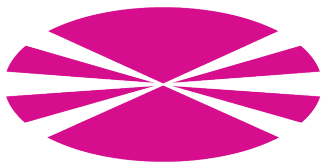
Unit 6: Design Patterns

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science and Information Technology
Faculty of Computer Science

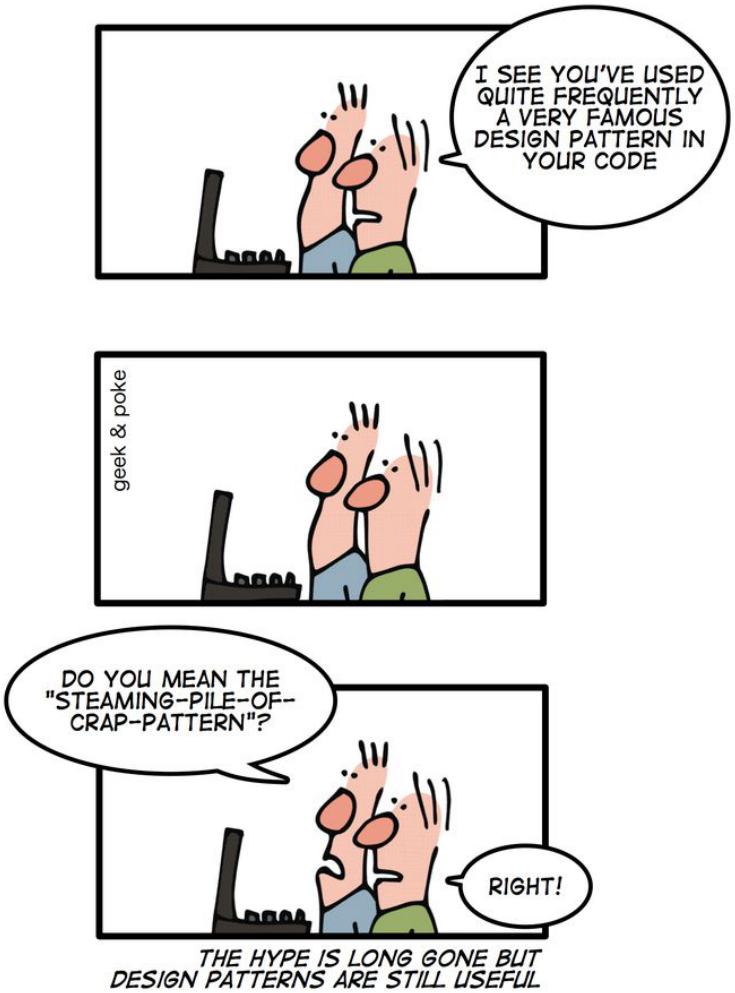


UNIVERSIDADE DA CORUÑA

Table of Contents

Table of Contents

Design patterns



Design patterns

Design patterns

A design pattern is a general reusable solution to a commonly occurring problem within a given context.

- Object-oriented design patterns typically show relationships and interactions between classes or objects.
- Their goal is to create a catalog that captures design experience in a form that people can use effectively.

History

- **1977 - Christopher Alexander**
  - Architect who first studied patterns in buildings and communities and developed a "pattern language" for generating them.
  - Published in 1977 a book titled: "*A Pattern Language: Towns, Buildings, Construction*"

Christopher Alexander on patterns...

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

History

- **1987 – Kent Beck and Ward Cunningham**
  - Adapted the idea of Pattern Language to object-oriented programming presenting five patterns used for designing window-based user interfaces in Smalltalk.
  - OOPSLA'87 Conference: "*Using Pattern Languages for Object-Oriented Programs*".
- **1991 – Jim Coplien**
  - Published: "*Advanced C++ Programming Styles and Idioms*"
  - A book with C++ specific and low-level patterns.

K. Beck

W.Cunningham

J. Coplien

History

- **1995 – Gang of Four (GoF)**
  - Published by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, known as the *Gang of Four* or simply GoF.
  - It is the main reference about design patters and is one of the most complete books on the subject.
  - It is not an introductory book, but advanced. The code examples are in C ++ and diagrams in OMT (quite similar to UML).

R. Johnson, E. Gamma, R. Helm, J. Vlissides

Types of patterns according to their conceptual level

- **Architectural Pattern**
  - Expresses a fundamental structural organization schema for software systems.
  - Provides a set of predefined subsystems, specifying their responsibilities and relationships.
  - **Example**: Model-View-Controller
- **Design pattern**
  - Provides a schema for refining the subsystems or components of a software system, or the relationships between them.
  - Solves a general design problem within a particular context.
  - **Example**: Communication between model and view is done through the Observer pattern.
- **Idiom**
  - Describes how to implement particular aspects of components or the relationships between them using the features of a given language.
  - **Example**: The observer pattern is implemented in Java using specialized classes from the API.

Antipatterns

Antipatterns

Represent a commonly occurring solution to a problem, creating decidedly negative consequences.

- They let us recognize problematic solutions and present a detailed plan for reversing them and implementing productive solutions.
- **Example, Antipattern "God Class"**: Occurs where one class monopolizes the processing and other classes primarily encapsulate data. $\Rightarrow$ Not fulfilling the Single Responsibility Principle.

Design patterns classification

- **Creational patterns**
  - They abstract the instantiation process and help make a system independent of how its objects are created, composed and represented
  - **Examples**: Singleton, Factory Method, etc.
- **Structural patterns**
  - Concerned with how classes and objects are composed to form larger structures.
  - **Examples**: Composite, Adapter, etc.
- **Behavioral patterns**
  - Concerned with algorithms and the assignment of responsibilities between objects.
  - Describe not just patterns of objects or classes but also the patterns of communication between them.
  - **Examples**: Strategy, Observer, etc.

Table of Contents

"Favor Immutability" principle

"Favor Immutability" principle

Classes should be immutable unless there is a very good reason to make them mutable.

- **Advantages**:
  - Immutable objects are simple. If you make sure that all constructors establish class invariants, then it is guaranteed that these invariants will always remain true.
  - They can be shared freely. Compare `String` (immutable) with `Date` (mutable).
  - They are inherently thread-safe and do not require synchronization.
  - They make great building blocks for other objects.

Immutable pattern

Immutable pattern

Allows creating immutable classes whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object.

Rules to make a class immutable

1. **Do not provide any methods that modify the object's state**
   - Known as mutators or setters (`set...`).
2. **Ensure that the class cannot be extended**
   - This prevents careless or malicious subclasses from compromising the immutable behavior of the class.
   - This is generally accomplished by making the class `final`.
3. **Make all fields `final`**
   - This clearly expresses your intent in a manner that is enforced by the system (*blank finals* should be initialized in the constructor).
4. **Make all fields private**
   - This prevents clients from obtaining access to mutable objects referred to by fields and modifying these objects directly.
5. **Ensure exclusive access to any mutable components**
   - If your class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain references to these objects (never initialize such a field to a client-provided object reference or return the object reference from an accessor -make defensive copies-, etc.)

Immutable pattern

Class `Point` immutable

```
public final class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public Point move(int x, int y) {
        return new Point(this.x + x, this.y + y);
    }
}
```

Class is defined as final.

Attributes are private and final and are initialized in the constructor.

Getter methods do not return references to mutable objects.

Methods that "simulate" modifying the state of the object but actually return a new instance of that object.

Immutable pattern

- **Drawbacks**
  - They require a separate object for each distinct value.
  - Creating these objects can be costly, especially if they are large.
  - The performance problem is magnified if you perform a multistep operation that generates a new object at every step, eventually discarding all objects except the final result.
- **Solution**
  - Define mutable companion classes that can be used to perform those operations that can be costly in immutable classes.
  - These classes do allow state changes in those operations in which the immutable one only "simulates" altering the state of the object.
  - It also includes methods for creating a mutable object from its immutable equivalent and vice versa.

Immutable pattern

Class `MutablePoint`

```
public class MutablePoint {
    private int x;
    private int y;
    public int getX() { return x; }
    public int getY() { return y; }

    public MutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public MutablePoint move(int x, int y) {
        this.x += x;
        this.y += y;
        return this;
    }
    public MutablePoint(Point p) {
        this(p.getX(), p.getY());
    }
    public Point getImmutable() {
        return new Point(x, y);
} } }
```

The class is not final.

Attributes are private but not final.

The `move` method does modify the state of the object.

We can convert an immutable point into a mutable one (using the constructor) and vice versa (via the `getImmutable` method).

Immutable pattern

Using the mutable class

```
public static void main(String[] args) {
    // Modifying directly on the immutable object.
    Point p1 = new Point(5, 7);
    for (int i = 1; i < 1000; i++) {
        p1 = p1.move(1, 2); // 1000 instances of Point are created.
    }
    System.out.println(p1.getX() + " " + p1.getY());

    // Modifying using the companion mutable class.
    Point p2 = new Point(5, 7);
    MutablePoint mp = new MutablePoint(p2);
    for (int i = 1; i < 1000; i++) {
        mp = mp.move(1, 2); // No new instaces are created.
    }
    p2 = mp.getImmutable();
    System.out.println(p2.getX() + " " + p2.getY());
}
```

Immutable pattern

- **Examples in the Java API**
  - The class `String` is immutable and has `StringBuilder` as its mutable companion class.
  - Wrapper classes like `Integer` or `Double` are immutable. Primitive types (`int`, `double`, etc.) can act as their mutable companions.
  - Classes like `BigDecimal` or `BigInteger` are immutable. There are mutable companion classes, such as `MutableBigInteger`. Interestingly, these classes were not defined as `final`, which is considered a flaw in their immutability.
  - Generally speaking, all classes that act as primitive types should be immutable. A notable exception in the Java API is `Date`, and this is one of the reasons why Java version 8 includes a new API for dates.

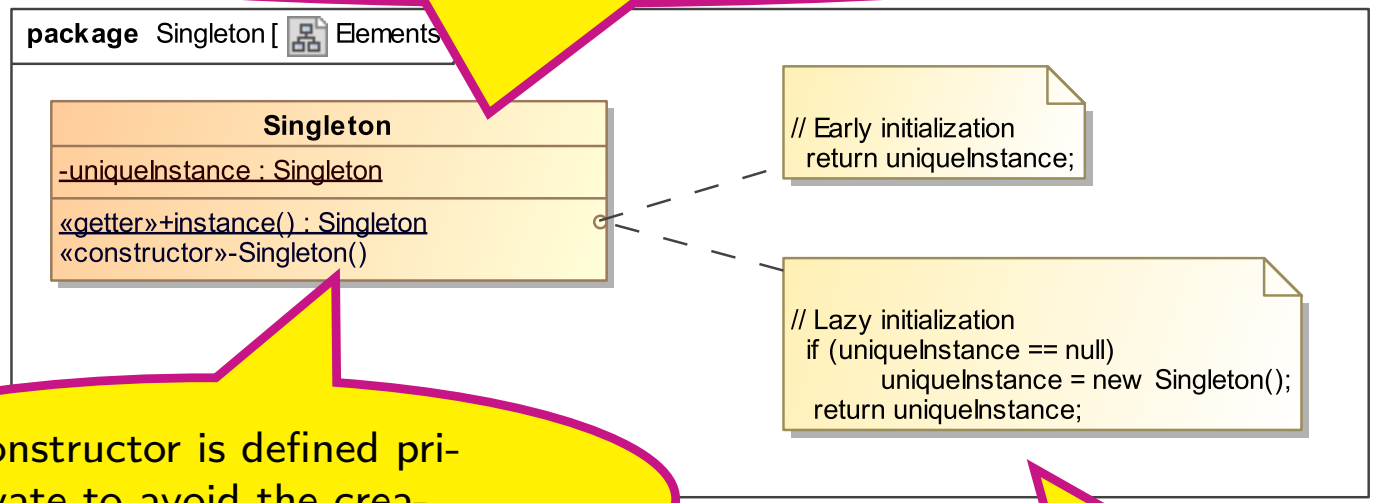
Singleton

Singleton

Ensures that a class has only one instance, and provides a global access point to it.

Singleton pattern elements

Singleton: Defines an Instance operation that lets clients access its unique instance.



Constructor is defined private to avoid the creation of more instances.

A static getter method allows access to the unique instance.

Singleton pattern

Class `Singleton` (early initialization)

```
class Singleton {

    private static final Singleton uniqueInstance = new Singleton();

    private Singleton() { }

    public static Singleton instance() {
        return uniqueInstance;
    }
}
```

`Singleton` instance is created when the class is loaded by the virtual machine.

Code is *thread-safe*, all the threads access the same instance.

It has the drawback of creating an instance that might never be used.

Singleton pattern

Class `SingletonLazy` (lazy initialization)

```
class SingletonLazy {

    private static SingletonLazy uniqueInstance = null;

    private SingletonLazy() { }

    public static SingletonLazy instance() {
        if (uniqueInstance == null) {
            uniqueInstance = new SingletonLazy();
        }
        return uniqueInstance;
    }
}
```

`SingletonLazy` instance is initialized to `null` to avoid creating it if not necessary.

The instance is created the first time `instance()` is called.

It can be problematic if we use multiple threads of execution (we can create a more complex version that includes synchronizations).

Singleton pattern problems

- **Singletons are frequently used to provide a global access point for some service**.
  - This is very similar to a global variable and represents a dependency in our design that is hidden in the code.
- **Singletons carry a state with them that lasts as long as the program lasts**.
  - Persistent state is the enemy of unit testing because each test has to be independent of all the others. Singletons depend on an instance held in a static variable and this is an invitation for test-dependency.
- **We can minimize these problems if the singleton keeps no state**.
  - For example, acting as if they were elements of an enumerated type $\Rightarrow$ see State Pattern.

Table of Contents

The "Encapsulate what varies" principle

The "Encapsulate what varies" principle

Identify the aspects of your application that vary and separate them from what stays the same.

Facing changes

- No matter how well you design an application, over time an application must grow and change or it will *die*.
- Take the parts that vary and encapsulate them, so that later you can modify or extend the parts that vary without affecting those that do not.

Strategy pattern

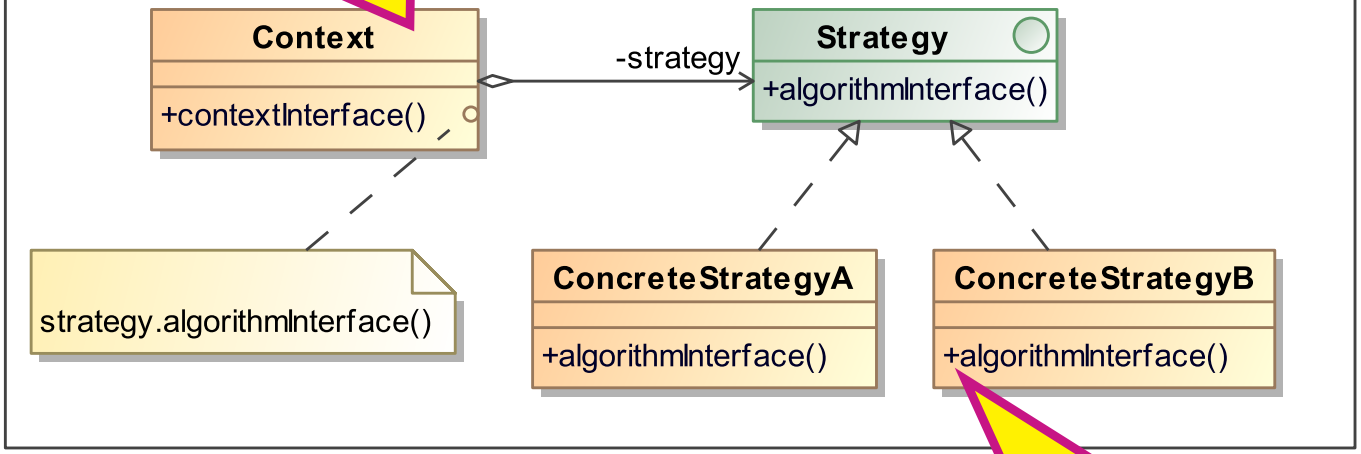
Strategy pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
Strategy lets the algorithm vary independently from its clients.

Strategy pattern elements

Context: Is configured with a `ConcreteStrategy` object and maintains a reference to a Strategy object

Strategy: Declares an interface common to all supported algorithms

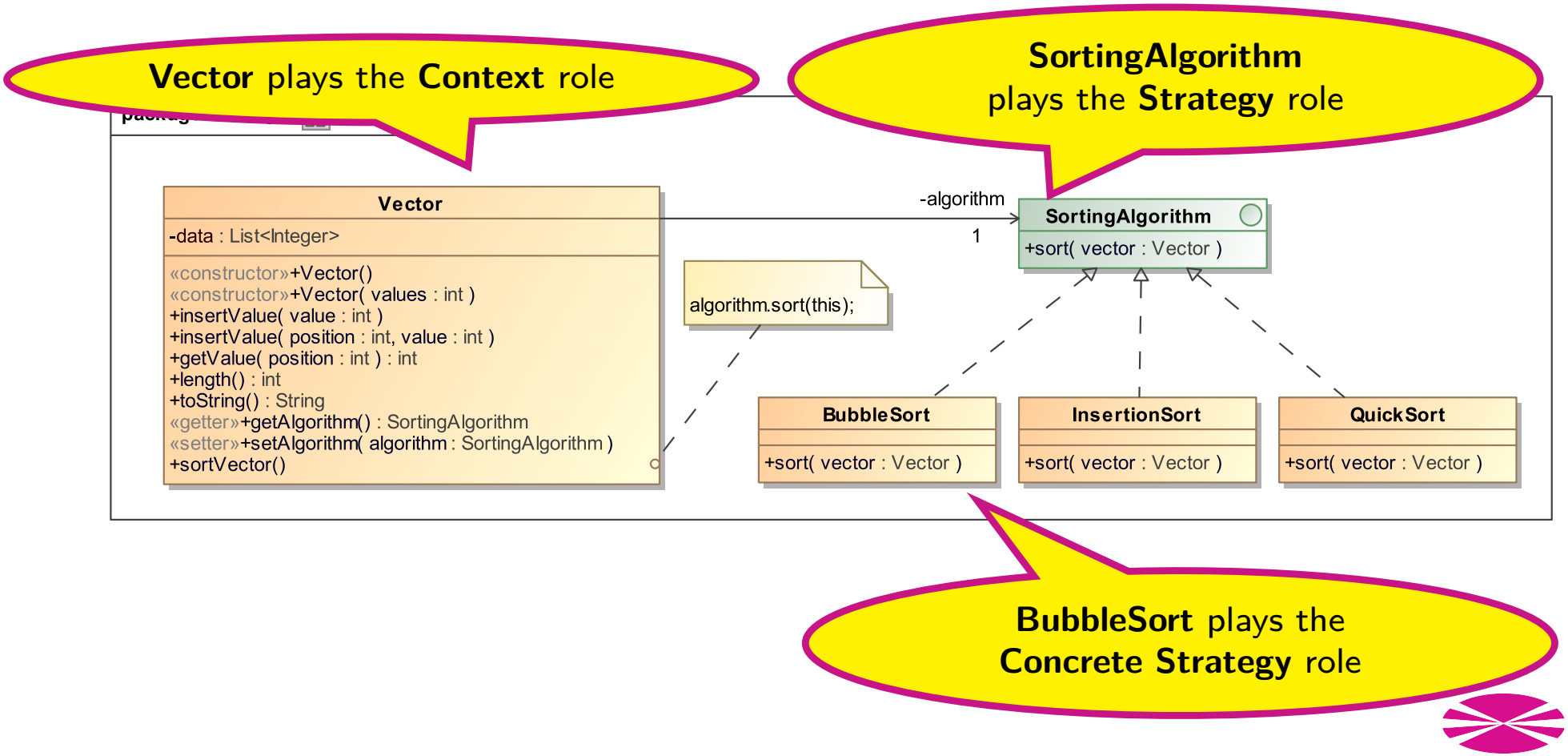


Concrete Stragety: Implements the algorithm using the `Strategy` interface

Strategy pattern example

- The class `Vector` represents a vector of integers.
- Apart from its typical operations we want to include a method for ordering the vector.
- There are many sorting algorithms and, depending on the context, we can choose one or another.
- We apply **Strategy** to separate the part that varies (the sorting algorithm) from the context (the vector) so we can dynamically change that sorting algorithm.

Strategy pattern example



Strategy pattern example

Class `Vector` (vector of integer)

`Vector` uses an `ArrayList` to store data.

We can create it empty or with random data.

We use delegation for the basic methods of `Vector`.

```
public class Vector {
    private List<Integer> data = new ArrayList<>();

    public Vector() { }

    public Vector(int values) {
        java.util.Random random = new java.util.Random();
        for (int i=0; i<values; i++)
            data.add(random.nextInt(100));
    }
    public void insertValue(int value) {
        data.add(value);
    }
    public void insertValue(int position, int value) {
        data.set(position, value);
    }
    public int getValue(int position) { return data.get(position); }
    public int length() { return data.size(); }
    // ...
```

Strategy pattern example

Class `Vector` (vector of integer)

```
    // ...
    private SortingAlgorithm algorithm;

    public SortingAlgorithm getAlgorithm() {
        return algorithm;
    }

    public void setAlgorithm(SortingAlgorithm algorithm) {
        this.algorithm = algorithm;
    }

    public void sortVector() {
        algorithm.sort(this);
    }
}
```

Favor composition over inheritance: `Vector` encapsulates an instance of `SortingAlgorithm`.

`Vector` delegates the ordering on the `algorithm` object, passing itself as a parameter.

Strategy pattern example

Other classes

Dependency inversion: Depend upon abstractions. Do not depend upon concretions.

```
public interface SortingAlgorithm {
    void sort(Vector vector);
}

public class BubbleSort implements SortingAlgorithm {
    @Override
    public void sort(Vector v) {
        int i, j, aux;
        for (i=0; i < v.length()-1; i++)
            for (j = 0; j < v.length()-i-1; j++)
                if (v.getValue(j + 1) < v.getValue(j)) {
                    aux = v.getValue(j + 1);
                    v.insertValue(j+1, v.getValue(j));
                    v.insertValue(j, aux);
                }
    }
}
```

Encapsulate what varies.

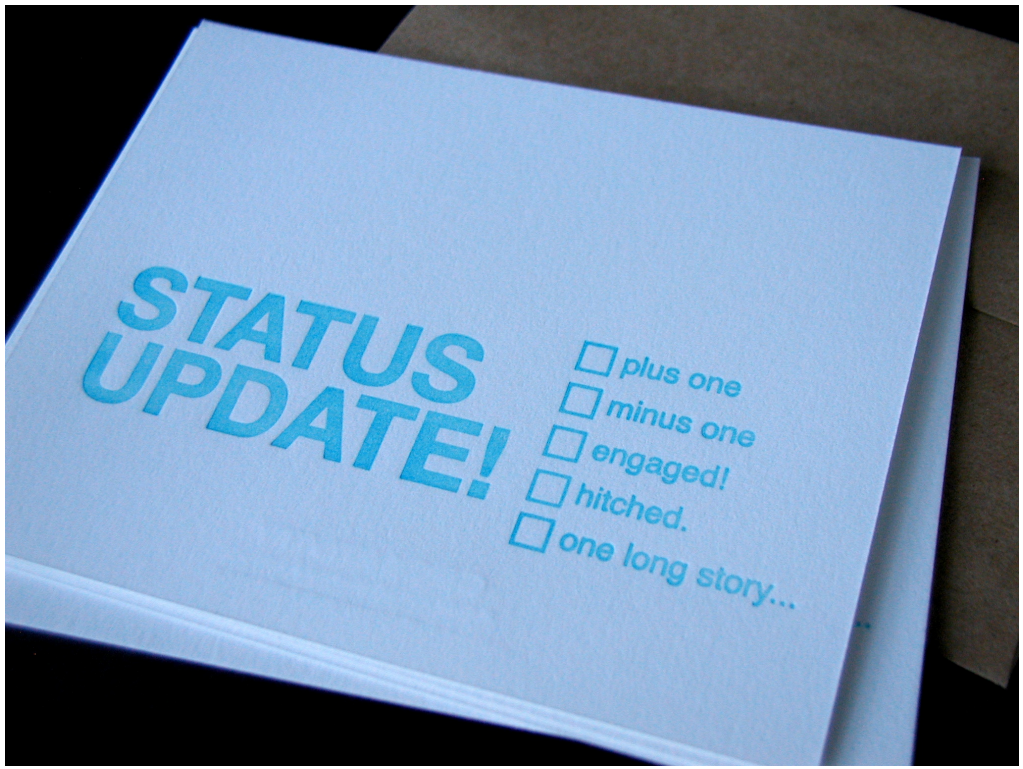
Strategy pattern advantages and disadvantages

- **Advantages**
  - Defines a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality.
  - Alternatively, you could create a subclass of `Context` to give it different behaviors. But this would mix the algorithm's implementation with the Context, making it harder to understand, maintain and extend.
  - Offers an alternative to conditional statements.
- **Disadvantages**
  - The `Strategy` interface is shared by all `ConcreteStrategy` classes whether the algorithms they implement are trivial or complex. Hence, it is likely that some concrete strategies won't use all the information passed to them through this interface.
  - Strategies increase the number of objects in an application. Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share.

State pattern

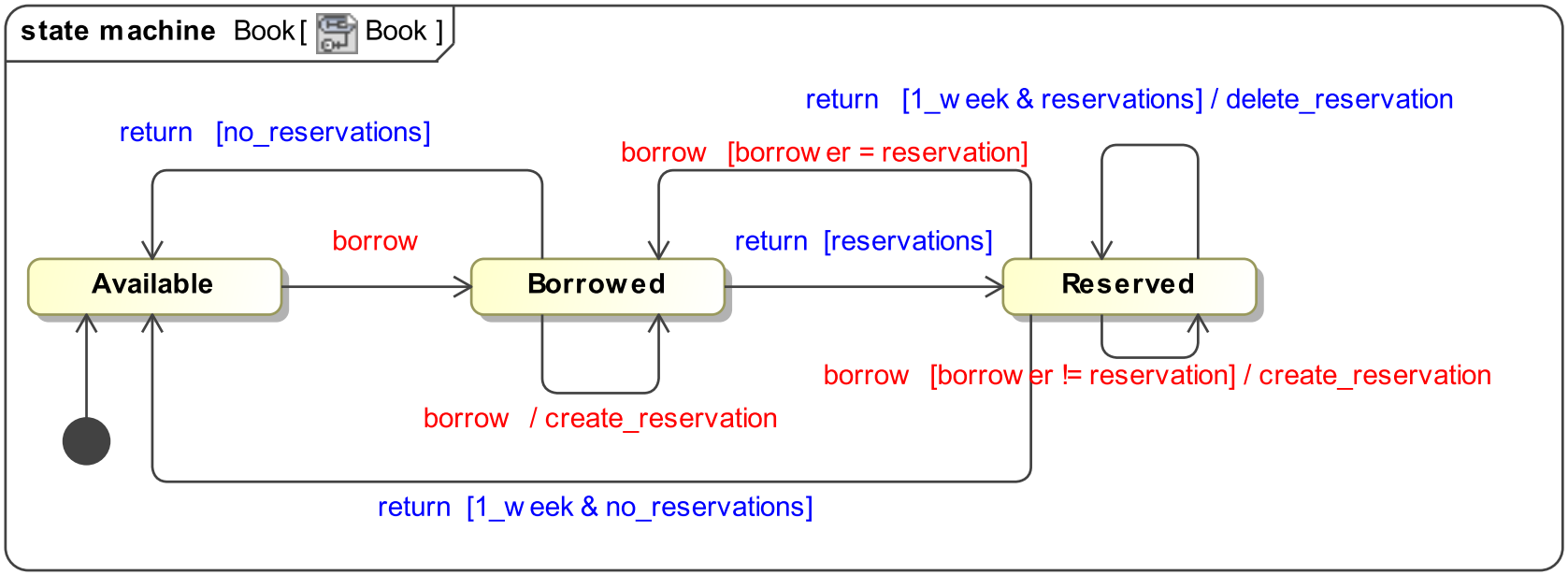
State pattern

Allows an object to alter its behavior when its internal state changes.



Example

- States of a book in a library.



Implementation with conditional statements

Class `Book` (1/3)

```
public class Book {
    enum BookState {AVAILABLE, BORROWED, RESERVED};
    BookState state = AVAILABLE;
    List<String> reservations = new ArrayList<>();
    String currentBorrower;
    boolean oneWeekBorrowed = false;
    // ...
```

States are defined as constants inside `Book`.

`Book` will maintain a reference to the current state and other properties (getters and setters are omitted).

Implementation with conditional statements

Class Book (2/3)

We define a method for each operation on the book.

```
public void borrow(String client) {
    if (state == AVAILABLE) {
        state = BORROWED;
        currentBorrower = client;
        System.out.println("Lending book to " + client);
    } else if (state == BORROWED) {
        reservations.add(client);
        System.out.println("Reservation from " + client);
    } else if (state == BORROWED) {
        if (client.equals(reservations.get(0))) {
            state = BORROWED;
            currentBorrower = client;
            System.out.println("Lending reserved book to " + client);
            reservations.remove(0);
        } else {
            reservations.add(client);
            System.out.println("New reservation from " + cliente);
        }
    }
}
```

We define conditional statements for each state of the book.

Implementation with conditional statements

Class `Book` (3/3)

```
public void return() {
    if (state == BORROWED) {
    ...
    ...
    ...
}   }   }
```

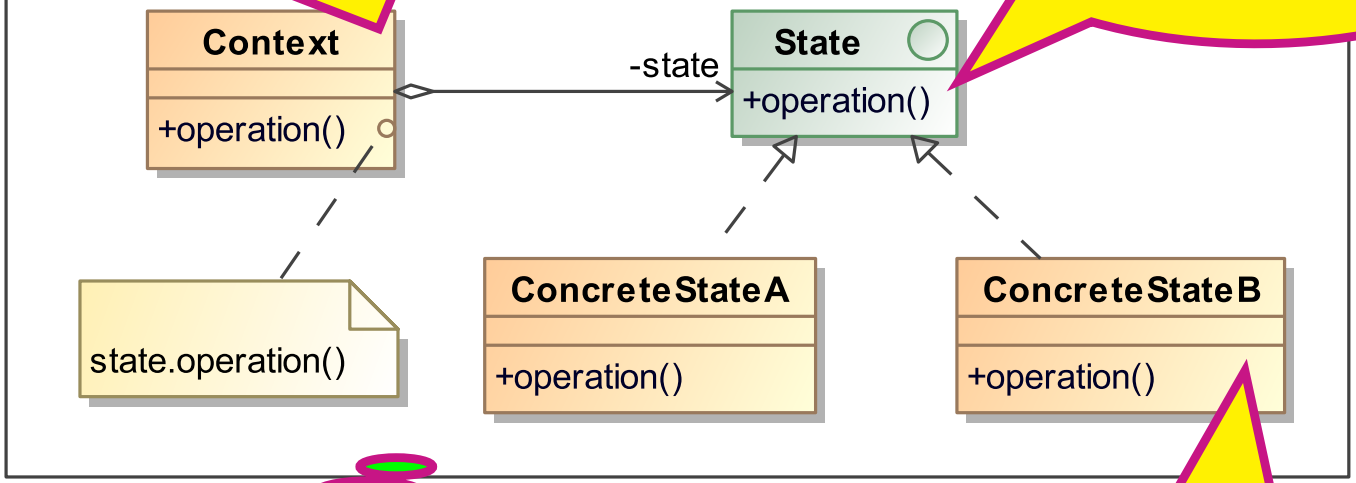
Implementation with conditional statements

- **Advantages**:
  - It is a simple and compact solution (all the behavior is inside one class).
  - It is the preferred solution when the number of states is small, there are only a few methods affected by changes in the state, and it is a stable situation (it is not going to change in a near future).
- **Disadvantages**:
  - *Confused solution*: Too many conditional statements tend to make the code less clear (it is difficult to know how the object behaves in a given state).
  - *Less adaptable to changes*: It is difficult to modify and extend. Every change obliges us to change the `Book` class, not fulfilling the Open-Closed Principle.
  - *Less cohesive*: `Book` has too many intermingled responsibilities (not fulfilling the Single Responsibility Principle).
  - **Solution $\Rightarrow$ State pattern**

State pattern elements

Context: Defines the interface for clients and keeps an instance of a ConcreteState subclass defining the current state.

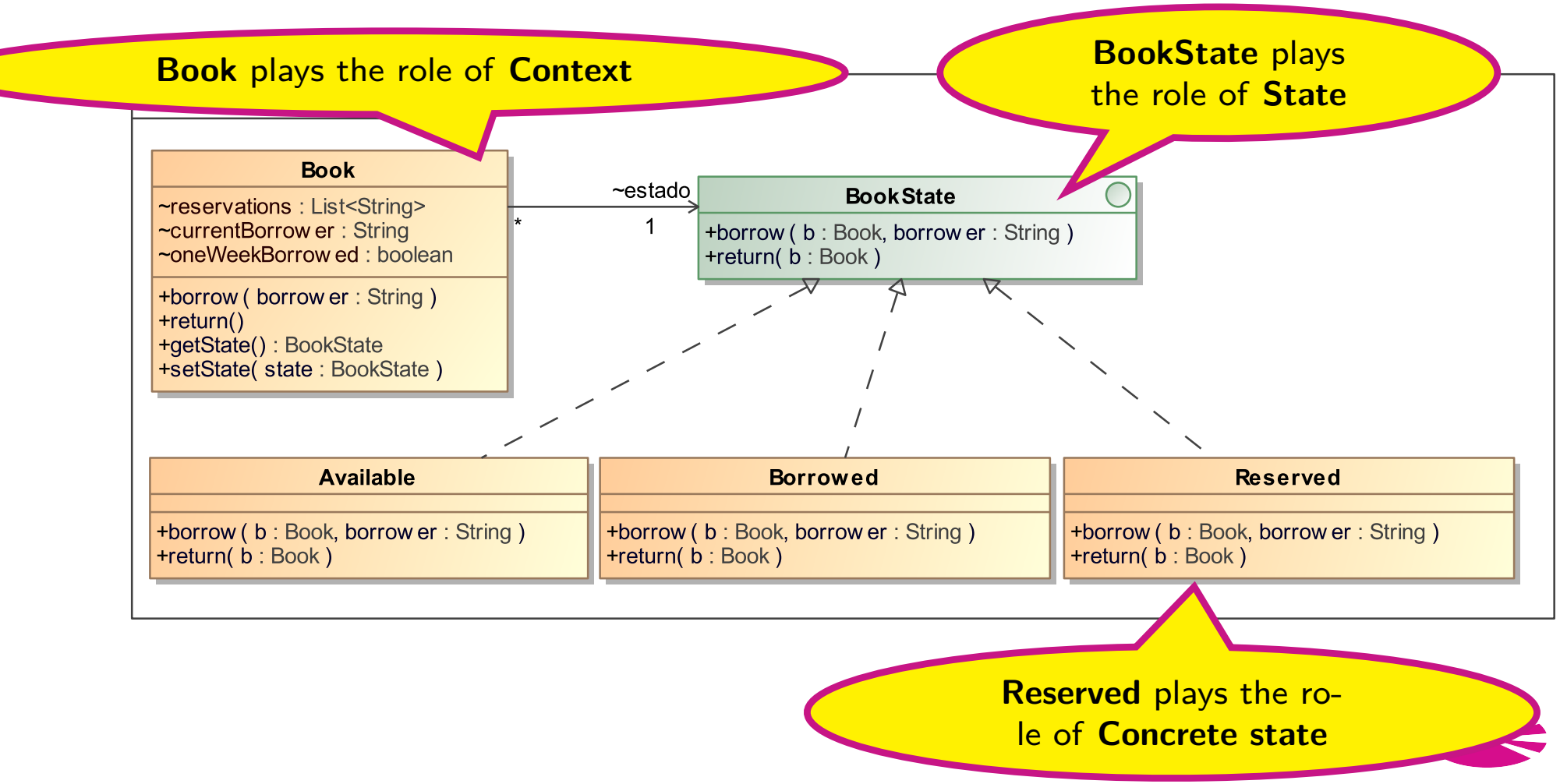
State: Defines an interface for encapsulating the behavior associated with a particular state of the Context.



This reminds me of something...

Concrete State: Each subclass implements a behavior associated with a state of the Context.

State pattern example



State pattern example

Class `Book` (State pattern)

```
public class Book {
    BookState state = Available.instance();
    List<String> reservations = new ArrayList<>();
    String currentBorrower;
    boolean oneWeekBorrowed = false;

    public void borrow(String client) {
        state.borrow(this, cliente);
    }
    public void return() {
        state.return(this);
    }

    public BookState getState() {
        return state;
    }
    public void setState(BookState state) {
        this.state = state;
    }
}
```

`Book` delegates the behavior of `borrow` and `return` on the state object.

`Book` offers methods to read and modify its internal state.

State pattern example

`BookState` is an interface that has all the methods affected by changes in the state.

Interface `BookState`

```
public interface BookState {
    void borrow(Book b, String client);
    void return(Book b);
}
```

`Book` passes itself as parameter so the different states can manipulate it.

State pattern example

States are defined as *singletons* to be shared by the different books.

Class `Available`

```
public class Available implements BookState {
    private static final Available uniqueInstance = new Available();
    private Available() { }
    public static Available instance() { return uniqueInstance; }

    @Override
    public void borrow(Book b, String client) {
        b.setState(Borrowed.instance());
        b.currentBorrower = client;
        System.out.println("Lending book to " + client);
    }

    @Override
    public void return(Book b) {
        // Cannot be returned because it is available
    }
}
```

Each possible state is developed as a class that implements the `BookState` interface.

State pattern example

Class Borrowed

```
public class Borrowed implements BookState {
    private static final Borrowed uniqueInstance = new Borrowed();
    private Borrowed() { }
    public static Borrowed instance() { return uniqueInstance; }

    @Override
    public void borrow(Book b, String client) {
        b.reservations.add(client);
        System.out.println("Reservation from " + client);
    }
    @Override
    public void return(Book b) {
        if (b.reservations.isEmpty()) {
           b.state = Available.instance();
           System.out.println("Book available...");
        } else {
           b.state = Reserved.instance();
           System.out.println("Book reserved by " + b.reservations.get(0));
        }
    }
}
```

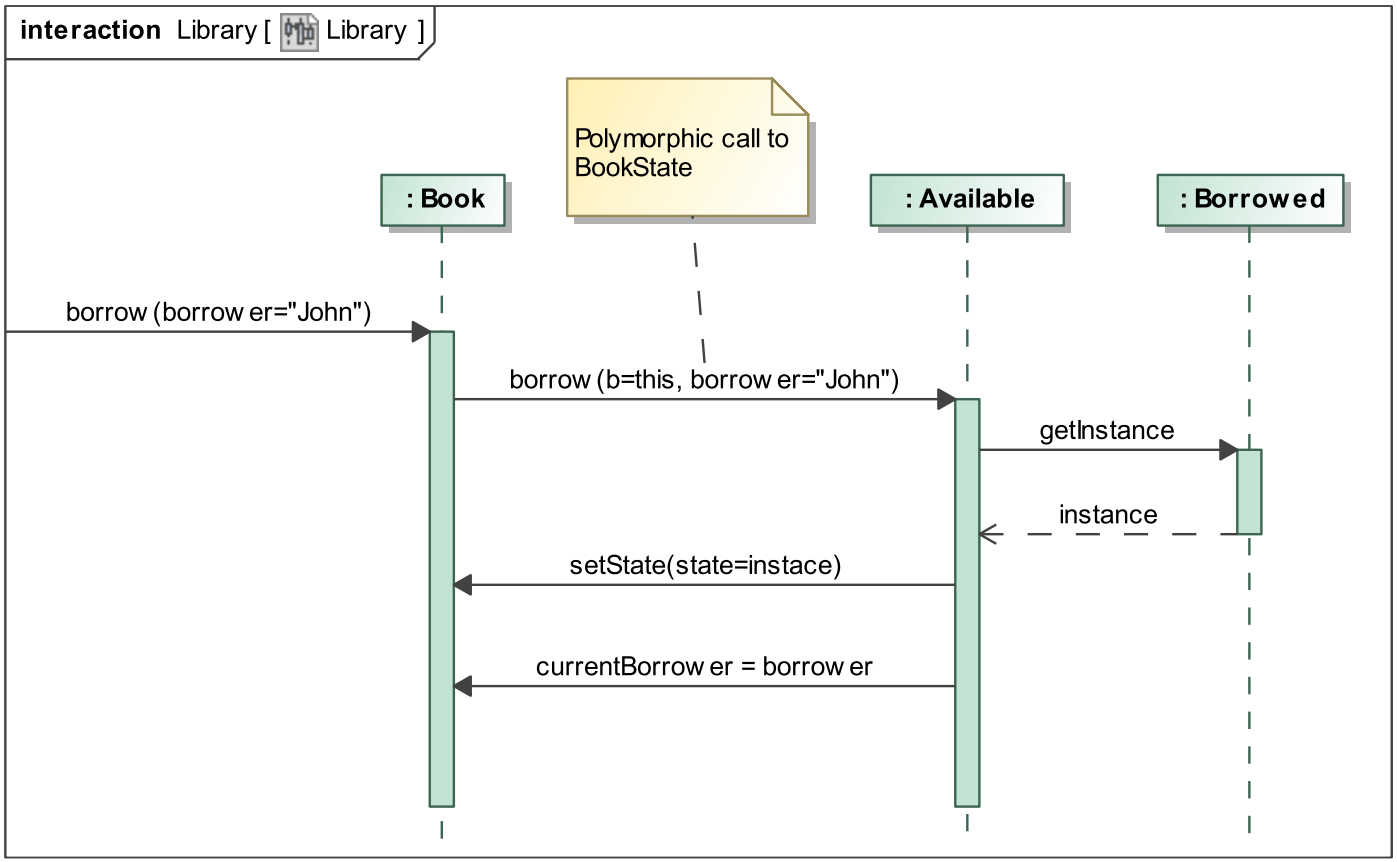
State pattern example

Class `Reserved`

What would the `Reserved` class code be like?

State pattern example

- Sequence diagram when an available book is lent.



State pattern advantages and disadvantages

- **Advantages**:
  - It localizes state-specific behavior and partitions behavior for different states.
  - It is easy to add new states, you only have to add a new class.
  - It makes state transitions explicit.
  - State objects can be shared if they have no instance variables.
- **Disadvantages**:
  - It increases the number of classes and is less compact than a single class.

Table of Contents

Composite pattern

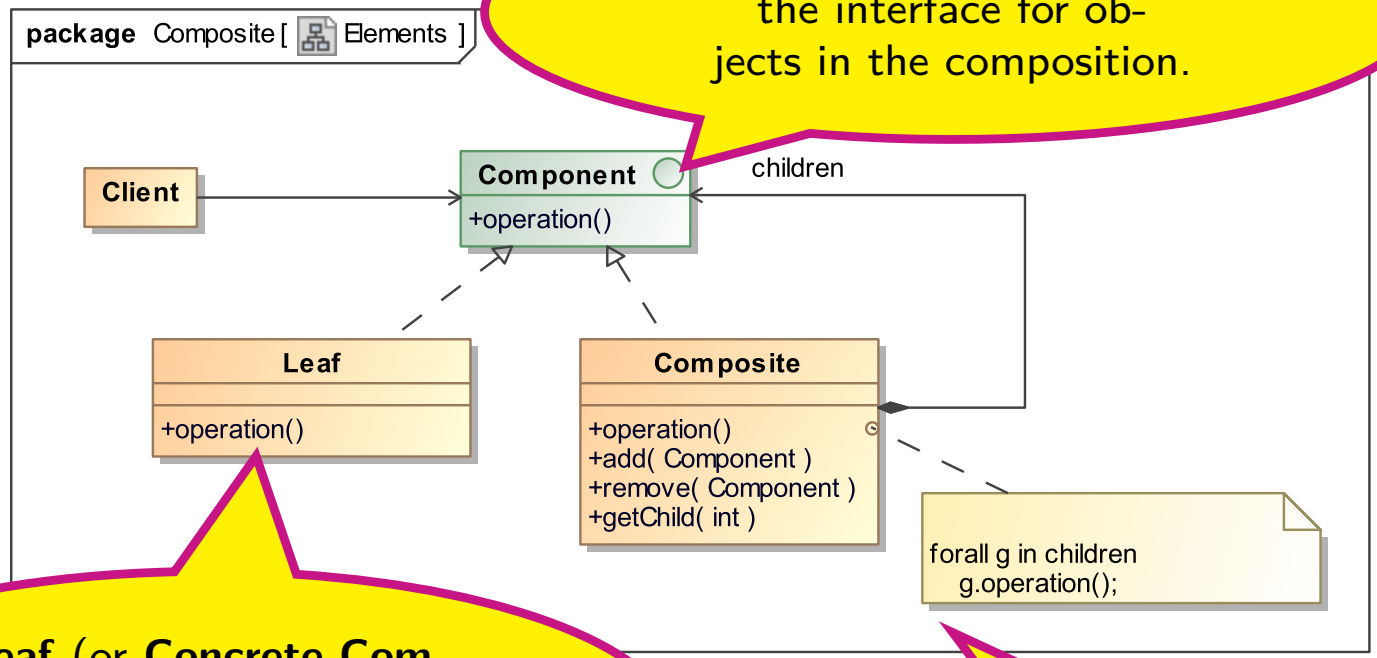
Composite pattern

Structural pattern that composes objects into tree structures to represent part-whole hierarchies.

Composite pattern characteristics

- Composite lets clients treat individual objects and compositions of objects uniformly.
- The key to the Composite pattern is an abstract class that represents both primitives and their containers.
- It is a good example of the joint use of inheritance and composition.

Composite pattern elements

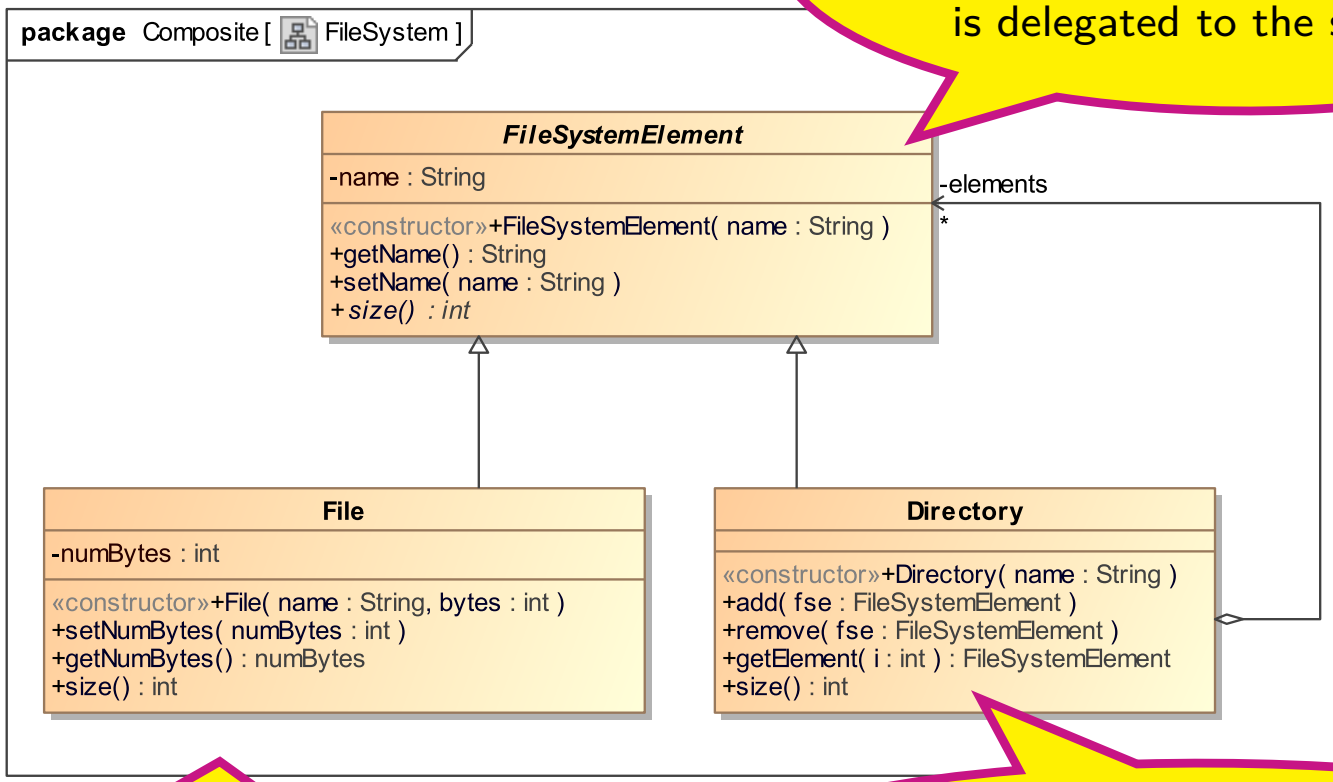


Component: Declares the interface for objects in the composition.

Leaf (or **Concrete Component**): Represents leaf objects in the composition. A leaf has no children.

Composite: Defines behavior for components with children.

Composite pattern example



FileSystemElement plays the role of **Component**, and `size()` is the operation that is delegated to the subclasses.

File plays the role of **Leaf**.

Directory plays the role of **Composite**.

Composite pattern example

Class `FileSystemElement`

```
public abstract class FileSystemElement {
    private String name;

    public FileSystemElement(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public abstract int size();
}
```

Composite pattern example

Class `File`

```
public class File extends FileSystemElement {
    private int numBytes;

    public File(String name, int bytes) {
        super(name);
        numBytes = bytes;
    }

    public void setNumBytes(int numBytes) { this.numBytes = numBytes; }
    public int getNumBytes() { return numBytes; }

    @Override
    public int size() {
        return numBytes;
    }
}
```

Calculating the size of a file consists in returning its number of bytes.

Composite pattern example

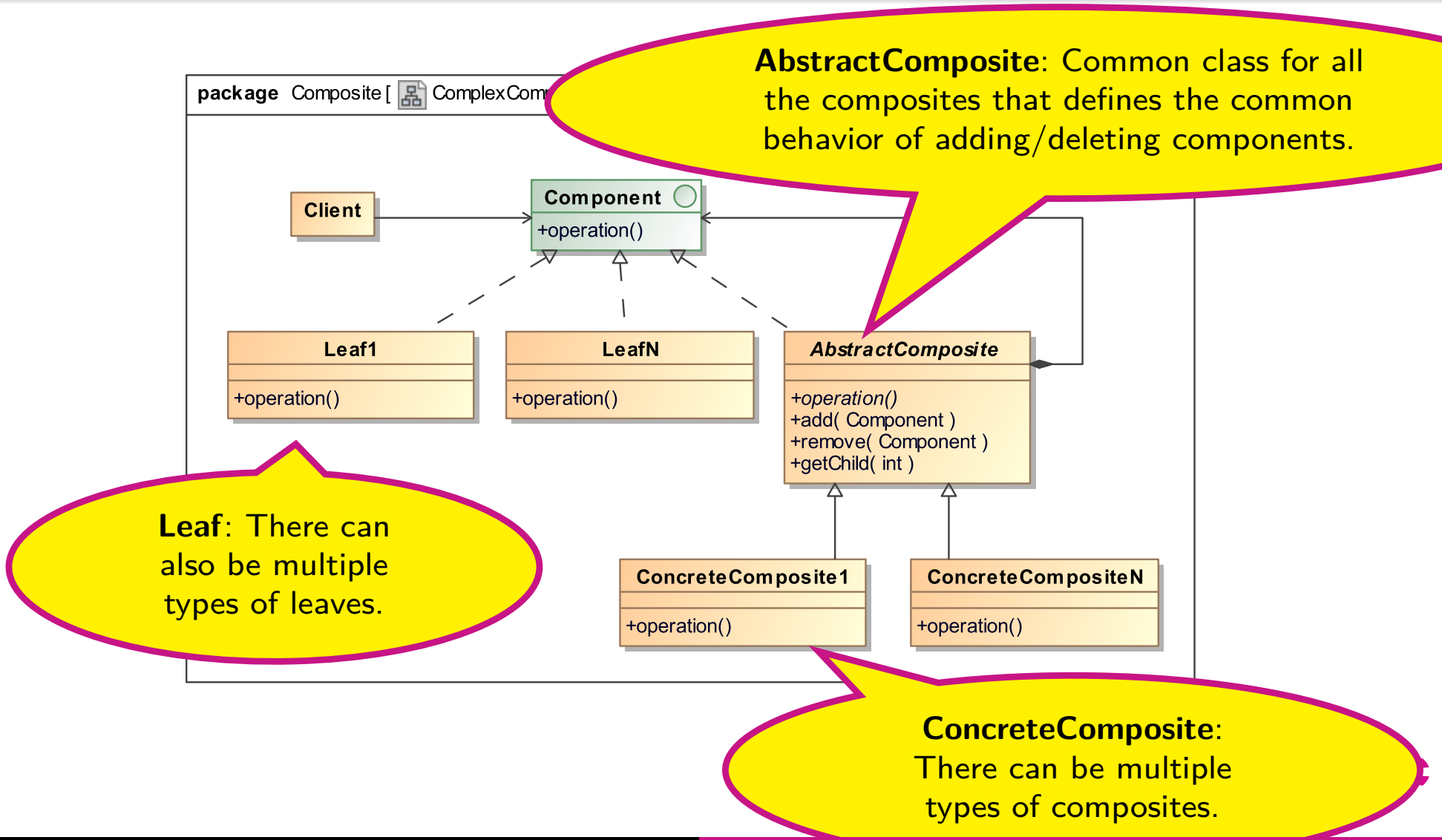
Class `Directory`

```
public class Directory extends FileSystemElement {
    private List<FileSystemElement> elements = new ArrayList<>();

    public Directory(String name) { super(name); }
    public void add(FileSystemElement fse) { elements.add(fse); }
    public void remove(FileSystemElement fse) { elements.remove(fse); }
    public FileSystemElement getElement(int i){ return elements.get(i);}

    @Override
    public int size() {
        int sum = 0;
        for(FileSystemElement fse : elements)
            sum += fse.size();
        return sum;
    }
}
```

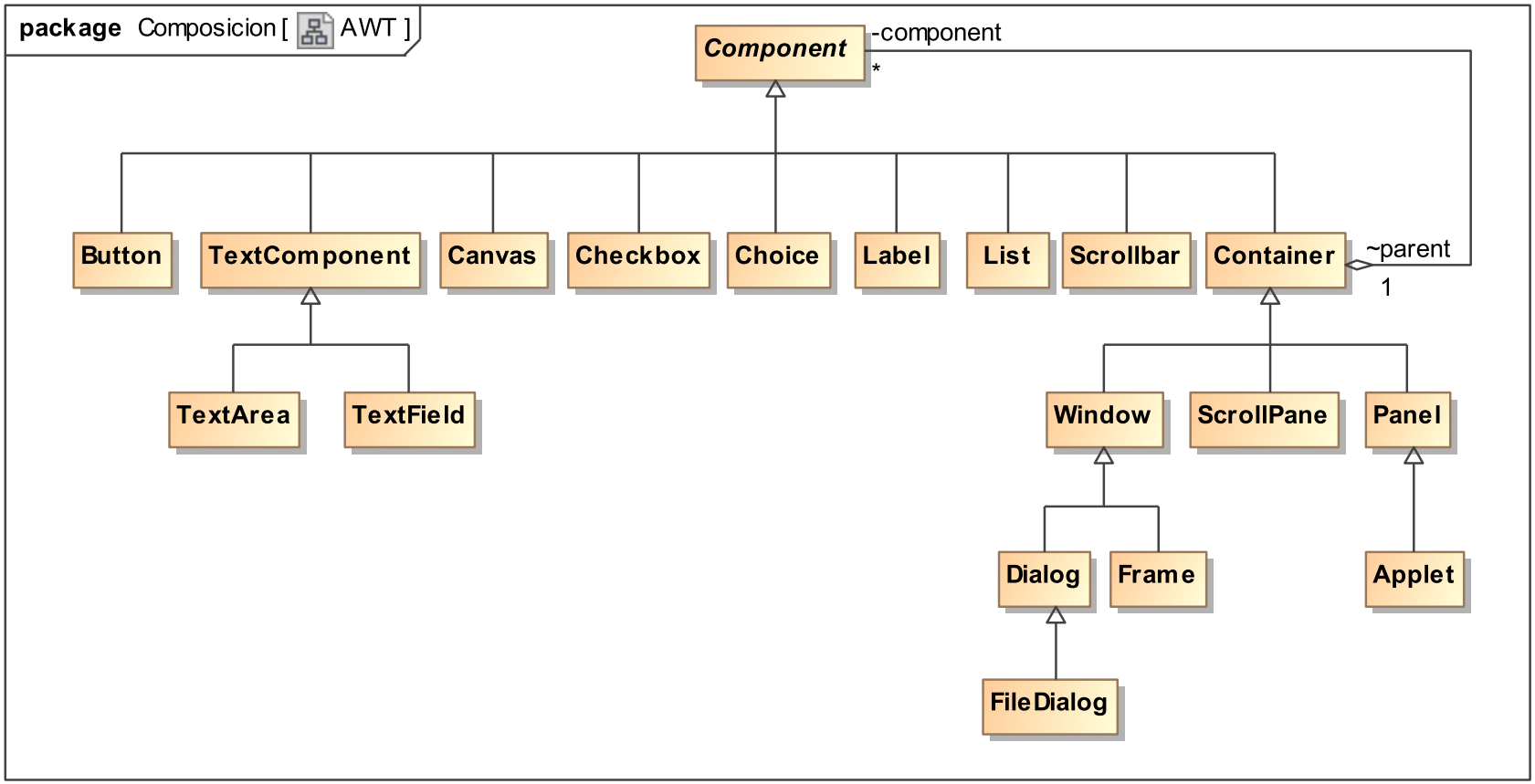
`Directory` includes methods for inserting and deleting new elements.

Calculating the size of a directory consists in adding the size of its elements (if these are directories the operation is recursive).

Complex Composite pattern example



Complex Composite pattern example



Composite pattern

- **Advantages**:
  - Clients will treat all objects in the composite structure uniformly, ignoring the distinction between compositions of objects and individual objects.
  - Makes it easier to add new kinds of components.
- **Disadvantages**:
  - Makes it harder to restrict the components of a composite. You cannot rely on the typing system to enforce those constraints for you. You'll have to use runtime checks instead.

Composite pattern

- **Which classes should declare the Add and Remove operations for managing children in the Composite class hierarchy?**
- The decision involves a trade-off between safety and transparency.

Defining child management in the Composite class

- **safety**: Any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like Java.
- **loss of transparency**: because leaves and composites have different interfaces.

Defining the child management interface at the root of the class hierarchy

- **transparency**: because you can treat all components uniformly.
- **costs in safety**: because clients may try to do meaningless things like adding and removing objects from leaves.

Iterator pattern

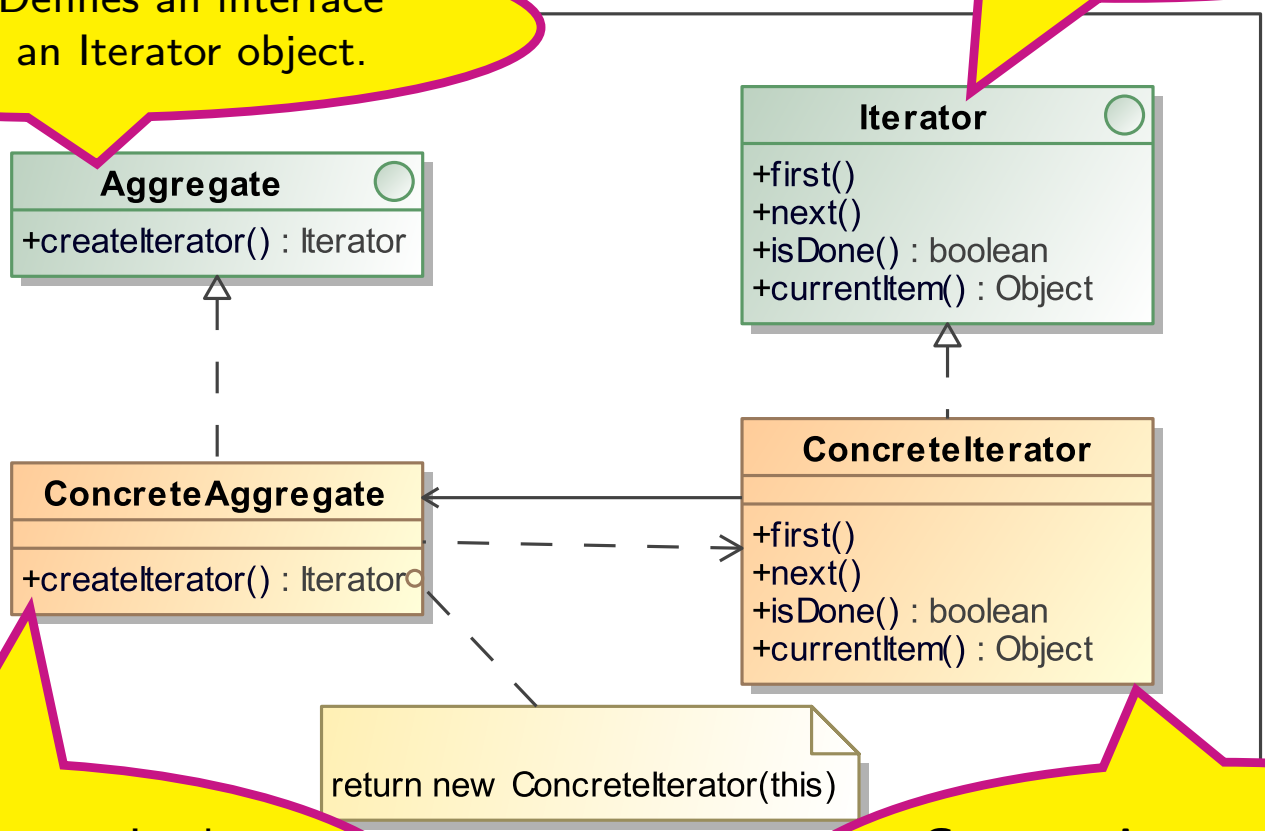
Iterator pattern

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator pattern elements

Iterator: Defines an interface for accessing and traversing elements.

Aggregate: Defines an interface for creating an Iterator object.

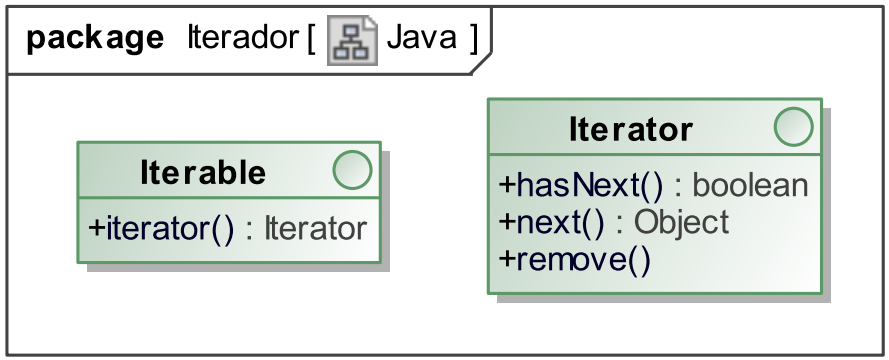


Concrete Aggregate: Implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

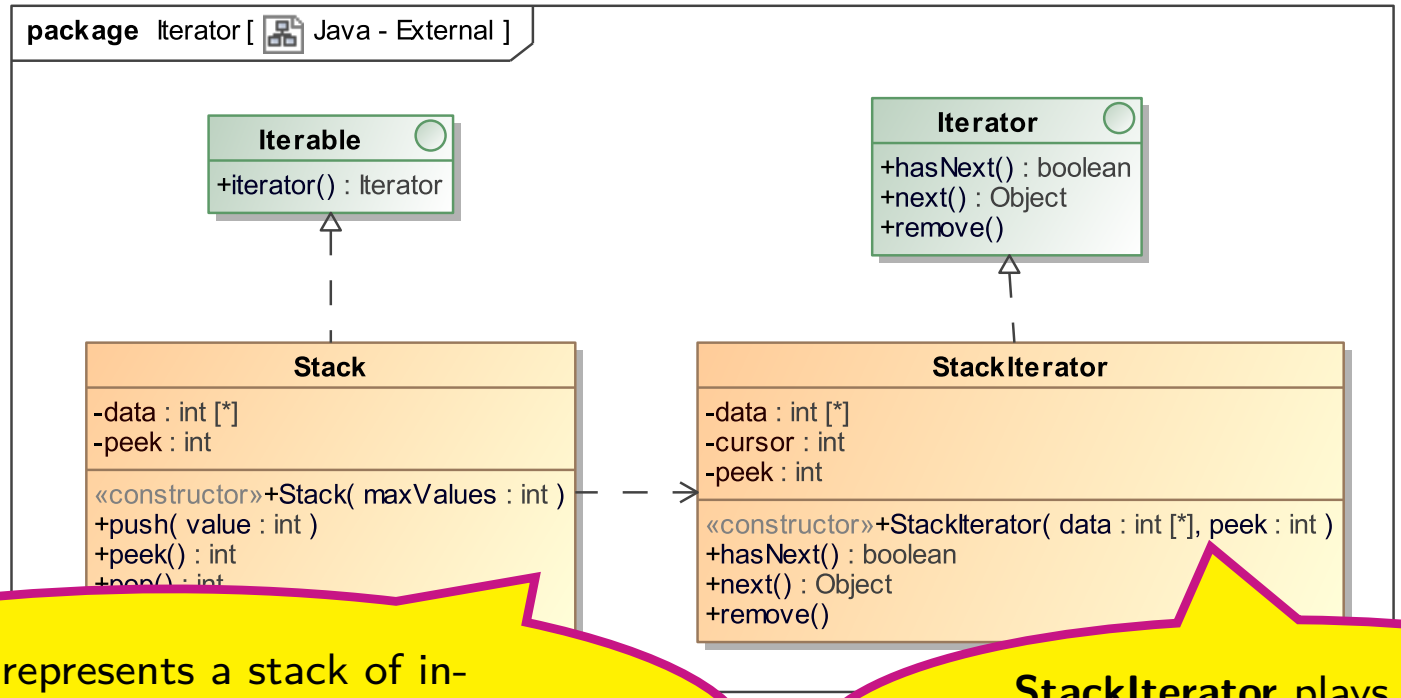
Concrete Iterator: Implements the Iterator interface and keeps track of the current position in the traversal of the aggregate.

Iterator pattern example

- **Iterator Pattern in Java**
  - Interface `Iterable` plays the role of `Aggregate` and defines a method `iterator()` that returns an instance of the iterator.
  - Interface `Iterator` plays the role of iterator and defines three methods:
    - `hasNext(): boolean`: Returns true if the iteration has more elements (equivalent to `isDone()`.)
    - `next(): Object`: Returns the next element in the iteration (it is a combination of `first()`, `next()` and `currentItem()`)
    - `remove()`: Removes from the collection the last element returned by this iterator (since Java 8, it has a default implementation that throws an exception, so it is not mandatory to overwrite it).



Iterator pattern example



Stack represents a stack of integers and plays the role of **Concrete Aggregate**, the collection that we want to iterate.

StackIterator plays the role of **Concrete Iterator** traversing the stack sequentially.

Iterator pattern example

Class `Stack`

```
public class Stack implements Iterable<Integer> {
    private int[] data;
    private int peek = -1;

    public Stack(int maxValues) { data = new int[maxValues]; }
    public void push(int value) { data[++peek] = value; }
    public int peek() { return data[peek]; }
    public int pop() { return data[peek--]; }

    @Override
    public Iterator<Integer> iterator() {
        return new StackIterator(data, peek);
    }
}
```

We implement a simple stack without worrying about possible exceptions (removing from an empty stack, etc.).

The usual behavior is to pass the current object as a parameter to the iterator constructor. Since the interface of the stack only allows access to the top element we must pass its internal structure.

Iterator pattern example

Class `StackIterator`

```
class StackIterator implements Iterator<Integer> {
    private final int[] data;
    private int cursor = 0;
    private int peek;

    public StackIterator(int[] data, int peek) {
        this.data = data;
        this.peek = peek;
    }

    @Override
    public boolean hasNext() { return cursor <= peek; }

    @Override
    public Integer next() { return data[cursor++]; }

    @Override
    public void remove() { throw new UnsupportedOperationException(); }
}
```

The iterator receives the internal structure of the stack...

...and traverses it using the methods of `Iterator`.

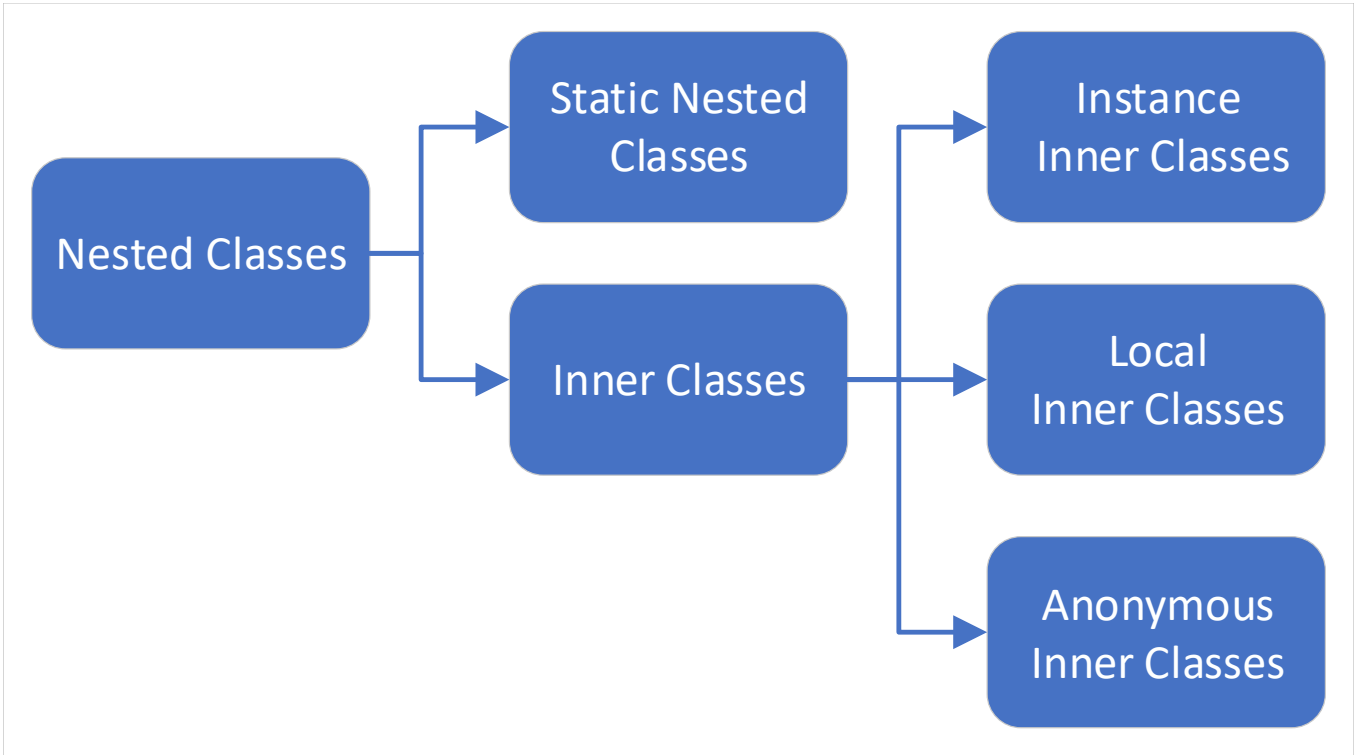
Iterators as inner classes

- The fact that the stack iterator is a distinct class from the stack is an advantage: the responsibilities are separated more clearly, we can have several iterators traversing the structure simultaneously, etc.
- The problem is that the interface of stack does not allow a sequential traversal, so we must pass its internal structure as a parameter to the iterator.
- **Solution** $\Rightarrow$ Define iterator as a inner class of `Stack`
  - An instance of an inner class is associated with an instance of its container class.
  - You will also have access to all elements of the container class (including the private ones).

Nested classes

Nested classes

Classes defined within another class.



Categories of nested classes

- **Static nested classes**: Nested classes that are declared static.
- **Inner classes**: Non-static nested classes.
  - **[Instance] inner classes**: Associated with an instance of its enclosing class.
  - **Local inner classes**: Like instance inner classes but declared inside a block (e.g. a method body, a for loop, etc.) and with access to local variables of that block (if they are *effectively* final).
  - **Anonymous inner classes**: Like local inner classes except that they do not have a name. You declare and instantiate a class at the same time $\Rightarrow$ Lambda Expressions.

Static nested classes

Static nested classes

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}

// Static nested classes are accessed using the enclosing class name.

OuterClass.StaticNestedClass nestedObject
    = new OuterClass.StaticNestedClass();
```

- **Example**: Nested enum types are implicitly static nested classes.

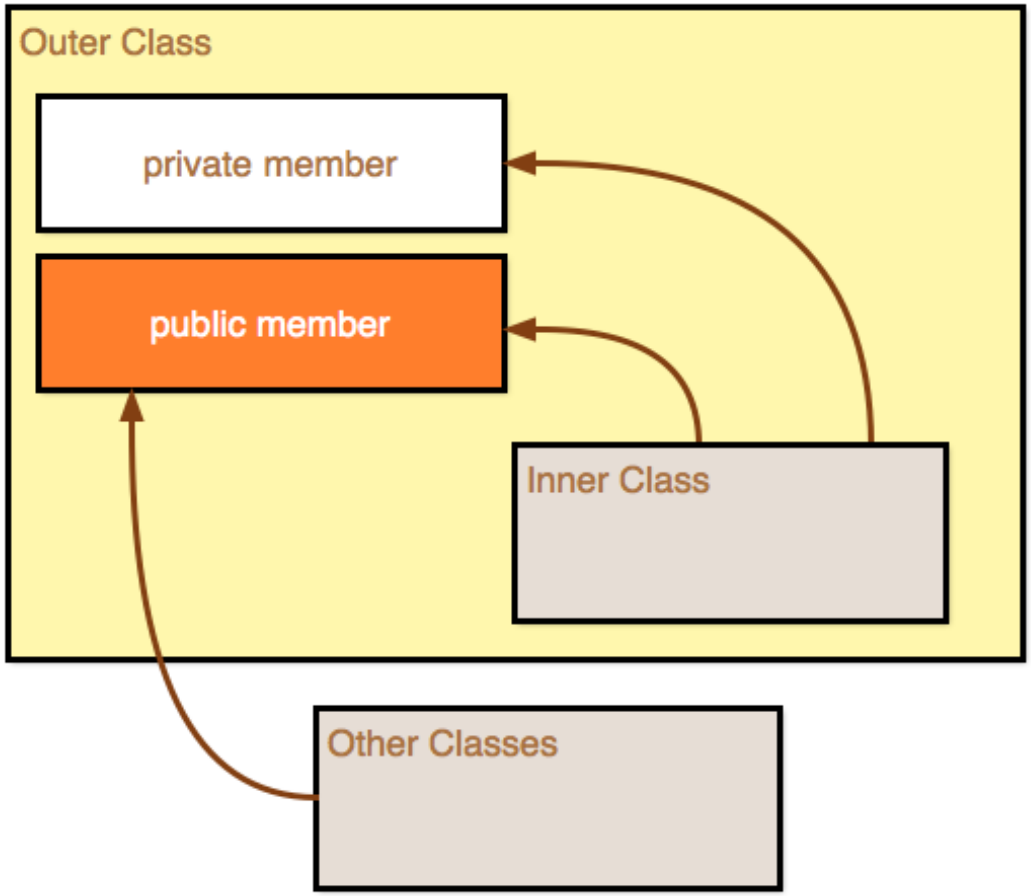
Inner classes

Inner classes

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}

// To instantiate an inner class, you must first instantiate the outer
// class and then create the inner object within the outer object.

OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Inner classes

- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.



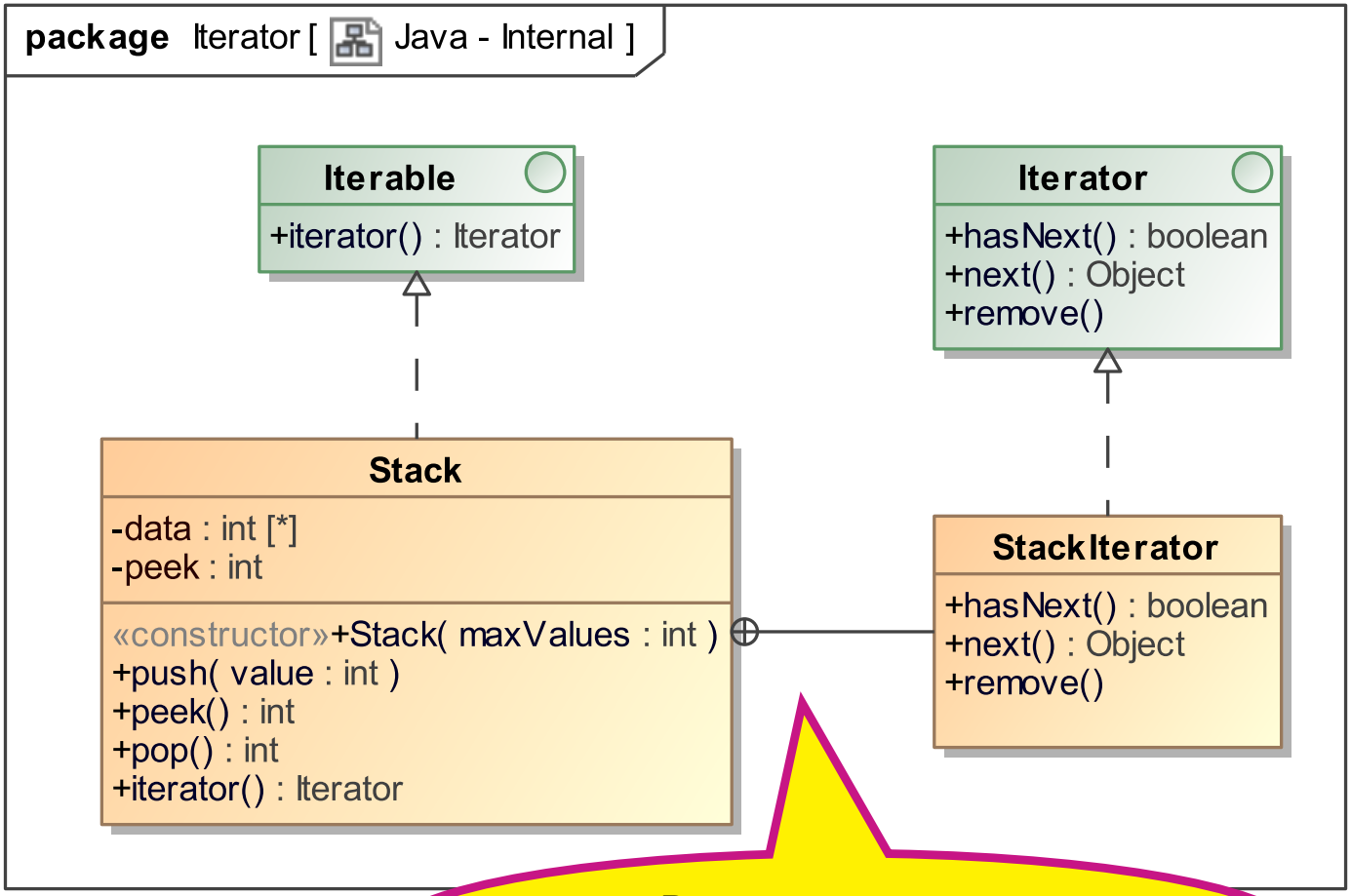
Iterators as inner classes

Class `StackIterator` as inner class of `Stack`

```
public class Stack implements Iterable<Integer> {
    private int[] data;
    private int peek = -1;
    public Stack(int maxValues) { data = new int[maxValues]; }
    public void push(int value) { data[++peek] = value; }
    public int peek() { return data[peek]; }
    public int pop() { return data[peek--]; }
    @Override
    public Iterator iterator()
    { return new StackIterator(); }
    class StackIterator implements Iterator<Integer> {
        private int cursor = 0;
        @Override
        public boolean hasNext() { return cursor <= peek; }
        @Override
        public Integer next() { return data[cursor++]; }
        @Override
        public void remove() { throw new UnsupportedOperationException();}
    }
}
```

`StackIterator` is an inner class of `Stack` and can access its private elements

Iterators as inner classes



Iterators as inner classes

- Java bases the functioning of the `for-each` loop on the `Iterable` and `Iterator` classes.
- Any class (such as `Stack`) that implements `Iterable` can be traversed using a `for-each` loop.

`for-each` loop and iterators

```
public static void main(String[] args) {
    Stack p = new Stack(10);
    p.push(2);
    p.push(4);
    p.push(6);
    p.push(8);
    for (Integer i : p) {
        System.out.print(i + " ");
    } // Prints 2 4 6 8
}
```

Iterator pattern

- **Advantages**
  - It is possible to access an aggregate object's contents without exposing its internal representation.
  - It may support multiple traversals of aggregate objects.
  - It provides a uniform interface for traversing different aggregate structures.
- **Complications**
  - What happens if the collection is modified during the iteration?
  - There are two different strategies here: *fail-fast* and *fail-safe*, each with advantages and disadvantages.

Iterator pattern

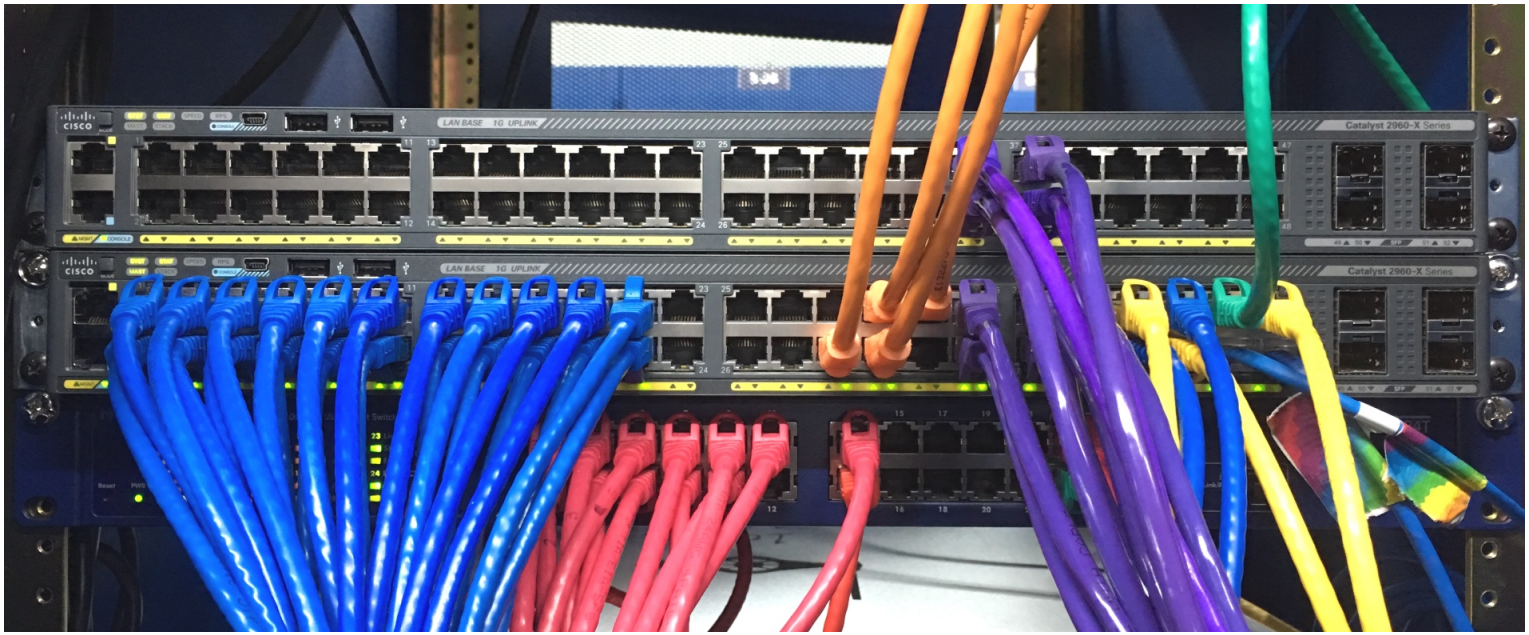
- **Fail-fast iterators**
  - Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation.
  - Fail-fast iterators **check if the underlying collection gets externaly modified**, if so they aborts the entire operation throwing `ConcurrentModificationException`.
  - Java API iterators are fail-fast.
- **Fail-safe iterators**
  - Fail-Safe systems don't abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible.
  - Those iterators **create a clone** of the actual collection and iterate over it. If any modification happens after the iterator is created, the copy still remains untouched.
  - The cost of the safe iteration is the **overhead of creating a copy** of the collection and that the iterator **isn't guaranteed to return updated data** from the collection.

Table of Contents

"Loose coupling" principle

"Loose coupling" principle

Strive for loosely coupled designs between interacting objects.



"Loose coupling" principle

Coupling

It is a measure of how closely connected two classes or modules are.

- **Advantages**
  - A loosely coupled code is more robust and adaptable because changes do not propagate to other classes of the system, only to a few classes nearby.
- **Coupling and cohesion**
  - The objective is to create cohesive objects (focused on a single responsibility) with low coupling (dependent on the lowest number of classes).

"Loose coupling" principle

- **How to reduce coupling**:
  - Working with abstractions and not with concretions, so we are less dependent on a given implementation $\Rightarrow$ **Dependency Inversion Principle**, **Design Patterns** in general.
  - Connecting our classes with the lowest number of other classes (some classes play the role of intermediary between others) $\Rightarrow$ **Least Knowledge Principle**, **Tell, Don't Ask**, **Façade Pattern**.
- **Coupling and unstable classes**
  - High coupling is a problem when one of the classes is unstable and changes frequently because it causes changes in the dependent elements.
  - If the coupled classes are stable, uncoupling them only adds complexity to the system.

Observer pattern

Observer pattern

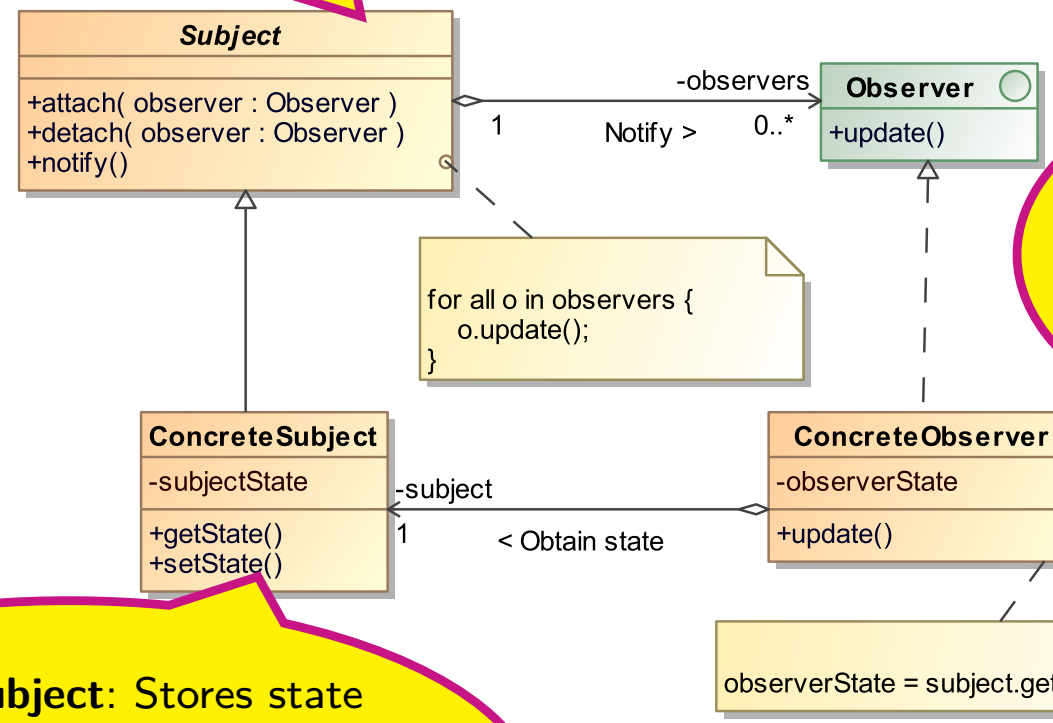
Define a one-to-many dependency between objects so that when one object changes its state, all its dependent objects are notified and updated automatically.

Observer pattern characteristics

- **Low coupling between Subject and Observer**. All a subject knows is that it has a list of observers, each conforming to a given interface.
- Lets you **add observers without modifying the subject** or other observers.
- Lets you **vary subjects and observers independently**.

Subject: Abstract class that provides an interface for attaching and detaching Observer objects.

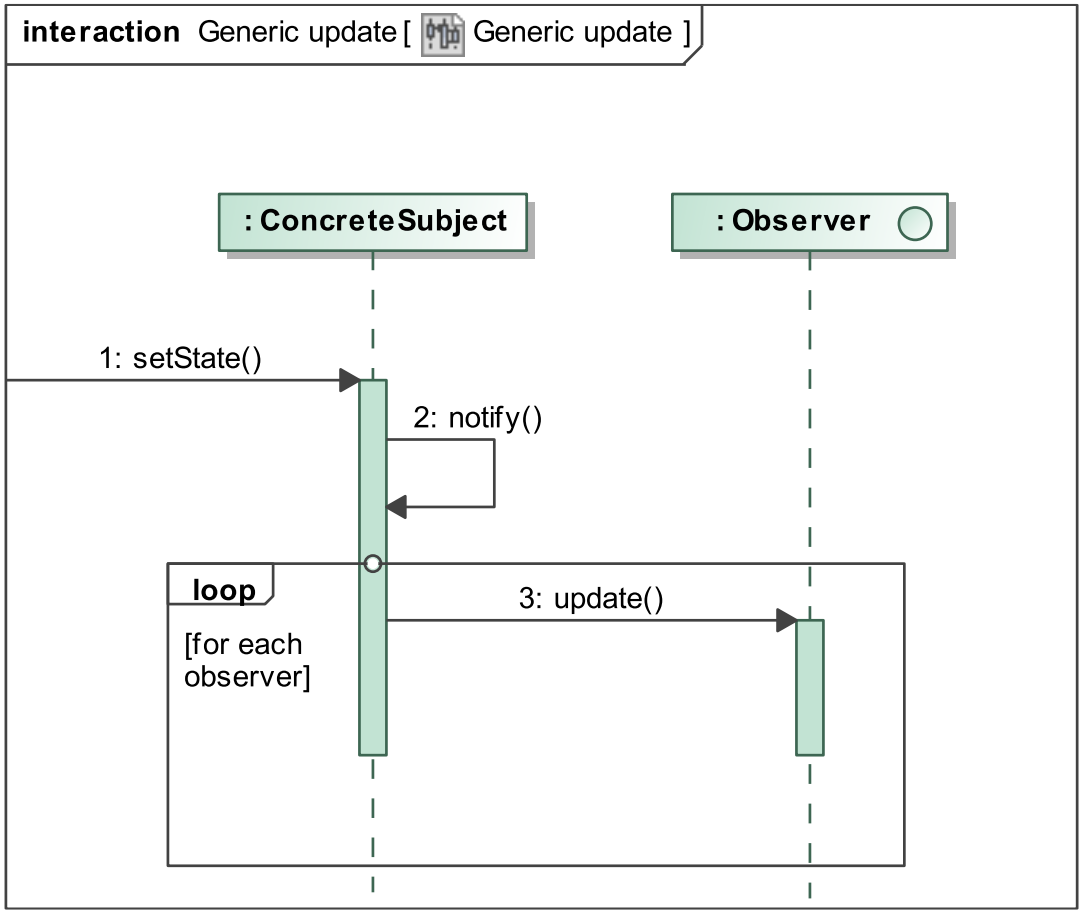
Observer: Defines an updating interface for objects that should be notified of changes in a subject.



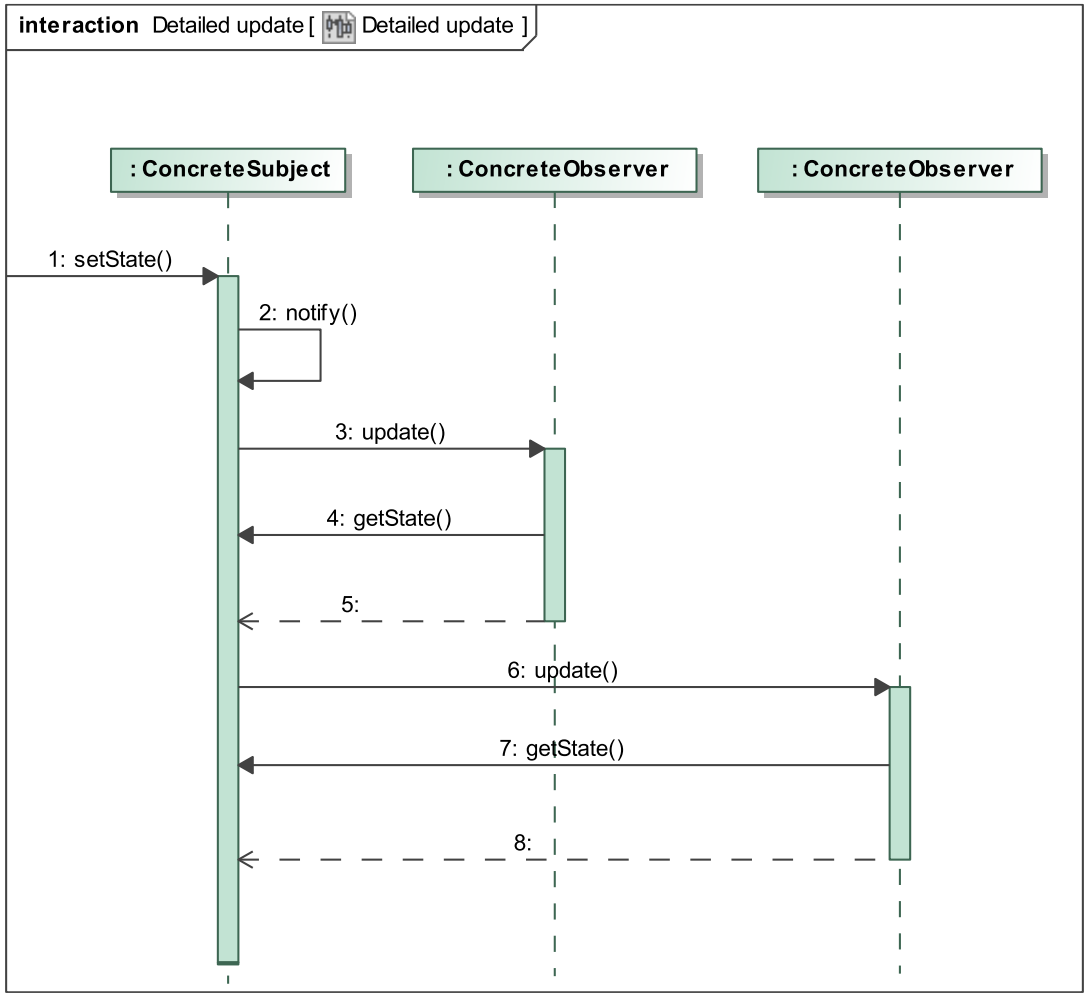
ConcreteObserver: Maintains a reference to a ConcreteSubject object and implements the updating interface.

ConcreteSubject: Stores state of interest to ConcreteObserver objects and sends a notification to its observers when its state changes.

Observer pattern functioning



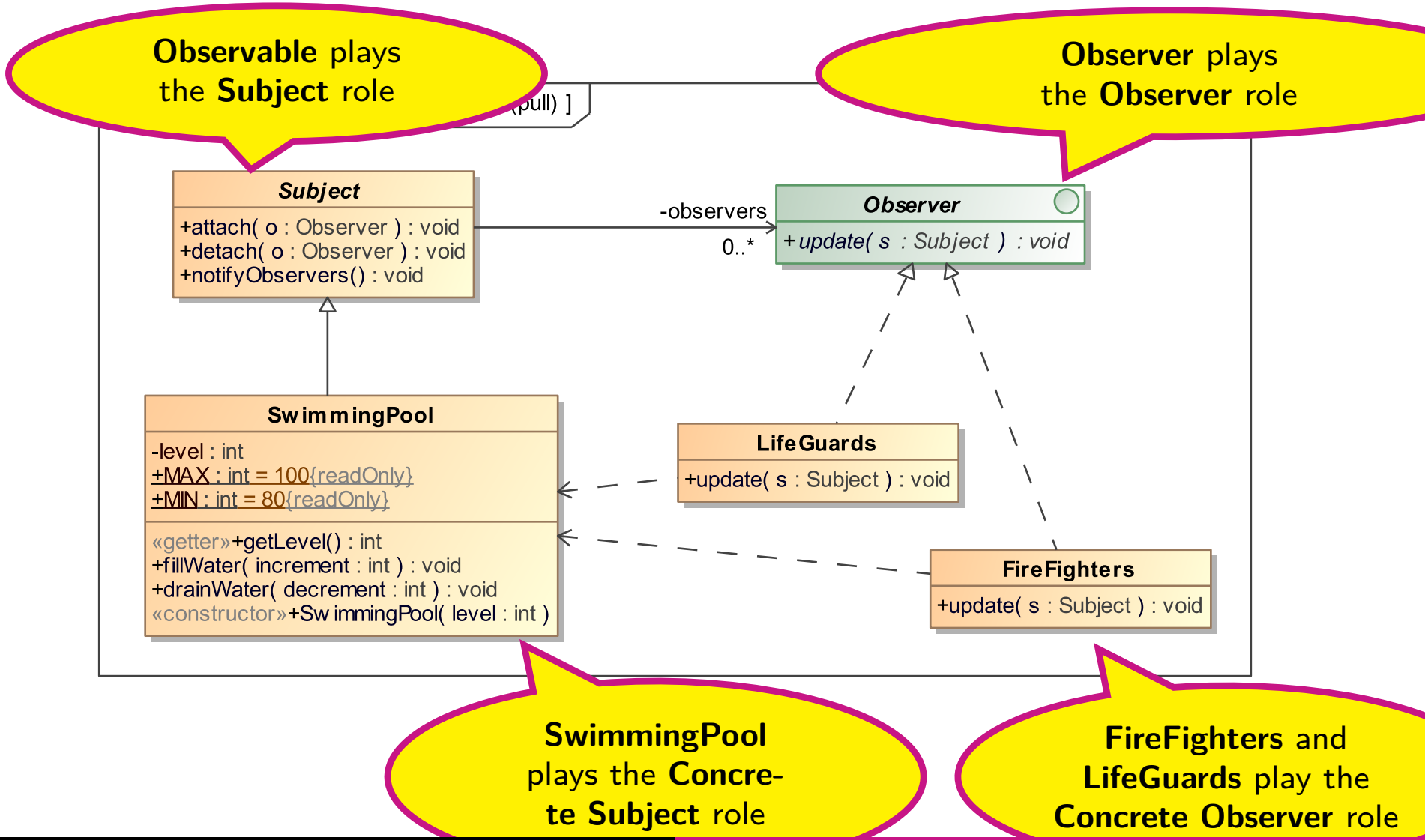
Observer pattern functioning



Observer pattern in Java

- To implement the Observer pattern in Java there is the **`Observable` class that plays the `Subject` role and the `Observer` class that plays the `Observer` role**.
- They are very simple classes and the pattern can be easily implemented without having to use them. That's why, **since Java version 9**, these classes have been declared ***deprecated***.
- In the following examples we will use **our own `Subject` and `Observer` classes**, which are similar to the Java classes but, as they are not part of the API, we can adapt them to our particular problem.
- To make a more complex and asynchronous Observer pattern there are other **alternatives in the Java API**. For example, the `Flow` API or the *RxJava* library for reactive streams style programming.

Observer pattern example



Ejemplo del Patrón Observador

Clase `Observer`

```
public interface Observer {
    void update(Subject s);
}
```

`Observer` defines the interface that the observers will implement

Clase `Subject`

```
public abstract class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer o) { observers.add(o); }
    public void detach(Observer o) { observers.remove(o); }

    public void notifyObservers() {
        for (Observer o : observers)
            o.update(this);
    }
}
```

`Subject` maintains a list of observers that it notifies through `update`.

Example of the Observer pattern

SwimmingPool class

```
public class SwimmingPool extends Subject {
    private int level;
    public static final int MAX = 100;
    public static final int MIN = 80;

    public int getLevel() { return level; }

    public void fillWater(int increment) {
        level += increment;
        if (level > MAX) notifyObservers();
    }

    public void drainWater(int decrement) {
        level -= decrement;
        if (level < MIN) notifyObservers();
    }

    public SwimmingPool(int level) { this.level = level; }
}
```

The swimming pool notifies observers whenever there is a relevant change in its state

Example of the Observer pattern

FireFighters class

```
public class FireFighters implements Observer {
    @Override
    public void update(Subject s) {
        int amount = 0;
        System.out.println("FireFighters receive a notification");
        SwimmingPool sp = (SwimmingPool) s;
        if (sp.getLevel() < SwimmingPool.MIN) {
            System.out.println("FireFighters: Not my problem");
        }
        else if (sp.getLevel() > SwimmingPool.MAX) {
            amount = sp.getLevel()-SwimmingPool.MAX;
            System.out.println("FireFighters: Let's drain " + amount
                               + " from the pool");
        }
    }
}
```

Firefighters are only interested in possible floods

Example of the Observer pattern

`LifeGuards` class

Lifeguards are interested in any level value outside the established margins

```
public class LifeGuards implements Observer {
    @Override
    public void update(Subject s) {
        int amount = 0;
        SwimmingPool sp = (SwimmingPool) s;
        System.out.println("LifeGuards receive a notification");
        if (sp.getLevel() < SwimmingPool.MIN) {
            amount = SwimmingPool.MIN-sp.getLevel();
            System.out.println("LifeGuards: Let's fill " + amount
                               + " in the pool");
        }
        else if (sp.getLevel() > SwimmingPool.MAX) {
            amount = sp.getLevel()-SwimmingPool.MAX;
            System.out.println("LifeGuards: Let's drain " + amount
                               + " from the pool");
        }
    }
}
```

Example of the Observer pattern

Use of `SwimmingPool` and its various observers

```
public class Main {
    public static void main(String[] args) {
        SwimmingPool myPool = new SwimmingPool(90);
        LifeGuards lifeGuards = new LifeGuards();
        FireFighters fireFighters = new FireFighters();

        myPool.attach(lifeGuards);
        myPool.attach(fireFighters);

        System.out.println("myPool level: " + myPool.getLevel()
                           + " let's fill 10...");
        myPool.fillWater(10);
        System.out.println("myPool level: " + myPool.getLevel()
                           + " let's fill 30...");
        myPool.fillWater(30);
        System.out.println("myPool level: " + myPool.getLevel()
                           + " let's drain 80");
        myPool.drainWater(80);
    }
}
```

Observers are attached to the pool

We delegated to the pool the responsibility to warn the observers

Advantages and disadvantages of the Observer Pattern

- **Advantages**
  - The notification that a subject sends needn't specify its receiver, it is broadcast automatically to all interested objects that subscribed to it.
  - The subject's only responsibility is to notify its observers. It's up to the observer to handle or ignore a notification.
- **Disadvantages**
  - Because observers have no knowledge of each other's presence, a seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.
  - The simple update protocol provides no details on what changed in the subject. Observers may have to work hard to discover what has changed $\Rightarrow$ **pull model** (the one implemented) vs. **push model**.

pull model vs. *push* model

- ***pull* model**
  - The observers *pull* the subject.
  - The subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.
  - **Advantage**: The subject is generally oblivious to the needs of the observers.
  - **Disadvantage**: It may be inefficient, because the observers must determine what has changed.

Notifications of temperature changes (*pull*)

- Let's suppose that the pool now wants to notify changes in temperature as well...

Modifications in `SwimmingPool`

```
public class SwimmingPool extends Subject {
    private int temperature;
    public static final int MIN_TEMP = 24;
    public static final int MAX_TEMP = 28;

    public int getTemperature() { return temperature; }

    public void setTemperature(int temperature) {
        this.temperature = temperature;
        if (temperature < MIN_TEMP || temperature > MAX_TEMP)
            notifyObservers();
    }

    // ...
}
```

A new field is added and relevant notifications are sent about it

Notifications of temperature changes (*pull*)

Modifications in `LifeGuards`

```
public class LifeGuards implements Observer {
    @Override
    public void update(Subject s) {
        int amount = 0;
        SwimmingPool sp = (SwimmingPool) s;
        System.out.println("LifeGuards receive a notification");
        // ...
        if (sp.getTemperature() < SwimmingPool.MIN_TEMP) {
            amount = SwimmingPool.MIN_TEMP-sp.getTemperature();
            System.out.println("LifeGuards: Let's heat the pool by"
                               + amount + " degrees");
        } else if (sp.getTemperature() > SwimmingPool.MAX_TEMP) {
            amount = sp.getTemperature()-SwimmingPool.MAX_TEMP;
            System.out.println("LifeGuards: Let's cool the pool by "
                               + amount + " degrees");
        }
    }
}
```

Clients who are interested in the new field are modified, the rest (`FireFigthers`) keep unchanged

pull model vs. *push* model

- ***push* model**
  - The subject *pushes* the observers.
  - The subject sends observers detailed information about the change, whether they want it or not.
  - Assumes subjects know something about their observers' needs.
  - **Advantage**: Communication is more efficient, Observer classes know in advance what changed in the Subject.
  - **Disadvantage**: Observers and Subject are less independent. Subject classes make assumptions about Observer classes that might not always be true.

Example of the Observer pattern

`Subject` class (*push*)

```
public abstract class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer o) {
        observers.add(o);
    }

    public void detach(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers(int level, int min, int max) {
        for (Observer o : observers)
            o.update(level, min, max);
    }
}
```

In *update* we pass all the information necessary to respond to the event

Example of the Observer pattern

`SwimmingPool` class (*push*)

```
public class SwimmingPool extends Subject {
    private int level;
    public static final int MAX = 100;
    public static final int MIN = 80;

    public int getLevel() { return level; }

    public void fillWater(int increment) {
        level += increment;
        if (level > MAX) notifyObservers(level, MIN, MAX);
    }

    public void drainWater(int decrement) {
        level -= decrement;
        if (level < MIN) notifyObservers(level, MIN, MAX);
    }

    public SwimmingPool(int level) { this.level = level; }
}
```

The notification from `SwimmingPool` now includes this additional information

Example of the Observer pattern

`FireFighters` class (*push*)

```
public class FireFighters implements Observer {
    @Override
    public void update(int level, int min, int max) {
        int amount = 0;
        System.out.println("FireFighters receive a notification");
        if (level < min) {
            System.out.println("FireFighters: Not my problem");
        }
        else if (level > max) {
            amount = level - max;
            System.out.println("FireFighters: Let's drain " + amount
                               + " from the pool");
        }
    }
}
```

Firefighters do not need to access the `SwimmingPool` object now to get the relevant information

Example of the Observer pattern

`LifeGuards` class (*push*)

neither the lifeguards

```
public class LifeGuards implements Observer {
    @Override
    public void update(int level, int min, int max) {
        int amount = 0;
        System.out.println("LifeGuards receive a notification");
        if (level < min) {
            amount = min - level;
            System.out.println("LifeGuards: Let's fill " + amount
            + " in the pool");
        }
        else if (level > max) {
            amount = level - max;
            System.out.println("LifeGuards: Let's drain " + amount
            + " from the pool");
        }
    }
}
```

Notifications of temperature changes (*push*)

- If we add the temperature notifications we have to change `update` and make changes in all classes.

Modifications in `Subject`

```
public abstract class Subject {
  // ...
 public void notifyObservers(int level, int min, int max, int temperature,
                             int min_temp, int max_temp) {
    for (Observer o : observers)
      o.update(level, min, max, temperature, min_temp, max_temp);
  }
}
```

Modifications in `Observer`

```
public interface Observer {
    void update(int level, int min, int max, int temperature,
                int min_temp, int max_temp);
}
```

Notifications of temperature changes (*push*)

Modifications in `SwimmingPool`

```
public class SwimmingPool extends Subject {  // ...
  private int temperature;
  public static final int MIN_TEMP = 24;
  public static final int MAX_TEMP = 28;

  public void fillWater(int increment) {
      level += increment;
      if (level > MAX)
        notifyObservers(level, MIN, MAX, temperature, MIN_TEMP, MAX_TEMP);
  }
  public void drainWater(int decrement) {
      level -= decrement;
      if (level < MIN)
        notifyObservers(level, MIN, MAX, temperature, MIN_TEMP, MAX_TEMP);
  }
  public void setTemperature(int temperature) {
     this.temperature = temperature;
     if (temperature < MIN_TEMP || temperature > MAX_TEMP)
        notifyObservers(level, MIN, MAX, temperature, MIN_TEMP, MAX_TEMP);
  }
}
```

Notifications of temperature changes (*push*)

Modifications in `FireFighters`

```
public class FireFighters implements Observer {
    public void update(int level, int min, int max, int temperature,
                       int min_temp, int max_temp) { /* ... */ } }
```

Modifications in `LifeGuards`

```
public class LifeGuards implements Observer {
    public void update(int level, int min, int max, int temperature,
                       int min_temp, int max_temp) {  // ...
        } else if (temperature < min_temp) {
            amount = min_temp-temperature;
            System.out.println("LifeGuards: Let's heat the pool");
        }
        else if (temperature > max_temp) {
            amount = temperature - max_temp;
            System.out.println("LifeGuards: Let's cool the pool");
        }
    }
}
```

Adapter pattern

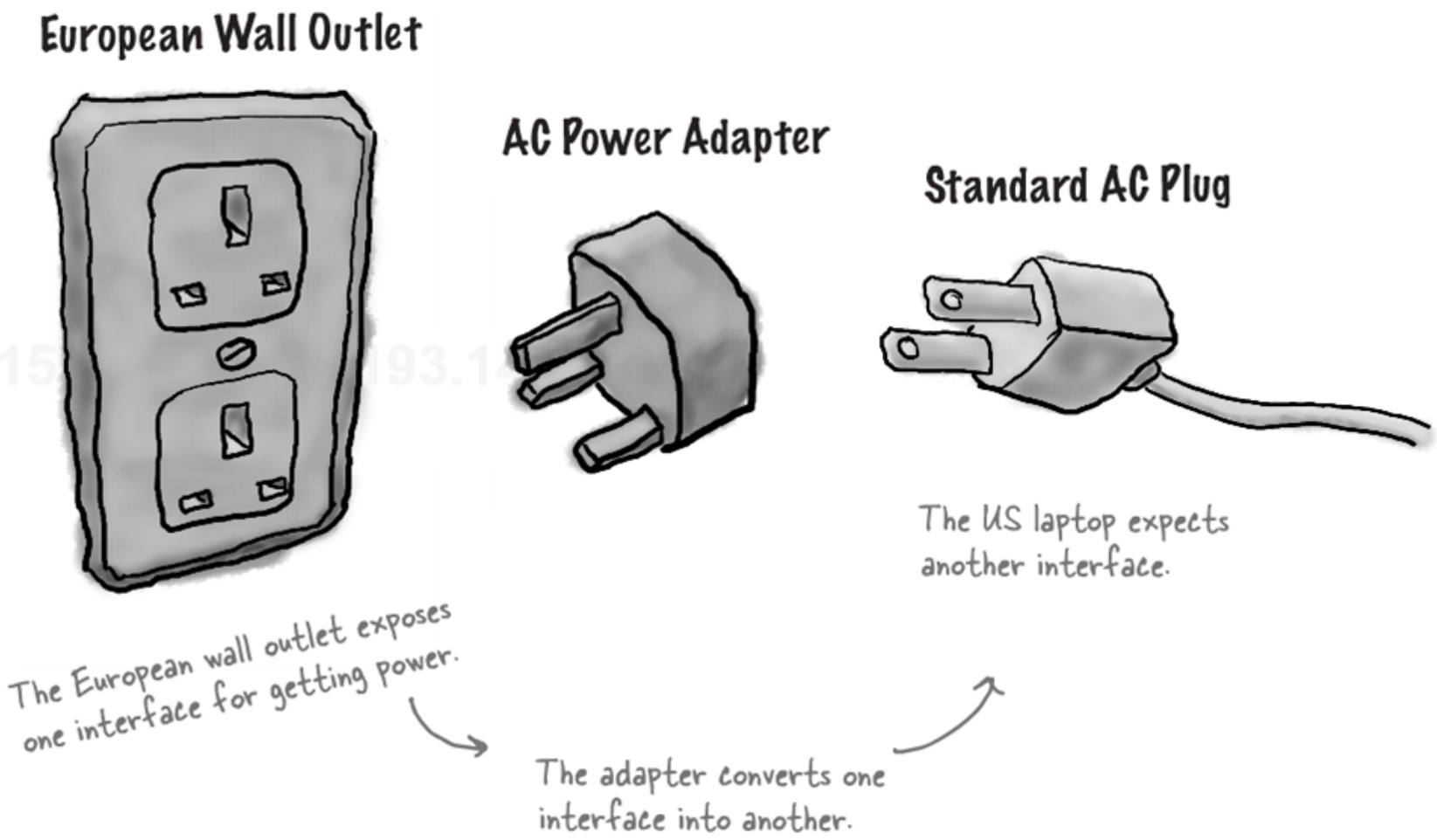
Adapter pattern

Converts the interface of a class into the one expected by clients. Adapter lets classes work together that couldn't otherwise due to incompatible interfaces.

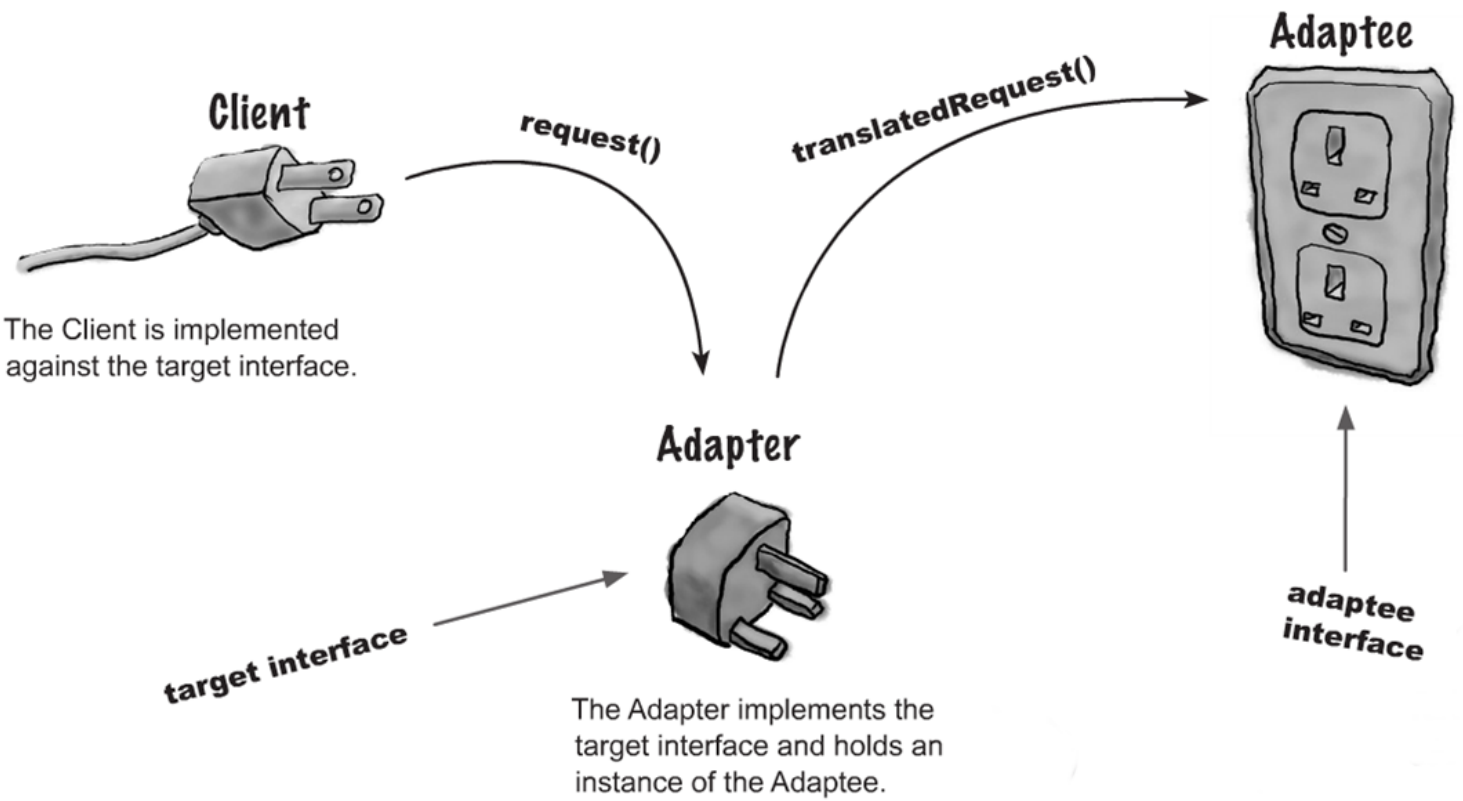
Adapter pattern characteristics

- The pattern assumes that changes cannot be made either in the interface that clients expect or in the interface provided by the server class.
- This could be because it is an external API, we do not have the source code or we simply do not want to modify the classes for a specific use.

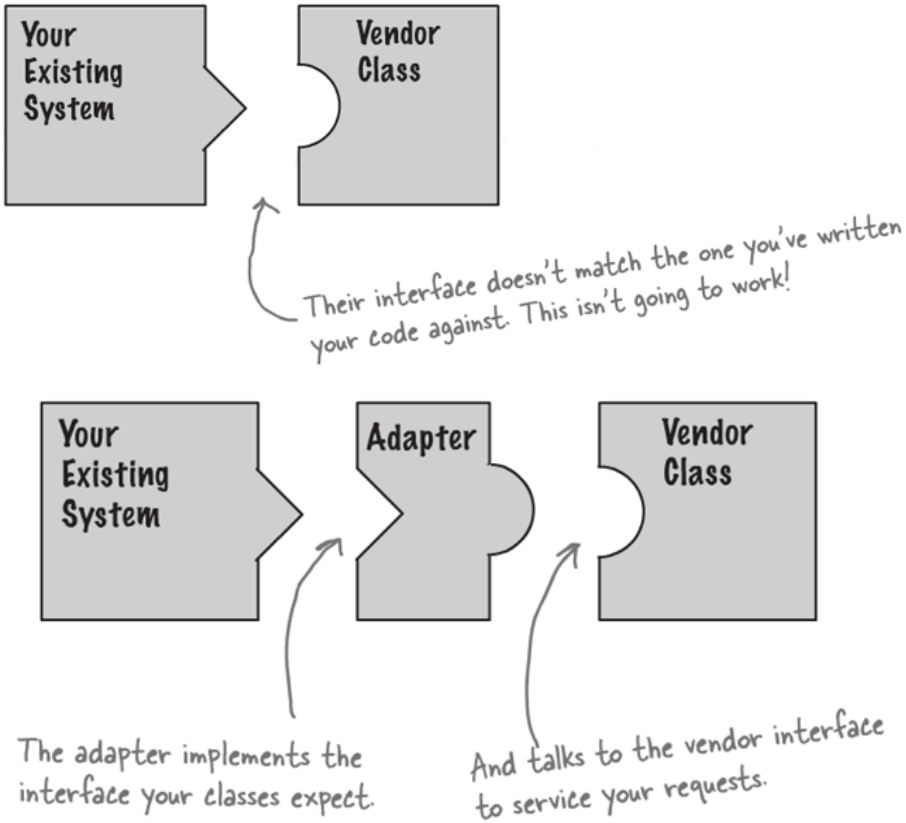
Adapter pattern

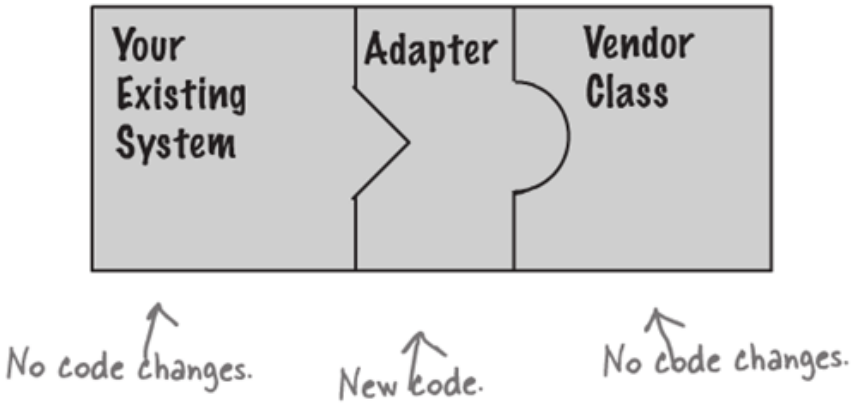


Adapter pattern

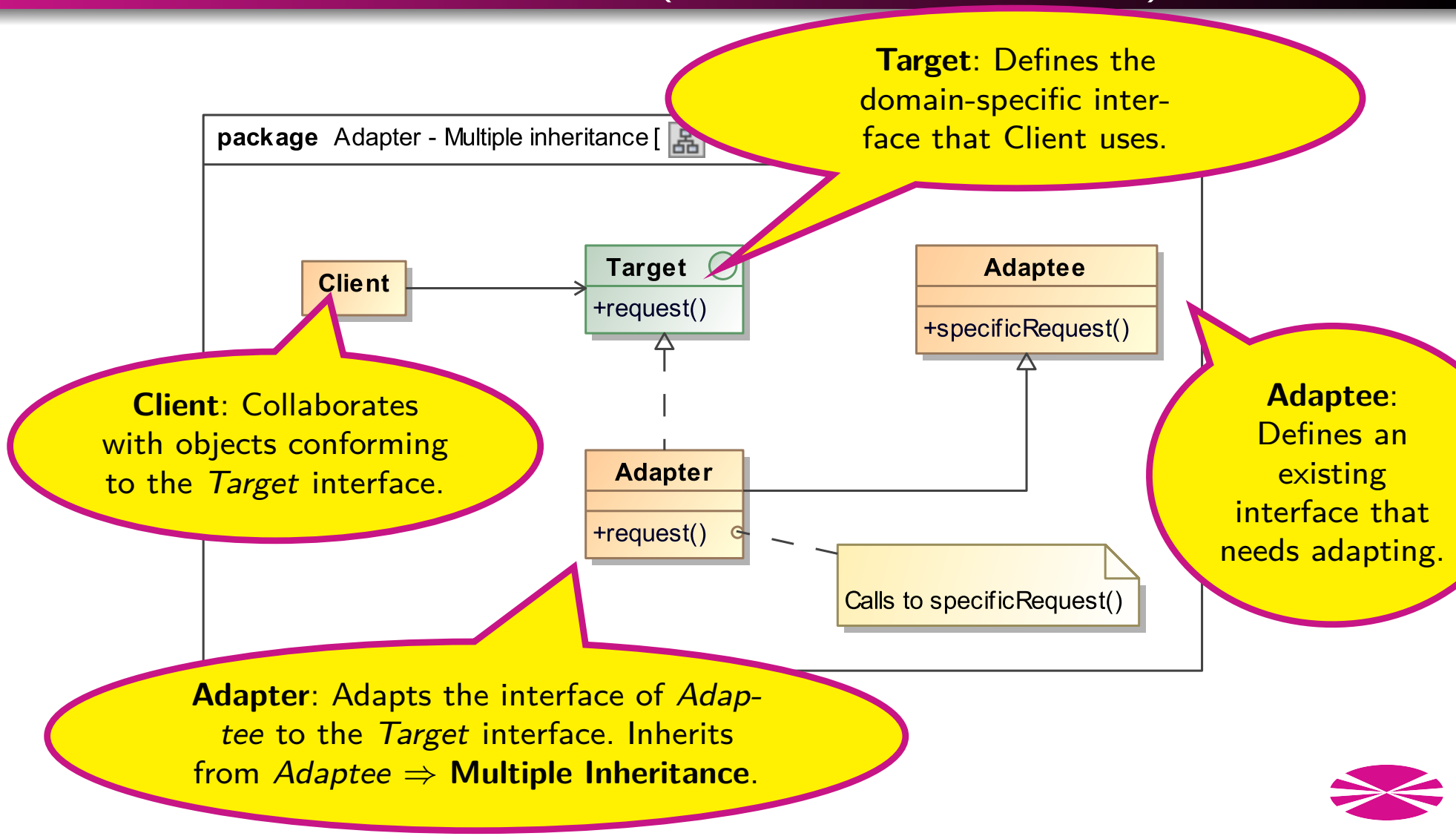


Adapter pattern

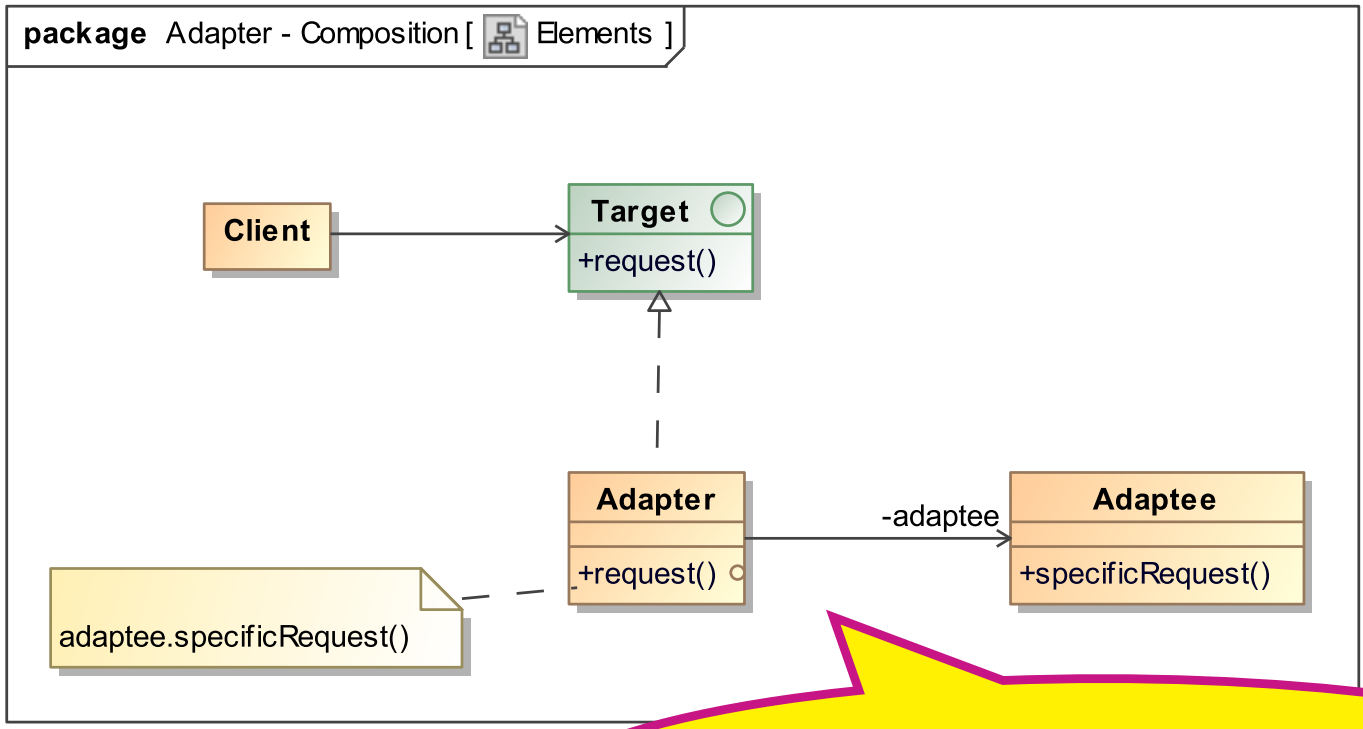




Adapter pattern elements (multiple inheritance)



Adapter pattern elements (composition)



Adapter: Adapts the interface of *Adaptee* to the *Target* interface. *Adaptee* is part of its state ⇒ **Composition**.

Differences between Multiple Inheritance and Composition

- **Multiple Inheritance** (class adapter)
  - A class adapter won't work when we want to adapt a class and all its subclasses.
  - Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - Introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
  - In Java it is only possible if `Target` is an interface and not an abstract class.
- **Composition** (object adapter)
  - Lets a single Adapter work with many Adaptees – that is, the Adaptee itself and all of its subclasses (if any).
  - The Adapter can also add functionality to all Adaptees at once.
  - Makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.
  - In Java, `Target` can be an abstract class or an interface.

Adapter pattern example

- The Java API defines a `Stack` class for implementing the behavior of a stack.
- `Stack` inherits from `Vector` (something that is a mistake) and includes the typical stack methods: `pop`, `push` and `peek`.
- The new Java API for collections included a `Queue` interface for defining the behavior of queues.
- The queue does not have to be strictly a FIFO queue (*First In, First Out*), it can be a priority queue, or even a LIFO queue (*Last In, First Out*).
- We have a client that uses `Queue` and we want to pass it a `Stack` object, but their interfaces are incompatible (and being API classes we can not modify them), so we need an adapter.

Adapter pattern example

- `Queue` methods

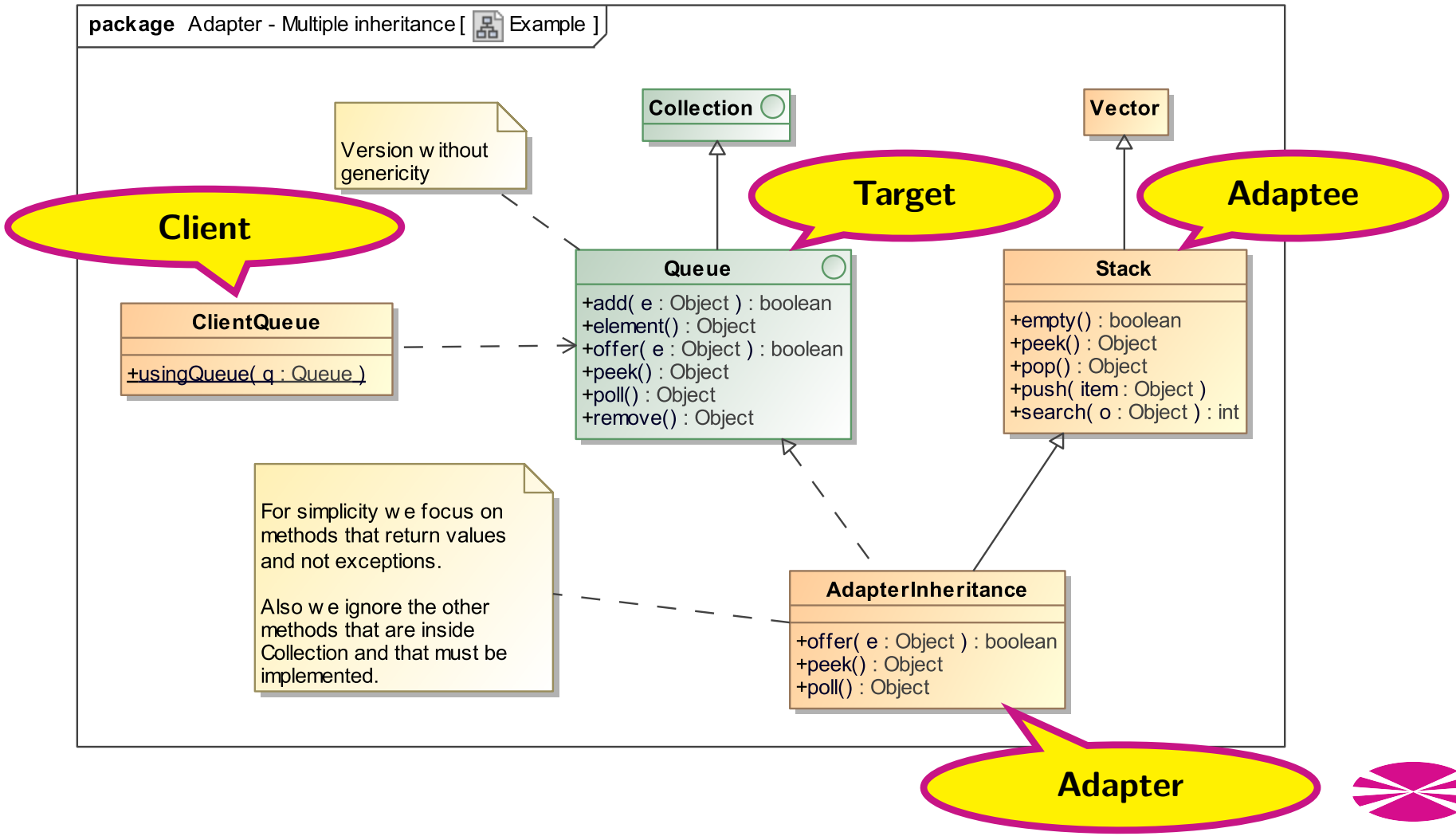
	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- `Stack` methods

Methods

Modifier and Type	Method and Description
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

Adapter pattern example (Multiple Inheritance)



Adapter pattern example (Multiple Inheritance)

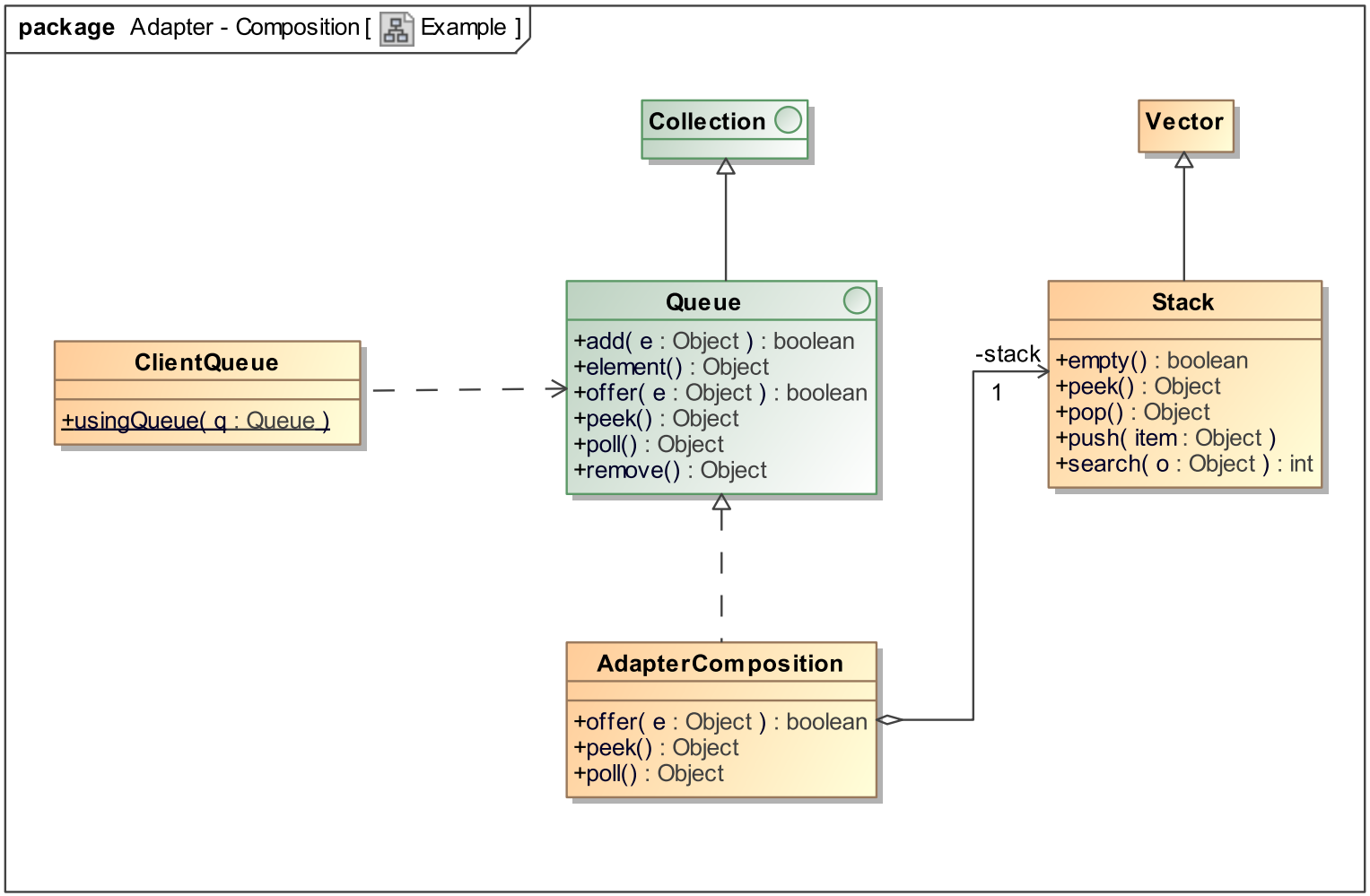
Class `AdapterInheritance`

```
public class AdapterInheritance extends Stack implements Queue {
    @Override
    public boolean offer(Object e) {
        push(e);
        return true;
    }

    @Override
    public Object peek() {
        if (empty()) return null;
        else return super.peek();
    }

    @Override
    public Object poll() {
        if (empty()) return null;
        else return pop();
    }
    // Other methods from Queue and Collection...
}
```

`AdapterInheritance` inherits from Adaptee and implements the Target interface.

Adapter converts the call to the `Queue` interface into a call to the `Stack` class making small adjustments.

Adapter pattern example (Composition)



Adapter pattern example (Composition)

Class `AdapterComposition`

```
public class AdapterComposition implements Queue {
    private Stack stack = new Stack();

    @Override
    public boolean offer(Object e) {
        stack.push(e);
        return true;
    }
    @Override
    public Object peek() {
        if (stack.empty()) return null;
        else return stack.peek();
    }
    @Override
    public Object poll() {
        if (stack.empty()) return null;
        else return stack.pop();
    }
    // Other methods from Queue and Collection...
}
```

`AdapterComposition` implements the Target interface and includes a private attribute that is an instance of Adaptee.

Adapter delegates the `Queue` interface calls on its internal object of class `Stack`.

Adapter pattern example

Class `ClientQueue`

`ClientQueue` expects a `Queue` object.

```
public class ClientQueue {
    public static void usingQueue(Queue q) {
        q.offer("A");
        q.offer("B");
        q.offer("C");
        System.out.println(q.poll() + " " + q.poll() + " " +  q.poll());
                                        // Prints C B A
        System.out.println(q.peek());// Prints the object already in
    }                                   // "q" or "null" if there was none

    public static void main(String[] args) {
        AdapterInheritance adapterH = new AdapterInheritance();
        ClientQueue.usingQueue(adapterH);

        AdapterComposition adapterC = new AdapterComposition();
        ClientQueue.usingQueue(adapterC);
    }
}
```

We pass it an adapter object so it can work with `Stack` as if it were a `Queue`.

Adapter pattern example in Java API (*Wrapper classes*)

Class `Integer`

```
public final class Integer extends Number
    private int value;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return value;
    }

    // Rest is omitted...
}
```

Adapter implements the Target objective, in our case it is `Object`. Our client can be, for example, an `ArrayList` that stores objects, not primitive types.

Adapter maintains a private instance of Adaptee, in our case a `int`.

Operations are delegated to the adaptee, in this case it is returning its value.

Principle of Least Knowledge (Law of Demeter)

- Patterns like **Observer** and **Adapter** try to reduce the coupling between two classes that have to collaborate communicating through interfaces or adapter classes.
- Another way to reduce coupling is reducing the number of connections between objects $\Rightarrow$ **Principle of Least Knowledge**, **"Tell, Don't Ask" principle**, **Facade pattern**.

Principle of Least Knowledge (Law of Demeter)

Class `Wallet`

```
public class Wallet {
  private int value;

  public int getTotalMoney() {
    return value;
  }

  public void addMoney(int deposit) {
    value += deposit;
  }

  public int subtractMoney(int debit) {
    if (debit > value)
      debit = value;
    value -= debit;
    return debit;
  }
}
```

Class `Customer`

```
public class Customer {
  private String name;
  private Wallet myWallet;

  public Customer(String n, Wallet w) {
    this.name = n;
    this.myWallet = w;
  }

  public String getName() {
    return name;
  }

  public Wallet getWallet() {
    return myWallet;
  }
}
```

Principle of Least Knowledge (Law of Demeter)

- **What's wrong with this code?**

Class `Deliveryman`

```
public class Deliveryman {
    public void obtainPayment(Customer customer, int payment) {
        Wallet w = customer.getWallet();
        int amountPaid = w.subtractMoney(payment);

        if (amountPaid >= payment)
            System.out.println("Thank you");
        else
            System.out.println("I'll be back");
    }
}
```

Principle of Least Knowledge (Law of Demeter)

Principle of Least Knowledge (Law of Demeter)

"Never talk to strangers... only talk to your friends".

Principle of Least Knowledge characteristics

- The fundamental notion is that a given object should assume as little as possible about the structure or properties of other objects (information hiding).
- It is a specific case of loose coupling, you only talk with your closest objects.
- Named for its origin in the Demeter Project. Demeter is the Greek goddess of agriculture.

Principle of Least Knowledge (Law of Demeter)

Principle of Least Knowledge (in object-oriented software)

An object's method should only invoke the methods of the following kinds of objects:

- The object itself (`this`).
- The method's parameters.
- Its direct component objects.
- An element of a collection which is an attribute of the enclosing object.
- Any objects the method creates/instantiates.

Principle of Least Knowledge (Law of Demeter)

Class that fulfills the principle

```
public class ClassA {
  private ClassC cc;
  private List<ClassD> collection = new ArrayList<>();

  public void methodA1 (ClassB cb)   {
    this.methodA2(); // Calling another method of the same object.
    cb.methodB();    // Calling a method of a parameter.
    cc.methodC();    // Calling a method of an attribute.
    collection.get(0).methodD(); // Calling a method of an element of a
                                 // collection that is an attribute
    ClassE ce = new ClassE();
    ce.methodE();    // Calling a method of a created object
  }

  public void methodA2()  { }
}
```

Principle of Least Knowledge (Law of Demeter)

- `Deliveryman` does not fulfill the principle because it calls a method (`subtractMoney`) of an object (`w`) that it is not an attribute, a parameter and was not created inside the `obtainPayment` method.

`Deliveryman` does not fulfill the principle

```
public class Deliveryman {
    public void obtainPayment(Customer customer, int payment) {
        Wallet w = customer.getWallet();
        int amountPaid = w.subtractMoney(payment);

        if (amountPaid >= payment)
            System.out.println("Thank you");
        else
            System.out.println("I'll be back");
    }
}
```

Does not fulfill the Law of Demeter

Principle of Least Knowledge (Law of Demeter)

Class `Customer`

```
public class Customer {
    private String name;
    private Wallet myWallet;

    public Customer(String n, Wallet w) {
        this.name = n;
        this.myWallet = w;
    }

    public String getName() {
        return name;
    }

    // public Wallet getWallet() {
    //     return myWallet;
    // }

    public int obtainPayment(int payment) {
        return myWallet.subtractMoney(payment);
    }
}
```

`getWallet()` is removed.

A new method to obtain payments is included.

Principle of Least Knowledge (Law of Demeter)

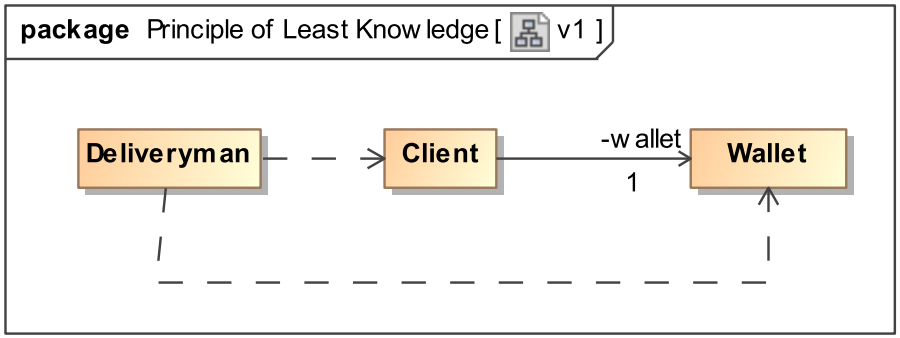
Class `Deliveryman`

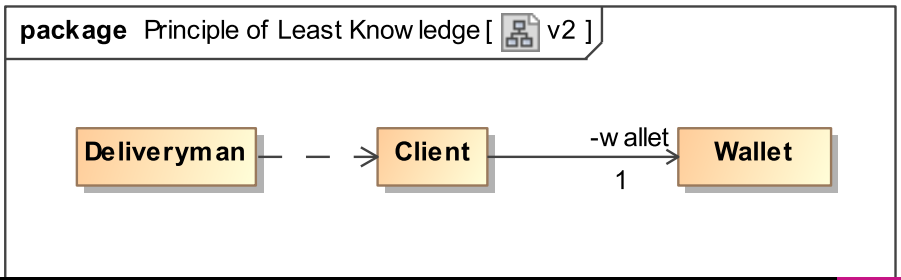
```
public class Deliveryman {
  public void obtainPayment(Customer customer, int payment) {
    int amountPaid = customer.obtainPayment(payment);

    if (amountPaid >= payment)
      System.out.println("Thank you");
    else
      System.out.println("I'll be back");
  }
}
```

`Deliveryman` only interacts with `Customer` and knows nothing about `Wallet`.

Principle of Least Knowledge (Law of Demeter)

- **Advantages**
  - Coupling is lower in this solution.
  - `Deliveryman` is not affected by changes in `Wallet`.
  - The payment mechanism is in `Customer`, so `Deliveryman` only knows the final result but not the mechanism (we can use the wallet or borrow money from a neighbor).
  - It is a better distribution of responsibilities between classes.





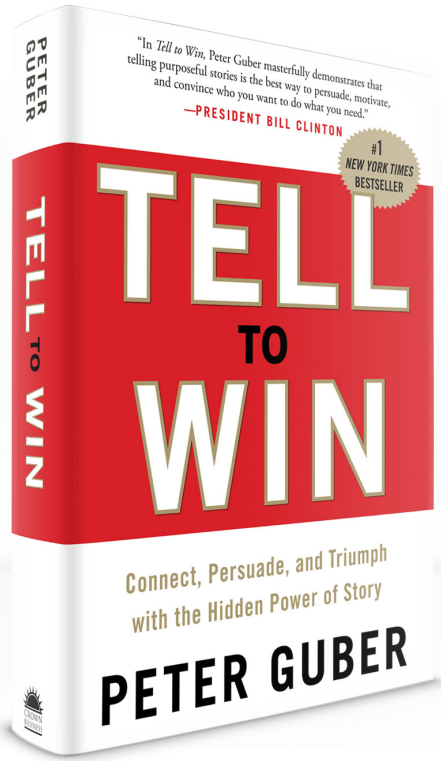
Principle of Least Knowledge (Law of Demeter)

- **Disadvantages**
  - The `Customer` class becomes more complex because it has to add a method (`obtainPayment`), but in exchange we have simplified `Deliveryman`.
  - Another drawback is trying to apply this solution to every case that seems appropriate (a police officer asks for a driver license, etc.).

The "Tell, Don't Ask" principle

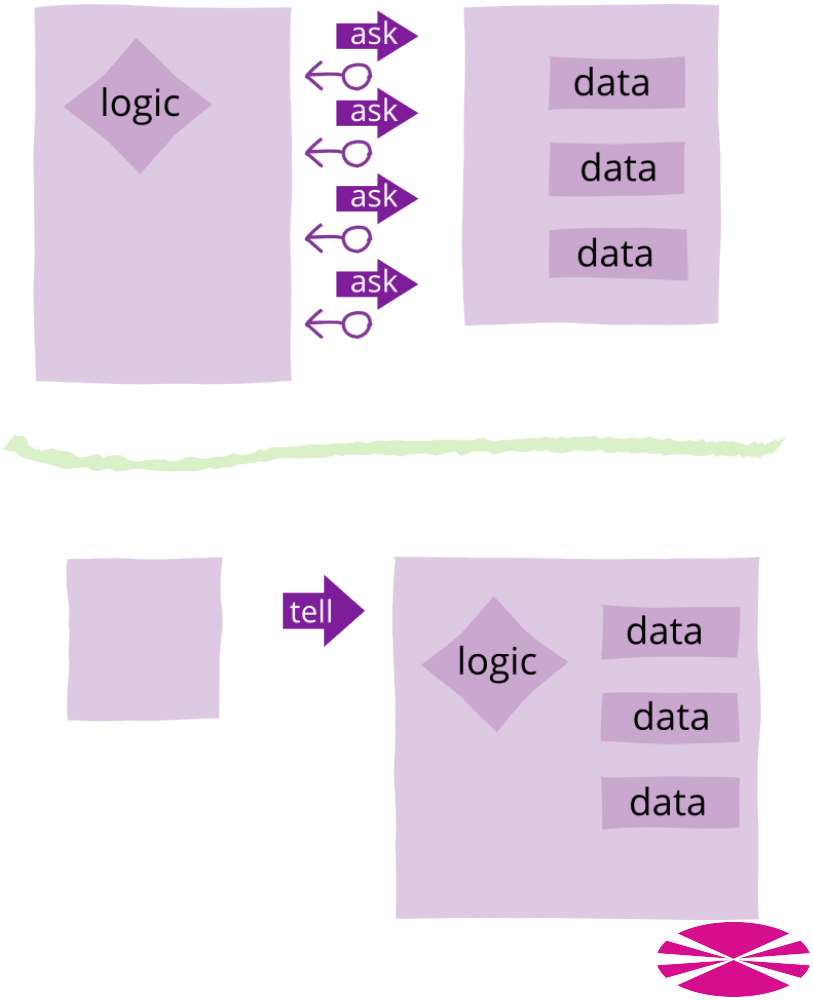
The "Tell, Don't Ask" principle

Procedural code asks information and then makes decisions.
Object-oriented code tells objects to do things.



"Tell, Don't Ask" principle

- **Ask**
  - Ask the object about its state, and then call methods on that object based on decisions made outside of the object.
  - Some of its behavior is located outside of the object, and internal state is exposed (leaky abstraction) to the outside world.
- **Tell**
  - Tell objects what you want them to do improving abstraction and reducing coupling.



"Tell, Don't Ask" principle example

- **Rational number**: a number that can be expressed as the quotient of two integers.

Class `Rational`

```
public class Rational {
    private int numerator;
    private int denominator;

    public Rational(int numerator, int denominator) {
        if (denominator == 0)
            throw new IllegalArgumentException();
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int getNumerator() { return numerator; }
    public int getDenominator() { return denominator; }

    // ...
}
```

"Tell, Don't Ask" principle example

- We assume that, within the code of another class (`UseRational`), we want to check if a rational is equal to a given value (for example `1/2`).

Class `UseRational`

```
public class UseRational {
    public static void main(String[] args) {
        Rational r1 = new Rational(1,2);

        // ...

        if (r1.getNumerator() == 1 && r1.getDenominator() == 2)
            System.out.println("You have half of a cake...");
    }
}
```

"Tell, Don't Ask" principle example

- **Problems**
  - `UseRational` is examining the internal state of Rational to make a decision.
  - We are unnecessarily exposing the internal structure of `Rational`.
  - We are locating behavior of `Rational` outside it $\Rightarrow$ the comparison of two rational values.
- **Solution**
  - Define an `equals` method in `Rational` that is responsible for making the comparisons.
  - This method could be even more complex, allowing it to detect equivalent fractions: $1/2 = 2/4$.

"Tell, Don't Ask" principle example

Simple `equals` method of `Rational`

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Rational other = (Rational) obj;
    if (this.numerator != other.numerator) {
        return false;
    }
    if (this.denominator != other.denominator) {
        return false;
    }
    return true;
}
```

"Tell, Don't Ask" principle example

Complex `equals` method of `Rational` using equivalences

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Rational other = (Rational) obj;
    // a/b and c/d, are equivalent if and only if a*d == b*c
    int value1 = this.numerator * other.denominator;
    int value2 = this.denominator * other.numerator;
    if (value1 != value2) {
        return false;
    }
    return true;
}
```

"Tell, Don't Ask" principle example

Class `UseRational` following the principle

```
public class UseRational {
    public static void main(String[] args) {
        Rational r1 = new Rational(1,2);
        Rational r2 = new Rational(2,4);
        Rational rHalf = new Rational(1,2);

        if (r1.equals(rHalf))
            System.out.println("You have half a cake...");

        if (r2.equals(rHalf))
            System.out.println("You have another half of a cake...");
    }
}
```

"Tell, Don't Ask" principle

- **Advantages**
  - Encourages combining data and behavior in the same abstraction. Things that are tightly coupled should be in the same component.
  - Leads to loosely coupled designs in a similar way to the Law of Demeter. If your code already follows the Law of Demeter, it will be easier to apply the Tell Don't Ask principle.
  - Focuses on the messages passing between objects. Sending instructions to an object is better than querying it.
- **Disadvantages**
  - May encourage people to become *getter eradicators*, seeking to get rid of all query methods. But there are times when objects collaborate effectively by providing information.
  - May conflict with the Single Responsibility Principle and objects can grow too large if not used carefully.

Facade pattern

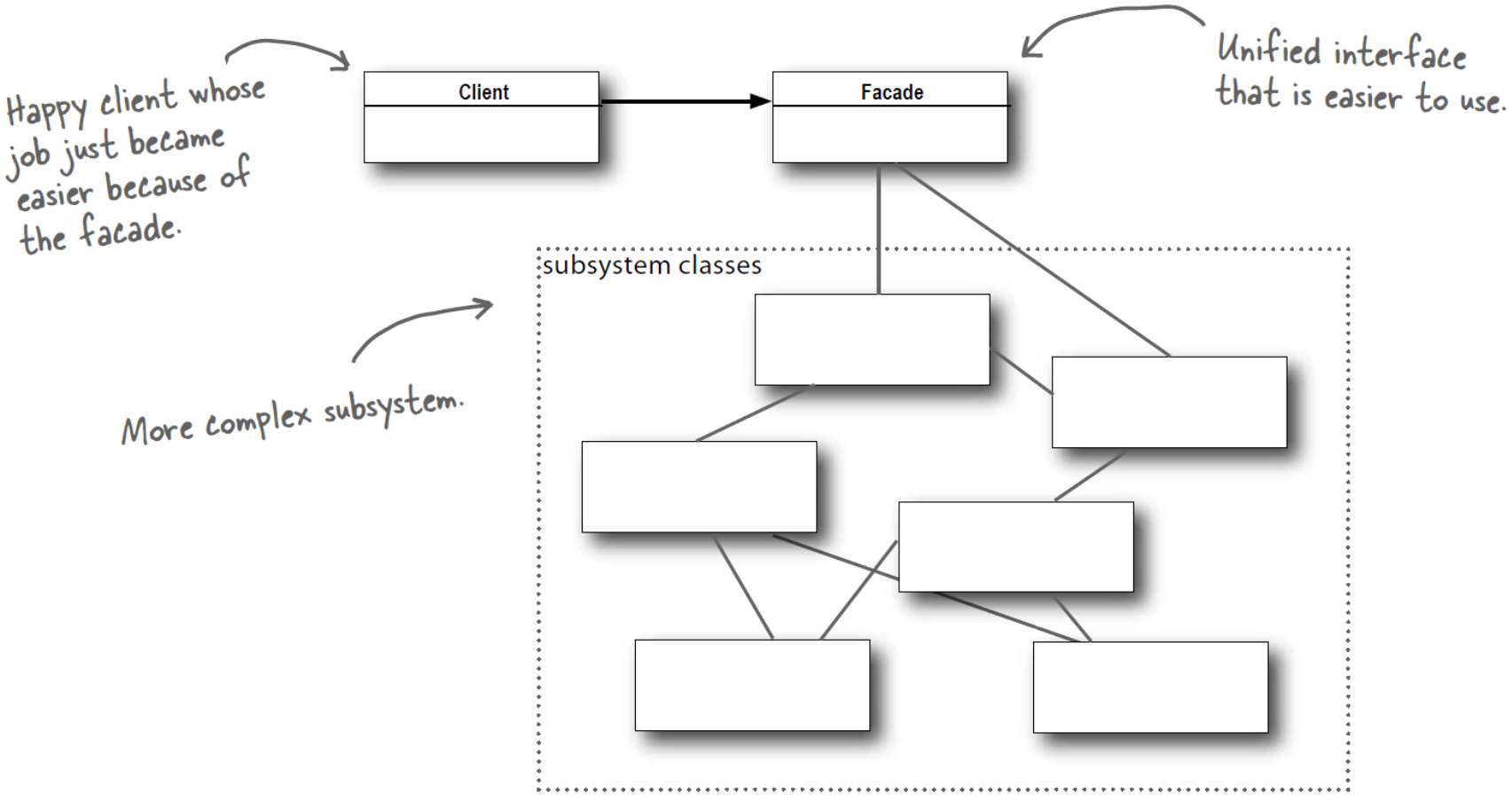
Facade pattern

Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade pattern elements

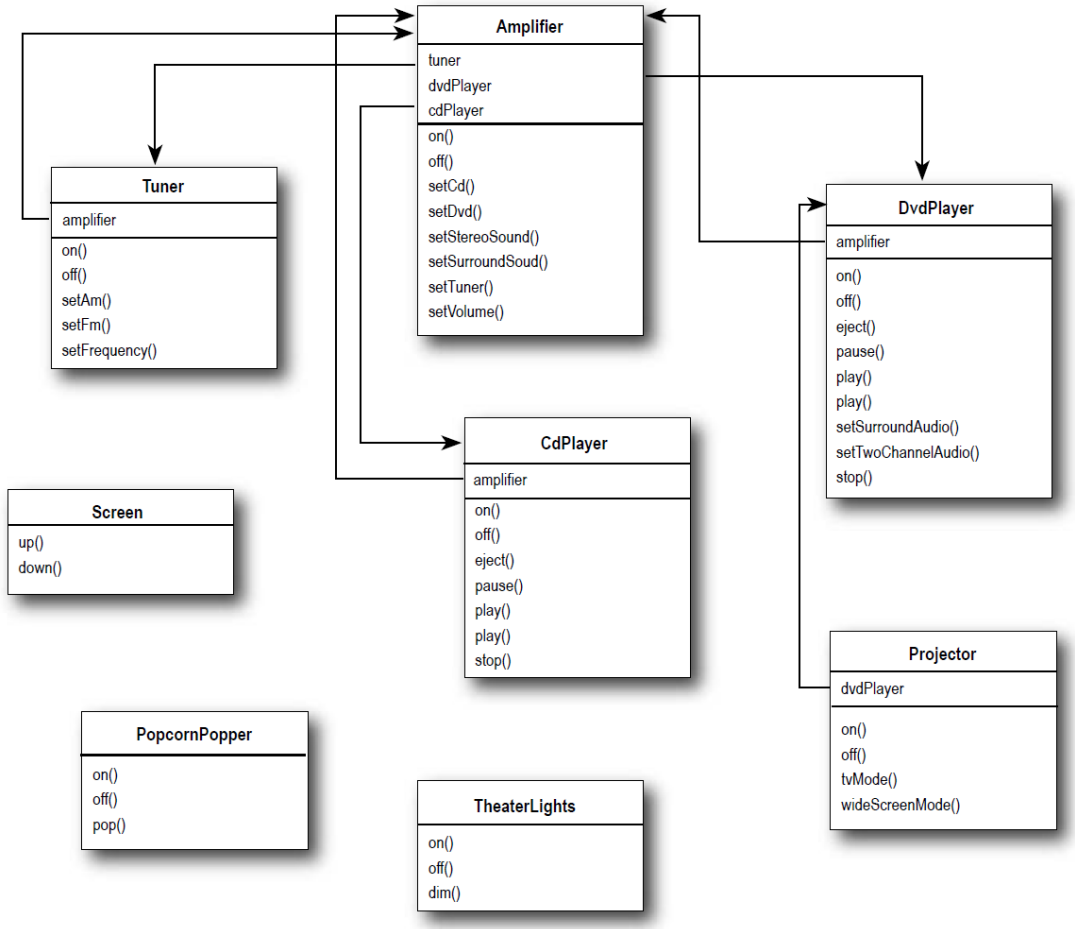
- **Facade**:
  - Knows which subsystem classes are responsible for a request and delegates client requests to suitable subsystem objects.
- **Subsystem classes**:
  - Implement subsystem functionality handling work assigned by the Facade object. They have no knowledge of the facade (keeping no references to it).

Facade pattern elements



Facade pattern example

- Classes for controlling a "Home Theater".

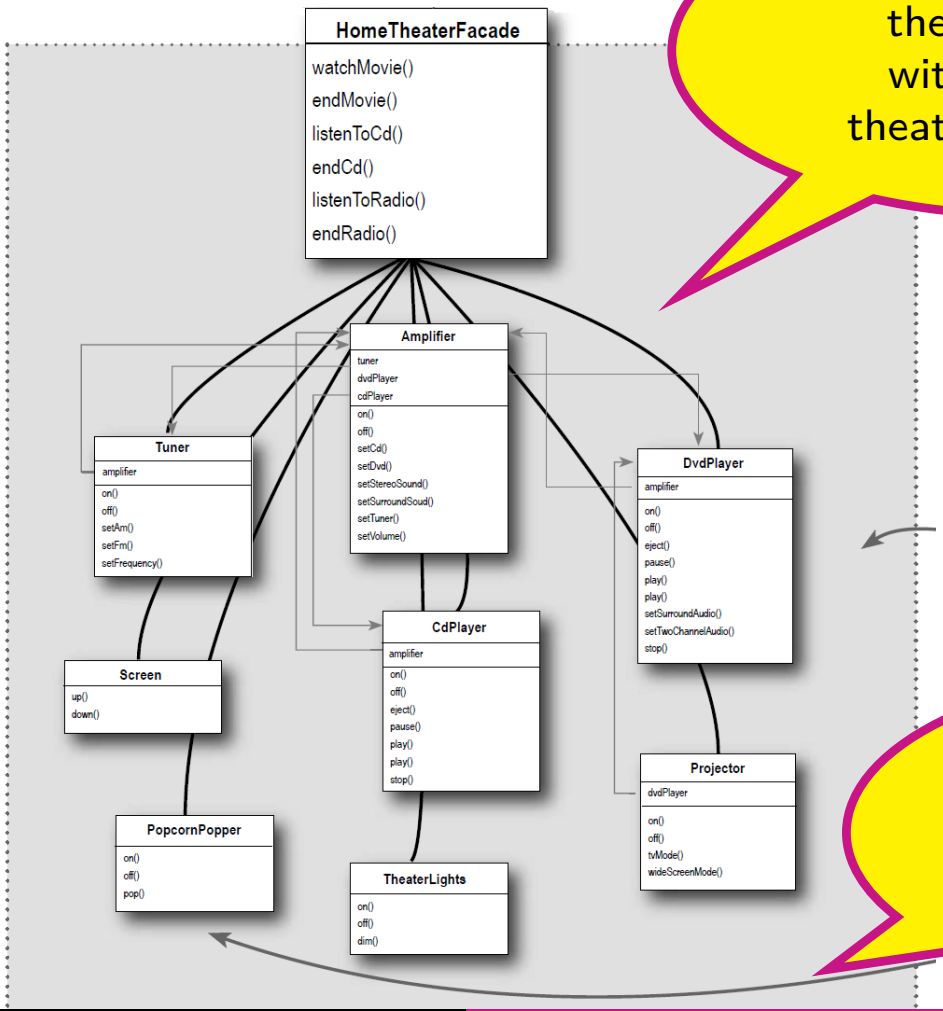


Facade pattern example

- **Actions to take when watching a movie**:
  - Turn on the popcorn popper.
  - Start the popper popping.
  - Dim the lights.
  - Put the screen down.
  - Turn the projector on.
  - Put the projector on wide-screen mode.
  - Turn the sound amplifier on.
  - Set the amplifier to DVD input.
  - Set the amplifier to surround sound.
  - Set the amplifier volume to medium (5).
  - Turn the DVD Player on.
  - Start the DVD Player.

Facade pattern example

- "Home Theater" using a facade.



`watchMovie` simplifies the interaction with the home theatre subsystem.

Accessing the subsystem classes avoiding the facade is still possible.

Facade pattern example

Class `HomeTheaterFacade`

```
public class HomeTheaterFacade {
    private Amplifier amp;
    private Tuner tuner;
    private DvdPlayer dvd;
    private CdPlayer cd;
    private Projector projector;
    private TheaterLights lights
    private Screen screen;
    private PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp, Tuner tuner, DvdPlayer dvd,
            CdPlayer cd, Projector projector, TheaterLights lights,
            Screen screen, PopcornPopper popper) {
        // ...
    }
    // ...
}
```

Facade knows the objects it works with.

Facade pattern example

Method `watchMovie` of `HomeTheaterFacade`

```
public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}
```

Facade pattern

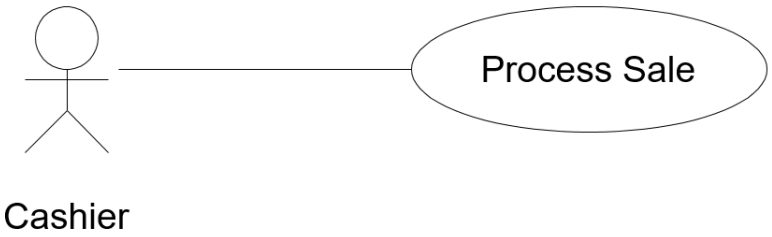
- **Advantages**
  - It abstracts clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
  - It promotes weak coupling between the subsystem and its clients. This lets you vary the components of the subsystem without affecting its clients.
  - It doesn't preclude applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.
- **Facade vs. Adapter**
  - They differ more in *intention* rather than structure.
  - The Adapter Pattern converts the interface of a class into the one expected by a client.
  - The Facade Pattern alters an interface, but for a different reason: to simplify the interface.

Facades and use cases

- Requirements analysis lets us identify the users who are going to use the system and how they will use it.
- Use cases can be used to obtain System Sequence Diagrams (high-level diagrams that try to identify the functions that a system offers to a given actor).
- These actions can be the origin of a facade that the system offers to the actor.
- Actually, the facade is between the layers of the application. They are the functions that the model offers to the user interface that is being managed by the actor.

Facades and use cases

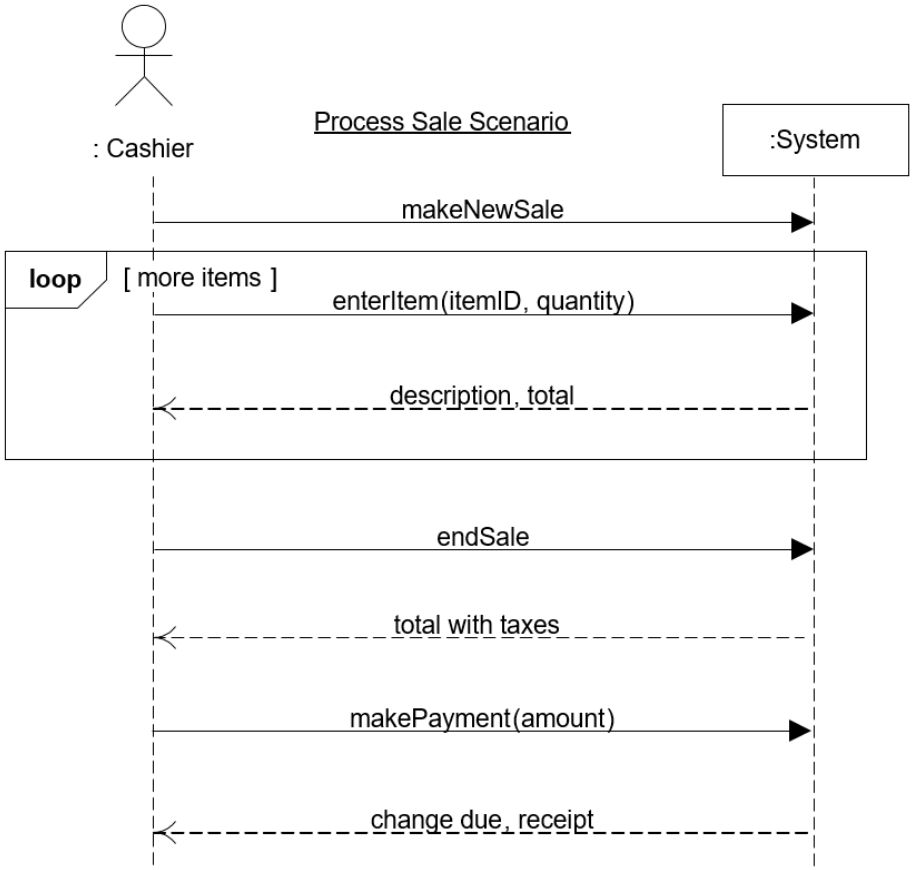
- **Process a sale in a supermarket**



- **Use case basic flow**
  1. Customer arrives at point of sale checkout with goods and/or services to purchase.
  2. Cashier starts a new sale.
  3. Cashier enters item identifier.
  4. System records sale line item and presents item description, price, and running total.
  5. Cashier repeats steps 3-4 until it is done.
  6. System presents total with taxes calculated.
  7. Cashier tells Customer the total, and asks for payment.
  8. Customer pays and System handles payment.
  9. System logs completed sale and sends sale and payment information to the external Accounting system and Inventory system.
  10. System presents receipt.
  11. Customer leaves with receipt and goods.

Facades and use cases

- **System Sequence Diagram**



- **Facade for a cashier (`Register`)**

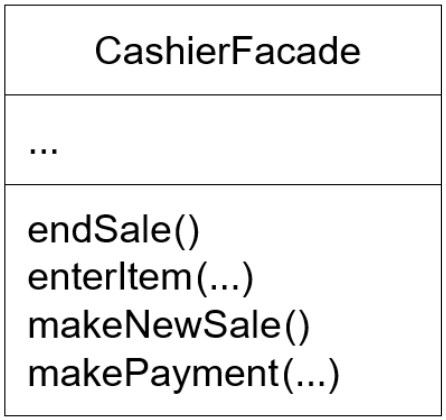


Table of Contents

Factory Method pattern

Factory Method pattern

Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

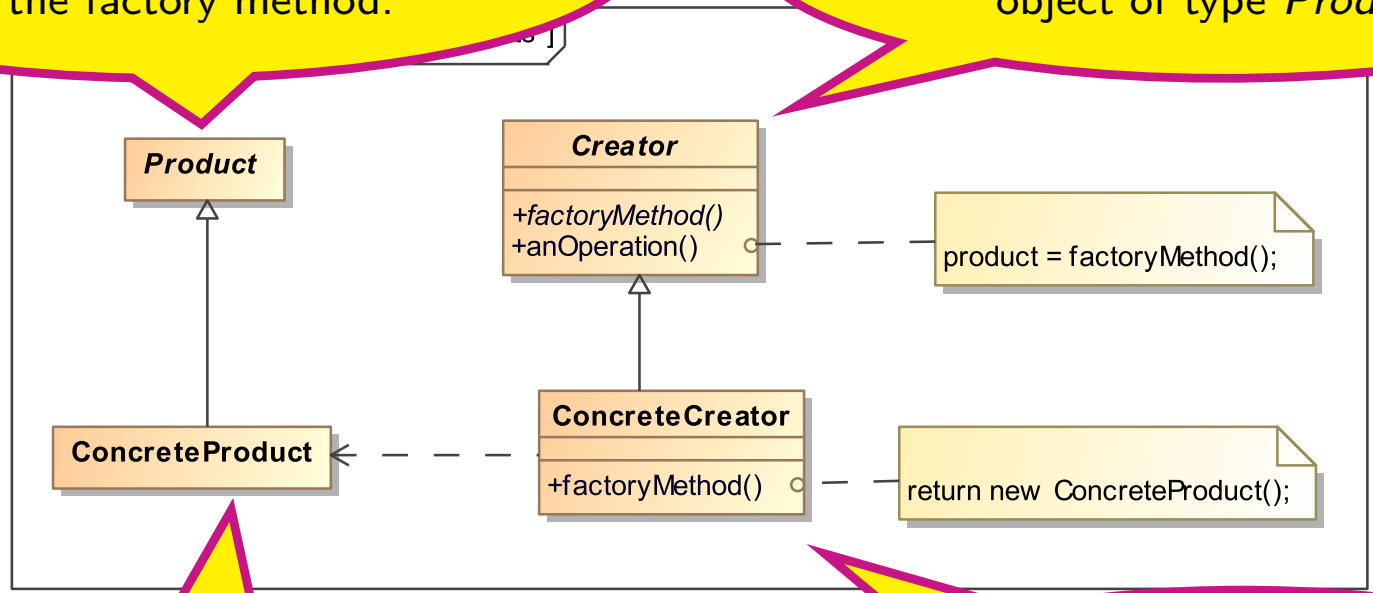
Factory Method pattern

- **Characteristics**
  - After a `new` operator, we must put a concrete class, not an abstract class or an interface.
  - So we are not fulfilling the Dependency Inversion Principle, since we are depending on concrete classes and not on abstract ones.
  - We can abstract the process of creation of objects and delegate it into subclasses using *factory methods*.

Factory Method pattern elements

Product: Defines the interface of objects created by the factory method.

Creator: Declares the factory method, which returns an object of type *Product*.



Concrete Product: Implements the *Product* interface.

Concrete Creator: Overrides the factory method to return an instance of a *Concrete Product*.

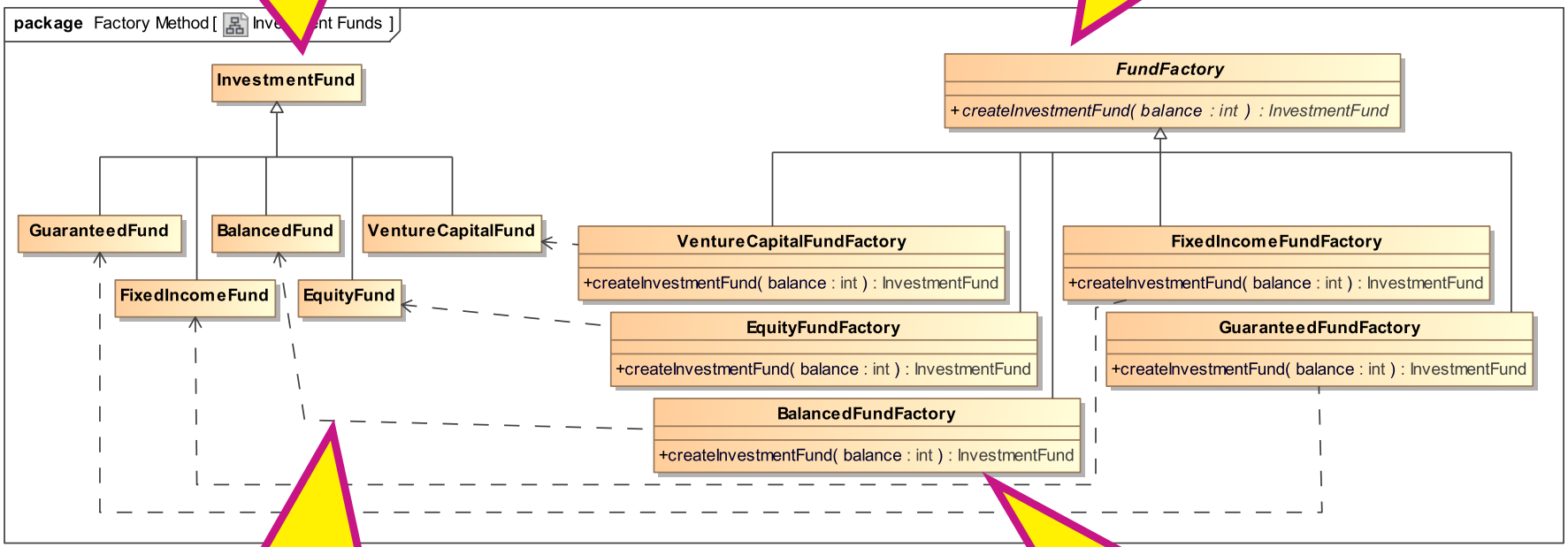
Factory Method pattern example

- Our financial institution invests the money of its clients in different types of mutual funds.
- In an increasing level of risk (but also of potential benefits) we have:
  - **Guaranteed funds**: Our company guarantees a given interest in a given investment period
  - **Fixed-income funds**: They invest in bonds, treasury bills, promissory notes, etc.
  - **Balanced funds**: Mixed investments from fixed-income funds and equity funds.
  - **Equity funds**: Invest in shares.
  - **Venture capital funds**: Invest in non-listed companies (e.g., *startups*)
- According to the profile of the investor (e.g., their tolerance to risk) we will invest the money in a certain type of product.

Factory Method pattern example

InvestmentFund plays the role of **Product**

FundFactory plays the role of **Creator**



The different types of funds play the role of **Concrete Product**

The different concrete factories play the role of **Concrete Creator**

Factory Method pattern example

Class `InvestmentFund`

```
public abstract class InvestmentFund {
    private int balance;
    private final String name;

    public InvestmentFund(String name) {
        this.name = name;
    }
    public InvestmentFund(String name, int balance) {
        this.name = name;
        this.balance = balance;
    }
    @Override
    public String toString() {
        return name + ": " + balance + " EUR";
    }
    public int getBalance() { return balance; }
    public void setBalance(int balance) { this.balance = balance; }
}
```

Factory Method pattern example

Class `GuaranteedFund`

```
public class GuaranteedFund extends InvestmentFund {
    public GuaranteedFund(int balance) {
        super("Guaranteed Fund", balance);

        // Invests in guaranteed assets
    }
}
```

Class `FixedIncomeFund`

```
public class FixedIncomeFund extends InvestmentFund {
    public FixedIncomeFund(int balance) {
        super("Fixed Income Fund", balance);

        // Invests in fixed income
    }
}
```

Factory Method pattern example

Class `FundFactory`

```
public abstract class FundFactory {
    public abstract InvestmentFund createInvestmentFund(int balance);
}
```

Class `FixedIncomeFundFactory`

```
public class FixedIncomeFundFactory extends FundFactory {
    @Override
    public InvestmentFund createInvestmentFund(int balance) {
        return new FixedIncomeFund(balance);
    }
}
```

Factory Method pattern example

Class `PortfolioManager`

`invest` delegates the creation of funds in the factory.

The factory is obtained according to the level of risk.

```
public class PortfolioManager {
  FundFactory factory;

  public InvestmentFund invest(int amount) {
    if (factory == null)
      factory = getFactory();
    return factory.createInvestmentFund(amount);
  }

  private FundFactory getFactory() {
    int riskLevel = calculateRiskLevel();
    if (riskLevel < 20) return new GuaranteedFundFactory();
    else if (riskLevel < 40) return new FixedIncomeFundFactory();
    else if (riskLevel < 60) return new BalancedFundFactory();
    else if (riskLevel < 80) return new EquityFundFactory();
    else return new VentureCapitalFundFactory();
  }
  private int calculateRiskLevel() {
    // do survey...
  }
}
```

Factory Method pattern example

- **Parameterized factory**:
  - This version of the factory method requires having two parallel hierarchies, one for the products and another for the factories.
  - This can be avoided with a parameterized version of the factory in which there is only one factory class that creates the products according to a parameter that is passed to it.
  - This eliminates the need for subclasses even though the creation process is less abstract (we tell the factory what to create).
  - It is still possible to create subclasses of the factory that alter the default behavior.

Factory Method pattern example

Class `FundFactory` parameterized using `switch`

```
class FundFactory {
    public InvestmentFund createInvestmentFund(String type, int balance) {
        switch (type) {
            case "GuaranteedFund":
                return new GuaranteedFund(balance);
            case "FixedIncomeFund":
                return new FixedIncomeFund(balance);
            case "BalancedFund":
                return new BalancedFund(balance);
            case "EquityFund":
                return new EquityFund(balance);
            case "VentureCapitalFund":
                return new VentureCapitalFund(balance);
            default:
                return new GuaranteedFund(balance);
        }
    }
}
```

Factory Method pattern example

- In order to avoid creating a new clause every time a new product is created (therefore, not fulfilling the open-closed principle) we can use other strategies to create the parameterized factory. For example using *reflection* techniques:

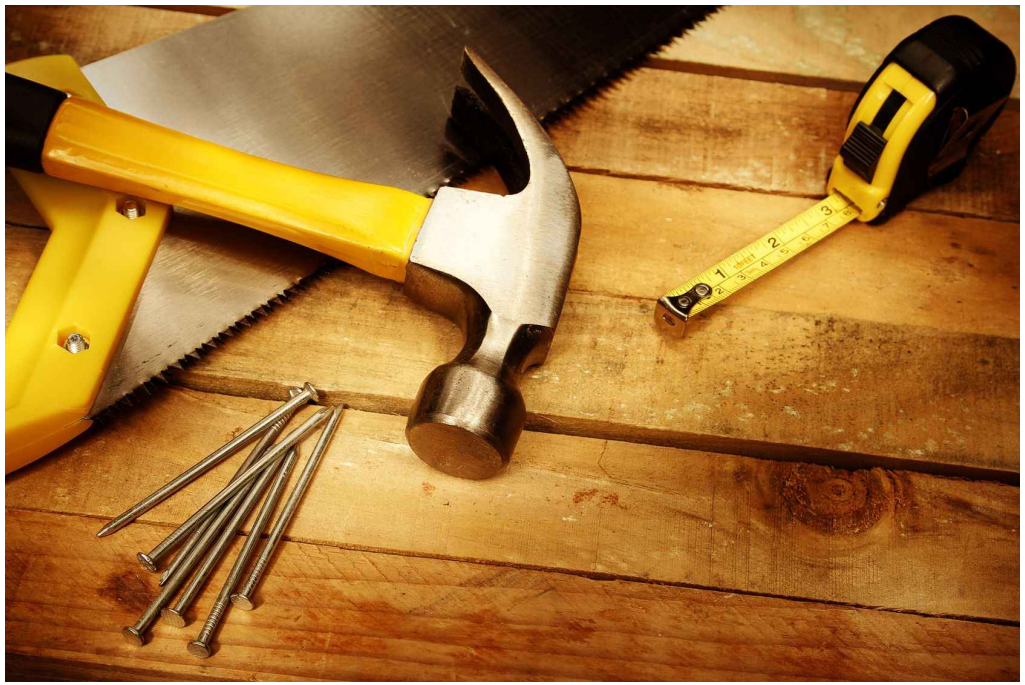
Class `FundFactory` parameterized using reflection

```
class FundFactory {
    public InvestmentFund createInvestmentFund(String type, int balance) {
        InvestmentFund fund = null;
        try {
            fund = (InvestmentFund) Class.forName("investment_funds."
                                                  + type).newInstance();
            fund.setBalance(balance);
        } catch (Exception e) {
            throw new IllegalArgumentException();
        }
        return fund;
    }
}
```

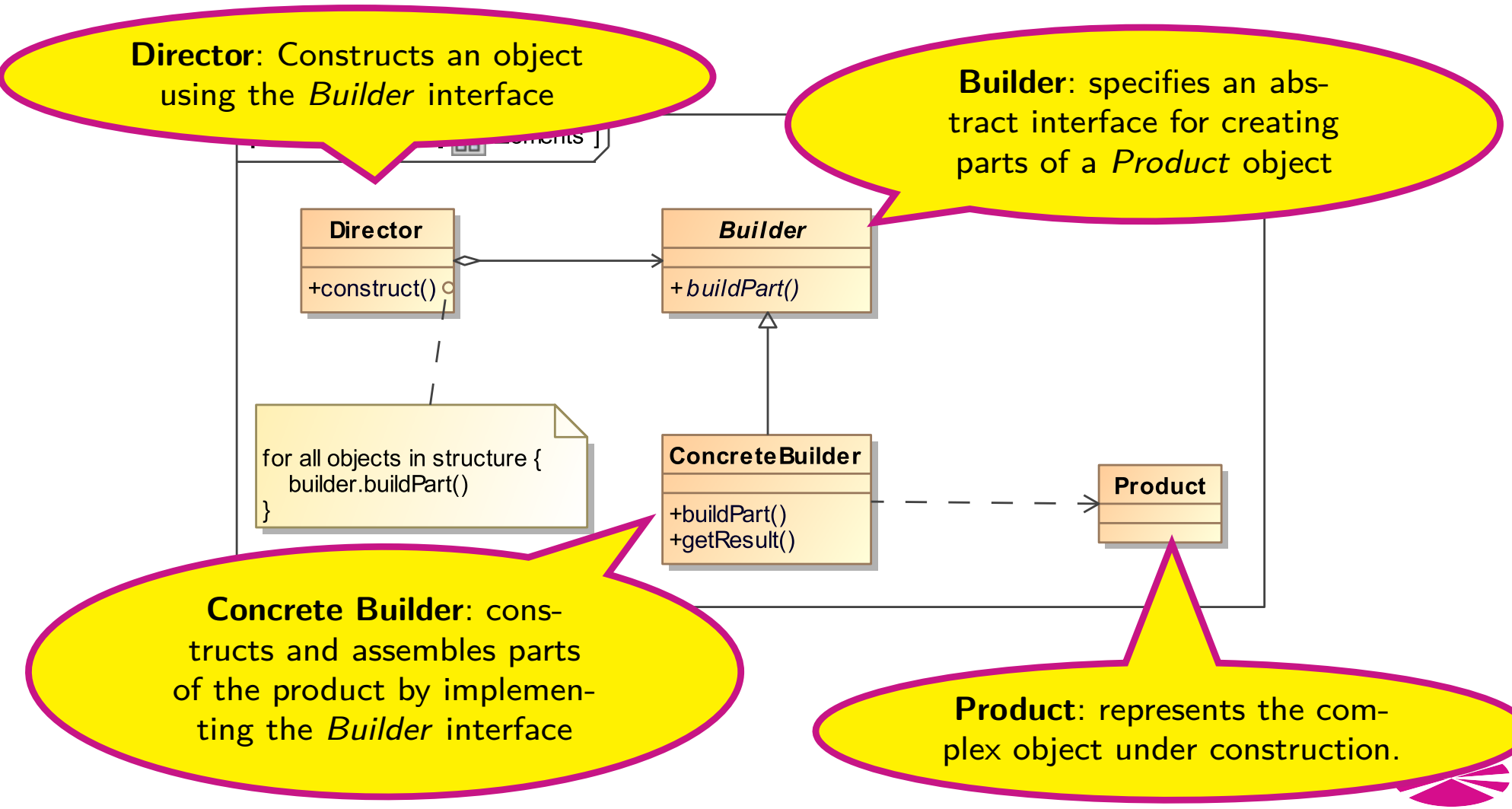
Builder pattern

Builder pattern

Separates the construction of a complex object from its representation so that the same construction process can create different representations.



Builder pattern elements



Telescoping constructor antipattern

- The traditional version of the builder pattern **emphasizes the possibility of using polymorphism to create different types of objects**.
  - The *Director* has the plan of how to build the object and uses a *builder* to build the different parts.
  - The *Director* doesn't know what *Concrete Builder* is using. Different *Concrete Builders* will allow the `Director` to build different versions of the same *Product*.
- There is another simpler version of the constructor pattern, **which emphasizes the possibility of constructing an object by parts** and not so much the construction of an object in a polymorphic way.
  - The idea is to build an object by putting together its constituent parts and not by using its constructor.
  - In Java, this approach is commonly used to solve the **Telescopic Constructor** antipattern.

Telescoping constructor antipattern

Constructor telescópico

A telescopic constructor occurs in those classes that overload the constructor to allow for different ways of creating them. Each new version of the constructor adds new parameters to the previous one by stretching the constructor parameters like a pirate telescope.

Telescoping constructor antipattern

- **Example**: Let's imagine a class that represents the nutritional aspects of a product called `NutritionFacts`.
- This class will have two **mandatory parameters**:
  - Serving size (`servingSize`).
  - Number of rations or servings included (`servings`).
- The **optional parameters** will be:
  - The `calories`.
  - The `fat`.
  - The `sodium`.
  - The `carbohydrate`.

Telescoping constructor antipattern

Class `NutritionFacts` (1/2)

```
public class NutritionFacts {
private final int servingSize;  // (mL) required
private final int servings;     // (per container) required
private final int calories;     // optional
private final int fat;          // (g) optional
private final int sodium;       // (mg) optional
private final int carbohydrate; // (g) optional

public NutritionFacts(int servingSize, int servings) {
    this(servingSize, servings, 0);
}
public NutritionFacts(int servingSize, int servings, int calories) {
    this(servingSize, servings, calories, 0);
}
public NutritionFacts(int servingSize,int servings,int calories,int fat) {
    this(servingSize, servings, calories, fat, 0);
}
public NutritionFacts(int servingSize, int servings, int calories,int fat,
                      int sodium) {
    this(servingSize, servings, calories, fat, sodium, 0);
}
```

Telescoping constructor antipattern

Clase `NutritionFacts` (2/2)

```
    public NutritionFacts(int servingSize, int servings, int calories,
                          int fat, int sodium, int carbohydrate) {
        this.servingSize = servingSize;
        this.servings = servings;
        this.calories = calories;
        this.fat = fat;
        this.sodium = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

Telescoping constructor antipattern

Building a `NutritionFacts` object

```
public static void main(String[] args) {
    NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
}
```

- **Problems**
  - Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won't complain, but the program will misbehave at runtime.
  - This constructor invocation will require many parameters that you don't want to set, but you're forced to pass a value for them anyway. There is no option for entering the `sodium` value but not entering the `fat` value.
  - Conclusion: the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.

Solution: Use the JavaBeans pattern

- *JavaBeans* are Java components.
- Basically they are characterized by:
  - Having a parameterless constructor.
  - Having private attributes.
  - These attributes are accessed by getter and setter methods that follow the (`get/set`) notation.

Solution: Use the JavaBeans pattern

Class `NutritionFacts` as a JavaBean

```
public class NutritionFacts {
    private int servingSize = -1; // Required; no default value
    private int servings = -1;    //      "       "      "      "
    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;
    public NutritionFacts() { }

    public void setServingSize(int val) { servingSize = val; }
    public void setServings(int val) { servings = val; }
    public void setCalories(int val) { calories = val; }
    public void setFat(int val) { fat = val; }
    public void setSodium(int val) { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

Solution: Use the JavaBeans pattern

Building a `NutritionFacts` JavaBean

```
public static void main(String[] args) {
    NutritionFacts cocaCola = new NutritionFacts();
    cocaCola.setServingSize(240);
    cocaCola.setServings(8);
    cocaCola.setCalories(100);
    cocaCola.setSodium(35);
    cocaCola.setCarbohydrate(27);
}
```

Solution: Use the JavaBeans pattern

- **Problems**
  - More code to write when creating an instance.
  - A JavaBean may be in an inconsistent state partway through its construction.
  - Precludes the possibility of making a class immutable.
- **Solution**
  - Use the *builder* pattern instead.

Builder pattern example

- The use of the Builder pattern in `NutritionFacts` is a concrete application of the pattern that attempts to solve the problem of telescopic builders.
- In this case there is no abstract builder, the product is `NutritionFacts` and the concrete builder is the `Builder` class, that is an inner class of `NutritionFacts`.
- `Director` class will be any class that creates the object using the builder.

Solution: Use the Builder pattern

Class `NutritionFacts` (1/3)

```
public final class NutritionFacts {

    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    private NutritionFacts(Builder builder) { // private
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }

    //...
```

Solution: Use the Builder pattern

Class `Builder` as an inner class of `NutritionFacts` (2/3)

```
    public static class Builder { // Inner class
        // Required parameters
        private final int servingSize;
        private final int servings;

        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;

        public Builder(int servingSize, int servings) { // Constructor
            this.servingSize = servingSize;
            this.servings = servings;
        }

    // ...
}
```

Solution: Use the Builder pattern

Class `Builder` as inner class of `NutritionFacts` (3/3)

```
        public Builder calories(int val) {  // Plays the build part role
            calories = val;  return this;
        }
        public Builder fat(int val) {
            fat = val; return this;
        }
        public Builder sodium(int val) {
            sodium = val; return this;
        }
        public Builder carbohydrate(int val) {
            carbohydrate = val;  return this;
        }
        public NutritionFacts build() { // Plays the obtain result role
            return new NutritionFacts(this);
        }
    } // End of Builder
} // End of de NutritionFacts
```

Solution: Use the Builder pattern

Class `Director`

```
public class Director {
    public static void main(String[] args) {
        NutritionFacts cocaCola = new NutritionFacts.Builder(240,8).
                  calories(100).sodium(35).carbohydrate(27).build();
        System.out.println(cocaCola);
    }
}
```

Builder pattern

- We have created an object using the expressive approach of JavaBeans but without its disadvantages
- The resulting class is immutable.
- The Builder pattern simulates named optional parameters as found in Python.

Named optional parameters in Python.

```
def info(object, spacing=10, collapse=1):

info(odbchelper)      # spacing and collapse use the default values
info(odbchelper, 12) # spacing is 12 and collapse is the default value
info(odbchelper, collapse=0) # collapse is 0 and spacing the default
                             # value
info(spacing=15, object=odbchelper) # Named parameters can be
                                    # in any order
```

Template Method

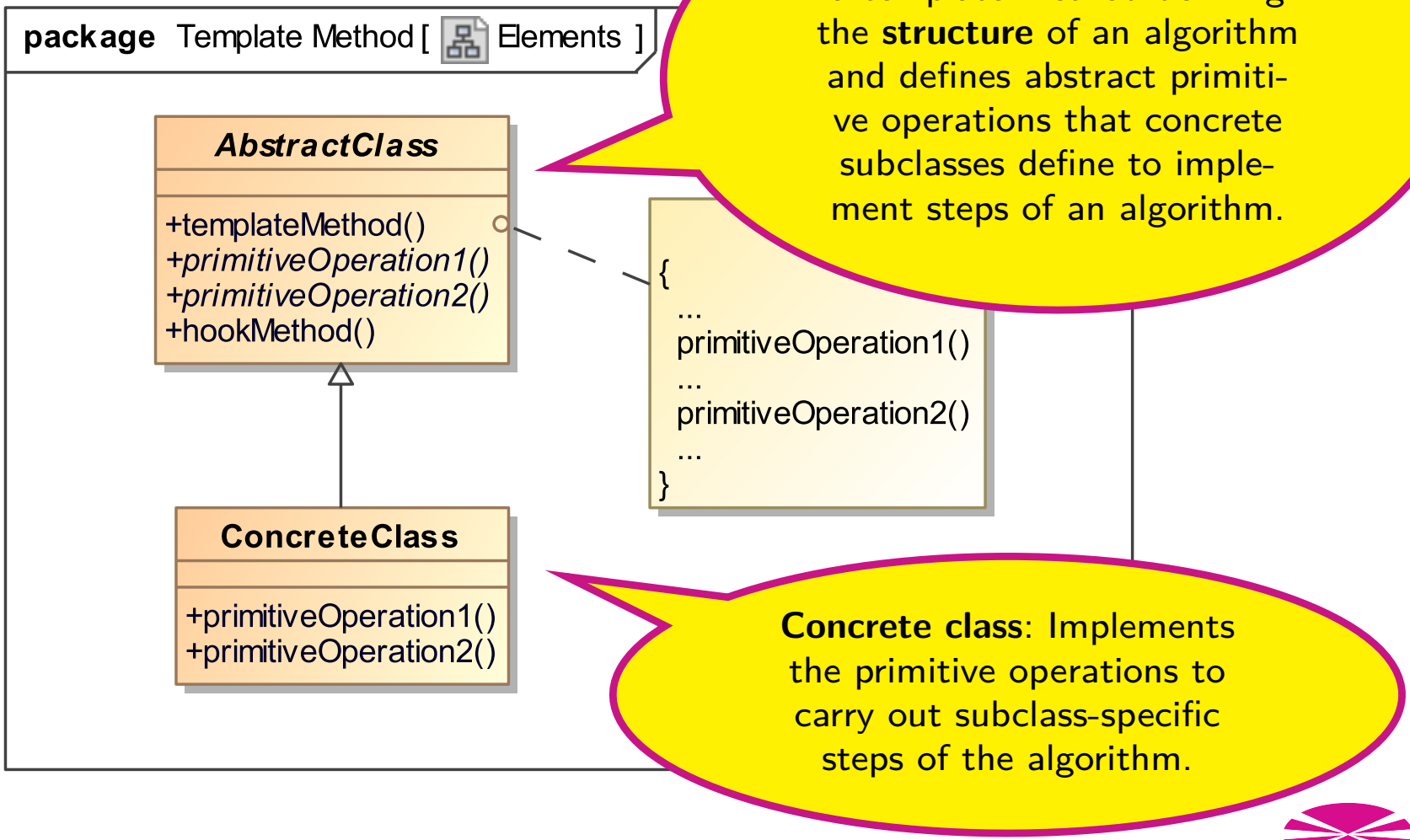
Template Method

Defines the structure of an algorithm in an operation, deferring some steps to subclasses.

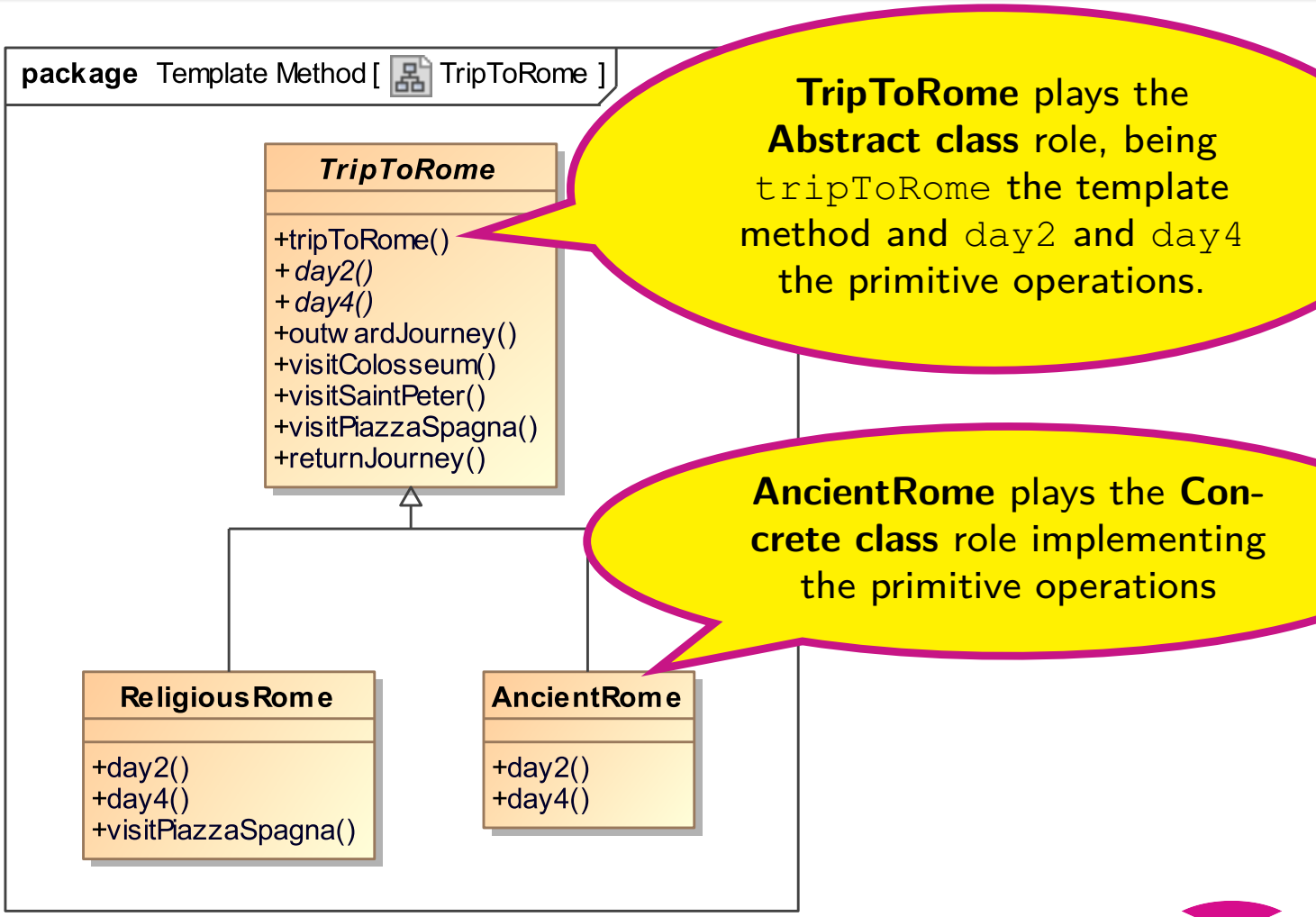
Template Method Characteristics

- Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- It is called template method because using it is like filling a template or form.

Template Method pattern elements



Template Method pattern example



Template Method pattern example

Class `TripToRome` (1/2)

```
public abstract class TripToRome {
    // Template method
    public final void tripToRome() {
        outwardJourney();
        visitColosseum();
        day2();  // Primitive operation
        visitSaintPeter();
        day4();  // Primitive operation
        visitPiazzaSpagna();
        returnJourney();
    }

    public abstract void day2();
    public abstract void day4();

    // ...
```

The template method defines the basic structure of a trip to Rome and cannot been overridden (as it is `final`).

Some steps are deferred to the subclasses.

Template Method pattern example

Class `TripToRome` (2/2)

Other methods have an implementation, although they can be overridden by the subclasses (hook methods).

```
    // ...
    public void outwardJourney() {
        System.out.println("Outward journey...");
    }

    public void visitColosseum() {
        System.out.println("Visiting Colosseum...");
    }

    public void visitSaintPeter() {
        System.out.println("Visiting Saint Peter...");
    }

    public void visitPiazzaSpagna() {
        System.out.println("Visiting Piazza Spagna...");
    }

    public void returnJourney() {
        System.out.println("Return journey...");
    }
}
```

Template Method pattern example

Class `ReligiousRome`

```
public class ReligiousRome extends TripToRome {
    @Override
    public void day2() {
        System.out.println("Visiting Castel Sant'Angelo...");
    }

    @Override
    public void day4() {
        System.out.println("Visiting the Basilica of St. Paul outside the
                            Walls...");
    }

    @Override
    public void visitPiazzaSpagna() {
        System.out.println("Visiting the churches of Piazza del Popolo");
    }
}
```

Primitive operations are implemented.

A hook method can be redefined to give it a more suitable implementation.

Template Method pattern example

Class `AncientRome`

```
public class AncientRome extends TripToRome {

@Override
    public void day2() {
        System.out.println("Visiting the Roman Forum...");
    }

    @Override
    public void day4() {
        System.out.println("Visiting Trajan's Market...");
    }
}
```

Template Method pattern vs. Strategy pattern

- **Similarities**
  - Both of them are used to implement several types of similar algorithms.
- **Differences**
  - Template methods use inheritance to vary part of an algorithm.
  - Strategies use delegation to vary the entire algorithm.

The Hollywood Principle

The Hollywood Principle

"Don't call us, we'll call you".

The Hollywood Principle

- **Characteristics**
  - Low-level components and high-level components participate in the computation.
  - Low-level components hook themselves into a system, but the high-level components determine when they are needed, and how.
  - A low-level component never calls a high-level component directly.
  - It is an *inversion of control* technique similar to the injection of dependencies.
- **Examples**
  - The Template Method pattern is a good example of "The Hollywood Principle".
  - The `sort` methods from the Java Collections API delegate the comparison part of the algorithm to the `compareTo` methods of the objects being compared.

DRY (Don't Repeat Yourself) principle

DRY (Don't Repeat Yourself) principle

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

DRY (Don't Repeat Yourself) principle

- The principle was formulated by **Andy Hunt** and **Dave Thomas** in the book **The Pragmatic Programmer**.
- **Adding additional, unnecessary code to a codebase increases the amount of work required to extend and maintain the software in the future**.
- Whether the duplication stems from *Copy Paste Programming* or poor understanding of how to apply abstraction, **it decreases the quality of the code**.
- **Violations of DRY are typically referred to as WET solutions**, which is commonly taken to stand for either "*write everything twice*", "*we enjoy typing*" or "*waste everyone's time*".

DRY (Don't Repeat Yourself) examples at code level

- **Magic numbers**.
  - Do not use *magic numbers* in your code, use constants instead.
- **Method abstraction**:
  - If you need to write the same piece of code in many places, don't copy-and-paste it, make it a separate method and call that method wherever it is required.
- **Generalization of behavior**:
  - If the subclasses share a behavior, factor it out to a common base class.
- **Encapsulate what varies**:
  - If an object has many possible variations in its behavior and you have to copy and paste "if-then" or "switch-case" structures to represent them, inject these behaviors using polymorphism and dynamic binding (e.g., the Strategy pattern).

DRY (Don't Repeat Yourself) principle

- **DRY can be applied to other levels**:
  - Duplication in process is also a waste if it can be automated. Manual testing, manual build and integration processes, etc., should all be eliminated whenever possible through the use of automation.
  - Normalization processes in database schemas.
  - Documentation, etc.
- **Related principles**:
  - The Open/Closed Principle only works in practice when DRY is followed.
  - The Single Responsibility Principle relies on DRY.

KISS and YAGNI principles

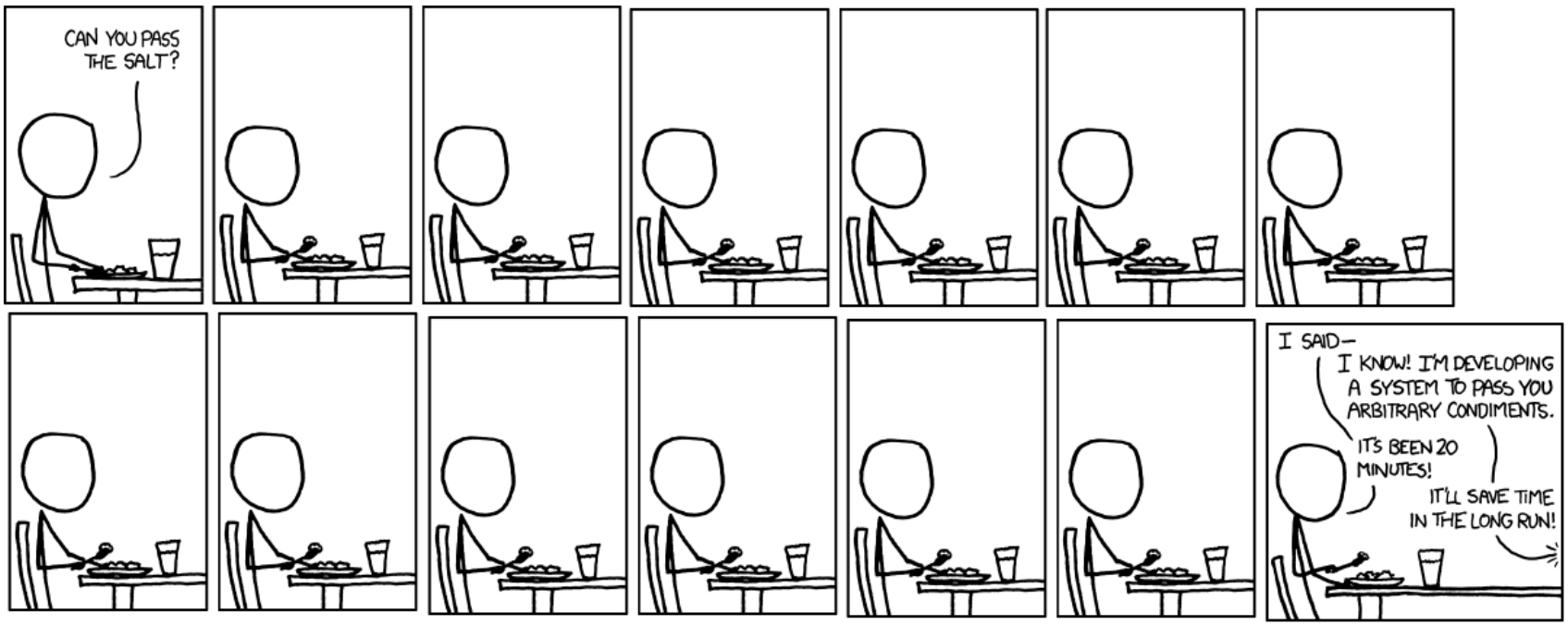
KISS principle "Keep It Simple, Stupid"

Simplicity should be a key goal in design and **unnecessary** complexity should be avoided.

KISS and YAGNI principles

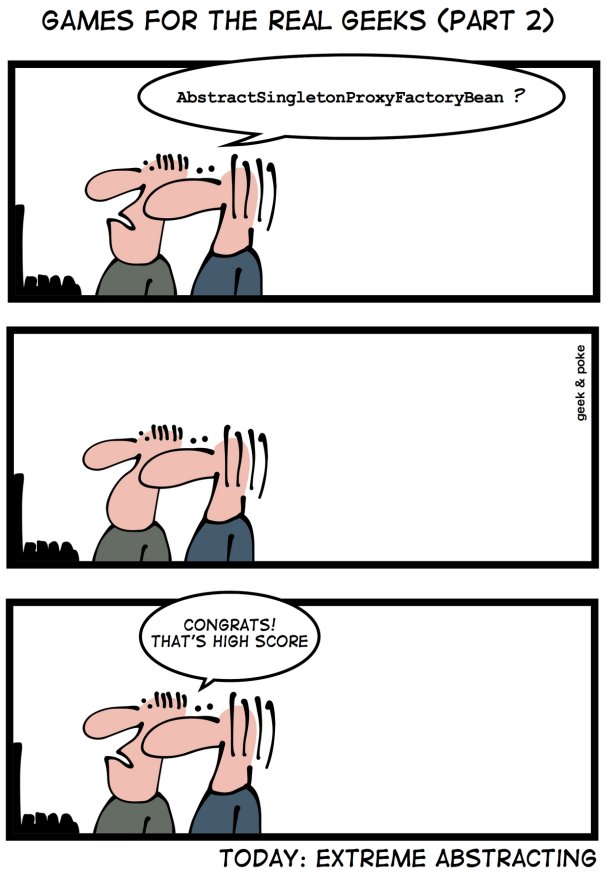
YAGNI principle "*You Aren't Gonna Need It*"

Always implement things when you actually need them, never when you just foresee that you need them.



Over-Engineering

- **Over-Engineering**
  - Developing a more complex code than what is needed.
  - It is done to anticipate future needs, but "*It is difficult to make predictions, especially about the future*".
  - If the predictions are incorrect we are wasting time developing unnecessary code.
  - It makes it difficult for people to understand your code. There's some piece built into the system that doesn't really need to be there, and the person reading the code can't figure out why it's there, or even how the whole system works.



No joke! See: http://docs.spring.io/spring/docs/2.5.x/api/org/springframework/aop/framework/AbstractSingletonProxyFactoryBean.html
(Thanks to Andreas Rummler for the idea!!!!)

Over-Engineering

- Implementing `Hello World` using the **Strategy**, **Factory** and **Singleton** patterns.

Class `HelloWorld`

```
class HelloWorld {
    public static void main(String[] args) {
        MessageBody mb = new MessageBody();
        mb.configure("Hello World!");
        AbstractStrategyFactory asf
                = DefaultFactory.getInstance();
        MessageStrategy strategy = asf.createStrategy(mb);
        mb.send(strategy);
    }
}
```

Over-Engineering

Class MessageBody

```
class MessageBody {
    private Object payload;
    public Object getPayload() { return payload; }
    public void configure(Object obj) { payload = obj;  }
    public void send(MessageStrategy ms) { ms.sendMessage(); }
}
```

Class MessageStrategy

```
interface MessageStrategy { public void sendMessage(); }
```

Class AbstractStrategyFactory

```
abstract class AbstractStrategyFactory {
    public abstract MessageStrategy createStrategy(MessageBody mb);
}
```

Over-Engineering

Class `DefaultFactory`

```
class DefaultFactory extends AbstractStrategyFactory {
    private static DefaultFactory instance;
    private DefaultFactory() {}

    public static AbstractStrategyFactory getInstance() {
        if (instance == null) { instance = new DefaultFactory(); }
        return instance;
    }
    @Override
    public MessageStrategy createStrategy(final MessageBody mb) {
        return new MessageStrategy() {
            MessageBody body = mb;

            @Override
            public void sendMessage() {
                Object obj = body.getPayload();
                System.out.println((String) obj);
            }
        };
    }
}
```

Under-Engineering

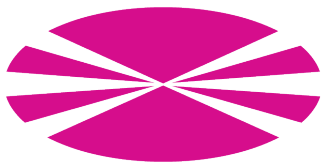
- **Under-Engineering**
  - Poorly designed code. It is a more common situation than over-engineering.
  - Causes: not enough time to develop, unclear requirements, lack of knowledge on how to develop a better design, etc.
  - A poor design that "works" will never be improved ("If it ain't broke, don't fix it").
  - As time goes by and developers add new functionalities, poorly designed code becomes more complicated to correct.
  - At the end the developers cry for a redesign.

Unit 6: Design Patterns

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science and Information Technology
Faculty of Computer Science



UNIVERSIDADE DA CORUÑA