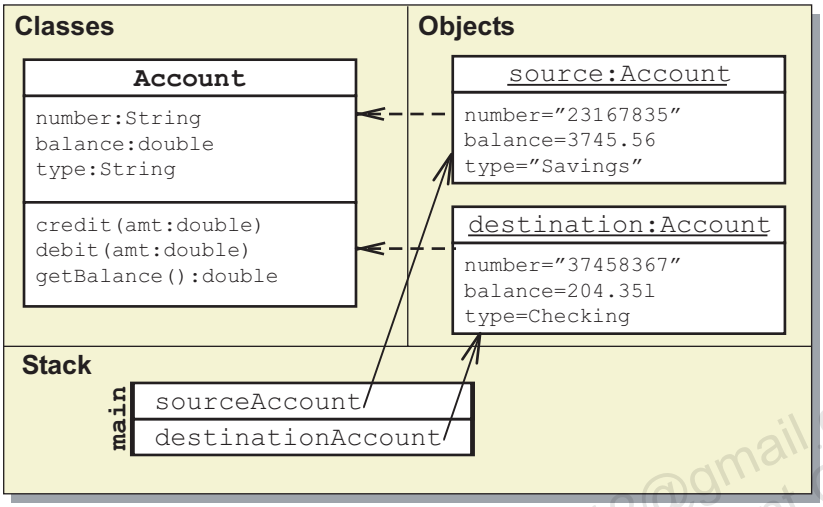


Figure 1-4 An Example Class

Note – We have kept the data types of the attributes simple for now.

Discussion – Discuss how you might do a transfer of funds from `sourceAccount` to `destinationAccount`

Abstraction

> Abstraction is "something that summarizes or concentrates the essentials of a larger thing" (Webster New Collegiate Dictionary)

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

- The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.
- The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.

Abstraction is a fundamental OO concept because finding the right abstractions enables you to design just those features defined by the system requirements and to ignore all other details that are irrelevant to the problem being solved.

Figure 1-5 illustrates two example `Engineer` classes. The class on the left has only the set of attributes and operations needed to satisfy the system requirements. The class on the right has a few attributes and objects that exist for most engineers (such as fingers and toes), but are irrelevant to the system requirements.

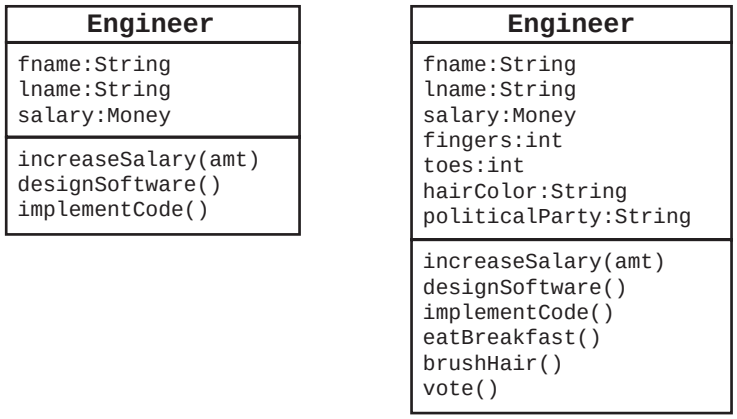


Figure 1-5 An Example of Good and Bad Abstractions

Another aspect of abstraction in software is the idea that you can hide implementation details behind a public interface (one or more methods). This is often called *information hiding* and is discussed in "Encapsulation" on page 1-17.

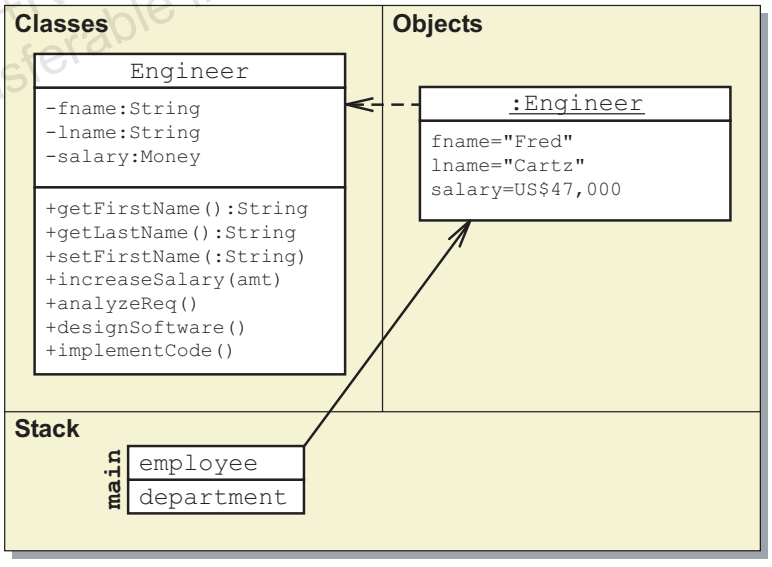
Encapsulation

> Encapsulation means "to enclose in or as if in a capsule." (Webster New Collegiate Dictionary)

Encapsulation is an essential property of an object. An object, as a programming construct, is a capsule that holds the object's internal state within its boundary. Most OO languages, such as the C++ and Java programming languages, have encapsulation built into the language itself.

In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: "hide implementation details behind a set of non-private methods." It is this aspect of encapsulation that most programmers think of, because in order to create proper encapsulation you must do two things: Make all attributes private and provide accessor and mutator methods to provide an abstract interface to the data held within the object capsule.

Figure 1-6 illustrates an example of encapsulation using the `Engineer` class. Notice that the attributes have been declared private (indicated by the minus sign before the attribute name) and accessor methods (such as `getFirstName`) and mutator methods (such as `setFirstName`) have been declared.




```
✗ name = employee.fname;                    ✓ name = employee.getFirstName();
✗ employee.fname = "Samantha";              ✓ employee.setFirstName("Samantha");
```

Figure 1-6 An Example of Encapsulation

Some programmers might argue that creating these methods is a waste of time. They might say: "Why do I need a method that just returns the value of an attribute?" or "Why do I need a method to set an attribute?" The following reasons might help relieve these concerns:

- Limiting the scope of bugs

  Strong encapsulation reduces the number of bugs in which one class incorrectly uses another class. For example, suppose that you are using a database to store employee records. Further, suppose that the first name field in the database has been defined as a `VARCHAR(10)`. This constraint means that the program must not try to store a string of length greater than 10 characters. If you did not use encapsulation, then a client of the `Engineer` class could set the `fname` attribute to a string of any length. With proper encapsulation, the `setFirstName` method could perform the length verification before setting the value. If the length check fails, then the mutator method could throw an exception.

- Abstraction

  To use an encapsulated class, you only need to know the purpose and signature of the public methods of the class. This tactic enables the programmer of the class to change the implementation inside the class without changing the public interface. The client of the class would never know that anything had changed.

Inheritance

Inheritance is "a mechanism whereby a class is defined in reference to others, adding all their features to its own." (Meyer page 1197)

Features of inheritance:

- Attributes and methods from the superclass are included in the subclass.

  Because managers and engineers share some common attributes (name and salary), a superclass `Employee` is defined to hold these common attributes and operations. The `Engineer` and `Manager` subclasses inherit all of the features of the superclass plus these classes can extend the superclass with new features specific to that type of employee.

- Subclass methods can override superclass methods.

  Notice that the `increaseSalary` method is declared by the `Employee` class. This declaration means that the subclasses will inherit this default behavior. However, the `Manager` class also includes the `increaseSalary` method. This declaration means that an object of type `Manager` will use its own implementation to "get a raise."

- The following conditions must be true for the inheritance relationship to be plausible:

  - A subclass object *is a (is a kind of)* the superclass object.

    Using the *is a* or *is a kind of* test helps you avoid using the inheritance relationship when it is either inappropriate now or in the future (if changes are made). For instance, *is a* `Product` an `Employee` fails the test, whereas *is a* `Engineer` an `Employee` passes the test. However, even if a relationship passes the test, it does not necessarily mean that using the inheritance relationship is the best approach. We will discuss these issues later in this module.

  - Inheritance should conform to Liskov's Substitution Principle (LSP)

    This principle was introduced by Barbara Liskov in 1987. A brief simplified interpretation is that if you substitute a subclass (for example, `Engineer`) for a superclass (for example, `Employee`), then any code that expects to use the superclass should have no surprises from a behavioral point of view. That

is, the code should find that all the methods it expected to find in the superclass are still provided and accessible, with any overridden methods having the same semantic meaning.

- A subclass can inherit from multiple superclasses (called multiple inheritance) *or* a subclass can *only* inherit from a single superclass (single inheritance).

  Some OO languages support multiple inheritance, which enables a subclass to inherit members from multiple superclasses. C++ supports this feature. Unfortunately, this feature has some significant drawbacks when it comes to language design. The Java programming language only supports single inheritance.

Figure 1-7 illustrates an example inheritance hierarchy. The HR example has been extended with a new type of employee: `Manager`.

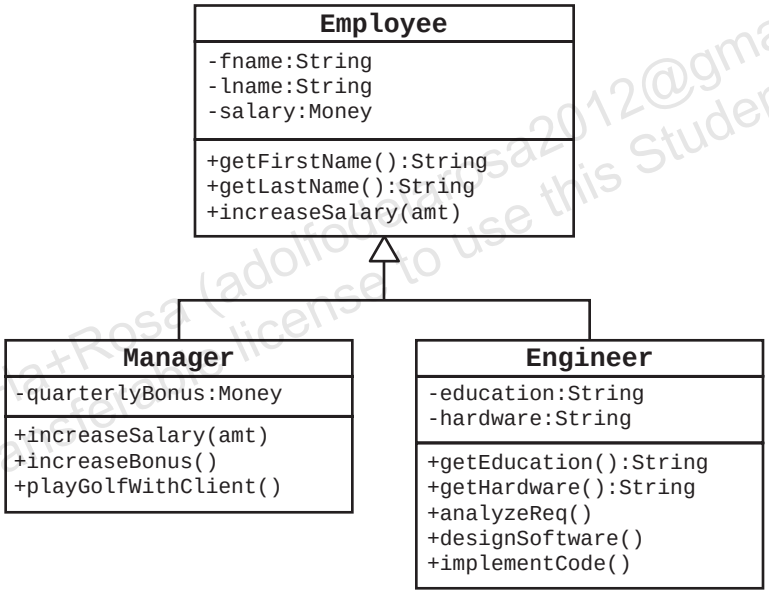


Figure 1-7 An Example Inheritance Hierarchy

Abstract Classes

> A class that contains one or more abstract methods, and therefore can never be instantiated. (Sun Glossary)

Features of abstract classes:

- Attributes are permitted.

  An abstract class may declare instance variables. These attributes are inherited by the subclasses of the abstract class.

- Methods are permitted and some might be declared abstract.

  Methods may be declared in an abstract class. One or more of the methods might also be declared abstract meaning that this class does not define an implementation of that method.

- Constructors are permitted, but no client may directly instantiate an abstract class.

  Constructors may be declared in an abstract class. These constructors can be used by the constructors of subclasses. However, the existence of constructors in an abstract class does not mean that the class may be instantiated. Instances cannot be created because one or more methods are abstract.

- Subclasses of abstract classes must provide implementations of all abstract methods; otherwise, the subclass must also be declared abstract.

  Subclasses of abstract classes may implement any of the abstract methods in its abstract superclass. The subclass may also override any non-abstract methods in the abstract superclass. However, if the subclass does not provide implementations for all abstract methods, then that subclass must also be declared abstract.

- In the UML, a method or a class is denoted as abstract by using italics, or by appending the method name or class name with `{abstract}`.

  It is difficult to use italics in handwritten drawings, which tend to use the `{abstract}` notation.

A *concrete class* is the opposite of an abstract class. A concrete class must have no abstract methods.

Figure 1-8 illustrates an example abstract class. The `Employee` class in the HR example has been declared abstract; likewise the `increaseSalary` method is abstract. Therefore, the `Manager` and `Engineer` classes must implement this method.

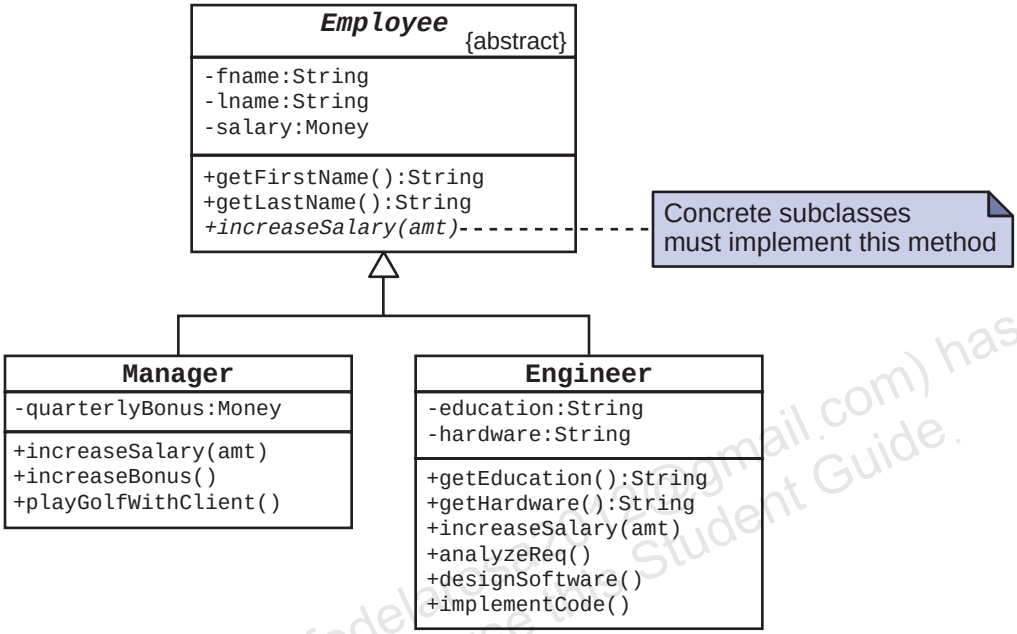


Figure 1-8 An Example Abstract Class

Interfaces

Features of Java technology interfaces:

- Attributes are not permitted (except constants).

  Interfaces by definition do not have attributes. However, interface may declare constants (with the modifiers `public static final`).

- Methods are permitted, but they must be abstract.

  Interfaces are a set of abstract methods. An interface may not declare a method with an implementation.

- Constructors are not permitted.

  Interfaces are not classes and no objects may be constructed from an interface. However, a class might implement an interface and objects of that class may be said to be *instances of the interface.* However, it is more appropriate to say the *instance supports the interface*.

- Subinterfaces may be defined, forming an inheritance hierarchy of interfaces.

  Interfaces may have subinterfaces. Therefore, a set of interfaces might form a hierarchy. For example, the Collections API in the `java.util` package includes a hierarchy of interfaces. The root interface is `Collection` with two subinterfaces `List` and `Set`; the `Set` interface also has subinterface `SortedSet`.

A class may implement one or more interfaces. This is sometimes called interface inheritance because the class inherits the methods of the interfaces. If the class does not implement these methods, then it must be declare abstract.

Note – The Java programming language supports interfaces directly. Other languages, such as C++, can support the concept of interfaces by defining classes that are completely abstract (all methods are virtual and empty) with no attributes.

Figure 1-9 illustrates an example interface. The HR example has been extended to include an interface for scheduling resources, such as people, rooms, and equipment.

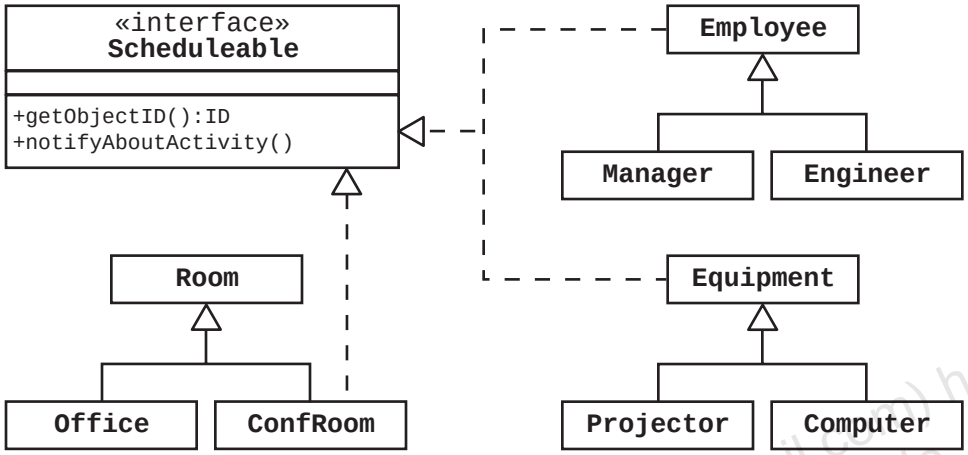


Figure 1-9 An Example Interface

Note – Interfaces can be implemented by any class in any hierarchy; classes that implement an interface are not bound to a single class hierarchy.

The `Employee` and `Equipment` classes implement the `Scheduleable` interface. Because these two classes are the root of a class hierarchy, any subclass of either of these superclasses are also "scheduleable." The `Room` class hierarchy is different because the system should not be permitted to schedule activities for any arbitrary room, such as an office or a closet. However, conference rooms need to be scheduled; therefore, the `ConfRoom` class implements the `Scheduleable` interface independent of the other classes in the rooms hierarchy.

Polymorphism

> Polymorphism is "a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass [type]." (Booch OOAD page 517)

Aspects of polymorphism:

- A variable declared to reference type `T` can be assigned different types of objects at runtime provided they are a subtype of the variable's type `T`.

  As shown in Figure 1-10 on page 1-26, the `employee` variable can reference any type of object whose class is a proper subclass of `Employee`. Polymorphism can be used with variables (local and instance), method parameters, and method return values.

- Method implementation is determined by the type of object, not the type of the declaration.

  Dynamic binding works by looking up the method (based on the signature) within the declared methods for the actual class of the object, not the apparent type (that is, the reference variable type). If the method is not found in that class, then the language environment performs the lookup on the superclass. If the method is not found in the superclass, then the method lookup is performed on the superclass's superclass, and so on.

- Only method signatures defined by the variable type can be called without casting.

  A method signature is the name of the method along with its parameters. A variable declared to reference type `T` can only be used to call methods that are supported by type `T`. Using the example shown in Figure 1-11 on page 1-26 any code using a variable declared to reference a `Scheduleable` type object can only call `getObjectID()` or `notifyAboutActivity()`.

Figure 1-10 illustrates an example of polymorphic behavior. The list contains three objects: one manager and two engineers. However, in the code, the `employee` variable is declared only as `Employee`; so how does the system know which `increaseSalary` method to call? The runtime environment finds the class of the actual object to determine which method to call. This language features is called *dynamic binding*.

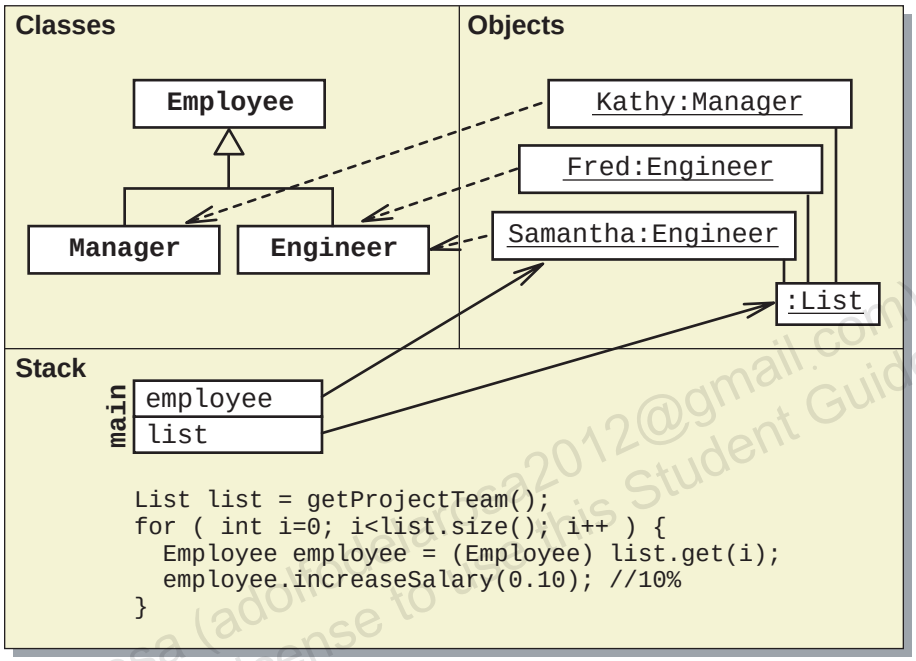


Figure 1-10 An Example of Polymorphism

Figure 1-11 illustrates an example using interfaces. The `Schedule` class enables any kind of scheduleable object to be added to a schedule. Therefore, employee objects, equipment objects, and conference room objects might be added to a schedule.

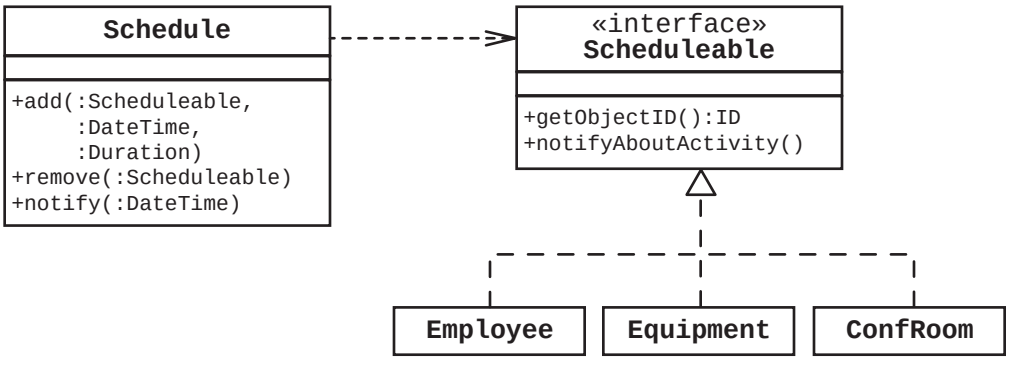


Figure 1-11 A Polymorphism Example Using Interfaces

Cohesion

> Cohesion is "the measure of how much an entity (component or class) supports a singular purpose within a system." (Knoernschild page 174)

In software, the concept of cohesion refers to how well a given component or method supports a single purpose.

Aspects of cohesion:

- *Low cohesion* occurs when a component is responsible for many unrelated features.

  Big classes with many unrelated methods tend to be hard to maintain.

- *High cohesion* occurs when a component is responsible for only one set of related features.

  Small classes with fewer, but highly related, methods tend to be easier to maintain.

- A component includes one or more classes. Therefore, cohesion applies to a single class, a subsystem, and a system.

- Cohesion also applies to other apsects including methods and packages.

- Components that do everything are often described with the Anti-Pattern terms of Blob components

  Whereas Patterns (as covered later in the course) are generally desirable, Anti-Patterns are generally undesirable.

Figure 1-12 illustrates a class, `SystemServices`, that has low cohesion. This class has functions that are completely unrelated: logging in and out, manipulating employees, and manipulating departments. The classes on the right side show a refactoring of the `SystemServices` class into three highly cohesive classes.

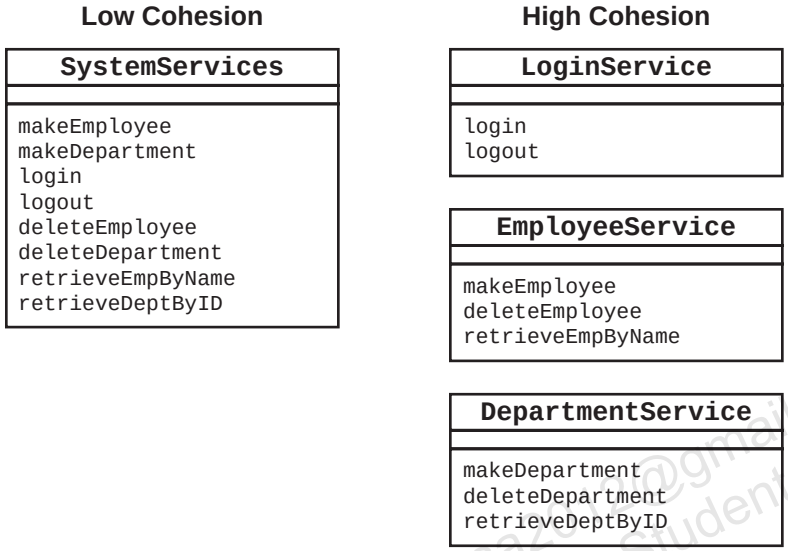


Figure 1-12 An Example of Low and High Cohesion

Coupling

> Coupling is "the degree to which classes within our system are dependent on each other." (Knoernschild page 174)

Coupling refers to the degree to which one class makes use of another class. The typical example is of a `Client` class that uses a `Service` class. In some cases a `Client` class never uses a `Service` class; this is called *no coupling*. A more interesting case is when the `Client` class that uses a concrete `Service` class; this is called loose coupling because the `Client` class has intimate knowledge of the `Service` class. Finally, tight coupling occurs when both the `Client` and `Service` class depend upon each other. The left side of Figure 1-13 shows these three cases.

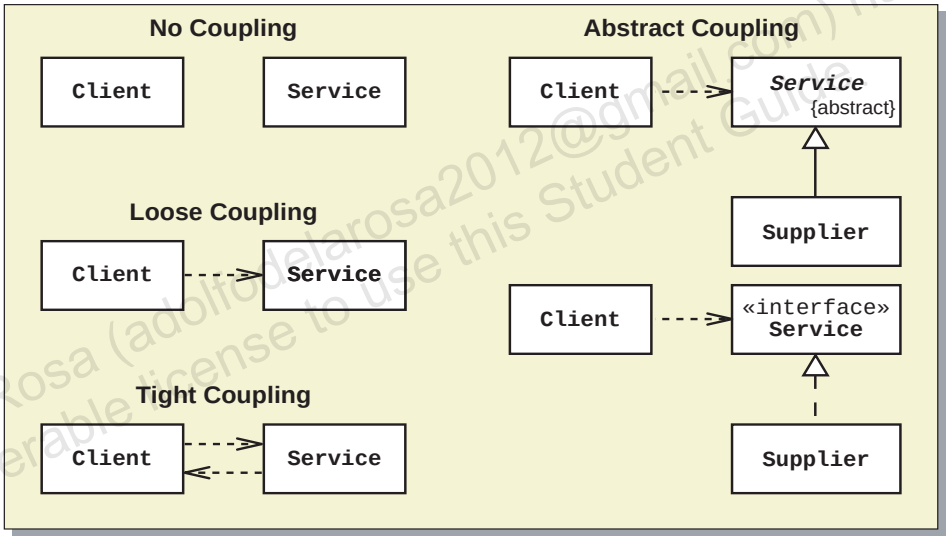


Figure 1-13 Four Forms of Coupling

The right side of Figure 1-13 shows two variations on a form of coupling called, *abstract coupling*. In this situation, the client requires the use of a service, but it does not know what concrete class is supplying that service. The Java programming language supports two abstract coupling mechanisms. One mechanism is to create an abstract `Service` class and then one or more concrete `Supplier` classes, which extend the `Service` class. Alternatively, you could create a `Service` interface and then one or more concrete `Supplier` classes, which implement the `Service` interface.

Note – Abstract coupling is an example of the Dependency Inversion Principle, which is covered later in the course.

Class Associations and Object Links

Class Associations

Class associations have many dimensions. These associations are discussed in greater detail later in the course. For now, you should know about these features:

- The roles that each class plays

  An association can be labeled with role names. These role names identify how each object is related to the other. For example, the association between `Department` and `Employee` as a role name of `dept` next to the `Department` class.

- The multiplicity of each role

  An association defines how many objects can participate in the association. For example, in the association between `Department` and `Employee`, an employee has only one (`1`) department, but a department can have one or more (`1..*`) employees.

Note – Multiplicity of roles is covered in more detail later in the course.

- The direction (or navigability) of the association

  An association can define the navigability of the association. This is the path leading from one object to another. For example, from the `department` object the system can navigate to any of the employees, and from an employee the system can navigate to that person's department. However, this structure specifies that the system can navigate from the manager to that manager's department; however, the `department` object cannot navigate to that department's manager.

Object Links

- Object links are instances of the class association.

  Just as objects are instances of classes, object links are instances of the class association.

- Object links are one-to-one relationships.

  In a one-to-many or many-to-many class relationship, each link represents a connection from a single object to a single object.

Figure 1-14 illustrates the use of class associations and object links. The top diagram shows a set of classes and their associations. The `Department` class is associated with one or more objects of the `Employee` class. The `Manager` class is associated with one (and only one) object of the `Department` class. The bottom diagram shows one example of a group of objects and their runtime links at a particular moment in time. The IT department contains three employees, and Kathy is the manager of the IT department. If we add another employee (or subtype) to the department, then the bottom (object) diagram would change, provided that the multiplicity constraints of the top (class) diagram allow the change.

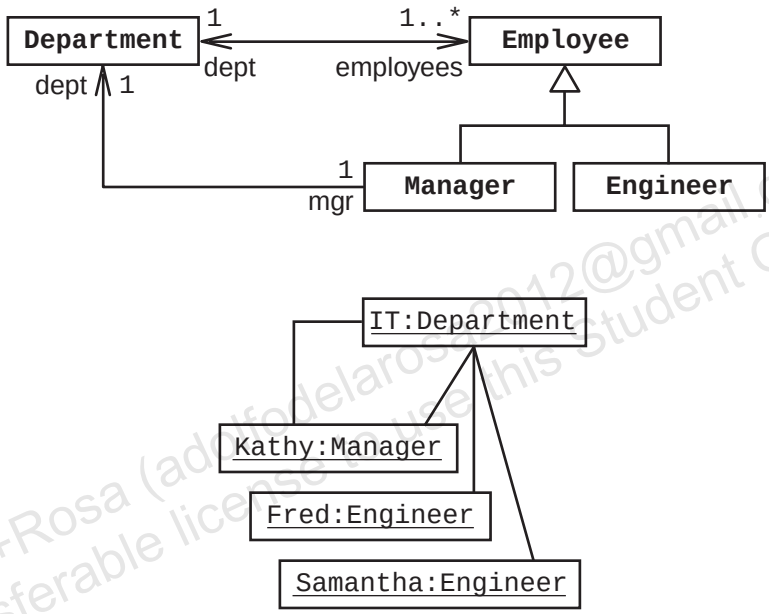


Figure 1-14 An Example of Object Associations

Delegation

Many computing problems can be easily solved by delegation to a more cohesive component (one or more classes) or method.

- Delegation is similar to how we humans behave. For example:
  - A manager often delegates tasks to an employee with the appropriate skills.
  - You often delegate plumbing problems to a plumber.
  - A car delegates accelerate, brake, and steer messages to its subcomponents, who in turn delegate messages to their subcomponents. This delegation of messages eventually affects the engine, brakes, and wheel direction respectively.
- OO paradigm frequently mimics the real world.
- The ways you delegate in OO paradigm include delegating to:
  - A more cohesive linked object
  - A collection of cohesive linked objects
  - A method in a subclass
  - A method in a superclass
  - A method in the same class

Figure 1-15 on page 1-33 illustrates one of the many problems that delegation can solve. The figure shows the example from Figure 1-8 on page 1-22 with some additional classes added. These are just a few of the additional classes whose objects might have links to the `Employee` objects at some time. In this business problem, it is likely that employees change their job roles. For example, an engineer might be promoted to a manager. However, you cannot change your class. So, for an engineer to be promoted to a manager, you would have to do the following:

1. Create a new object of type `Manager` and copy the common attribute values (`fname`, `lname`, and `salary`) to the new object.
2. Ask all the objects that use the `Engineer` object to replace that link with a link to the `Manager` object.
3. Delete the `Engineer` object.

Note – We have not addressed the issues of keeping a historical record of an employee's employment.

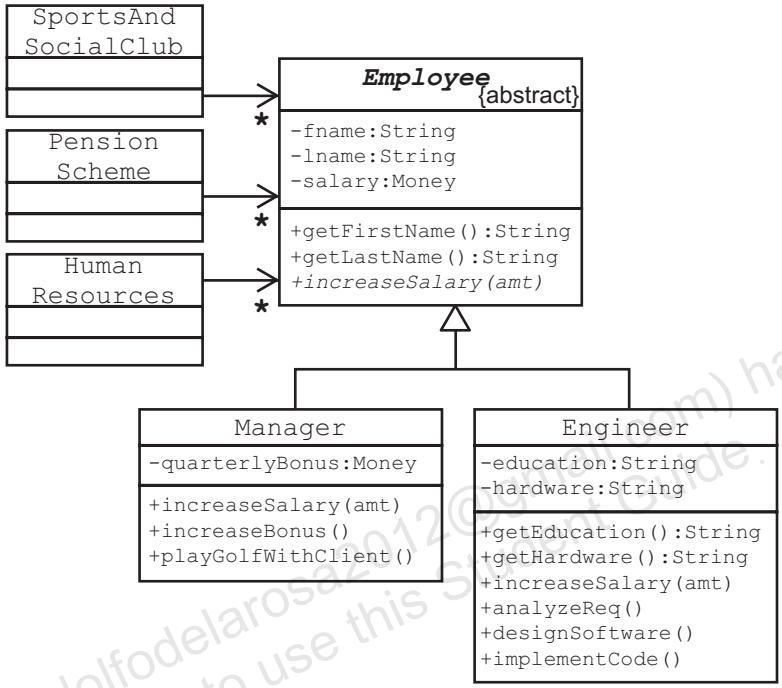


Figure 1-15 Issues with Inheritance on Non-Coherent Classes

The problem in this example is because the details of job roles and non-job roles are mixed together, which is not coherent. Figure 1-16 shows one possible solution that separates these two aspects and associates the two classes. In this case, the `Employee` will delegate the job role to the `JobRole` class that is subclassed into `Engineer` or `Manager`.

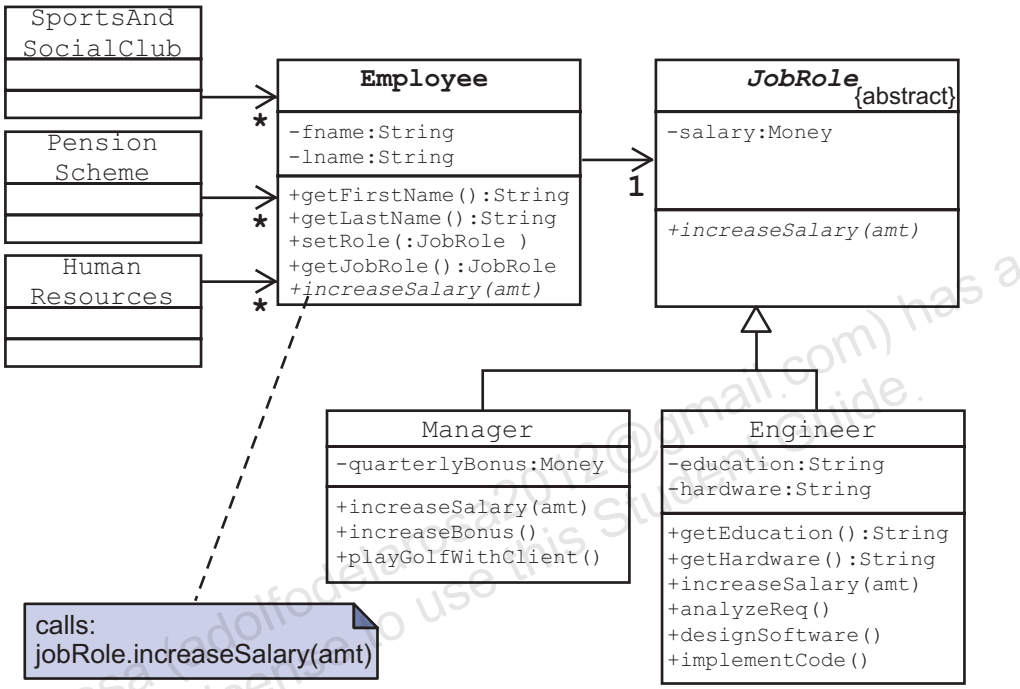


Figure 1-16 Delegation to a more Cohesive Class

Note – As this is an introduction, we have not covered all of the aspects of delegation. For example, we have not covered having multiple delegates, such as roles or types. These aspects will be covered later in the course.

Note – Delegation is not only for roles; it has many other discriminators—for example, types.

Summary

Object orientation is a model of computation that is closer to how humans think about problems. Using an OO paradigm for developing systems has many benefits over procedural development because of the closer mapping between your mental models (the system requirements), your analysis and design models, and your implementation.

OO provides a set of fundamental concepts:

- Object
- Class
- Abstraction
- Encapsulation and information hiding
- Inheritance
- Abstract classes
- Interfaces
- Polymorphism
- Cohesion
- Coupling
- Class associations and object links
- Delegation

These OO concepts are the basis of most OO principles and patterns used throughout this course.