

INTRODUCTION TO OOAD WITH OO BASICS

Object Oriented analysis and design skills are essential for the creation of well-designed, robust, and maintainable software using OO technologies and languages such as Java or C#.

Requirement's analysis and OOA/D needs to be presented and practiced in the context of some development process. In this case, an **agile** (light, flexible) approach to the well-known **Unified Process** (UP) is used as the sample **iterative development process** are taken. It includes

- Apply principles and patterns to create better object designs.
- Iteratively follow a set of common activities in analysis and design, based on an agile approach to the UP as an example.
- Create frequently used diagrams in the UML notation.

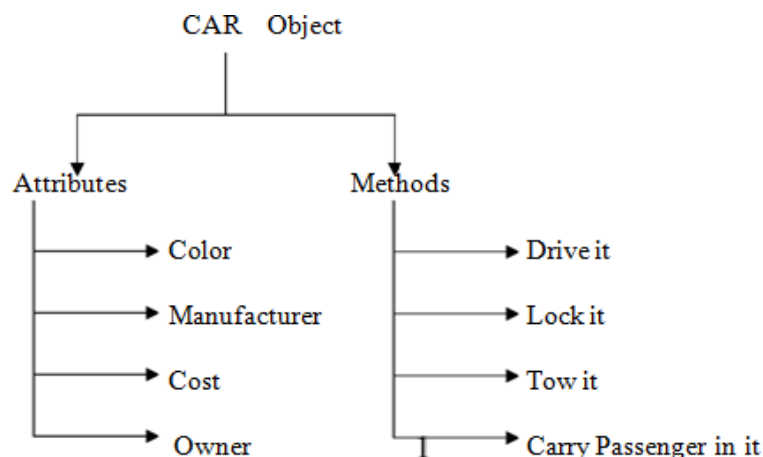
What is Analysis and Design?

Analysis (do the right thing) emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new online trading system is desired, how will it be used? What are its functions?

Design (do the thing right) emphasizes a conceptual solution (in software and hardware) that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Design ideas often exclude low-level or "obvious" detail.

OO Basics

Object: A car is an object a real-world entity, identifiably separate from its surroundings. A car has a well-defined set of attributes in relation to other object.



Attributes: Data of an object. ,Properties of an object.

Methods: Procedures of an object. or Behavior of an object.

The term object was for formal utilized in the similar language. The term object means a combination of data and logic that represent some real-world entity.

When developing an object oriented applications, two basic questions arise,

1. What objects does the application need?
2. What functionality should those objects have?

Programming in an object-oriented system consists of adding new kind of objects to the system and defining how they behave. The new object classes can be built from the objects supplied by the object-oriented system.

Object state and properties (Attributes)

Properties represent the state of an object. In an object oriented methods we want to refer to the description of these properties rather than how they are represented in a particular programming language.

Attributes of car object

We could represent each property in several ways in a programming languages.

For example: Color

1. Can be declared as character to store sequence or character [ex: red, blue, ..]
2. Can declared as number to store the stock number of paint [ex: red paint, blue paint, ..]
3. Can be declared as image (or) video file to refer a full color video image.

The importance of this distinction is that an object abstract state can be independent of its physical representation.

Object Behavior and Methods:

We can describe the set of things that an object can do on its own (or) we can do with it. For example: Consider an object car, We can drive the car. We can

stop the car.

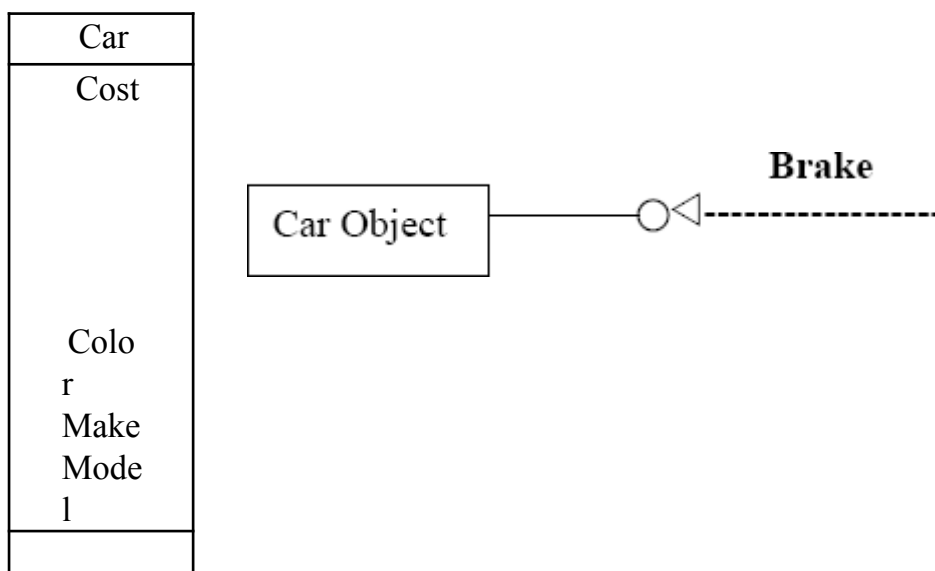
Each of the above statements is a description of the object's behavior. The

objects behavior is described in methods or procedures. A method is a function or procedures that is defined in a class and typically can access to perform some operation. Behavior denotes the collection of methods that abstractly describes what an object is capable of doing. The object which operates on the method is called receiver. Methods encapsulate the behavior or the object, provide interface to the object and hide any of the internal structures and states maintained by the object. The procedures provide us the means to communicate with an object and access its properties.

For example: An employee object knows how to compute salary. To compute an employee salary, all that is required is to send the compute payroll message to the employee object.

Objects Respond to Messages: The capability of an object's is determined by the methods defined for it. To do an operation, a message is sent to an object. Objects represented to messages according to the methods defined in its class.

For example:



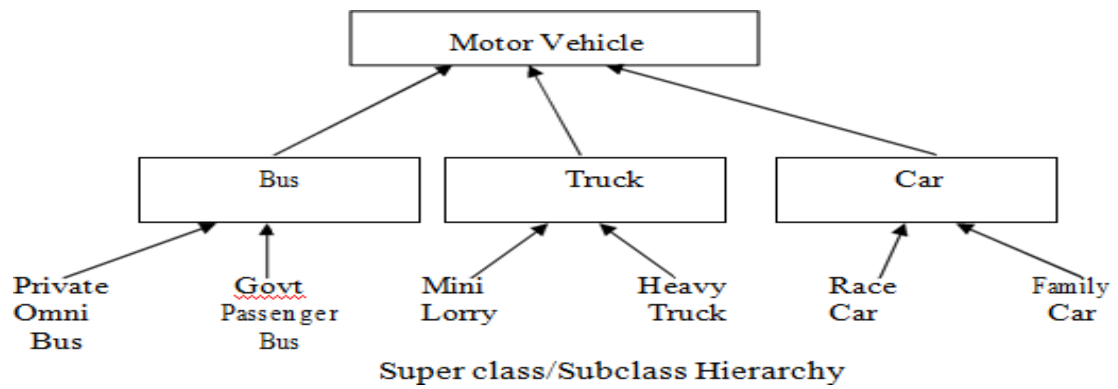
When we press on the brake pedal of a car, we send a stop message to the car object. The car object knows how to respond to the stop message since brake have been designed with specialized parts such as brake pads and drums precisely respond to that message.

Different object can respond to the same message in different ways. The car, motorcycle and bicycle will all respond to a stop message, but the actual operations performed are object specific.

It is the receiver's responsibility to respond to a message in an appropriate manner. This gives the great deal of flexibility, since different object can respond to the same message in different ways. This is known as polymorphism.

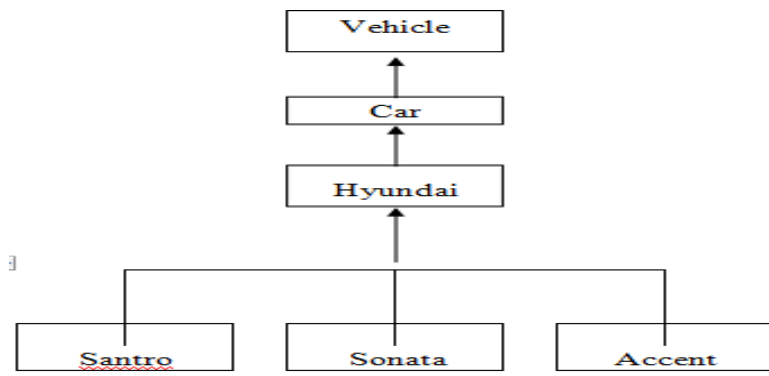
Class Hierarchy

An object-oriented system organizes classes into a subclass super class hierarchy.



The properties and behaviors are used as the basis for making distinctions between classes are at the top and more specific are at the bottom of the class hierarchy. The family car is the subclass of car. A subclass inherits all the properties and methods defined in its super class.

Inheritance: It is the property of object-oriented systems that allow objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The derived class holds the properties and behavior of base class in addition to the properties and behavior of derived class.



Dynamic Inheritance

Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class.

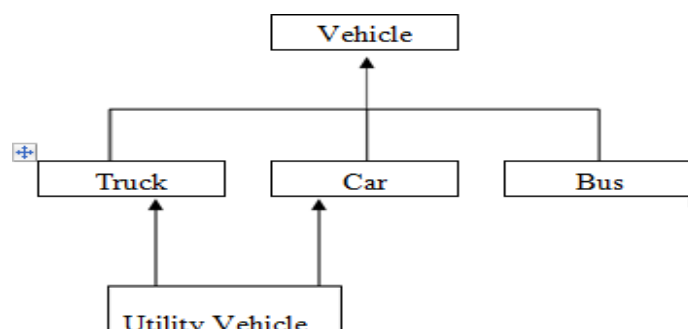
Example:

A window objects change to icon and back again. When we double click the folder the contents will be displayed in a window and when close it, changes back to icon. It involves changing a base class between a windows class and icon class.

Multiple Inheritances

Some object-oriented systems permit a class to inherit its state (attributes) and behavior from more than one super class. This kind of inheritance is referred to as multiple inheritances.

For example: Utility vehicle inherits the attributes from the Car and Truck classes.



Encapsulation and Information Hiding

Information hiding is the principle of concealing the internal data and procedures of an object. In C++ , encapsulation protection mechanism with private, public and protected members.

A car engine is an example of encapsulation. Although engines may differ in implementation, the interface between the driver and car is through a common protocol.

Polymorphism

Poly □ "many" Morph □ "form"

It means objects that can take on or assume many different forms. Polymorphism means that the same operations may behave differently on different classes. Booch defines polymorphism as the relationship of objects many different classes by some common super class. Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation detail to the objects involved.

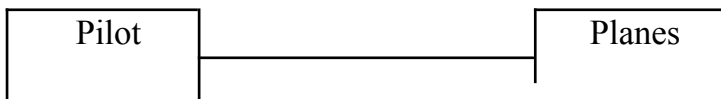
Example: In a pay roll system, manager, office worker and production worker objects all will respond to the compute payroll message, but the actual operations performed are object specific.

Object Relationship and Associations

Association represents the relationships between objects and classes. Associations are bi-directional. The directions implied by the name are the forward direction and the opposite is the inverse direction.

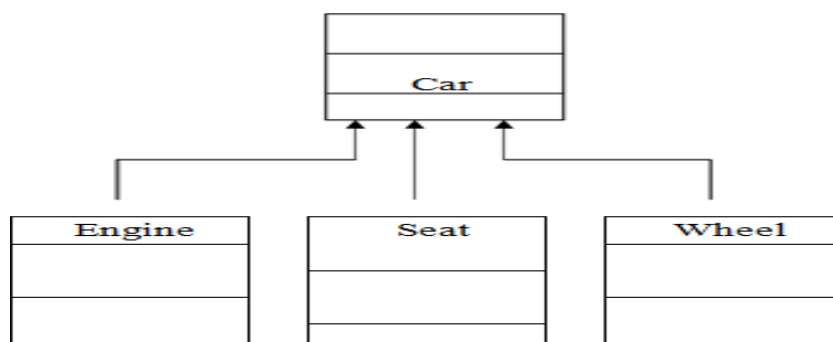
Can fly

Flown by



A pilot "can fly" planes. The inverse of can fly is "is flown by ". Plane "is flown by" pilot

Aggregations: All objects, except the most basic ones, are composed of and may contain other objects. Breaking down objects in to the objects from which they are composed is de



composition. This is possible because an object attributes need not be simple data fields, attributes can reference other objects. Since each object has an identity, one object can refer to other objects. This is known as aggregation. The car object is an aggregation of other objects such as engine, seat and wheel objects.

Static and Dynamic Binding:

Determining which function has to be involved at compile time is called static binding. Static binding optimized the calls. (Ex) function call.

The process of determining at run time which functions to involve is termed dynamic binding. Dynamic binding occurs when polymorphic call is issued. It allows some method invocation decision to be deferred until the information is known.

Object Persistence:

Objects have a lifetime. They are explicitly created and can exist for a period of time that has been the duration of the process in which they were created. A file or database can provide support for objects having a longer lifeline, longer than the duration of the process for which they are created. This characteristic is called object persistence.

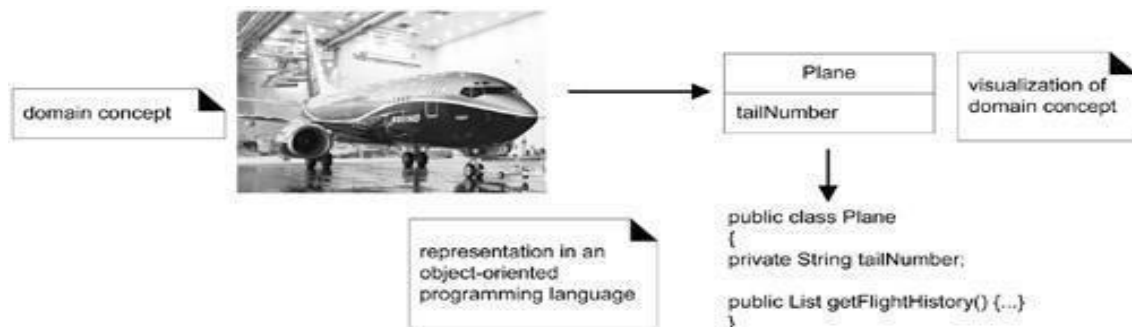
What is Object-Oriented Analysis and Design?

Object-Oriented Analysis : It emphasis on finding and describing the objects or concepts in the problem domain. For example, in the case of the flight information system, some of the concepts include Plane, Flight, and Pilot.

Object-Oriented Design : It emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, a Plane software object may have a tail Number attribute and a get FlightHistory method.

During implementation or object-oriented programming, design objects are implemented, such as a Plane class in Java.

Object-orientation emphasizes representation of objects.



Example: a simple example a "dice game" in which software simulates a player rolling two dice. If the total is seven, they win; otherwise, they lose.



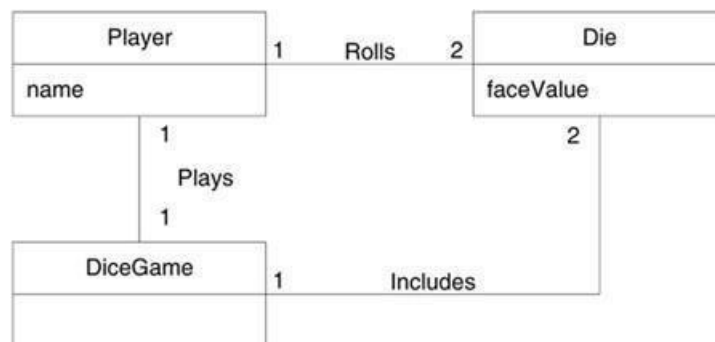
i) **Define Use Cases:** Requirements analysis may include stories or scenarios of how people use the application; these can be written as use cases. They are a popular tool in requirements analysis. For example, the Play a Dice Game use case:

Play a Dice Game: Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.

ii) Define a Domain Model

There is an identification of the concepts, attributes, and associations that are considered important. The result can be expressed in a domain model that shows the important domain concepts or objects.

Partial domain model of the dice game.



This model illustrates the important concepts Player, Die, and Dice Game, with their associations and attributes. It is a visualization of the concepts or mental models of a real-world domain. Thus, it has also been called a **conceptual object model**.

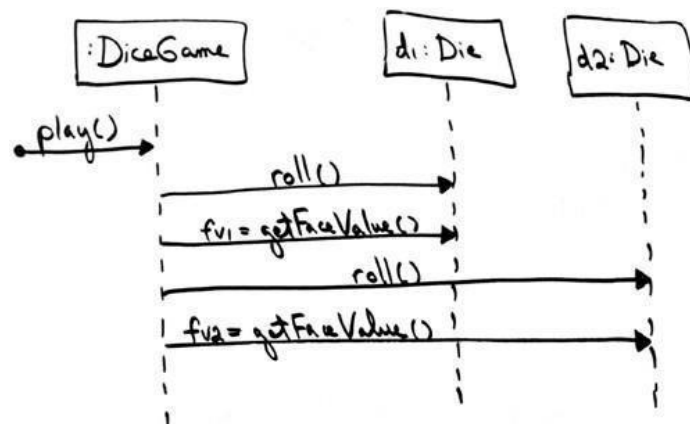
iii) Assign Object Responsibilities and Draw Interaction Diagrams

Object-oriented design is concerned with defining software objects their responsibilities and collaborations. It shows the flow of messages between software objects, and thus the invocation of methods.

For example, the sequence diagram in Figure 1.4 illustrates an OO software design, by sending messages to instances of the DiceGame and Die classes.

Notice that although in the real world a player rolls the dice, in the software design the DiceGame object "rolls" the dice (that is, sends messages to Die objects). Software object designs and programs do take some inspiration from real-world domains, but they are not direct models or simulations of the real world.

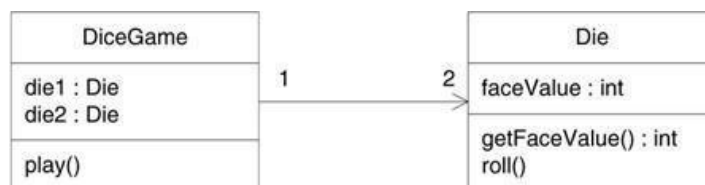
Sequence diagram illustrating messages between software objects.



iv) **Define Design Class Diagrams:** a static view of the class definitions is usefully shown with a design class diagram. This illustrates the attributes and methods of the classes.

For example, in the dice game, an inspection of the sequence diagram leads to the partial design class diagram shown in Figure 1.5. Since a play message is sent to a DiceGame object, the DiceGame class requires a play method, while class Die requires a roll and get FaceValue method.

Partial design class diagram.



UML DIAGRAMS

What is the UML?

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems. The word visual in the definition is a key point -the UML is the de facto standard diagramming notation for drawing or presenting pictures. The standard is managed, and was created, by the Object Management Group. It was first added to the list of OMG adopted technologies in 1997.

UML is composed of 9 graphical diagrams:

- 1) **Class Diagram** - describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- 2) **Use – Case Diagram** - describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use

cases.

3) Behavior Diagram

a. Interaction Diagram

- i. **Sequence Diagram** - shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespan of objects relative to those messages.
- ii. **Communication diagram:-** how the interactions between objects or parts in terms of sequenced messages.

b. **State Chart Diagram** - describes the states and state transitions of the system.

c. **Activity Diagram** - describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.

4) Implementation Diagram

- a. **Component Diagram** - describes how a software system is split up into components and shows the dependencies among these components.
- b. **Deployment Diagram** - describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.

Three Ways to Apply UML

UML as sketch : Informal and incomplete diagrams created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.

UML as blueprint: Relatively detailed design diagrams used either for

1) **Reverse Engineering:** UML tool reads the source or binaries and generates UML package, class, and sequence diagrams to visualize and better understanding of existing code in UML diagrams.

2) **Forward Engineering:** code generation. Before programming, some detailed diagrams can provide guidance for code generation (e.g., in Java), either manually or automatically with a tool. It's common that the diagrams are used for some code, and other code is filled in by a developer while coding

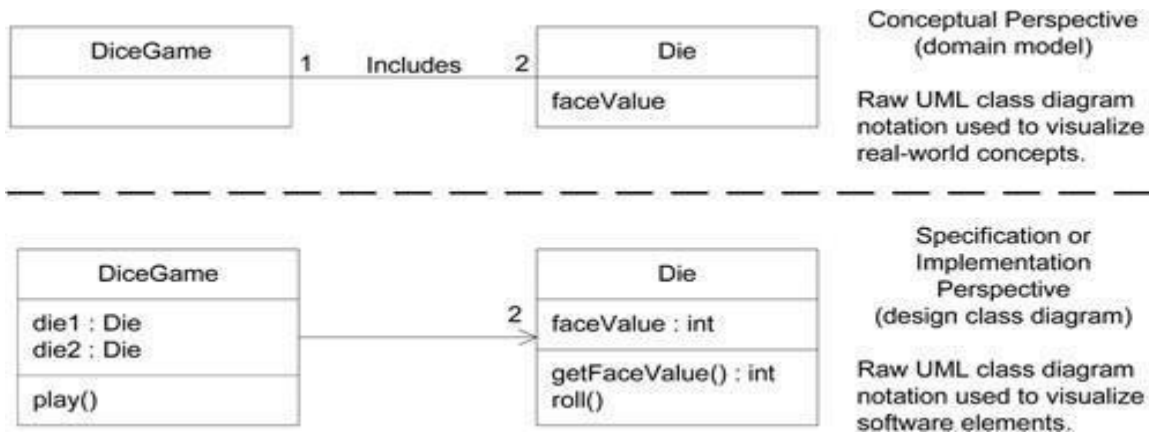
UML as programming language: Complete executable specification of a software system in UML. Executable code will be automatically generated

Three Perspectives to Apply UML

The same UML class diagram notation can be used to draw pictures of concepts in the real

world or software classes in Java.

1. **Conceptual perspective** - the diagrams are interpreted as describing things in a situation of the real world or domain of interest.
2. **Specification (software) perspective** - the diagrams (using the same notation as in the conceptual perspective) describe software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
3. **Implementation (software) perspective** - the diagrams describe software implementations in a particular technology (such as Java).



The Meaning of "Class" in Different Perspectives

Conceptual class real-world concept or thing. A conceptual or essential perspective. The UP-Domain Model contains conceptual classes.

Software class a class representing a specification or implementation perspective of a software component, regardless of the process or method.

Implementation class a class implemented in a specific OO language such as Java.

UNIFIED PROCESS (UP)

What is the UP?

A software development process describes an approach to building, deploying, and possibly maintaining software. The Unified Process has emerged as a popular iterative software development process for building object-oriented systems. In particular, the Rational Unified Process or RUP a detailed refinement of the Unified Process, has been widely adopted.

The UP combines commonly accepted best practices, such as an **iterative lifecycle and**

risk-driven development, into a cohesive and well-documented process description.

UP for three reasons

1. The UP is an iterative process.
2. UP practices provide an example structure for how to do and thus how to explain OOA/D.
3. The UP is flexible, and can be applied in a lightweight and agile approach that includes practices from other agile methods

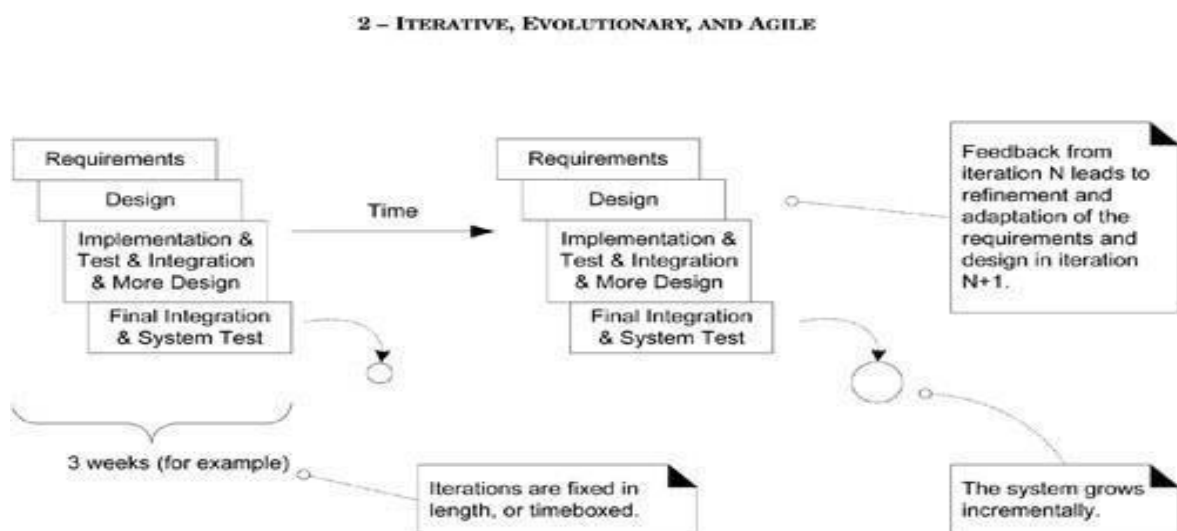
Iterative and Evolutionary Development

A key practice in both the UP and most other modern methods is iterative development.

In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called iterations; the outcome of each is a tested, integrated, and executable partial system. Each iteration includes its own requirements analysis, design, implementation, and testing activities.

The system grows incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental development. Because feedback and adaptation evolve the specifications and design, it is also known as iterative and evolutionary development.

Iterative and evolutionary development.



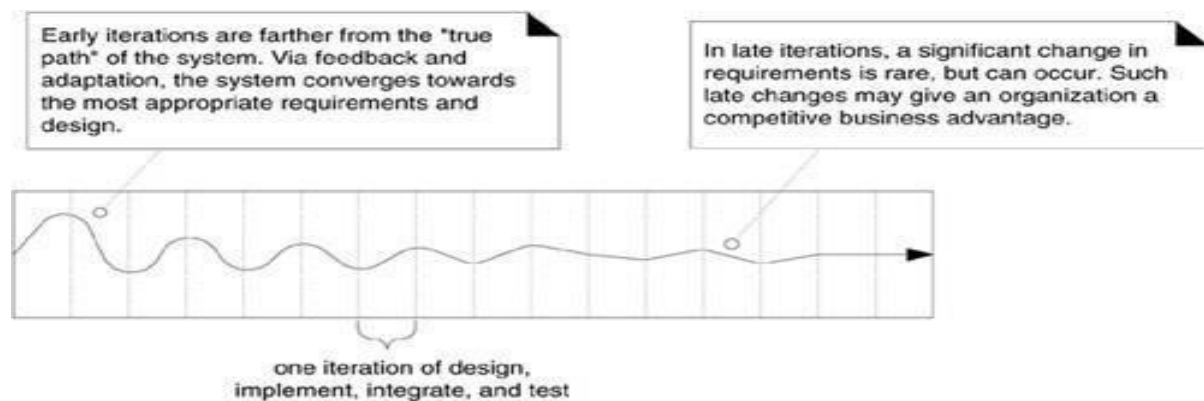
Notice in this example that there is neither a rush to code, nor a long drawn-out design step that attempts to perfect all details of the design before programming. The system may not be eligible for production deployment until after many iterations; for example, 10 or 15 iterations.

To Handle Change on an Iterative Project

Each iteration involves choosing a small subset of the requirements, and quickly designing, implementing, and testing

In addition to requirements clarification, activities such as load testing will prove if the partial design and implementation are on the right path, or if in the next iteration, a change in the core architecture is required.

Iterative feedback and evolution leads towards the desired system. The requirements and design instability lowers over time.



Work proceeds through a series of structured build-feedback-adapt cycles. In early iterations the deviation from the "true path" of the system (in terms of its final requirements and design) will be larger than in later iterations. Over time, the system converges towards this path, as illustrated in Figure

Benefits of Iterative Development

less project failure, better productivity, and lower defect rates; shown by research into iterative and evolutionary methods

early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth)

early visible progress

early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders

managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps

the learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

What is Iteration Time boxing?

Most iterative methods recommend an iteration length between two and six weeks.

Small steps, rapid feedback, and adaptation are central ideas in iterative development; long iterations subvert the core motivation for iterative development and increase project risk.

A very long time-boxed iteration misses the point of iterative development. Short is good.

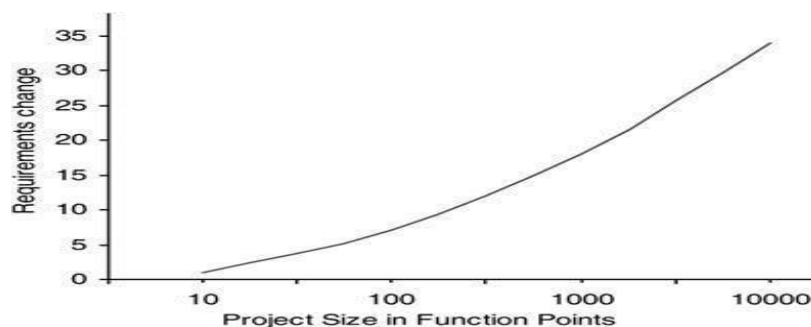
Iterations are time-boxed, or fixed in length. For example, if the next iteration is chosen to be three weeks long, then the partial system must be integrated, tested, and stabilized by the scheduled date-date slippage is illegal. If it seems that it will be difficult to meet the deadline, the recommended response is to de-scope-remove tasks or requirements from the iteration, and include them in a future iteration, rather than slip the completion date.

In a waterfall lifecycle process, there is an attempt to define all or most of the requirements before programming. It is strongly associated with

high rates of failure

lower productivity

higher defect rates



Percentage of change on software projects of varying sizes.

The Need for Feedback and Adaptation

In complex, changing systems feedback and adaptation are key ingredients for success.

Feedback from early development, programmers trying to read specifications, and client demos to refine the requirements.

Feedback from tests and developers to refine the design or models.

Feedback from the progress of the team tackling early features to refine the schedule and estimates.

Feedback from the client and marketplace to re-prioritize the features to tackle in the next iteration.

What is Risk-Driven and Client-Driven Iterative Planning?

The UP encourages a combination of risk-driven and client-driven iterative planning. This means that the goals of the early iterations are chosen to 1) identify and drive down the highest risks, and 2) build visible features that the client cares most about.

Risk-driven iterative development includes more specifically the practice of architecture-centric iterative development, i.e early iterations focus on building, testing, and stabilizing the core architecture

What are Agile Methods and Attitudes?

Agile development methods usually apply time boxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage agility rapid and flexible response to change.

The Agile Manifesto and Principles

The Agile Manifesto

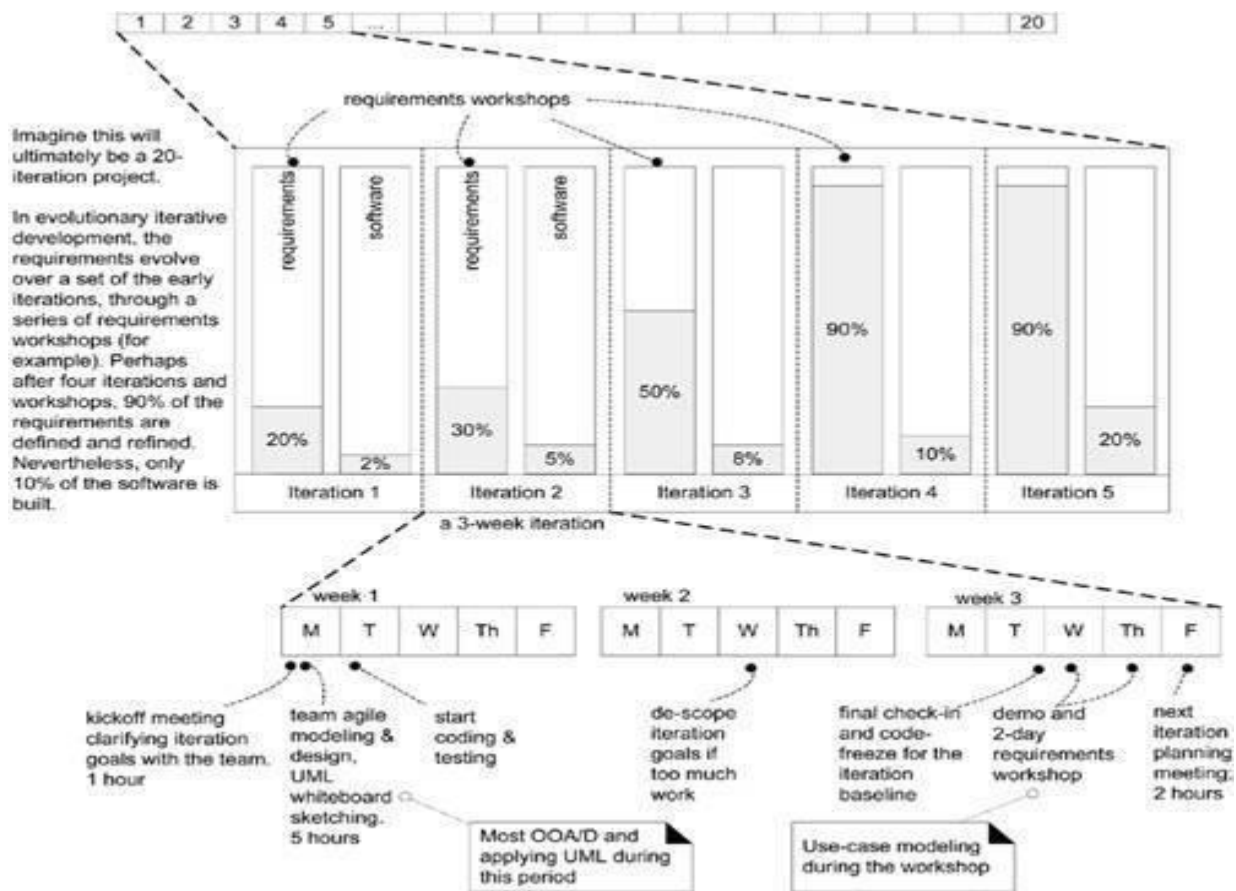
Individuals and interactions	over processes and tools
Working software	over comprehensive documentation
Customer collaboration	over contract negotiation
Responding to change	over following a plan

The Agile Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development.
9. The sponsors, developers, and users should be able to maintain a constant pace indefinitely
10. Continuous attention to technical excellence and good design enhances agility
11. Simplicity the art of maximizing the amount of work not done is essential
12. The best architectures, requirements, and designs emerge from self-organizing teams.

13. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

This example assumes there will ultimately be 20 iterations on the project before delivery:



Evolutionary analysis and design the majority in early iterations.

UP Phases

A UP project organizes the work and iterations across four major phases:

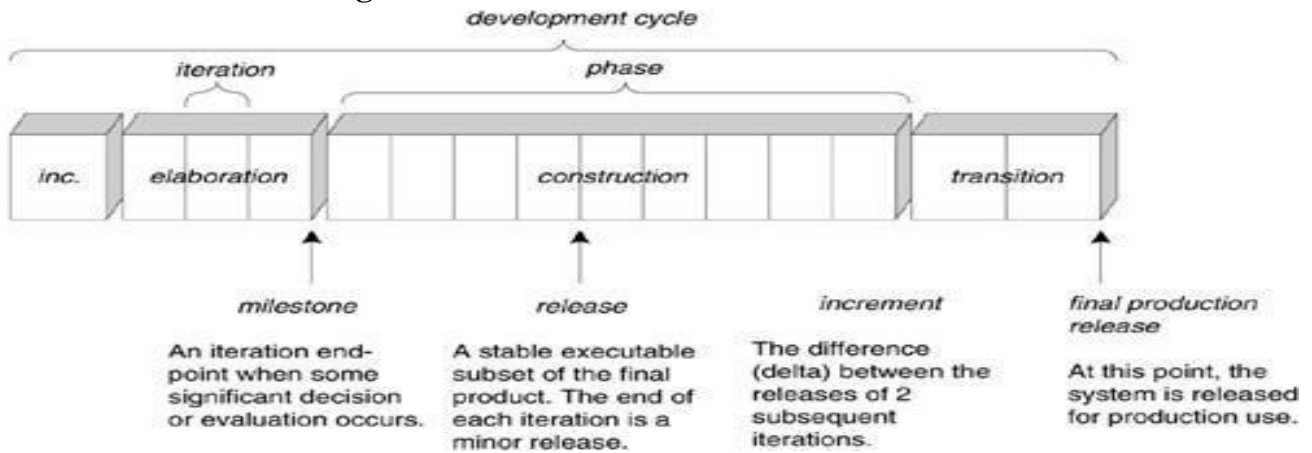
1. **Inception** - approximate vision, business case, scope, vague estimates.
2. **Elaboration** - refined vision, iterative implementation of the core architecture, resolution of high risks, identification of most requirements and scope, more realistic estimates.
3. **Construction** - iterative implementation of the remaining lower risk and easier elements, and preparation for deployment.
4. **Transition** - beta tests, deployment.

This is not the old "waterfall" or sequential lifecycle of first defining all the requirements, and then doing all or most of the design. Inception is not a requirements phase; rather, it is a feasibility phase, where investigation is done to support a decision to

continue or stop.

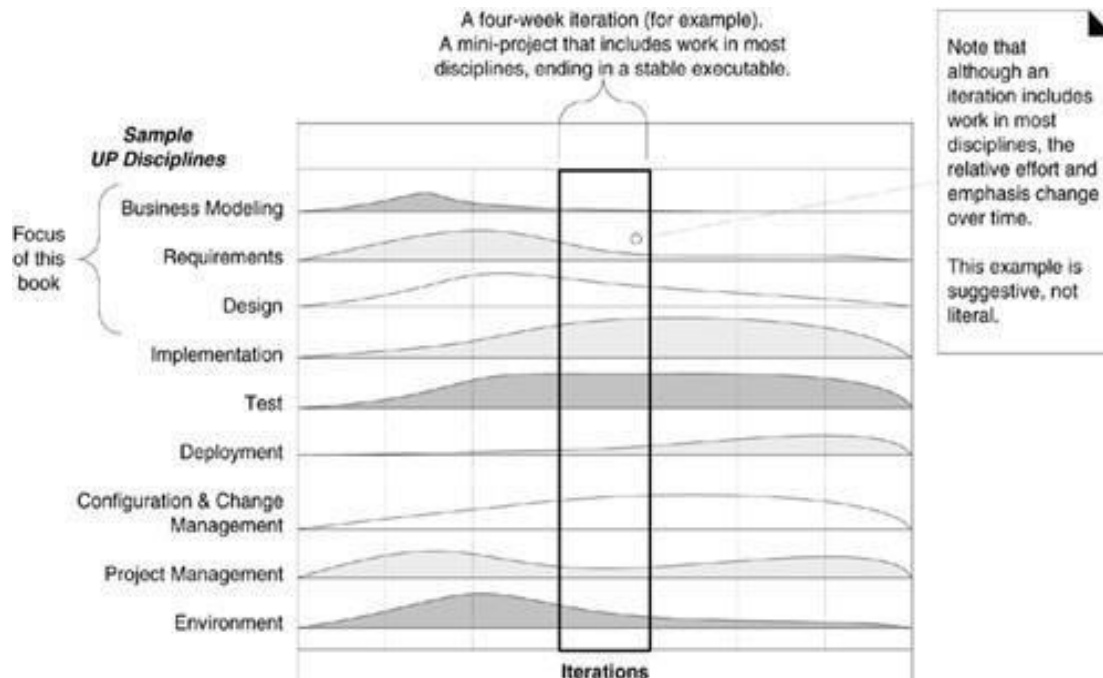
Similarly, elaboration is a phase where the core architecture is iteratively implemented, and high-risk issues are mitigated.

Figure - Schedule-oriented terms in the UP.



The UP Disciplines

Disciplines a set of activities (and related artifacts) in one subject area, such as the activities within requirements analysis. In the UP, an artifact is the general term for any work product: code, Web graphics, database schema, text documents, diagrams, models, and so on. There are several disciplines in the UP:

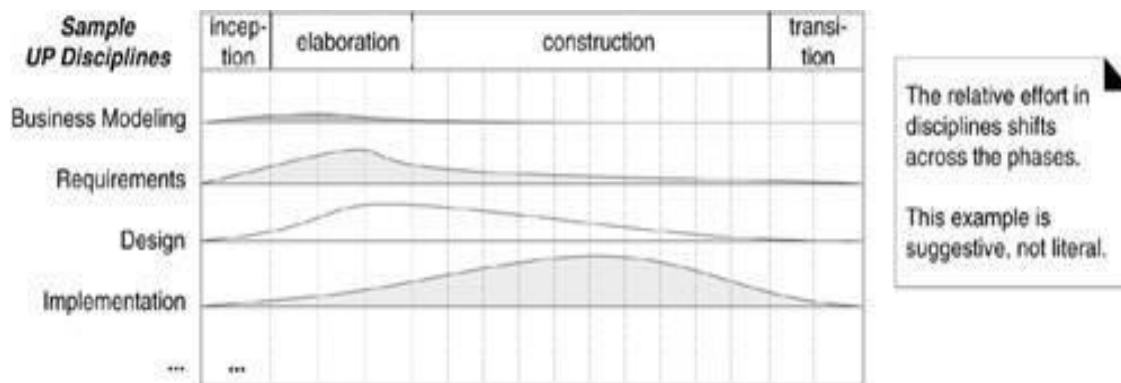


Business Modeling - The Domain Model artifact, to visualize noteworthy concepts in the application domain.

Requirements - The Use-Case Model and Supplementary Specification artifacts to capture functional and non-functional requirements.

Design - The Design Model artifact, to design the software objects

What is the Relationship Between the Disciplines and Phases?



Definition: the Development Case: The choice of practices and UP artifacts for a project may be written up in a short document called the Development Case (an artifact in the Environment discipline).

Discipline	Practice	Artifact	Incep.	Elab.	Const.	Trans.
			I1	E1..En	C1..Cn	T1..T2
		Iteration				
Business Modeling	agile modeling req. workshop	Domain Model		s		
Requirements	req. workshop vision box exercise dot voting	Use-Case Model	s	r		
		Vision	s	r		
		Supplementary Specification	s	r		
		Glossary	s	r		
Design	agile modeling test-driven dev.	Design Model		s	r	
		SW Architecture Document		s		
		Data Model		s	r	
Implementation	test-driven dev. pair programming continuous integration coding standards	...				
Project Management	agile PM daily Scrum meeting	...				
...						

Table- Sample Development Case. s - start; r - refine

CASE STUDY

Applications include UI elements, core application logic, database access, and collaboration with external software or hardware components. This introduction to OOA/D focuses on the core application logic layer, with some secondary discussion of the other layers.

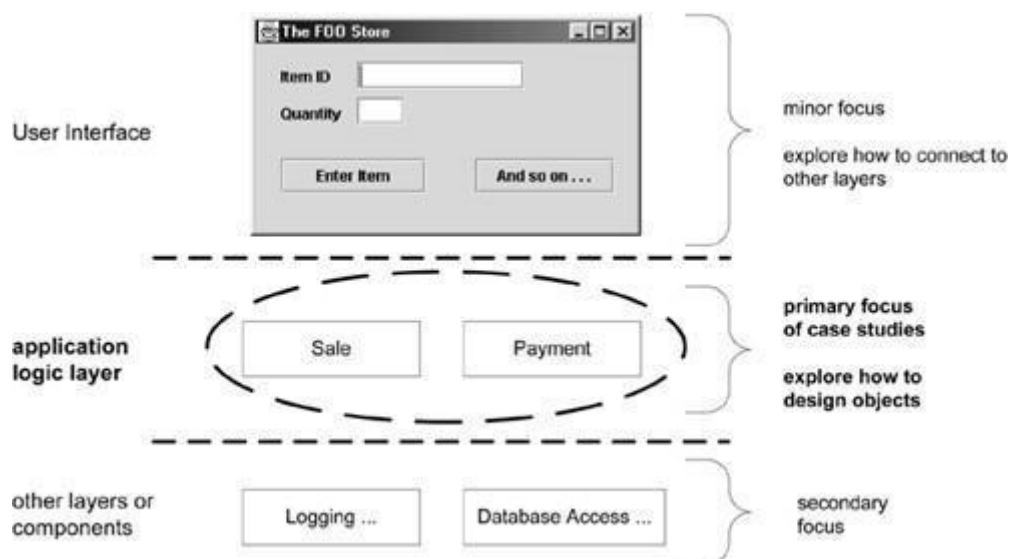
Why focus on OOA/D in the core application logic layer?

Other layers are usually technology/platform dependent. For example, to explore the OO design of a Web UI or rich client UI layer in Java, we would need to learn in detail about a framework such as Struts or Swing.

In contrast, the OO design of the core logic layer is similar across technologies.

The essential OO design skills learned in the context of the application logic layer are applicable to all other layers or components.

The design approach/patterns for the other layers tends to change quickly as new frameworks or technologies emerge.



Sample layers and objects in an object-oriented system, and the case study focus.

NEXTGEN POS SYSTEM

The NextGen POS(Point of Sale) System

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store.

It includes hardware components such as a computer and bar code scanner, and software to run the system.

It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if

remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).



A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and implementation.

INCEPTION

Inception: The purpose of the inception phase is not to define all the requirements, or generate a believable estimate or project plan. Most requirements analysis occurs during the elaboration phase, in parallel with early production-quality programming and testing.

Inception (in one sentence) - Envision the product scope, vision, and business case. It may include the first requirements workshop, planning for the first iteration, and then quickly moving forward to elaboration. Common inception artifacts and indicates the issues they address.

For example, the Use-Case Model may list the names of most of the expected use cases and actors, but perhaps only describe 10% of the use cases in detail done in the service of developing a rough high-level vision of the system scope, purpose, and risks.

Note that some programming work may occur in inception in order to create "proof of concept" prototypes, to clarify a few requirements via UI-oriented prototypes, and to do programming experiments for key "show stopper" technical questions.

Table -Sample inception artifacts.

Artifact	Comment
Vision and Business Case	Describes the high-level goals and constraints, the business case, and provides an executive summary.
Use-Case Model	Describes the functional requirements. During inception, the names of most use cases will be identified, and perhaps 10% of the use cases will be analyzed in detail.
Supplementary Specification	Describes other requirements, mostly non-functional. During inception, it is useful to have some idea of the key non-functional requirements that have will have a major impact on the architecture.
Glossary	Key domain terminology, and data dictionary.
Risk List & Risk Management Plan	Describes the risks (business, technical, resource, schedule) and ideas for their mitigation or response.
Prototypes and proof-of- concepts	To clarify the vision, and validate technical ideas.
Iteration Plan	Describes what to do in the first elaboration iteration.
Phase Plan & Software Development Plan	Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources.
Development Case	A description of the customized-UP steps and artifacts for this project. In the UP, one always customizes it for the project.

Activities in Inception:

Inception is a short step to elaboration. It determines basic feasibility, risk, and scope, to decide if the project is worth more serious investigation.

Activities and artifacts in inception include:

- a short requirement workshops
- most actors, goals, and use cases named
- most use cases written in brief format; 10-20% of the use cases are written in fully dressed detail to improve understanding of the scope and complexity
- most influential and risky quality requirements identified
- version one of the Vision and Supplementary Specification written
- risk list
- Technical proof-of-concept prototypes and other investigations to explore the technical feasibility of special requirements
- user interface-oriented prototypes to clarify the vision of functional requirements
- recommendations on what components to buy/build/reuse, to be refined in

elaboration

- o For example, a recommendation to buy a tax calculation package.

high-level candidate architecture and components proposed

- o This is not a detailed architectural description, and it is not meant to be final or correct. Rather, it is brief speculation to use as a starting point of investigation in elaboration. For example, "A Java client-side application, no application server, Oracle for the database," In elaboration, it may be proven worthy, or discovered to be a poor idea and rejected.

plan for the first iteration

USE CASES & USECASE MODELLING

Use cases are text stories, widely used to discover and record requirements. use cases are text stories of some actor using a system to meet goals. There are 3 formats to represent the use case

- I) Brief Format
- II) Casual Format
- III) Fully Dressed Format

Definition: What are Actors, Scenarios, and Use Cases

An **actor** is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

A **scenario** is a specific sequence of actions and interactions between actors and the system; it is also called a use case instance. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.

A **use case** is a collection of related success and failure scenarios that describe an actor using a system to support a goal.

Use Cases and the Use-Case Model

Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing diagrams.

There are also the Supplementary Specification, Glossary, Vision, and Business Rules. These are all useful for requirements analysis.

The Use-Case Model may optionally include a UML use case diagram to show the names of use cases and actors, and their relationships. This gives a nice context diagram of a system and its environment. It also provides a quick way to list the use cases by name.

Motivation: Why Use Cases?

Lack of user involvement in software projects is near the top of the list of reasons for project failure. Use cases are a good way to help keep it simple, and make it possible for domain experts or requirement donors to themselves write use cases.

Another value of use cases is that they emphasize the user goals and perspective; we ask the question "Who is using the system, what are their typical scenarios of use, and what are their goals?" This is a more user-centric emphasis compared to simply asking for a list of system features.

Definition: Are Use Cases Functional Requirements

Use cases are requirements, primarily functional or behavioral requirements that indicate what the system will do. A related viewpoint is that a use case defines a contract of how a system will behave

What are Three Kinds of Actors?

An actor is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems. **Primary and supporting** actors will appear in the action steps of the use case text. **Actors are roles played not only by people, but by organizations, software, and machines.** There are three kinds of external actors in relation to the SuD:

1. **Primary actor** has user goals fulfilled through using services of the SuD. For example, the cashier.
 - o Why identify? To find user goals, which drive the use cases.
2. **Supporting actor** provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
 - o Why identify? To clarify external interfaces and protocols.
3. **Offstage actor** has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
 - o Why identify? To ensure that all necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

Three Common Use Case Formats

Brief - Terse one-paragraph summary, usually of the main success scenario. The prior Process Sale example was brief. It was created during early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.

Example: Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items. Notice that use cases are not diagrams, they are text.

Casual - Informal paragraph format. Multiple paragraphs that cover various scenarios. It was created during early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.

Example: Handle Returns Use case

Main Success Scenario: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...

Alternate Scenarios: If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.

If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code . If the system detects failure to communicate with the external accounting system,

Fully Dressed - All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees. It was created , after many use cases have been identified and written in a brief format, then during the first requirements workshop a few (such as 10%) of the architecturally significant and high-value use cases are written in detail.

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, and worth telling the reader?
Success Guarantee	What must be true on successful completion, and worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.

Special Requirements	Related non-functional requirements.
----------------------	--------------------------------------

Use Case Section	Comment
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Use Case UC1: Process Sale: Fully Dressed Format Example

Scope: NextGen POS application

Level: user goal

Primary Actor: Cashier

Stakeholders and Interests:- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.

- **Salesperson:** Wants sales commissions updated.

- **Customer:** Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.

- **Company:** Wants to accurately record transactions and satisfy customer interests.

Wants to ensure that Payment Authorization Service payment receivables are recorded.

Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.

- **Manager:** Wants to be able to quickly perform override operations, and easily debug Cashier problems.

- **Government Tax Agencies:** Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.

- **Payment Authorization Service:** Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store. **Preconditions:** Cashier is identified and authenticated.

Success Guarantee (or Postconditions): Sale is saved. Tax is correctly calculated.

Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Payment authorization approvals are recorded.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Somehow, we want robust recovery when access to remote services such the inventory system is failing.
- Language internationalization on the text displayed
- Pluggable business rules to be insertable at steps 3 and 7.

Technology and Data Variations List:

*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.

3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard. 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

- What are the tax law variations?
- Explore the remote service recovery issue.
- What customization is needed for different businesses?
- Must a cashier take their cash drawer when they log out?
- Can the customer directly use the card reader, or does the cashier have to do it?

Format Description:

Scope: It can be either system use case or Business Use case.

system use case: A use case describes use of one software system

business use case: At a broader scope, use cases can also describe how a business is used by its customers and partners. Such an enterprise-level process description is called a business use case.

Level: Use cases are classified as at the user-goal level or the sub function level, among others.

A user-goal level use case is the common kind that describe the scenarios to fulfill the goals of a primary actor to get work done.

A subfunction-level use case describes sub steps required to support a user goal, and is usually created to factor out duplicate sub steps shared by several regular use cases an example is the subfunction use case Pay by Credit, which could be shared by many regular use cases.

Primary Actor: The principal actor that calls upon system services to fulfill a goal.

Stakeholders and Interests List It satisfies all the stakeholders' interests. by starting with the stakeholders and their interests before writing the remainder of the use case, we have a method to remind us what the more detailed responsibilities of the system should be.

Stakeholders and Interests:

Stakeholders and Interests:

- Cashier: Wants accurate, fast entry and no payment errors, as cash drawer shortages are deducted from his/her salary.
 - Salesperson: Wants sales commissions updated.
 - ...
-

Preconditions and Success Guarantees (Postconditions)

Preconditions state what must always be true before a scenario is begun in the use case. Preconditions are not tested within the use case; rather, they are conditions that are assumed to be true. Typically, a precondition implies a scenario of another use case, such as logging in, that has successfully completed.

Main Success Scenario and Steps (or Basic Flow) This has also been called the "happy path" scenario, or the more prosaic "Basic Flow" or "Typical Flow." It describes a typical success path that satisfies the interests of the stakeholders.

The scenario records the steps, of which there are three kinds:

1. An interaction between actors.
2. A validation (usually by the system).
3. A state change by the system (for example, recording or modifying something).

Main Success Scenario:

1. Customer arrives at a POS checkout with items to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. ...

Cashier repeats steps 3-4 until indicates done.

5. ...

Extensions (or Alternate Flows)

Extension scenarios are branches (both success and failure) from the main success scenario, and so can be notated with respect to its steps 1...N. For example, at Step 3 of the main success scenario there may be an invalid item identifier, either because it was incorrectly entered or unknown to the system. An extension is labeled "3a"; it first identifies the condition and then the response. Alternate extensions at Step 3 are labeled "3b" and so forth.

Extensions:

3a. Invalid identifier:

1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):

1. Cashier can enter item category identifier and the quantity.

An extension has two parts: the condition and the handling.

Guideline: When possible, write the condition as something that can be detected by the system or an actor.

This extension example also demonstrates the notation to express failures within extensions.

7b. Paying by credit:

- Customer enters their credit account information.
- System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.

2a. System detects failure to collaborate with external system:

- System signals error to Cashier.
- Cashier asks Customer for alternate payment.

Special Requirements

If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely.

Special Requirements:

- Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
- Credit authorization response within 30 seconds 90% of the time.
- Language internationalization on the text displayed.
- Pluggable business rules to be insertable at steps 2 and 6.

Technology and Data Variations List

A common example is a technical constraint imposed by a stakeholder regarding input or output technologies. For example, a stakeholder might say, "The POS system must

support credit account input using a card reader and the keyboard." It is also necessary to understand variations in data schemes, such as using UPCs or EANs for item identifiers, encoded in bar code symbology.

Technology and Data Variations List:

- 3a. Item identifier entered by laser scanner or keyboard.
- 3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- 7a. Credit account information entered by card reader or keyboard.
- 7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Guidelines For Use Case Modeling:

Guideline 1. Write in an Essential UI-Free Style

Guideline : Write use cases in an essential style; keep the user interface out and focus on actor intent.

Essential Style

Assume that the Manage Users use case requires identification and authentication:

- Administrator identifies self.
- System authenticates identity. 3. ...

The design solution to these intentions and responsibilities is wide open: biometric readers, graphical user interfaces (GUIs), and so forth.

Concrete Style Avoid During Early Requirements Work

In contrast, there is a concrete use case style. In this style, user interface decisions are embedded in the use case text. The text may even show window screen shots, discuss window navigation, GUI widget manipulation and so forth. For example:

- Administrator enters ID and password in dialog box
- System authenticates Administrator.
- System displays the "edit users" window 4. ...

Guideline 2. Write Terse Use Cases : Delete "noise" words. Even small changes add up, such as "System authenticates..." rather than "The System authenticates..."

Guideline 3 : Write Black-Box Use Cases : Black-box use cases are the most common and recommended kind; they do not describe the internal workings of the system, its components, or design. Rather, the system is described as having responsibilities, which is a common unifying metaphorical theme in object-oriented thinking software elements have responsibilities and collaborate with other elements that have responsibilities.

Black-box style	Not Recommended
The system records the sale.	The system writes the sale to a database. ...or (even worse): The system generates a SQL INSERT statement for the sale...

Guideline 4 : Take an Actor and Actor-Goal Perspective : Here's the RUP use case definition, from the use case founder Ivar Jacobson:

A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor. It stresses two attitudes during requirements analysis:

Write requirements focusing on the users or actors of a system, asking about their goals and typical situations.

Focus on understanding what the actor considers a valuable result.

Guideline 5: To Find Use Cases

Use cases are defined to satisfy the goals of the primary actors. Hence, the basic procedure is:

1.	Choose the system boundary. Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
2.	Identify the primary actors those that have goals fulfilled through using services of the system.
3.	Identify the goals for each primary actor.
4.	Define use cases that satisfy user goals; name them according to their goal. Usually, user-goal level use cases will be one-to-one with user goals, but there is at least one exception, as will be examined.

Step 1: Choose the System Boundary

The POS system itself is the system under design; everything outside of it is outside the system boundary, including the cashier, payment authorization service, and so on. For example, is the complete responsibility for payment authorization within the system boundary? No, there is an external payment authorization service actor.

Steps 2 and 3: Find Primary Actors and Goals

Guideline: Identify the primary actors first, as this sets up the framework for further investigation. The following questions help to identify other actors.

Who starts and stops the system?	Who does system administration?
Who does user and security management?	Is "time" an actor because the system does something in response to a time event?
Is there a monitoring process that restarts the system if it fails?	Who evaluates system activity or performance?
How are software updates handled? Push or pull update?	Who evaluates logs? Are they remotely retrieved?
In addition to human primary actors, are there any external software or robotic systems that call upon services of the system?	Who gets notified when there are errors or failures?

Representing Goals of an Actor:

There are at least two approaches:

1. Draw them in a use case diagram, naming the goals as use cases.
2. Write an actor-goal list first, review and refine it, and then draw the use case diagram.

For example:

Actor	Goal	Actor	Goal
Cashier	process sales process rentals handle returns cash in cash out...	System Administrator	add users modify users delete users manage security manage system tables...
Manager	start up shut down...	Sales Activity System	analyze sales and performance data
...	...	...	...

Is the Cashier or Customer the Primary Actor?

The answer depends on the system boundary of the system under design, and who we are primarily designing the system for the viewpoint of the POS system the system services the goal of a trained cashier (and the store) to process the customer's sale.

Primary actors and goals at different system boundaries.



The customer is an actor, but in the context of the NextGen POS, not a primary actor; rather, the cashier is the primary actor because the system is being designed to primarily serve the trained cashier's "power user" goals. The system does not have a UI and functionality that could equally be used by the customer or cashier. Rather, it is optimized to meet the needs and training of a cashier.

Second Method to Find Actors and Goals - Event Analysis

Another approach to aid in finding actors, goals, and use cases is to identify external events. What are they, where from, and why? Often, a group of events belong to the same use case. For example:

External Event	From Actor	Goal/Use Case
enter sale line item	Cashier	process a sale
enter payment	Cashier or Customer	process a sale
...		

Step 4: Define Use Cases

In general, define one use case for each user goal. Name the use case similar to the user goal for example,

Goal: process a sale; Use Case: Process Sale.

Start the name of use cases with a verb. A common exception to one use case per goal is to collapse CRUD (create, retrieve, update, delete) separate goals into one CRUD use case, idiomatically called Manage <X>. For example, the goals "edit user," "delete user," and so forth are all satisfied by the Manage Users use case.

Guideline 6: Tests To Find Useful Use Cases

There are several rules of thumb, including:

The Boss Test

The EBP Test

The Size Test

The Boss Test: To check for achieving results of measurable value

Your boss asks, "What have you been doing all day?" You reply: "Logging in!" Is your boss happy? If not, the use case fails the Boss Test, which implies it is not strongly related to achieving results of measurable value. It may be a use case at some low goal level, but not the desirable level of focus for requirements analysis.

The EBP Test

An Elementary Business Process (EBP) is a term from the business process engineering field, defined as:

EBP is similar to the term user task in usability engineering, although the meaning is less strict in that domain. Focus on use cases that reflect EBPs.

A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state, e.g., Approve Credit or Price Order

The EBP Test is similar to the Boss Test, especially in terms of the measurable business value qualification.

The Size Test

A use case typically contains many steps, and in the fully dressed format will often require 3- 10 pages of text. A common mistake in use case modeling is to define just a single step within a series of related steps as a use case by itself, such as defining a use case called Enter an Item ID. You can see a hint of the error by its small size the use case name will wrongly suggest just one step within a larger series of steps, and if you imagine the length of its fully dressed text, it would be extremely large.

Example: Applying the Tests

Negotiate a Supplier Contract

- o Much broader and longer than an EBP. Could be modeled as a business use case, rather than a system use case.

Handle Returns

- o OK with the boss. Seems like an EBP. Size is good.

Log In

- o Boss not happy if this is all you do all day!

Move Piece on Game Board

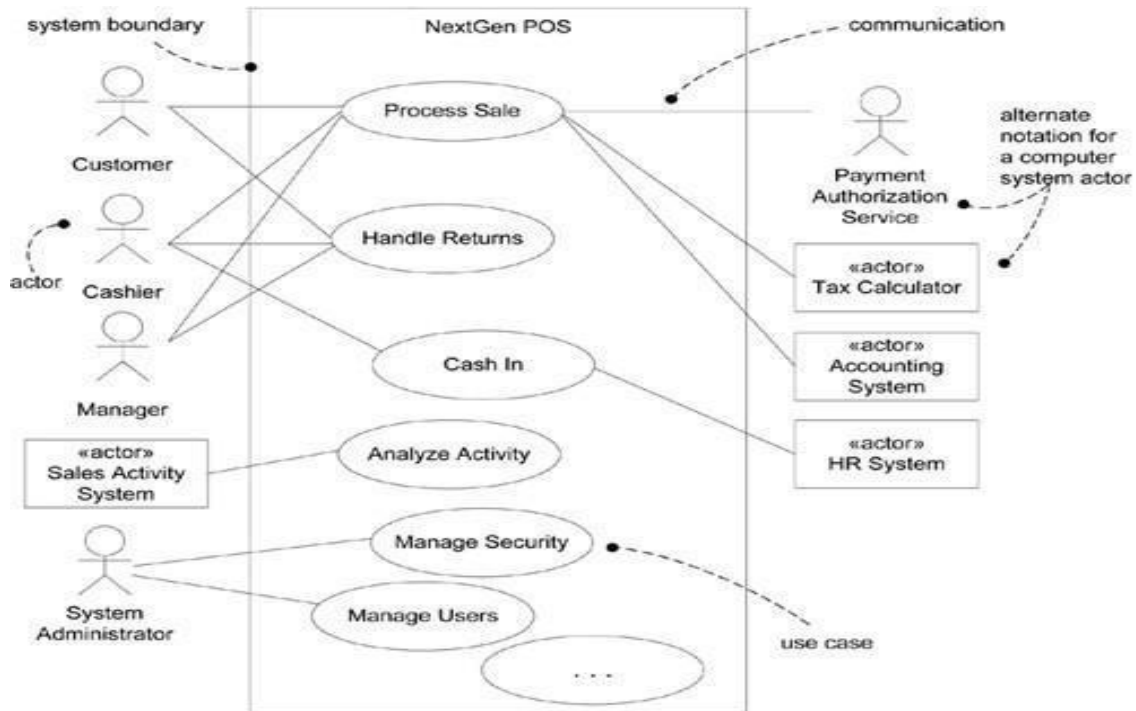
- o Single step fails the size test.

Applying UML: Use Case Diagrams

The UML provides use case diagram notation to illustrate the names of use cases and actors, and the relationships between them

Use case diagrams and use case relationships are secondary in use case work. Use cases are text documents. Doing use case work means to write text.

Partial use case Diagram - context diagram.



Guideline

Use case diagram is an excellent picture of the system context

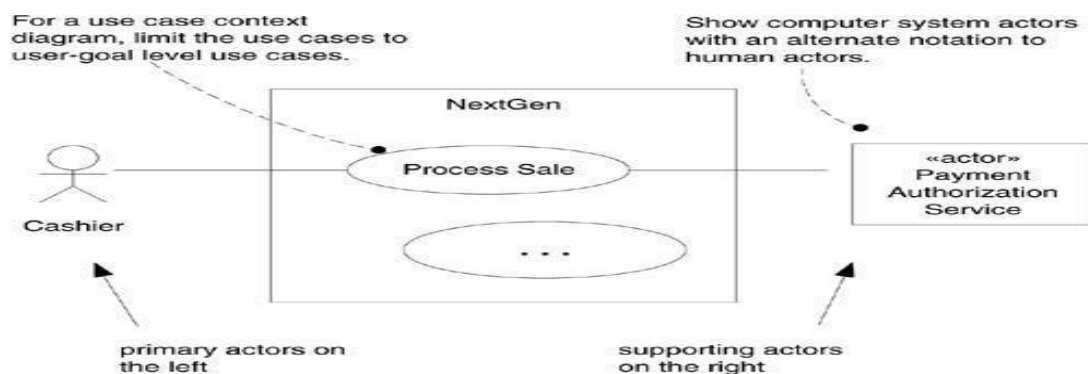
It makes a good context diagram that is, showing the boundary of a system, what lies outside of it, and how it gets used.

It serves as a communication tool that summarizes the behavior of a system and its actors.

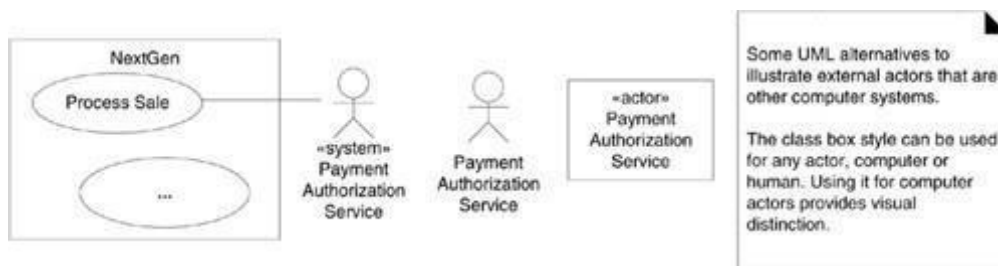
Guideline: Diagramming

Notice the actor box with the symbol «actor». This style is used for UML keywords and stereotypes, and includes guillemet symbols special single-character brackets («actor», not <<actor>>)

Notation suggestions.



Alternate actor notation.



RELATING USE CASES

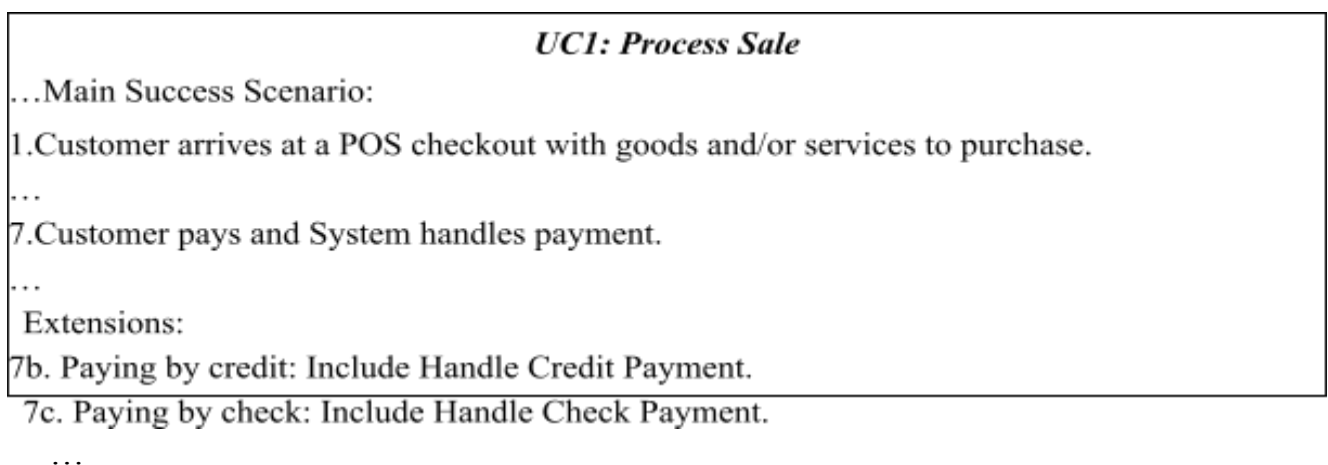
Use cases can be related to each other. For example, a sub function use case such as Handle Credit Payment may be part of several regular use cases, such as Process Sale and Process Rental. It is simply an organization mechanism to (ideally) improve communication and comprehension of the use cases, reduce duplication of text, and improve management of the use case documents.

Kinds of relationships

1. The include Relationship
2. The extend Relationship
3. The generalize Relationship

1. The include Relationship

It is common to have some partial behavior that is common across several use cases. For example, the description of paying by credit occurs in several use cases, including Process Sale, Process Rental, Contribute to Lay-away Plan.. Rather than duplicate this text, it is desirable to separate it into its own subfunction use case, and indicate its inclusion.



UC7: Process Rental

...

Extensions:

6b. Paying by credit: Include Handle Credit Payment.

...

This is the include relationship. A slightly shorter (and thus perhaps preferred) notation to indicate an included use case is simply to underline it or highlight it in some fashion. For example:

UC1: Process Sale

...

Extensions:

7b. Paying by credit: Handle Credit Payment.

7c. Paying by check: Handle Check Payment.

...

Notice that the Handle Credit Payment subfunction use case was originally in the Extensions section of the Process Sale use case, but was factored out to avoid duplication.

Another use of the include relationship is to describe the handling of an asynchronous event, such as when a user is able to, at any time, select or branch to a particular window, function, or Web page, or within a range of steps.

The basic notation is to use the a*, b*, ... style labels in the Extensions section

UC1: Process Foo Bars

...

Main Success

Scenario: 1. ...

Extensions:

a*. At any time, Customer selects to edit personal information: Edit Personal Information. b*. At any time, Customer selects printing help: Present Printing Help. 2-11. Customer cancels: Cancel Transaction Confirmation.

Use cases and use the include relationship when:

- They are duplicated in other use cases.

- A use case is very complex and long, and separating it into subunits aids comprehension.

Concrete, Abstract, Base, and Addition Use Cases

A **concrete use case** is initiated by an actor and performs the entire behavior desired by the actor. These are the elementary business process use cases. For example, Process Sale is a concrete use case.

An **abstract use case** is never instantiated by itself; it is a subfunction use case that is part of another use case. Handle Credit Payment is abstract; it doesn't stand on its own, but is always part of another story, such as Process Sale.

A use case that includes another use case, or that is extended or specialized by another use case is called **a base use case**. Process Sale is a base use case with respect to the included Handle Credit Payment subfunction use case. On the other hand, the use case that is an inclusion, extension, or specialization is called an addition use case. Handle Credit Payment is the addition use case in the include relationship to Process Sale. Addition use cases are usually abstract. Base use cases are usually concrete.

2. The extend Relationship

The idea is to create an extending or addition use case, and within it, describe where and under what condition it extends the behavior of some base use case. For example:

UC1: Process Sale (the base use case)

...

Extension Points: VIP Customer, step 1. Payment, step 7.

Main Success Scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.

...

7. Customer pays and System handles payment.

...

UC15: Handle Gift Certificate Payment (the extending use case)

...Trigger: Customer wants to pay with gift certificate.

Extension Points: Payment in Process Sale.

Level: Subfunction

Main Success

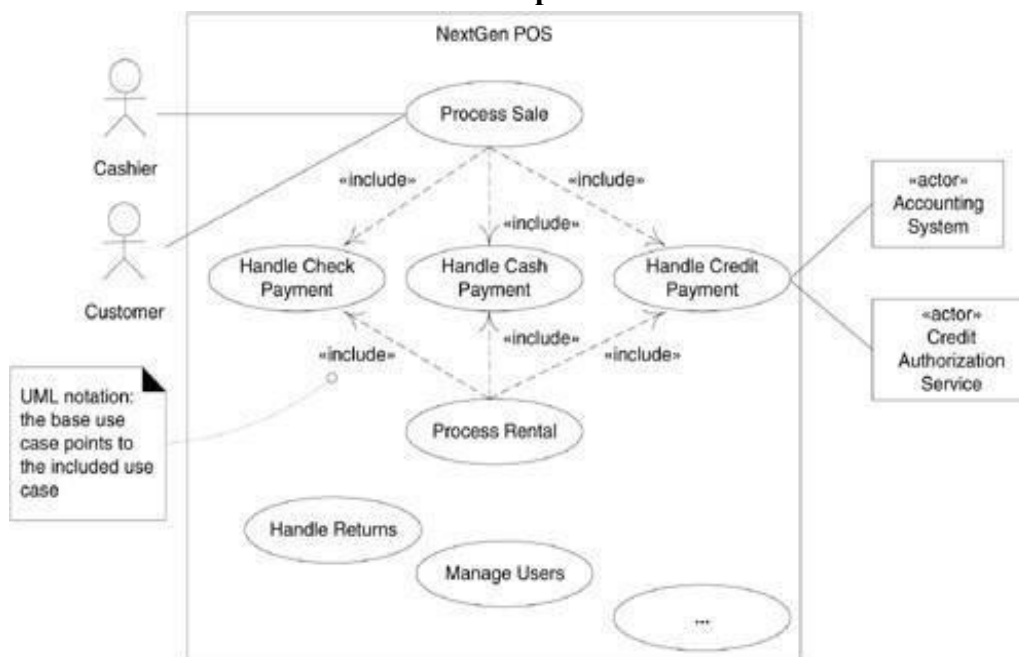
Scenario:

1. Customer gives gift certificate to Cashier.
2. Cashier enters gift certificate ID.

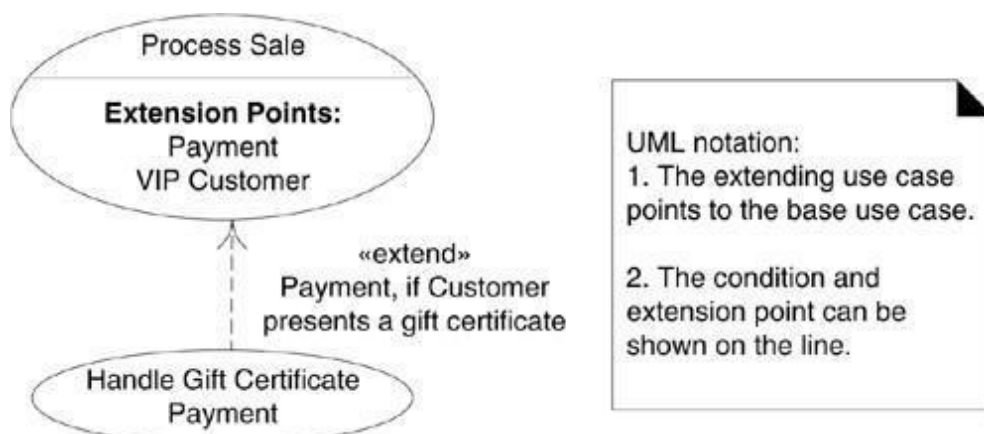
...

This is an example of an extend relationship. The use of an extension point, and that the extending use case is triggered by some condition. Extension points are labels in the base use case which the extending use case references as the point of extension, so that the step numbering of the base use case can change without affecting the extending use case.

Use case include relationship in the Use-Case Model.



The extend relationship.



3. The Generalization Relationship

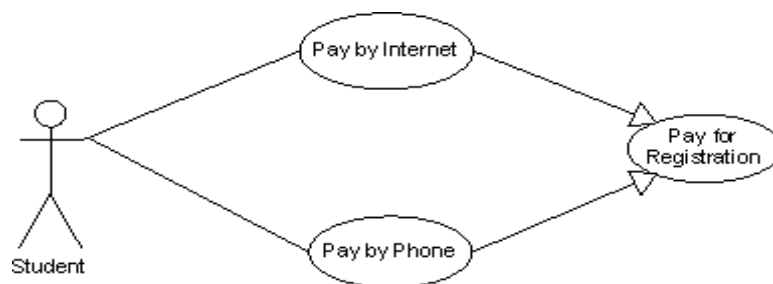
In the context of use case modeling the use case generalization refers to the relationship which can exist between two use cases and which shows that one use case (child) inherits the structure, behavior, and relationships of another actor (parent). The child use case is also referred to the more specialized use case while the parent is also referred to as the more abstract use case of the relationship.

For those of you familiar with object-oriented concepts: use cases in UML are classes and the generalization is simply the inheritance relationship between two use cases by which one use case inherits all the properties and relationships of another use case.

You can use the generalization relationship when you find two or more use cases which have common behavior/logic. In this instance, you can describe the common parts in a separate use case (the parent) which then is specialized into two or more specialized child use cases.

Example:

If you are creating a payment system which allows students of a training provider to pay for courses both on-line and by phone, there will many things in common between the two scenarios: specifying personal info, specifying payment info, etc. However, there would also be differences between the two. So, the best way to accomplish this is to create one use case (the parent) which contains the common behavior and then create two specialized child use cases which inherit from the parent and which contain the differences specific to registering on-line vs. by phone.



WHEN TO USE USECASE DIAGRAM

When to Use USECASE DIAGRAM

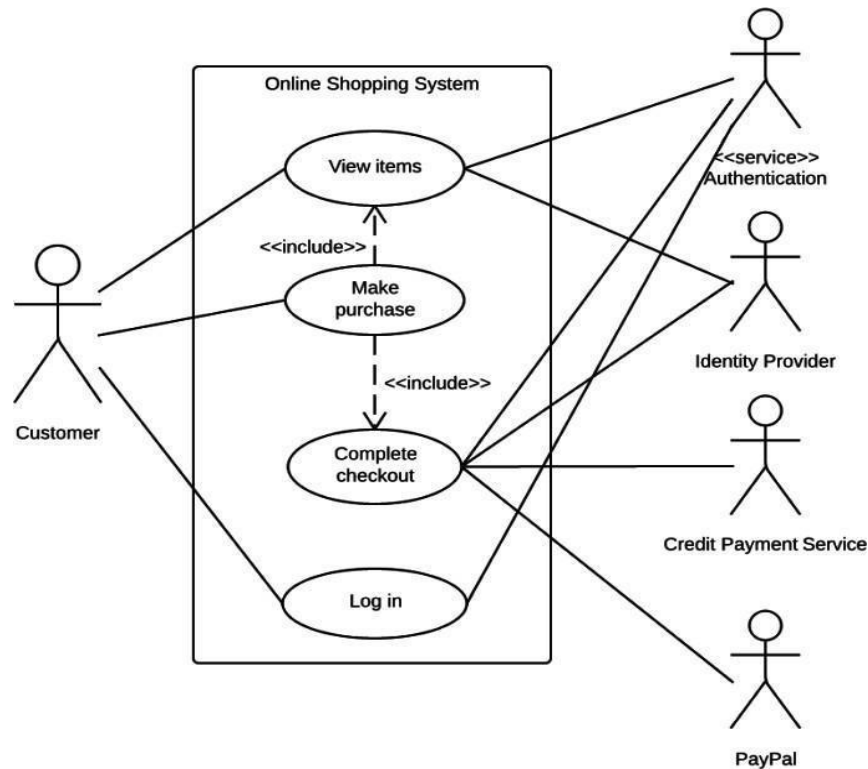
UML use case diagrams are ideal for:

- Representing the goals of system-user interactions
- Defining and organizing functional requirements in a system

- Specifying the context and requirements of a system
- Modeling the basic flow of events in a use case

- Reverse engineering.
- Forward engineering.

Ex: Online shopping System



Use case diagrams are valuable for visualizing the functional requirements of a system that will translate into design choices and development priorities. They also help identify any internal or external factors that may influence the system and should be taken into consideration.

Use case diagrams specify the events of a system and their flows. But use case diagram never describes how they are implemented. Use case diagram can be imagined as a black box where only the input, output, and the function of the black box is known. Although use case is not a good candidate for forward and reverse engineering, still they are used in a slightly different way to make forward and reverse engineering. The same is true for reverse engineering. Use case diagram is used differently to make it suitable for reverse engineering. In forward engineering, use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.