

Builder Pattern

in Flutter 

Design Patterns für moderne App-Entwicklung



Einstiegsproblem [1]

Der einfache Konstruktor

```
class Burger {  
    // Zutaten  
    String _bun;  
    String _patty;  
    bool _cheese;  
  
    // Konstruktor  
    Burger(this._bun, this._patty, this._cheese);  
  
    // Setter  
    set bun(String bun) => _bun = bun;  
  
    // Getter  
    String getDescription() {  
        return 'Burger mit $_bun Brot, $_patty Patty,'  
            ' ${_cheese ? "mit" : "ohne"} Käse';  
    }  
}
```

Einstiegsproblem [1]

Der einfache Konstruktor

```
void main() {  
  
    var burger = Burger('Sesam', 'Rind', false);  
    //                                ^cheese  
  
    burger.patty = 'Veggie';  
}
```

Einstiegsproblem [1]

Telescoping Constructor Pattern (Anti-Pattern)

```
class Burger {  
    // Zutaten  
    final String _bun;  
    final String _patty;  
    final bool _cheese;  
    final bool _sauce;  
    final bool _onion;  
  
    // Konstruktor  
    Burger(  
        this._bun,  
        this._patty,  
        this._cheese,  
        this._sauce,  
        this._onion,  
    );  
}
```

Einstiegsproblem [1]

Telescoping Constructor Pattern (Anti-Pattern)

```
void main() {  
  
    var burger = Burger('Sesam', 'Rind', true);  
    //                                     ^cheese  
  
    var burger2 = Burger('Sesam', 'Rind', true, false, true);  
    //                                     ^sauce, ^onion  
  
    var special400 = Burger('Sesam', 'Rind', true, true, true,  
        false, true, false, false, true, false, true, false, 'Mais',  
        'Oliven', false, true, 'BBQ', false, 5, 2, true,  
        false, true, 'Honig', true); //      ^ớt-Schoten 🥒  
  
    print(burger.getDescription());  
}
```

Einstiegsproblem [1]

Der Subclass-Ansatz

```
class Burger { /* Basisklasse */ }

class Cheeseburger extends Burger {
    // Konstruktor der Basisklasse aufrufen
    Cheeseburger() : super('Sesam', 'Rind', true, false, false);
}

class VeggieBurger extends Burger {
    VeggieBurger() : super('Vollkorn', 'Gemüse', false, true, true);
}

class SwabBurger extends Burger {
    SwabBurger() : super('Brioche', 'Käsespätzle', true, true, true);
}
```

Builder Pattern

"... separates the construction of a complex object from its representation." [2]

^Produkt-Klasse

Vergleich [1]

Vorher: Ohne Builder Pattern

Nachher: Mit Builder Pattern

Problem: Parameter unklar, unflexibel, nicht schön

Lösung: Selbstdokumentierend!

BurgerBuilder [1]

Verwendung (main)

```
void main() {  
    var burgerBuilder = BurgerBuilder('Brioche','Rind');  
    burgerBuilder.setCheese();  
    var burger = burgerBuilder.build();  
}
```

Builder-Klasse

```
class BurgerBuilder {  
    // Felder  
    final String _bun;  
    final String _patty;  
    bool _cheese = false;  
    bool _pickles = false;  
  
    // Konstruktor  
    BurgerBuilder(  
        this._bun,  
        this._patty,  
    );  
  
    // ...  
}
```

BurgerBuilder - void Setter [1]

Verwendung (main)

```
void main() {  
    var burgerBuilder = BurgerBuilder('Brioche','Rind');  
    burgerBuilder.setCheese();  
    var burger = burgerBuilder.build();  
}
```

Setter-Methoden

```
class BurgerBuilder {  
    // ...  
    bool _cheese = false;  
    // ...  
  
    void setCheese() {  
        _cheese = true;  
    }  
  
    void setPickles() {  
        _pickles = true;  
    }  
  
    // ...  
}
```

BurgerBuilder - build() [1]

Verwendung (main)

```
void main() {  
    var burgerBuilder = BurgerBuilder('Brioche','Rind');  
    burgerBuilder.setCheese();  
    var burger = burgerBuilder.build();  
}
```

build()-Methode

```
class BurgerBuilder {  
    final String _bun;  
    final String _patty;  
    bool _cheese = false;  
    bool _pickles = false;  
  
    // BurgerBuilder(...)  
  
    void setCheese() {  
        _cheese = true;  
    }  
  
    // void setPickles() {...}  
  
    Burger build() {  
        return Burger(_bun, _patty, _cheese, _pickles);  
    }  
}
```

Immutable vs. Mutable [3]

Mutable (veränderbar)

```
class Burger {  
    // Private Attribute  
  
    // Builder Parameter werden hier übergeben  
    Burger(this._bun, this._patty,  
           this._cheese, this._pickles);  
  
    // Public Setter - Objekt veränderbar  
    void setCheese(bool value) {  
        _cheese = value;  
    }  
  
    void setPickles(bool value) {  
        _pickles = value;  
    }  
  
    // Getter...  
}
```

Immutable (unveränderbar)

```
class Burger {  
    // Private Attribute  
  
    Burger(this._bun, this._patty,  
           this._cheese, this._pickles);  
  
    // Setter entfernt!  
  
    // Getter...  
}
```

Objekt kann nach Erstellung nicht geändert werden!

Objekt kann nach Erstellung geändert werden

Void vs Method Chaining [3]

Vorher: Mit void Settern

```
void main() {  
    var burgerBuilder = BurgerBuilder('Brioche','Rind');  
    burgerBuilder.setCheese();  
    var burger = burgerBuilder.build();  
}
```

Nachher: Mit Method Chaining

```
void main() {  
    var burger = BurgerBuilder('Brioche','Rind')  
        .setCheese()  
        .build();  
}
```

Funktioniert, aber viel Schreibarbeit

Kürzer und eleganter!

Method Chaining - Setter [3]

Verwendung (main)

```
void main() {  
    var burger = BurgerBuilder('Brioche', 'Rind')  
        .setCheese()  
        .build();  
}
```

Setter-Methoden

```
class BurgerBuilder {  
    // ...  
    bool _cheese = false;  
    // ...  
  
    BurgerBuilder setCheese() {  
        _cheese = true;  
        return this;  
    }  
  
    BurgerBuilder setPickles() {  
        _pickles = true;  
        return this;  
    }  
  
    // ...  
}
```

Method Chaining - build() [3]

Verwendung (main)

```
void main() {  
    var burger = BurgerBuilder('Brioche', 'Rind')  
        .setCheese()  
        .build();  
}
```

build()-Methode

```
class BurgerBuilder {  
    final String _bun;  
    final String _patty;  
    bool _cheese = false;  
    bool _pickles = false;  
  
    // BurgerBuilder(...)  
  
    BurgerBuilder setCheese() {  
        _cheese = true;  
        return this;  
    }  
  
    // BurgerBuilder setPickles() {...}  
  
    Burger build() {  
        return Burger(_bun, _patty, _cheese, _pickles);  
    }  
}
```

Director - Optional [2]

Ohne Director

```
void main() {  
    var burger1 = BurgerBuilder('Brioche', 'Rind')  
        .setCheese()  
        .setPickles()  
        .build();  
  
    var burger2 = BurgerBuilder('Brioche', 'Rind')  
        .setCheese()  
        // Gürkchen vergessen!  
        .build();  
}
```

Mit Director

```
void main() {  
    var director = BurgerDirector();  
  
    var burger1 = director.makeFullyLoaded();  
    var burger2 = director.makeCheeseLover();  
}
```

- Client Code muss Rezept kennen
- Reihenfolge nicht garantiert
- Code-Duplikation

Director - Optional [2]

Mit Director

```
void main() {  
  
    var director = BurgerDirector();  
    var burger = director.makeFullyLoaded();  
  
}
```

Client bekommt Burger - muss Details nicht kennen!



Quelle: facebook.com/spongebob

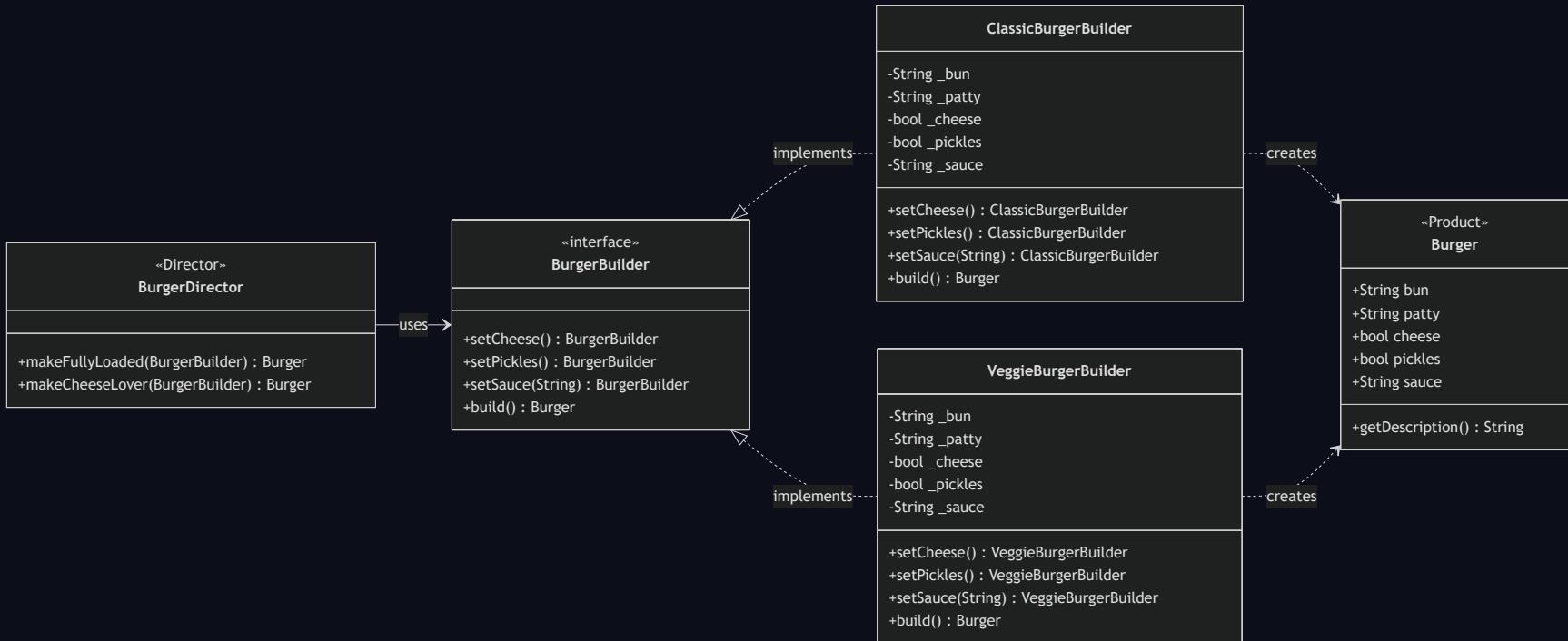
Live-Demo



Scan Me

github.com/Bachelor-MMB7/builder-pattern-demo

Klassendiagramm [4]



Vorteile und Nachteile [3]

Vorteile

Garantierte Immutability

Objekt kann nach der Erstellung nicht geändert werden

Bessere Lesbarkeit

Keine überfüllten Konstruktoren oder Parameter-Verwechslung.

Trennung Konstruktion / Repräsentation

Gleicher Prozess kann unterschiedliche Objekte erzeugen.

Nachteile

Zusätzliche Klasse

Builder-Klasse bedeutet mehr Code und Overhead.

Code-Duplizierung

Builder enthält dieselben Felder wie das Produkt.

Unnötige Komplexität

Overkill für einfache Objekte. Mögliche Performance-Nachteile.

Verwandt zu Factory Pattern [2, 5]

Builder Pattern

Konstruiert komplexe Objekte **Schritt für Schritt**

Produkt wird erst bei `build()` zurückgegeben

Erlaubt zusätzliche Konstruktionsschritte vor der Rückgabe

Abstract Factory

Erstellt feste Produkt-Kombinationen **in einem Schritt**

Gibt das Produkt **sofort** zurück

Verhindert falsche Kombinationen

Gemeinsamkeit: Beide sind **Creational Patterns** – sie kapseln die Objekterstellung

Quellen

Referenzen und weiterführende Links

- [1] A. Shvets, "Builder," *Refactoring.Guru*, 2014. Online. Available: <https://refactoring.guru/design-patterns/builder>
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995.
- [3] O. Musch, *Design Patterns mit Java: Eine Einführung*. Wiesbaden: Springer Vieweg, 2021.
- [4] GeeksforGeeks, "Builder Design Pattern," *GeeksforGeeks*, Dec. 1, 2025. Online. Available: <https://www.geeksforgeeks.org/system-design/builder-design-pattern/>
- [5] F. Kiwy, "Exploring Joshua Bloch's Builder design pattern in Java," *Oracle Java Magazine*, May 28, 2021. Online. Available: https://intojava.wordpress.com/wp-content/uploads/2021/06/exploring_joshua_bloch_s_builder_design_pattern_in_java.pdf

Abbildungen

Titelbild: Erstellt mit Google Gemini (2025)

Diskussion

Wann braucht man ein Interface?

Wenn du mehrere Builder mit unterschiedlichen Implementierungen haben willst, die aber austauschbar sein sollen.

Warum nicht einfach Konstruktor + Setter der Produktklasse nutzen um ein Objekt zu erstellen? Damit spar ich mir doch den Builder? (JavaBeans Pattern [3])

Unfertige Objekte können versehentlich genutzt werden → Fehler. Lösung: Setter entfernen + Felder `final` → Objekt sofort immutable.

Habt ihr das Pattern schon mal verwendet?