How To Avoid Java’s instanceof Operator?

Bern, 15 February 2021

Raoul Adler

Definition

Java **instanceof** Operator…

*…is used to **test whether the object is an instance of the specified type** (class or subclass or interface).*

*..it is also known as **type comparison operator** because it compares the instance with type. ...*

In A Pure OO Model

*the **use of the instanceof operator** is definitely a*

General Rule

Avoid **instanceof** whenever possible

Why?

- is a symptom of a bad/poor design
- makes code extensions difficult
- makes your code procedural
- can end up in a runtime exception

But

A common exception to this rule is the use of instanceof within an equals method.

Real Life Example

What does this code?

Depending on a type of the Standarddokument interface’s implementation, we choose a way of creating a “Bemerkung”.

In case we don't find the type that matches, we just throw an exception.

Take a Look at the Code…

Do you like it?

What’s wrong here? (1)

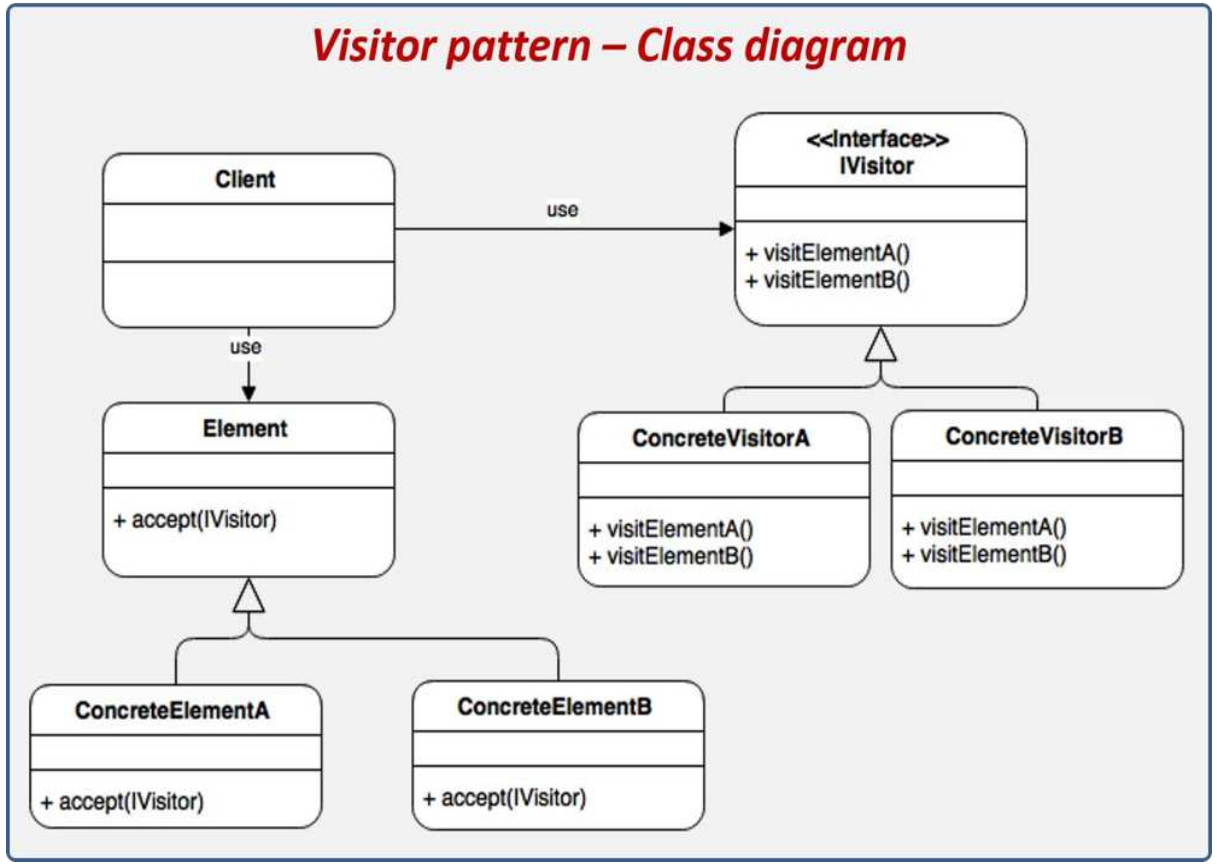
- Method is very long.
- If-Else hell.
- Method body does not depend on its input parameter.
- Casting in method body into a specific type
  → results in an increased number of dependencies.

What’s wrong here? (2)

- A new type is not so easy to add.
- Using an exception to signalize the need of support of new functionality is a misusing of exceptions
  → could end up in a runtime exception.

A way to get rid of instanceof

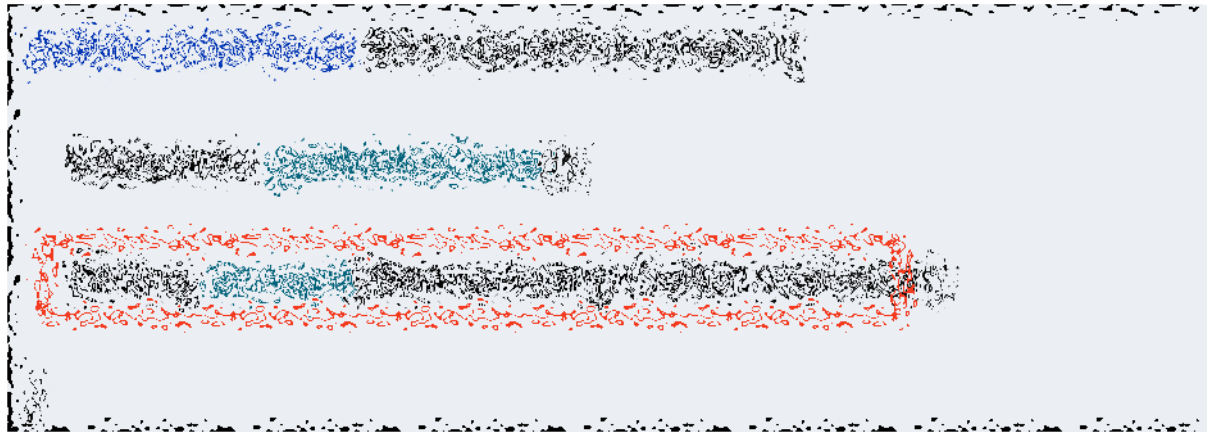
Solution: **Visitor Pattern**



CSS Versicherung

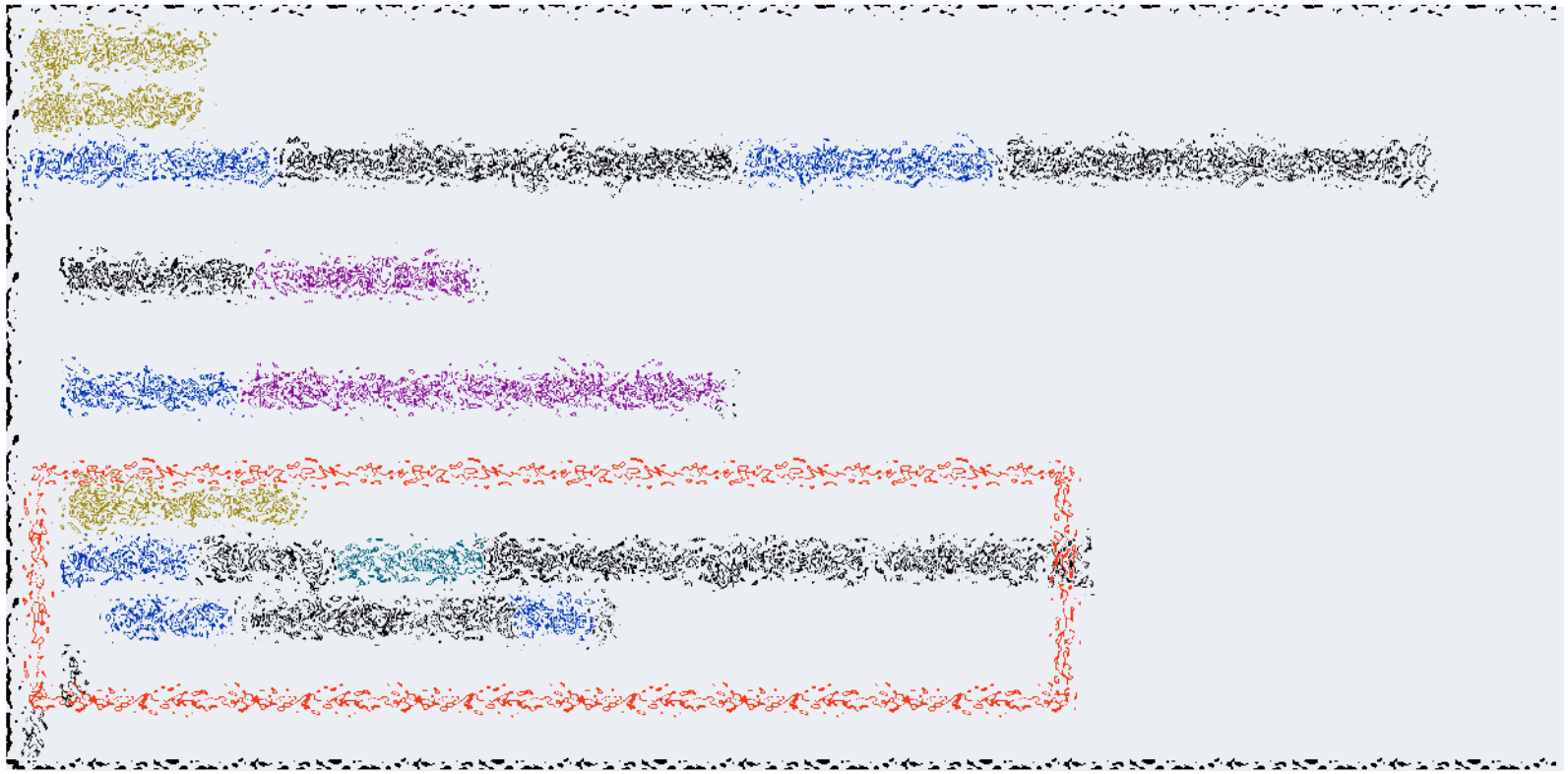
Applying The Pattern (Step 1)

Add a method to the Standarddokument interface which allows to pass a visitor:



Applying The Pattern (Step 2)

An implementation of the Standarddokument interface now looks like this:



Applying The Pattern (Step 3)

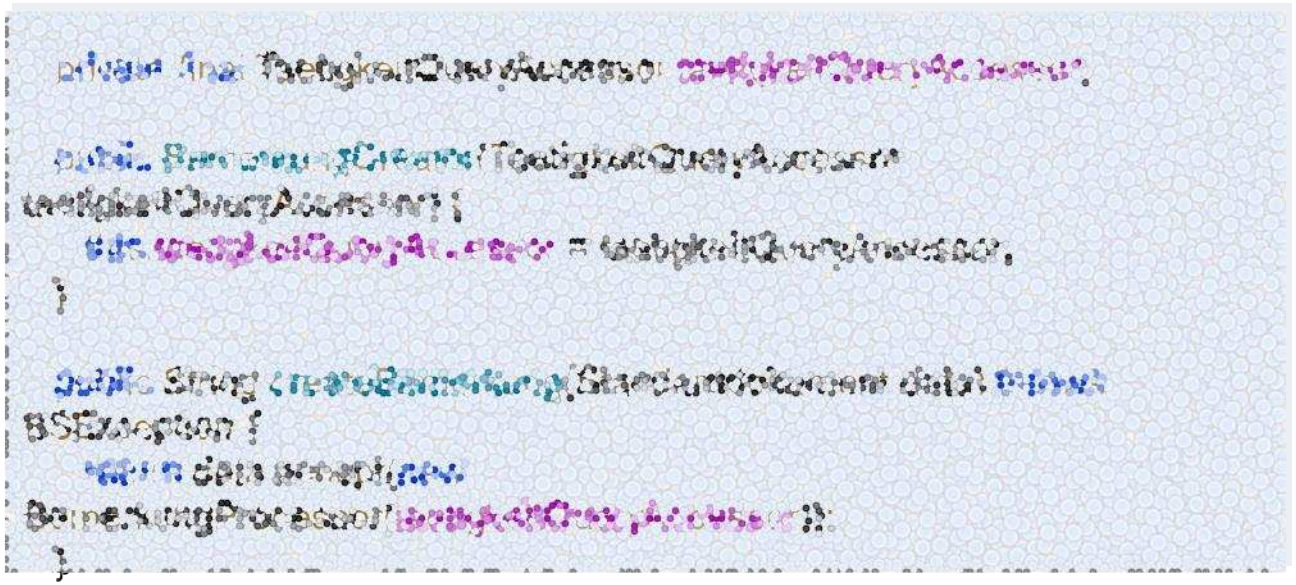
1. The Visitor interface:

2. Its implementation:

Applying The Pattern (Final Step)

Here is the refactored **BemerkungCreator** class:

```
public class BemerkungCreator {
}
```



Applying The Pattern (Conclusion)

- **The exception is no longer needed** → the required support for newly introduced implementation would be signaled by non-compiling code.

- **Single Responsibility Principle** comes into play → each specific implementation of Visitor interface is responsible only for one functionality.

- **Design is behavior-oriented** (interfaces), not implementation-oriented (instanceof + casting). In this way, we are hiding implementation details.

- **Open Closed Principle:** Design is open for extensions →It is really easy to introduce new functionality which implementation differ for specific objects.

CSS Versicherung


THANK YOU
FOR YOUR
ATTENTION