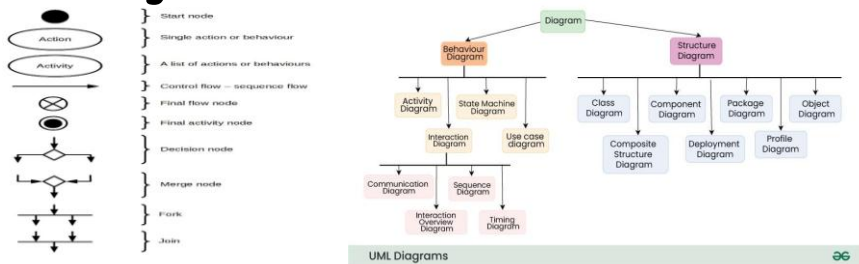


Software Engineering And Testing

Object Modeling using UML



State Diagram

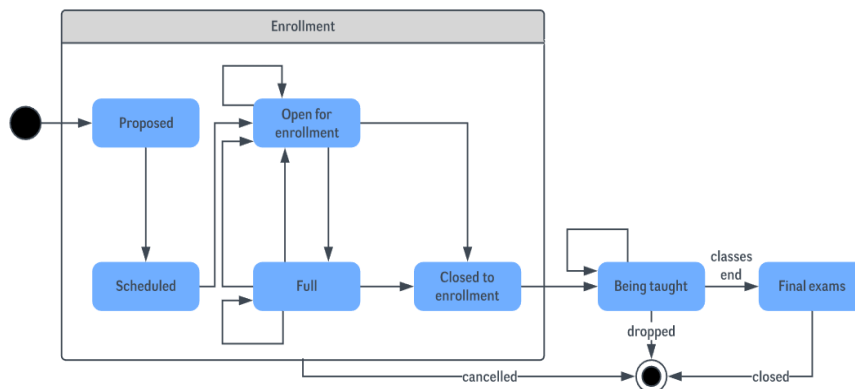
- States refer to the different combinations of information that an object can hold, not how the object behaves.
- Each state diagram typically begins with a dark circle that indicates the initial state and ends with a bordered circle that denotes the final state.
- States are represented with rectangles with rounded corners that are labeled with the name of the state.
- Transitions are marked with arrows that flow from one state to another, showing how the states change.

State Diagram


- The main applications are as follows:
 - Depicting event-driven objects in a reactive system.
 - Illustrating use case scenarios in a business context.
 - Describing how an object moves through various states within its lifetime
 - Showing the overall behavior of a state machine or the behavior of a related set of state machines.

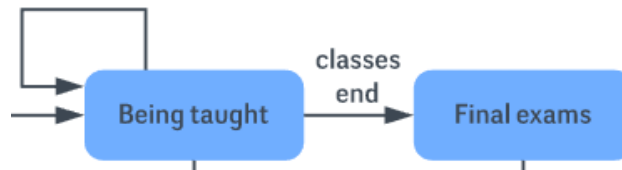
State Diagram Component

- **Composite state** : A state that has substates nested into it. See the university state diagram example below. “Enrollment” is the composite state in this example because it encompasses various substates in the enrollment process.



State Diagram Component

- **Choice pseudostate** : A diamond symbol that indicates a dynamic condition with branched potential results. 
- **Event** : An instance that triggers a transition, labeled above the applicable transition arrow. In this case, “classes end” is the event that triggers the end of the “Being taught” state and the beginning of the “Final exams” state.



- **First state** : A marker for the first state in the process, shown by a dark circle with a transition arrow.



State Diagram Component

- **Exit point** : The point at which an object escapes the composite state or state machine, denoted by a circle with an X through it. The exit point is typically used if the process is not completed but has to be escaped for some error or other issue.



- **State** : A rectangle with rounded corners that indicates the current nature of an object.



- **Terminator** : A circle with a dot in it that indicates that a process is terminated.



- **Transition**: An arrow running from one state to another that indicates a changing state.



State Diagram Component

- **Exit point** : The point at which an object escapes the composite state or state machine, denoted by a circle with an X through it. The exit point is typically used if the process is not completed but has to be escaped for some error or other issue.



- **State** : A rectangle with rounded corners that indicates the current nature of an object.



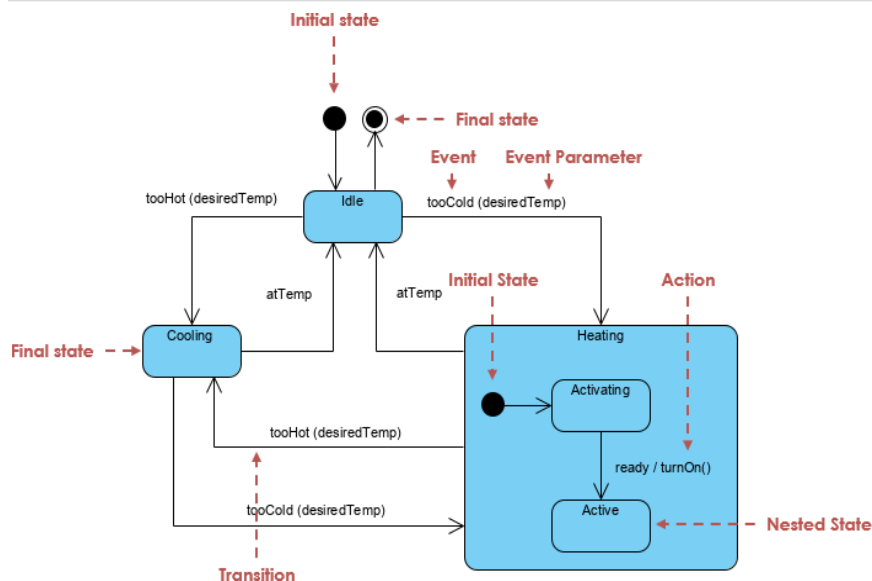
- **Terminator** : A circle with a dot in it that indicates that a process is terminated.



- **Transition**: An arrow running from one state to another that indicates a changing state.



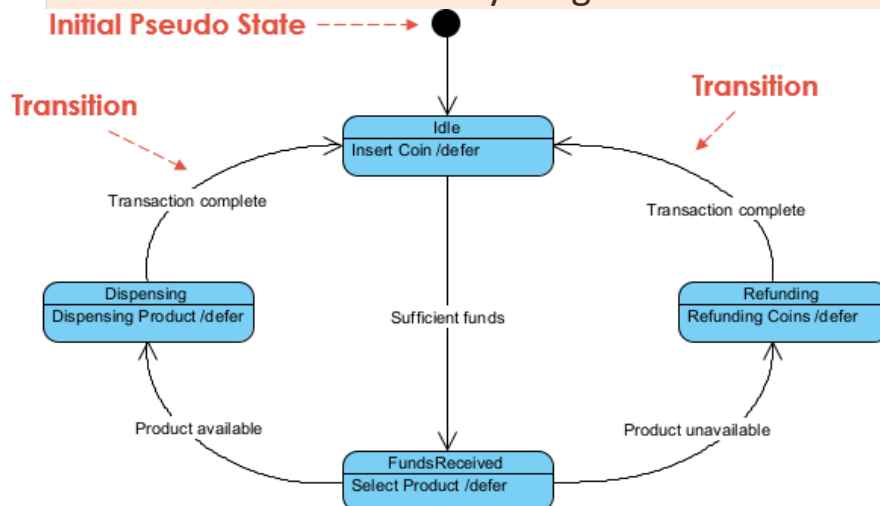
State Diagram Example



Similarities And Differences Between State Chart And Activity Diagrams

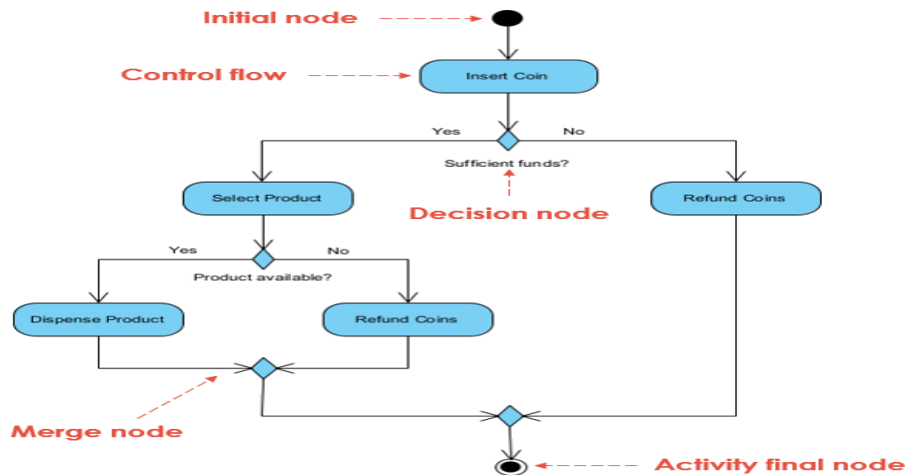
	State Chart Diagram	Activity Diagram
Purpose	Model reactive systems	Model non-reactive systems
Focus	Internal state of an object or system	Sequence of activities involved in a process or workflow
Elements	States, transitions, events	Activities, actions, transitions
Complexity	More complex	Less complex
Use cases	Embedded systems, control systems, real-time systems	Business processes, workflows, software processes

Similarities And Differences Between State Chart And Activity Diagrams



Modeling a Vending Machine by a State Chart

Similarities And Differences Between State Chart And Activity Diagrams



Modeling a Vending Machine by a activity Diagram

Patterns

- Pattern is a body of literature to help software developers resolve common, difficult problem and a vocabulary for communicating insight and experience about problems and their solutions
- A pattern is an instructive information that captures the essential structure and insight of a **successful family of proven solutions** to a **recurring problem** that arises within a certain context and system of forces



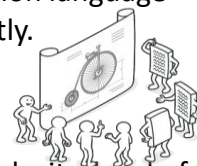
Patterns

- A good pattern will do the following:
 - *It solves a problem.* Patterns capture solutions, not just abstract principles or strategies
 - *It is a proven concept.* Patterns capture solutions with a track record, not theories or speculation
 - *The solution is not obvious.* The best patterns generate a solution to a problem indirectly—a necessary approach for the most difficult problems of design
 - *It describes a relationship.* Patterns do not just describe modules, but describe deeper system structures and mechanisms
 - *The pattern has a significant human component.* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility

13

Patterns

- **Design patterns** are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- **Benefits of patterns** are a toolkit of solutions to common problems in software design. They define a common language that helps your team communicate more efficiently.
- **Classification** Design patterns differ by their complexity, level of detail and scale of applicability. In addition, they can be categorized by their intent and divided into three groups.



14

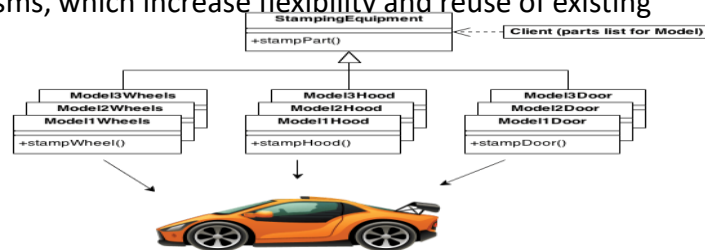
Design Patterns

- Creational Design patterns
- Structural design patterns
- Behavioral design patterns

15

Creational Design Patterns

- Creational patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



1. **Factory Method** : Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

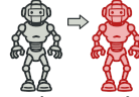


2. **Abstract Factory** : Lets you produce families of related objects without specifying their concrete classes.

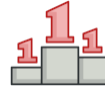


Creational Design Patterns

4. **Prototype** : Lets you copy existing objects without making your code dependent on their classes.



5. **Singleton** : Lets you ensure that a class has only one instance, while providing a global access point to this instance



6. **Builder** : Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

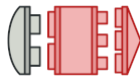


17

Structural Design Patterns

- Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

- **Adapter** : Allows objects with incompatible interfaces to collaborate.



- **Bridge** : Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



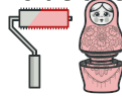
- **Composite** : Lets you compose objects into tree structures and then work with these structures as if they were individual objects



18

Structural Design Patterns

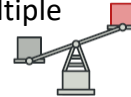
- **Decorator** : Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors



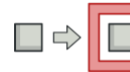
- **Facade** : Provides a simplified interface to a library, a framework, or any other complex set of classes.



- **Flyweight** : Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object



- **Proxy** : Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object



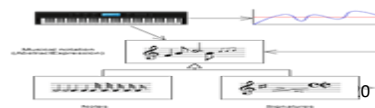
19

Behavioral Design Patterns

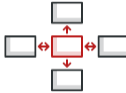
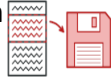

- Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects.
- **Chain of Responsibility** : Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



- **Command** : Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations

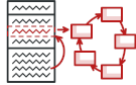
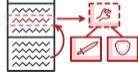

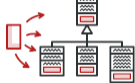


Behavioral Design Patterns

- **Iterator** : Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
- **Mediator** : Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
 
- **Memento** : Lets you save and restore the previous state of an object without revealing the details of its implementation
 
- **Observer** : Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing
 

21

Behavioral Design Patterns

- **State** : Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
 
- **Strategy** : Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
 
- **Template Method** : Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
 
- **Visitor** : Lets you separate algorithms from the objects on which they operate.
 

22

Object Oriented Methodologies

- Object oriented methodologies are set of methods, models, and rules for developing systems.
- Modeling provides means of communicating ideas which is easy to understand the system complexity .
- Object-Oriented Methodologies are widely classified into three
 1. The Rumbaugh et al. OMT (Object modeling technique)
 2. The Booch methodology
 3. Jacobson's methodologies

23



24