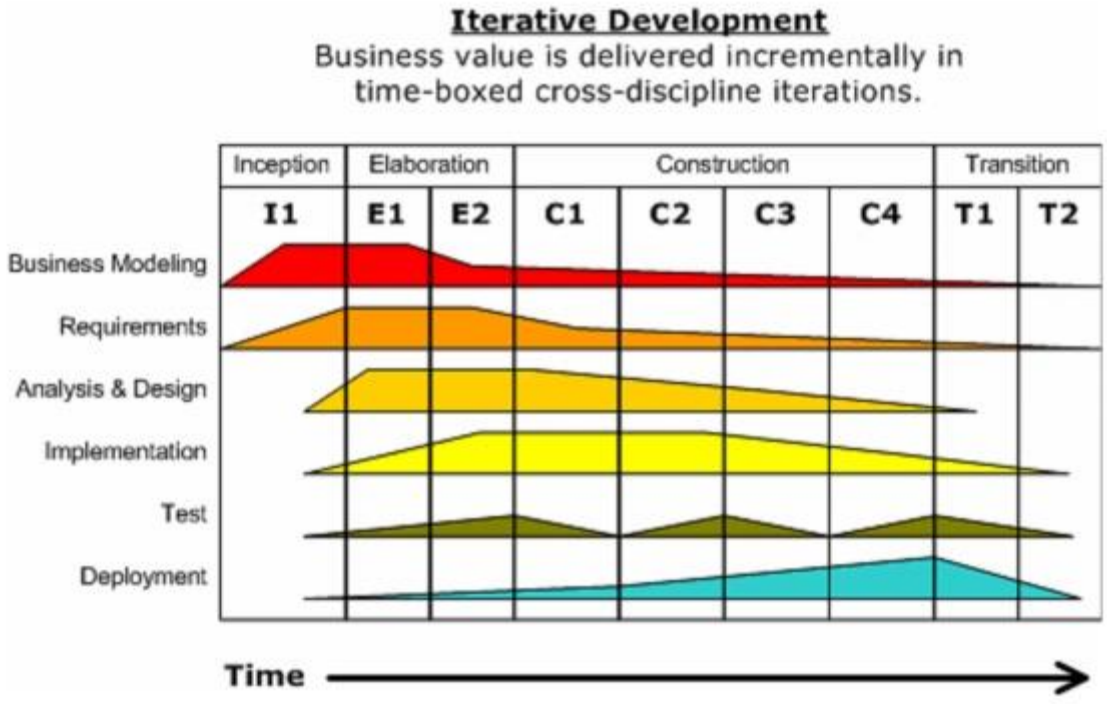
Unified Process

The Unified Software Development Process or Unified Process is a popular iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP)
Profile of a typical project showing the relative sizes of the four phases of the Unified Process.

Overview

The Unified Process is not simply a process, but rather an extensible framework which should be customized for specific organizations or projects. The Rational Unified Process is, similarly, a customizable framework. As a result it is often impossible to say whether a refinement of the process was derived from UP or from RUP, and so the names tend to be used interchangeably.

The name Unified Process as opposed to Rational Unified Process is generally used to describe the generic process, including those elements which are common to most refinements. The Unified Process name is also used to avoid potential issues of trademark infringement since Rational Unified Process and RUP are trademarks of IBM. The first book to describe the process was titled The Unified Software Development Process and published in 1999 by Ivar Jacobson, Grady Booch and James Rumbaugh. Since then various authors unaffiliated with Rational Software have published books and articles using the name Unified Process, whereas authors affiliated with Rational Software have favored the name Rational Unified Process.



1. Unified Process Characteristics

Iterative and Incremental

The Unified Process is an iterative and incremental development process. The Elaboration, Construction and Transition phases are divided into a series of timeboxed iterations. (The Inception phase may also be divided into iterations for a large project.)
Each iteration results in an increment, which is a release of the system that contains added or improved functionality compared with the previous release.

Although most iterations will include work in most of the process disciplines (e.g. Requirements, Design, Implementation, Testing) the relative effort and emphasis will change over the course of the project.

Use Case Driven

In the Unified Process, use cases are used to capture the functional requirements and to define the contents of the iterations. Each iteration takes a set of use cases or scenarios from requirements all the way through implementation, test and deployment.

Architecture Centric

The Unified Process insists that architecture sit at the heart of the project team's efforts to shape the system. Since no single model is sufficient to cover all aspects of a system, the Unified Process supports multiple architectural models and views.

One of the most important deliverables of the process is the executable architecture baseline which is created during the Elaboration phase. This partial implementation of the system serves and validate the architecture and act as a foundation for remaining development.

Risk Focused

The Unified Process requires the project team to focus on addressing the most critical risks early in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first.

Project Lifecycle

The Unified Process divides the project into four phases:

- Inception
- Elaboration
- Construction
- Transition

Inception Phase

Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it may be an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process.

The following are typical goals for the Inception phase.

- Establish a justification or business case for the project
- Establish the project scope and boundary conditions

Outline the use cases and key requirements that will drive the design tradeoffs

Outline one or more candidate architectures

- Identify risks
- Prepare a preliminary project schedule and cost estimate
- The Lifecycle Objective Milestone marks the end of the Inception phase.

Elaboration Phase

- During the Elaboration phase the project team is expected to capture a healthy majority of the system requirements. However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture. Common processes undertaken in this phase include the creation of use case diagrams,

conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams).

- The architecture is validated primarily through the implementation of an Executable Architecture Baseline. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, timeboxed iterations. By the end of the Elaboration phase the system architecture must have stabilized and the executable architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost.

- The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase. At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase.

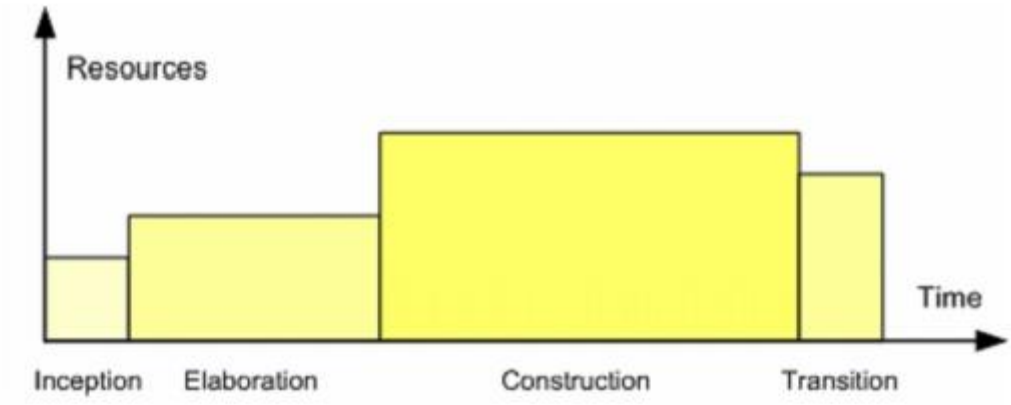
The Lifecycle Architecture Milestone marks the end of the Elaboration phase.

Construction Phase

Construction is the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration. System features are implemented in a series of short, timeboxed iterations. Each iteration results in an executable release of the software. It is customary to write full text use cases during the construction phase and each one becomes the start of a new iteration. Common UML (Unified Modelling Language) diagrams used during this phase include Activity, Sequence, Collaboration,State (Transition) and Interaction Overview diagrams. The Initial Operational Capability Milestone marks the end of the Construction phase.

Transition Phase

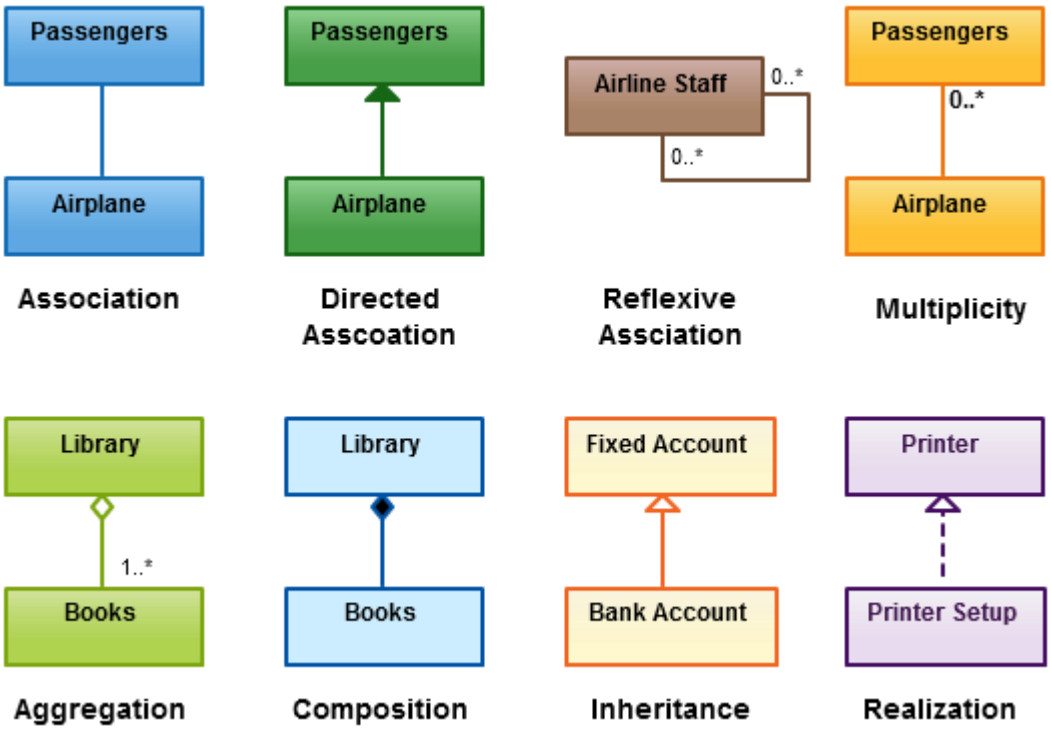
The final project phase is Transition. In this phase the system is deployed to the target users. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training. The Product Release Milestone marks the end of the Transition phase.



Class diagram

UML Class Diagram Relationships Explained with Examples

Many people consider class diagrams a bit more complicated to build compared with ER diagrams. Most of the time it’s because of the inability to understand the different relationships in class diagrams. This article explains how to correctly determine and implement the different class diagram relationships that are applicable in object-oriented modeling. What’s more, you can easily create class diagrams online using our diagramming tool.

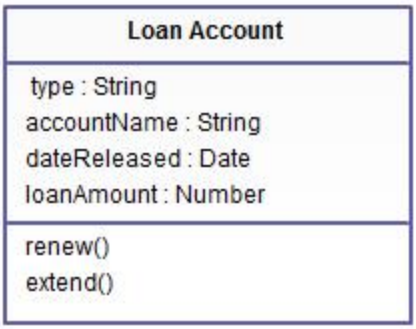


Relationships in UML class diagrams

What are the Class Diagrams?

Class diagrams are the main building block in object-oriented modeling. They are used to show the different objects in a system, their attributes, their operations and the relationships among them.

The following figure is an example of a simple class:



Simple class diagram with attributes and operations

In the example, a class called “loan account” is depicted. Classes in class diagrams are represented by boxes that are partitioned into three:

1. The top partition contains the name of the class.
2. The middle part contains the class's attributes.
3. The bottom partition shows the possible operations that are associated with the class.

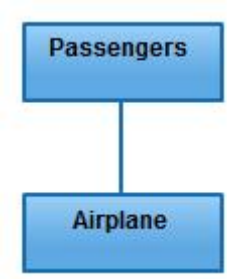
The example shows how a class can encapsulate all the relevant data of a particular object in a very systematic and clear way. A class diagram is a collection of classes similar to the one above.

Relationships in Class Diagrams

Classes are interrelated to each other in specific ways. In particular, relationships in class diagrams include different types of logical connections. The following are such types of logical connections that are possible in UML:

- Association
- Directed Association
- Reflexive Association
- Multiplicity
- Aggregation
- Composition
- Inheritance/Generalization
- Realization

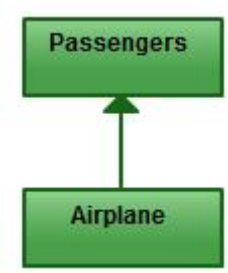
Association



Association

is a broad term that encompasses just about any logical connection or relationship between classes. For example, passenger and airline may be linked as above:

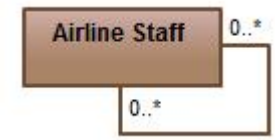
Directed Association



Directed Association

refers to a directional relationship represented by a line with an arrowhead. The arrowhead depicts a container-contained directional flow.

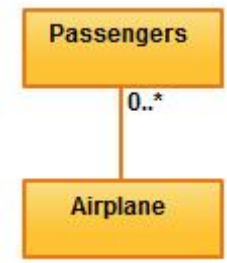
Reflexive Association



Reflexive Association

This occurs when a class may have multiple functions or responsibilities. For example, a staff member working in an airport may be a pilot, aviation engineer, a ticket dispatcher, a guard, or a maintenance crew member. If the maintenance crew member is managed by the aviation engineer there could be a managed by relationship in two instances of the same class.

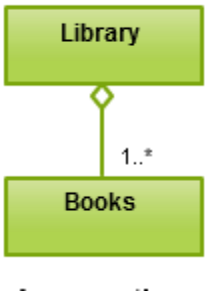
Multiplicity



Multiplicity

is the active logical association when the cardinality of a class in relation to another is being depicted. For example, one fleet may include multiple airplanes, while one commercial airplane may contain zero to many passengers. The notation 0..* in the diagram means “zero to many”.

Aggregation

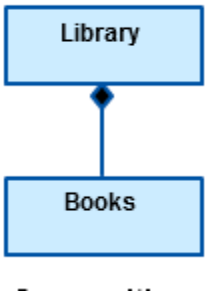


Aggregation

refers to the formation of a particular class as a result of one class being aggregated or built as a collection. For example, the class “library” is made up of one or more books, among other materials. In aggregation, the contained classes are not strongly dependent on the lifecycle of the container. In the same example, books will remain so even when the library is dissolved. To show aggregation in a diagram, draw a line from the parent class to the child class with a diamond shape near the parent class.

To show aggregation in a diagram, draw a line from the parent class to the child class with a diamond shape near the parent class.

Composition

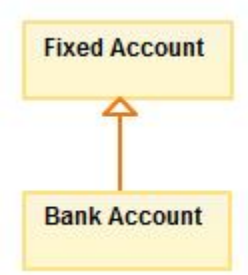


Composition

The composition relationship is very similar to the aggregation relationship. with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class. That is, the contained class will be obliterated when the container class is destroyed. For example, a shoulder bag's side pocket will also cease to exist once the shoulder bag is destroyed.

To show a composition relationship in a UML diagram, use a directional line connecting the two classes, with a filled diamond shape adjacent to the container class and the directional arrow to the contained class.

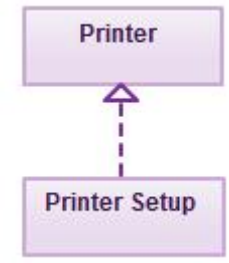
Inheritance / Generalization



Inheritance

refers to a type of relationship wherein one associated class is a child of another by virtue of assuming the same functionalities of the parent class. In other words, the child class is a specific type of the parent class. To show inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead.

Realization



Realization

denotes the implementation of the functionality defined in one class by another class. To show the relationship in UML, a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality of the class that implements the function. In the example, the printing preferences that are set using the printer setup interface are being implemented by the printer.

Drawing class diagrams using Creately

We've given a lot of thought to relationships when we built our class diagramming tools. Our connectors adjust to the context and show only the most logical relationships when connecting classes. This significantly reduced your chances of making a mistake.

Drawing from scratch can be cumbersome. You can get started immediately using our professionally designed class diagrams. Browse our class diagram examples and pick the one that's closely related to your system.

Any more questions about class diagram relationships?

I hope I've clearly explained the various relationships between class diagrams. They are not as complex as you think and can be mastered with some practice. And by using our tool you shouldn't have any trouble coming up with class diagrams. If you have any more questions don't hesitate to ask in the comments section. Also, check out this guide to UML Diagram Types with Examples for further reading.

Identify and Model Classes—Which Classes do We Need?

An analysis of the interrelationships, information needs, and actors and prototypes is conducted on the basis of general domain knowledge, discussions with experts, and documents. The questions that should be asked are:

- What are the most important things that will be worked with in the IT system?
- What classes can be created from this?

The answers to these questions provide a number of potential classes, which we model in a first draft of the class diagram. In practice, the results of this first work step vary greatly. However, we have never experienced a case in which nothing at all was found. If you are still inexperienced in identifying classes, it has proven

helpful to run through top-down analysis repeatedly. With time, you will develop a sense for what is a class and what is not (Figure 4.36):

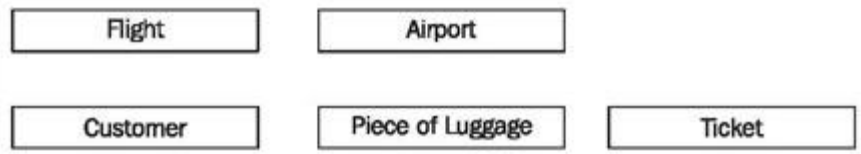


Figure 4.36 Potential classes

Identify and Model Associations—How Are the Classes Connected?

We model the interconnections between the obtained classes and business rules in class diagrams as associations with meaningful names and multiplicities, as shown in Figure 4.37. The questions are:

- What relationships exist among objects??
- How many objects of each class are involved in a relationship?

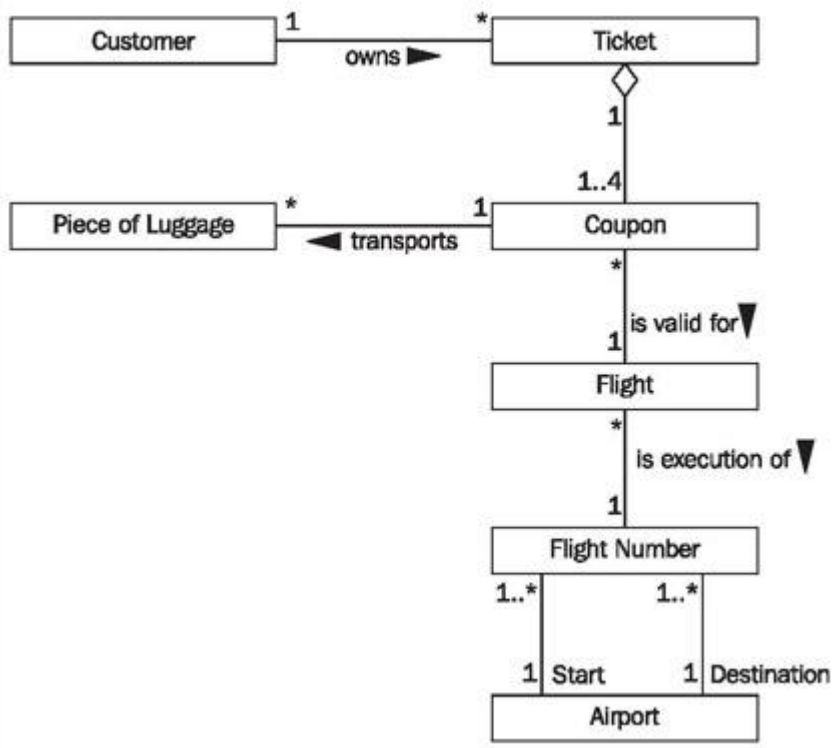


Figure 4.37 Class and Associations

The first question has to be asked for objects of each pair of different classes, for instance, for the classes flight and customer from our case study. Here, it is important to recognize whether the relationship is direct, or if the relationship only exists indirectly through other objects. In our example it turns out that a customer owns a ticket, which in turn, consists of coupons, which are valid for a flight. The goal of the second question is to determine the multiplicity of the relationship, for instance, how many tickets a customer can have, and to how many customers a ticket belongs (Figure 4.37).

Even though at the beginning of this work step we started with previously found classes, because of the domain discussions, we generally find more classes in this work step.

Define Attributes—What do We Want to Know about the Objects?

The required information about a class has to be identified and modeled in the form of attributes. The question for this is:

- Which information about a certain class am I interested in??

This question is about finding obviously needed attributes of the individual classes (see Figure 4.38). This question cannot be answered completely without precisely analyzing inputs and queries, as takes place in bottom-up analysis. Because of this, not too much time should be spent answering this question.

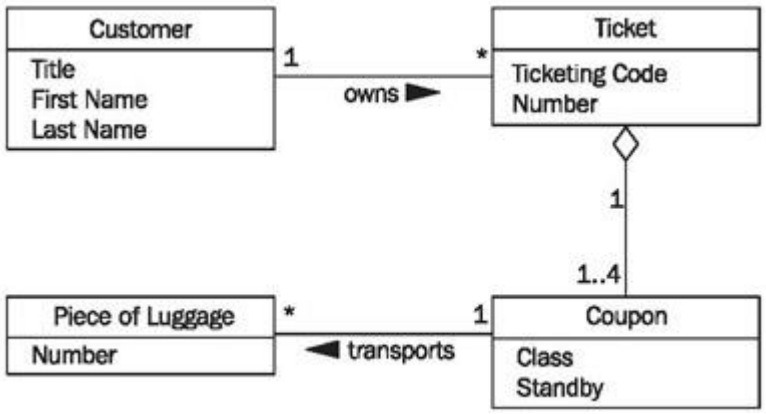


Figure 4.38 Classes and Attributes

List Required Queries and Inputs—What does the IT System Need to Deliver and Accept?

In this first work step of bottom-up analysis, the individual queries and inputs of the IT system have to be identified. The queries are more important here, because answering queries is the real purpose of IT systems. The questions are:

- What information does the IT system have to be able to provide??
- What information does the IT system have to be able to accept?

When answering these questions, you can build upon the use cases already found. Which queries and mutations occur in a use case is already drafted in the use case sequence diagram. Another source of information are the business processes of the business system (see ***Modeling Business Systems***). The result of this work step is a list of information requirements, as illustrated in Figure 4.39:

Requirements	Type	Use Case
Boarding Pass	Output	Generating Boarding Pass
Passenger List	Output	Complete Boarding
Ticket Details	Output	Check-In, Express Check-In
Customer Details	Output	Check-In, Express Check-In
Coupon	Input	Check-In, Express Check-In
...		

Figure 4.39 List of information requirements

Formulate Queries and Inputs—How Exactly Should the Display Look?

In order to create individual class diagrams for the individual queries and inputs, we first need to be define how they look. Complex query results or inputs are collected or drafted. Figure 4.40 shows a passenger list; further examples can be found in Figure 4.66 (display) and Figure 4.67 (boarding pass). The question is:

- How precisely does the display of a query or input look??

Good sources of information are already existing forms (for example, the passenger list from Figure 4.40) and displays from the prototypes:

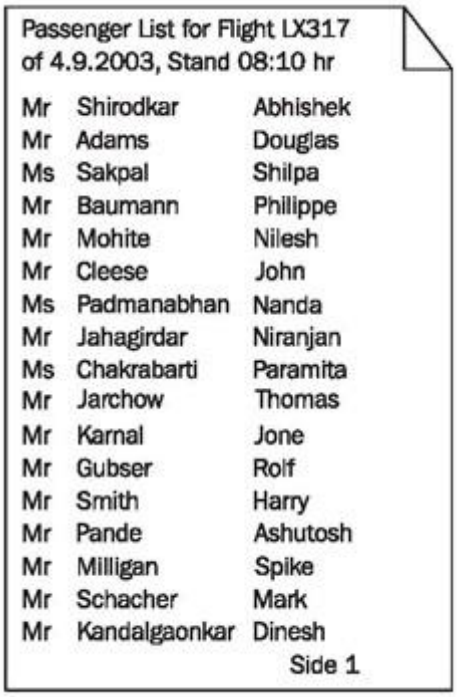


Figure 4.40 Passenger list

Conduct Information Analysis—Which Classes, Associations, and Attributes Do We Need?

In this work step the main part of the bottom-up analysis is performed. For each query or input a small class diagram is created on the basis of the existing classes.

This is achieved by modeling the drafted inputs and outputs of the IT system. Class modeling on a small scale takes place. The questions are:

- What data elements exist in input and output??
- What objects hide behind these data elements?
- What relationships exist between the objects that were found?
- Which of the objects that have already been modeled can be used?

For the passenger list in Figure 4.40, the class diagram in Figure 4.41 can be constructed:

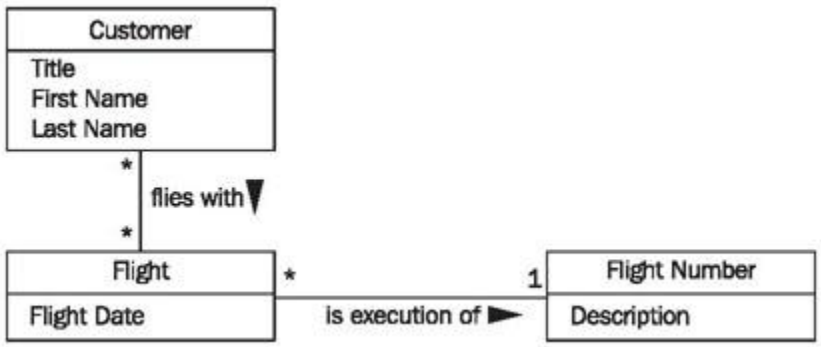


Figure 4.41 Class diagram for the passenger list

Taking into consideration the classes that were already found in the top-down analysis, the class diagram in Figure 4.42 is constructed:

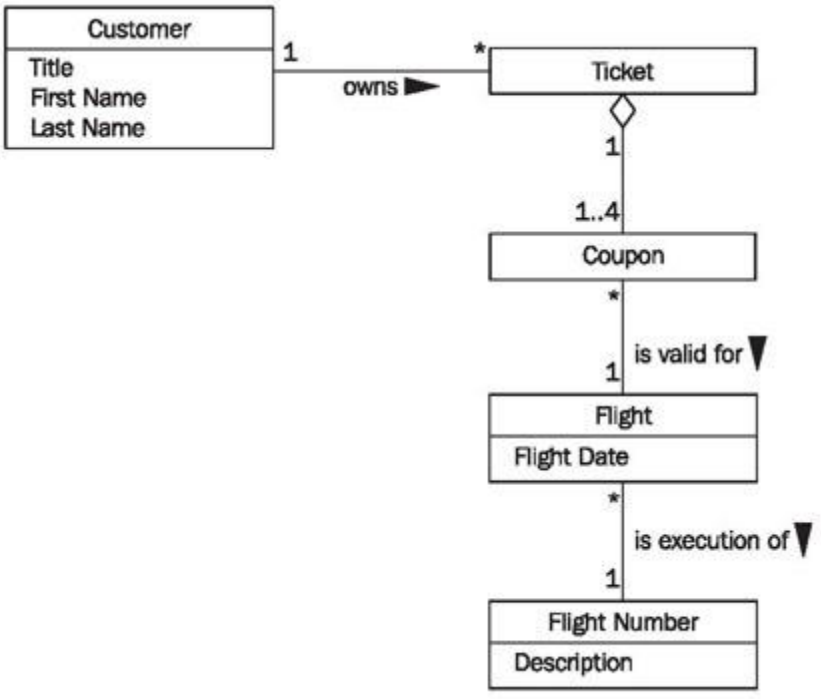


Figure 4.42 Edited class diagram for passenger list

Consolidate Class Diagrams—How Does Everything Fit Together?

In this last work step, if it has not been done yet, the individual class diagrams have to be consolidated into one cumulative class diagram. Here, inconsistencies have to be discovered and corrected. Applicable questions are:

- Are there classes in the individual class diagrams that have different names, but represent the same thing?
- Are there multiple relationships in individual class diagrams that have the same meaning?
- Are there attributes within classes that are named differently, but that have the same meaning?

In fact, when all individual class diagrams are being consolidated to one cumulative diagram, these questions almost pose themselves. Once inconsistencies have been recognized, they can usually be corrected easily. If you used the classes found during top-down analysis for modeling the drafted inputs and outputs, overlaps and conflicts during the consolidation of the individual class diagrams should be limited anyway.

Verify the Class Diagrams—Is Everything Correct?

The completed class diagram in the structural view can be verified with the following checklist:

Checklist 4.5 Verifying Class Diagrams of the Structural View

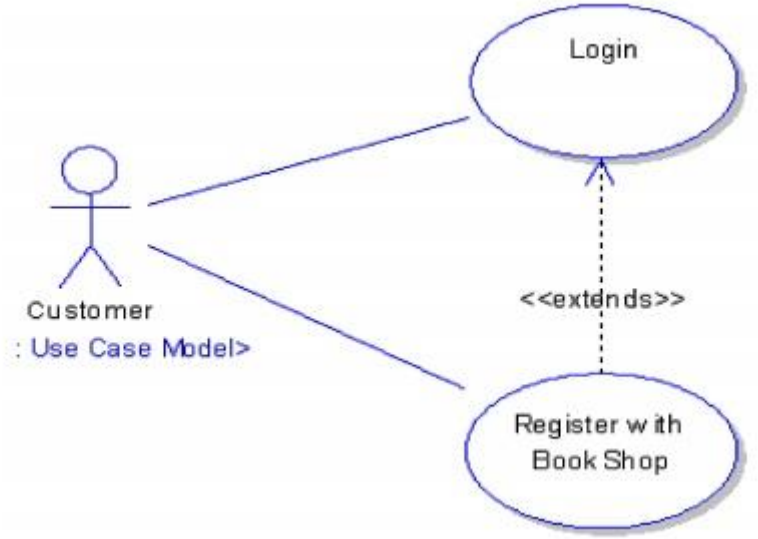
- Is the class diagram complete? This question will be answered in ***Interaction View***. In the interaction view, we will show how the class diagram can be used to answer all required queries of the IT system. If this is possible, the class diagram is complete.
- Is the class diagram correct? The second question, the question about correctness, is a little bit more difficult to answer. Our experience shows that intensive collaborative reading of class diagrams together with the knowledge carriers will bring to light most mistakes. In addition to that, the class diagram can be tested for suspect structure patterns. The best way to do this is with a suitable tool. An introduction to the analysis of structure patterns would go beyond the scope of the text

The Use Case Model

The Use Case Model describes the proposed functionality of the new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system. A Use Case is a single unit of meaningful work; for example login to system, register with system and create order are all Use Cases. Each Use Case has a description which describes the functionality

that will be built in the proposed system. A Use Case may 'include' another Use Case's functionality or 'extend' another Use Case with its own behaviour.

Use Cases are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work



A Use Case description will generally include:

1. General comments and notes describing the use case.
2. Requirements
3. Constraints
4. Scenarios
5. Scenario diagram

Actors

An Actor is a user of the system. This includes both human users and other computer systems. An Actor uses a Use Case to perform some piece of work which is of value to the business. The set of Use Cases an actor has access to defines their overall role in the system and the scope of their action.

Constraints, Requirements and Scenarios

The formal specification of a Use Case includes:

1. Requirements. These are the formal functional requirements that a Use Case must provide to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.

2. Constraints. These are the formal rules and limitations that a Use Case operates under, and includes pre- post- and invariant conditions. A pre-condition specifies what must have already occurred or be in place before the Use Case may start. A post-condition documents what will be true once the Use Case is complete. An invariant specifies what will be true throughout the time the Use Case operates.
3. Scenarios. Scenarios are formal descriptions of the flow of events that occurs during a Use Case instance. These are usually described in text and correspond to a textual representation of the Sequence Diagram.

Includes and Extends relationships between Use Cases

One Use Case may include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case will be called every time the basic path is run. An example may be to list a set of customer orders to choose from before modifying a selected order - in this case the <list orders> Use Case may be included every time the <modify order> Use Case is run.

A Use Case may be included by one or more Use Cases, so it helps to reduce duplication of functionality by factoring out common behaviour into Use Cases that are re-used many times.

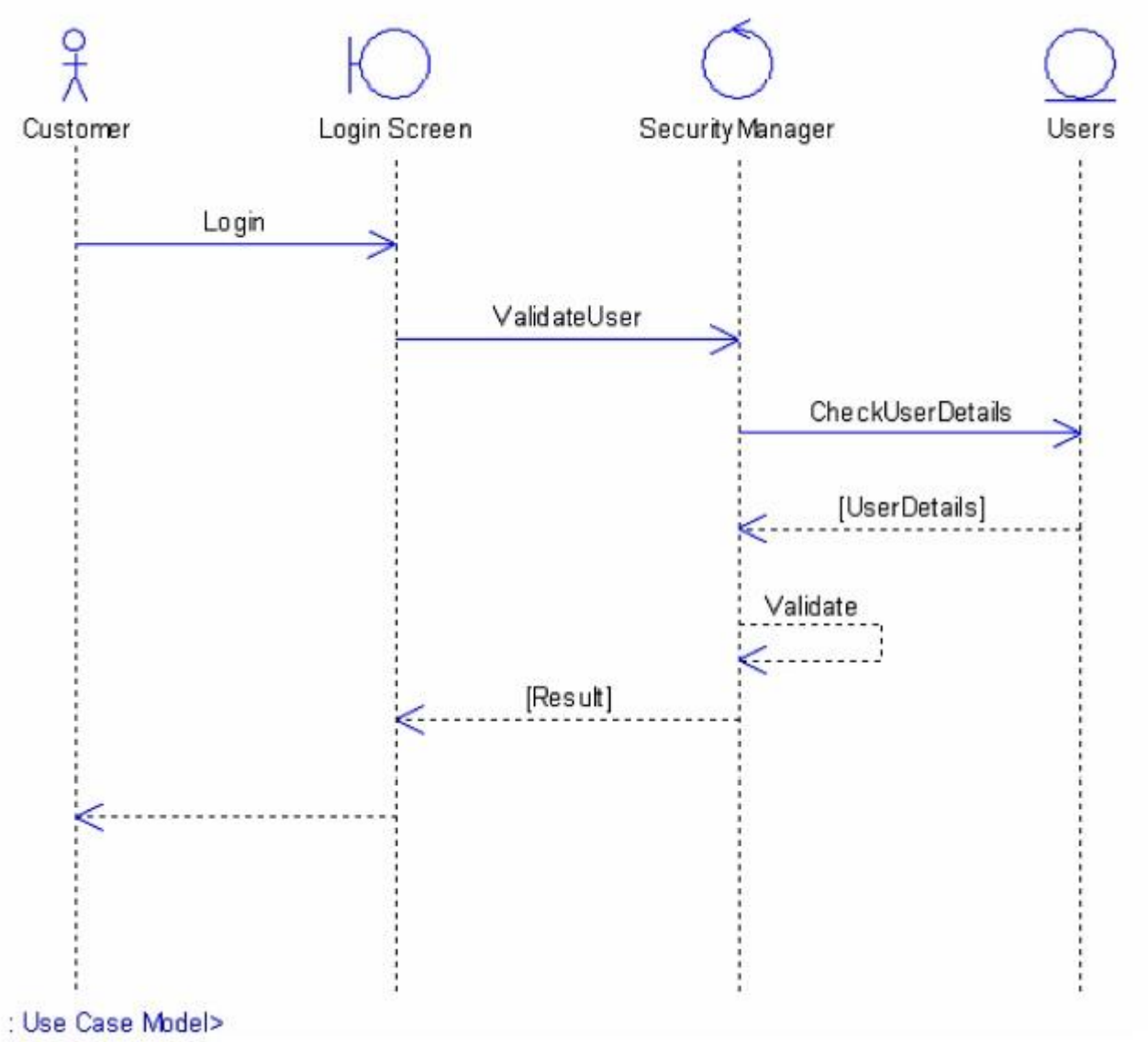
One Use Case may extend the behaviour of another - typically when exceptional circumstances are encountered. For example, if before modifying a particular type of customer order, a user must get approval from some higher authority, then the <get approval> Use Case may optionally extend the regular <modify order> Use Case.

Sequence Diagrams

UML provides a graphical means of depicting object interactions over time in Sequence Diagrams. These typically show a user or actor, and the objects and components they interact with in the execution of a use case. One sequence diagram typically represents a single Use Case 'scenario' or flow of events.

Sequence diagrams are an excellent way to document usage scenarios and to both capture required objects early in analysis and to verify object usage later in design. Sequence diagrams show the flow of messages from one object to another, and as such correspond to the methods and events supported by a class/object.

The diagram illustrated below shows an example of a sequence diagram, with the user or actor on the left initiating a flow of events and messages that correspond to the Use Case scenario. The messages that pass between objects will become class operations in the final model.



Implementation Diagram

A Use Case is a formal description of functionality the system will have when constructed. An implementation diagram is typically associated with a Use Case to document what design elements (eg. components and classes) will implement the Use Case functionality in the new system. This provides a high level of traceability for the system designer, the customer and the team that will actually build the system. The list of Use Cases that a component or class is linked to documents the minimum functionality that must be implemented by the component

Relationships Between Use Cases

Use cases could be organized using following relationships:

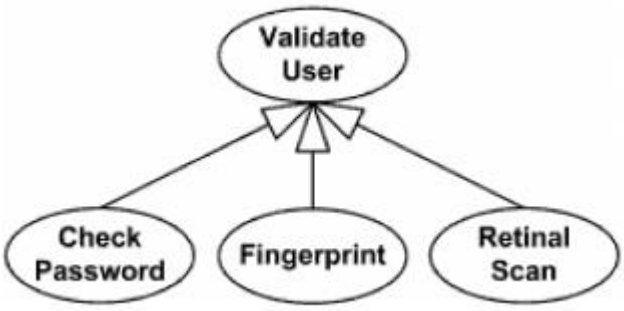
1. Generalization
2. Association
3. Extend
4. Include

1. Generalization Between Use Cases

Generalization between use cases is similar to generalization between classes – child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.

Notation:

Generalization is rendered as a solid directed line with a large open arrowhead (same as generalization between classes).



Generalization between use cases

2. Association Between Use Cases

Use cases can only be involved in binary Associations. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the system.

3. Extend Relationship

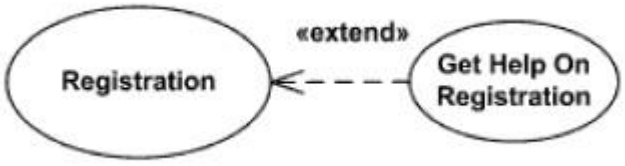
Extend is a directed relationship from an **extending use case** to an extended use case that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the use case to be extended

Note: Extended use case is meaningful on its own, independently of the extending use case, while the extending use case typically defines behavior that is not necessarily meaningful by itself.

The extension takes place at one or more extension points defined in the extended use case.

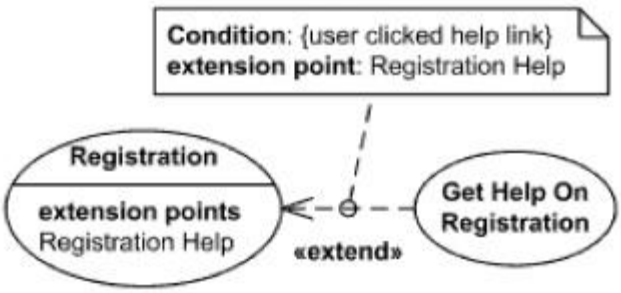
The extend relationship is owned by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended.

Extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the extending use case to the extended (base) use case. The arrow is labeled with the keyword «extend».



Registration use case is meaningful on its own, and it could be extended with optional
Get Help On Registration use case

The condition of the extend relationship as well as the references to the extension points are optionally shown in a Note attached to the corresponding extend relationship.



⎪ Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

Extension Point

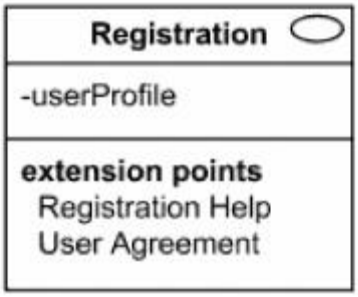
⎪ An extension point is a feature of a use case which identifies (references) a point in the behavior of the use case where that behavior can be extended by some other (extending) use case, as specified by an extend relationship.

⎪ Extension points may be shown in a compartment of the use case oval symbol under the heading extension points. Each extension point must have a name, unique within a use case. Extension points are shown as text string according to the syntax: <extension point> ::= <name> [: <explanation>]

⎪ The optional description is given usually as informal text, but can also be given in other forms, such as the name of a state in a state machine, an activity in an activity diagram, a precondition, or a postcondition.

⎪ Registration use case with extension points Registration Help and User Agreement

⎪ Extension points may be shown in a compartment of the use case rectangle with ellipse icon under the heading extension points.



Extension points of the Registration use case shown using the rectangle notation

4. Include Relationship

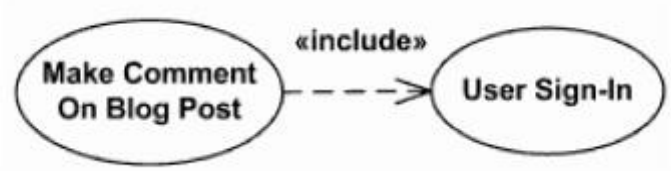
⎪ An include relationship is a directed relationship between two use cases, implying that the behavior of the required (not optional) included use case is inserted into the behavior of the including (base) use case. Including use case depends on the addition of the included use case.

⎪ The include relationship is intended to be used when there are common parts of the behavior of two or more use cases. This common part is extracted into a separate use case to be included by all the base use cases having this part in common.

⎪ Execution of the included use case is analogous to a subroutine call or macro command in programming. All of the behavior of the included use case is executed at a single location in the including use case before execution of the including use case is resumed.

⎪ As the primary use of the include relationship is to reuse common parts, including use cases are usually not complete by themselves but dependent on the included use cases.

⎪ Include relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the keyword «include».



Four Phases of Unified Process

⎫ The Unified Process consists of cycles that may repeat over the long-term life of a system. A cycle consists of four phases: Inception, Elaboration, Construction and Transition. Each cycle is concluded with a release, there are also releases within a cycle. Let's briefly review the four phases in a cycle:

Inception Phase - During the inception phase the core idea is developed into a product vision. In this phase, we review and confirm our understanding of the core business drivers. We want to understand the business case for why the project should be attempted. The inception phase establishes the product feasibility and delimits the project scope.

Elaboration Phase - During the elaboration phase the majority of the Use Cases are specified in detail and the system architecture is designed. This phase focuses on the "Do-Ability" of the project. We identify significant risks and prepare a schedule, staff and cost profile for the entire project.

Construction Phase - During the construction phase the product is moved from the architectural baseline to a system complete enough to transition to the user community. The architectural baseline grows to become the completed system as the design is refined into code.

Transition Phase - In the transition phase the goal is to ensure that the requirements have been met to the satisfaction of the stakeholders. This phase is often initiated with a beta release of the application. Other activities include site preparation, manual completion, and defect identification and correction. The transition phase ends with a postmortem devoted to learning and recording lessons for future cycles.