L2 - Design Patterns and UML - Class and Object diagrams

Definition 1 Patterns identify and specify abstractions that are above the level of single classes and instances, or of components.

Definition 2 Design Patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development.

Definition 3 Design Patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detail design and implementation

Definition 4 A pattern addresses a recurring design problem that arises in specific design situations and presents a solution to it.

Definition 5 Design Patterns are recurring solutions to design problems you see over and over.

Definition 6 Experienced OO developers build up a repertoire of general principles and idiomatic solutions that guide them in the creation of software. These may be called patterns.

Definition 7 Design Patterns are programming tools to improve code to be: easier to implement, and easier to maintain. are good answers to common and specialised problems. define a common (programming language independent) programming model that standardise common programming tasks into recognisable forms, giving your projects better cohesiveness.

When design patterns are applied we achieve: Improved maintainability of code Improved adaptability of code Improved reliability of code

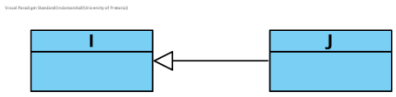
There are 23 classic patterns, identified by the GoF, categorised as:
Creational - creation of objects

Behavioural - interaction between objects

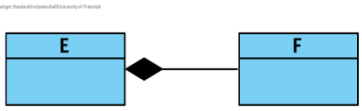
Structural - composition of objects

Category	Pattern	Strategy	
		Delegation	Inheritance
Creational	Factory Method		X
	Abstract Factory	X	
	Prototype	X	
	Builder	X	
	Singleton	X	
Behavioural	Memento	X	
	Template Method		X
	Strategy	X	
	State	X	
	Observer	X	
	Iterator	X	
	Mediator	X	
	Command	X	
	Chain of Responsibility	X	
	Interpreter		X
	Visitor	X	
Structural	Composite	X	
	Decorator	X	
	Adapter	X	X
	Bridge	X	
	Façade	X	
	Proxy	X	
	Flyweight	X	

Generalisation is a relationship for modelling **inheritance** and specifically *public* inheritance. It is the manifestation of the *is-a* relationship.



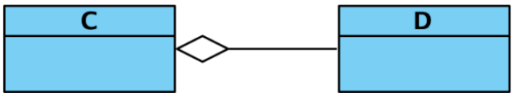
Composition (owns-a)



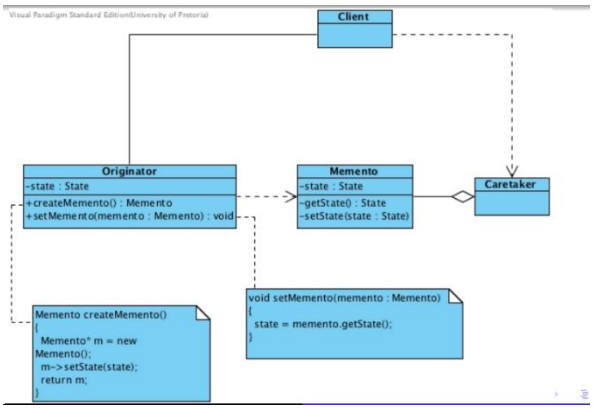
Dependency (uses-a)

Interface SiteSearch is used (required) by SearchController

Aggregation (has-a)



L3 – Memento



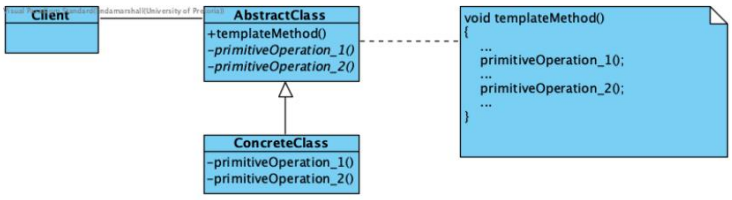
Name and Classification: Memento (Behavioural) Delegation (Object) Intent: "Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later." GoF(283)

The Memento: Stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary, at its originator's discretion.

The Originator: Creates a memento containing a snapshot of its current internal state (createMemento). Uses the memento to restore its internal state (setMemento).

The Caretaker: is responsible for the safekeeping of the memento's state, never operates on or examines the contents of a memento.

L4 – Template Method

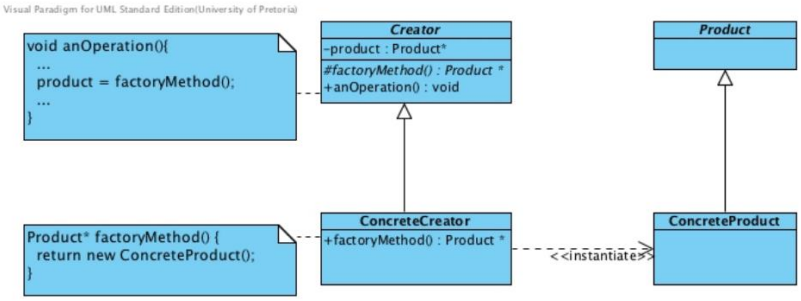


Name and Classification: Template Method (Class Behavioural) Intent: "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." GoF(325)

Abstract Class: defines abstract primitive operations that need to be defined by the concrete classes implements the template method operation that provides a skeleton of an algorithm

Concrete Class: implements the primitive operations defined by the Abstract class

L5 – Factory Method



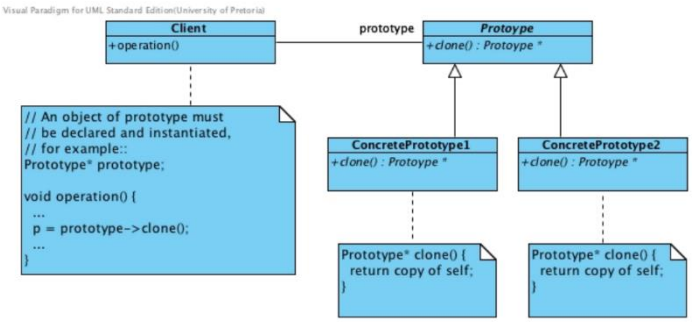
Name and Classification: Factory Method (Class Creational) Intent: "Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses." GoF(107)

- A Creator creates product which the client uses.
- **Product**: is always created by Creator.
- **ConcreteCreators**: create specific concrete product.
- Makes use of the Template Method design pattern.
- Forces the creation of an object to occur in a common factory rather than scattered around the code

- A factory can be implemented by using a static factory member, or by making use of polymorphism
- **Product**: defines the product interface for the factory method to create
- **Concrete Product**: implements the interface for the product
- **Creator**: declares the factory method which returns a product object default factory method implementations may return a default concrete product
- **Concrete Creator**: overrides the factory method to return an instance of the product

L6 – Prototype

Name and Classification: Prototype (Object Creational) Intent: "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype" GoF(117)



Gives flexible alternatives to inheritance The client creates a prototype and each time it requires a new object, the prototype is asked to clone itself The state of this clone may be that of the current object, or that of the initial object.

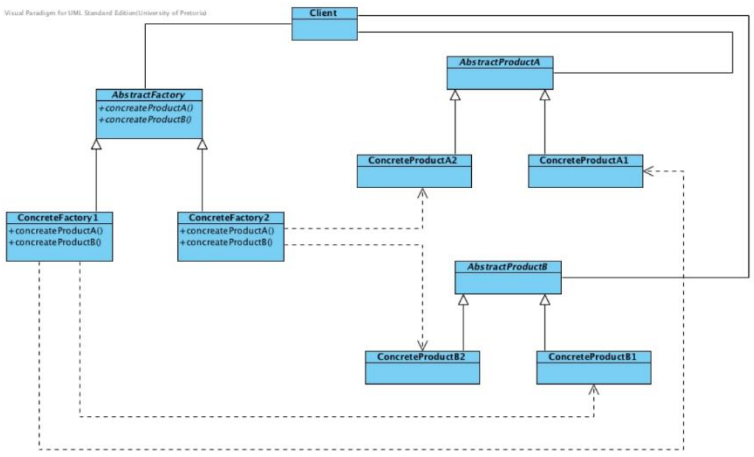
Prototype: defines an interface for cloning

ConcretePrototypeN: implementation of operation for cloning

Client: asks the prototype to clone so that a new object can be created

L7 – Abstract Factory

Name and Classification: Abstract Factory (Object Creational) Intent: "Provide an interface for creating families of related or dependent objects without specifying the concrete classes." GoF(87)



- Makes use of Factory Methods
- **ConcreteFactory**: implements the **AbstractFactory**: interface. Abstract Factory therefore does not directly create product
- **ConcreteFactory**: creates product

AbstractFactory: provides an interface to produce abstract product objects

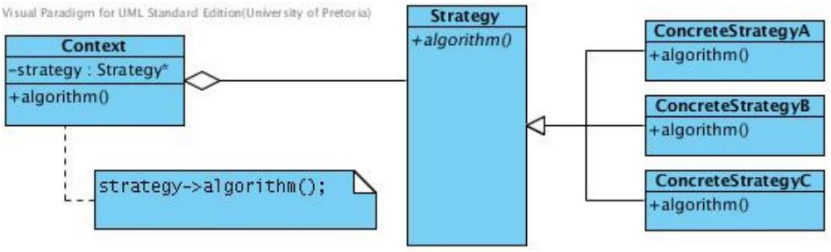
ConcreteFactory: implements the abstract operations to produce concrete product objects

AbstractProduct: provides an interface for product objects

ConcreteProduct: implements the abstract operations that produce product objects that are created by the corresponding ConcreteFactory

L8 – Strategy

Name and Classification: Strategy (Behavioural) Intent: “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” GoF(315)



- Context holds a pointer to a strategy object.
- The strategy object may vary in implementation in terms of the ConcreteStrategy to which is being referred.
- The pattern alleviates the need for a complex conditional to select the desired strategy.

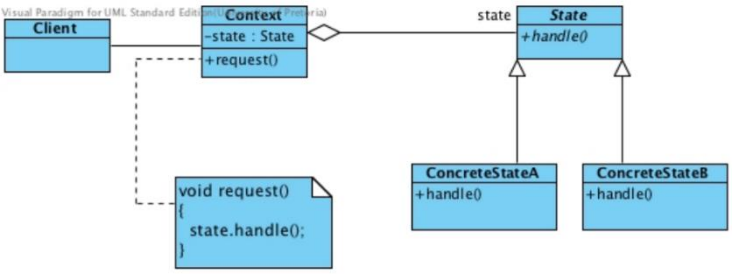
Strategy: Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

Concrete Strategy: Implements the algorithm defined by the Strategy interface.

Context: Is configured with a ConcreteStrategy object. Maintains a reference to a Strategy object. May define an interface that lets Strategy access its data.

L9 – State

Name and Classification: State (Behavioural) Intent: “Allow an object to alter its behaviour when its internal state changes.The object will appear to change its class.” GoF(305)



State: Defines an interface for encapsulating the behaviour associated with a particular state of the Context.

ConcreteState: Implements a behaviour associated with a state of the Context.

Context: Maintains an instance of a ConcreteState subclass that defines the current state. Defines the interface of interest to clients.

L10 – UML State Diagrams

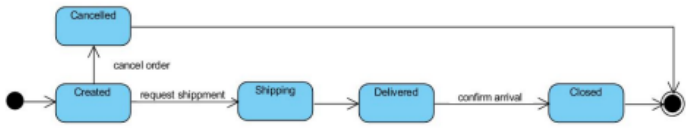
A state is a condition in which an object can be at some point during its lifetime, for some finite period of time. A state diagram is:

- used to model critical states within a system and the events that may trigger a change in state (mostly attribute values).
- good for describing the behaviour of one object over time.

A UML State diagram consists of nodes and edges. Nodes are classified as State nodes or Control nodes. Edges, more commonly referred to as transitions, are labelled arrows connecting the nodes.

Initial Node: ● State node: 

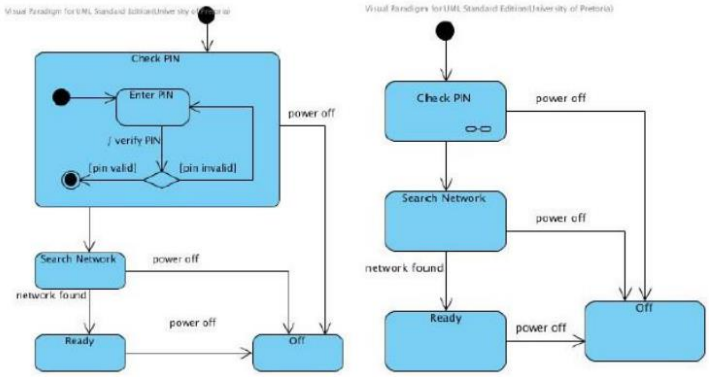
End Node: ◉



Composite State:

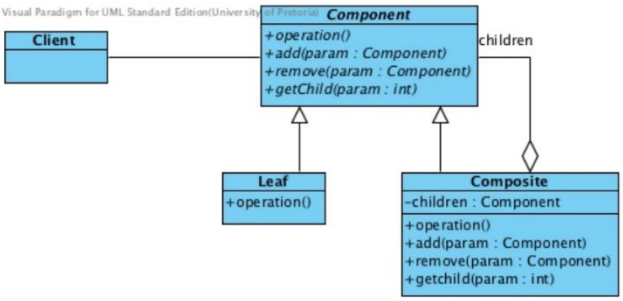
A composite state node is a special type of state node containing another state diagram. composite state nodes may be named, or be left anonymous. If they are anonymous, detailed flow needs to be modelled.

Examples of composite states



L11 – Composite

Name and Classification: Composite (Object Structural) Intent: “Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.” GoF(163)



Component: provides the interface with which the client interacts

Leaf: do not have children, define the primitive objects of the composition

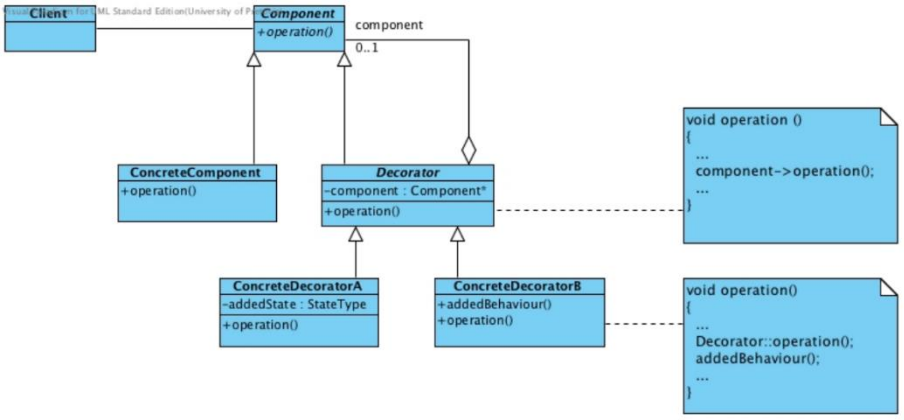
Composite: contain children that are either composites or leaves

Client: manipulates the objects that comprise the composite

- Used in hierarchies where some objects are composites of others
- Makes use of a “structure” for the children defined by Composite

L13 – Decorator

Name and Classification: Decorator (Object Structural) Intent: “Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.” GoF(175)



Looks similar to the Composite. A composite may comprise 0..* or 1..* components, while Decorators comprise of 0..1 component.
Composite themselves do not have specialisations, while Decorators do.

Component: interface for objects that can have responsibilities dynamically added to them.
Concrete Component: the object to which the additional responsibilities can be attached

Decorator: defines a reference to a Component-type object

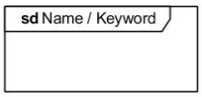
Concrete Decorator: adds the responsibilities to the component

L14 – Sequence

A sequence diagram:

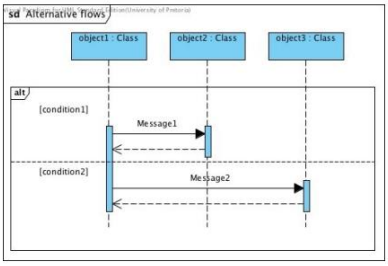
- is used to model how objects interact with one another in terms of the messages (method calls) they pass to one another.
- emphasises the order of message execution as a reaction to some event.
- arranged interactions from top to bottom, following their order of

Sequence diagrams are drawn in *frames* - a rectangle with a heading

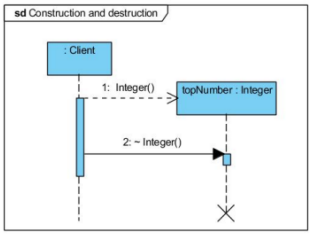


- *Name* - to name the diagram or
- *Keyword* - to indicate the scope of loop structures, conditional statements or parallel flows.

Branching happens when the program flow contains conditional statements.

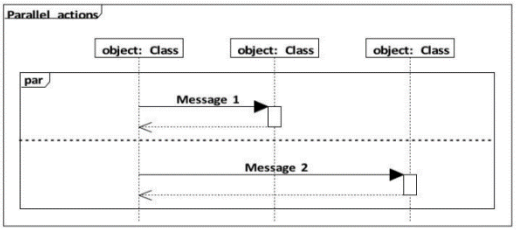


Creation and Destruction



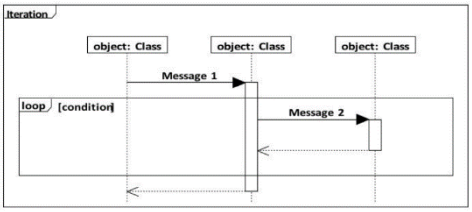

```
Integer* topNumber = new Integer();
delete topNumber;
```

Parallel actions model interactions that are executed at the same time (in parallel).



Syntax for parallel actions

Iteration happens when the program flow contains looping statements.



Syntax for a loop structure