Design Patterns

SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#python-advanced

Table of Contents

Definition, Solutions and Elements

Design Patterns

What are Design Patterns?

- **General** and **reusable solutions** to common problems in software design
- A **pattern** for solving given problems
- Add additional layers of **abstraction** in order to reach flexibility

What do Design Patterns Solve?

- Patterns solve **software structural problems** like
  - Abstraction
  - Encapsulation
  - Separation of concerns
  - Coupling and cohesion
  - Separation of interface and implementation

Elements of a Design Pattern

- Pattern name
  - Increases **vocabulary** of designers
- Problem
  - **Intent**, context, and when to apply
- Solution
  - **Abstract** code
- Consequences
  - **Results** and trade-offs

Benefits and Drawbacks

Why Design Patterns?

Benefits

- Names form a common vocabulary
- Enable large-scale **reuse** of software architectures
- Help improve developer **communication**
- Can **speed up** the development

Drawbacks

- Do not lead to a direct code reuse
- Deceptively simple
- Developers may suffer from **pattern overload** and **overdesign**
- Validated by **experience** and discussion, not by automated testing
- Should be used only if **understood well**

Types of Design Patterns

Main Types

- Creational patterns
  - Deal with **initialization and configuration** of classes and objects
- Structural patterns
  - Describe ways to **assemble** objects to implement **new functionality**
  - **Composition** of classes and objects
- Behavioral patterns
  - Deal with dynamic **interactions** among societies of classes
  - Distribute **responsibility**

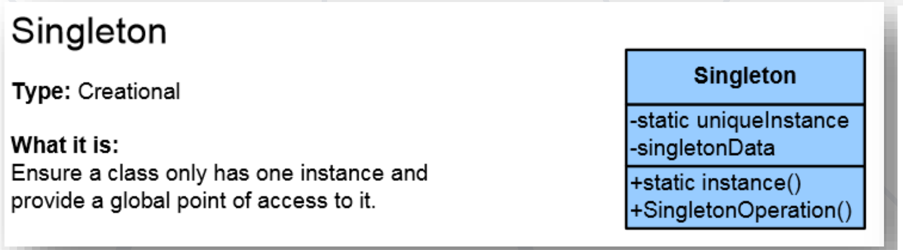
Creational Patterns

Purposes

- Deal with **object creation** mechanisms
- Trying to create objects in a **manner suitable** to the **situation**
- Two main ideas
  - **Encapsulating** knowledge about which classes the system uses
  - **Hiding** how instances of these classes are created

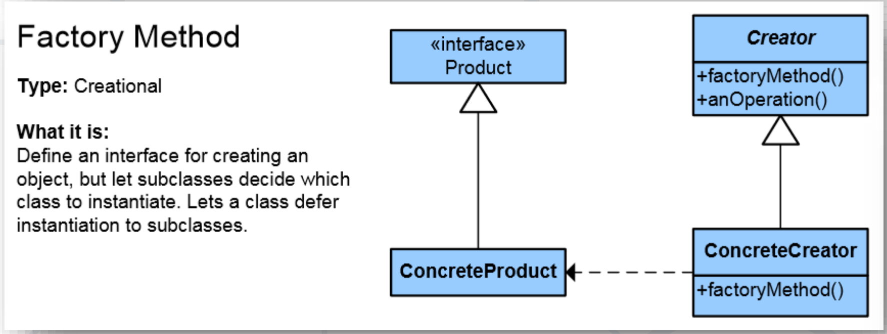
List of Creational Patterns

- Singleton
- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Prototype
- Fluent Interface
- Object Pool
- Lazy Initialization


Singleton

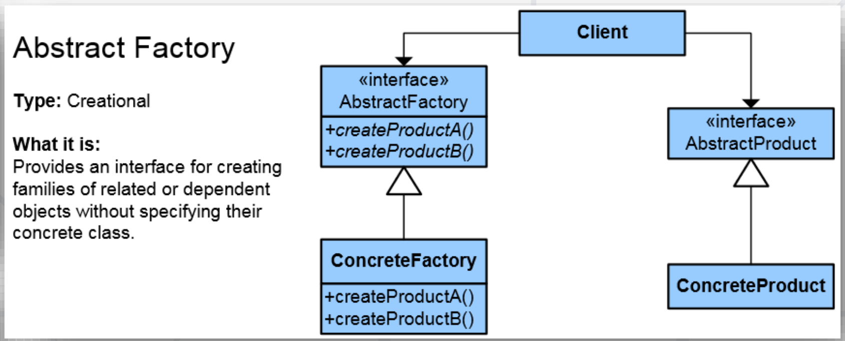
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.
Singleton
-static uniqueInstance
-singletonData
+static instance()
+SingletonOperation()


Factory Method

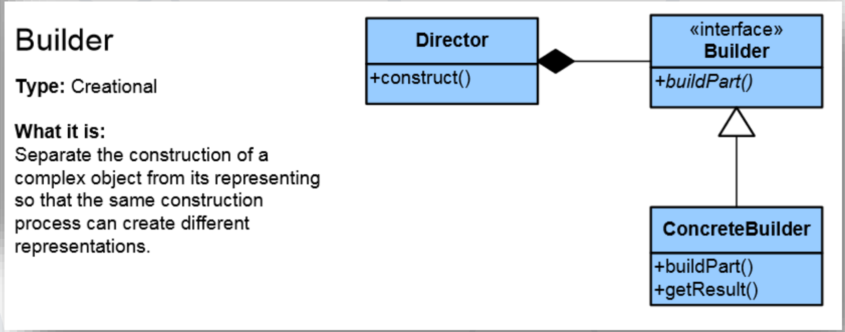
Type: Creational

What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.
«interface» Product
ConcreteProduct
Creator
+factoryMethod()
+anOperation()
ConcreteCreator
+factoryMethod()


Abstract Factory

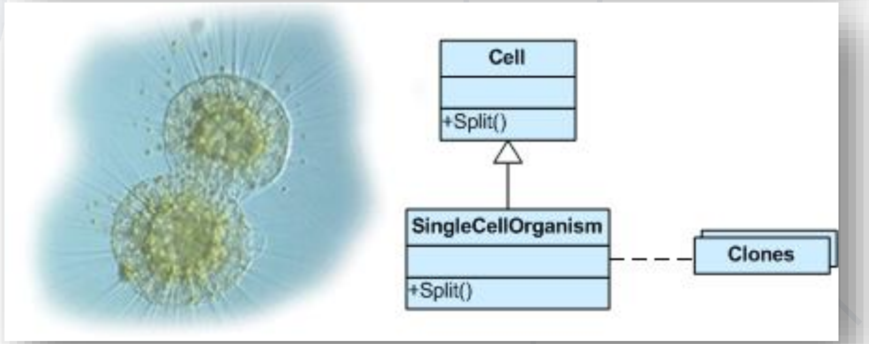
Type: Creational

What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.
Client
«interface» AbstractFactory
+createProductA()
+createProductB()
ConcreteFactory
+createProductA()
+createProductB()
«interface» AbstractProduct
ConcreteProduct


Builder

Type: Creational

What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.
Director
+construct()
«interface» Builder
+buildPart()
ConcreteBuilder
+buildPart()
+getResult()



Creational Patterns in Python

- The language itself provides us with all the **flexibility** we need to create objects in an **elegant fashion**
- We rarely need to implement anything on top, like **Singleton** or **Factory**
- **Factories** are abstraction on top of **constructors**
- **Builders** are abstraction on top of **factories**

Singleton

- The Singleton pattern is used when we want to guarantee that only **one instance** of a given class exists during runtime
- The Singleton is considered an anti-pattern because:
  - It makes the code more complex and less useful
  - It introduces unnecessary restrictions
  - It is hard to test

Singleton: Example

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
# Continues on the next slide
```

Singleton: Example

```
# Continues from the previous slide

@singleton
class DBConnection(object):

    def __init__(self):
        """Initialize your database connection here."""
        pass

    def __str__(self):
        return 'Database connection object'
```

Factory Method

```python
from abc import ABC, abstractmethod

class DataExporter(ABC):
        @abstractmethod
        def export(self, data):
                pass

class CsvDataExporter(ABC):
        @abstractmethod
        def export(self, data) -> str:
                pass

# Continues on the next slide
```

Factory Method

```
# Continues from the previous slide

class DataExporterFactory(ABC):
        @abstractmethod
        def get_exporter(self) -> DataExporter:
                pass

class CsvDataExporterFactory(DataExporterFactory):
        def get_exporter(self) -> DataExporter:
                return CsvDataExporter()
```

Abstract Factory

```
from abc import ABC, abstractmethod
import json

class JsonDataExporter(ABC):
   @abstractmethod
   def export(self, data) -> str:
      pass

class CsvDataExporter(ABC):
   @abstractmethod
   def export(self, data) -> str:
      pass

# Continues on the next slide
```

Abstract Factory

```
# Continues from the previous slide

class DataExporterFactory(ABC):
    @abstractmethod
    def get_json_exporter(self) -> JsonDataExporter:
        pass

    @abstractmethod
    def get_csv_exporter(self) -> CsvDataExporter:
        pass
```

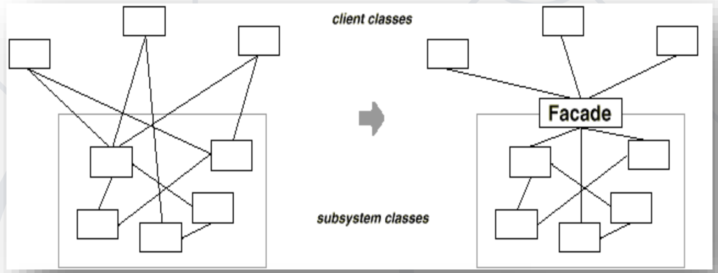
Structural Patterns

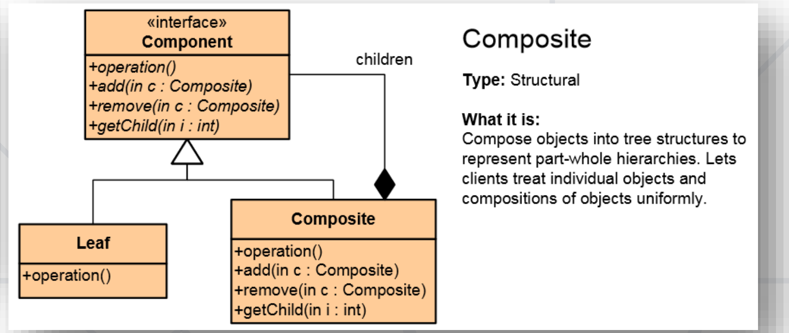
Purposes

- Describe ways to assemble **objects** to implement a **new functionality**
- Ease the design by identifying a simple way to realize the **relationship** between entities
- All about Class and Object composition
  - **Inheritance** to compose interfaces
  - Ways to compose objects to obtain **new functionality**

List of Structural Patterns

- Façade
- Composite
- Flyweight
- Proxy
- Decorator
- Adapter
- Bridge

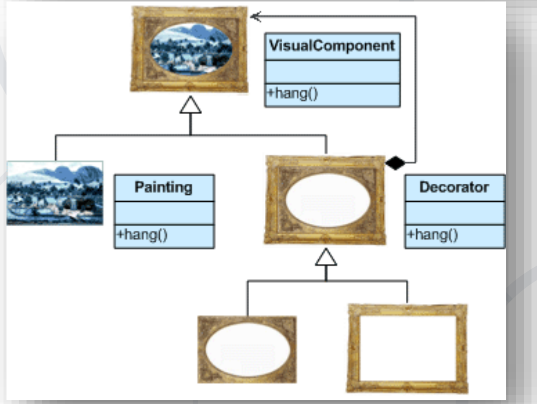


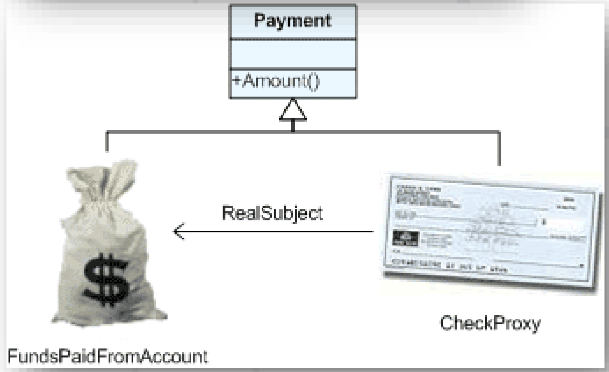


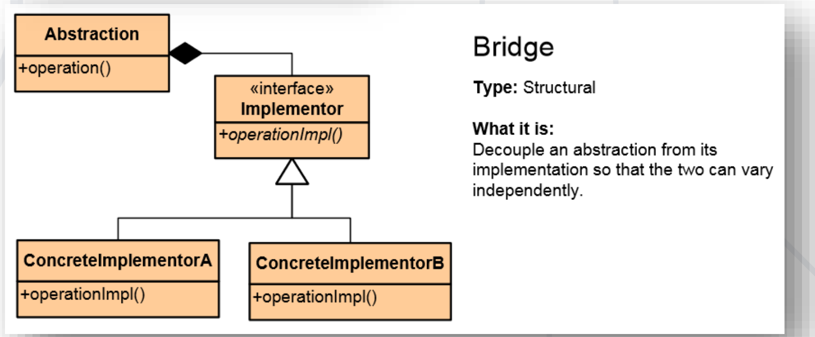
Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.







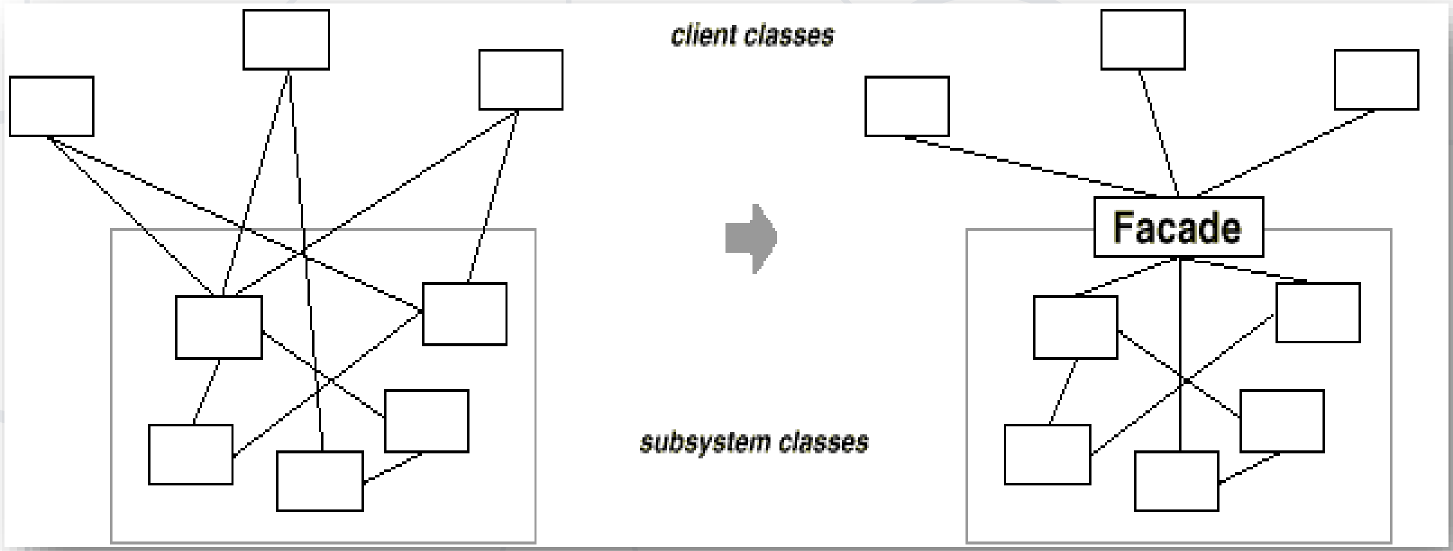
Bridge

Type: Structural

What it is:
Decouple an abstraction from its implementation so that the two can vary independently.

Façade Pattern

- Provides a **unified interface** to a set of interfaces in a subsystem
- Defines a **higher-level interface** that makes the subsystem easier to use



Façade Example

```
class Cook(object):
    def prepareDish(self):
        self.cutter = Cutter()
        self.cutter.cutVegetables()
        self.boiler = Boiler()
        self.boiler.boilVegetables()
```

Façade Example

```
class Cutter(object):
    def cutVegetables(self):
        print("All vegetables are cut")

class Boiler(object):
    def boilVegetables(self):
        print("All vegetables are boiled")
```

Decorator Pattern

```
from abc import ABC, abstractmethod

class DataSource(ABC):
    @abstractmethod
    def writeData(self, data):
        pass

    @abstractmethod
    def readData(self) -> str:
        pass
```

```
class FileDataSource(DataSource):
    def __init__ (self, filename):
        self._file = filename

    def writeData(self, data):
        # write data to file.
        pass

    def readData(self) -> str:
        # read data from file.
        pass
```

Decorator Pattern

```
class EncryptionDecorator(DataSource):
    def writeData(self, data):
        # encrypt the data
        # pass encrypted data to wrapper
        pass

    def readData(self) -> str:
        # get encrypted data
        # decrypt it
        # return it
        pass
```

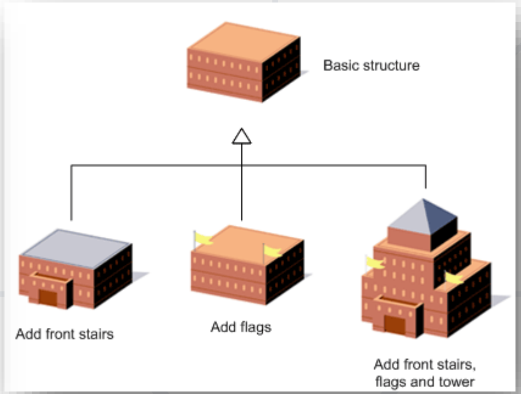
Behavioral Patterns

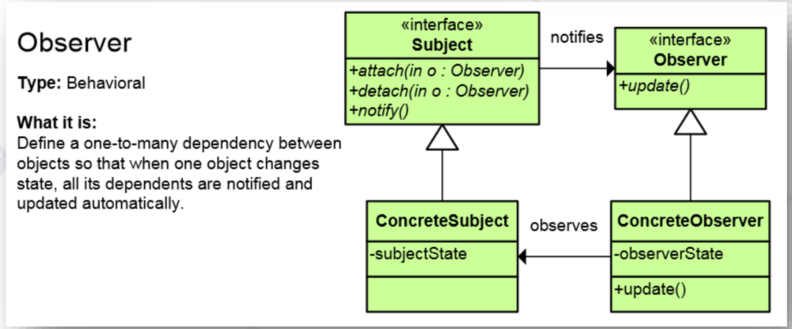
Purposes

- Concerned with the **interaction** between objects
  - Either with the **assignment of responsibilities** between objects
  - Or **encapsulating behavior** in an object and delegating requests to it
- Increases **flexibility** in carrying out cross-classes communication

List of Behavioral Patterns

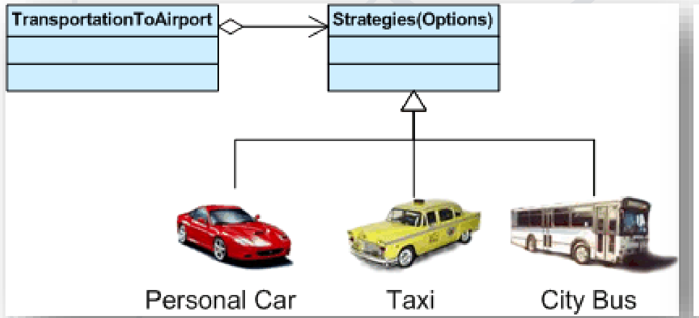
- Chain of Responsibility
- Iterator
- Command
- Template Method
- Strategy
- Observer

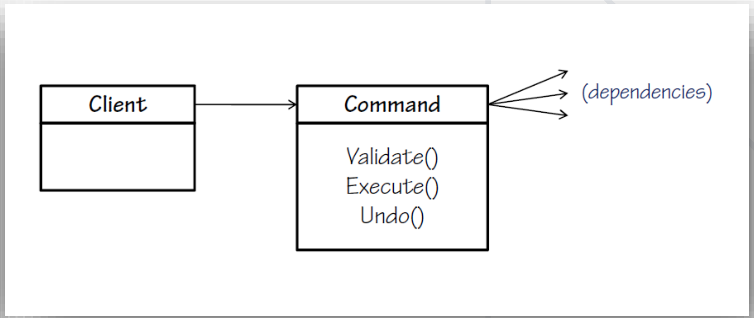



Observer

Type: Behavioral

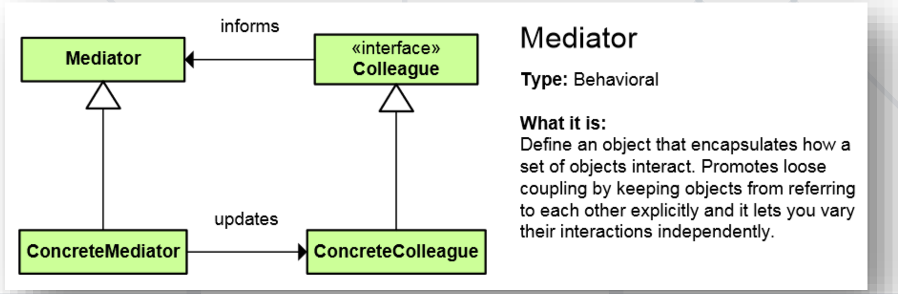
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.





List of Behavioral Patterns

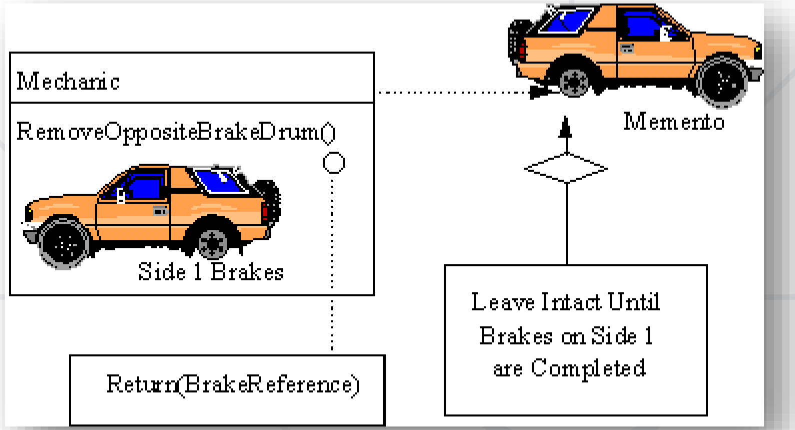
- Mediator
- Memento
- State
- Interpreter
- Visitor

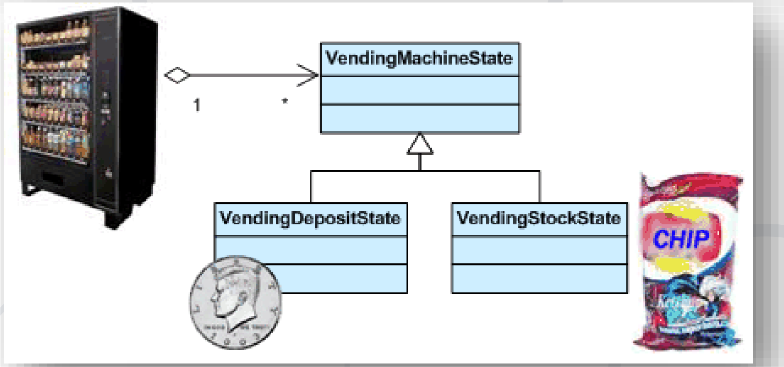


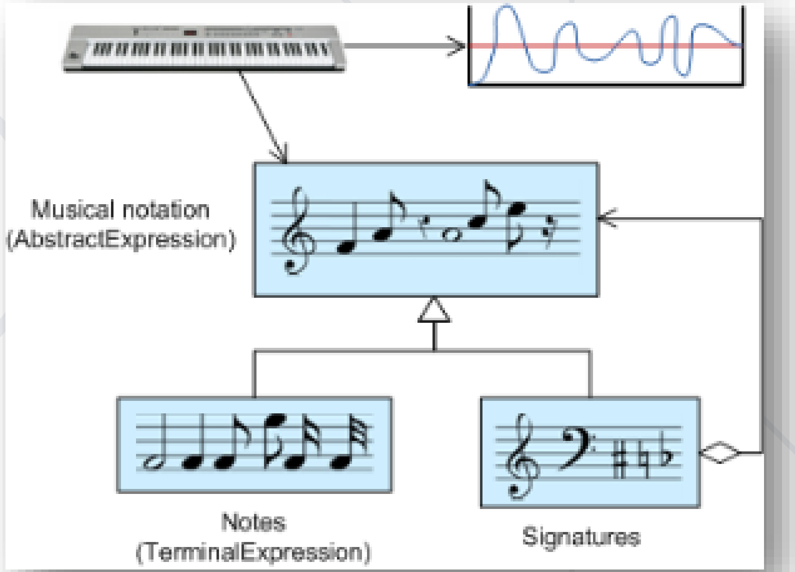
Mediator

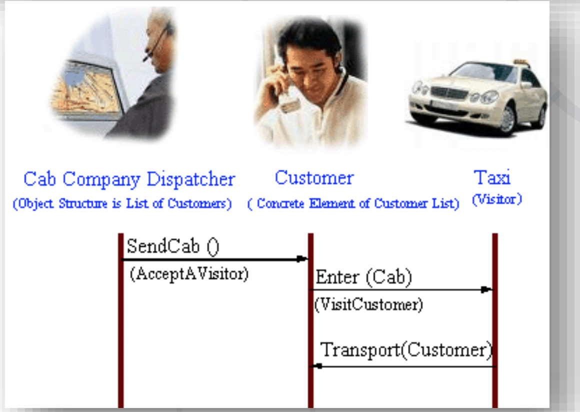
Type: Behavioral

What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.


Mechanic
RemoveOppositeBrakeDrum()
Side 1 Brakes
Memento
Leave Intact Until Brakes on Side 1 are Completed
Return(BrakeReference)

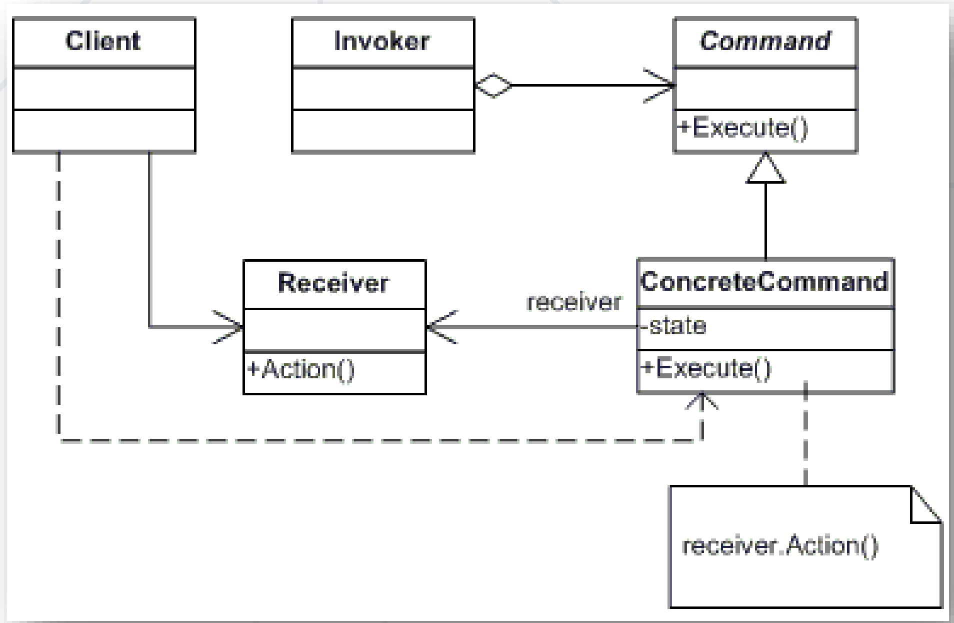



Musical notation (AbstractExpression)
Notes (TerminalExpression)
Signatures



Command Pattern

- An object **encapsulates** all the information needed to call a method later
  - Let's you **parameterize** clients with different requests, queue or log requests, and support undoable operations



The Invoker Class

```python
from abc import ABC, abstractmethod
class Invoker:
    def __init__(self):
        self._commands = []

    def store_command(self, command):
        self._commands.append(command)

    def execute_commands(self):
        for command in self._commands:
            command.execute()
```

Command and Concrete Command Class

```python
class Command(ABC):
    def __init__(self, receiver):
        self._receiver = receiver

    @abstractmethod
    def execute(self):
        pass
```

```python
class ConcreteCommand(Command):
    def execute(self):
        self._receiver.action()

class Receiver:
    def action(self):
        pass
```

Example

```python
def main():
    receiver = Receiver()
    concrete_command = ConcreteCommand(receiver)
    invoker = Invoker()
    invoker.store_command(concrete_command)
    invoker.execute_commands()


if __name__ == "__main__":
    main()
```

Summary

- Design Patterns
  - Provide solutions to common problems
  - Add additional layers of abstraction
- Three main types of Design Patterns
  - Creational
  - Structural
  - Behavioral

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank

Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR

Serving the high achievers

AMBITIONED

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License