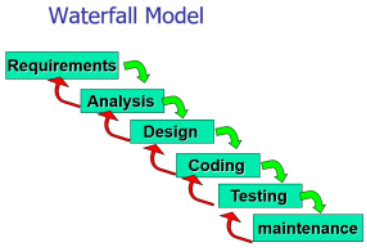
Introduction

Provided by: Hanyuu

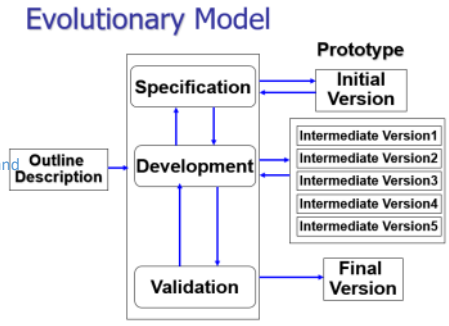
Software Process Model

- Waterfall Model
- Evolutionary model
- Spiral model
- Iterative and Incremental Process Model
- UP Model

- Waterfall Model
  - (→)
  - Requirement Analysis Design Coding Testing maintenance
  - (←)

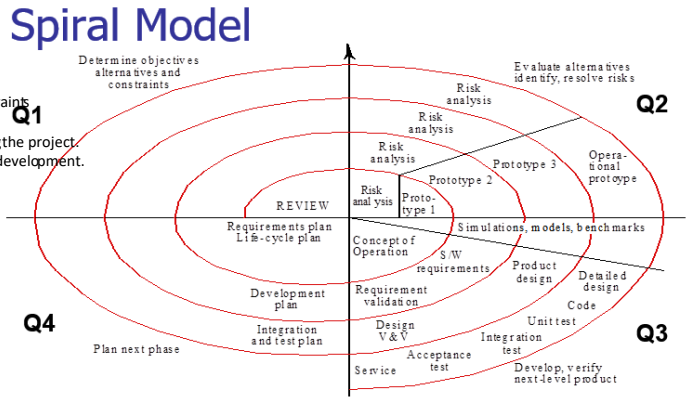


- About waterfall model:
  - The different stages are to be carried out consecutively
  - At each stage, the work done will be documented, which will form the basis for the work that is done at the next stage
  - After deployment, the system will be maintained as long as it remains in service
- Advantages：
  - It is an intuitive, sensible and general-purpose engineering approach to software engineering.
  - It emphasizes planning for an activity before carrying it out, better than just start coding without considering overall architecture of the program.
  - It recommends a top-down approach, from high level of abstraction to more details.
  - It provides a strong management tool for controlling software projects.
- Disadvantages：
  - If errors are found when working on one stage, it is reasonable to correct the problems before proceeding. However, Some earlier versions of the waterfall model restricted feedback to the previous stage in the process only.
- Evolutionary Model
  - Evolutionary model suggests that software should be developed in a more 'evolutionary' manner.
- How evolutionary model works?
  - Development starts with the production of a prototype system implementing only the core functionality of the complete system.
  - The prototype will be used to discuss with the users
  - Feedback from the users would then guide subsequent development, …
  - The prototype would evolve into a more complete system, with small increments of functionality being added at any given stage and repeated consultations with the users taking place.



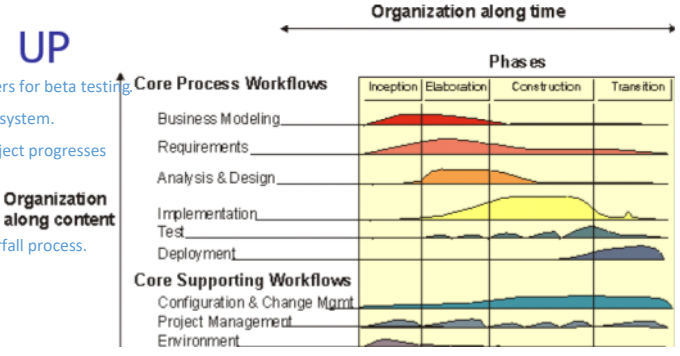
- Advantages:
  - By involving users in the development process, the problem of discovering errors late has been overcome.
  - A series of working prototypes mean that the evolving system was being tested from an early stage in a project
- Disadvantages:
  - It is very difficult to see how project managers could sensibly plan a project.
  - The model gives no guarantee that the evolutionary process will in fact ever converge on a stable system.
- Spiral Model
- The spiral model was developed in an attempt
  - to develop an explicit software process that addressed the weakness of the waterfall model
  - to reserve more of its traditional management oriented advantages than the evolutionary model

Q1: consider the objectives for the iteration, the various alternative solutions that should be considered and the constraints under which a solution must be developed.
Q2: a risk analysis is carried out. The aim of this is to ensure that each iteration focuses on the highest risks threatening the project.
Q3: construct prototypes to help evaluate possible approaches to resolving the risks, before committing to a detailed development.
Q4:a review of the work done in the iteration and plans are made for the subsequent iteration.

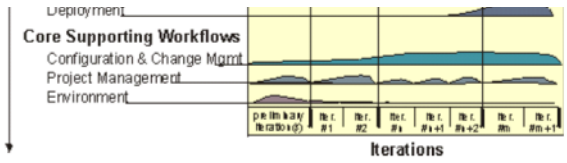


- Incremental development :
- develop core functionality first, to the level of getting a working system, and then
- add the remaining functionality incrementally, in a series of increments
- Iterative development:
- Software projects be managed as a series of iterations;
- Not just as a single pass through the various activities identified in the waterfall model

- UP Model
- The inception phase is primarily concerned with project planning and risk evaluation.
- The elaboration phase is concerned with defining the overall architecture of the system.
- In the construction phase, the system is built, ending with a release that can be delivered to the customers for beta testing.
- The transition phase covers the period of beta testing and terminates with the release of the completed system.
- Each activity can occur in any iteration, but the balance of activities in an iteration will change as the project progresses from inception to completion.
- Each phase terminates with a milestone to capture the points within the project lifecycle
- Each phase can contain a number of iterations; Each iteration will typically be structured as a mini-waterfall process.
- An iteration should result in an incremental improvement to the product being constructed.
- UP is:
  - Use case (requirements) driven



- An iteration should result in an incremental improvement to the product being constructed.
- UP is:
  - Use case (requirements) driven
  - Architecture centric
  - Iterative and incremental

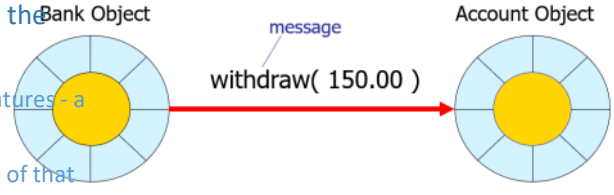


Object-Oriented Technology

Provided by Hanyuu

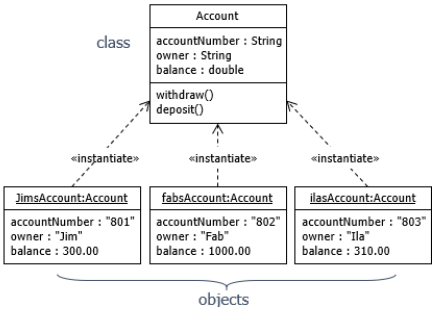
- Bases on object concept
- Uses class and inheritance as organize mechanism
- Utilizes polymorphism to provide flexibility
- Principles（四种主要原则和三种次要原则）
- Abstraction 抽象
- Encapsulation 封装
- Modularity 模块化
- Hierarchy 层次
- Typing 类型
- Concurrency 并发
- Persistence 持久

  综合考虑，相互协作
- Advantages
- Promotes abstract level
- Simplifies software development process
- Improves software’s structure
- Supports software reusing
- Concepts
- Object:Objects consist of data and function packaged together in a reusable unit.
  - Every object is an instance of some class which defines the common set of features (attributes and operations) shared by all of its instances. Objects have:
    - Attribute values – the data part
    - Operations – the behaviour part
  - All objects have:
    - Identity: Each object has its own unique identity and can be accessed by a unique handle
    - State: This is the actual data values stored in an object at any point in time
    - Behaviour: The set of operations that an object can perform
  - Encapsulation 封装
    - Data is hidden inside the object. The only way to access the data is via one of the operations
    - This is encapsulation or data hiding and it is a very powerful idea. It leads to more robust software and reusable code.
- Class:Every object is an instance of one class - the class describes the "type" of the object.
  - Classes allow us to model sets of objects that have the same set of features - a class acts as a template for objects:
    - The class determines the structure (set of features) of all objects of that class

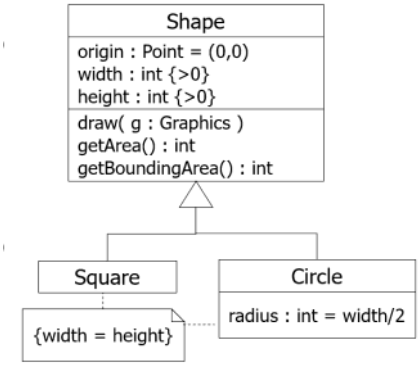


- All objects of a class must have the same set of operations, must have the same attributes, but may have different attribute values

- Message
  - In OO systems, objects send messages to each other over links
  - These messages cause an object to invoke an operation
- Inheritance
  - Inheritance implies a generalization/specialization hierarchy, wherein a subclass specializes the more general structure or behavior of its superclasses.
  - As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses.
  - Subclasses inherit all features of their superclasses:
    - attributes
    - operations
    - relationships
    - stereotypes, tags, constraints
  - Subclasses can add new features
  - Subclasses can override superclass operations
  - We can use a subclass instance anywhere a superclass instance is expected
  - Overriding
  - Multiple inheritance

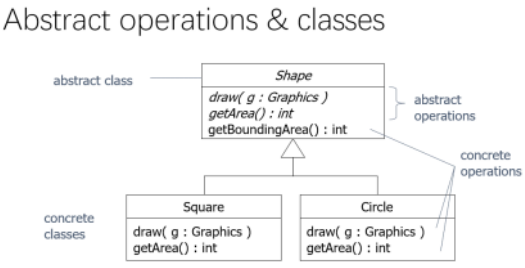


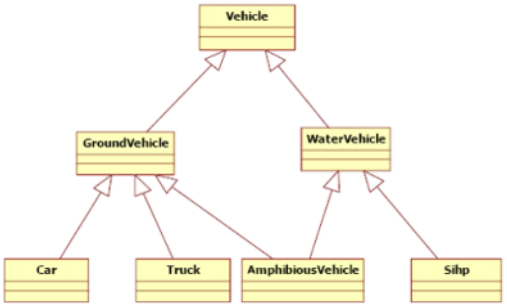
- Polymorphism
  - Polymorphism allows instances of different classes to respond to the same message in different ways.
- Generalization
  - A relationship between a more general element and a more specific element
  - The more specific element is entirely consistent with the more general element but contains more information
  - An instance of the more specific element may be used where an instance of the more general element is expected



【Specialisation&generalization】"is a kind of"

- Class inhertance
  - Semantically, inheritance denotes an “is a” relationship. For example
    - a bear “is a” kind of mammal
  - Inheritance thus implies a generalization/specialization hierarchy, wherein a subclass specializes the more general structure or behavior of its superclasses.
    - Indeed, this is the litmus test for inheritance: If B is not a kind of A, then B should not inherit from A.
  - As we evolve our inheritance hierarchy, the structure and behavior that are common for different classes will tend to migrate to common superclasses.
  - Subclasses inherit all features of their superclasses:
    - attributes
    - operations
    - relationships
    - stereotypes, tags, constraints
  - Subclasses can add new features
  - Subclasses can override superclass operations
  - We can use a subclass instance anywhere a superclass instance is expected
  - Subclasses often need to override superclass behaviour
  - To override a superclass operation, a subclass must provide an operation with the same signature
    The operation signature is the operation name, return type and types of all the parameters





- Abstract
  - Operations that lack an implementation are abstract operations
  - A class with any abstract operations can’t be instantiated and is therefore an abstract class
- Multiple inheritance

    - Sometimes a class may have more than one superclass
    - The "is kind of" and substitutability principles must apply for all of the classifications
    - Multiple inheritance is sometimes the most elegant way of modelling something.
- Polymorphism
    - Polymorphism = "many forms"
        - A polymorphic operation has many implementations
        - Square and Circle provide implementations for the polymorphic operations Shape::draw() and Shape::getArea()
    - All concrete subclasses of Shape must provide concrete draw() and getArea() operations because they are abstract in the superclass
        - For draw() and getArea() we can treat all subclasses of Shape in a similar way - we have defined a contract for Shape subclasses

- Engineering
- OOA
    - Produce an Analysis Model of the system's desired behaviour:
        - This model should be a statement of what the system does not how it does it
        - We can think of the analysis model as a "first-cut" or "high level" design model
        - It is in the language of the business
    - The analysis model:
        - is always in the language of the business;
        - captures the big picture;
        - contains artifacts that model the problem domain;
        - tells a story about the desired system;
        - is useful to as many of the stakeholders as possible.
- OOD
    - Decide how the system's functions are to be implemented
    - Decide on strategic design issues such as persistence, distribution etc.
    - Create policies to deal with tactical design issues
    - Open/Closed Principle, OCP 开闭原则

      Software entities should be open for extension,but closed for modification
      软件实体应当对扩展开放，对修改关闭
    - Liskov Substitution Principle, LSP 里氏代换原则

      如果每一个类型为T1的对象o1，都有类型为T2的对象o2，使得以T1定义的所有程序P在所有的对象o1都代换为o2时,程序P的行为没有变化，那么类型T2是类型T1的子类型。
    - Dependency Inversion Principle, DSP 依赖倒置原则

      A.高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。
      B.抽象不应该依赖于具体实现，具体实现应该依赖于抽象。
    - Interface Segregation Principle, ISP 接口分离原则

      采用多个与特定客户类有关的接口比采用一个通用的接口要好
- OOD - GRASP(General Responsibility Assignment Software Patterns):
    - Information Expert 信息专家

      如果某个类拥有完成某个职责所需要的所有信息，那么这个职责

就应该分配给这个类来实现。

- Creator 创造者

  实际应用中，符合下列任一条件的时候，都应该由类A来创建类B，这时A是B的创建者：

  a. A是B的聚合

  b. A是B的容器

  c. A持有初始化B的信息(数据)

  d. A记录B的实例

  e. A频繁使用B

  如果一个类创建了另一个类，那么这两个类之间就有了耦合，也可以说产生了依赖关系。依赖或耦合本身是没有错误的，但是它们带来的问题就是在以后的维护中会产生连锁反应，而必要的耦合是逃不掉的，我们能做的就是正确地创建耦合关系，不要随便建立类之间的依赖关系，那么该如何去做呢？就是要遵守创建者模式规定的基本原则，凡是不符合以上条件的情况，都不能随便用A创建B。

- Low Coupling 低耦合

  尽可能地减少类之间的连接

- High Cohesion 高内聚

  给类尽量分配内聚的职责

- Controller 控制器

  用来接收和处理系统事件的职责，一般应该分配给一个能够代表整个系统的类

  a. 系统事件的接收与处理通常由一个高级类来代替。

  b. 一个子系统会有很多控制器类，分别处理不同的事务。

- Polymorphism 多态

- Indirection 间接

- Pure Fabrication 纯虚构

  由一个纯虚构的类来协调内聚和耦合

- Protected Variations 受保护变化

  开闭原则要求

- OOD - GoF Patterns

  - Creational patterns

    1. Abstract Factory
    2. Builder
    3. Factory Method
    4. Prototype
    5. Singleton

  - Structural patterns

    1. Adapter
    2. Bridge
    3. Composite
    4. Decorator
    5. Facade
    6. Flyweight

7. Proxy

- Behavioral patterns
    1. Chain of Responsibility
    2. Command
    3. Interpreter
    4. Iterator
    5. Mediator
    6. Memento
    7. Observer
    8. State
    9. Strategy
    10. Template Method
    11. Visitor

- OOP

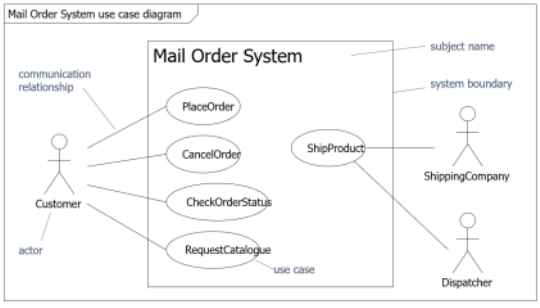
Use case modeling

Powered by: Hanyuu

- Use case modelling is a form of requirements engineering.
- Use cases are a technique for capturing the functional requirements of a system.
- Use case models :
  - Describe how users use the system;
  - Describe how actors interact with the system;
  - Capture the system requirements from the perspective of users;
  - Are easy to understand;
  - …
- A Use case Diagram:
  - Describes actors, use cases and relationships between them.
  - Describes who or what uses which function of a system.
- A use case specification:
  - Describe the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used.
  - Brings together all of the behavior relevant to a slice of system functionality.

  - The use case diagram shows the system boundary and the interactions with the outside world.
  - The use case diagram shows the actors, the use cases, and the relationships between them.
- Subject – the edge of the system
  - also known as the system boundary
- Actors – who or what uses the system
  - An actor is anything that interacts directly with the system
    - Actors identify who or what uses the system and so indicate where the system boundary lies
  - Actors are external to the system
  - An Actor specifies a role that some external entity adopts when interacting with the system
- Use Cases – things actors do with the system
  - A use case is something an actor needs the system to do. It is a “case of use” of the system by a specific actor
  - Use cases are always started by an actor
    - The primary actor triggers the use case
    - Zero or more secondary actors interact with the use case in some way
  - Use cases are always written from the point of view of the actors
  - Naming
    - Use cases describe something that happens
    - They are named using verbs or verb phrases

The use case diagram



- Relationships - between actors and use cases
  - Extra information:
    - Actor generalization
    - Use case generalization
    - «include» – between use cases
    - «extend» – between use cases
- Actor generalization

  If two actors communicate with the same set of use cases in the same way, then we can express this as a generalisation to another (possibly abstract) actor

  The descendent actors inherit the roles and relationships to use cases

held by the ancestor actor

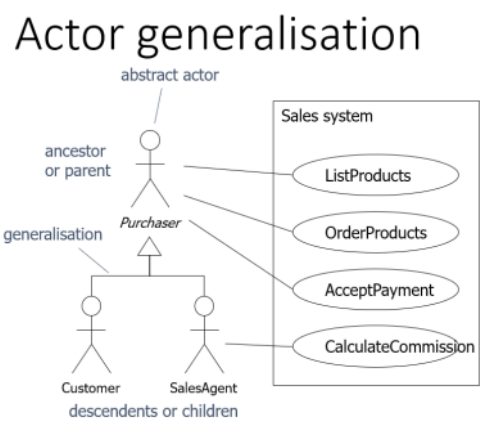
We can substitute a descendent actor anywhere the ancestor actor is expected. This is the substitutability principle

Use case generalisation

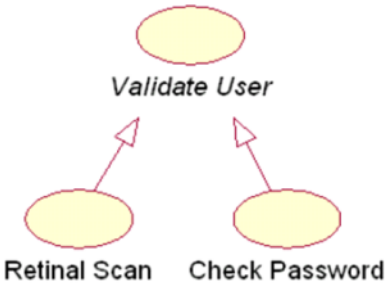
The ancestor use case must be a more general case of one or more descendant use cases

Child use cases are more specific forms of their parent

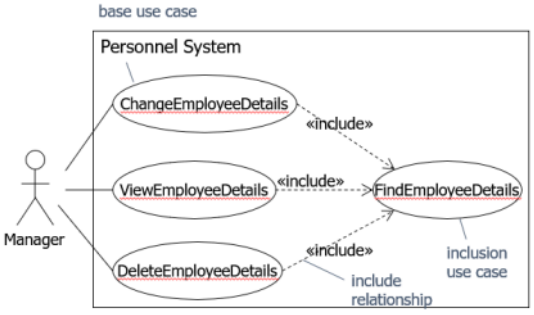
They can inherit, add and override features of their parent



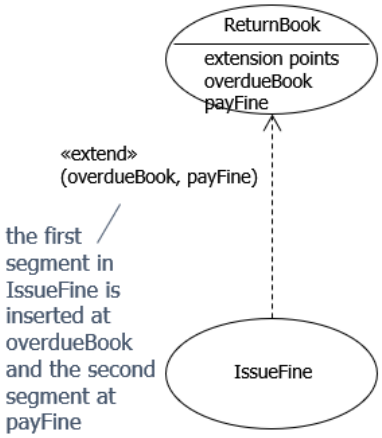
- «include»
  - The base use case executes until the point of inclusion: include(InclusionUseCase)
    - Control passes to the inclusion use case which executes
    - When the inclusion use case is finished, control passes back to the base use case which finishes execution
  - Note:
    - Base use cases are not complete without the included use cases
    - Inclusion use cases may be complete use cases, or they may just specify a fragment of behaviour for inclusion elsewhere

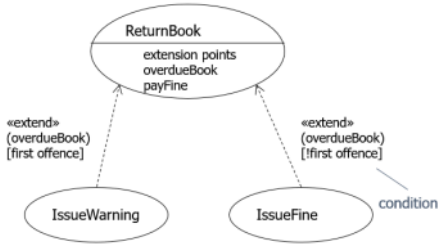


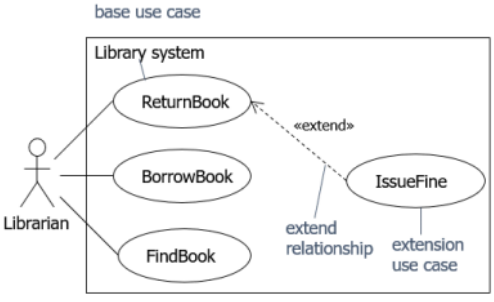
- «extend»
  - «extend» is a way of adding new behaviour into the base use case by inserting behaviour from one or more extension use cases
    - The base use case specifies one or more extension points in its flow of events
  - The extension use case may contain several insertion segments
  - The «extend» relationship may specify which of the base use case extension points it is extending



- Multiple insertion points
  - If more than one extension point is specified in the «extend» relationship then the extension use case must have the same number of insertion segments







- Conditional extensions
  - We can specify conditions on «extend» relationships
    - Conditions are Boolean expressions
    - The insertion is made if and only if the condition evaluates to true

- Use case specification

use case name	Use case: PaySalesTax
use case identifier	ID: 1
brief description	Brief description: Pay Sales Tax to the Tax Authority at the end of the business quarter.
the actors involved in the use case	Primary actors: Time
	Secondary actors: TaxAuthority
the system state before the use case can begin	Preconditions: 1. It is the end of the business quarter.
the actual steps of the use case	Main flow: (implicit time actor) 1. The use case starts when it is the end of the business quarter. 2. The system determines the amount of Sales Tax owed to the Tax Authority. 3. The system sends an electronic payment to the Tax Authority.
the system state when the use case has finished	Postconditions: 1. The Tax Authority receives the correct amount of Sales Tax.
alternative flows	Alternative flows: None.

- Pre and postconditions
  - ○ Preconditions and postconditions are constraints
  - ○ Preconditions constrain the state of the system before the use case can start
  - ○ Postconditions constrain the state of the system after the use case has executed
  - ○ If there are no preconditions or postconditions write "None" under the heading
- Main flow
  - <number> The <something> <some action>
    - The flow of events lists the steps in a use case
    - It always begins by an actor doing something
      - A good way to start a flow of events is:
        The use case starts when an <actor> <function>
    - The flow of events should be a sequence of short steps that are:
      - Declarative
      - Numbered,
      - Time ordered
    - The main flow is always the happy day or perfect world scenario
      - Everything goes as expected and desired, and there are no errors, deviations, interrupts, or branches
      - Alternatives can be shown by branching or by listing under Alternative flows
- Branching within a flow: If
  - ○ Use the keyword if to indicate alternatives within the flow of events
    - ○ There must be a Boolean expression immediately after if
  - ○ Use indentation and numbering to indicate the conditional part of the flow

Use else to indicate what happens if the condition is false

1. The shopping basket contents are visible.

Main flow:
1. The use case starts when the Customer selects an item in the basket.
2. If the Customer selects "delete item"
   2.1 The system removes the item from the basket.
3. If the Customer types in a new quantity
   3.1 The system updates the quantity of the item in the basket.

Postconditions:

- Repetition within a flow: For
  - We can use the keyword For to indicate the start of a repetition within the flow of events
  - The iteration expression immediately after the For statement indicates the number of repetitions of the indented text beneath the For statement.

4. The system searches for products that match the Customer's criteria.
5. For each product found
   5.1. The system displays a thumbnail sketch of the product.
   5.2. The system displays a summary of the product details.
   5.3. The system displays the product price.

Postconditions:

- Repetition within a flow: While
  - We can use the keyword while to indicate that something repeats while some Boolean condition is true

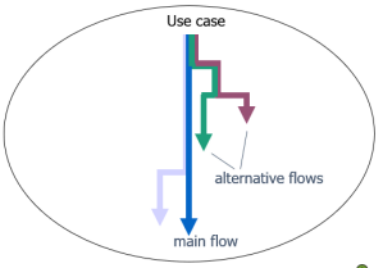
None.

Main flow:
1. The use case starts when the Customer selects "show company details".

Boolean condition is true

None.

Main flow:
1. The use case starts when the Customer selects "show company details".
2. The system displays a web page showing the company details.
3. While the Customer is browsing the company details
4. The system searches for products that match the Customer's criteria.
   4.1. The system plays some background music.
   4.2. The system displays special offers in a banner ad.



- Branching: Alternative flows
  - We may specify one or more alternative flows through the flow of events:
    - Alternative flows capture errors, branches, and interrupts
  - Potentially very many alternative flows! You need to manage this:
    - Pick the most important alternative flows and document those.
    - If there are groups of similar alternative flows - document one member of the group as an exemplar and (if necessary) add notes to this explaining how the others differ from it.
- Referencing alternative flows
  - List the names of the alternative flows at the end of the use case
  - Find alternative flows by examining each step in the main flow and looking for:
    - Alternatives
    - Exceptions
    - Interrupts
  - The alternative flow may be triggered instead of the main flow - started by an actor
  - The alternative flow may be triggered after a particular step in the main flow - after
  - The alternative flow may be triggered at any time during the main flow - at any time
- Modeling Methods
  - Establish the context of the system by identifying the actors that surround it.
  - For each actor, consider the behavior that each expects or requires the system to provide.
  - Name these behaviors as use cases.
  - Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
  - For each use case, describe the typical interactions between the actors and the system.
  - Model these use cases, actors, and their relationships in a use case diagram.
  - Adorn these use cases with notes or constraints that assert nonfunctional requirements.
- When to use case analysis
  - Use cases describe system behaviour from the point of view of one or more actors. They are the best choice when:
    - The system is dominated by functional requirements
    - The system has many types of user to which it delivers different functionality
    - The system has many interfaces
  - Use cases are designed to capture functional requirements. They are a poor choice when:
    - The system is dominated by non-functional requirements
    - The system has few users
    - The system has few interfaces