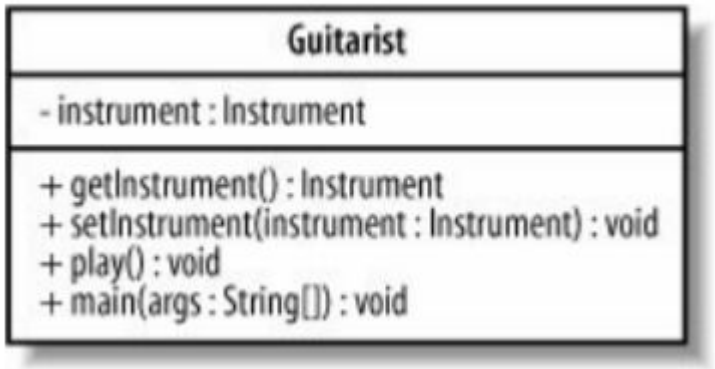
Unified Modeling Language (UML)

A modeling language can be made up of pseudo-code, actual code, pictures, diagrams, or long passages of description; in fact, it's pretty much anything that helps you describe your system.

The elements that make up a modeling language are called its notation.



Unified Modeling Language (UML)

The Unified Modeling Language (UML) is the standard modeling language for software and systems development.

Software can span from simple application to a complex multi-tier enterprise scale application.

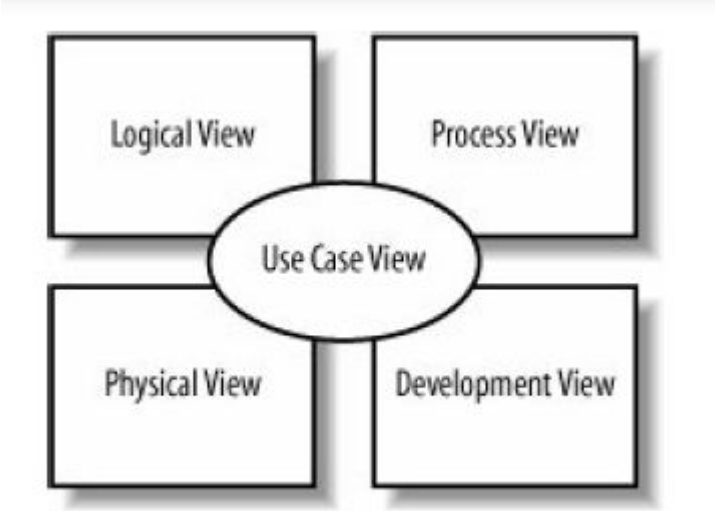
How do you (and your team) keep track of which components are needed, what their jobs are, and how they meet your customers' requirements?

Furthermore, how do you share your design with your colleagues to ensure the pieces work together?

There are just too many details that can be misinterpreted or forgotten when developing a complex system without some help

Views of Model

There are a number of ways to break up your UML model diagrams into perspectives or views that capture a particular facet of your system.

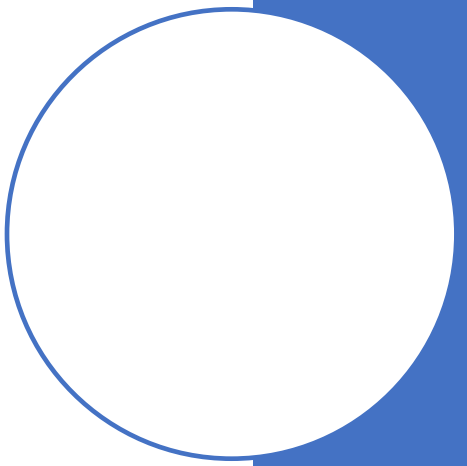


Logical View

Logical view describes the abstract descriptions of a system's parts.

Used to model what a system is made up of and how the parts interact with each other.

The types of UML diagrams that typically make up this view include class, object, state machine, and interaction diagrams.

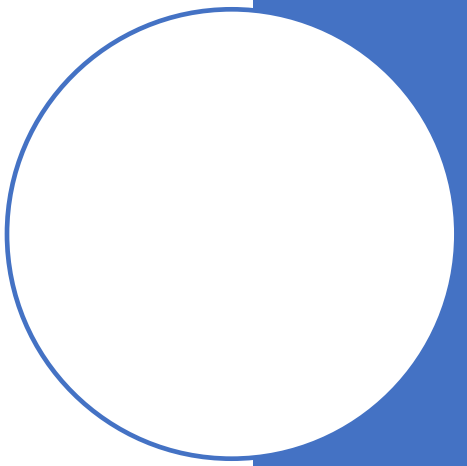


Process view

Describes the processes within your system.

It is particularly helpful when visualizing what must happen within your system.

This view typically contains activity diagrams.

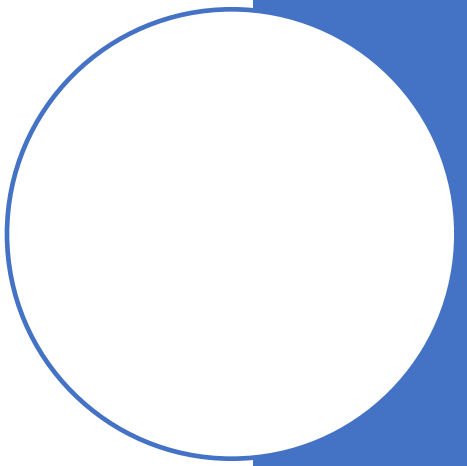


Development view

Describes how your system's parts are organized into modules and components.

It is very useful to manage layers within your system's architecture.

This view typically contains package and component diagrams.



Physical view

Describes how the system's design, as described in the three previous views, is then brought to life as a set of real-world entities.

The diagrams in this view show how the abstract parts map into the final deployed system.

This view typically contains deployment diagrams.

Use case view

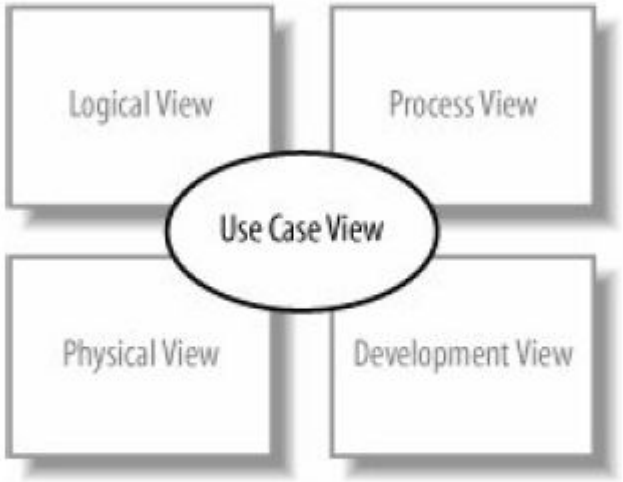
Describes the functionality of the system being modeled from the perspective of the outside world.

This view is needed to describe what the system is supposed to do.

All of the other views rely on the use case view to guide them.

This view typically contains use case diagrams, descriptions, and overview diagrams.

Modeling Requirements: Use Cases



Modeling Requirements: Use Cases

Use cases are an excellent starting point for just about every facet of object-oriented system development, design, testing, and documentation.

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.

Use cases specify only what your system is supposed to do, i.e., the system's functional requirements.

Your use cases can be assigned to teams or individuals to be implemented and, since a use case represents tangible user value, you can track the progress of the project by use cases delivered.

Last but not least, use cases also help construct tests for your system.

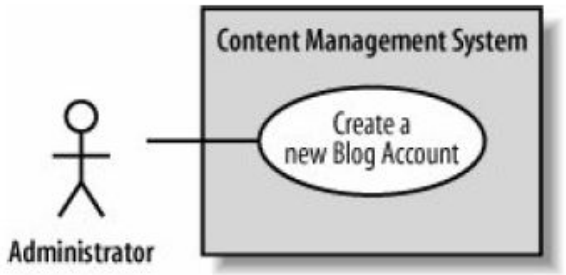
Use Case

A use case describes the way your system behaves to meet a requirement.

i) First identify actor(s).

ii) Once you have captured an initial set of actors that interact with your system, you can assemble the exact model of those interactions. Find cases where the system is being used (by actor) to complete a specific job.

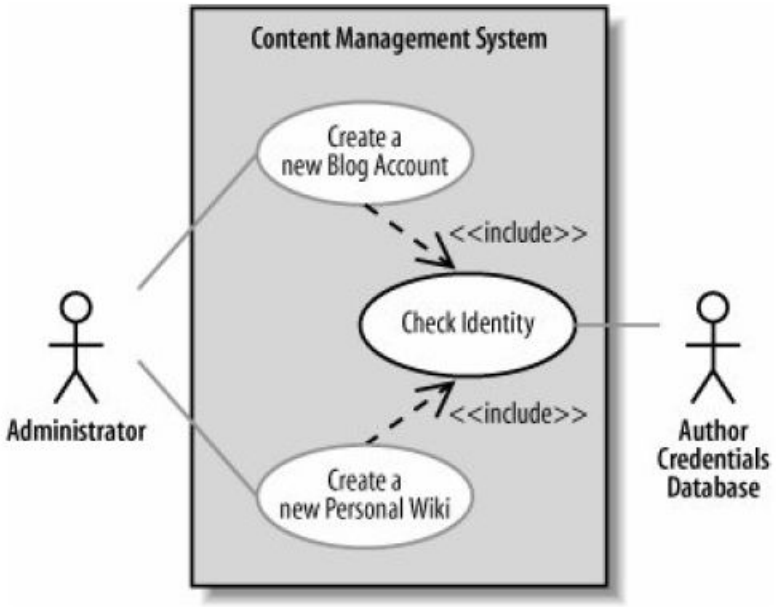
iii) Oval shape represents a use case, box represents a System boundary.



Use Case Relationships: include relationship

The <<include>> relationship declares that the use case at the head of the dotted arrow completely reuses all of the steps from the use case being included.

The ‘Create a new Blog Account’ and ‘Create a new Personal Wiki’ completely reuse all of the steps declared in the Check Identity use case.



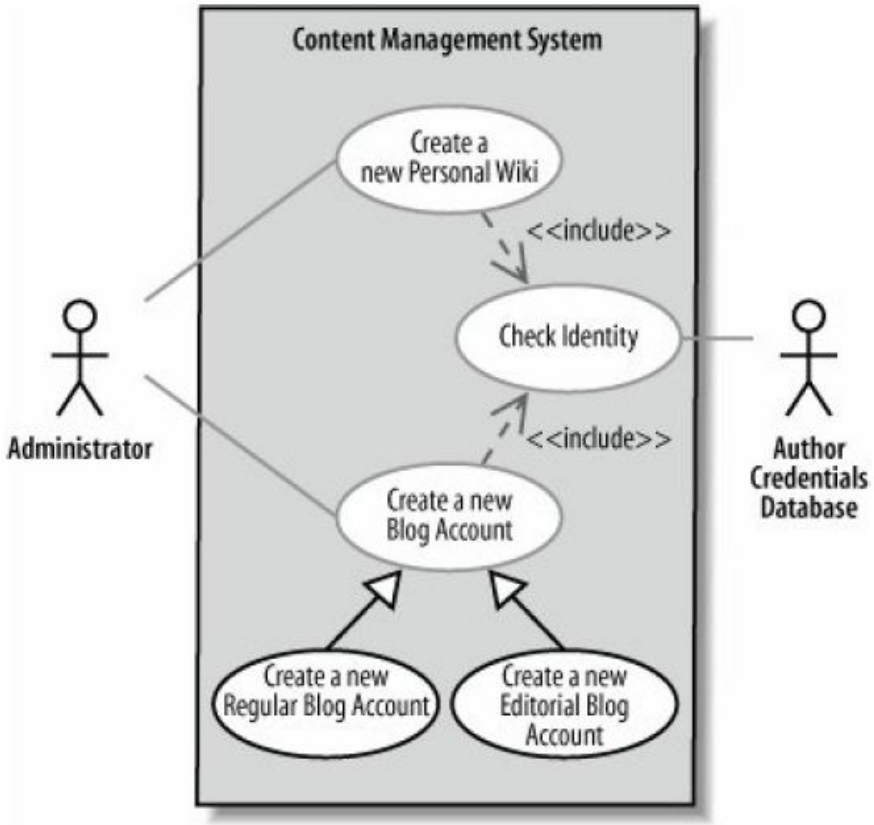
Inheritance relationship / Generalization

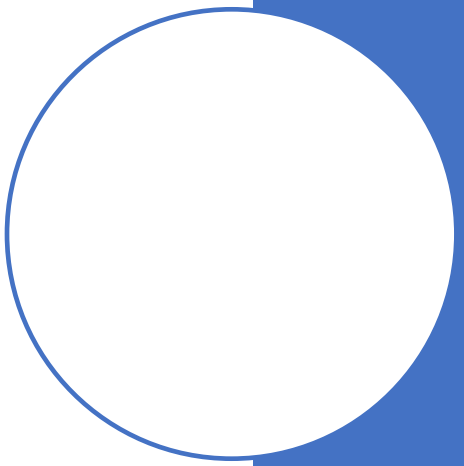
You want to describe the general behavior for creating a blog account captured in the Create a new Blog Account use case and then define specialized use cases in which the account being created is a specific type, such as a regular account with one blog or an editorial account that can make changes to entries in a set of blogs.

This is where use case generalization comes in. A more common way of referring to generalization is using the term inheritance . Use case inheritance is useful when you want to show that one use case is a special type of another use case. To show use case inheritance , use the generalization arrow to connect the more general, or parent, use case to the more specific use case.

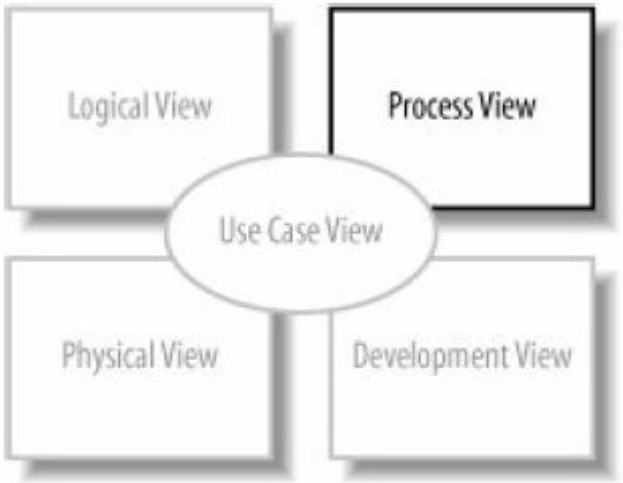
Inheritance relationship / Generalization

The Figure shows how you could extend the CMS's use cases to show that two different types of blog accounts can be created.





Modeling System Workflows: Activity Diagrams



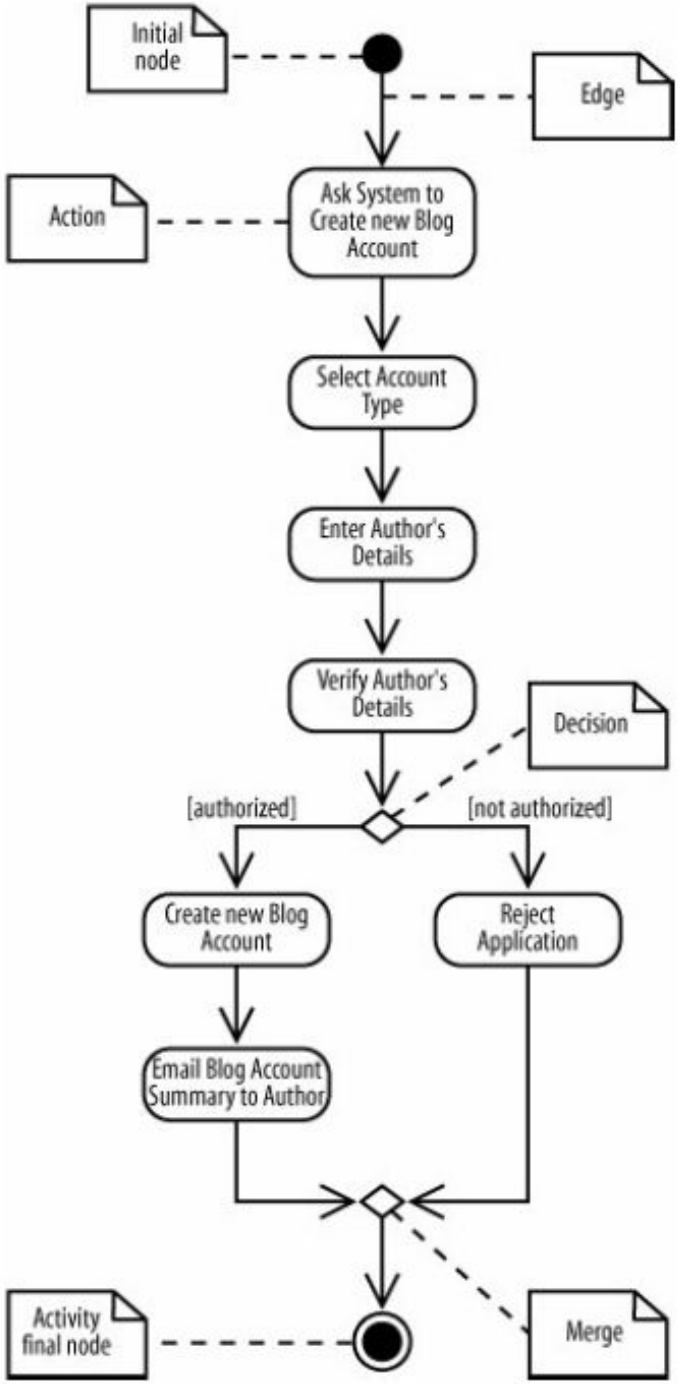
Modeling System Workflows: Activity Diagrams

Use cases show what your system should do. Activity diagrams allow you to specify how your system will accomplish its goals.

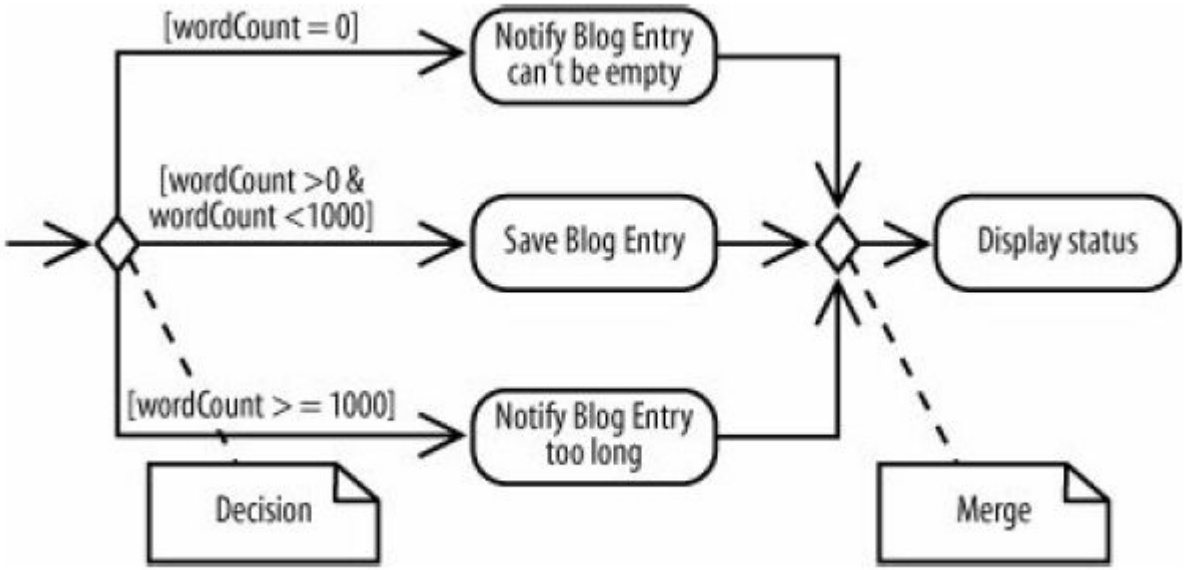
Activity diagrams show high-level actions chained together to represent a process occurring in your system.

For example, you can use an activity diagram to model the steps involved with creating a blog account.

Activity diagrams are particularly good at modeling business processes . A business process is a set of coordinated tasks that achieve a business goal, such as shipping customers' orders.

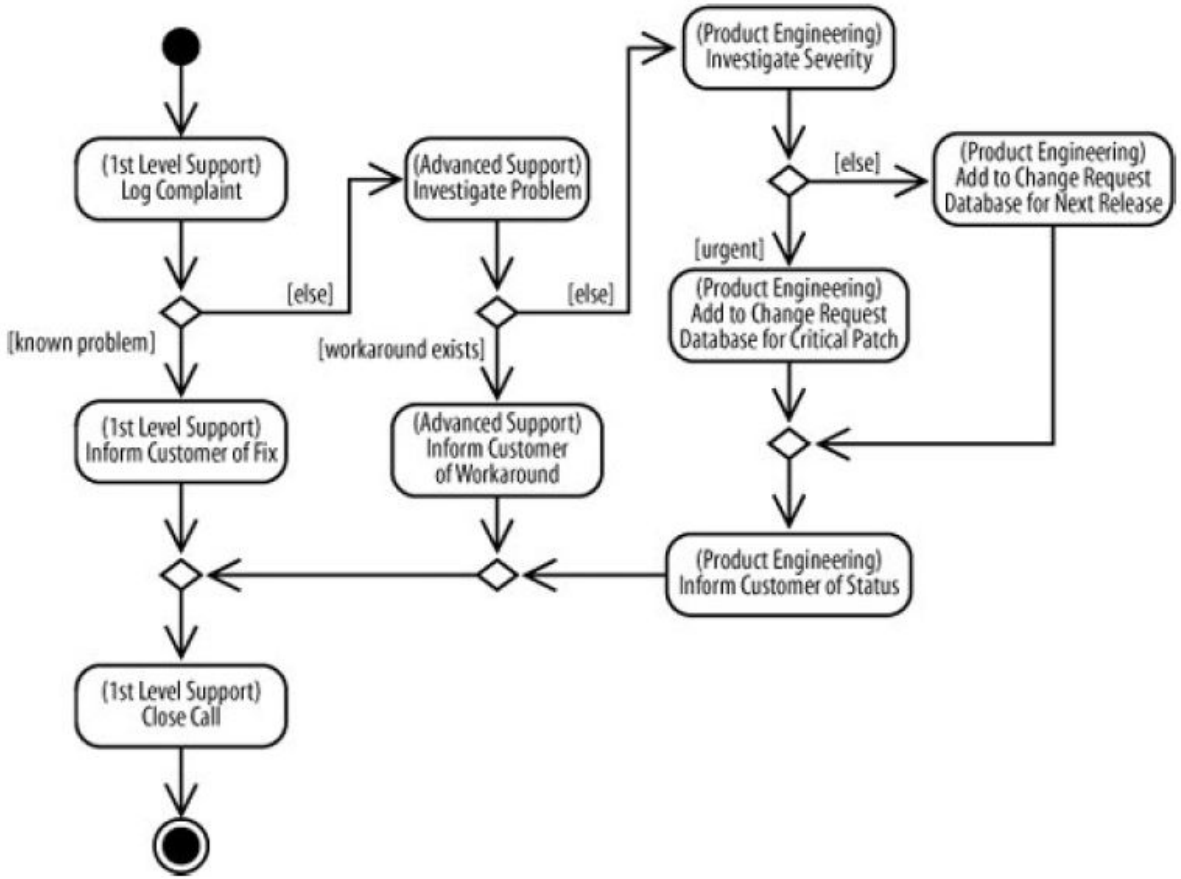

Initial node
Edge
Action
Ask System to Create new Blog Account
Select Account Type
Enter Author's Details
Verify Author's Details
Decision
[authorized]
[not authorized]
Create new Blog Account
Reject Application
Email Blog Account Summary to Author
Activity final node
Merge

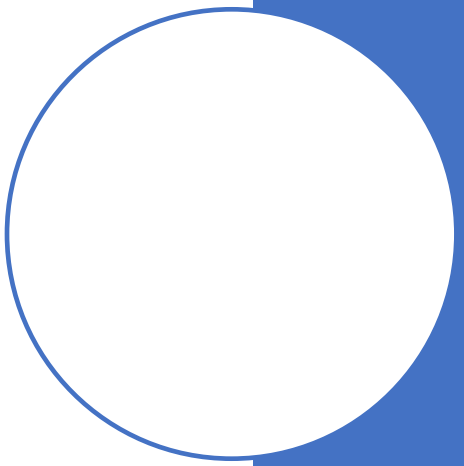
Using Annotation



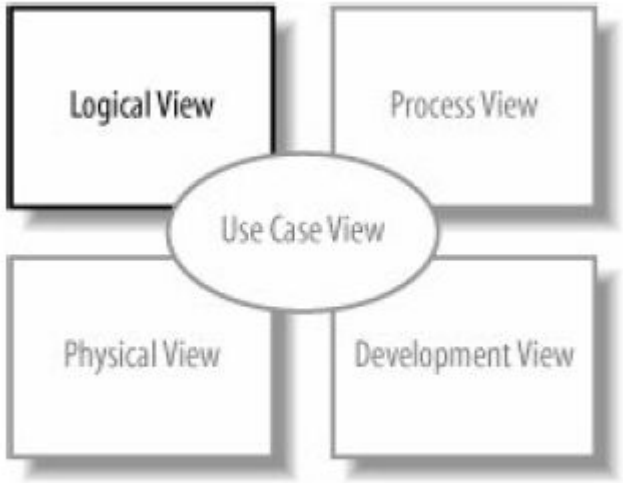
If the input value is 1000, then the “Notify Blog Entry too long” action is performed.

Using Annotation





Modeling a System's Logical Structure: Introducing Classes and Class Diagrams

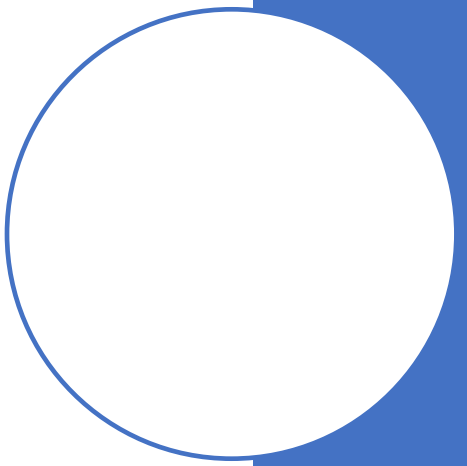


Modeling a System's Logical Structure: Introducing Classes and Class Diagrams

Classes are at the heart of any object-oriented system; therefore, it follows that the most popular UML diagram is the class diagram.

A system's structure is made up of a collection of pieces often referred to as objects.

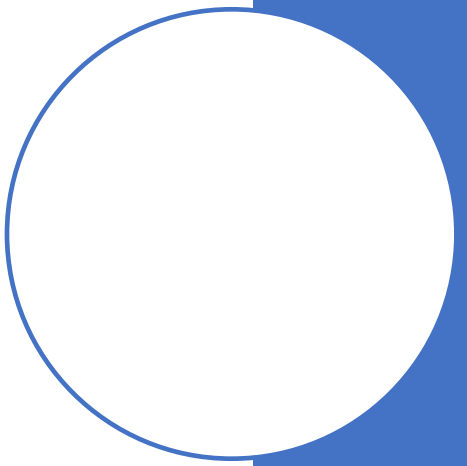
Classes describe the different types of objects that your system can have, and class diagrams show these classes and their relationships.



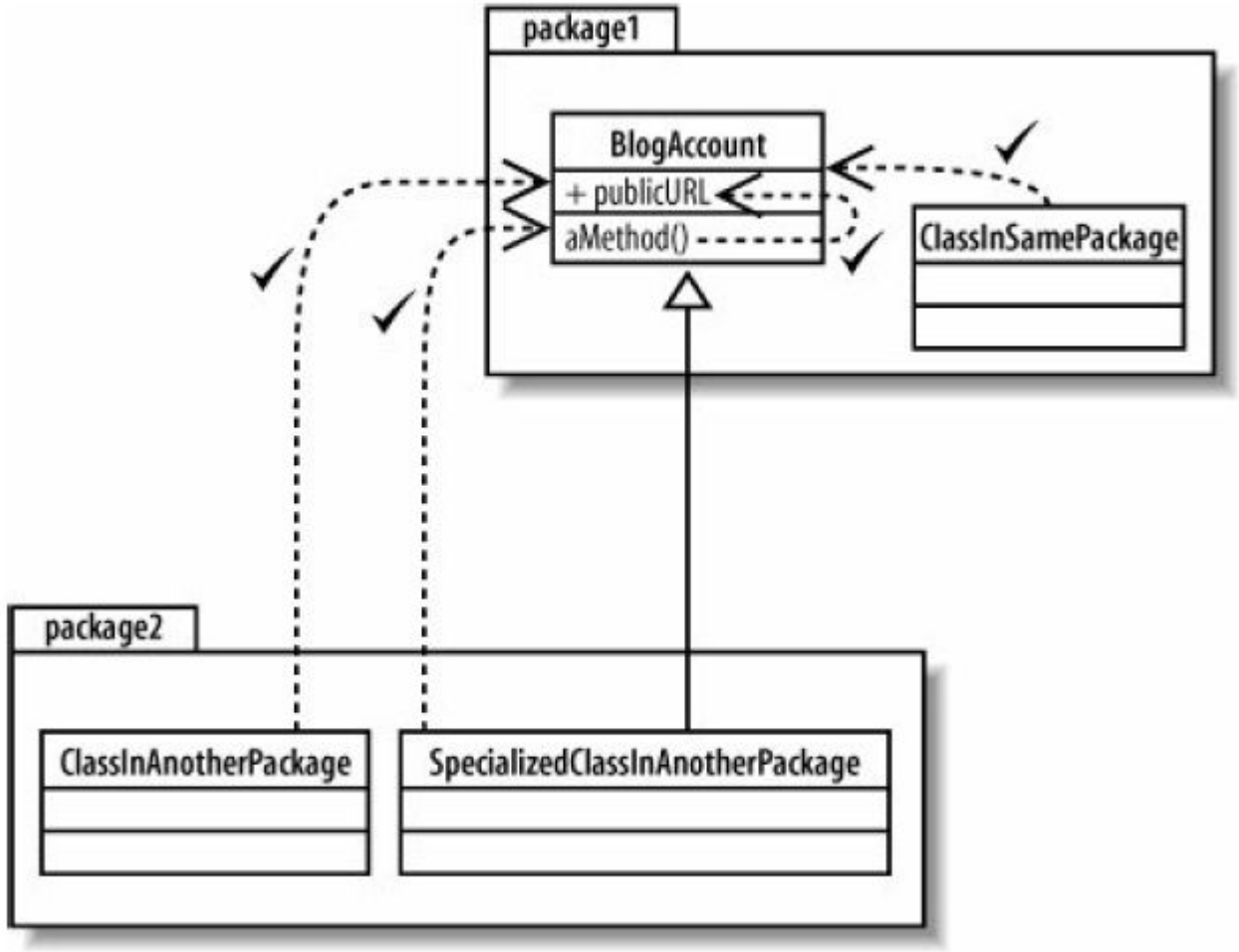
Modeling a System's Logical Structure: Introducing Classes and Class Diagrams

Let see the pillars of OOP.

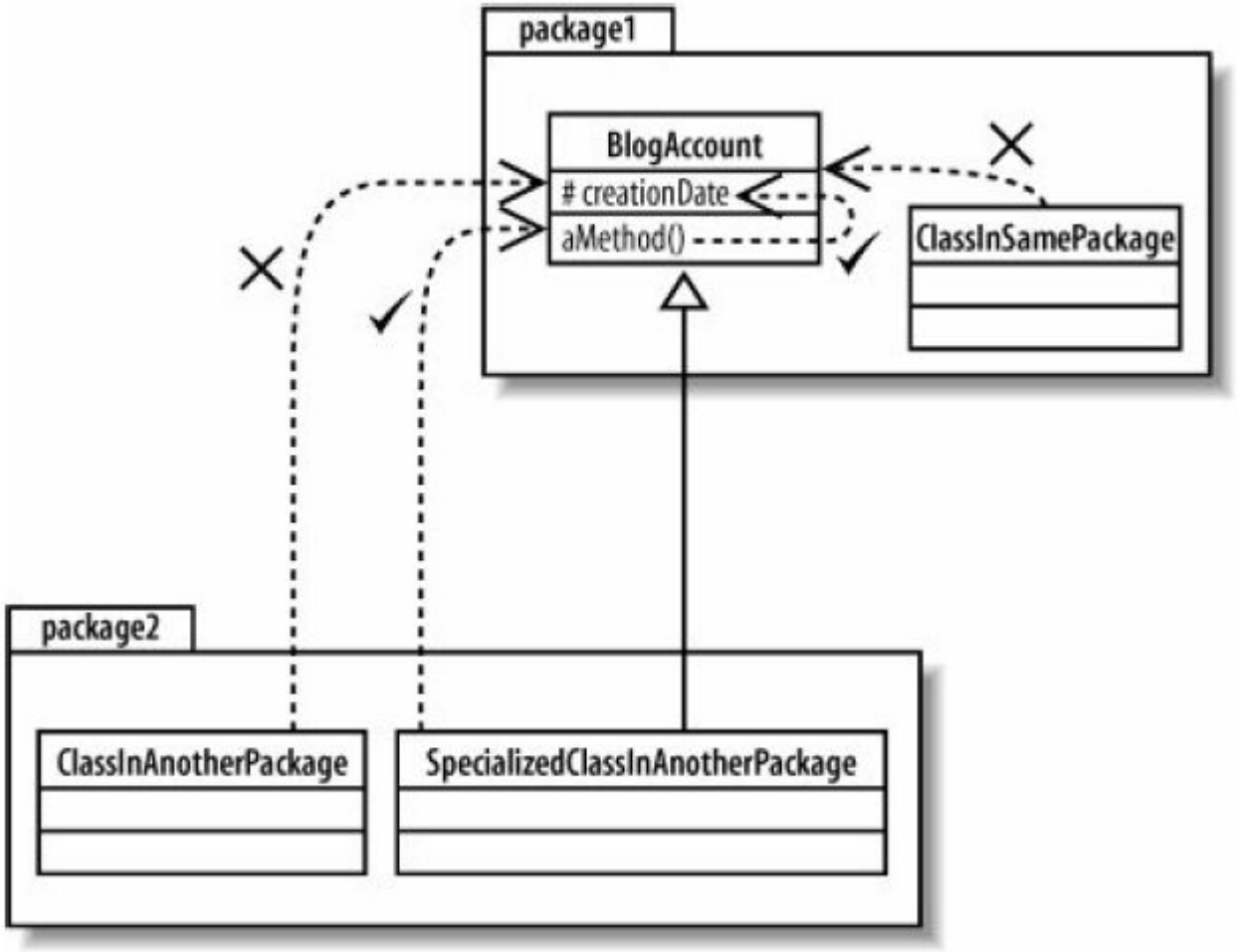
Abstraction
Encapsulation
Inheritance
Polymorphism



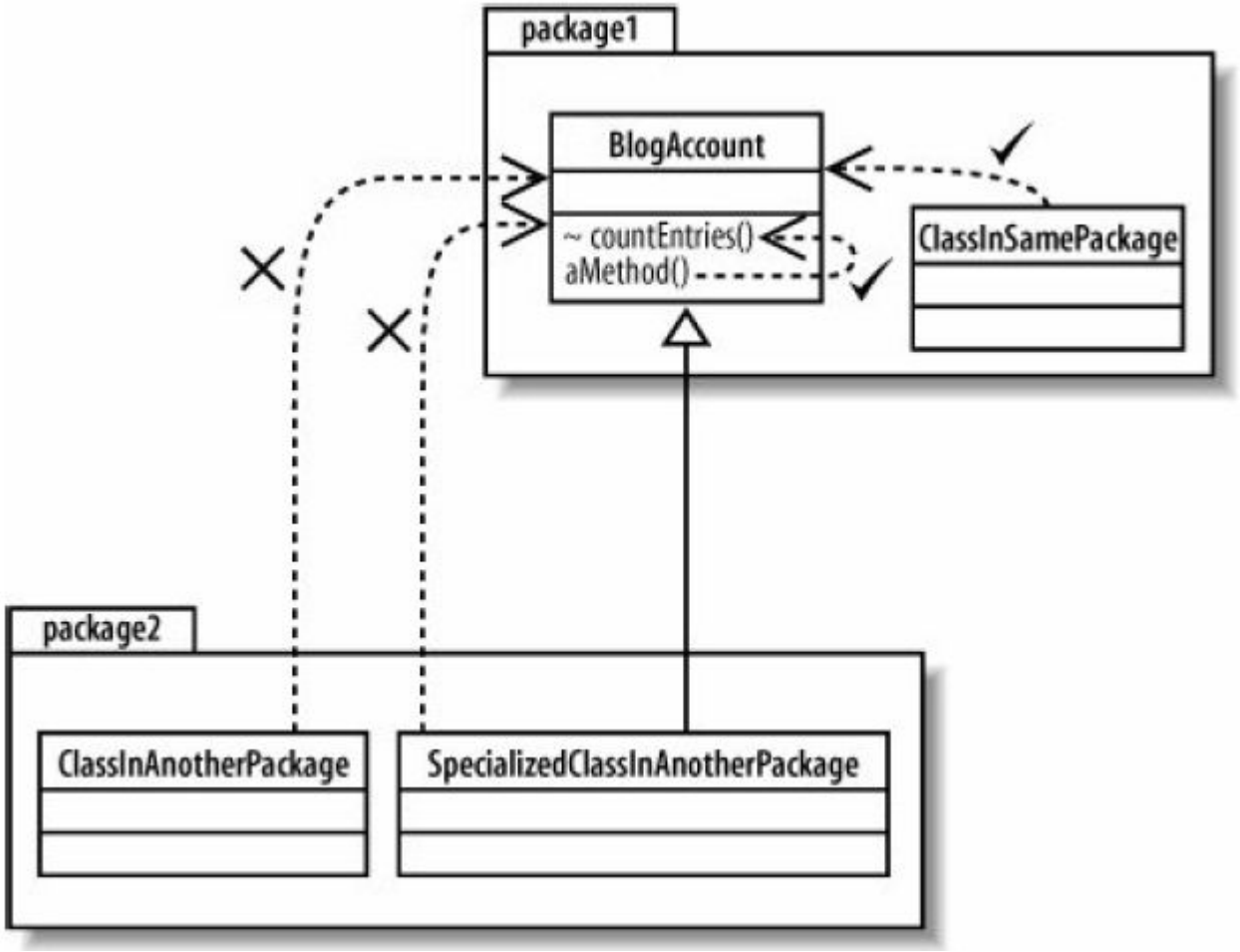
Public Visibility



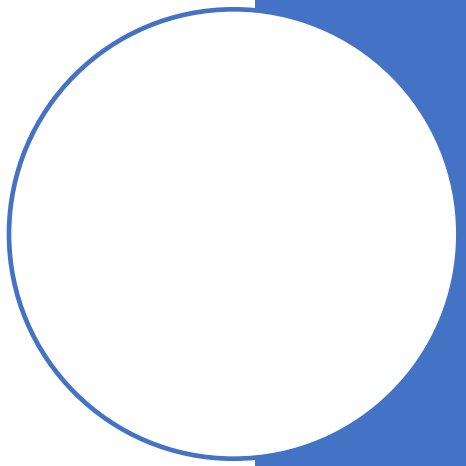
Protected Visibility



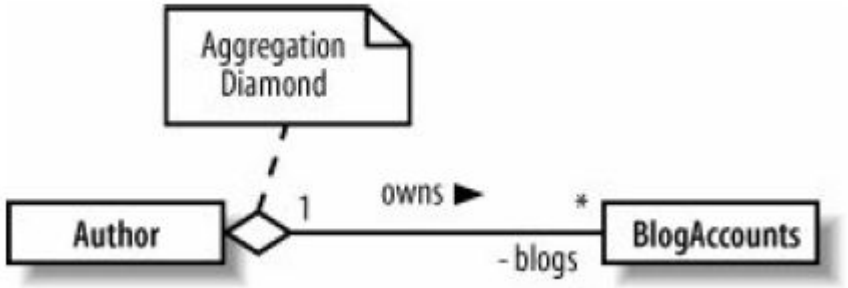
Private Visibility



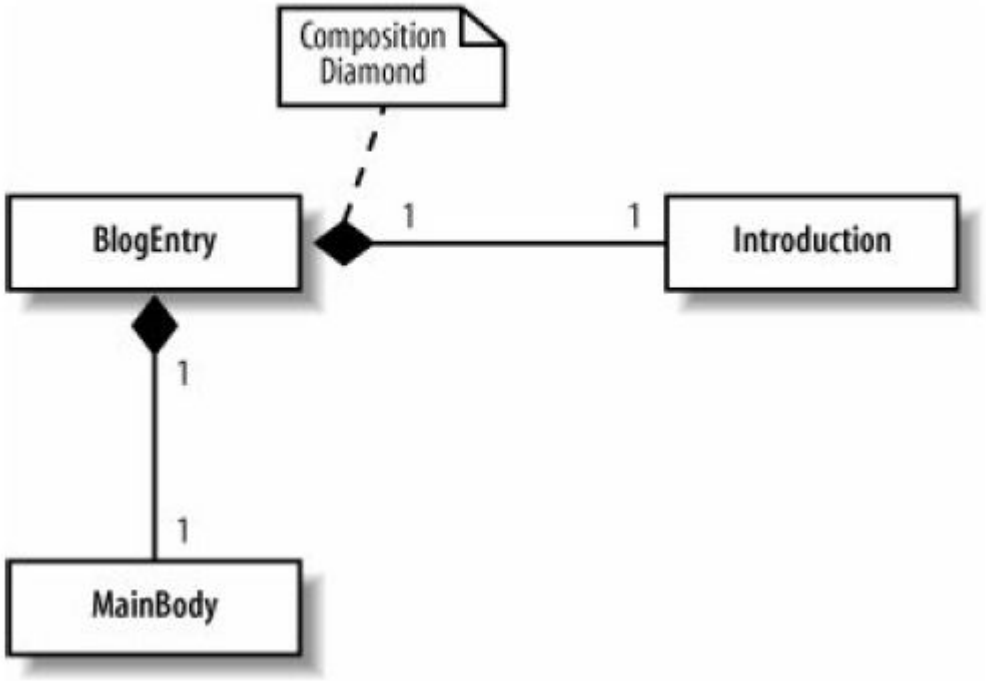
Class Relationships



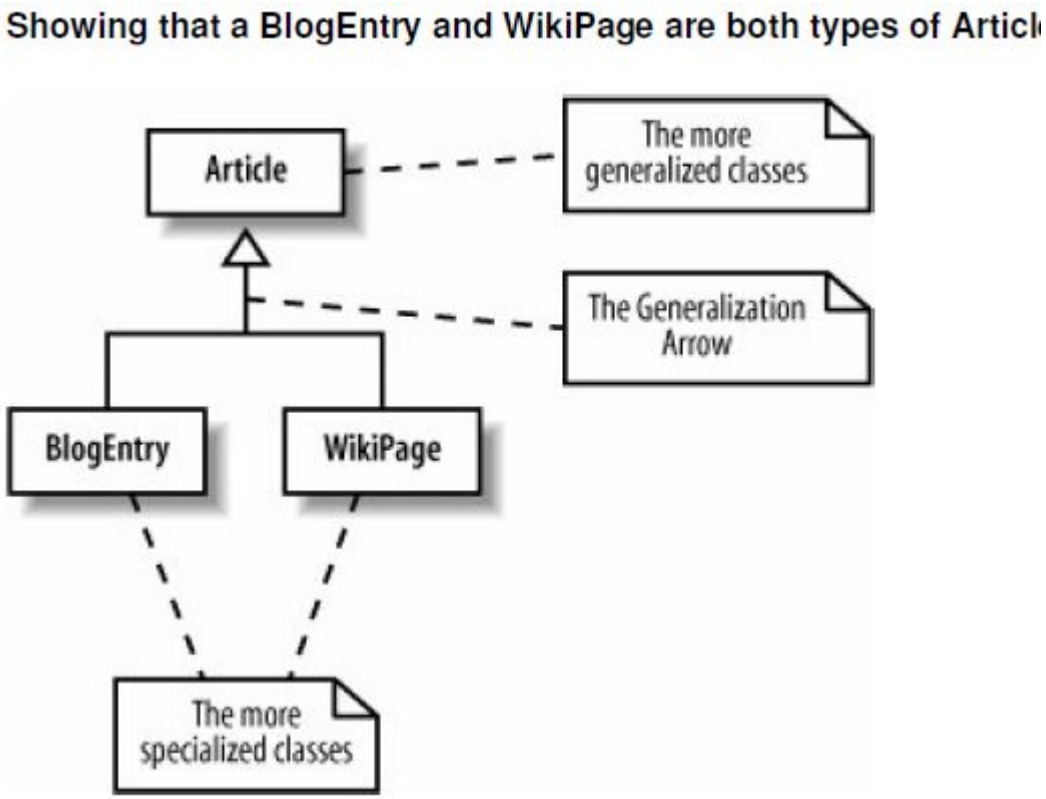
Class Relationships
1) Aggregation



Class Relationships
2) Composition

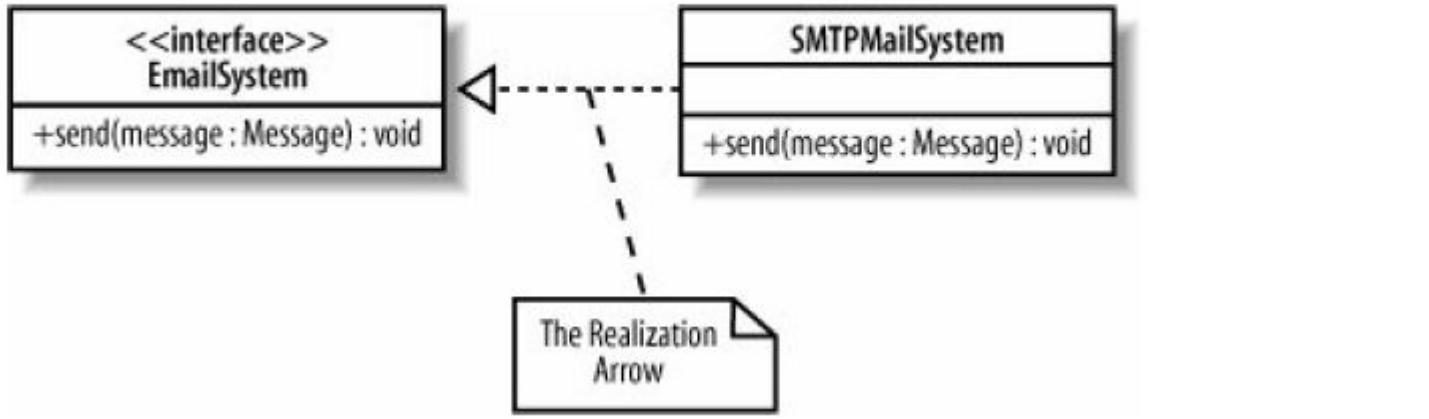


Generalization / Inheritance
Is-a relationship



Interfaces
Realization / Implementation

The realization arrow specifies that the SMTPMailSystem realizes the EmailSystem interface



Class Diagram (Quick Recap)