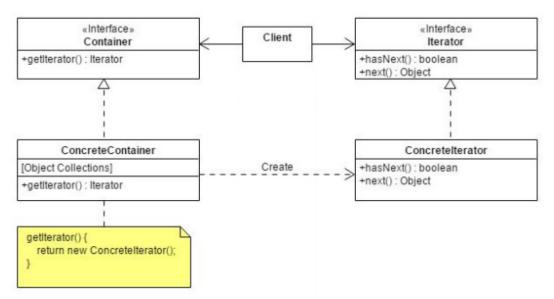
# Behavioral Design Patterns

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

### **Iterator**

- abstrakce algoritmu procházení (iterování) od implementace datové struktury
- klient řídí iteraci pomocí metod next(), isDone() a někdy i hasNext()
  - o nemusí znát vlastnosti ani implementaci datové struktury
- iterátor si udržuje stav procházení, aktuální položku a je schopen posunu na další prvek



- Container = struktura, kterou se snažíme procházet; vrací jednotlivé iterátory
- některé iterátory mají i metodu remove(), ale ne všechny struktury jsou modifikovatelné během procházení

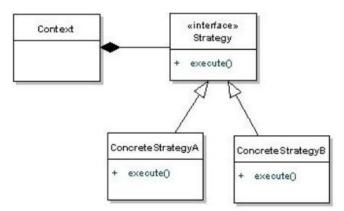
```
public class Client {

   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);
      }
   }
}
```

```
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;
       }
   }
```

## Strategy

- změna chování (nebo algoritmu) v průběhu běhu programu
- vytváříme tzv. context, který se chová různě dle toho, jakou má nastavenou strategii
- pro realizaci jednoho tasku, když chceme přepínat mezi různými algoritmy



- příklad:
  - o máme různé policejní okrsky
  - o chceme pro ně nastavit různé strategie při kontrole řidičů
  - o policejní okrsek reprezentován pomocí Context objektu

```
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");
    }
}
public class Context {
```

```
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);

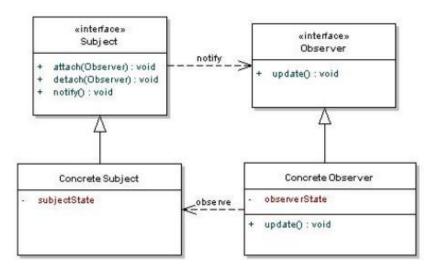
      Context czechPoliceDepartment = new Context(niceStrategy);
      czechPoliceDepartment.patrol(95); //Does nothing

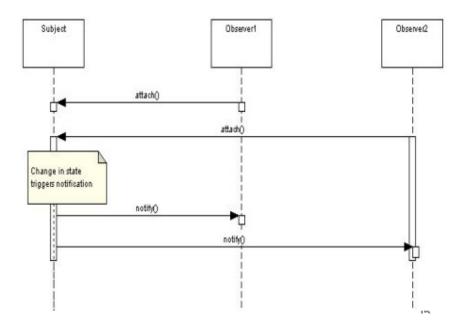
      czechPoliceDepartment.setStrategy(lazyStrategy);
      czechPoliceDepartment.patrol(150);
      czechPoliceDepartment.patrol(150);
      czechPoliceDepartment.patrol(150);
      czechPoliceDepartment.patrol(150);
    }
}
```

```
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 změny stavu komponenty s komponentami reagující na tuto změnu
- často používán při implementaci GUI frameworků (MVC pattern)
- **Subject** udržuje si seznam objektů "**observers**"
  - ty notifikuje pokaždé, když dojde k jeho změně
  - o notifikace probíhá zavoláním metody na observeru





#### - příklad:

- 1.) vytvoříme 3 různé observery, každý z nich vypisuje stav Subjektu v jiném formátu
- 2.) nadefinujeme abstraktní Observer přes metodu update() notifikuje Subject jednotlivé observery
- 3.) Subject umožňuje správu observerů, při změně stavu všechny notifikuje
- 4.) vytvoříme subjekt, zaregistrujeme observery, modifikujeme stav subjektu, observery jsou notifikovány a provedou metodu navázanou na změnu stavu

```
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());}
}

public abstract class Observer {
```

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

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

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

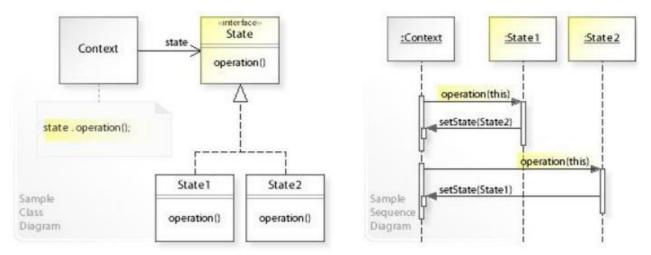
      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");

      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

- implementace principů stavového automatu pomocí OOP
- každý stav systému/komponenty realizován pomocí objektu State
  - o ten má také předdefinované rozhraní pro realizaci přechodů (transitions) mezi stavy
  - o State může obsahovat kód pro kontrolu preconditions/postconditions



- příklad přehrávač CD:
  - 1.) 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;
   }
}
```

2.) nadefinujeme rozhraní, přes které budeme posouvat stavový automat

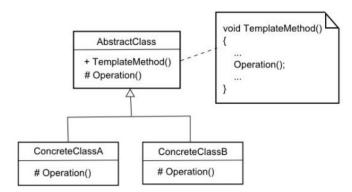
```
public interface AudioPlayerState {
   public void insertCD(AudioPlayerContext context, CD cd);
   public CD ejectCD(AudioPlayerContext context);
   public void play(AudioPlayerContext context);
   public void stop(AudioPlayerContext context);
}
```

- 3.) nadefinujeme stavy přehrávače ty implementují rozhraní, které jsme definovali
  - a. nedává-li stavový přechod smysl, neděje se nic
  - b. pokud ano, na kontextu se provede přenastavení stavu
- 4.) vlastní audio přehrávač = context objekt ze state design patternu

```
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){}
                                                              ... podobně pro další dva stavy
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;}
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
- pro variantní části chování se liší dle toho, která subclass je implementuje
- podobný princip jako u IoC místo subclass řídí chování jejich abstraktní předek



- příklad:
  - 1.) implementujeme abstraktní třídu Game (ta předepisuje tři metody + algoritmus v jakém pořadí se budou volat)

2.) nadefinujeme dva konkrétní potomky třídy Game

```
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!");
   }
}
```

3.) využití vypadá následovně:

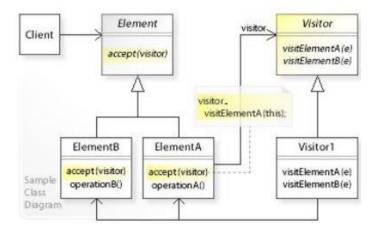
```
public class Client {
   public static void main(String[] args) {
      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

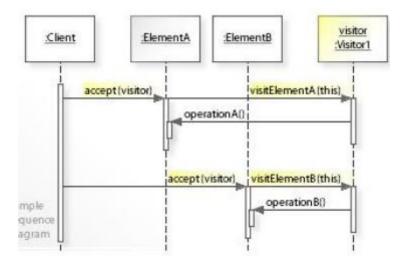
```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

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 struktury přidávat další operace aniž bych ji musel modifikovat (open-closed princip)
- realizace patternu pomocí tzv. double dispatch principu
  - elementy, u kterých chci v budoucnosti přidávat operace mají metodu accept(Visitor)
    - uvnitř ní zavolám na vloženém visitoru metodu visitElement(Element)
    - tím předám řízení visitoru a zároveň mu předám referenci na vlastní element, který potřebuji v nové operaci





příklad:

1.) Implementujeme datovou strukturu pro osobní počítač. Do klíčových komponent PC 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);
    }
}
```

2.) Vytvoříme interface pro různé visitory pracující nad komponentami PC.

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

3.) Implementujeme konkrétní realizaci daného visitoru. Tento vypisuje info o komponentách do konzole.

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

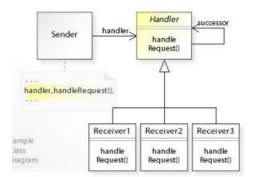
```
public class Client {
    public static void main(String[] args) {

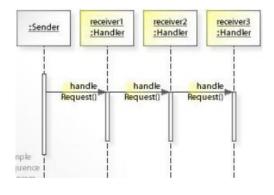
        ComputerPart computer = new Computer();
        computer.accept(new ComputerPartDisplayVisitor());
    }
}
```

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
- každá komponenta má jiný účel, popř. řeší jinou doménu problému
- každá komponenta rozhoduje, má-li požadavek poslat další komponentě v pořadí
- ChoR využívá EDA (Event Driven Architecture) nebo frameworky, kdy je třeba na request aplikovat různé aspekty, anebo nalézt komponentu, která jej je schopná odbavit





- komponenty řešící požadavek lze řetězit do lineárních seznamů nebo i stromů
- **příklad**: Chceme implementovat logovací systém. Vytvoříme jej pomocí logovacích komponent, kde každá loguje jiný druh informací a je možné řetězit je za sebe.

### 1.) 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);
}</pre>
```

#### 2.) Implementujeme jednotlivé logovací komponenty.

```
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);
    }
}
```

```
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.

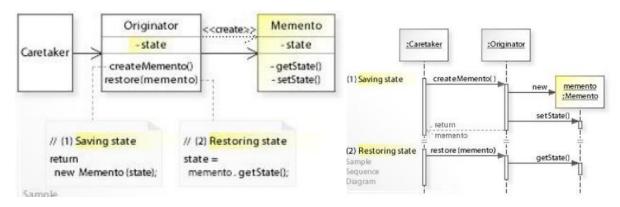
Standard Console::Logger: This is an error information.
```

#### Memento

- slouží pro navrácení objektu k jeho přechozímu stavu
- např. pro realizaci undo/redo operací
- tři základní entity:
  - Originator
    - komponenta, kterou modifikujeme (měníme její stavy)
    - chceme mít možnost se k těmto stavům vracet
  - Caretaker
    - provádí dané změny
    - než je provede, vyžádá si aktuální stav

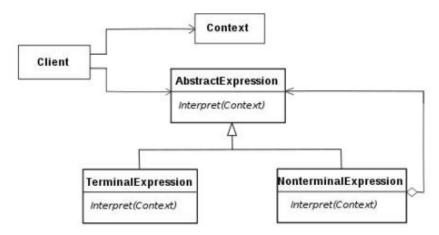
#### o Memento

- v tomto objektu mu Originator vrací aktuální stav
- Caretaker si udržuje list "Mement", může se tak k nim vracet



#### Interpreter

- zpracování (vyhodnocování) vět vytvořených v určitém jazyce
- pro každý symbol je vytvořena třída, která ho dokáže zpracovat
- pro prezentaci se často používá tzv. syntaktický strom (pro strukturu věty)
- symboly mohou být:
  - o terminální dále se nepřepisují (vyhodnocení v nich končí)
  - o **neterminální** řídí zpracování dalších symbolů



- příklad:
  - O Chceme spočítat výraz zapsaný větou "w x z +".
  - o Jedná se o matematický výraz zapsaný v postfix zápisu "x-z+w".
  - Věta obsahuje tři typy symbolů:
    - číslo (terminální)
    - sčítání (neterminální)
    - odčítání (neterminální)
  - o Z výrazu lze sestavit následující syntaktický strom:



1.) Definujeme Exression pro reprezentaci terminálních/neterminálních symbolů.

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

2.) Implementujeme jej pro operátory (+-) a operandy (čísla).

```
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);
    }
}
```

3.) Implementujeme parset vět do stromu. Využijeme zásobníku pro sestavení postfix zápisu.

```
{\bf class} \ {\bf Evaluator} \ {\bf implements} \ {\bf 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);
```

4.) Provedeme interpretaci při dosazení proměnných (w = 6, x = 8, z = 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);
    }
}
```