Difference uml:

Aggregation vs. Composition

An aggregation is a special form of association between classes that represents the concept of “WHOLE - PART”. Each object of one of the classes that belong to the aggregation (the composed class) is composed of objects of the other class of the aggregation (the component class). The composed class is often called "whole" and the component classes are often called "parts". An intuitive example of aggregation is the relationship between a wood and its threes. The wood can be considered as the whole and the threes would be the parts that belong to the wood.

However, UML distinguishes only

between two kind of aggregation: simple aggregation and composed aggregation .

Aggregation

A simple aggregation is an aggregation where each part can be part of more than one

whole. This kind of aggregation is very common in the literature and has been oftenrefereed as logical or catalogue aggregation, even in the first drafts of UML [6]. Asan example of simple aggregation we can think in the catalogue of dolls of the“ToysX” store that contains n Barbie models.

However, the same Barbie models can appear in the catalogue of dolls of different toy-stores. This is possible because there exists a logical aggregation, and the dolls do not compound physically the catalogues. Figure 3 shows a simple aggregation. It is

represented placing a diamond next to the whole
class.

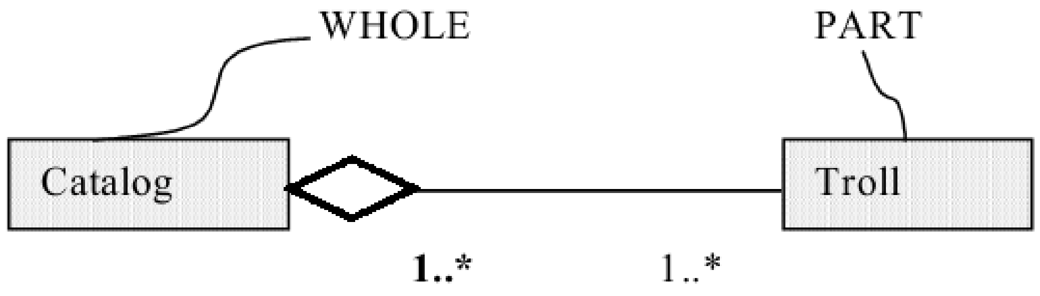


Fig.3. Simple Aggregation Example

Simple aggregation does not imply any kind of restriction over the life of the parts with regard to its whole.

Composition

A composition, also called composed aggregation, is a special kind of aggregation in which the parts are physically included in the whole. Once a part has been created it lives and dies with its whole. A part can be explicitly removed before removing its associated whole. As it is a physical aggregation, a part can only belong to a whole. Example of composition. University is the whole and departments
are its parts. The live of a department depends on the live of the university to which it belongs. If the university disappears its departments disappears as well. Besides, a department can be joined only to a university. The representation of the composition is similar to the representation of the simple aggregation. The only difference is that the diamond is fulfilled.

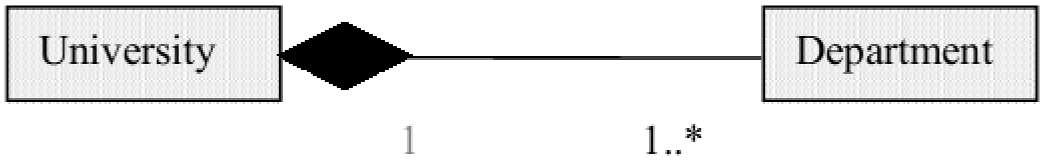


Fig.4. Composition Example

Differences

1. Both aggregation and composition are special kinds of associations.

2. Aggregation - compose of somethings
Composition - is an aggregation class in the composite design pattern

composition - compose of something
for example, a car has engine,wheels, seats

aggregation - a subclass of something (inheritance class)
parrot is a type of bird
car is a type of vehicle

3. Aggregation is used to represent ownership or a whole/part relationship, and composition is used to represent an even stronger form of ownership.

With composition, we get coincident lifetime of part with the whole. The composite object has sole responsibility for the disposition of its parts in terms of creation and destruction. In implementation terms, the composite is responsible for memory allocation and deallocation.

4. Moreover, the multiplicity of the aggregate end may not exceed one; i.e., it is unshared. An object may be part of only one composite at a time.

If the composite is destroyed, it must either destroy all its parts or else give responsibility for them to some other object. A composite object can be designed with the knowledge that no other object will destroy its parts.

5. Composition can be used to model by-value aggregation, which is semantically equivalent to an attribute. In fact, composition was originally called aggregation-by-value in an earlier UML

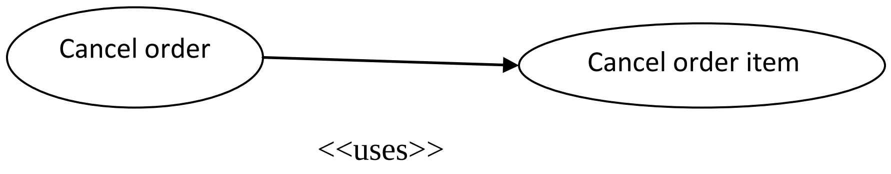
draft, with "normal" aggregation being thought of as aggregation-by-reference. The definitions have changed slightly, but the general ideas still apply.

The distinction between aggregation and composition is more of a design concept and is not usually relevant during analysis.

Finally, a word of warning on terminology. While UML uses the terms association, aggregation, and composition with specific meanings, some object-oriented authors use one or more of these terms with slightly different interpretations.

USES /INCLUDE Relationship:-

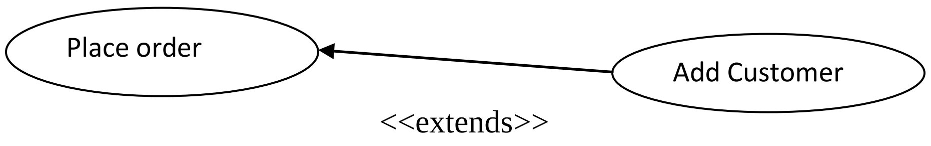
- Multiple use cases share a piece of some functionality.
- This functionality is placed in separate use case rather than documenting in every use case that needs it.
- A uses relationship shows behavior that is common to one or more use cases.
- Instance of use case 'A' will also include the behavior as specified by 'B'.



- The behavior of "Cancel order item" is always included in the "Cancel order" use case.

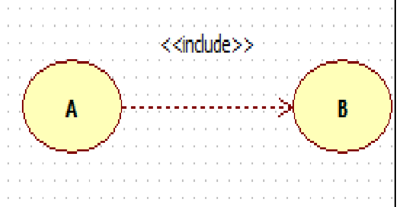
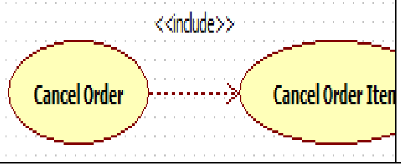
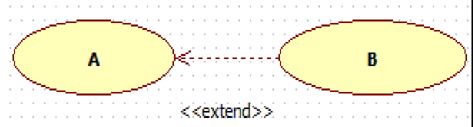
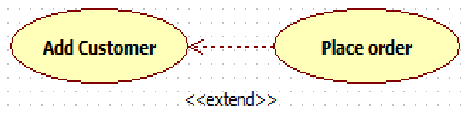
1. **EXTENDS relationship:-**

- It is used to show optional behavior which is required only user certain condition.
- Extends relationship from use case 'A' to use case 'B' may include the behavior specified by A.



- The use case "place order" sometimes includes the behavior of "Add customer".

(For eg. When place order attempts to create an order for a customer who has not yet been added to the system.) Sometimes, while placing order, the customer is added.

Extends	Include/Uses
Extension use case references the base use case.	Base use case refers the inclusion use case.
Extension may be invoked independently.	Inclusion may be invoked independently.
The base use case must be well formed even if extension use case is not invoked.	Base use case may depend on the effects of inclusion use case.
Extension usually takes care of alternate scenario (which might happen to be exception)	Includes relationship may be linked to aggregation of classes.
1. The uses arrow is drawn from a use case A to another use case B to indicate the process of doing A always involves doing B at least once. 2. A uses relation from use case A to use case B indicates that an instance of the use case A will also be include in the behavior as specified by use-case B. 3. The notation is a close-headed arrow annotated with the stereo type name uses. 4. **Symbol:**  5. **Example:** 	1) The extends arrow is drawn if a use case A to a use case B, indicate that the process A is a special case behavior of the same as the more general process B. 2) An Extends relationship from a use case A to use case B indicate that an instance of use case may include the behavior. 3) The notation is the enclosed headed arrow directed arrow from the extension with the <<extends>> stereotype annotation. 4) **Symbol:**  5) **Example:** 

Architectural vs. Design Pattern

- The reuse of classes, components, and frameworks differ in the scale and mechanism of reuse, but the three share one important property: in all these cases, we are reusing the code.
- A pattern is to reuse the knowledge. A pattern is a way of solving the problem.
- A pattern typically promotes a design principle while solving a problem.
- Some patterns make the use of encapsulation other reduces various type of coupling and increasing cohesion.
- A pattern provides a solution to make your design more flexible. Because a pattern embodies the reuse of knowledge rather than the code developing a catalogs of patterns has become an important point of programming communication.

Architecture pattern versus Design pattern

1. Architectural pattern:-
    - It is a problem independent way of organizing a system or subsystem. It describes a structure by which different parts of system are organized or interact.
1. Design pattern:-
    - A design pattern is a solution to small problem i.e. independent of any problem domain. It represents a solution to a design problem that to occur in any kind of application. Some design pattern can be applied to different areas like order processing, online shopping cart, banking application or any other system.
2. Architecture pattern:-
    - Architectural means how data is flowing in your application.
2. Design pattern:-
    - Design pattern means how you are approaching to problem.
3. Architectural pattern:-
    - Architectural pattern concentrate on ‘what’.
3. Design pattern:-
    - Design pattern concentrate on ‘how’.
4. Architectural pattern:-
    - Architecture means flow of particular business logic or project.
4. Design pattern:-
    - Design means approach for that particular business logic or project.
5. Architecture pattern:-
    - Architectural states how that particular logic to be implemented.
5. Design pattern:-
    - Design pattern states the rules and regulation to be followed while implementing some software.
6. Architecture pattern:-
    - Architecture is concern with the selection of architectural elements, their interaction and the constraints on those elements and their interaction.
6. Design pattern:-

- Design is concern with the modularization and detects interfaces of design element, their procedure and datatype needed to support the architecture and to satisfy the requirements.

Functional modeling vs object modeling

1. It describes data flow & constraints to describe business process.	a) It describes the structure of the objects in the system in terms of identity relationship, attribute and operation.
2. Weaker approach, then object modeling.	b) Strongest approach then functional modeling.
3. It uses DFD for describing business processes.	c) It uses class diagram to describe the system.
4. It is difficult to prototype.	d) It is easy to prototype.
5. It cannot map the model to real world scenario	e) It maps model to real worlds scenario.

Inheritance vs association

1. It is identifying and defining a system of hierarchies between classes.	i. Associations are relationships between attributes & also between classes.
2. Inheritance can be:- Generalization Specialization	ii. An association could be : One-To-One One-To-Many Many-To-One
3. There is concept of super class and sub class.	iii. There is no concept of super & sub-class.
4. Inheritance does not include any constraints.	iv. Associations include cardinality constraints on its end.

Qualified Association and Association

Qualified association :

A **qualified association** has a **qualifier**that is used to select an object (or objects) from a larger set of related objects, based upon the qualifier key. Informally, in a software perspective, it suggests looking things up by a key, such as objects in a *HashMap*.

The standard in memory implemetation idiom for aqualified association is a keyed list where key is a qualifier .The implicit cardinality opposite the qualifier is always many

Association:

An association is a relationship between classes in a class diagram

A link between objects is an instance of an association between classes

Therefore any link in an object diagram must be derived from an association in the class diagram The diagramatic conventions for links and associations are the same except the name of the link is underlined

Abstract classes and parametrised classes:

Abstract classes

Abstract class does not provide a complete declaration and can typically not be instantiated

An abstact class is intended to be used by other classes.

Any class with an abstract method must be abstarct

An abstract class should not necessarily have abstarct method thus we may have classes whose interfaces are fully implemented but u do not want to create instances of them.

Parametrised classes:

Parameterised classes are implemented in different object oriented languages

Parameterised classes are extension of abstarct classes

Parameterised classes contain all concrete methods

They can be instantiated unlike abstarct classes

Activity and Statechart diagrams:

Activity diagram:

- ***Flow***: permits the interaction between two nodes of the activity diagram (represented by edges in activity diagram)
- ***State***: a condition or situation in the life of an object during which it satisfies some conditions, performs some activities, or waits for some events
- ***Type***: specifies a domain of objects together with the operations applicable to the objects (also none); includes primitive built-in types (such as integer and string) and enumeration types
- ***Token***: contains an object, datum, or locus of control, and is present in the activity diagram at a particular node; each token is distinct from any other, even if it contains the same value as another
- ***Value***: an element of a type domain

They are useful to specify software and hardware system behaviour

They are based on data flow models

The fundamental unit of executable functionality in an activity

Statechart diagrams:

It shows the sequence of states that object of a class goes through during its lifecycle in response to the external events and responses and actions in reaction to an event

Model Elements

States:

Transitions :

Events:

Actions and activities:

Elements:

Event:significant or noteworthy occurrence

State:Condition of an object at a moment in time

Transition:Relationship between two states that indicates that when event occurs object moves from one state to another

Graph relating events and states

Nodes are states and arcs are events

Describes behaviour of a single class of objects

Event vs state

Event	State
1. n event is something that happens at a point in time such as user depresses the left button of mouse. 2. n event does not have any duration.	The attribute values and links held by an object are called its states. A state is a condition or situation during the life of an

	object during which it satisfies some condition, performs some activity, or waits for some event.

Component vs node

Node	**Component**
1. node corresponds to hardware. 2. he node can represent a piece of a larger hardware component, represented by encapsulating the node (having one node inside the other node). 3. node represents a physical component of the system.	Components represent software. Each component is representative of one or more classes that implement one function within the system. Component is used to represent any part of a system for which UML

	diagrams are made.

Aggregation and Inheritance

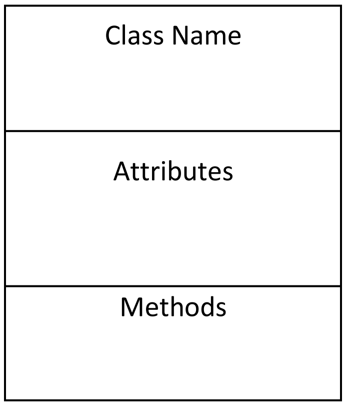
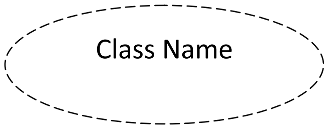
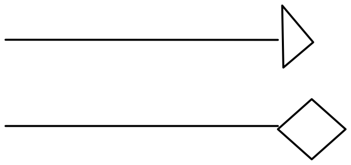
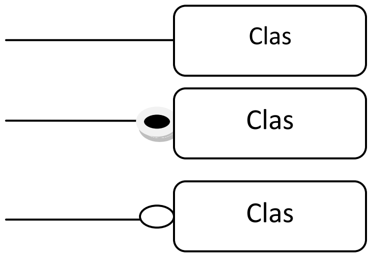
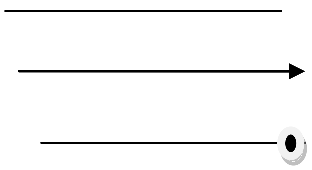
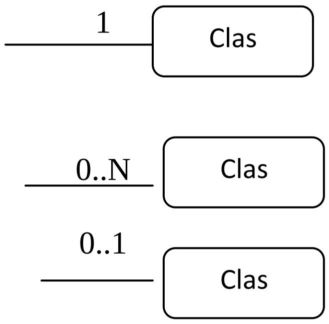
1. ggregation is a special form of association, between a whole and its parts, in which the whole is composed of the part. 2. n aggregation is relating an assembly class to one component. 3. t is also called 'has a' relationship. 4. omponent objects do not have a separate existence, they depend on composite objects.	Inheritance is that property of object-oriented systems that allows objects to be built from other objects. Inheritance is a relationship between classes where one class is the parent class of another class It is called 'is a' relationship. The derived class has a separate existenc

MissionMca.com

	e, they only redefine its parent class and add a few properties of its own.

Differentiate between OMT Technique & Booch Methodology

OMT Technique	**Booch Methodology**
Phases of OMT: 1. Analysis 2. System Design 3. Object Design 4. Implementation	Phases Of Booch Methodology : 1. Conceptualization 2. Analysis 3. Design 4. Evolution 5. Maintenance
OMT Diagrams : 1. Object Model 2. Dynamic Model 3. Functional Model	Diagrams of Booch Methodology : 1. Class Diagram 2. Object Diagram 3. State Transition Diagram 4. Module Transition Diagram 5. Process Diagram 6. Interaction Diagram

OMT Notation for Class 	Booch Notation for Class 
OMT Notation for class Relationship Association Inheritance Aggregation  OMT notation for Multiplicity Exactly one 0 or more 0 or 1 	Booch Notation for Class Relationship Association Inheritance Aggregation  Booch Notation for Multiplicirty Exactly one 0 or more 0 or1 

specialization and aggregation (page 128-UML-charles Richter)

Specialization versus aggregation:

- The use of class specialization can result in an inflexible design.
- In some cases, employing aggregation instead produces a superior solution. One such example is the specialization of a class along different (unrelated) dimensions: Another example is a property shared by some, but not all, subclasses. In some cases, multiple inheritance is used in situations that call for aggregation. These problems and their solutions are described.

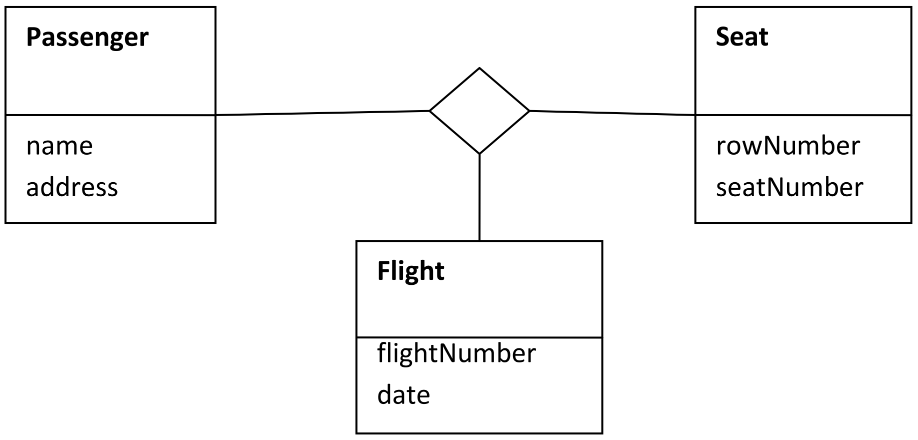
• **Aggregation**:

- The advantages of hiding an aggregate's parts within the aggregate are outlined. Different approaches that allow an aggregate's parts to be reused in other applications are examined.

2. Ternary and Reflexive associations (page 61-UML-charles Richter)

Ternary and Reflexive Associations :

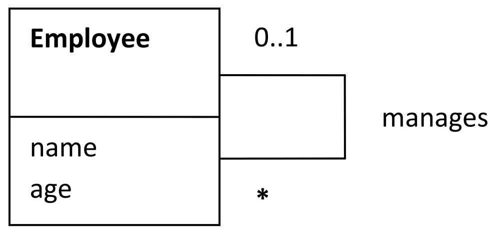
The degree of all of the associations described until now is two, meaning that each relates two classes. UML permits the specification of associations of any degree, however. The notation for associations of degree three or greater is a diamond with "legs" to all classes that participate in the association, as depicted in Figure 2.22.



While ternary (and larger) associations are permitted in UML, you should normally try to avoid them during design. One reason is based on human cognition: Such associations are not intuitive and are somewhat difficult to describe. In a ternary association, for instance, there are six cardinality constraints. What is the best means to specify those constraints?

Another problem with ternary associations is that they have no standard implementation idioms. Binary associations have idioms, such as the containers (lists) described above, but how would you implement the association in Figure 2.22? If the association is implemented as a class (where each instance of the class is a link), why not simply introduce that class in the design? In Figure 2.23, for example, the ternary association in Figure 2.22 has been transformed into the Seat Assignment class. An instance of this class represents a ternary link between a Passenger, Seat, and Flight instance.

UML also allows **reflexive**(or recursive) associations in which a class is linked to another.An Example such association is shown below:



3. method overriding and polymorphism (page 78-UML-charles Richter)

Method Overriding and Polymorphism :

Consider the example in Figure 2.43. A run method has been introduced in the Automobile class. Note also that a run method is defined in Vehicle, the super class of Automobile. The Automobile class is overriding the run implementation it inherits from Vehicle, which means that it is replacing the inherited run behav-ior with its own variant. The run implementation in Automobile may either call the run implementation inherited from Vehicle (in which case it extends, rather than replaces, the behavior), or it may ignore the inherited implementation altogether.

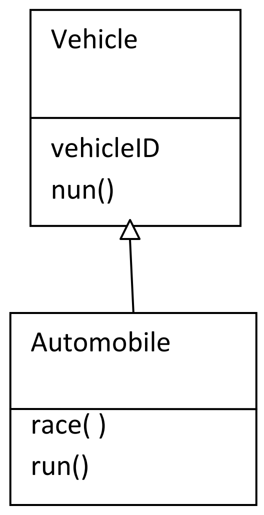


FIGURE 2 .43 Overriding a method.

Consider the Java code fragment for method x .-f (repeated from Section 2.5.2, "Type Promotion"):

```
class X {
 public static void f(Vehicle v) {
V. run() ;
          }
}
```

Suppose that you call x.f,passing a reference to an Automobile. Which run implementation will be invoked? The compile-time type of the reference in x.f is Vehicle, which means that if the decision is made at compilation (or link) time, the run method in Vehicle will be called. On the other hand, the execution-time class of the object to which v refers in this example is Automobile; therefore, if the binding is deferred until execution time, Automobile's run method could be invoked.

In the typical case, the binding is delayed until execution time; so, the run method for Automobile will be called. (There are ways to force binding at com-pilation time, but that is beyond the scope of this book.) This means that, even though an Automobile object is being referenced as a vehicle, it still acts like what it really is anAutomobile. This is referred to as run-time (or execution-time) binding, polymorphism, or single dispatching. (The "dispatching" part of the last term refers to the selection of a method implementation, whereas the "sin-gle" portion indicates that the selection is based on the type of a single object. In the example, the correct implementation of run is selected based on the type of the object on which run is invoked.)

Note: Polymorphism means many forms; therefore, the mere overriding of a method technically would qualify as polymorphism, regardless of when the binding occurs. The customary use of the term, however, includes execution-time binding.

What are the advantages of type promotion and polymorphism? Consider this scenario of a system that has text and binary files. You occasionally must print files, but text and binary files employ different print implementations. If you were to write C-style code to print files, you would do something like the following:

```
class Client {
 public void handleFile (File f) {
 switch (f.getType()) {
case TEXT: // print text file
case BINARY: // Print binary file
            }
      }
}
```

In other words, each time you must print a file, you introduce conditional code that is based on the type of the file . You originally intended to write this condi-tional code only once, but over time, the code is duplicated in several places. When confronted with your colleagues' comments about the proliferation of conditional code, you offer that there shouldn't be any additional types of files. After all, other than text and binary files, what types of files could possibly exist?

Over time, of course, you encounter several new file types, each printed in a different way. PostscriptTM files, for example, require the invocation of a Postscript driver. Image files require some rendering of the image. In each case, You must find every occurrence of the conditional printing code and add a new condition with the code to print the new type of file. The alternative to the preceding conditional code is to employ type promotion and polymorphism. A File superclass defines the print method. Each different type of file is represented by a subclass of File that implements the print method appropriately for that type of file. An initial hierarchy with Text File and Binary File is depicted in Figure 2.44.

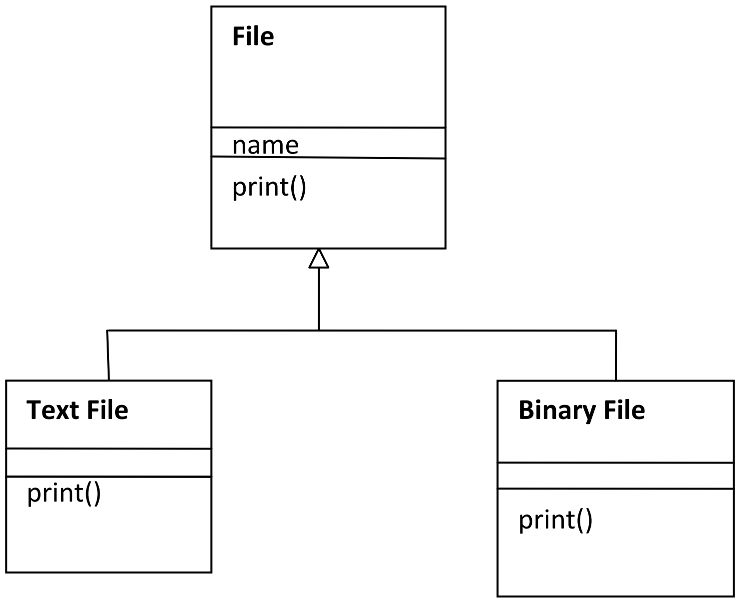


FIGURE 2.44 The initial File class hierarchy.

A client that prints files is passed a reference to a specific type of file, but the client receives it as a reference to a generic File. The client code for printing files is as follows:

```
class Client {
        public void dolt (File f) {
                f.print(); // polymorphic print call
                }
}
```

Note that this client code is completely ignorant of the specific (sub)types of files. In some part of your application, of course, you must create the specific types of files. You should strive to restrict the code that is aware of the File subclasses to a small part of your application. After creating instances of specific type of File subclasses, promote those references to be of type File(and deal only with files in the rest of the application).

Adding a new type of file requires adding a new subclass. Figure 2.45 illus-trates the addition of an Image File. The preceding client code does not change, because it deals only with generic File references.

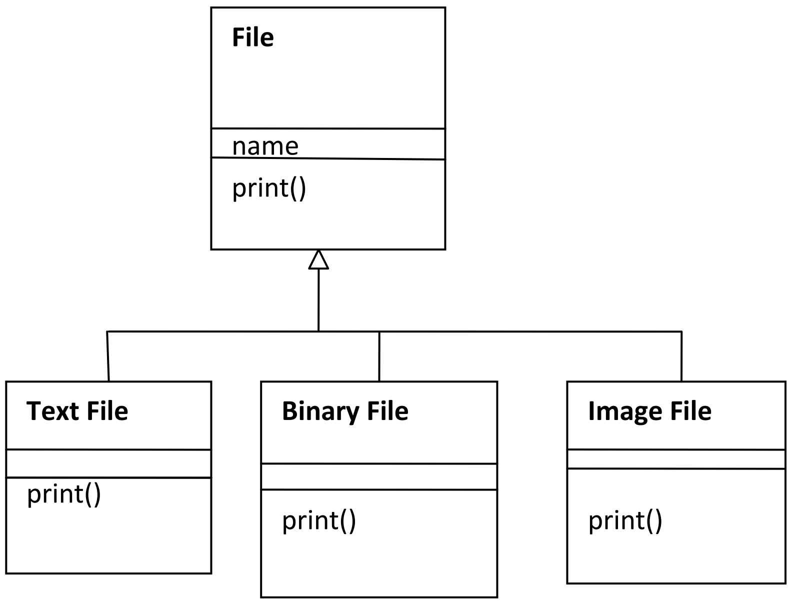


FIGURE 2 . 4 5 Introducing an Image File class.

Following is a list of the benefits of type promotion and polymorphism:

• They allow clients to be unaware of specific subclasses. The client is given a reference to the more abstract superclass type (avoiding a form of cou-pling called subclass coupling or subtype coupling).

• They shield the client from changes in the subclasses. Adding or removing subclasses does not affect the client. Changing existing subclasses does not affect the client (as long as the interface of the superclass does not change).

• In some languages, they can lessen the need for recompilation. As long as the public interface of the superclass is unchanged, the client need not be recompiled. This is a major issue in large systems (Lakos 1996, 327-471). A client should therefore refer to the most general type possible.

Class Diagram and Object Diagram

Class Diagram :

- A class diagram is a static representation of your system. It shows the types of classes, and how these classes are linked to each other
- A class diagram is a graph of Classifier elements connected by their various static relationships
- Class diagrams show the logical, static structure of a system and are central the Unified Modeling Language.

Object Diagram :

- A pictorial representation of the relationships between these instantiated classes at any point of time (called objects) is called an "Object diagram."
- Class diagrams can contain objects, so a class diagram with objects and no classes is an "object diagram."
- An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time.
- Object diagrams play a smaller role in UML. They are used primarily for documenting test cases and scenarios

Aggregation and Specialization

Aggregation :

- Aggregation is to create new functionality by taking other classes and combining them into a new class
- An aggregation is the "a part of" relationship in which objects represents the components in same assembly. Aggregation may be the special form of Association.
- With aggregation, you get to choose either implementation or interface, or both -- but you're not forced into either. The functionality of an object is left up to the object itself
- Example (aggregation) : A Paragraph consists of many sentences. Here Paragraph contains sentences.

Specialization :

- Specialization is relationship between the classes.
- Specialization is the *is-a-kind-of* relationship, in which the specialization is the subclass, or subtype.
- Example:
  If we have class Vehicle, Automobile and Truck. Then Automobile and Truck *is-a-kind-of* Vehicle.
  Vehicle is a superclass and Automobile and Truck are subclasses.

State and Event

Events:

- State machines communicate by sending events to one another.
- Events are typically treated as operations.
- In some cases, an event may corrospond to a signal, such as a hardware interrupt. In other situation, it may corrospond to a message, such as a method invocation or an inter-process message.
- The term event refers to the type of occurrence rather than to any concrete instance of that occurrence. For example, Keystroke is an event for the keyboard, but each press of a key is not an event but a concrete instance of the Keystroke event.
- An event can have associated parameters, allowing the event instance to convey not only the occurrence of some interesting incident but also quantitative information regarding that occurrence. For example, the Keystroke event generated by pressing a key on a computer keyboard has associated parameters that convey the character scan code as well as the status of the Shift, Ctrl, and Alt keys.

State :

- A state is an abstraction of an object's state. It can represent particular values for a subset of the object's attributes an links.
- A transition between states, therefore, represents some changes in those values.
- A state captures the relevant aspects of the system's history very efficiently. For example, when you strike a key on a keyboard, the character code generated will be either an uppercase or a lowercase character, depending on whether the Caps Lock is active.
- A state is depicted in UML as a capsule with name and/or a set of actions.

Macro vs micro process

Micro Processes	Macro Processes
A. The micro-processes of object-oriented development is largely driven by the stream of scenarios and architectural products. B. It represents the daily activities of the individual developer or a small team of developer. C. It applies equally to the software engineering and software architecture. D. The traditional phases of analysis and design are intentionally blurred, and the process is under opportunistic control.	A. The macro process serves as the controlling framework for the micro process. B. It represents the activities of the entire development team on the scale of weeks to months at a time. C. Many elements of the macro process are simply sound software management practice, and so apply equally to object-oriented as well as non-object-oriented systems. D. The traditional phases of analysis and design are to a large extent retained, and the process is reasonably well defined.

White vs black

- A white-box framework allows for application-specific implementation through inheritance (by offering set of abstractions whose interfaces are defined and implemented).
  A black-box framework allows for customization through additional classes that implement the interfaces defined in the framework, i.e., you just plug in classes that conform to the interface or role.

- With white-box frameworks you must understand the classes in them in order to tailor them for an application and in black-box frameworks you must understand what interfaces have to be implemented.

  White box Framework
  - consists of a set of classes that define abstractions.
  - you define subclasses inherit that abstraction.
  - inheritance of application.
  - you must understand the implementation details of the classes in the framework.
  - subclasses add state and behaviour specific to application.
  - some framework classes are abstract, many are concrete and some can be used as it is.

Black box framework

- consists of a set of classes that operate on specific interfaces.
- each interface defines a *role*.
- you introduce classes that play those roles by implementing the interfaces.
- interface inheritance and polymorphism as principle specialization mechanism.
- you are “plugging in” classes to a set of interfaces.
- Black-box frameworks are easier to use than white-box.
- You must understand only interfaces.