

04834580 Software Engineering (Honor Track) 2025-26

# Creational & Structural Design Patterns

Sergey Mechtaev

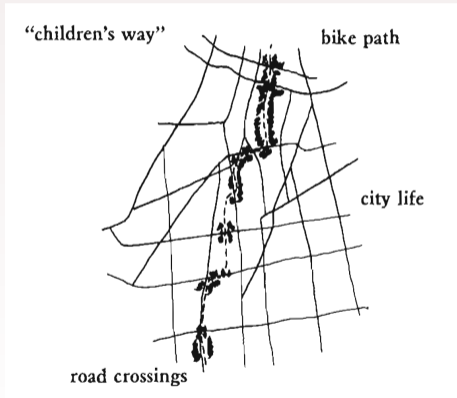
[mechtaev@pku.edu.cn](mailto:mechtaev@pku.edu.cn)

School of Computer Science, Peking University



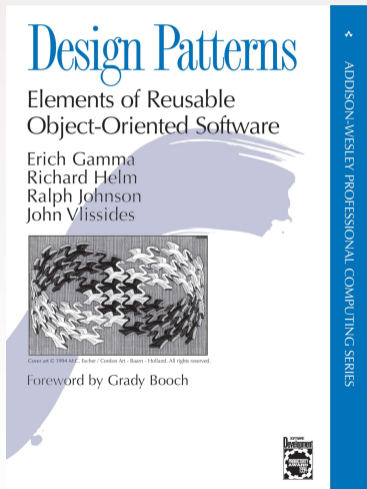
In “A Pattern Language” [1],  
Christopher Alexander wrote

*Each patterns describes a problem  
which occurs over and over again in  
our environment, and then describes  
the core of the solution to that prob-  
lem, in such a way that you can use  
this solution a million times over...*



The “Gang of Four” [2] introduced patterns for software design:

- ▶ Creational:
  - ▶ Singleton
  - ▶ Factory
  - ▶ Builder
  - ▶ ...
- ▶ Structural:
  - ▶ Adapter
  - ▶ Decorator
  - ▶ ...
- ▶ Behavioral:
  - ▶ Template Method
  - ▶ Iterator
  - ▶ Visitor
  - ▶ Observer
  - ▶ ...



## Definition ([2])

Ensure a class only has one instance, and provide a global point of access to it.

Erich Gamma:

“I’m in favor of dropping Singleton. Its use is almost always a design smell”.

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

```
public final class Singleton {
    private static volatile Singleton instance = null;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

```
public final class Singleton {
    private static volatile Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Rust does not allow directly using global mutable variables.

```
use std::sync::Mutex;

static ARRAY: Mutex<Vec<i32>> = Mutex::new(Vec::new());

fn do_a_call() {
    ARRAY.lock().unwrap().push(1);
}

fn main() {
    do_a_call();
    do_a_call();
    do_a_call();

    let array = ARRAY.lock().unwrap();
    println!("Called {} times: {:?}", array.len(), array);
    drop(array);

    *ARRAY.lock().unwrap() = vec![3, 4, 5];

    println!("New singleton object: {:?}", ARRAY.lock().unwrap());
}
```

## Definition ([2])

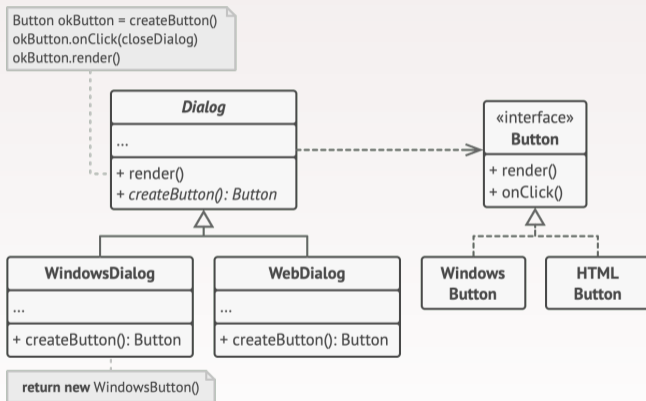
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

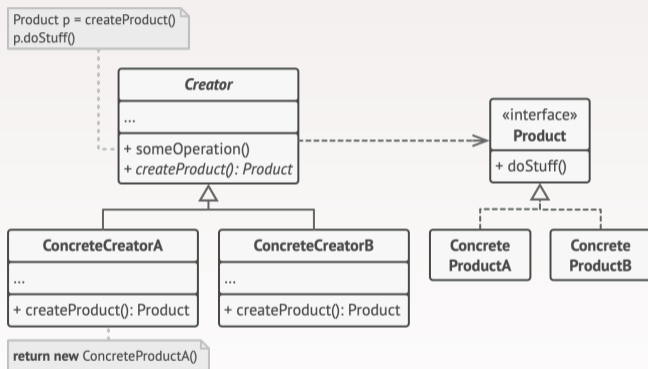
Example: different types of dialogs (Windows, Mac, HTML, etc) require their own types of elements. That's why we create a subclass for each dialog type and override their factory methods.

```
public abstract class Dialog {  
  
    public void renderWindow() {  
        // ... other code ...  
  
        Button okButton = createButton();  
        okButton.render();  
    }  
  
    public abstract Button createButton();  
}
```

Now, each dialog type will instantiate proper button classes. Base dialog works with products using their common interface, that's why its code remains functional after all changes.

```
public class HtmlDialog extends Dialog {  
  
    @Override  
    public Button createButton() {  
        return new HtmlButton();  
    }  
}
```





## Definition ([2])

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

In this example, the Builder pattern allows step by step construction of different car models.

```
public interface Builder {  
    void setCarType(CarType type);  
    void setSeats(int seats);  
    void setEngine(Engine engine);  
    void setTransmission(Transmission transmission);  
    void setTripComputer(TripComputer tripComputer);  
    void setGPSNavigator(GPSNavigator gpsNavigator);  
}
```

```
public class CarBuilder implements Builder {
    private CarType type;
    private int seats;
    private Engine engine;
    private Transmission transmission;
    private TripComputer tripComputer;
    private GPSNavigator gpsNavigator;

    @Override
    public void setSeats(int seats) {
        this.seats = seats;
    }

    @Override
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    ...
}
```

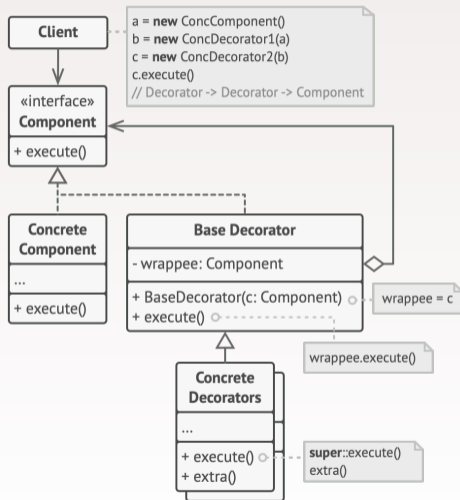
```
public class CarBuilder implements Builder {  
  
    ...  
  
    public Car getResult() {  
        return new Car(type, seats, engine, transmission, tripComputer, gpsNavigator);  
    }  
}
```

```
Command::new("git")
...
.subcommand(
    Command::new("diff")
        .about("Compare two commits")
        .arg(arg!(base: [COMMIT]))
        .arg(arg!(head: [COMMIT]))
        .arg(arg!(path: [PATH]).last(true))
        .arg(
            arg!(--color <WHEN>)
                .value_parser(["always", "auto", "never"])
                .num_args(0..=1)
                .require_equals(true)
                .default_value("auto")
                .default_missing_value("always"),
        ),
    ),
)
```

Check examples: <https://github.com/clap-rs/clap/tree/master/examples>

## Definition ([2])

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



## Definition ([2])

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

```
public class RoundHole {
    private double radius;

    public RoundHole(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public boolean fits(RoundPeg peg) {
        boolean result;
        result = (this.getRadius() >= peg.getRadius());
        return result;
    }
}
```



RoundPegs are compatible with RoundHoles:

```
public class RoundPeg {
    private double radius;

    public RoundPeg() {}

    public RoundPeg(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }
}
```

SquarePegs are **incompatible** with RoundHoles:

```
public class SquarePeg {
    private double width;

    public SquarePeg(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }

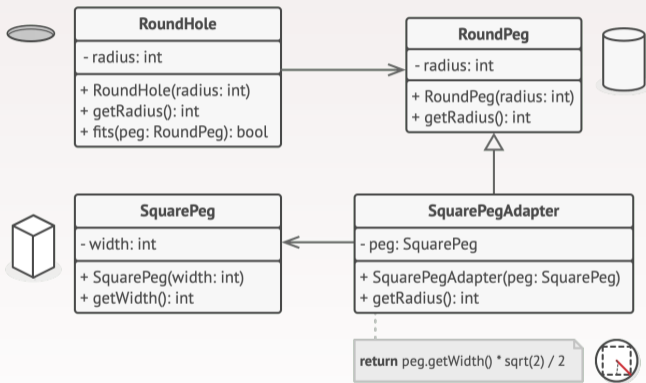
    public double getSquare() {
        double result;
        result = Math.pow(this.width, 2);
        return result;
    }
}
```

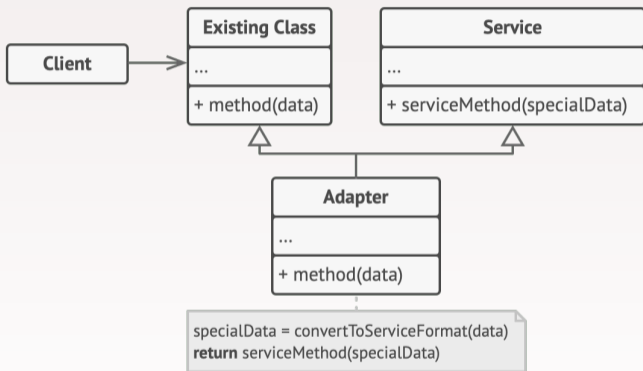
This adapter allows fitting square pegs into round holes:

```
public class SquarePegAdapter extends RoundPeg {
    private SquarePeg peg;

    public SquarePegAdapter(SquarePeg peg) {
        this.peg = peg;
    }

    @Override
    public double getRadius() {
        double result;
        // Calculate a minimum circle radius, which can fit this peg.
        result = (Math.sqrt(Math.pow((peg.getWidth() / 2), 2) * 2));
        return result;
    }
}
```





- [1] Christopher Alexander.  
*A pattern language: towns, buildings, construction.*  
Oxford university press, 1977.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.  
*Design patterns: elements of reusable object-oriented software.*  
Pearson Deutschland GmbH, 1995.