REQUIREMENTS ELICITATION

REQUIREMENTS ELICITATION : → definition of the system in terms understood by the customer

ANALYSIS : → definition of the system in terms understood by the developer

1. FUNCTIONAL REQUIREMENTS :

- describe user tasks that the system needs to support
- phrased as actions
- the use-case diagram

2. NON-FUNCTIONAL REQUIREMENTS :

- describe properties of the system or the domain
- phrased as constraints or negative assertions

TYPES :

- usability
- reliability
- performance requirements
- supportability requirements
- implementation requirements
- interface requirements
- operations requirements
- packaging requirements

→ QUALITY REQUIREMENTS (FURPS+)

→ CONSTRAINTS (PSEUDO REQUIREMENTS)

REQUIREMENTS VALIDATION :

- a quality assurance step, usually performed after requirements elicitation or analysis
- correctness, completeness, consistency, clarity, realism, traceability

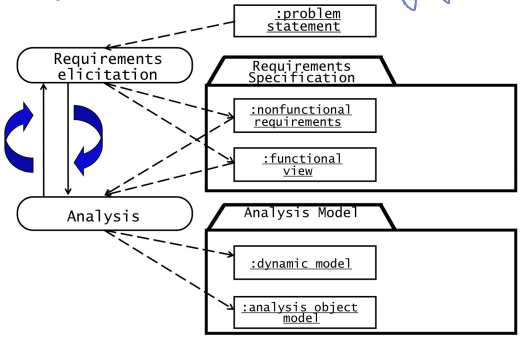
TYPES OF REQUIREMENTS ELICITATION :

1. Greenfield Engineering : → development starts from scratch
   → triggered by user needs
2. Re-engineering : → re-design / re-implementation of an existing system
   → triggered by technology enabler
3. Interface Engineering : → provision of existing services in a new environment
   → triggered by technology enabler or new market needs

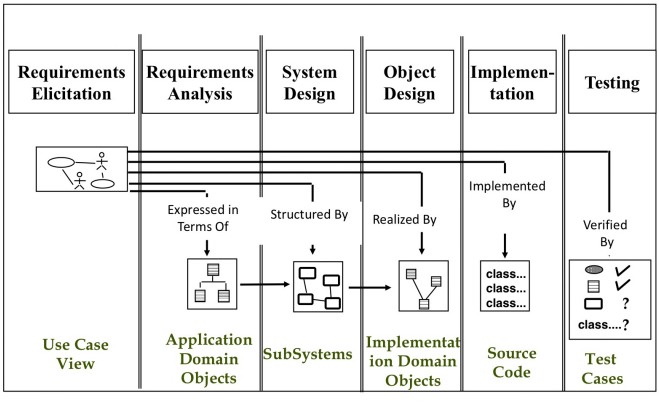
REQUIREMENTS SPECIFICATION : → uses natural language

ANALYSIS MODEL : → uses formal or semi-formal language (UML)

REQUIREMENTS PROCESS :



SOFTWARE LIFECYCLE ACTIVITIES :



SCENARIO : → a description of what actors do as they use the system

- each scenario has : → an use case
  → an acceptance use case

USE CASE ASSOCIATIONS :

- dependencies between use cases are represented with the use case associations
- used to reduce complexity
- types : → includes
  → extends
  → generalization

<< INCLUDE >> : → problem : a function is too complex
→ solution : describe the function as the aggregation of a set of simpler functions

<< EXTEND >> : → problem : the functionality in the original statement needs to be extended
→ solution : an extend association from use case A to use case B

GENERALIZATION : → problem : we want to factor out common behavior
→ solution : the child use cases inherit the behavior and meaning of the parent use case and add or override some behavior

THE ANALYSIS MODEL

ANALYSIS OBJECT VIEW :

1. Entity objects : ⟶ the persistent information tracked by the system
2. Boundary objects : ⟶ the interaction between the actors and the system
3. Control objects : ⟶ are in charge of realizing use cases

GENERALIZATION & SPECIALIZATION :

- result in the specification of inheritance relationships between concepts

ANALYSIS ACTIVITIES → FROM USE CASES TO OBJECTS :

- identifying : → Entity Objects, Boundary Objects, Control Objects
- mapping Use Cases to Objects with Sequence Diagrams
- mapping interactions among Objects with CRC Cards
- identifying → Associations, Aggregates, Attributes
- modelling State-Dependent Behavior of Individual Objects
- modelling Inheritance Relationships
- reviewing the Analysis Model

BOUNDARY OBJECTS : ⟶ represent the system interface with the actors

- in each use case, each actor interacts with at least one boundary object
- it collects the information from the actor and translates it into a form that can be used by both entity and control objects

CONTROL OBJECTS : → are responsible for coordenating boundary and entity objects

SEQUENCE DIAGRAM :
- → ties use cases with objects
- it shows how the behaviour of a use case is distributed among participating objects
- first column ⇒ the actor
- second column ⇒ a boundary object
- third column ⇒ the control object

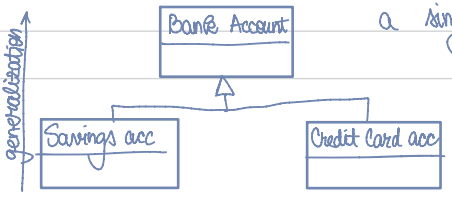
ASSOCIATION : → two classes need to communicate with each other

→ a student can associate with multiple teachers

AGGREGATION : → implies a relationship where the child can exist independently of the parent

COMPOSITION : → implies a relationship where the child cannot exist independent of the parent

GENERALIZATION : → a mechanism for combining similar classes of objects into a single, more general class



SYSTEM DESIGN

SYSTEM DESIGN : ⟶ is the transformation of an analysis model into a system design model

design patterns

THE ADAPTER PATTERN :

- works as a bridge between two incompatible interfaces.
- it is a Structural Pattern because it combines the capability of two independent interfaces
- it involves a single class which is responsible to join functionalities of independent or incompatible interfaces

THE PROXY PATTERN :

- the ProxyObject class acts on behalf of a RealObject class.
- both classes implement the same interface
- the ProxyObject stores a subset of the attributes of the RealObject
- the ProxyObject handles certain requests completely, whereas others are delegated to the RealObject
- after delegation, the RealObject is created and loaded in memory

THE BRIDGE PATTERN :

- bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently
- this pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes
- both types of classes can be altered structurally without affecting each other concrete classes

THE COMPOSITE PATTERN :

- modern toolkits enable developers to organize the user interface objects into hierarchies of aggregate nodes, called panels, that can be manipulated the same way as the concrete user interface objects
- each panel is responsible for the layout of its subpanels called children, and the overall dialog must deal with x panels

THE COMMAND PATTERN :

- a request is wrapped under an object as command and passed to invoker object
- invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command

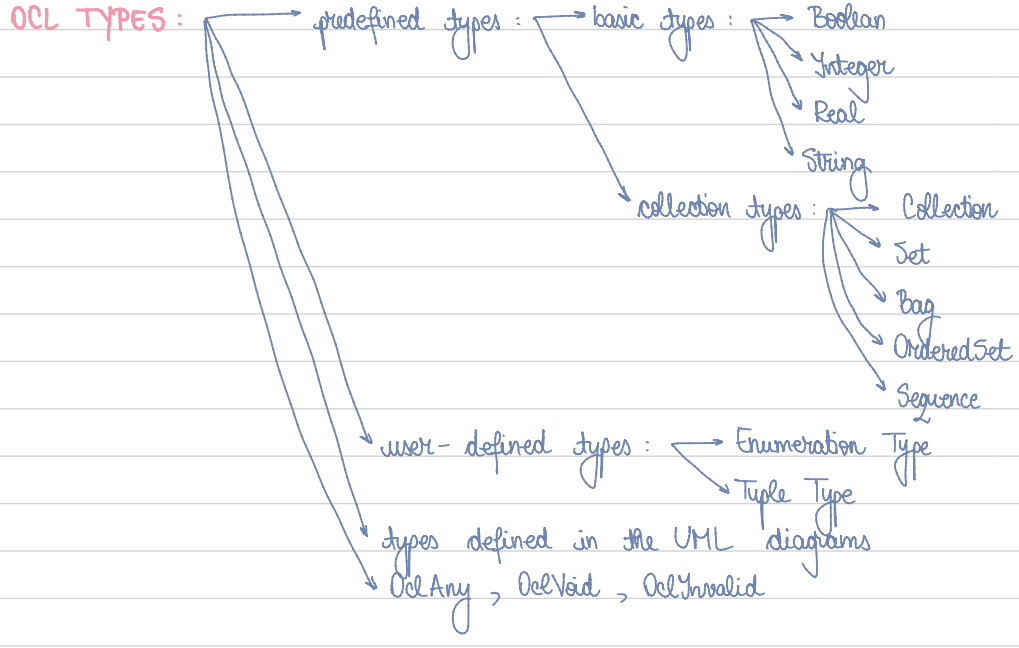
THE OBSERVER PATTERN :

- is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically
- it falls under behavioral pattern category
- uses 3 actor classes :
  - Subject
  - Observer
  - Client
- subject is an object having methods to attach and deattach observers to a client object

THE STRATEGY PATTERN :

- a class behavior or its algorithm can be changed at run time
- it is a Behavior pattern
- we create objects which represent various strategies and a context object whose behavior varies as per its strategy object
- the strategy object changes the executing algorithm of the context object

OCL



PRECONDITIONS : → must be true just prior to the execution of an operation

→ syntax : context < classifier > :: < operation > (< parameters >)
pre [< constraint name >] :
< Boolean OCL expression >

POSTCONDITIONS : → constraint that must be true just after to the execution of an operation

→ are the way how the actual effect of an operation is described in OCL

OCLANY :

- the supertype of all types in UML models
- operations :
  - =(object2 : OclAny) : Boolean
  - <>(object2 : OclAny) : Boolean
  - oclIsTypeOf (type : Classifier) : Boolean
  - oclIsKindOf (type : Classifier) : Boolean
  - oclType ( ) : Classifier
  - oclIsInState (statespec : OclState) : Boolean
  - oclAsType (type : Classifier) : T → type casting
  - oclIsNew ( ) : Boolean
  - oclIsUndefined ( ) : Boolean
  - oclIsInvalid ( ) : Boolean

pt. inheritance

COLLECTION OPERATIONS :

- iterate : collection → iterate (elem : Type ; acc : Type = < expression > |
  expression – with – elem – and – acc)
  - elem iterates over the collection and the expression – with – elem – and – acc is evaluated for each elem and its value is assigned to acc
- select :
  - collection → select (elem : T | expression)
  - collection → select (elem | expression)
  - collection → select (expression)
  - ⇒ subset of all elements for which expression is true
- forAll :
  - collection → forAll (elem : T | expr)
  - collection → forAll (elem | expr)
  - collection → forAll (expr)
  - ⇒ true if expr is true for all elements of the collection

- exists :
  - collection → exists (elem : T | expr)
  - collection → exists (elem | expr)
  - collection → exists (expr)
  - ⇒ True if there is at least one element in the collection for which expr is True

- *isEmpty*: **true** if collection has no elements
- *notEmpty()*: **true** if collection has at least one element
- *size*: number of elements in collection
- *count(elem)*: number of occurences of elem in collection
- *includes(elem)*: **true** if elem is in collection
- *including(elem)*: **returns** a collection a similar collection including *elem*
- *excludes(elem)*: **true** if elem is not in collection
- *includesAll(coll)*: **true** if all elements of coll are in collection

testing

TESTING : → the process of finding differences between the expected behavior specified by system models and the observed behavior of the implemented system

TEST COMPONENT → a part of the system that can be isolated for testing
- an object / a group of objects / one or more subsystems

FAULT / BUG / DEFECT → a design or coding mistake that may cause abnormal component behavior

ERRONEOUS STATE → manifestation of a fault during the execution of the system

FAILURE → a deviation between the specification and the actual behavior

TEST CASE → a set of inputs and expected results (= oracles) that exercises a test component with the purpose of causing failures and detecting faults.

TEST STUB → the called program

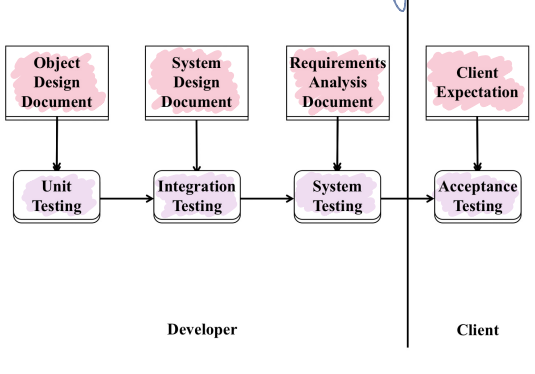
TEST DRIVER → the calling program

example : We test the service and it calls the repo => repo = stub & service = driver

CORRECTION → a change to a component with the purpose to repair a fault

BLACKBOX TESTS → focus on the input / output behavior of the component

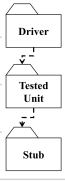
WHITEBOX TESTS → focus on the internal structure of the component
- code coverage

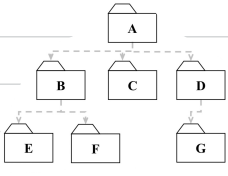


UNIT TESTING: → tests: individual component (class or subsystem)
→ types: → Static Testing (at compile time)
→ Dynamic Testing (at run time): → Black-box testing
→ White-box testing

INTEGRATION TESTING: → tests: groups of subsystems and eventually the entire system

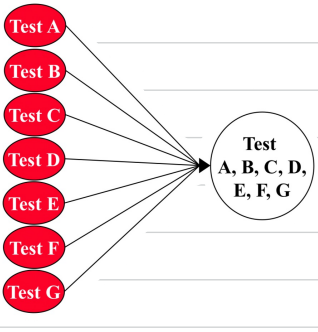
- Driver → a component that calls the TestedUnit
- Stub → a component the TestedUnit depends on



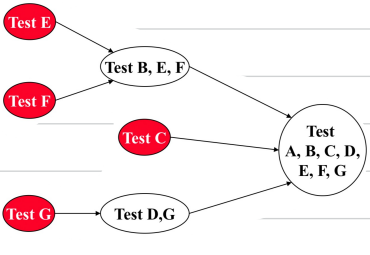


TYPES:

1. Big-Bang Approach:

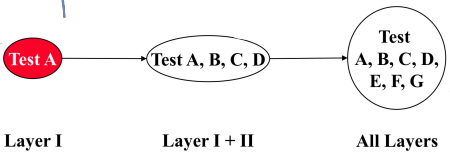


2. Bottom-up:



- pro: → no stubs needed
  → useful for: Object-oriented and real-time systems
- con: → drivers needed
  → tests the most important subsystem (UI) last

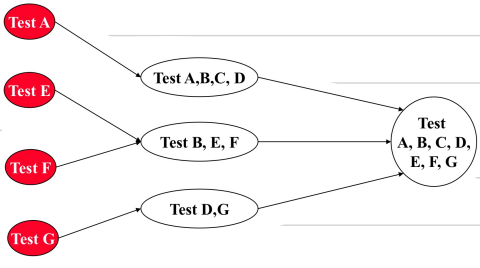
3. Top - Down :



→ pro : → no drivers needed
→ test cases defined in terms of the functionality of the system

→ con : → stubs needed

4. Sandwich :



→ combines top-down with bottom-up
→ system viewed as having 3 layers :
- target layer → middle
- a layer above target
- a layer below target

CONTINUOUS TESTING : → testing from day one

SYSTEM TESTING :
- Functional testing : → validates functional requirements
- Performance testing : → validates non-functional requirements
- Acceptance testing : → validates clients expectations

DIAGRAMS

diagrams

- use case diagram
  - → demonstrate the different ways that a user might interact with a system
  - → can help your team discuss and represent
    1. scenarios in which your system or application interacts with people, organizations or external systems
    2. goals that your system or application helps those entities (known as actors) achieve
    3. the scope of your system

Components

- ACTORS the user that interacts with a system
  ex a person, an organization, an outside system
- SYSTEM a specific sequence of actions and interactions between actors and system
- GOALS The end result of most use cases

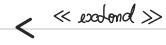
Symbols and notation

actor

use case

association
(actor → use case)

include relationship



extend relationship

- class diagram
  → map out the structure of a particular system by modeling its classes, attributes, operations and relationships between objects

Components

- UPPER SECTION name of the class
- MIDDLE SECTION attributes of the class (describe the qualities of the class)
- BOTTOM SECTION includes class operations (methods)

Symbols and notation

→ access levels depending on the visibility

+	public
−	private
#	protected
underlined	static

private:
Scope:
Limited to the same class.
Visibility:
Members (variables, methods, constructors) marked as private are accessible only within the same class.
Example:
private int age; (A private variable can only be accessed within the class it is declared in.)

public:
Scope:
Accessible from anywhere.
Visibility:
Members marked as public are accessible from any class or package.
Example:
public void displayInfo() { ... } (A public method can be accessed from any class or package.)

protected:
Scope:
Accessible within the same package and subclasses (even if they are in a different package).
Visibility:
Members marked as protected are accessible within the same package, as well as in subclasses (regardless of the package).
Example:
protected String name; (A protected variable can be accessed within the same package and subclasses.)

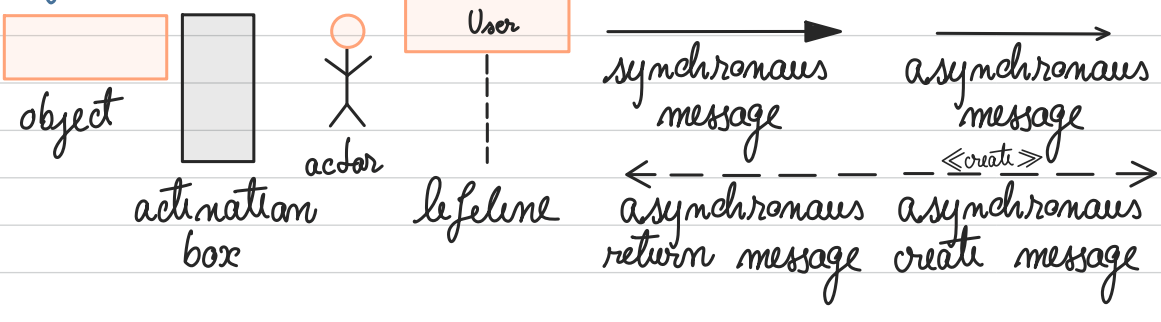
static:
Usage:
Associated with a class itself rather than with instances of the class.
Behavior:
Static members belong to the class itself and are not tied to specific instances. They can be accessed using the class name without creating an object.
Example:
public static int count; (A static variable is shared across all instances of the class and can be accessed using the class name, e.g., ClassName.count.)

- sequence diagram
  - a popular dynamic modeling solution
  - focus on lifelines, or the processes and objects that live simultaneously, and the messages exchanged between them to perform a function before the lifeline ends
  - a type of interaction diagram because it describes how (and in what order) a group of objects works together (known as event diagram or event scenarios)

Benefits

1. represent the details of a UML use case
2. model the logic of a sophisticated procedure, function or operation
3. see how objects and components interact with each other to complete a process
4. plan and understand the detailed functionality of an existing or future scenario

Symbols and notation



- **activity diagram**
  - → essentially a flowchart that shows activities performed by a system
  - → help people on the business and development sides of an organization come together to understand the same process and behavior

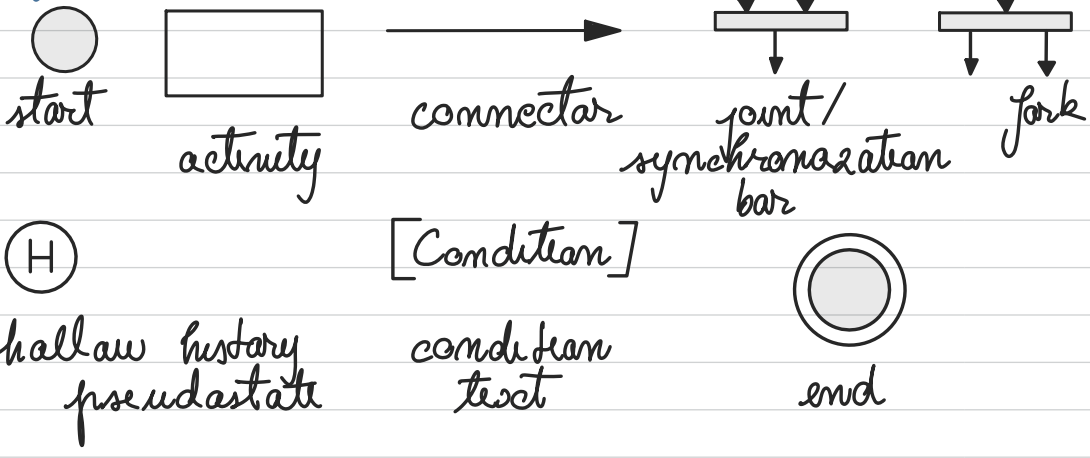
Benefits

1. demonstrate the logic of an algorithm
2. describe the steps performed in a UML use case
3. illustrate a business process or workflow between users and the system
4. simplify and improve any process by clarifying complicated use cases
5. model software architecture elements, such as method, function and operation

Components

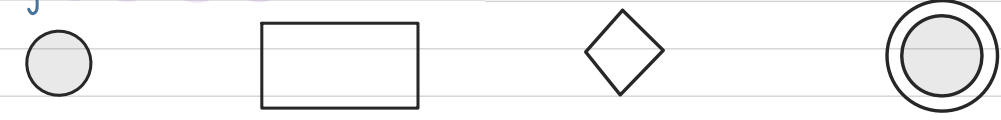
- ACTION — a step in the activity wherein the users or software perform a given task
- DECISION NODE — a conditional branch in the flow
- CONTROL FLOWS — connectors that show the flow between steps in the diagram
- START NODE — the beginning of the activity
- END NODE — the final step in the activity

Symbols and notation



- **state diagram**
  - → known as a state machine diagram, is a type of behaviour diagram
  - → shows transitions between various objects

Symbols and notation



- component diagram
  - → integral to building your software system
  - → help understand structure of existing system

Benefits

1. imagine the system's physical structure
2. pay attention to the system's components and how they relate
3. emphasize the service behavior as it relates to the interface

Symbols and notation