Search . . .

06

Dušičky edition 2.11.2023

Implementace chování aplikace

- Iterátor
- Chain of responsibility
- Strategy
- Visitor
- Observer
- Template method
- State
- Memento
- Interpreter

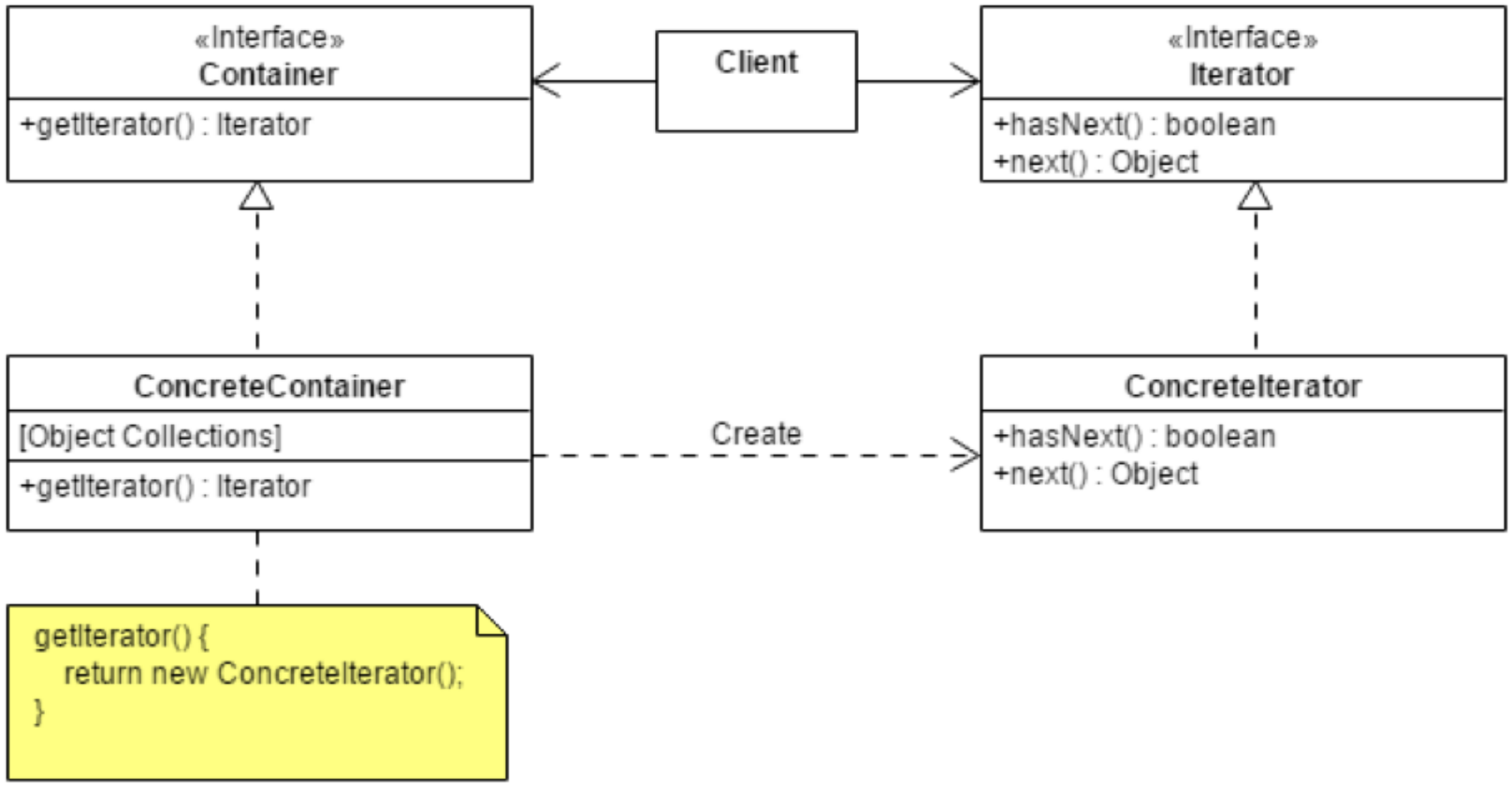
Next >

Objektové Modelování - podzim 2022

kadlecd@fel.cvut.cz, www.czm.fel.cvut.cz

06 ITERÁTOR

Při procházení datové struktury jsme pevně svázáni s její implementací. Pomocí iterátoru budeme abstrahovat samotný algoritmus procházení (tzv. iterování) od implementace datové struktury. Klient používá iterátor tak, že pouze řídí procházení datové struktury pomocí metod ***next()*** a ***hasNext()***, ale vůbec nemusí znát vlastnosti ani implementaci datové struktury. Iterátor si udržuje stav procházení, aktuální položku a je schopen posunout se na další prvek.



06 ITERÁTOR

Container je datová struktura, kterou se snažíme procházet. Container vrací jednotlivé iterátory pomocí kterých může klient tuto datovou strukturu procházet.

```
public interface Container {
   Iterator getIterator();
}
```

Všechny interátory by měly mít stejné rozhraní s metodami pro posun iterátoru o jeden prvek a metodu vracející jestli existuje další prvek pro pro procházení. Některé iterátory mají ještě metodu *remove()*, která umožňuje aktuální prvek odebrat. Ne všechny datové struktury umožňují modifikaci v průběhu procházení.

```
public interface Iterator {
   boolean hasNext();
   Object next();
}
```

06 ITERÁTOR

MonsterHouse je datová struktura na správu strašidel, interně je reprezentována přes pole. Vrací iterátor na procházení přes strašidla v

```
public class MonstersHouse implements Container {
   public String monsters[] = {
           "Dracula",
           "Frankenstein",
           "Werewolf",
           "Headless Horseman",
           "Zombie",
           "Witch",
           "Gremlin"};
   public Iterator getIterator() {
       return new NameIterator();
   }
   private class NameIterator implements Iterator {
       int index = 0;
       public boolean hasNext() {
           return index < monsters.length;
       }
       public Object next() {
           if (hasNext()) {
               return monsters[index++];
           }
           return null;
       }
   }
}
```

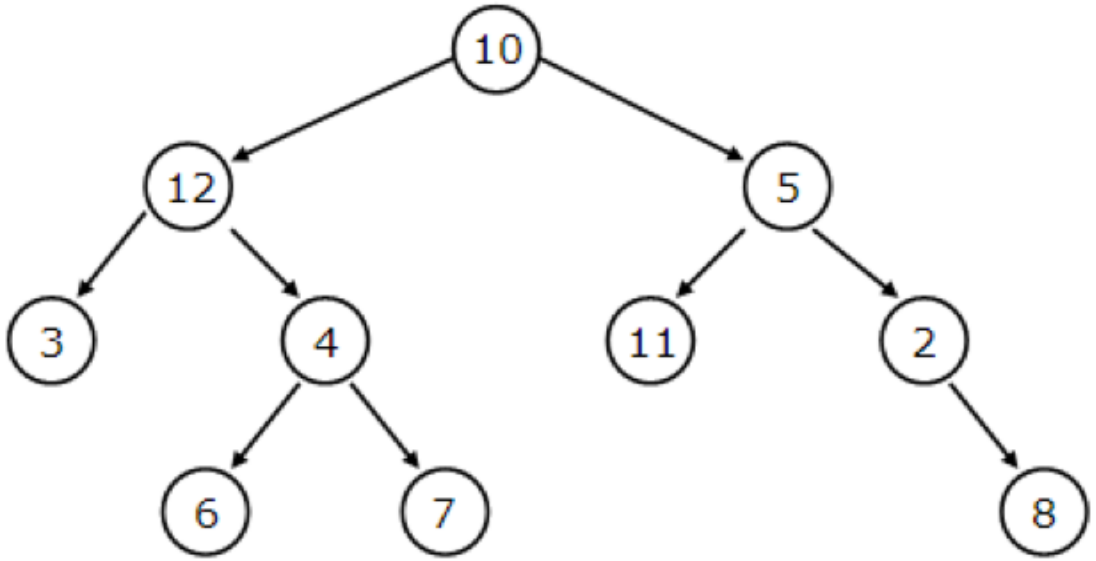
06 ITERÁTOR

MonsterHouse procházíme (iterujeme) pomocí metod *next()* a *hasNext()*.

```
public class Client {

   public static void main(String[] args) {
MonstersHouse monsters = new MonstersHouse();
for (Iterator it = monsters.getIterator(); it.hasNext(); ) {
            String name = (String) it.next();
            System.out.println(name);
        }
   }
}
```

06 ITERÁTOR

Jako další příklad implementujeme iterátor, který prochází strom, čtyřmi různými způ



Levelorder tree traversal
10, 12, 5, 3, 4, 11, 2, 6, 7, 8

Inorder tree traversal
3, 12, 6, 4, 7, 10, 11, 5, 2, 8

Preorder tree traversal
10, 12, 3, 4, 6, 7, 5, 11, 2, 8

Postorder tree traversal
3, 6, 7, 4, 12, 11, 8, 2, 5, 10

06 ITERÁTOR

```
public class BTree<T> {
    BTreeNode<T> root;
    BTree(BTreeNode<T> root){
        this.root = root;
    }
    Iterator<T> getInOrderIterator(){
        return new InorderIterator<T>(root);
    }
    Iterator<T> getPreOrderIterator(){
        return new PreorderIterator<T>(root);
    }
    Iterator<T> getPostOrderIterator(){
        return new PostorderIterator<T>(root);
    }
}
```

```
class BTreeNode<T> {
    T element;
    BTreeNode<T> leftChild, rightChild;
    BTreeNode( T element, BTreeNode<T> leftChild, BTreeNode<T> rightChild ){
        this.element = element;
        this.leftChild = leftChild;
        this.rightChild = rightChild;
    }
    T getElement(){ return element; }
    BTreeNode<T> getLeft(){ return leftChild; }
    BTreeNode<T> getRight(){ return rightChild; }
}
```

```
class StackElement<T> {
    BTreeNode<T> btreeNode;
    boolean visited;
    StackElement( BTreeNode<T> btreeNode, boolean visited ){
        this.btreeNode = btreeNode;
        this.visited = visited;
    }
    BTreeNode<T> getNode(){    return btreeNode; }
    boolean isVisited(){ return visited; }
}
```

06 ITERÁTOR

Naimplementujeme *InorderIterator.*

Pracujeme tak, že procházíme stromem a do zásobníku vkládáme elementy v pořadí pravý child, aktualní a levý child. Aktuální nastavujeme jako navštívený a pak pokračujeme prvkem z vrcholu zásobníku… ve chvíli, kdy máme na vrchu zásobníku navštívený prvek, tak je to náš aktuální prvek

```
class InorderIterator<T> implements Iterator<T>{
    Stack<StackElement<T>> iteratorStack;
    InorderIterator(BTreeNode<T> root){
        iteratorStack = new Stack<StackElement<T>>();
        iteratorStack.push(new StackElement<T>(root, false));
    }
    public boolean hasNext() {
        return !iteratorStack.isEmpty();
    }
    public T next() {
        // Repeat until we have a visited node on top of the stack.
        while( !iteratorStack.peek().isVisited() ){
            // Get current node from top of the stack.
            BTreeNode<T> curNode = iteratorStack.pop().getNode();
            // If right child is present, push it on the stack as not-visited.
            if( curNode.getRight() != null )
                iteratorStack.push( new StackElement<T>(curNode.getRight(), false) );
            // Push current node on the stack as visited.
            iteratorStack.push(new StackElement<T>(curNode, true));
            // If left child is present, push it on the stack as not-visited.
            if( curNode.getLeft() != null )
                iteratorStack.push(new StackElement<T>(curNode.getLeft(), false));
        }
//Pop the node on top of the stack (this node is a visited node) and return its data.
        return iteratorStack.pop().getNode().getElement();
    }
    public void remove() {}
}
```

06 ITERÁTOR

PreorderIterator

```
...
        while( !iteratorStack.peek().isVisited() ){
            BTreeNode<T> curNode = iteratorStack.pop().getNode();
        // If right child is present, push it on the stack as not-visited.
            if( curNode.getRight() != null )
                iteratorStack.push( new StackElement<T>(curNode.getRight(), false) );
            if( curNode.getLeft() != null )
                iteratorStack.push( new StackElement<T>(curNode.getLeft(), false) );
            iteratorStack.push(new StackElement<T>(curNode, true));
        }
...
```

PostorderIterator

```
...
        while( !iteratorStack.peek().isVisited() ){
            BTreeNode<T> curNode = iteratorStack.pop().getNode();
            iteratorStack.push(new StackElement<T>(curNode, true));
            if( curNode.getRight() != null )
                iteratorStack.push( new StackElement<T>(curNode.getRight(), false) );
            if( curNode.getLeft() != null )
                iteratorStack.push( new StackElement<T>(curNode.getLeft(), false) );
        }
...
```

06 ITERÁTOR

Klient, ze kterého použijeme iterátor =>

Použití je vždy stejné nehledě na to, jakým způsobem strom procházíme

```
public class BTreeClient {
   public static void main(String[] args) {
       BTree<Integer> testTree = setupTree();
       //Inorder tree traversal
       Iterator<Integer> inOrderIterator = testTree.getInOrderIterator();
       while( inOrderIterator.hasNext() )
           System.out.print(inOrderIterator.next()+".");

       //Preorder tree traversal
       Iterator<Integer> preOrderIterator = testTree.getPreOrderIterator();
       while( preOrderIterator.hasNext() )
           System.out.print(preOrderIterator.next()+".");

       //Postorder tree traversal
       Iterator<Integer> postOrderIterator = testTree.getPostOrderIterator();
       while( postOrderIterator.hasNext() )
           System.out.print(postOrderIterator.next()+".");
   }
   static BTree<Integer> setupTree(){
     BTreeNode<Integer> n6 = new BTreeNode<Integer>(6, null, null);
     BTreeNode<Integer> n7 = new BTreeNode<Integer>(7, null, null);
     BTreeNode<Integer> n4 = new BTreeNode<Integer>(4, n6, n7);
     BTreeNode<Integer> n3 = new BTreeNode<Integer>(3, null, null);
     BTreeNode<Integer> n12 = new BTreeNode<Integer>(12, n3, n4);
     BTreeNode<Integer> n8 = new BTreeNode<Integer>(8, null, null);
     BTreeNode<Integer> n2 = new BTreeNode<Integer>(2, null, n8);
     BTreeNode<Integer> n11 = new BTreeNode<Integer>(11, null, null);
     BTreeNode<Integer> n5 = new BTreeNode<Integer>(5, n11, n2);
     BTreeNode<Integer> n10 = new BTreeNode<Integer>(10, n12, n5);
   return new BTree<Integer>(n10);
   }
}
```

06 ITERÁTOR

Funkcionální programování

Ve funkcionálním programování je design pattern Iterator často nahrazeny Higher Order Funkcemi a transformacemi immutable dat nebo rekurzí. Tyto přístupy kladou důraz na deklarativní zpracování dat namísto explicitní iterace.

```java
List<Integer> largeList = List.of(...); // Assume a large list

// Lazy evaluation: no elements are processed until terminal operation
largeList.stream()
         .map(x -> x * 2)
         .filter(x -> x > 10)
         .forEach(System.out::println);
```

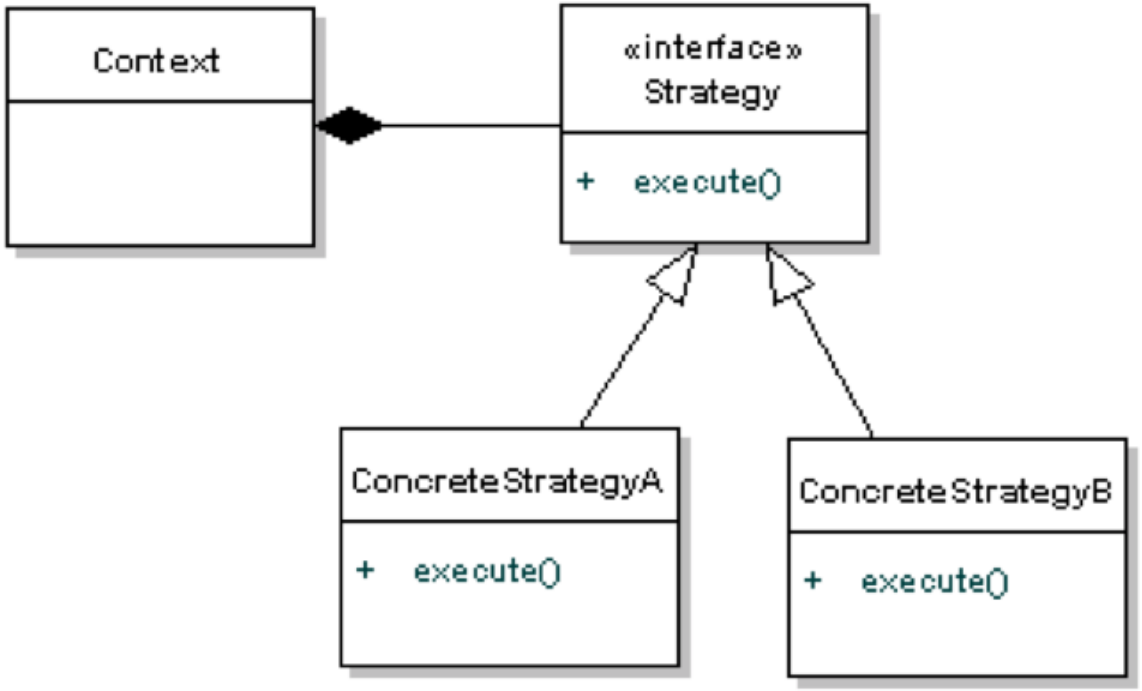
```java
// Recursive summing function in Java
public static int sum(List<Integer> numbers) {
    return sumHelper(numbers, 0);
}

private static int sumHelper(List<Integer> numbers, int acc) {
    if (numbers.isEmpty()) return acc;
    return sumHelper(numbers.subList(1, numbers.size()), acc + numbers.get(0));
}
```

Bude vysvětleno v dalších přednáškách

06 STRATEGY

Změna chování (nebo algoritmu) v průběhu běhu programu. Vytváříme tzv. context, který se chová různě podle toho, jakou má aktuálně nastavenou strategii. Strategy se používá, když pro realizaci jednoho tasku chceme flexibilně přepínat mezi různými algoritmy.

Analogie s funkcionálním programováním, kde předáváme funkci jako parameter do jiné funkce




```
interface Strategy {
   public void execute();
}
```

06 STRATEGY

Máme různé policejní okrsky a pro ně chceme nastavit různé strategie při kontrole řidičů


```java
interface PoliceStrategy {
    static final int speedLimit = 66;
    public void execute(int speed);
}
```

```java
class NicePoliceStrategy implements PoliceStrategy {
    public void execute(int speed) {
        if (speed > speedLimit + 10)
            System.out.println("Nice police: You will pay fine 50USD");
    }
}
class HardPoliceStrategy implements PoliceStrategy {
    public void execute(int speed) {
        if (speed > speedLimit)
            System.out.println("Hard police: You will pay fine 50USD");
    }
}
class LazyPoliceStrategy implements PoliceStrategy {
    public void execute(int speed) {
        if (speed > speedLimit && Math.random() > 0.5)
            System.out.println("Lazy police: You will pay fine 50USD");
        else
            System.out.println("Lazy police: Let go");
    }
}
```

06 STRATEGY

Náš policejní okrsek je reprezentován pomocí *Context* objektu.

```
public class Context {
   PolicyStrategy strategy;
   public void setStrategy(PolicyStrategy strategy) {
       this.strategy = strategy;
   }
   public Context(PolicyStrategy strategy){
       this.strategy = strategy;
   }
   public void patrol(int speed){ strategy.execute(speed);}
}
```

```
public class Client {
   public static void main(String [] args) {
       PolicyStrategy hardStrategy = new HardPoliceStrategy();
       PolicyStrategy niceStrategy = new NicePoliceStrategy();
       PolicyStrategy lazyStrategy = new LazyPoliceStrategy();

       Context newYorkPoliceDepartment = new Context(hardStrategy);
       newYorkPoliceDepartment.patrol(70);

       Context czechPoliceDepartment = new Context(niceStrategy);
       czechPoliceDepartment.patrol(70); //Does nothing

       czechPoliceDepartment.setStrategy(lazyStrategy);
       czechPoliceDepartment.patrol(150);
       czechPoliceDepartment.patrol(150);
       czechPoliceDepartment.patrol(150);
   }}
```

```
Hard police: You will pay fine 50USD
Nice police: Let go
Lazy police: You will pay fine 50USD
Lazy police: Let go
Lazy police: You will pay fine 50USD
```

06 STRATEGY

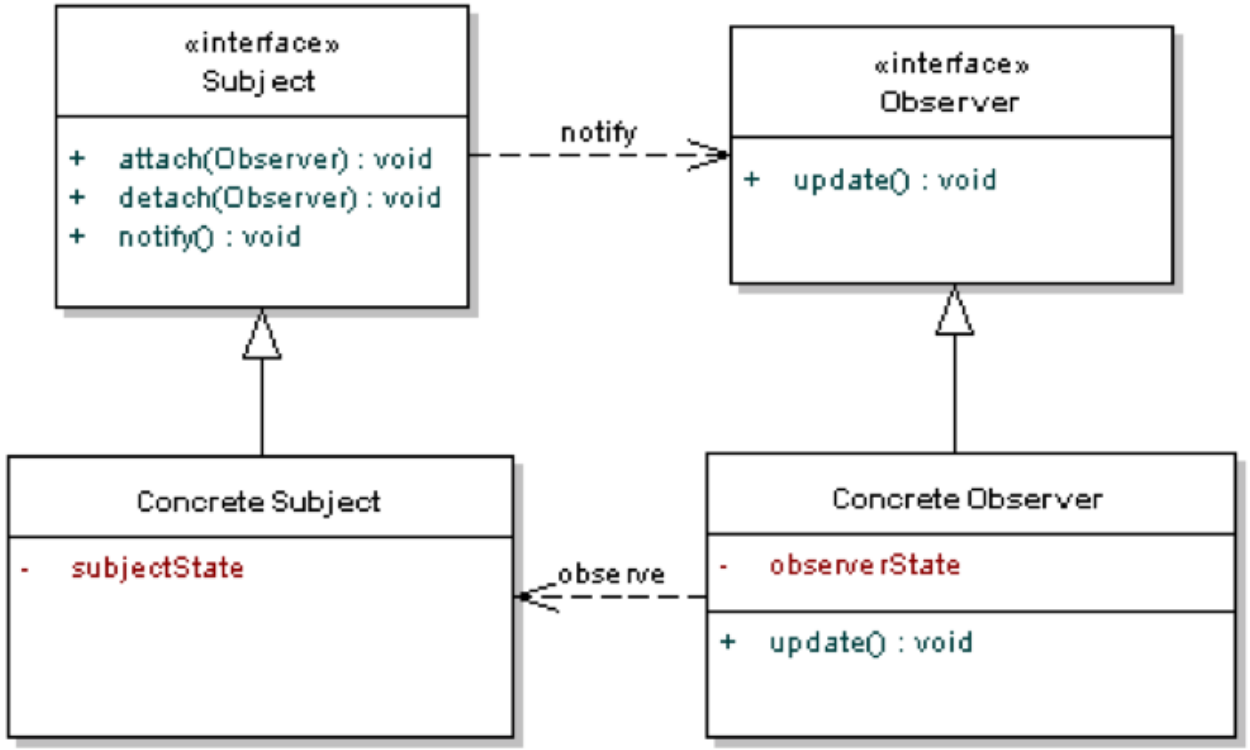
Funkcionální programování

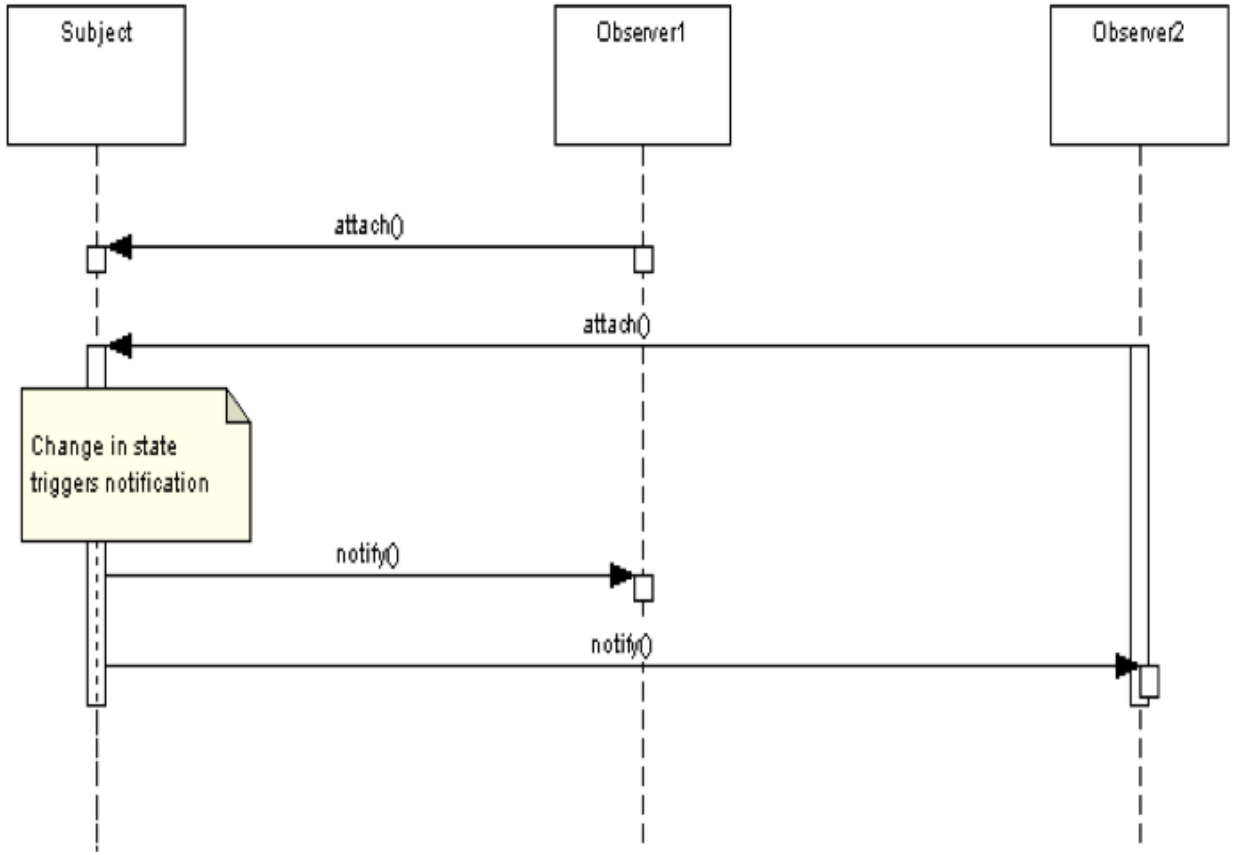
Injectujeme funkci podle zvolené strategie….

```
class PricingStrategy {
    public static double walkInPricing(CustomerOrder order) {
        // Calculate the price per item in order according to the walk-in pricing
        double totalPrice = 10;
        return totalPrice;
    }

    public static double onlinePricing(CustomerOrder order) {
        // Calculate the price per item in order according to the online pricing
        double totalPrice = 20;
        return totalPrice;
    }

    public static double getTotalPrice(Function<CustomerOrder, Double> pricingStrategy, CustomerOrder order) {
        return pricingStrategy.apply(orders);
    }

    public static void main(String[] args) {
        CustomerOrder order = new CustomerOrder();
        String customerType = "walkIn";  // or "online" based on the customer

        Function<CustomerOrder, Double> pricing;
        switch (customerType) {
            case "walkIn":
                pricing = PricingStrategy::walkInPricing;
                break;
            case "online":
                pricing = PricingStrategy::onlinePricing;
                break;
            default:
                throw new IllegalArgumentException("Invalid customer type");
        }

        double totalPrice = getTotalPrice(pricing, order);
    }
}

class CustomerOrder {
    // Define order items
}
```

06 OBSERVER

Propojení změnu stavu komponenty s komponentami reagujícími na tuto změnu. Objekt nazvaný **subject** si udržuje seznam objektů **observers**, které notifikuje pokaždé, když dojde ke změně stavu subjectu. Notifikace probíhá zavoláním metody na observeru. Tento pattern je často používán při implementaci GUI frameworků (MVC pattern).





06 OBSERVER

Vytvoříme tři různé observers, každý z nich vypisuje stav subjektu v jiném formátu.

```
public class BinaryObserver extends Observer{
   public BinaryObserver(Subject subject){
       super(subject);
   }
   public void update() {
       System.out.println("Binary:" + Integer.toBinaryString(subject.getState()));
   }
}

public class OctalObserver extends Observer{
   public OctalObserver(Subject subject){ super(subject);}
   public void update() {
      S ystem.out.println( "Octal:"+Integer.toOctalString(subject.getState()));
   }
}
public class HexaObserver extends Observer{
   public HexaObserver(Subject subject){ super(subject);}
   public void update(){
System.out.println( "Hex:"+Integer.toHexString(subject.getState()).toUpperCase());
}
}
```

06 OBSERVER

Nadefinujeme abstraktní třídu Observer, která má metodu ***update()*** přes kterou notifikuje *Subject* jednotlivé observers.

```
public abstract class Observer {
    protected Subject subject;
    public abstract void update();
    public Observer(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }
}
```

Třída *Subject* umožňuje správu observers a při změně stavu všechny notifikuje.

```
public class Subject {
    private List<Observer> observers = new ArrayList<Observer>();
    private int state;
    public int getState() {
        return state;
    }
    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }
    public void attach(Observer observer){
        observers.add(observer);
    }
    public void notifyAllObservers(){
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

06 OBSERVER

Vytvoříme subject a do něj zaregistrujeme tři různé observers. Poté modifikujeme stav *Subjectu*, všechny navázané observers jsou notifikovány a provedou metodu navázanou změnu stavu.

```
public class Client {
  public static void main(String[] args) {
    Subject subject = new Subject();
    new HexaObserver(subject);
    new OctalObserver(subject);
    new BinaryObserver(subject);
    System.out.println("First state change: 15");
    subject.setState(15);
    System.out.println("Second state change: 10");
    subject.setState(10);
  }
}
```

=>

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

06 OBSERVER

Funkcionální programování

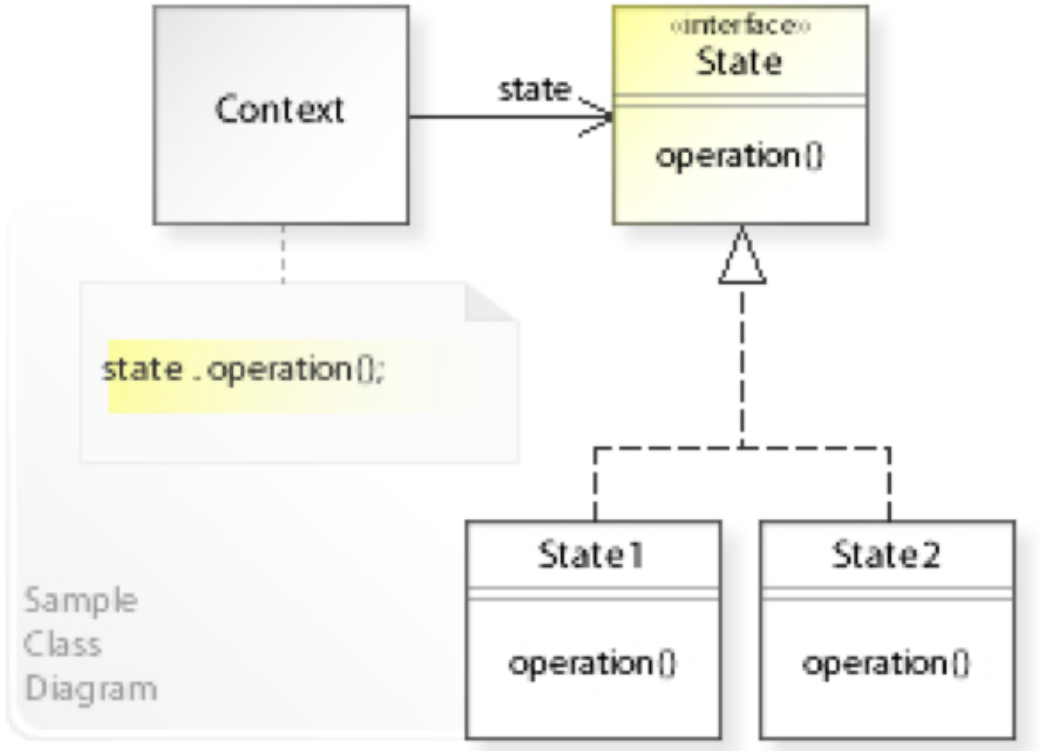
Alternativou pro observer ve funkcionálním programování jsou:

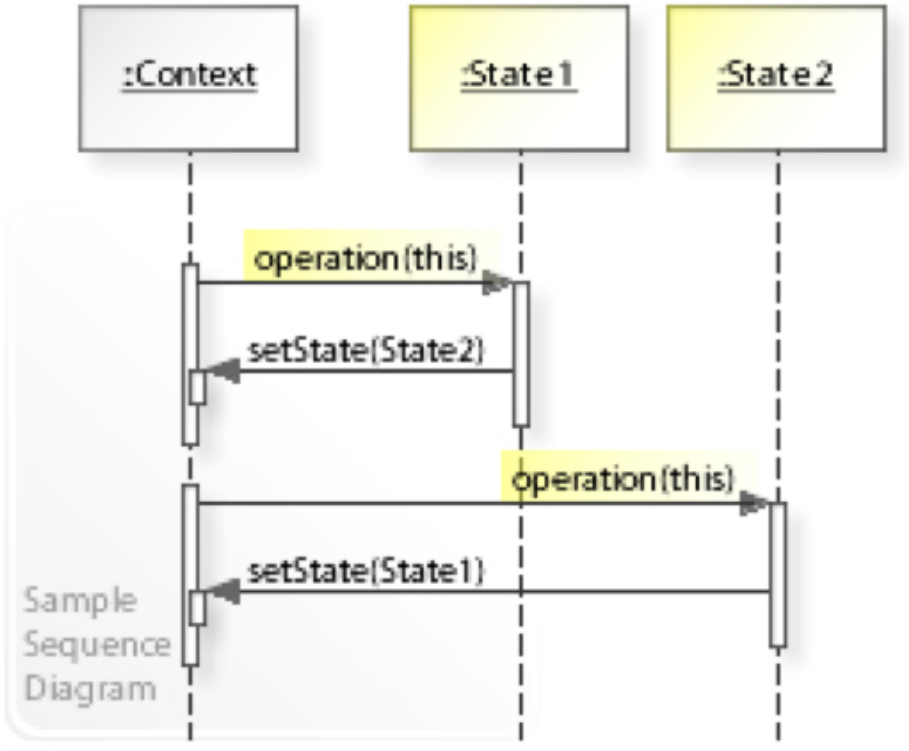
- Event streams
- Publish subscribe model
- Funkcionální reaktivní programování

Bude vysvětleno v dalších přednáškách

06 STATE/STATE MODEL

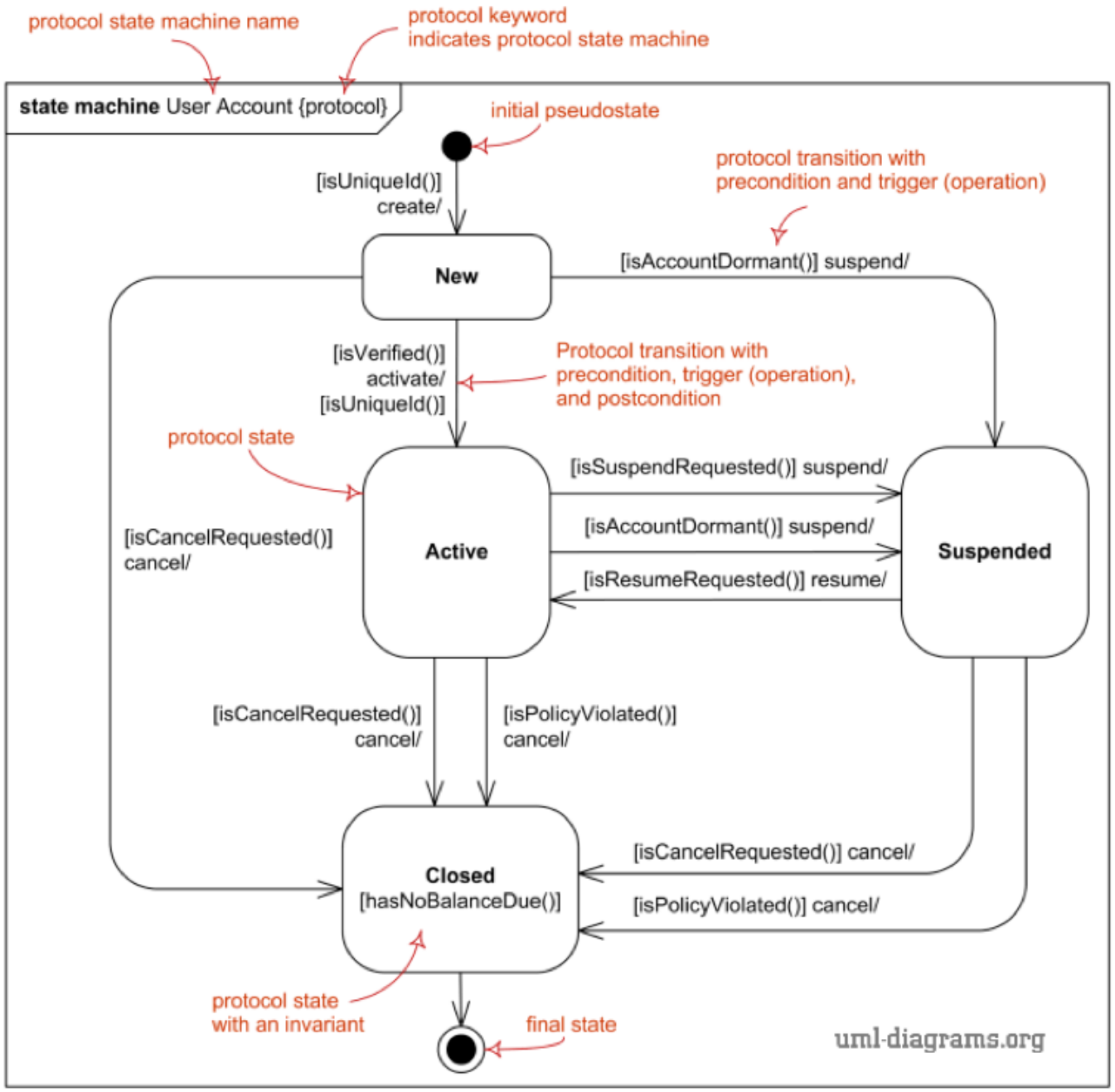
Implementace principů stavového automatu pomocí OOP. Každý stav systému nebo komponenty je realizován pomocí objektu State, který má předdefinované rozhraní pro realizaci přechodů (transitions) mezi jednotlivými stavu systému. State objekt tedy definuje stav systému a také metody pro přechody. State také může obsahovat kód pro kontrolu preconditions





06 STATE/STATE MODEL

Připomeneme si jak vypadá UML state diagram



06 STATE/STATE MODEL

Implementujeme **přehrávač na CD**. Začneme tím, že nadefinujeme třídu pro reprezentaci CD (public fields pro zkrácení kódu)

```java
public class CD {
    public String name;
    public byte format;

    public CD(String name, byte format) {
        this.name = name;
        this.format = format;
    }
}
```

Pak nadefinujeme rozhraní, přes které budeme posouvat náš stavový automat.

```java
public interface AudioPlayerState {
    public void insertCD(AudioPlayerContext context, CD cd);
    public CD ejectCD(AudioPlayerContext context);
    public void play(AudioPlayerContext context);
    public void stop(AudioPlayerContext context);
}
```

06 STATE/STATE MODEL

Nadefinujeme stavy přehrávače. Stavy implementují rozhraní, které jsme definovali na předchozím slide. Pokud stavový přechod nedává smysl, tak se neději nic. Pokud ano, tak stavy provede přenastavení stavu na kontexu.

Zde implementujeme stav ready.

U metody *play()* se kontroluje precondition na správný formát CD.

```
public class ReadyState implements AudioPlayerState {
   public ReadyState(){
System.out.println("Player ready");
   }

   public void play(AudioPlayerContext context) {
       //Test precondition
       if (context.getCd() != null &&
context.getCd().format > 0)
           context.setState(new PlayingState());
       else
           System.out.println("Cannot play this CD");
   }

   public CD ejectCD(AudioPlayerContext context) {
       context.setState(new NoCDState());
       return context.getCd();
   }

   public void stop(AudioPlayerContext context) { }
   public void insertCD(AudioPlayerContext context,CD cd){}
}
```

06 STATE/STATE MODEL

Pokračujeme pro další dva stavy

```
public class NoCDState implements AudioPlayerState {
    public NoCDState(){
        System.out.println("Player empty");
    }
    public void insertCD(AudioPlayerContext context, CD cd) {

        context.setState(new ReadyState());
        context.setCd(cd);
    }
    public CD ejectCD(AudioPlayerContext context) { return null;}
    public void play(AudioPlayerContext context) { }
    public void stop(AudioPlayerContext context) { }
}

public class PlayingState implements AudioPlayerState {
    public PlayingState(){
        System.out.println("Player playing");
    }
    public void stop(AudioPlayerContext context) {
        context.setState(new ReadyState());
    }
    public CD ejectCD(AudioPlayerContext context) { return null;}
    public void play(AudioPlayerContext context) { }
    public void insertCD(AudioPlayerContext context, CD cd) { }
}
```

06 STATE/STATE MODEL

Vlastní audio přehrávač realizujeme jako context objekt ze state design patternu.

```
public class AudioPlayerContext {
    CD cd;
    private AudioPlayerState state;

    public CD getCd() { return cd;}

    public AudioPlayerContext(){
state = new NoCDState();
    }
    public void setCd(CD cd) {
        this.cd = cd;
        System.out.println(cd.name);
    }

    public void setState(AudioPlayerState state){this.state=state;}

    public void play(){ state.play(this);}

    public void insertCD(CD cd){ state.insertCD(this, cd);}

    public void stop(){ state.stop(this);}

    public void ejectCD(){ state.ejectCD(this);}

    public AudioPlayerState getState(){ return state;}
}
```

06 STATE/STATE MODEL

Vlastní audio přehrávač realizujeme jako context objekt ze state design patternu.

```
public class AudioPlayerContext {
    CD cd;
    private AudioPlayerState state;

    public CD getCd() { return cd;}

    public AudioPlayerContext(){
state = new NoCDState();
    }
    public void setCd(CD cd) {
        this.cd = cd;
        System.out.println(cd.name);
    }

    public void setState(AudioPlayerState state){this.state=state;}

    public void play(){ state.play(this);}

    public void insertCD(CD cd){ state.insertCD(this, cd);}

    public void stop(){ state.stop(this);}

    public void ejectCD(){ state.ejectCD(this);}

    public AudioPlayerState getState(){ return state;}
}
```

06 STATE/STATE MODEL

Následující klient ukazuje využití našeho audio přehrávače

```
public class Client {
   public static void main(String[] args) {
       AudioPlayerContext player = new AudioPlayerContext();

       player.insertCD(new CD("Helena Vondrackova", (byte)-1));
       player.play(); //Nothing - precondition
       player.ejectCD(); //Ejects CD
       player.insertCD(new CD("Imagine Dragons", (byte)1));
       player.play();
       player.ejectCD(); //Nothing - invalid state
       player.stop(); //Stop playing
       player.ejectCD(); //Ejects CD
   }
}
```

06 STATE/STATE MODEL

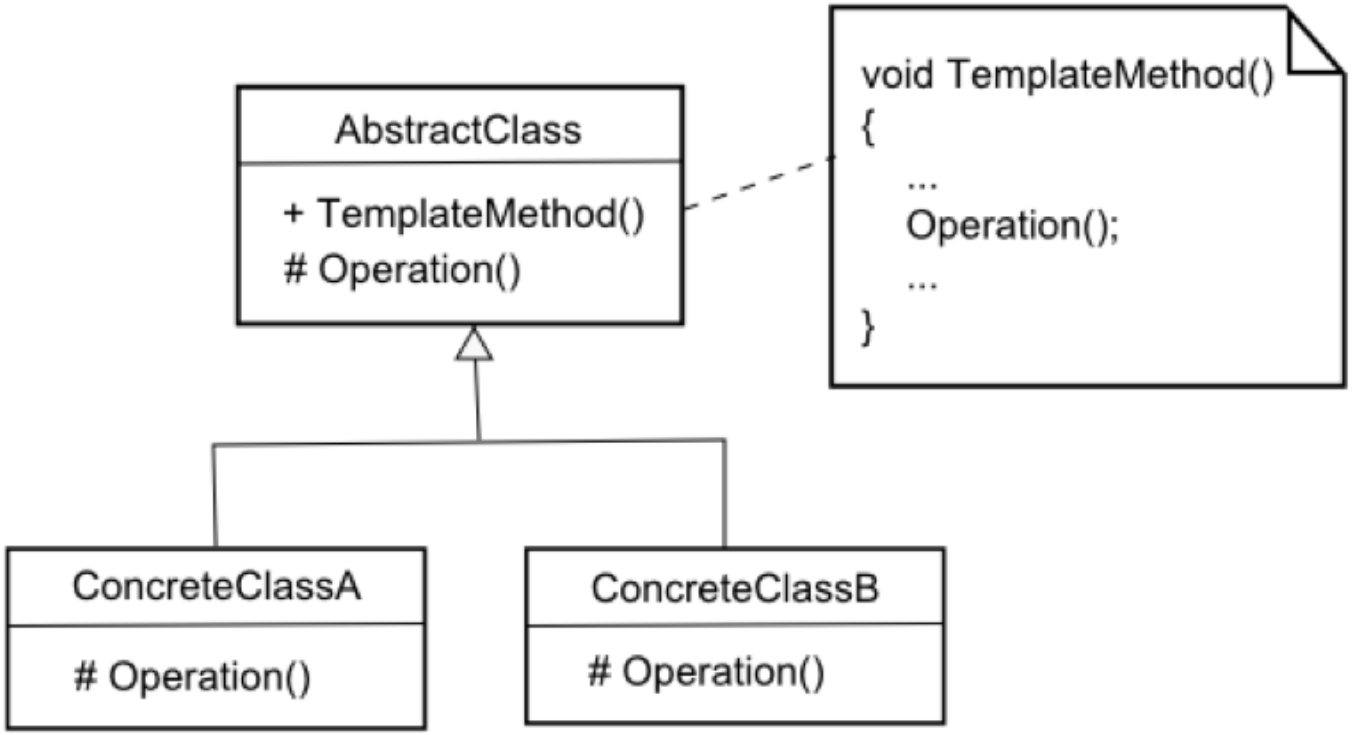
Funkcionální programování

Alternativou pro observer ve funkcionálním programování je realizace pomocí patternu monády.

Bude vysvětleno v dalších přednáškách

06 TEMPLATE METHOD

Předepisuje abstraktní metody pro variantní části chování systému. Abstraktní třída implementuje invariantní části chování a pro ty variabilní pak implementuje sekvenci (algoritmus) ve které se volají variantní části chování. Tyto variantní části chování se liší podle toho, která subclass je implementuje. Je zde tedy použit podobný princip jako u inversion of control - tedy místo toho, aby si subclasses řídily chování, tak to řídí jejich abstraktní předek.



06 TEMPLATE METHOD

Implementujeme abstraktní třídu *Game*, která svým potomkům předepisuje tři metody. Sama pak definuje algoritmus definující pořadí ve kterém se budou volat.

Pak nadefinujeme dva konkrétní potomky třídy

```java
public abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();
    //template method
    public final void play(){
        //common operations go here
        …
        //variable operations go here
        initialize();//initialize the game
        startPlay(); //start game
        endPlay(); //end game
    }
}
```

```java
public class Football extends Game {
    void endPlay() {
        System.out.println("Football Game Finished!");
    }
    void initialize() {
        System.out.println("Football Game Initialized! Start play");
    }
    void startPlay() {
        System.out.println("Football Game Started. Enjoy game!");
    }
}

public class Cricket extends Game {
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
}
```

06 TEMPLATE METHOD

```java
public class Client {
    public static void main(String[] args) {
        Game game = new Cricket();
        game.play();
        System.out.println();
        game = new Football();
        game.play();
    }
}
```

=>

```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!
```

06 TEMPLATE METHOD

Funkcionální programování

Alternativy k template metodě využívající funkcionální programování jsou:

1. **Higher-order functions**: zapouzdření sdílené funkcionality a operace jsou vkládány jako argumenty
2. **Function composition**: kompozice funkcí
3. **Pipelines**: Zřetězení funkcí (srovnej s chain of responsibility)
4. **Data-driven design**: použijeme datové struktury pro definici kroků a jejich exekuci v sekvenci

```java
public class FunctionComposition {

    public static void main(String[] args) {
        // Functions for each step
        Function<Integer, Integer> preprocess = item -> item * 2; // Double the value
        Function<Integer, Integer> process = item -> item + 1; // Add one to the value
        Function<Integer, Integer> postprocess = result -> result; // No change in this case

        // Composing the functions
        Function<Integer, Integer> completeProcess = preprocess.andThen(process).andThen(postprocess);

        Integer data = 5;
        Integer finalResult = completeProcess.apply(data);
        System.out.println("Final Result: " + finalResult); // Output: 11
    }
}
```

06 TEMPLATE METHOD

Funkcionální programování

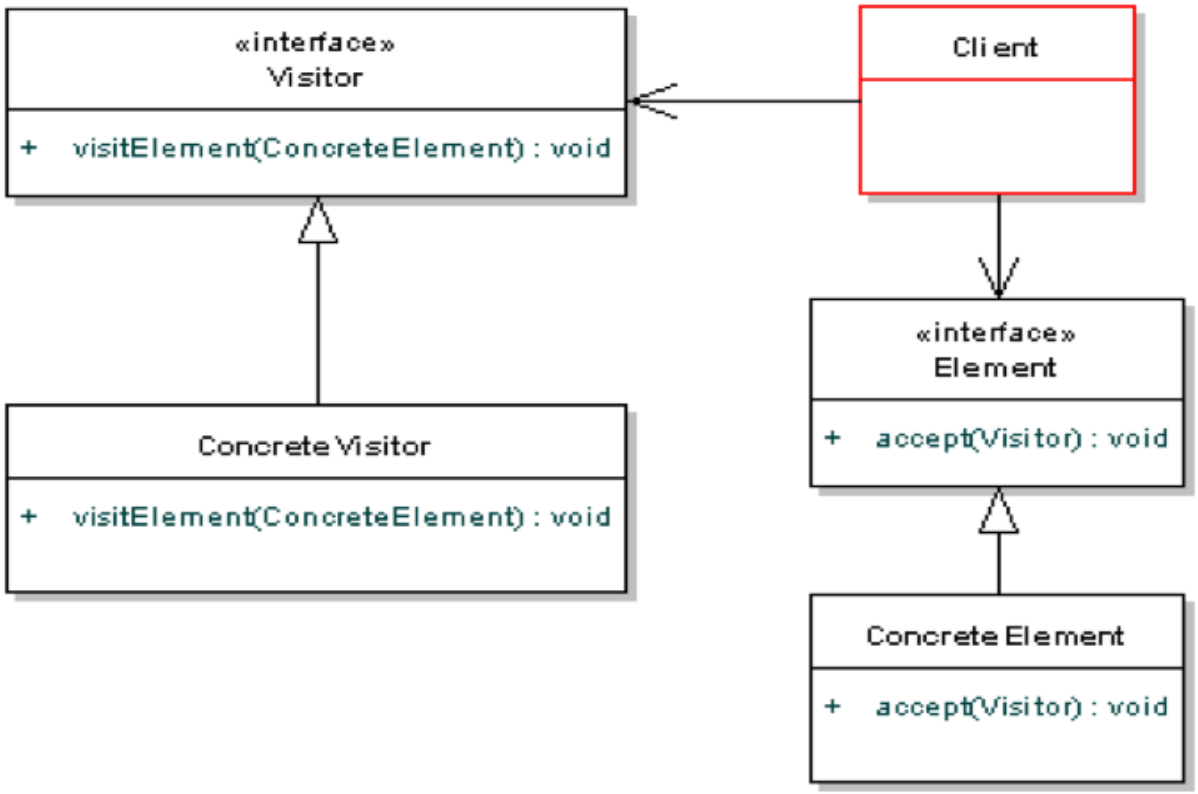
```
public class DataDrivenDesign {

    public static void main(String[] args) {
        // Define the steps
        Step<Integer, Integer> preprocess = new Step<>(item -> item * 2); // Double the value
        Step<Integer, Integer> process = new Step<>(item -> item + 1); // Add one to the value
        Step<Integer, Integer> postprocess = new Step<>(result -> result); // No change in this case

        // Create a data structure for steps
        Step<?, ?>[] steps = new Step[] { preprocess, process, postproc };

        Integer data = 5;
        Object result = executeSteps(data, steps);
        System.out.println("Final Result: " + result); // Output: 11
    }

    // Method to execute the steps
    public static Object executeSteps(Object data, Step<?, ?>[] steps) {
        Object currentData = data;
        for (Step<?, ?> step : steps) {
            currentData = step.getFunction().apply(currentData);
        }
        return currentData;
    }
}

// Helper class to define a step in the process
class Step<T, R> {
    private final Function<T, R> function;

    public Step(Function<T, R> function) {
        this.function = function;
    }

    public Function<T, R> getFunction() {
        return function;
    }
}
```

kolo

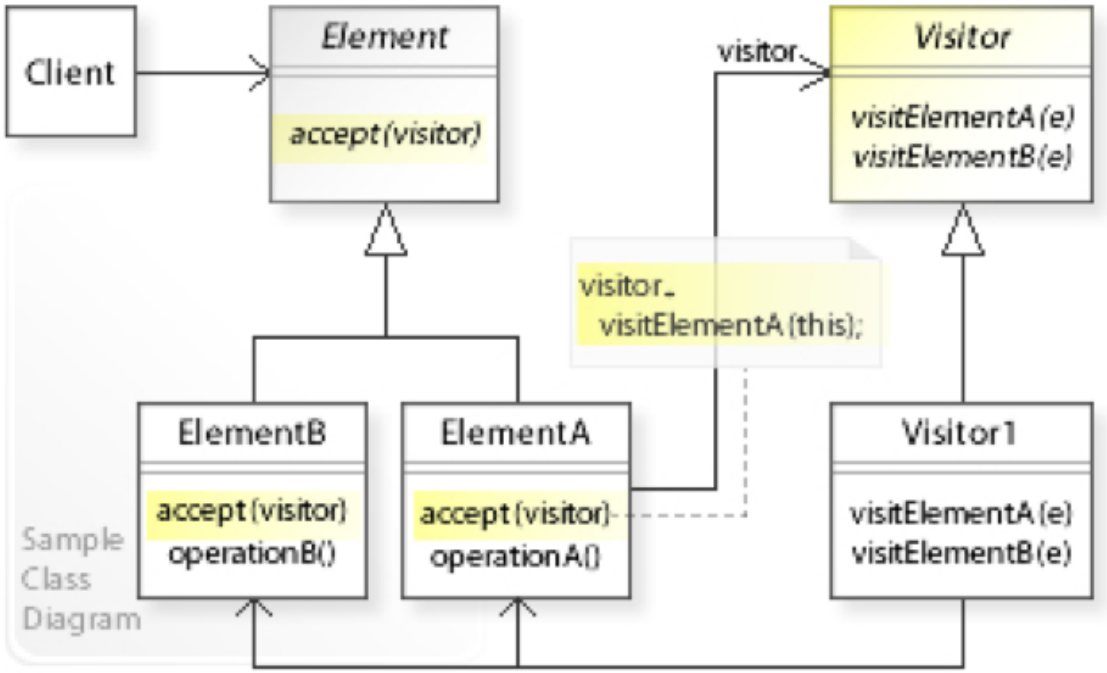
06 VISITOR

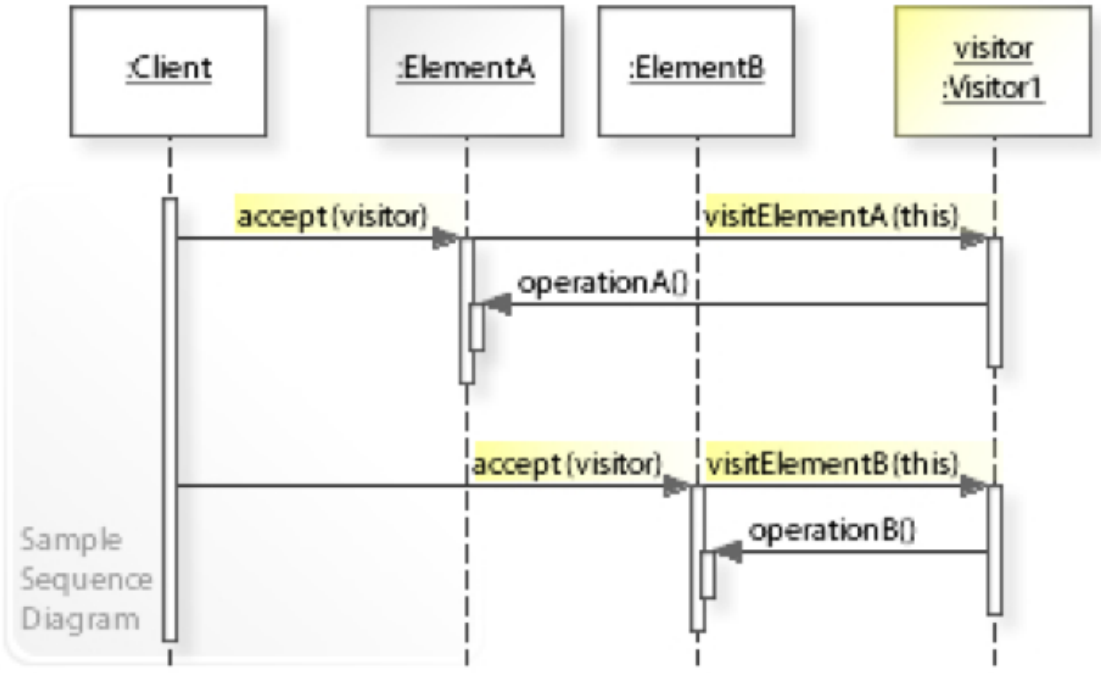
Oddělení algoritmu od datové struktury na které algoritmus pracuje. Chci do datové struktury přidávat další operace aniž bych ji musel modifikovat (podpora open-closed principu).

Realizace patternu je pomocí tzv. *double dispatch* principu. Elementy datové struktury u kterých chci v budoucnosti přidávat operace, mají metodu *accept(Visitor)* uvnitř které zavolám na vloženém visitoru metodu visitElement(Element). Tím předám řízení visitoru a zároveň mu předám referenc



06 VISITOR


Client
Element
accept(visitor)
visitor
Visitor
visitElementA(e)
visitElementB(e)
visitor.
visitElementA(this);
ElementB
accept(visitor)
operationB()
ElementA
accept(visitor)
operationA()
Visitor1
visitElementA(e)
visitElementB(e)
Sample
Class
Diagram


:Client
:ElementA
:ElementB
visitor
:Visitor1
accept(visitor)
visitElementA(this)
operationA()
accept(visitor)
visitElementB(this)
operationB()
Sample
Sequence
Diagram

06 VISITOR

Implementujeme datovou strukturu pro osobní počítač. Do klíčových komponent počítače vložíme metodu *accept()* pro double dispatch.

```
public interface ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor);
}

public class Keyboard implements ComputerPart {
  public void accept(ComputerPartVisitor computerPartVisitor) {
       computerPartVisitor.visit(this);
   }
}

public class Monitor implements ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor) {
       computerPartVisitor.visit(this);
   }
}

public class Mouse implements ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor) {
       computerPartVisitor.visit(this);
   }
}
```

```
public class Computer implements ComputerPart {

   ComputerPart[] parts;

   public Computer(){
       parts = new ComputerPart[] {new Mouse(),
  new Keyboard(),
  new Monitor()};
   }

   public void accept(ComputerPartVisitor computerPartVisitor) {
       for (int i = 0; i < parts.length; i++) {
           parts[i].accept(computerPartVisitor);
       }
       computerPartVisitor.visit(this);
   }
}
```

06 VISITOR

Vytvoříme interface pro různé visitory pracující nad komponentami počítače.

Implementujeme konkrétní realizaci tohoto visitoru. Tato vypisuje informace o klíčových komponentách na konzoli

```java
public interface ComputerPartVisitor {
    public void visit(Computer computer);
    public void visit(Mouse mouse);
    public void visit(Keyboard keyboard);
    public void visit(Monitor monitor);
}

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {
    public void visit(Computer computer) {
        System.out.println("Displaying Computer.");
    }

    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");
    }

    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");
    }

    public void visit(Monitor monitor) {
        System.out.println("Displaying Monitor.");
    }
}
```

06 VISITOR

```
public class Client {
   public static void main(String[] args) {

       ComputerPart computer = new Computer();
       computer.accept(new ComputerPartDisplayVisitor());
   }
}
```

=>

```
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```

06 VISITOR

Funkcionální programování

Klasický visitor lze modifikovat v Java pomocí:

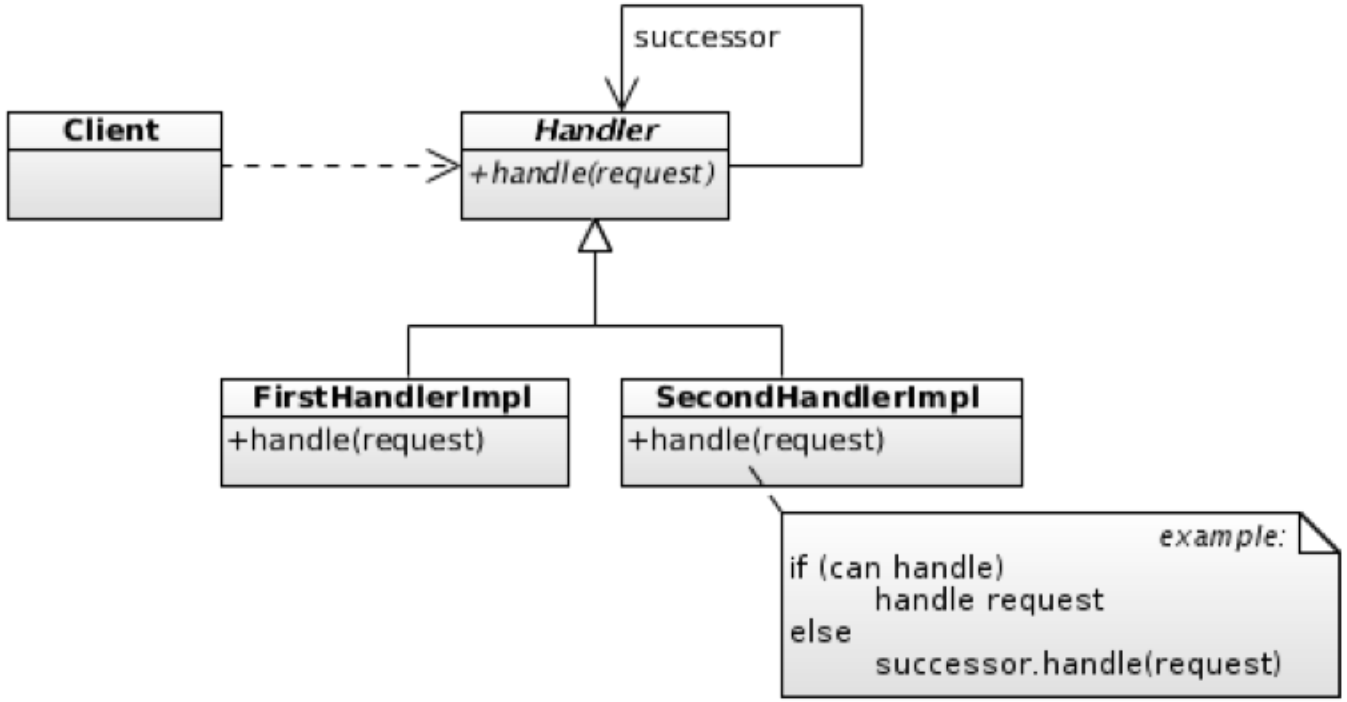
- Polymorfních metod
- Pattern Matching s instanceof a switch
- Higher Order Functions

```java
class ShapeCalculator {

    // Function to calculate area based on the shape type
    public static Function<Shape, Double> areaFunction() {
        return shape -> {
            if (shape instanceof Circle circle) {
                return Math.PI * circle.radius * circle.radius;
            } else if (shape instanceof Rectangle rectangle) {
                return rectangle.width * rectangle.height;
            } else if (shape instanceof Triangle triangle) {
                return 0.5 * triangle.base * triangle.height;
            }
            throw new IllegalArgumentException("Unknown shape type for area calculation");
        };
    }

    // Function to calculate circumference based on the shape type
    public static Function<Shape, Double> circumferenceVisitorFunction() {
        return shape -> {
            if (shape instanceof Circle circle) {
                return 2 * Math.PI * circle.radius;
            } else if (shape instanceof Rectangle rectangle) {
                return 2 * (rectangle.width + rectangle.height);
            } else if (shape instanceof Triangle triangle) {
                return triangle.base + triangle.sideA + triangle.sideB;
            }
            throw new IllegalArgumentException("Unknown shape type for circumference calculation");
        };
    }

    public static void main(String[] args) {
        Shape circle = new Circle(5); Shape rectangle = new Rectangle(4, 6); Shape triangle = new Triangle(3, 4, 3, 5);
        List<Shape> shapes = Arrays.asList(circle, rectangle, triangle);

        // Get the area and circumference functions
        Function<Shape, Double> areaVisitor = areaFunction();
        Function<Shape, Double> circumferenceVisitor = circumferenceVisitorFunction();

        //Iterate the structure and calculate areas
        for (Shape shape : shapes) {
            System.out.printf("%s %f ", shape.getClass().getSimpleName(), areaVisitor.apply(shape));
        }

        //Iterate the structure and calculate circumferences
        for (Shape shape : shapes) {
            System.out.printf("%s %f ", shape.getClass().getSimpleName(), circumferenceVisitor.apply(shape));
        }
    }
}
```

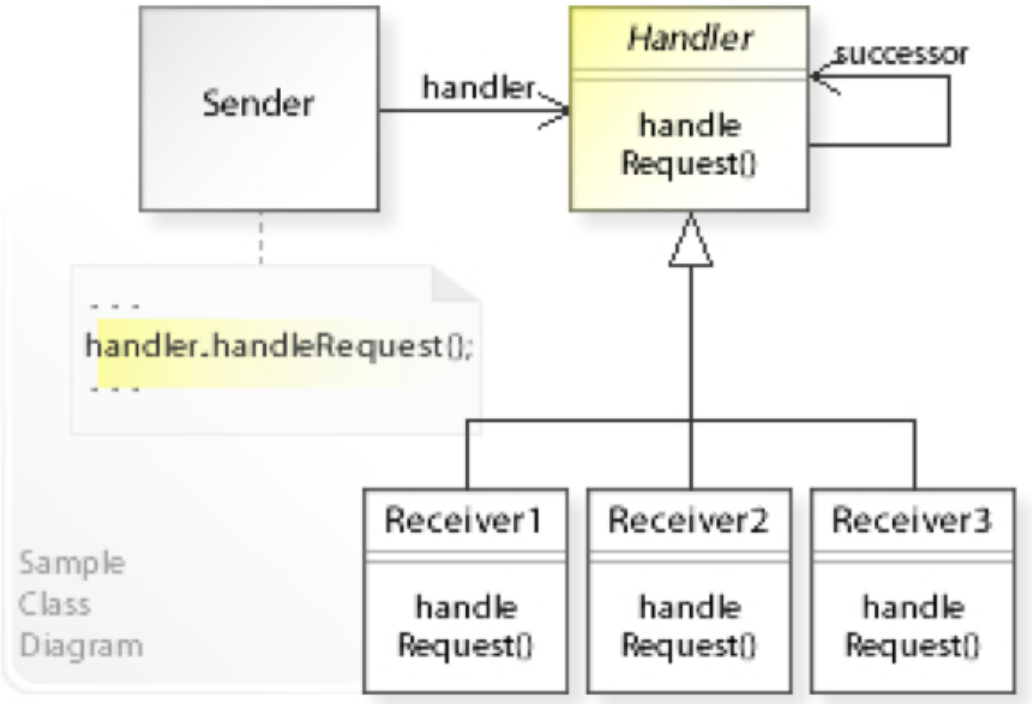
06 CHAIN OF RESPONSIBILITY

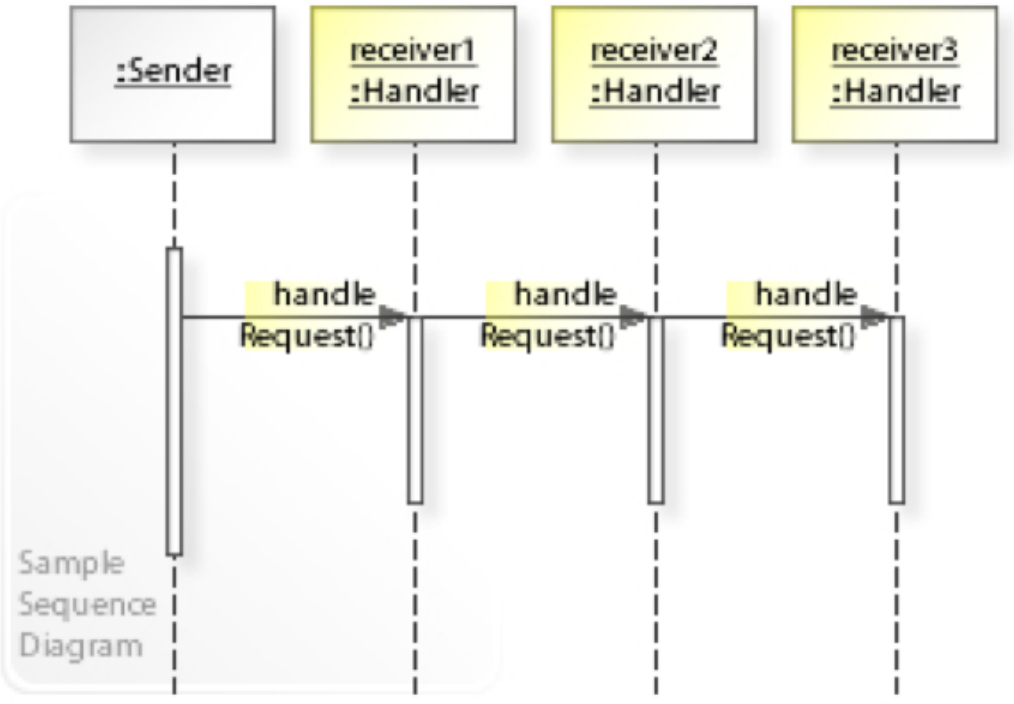
Vytvoření řetězu komponent, které postupně zpracovávají požadavek. Snižuje coupling mezi objektem posílajícím požadavek a objektem přijímající požadavek. Pošlu tedy požadavek a ten je "opracováván" nebo na něj reagují komponenty. Každá komponenta má nějaký jiný účel nebo řeší jinou doménu problémy. Každá komponenta se rozhoduje jestli má požadavek poslat na další komponentu v pořadí.

Tento pattern využívá EDA (Event Driven Architecture) nebo celá řada frameworků, kde je třeba na requets třeba aplikovat různé aspekty nebo naleznout komponentu, která je schopná požadavek odbavit.



06 CHAIN OF RESPONSIBILITY





06 CHAIN OF RESPONSIBILITY

Komponenty, které řeší požadavek můžeme řetězit do lineárních seznamů nebo i stromů

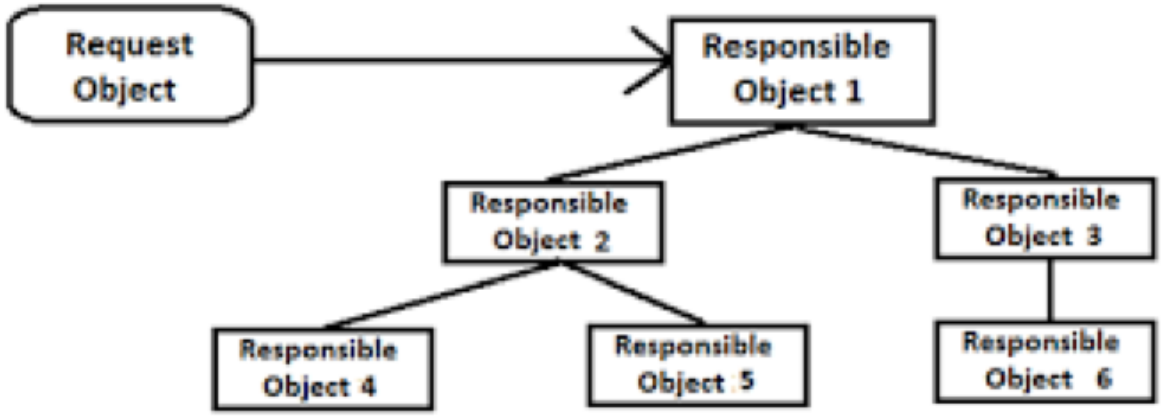
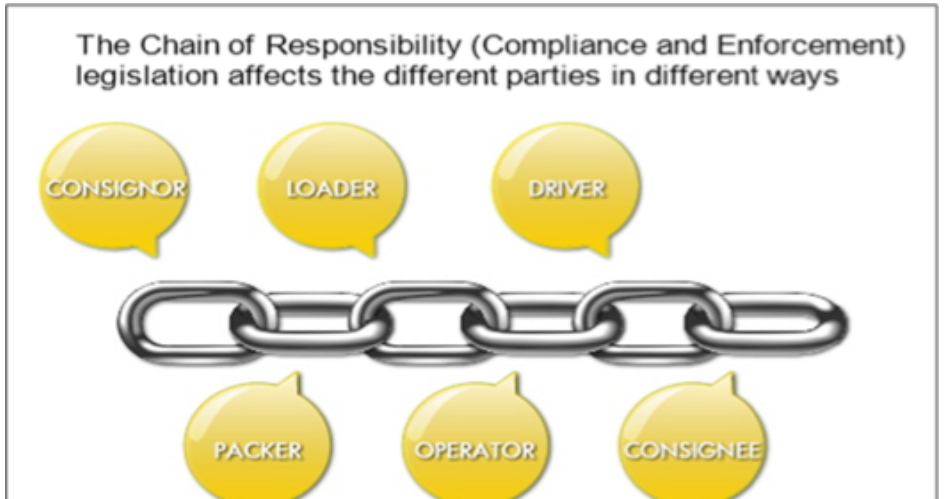


Fig: Tree Chain of Responsibility



06 CHAIN OF RESPONSIBILITY

Chceme implementovat logovací systém. Ten vytvoříme pomocí logovacích komponent, kde každá loguje jiný druh informací a které je možné řetězit za sebe.

Začneme implementací abstraktní logovací třídy

```
public abstract class AbstractLogger {
   public static int INFO = 1;
   public static int DEBUG = 2;
   public static int ERROR = 3;
   protected int level;

   //next element in chain or responsibility
   protected AbstractLogger nextLogger;

   public void setNextLogger(AbstractLogger nextLogger){
      this.nextLogger = nextLogger;
   }

   public void logMessage(int level, String message){
      if(this.level <= level){
         write(message);
      }
      if(nextLogger !=null){
         nextLogger.logMessage(level, message);
      }
   }

   abstract protected void write(String message);
}
```

06 CHAIN OF RESPONSIBILITY

Implementujeme jednotlivé logovací komponen

```
public class ConsoleLogger extends AbstractLogger {
   public ConsoleLogger(int level){
       this.level = level;
   }
   protected void write(String message) {
      System.out.println("Standard Console::Logger: " + message);
   }
}
public class ErrorLogger extends AbstractLogger {
   public ErrorLogger(int level){
       this.level = level;
   }
   protected void write(String message) {
       System.out.println("Error Console::Logger: " + message);
   }
}
public class FileLogger extends AbstractLogger {
   public FileLogger(int level){
       this.level = level;
   }
   protected void write(String message) {
       System.out.println("File::Logger: " + message);
   }
}
```

06 CHAIN OF RESPONSIBILITY

```java
public class Client {
    private static AbstractLogger getChainOfLoggers(){
       AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
       AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
       AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);
       errorLogger.setNextLogger(fileLogger);
       fileLogger.setNextLogger(consoleLogger);
       return errorLogger;
    }

    public static void main(String[] args) {
       AbstractLogger loggerChain = getChainOfLoggers();
       loggerChain.logMessage(AbstractLogger.INFO, "This is an information.");
       loggerChain.logMessage(AbstractLogger.DEBUG, "This is an debug level information.");
       loggerChain.logMessage(AbstractLogger.ERROR,  "This is an error information.");
    }
}
```

=>

```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error information.
```

06 CHAIN OF RESPONSIBILITY

Funkcionální programování

1.Processory jako funkce: Každý procesní krok, *validateCustomer, validateItems, applyDiscounts*, je *UnaryOperator<CustomerOrder>* protože bere a vrací *CustomerOrder*.

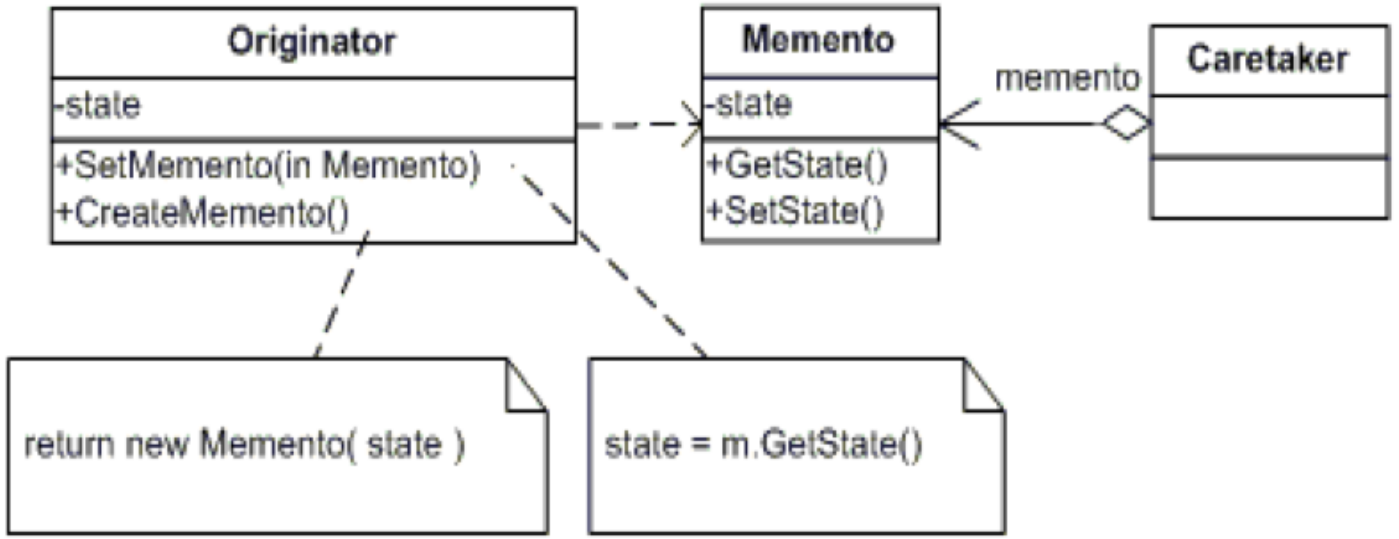
2.Chaining funkce: *orderProcessorChain* je kompozice procesních kroků spojená přes *.andThen().*

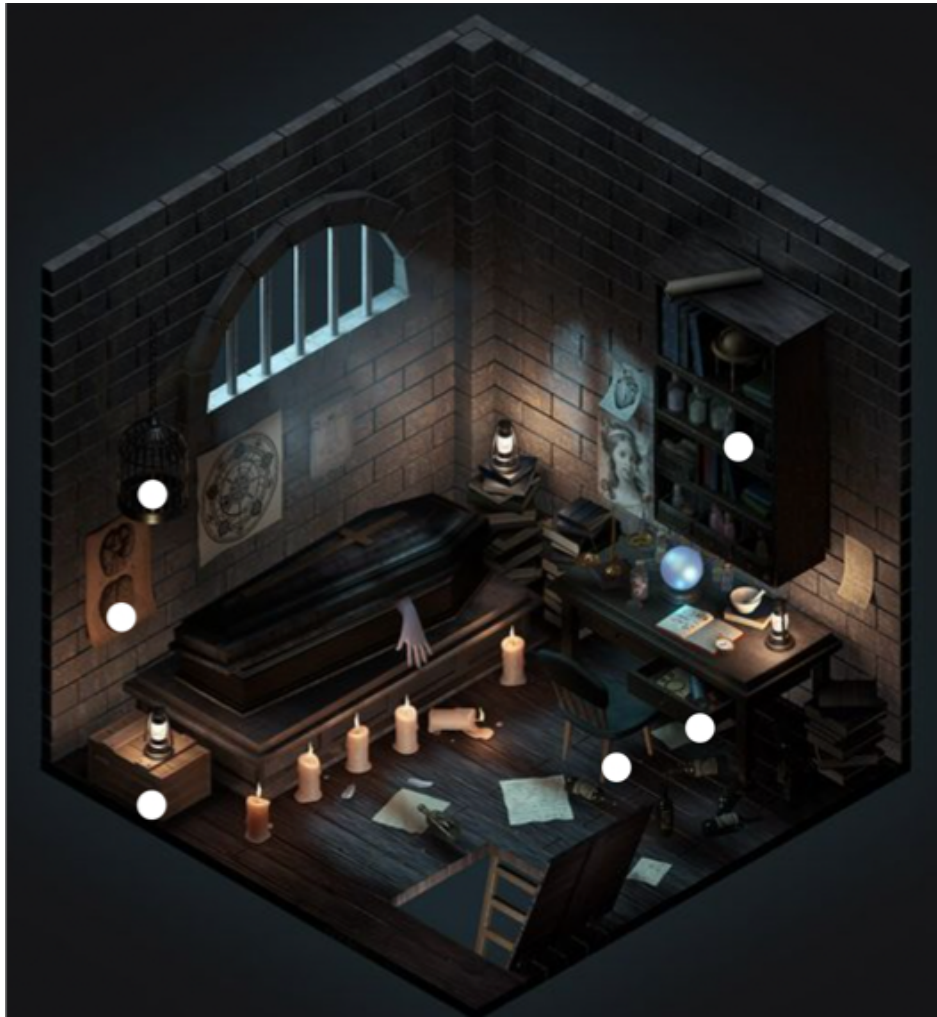
3.Exekuce: Main funkce inicializuje CustomerOrder a aplikuje na ni orderProcessorChain. Každý krok provede operaci na CustomerOrder a předá řízení do následujícího kroku.

```
class OrderChainOfResponsibility {
    // Step 1: Define processors for the chain as functions
    static UnaryOperator<CustomerOrder> validateCustomer = order -> {
        if (order.getCustomerName() == null || order.getCustomerName().isEmpty()) {
            throw new IllegalArgumentException("Invalid customer name");
        }
        System.out.println("Customer validation passed.");
        return order;
    };

    static UnaryOperator<CustomerOrder> validateItems = order -> {
        if (order.getItems() == null || order.getItems().isEmpty()) {
            throw new IllegalArgumentException("Order must contain items");
        }
        System.out.println("Item validation passed.");
        return order;
    };

    static UnaryOperator<CustomerOrder> applyDiscounts = order -> {
        if (order.getTotalPrice() > 100) {
            order.setTotalPrice(order.getTotalPrice() * 0.9); // Apply 10% discount
            System.out.println("Discount applied.");
        }
        return order;
    };

    // Step 2: Build the chain by composing functions
    static Function<CustomerOrder, CustomerOrder> orderProcessorChain =
            validateCustomer
            .andThen(validateItems)
            .andThen(applyDiscounts);

    public static void main(String[] args) {
        CustomerOrder order = new CustomerOrder("John Doe", 150.0, List.of("Item1", "Item2"));
        // Process the order through the chain
        CustomerOrder processedOrder = orderProcessorChain.apply(order);
        System.out.println("Final Price: " + processedOrder.getTotalPrice());
    }
}
```

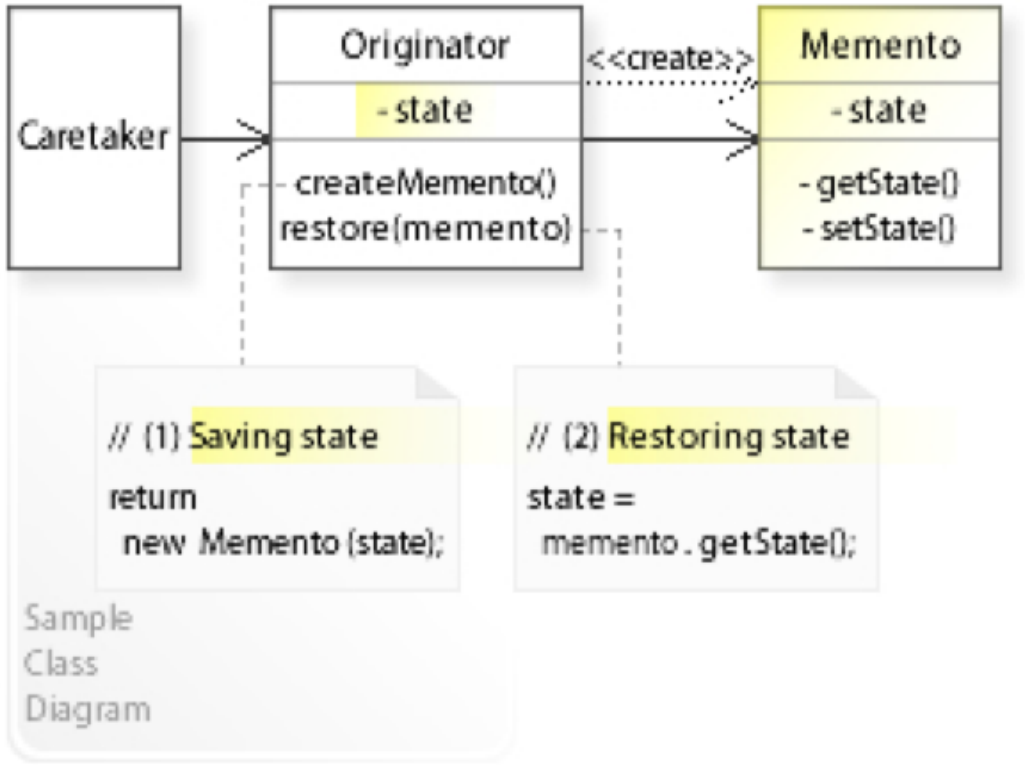
06 MEMENTO

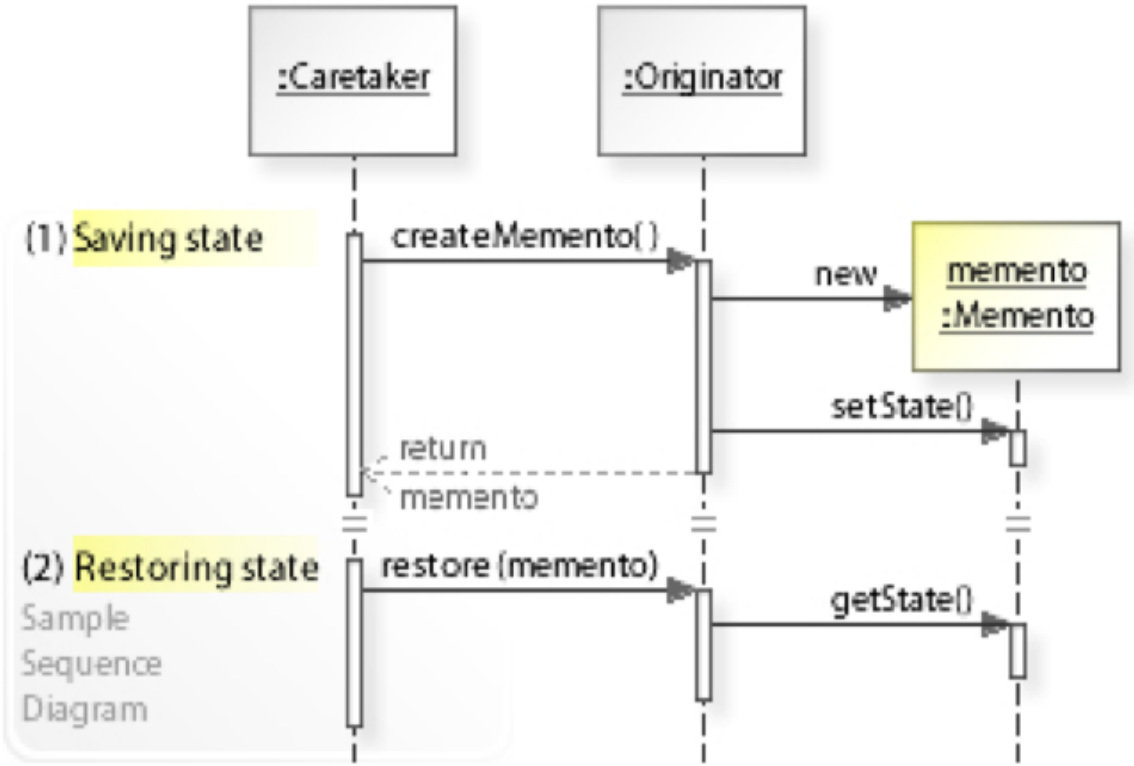
Navrácení objektu k jeho předcházejícímu stavu. Design pattern se používá např. pro realizaci undo/redo operací. Pattern se skládá ze tří základních entit: *Originator, Caretaker* a *Memento*. Originator je naše komponenta, kterou modifikujeme (měníme její stavy) a chceme mít možnost se k těmto stavům vracet. Tyto změny s naší komponentou provádí Caretaker. Předtím, než Caretaker provede změnu, tak si vyžádá aktuální stav. Ten mu Originator vrátí v objektu Memento. Caretaker si pak udržuje list těchto "mement" a může se tak vracet k libovolnému stavu.





MEMENTO


Caretaker
Originator
- state
createMemento()
restore(memento)
<<create>>
Memento
- state
- getState()
- setState()
// (1) Saving state
return
new Memento (state);
// (2) Restoring state
state =
memento.getState();
Sample
Class
Diagram


:Caretaker
:Originator
(1) Saving state
createMemento()
new
memento
:Memento
setState()
return
memento
(2) Restoring state
restore (memento)
getState()
Sample
Sequence
Diagram

06 MEMENTO

```
import java.util.List;
import java.util.ArrayList;
class Originator {
    private String state;
    // The class could also contain additional data that is not part of the
    // state saved in the memento..

    public void set(String state) {
        this.state = state;
        System.out.println("Originator: Setting state to " + state);
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(this.state);
    }

    public void restoreFromMemento(Memento memento) {
        this.state = memento.getSavedState();
        System.out.println("Originator: State after restoring from Memento: " + state);
    }
```

06 MEMENTO

```
    public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        // accessible by outer class only
        private String getSavedState() {
            return state;
        }
    }
}

class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new ArrayList<Originator.Memento>();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to roll back to.
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

06 MEMENTO

Funkcionální programování

Ve funkcionálním programování dosahujeme stejných cílů jako u mementa bez potřeby mutable stavů. Hlavní přístupy jsou:

1. **Immutable Data Structures**: Vytváří se neustále nové stavy
2. **Versioning**: Transformace datové struktury do tvaru, který udržuje verze
3. **Functional State Machines**: Využití pure funkcí, které řídí stavové transakce bez nutnosti držení vnitřních reprezentací

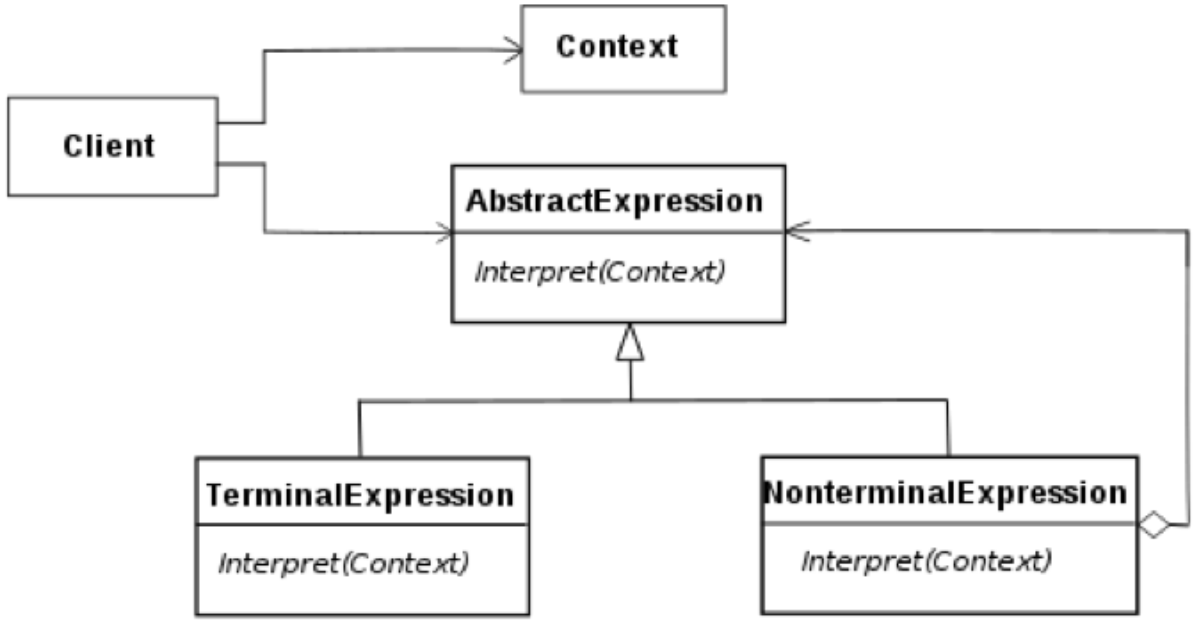
Bude vysvětleno v dalších přednáškách

06 INTERPRETER

Zpracování (vyhodnocování) výrazů vytvořených v určitém jazyce. Princip je takový, že pro každý symbol je vytvořena třída, která ho dokáže zpracovat. Symboly jsou buď:

- *terminální* - dále se nepřepisují - tedy vyhodnocení v nich končí
- *neterminální* - řídí zpracování dalších symbolů

Pro reprezentaci věty se často používá tzv. Syntaktický strom (syntax tree) = stromová reprezentace syntaktické struktury věty



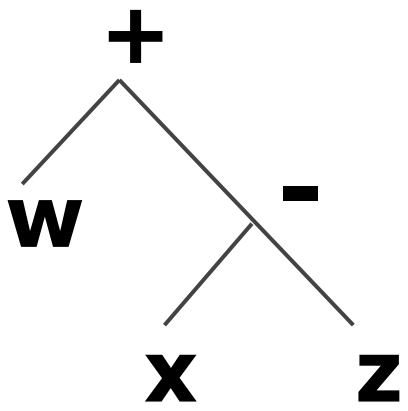
06 INTERPRETER

Chceme spočítat výraz zapsaný větou "*w x z - +*". Jedná se o matematický výraz zapsaný v postfix zápisu *"x-z+w"*.

Věta obsahuje tři typy symbolů:

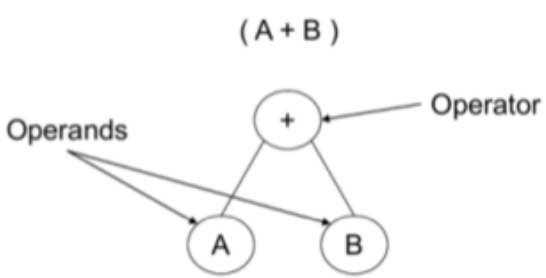
- číslo (terminální)
- sčítání (neterminální)
- odčítání (neterminální)

Z výrazu lze sestavit následující syntaktický strom:




```
interface Expression {
  public int interpret(final Map<String, Expression> variables);
}
```

06 INTERPRETER



Traversals?

InOrder → Infix
A + B

PreOrder → Prefix
+ A B

PostOrder → Postfix
A B +

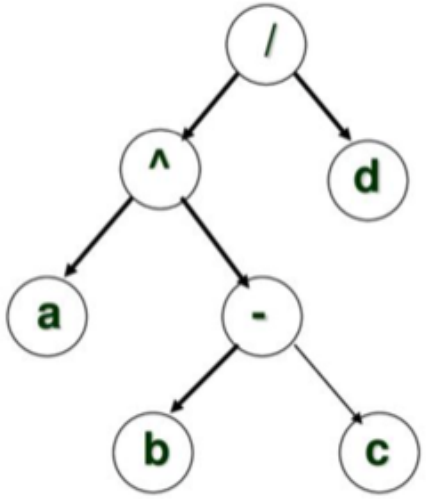
A binary tree can store anything where the number '2' comes into picture.

In an arithmetic expression:

Operators normally have maximum of 2 operands.

So an arithmetic expression can be easily represented using Binary Tree.

a ^ (b – c) / d



Infix: ((a ^ (b – c)) / d)

Postfix: a b c - ^ d /

Prefix: / ^ a – b c d

06 INTERPRETER

Definujeme interface Expression pro reprezentaci terminálních a neterminálních symbolů a implementujeme ho pro operátory (+-) a operandy (čísla). Pozn. odčítání obdobné jako sčítání

```
class Variable implements Expression {
  private String name;
  public Variable(final String name) {
    this.name = name;
  }
  public int interpret(final Map<String, Expression> variables) {
    if (null == variables.get(name))
            return 0; // Either return new Number(0).
    return variables.get(name).interpret(variables);
  }
}
```

```
class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(final Expression left, final Expression right) {
      leftOperand = left;
      rightOperand = right;
    }
    public int interpret(final Map<String, Expression> variables) {
        return leftOperand.interpret(variables) + rightOperand.interpret(variables);
    }
}
```

06 INTERPRETER

Implementujeme parser vět do syntaktického stromu. Využijeme zásobníku pro sestavení z postfix zápisu:

```
class Evaluator implements Expression {
  private Expression syntaxTree;
  public Evaluator(final String expression) {
    final Stack<Expression> expressionStack = new Stack<Expression>();
    for (final String token : expression.split(" ")) {
      if (token.equals("+")) {
        final Expression subExpression=new Plus(expressionStack.pop(), expressionStack.pop());
        expressionStack.push(subExpression);
      } else if (token.equals("-")) {
        // it's necessary remove first the right operand from the stack
        final Expression right = expressionStack.pop();
        // ..and after the left one
        final Expression left = expressionStack.pop();
        final Expression subExpression = new Minus(left, right);
        expressionStack.push(subExpression);
      } else
        expressionStack.push(new Variable(token));
    }
    syntaxTree = expressionStack.pop();
  }
  public int interpret(final Map<String, Expression> context) {
    return syntaxTree.interpret(context);
  }
}
```

06 INTERPRETER

A provedeme interpretaci při dosazení proměnných *w = 6, x = 8, x = 30*

```
public class InterpreterExample {
  public static void main(final String[] args) {
    final String expression = "w x z - +";
    final Evaluator sentence = new Evaluator(expression);
    final Map<String, Expression> variables = new HashMap<String, Expression>();
    variables.put("w", new Number(6));
    variables.put("x", new Number(8));
    variables.put("z", new Number(30));
    final int result = sentence.interpret(variables);
    System.out.println(result);
  }
}
```

06 SHRNUTÍ

- **Iterator** - abstrakce procházení datové struktury od její implementace
- **Chain of responsibility** - vytvoření řetězu komponent, které postupně zpracovávají požadavek
- **Strategy** - dynamické změny chování komponenty (nebo jejího algoritmu) v průběhu běhu programu
- **Visitor** - oddělení algoritmu od datové struktury na které pracuje
- **Observer** - propojení změnu stavu komponenty s komponentami reagujícími na tuto změnu
- **Template method** - předepisuje abstraktní metody pro variantní části chování
- **State** - implementace principů stavového automatu pomocí OOP
- **Memento** - navrácení objektu k jeho předcházejícímu stavu (undo/redo)
- **Interpreter** - zpracování (vyhodnocování) vět vytvořených v určitém jazyce