5. Modeling a System's Logical Structure: Advanced Class Diagrams

Shaoning Zeng, http://zsn.cc

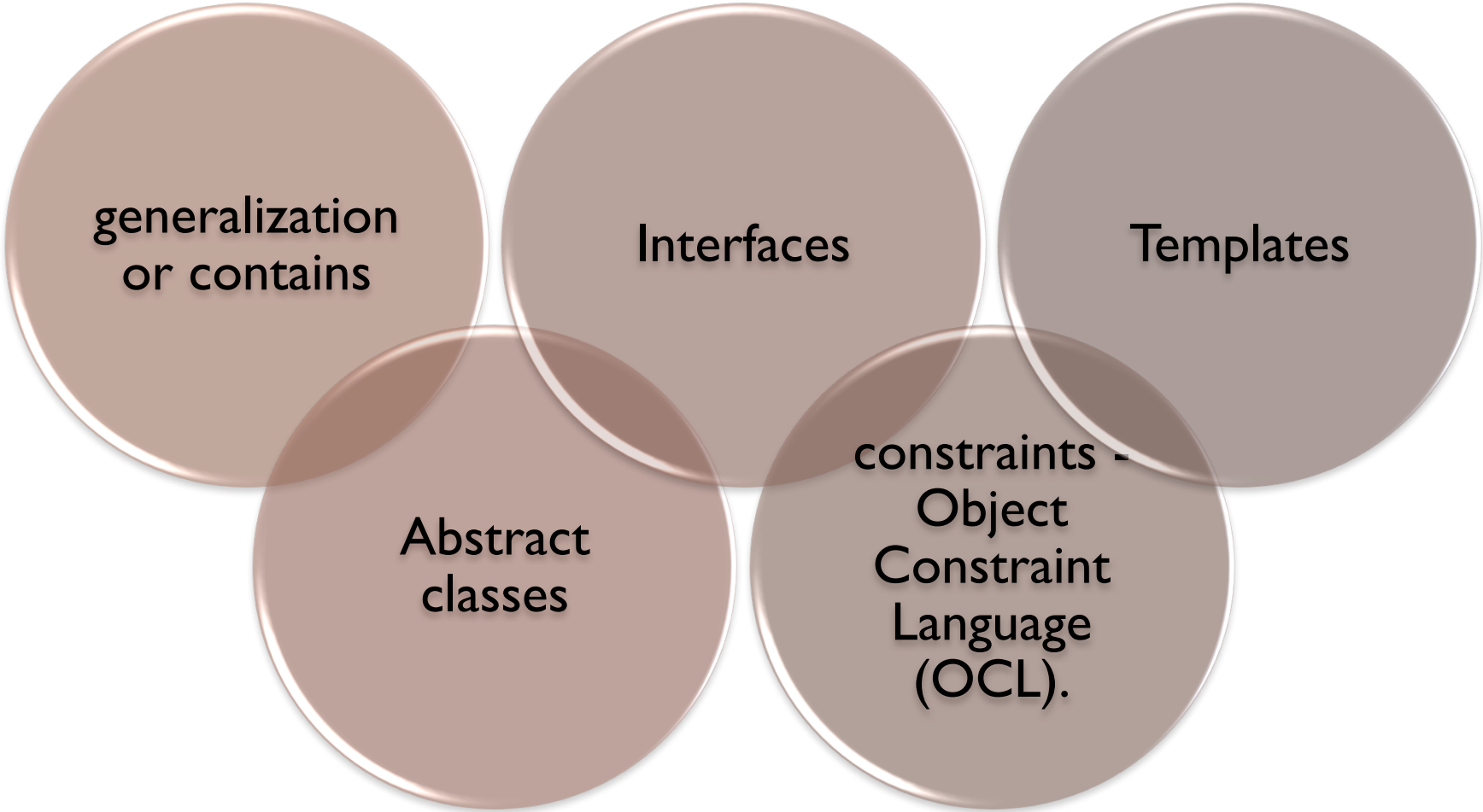
What we Learned?

- 4. Modeling a System's Logical Structure: Introducing Classes and Class Diagrams
  - 4.1. What Is a Class?
  - 4.2. Getting Started with Classes in UML
  - 4.3. Visibility
  - 4.4. Class State: Attributes
  - 4.5. Class Behavior: Operations
  - 4.6. Static Parts of Your Classes

5. Modeling a System's Logical Structure: Advanced Class Diagrams

- 5.1. Class Relationships 类的关系
- 5.2. Constraints 约束
- 5.3. Abstract Classes 抽象类
- 5.4. Interfaces 接口
- 5.5. Templates 模板

5. Modeling a System's Logical Structure: Advanced Class Diagrams



5. Modeling a System's Logical Structure: Advanced Class Diagrams

- For starters, classes can have relationships to one another.
  - A class can be a type of another class generalization or it can contain objects of another class in various ways depending on how strong the relationship is between the two classes.
- Abstract classes help you to partly declare a class's behavior,
  - allowing other classes to complete the missing abstract bits of behavior as they see fit.
- Interfaces take abstract classes one stage further
  - by specifying only the needed operations of a class but without any operation implementations.
- You can even apply constraints to your class diagrams that
  - describe how a class's objects can be used with the Object Constraint Language (OCL).
- Templates complete the picture
  - by allowing you to declare classes that contain completely generic and reusable behavior.
  - With templates , you can specify what a class will do and then wait as late as runtime if you choose to decide which classes it will work with.

5.1. Class Relationships

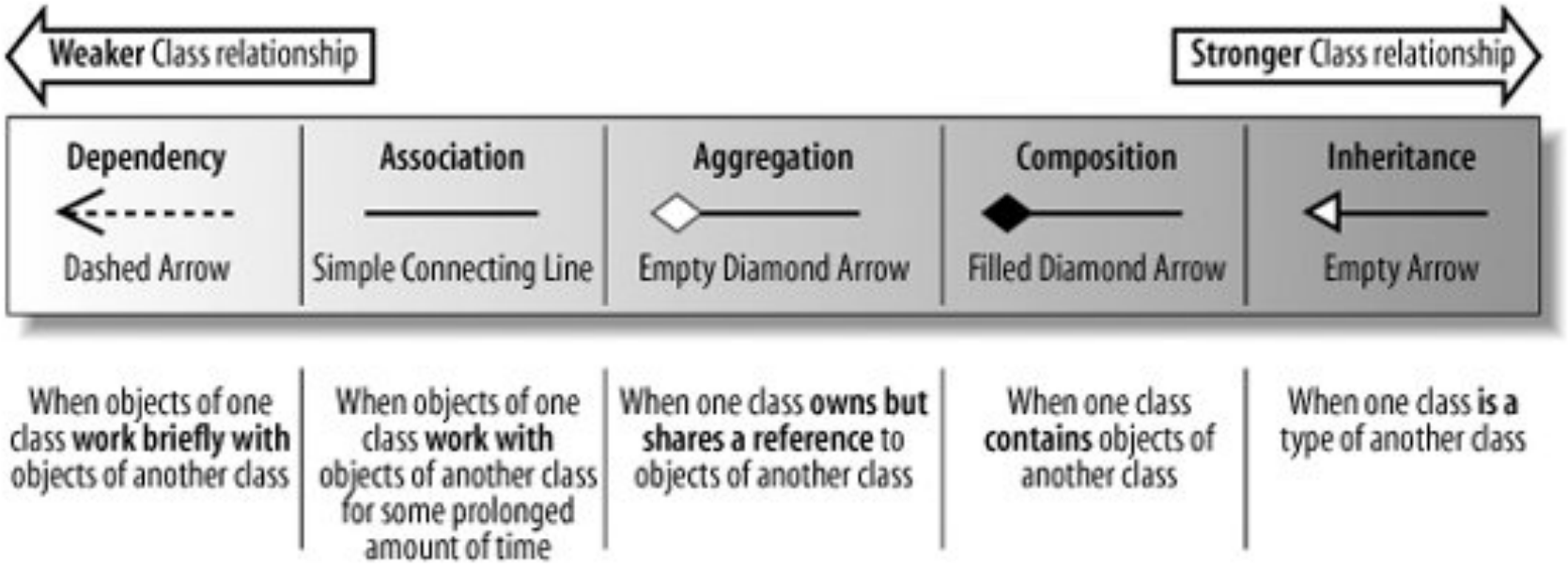


Figure 5-1. UML offers five different types of class relationship

5.1.1. Dependency

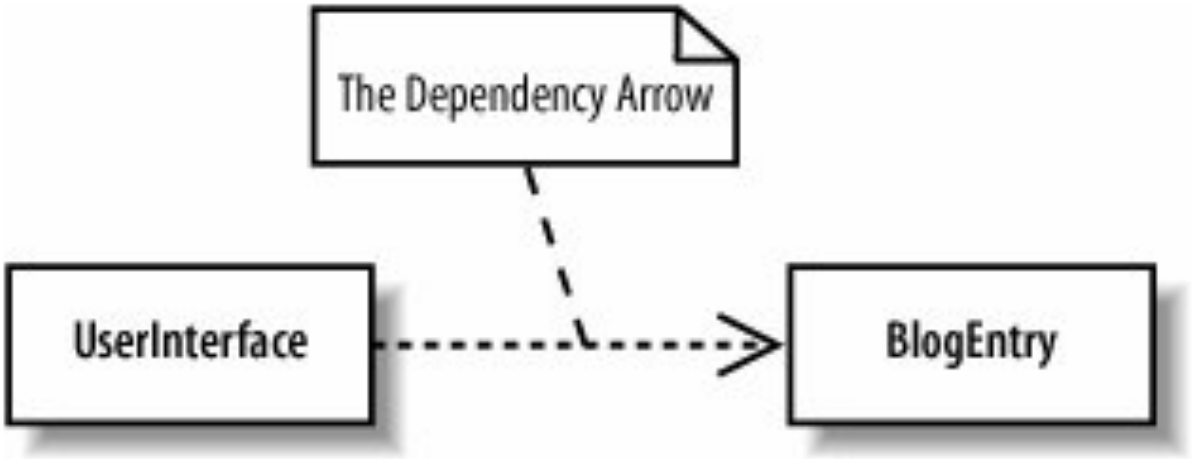
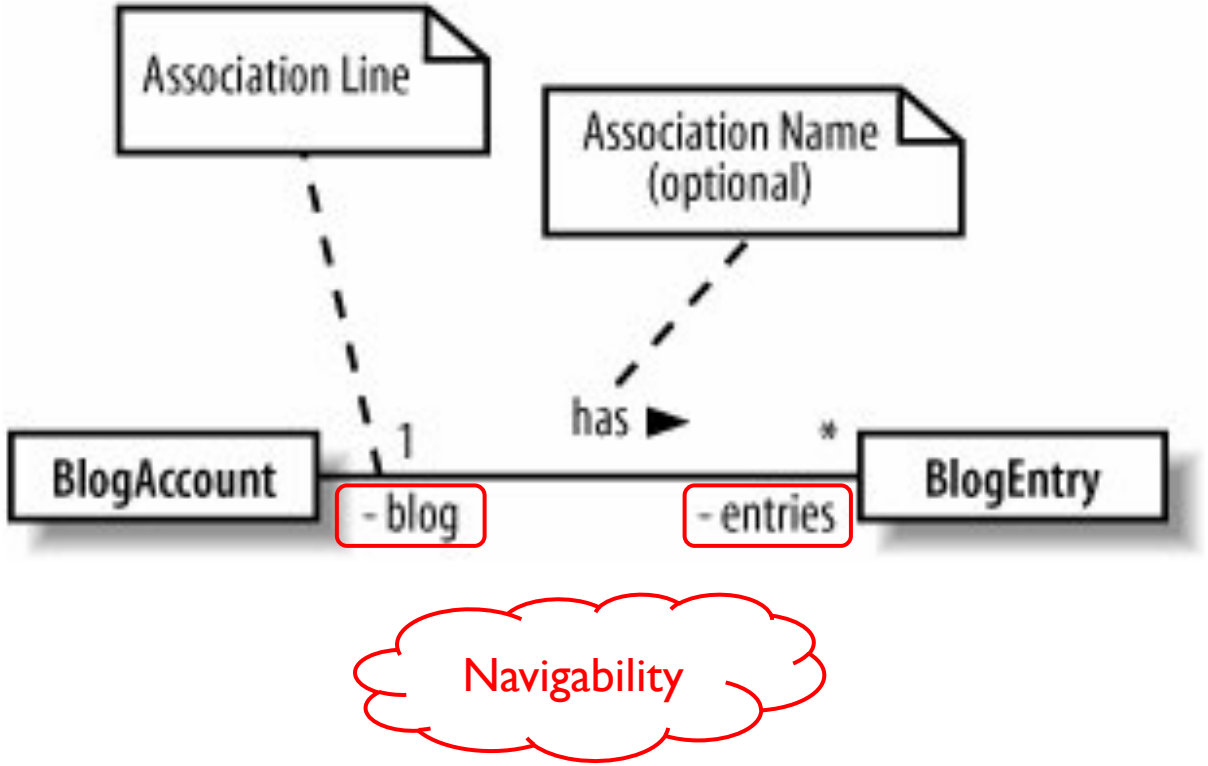


Figure 5-2. The UserInterface is dependent on the BlogEntry class because it will need to read the contents of a blog's entries to display them to the user

5.1.2. Association 关联

- Although dependency simply allows one class to use objects of another class, association means that a class will actually contain a reference to an object, or objects, of the other class in the form of an attribute.
- If you find yourself saying that a class works with an object of another class, then the relationship between those classes is a great candidate for association rather than just a dependency.
- Association is shown using a simple line connecting two classes.

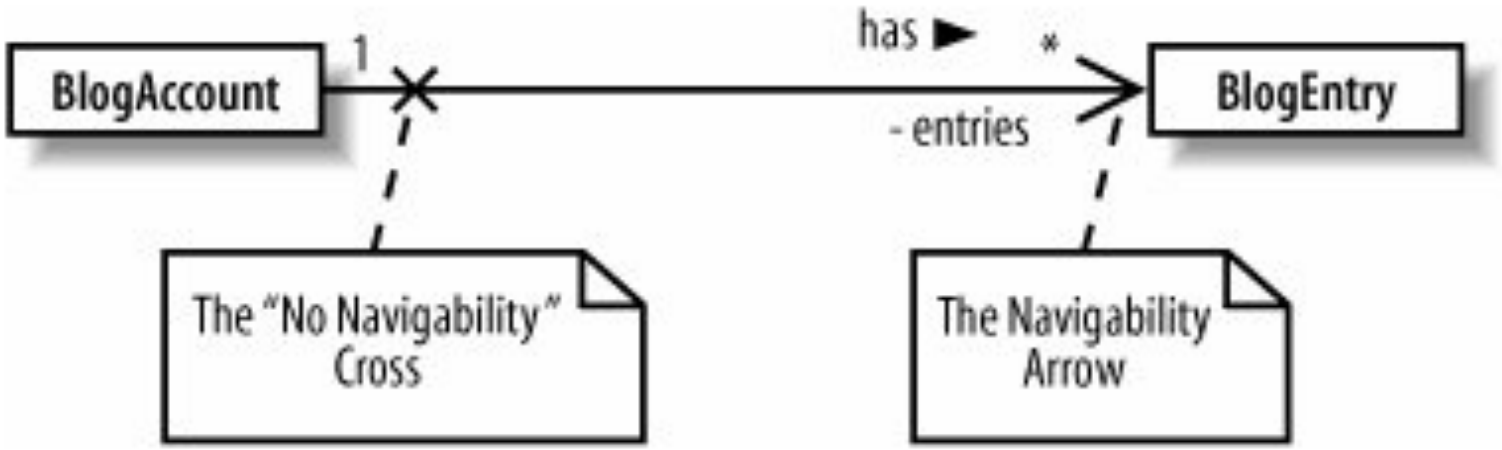
Figure 5-3. The BlogAccount class is optionally associated with zero or more objects of the BlogEntry class; the BlogEntry is also associated with one and only one BlogAccount



Example 5-1. The BlogAccount and BlogEntry classes without navigability applied to their association relationship

```
public class BlogAccount {

   // Attribute introduced thanks to the association with the BlogEntry class
   private BlogEntry[] entries;

   // ... Other Attributes and Methods declared here ...
}

public class BlogEntry {

   // Attribute introduced thanks to the association with the Blog class
   private BlogAccount blog;

   // ... Other Attributes and Methods declared here ...
}
```

Figure 5-4. If we change Figure 5-3 to incorporate the navigability arrow, then we can declare that you should be able to navigate from the blog to its entries



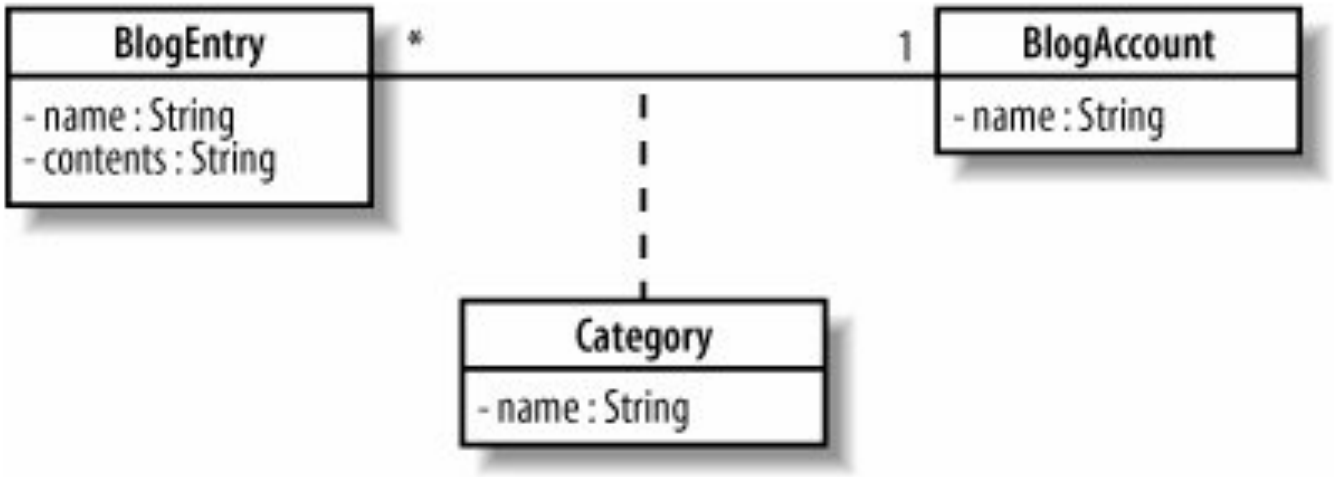
Example 5-2. With navigability applied, only the BlogAccount class contains an association introduced attribute

```
public class BlogAccount {

   // Attribute introduced thanks to the association with the BlogEntry class
   private BlogEntry[] entries;

   // ... Other Attributes and Methods declared here ...
}

public class BlogEntry
{
   // The blog attribute has been removed as it is not necessary for the
   // BlogEntry to know about the BlogAccount that it belongs to.

   // ... Other Attributes and Methods declared here ...
}
```

Association classes

- Sometimes an association itself introduces new classes.
- Association classes are particularly useful in complex cases when you want to show that a class is related to two classes because those two classes have a relationship with each other.

Figure 5-5. A BlogEntry is associated with an Author by virtue of the fact that it is associated with a particular BlogAccount



Example 5-3. One method of implementing the BlogEntry to BlogAccount relationship and the associated Category class in Java

```
public class BlogAccount {
   private String name;
   private Category[] categories;
   private BlogEntry[] entries;
}

public class Category {
   private String name;
}

public class BlogEntry {
   private String name;
   private Category[] categories
}
```

5.1.3. Aggregation 聚合

- Aggregation is really just a stronger version of association and is used to indicate that a class actually owns but may share objects of another class.
- Aggregation is shown by using an empty diamond arrowhead next to the owning class

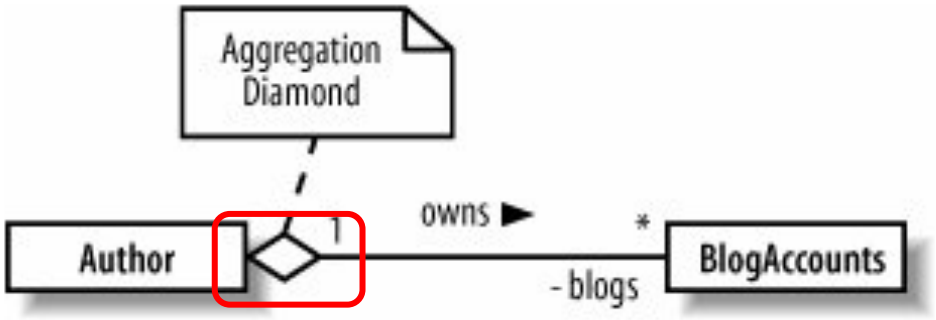


Figure 5-6. An aggregation relationship can show that an Author owns a collection of blogs

5.1.4. Composition 组合

- Moving one step further down the class relationship line, composition is an even stronger relationship than aggregation, although they work in very similar ways.
- Composition is shown using a closed, or filled, diamond arrowhead.

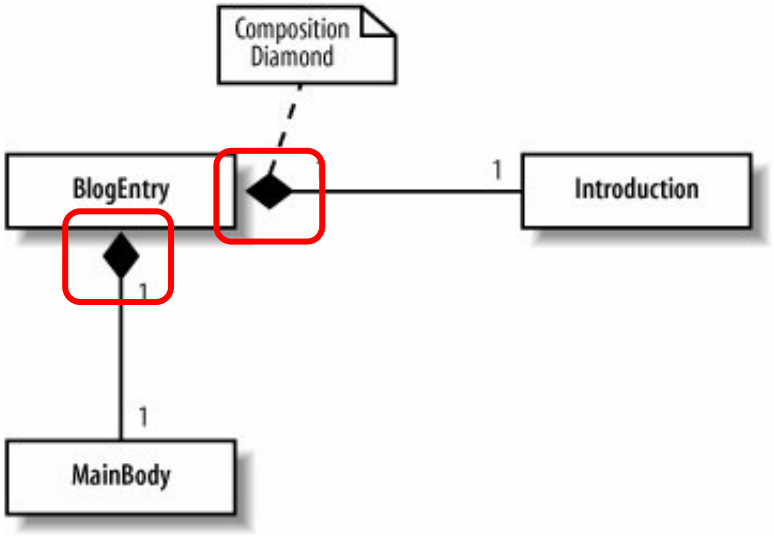
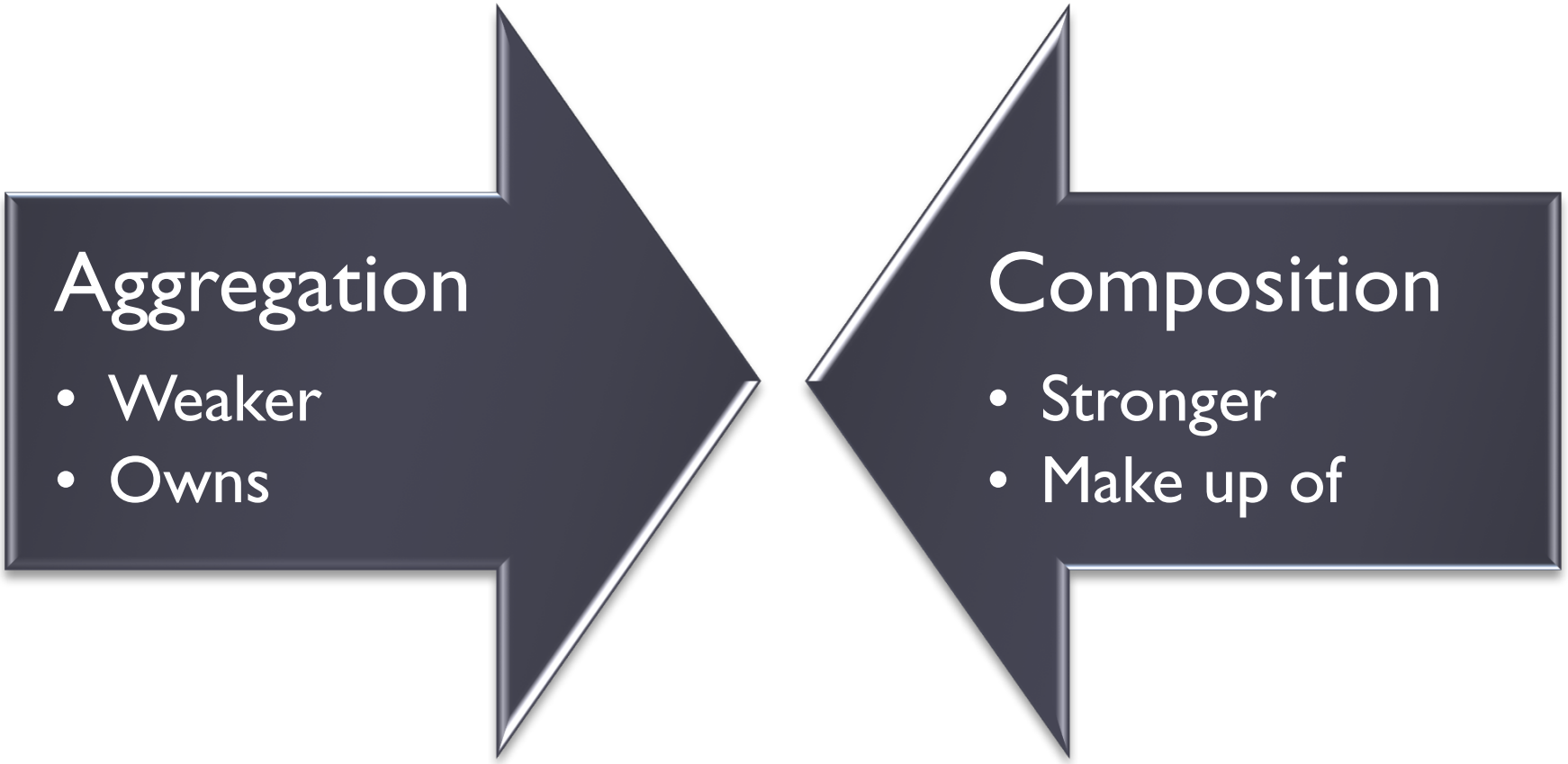


Figure 5-7. A BlogEntry is made up of an Introduction and a MainBody

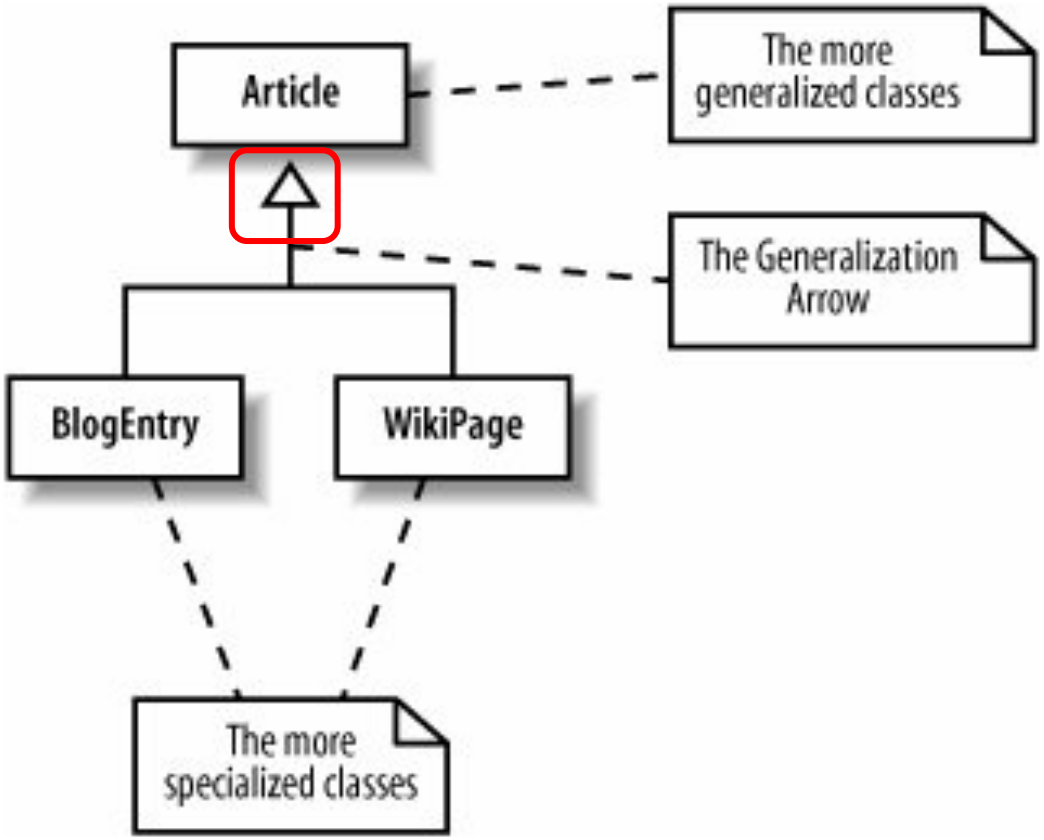
Aggregation v.s. Composition



5.1.5. Generalization (Otherwise Known as Inheritance) 泛化 | 继承

- Generalization and inheritance are used to describe a class that is a type of another class.
- The terms has a and is a type of have become an accepted way of deciding whether a relationship between two classes is aggregation or generalization for many years now.
  - If you find yourself stating that a class has a part that is an object of another class, then the relationship is likely to be one of association, aggregation, or composition.
  - If you find yourself saying that the class is a type of another class, then you might want to consider using generalization instead.
- In UML, the generalization arrow is used to show that a class is a type of another class.

Figure 5-8. Showing that a BlogEntry and WikiPage are both types of Article



Generalization and specialization 泛化与具体化

- The more generalized class that is inherited from at the arrow end of the generalization relationship, Article in this case is often referred to as the parent, base, or superclass.
- The more specialized classes that do the inheriting BlogEntry and WikiPage in this case are often referred to as the children or derived classes.
  - The specialized class inherits all of the attributes and methods that are declared in the generalized class and may add operations and attributes that are only applicable in specialized cases.
- The key to why inheritance is called generalization in UML is in the difference between what a parent class and a child class each represents.
  - Parent classes describe a more general type, which is then made more specialized in child classes .

5.1.5.1. Generalization and implementation reuse

- A child class inherits and reuses all of the attributes and methods that the parent contains and that have public, protected, or default visibility. So, generalization offers a great way of expressing that one class is a type of another class, and it offers a way of reusing attributes and behavior between the two classes.
- That makes generalization look like the answer to your reuse prayers, doesn't it? Just hold on a second!
  - If you are thinking of using generalization just so you can reuse some behavior in a particular class, then you probably need to think again.
  - Since a child class can see most of the internals of its parent, it becomes tightly coupled to its parent's implementation.

5.1.5.1. Generalization and implementation reuse (Cont.)

- One of the principles of good object-oriented design is to avoid tightly coupling classes so that when one class changes, you don't end up having to change a bunch of other classes as well.
  - Generalization is the strongest form of class relationship because it creates a tight coupling between classes.
  - Therefore, it's a good rule of thumb to use generalization only when a class really is a more specialized type of another class and not just as a convenience to support reuse.

Do not use generalization for reuse purpose!

5.1.5.2. Multiple inheritance 多重继承

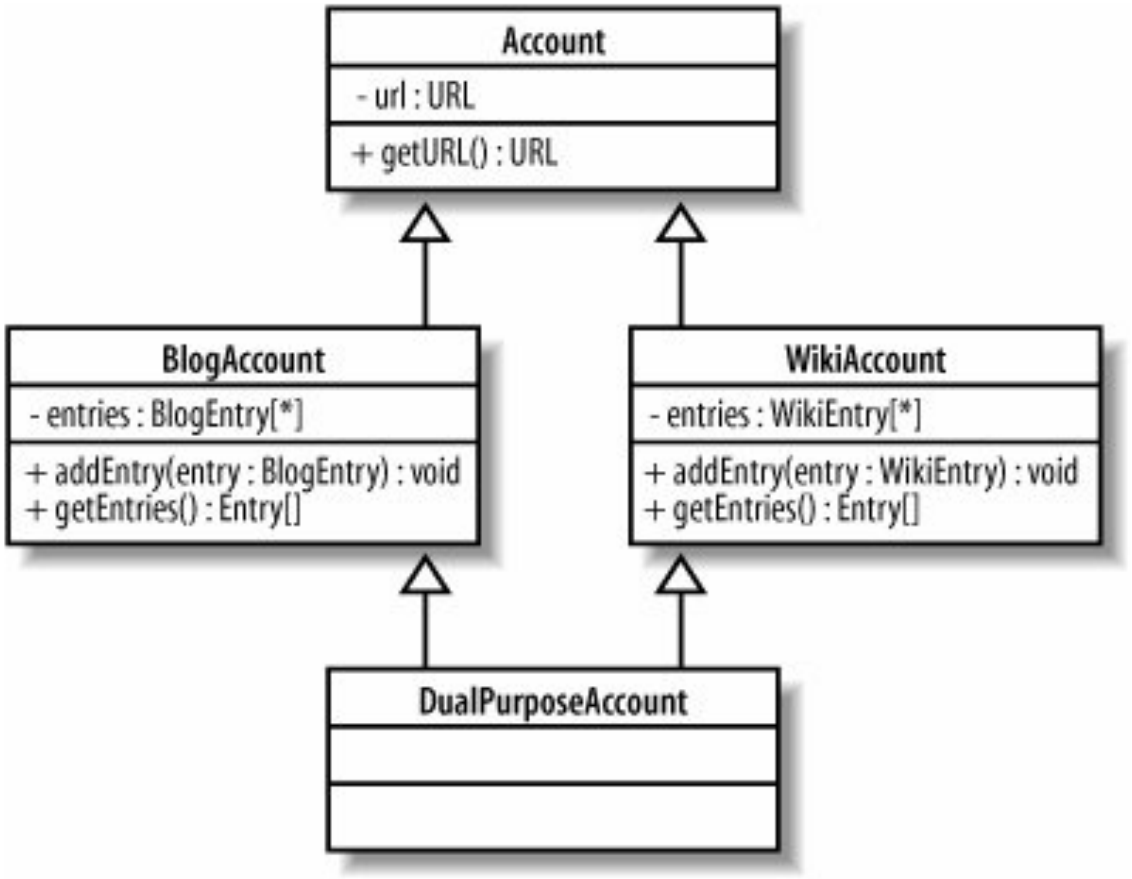
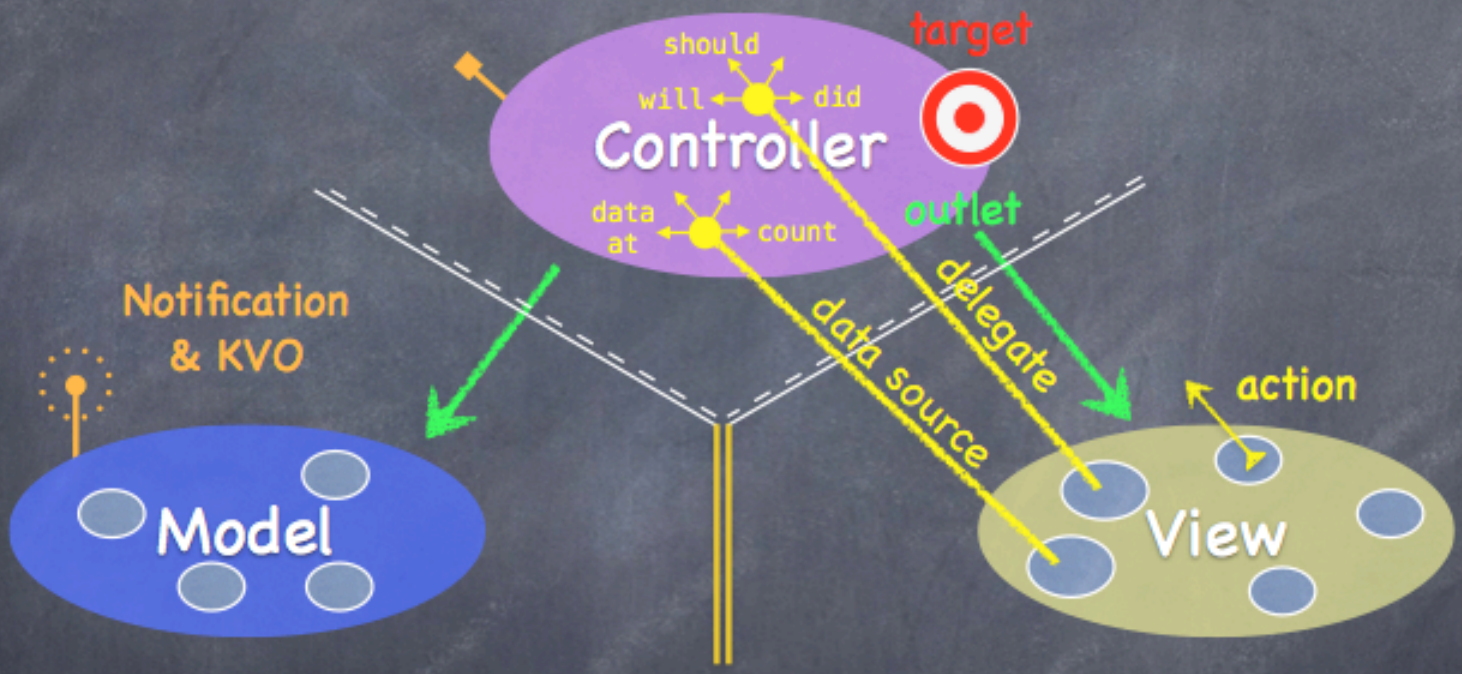


Figure 5-9. The DualPurposeAccount is a BlogAccount and a WikiAccount all combined into one

5.1.5.2. Multiple inheritance

- Although multiple inheritance is supported in UML, it is still not considered to be the best practice in most cases. This is mainly due to the fact that multiple inheritance presents a complicated problem when the two parent classes have overlapping attributes or behavior.
- The answers to these question are unfortunately often hidden in implementation details.
  - For example, if you were using the C++ programming language, which supports multiple inheritance, you would use the C++ language's own set of rules about how to resolve these conflicts.
  - Another implementation language may use a different set of rules completely.
  - Because of these complications, multiple inheritance has become something of a taboo (禁忌) subject in object-oriented software development to the point where the current popular development languages, such as Java and C#, do not even support it.
  - However, the fact remains that there are situations where multiple inheritance can make sense and be implemented in languages such as C++, for example so UML still needs to support it.

MVC

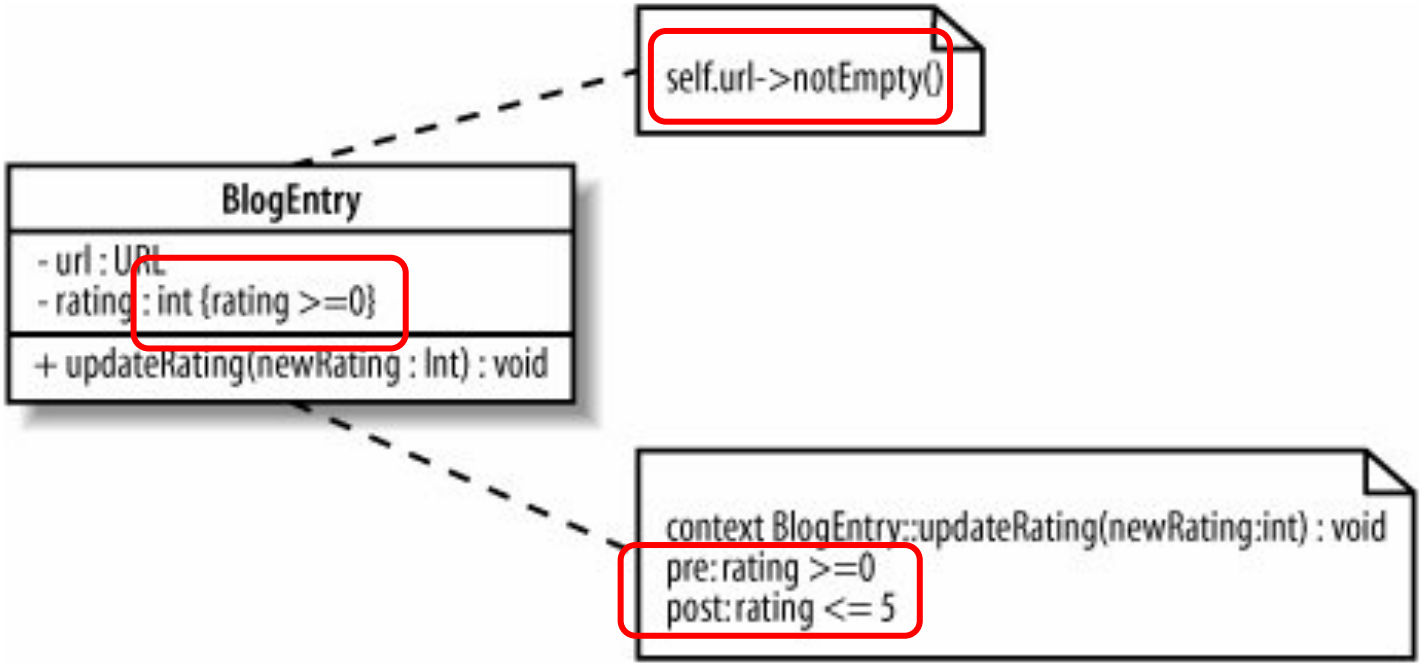


Now combine MVC groups to make complicated programs ...

5.2. Constraints 约束

- There are three types of constraint that can be applied to class members using OCL:
  - Invariants 不变性: An *invariant* is a constraint that must always be true; otherwise the system is in an invalid state.
    - Invariants are defined on class attributes.
  - Preconditions 前置条件: A *precondition* is a constraint that is defined on a method and is checked before the method executes.
    - Preconditions are frequently used to validate input parameters to a method.
  - Postconditions后置条件: A *postcondition* is also defined on a method and is checked after the method executes.
    - Postconditions are frequently used to describe how values were changed by a method.
- Constraints are specified using either the OCL statement in curly brackets next to the class member or in a separate note

Figure 5-10. Three constraints are set on the BlogEntry class: self.url>notEmpty( ) and rating>=0 are both invariants, and there is a postcondition constraint on the updateRating(..) operation



5.3. Abstract Classes

Figure 5-11. Using regular operations, the Store class needs to know how to store and retrieve a collection of articles

Abstract Methods

- To indicate that the implementation of the store(..) and retrieve(..) operations is to be left to subclasses by declaring those operations as abstract, write their signatures in italics.

Figure 5-12. The store(..) and retrieve(..) operations do not now need to be implemented by the Store class

Abstract Classes

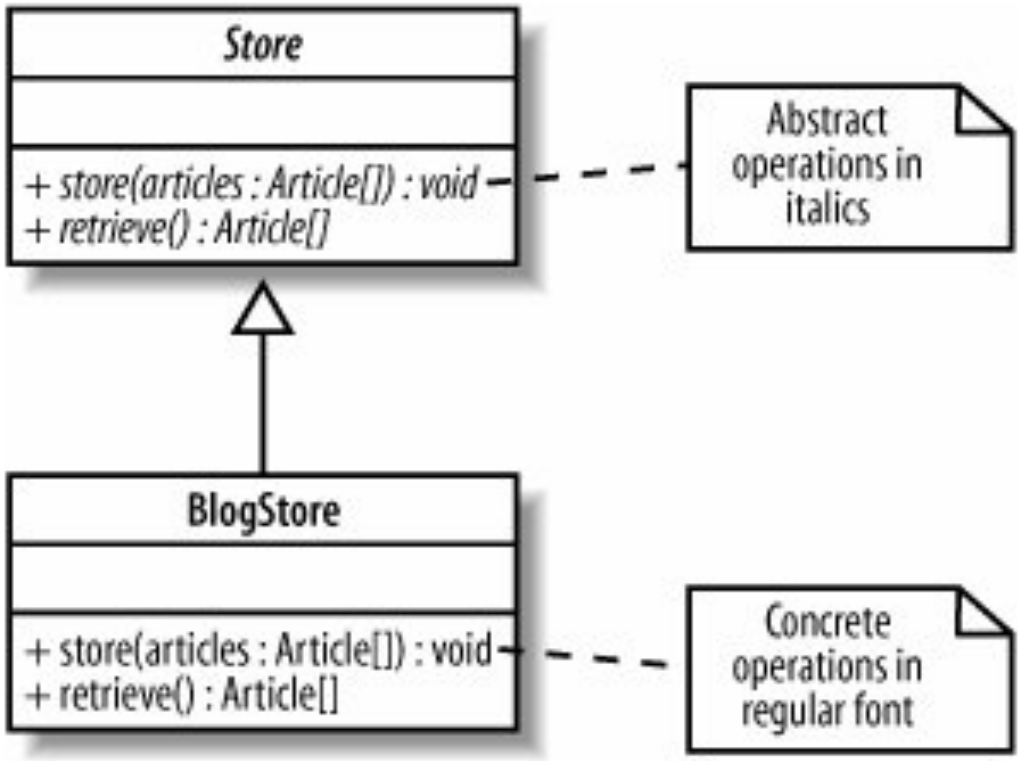
- An abstract operation does not contain a method implementation and is really a placeholder that states, "I am leaving the implementation of this behavior to my subclasses."
- If any part of a class is declared abstract, then the class itself also needs to be declared as abstract by writing its name in italics.

Figure 5-13. The complete abstract Store class

Example 5-4. The problem of what code to put in the implementation of the play( ) operation is solved by declaring the operation and the surrounding class as abstract

```
public abstract class Store {
   public abstract void store(Article[] articles);
   public abstract Article[] retrieve(  );
}
```

Figure 5-14. The BlogStore class inherits from the abstract Store class and implements the store(..) and retrieve(..) operations; classes that completely implement all of the abstract operations inherited from their parents are sometimes referred to as "concrete"



Example 5-5. The BlogStore class completes the abstract parts of the Store class

```
public abstract class Store {

   public abstract void store(Article[] articles);
   public abstract Article[] retrieve(  );
}

public class BlogStore {

   public void store(Article[] articles) {
      // Store away the blog entries here ...
   }

   public Article[] retrieve(  ) {
      // Retrieve and return the stored blog entries here...
   }
}
```

Example 5-6. You can create objects of non-abstract clas eds to imp rited

```
public abstract class Store {

   public abstract void store(Article[] articles);
   public abstract Article[] retrieve(  );
}

public class BlogStore {

   public void store(Article[] articles) {
      // Store away the blog entries here ...
   }

   public Article[] retrieve(  ) {
      // Retrieve and return the stored blog entries here...
   }
}
public class MainApplication {

   public static void main(String[] args) {

      // Creating an object instance of the BlogStore class.
      // This is totally fine since the BlogStore class is not abstract.
      BlogStore store = new BlogStore(  );
      blogStore.store(new Article[]{new BlogEntry(  )});
      Article[] articlesInBlog = blogStore.retrieve(  );

      // Problem! It doesn't make sense to create an object of
      // an abstract class because the implementations of the
      // abstract pieces are missing!
      Store store = new Store(  ); // Compilation error here!
   }
}
```

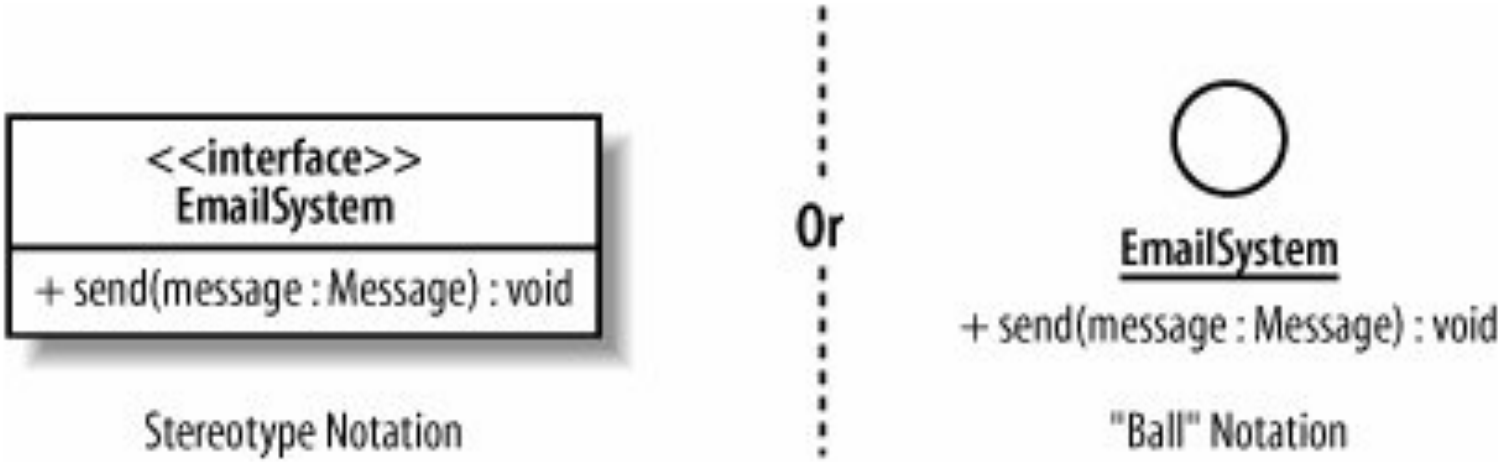
Problem 使用抽象类的问题

- Abstract classes are a very powerful mechanism that enable you to define common behavior and attributes, but they leave some aspects of how a class will work to more concrete subclasses.
  - A great example of where abstract classes and interfaces are used is when defining the generic roles and behavior that make up design patterns.
- However, to implement an abstract class, you have to use inheritance; therefore, you need to be aware of all the baggage that comes with the strong and tightly coupling generalization relationship.

5.4. Interfaces

- An interface is a collection of operations that have no corresponding method implementations, very similar to an abstract class that contains only abstract methods.
  - In some software implementation languages, such as C++, interfaces are implemented as abstract classes that contain no operation implementations.
  - In newer languages, such as Java and C#, an interface has its own special construct.
- Think of an interface as a very simple contract that declares,
  - "These are the operations that must be implemented by classes that intend to meet this contract."
- Sometimes an interface will contain attributes as well, but in those cases, the attributes are usually static and are often constants.
- In UML, an interface can be shown as a stereotyped class notation or by using its own ball notation.

Figure 5-15. Capturing an interface to an EmailSystem using the stereotype and "ball " UML notation; unlike abstract classes, an interface does not have to show that its operations are not implemented, so it doesn't have to use italics



Example 5-7. The EmailSystem interface is implemented in Java by using the interface keyword and contains the single send(..) operation signature with no operation implementation

```
public interface EmailSystem {
    public void send(Message message);
}
```

Figure 5-16. The SMTPMailSystem class implements, or realizes, all of the operations specified on the EmailSystem interface

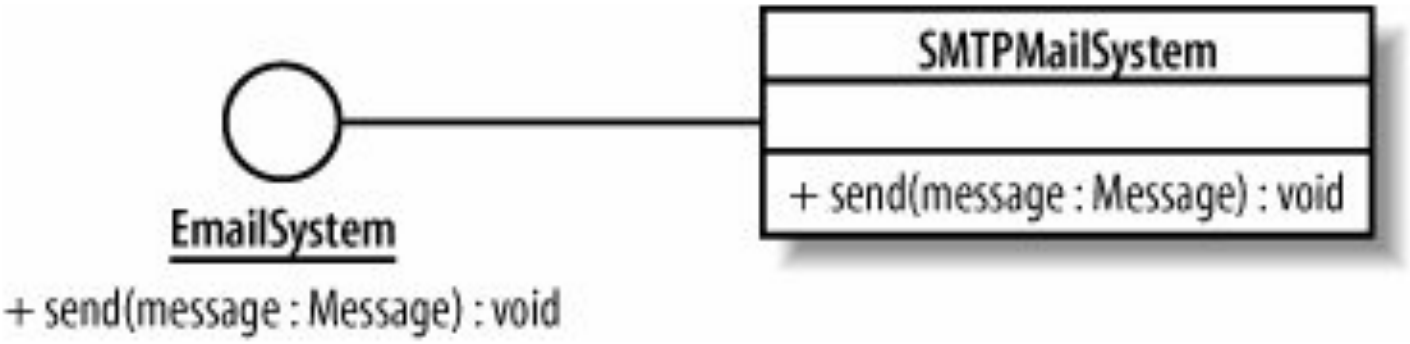
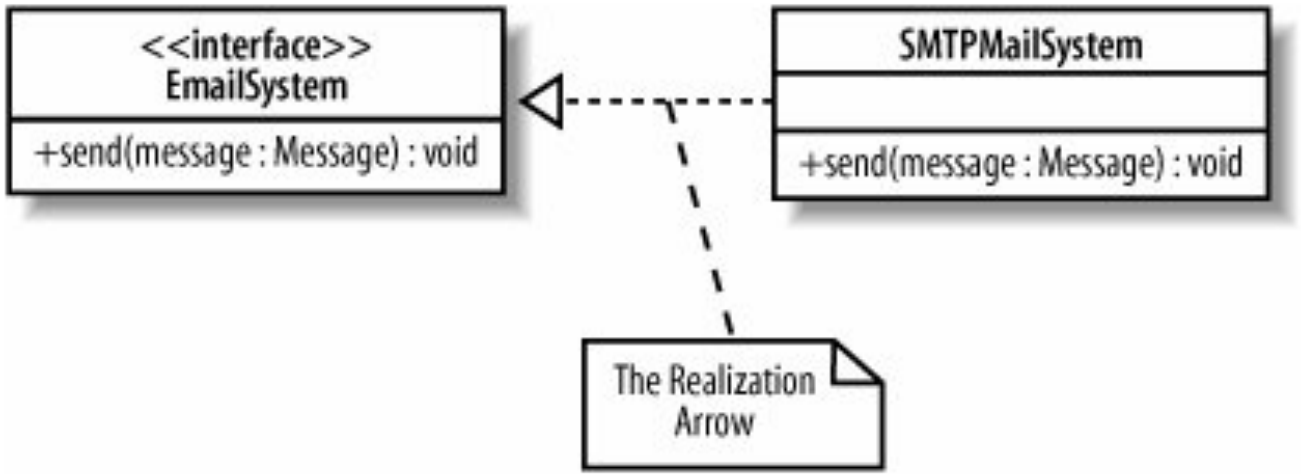


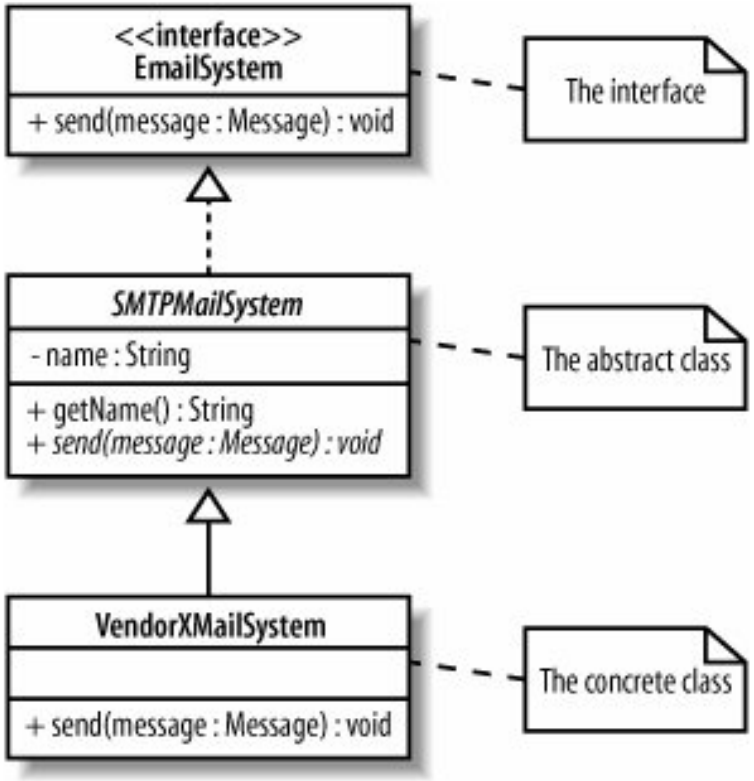
Figure 5-17. The realization arrow specifies that the SMTPMailSystem realizes the EmailSystem interface



Example 5-8. Java classes realize interfaces using the implements keyword

```
public interface EmailSystem
{
   public void send(Message message));
}

public class SMTPMailSystem implements EmailSystem
{
   public void send(Message message)
   {
      // Implement the interactions with an SMTP server to send the message
   }

   // ... Implementations of the other operations on the Guitarist class ...
}
```

Figure 5-18. Because the SMTPMailSystem class does not implement the send(..) operation as specified by the EmailSystem interface, it needs to be declared abstract; the VendorXMailSystem class completes the picture by implementing all of its operations



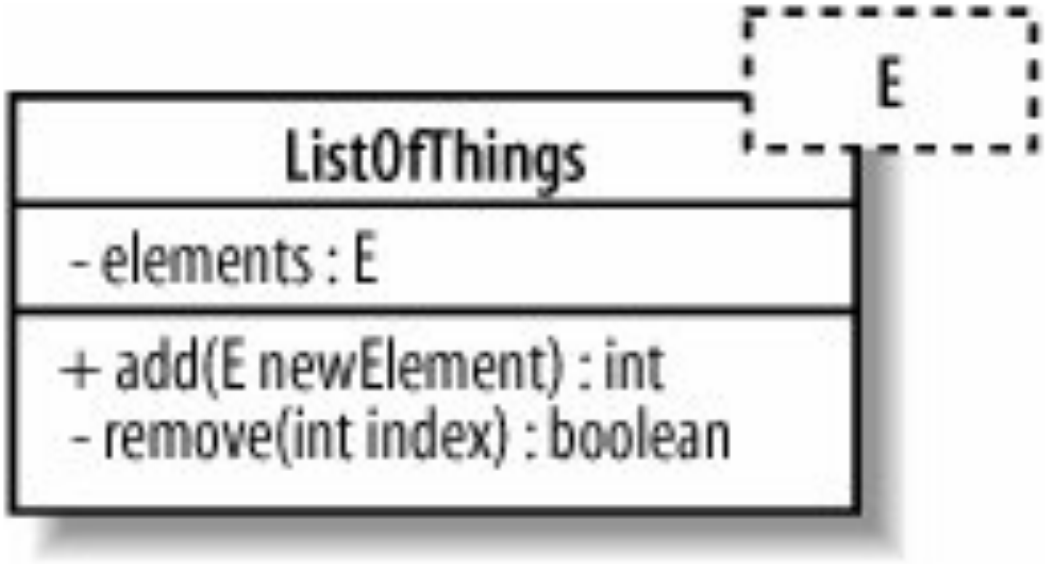
Using Interfaces

- It is good practice to de-couple dependencies between your classes using interfaces; some programming environments, such as the Spring Framework, enforce this interface-class relationship. 解耦
- The use of interfaces, as opposed to abstract classes, is also useful when you are implementing design patterns.
- In languages such as Java, you don't really want to use up the single inheritance relationship just to use a design pattern.
- A Java class can implement any number of interfaces, so they offer a way of enforcing a design pattern without imposing the burden of having to expend that one inheritance relationship to do it.

5.5. Templates 模板

- Templates are an advanced but useful feature of object orientation.
- A template or parameterized class, as they are sometimes referred to is helpful when you want to postpone (推迟) the decision as to which classes a class will work with.
- When you declare a template, it is similar to declaring, "I know this class will have to work with other classes, but I don't know or necessarily care what those classes actually end up being."

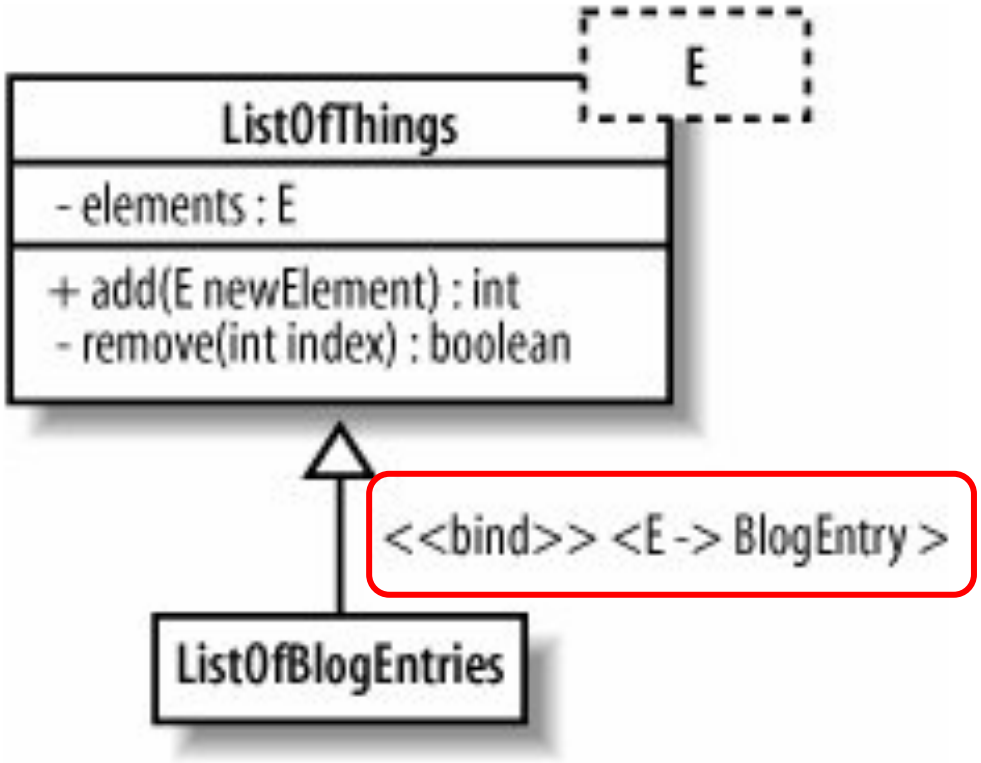
Figure 5-19. A template in UML is shown by providing an extra box with a dashed border to the top right of the regular class box



Lists

- Lists tend to be the most common examples of how to use templates , and with very good reason.
- Lists and their cousins, such as maps and sets, all store objects in different ways, but they don't actually care what classes those objects are constructed from.
- For this reason, one of the best real-world uses of templates is in the Java collection classes.
  - Prior to Java 5, the Java programming language did not have a means of specifying templates.
  - With the release of Java 5 and its generics feature, you can now not only create your own templates, but the original collection classes are all available to use as templates as well.

Figure 5-20. The ListOfThings class is subclassed into a ListOfBlogEntries, binding the single parameter E to the concrete BlogEntry class



Summary

- Chapter 4. Modeling a System's Logical Structure: Introducing Classes and Class Diagrams
  - 4.1. What Is a Class?
  - 4.2. Getting Started with Classes in UML
  - 4.3. Visibility
  - 4.4. Class State: Attributes
  - 4.5. Class Behavior: Operations
  - 4.6. Static Parts of Your Classes
- Chapter 5. Modeling a System's Logical Structure: Advanced Class Diagrams
  - 5.1. Class Relationships
  - 5.2. Constraints
  - 5.3. Abstract Classes
  - 5.4. Interfaces
  - 5.5. Templates

类建模练习

确定下面的关系分类（泛化、聚合或关联）

- 一个国家有一个首都
- 哲学家用餐时使用餐叉
- 文件要么是普通文件，要么是目录文件
- 文件包含记录
- 多边形由有序点集组成
- 绘制的对象是文本、几何对象或分组
- 某人在某个项目里使用一门计算机语言
- 调制解调器和键盘是输入/输出设备
- 类有多个属性
- 某人在某年为某队打球
- 路线连接两座城市
- 某学生选择了某教授的课程

绘制类图

- **为下面每一组类绘制一个类图。**
  - 每一个图至少增加10种关系（关联和泛化）。
  - 可以使用关联名和关联端名。
  - 可以使用限定关联和多样性。
  - 不需要显示属性或操作。
  - 绘图时可以增加必要的类。

UML类图练习1

School （学校）	Playground （操场）	Principal （校长）	School Board （校董事会）
Classroom （教室）	Book （书本）	Student （学生）	Teacher （教师）
Cafeteria （自助餐厅）	Restroom （公共厕所）	Computer （计算机）	Desk （课桌）
Chair （椅子）	Ruler （尺子）	Door （门）	Swing （秋千）

UML类图练习2

Automobile（汽车）	Engine（发动机）	Wheel（车轮）
Brake（刹车）	Brake Light（刹车灯）	Door（车门）
Battery（电池）	Muffler（消声器）	Tail Pipe（排气尾管）

UML类图练习3

Expression（表达式）

Constant（常量）

Variable（变量）

Function（函数）

Argument List（参数列表）

Statement（语句）

Computer Program（计算机程序）

UML类图练习4

File System
（文件系统）

File
（文件）

ASCII File
（ASCII文件）

Binary File
（二进制文件）

Directory File
（目录文件）

Disc
（磁盘）

Drive
（驱动器）

Track
（磁道）

Sector
（扇区）

UML类图练习5

Gas Furnace（煤气炉）

Blower（吹风机）

Blower Motor（吹风机马达）

Room Thermostat（室用恒温器）

Furnace Thermostat（恒温调节炉）

Humidifier（湿度调节器）

Humidity Sensor（湿度传感器）

Gas Control（煤气控制器）

Blower Control（吹风机控制器）

Hot Air Vent（热空气通风孔）

UML类图练习6

Chess Piece
（棋子）

Rank
（横排）

File
（纵列）

Square
（格子）

Board
（棋盘）

Move
（走棋）

Tree Of Moves
（对局棋谱）

UML类图练习7

Sink （水槽）	Freezer （冰柜）	Refrigerator （冰箱）	Table （桌子）
Light （灯）	Switch （开关）	Window （窗户）	Smoke Alarm （烟雾报警器）
Burglar Alarm （防盗报警器）	Cabinet （橱柜）	Bread （面包）	Cheese （干酪）
Ice （冰）	Door （门）	Kitchen （厨房）	

UML类图练习8

- 为支持分组的图形化文档编辑器绘制一个类图
  - 假定一篇文档由几张表单组成。
  - 每张表单包含文本、几何对象和分组等绘制对象。
  - 分组只是一组绘图对象，也可能包含其他分组。
  - 一个分组必须至少包含两个绘制对象。
  - 一个绘制对象至多是一个分组中的直接成员。
  - 几何对象包括圆、椭圆、矩形、直线和正方形。

What's next?

- 6. Bringing Your Classes to Life: Object Diagrams
  - 6.1. Object Instances
  - 6.2. Links
  - 6.3. Binding Class Templates
- 7. Modeling Ordered Interactions: Sequence Diagrams
  - 7.1. Participants in a Sequence Diagram
  - 7.2. Time
  - 7.3. Events, Signals, and Messages
  - 7.4. Activation Bars
  - 7.5. Nested Messages
  - 7.6. Message Arrows
  - 7.7. Bringing a Use Case to Life with a Sequence Diagram
  - 7.8. Managing Complex Interactions with Sequence Fragments

See you …