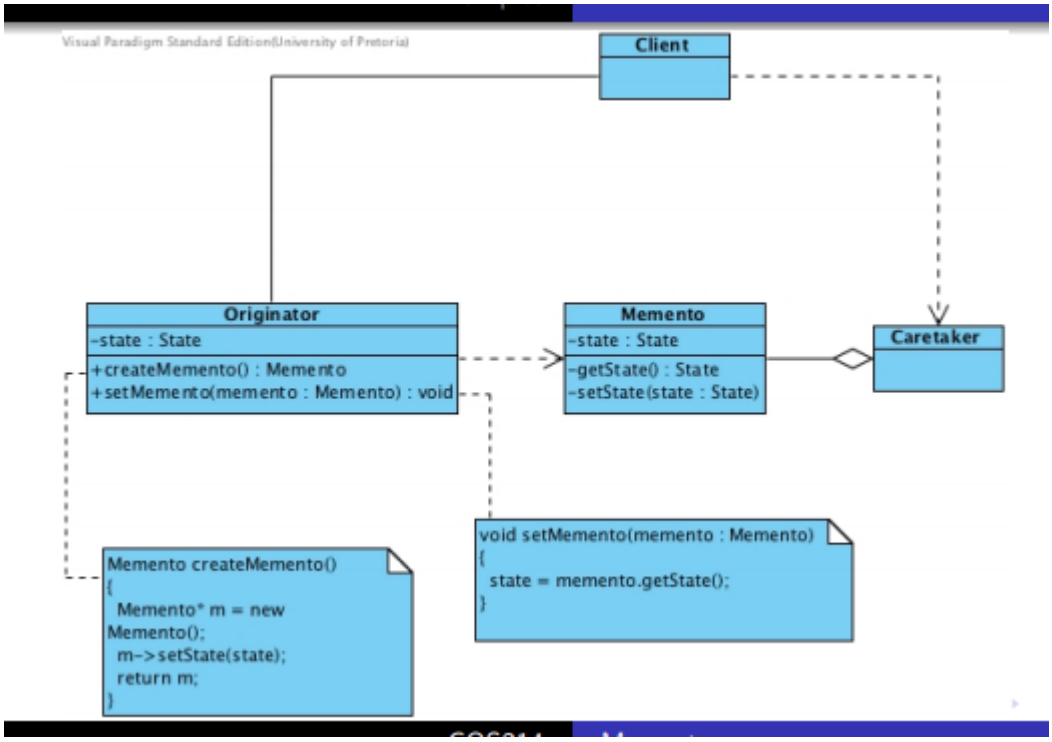
Memento:

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."



Participants:

Memento:

- Stores internal state of the originator object. May store as much or as little of the originator's internal state as necessary, as its originator's discretion.
- Protects against access by objects other than the originator.
    - Caretaker sees a narrow interface to the memento – it can only pass the memento to other objects.
    - Originator sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

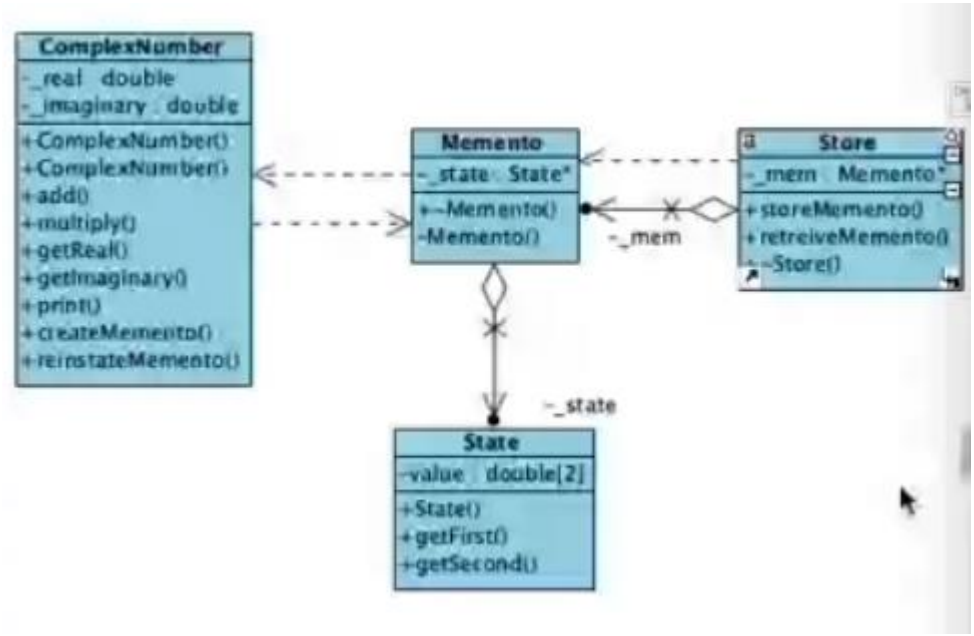
Originator:

- Creates a memento containing a snapshot of its current internal state.
- Uses the memento to restore its internal state.

Caretaker:

- Is responsible for the memento's safekeeping i.e. the state.
- Never operates on or examine the contents of a memento.

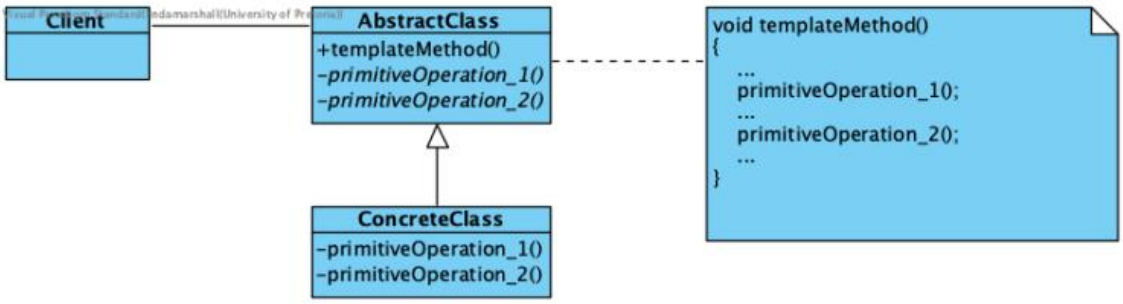
Examples:


ComplexNumber
-_real : double
-_imaginary : double
+ComplexNumber()
+ComplexNumber()
+add()
+multiply()
+getReal()
+getImaginary()
+print()
+createMemento()
+reinstateMemento()
Memento
-_state : State*
+~Memento()
-Memento()
-_mem
Store
-_mem : Memento*
+storeMemento()
+retreiveMemento()
+~Store()
-_state
State
-value : double[2]
+State()
+getFirst()
+getSecond()

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.



Participants:

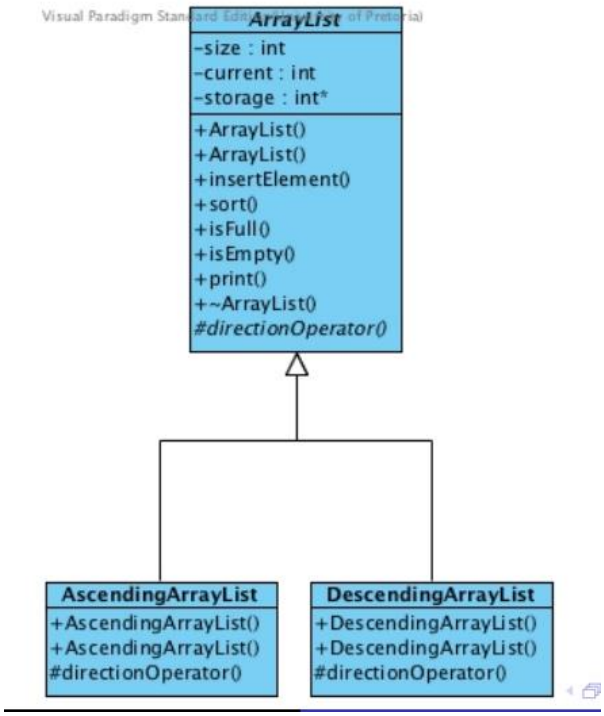
Abstract Class:

- 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 define by the abstract class.

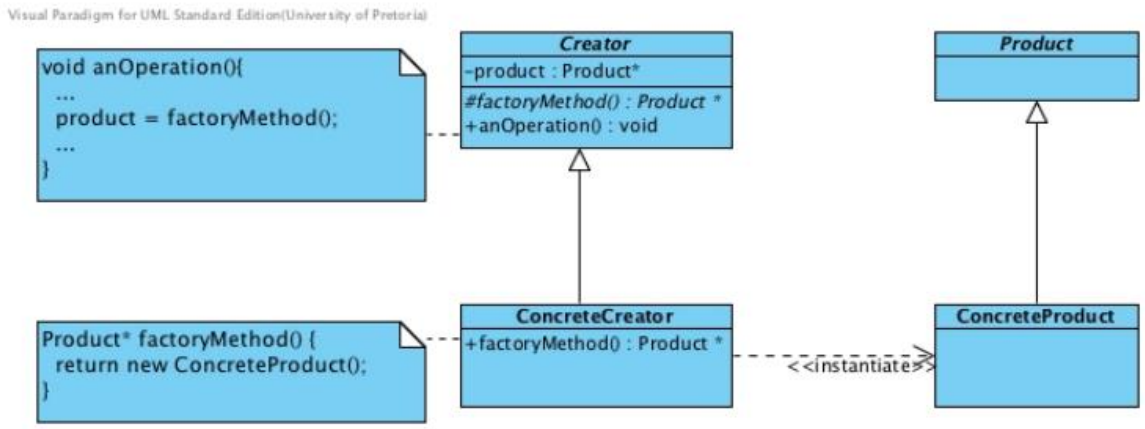
Examples:



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.



- A **Creator** creates **product** which the **client** uses.
- **Product** is always created by **Creator**.
- **Concrete Creators** 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.

Participants:

Product:

- Defines the product interface for the factory method to create.

Concrete Product:

- Implements the interface for the product.

Creator:

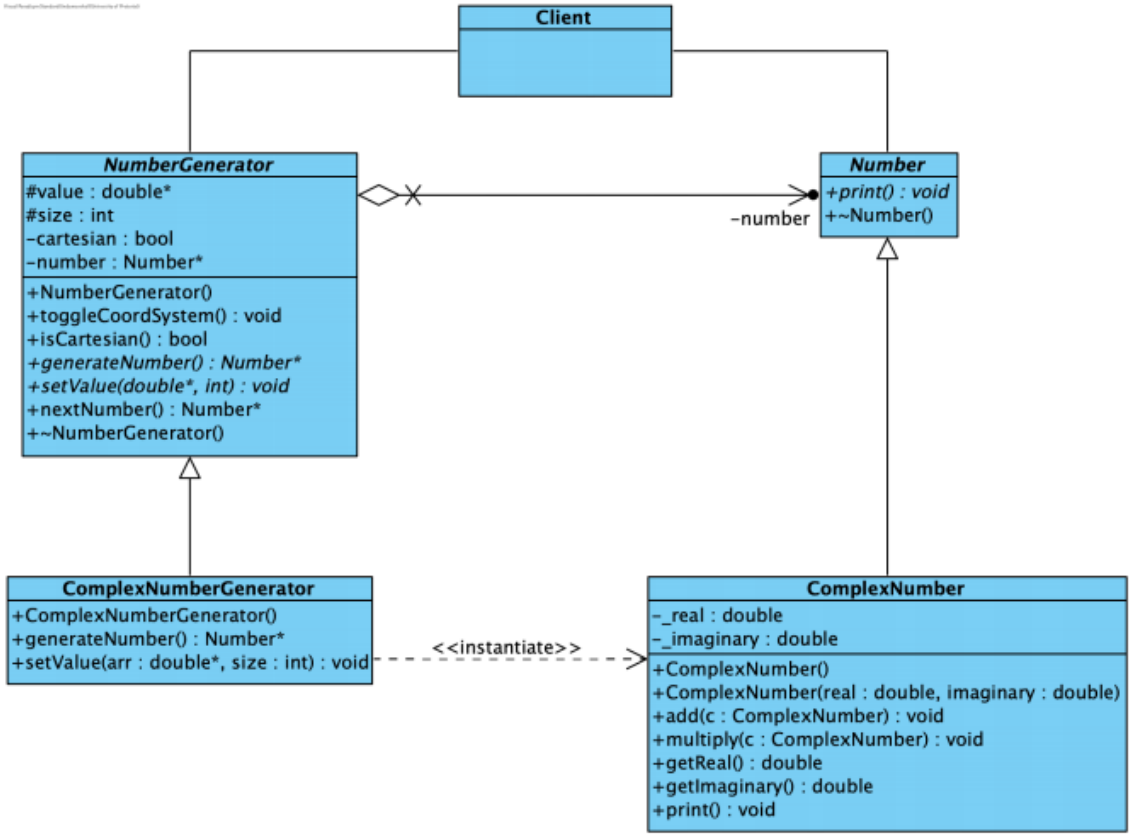
Declares the factory method which returns a product object.

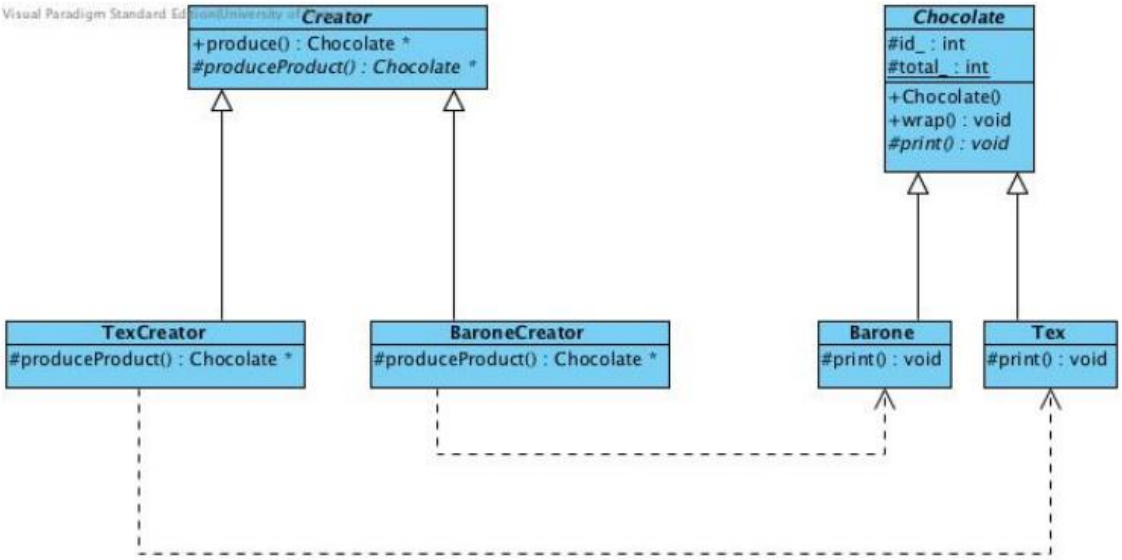
- Default factory method implementations may return a default concrete product.

Concrete Creator:

- Overrides the factory method to return an instance of the product.

Examples:

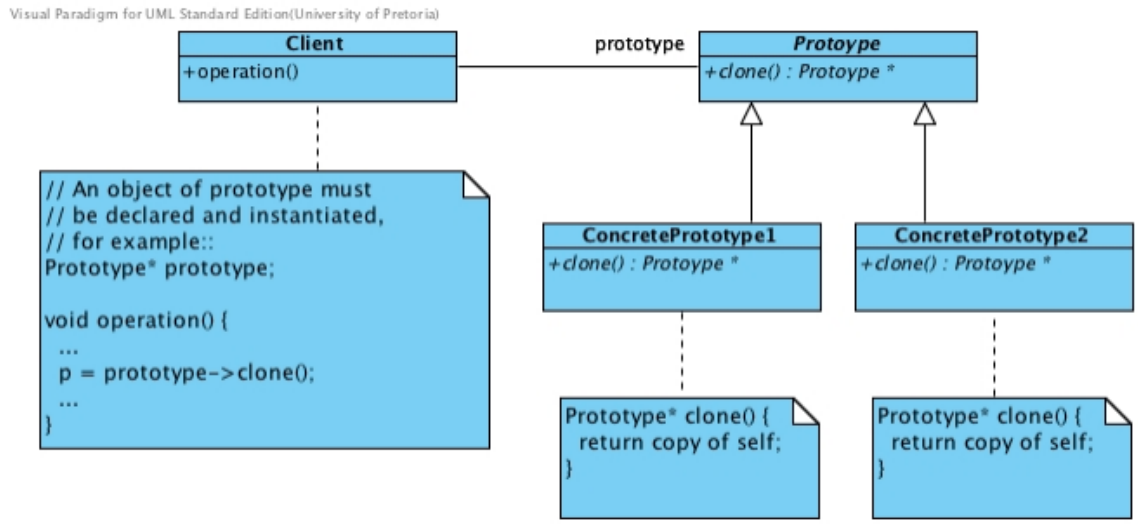

Client
NumberGenerator
#value : double*
#size : int
-cartesian : bool
-number : Number*
+NumberGenerator()
+toggleCoordSystem() : void
+isCartesian() : bool
+generateNumber() : Number*
+setValue(double*, int) : void
+nextNumber() : Number*
+~NumberGenerator()
Number
+print() : void
+~Number()
-number
ComplexNumberGenerator
+ComplexNumberGenerator()
+generateNumber() : Number*
+setValue(arr : double*, size : int) : void
<<instantiate>>
ComplexNumber
-_real : double
-_imaginary : double
+ComplexNumber()
+ComplexNumber(real : double, imaginary : double)
+add(c : ComplexNumber) : void
+multiply(c : ComplexNumber) : void
+getReal() : double
+getImaginary() : double
+print() : void


Creator
+produce() : Chocolate *
#produceProduct() : Chocolate *
Chocolate
#id_ : int
#total_ : int
+Chocolate()
+wrap() : void
#print() : void
TexCreator
#produceProduct() : Chocolate *
BaroneCreator
#produceProduct() : Chocolate *
Barone
#print() : void
Tex
#print() : void

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.



- 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.

Participants:

Prototype:

Defines an interface for cloning.

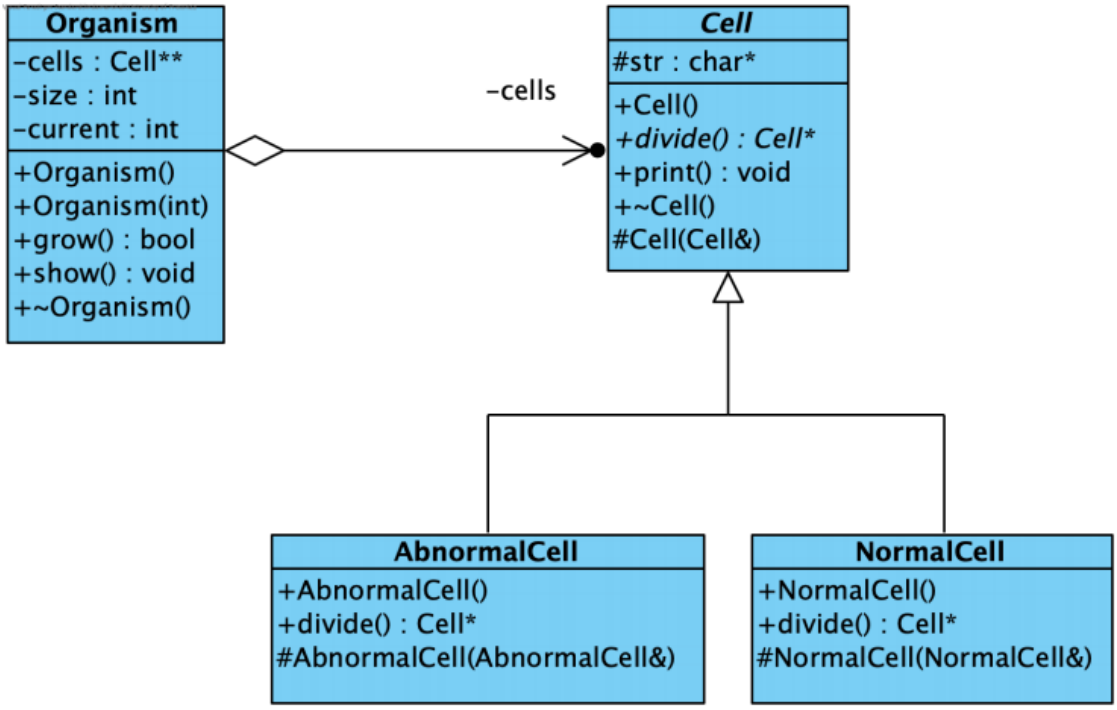
Concrete PrototypeN:

Implementation of operation for cloning.

Client:

Asks the prototype to clone so that a new object can be created.

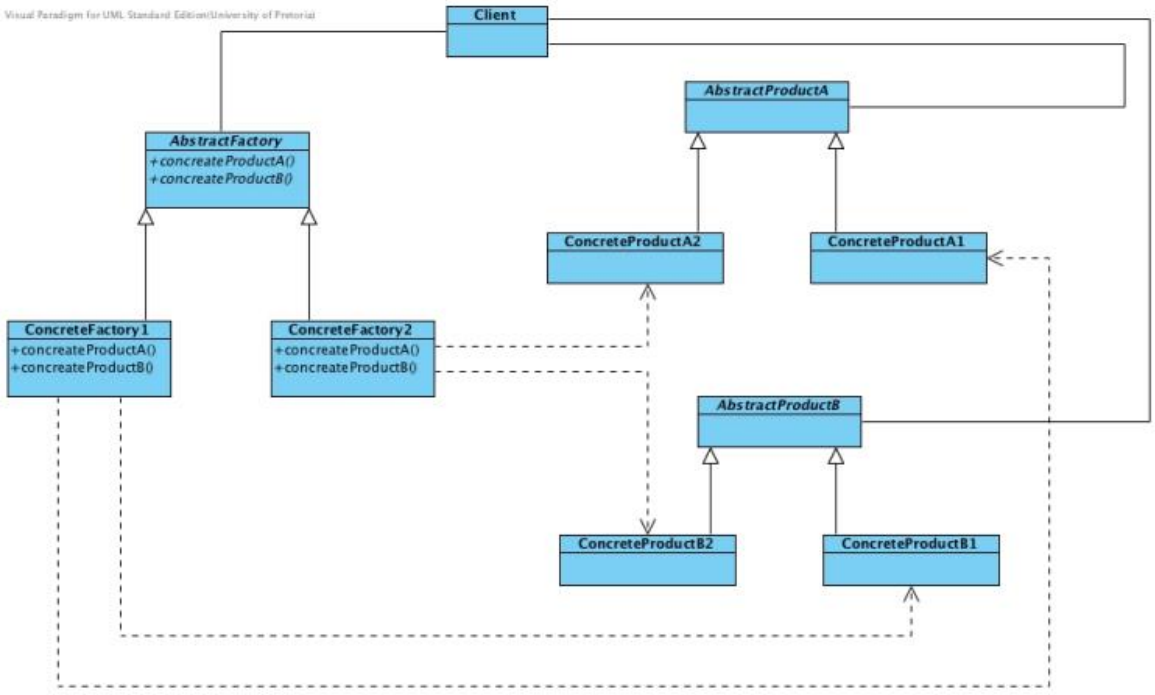
Examples:


Organism
-cells : Cell**
-size : int
-current : int
+Organism()
+Organism(int)
+grow() : bool
+show() : void
+~Organism()
-cells
Cell
#str : char*
+Cell()
+divide() : Cell*
+print() : void
+~Cell()
#Cell(Cell&)
AbnormalCell
+AbnormalCell()
+divide() : Cell*
#AbnormalCell(AbnormalCell&)
NormalCell
+NormalCell()
+divide() : Cell*
#NormalCell(NormalCell&)

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.



- Makes use of **factory methods.**
- **Concrete Factory** implements the **Abstract Factory** interface. **Abstract Factory** therefore **does not** directly create **product**.
- **Concrete factory** creates **product**.

Participants:

AbstractFactory:

- Provides an interface to produce abstract product objects.

ConcreteFactory:

- Implements the abstract operations to produce concrete product objects.

AbstractProduct:

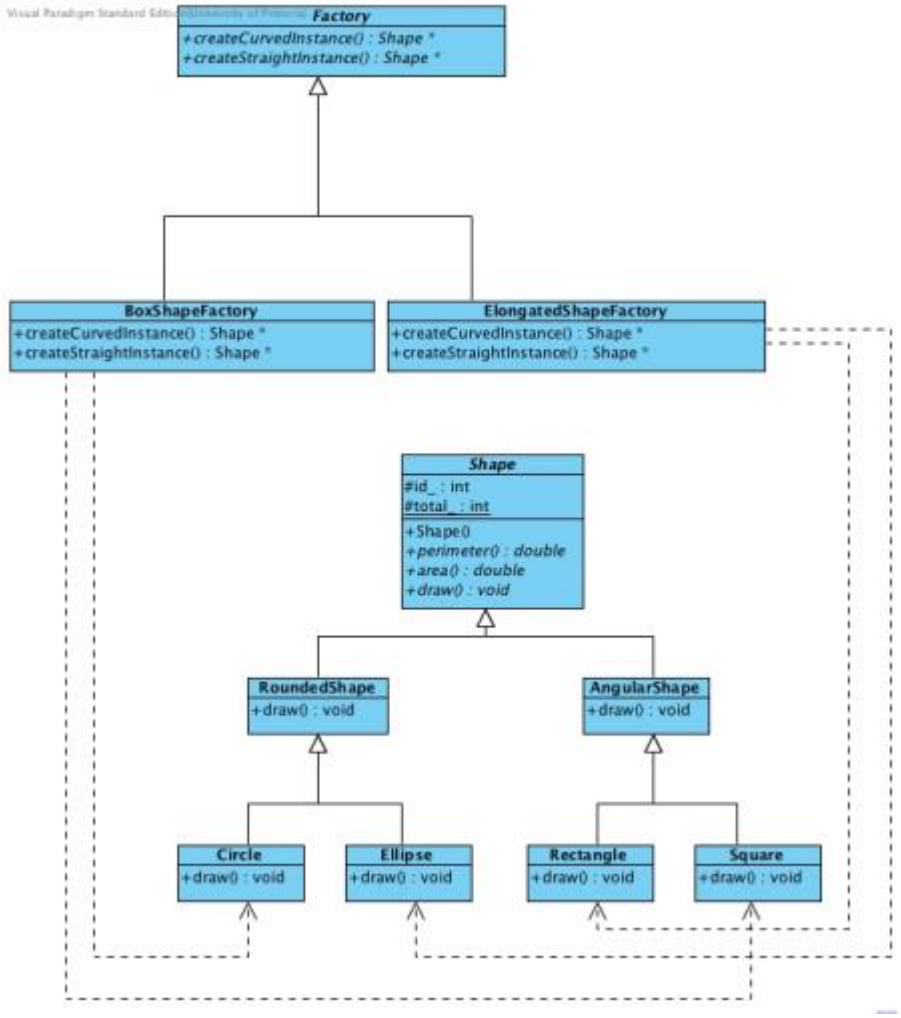
- Provides an interface for product objects.

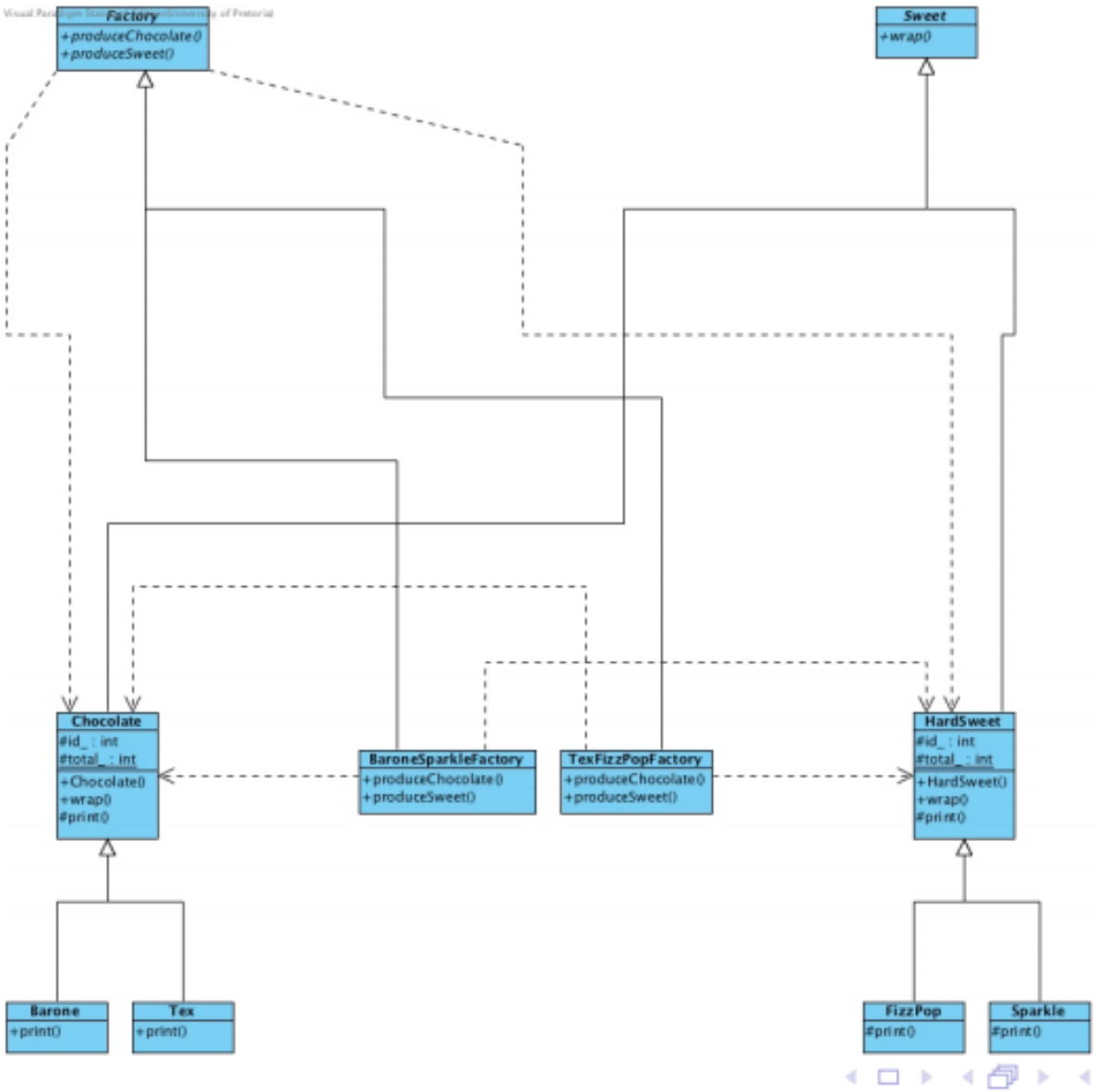
ConcreteProduct:

- Implements the abstract operations that produce objects that are created by the corresponding ConcreteFactory.

Client:

- Uses the interfaces defined by AbstractFactory and AbstractProduct.

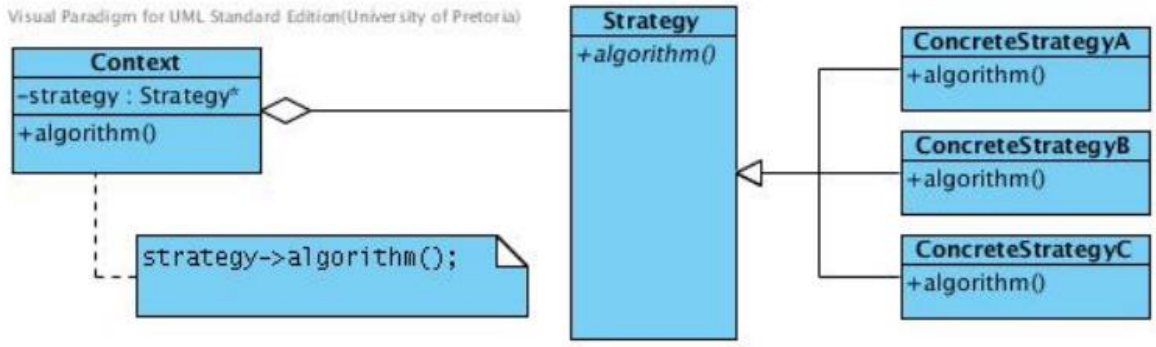



Factory
+produceChocolate()
+produceSweet()
Sweet
+wrap()
Chocolate
#id_ : int
#total_ : int
+Chocolate()
+wrap()
#print()
BaroneSparkleFactory
+produceChocolate()
+produceSweet()
TexFizzPopFactory
+produceChocolate()
+produceSweet()
HardSweet
#id_ : int
#total_ : int
+HardSweet()
+wrap()
#print()
Barone
+print()
Tex
+print()
FizzPop
#print()
Sparkle
#print()

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.



- 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.

Participants:

Strategy

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

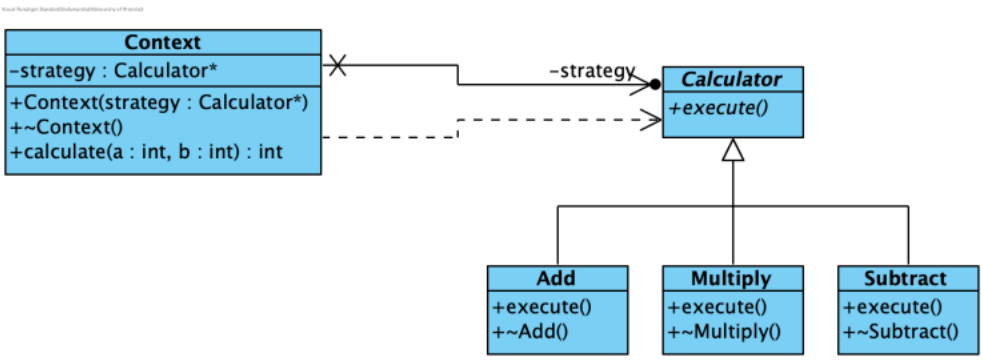
ConcreteStrategy:

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.

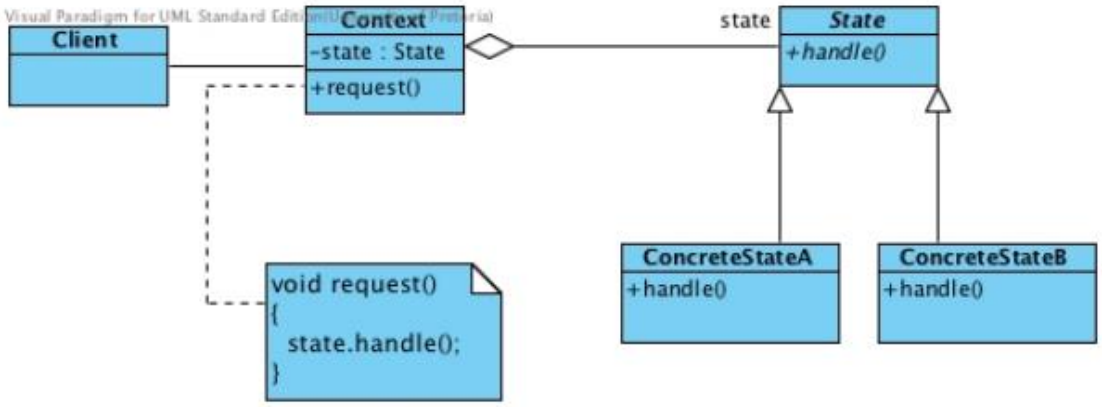
Examples:



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.



Participants:

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.

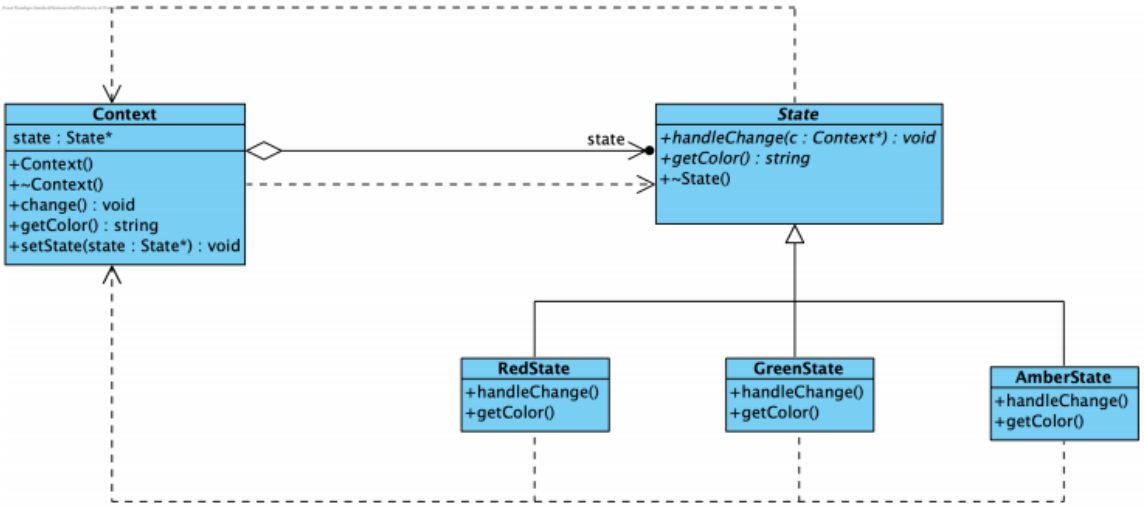
Context:

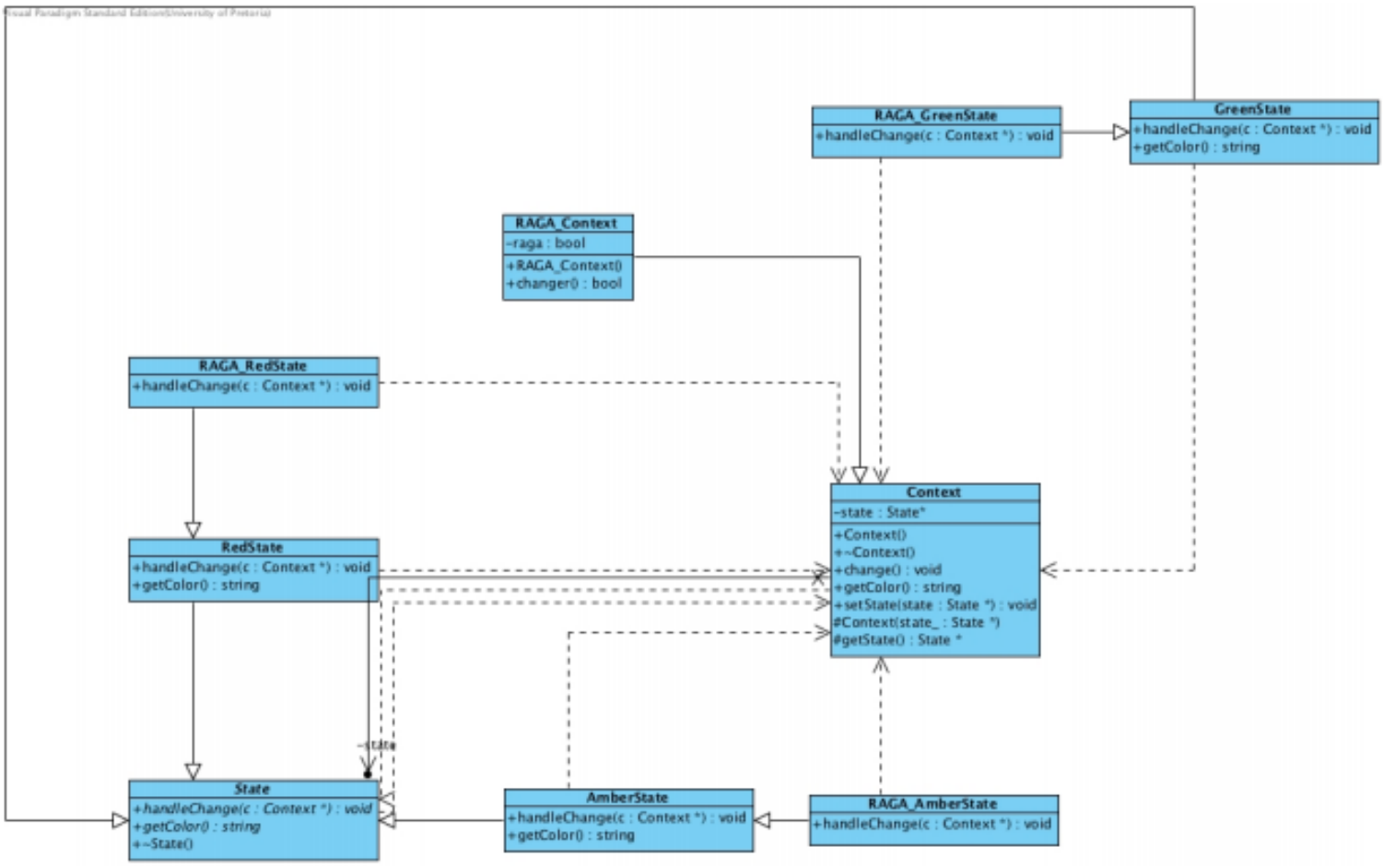
- Maintains an instance of a concrete state subclass that defines the current state.
- Defines the interface of interest to clients.

Effective when:

- An object becomes large - state of the object is managed externally to the object itself
- An object can experience an extensive number of state changes - specifically when flow-control is characterised by multiple if or switch statements. State is managed externally, by modelling each state as an individual object

Examples:


Context
state : State*
+Context()
+~Context()
+change() : void
+getColor() : string
+setState(state : State*) : void
state
State
+handleChange(c : Context*) : void
+getColor() : string
+~State()
RedState
+handleChange()
+getColor()
GreenState
+handleChange()
+getColor()
AmberState
+handleChange()
+getColor()


RAGA_GreenState
+handleChange(c : Context *) : void
GreenState
+handleChange(c : Context *) : void
+getColor() : string
RAGA_Context
-raga : bool
+RAGA_Context()
+changer() : bool
RAGA_RedState
+handleChange(c : Context *) : void
Context
-state : State*
+Context()
+~Context()
+change() : void
+getColor() : string
+setState(state : State *) : void
#Context(state_ : State *)
#getState() : State *
RedState
+handleChange(c : Context *) : void
+getColor() : string
-state
State
+handleChange(c : Context *) : void
+getColor() : string
+~State()
AmberState
+handleChange(c : Context *) : void
+getColor() : string
RAGA_AmberState
+handleChange(c : Context *) : void