# Advanced Programming

*Software Design Patterns*
*Architectural Patterns*

Álvaro Santos@DEIS-ISEC

# Software Design Patterns

FSM – *Finite-State Machine*

# Problem complexity

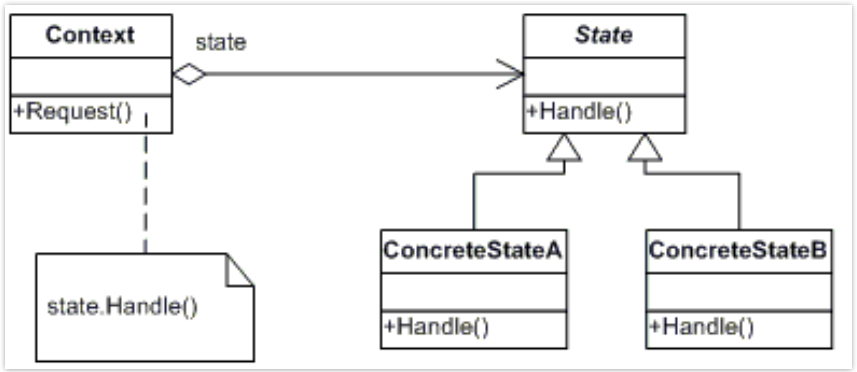
- When a problem to be solved or a program to be developed becomes too complicated, organization and/or complexity reduction techniques are used to enable its resolution

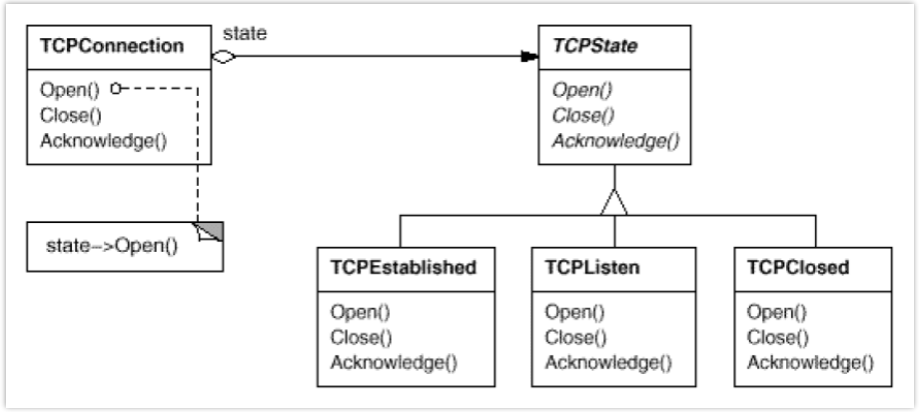
- In problems where it is possible to identify program evolution flows between a succession of well-identified situations, usually referred to as "states", the development pattern known as State Machine is used
  - **FSM – *Finite-State Machine***

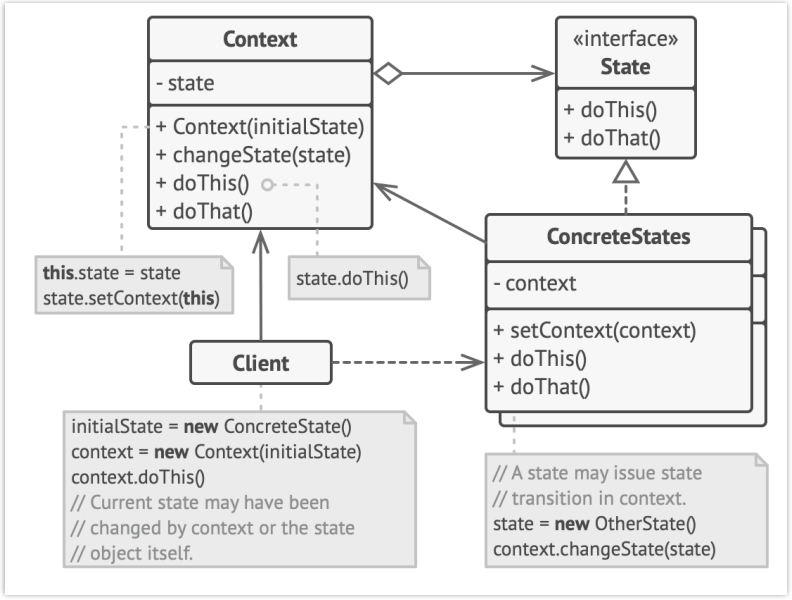
# FSM – *Finite-State Machine*

- Software design pattern used when a particular entity undergoes a process of evolution between successive states, in the context of which it may be subject to events or actions appropriate to each state, allowing it to evolve/adapt behaviors or data itself
  - The entity can be an object, a group of objects, or the entire application
  - In each state of its evolution, there are associated operations, which may be different from those in other states

# State design pattern



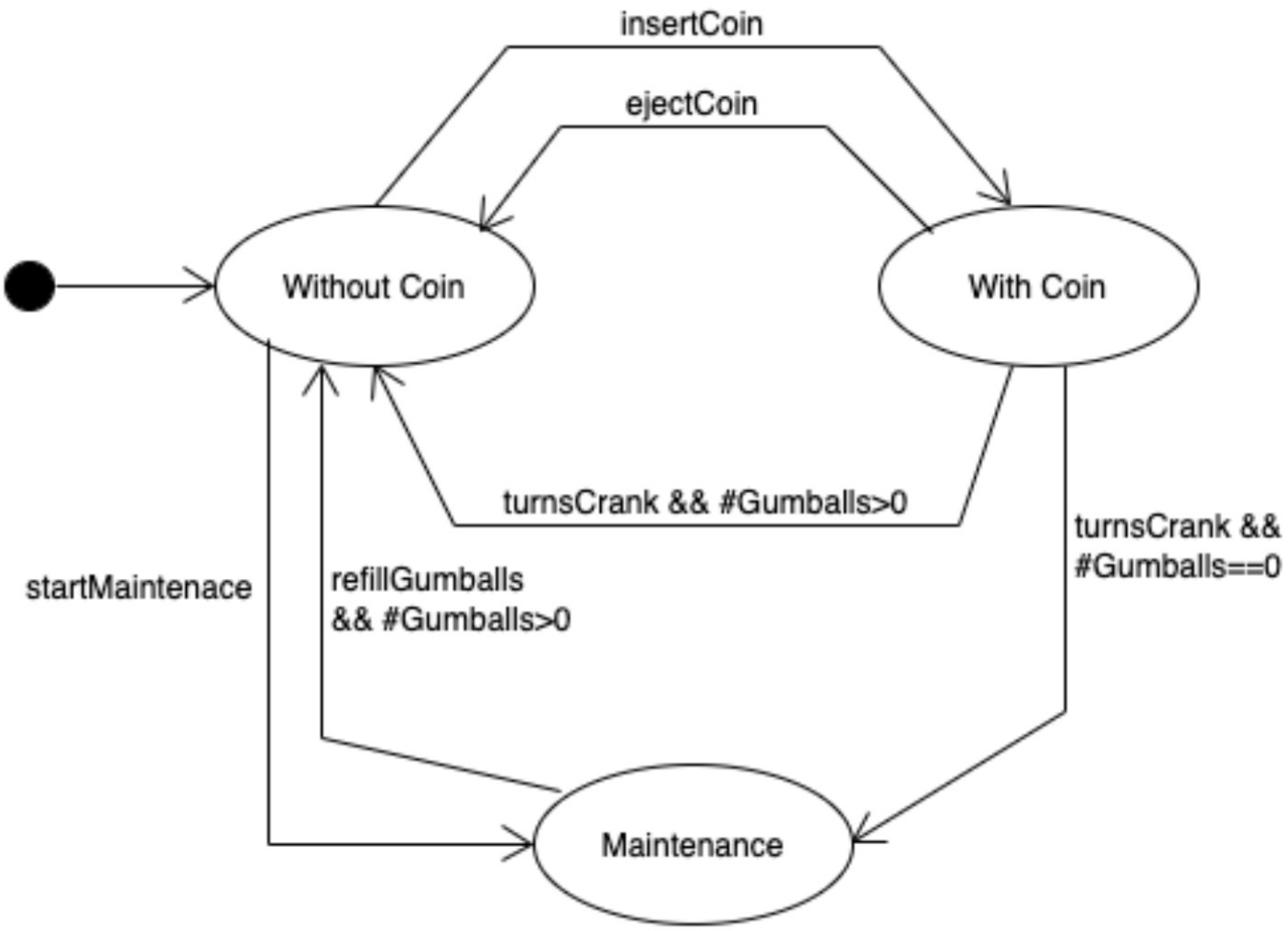



# Example – "Gumball Machine"

- Let's assume that a simulator of a gumball machine is to be implemented...
- The gumball machine dispenses gumballs when a coin is inserted and the crank/handle is turned
- Actions (all actions allowed regardless of state):
  - Insert coin
  - Withdraw/eject coin
  - Turn crank
  - Put in maintenance mode/Start maintenance
  - Refill gumballs
- States
  - Without coin
  - With coin
  - In maintenance mode

# State diagram



# State machine

- The reaction to various events depends on the situation (state) in which the 'gumball machine' finds itself

- The machine's reaction to an event can be:
  - Ignore the event
    - The event may not make sense for the current state
  - Change its state
    - Execute actions, which may trigger a change to a different state

# Creation of the state machine

- After studying the problem to be solved and creating the state diagram...

- Create the classes that support the data model
  - The data model should not depend on the specific way the actions and data transformations will be performed
    - The data model, usually, should be prepared in a generic way to adapt to any behavioral pattern/execution model, not necessarily the state machine pattern
    - All the rules/logic of the model ('business rules') must be ensured

# Creation of the state machine

- Create a set of constants or an `enum` that allows the identification of all states
- Create an interface with methods that represent all transitions
  - This interface should include a method `getState` that allows returning the current state (defined by a constant or an `enum` value)

# Creation of the state machine

- Create an abstract class called *StateAdapter* that allows:
  - Providing default implementations for all transitions
  - Managing references to the base class that represents the data model and to the class that represents the overall context of the state machine
  - Providing a method to change the current state in the context
- Creation of the classes that represent each state (derived from the *StateAdapter* class)

# Creation of the state machine

- Create the *Context* class with:
  - A reference to the current state, which may be initialized in its constructor
  - A reference to the data model
  - A public method that allows obtaining the current state
  - A *package-private* method that allows changing the current state
  - Methods that forward actions/events to the active state
  - A set of methods that allow obtaining the data necessary for interaction with the user or with other modules of the program
- The *Context* class must ensure that changes to the data model only occur within the context of methods related to state transitions
  - It should not provide methods or return references to objects that allow direct modification of the data

# GumballMachineData

```
public class GumballMachineData {
    private int count = 0;

    public GumballMachineData(int count) { this.count = count; }

    public int getCount() { return count; }

    public void refillGumballs(int count) {
        if (count>0)
            this.count += count;
    }

    public boolean getGumball() {
        if (count>0) {
            count--;
            return true;
        }
        return false;
    }

    @Override
    public String toString() {
        return String.format("Gumball Machine with %d gumball(s)",count);
    }
}
```

# IState and enum State

```
public enum State {
    MAINTENANCE, WITH_COIN, WITHOUT_COIN
}
```

---

```
public interface IState {
    boolean insertCoin();
    boolean ejectCoin();
    boolean turnsCrank();
    boolean startMaintenance();
    boolean refillGumballs(int count);

    State getState();
}
```

# StateAdapter

```
abstract class StateAdapter implements IState {
    protected Context context;
    protected GumballMachineData data;

    protected StateAdapter(Context context, GumballMachineData data) {
        this.context = context;
        this.data = data;
    }

    protected void changeState(IState newState) { context.changeState(newState); }

    @Override
    public boolean insertCoin() { return false; }

    @Override
    public boolean ejectCoin() { return false; }

    @Override
    public boolean turnsCrank() { return false; }

    @Override
    public boolean startMaintenance() { return false; }

    @Override
    public boolean refillGumballs(int count)  { return false; }
}
```

# WithoutCoinState

```
class WithoutCoinState extends StateAdapter {
    WithoutCoinState(Context context, GumballMachineData data) {
        super(context,data);
    }

    @Override
    public boolean insertCoin() {
        changeState(new WithCoinState(context,data));
        return true;
    }

    @Override
    public boolean startMaintenance() {
        changeState(new MaintenanceState(context,data));
        return true;
    }

    @Override
    public State getState() { return State.WITHOUT_COIN; }
}
```

# WithCoin

```
class WithCoinState extends StateAdapter {
    WithCoinState(Context context, GumballMachineData data) {
        super(context,data);
    }

    @Override
    public boolean ejectCoin() {
        changeState(new WithoutCoinState(context, data));
        return true;
    }

    @Override
    public boolean turnsCrank() {
        if (data.getGumball() && data.getCount()>0) {
            changeState(new WithoutCoinState(context, data));
            return true;
        }
        changeState(new MaintenanceState(context,data));
        return false;
    }

    @Override
    public State getState() { return State.WITH_COIN; }
}
```

# MaintenanceState

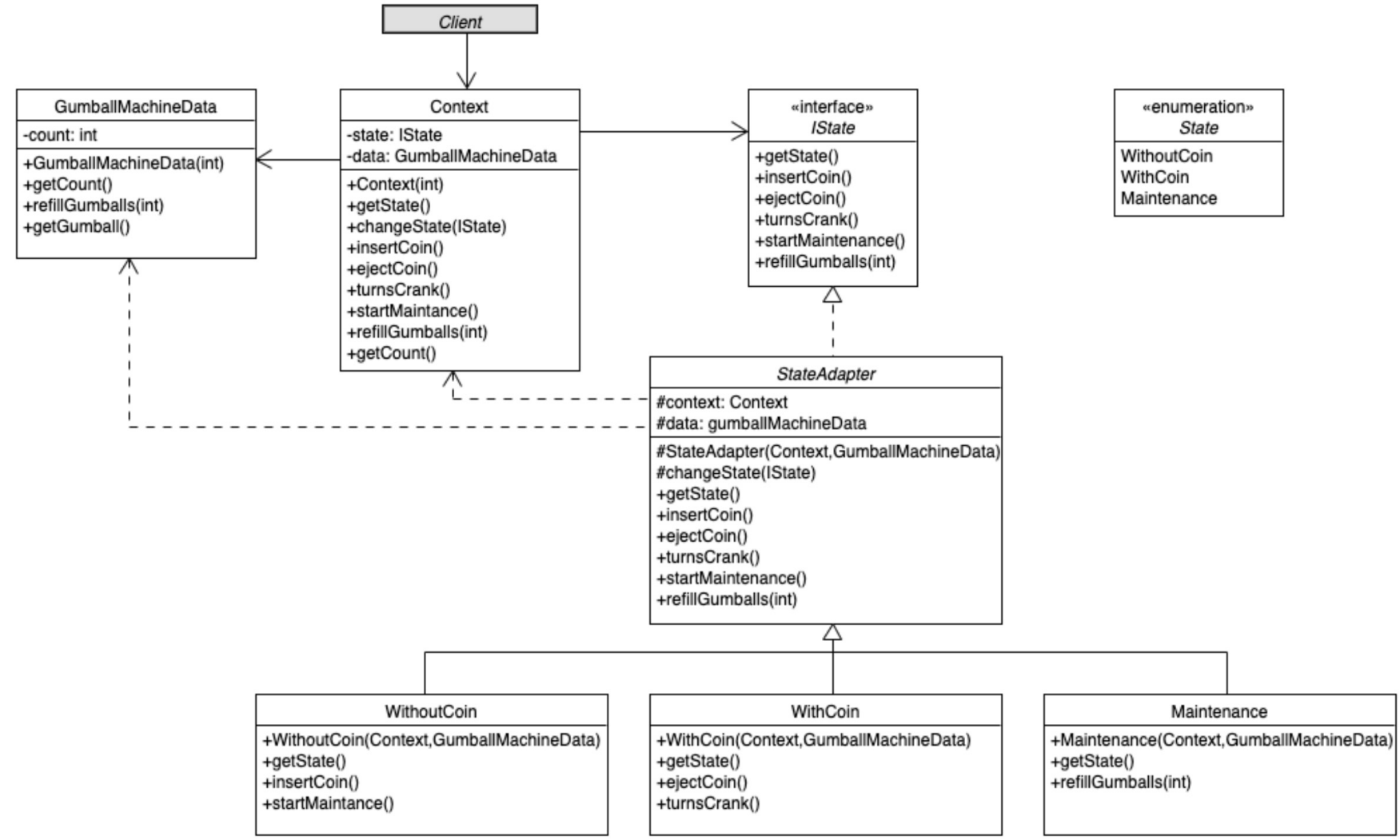
```
class MaintenanceState extends StateAdapter {
    MaintenanceState(Context context, GumballMachineData data) {
        super(context,data);
    }

    @Override
    public boolean refillGumballs(int count) {
        data.refillGumballs(count);
        if (data.getCount()>0) {
            changeState(new WithoutCoinState(context, data));
            return true;
        }
        return false;
    }

    @Override
    public State getState() {
        return State.MAINTENANCE;
    }
}
```

# Context – *State Machine*

```
public class Context {
    private GumballMachineData data;
    private IState state;

    public Context(int count) {
      data = new GumballMachineData(count);
      state = new
WithoutCoinState(this,data);
    }

    public State getState() {
        return state.getState();
    }

    void changeState(IState newState) {
        this.state = newState;
    }

    // ---- get data ----

    public int getCount() {
        return data.getCount();
    }

// ...
```

```
// ...

    // Transitions
    public boolean insertCoin() {
        return state.insertCoin();
    }

    public boolean ejectCoin() {
        return state.ejectCoin();
    }

    public boolean turnsCrank() {
        return state.turnsCrank();
    }

    public boolean startMaintenance() {
        return state.startMaintenance();
    }

    public boolean refillGumballs(int
count) {
        return
state.refillGumballs(count);
    }

}
```

# "Gumball Machine"



# Main

```
public class Main {
    public static void main(String[] args) {
        Context fsm = new Context(100);
        GumballMachineUI ui = new GumballMachineUI(fsm);
        ui.start();
    }
}
```

# GumballMachineUI (1/2)

```
public class GumballMachineUI {
    private Context fsm;
    public GumballMachineUI(Context fsm) { this.fsm = fsm; }

    public boolean start() {
        while (switch (fsm.getState()) {
                  case MAINTENANCE  -> maintenance();
                  case WITH_COIN    -> withCoin();
                  case WITHOUT_COIN -> withoutCoin();
               }) {
            System.out.printf("\nCurrent state: %s\n\n",fsm.getState()); // (only for debug)
        }
        return false;
    }

    public boolean withoutCoin() {
        System.out.printf("\nGumball Machine with %d gumballs\n",fsm.getCount());
        switch (PAInput.chooseOption("Machine without coin","Insert coin","Start maintenance","Stop
machine")) {
            case 1 -> fsm.insertCoin();
            case 2 -> fsm.startMaintance();
            default -> {
                return false;
            }
        }
        return true;
    }   // ========= next page ======>
```

# GumballMachineUI (2/2)

```
    // <=============== previous page =======
    public boolean withCoin() {
        System.out.printf("\nGumball Machine with %d gumballs\n",fsm.getCount());
        switch (PAInput.chooseOption("Machine with a coin","Eject coin","Turns crank","Stop Machine")) {
            case 1 -> fsm.ejectCoin();
            case 2 -> fsm.turnsCrank();
            default -> {
                return false;
            }
        }
        return true;
    }

    public boolean maintenance() {
        switch (PAInput.chooseOption("Maintenance/Machine without gumballs","Refill gumballs","Stop
machine")) {
            case 1 -> {
                int count = PAInput.readInt("Number of Gumballs: ");
                fsm.refillGumballs(count);
            }
            default -> {
                return false;
            }
        }
        return true;
    }
}
```

# Software Design Patterns

*Factory method*

# Factory method pattern

- The *Factory* pattern, also known as the *Factory method*, provides a mechanism (method) that allows the creation of an instance of a class from a given set of possibilities
  - It's typically applied in a context where there's a hierarchy, from which a set of subclasses derive, and this pattern is used as a mechanism to create an instance of one of these subclasses

# Factory method pattern

- Usual implementation:
  - A `static` method is defined
    - The method can be created in several places (choosing the most adequate place for each project). Examples:
      - In the interface or base class of the hierarchy where the types of objects to be created are derived from
      - In a class created for this purpose (for example: *`GumballStateFactory`*)
      - In a Java *`enum`*
  - The method receives as parameters:
    - The necessary data to identify the type of object desired
    - The parameters necessary for its creation
  - The method will return the type of object from the base of the hierarchy that includes all possible types created by the method

# Example (base class/interface)

```
public interface IState {
  boolean insertCoin();
  boolean ejectCoin();
  boolean turnsCrank();
  boolean startMaintenance();
  boolean refillGumballs(int count);

  State getState();

  static IState getInstance(State state,
                            Context context, GumballMachineData data) {
    return switch (state) {
      case MAINTENANCE  -> new MaintenanceState(context, data);
      case WITH_COIN    -> new WithCoinState(context, data);
      case WITHOUT_COIN -> new WithoutCoinState(context, data);
    };
  }
}
```

# Example (independent class)

```
class GumballStateFactory {
  static IState getInstance(State state,
                            Context context, GumballMachineData data) {
    return switch (state) {
      case MAINTENANCE  -> new MaintenanceState(context, data);
      case WITH_COIN    -> new WithCoinState(context, data);
      case WITHOUT_COIN -> new WithoutCoinState(context, data);
    };
  }
}
```

# Example (enum – static)

```
public enum State {
  MAINTENANCE, WITH_COIN, WITHOUT_COIN;

  //Factory: static method
  static IState getInstance(State state,
                            Context context, GumballMachineData data) {
    return switch (state) {
      case MAINTENANCE  -> new MaintenanceState(context, data);
      case WITH_COIN    -> new WithCoinState(context, data);
      case WITHOUT_COIN -> new WithoutCoinState(context, data);
    };
  }
}
```

# Example (enum – instance)

```
public enum State {
    MAINTENANCE, WITH_COIN, WITHOUT_COIN;

    //Factory: instance method (package-private)
    IState getInstance(Context context, GumballMachineData data) {
        return switch (this) {
            case MAINTENANCE  -> new MaintenanceState(context,data);
            case WITH_COIN    -> new WithCoinState(context,data);
            case WITHOUT_COIN -> new WithoutCoinState(context, data);
        };
    }
}
```

# Example (enum – instance *v2*)

```
public enum State {
    MAINTENANCE {
        @Override
        IState getInstance(Context context, GumballMachineData data) {
            return new MaintenanceState(context,data);
        }
    },
    WITH_COIN {
        @Override
        IState getInstance(Context context, GumballMachineData data) {
            return new WithCoinState(context,data);
        }
    },
    WITHOUT_COIN {
        @Override
        IState getInstance(Context context, GumballMachineData data) {
            return new WithoutCoinState(context, data);
        }
    };

    abstract IState getInstance(Context context, GumballMachineData data);
}
```

# Usage (Gumball FSM examples)

- Original

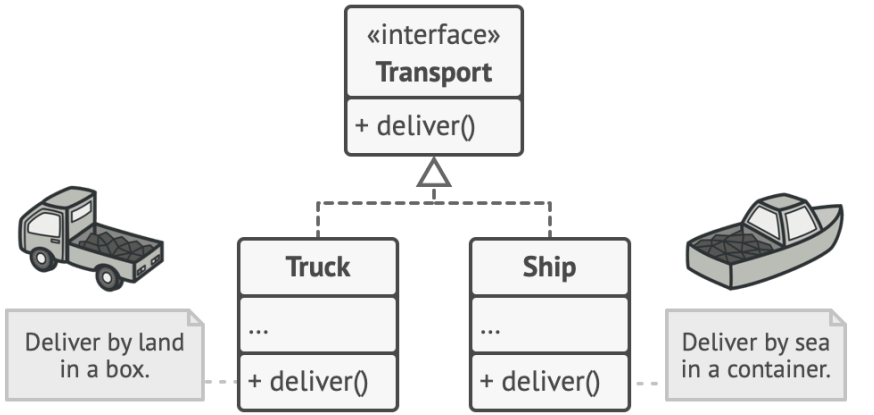
```
protected void changeState(IState newState) {
  context.changeState(newState);
}
```

- Alternatives

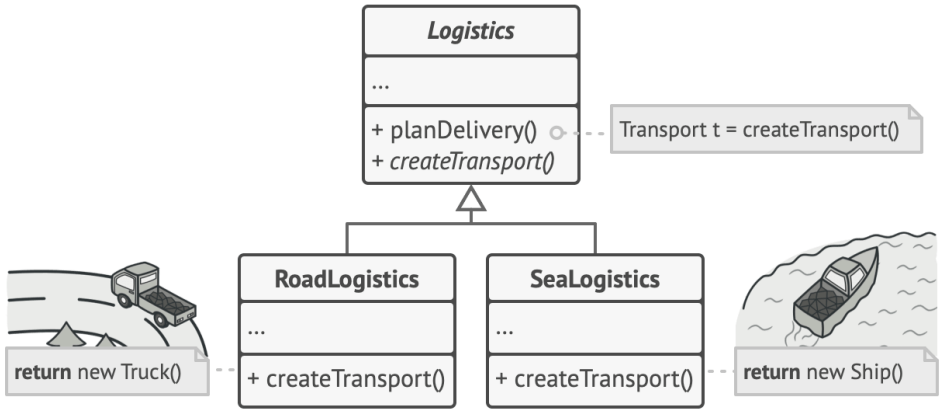
```
protected void changeState(State newState) {
  context.changeState(
    GumballStateFactory.getInstance(newState, context, data)
  );
}

protected void changeState(State newState) {
  context.changeState(
    State.getInstance(newState, context, data)
  );
}

protected void changeState(State newState) {
  context.changeState(newState.getInstance(context, data));
}
```

# Polymorphic Factory

- The pattern can be implemented in other ways, for example, by using instance methods in subclasses to create the object instances
  - This form allows the redefinition of these methods in the derived classes (to include other types of objects in the factory)




*All products must follow the same interface.*




*Subclasses can alter the class of objects being returned by the factory method.*

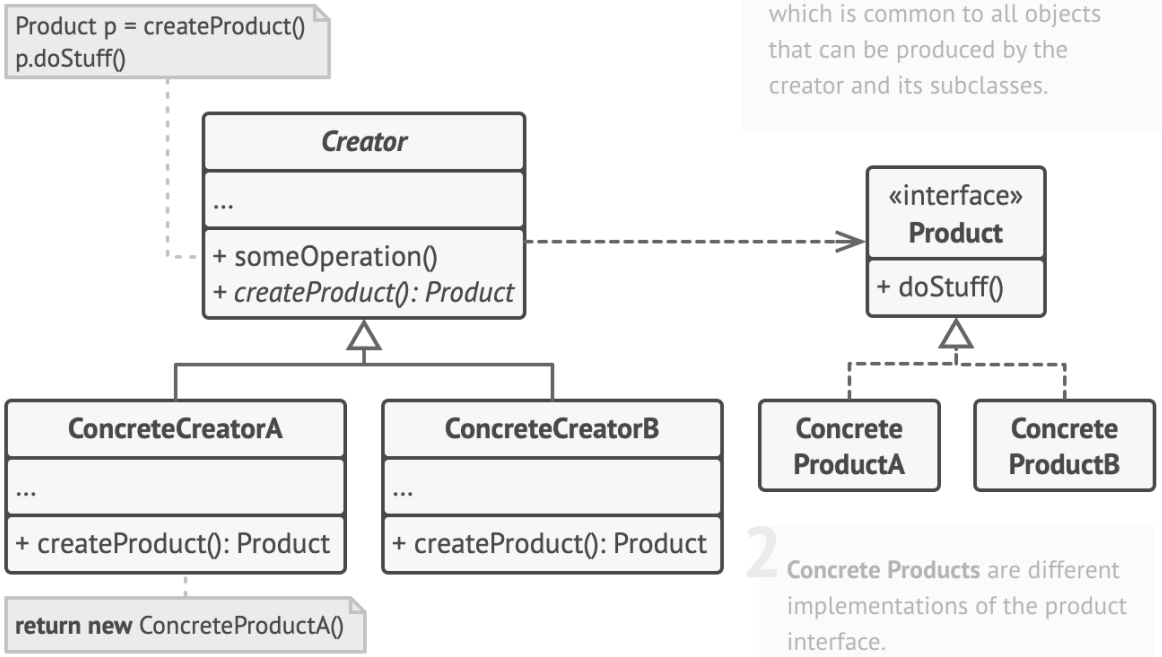
https://refactoring.guru/design-patterns/factory-method

# Polymorphic Factory

3 The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

You can declare the factory method as `abstract` to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.




1 The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2 **Concrete Products** are different implementations of the product interface.

4 **Concrete Creators** override the base factory method so it returns a different type of product.

Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

https://refactoring.guru/design-patterns/factory-method

# Software Design Patterns

*Singleton pattern*

# *Singleton*

- The *Singleton* pattern allows ensuring that only one instance of a particular class exists
- It allows easy access to the object's functionalities without the need to pass the reference to it, in successive calls between functions
- It can be implemented using a class with all static members, usually designated as "*static singleton*"
  - This approach has disadvantages in terms of *Thread safety* and *Data serialization*

# *Singleton*

- Usual implementation:
  - Creating objects using the `new` keyword is denied
    - Private constructor
  - Access to the *singleton* instance will be achieved through a static method (*Factory method*), provided by the class itself
    - Within this method's context, an instance is created if it doesn't already exist, and stored in a static variable
    - The created instance or the previously created one is then returned
- Alternatively, an enumeration can be used to represent the instance

# Singleton

- Usual implementation

```
public class Singleton {
   private static Singleton instance = null;

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }

   private Singleton() { }

   // other variables and methods
   // Example:
   private int count = 0;
   public int getNewID() { return ++count; }
}
```

- Usage: `int i = Singleton.getInstance().getNewID();`

# Singleton (final instance)

- Alternative implementation

```
public class Singleton {
   private static final Singleton instance = new Singleton();

   public static Singleton getInstance() {
      return instance;
   }

   private Singleton() { }

   // other variables and methods
   // Example:
   private int count = 0;
   public int getNewID() { return ++count; }
}
```

- Usage: `int i = Singleton.getInstance().getNewID();`

# Singleton (enum)

- Alternative implementation

```
public enum Singleton {
    INSTANCE;

    // other variables and methods
    // Example:
    private int count = 0;
    public int getNewID() { return count; }
}
```

- Usage examples:

```
int i = Singleton.INSTANCE.getNewID();
int j = INSTANCE.getNewID(); //with a static import
```

# Singleton example - ModelLog

```
public class ModelLog {
    private static ModelLog _instance=null;

    public static ModelLog getInstance() {
        if (_instance == null)
            _instance = new ModelLog();
        return _instance;
    }

    protected ArrayList<String> log;

    private ModelLog() {
        log = new ArrayList<>();
    }

    public void reset() {
        log.clear();
    }

    public void log(String msg) {
        log.add(msg);
    }

    public List<String> getLog() {
        return new ArrayList<>(log);
    }

    //...
}
...
    ModelLog.getInstance().log("Optimization finished");
```

# Software Design Patterns

*Command pattern*

# *Command design pattern*

- The *Command pattern* allows the encapsulation of commands through object classes to be executed on a specific target – the *Command Receiver*
- Each command defines the operations to be executed when invoked through a *Command Manager*
- Additionally, if feasible, an *undo* action to revert the performed operation can be defined
  - If *undo* operations are provided for the commands, the *manager* can provide *undo* and *redo* functionalities

# ICommand

- `ICommand` – Base interface for all commands

```
public interface ICommand {
    boolean execute();
    boolean undo();
}
```

  - The *undo* action may not be justified in certain contexts
    - The `execute` method should return `true` if the operation can be subject to an *undo* action

# AbstractCommand

- `AbstractCommand` (optional) – class that provides common implementations for commands
  - For example, maintaining a reference to the *Command Receiver*

```
abstract class AbstractCommand implements ICommand {
    protected ReceiverClass receiver;

    protected AbstractCommand(ReceiverClass receiver) {
        this.receiver = receiver;
    }
}
```

# ConcreteCommand

- Each class that derives from `AbstractCommand` typically corresponds to a specific command that can be executed and, optionally, undone

```
public class ConcreteCommand extends AbstractCommand {
    private <variables needed to execute and/or undo this command>

    public ConcreteCommand(ReceiverClass receiver, <additional parameters> ) {
        super(receiver);
        // store parameters
    }

    @Override
    public boolean execute() {
        ... receiver.concreteCommand(<execute_parameters>) ...
        return <undo_is_possible>;
    }

    @Override
    public boolean undo() {
        ... receiver.undoConcreteCommand(<undo_parameters>) ...
        return <result>;
    }
}
```

# *Invoker / CommandManager*

- The *Command Manager* will be responsible for...
  - executing the command (calling the `execute` method)
  - managing the command history to enable *undo* operations (if available)
  - managing a list of commands to perform *redo* operations

- Considering the typical operation of *undo* and *redo*, their management can be assisted by `Stack` or `Deque` objects
  - Both allow implementing *Last-In-First-Out* (LIFO) behavior
  - They provide functions for *push* and *pop* operations

# Invoker / CommandManager

```
public class CommandManager {
    private Deque<ICommand> history;
    private Deque<ICommand> redoCmds;
    //private Stack<ICommand> history;
    //private Stack<ICommand> redoCmds;

    public CommandManager() {
        history = new ArrayDeque<>();
        redoCmds = new ArrayDeque<>();
        //history = new Stack<>();
        //redoCmds = new Stack<>();
    }

    public boolean invokeCommand(ICommand cmd) {
        redoCmds.clear();
        if (cmd.execute()) {
            history.push(cmd);
            return true;
        }
        history.clear();
        return false;
    }

// ... => ...
```

```
// ... => ...

    public boolean undo() {
        if (history.isEmpty())
            return false;
        ICommand cmd = history.pop();
        cmd.undo();
        redoCmds.push(cmd);
        return true;
    }

    public boolean redo() {
        if (redoCmds.isEmpty())
            return false;
        ICommand cmd = redoCmds.pop();
        cmd.execute();
        history.push(cmd);
        return true;
    }

    public boolean hasUndo() {
        return history.size()>0;
    }

    public boolean hasRedo() {
        return redoCmds.size()>0;
    }
}
```

# Software Design Patterns

*Memento pattern*

# Memento

- The *Memento design pattern* allows saving the state of an object so that it can be restored later
  - It involves creating *snapshots* of the object's state and managing them
  - By maintaining a history of the *state snapshots*, it becomes possible to implement *undo* and, eventually, *redo* operations

# Memento

- Entities that constitute the *pattern*:
  - *Originator*
    - The object that holds the data/state to be safeguarded
  - *Memento*
    - *Snapshot* of the information/state
    - Should store the information in a way that only the *Originator* can access it
  - *CareTaker*
    - Entity responsible for managing the *mementos*
      - Operations for creating *mementos* are triggered from this entity
    - Maintains a history of the *mementos* and provides undo functionality
      - May have an *undo* history to enable *redo* operations

# Originator

- It should have methods that allow:
  - Saving the current state, returning a *memento* (*snapshot*) with all the information needed to restore it in the future
  - Restoring a previously saved *memento*, restoring the situation as it was at the time the *snapshot* was generated

- The class representing the object should implement the interface `IOriginator`

```
public interface IOriginator {
    IMemento save();
    void restore(IMemento memento);
}
```

# Memento

- Objects that allow representing/storing *snapshots* of a particular object or situation

```
public interface IMemento {
    default Object getSnapshot() { return null; }
}
```

  - The way to store the *snapshot* depends on the *Originator's* information
    - References to the original objects should not be stored
    - Copies of these objects should be stored

# Memento

- The *memento* objects can be implemented in various ways, highlighting two of them:
  - Nested class (private) of the *Originator*
    - This way, the data in the *memento* can be stored in private variables of the nested class, accessible by the *Originator* class but not accessible by external entities
  - Class, possibly external to the *Originator*, that stores the "raw data"
    - Typically using *byte arrays* and the *serialization* process
    - Suitable method when dealing with a more complex data structure in the *Originator*

# Memento with a nested class

```
class MyOriginator implements IOriginator {
   MyObject data;
   // ...
   private static class MyMemento implements IMemento {
      MyObject data;

      MyMemento(MyOriginator base) {
         this.data = base.data.clone(); // or similar
      }
   }

   @Override
   public IMemento save() { return new MyMemento(this); }

   @Override
   public void restore(IMemento memento) {
      if (memento instanceof MyMemento m)
         data = m.data;
   }
}
```

# Memento with serialization

```
class MyOriginator implements
                Serializable, IOriginator {
  MyObject data;
  // ...
  @Override
  public IMemento save() {
    return new Memento(this);
  }

  @Override
  public void restore(IMemento memento) {
    Object obj = memento.getSnapshot();
    if (obj instanceof MyOriginator m)
      data = m.data;
  }
}
```

```
class Memento implements IMemento {
  byte[] snapshot;

  public Memento(Object obj) {
    try (ByteArrayOutputStream baos =
                      new ByteArrayOutputStream();
         ObjectOutputStream oos =
                    new ObjectOutputStream(baos)) {
      oos.writeObject(obj);
      snapshot = baos.toByteArray();
    } catch (Exception e) { snapshot = null; }
  }

  @Override
  public Object getSnapshot() {
    if (snapshot == null) return null;
    try (ByteArrayInputStream bais =
              new ByteArrayInputStream(snapshot);
         ObjectInputStream ois =
                    new ObjectInputStream(bais)) {
      return ois.readObject();
    } catch (Exception e) { return null; }
  }
}
```

# CareTaker

```
public class CareTaker {
    IOriginator originator;
    Deque<IMemento> history;
    Deque<IMemento> redoHist;

    public CareTaker(IOriginator originator) {
        this.originator = originator;
        history = new ArrayDeque<>();
        redoHist= new ArrayDeque<>();
    }

    public void save() {
        redoHist.clear();
        history.push(originator.save());
    }

    public void undo() {
        if (history.isEmpty())
            return;
        redoHist.push(originator.save());
        originator.restore(history.pop());
    }
//...... =>
```

```
// =>.......
    public void redo() {
        if (redoHist.isEmpty())
            return;
        history.push(originator.save());
        originator.restore(redoHist.pop());
    }

    public void reset() {
        history.clear();
        redoHist.clear();
    }

    public boolean hasUndo() {
        return !history.isEmpty();
    }

    public boolean hasRedo() {
        return !redoHist.isEmpty();
    }
}
```

# CareTaker

- Usage:
  - Create an instance of `CareTaker`, associating it with the *Originator*
  - Whenever there is an action that changes the data/state of the *Originator*, the `save()` method of `CareTaker` should be called
  - When an *undo* operation is needed, call the `undo()` method
  - When a *redo* operation is needed, call the `redo()` method
  - If an action is performed that changes the data/state of the *Originator* but for which a *snapshot* is not desired, it is advisable to call the `reset()` method of `CareTaker` to clear the *undo* and *redo* history

# Software Design Patterns

*Facade pattern*

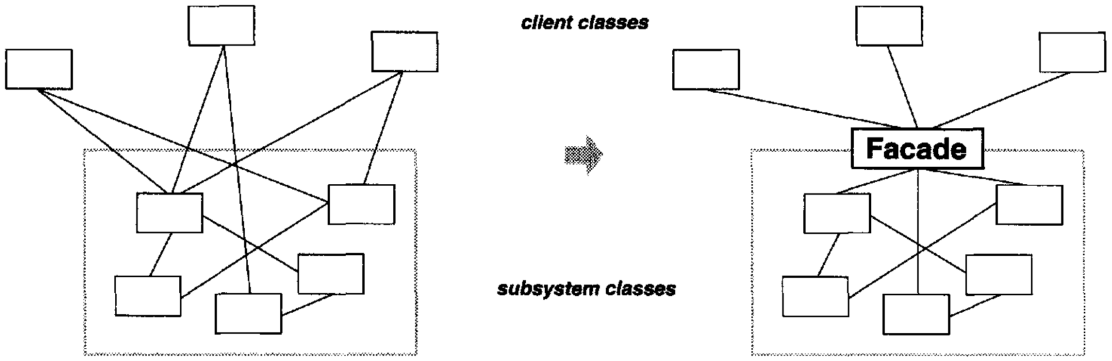
# *Facade*

- The use of the *Command pattern* or the *State pattern*, as well as other patterns or more complex implementations, can and should be hidden from the classes that will benefit from their functionalities, such as user interface classes

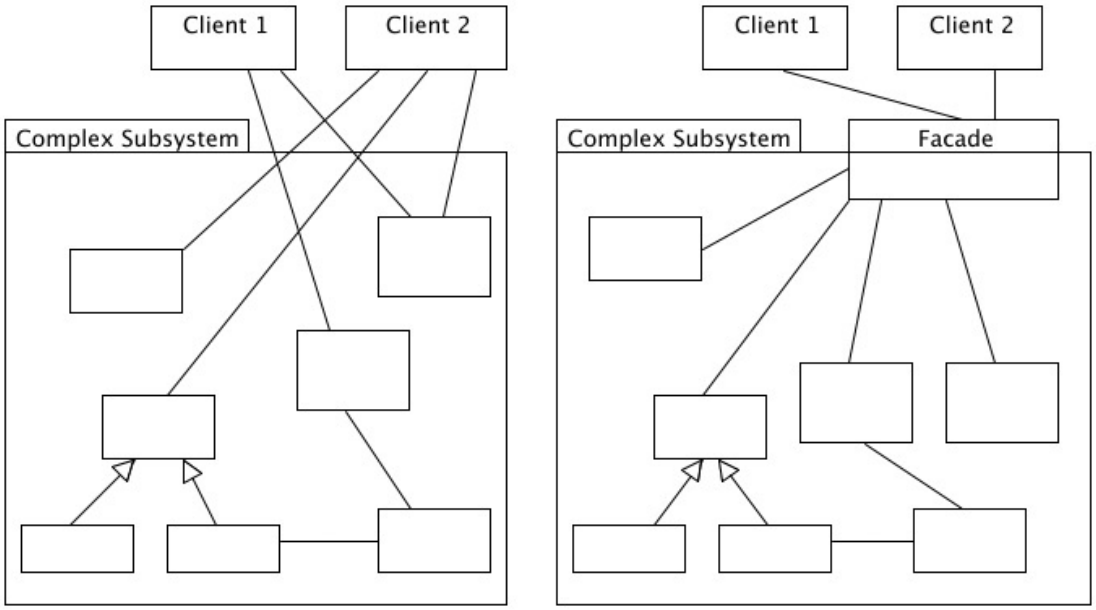
- The implementation details can be hidden by introducing an intermediary class that serves as an access point to the functionalities provided by these systems, classes, or algorithms

- This simplified access interface is called *Facade*

# Facade




Erich Gamma et al., 1995, "Design Patterns", Addison-Wesley




http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/facade.html

# *Facade*

- A *Facade* class should provide simple methods that allow redirecting desired executions to the internal objects of the *packages* (whose visibility is intended to be limited)
  - In some applications, *Facade* classes can be implemented following the *Singleton pattern*

# *Facade* example for the *Command pattern*

```
public class ReceiveManager { // Facade
    ReceiverClass rc;
    CommandManager cm;

    public ReceiveManager() {
        rc = new ReceiverClass();
        cm = new CommandManager();
    }

    public boolean concreteCommand1(<parameters>) {
        return cm.invokeCommand(new ConcreteCommand1(rc,<parameters>));
    }

    public boolean concreteCommand2(<parameters>) {
        return cm.invokeCommand(new ConcreteCommand2(rc,<parameters>));
    }

    // ...

    public boolean hasUndo() { return cm.hasUndo(); }

    public boolean undo()    { return cm.undo();    }

    public boolean hasRedo() { return cm.hasRedo(); }

    public boolean redo()    { return cm.redo();    }

    // ...
}
```

# Software Design Patterns

*Decorator pattern*

# Decorator pattern

- The Decorator pattern is a structural design pattern that allows new behaviors to be attached to objects by encapsulating them within special wrapper objects containing the additional behaviors

- In this pattern, the decorator object typically receives (usually in its constructor) a reference to the object it intends to augment

# Decorator pattern example

```
interface Component {
  void operation();
}


class ConcreteComponent implements Component {
  @Override
  public void operation() {
    System.out.println("Concrete operation");
  }
}


public class Main {
  public static void main(String[] args) {
    Component component = new ConcreteComponent();

    Component decoratedComponent =
                  new ConcreteDecorator(component);

    decoratedComponent.operation();
  }
}
```

```
class Decorator implements Component {
  private Component component;

  public Decorator(Component component) {
    this.component = component;
  }

  @Override
  public void operation() {
    component.operation();
  }
}

class ConcreteDecorator extends Decorator {
  public ConcreteDecorator(Component component) {
    super(component);
  }

  @Override
  public void operation() {
    super.operation();
    addedBehavior();
  }

  private void addedBehavior() {
    System.out.println("Added behavior");
  }
}
```

# Software Design Patterns

*Proxy pattern*

# Proxy pattern

- The Proxy design pattern is a structural approach that provides a substitute or placeholder for an object
- This proxy controls access to the original object, facilitating actions either before or after the request is forwarded to the original object
- This pattern can be used, for example, to access local or remote objects/entities while obscuring this process from their clients

# Proxy pattern example

```
class Data {
    String strData;

    public String getStrData() {
        return strData;
    }

    public void setStrData(String strData) {
        this.strData = strData;
    }
}
```

```
class ProxyData {
    Data target;

    public ProxyData() {
        target = new Data();
    }

    private void preProc(/*...*/) { /* ... */ }
    private void posProc(/*...*/) { /* ... */ }

    public String getStrData() {
        preProc(/*....*/);
        String result = target.getStrData();
        posProc(/*....*/);
        return result;
    }

    public void setStrData(String strData) {
        preProc(/*....*/);
        target.setStrData(strData);
        posProc(/*....*/);
    }

}
```

# Software Design Patterns

*Observer/Observable pattern*

*Publisher/Subscriber pattern*

# Observer pattern

- The *Observer*/*Observable* pattern facilitates the implementation of mechanisms where a group of entities (*Observers*) express interest in the data or notifications emitted by another entity (*Observable*)

- The *Observable* can be an object, a component of the data model, or all the model data itself
  - As model data typically consists of multiple classes, the *Observable* can be represented by a *Facade*, a *Decorator*, or a *Proxy* object

# Publisher/Subscriber

- The *Observer pattern* is also known by various other names, including:
  - *Publisher/Subscriber*
  - *Event-subscriber*
  - *Listener*

- Using these names, the *Observable* is the *Publisher*, and the entities observing changes are the *Subscribers*

- This terminology is particularly fitting in scenarios where:
  - The *Publisher* manages publications that can originate from third-party entities (in this context the *Publisher* is often referred to as '*Broker*')
  - The *Publisher* can disseminate multiple types of information, known as *'topics*', allowing *Subscribers* to selectively subscribe to the relevant *'topics*' based on their interests

# Working mode

- The *Observable*/*Publisher* maintains a list of *Observers*/*Subscribers*
  - Allows operations to add *Observer*/*Subscriber*, as well as remove/unsubscribe
  - Each *Observer*/*Subscriber* implements a common interface that includes a function to be invoked when a topic/announcement is activated – the *listener* function
- When the *Observable*/*Publisher* detects or wishes to notify the *Observers*/*Subscribers*, it iterates through the list to call the *listener* function
  - In cases where the information to disseminate is related to a specific topic, the *Observable*/*Publisher* notifies only the interested *Observers*/*Subscribers*

# *Observer/Observable*

- In the `java.util` *package* an implementation of the *Observer/Observable pattern* is provided, however it is marked as *deprecated*

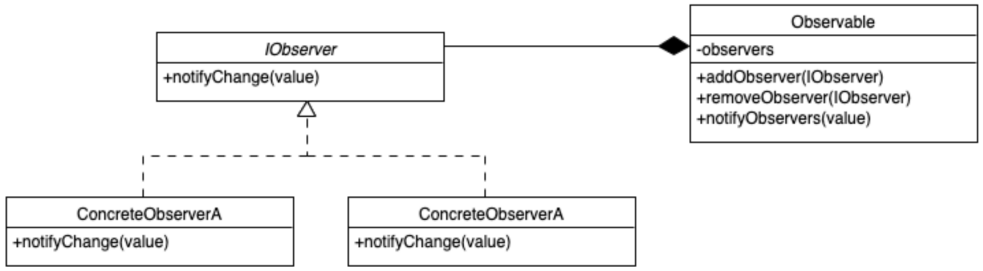
# Observer/Observable

```
interface IObserver {
    void notifyChange(Object value);
}

class A implements IObserver {
    @Override
    public void notifyChange(Object value) {
        System.out.println("A: "+value);
    }
}

class B implements IObserver {
    @Override
    public void notifyChange(Object value) {
        System.out.println("B: "+value);
    }
}
```

```
class Observable {
    HashSet<IObserver> observers = new HashSet<>();

    public void addObserver(IObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(IObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers(Object value) {
        for(IObserver observer : observers)
            observer.notifyChange(value);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Observable observable = new Observable();
        A a = new A();
        B b = new B();
        observable.addObserver(a);
        observable.addObserver(b);
        observable.notifyObservers("DEIS-ISEC");
    }
}
```

```
Output:
A: DEIS-ISEC
B: DEIS-ISEC
```



# *Property Change*

- In the `java.beans` *package* a set of classes is provided that allow implementing the *Observer pattern*'s functionalities in a more robust way and with more functionalities, called *Property Change*

- Main classes
  - `PropertyChangeSupport`
  - `PropertyChangeListener`
  - `PropertyChangeEvent`

# Property Change

- In the `java.beans` *package* a set of classes is provided that allow implementing the *Observer pattern*'s functionalities in a more robust way and with more functionalities, called *Property Change*

- Main classes
  - `PropertyChangeSupport`
    - Manages the target where there will be changes to the data that must be signaled
    - Manages the set of *listeners* (similar to *Observers*), who manifest interest in changes
    - Interests can be manifested in specific properties (defined by a name) or of general interest (no name is indicated - `null`)
    - Allows to signal a change that has occurred:
      - `firePropertyChange(<property_or_null>, <old_value>, <new_value>)`
  - `PropertyChangeListener`
  - `PropertyChangeEvent`

# *Property Change*

- In the `java.beans` *package* a set of classes is provided that allow implementing the *Observer pattern*'s functionalities in a more robust way and with more functionalities, called *Property Change*

- Main classes
  - `PropertyChangeSupport`
  - `PropertyChangeListener`
    - Corresponds to the object that manifests interest in the changes that may occur
  - `PropertyChangeEvent`

# *Property Change*

- In the `java.beans` *package* a set of classes is provided that allow implementing the *Observer pattern*'s functionalities in a more robust way and with more functionalities, called *Property Change*

- Main classes
  - `PropertyChangeSupport`
  - `PropertyChangeListener`
  - `PropertyChangeEvent`
    - Object that represents the notification about a change that has happened
    - Allows obtaining different values, such as new property value and previous value

# Example with *Property Change*

```
class MyModel {
  String data;

  public String getData() {
    return data;
  }

  public void setData(String data) {
    this.data = data;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    ModelManager mm = new ModelManager();

    mm.addPropertyChangeListener(ModelManager.TYPE1, evt-> {
      System.out.println("L1: "+evt.getNewValue());
    });

    mm.addPropertyChangeListener(ModelManager.TYPE2, evt-> {
      System.out.println("L2: "+evt.getNewValue());
    });

    mm.addPropertyChangeListener(evt -> {
      System.out.println("L3: "+evt.getNewValue());
    });

    mm.setData1("DEIS-ISEC");
    mm.setData2("ISEC-DEIS");
  }
}
```

```
Output:
L3: DEIS-ISEC
L1: DEIS-ISEC
L3: ISEC-DEIS
L2: ISEC-DEIS
```

```
class ModelManager {
  public static final String TYPE1 = "type1";
  public static final String TYPE2 = "type2";
  MyModel myModel;
  PropertyChangeSupport pcs;

  public ModelManager() {
    myModel = new MyModel();
    pcs = new PropertyChangeSupport(this);
  }
  public void addPropertyChangeListener(
                  String prop,
                  PropertyChangeListener listener) {
    pcs.addPropertyChangeListener(prop,listener);
  }
  public void addPropertyChangeListener(
                  PropertyChangeListener listener) {
    pcs.addPropertyChangeListener(listener);
  }
  public String getData(){return myModel.getData();}
  public void setData1(String value) {
    myModel.setData(value);
    pcs.firePropertyChange(TYPE1,null,value);
  }
  public void setData2(String value) {
    myModel.setData(value);
    pcs.firePropertyChange(TYPE2,null,value);
  }
}
```

# Software Design Patterns

*Multiton pattern*

# Multiton pattern

- *Multiton* is a design pattern that extends the *Singleton pattern,* enabling the centralized management of multiple instances identified based on a specified element

```
public class ModelMultiton {
    private ModelMultiton() {}

    private static final HashMap<Object,ModelData> models = new HashMap<>();

    public static ModelData getModel(Object scope) {
        ModelData model = models.get(scope);
        if (model == null) {
            model = new ModelData();
            models.put(scope, model);
        }
        return model;
    }
}
```

- For instance, the reference to the `Stage` can be used as a parameter for the `getModel` method, serving as the `HashMap` key
  - This enables the processing of multiple documents or models within the same *JavaFX* application, each document associated with its own `Stage` (*JavaFX Window*)

# Architectural Patterns

MVC

MVP

MVVM

# Project organization

- In order to maintain independence between the data model and the forms of visualization and/or interaction with the user, it is necessary to organize the entities that constitute the programs

- This separation/organization favors other aspects of development, for example:
  - Division of program complexity
  - Evolution of the different parts of the application independently
  - Allocation of different tasks to development teams or members
  - Code reuse
  - Application testing
  - Easy support
  - ...

# Architectural patterns

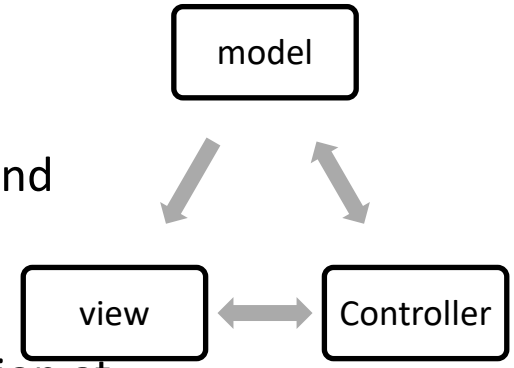
- Architectural patterns define the overall structure and organization of software systems

- They address higher-level concerns such as
  - how different components of a system interact
  - how data flows through the system
  - how responsibilities are distributed among the components

- Architectural patterns help in designing systems that are scalable, modular, and maintainable.
  - Examples of architectural patterns include:
    - *Model-View-Controller* (MVC)
    - *Model-View-Presenter* (MVP)
    - *Model-View-ViewModel* (MVVM)

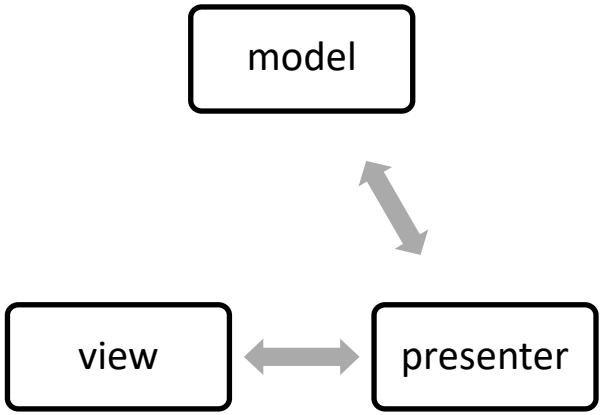
# *Model-View-Controller* (MVC)

- In the MVC pattern responsibilities are divided between:
  - Model
    - Set of classes that manages data, business logic/rules and respective algorithms
  - View
    - Classes and components that display relevant information at all times
    - Presents the necessary elements for the user to interact with the application
  - Controller
    - Performs the appropriate model tasks based on the user commands
    - Ensures that views are updated to reflect the most current information



# *Model-View-Presenter* (MVP)

- The MVP pattern is based on MVC, however …
  - All exchanges of information between the model and the view must pass through the *Presenter*
  - Usually there is a *one-to-one* relationship between *View* and *Presenter*



# *Model-View-ViewModel* (MVVM)

- The MVVM pattern is built on the previous ones where:
  - A layer is created between the view and the model, called *View-Model*, which makes it possible to provide essential information for the view
    - This intermediate layer serves as a model adapted to the needs of the view
    - User actions on the view are triggered on the *View-Model*, which performs the necessary operations on the model
  - In this pattern, the communication from *View-Model* to *View* is based on the *Observer/Observable pattern,* allowing changes be made to the views asynchronously and more transparently