Design principle 1: Identificam aspectele aplicatiei care variaza si le separam de cele care raman neschimbate.

Design principle 2:Programam catre o interfata(*supertip*), nu catre o implementare.
Ex: - programarea catre o implementare

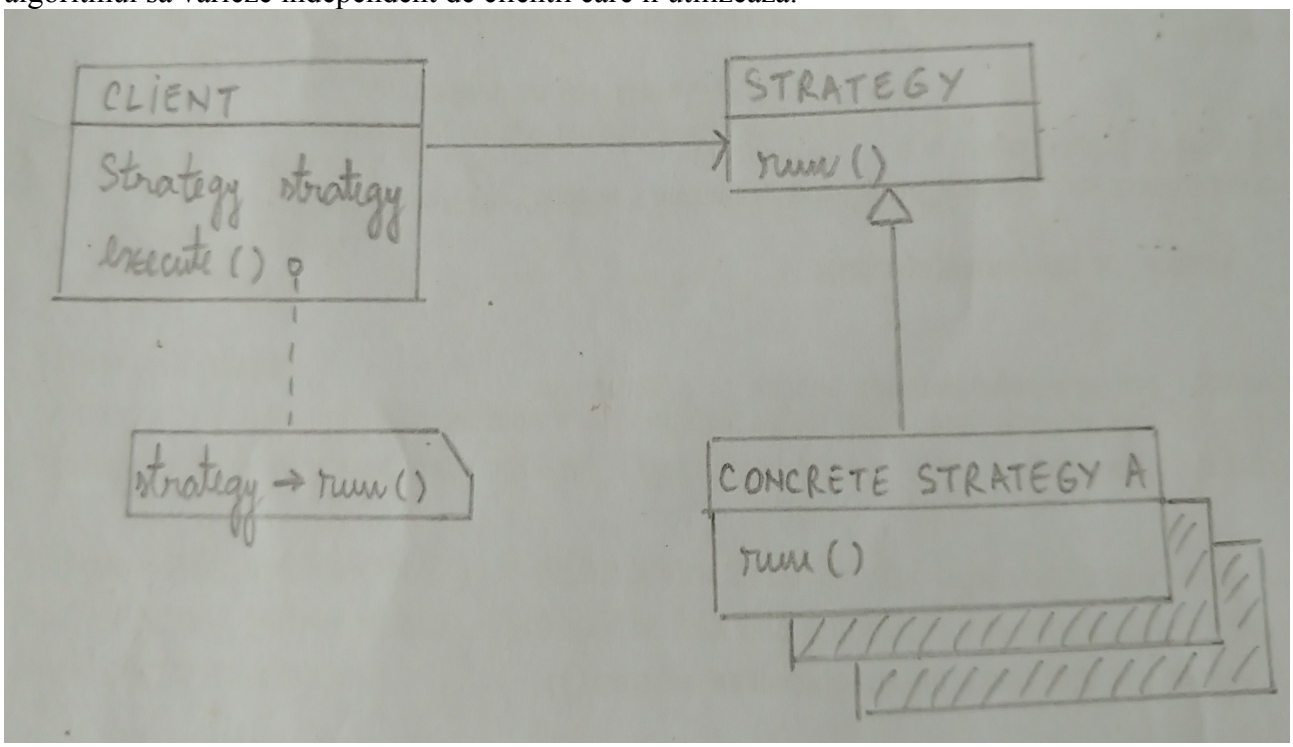
```
Dog* dog=new Dog;
dog->bark();
```

- programarea catre o interfata

```
Animal* animal=new Dog;
animal->makeSound();
```

Design principle 3: Favorizam compozitia, in locul mostenirii.

Strategy Pattern
Defineste o familie de algoritmi, ii incapsuleaza pe fiecare, si ii face interschimbabili.*Strategy* lasa algoritmul sa varieze independent de clientii care il utilizeaza.




```
#include <iostream>
#include <memory>
//Strategy
class WeaponBehaviour
{
public:
	virtual void useWeapon()const = 0;  //run()
};
//Concrete strategy
class SwordBeahaviour :public WeaponBehaviour
{
public:
	void useWeapon()const override
	{
		std::cout << "Swinging a sword\n";
	}
};
//Concrete strategy
class KnifeBehaviour :public WeaponBehaviour
{
```

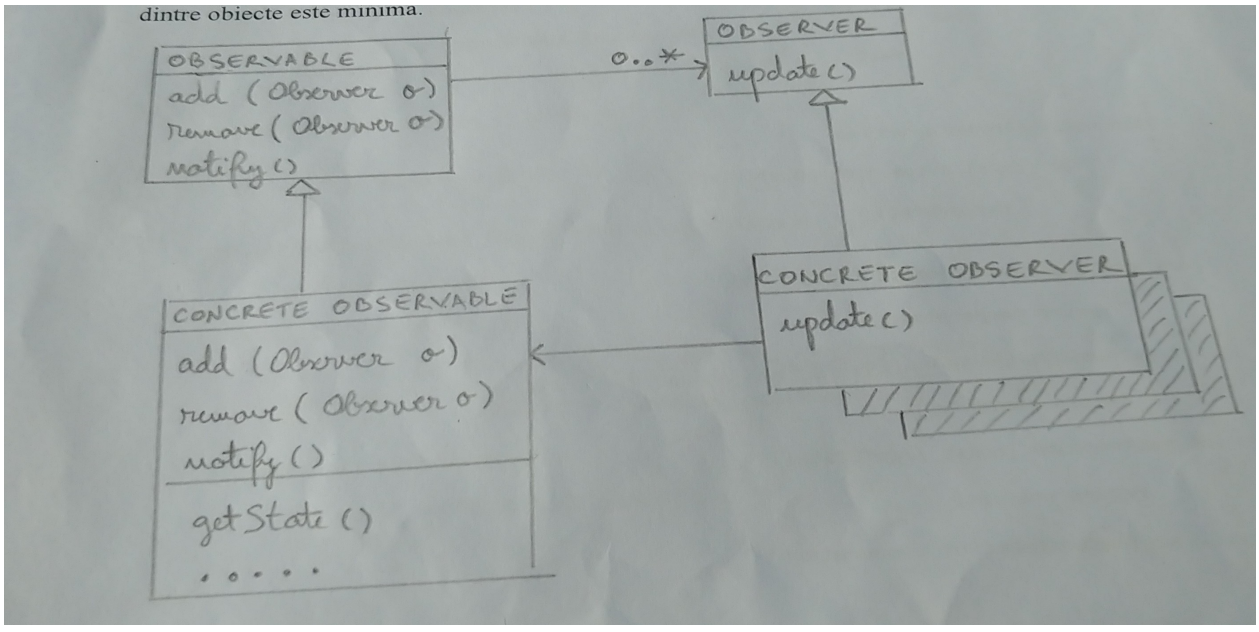
```
public:
	void useWeapon()const override
	{
		std::cout << "Cutting with a knife\n";
	}
};
//Concrete strategy
class AxeBehaviour:public WeaponBehaviour
{
public:
	void useWeapon()const override
	{
		std::cout << "Chopping with an axe\n";
	}
};
//Concrete strategy
class BowAndArrowBehaviour :public WeaponBehaviour
{
public:
	void useWeapon()const override
	{
		std::cout << "Shooting an arrow with a bow\n";
	}

};
//Client
class Character
{
public:
	virtual void fight()const = 0;  //execute()
	void setWeapon(std::unique_ptr<WeaponBehaviour> weapon)
	{
		weapon_ = std::move(weapon);
	}
protected:
	std::unique_ptr<WeaponBehaviour> weapon_;  //strategy
};
class King :public Character
{
public:
	King()
	{
		weapon_ = std::make_unique<SwordBeahaviour>();
	}
	void fight()const override
	{
		weapon_->useWeapon();
	}
};
class Queen :public Character
{
public:
	Queen()
	{
		weapon_ = std::make_unique<BowAndArrowBehaviour>();
	}
	void fight()const override
	{
		weapon_->useWeapon();
	}
};
class Knight :public Character
{
public:
	Knight()
```

```
	{
		weapon_ = std::make_unique<KnifeBehaviour>();
	}
	void fight()const override
	{
		weapon_->useWeapon();
	}
};
class Troll :public Character
{
public:
	Troll()
	{
		weapon_ = std::make_unique<AxeBehaviour>();
	}
	void fight()const override
	{
		weapon_->useWeapon();
	}
};
int main()
{
	std::unique_ptr<Character> king = std::make_unique<King>();
	king->fight();
	std::unique_ptr<Character> queen = std::make_unique<Queen>();
	queen->fight();
	std::unique_ptr<Character> knight = std::make_unique<Knight>();
	knight->fight();
	std::cout << "***Changed at run-time***\n";
	knight->setWeapon(std::make_unique<SwordBeahaviour>());
	knight->fight();
	std::unique_ptr<Character> troll = std::make_unique<Troll>();
	troll->fight();
	return 0;
}
```

Observer Pattern

Defineste *una la mai multe* dependente intre obiecte astfel incat cand obiectul isi schimba starea, toate obiectele dependente sunt notificate si updatate automat.

Design Principle 4:Trebuie sa ne straduim pentru un design *slab cuplat* intre obiecte ce interactioneaza.Aceasta permite aplicatiilor sa faca fata schimbarilor pentru ca interdependenta dintre obiecte este minima.



```
#include <iostream>
#include <list>
#include <memory>
//Observer
class Observer
{
public:
	virtual void update()const = 0;
};
class Display
{
public:
	virtual void display()const = 0;
};
//Observable
class Subject
{
public:
	virtual void add(Observer* observer)= 0;
	virtual void remove(Observer* observer) = 0;
	virtual void notify()const = 0;
};
//Concrete observable
class WeatherStation :public Subject
{
public:
	void add(Observer* observer)override
	{
		observersList.push_back(observer);
	}
	void remove(Observer* observer)override
	{
		for (auto it = observersList.begin();it != observersList.end();)
		{
			if (*it==observer)
			{
				it = observersList.erase(it);
			}
			else
			{
				++it;
			}
		}
	}
	void notify()const override
	{
		for (const auto& o:observersList)
		{
			o->update();
		}
	}
	int getTemperature()const //getState()
	{
		return temperature_;
	}
	int getHumidity()const //getState()
	{
		return humidity_;
	}
	int getPressure()const //getState()
	{
		return pressure_;
	}
	void setTemperature(int temperature)
	{
```

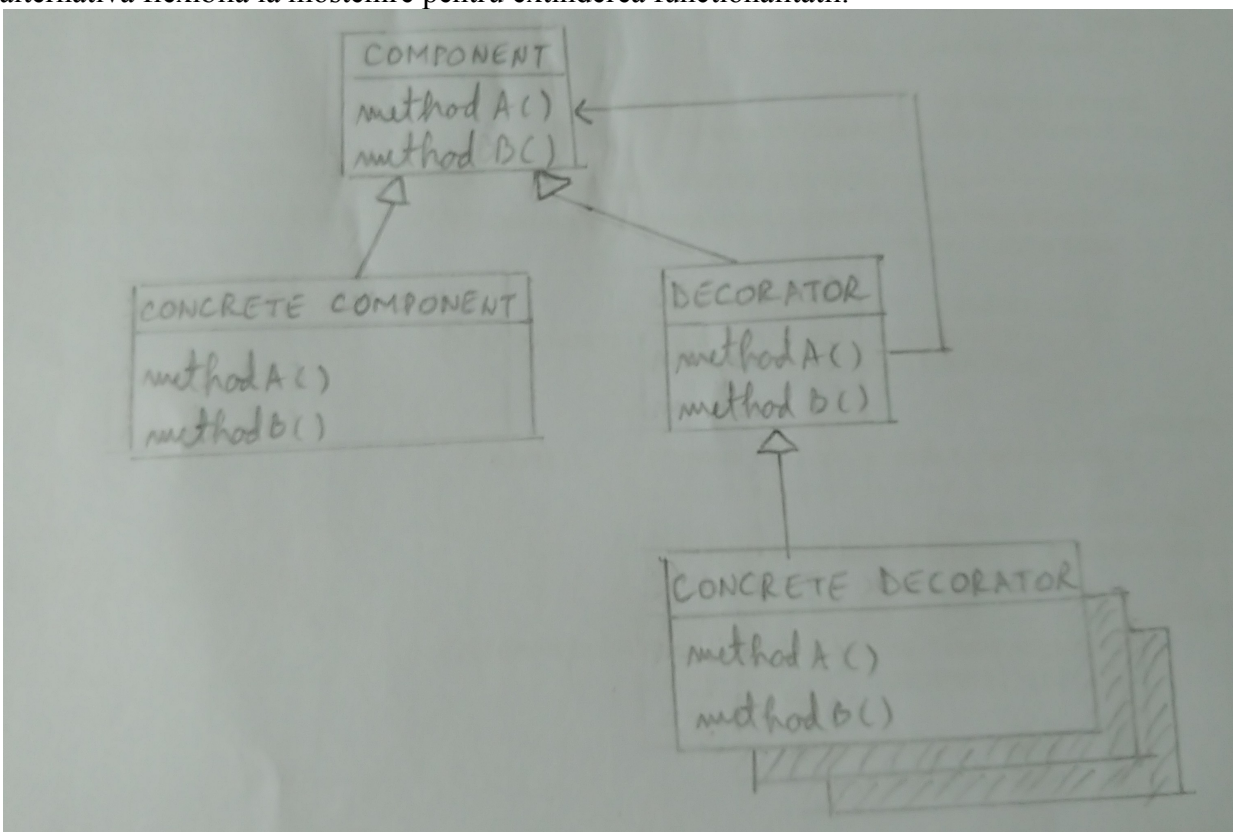
```
		temperature_ = temperature;
	}
	void setHumidity(int humidity)
	{
		humidity_ = humidity;
	}
	void setPressure(int pressure)
	{
		pressure_ = pressure;
	}
	void setMeasurements(int temperature,int humidity,int pressure)
	{
		temperature_ = temperature;
		humidity_ = humidity;
		pressure_ = pressure;
		notify();
	}
private:
	std::list<Observer*> observersList; //0..*
	int temperature_=0;
	int humidity_=0;
	int pressure_=0;
};
//Concrete observer
class SmartPhone :public Observer, public Display
{
public:
	SmartPhone(WeatherStation* station) :station_{station}
	{
		station_->add(this);
	}
	void display()const override
	{
		std::cout << "Temperature:" << station_->getTemperature() << " degrees\n";
		std::cout << "Humidity:" << station_->getHumidity() << "%\n";
		std::cout << "Presure:" << station_->getPressure() << " Pa\n";
	}
	void update()const override
	{
		std::cout << "SMARTPHONE display:\n";
		display();
	}
private:
	WeatherStation* station_;  //HAS-A
};
//Concrete observer
class Tablet :public Observer, public Display
{
public:
	Tablet(WeatherStation* station) :station_{station}
	{
		station_->add(this);
	}
	void display()const override
	{
		std::cout << "Temperature:" << station_->getTemperature() << " degrees\n";
		std::cout << "Humidity:" << station_->getHumidity() << "%\n";
		std::cout << "Presure:" << station_->getPressure() << " Pa\n";
	}
	void update()const override
	{
		std::cout << "TABLET display\n";
		display();
	}
private:
```

```
	WeatherStation* station_;  //HAS-A
};
//Concrete observer
class Monitor :public Observer, public Display
{
public:
	Monitor(WeatherStation* station) :station_{ station }
	{
		station_->add(this);
	}
	void display()const override
	{
		std::cout << "Temperature:" << station_->getTemperature() << " degrees\n";
		std::cout << "Humidity:" << station_->getHumidity() << "%\n";
		std::cout << "Presure:" << station_->getPressure() << " Pa\n";
	}
	void update()const override
	{
		std::cout << "MONITOR display\n";
		display();
	}
private:
	WeatherStation* station_;  //HAS-A
};
//Concrete observer
class ForecastOnTV :public Observer, public Display
{
public:
	ForecastOnTV(WeatherStation* station) :station_{station}
	{
		station_->add(this);
	}
	void display()const override
	{
		std::cout << "Temperature:" << station_->getTemperature() << " degrees\n";
		std::cout << "Humidity:" << station_->getHumidity() << "%\n";
		std::cout << "Presure:" << station_->getPressure() << " Pa\n";
	}
	void update()const override
	{
		//Random forecast
		station_->setTemperature(station_->getTemperature()+5);
		station_->setHumidity(station_->getHumidity()+10);
		station_->setPressure(station_->getPressure()+50);
		std::cout << "FORECAST ON TV display\n";
		display();
	}
private:
	WeatherStation* station_;  //HAS-A
};
int main()
{
	auto weatherStation = std::make_unique<WeatherStation>();
	SmartPhone phone(weatherStation.get());
	Tablet tablet(weatherStation.get());
	Monitor monitor(weatherStation.get());
	ForecastOnTV tv(weatherStation.get());
	std::cout << "Simulating measurements.....\n";
	weatherStation->setMeasurements(37,40,100);
	weatherStation->remove(&tablet);
	std::cout << "*****Removed the tablet*****\n";
	weatherStation->setMeasurements(33,60,200);
	weatherStation->remove(&monitor);
	std::cout << "*****Removed the monitor*****\n";
	weatherStation->setMeasurements(20,45,300);
```

```
	return 0;
}
```

Decorator Pattern

Ataseaza responsabilitati aditionale la un obiect, in mod dinamic(la run-time).Decoratorii furnizeaza o alternativa flexibila la mostenire pentru extinderea functionalitatii.



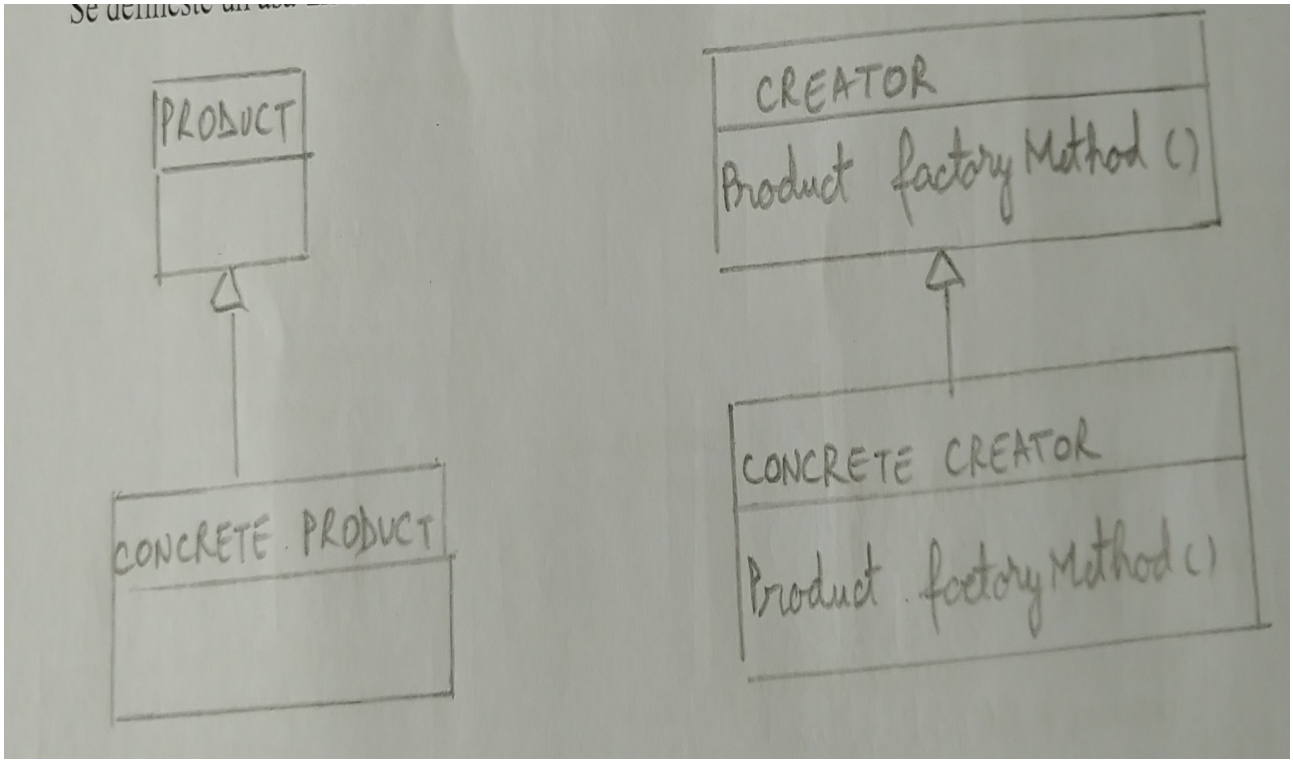

```
#include <iostream>
#include <string>
#include <memory>
//Component
class Animal
{
public:
	virtual void play() = 0;
	virtual ~Animal() = default;
};
//Concrete component
class Dog :public Animal
{
public:
	Dog(const std::string& dogName) :dogName_{dogName} {}
	void play() override
	{
		std::cout << dogName_ << " is playing outside\n";
	}
private:
	std::string dogName_;
};
//Decorator
class AnimalDecorator :public Animal //IS-A
{
public:
	AnimalDecorator(Animal* animal) :animal_{animal} {}
	virtual ~AnimalDecorator()=default;
	Animal* getAnimal()const
	{
```

```
		return animal_;
	}
private:
	Animal* animal_;  //HAS-A
};
//Concrete decorator
class StrayDog :public AnimalDecorator
{
public:
	StrayDog(Animal* animal) :AnimalDecorator(animal){}
	void play() override
	{
		status = "found an owner";
		std::cout << "The stray dog "<<status<<'\n';
		getAnimal()->play();
	}
private:
	std::string status = "has no owner";     //state
};
//Concrete decorator
class OwnedDog :public AnimalDecorator
{
public:
	OwnedDog(Animal* animal) :AnimalDecorator(animal){}
	void play()override
	{
		getAnimal()->play();
		catchFrisbee();
	}
protected:
	void catchFrisbee()const //behaviour
	{
		std::cout << "He is catching the frisbee thrown by the owner\n";
	}
};
//Concrete decorator
class GuardDog :public OwnedDog //wraps another decorator
{
public:
	GuardDog(Animal* animal) :OwnedDog(animal) {}
	void play()override
	{
		getAnimal()->play();
		catchFrisbee();//Can, or not, use the behaviour of the thing it wraps
		guardFlock();
	}
private:
	void guardFlock()const
	{
		std::cout << "He is guarding the flock of sheep\n";
	}
};
int main()
{
	auto Fluffy = std::make_unique<Dog>("Fluffy");
	Fluffy->play();
	std::unique_ptr<Animal> Povic = std::make_unique<Dog>("Povic");
	//Decorated Povic
	std::unique_ptr<Animal> strayDog = std::make_unique <StrayDog>(Povic.get());
	strayDog->play();
	std::unique_ptr<Animal> Rex = std::make_unique<Dog>("Rex");
	//Decorated Rex
	std::unique_ptr<Animal> ownedDog = std::make_unique<OwnedDog>(Rex.get());
	ownedDog->play();
	std::unique_ptr<Animal> Grivei = std::make_unique<Dog>("Grivei");
```

```
	//Decorated Grivei
	std::unique_ptr<Animal> guardDog = std::make_unique<GuardDog>(Grivei.get());
	guardDog->play();
	return 0;
}
```

Factory Method Pattern

Defineste o interfata pentru crearea unui obiect, dar lasa subclasele sa decida ce clasa sa instantieze. Se defineste un asa-zis *constructor virtual*.

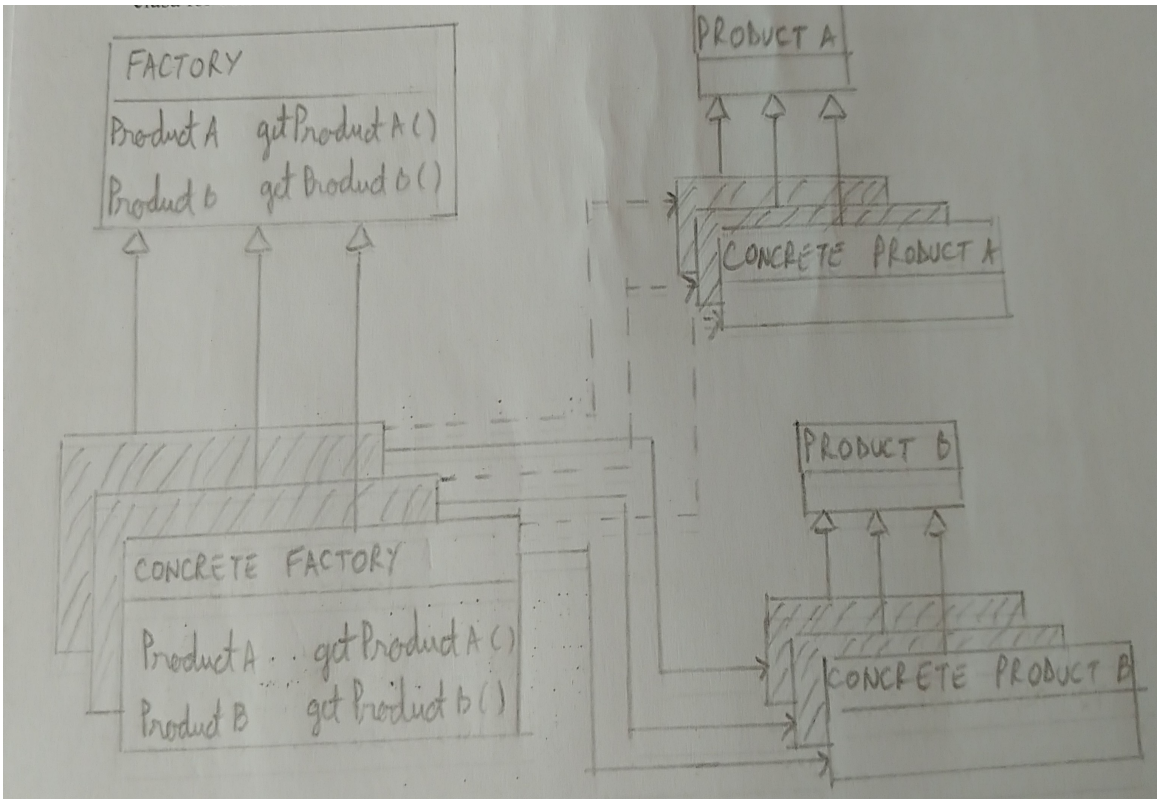



```
#include <iostream>
#include <string>
#include <memory>
#include <ctime>
class Toy //Product
{
public:
	virtual void combineParts()const = 0;
	virtual void applyLabel() = 0;
	const std::string& getName()const
	{
		return name_;
	}
	void setName(const std::string& name)
	{
		name_ = name;
	}
	virtual ~Toy() = default;
private:
	std::string name_;
};
//Concrete product
class Car :public Toy
{
public:
	void combineParts()const override
	{
		std::cout << "Combining car parts\n";
```

```
	}
	void applyLabel()override
	{
		std::cout << "Applying label\n";
		setName("CAR");
	}
};
//Concrete product
class Bike :public Toy
{
public:
	void combineParts()const override
	{
		std::cout << "Combining bike parts\n";
	}
	void applyLabel()override
	{
		std::cout << "Applying label\n";
		setName("BIKE");
	}
};
//Concrete product
class Plane :public Toy
{
public:
	void combineParts()const override
	{
		std::cout << "Combining plane parts\n";
	}
	void applyLabel()override
	{
		std::cout << "Applying label\n";
		setName("PLANE");
	}
};
//Creator
class ToyFactory
{
public:
	//Factory method
	virtual std::unique_ptr<Toy> createToy(const std::string&)const =0;
};
//Concrete creator
class NormalToyFactory:public ToyFactory
{
public:
	std::unique_ptr<Toy> createToy(const std::string& toyName)const override
	{
		if (toyName=="car")
		{
			return std::make_unique<Car>();
		}
		else if (toyName=="bike")
		{
			return std::make_unique<Bike>();
		}
		else if (toyName=="plane")
		{
			return std::make_unique<Plane>();
		}
		else
		{
			std::cout << "Please enter the correct name of the toy!!!\n";
			return  nullptr;
		}
```

```
	}
};
//Concrete creator
class RandomToyFactory :public ToyFactory
{
public:
	std::unique_ptr<Toy> createToy(const std::string& dummy)const override
	{
		if (rand() % 3 == 0)
		{
			return std::make_unique<Car>();
		}
		else if (rand() % 5 == 0)
		{
			return std::make_unique<Bike>();
		}
		else
		{
			return std::make_unique<Plane>();
		}
	}
};
void showProduct(Toy* toy)
{
	toy->combineParts();
	toy->applyLabel();
	std::cout << "The toy is a "<<toy->getName()<<'\n';
}
int main()
{
	srand(time(0));
	std::string input;
	while (true)
	{
		std::cout << "Enter 'car' or 'bike' or 'plane' or 'random' -- 'exit' for
                                                                    termination:";
		std::cin >> input;
		if (input=="exit")
		{
			break;
		}
		if (input=="random")
		{
			std::unique_ptr<ToyFactory> randomFactory =
                        std::make_unique<RandomToyFactory>();
			auto randomToy = randomFactory->createToy(input);
			std::cout << "Randomly creating toy...\n";
			showProduct(randomToy.get());
			std::cout << "**********************\n";
		}
		else
		{
			std::unique_ptr<ToyFactory> factory =
                        std::make_unique<NormalToyFactory>();
			auto toy = factory->createToy(input);
			if (toy != nullptr)
			{
				showProduct(toy.get());
				std::cout << "**********************\n";
			}
		}
	}
	return 0;
}
```

Abstract Factory Pattern
Furnizeaza o interfata pentru crearea de familii de obiecte inrudite sau dependente fara a specifica clasa lor concreta.




```
#include <iostream>
#include <string>
#include <memory>
//Abstract Product A
class WoodenToy
{
public:
	virtual ~WoodenToy() = default;
	virtual void display()const = 0;
};
//Concrete Product A1
class WoodenShip :public WoodenToy
{
public:
	void display()const override
	{
		std::cout << "Wooden Ship created\n";
	}
};
//Concrete Product A2
class WoodenRocket :public WoodenToy
{
public:
	void display()const override
	{
		std::cout << "Wooden Rocket created\n";
	}
};
//Concrete Product A3
```

```
class WoodenHorse :public WoodenToy
{
public:
	void display()const override
	{
		std::cout << "Wooden Horse created\n";
	}
};

//Abstract Product B
class PlasticToy
{
public:
	virtual ~PlasticToy() = default;
	virtual void display()const = 0;
};
//Concrete Product B1
class PlasticCar :public PlasticToy
{
public:
	void display()const override
	{
		std::cout << "Plastic Car created\n";
	}
};
//Concrete Product B2
class PlasticTrain :public PlasticToy
{
public:
	void display()const override
	{
		std::cout << "Plastic Train created\n";
	}
};
//Concrete Product B3
class PlasticDoll :public PlasticToy
{
public:
	void display()const override
	{
		std::cout << "Plastic Doll created\n";
	}
};
//Abstract Factory
class ToyFactory
{
public:
	virtual std::unique_ptr<WoodenToy> createWoodenToy()const = 0;
	virtual std::unique_ptr<PlasticToy> createPlasticToy()const = 0;
};
//Concreate Factory 1
class SimpleFactory :public ToyFactory
{
public:
	std::unique_ptr<WoodenToy> createWoodenToy()const override
	{
		return std::make_unique<WoodenShip>();
	}
	std::unique_ptr<PlasticToy> createPlasticToy()const override
	{
		return std::make_unique <PlasticCar>();
	}
};
//Concrete Factory 2
class RobustFactory :public ToyFactory
```

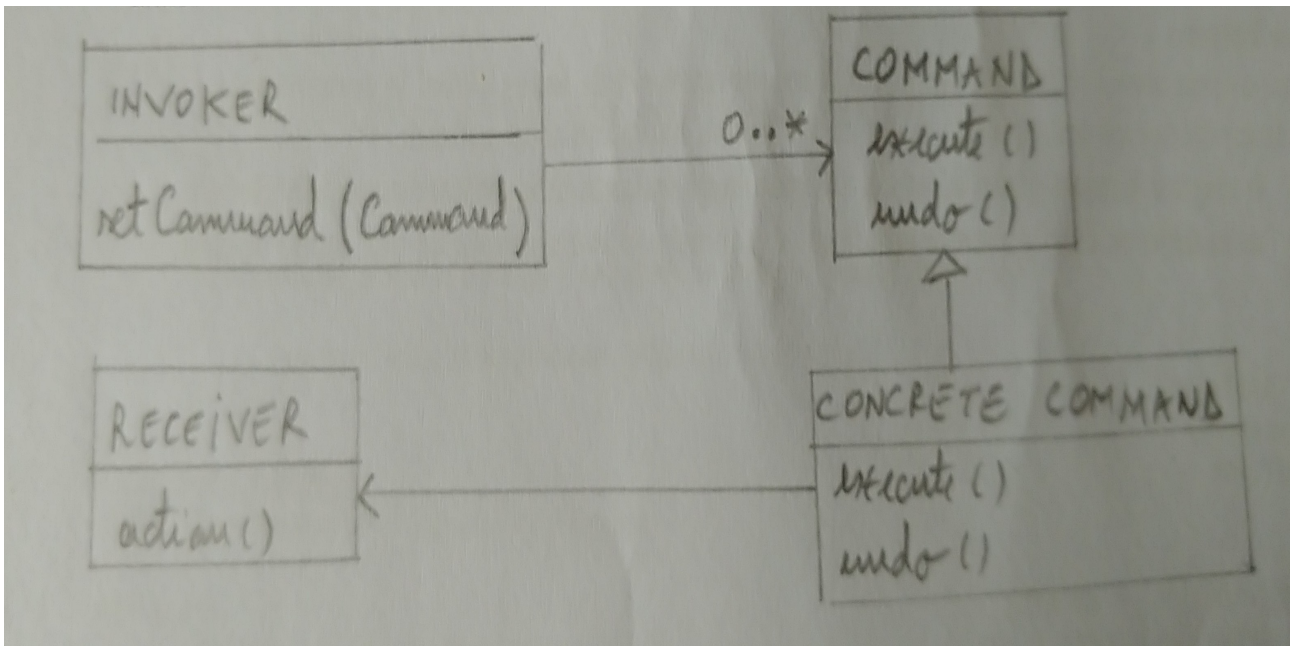
```
{
public:
	std::unique_ptr<WoodenToy> createWoodenToy()const override
	{
		return std::make_unique<WoodenRocket>();
	}
	std::unique_ptr<PlasticToy> createPlasticToy()const override
	{
		return std::make_unique <PlasticTrain>();
	}
};
//Concrete Factory 3
class FigurineFactory :public ToyFactory
{
public:
	std::unique_ptr<WoodenToy> createWoodenToy()const override
	{
		return std::make_unique<WoodenHorse>();
	}
	std::unique_ptr<PlasticToy> createPlasticToy()const override
	{
		return std::make_unique <PlasticDoll>();
	}
};
int main()
{
	std::string input;
	while (true)
	{
		std::cout << "Enter 'simple', 'robust', 'figurine' -- 'exit' for
                                                                termination:";
		std::cin >> input;
		if (input=="exit")
		{
			break;
		}
		if (input=="simple")
		{
			std::unique_ptr<ToyFactory> factory =
                                        std::make_unique<SimpleFactory>();
			factory->createWoodenToy()->display();
			factory->createPlasticToy()->display();
		}
		else if (input=="robust")
		{
			std::unique_ptr<ToyFactory> factory =
                                       std::make_unique<RobustFactory>();
			factory->createWoodenToy()->display();
			factory->createPlasticToy()->display();
		}
		else if (input == "figurine")
		{
			std::unique_ptr<ToyFactory> factory =
                                        std::make_unique<FigurineFactory>();
			factory->createWoodenToy()->display();
			factory->createPlasticToy()->display();
		}
		else
		{
			std::cout << "Please enter the correct category!!!\n";
		}
	}
	return 0;
}
```

Singleton Pattern
Asigura ca o clasa are o *singura* instanta si furnizeaza un punct de acces global la acea instanta.
Doua motive pentru care unii spun ca nu ar trebui folosit niciodata acest pattern:

- o singura instanta globala, adica o variabila globala; trebuie evitate variabilele globale
- daca dupa un timp cerintele aplicatiei se schimba si avem nevoie de mai multe instante

```
#include <iostream>
#include <memory>
class Singleton
{
public:
	Singleton(int) = delete;
	int getValue()const;
	void setValue(int);
	static Singleton* getInstance();
private:
	Singleton();
	int value_;
	static std::unique_ptr<Singleton> instance;
};
std::unique_ptr<Singleton> Singleton::instance{ nullptr };
Singleton::Singleton() :value_{} {}
int Singleton::getValue()const
{
	return value_;
}
void Singleton::setValue(int value)
{
	value_ = value;
}
Singleton* Singleton::getInstance()
{
	if (instance==nullptr)
	{
		//Nu merge cu make_unique
		instance = std::unique_ptr<Singleton>(new Singleton);
	}
	return instance.get();
}
void foo()
{
	Singleton::getInstance()->setValue(99);
	std::cout << "Value in foo() is:"<<Singleton::getInstance()->getValue()<<'\n';
}
int main()
{
	Singleton::getInstance()->setValue(33);
	std::cout << "First value in main() is:" << Singleton::getInstance()->getValue() <<
                                                                                '\n';
	foo();
	std::cout << "Now value in main() is:" << Singleton::getInstance()->getValue() <<
                                                                                '\n';
	return 0;
}
```

Command Pattern
Incapsuleaza o cerere intr-un obiect, permitand astfel parametrizarea altor obiecte cu cereri diferite, permitand punerea in coada si inregistrarea cererilor, si permitand suport pentru operatii de tip *undo*.




```
#include <iostream>
#include <string>
#include <memory>
#include <array>
//Command interface
class Command
{
public:
	virtual ~Command() = default;
	virtual void execute() = 0;
	virtual void undo() = 0;
};
//Receiver
class Light
{
public:
	Light(const std::string& roomName) :roomName_{ roomName } {}
	void on()const
	{
		std::cout << roomName_ <<" light is ON\n";
	}
	void off()
	{
		std::cout << roomName_ << " light is OFF\n";
	}
private:
	std::string roomName_;
};
//Receiver
class CeilingFan
{
public:
	CeilingFan(const std::string& roomName) :roomName_{ roomName }, speed{OFF}{}
	void high()
	{
		speed = HIGH;
		std::cout << roomName_ << " ceiling fan is on HIGH\n";
	}
	void medium()
	{
		speed = MEDIUM;
		std::cout << roomName_ << " ceiling fan is on MEDIUM\n";
```

```
	}
	void low()
	{
		speed = LOW;
		std::cout << roomName_ << " ceiling fan is on LOW\n";
	}
	void on()
	{
		speed = ON;
		std::cout << roomName_ << " ceiling fan is ON\n";
	}
	void off()
	{
		speed = OFF;
		std::cout << roomName_ << " ceiling fan is OFF\n";
	}
	int getSpeed()const
	{
		return speed;
	}
	const static int ON = -1;
	const static int OFF = 0;
	const static int LOW = 1;
	const static int MEDIUM = 2;
	const static int HIGH = 3;
private:
	std::string roomName_;
	int speed;
};



//Receiver
class Stereo
{
public:
	Stereo(const std::string& roomName) :roomName_{roomName} {}
	void on()const
	{
		std::cout << roomName_ <<" stereo is ON\n";
	}
	void off()const
	{
		std::cout << roomName_ << " stereo is OFF\n";
	}
	void setCD()const
	{
		std::cout << roomName_ << " stereo is set for CD input\n";
	}
	void setDVD()const
	{
		std::cout << roomName_ << " stereo is set for DVD input\n";
	}
	void setRadio()const
	{
		std::cout << roomName_ << " stereo is playing a radio station\n";
	}
	void setVolume(int volume)const
	{
		std::cout << roomName_ << " stereo volume is set to "<< volume <<'\n';
	}
private:
	std::string roomName_;
};
```

```
//Receiver
class GarageDoor
{
public:
	void up()const
	{
		std::cout << "The garage door is UP\n";
	}
	void down()const
	{
		std::cout << "The garage door is DOWN\n";
	}
	void stop()const
	{
		std::cout << "The garage door has STOPPED\n";
	}
	const static int UP = 1;
	const static int DOWN = -1;
	const static int STOP = 0;
	int getPosition()const
	{
		return position;
	}
private:
	int position;
};
//...Other receivers....
//Concrete command
class LightOnCommand :public Command
{
public:
	LightOnCommand(Light* light) :light_{light} {}
	void execute() override
	{
		light_->on();
	}
	void undo() override
	{
		light_->off();
	}
private:
	Light* light_;
};
//Concrete command
class LightOffCommand :public Command
{
public:
	LightOffCommand(Light* light) :light_{ light } {}
	void execute() override
	{
		light_->off();
	}
	void undo() override
	{
		light_->on();
	}
private:
	Light* light_;
};
//Concrete command
class CeilingFanOnCommand :public Command
{
public:
	CeilingFanOnCommand(CeilingFan* ceilingFan) :ceilingFan_{ceilingFan} {}
	void execute() override
```

```
	{
		previousSpeed = ceilingFan_->getSpeed();//saved before change
		ceilingFan_->on();
	}
	void undo() override
	{
		if (previousSpeed == ceilingFan_->HIGH)
		{
			ceilingFan_->high();
		}
		else if (previousSpeed == ceilingFan_->MEDIUM)
		{
			ceilingFan_->medium();
		}
		else if (previousSpeed == ceilingFan_->LOW)
		{
			ceilingFan_->low();
		}
		else if (previousSpeed == ceilingFan_->ON)
		{
			ceilingFan_->on();
		}
		else if (previousSpeed == ceilingFan_->OFF)
		{
			ceilingFan_->off();
		}
	}
private:
	CeilingFan* ceilingFan_;
	int previousSpeed =ceilingFan_->OFF;
};
//Concrete command
class CeilingFanOffCommand :public Command
{
public:
	CeilingFanOffCommand(CeilingFan* ceilingFan) :ceilingFan_{ ceilingFan } {}
	void execute() override
	{
		previousSpeed = ceilingFan_->getSpeed();//saved before change
		ceilingFan_->off();
	}
	void undo() override
	{
		if (previousSpeed == ceilingFan_->HIGH)
		{
			ceilingFan_->high();
		}
		else if (previousSpeed == ceilingFan_->MEDIUM)
		{
			ceilingFan_->medium();
		}
		else if (previousSpeed == ceilingFan_->LOW)
		{
			ceilingFan_->low();
		}
		else if (previousSpeed == ceilingFan_->ON)
		{
			ceilingFan_->on();
		}
		else if (previousSpeed == ceilingFan_->OFF)
		{
			ceilingFan_->off();
		}
	}
private:
```

```
	CeilingFan* ceilingFan_;
	int previousSpeed = ceilingFan_->OFF;
};
//Concrete command
class CeilingFanHighCommand :public Command
{
public:
	CeilingFanHighCommand(CeilingFan* ceilingFan) :ceilingFan_{ceilingFan} {}
	void execute() override
	{
		previousSpeed = ceilingFan_->getSpeed();//saved before change
		ceilingFan_->high();
	}
	void undo() override
	{
		if (previousSpeed==ceilingFan_->HIGH)
		{
			ceilingFan_->high();
		}
		else if (previousSpeed == ceilingFan_->MEDIUM)
		{
			ceilingFan_->medium();
		}
		else if (previousSpeed==ceilingFan_->LOW)
		{
			ceilingFan_->low();
		}
		else if (previousSpeed==ceilingFan_->ON)
		{
			ceilingFan_->on();
		}
		else if (previousSpeed==ceilingFan_->OFF)
		{
			ceilingFan_->off();
		}

	}
private:
	CeilingFan* ceilingFan_;
	int previousSpeed= ceilingFan_->OFF;
};
//Concrete command
class CeilingFanMediumCommand :public Command
{
public:
	CeilingFanMediumCommand(CeilingFan* ceilingFan) :ceilingFan_{ ceilingFan } {}
	void execute() override
	{
		previousSpeed = ceilingFan_->getSpeed();//saved before change
		ceilingFan_->medium();
	}
	void undo() override
	{
		if (previousSpeed == ceilingFan_->HIGH)
		{
			ceilingFan_->high();
		}
		else if (previousSpeed == ceilingFan_->MEDIUM)
		{
			ceilingFan_->medium();
		}
		else if (previousSpeed == ceilingFan_->LOW)
		{
			ceilingFan_->low();
		}
```

```
		else if (previousSpeed == ceilingFan_->ON)
		{
			ceilingFan_->on();
		}
		else if (previousSpeed == ceilingFan_->OFF)
		{
			ceilingFan_->off();
		}

	}
private:
	CeilingFan* ceilingFan_;
	int previousSpeed = ceilingFan_->OFF;
};
//Concrete command
class CeilingFanLowCommand :public Command
{
public:
	CeilingFanLowCommand(CeilingFan* ceilingFan) :ceilingFan_{ ceilingFan } {}
	void execute() override
	{
		previousSpeed = ceilingFan_->getSpeed();//saved before change
		ceilingFan_->low();
	}
	void undo() override
	{
		if (previousSpeed == ceilingFan_->HIGH)
		{
			ceilingFan_->high();
		}
		else if (previousSpeed == ceilingFan_->MEDIUM)
		{
			ceilingFan_->medium();
		}
		else if (previousSpeed == ceilingFan_->LOW)
		{
			ceilingFan_->low();
		}
		else if (previousSpeed == ceilingFan_->ON)
		{
			ceilingFan_->on();
		}
		else if (previousSpeed == ceilingFan_->OFF)
		{
			ceilingFan_->off();
		}

	}
private:
	CeilingFan* ceilingFan_;
	int previousSpeed = ceilingFan_->OFF;
};
//Concrete command
class StereoOnWithCDCommand :public Command
{
public:
	StereoOnWithCDCommand(Stereo* stereo) :stereo_{stereo} {}
	void execute() override
	{
		stereo_->on();
		stereo_->setCD();
		stereo_->setVolume(11);
	}
	void undo() override
	{
```

```
		stereo_->off();
	}
private:
	Stereo* stereo_;
};
//Concrete command
class StereoOffCommand :public Command
{
public:
	StereoOffCommand(Stereo* stereo) :stereo_{ stereo } {}
	void execute() override
	{
		stereo_->off();
	}
	void undo() override
	{
		stereo_->on();
	}
private:
	Stereo* stereo_;
};
//....Other concrete commands of this kind
//Concrete command
class GarageDoorUpCommand :public Command
{
public:
	GarageDoorUpCommand(GarageDoor* garageDoor) :garageDoor_{garageDoor} {}
	void execute() override
	{
		previousPosition = garageDoor_->getPosition();//saved before change
		garageDoor_->up();
	}
	void undo() override
	{
		if (previousPosition==garageDoor_->STOP)
		{
			garageDoor_->stop();
		}
		else if (previousPosition == garageDoor_->DOWN)
		{
			garageDoor_->down();
		}
		else if (previousPosition == garageDoor_->UP)
		{
			garageDoor_->up();
		}
	}
private:
	GarageDoor* garageDoor_;
	int previousPosition=garageDoor_->STOP;
};
class GarageDoorDownCommand :public Command
{
public:
	GarageDoorDownCommand(GarageDoor* garageDoor) :garageDoor_{ garageDoor } {}
	void execute() override
	{
		previousPosition = garageDoor_->getPosition();//saved before change
		garageDoor_->down();
	}
	void undo() override
	{
		if (previousPosition == garageDoor_->STOP)
		{
			garageDoor_->stop();
```

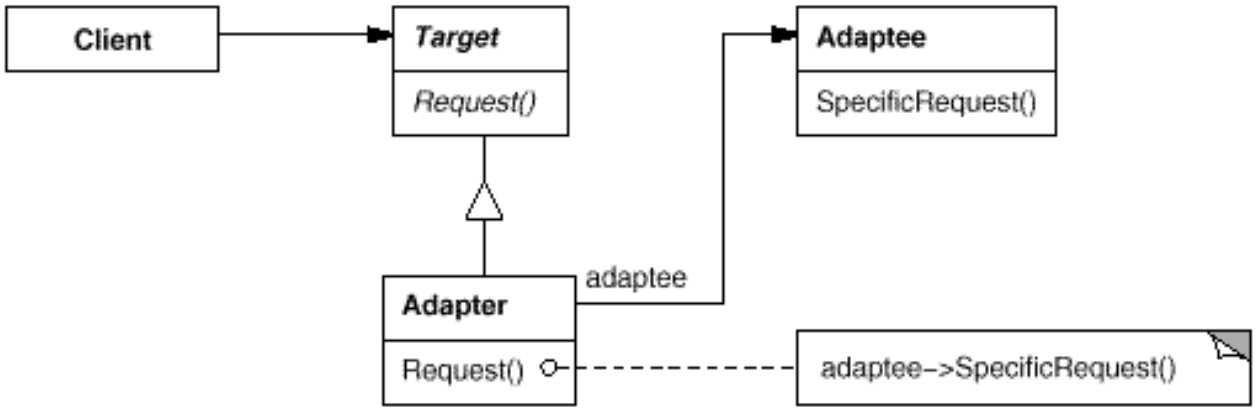
```
		}
		else if (previousPosition == garageDoor_->DOWN)
		{
			garageDoor_->down();
		}
		else if (previousPosition == garageDoor_->UP)
		{
			garageDoor_->up();
		}
	}
private:
	GarageDoor* garageDoor_;
	int previousPosition = garageDoor_->STOP;
};
class GarageDoorStopCommand :public Command
{
public:
	GarageDoorStopCommand(GarageDoor* garageDoor) :garageDoor_{ garageDoor } {}
	void execute() override
	{
		previousPosition = garageDoor_->getPosition();//saved before change
		garageDoor_->stop();
	}
	void undo() override
	{
		if (previousPosition == garageDoor_->STOP)
		{
			garageDoor_->stop();
		}
		else if (previousPosition == garageDoor_->DOWN)
		{
			garageDoor_->down();
		}
		else if (previousPosition == garageDoor_->UP)
		{
			garageDoor_->up();
		}
	}
private:
	GarageDoor* garageDoor_;
	int previousPosition = garageDoor_->STOP;
};
class NoCommand :public Command
{
public:
	void execute() override {}
	void undo() override {}
};
//Invoker
class RemoteControl
{
public:
	RemoteControl()
	{
		auto noCommand = std::make_unique<NoCommand>();
		for (int i = 0;i < 7;++i)
		{
			onCommands[i] = noCommand.get();
			offCommands[i] = noCommand.get();
		}
		undoCommand = noCommand.get();
	}
	void setCommand(int slot,Command* onCommand,Command* offCommand)
	{
		onCommands[slot] = onCommand;
```

```
		offCommands[slot] = offCommand;
	}
	void onButtonWasPushed(int slot)
	{
		onCommands[slot]->execute();
		undoCommand = onCommands[slot];//save the last command
	}
	void offButtonWasPushed(int slot)
	{
		offCommands[slot]->execute();
		undoCommand = offCommands[slot];//save the last command
	}
	void undoButtonWasPushed(int slot)
	{
		undoCommand->undo();
	}
private:
	std::array<Command*, 7> onCommands;
	std::array<Command*, 7> offCommands;
	Command* undoCommand;
};
int main()
{
	auto livingRoomLight = std::make_unique<Light>("Living Room");
	auto kitchenLight = std::make_unique<Light>("Kitchen");
	auto ceilingFan = std::make_unique<CeilingFan>("Bedroom");
	auto garageDoor = std::make_unique<GarageDoor>();
	auto stereo = std::make_unique<Stereo>("Living Room");

	auto livingRoomLightOn = std::make_unique<LightOnCommand>(livingRoomLight.get());
	auto livingRoomLightOff = std::make_unique<LightOffCommand>(livingRoomLight.get());
	auto kitchenLightOn= std::make_unique<LightOnCommand>(kitchenLight.get());
	auto kitchenLightOff = std::make_unique<LightOffCommand>(kitchenLight.get());
	auto ceilingFanOn = std::make_unique<CeilingFanOnCommand>(ceilingFan.get());
	auto ceilingFanMedium = std::make_unique<CeilingFanMediumCommand>(ceilingFan.get());
	auto ceilingFanOff = std::make_unique<CeilingFanOffCommand>(ceilingFan.get());
	auto garageDoorUp = std::make_unique<GarageDoorUpCommand>(garageDoor.get());
	auto garageDoorDown = std::make_unique<GarageDoorDownCommand>(garageDoor.get());
	auto garageDoorStop = std::make_unique<GarageDoorStopCommand>(garageDoor.get());
	auto stereoOnWithCD = std::make_unique<StereoOnWithCDCommand>(stereo.get());
	auto stereoOff = std::make_unique<StereoOffCommand>(stereo.get());

	auto remoteControl = std::make_unique<RemoteControl>();
	remoteControl->setCommand(0, livingRoomLightOn.get(), livingRoomLightOff.get());
	remoteControl->setCommand(1, kitchenLightOn.get(), kitchenLightOff.get());
	remoteControl->setCommand(2, ceilingFanOn.get(), ceilingFanOff.get());
	remoteControl->setCommand(3, ceilingFanMedium.get(), ceilingFanOff.get());
	remoteControl->setCommand(4, garageDoorUp.get(), garageDoorDown.get());
	remoteControl->setCommand(5, garageDoorStop.get(), garageDoorDown.get());
	remoteControl->setCommand(6,stereoOnWithCD.get(),stereoOff.get());

	remoteControl->onButtonWasPushed(0);
	remoteControl->offButtonWasPushed(0);
	remoteControl->onButtonWasPushed(1);
	remoteControl->offButtonWasPushed(1);
	remoteControl->onButtonWasPushed(2);
	remoteControl->offButtonWasPushed(2);
	remoteControl->onButtonWasPushed(3);
	remoteControl->offButtonWasPushed(3);
	remoteControl->undoButtonWasPushed(3);
	remoteControl->onButtonWasPushed(4);
	remoteControl->offButtonWasPushed(4);
	remoteControl->onButtonWasPushed(5);
	remoteControl->offButtonWasPushed(5);
	remoteControl->undoButtonWasPushed(5);
```

```
	remoteControl->onButtonWasPushed(6);
	remoteControl->offButtonWasPushed(6);
	remoteControl->undoButtonWasPushed(6);
	return 0;
}
```

Adaptern Pattern(*Wrapper*)

Converteste interfata unei clase(*Adaptee*) in una pe care o asteapta clientul(*Target*).Pattern-ul lasa clasele sa lucreze impreuna, ce in alte conditii nu ar putea din cauza interfetelor incompatibile.



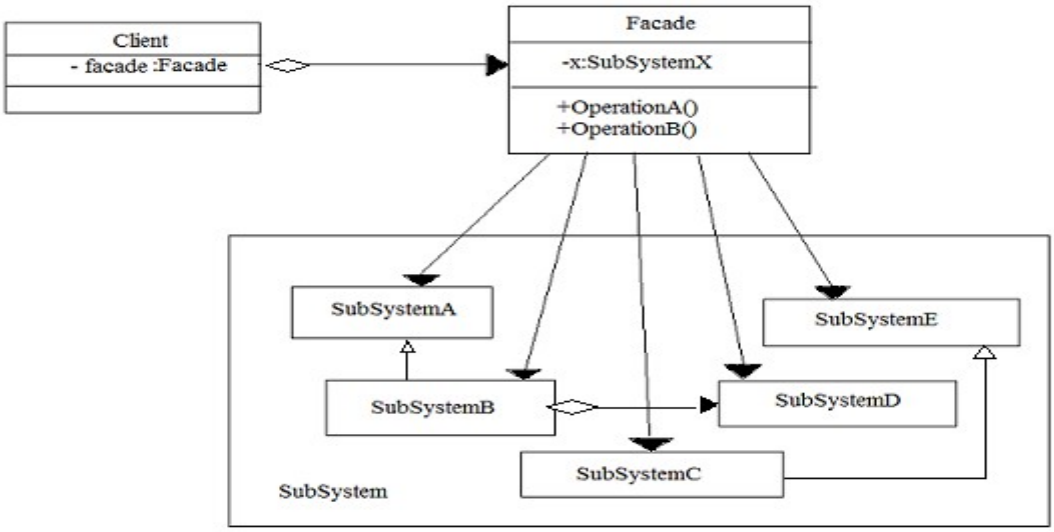

```
#include <iostream>
#include <string>
#include <cstdlib> //for rand()
#include <ctime>    //for time()
#include <memory>
//Target
class Player
{
public:
	virtual ~Player() = default;
	virtual void useWeapon() const = 0; //Request
	virtual void runForward() const = 0; //Request
	virtual void assignAcolyte(const std::string&) const = 0; //Request
};
class Paladin :public Player
{
public:
	void useWeapon() const override
	{
		int attackDamage = rand() % 10 + 1;
		std::cout << "Paladin does " << attackDamage << " damage with the sword\n";
	}
	void runForward() const override
	{
		int movement = rand() % 20 + 1;
		std::cout << "Paladin runs " << movement << " spaces forward\n";
	}
	void assignAcolyte(const std::string& acolyte) const override
	{
		std::cout << "Paladin fights together with " << acolyte << "\n";
	}
};
//Adaptee
class Mage
```

```
{
public:
	void throwFireball() const //Specific request
	{
		int attackDamage = rand() % 15 + 1;
		std::cout << "Mage does " << attackDamage << " damage with fireball\n";
	}
	void walkForward() const //Specific request
	{
		int movement = rand() % 10 + 1;
		std::cout << "Mage walks " << movement << " spaces forward\n";
	}
	void summonWolf(const std::string& wolf) const //Specific request
	{
		std::cout << "Mage summons " << wolf << " the wolf, to fight with him\n";
	}
};
//Adapter/Wrapper
class MageAdapter :public Player
{
public:
	MageAdapter(Mage* mage) :mage_{ mage } {}
	void useWeapon() const override
	{
		mage_->throwFireball();
	}
	void runForward() const override
	{
		mage_->walkForward();
	}
	void assignAcolyte(const std::string& acolyte) const override
	{
		mage_->summonWolf(acolyte);
	}
private:
	Mage* mage_;  //HAS-A
};
//Client
int main()
{
	srand(time(0));
	std::cout << "==PALADIN==\n";
	std::unique_ptr<Player> paladin = std::make_unique<Paladin>();
	paladin->useWeapon();
	paladin->runForward();
	paladin->assignAcolyte("\"Nocturno Culto\"");

	std::cout << "==MAGE==\n";
	auto mage = std::make_unique<Mage>();
	mage->throwFireball();
	mage->walkForward();
	mage->summonWolf("\"Black Fang\"");

	std::cout << "==MAGE with adapter==\n";
	std::unique_ptr<Player> magus = std::make_unique<MageAdapter>(mage.get());
	magus->useWeapon();
	magus->runForward();
	magus->assignAcolyte("\"Fenrir\"");

	std::cout << "==MAGE with adapter==\n";
	std::unique_ptr<Player> warlock =
	std::make_unique<MageAdapter>(std::make_unique<Mage>().get());
	warlock->useWeapon();
	warlock->runForward();
	warlock->assignAcolyte("\"Cerber\"");
```

```
	return 0;
}
```

Facade Pattern

Furnizeaza o interfata unificata la un set de interfete dintr-un subsistem.Defineste o interfata de nivel inalt astfel incat subsistemul este mai usor de utilizat.



Design Principle 5:Principle of Least Knowledge(*Law of Demeter*):Vorbeste numai cu prietenii imediati.

Guidelines:

Luam un obiect oarecare si din orice metoda a acelui obiect ar trebui sa invocam doar metodele ce apartin:

- obiectului insusi
- obiectelor pasate ca argumente ale metodei
- oricarui obiect pe care metoda il creaza sau instantiaza
- oricarui component al obiectului

Observatie: NU ar trebui sa apelam metode ale obiectelor care sunt returnate de apelul altor metode.Daca am face-o, atunci am creste numarul obiectelor pe care le cunoastem direct.Principiul ne forteaza ca sa cerem obiectului sa faca cererea pentru noi:

```
//fara principiu
float getTemp()
{
	Thermometer thermometer=station.getThermometer();
	return  thermometer.getTemperature();
}
//cu principiu
float getTemp()
{
	return station.getTemperature();
}
```

Am adaugat metoda *getTemperature()* clasei *Station*, care face cererea catre termometru pentru noi, reducand astfel numarul claselor de care depindem.

Ex. de asa **DA**:

```
class Car
{
	Engine engine;//Component(1)
public:
	Car(){//initializare engine...etc}
	void start(Key key)//Obiect pasat ca argument(2)
	{
		Doors* doors=new Doors;//Creem un obiect nou(3)
		bool authorized=key.turns();//putem apela(2)
		if(authorized)
		{
			engine.start();//putem apela(1)
			updateDisplay();//putem apela(4)
			doors->lock();//putem apela(3), daca creem sau instantiem
			delete doors;
		}
	}
	void updateDisplay(){//update display....}//metoda locala a obiectului(4)
}
```

Ex. de asa **NU**:

```
class House
{
	WheatherStation station;
	//.....
	float getTemp()
	{
		return station.getThermometer().getTemperature();
	}
};
```

Apelam metoda unui obiect returnat de alt apel.

```
#include <iostream>
#include <memory>
//Observatie: fiecare pointer utilizat tb. verificat daca nu e nullptr
class Tuner;
class DVDPlayer;
class CDPlayer;
class Amplifier
{
public:
	Amplifier(const std::string& ampName) :ampName_{ ampName } {}
	void on()const
	{
		std::cout << ampName_ << " amplifier is ON\n";
	}
	void off()const
	{
		std::cout << ampName_ << " amplifier is OFF\n";
	}
	void setTuner(Tuner* tuner)
	{
		std::cout << ampName_ << " amplifier is setting the tuner\n";
		tuner_ = tuner;
	}
	void setDVD(DVDPlayer* dvdPlayer)
	{
		std::cout << ampName_ << " amplifier is setting the DVD player\n";
		dvdPlayer_ = dvdPlayer;
	}
	void setCD(CDPlayer* cdPlayer)
	{
		std::cout << ampName_ << " amplifier is setting the CD player\n";
```

```
		cdPlayer_ = cdPlayer;
	}
	void setStereoSound()const
	{
		std::cout << ampName_ << " amplifier is setting the stereo sound ON\n";
	}
	void setSurroundSound()const
	{
		std::cout << ampName_ << " amplifier is setting the surround sound ON (5
                                                        speakers, 1 subwoofer)\n";
	}
	void setVolume(int level)const
	{
		std::cout << ampName_ << " amplifier is setting the volume to " << level <<
                                                                        '\n';
	}
	const std::string& getAmpName()const
	{
		return ampName_;
	}
private:
	Tuner* tuner_;
	DVDPlayer* dvdPlayer_;
	CDPlayer* cdPlayer_;
	std::string ampName_;
};
class Tuner
{
public:
	Tuner(const std::string& tunerName, Amplifier* amplifier) :tunerName_{ tunerName },
                                                            amplifier_{amplifier} {}
	void on()const
	{
		std::cout <<"Radio tuner "<< tunerName_<< " is ON\n";
	}
	void off()const
	{
		std::cout << "Radio tuner " << tunerName_ << " is OFF\n";
	}
	void setFrequency(double frequency)const
	{
		std::cout << "Frequency is set to "<< frequency <<" MHz\n";
		std::cout << "Amplifier "<< amplifier_->getAmpName() <<" is adapting to
                                                                frequency\n";
	}
private:
	Amplifier* amplifier_;
	std::string tunerName_;
};
class Screen
{
public:
	void up()const
	{
		std::cout << "The screen is UP\n";
	}
	void down()const
	{
		std::cout << "The screen is DOWN\n";
	}
};
class TheaterLights
{
public:
	void on()const
```

```
	{
		std::cout << "Theater lights are ON\n";
	}
	void off()const
	{
		std::cout << "Theater lights are OFF\n";
	}
	void dim(int level)const
	{
		std::cout << "Theater lights dimming to "<< level <<"%\n";
	}
};
class CDPlayer
{
public:
	CDPlayer(const std::string& cdPlayerName, Amplifier* amplifier)
	:cdPlayerName_{ cdPlayerName }, amplifier_{amplifier} {}
	void on()const
	{
		std::cout << cdPlayerName_ << " CD Player is ON\n";
	}
	void off()const
	{
		std::cout << cdPlayerName_ << " CD Player is OFF\n";
	}
	void play(const std::string& title)const
	{
		std::cout << cdPlayerName_ << " CD Player is playing "<< title <<"\n";
	}
	void pause()const
	{
		std::cout << cdPlayerName_ << " CD Player is on pause\n";
		std::cout << "Amplifier " << amplifier_->getAmpName() << " is stopped\n";
	}
	void stop()const
	{
		std::cout << cdPlayerName_ << " CD Player is stopped\n";
		std::cout << "Amplifier " << amplifier_->getAmpName() << " is stopped\n";
	}
	void eject()const
	{
		std::cout << cdPlayerName_ << " CD Player is ejecting the CD\n";
	}
private:
	Amplifier* amplifier_;
	std::string cdPlayerName_;
};
class DVDPlayer
{
public:
	DVDPlayer(const std::string& dvdPlayerName, Amplifier* amplifier)
	:dvdPlayerName_{ dvdPlayerName }, amplifier_{amplifier}{}
	void on()const
	{
		std::cout<< dvdPlayerName_ << " DVD Player is ON\n";
	}
	void off()const
	{
		std::cout << dvdPlayerName_ << " DVD Player is OFF\n";
	}
	void play(const std::string& movie)const
	{
		std::cout << dvdPlayerName_ << " DVD Player is playing "<< movie <<"\n";
	}
	void pause()const
```

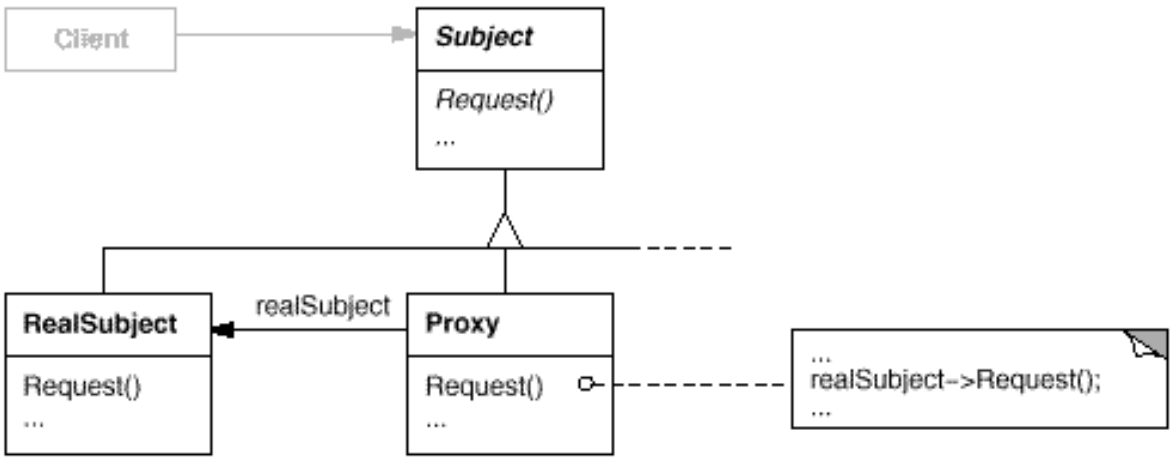
```
	{
		std::cout << dvdPlayerName_ << " DVD Player is on pause\n";
		std::cout << "Amplifier " << amplifier_->getAmpName() << " is stopped\n";
	}
	void stop()const
	{
		std::cout << dvdPlayerName_ << " DVD Player is stopped\n";
		std::cout << "Amplifier " << amplifier_->getAmpName() << " is stopped\n";
	}
	void eject()const
	{
		std::cout << dvdPlayerName_ << " DVD Player is ejecting the DVD\n";
	}
	const std::string& getDVDName()const
	{
		return dvdPlayerName_;
	}
private:
	Amplifier* amplifier_;
	std::string dvdPlayerName_;

};
class Projector
{
public:
	Projector(const std::string& projectorName, DVDPlayer* dvdPlayer)
	:projectorName_{ projectorName }, dvdPlayer_{dvdPlayer}{}
	void on()const
	{
		std::cout << projectorName_ << " projector is ON\n";
	}
	void off()const
	{
		std::cout << projectorName_ << " projector is OFF\n";
	}
	void tvMode()const
	{
		std::cout << projectorName_ <<" projector is on TV Mode\n";
		std::cout << dvdPlayer_->getDVDName() << " DVD player set to TV mode\n";
	}
	void wideScreenMode()const
	{
		std::cout << projectorName_ <<" projector is on widescreen mode\n";
		std::cout << dvdPlayer_->getDVDName() << " DVD player set to widescreen
                                                                    mode\n";
	}
private:
	DVDPlayer* dvdPlayer_;
	std::string projectorName_;
};
class HomeTheaterFacade
{
public:
	//Observatie: fiecare pointer utilizat tb. verificat daca nu e nullptr
	HomeTheaterFacade(Amplifier* amp,Tuner* tuner,DVDPlayer* dvdPlayer,CDPlayer*
	cdPlayer,Projector* projector,TheaterLights* lights,Screen* screen)
	{
		amp_ = amp;
		tuner_ = tuner;
		dvdPlayer_ = dvdPlayer;
		cdPlayer_ = cdPlayer;
		projector_ = projector;
		lights_ = lights;
		screen_ = screen;
	}
```

```
	void watchMovie(const std::string& movie)const
	{
		std::cout << "Get ready to watch a movie...\n";
		lights_->dim(10);
		screen_->down();
		projector_->on();
		projector_->wideScreenMode();
		amp_->on();
		amp_->setDVD(dvdPlayer_);
		amp_->setSurroundSound();
		amp_->setVolume(19);
		dvdPlayer_->on();
		dvdPlayer_->play(movie);
	}
	void endMovie()const
	{
		std::cout << "Shutting movie theater down...\n";
		lights_->on();
		screen_->up();
		projector_->off();
		amp_->off();
		dvdPlayer_->stop();
		dvdPlayer_->eject();
		dvdPlayer_->off();
	}
	void listenToCD(const std::string& cdName)const
	{
		std::cout << "Get ready to listen to CD...\n";
		lights_->on();
		amp_->on();
		amp_->setVolume(25);
		amp_->setCD(cdPlayer_);
		amp_->setStereoSound();
		cdPlayer_->on();
		cdPlayer_->play(cdName);
	}
	void endCD()const
	{
		std::cout << "Shutting down CD....\n";
		cdPlayer_->stop();
		amp_->off();
		cdPlayer_->eject();
		cdPlayer_->off();
	}
	void listenToRadio(double frequency)const
	{
		std::cout << "Get ready to listen to the radio...\n";
		tuner_->on();
		tuner_->setFrequency(frequency);
		amp_->on();
		amp_->setVolume(11);
		amp_->setTuner(tuner_);
	}
	void endRadio()const
	{
		std::cout << "Shutting down the tuner...\n";
		tuner_->off();
		amp_->off();
	}
private:
	Amplifier* amp_;
	Tuner* tuner_;
	DVDPlayer* dvdPlayer_;
	CDPlayer* cdPlayer_;
	Projector* projector_;
```

```
	TheaterLights* lights_;
	Screen* screen_;
};
int main()
{
	auto amp = std::make_unique<Amplifier>("Top-O-Line");
	auto tuner = std::make_unique<Tuner>("Best Tuner",amp.get());
	auto dvdPlayer = std::make_unique<DVDPlayer>("Hitachi",amp.get());
	auto cdPlayer = std::make_unique<CDPlayer>("Denon",amp.get());
	auto projector = std::make_unique<Projector>("Sony",dvdPlayer.get());
	auto lights = std::make_unique<TheaterLights>();
	auto screen = std::make_unique<Screen>();

	auto homeTheater = std::make_unique<HomeTheaterFacade>(amp.get(),tuner.get(),
	dvdPlayer.get(),cdPlayer.get(),projector.get(),lights.get(),screen.get());

	homeTheater->watchMovie("Terminator Salvation");
	homeTheater->endMovie();

	homeTheater->listenToCD("Scorpions - Best of");
	homeTheater->endCD();

	homeTheater->listenToRadio(87.80);
	homeTheater->endRadio();
	return 0;
}
```

Proxy Pattern
Furnizeaza un surogat sau inlocuitor pentru alt obiect cu scopul de a controla accesul la acesta.

Pattern-ul poate fi:
Remote proxy: controleaza accesul la un obiect *remote*.
Virtual proxy: controleaza accesul la o resursa care este costisitoarea de creat.
Protection proxy: controleaza accesul la o resursa bazat pe drepturi de acces.



Virtual Proxy

```
#include <iostream>
#include <string>
#include <memory>
//Subject
class Image
```

```
{
public:
	virtual ~Image() = default;
	virtual void showImage() = 0;
};
//Real subject
class HighResolutionImage :public Image
{
public:
	HighResolutionImage(const std::string& imageFilePath)
	{
		loadImage(imageFilePath);
	}
	void showImage() override
	{
		std::cout << "Showing the high resolution image\n";
	}
private:
	void loadImage(const std::string& imageFilePath)
	{
		// This is heavy and costly operation
		std::cout << "Load Image from disk into memory\n";
	}
};
//Proxy
class ImageProxy :public Image
{
public:
	ImageProxy(const std::string& imageFilePath) :imageFilePath_{imageFilePath} {}
	void showImage() override
	{
		//lazy creation
		proxifiedImage = std::make_unique<HighResolutionImage>(imageFilePath_);
		proxifiedImage->showImage();
	}
private:
	//HAS-A Real subject (Image -> polimorfism)
	std::unique_ptr<Image> proxifiedImage;
	std::string imageFilePath_;
};
int main()
{
	// Assuming that the user selects a folder that has 3 images
	//Create the 3 images
	std::unique_ptr<Image> highResolutionImage1 =
	std::make_unique<ImageProxy>("sample/veryHighResPhoto1.jpeg");
	std::unique_ptr<Image> highResolutionImage2 =
	std::make_unique<ImageProxy>("sample/veryHighResPhoto2.jpeg");
	std::unique_ptr<Image> highResolutionImage3 =
	std::make_unique<ImageProxy>("sample/veryHighResPhoto3.jpeg");

	// Assume that the user clicks on Image one item in a list
	// this would cause the program to call showImage() for that image only
	// note that in this case only image one was loaded into memory
	highResolutionImage1->showImage();

	// consider using the high resolution image objects directly
	std::unique_ptr<Image> highResolutionImageNoProxy1 =
	std::make_unique<HighResolutionImage>("sample/veryHighResPhoto1.jpeg");
	std::unique_ptr<Image> highResolutionImageNoProxy2 =
	std::make_unique<HighResolutionImage>("sample/veryHighResPhoto2.jpeg");
	std::unique_ptr<Image> highResolutionImageNoProxy3 =
	std::make_unique<HighResolutionImage>("sample/veryHighResPhoto3.jpeg");

	// Assume that the user selects image two item from images list
```

```
	highResolutionImageNoProxy2->showImage();

	// Note that in this case all images have been loaded into memory
	// and not all have been actually displayed
	// This is a waste of memory resources
	return 0;
}
Output:
Load Image from disk into memory
Showing the high resolution image
Load Image from disk into memory
Load Image from disk into memory
Load Image from disk into memory
Showing the high resolution image
```

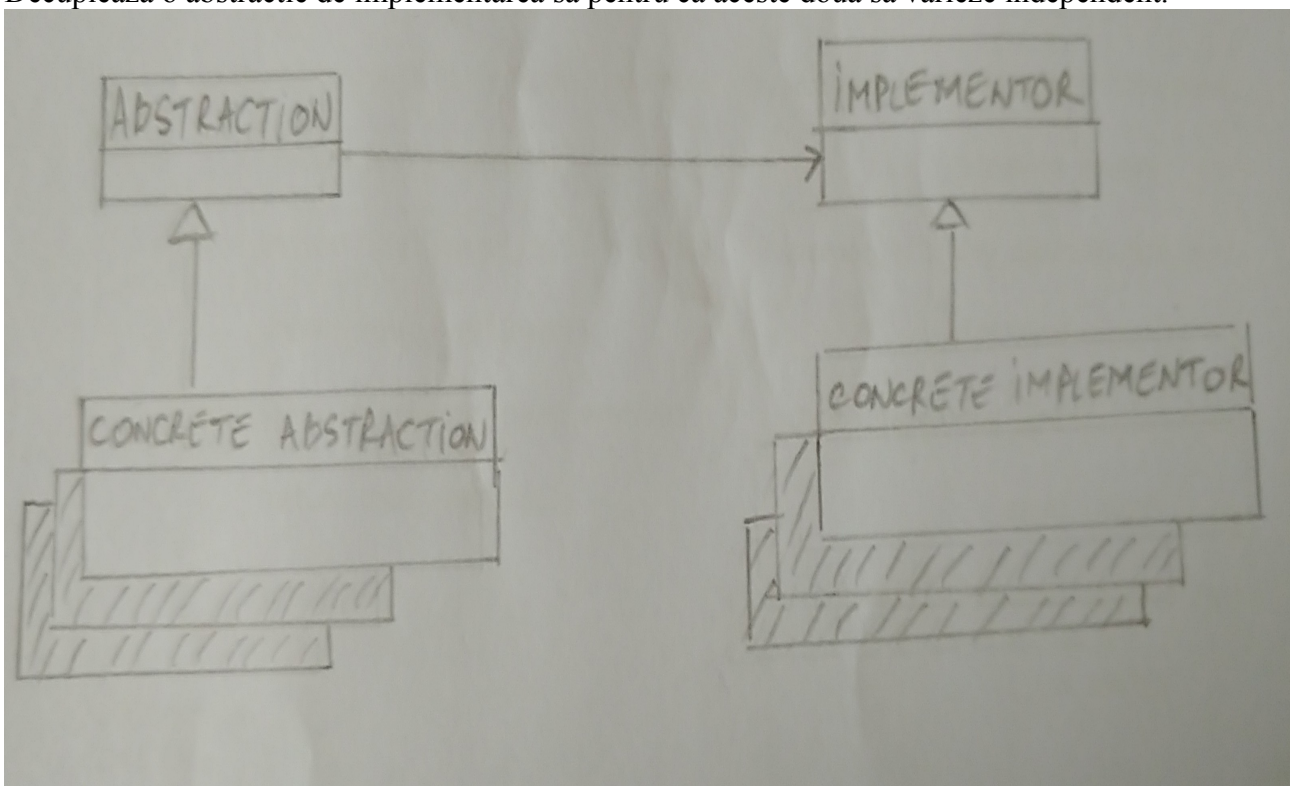
Protection Proxy

```
#include <iostream>
#include <string>
#include <memory>
class Person
{
public:
	virtual ~Person() = default;
	Person(const std::string& name) :name_{name} {}
	const std::string& getName()const
	{
		return name_;
	}
	virtual bool HasEarlyAccess()const = 0;
	virtual bool HasAnimalShowAccess()const = 0;
private:
	std::string name_;
};
class Member :public Person
{
public:
	Member(const std::string& name) :Person(name){}
	bool HasEarlyAccess()const override
	{
		return true;
	}
	bool HasAnimalShowAccess()const override
	{
		return true;
	}
};
class Guest :public Person
{
public:
	Guest(const std::string& name) :Person(name) {}
	bool HasEarlyAccess()const override
	{
		return false;
	}
	bool HasAnimalShowAccess()const override
	{
		return false;
	}
};
//Subject
class Zoo
{
public:
	virtual ~Zoo() = default;
	virtual void EnterZooEarly(Person* person)const = 0;
```

```
	virtual void AttendAnimalShow(Person* person)const = 0;
};
//Real subject
class RealZoo :public Zoo
{
public:
	void EnterZooEarly(Person* person)const override
	{
		std::cout << "Welcome "<< person->getName() <<", to our early access zoo
hours\n";
	}
	void AttendAnimalShow(Person* person)const override
	{
		std::cout << "Welcome " << person->getName() << ", to our animal show\n";
	}
};
//Proxy
class ProxyZoo :public Zoo
{
public:
	ProxyZoo(RealZoo* realZoo) :realZoo_{realZoo} {}
	void EnterZooEarly(Person* person)const override
	{
		if (person->HasEarlyAccess())
		{
			realZoo_->EnterZooEarly(person);
		}
		else
		{
			std::cout << "Sorry "<<person->getName()<<", you don't have early zoo
access privileges\n";
		}
	}
	void AttendAnimalShow(Person* person)const override
	{
		if (person->HasAnimalShowAccess())
		{
			realZoo_->AttendAnimalShow(person);
		}
		else
		{
			std::cout << "Sorry " << person->getName() << ", you don't have access
to the animal show\n";
		}
	}
private:
	RealZoo* realZoo_;
};
int main()
{
	auto member = std::make_unique<Member>("Dorel Amariei");
	auto guest = std::make_unique<Guest>("Mel Gibson");

	auto realZoo = std::make_unique<RealZoo>();
	auto proxyZoo = std::make_unique<ProxyZoo>(realZoo.get());

	proxyZoo->EnterZooEarly(member.get());
	proxyZoo->EnterZooEarly(guest.get());
	proxyZoo->AttendAnimalShow(member.get());
	proxyZoo->AttendAnimalShow(guest.get());
	return 0;
}
```

Bridge Pattern

Decupleaza o abstractie de implementarea sa pentru ca aceste doua sa varieze independent.




```
#include <iostream>
#include <string>
#include <memory>
class Artist
{
public:
	Artist(const std::string& artistName) :artistName_{artistName} {}
	void setBiography(const std::string& biography,const std::string& image
	,const std::string& url)
	{
		biography_ = biography;
		image_ = image;
		url_ = url;
	}
	const std::string& getBiography()const
	{
		return biography_;
	}
	const std::string& getImage()const
	{
		return image_;
	}
	const std::string& getUrl()const
	{
		return url_;
	}
	const std::string& getName()const
	{
		return artistName_;
	}
private:
	std::string biography_;
	std::string artistName_;
	std::string image_;
	std::string url_;
```

```
};
class Book
{
public:
	Book(const std::string& bookTitle) :bookTitle_{bookTitle} {}
	void setSynopsis(const std::string& synopsis, const std::string& image
	,const std::string& url)
	{
		synopsis_ = synopsis;
		image_ = image;
		url_ = url;
	}
	const std::string& getSynopsis()const
	{
		return synopsis_;
	}
	const std::string& getTitle()const
	{
		return bookTitle_;
	}
	const std::string& getImage()const
	{
		return image_;
	}
	const std::string& getUrl()const
	{
		return url_;
	}
private:
	std::string synopsis_;
	std::string bookTitle_;
	std::string image_;
	std::string url_;
};
//Implementor
class Resource
{
public:
	virtual ~Resource() = default;
	virtual const std::string& snippet()const = 0;
	virtual const std::string& title()const = 0;
	virtual const std::string& image()const = 0;
	virtual const std::string& url()const = 0;
};
//Concrete implementor
class ArtistResource:public Resource
{
public:
	ArtistResource(Artist* artist) :artist_{artist} {}
	const std::string& snippet()const override
	{
		return artist_->getBiography();
	}
	const std::string& title()const override
	{
		return artist_->getName();
	}
	const std::string& image()const override
	{
		return artist_->getImage();
	}
	const std::string& url()const override
	{
		return artist_->getUrl();
	}
```

```
private:
	Artist* artist_;
};
//Concrete implementor
class BookResource :public Resource
{
public:
	BookResource(Book* book) :book_{book} {}
	const std::string& snippet()const override
	{
		return book_->getSynopsis();
	}
	const std::string& title()const override
	{
		return book_->getTitle();
	}
	const std::string& image()const override
	{
		return book_->getImage();
	}
	const std::string& url()const override
	{
		return book_->getUrl();
	}
private:
	Book* book_;
};
//Abstraction
class View
{
public:
	virtual ~View() = default;
	View(Resource* resource) :resource_{resource} {}
	virtual void show()const = 0;
protected:
	Resource* resource_;  //HAS-A
};
//Concrete abstraction
class LongFormView :public View
{
public:
	LongFormView(Resource* resource) :View(resource){}
	void show()const override
	{
		std::cout << "This is the LONG FORM VIEW\n";
		std::cout << resource_->title() << '\n';
		std::cout << resource_->snippet() << '\n';
		std::cout << resource_->image() << '\n';
		std::cout << resource_->url() << '\n';
	}
};
//Concrete abstraction
class ShortFormView :public View
{
public:
	ShortFormView(Resource* resource) :View(resource){}
	void show()const override
	{
		std::cout << "This is the SHORT FORM VIEW\n";
		std::cout << resource_->title() << '\n';
		std::cout <<resource_->snippet() <<'\n';
		std::cout << resource_->image() << '\n';
		std::cout << resource_->url() << '\n';
	}
};
```

```
int main()
{
	auto artist = std::make_unique<Artist>("Michael Jackson");
	artist->setBiography("Born in 1959, Died in 2009", "michael.jpeg",
                                          "www.michael_jackson.com");
	auto book = std::make_unique<Book>("The name of the Rose");
	book->setSynopsis("A detective story set in an
                            abbey","rose.jpeg","www.umberto_eco.com");

	std::unique_ptr<Resource> artistResource =
                        std::make_unique<ArtistResource>(artist.get());
	std::unique_ptr<Resource> bookResource = std::make_unique<BookResource>(book.get());

	//all combination
	std::unique_ptr<View> longArtistView =
                 std::make_unique<LongFormView>(artistResource.get());
	std::unique_ptr<View> longBookView =
                   std::make_unique<LongFormView>(bookResource.get());
	std::unique_ptr<View> shortArtistView =
                std::make_unique<ShortFormView>(artistResource.get());
	std::unique_ptr<View> shortBookView =
                 std::make_unique<ShortFormView>(bookResource.get());

	longArtistView->show();
	longBookView->show();
	shortArtistView->show();
	shortBookView->show();
	return 0;
}
```

Template Method Pattern
Defineste scheletul unui algoritm intr-o metoda, amanand niste pasi pentru subclase.Lasa subclasele sa redefineasca anumiti pasi ai algoritmului fara sa ii schimbe structura.Clasa de baza declara niste *inlocuitori*(**hooks**) si clasele derivate ii implementeaza.

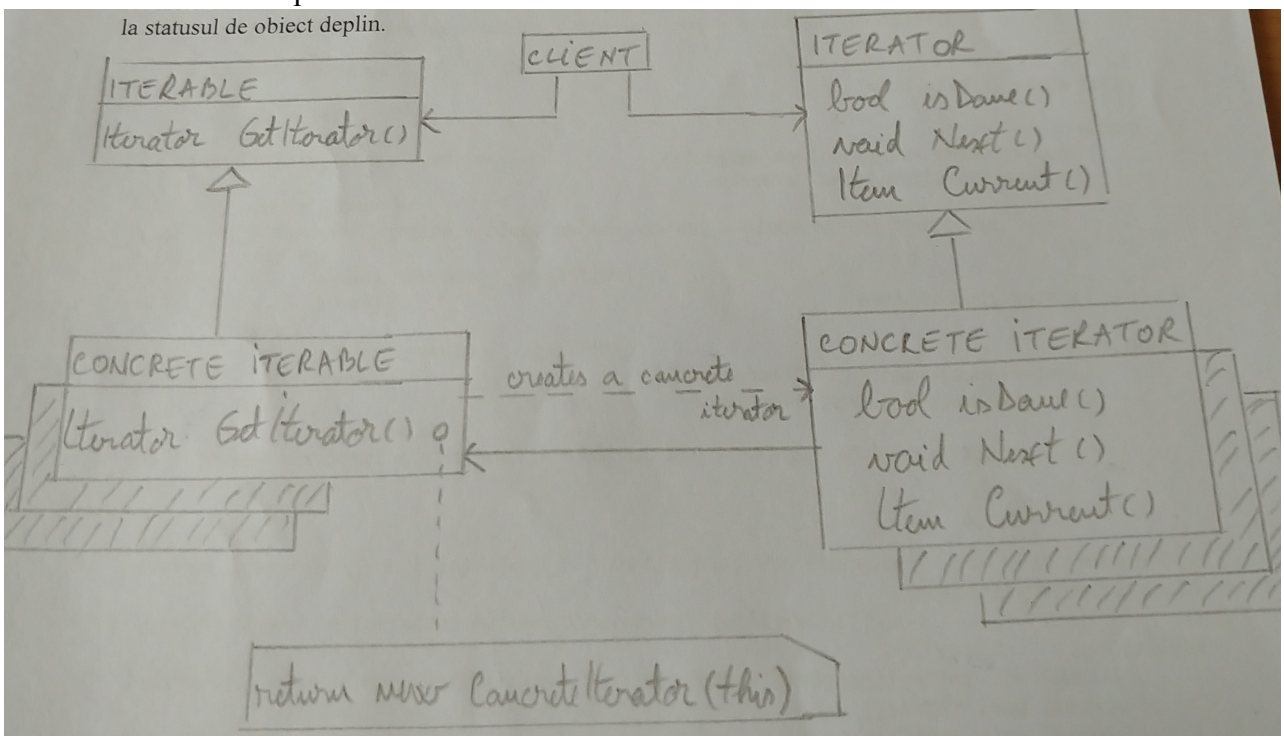
Design Principle 6:Acest pattern conduce la o structura de control inversata denumita *Principiul Hollywood* :”Nu ne suna pe noi, noi o sa te sunam pe tine”.Adica, clasa de baza apeleaza metodele unei subclase, nu invers.

```
#include <iostream>
#include <memory>
class Worker
{
public:
	virtual ~Worker() = default;
	void dailyRoutine()const   //Template method
	{
		getUp();
		eatBreakfast();
		goToWork();
		work();
		returnToHome();
		relax();
		sleep();
	}
private:
	void getUp()const { std::cout << "Get up\n"; }
	void eatBreakfast()const { std::cout << "Eat breakfast\n"; }
	void goToWork()const { std::cout << "Go to work\n"; }
	virtual void work()const = 0;     //hook
	void returnToHome()const { std::cout << "Return to home\n"; }
	//The hook relax() can have an empty implementation
	virtual void relax()const { std::cout << "Drink a beer\n"; }  //hook
```

```
	void sleep()const { std::cout << "Sleep\n"; }
};
class FireFighter :public Worker
{
private:
	void work()const override { std::cout << "Stop the fire\n"; }
};
class Lumberjack :public Worker
{
private:
	void work()const override { std::cout << "Cut the trees\n"; }
};
class Postman :public Worker
{
private:
	void work()const override { std::cout << "Deliver the mail\n"; }
};
class Manager :public Worker
{
private:
	void work()const override { std::cout << "Manage the firm\n"; }
	void relax()const override { std::cout << "Drink a tequila\n"; }
};
int main()
{
	std::unique_ptr<Worker> workers[]{ std::make_unique<FireFighter>(),
					   std::make_unique<Lumberjack>(),
					   std::make_unique<Postman>(),
					   std::make_unique<Manager>() };
	for (const auto& worker : workers)
	{
		worker->dailyRoutine();
		std::cout << "*******************************\n";
	}
	return 0;
}
```

Iterator Pattern

Traversarea elementelor unei colectii, fara ai expune reprezentarea.Promovarea traversarii colectiei la statusul de obiect deplin.



```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>
//Item
class Item
{
public:
	virtual ~Item() = default;
	Item(double weight) :weight_{weight} {}
	virtual void displayItem()const = 0;
	double getWeight()const
	{
		return weight_;
	}
protected:
	double weight_;
};
class Sword :public Item
{
public:
	Sword(double weight) :Item(weight){}
	void displayItem()const override
	{
		std::cout << "A sword - you can swing a sword - weight="<<weight_<<'\n';
	}
};
class Shield :public Item
{
public:
	Shield(double weight) :Item(weight){}
	void displayItem()const override
	{
		std::cout << "A shield - you can hold a shield - weight=" << weight_ << '\n';
	}
};
class Apple :public Item
{
public:
	Apple(double weight) :Item(weight){}
	void displayItem()const override
	{
		std::cout << "An apple - you can eat an apple - weight=" << weight_ << '\n';
	}
};
class Lighter :public Item
{
public:
	Lighter(double weight) :Item(weight){}
	void displayItem()const override
	{
		std::cout << "A lighter - you can make a fire with a lighter - weight=" <<
                                                                    weight_ << '\n';
	}
};
//Iterator
class InventoryIterator
{
public:
	virtual ~InventoryIterator() = default;
	virtual bool isDone() = 0;
	virtual void Next() = 0;
	virtual Item* Current()const = 0;
};
//Iterable
```

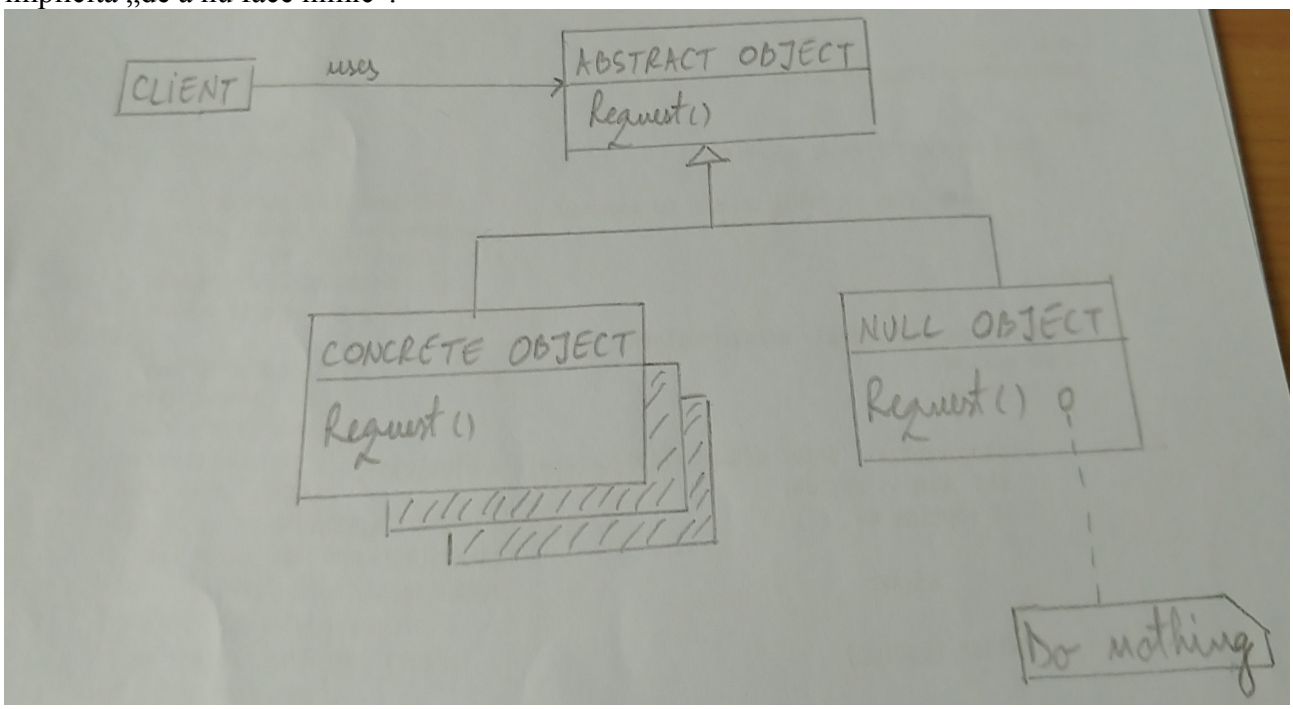
```
class Inventory
{
public:
	virtual ~Inventory() = default;
	virtual std::unique_ptr<InventoryIterator> getIterator() = 0;
};
//Concrete Iterable
class HandHeldInventory :public Inventory
{
public:
	HandHeldInventory(Item* right, Item* left) :right_{ right }, left_{ left } {}
	Item* getRight()const
	{
		return right_;
	}
	Item* getLeft()const
	{
		return left_;
	}
	//Creates a Concrete Iterator
	std::unique_ptr<InventoryIterator> getIterator() override;
private:
	Item* right_;
	Item* left_;
};
//Concrete Iterator
class HandHeldInventoryIterator :public InventoryIterator
{
public:
	HandHeldInventoryIterator(HandHeldInventory* const inventory) :inventory_{
                                                                    inventory } {}
	bool isDone() override
	{
		return index >= 2;
	}
	void Next()override
	{
		index++;
	}
	Item* Current()const override
	{
		switch (index)
		{
		case 0:
			return inventory_->getRight();
			break;
		case 1:
			return inventory_->getLeft();
			break;
		default:
			return nullptr;
		}
	}
private:
	int index = 0;
	HandHeldInventory* const inventory_;  //HAS-A Concrete Iterable
};
std::unique_ptr<InventoryIterator> HandHeldInventory::getIterator()
{
	return std::make_unique<HandHeldInventoryIterator>(this);
}
//Concrete Iterable
class BackpackInventory :public Inventory
{
public:
```

```
	BackpackInventory(const std::vector<Item*>& capacity)
	{
		for (const auto& item:capacity)
		{
			capacity_.push_back(item);
		}
	}
	Item* getElement(const int index)
	{
		if (index < capacity_.size())
		{
			return capacity_[index];
		}
		else
		{
			return nullptr;
		}
	}
	//Creates a Concrete Iterator
	std::unique_ptr<InventoryIterator> getIterator() override;
private:
	std::vector<Item*> capacity_;
};
//Concrete Iterator
class BackpackInventoryIterator :public InventoryIterator
{
public:
	BackpackInventoryIterator(BackpackInventory* const inventory) :inventory_{inventory}
                                                                                {}
	bool isDone() override
	{
		if (inventory_->getElement(index) == nullptr)
		{
			return true;
		}
		capacity -= inventory_->getElement(index)->getWeight();
		return capacity < 0; ;
	}
	void Next()override
	{
		index++;
	}
	Item* Current()const override
	{
		return inventory_->getElement(index);
	}
private:
	double capacity = 40;
	int index = 0;;
	BackpackInventory* const inventory_;  //HAS-A Concrete Iterable
};
std::unique_ptr<InventoryIterator> BackpackInventory::getIterator()
{
	return std::make_unique<BackpackInventoryIterator>(this);
}
void displayInventory(Inventory* inventory)
{
	std::cout << "Things in the inventory....\n";
	std::unique_ptr<InventoryIterator> iterator = inventory->getIterator();
	while (!iterator->isDone())
	{
		auto item = iterator->Current();
		if (item != nullptr)
		{
			item->displayItem();
```

```
            }
            iterator->Next();
        }
}
int main()
{
        auto sword = std::make_unique<Sword>(15);
        auto shield = std::make_unique<Shield>(49);
        displayInventory(std::make_unique<HandHeldInventory>(sword.get(),
                                                    shield.get()).get());
        auto apple1 = std::make_unique<Apple>(0.3);
        auto apple2 = std::make_unique<Apple>(0.4);
        auto apple3 = std::make_unique<Apple>(0.5);
        auto lighter = std::make_unique<Lighter>(0.2);
        std::vector<Item*> backpackItems
        {sword.get(),shield.get(),apple1.get(),apple2.get(),apple3.get(),lighter.get()};
        auto ascending = [](Item* a, Item* b) { return a->getWeight() < b->getWeight();};
        std::sort(backpackItems.begin(),backpackItems.end(),ascending);
        displayInventory(std::make_unique<BackpackInventory>(backpackItems).get());
        return 0;
}
```

Null Object Pattern

Incapsuleaza absenta unui obiect prin furnizarea unei alternative convenabile ce ofera o comportare implicita „de a nu face nimic”.

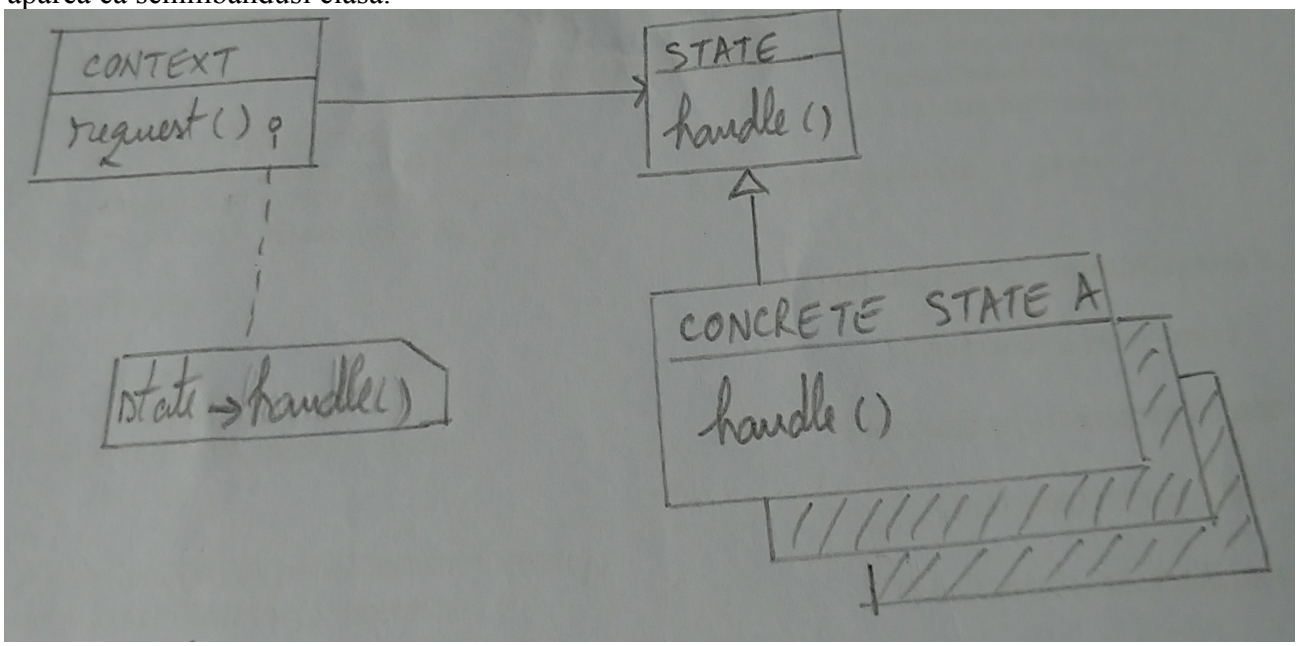



```
#include <iostream>
#include <memory>
//Abstract object
class Animal
{
public:
        virtual ~Animal() = default;
        virtual void makeSound()const = 0;       //Request()
};
//Concrete object
class Dog :public Animal
{
public:
        void makeSound()const override
```

```
	{
		std::cout << "Woof woof\n";
	}
};
//Concrete object
class Cat :public Animal
{
public:
	void makeSound()const override
	{
		std::cout << "Miau miau\n";
	}
};
//Concrete object
class Cow :public Animal
{
public:
	void makeSound()const override
	{
		std::cout << "Muu muu\n";
	}
};
//Null object
class NullAnimal :public Animal
{
public:
	void makeSound()const override
	{
		std::cout << "Null animal: no sound\n";
	}
};
int main()
{
	std::unique_ptr<Animal> animal(nullptr);
	char option;
	while (true)
	{
		std::cout << "Enter d(dog),c(cat),w(cow) -- e(exit):";
		std::cin >> option;
		if (option == 'e')
		{
			break;
		}
		switch (option)
		{
		case 'd':
			animal = std::make_unique<Dog>();
			animal->makeSound();
			break;
		case 'c':
			animal = std::make_unique<Cat>();
			animal->makeSound();
			break;
		case 'w':
			animal = std::make_unique<Cow>();
			animal->makeSound();
			break;
		default:
			animal = std::make_unique<NullAnimal>();
			animal->makeSound();
		}
	}
	return 0;
}
```

State Pattern

Permite unui obiect sa isi schimbe comportarea atunci cand starea sa interna se schimba.Obiectul va aparea ca schimbandusi clasa.




```
//GumballMachine.h
#pragma once
#include "State.h"
class GumballMachine
{
	friend class NoQuarterState;
	friend class HasQuarterState;
	friend class SoldState;
	friend class SoldOutState;
	friend class WinnerState;
public:
	GumballMachine(int);
	void insertQuarter();
	void ejectQuarter();
	void turnCrank();
	void refill(int);
	State* getSoldOutState()const;
	State* getNoQuarterState()const;
	State* getHasQuarterState()const;
	State* getSoldState()const;
	State* getWinnerState()const;
	int getCount()const;
private:
	void setState(State*); //clase  prietene pt. aceasta
	void releaseBall(); //clase prietene pt. aceasta
	std::unique_ptr<State> soldOutState;
	std::unique_ptr<State> noQuarterState;
	std::unique_ptr<State> hasQuarterState;
	std::unique_ptr<State> soldState;
	std::unique_ptr<State> winnerState;
	State* state_ = soldOutState.get();
	int count = 0;
};
//GumballMachine.cpp
#include "GumballMachine.h"
#include "SoldOutState.h"
#include "NoQuarterState.h"
#include "HasQuarterState.h"
#include "SoldState.h"
```

```
#include "WinnerState.h"
GumballMachine::GumballMachine(int numberOfGumballs)
{
	soldOutState = std::make_unique<SoldOutState>(this);
	noQuarterState = std::make_unique<NoQuarterState>(this);
	hasQuarterState = std::make_unique<HasQuarterState>(this);
	soldState = std::make_unique<SoldState>(this);
	winnerState = std::make_unique<WinnerState>(this);
	count = numberOfGumballs;
	if (numberOfGumballs > 0)
	{
		state_ = noQuarterState.get();
	}
}
void GumballMachine::insertQuarter()
{
	state_->insertQuarter();
}
void GumballMachine::ejectQuarter()
{
	state_->ejectQuarter();
}
void GumballMachine::turnCrank()
{
	state_->turnCrank();
	state_->dispense();
}
void GumballMachine::setState(State* state)
{
	state_ = state;
}
void GumballMachine::releaseBall()
{
	std::cout << "A gumball comes rolling out the slot...\n";
	if (count != 0)
	{
		count--;
	}
}
void GumballMachine::refill(int numberOfGumballs)
{
	count = numberOfGumballs;
	state_ = noQuarterState.get();
}
State* GumballMachine::getSoldOutState()const
{
	return soldOutState.get();
}
State* GumballMachine::getNoQuarterState()const
{
	return noQuarterState.get();
}
State* GumballMachine::getHasQuarterState()const
{
	return hasQuarterState.get();
}
State* GumballMachine::getSoldState()const
{
	return soldState.get();
}
State* GumballMachine::getWinnerState()const
{
	return winnerState.get();
}
int GumballMachine::getCount()const
```

```
{
	return count;
}
```
//State.h
```
#pragma once
#include <iostream>
class State
{
public:
	virtual void insertQuarter() = 0;
	virtual void ejectQuarter() = 0;
	virtual void turnCrank() = 0;
	virtual void dispense() = 0;
};
```
//NoQuarterState.h
```
#pragma once
#include "State.h"
#include "GumballMachine.h"
class NoQuarterState :public State
{
public:
	NoQuarterState(GumballMachine* const);
	void insertQuarter() override;
	void ejectQuarter() override;
	void turnCrank() override;
	void dispense() override;
private:
	GumballMachine* const gumballMachine_;
};
```
//NoQuarterState.cpp
```
#include "NoQuarterState.h"
NoQuarterState::NoQuarterState(GumballMachine* const gumballMachine) :gumballMachine_{
                                                                    gumballMachine } {}
void NoQuarterState::insertQuarter()
{
	std::cout << "You inserted a quarter\n";
	gumballMachine_->setState(gumballMachine_->getHasQuarterState());
}
void NoQuarterState::ejectQuarter()
{
	std::cout << "You haven't inserted a quarter\n";
}
void NoQuarterState::turnCrank()
{
	std::cout << "You turned but there is no quarter\n";
}
void NoQuarterState::dispense()
{
	std::cout << "You need to pay first\n";
}
```
//HasQuarterState.h
```
#pragma once
#include "State.h"
#include "GumballMachine.h"
class HasQuarterState :public State
{
public:
	HasQuarterState(GumballMachine* const);
	void insertQuarter() override;
	void ejectQuarter() override;
	void turnCrank() override;
	void dispense() override;
private:
	GumballMachine* const gumballMachine_;
};
```

//HasQuarterState.cpp

```
#include "HasQuarterState.h"
HasQuarterState::HasQuarterState(GumballMachine* const gumballMachine) :gumballMachine_{
                                                                        gumballMachine } {}
void HasQuarterState::insertQuarter()
{
	std::cout << "You can't insert another quarter\n";
}
void HasQuarterState::ejectQuarter()
{
	std::cout << "Quarter returned\n";
	gumballMachine_->setState(gumballMachine_->getNoQuarterState());
}
void HasQuarterState::turnCrank()
{
	std::cout << "You turned...\n";
	int winner = rand() % 10;
	if (winner == 0 && gumballMachine_->getCount() > 1)
	{
		gumballMachine_->setState(gumballMachine_->getWinnerState());
	}
	else
	{
		gumballMachine_->setState(gumballMachine_->getSoldState());
	}
}
void HasQuarterState::dispense()
{
	std::cout << "No gumball dispensed\n";
}
```

//SoldState.h

```
#pragma once
#include "State.h"
#include "GumballMachine.h"
class SoldState :public State
{
public:
	SoldState(GumballMachine* const);
	void insertQuarter() override;
	void ejectQuarter() override;
	void turnCrank() override;
	void dispense() override;
private:
	GumballMachine* const gumballMachine_;
};
```

//SoldState.cpp

```
#include "SoldState.h"
SoldState::SoldState(GumballMachine* const gumballMachine):gumballMachine_{ gumballMachine }
                                                                                        {}
void SoldState::insertQuarter()
{
	std::cout << "Please wait, we are already giving you a gumball\n";
}
void SoldState::ejectQuarter()
{
	std::cout << "Sorry, you already turned the crank\n";
}
void SoldState::turnCrank()
{
	std::cout << "Turning twice dosn't get you another gumball\n";
}
void SoldState::dispense()
{
	gumballMachine_->releaseBall();
```

```
	if (gumballMachine_->getCount() > 0)
	{
		gumballMachine_->setState(gumballMachine_->getNoQuarterState());
	}
	else
	{
		std::cout << "Oops, out of gumballs!\n";
		gumballMachine_->setState(gumballMachine_->getSoldOutState());
	}
}
```

//SoldOutState.h

```
#pragma once
#include "State.h"
#include "GumballMachine.h"
class SoldOutState :public State
{
public:
	SoldOutState(GumballMachine* const);
	void insertQuarter() override;
	void ejectQuarter() override;
	void turnCrank() override;
	void dispense() override;
private:
	GumballMachine* const gumballMachine_;
};
```

//SoldOutState.cpp

```
#include "SoldOutState.h"
SoldOutState::SoldOutState(GumballMachine* const gumballMachine) :gumballMachine_{
                                                                   gumballMachine } {}
void SoldOutState::insertQuarter()
{
	std::cout << "You can't insert a quarter, the machine is sold out\n";
}
void SoldOutState::ejectQuarter()
{
	std::cout << "You can't eject, you haven't inserted a quarter yet\n";
}
void SoldOutState::turnCrank()
{
	std::cout << "You turned, but there are no gumballs\n";
}
void SoldOutState::dispense()
{
	std::cout << "No gumball dispensed\n";
}
```

//WinnerState.h

```
#pragma once
#include "State.h"
#include "GumballMachine.h"
class WinnerState :public State
{
public:
	WinnerState(GumballMachine* const);
	void insertQuarter() override;
	void ejectQuarter() override;
	void turnCrank() override;
	void dispense() override;
private:
	GumballMachine* const gumballMachine_;
};
```

//WinnerState.cpp

```
#include "WinnerState.h"
WinnerState::WinnerState(GumballMachine* const gumballMachine) :gumballMachine_{
                                                                  gumballMachine } {}
void WinnerState::insertQuarter()
```

```
{
	std::cout << "Please wait, we are already giving you a gumball\n";
}
void WinnerState::ejectQuarter()
{
	std::cout << "Sorry, you already turned the crank\n";
}
void WinnerState::turnCrank()
{
	std::cout << "Turning twice dosn't get you another gumball\n";
}
void WinnerState::dispense()
{
	std::cout << "YOU ARE A WINNER!!! You get two gumballs for your quarter\n";
	gumballMachine_->releaseBall();
	if (gumballMachine_->getCount() == 0)
	{
		gumballMachine_->setState(gumballMachine_->getSoldOutState());
	}
	else
	{
		gumballMachine_->releaseBall();
		if (gumballMachine_->getCount() > 0)
		{
			gumballMachine_->setState(gumballMachine_->getNoQuarterState());
		}
		else
		{
			std::cout << "Oops, out of gumballs!!!\n";
			gumballMachine_->setState(gumballMachine_->getSoldOutState());
		}
	}
}
//Source.cpp
#include <iostream>
#include <memory>
#include <ctime>
#include "GumballMachine.h"
void showMachine(const GumballMachine& machine)
{
	std::cout << "Gumball Machine Model #2019\n";
	std::cout << "Inventory:"<<machine.getCount()<<" gumballs\n";
	std::cout << "Machine is waiting for a quarter\n";
	std::cout << "_____________________________________________\n";
}
int main()
{
	srand(time(0));
	GumballMachine gumballMachine(3);
	gumballMachine.insertQuarter();
	gumballMachine.turnCrank();
	showMachine(gumballMachine);

	gumballMachine.insertQuarter();
	gumballMachine.ejectQuarter();
	gumballMachine.turnCrank();
	showMachine(gumballMachine);

	gumballMachine.insertQuarter();
	gumballMachine.turnCrank();
	showMachine(gumballMachine);

	gumballMachine.turnCrank();
	return 0;
}
```