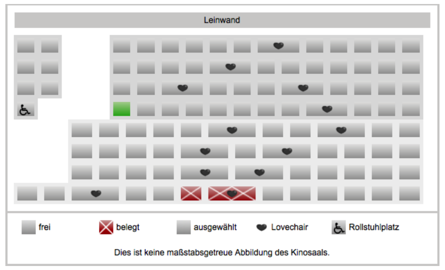
SE II
Software Design Pattern

Prof. Dr.-Ing. S. Becker

Motivation

Movie Theater



Webfrontend

Mobile App

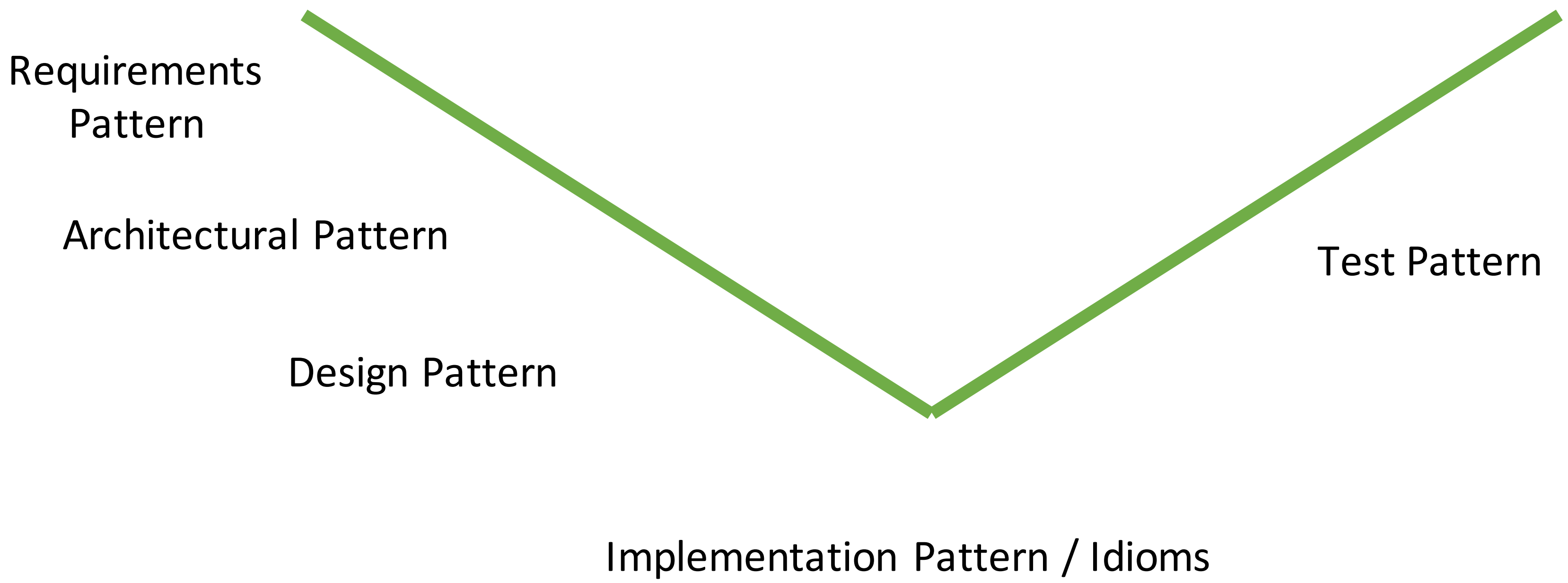
Cash Desk-PC

Unique Problem?

- No, Problem reoccurs
  (Train-/Flightbooking, Excel Charts, …)
  - Central Subject
  - Set of flexibale and extensible viewers
- Not reinvent a solution but reuse existing ones
  - —> Pattern

Pattern: Definition

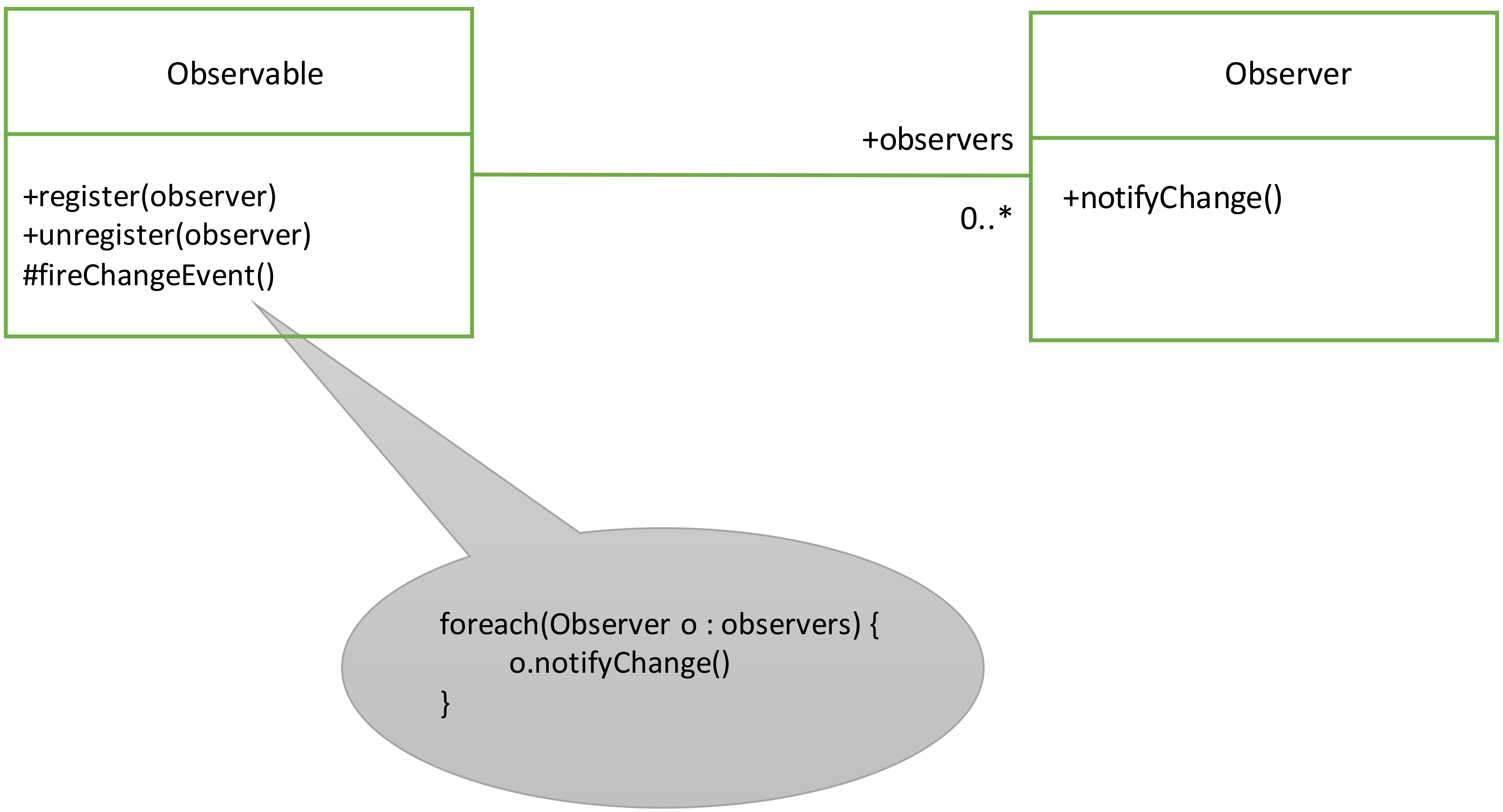
- Pattern are established solution schemata for reoccuring problems
- Depending on the type of problem we distinish different types



Observer: Behaviour

- How to realize an observer?
  - 1. View registers itself at the observable
  - 2. Observable notifies all views on any state change
  - 3. Views unregister themselves

Observer: Implementation



Back to the problem…

- Reminder: Which characteristics had our problem?
  - All applications should display the same, consistent view of the theater
  - Changes should update the viewers as soon as possible
- How to efficiently **find** a solution schema for this?

Observer: Specification schema

- *Name*: Observer
- *Alternative Names*: Publish-Subscribe, Listener
- *Problem*: Changes on a central subject should alter the display of a unknown number of dependant viewers
- *Advantages*: Abstract coupling, Broadcast notifications
- *Disadvantages*: Unexpected updates
- [..]

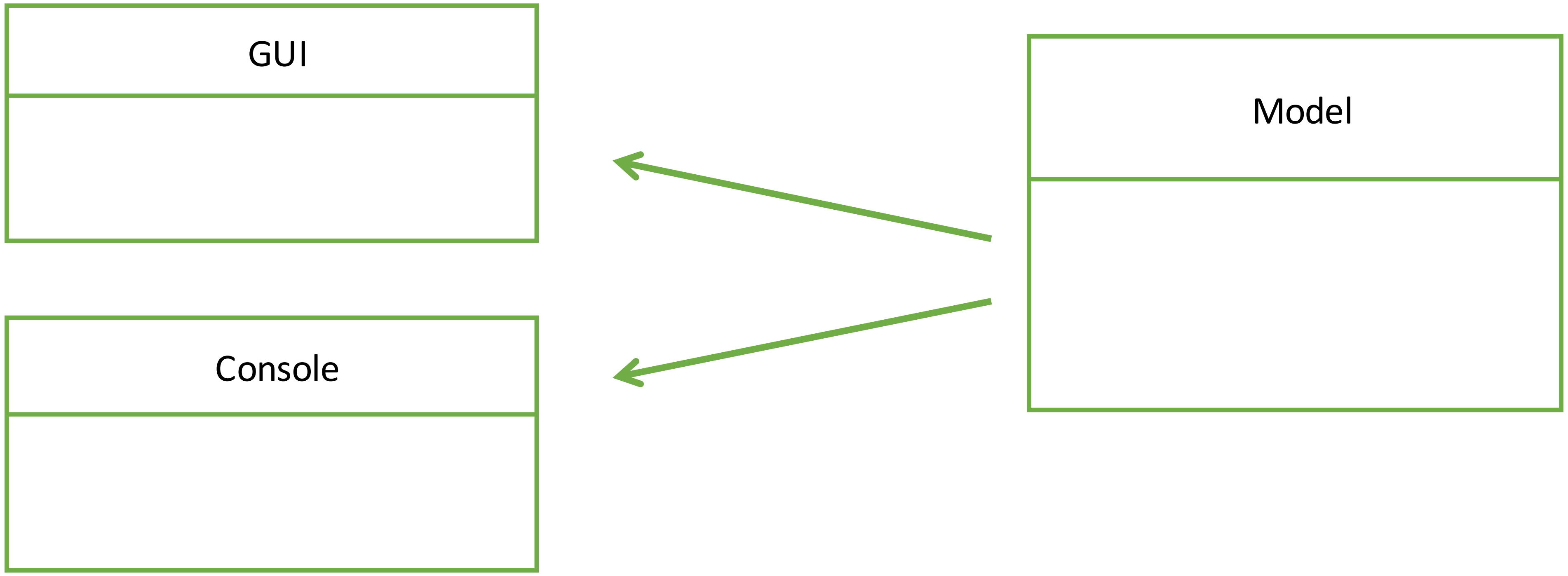
As a result we get a pattern catalouge

Beobachtermuster: Varianten

- Pull-Model
  - Observable lets the viewers pull for the state they are interested in
  - Decision what to query left to the viewers, they only get to know *that* something has changed, but not *what*
  - Observable has (public) methods for viewers to get relevant state info
- Push-Model
  - Observable sends all state change information in the notification
  - Viewers do not need to query the observable for its state
  - However, viewers cannot be reused that easy any longer (depend on the notification parameter's type)

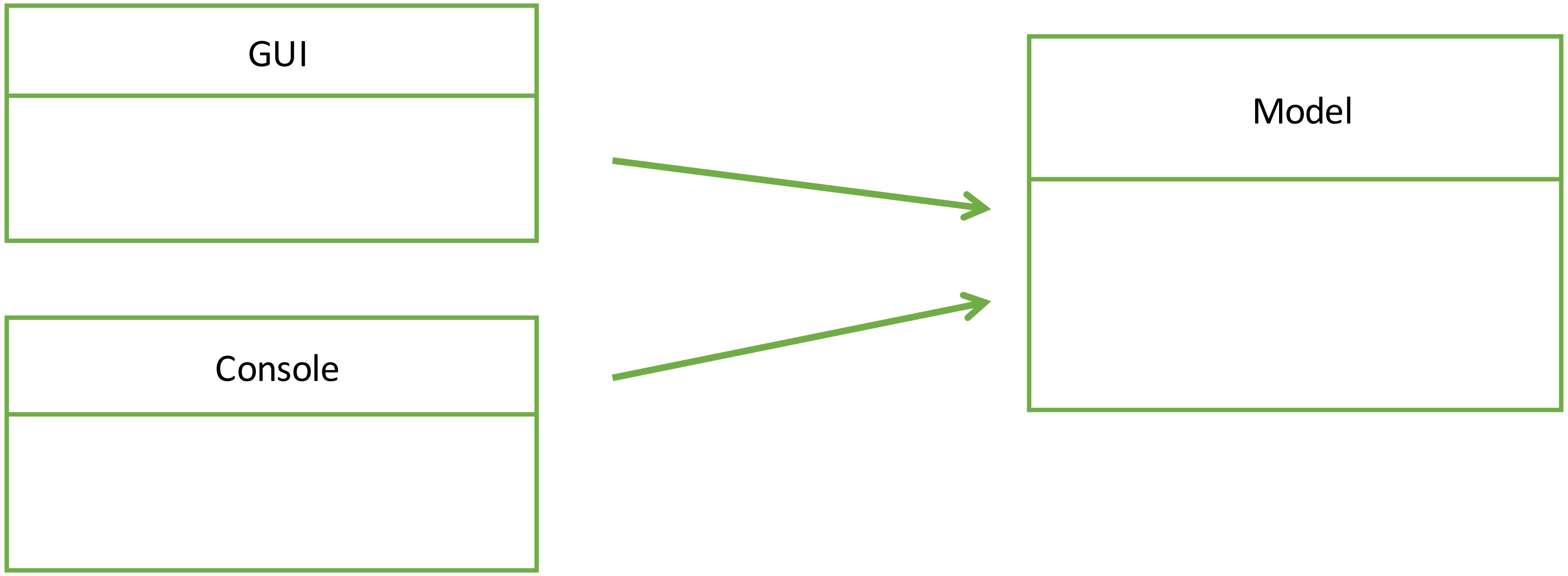
Consequences: Dependencies

- Without observer pattern, model depends on its views
  - The model has to know its views
  - The model might even update the views accordingly
  - → Adding more views / view types is difficult



Consequences: Dependencies

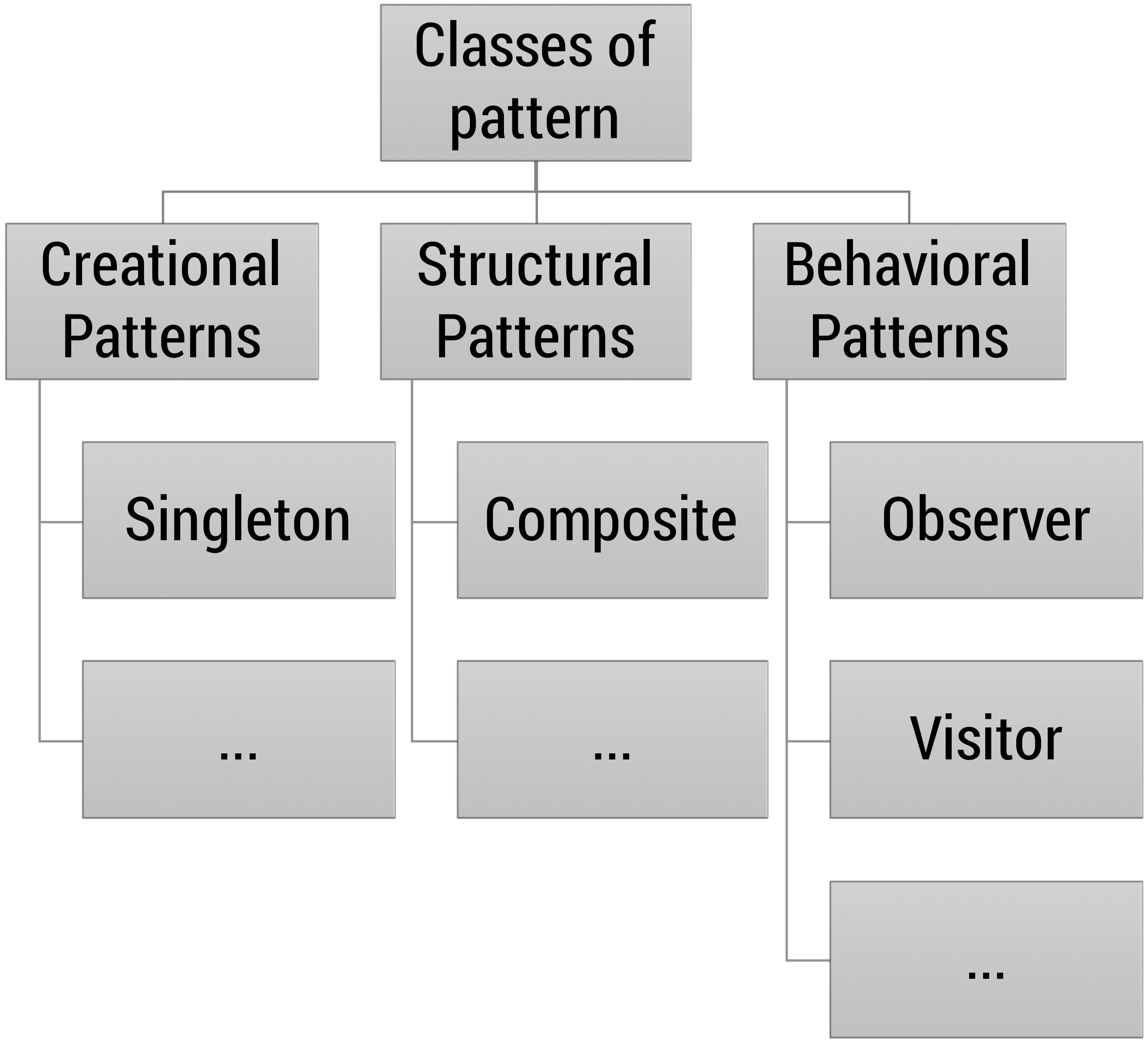
- With observer, views depend on the model
  - The model only knows about the abstract observer interface
  - Any concrete view know the state query methods of the model it displays



Overall effect
Dependency Inversion and Inversion of Control

- Dependency Inversion
  - W/o observer:
    Model knows views
  - With observer:
    Views know the model
- Inversion of control
  - W/o observer:
    Model controls all updates
  - With observer
    Each view updates itself

Classification of design patterns



Classification
Even more dimensions…

- What?
    - Creational pattern
    - Structural pattern
    - Behavioural pattern
- How?
    - Whole objects
    - Split objects
- When?
    - Design
    - Architecture
    - …

Creational pattern: Singleton

Design Pattern: Singleton

Problem

- A class should have only a single instance, as the underlying ressources it controls are unique
- e.g. one Window Manager, one file system, one universe, ...

Solution idea

- „Ensure a class only has one instance (object), and provide a global point of access to it.“
- The class itself has the control over its one and only instance

Pattern type

- Creational pattern

How to implement this in Java?

Java Realization

```java
package lecture1;

public class Universe {
  private final static Universe theUniverse
     = new Universe();

  private Universe() {
     super();
  }

  public static Universe getUniverse() {
     return theUniverse;
  }
}
```

Constructor and class variable private

Only a single method returns the singleton object

Universe.java

Issues?

- Multithreading
  - The construction of the singleton object is not thread-safe
    → Race condition
  - Multiple objects of the class Universe might be created
- Virtual machine class loader
  - Static variables are unique per class loader
  - However, clients can instantiate multiple class loaders

Thread-safety: Double null check solution

```java
package lecture1;

public class Universe {
  private final static Universe theUniverse;

  private Universe() {
      super();
  }

  public static Universe getUniverse() {
      if (theUniverse == null) {
        synchronized(Universe.class) {
           if (theUniverse == null) {
               theUniverse = new Universe();
           }
        }
      }
      return theUniverse;
  }
}
```

Universe.java

Thread-safety and class loader

```
package lecture1;

public enum Universe {
  THE_UNIVERSE;

  public void doSth() {
    System.out.println(THE_UNIVERSE);
  }
}
```

Universe.java

Insight

- Even simple patterns might get tricky to implement, depending on their usage context
  - Multi-threaded systems
  - Multiple class loaders
  - Distributed environments
  - No synchronized clocks
  - …
- Consider your context before starting to implement a solution

Design pattern
Visitor

Design Pattern: Visitor

Problem

- Elements of a datastructure should be manipulated/traversed (in general: an operation should work on them) in many different ways. Operations might depend on the type of node they encounter
- Operations should be extensible by third-parties

Solution idea

- „Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.“
- The datastructure has to provide a method that allows visitors to operate on them

Class of pattern

- Behavioural pattern

Application scenarios

- Visitor often used to manipulate graphs, e.g., trees
- In particular Abstract Syntax Trees (AST)
- Created by the compiler while compiling source code
- Used for the following set of operations
  - Type checks
  - Optimizations
  - Code generation
  - …

Generic classes
A short excursion

- Example: Java
  - Generic classes
  - Generic interfaces

- Generic classes are templates for concrete classes in which certain component types have not been defined yet. Such types are represented as type variables only. These classes form so called open types.
- A concrete class is created by assigning concrete types (type arguments) to the type variables. The class becomes a closed type.

An example: A generic pair

```
package lecture1;

class Pair<T> {
  private T first;
  private T second;
  Pair(T e, T z) {
    first = e;
    second = z;
  }
  public void setFirst (T e) {
    first = e;
  }
  public T getFirst ()  {
    return first;
  }
  // ...
}
```

Declaring a type variable

Using the type variable

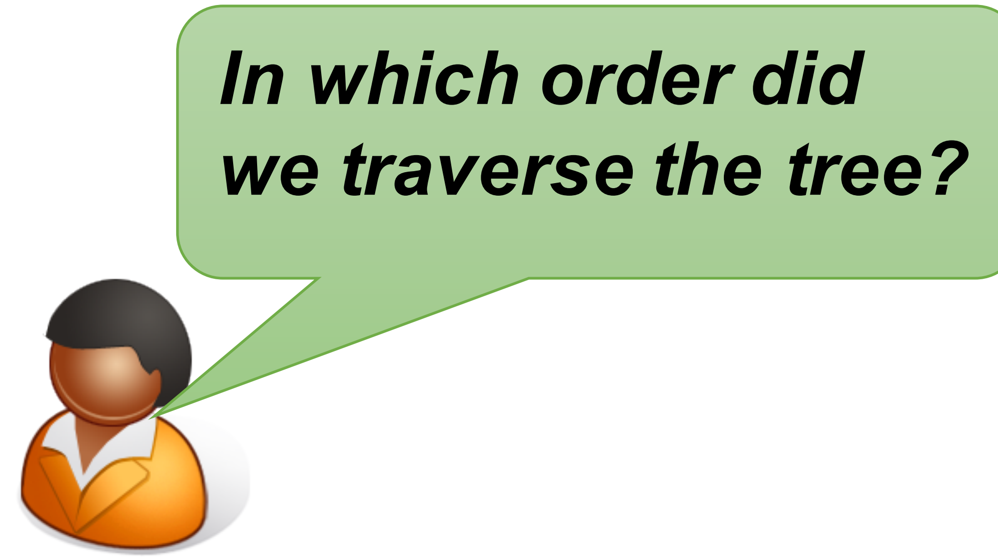
Pair.java

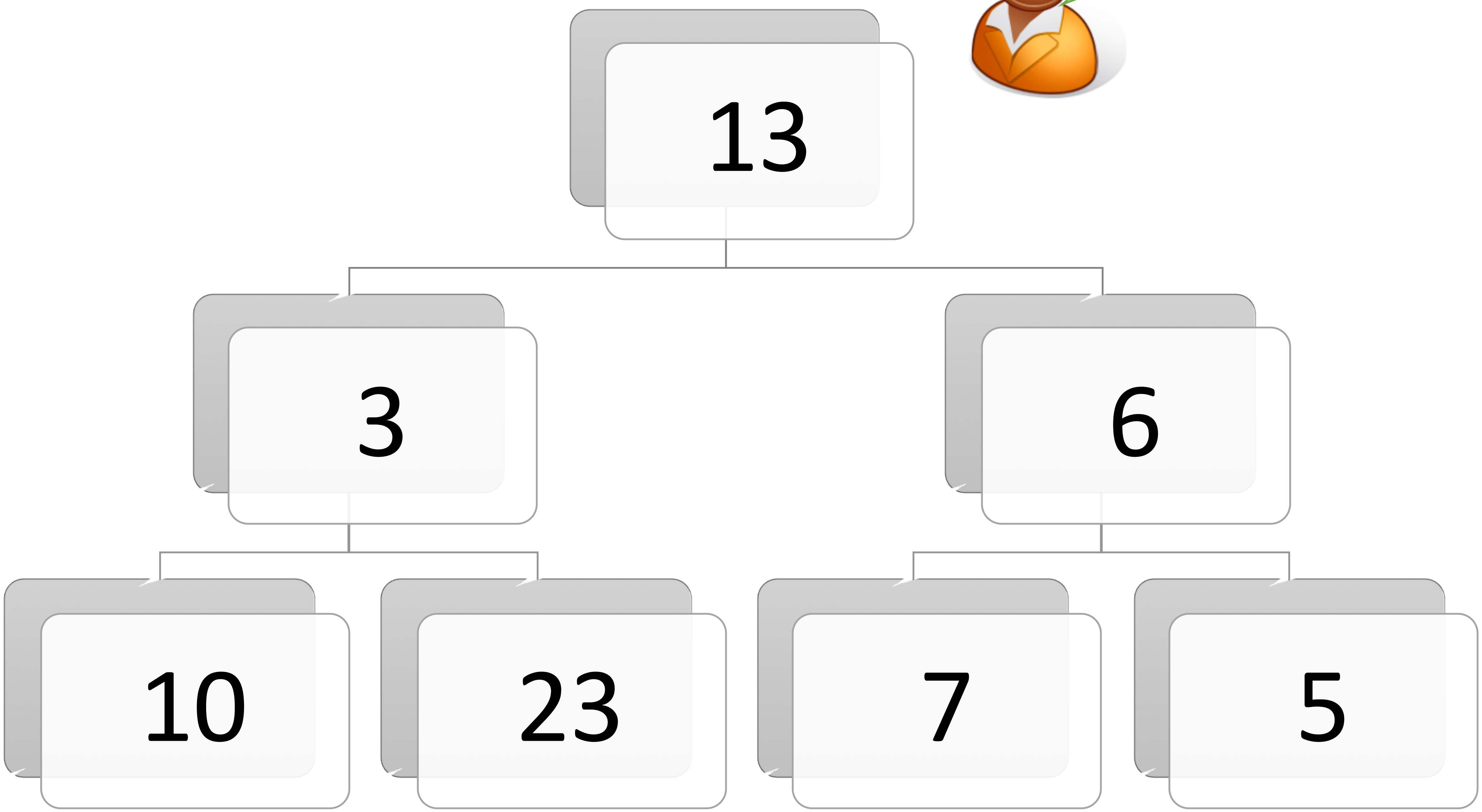
Using Pair

```
package lecture1;

class Student {}

class PairTest {
  public static void main(String[] args){
    Pair<Integer> pi =
        new Pair<Integer>(
            new Integer(4),
            new Integer(32));
    Pair<String> ps =
        new Paar<String>("bar","foo");
    Pair<Student> pst = new Paar<Student>(
        new Student(), new Student());
    // etc.

    // The following becomes valid now…
    Integer i = pi.getFirst();
    ps.setFirst("bla");
  }
}
```

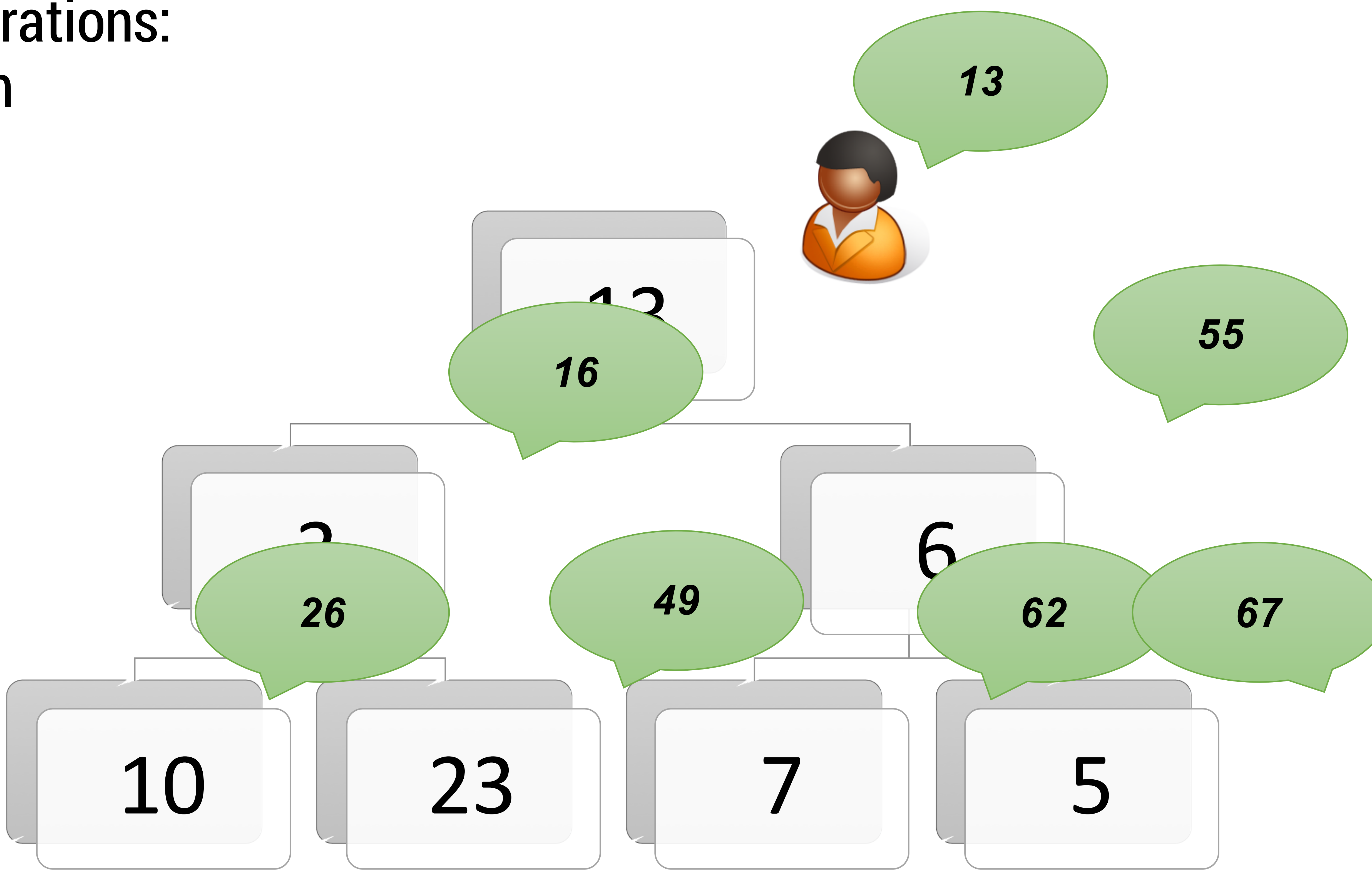
PaarTest.java

Back to the visitor…
Example: Binary tree

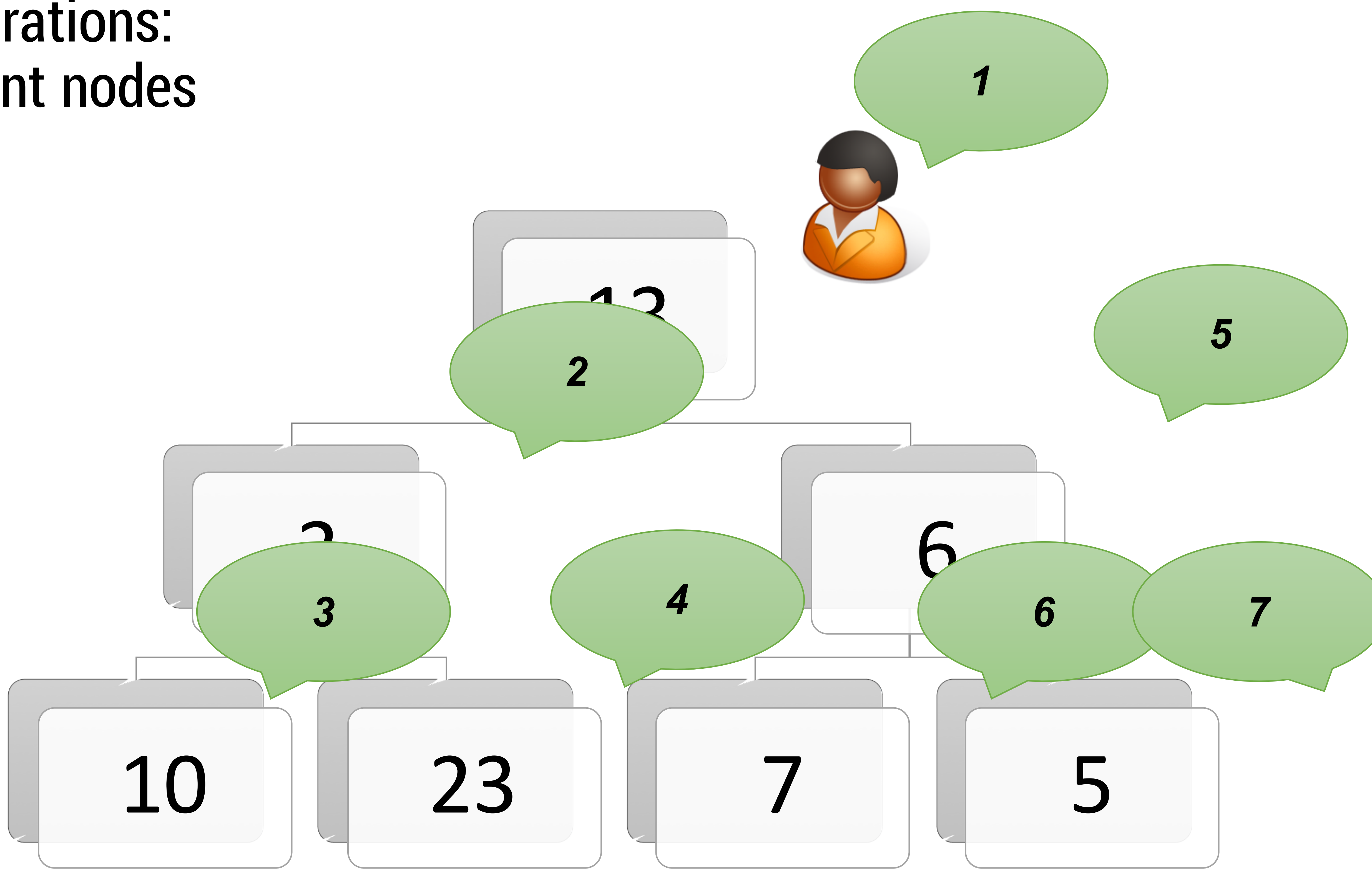




Operations:
Sum



Operations:
Count nodes



Subtasks

Implement tree

- Dynamic datastructure
- Using arbitrary node types

Traverse the tree

- Determine order
- Stop at leave nodes

Perform node operatios

- Examples
  - Sum
  - Count nodes
  - Print node values

First solution: A generic tree in Java

```
package lecture1;

public class Tree<G> {
  private final Tree<G> left;
  private final Tree<G> right;
  public G val;
  public Tree(Tree<G> left,G val,Tree<G>
    super();
    this.left = left;
    this.val = val;
    this.right = right;
  }
  public void inOrder () {
    if (!isEmpty(left))
      left.inOrder();
    System.out.println(val);
    if (!isEmpty(right))
      right.inOrder();
  }
  private boolean isEmpty(Tree<G> node){
    return node == null;
  }
}
```

Implements inorder tree traversal

Recursion

Tree.java

Example

```
package lecture1;

public class TreeTest {
  public static void main(String[] args){
    Tree<Integer> t = new Tree<Integer>(
        new Tree<Integer>(
          new Tree<Integer>(
            null,
            1,
            null),
          4,
          new Tree<Integer>(null,7,null)),
        3,
        new Tree<Integer>(null,28,null));
    t.inOrder();
  }
}
```

How does the tree look like?
What is the result of t.inOrder()?

TreeTest.java

Traversal revisited...

```
public void whateverOrder () {
  if (right != null)
    right.whateverOrder();
  System.out.println(val);
}
```

In which order does this method traverse the tree?

Tree.java

Issue: Operations not exchangable

```
public void inOrder () {
    if (!isEmpty(left))
        left.inOrder();
```

Solution so far consists of two parts

- Iterate all/some nodes (green)
- Perform operation on nodes found

However, both aspects are
(a) mixed and hence (b) **cannot be exchanged**
→ (Quality-) Goal not yet reached

Visitor for trees

Next step

- Generalize the inOrder traversal so that we can execute **arbitrary operations**
- We keep iteration schema constant
- → Operations become flexible/exchangable

Realization

- inOrder gets additional parameter of type interface IVisitor
- Each Visitor has method process, realizing the operation to perform on a node
- The visitor interface is implemented by different classes, e.g.,
  - TreePrinter: prints visited nodes
  - TreeCounter: counts the nodes
  - TreeIncrementer: increments each node‘s value
- inOrder calls the process–Method of conrete instances → polymorphic dispatch

inOrder

```java
public void inOrder (IVisitor<G> visitor) {
   if (!isEmpty(left))
     left.inOrder(visitor);
   visitor.process(this);
   if (!isEmpty(right))
     right.inOrder(visitor);
}
```

Due to the interface abstraction, the datastructure does not need to know any concrete visitor

Tree.java

Visitor Interface

```java
package lecture1;

public interface IVisitor<G> {

  public void process(Tree<G> tree);

}
```

A single method to encapsulate the operation to be performed on each node

IVIsitor.java

Implementing classes

```java
package lecture1;

class TreePrinter implements IVisitor<Object> {
   public void process(Tree<Object> t) {
      System.out.println(t.val.toString());
   }
}

class TreeCounter implements IVisitor<G> {
   private int count;
   TreeCounter() {
      count = 0;   }
   public void process(Tree<G> t) {
      count++;  }
   public int getCount () {
      return count;
   }
}
```

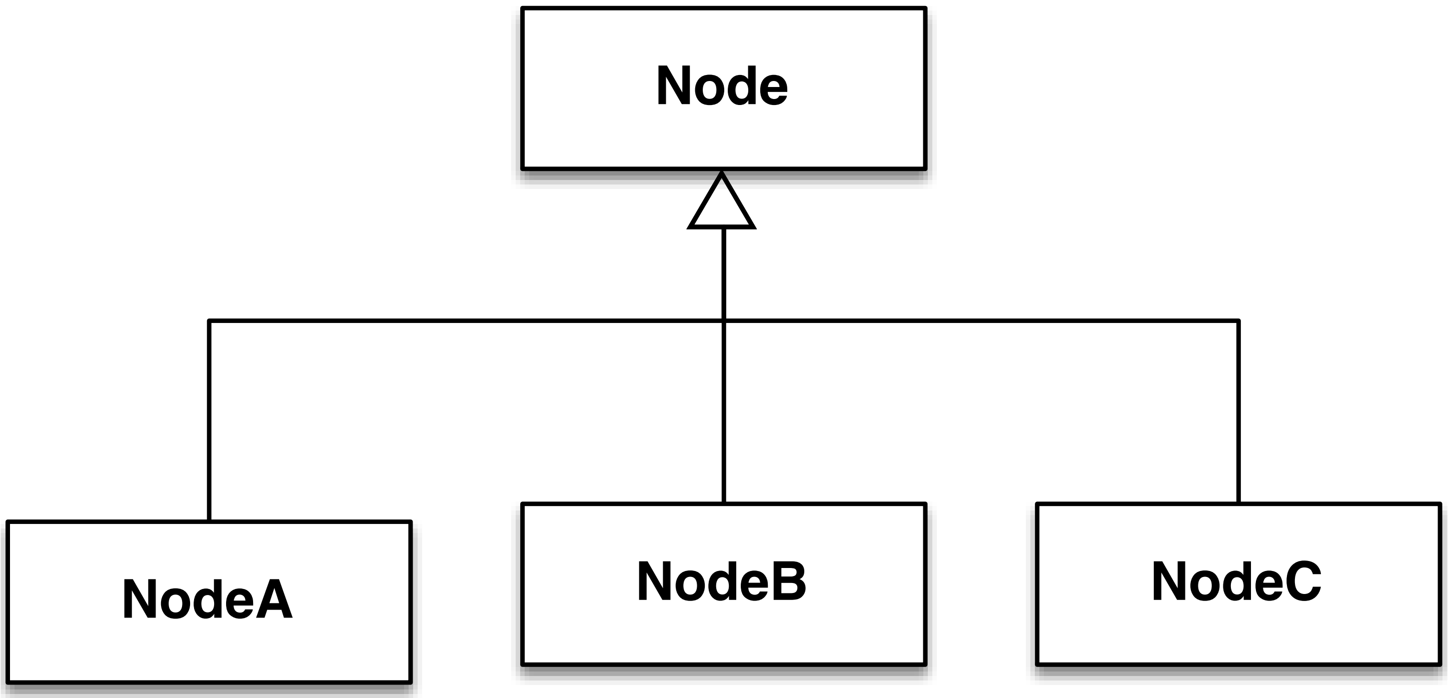
Tree*.java

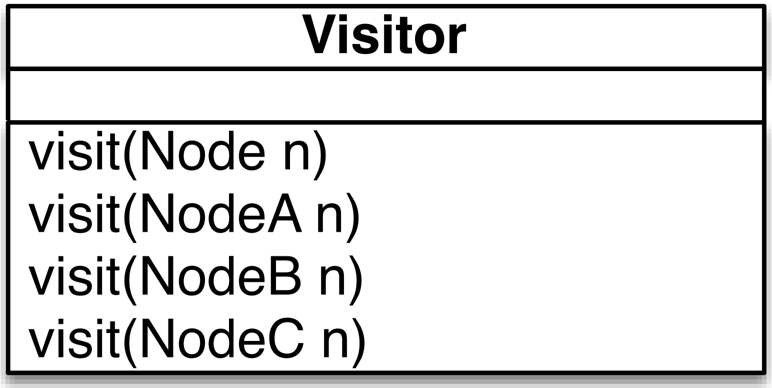
Visitor so far...

- Encapsulate data structure realisation from the operations working on the datastructure
- Advantage: We can extend the set of operations
  - Extensible
  - More flexible
- So far: Simple visitor variant
  - All visited nodes have the same type
  - Next we realize different node types where the type in addition determines the operation to perform

Vistor and Node specific behaviour

Node specific behaviour





Implementing classes: Visitor

```
package lecture1;

class Visitor implements IVisitor<Object> {
  public void visit(Node n) {
     System.out.println(“Node!”);
  }

  public void visit(NodeA n) {
     System.out.println(“NodeA!”);
  }

  …
}
```

Tree*.java

Implementing classes: Visitable nodes

```
package lecture1;

class Node{
  public void visit(IVisitor visitor) {
    visitor.visit(this);
  }
}

class NodeA{
  public void visit(IVisitor visitor) {
    visitor.visit(this);
  }
}
```

Dynamic dispatched visit calls

Tree*.java

Lessons learned

- This lecture…
  - Know and apply the pattern concept
  - Use specification schemata for pattern
  - Implement some example pattern
- Next lecture: More patterns...