04834580 Software Engineering (Honor Track) 2024-25

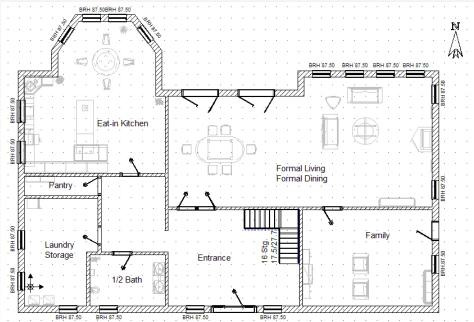
Modeling

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Definition (from SWEBOK [1])

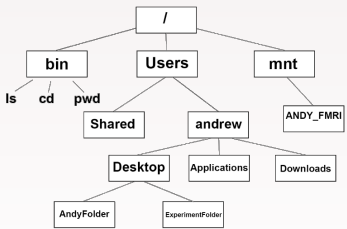
Modeling provides the software engineer with an organized and systematic approach for representing significant aspects of the software under study, facilitating decision-making about the software or elements, and communicating those significant decisions to others in the stakeholder communities.

Software is **invisible** *and unvisualizable. Geometric abstractions are powerful tools. The floor plan of a building helps both architect and client evaluate spaces, traffic flows, and views. Contradictions become obvious, omissions can be caught. Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in a geometric abstraction.* **The reality of software is not inherently embedded in space.** — *Fred Brooks [2]*

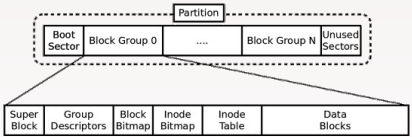


A model is an abstraction or simplification of a system. For example, a filesystem can be modeled

as a directory tree:



as blocks and partitions:



Unified Modeling Language (UML)

UML is a formal modeling language that contains

- a notation (a way of expressing the model), and
- a description of what that notation means (a meta-model).

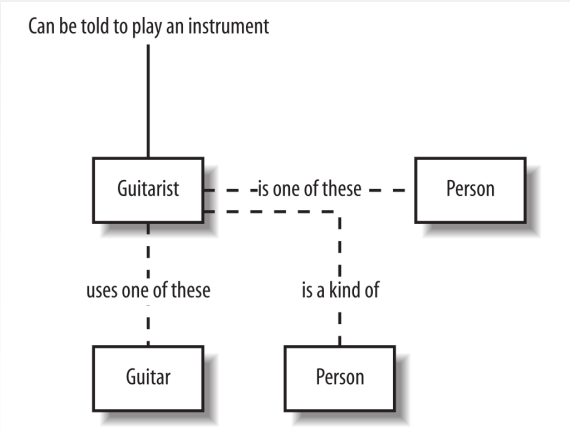
Formal means each element of the language has a strongly defined meaning, so you can be confident that when you model a particular facet of your system it will not be misunderstood.

Example: Guitarist Class [3]

北京大学

```java
public class Guitarist extends Person implements MusicPlayer {
    Guitar favoriteGuitar;
    public Guitarist (String name) {
        super(name);
    }
    // A couple of local methods for accessing the class's properties
    public void setInstrument(Instrument instrument) {
        if (instrument instanceof Guitar) {
            this.favoriteGuitar = (Guitar) instrument;
        }
        else {
            System.out.println("I'm not playing that thing!");
        }
    }
    public Instrument getInstrument() {
        return this.favoriteGuitar;
    }
    // Better implement this method as MusicPlayer requires it
    public void play() {
        System.out.println(super.getName() + "is going to do play the guitar now ...");
        if (this.favoriteGuitar != null) {
            for (int strum = 1; strum < 500; strum++) {
                this.favoriteGuitar.strum();
            }
            System.out.println("Phew! Finished all that hard playing");
        }
        else {
            System.out.println("You haven't given me a guitar yet!");
        }
    }
    // I'm a main program so need to implement this as well
    public static void main(String[] args) {
        MusicPlayer player = new Guitarist("Russ");
        player.setInstrument(new Guitar("Burns Brian May Signature"));
        player.play();
    }
}
```

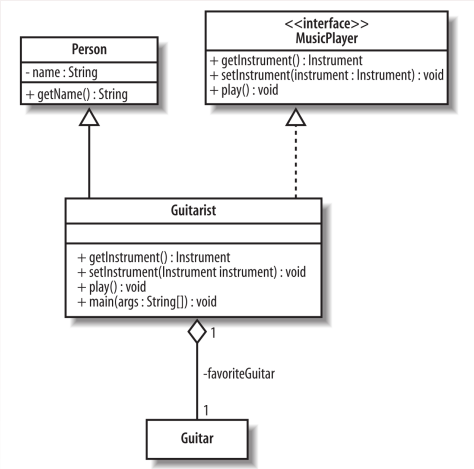
Informal languages suffer from the dual problem of **verbosity** and **ambiguity**.



Example: Formal UML Notation

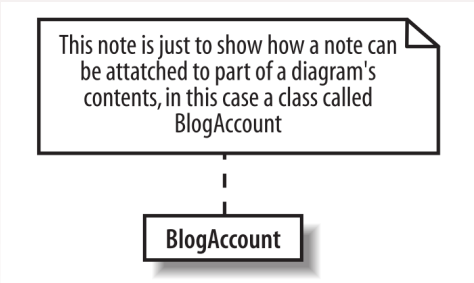
北京大学

Formal notation such as **class diagram** is informative and precise.



Type	What can be modeled?
Use Case	Interactions between your system and users/external systems.
Activity	Sequential and parallel activities within your system.
Class	Classes, types, interfaces, and the relationships between them.
Object	Object instances of the classes defined in class diagrams.
Sequence	The order of the interactions between objects.
Communication	The ways in which objects interact and connections.
Timing	Interactions between objects where timing is important.
Interaction Overview	Collect sequence, communication, and timing.
Composite Structure	The internals of a class or component.
Component	Components within your system and their interfaces.
Package	Hierarchy of groups of classes and components.
State Machine	The state of an object throughout its lifetime.
Deployment	How your system is deployed.

Notes can be attached to elements of diagrams.



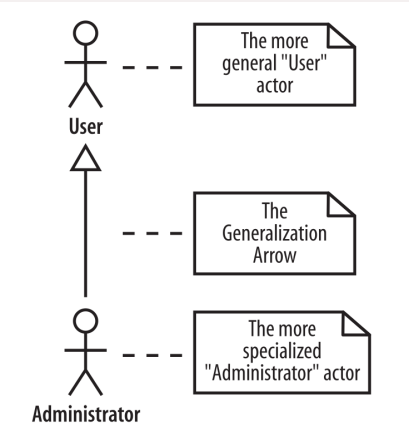
A **use case** is a case (or situation) where your system is used to fulfill one or more of your user's requirements; a use case captures a piece of functionality that the system provides.

Example requirement:

> *The content management system shall allow an administrator to create a new blog account, provided the personal details of the new blogger are verified using the author credentials database.*

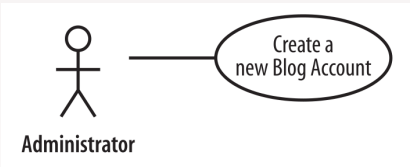
Administrator is an *actor*:

Administrator generalizes to User:



A use case is something that provides some measurable result to the user or an external system:

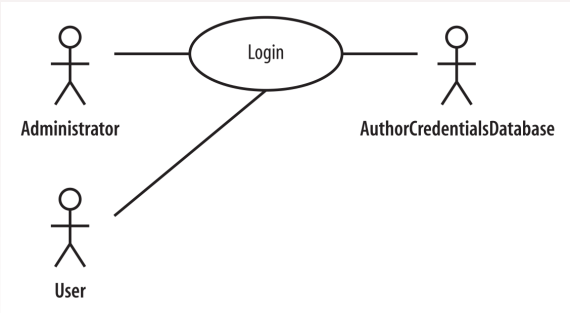
A communication line shows that an actor participating in a use case:



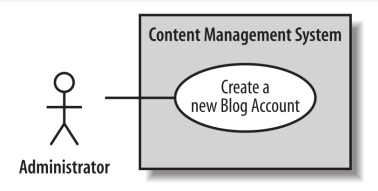
Use Case Diagrams: Multiple Actors

北京大学

A connection implies that an actor is simply *involved* in a use case, not to imply an information exchange in any particular direction or that the actor starts the use case.

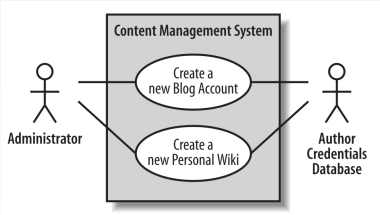


Administrator is outside of the system:

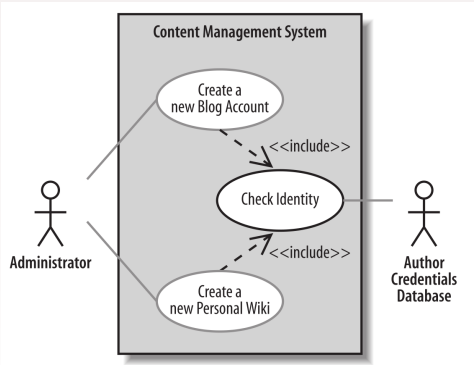


Another requirements:

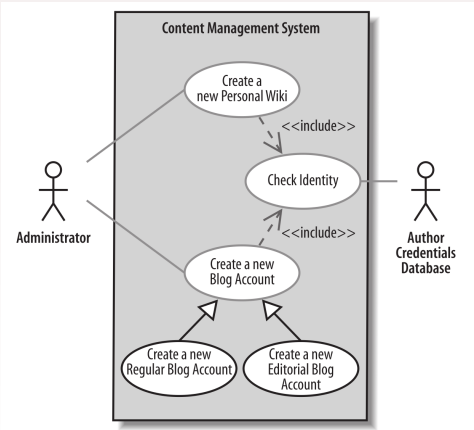
> *The content management system shall allow an administrator to create a new personal Wiki, provided the personal details of the applying author are verified using the Author Credentials Database.*



This repetitive behavior shared between two use cases is best separated and captured within a totally new use case.



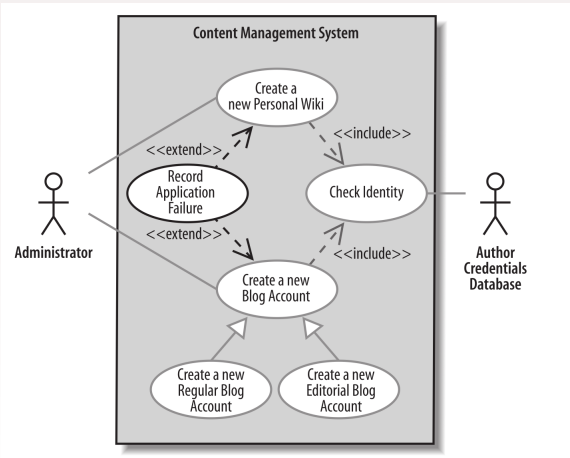
Use case inheritance is useful when you want to show that one use case is a special type of another use case.



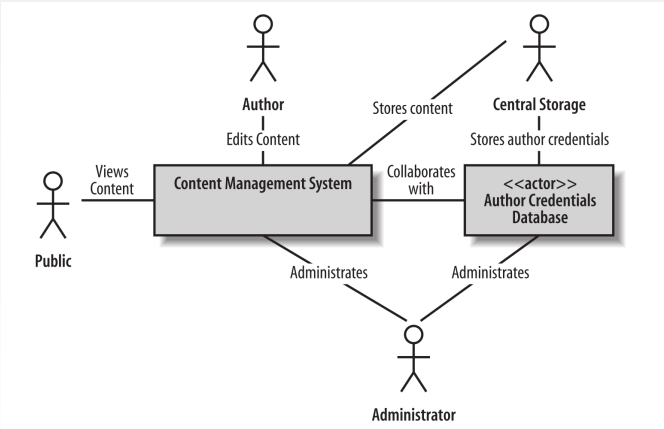
Use Case Diagrams: Extend Relationship

北京大学

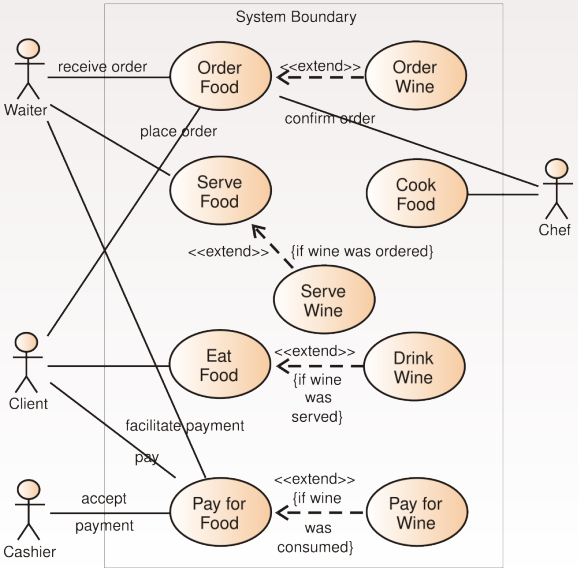
A use case might completely reuse another use case's behavior, but that this reuse was optional and dependent either on a runtime or system implementation decision



Shows your system's context or domain:

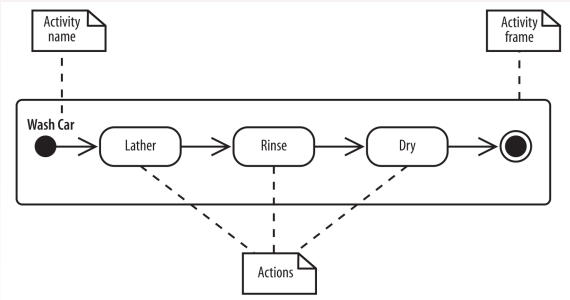


Use Case Diagrams: Example



Activity diagrams allow you to specify how your system will accomplish its goals.

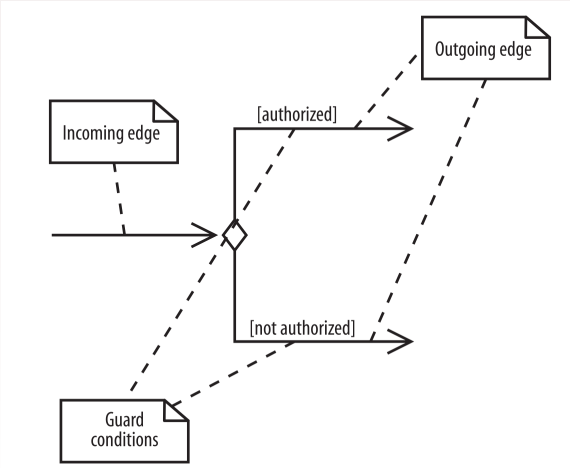
Actions & Activities:



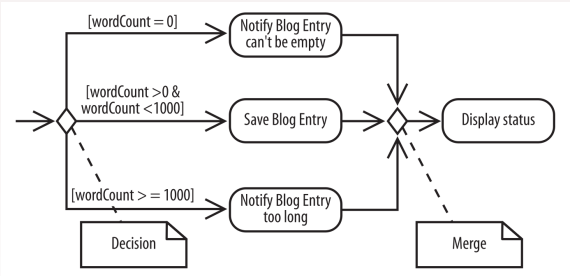
Activity Diagrams: Decisions & Merges

北京大学

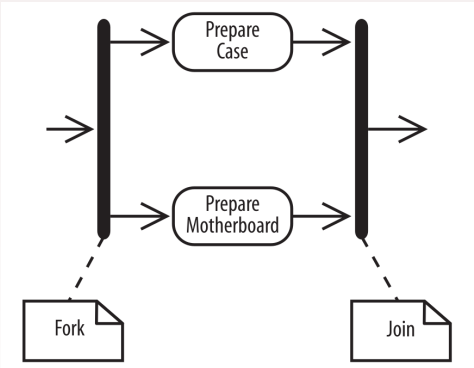
Decisions are used when you want to execute a different sequence of actions depending on a condition.



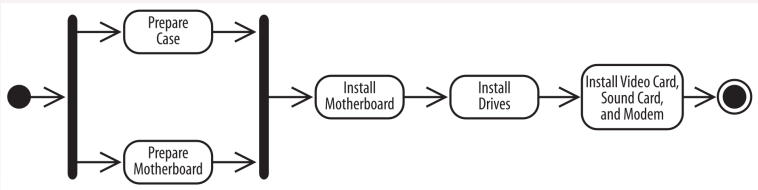
The branched flows join together at a merge node, which marks the end of the conditional behavior started at the decision node.



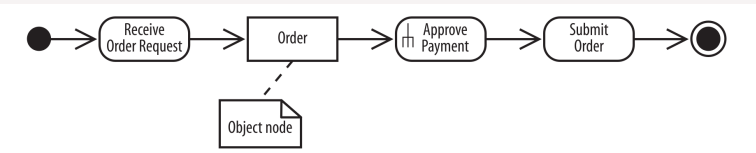
Represent parallel actions in activity diagrams by using forks and joins.



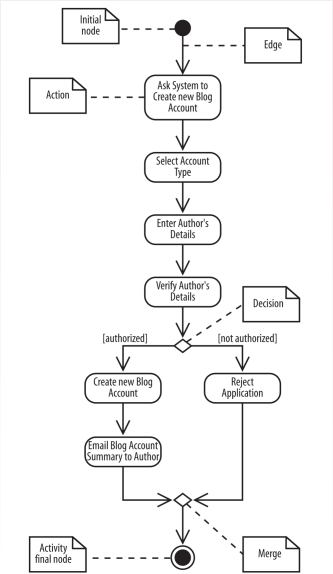
Activity diagram for the computer assembly workflow:



Object nodes to show data flowing through an activity:

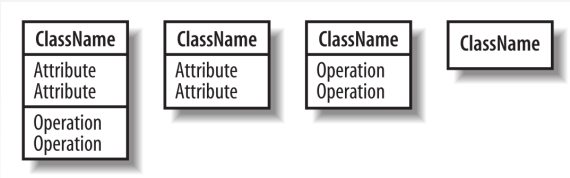


Activity Diagrams: Example



Class Diagrams

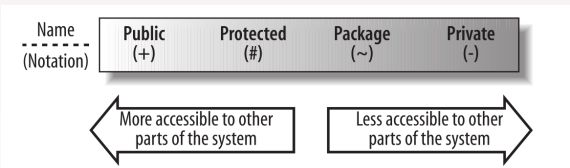
Visualize design of object-oriented systems.



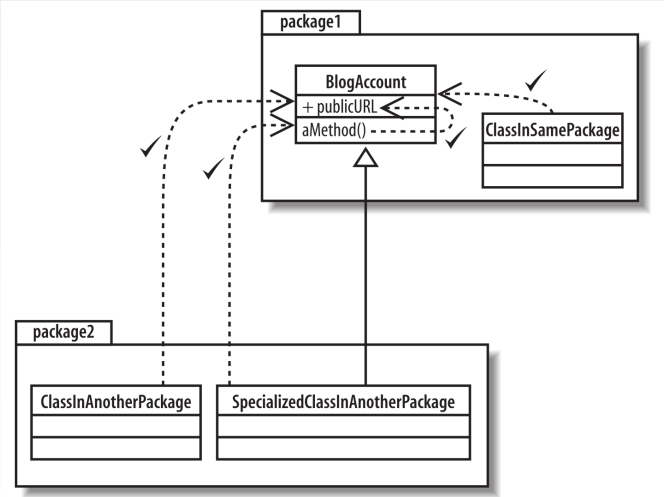
Class Diagrams: Visibility

北京大学

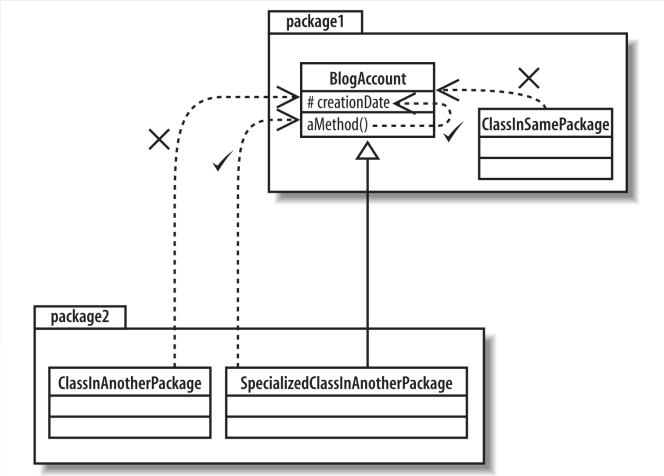
Control access to attributes, operations, and even entire classes to effectively enforce encapsulation.



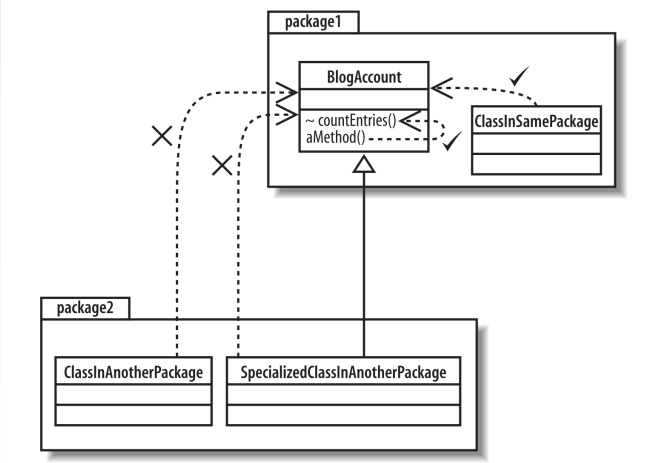
Class Diagrams: Public Visibility



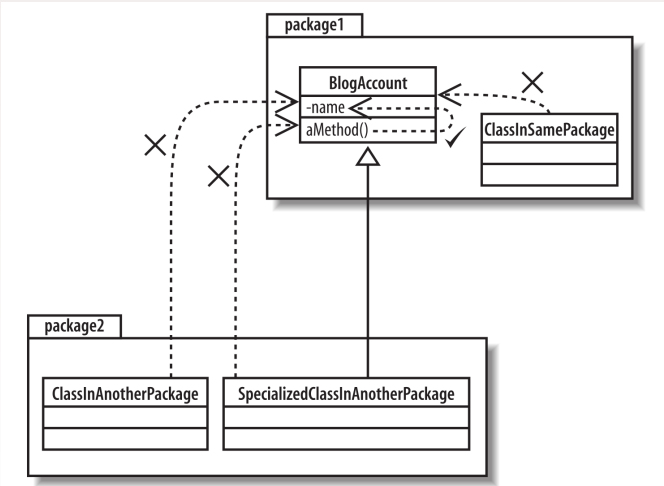
Class Diagrams: Protected Visibility



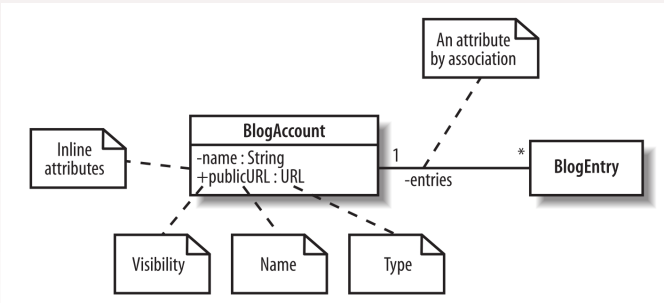
Class Diagrams: Package Visibility



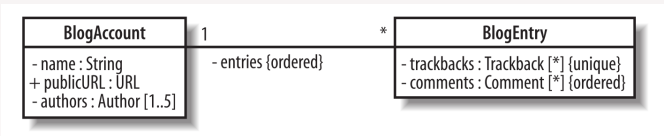
Class Diagrams: Private Visibility



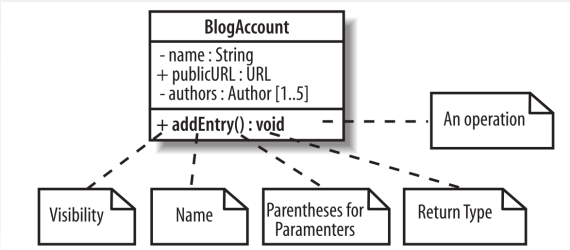
A class's attributes are the pieces of information that represent the state of an object.



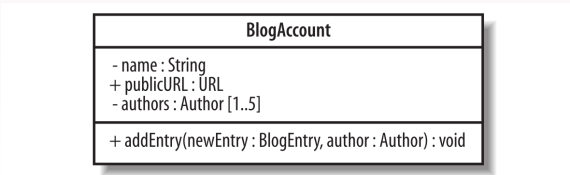
Multiplicity allows you to specify that an attribute actually represents a collection of objects.



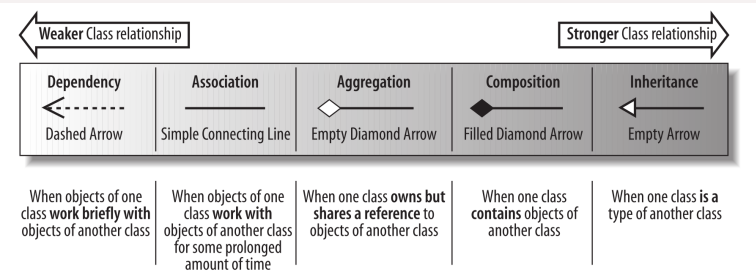
A class's attributes are the pieces of information that represent the state of an object.



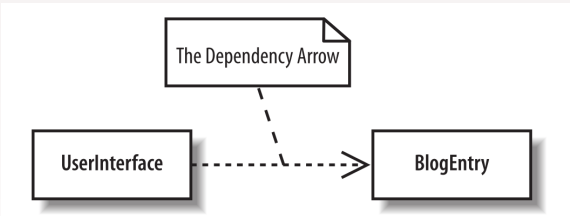
Parameters:



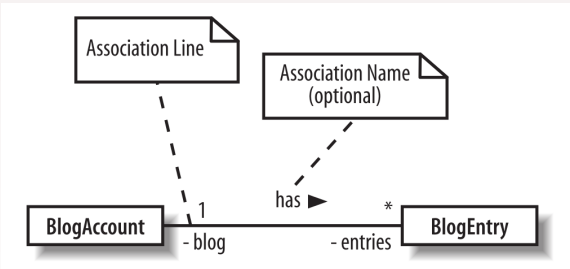
UML offers five different types of class relationship:



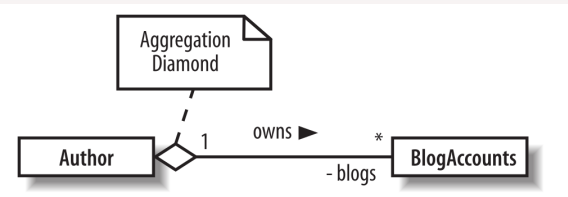
A dependency between two classes declares that a class needs to know about another class to use objects of that class.



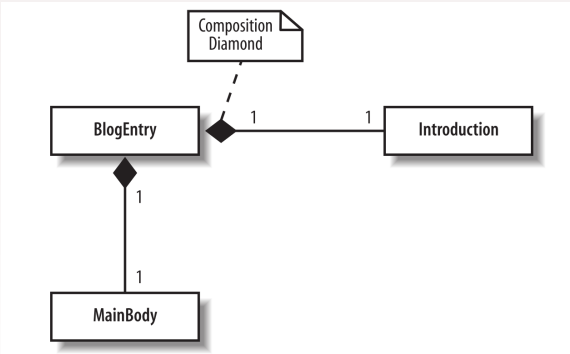
Association means that a class will actually contain a reference to an object, or objects, of the other class in the form of an attribute.



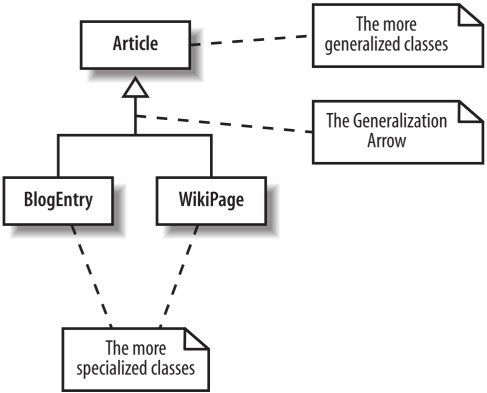
Aggregation is really just a stronger version of association and is used to indicate that a class actually owns but may share objects of another class.



Even stronger relationship than aggregation, *part of*. If the blog entry is deleted, then its corresponding parts are also deleted.



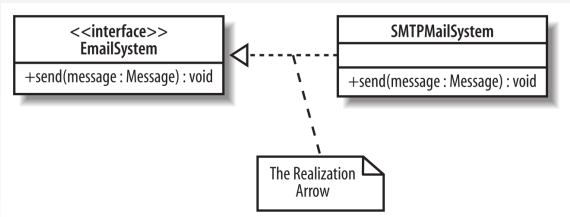
A class that is a type of another class.



Class Diagrams: Interfaces

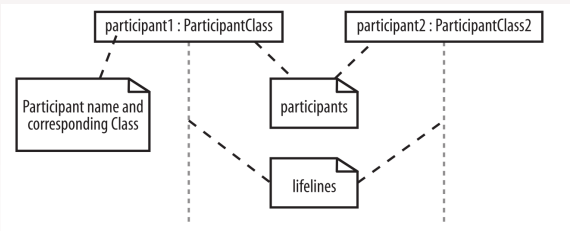
北京大学

An interface is a collection of operations that have no corresponding method implementations.

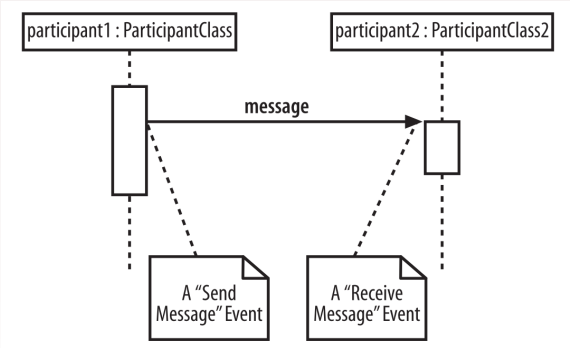


Sequence diagrams are all about capturing the order of interactions between parts of your system.

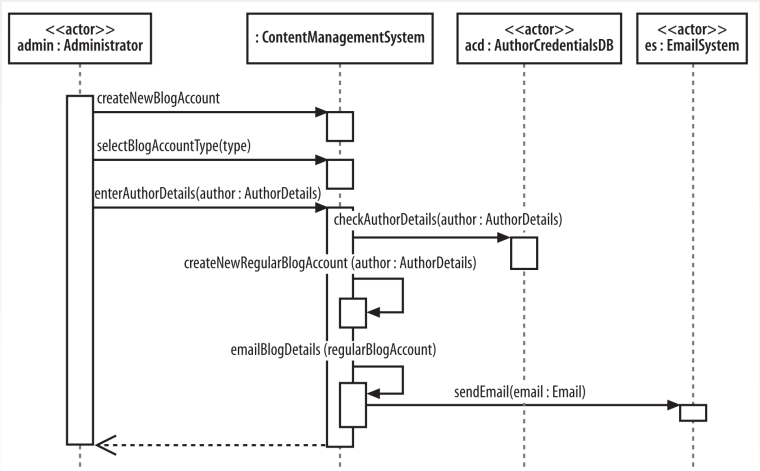
Participants in a Sequence Diagram:



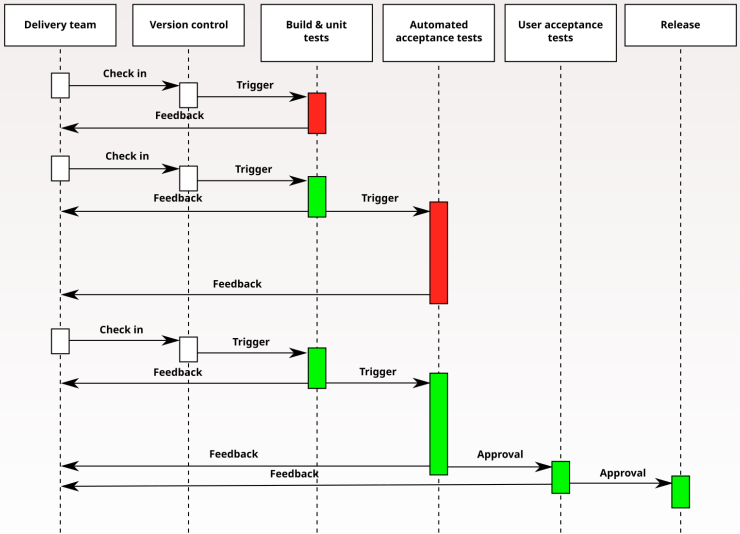
An event is any point in an interaction where something occurs.



Sequence Diagrams: Example



Sequence Diagrams: Example



[1] Alain Abran, James W Moore, Pierre Bourque, Robert Dupuis, and L Tripp.
Software engineering body of knowledge.
IEEE Computer Society, Angela Burgess, 25:1235, 2004.

[2] Frederick Brooks and H Kugler.
No silver bullet.
April, 1987.

[3] Russ Miles and Kim Hamilton.
Learning UML 2.0: a pragmatic introduction to UML.
" O'Reilly Media, Inc.", 2006.