OMO

6 - Behavioral Design Patterns

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

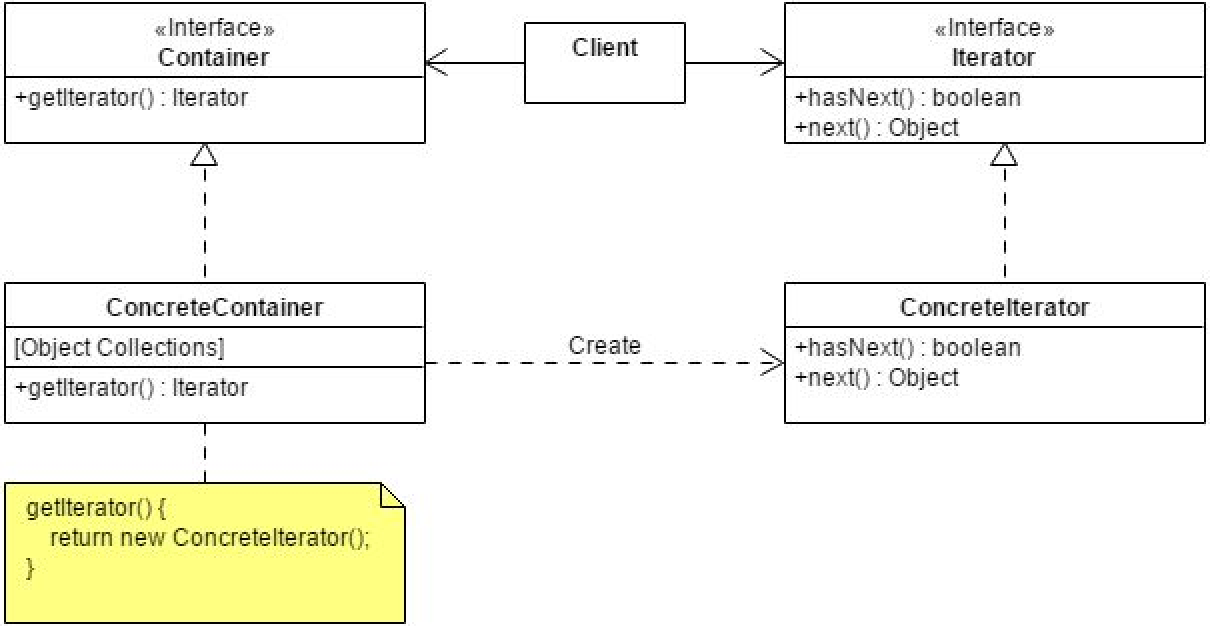
Ing. David Kadleček, PhD.

Creational design patterns

- **Iterator** - abstrakce procházení datovové 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** - implemenace principů stavového automatu pomocí OOP
- **Memento** - navrácení objektu k jeho předcházejícímu stavu (undo/redo)
- **Interpreter** - zpracování (vyhodnocování) vět vytvořených v určitém jazyce

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 *isDone()* (někdy *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



```
getIterator() {
   return new ConcreteIterator();
}
```

Iterátor

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

```
// Container.java
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();
}
```

Iterátor

PetRepository je datová struktura na správu dinosaurů, interně je reprezentována přes pole. Vrací iterátor na procházení přes tyto dinosaury.

```java
public class PetRepository implements Container {
   public String pets[] = {
           "Tyrannosaurus Rex",
           "Stegosaurus",
           "Velociraptor",
           "Triceratops",
           "Pterosauria",
           "Ichthyosaur",
           "Tanystropheus"};
   public Iterator getIterator() {
       return new NameIterator();
   }
   private class NameIterator implements Iterator {
       int index = 0;
       public boolean hasNext() {
           return index < pets.length;
       }
       public Object next() {
           if (hasNext()) {
               return pets[index++];
           }
           return null;
       }
   }
}
```

Iterátor

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

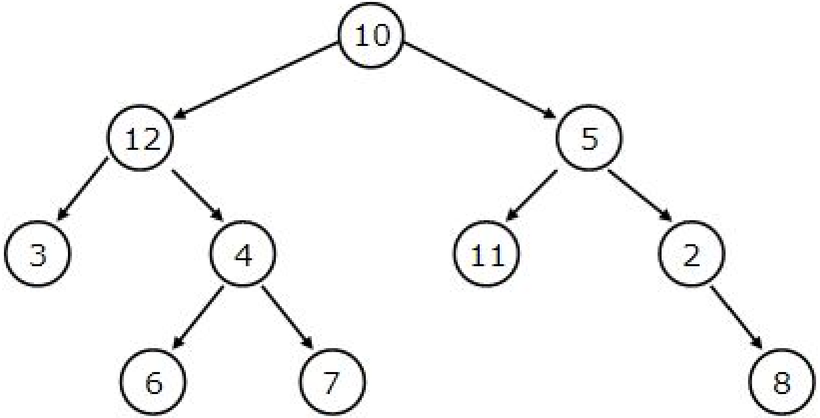
```
public class Client {

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

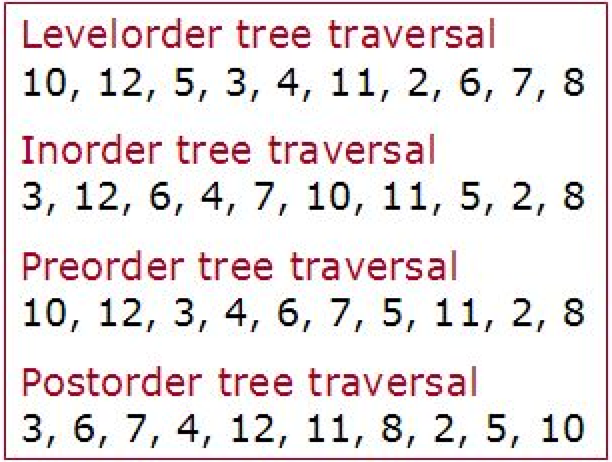
=>

```
Tyrannosaurus Rex
Stegosaurus
Velociraptor
Triceratops
Pterosauria
Ichthyosaur
Tanystropheus
```

Iterátor

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





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

Iterátor

Začneme tím, že si vytvoříme strukturu pro reprezentaci binárního stromu a element zásobníku, který nám vrací informaci o tom, že byl element navštíven iterátorem.

```
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; }
}
```

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

```java
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() {}
}
```

Iterátor

PreorderIterator a PostorderIterator vytváříme obdobně

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) );
        }
...
```

Iterátor

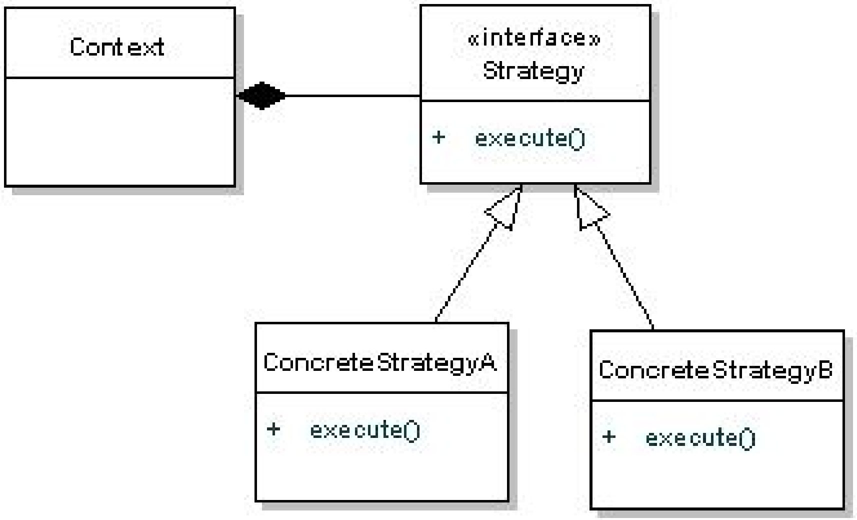
Klient, ze kterého použijeme iterátor =>

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

```java
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);
    }
}
```

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, když chceme flexibilně přepínat mezi různými algoritmy.



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

Strategy

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

```
interface PoliceStrategy {
    static final int speedLimit = 90;
    public void execute(int speed);
}
```

```
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");
    }
}
```

Strategy

Náš policejní okrsek je reprezentován pomocí Context objektu.

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

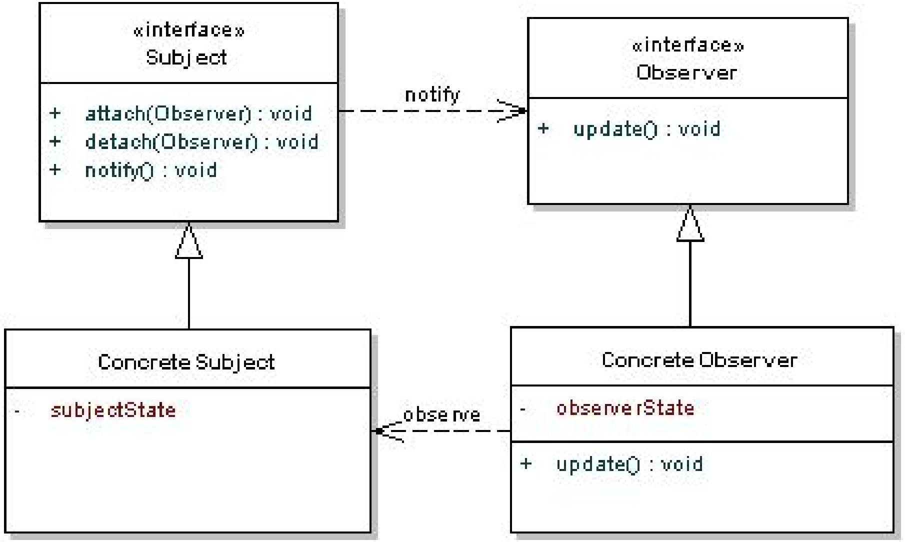
```
public class Client {
    public static void main(String [] args) {
        PoliceStrategy hardStrategy = new HardPoliceStrategy();
        PoliceStrategy niceStrategy = new NicePoliceStrategy();
        PoliceStrategy lazyStrategy = new LazyPoliceStrategy();

        Context newYorkPoliceDepartment = new Context(hardStrategy);
        newYorkPoliceDepartment.patrol(95);

        Context czechPoliceDepartment = new Context(niceStrategy);
        czechPoliceDepartment.patrol(95); //Does nothing

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

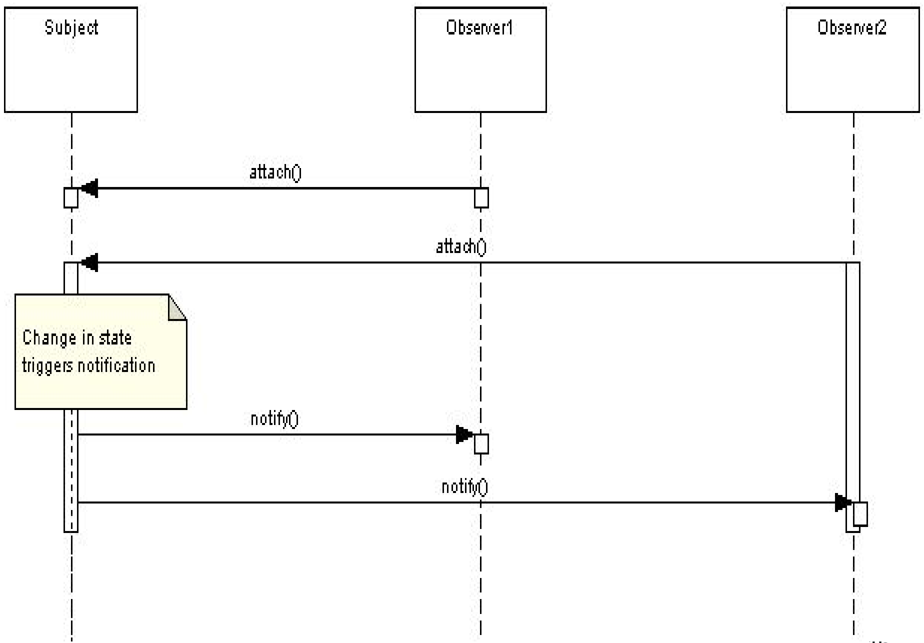
<=

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

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. Notofikace probíhá zavoláním metody na observeru. Tento pattern je často používán při implementaci GUI frameworků (MVC pattern)





Observer

Vytvoříme tři různé observers, každý z nich vypisuje stav Subjectu 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() {System.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());}
}
```

Observer

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

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

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

```
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();
       }
   }
}
```

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.

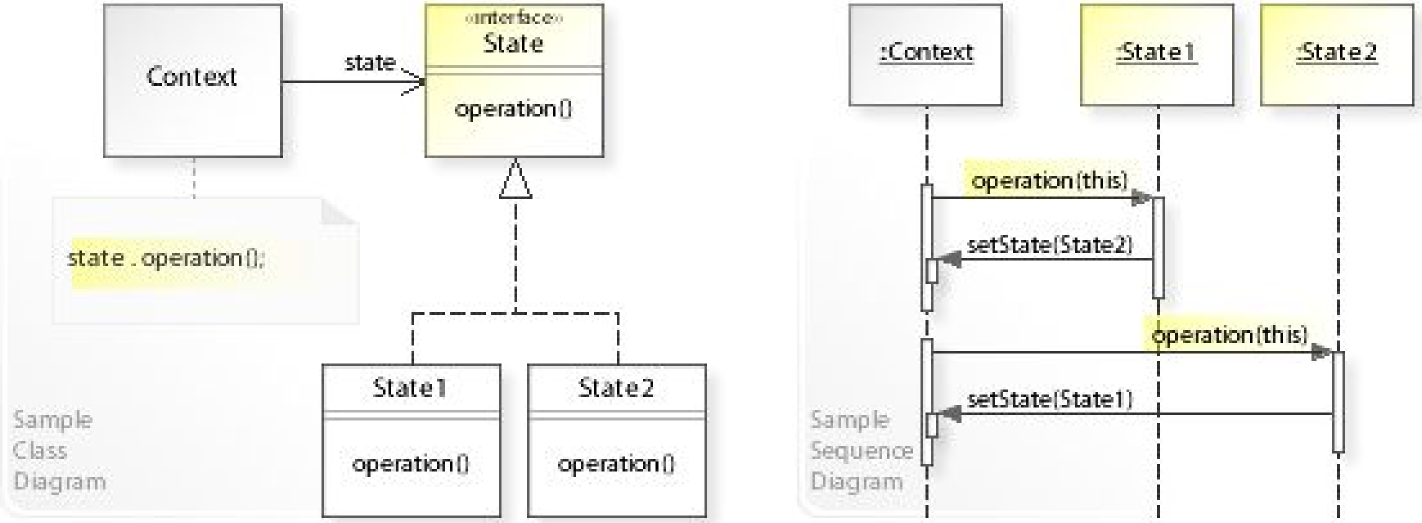
```java
public class Client {
   public static void main(String[] args) {
      Subject subject = new Subject();

      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
```

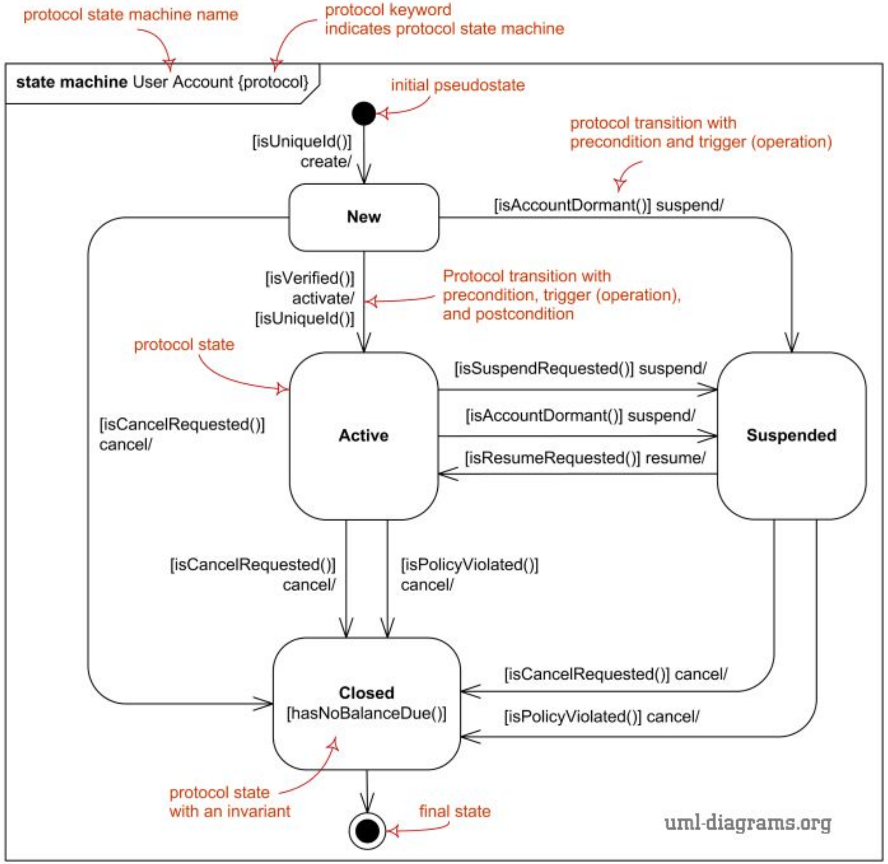
State

Implemenace principů stavového automatu pomocí OOP. Každý stav systému nebo komponenty je realizován mocí objektu State, který má předefinované 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 a postconditions.



State

Připomeneme si jak vypadá UML state diagram



State

Naimplementujeme přehrávač na CD. Začnem tím, že nadefinujeme třídu pro reprezentaci CD (public fields pro zkrácení kódu)

```
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.

```
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);
}
```

State

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.

```java
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){}
}
```

State

Pokračujeme pro další dva stavy

```java
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) { }
}
```

State

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;}
}
```

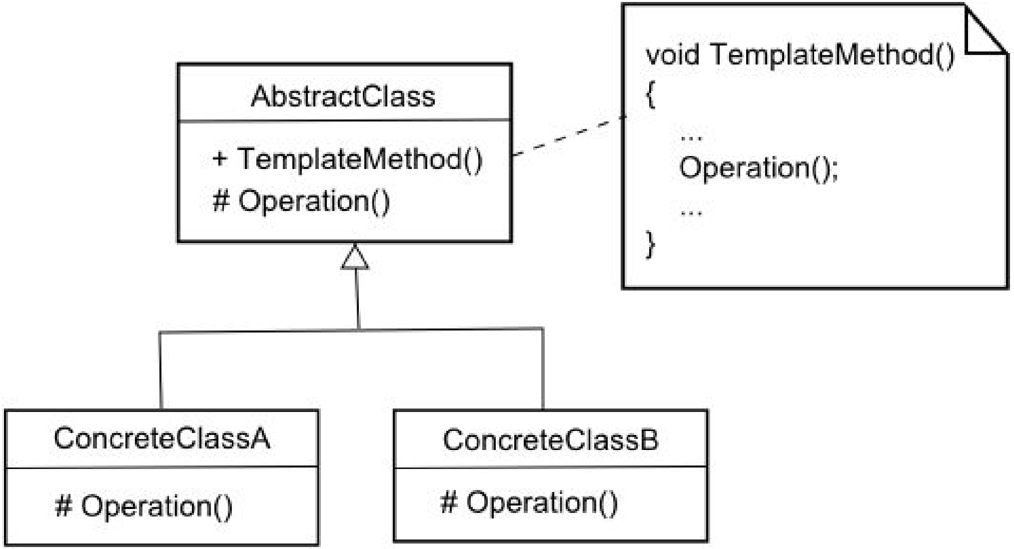
State

Následující klient ukazuje využití našeho audio přehrávače

```java
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
   }
}
```

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 subcslasses řídili chování, tak to řídí jejich abstraktní předek.



Template method

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

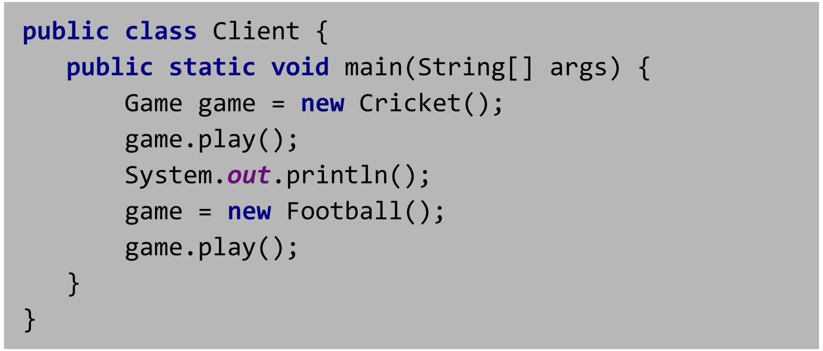
Pak nadefinujeme dva konkrétní potomky třídy Game.

```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 playing.");
    }
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the 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!");
    }
}
```

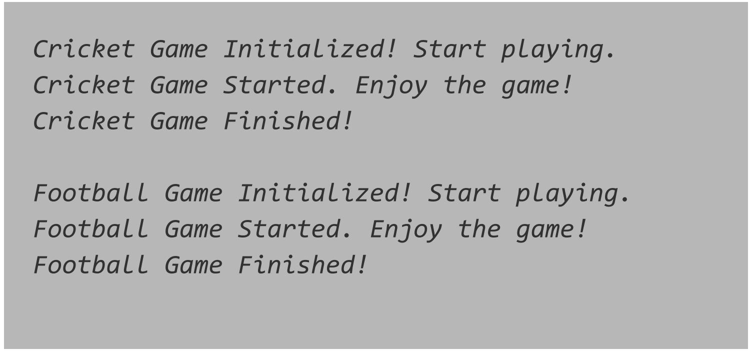
Template method

Využití pak vypadá následovně



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

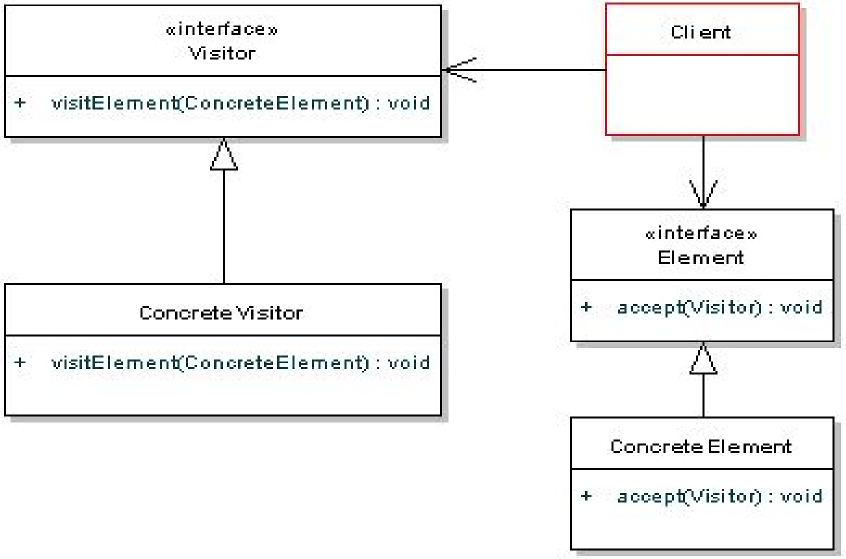
=>



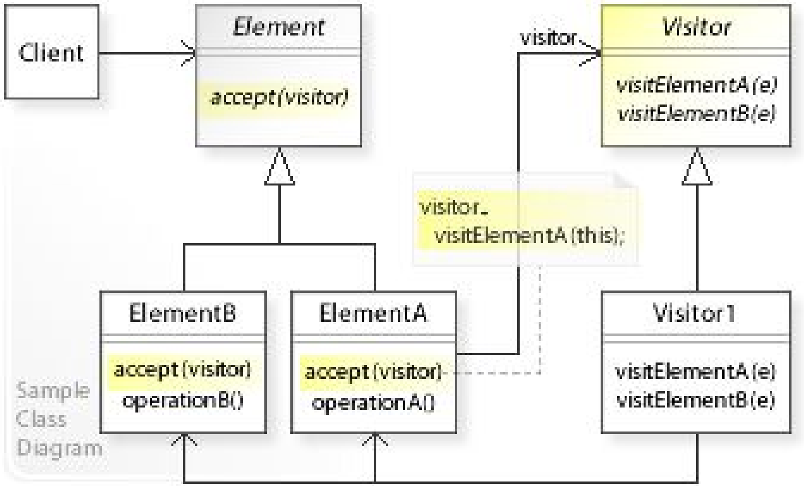
```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

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

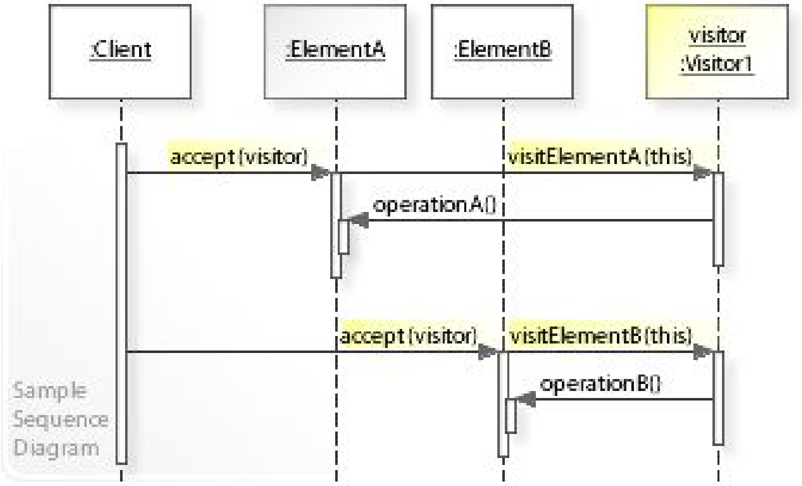
Visitor

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 referenci na vlastní element, který potřebuju v nové operaci.



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

Visitor

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

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

```java
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);
   }
}
```

```java
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);
   }
}
```

Visitor

Vytvoříme interface pro různé visitory pracující nad komponentami počítače.

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

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

```java
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.");
   }
}
```

Visitor

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

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

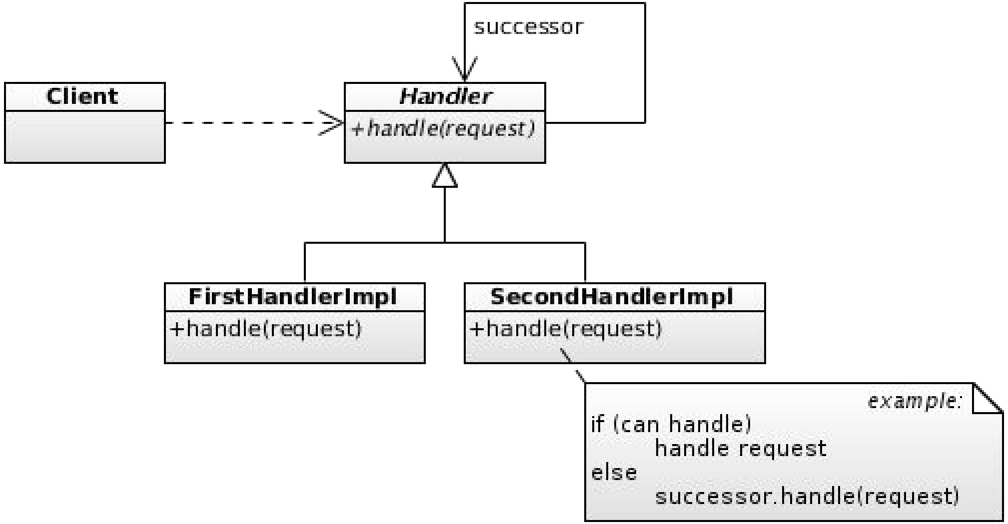
=>

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

Chain of responsibility

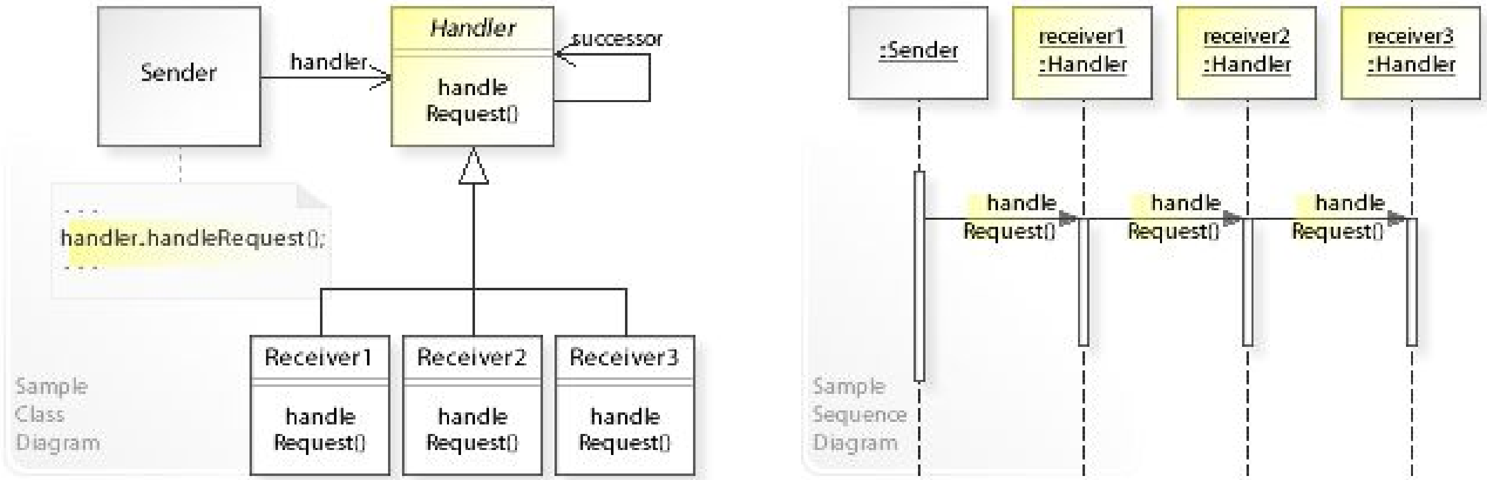
Vytvoření řetězu komponent, které postupně zpracovávají požadavek. Snižuje coupling mezi objektem posílajícím požadavek a objektem přijímající požadavek. Pošlu tedy požadavek a ten je “opracováván” nebo na něj reagují komponenty. Každá komponenta má nějaký jiný účel nebo řeší jinou doménu problémy. Každá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

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



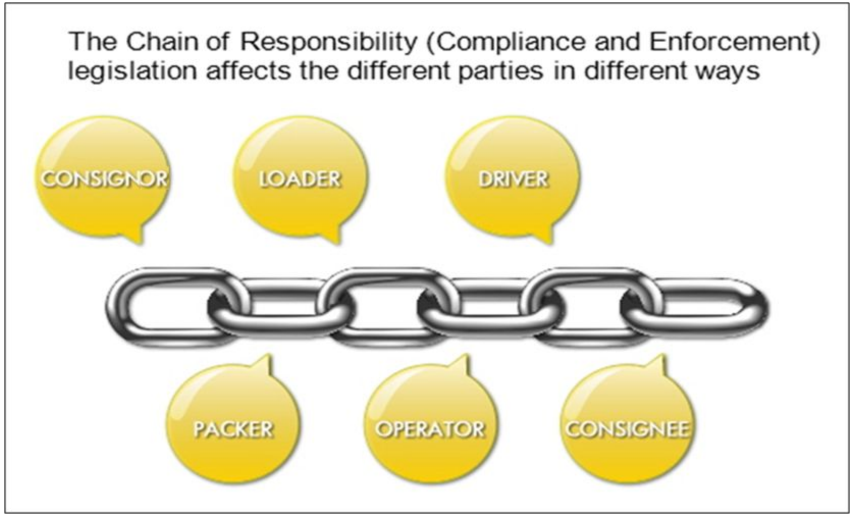
```
if (can handle)
        handle request
else
        successor.handle(request)
```

Chain of responsibility

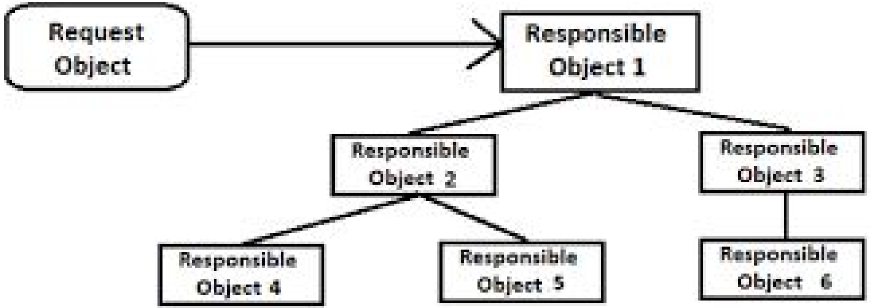


Chain of responsibility

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



The Chain of Responsibility (Compliance and Enforcement) legislation affects the different parties in different ways



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);
}
```

Chain of responsibility

Implementujeme jednotlivé logovací komponenty

```java
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);
   }
}
```

Chain of responsibility

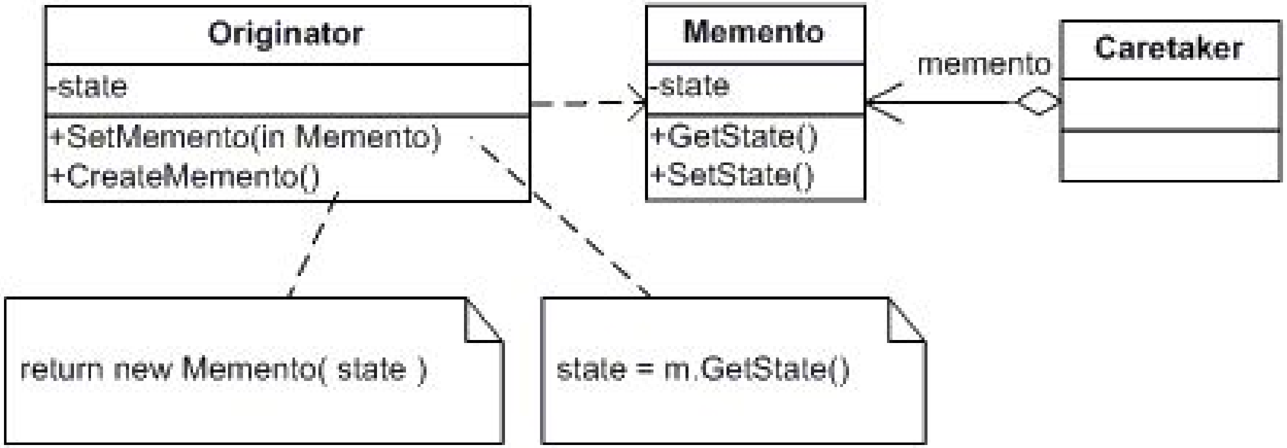
```java
public class Client {
    private static AbstractLogger getChainOfLoggers(){
        AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);
        AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);
        AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);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

=>

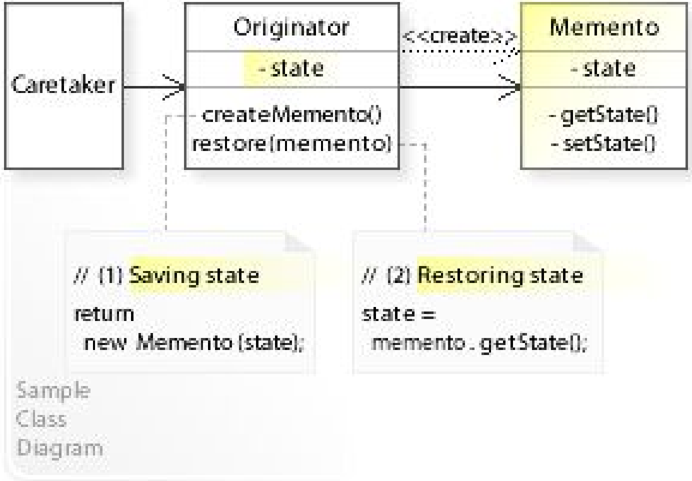
```
Standard Console::Logger: This is an information.
File::Logger: This is an debug level information.
Standard Console::Logger: This is an debug level information.
Error Console::Logger: This is an error information.
File::Logger: This is an error information.
Standard Console::Logger: This is an error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

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.



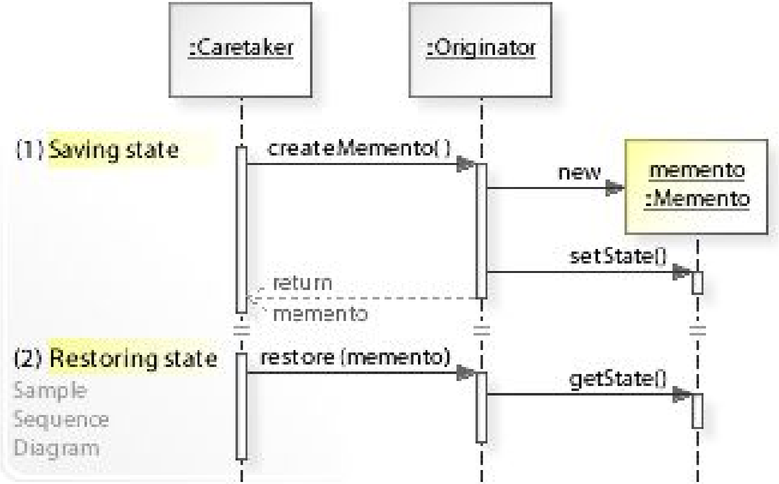
Memento



```
// (1) Saving state
return
  new Memento (state);
```

```
// (2) Restoring state
state =
  memento . getState();
```

Sample Class Diagram



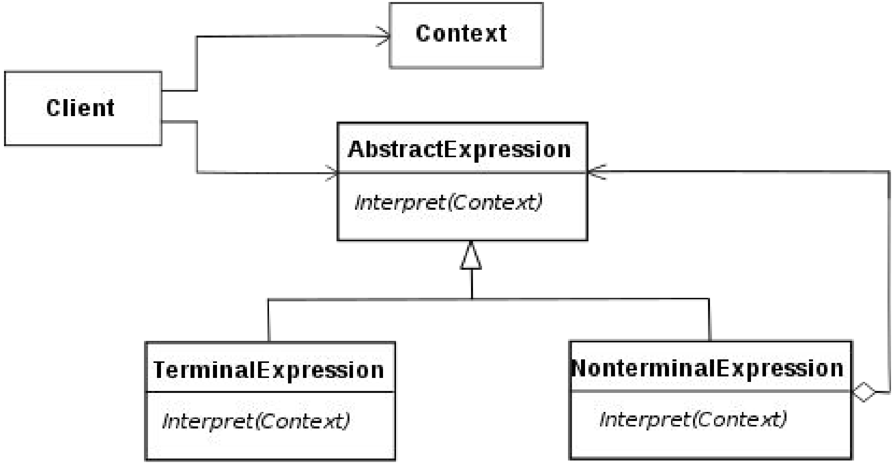
Sample Sequence Diagram

Interpreter

Zpracování (vyhodnocování) vět 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
- *neterminání* - řídí zpracování dalších symbolů

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



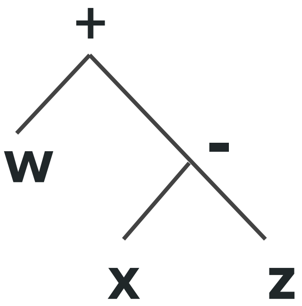
Interpreter

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:



Interpreter

Definujeme interface Expression pro reprezentaci terminálních a neterminálních symbolů:

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

A implementujeme ho pro operátory (+-) a operandy (čísla). Pozn. odčítání obdobné jako sčítání

```
class Number implements Expression {
    private int number;
    public Number(final int number)       { this.number = number; }
    public int interpret(final Map<String, Expression> variables)  { return number; }
}
```

```
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);
    }
}
```

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);
    }
}
```

Interpreter

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

```java
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);
    }
}
```