

The principal goal of information systems analysis is to state accurately users' requirements for a new information processing system.

A six-step process for object-oriented analysis is introduced. Together, these six steps produce an event model, a use case model, system sequence diagrams, a model of the problem domain, and system operation contracts.

An event is an occurrence which takes place at a specific time and triggers a predetermined response from the system.

Procedure for Object-Oriented Systems Analysis

Step 1. Identify the business events and make an event table.

Step 2. Identify the use cases and produce a use case diagram for the system.

Step 3. Write a use case narrative describing the system's response to each business event.

Step 4. Draw a system sequence diagram for each use case scenario.

Step 5. Produce a domain model showing the concepts, attributes and associations in the problem domain of the system.

Step 6. Write a contract for each system operation.

An event is an occurrence which takes place at a specific time and initiates or triggers a predetermined response from the system.

- An external event is an event which occurs outside the system boundary.
- An internal event is an event which occurs inside the system boundary.
- A temporal event is an event which occurs at a prespecified time.

Event analysis creates a system description by identifying:

1. The events to which the system is expected to respond
2. The incoming message (event flow or data flow) associated with each event
3. The desired response
4. The actions or behaviours required to generate the response for each stimulus

The goal of systems analysis is to state users' requirements for a new information system correctly.

The initial step in object-oriented systems analysis is event analysis.

Event analysis identifies external and temporal events and the systems expected responses.

EVENT NUMBER	EVENT DESCRIPTION	SYSTEM INPUT	ACTOR PROVIDING INPUT	SYSTEM OUTPUT	ACTOR RECEIVING OUTPUT
1.	Department submits class schedule.	Department Class Schedule	Department		
2.	Time to produce class schedule			University Class Schedule	Student Department Professor
3.	Student registers for classes	Registration Request	Student	Student Class List	Student
4.	Time to produce class roster			Class Roster	Professor

Operations contract

Contract

Name:

requestSection

(departmentCode,
courseNumber,
sectionNumber)

Responsibilities:

Enroll the Student in the Section.

Type:

System

Exceptions:

If the combination of department code, course number and section number is not valid, indicate that it was an error.
If no seats are available, inform the Student.

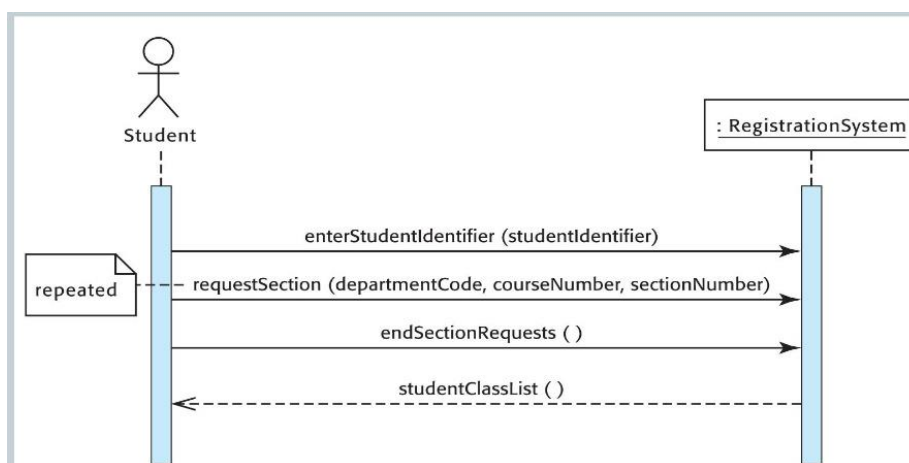
Output:

Preconditions:

Department and Section are known to the system.

Postconditions:

A new instance of the Enrolled In association was created, linking the Student and the Section.



Use Cases

Use case diagrams show the use cases within the scope of the system and the actors in the environment with which each use case is associated.

There is at least one use case narrative for each use case. A use case narrative is a structured narrative showing what the system must do to respond to a specific event.

A use case is the sequence of actions which occur when an actor uses a system to complete a process.

A use case is a model of a requirement.

A use case name is a short phrase beginning with a verb.

Each event corresponds to at least one use case.

An actor is a person, organization, or system which interacts with a system by sending messages to the system or receiving messages from the system.

An initiating actor initiates a use case by initiating an external event. Thus, initiating actors provide system inputs.

A participating actor is involved in a use case but does not initiate it. Thus, participating actors receive system outputs.

Use case:	Register for Classes
Actors:	Student
Purpose:	Register a student for classes and record the student's schedule.
Description:	A Student requests the sections of classes desired for a term. The system adds the Student to each section if space is available. On completion, the Student receives a list of the classes in which he or she is enrolled.

Use case:	Register for Classes
Actors:	Student
Purpose:	Register a student for classes and record the student's schedule.
Overview:	A Student requests the sections of class desired for a term. The system adds the Student to each section if there is space available. On completion, the system provides the Student with a list of the classes in which he or she is enrolled.
Type:	Essential
Preconditions:	Class schedule must exist. Student is known by the system.
Postconditions:	Student was enrolled in the section.
Special Requirements:	Student must get a system response within 10 seconds.

Flow of Events

ACTOR ACTION	SYSTEM RESPONSE
1. This use case begins when a Student desires to register for classes.	
2. The Student provides the Student's identifier and a list of the department code, course number, and section number for each section desired.	3. Adds the student to the section if there are seats available.
4. On completion of entry of the section requests, the Student indicates that the request is complete.	5. Produces a student class list for the Student.
6. The Student receives the student class list.	

Alternative Flow of Events

- Line 3: Invalid department code and course number entered. Indicate error.
Return to Step 2.
Invalid section number entered. Indicate error. Return to Step 2.
No seats remaining. Inform the Student. Return to Step 2.

A use case scenario is a narrative of a single occurrence of a use case. It describes specifics of a real world enactment of the use case.
Use case scenarios can help discover alternative paths through a use case or test the completeness or correctness of a use case narrative.

Class Diagram Notes

- association -- a relationship between instances of the two classes. There is an association between two classes if an instance of one class must know about the other in order to perform its work. In a diagram, an association is a link connecting two classes.
- aggregation -- an association in which one class belongs to a collection. An aggregation has a diamond end pointing to the part containing the whole. In our diagram, Order has a collection of OrderDetails.
- generalization -- an inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass. *Payment* is a superclass of Cash, Check, and Credit.

An association has two ends. An end may have a role name to clarify the nature of the association. For example, an OrderDetail is a line item of each Order.

A navigability arrow on an association shows which direction the association can be traversed or queried. An OrderDetail can be queried about its Item, but not the other way around. The arrow also lets you know who "owns" the association's implementation; in this case, OrderDetail has an Item. Associations with no navigability arrows are bi-directional.

The multiplicity of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers. In our example, there can be only one Customer for each Order, but a Customer can have any number of Orders.

Software Design Principles

Rigidity is the tendency for software to be difficult to change, even in simple ways.

- Symptom: Every change causes a cascade of subsequent changes in dependent modules.
- Effect: When software behaves this way, managers fear to allow developers to fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the developers will be finished.

Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.

- Symptom: Every fix makes it worse, introducing more problems than are solved.
- Effect: Every time managers/ team leaders authorize a fix, they fear that the software will break in some unexpected way.

Immobility is the inability to reuse software from other projects or from parts of the same project.

- Symptom: A developer discovers that he needs a module that is similar to one that another developer wrote. But the module in question has too much baggage that it depends upon. After much work, the developer discovers that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate.
- Effect: And so the software is simply rewritten instead of reused.

Viscosity is the tendency of the software/development environment to encourage software changes that are hacks rather than software changes that preserve original design intent.

- Symptom: It is easy to do the wrong thing, but hard to do the right thing.
- Effect: The software maintainability degenerates due to hacks, workarounds, shortcuts, temporary fixes etc.

Law of Demeter

Also known as the Principle of least knowledge

Is a design principle which provides guidelines for designing a system with minimal dependencies. It is typically summarized as "Only talk to your immediate friends."

Singleton Pattern

Singleton pattern used when we want to allow only a single instance of a class can be created inside our application. Using this pattern ensures that a class only have a single instance by protecting the class creation process, by setting the class constructor into private access modifier




Delegation Pattern

1. It is a technique where an object expresses certain behaviour to the outside but in reality delegates responsibility for implementing that behaviour to an associated object
2. Delegation is the simple yet powerful concept of handing a task over to another part of the program. In object-oriented programming it is used to describe the situation where one object assigns a task to another object, known as the delegate.
3. Generally spoken: use delegation as alternative to inheritance.
4. Inheritance is a good strategy, when a close relationship exist in between parent and child object, however, inheritance couples objects very closely. Often, delegation is the more flexible way to express a relationship between classes.
5. "Delegation is like inheritance done manually through object composition.

Interface Segregation Principle

Interface segregation is a design principle that deals with the disadvantages of “fat” interfaces. Interfaces containing methods that are not specific to it are called polluted or fat interfaces.

Open Closed Principle

-  We should write our modules so that they can be extended, without requiring them to be modified. In other words, we want to be able to change what the modules do, without changing the source code of the modules.
-  How?: Abstraction and Polymorphism
-  Allow the modules (classes) to depend on the abstractions, there by new features can be added by creating new extensions of these abstractions.