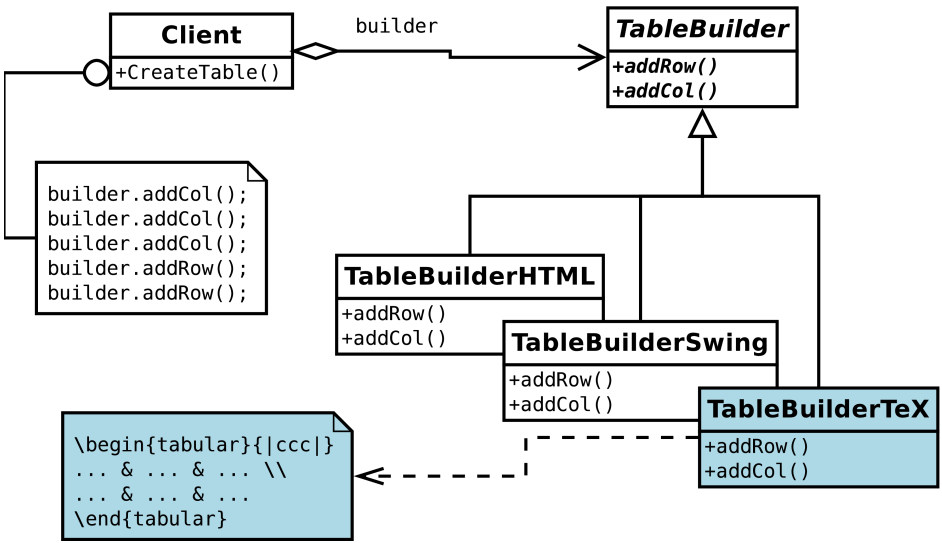
Builder (budowniczy)

Cel:

Oddzielenie konstruowania złożonego obiektu od jego reprezentacji, tak aby ten sam proces konstrukcji mógł tworzyć różne reprezentacje.

Przykład:

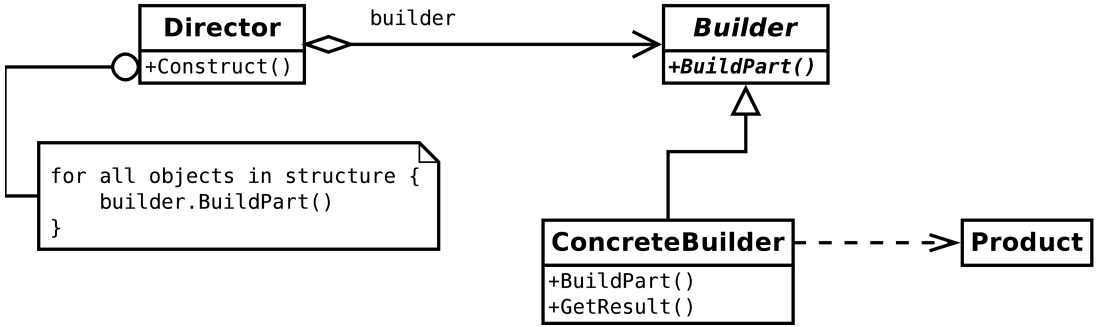


```
abstract class TableBuilder {
     public abstract void addRow();
     public abstract void addCol();
}
class TableBuilderHTML extends TableBuilder {
     public void addRow() { /* ... */ }
     public void addCol() { /* ... */ }
     public String getTable() { /* ... */ }
}
// ... Client
     builder = new TableBuilderHTML();
     builder.addRow(); // ...
     builder.addCol(); // ...
     // ...
     String table = builder.getTable();
```

Zastosowanie:

- Gdy algorytm tworzenia złożonego obiektu powinien być niezależny od części, które składają się na ten obiekt i sposobu ich składania
- Gdy proces konstrukcji powinien pozwalać na różne reprezentacje tworzonego obiektu

Struktura:



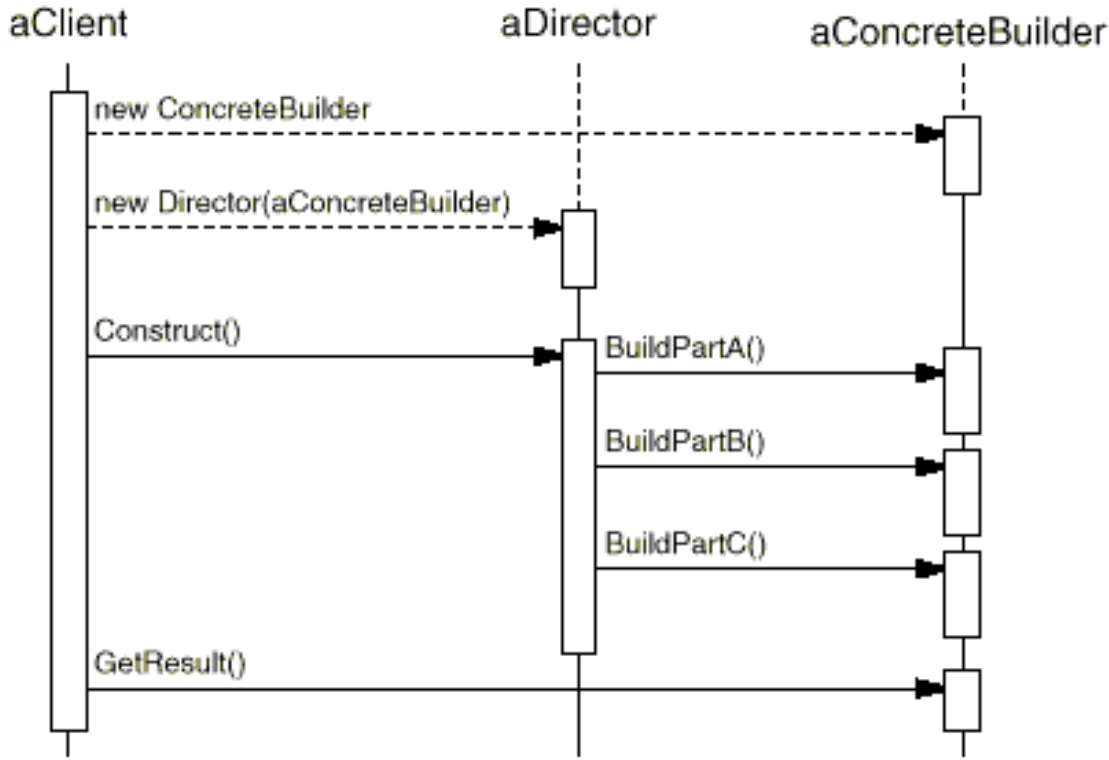
Składniki:

- Builder – deklaruje abstrakcyjny interfejs tworzenia części produktu
- ConcreteBuilder – tworzy i składa części produktu, implementując interfejs Buildera; definiuje i przechowuje tworzoną reprezentację produktu; udostępnia interfejs pozwalający na odebranie stworzonego produktu
- Director – konstruuje obiekt przy pomocy interfejsu Buildera
- Product – reprezentuje złożony obiekt, który konstruujemy; wewnętrzną reprezentację produktu definiuje ConcreteBuilder, on też definiuje proces jego składania; product przeważnie składa się z klas reprezentujących jego części składowe

Zależności:

- Klient tworzy obiekt Directora i konfiguruje go konkretnym rodzajem Buildera.
- Director kieruje do Buildera żądania tworzenia kolejnych części produktu.
- Builder wykonuje te żądania: tworzy części i do daje je do produktu.
- Klient odbiera od Buildera gotowy produkt.

```
class Director {
     public Director(TableBuilder builder) { /* ... */ }
     public void createTable(int rows, int cols) {
          for(int i = 0; i < rows; ++i)
               builder.addRow();
          for(int i = 0; i < cols; ++i)
               builder.addCol();
     }
}
// ... Client
     TableBuilderSwing builder = new
TableBuilderSwing();
     Director director = new Director(builder);
     director.createTable(5, 3);
     JComponent table = builder.GetResult();
```


aClient
aDirector
aConcreteBuilder
new ConcreteBuilder
new Director(aConcreteBuilder)
Construct()
BuildPartA()
BuildPartB()
BuildPartC()
GetResult()

Konsekwencje:

1. umożliwia zmianę wewnętrznej reprezentacji produktu – builder dostarcza directorowi abstrakcyjny interfejs do tworzenia produktu, ukrywając przed nim reprezentację i wewnętrzną strukturę, oraz to, w jaki sposób jest on składany; aby zmienić wewnętrzną reprezentację produktu wystarczy zdefiniować nowy rodzaj buildera;
2. izoluje kod konstrukcji i reprezentacji – poprawia modularność hermetyzując sposób tworzenia złożonego obiektu – klient nie musi wiedzieć jakie klasy tworzą wewnętrzną strukturę produktu, gdyż nie są one widoczne w interfejsie buildera; każdy konkretny builder zawiera cały kod potrzebny do stworzenia i złożenia danego rodzaju produktu, director może używać raz napisanego kodu do konstrukcji innych wariantów produktu z tych samych kawałków;
3. daje bardziej precyzyjną kontrolę nad procesem konstrukcji – w przeciwieństwie do wzorców tworzących produkt w jednym ruchu, builder tworzy go krok po kroku, pod kontrolą directora – produkt jest zwracany dopiero po skończeniu produkcji, zatem interfejs buildera lepiej przedstawia proces tworzenia obiektu i daje nad nim pełniejszą kontrolę;

Implementacja:

Zazwyczaj mamy abstrakcyjnego buildera, który definiuje operację dla każdego komponentu, o którego stworzenie może prosić director (domyślnie pustą) – konkretny builder przesłania operacje tworzące interesujące go komponenty. A poza tym:

1. interfejs tworzenia i składania – jak konstruować produkty? tworzymy je krok po kroku, zatem interfejs buildera musi być wystarczająco uniwersalny, by pozwolić na tworzenie produktów wszystkim rodzajom konkretnych builderów; czasem wystarczy po kolei tworzyć kolejne fragmenty i składać je ze sobą (np. tekst formatowany), wtedy interfejs tworzenia produktu jest prosty, ale czasem potrzebujemy odwołań do części już stworzonych (grafy, drzewa) – wówczas builder będzie zwracał jakieś węzły dla directora, a ten przekaże je dalej w kolejnych wywołaniach;

```
class TreeBuilder {
     public Node addNode(Node parent, String text) {
          Node node = new Node();
          node.text = text; node.parent = parent;
          return node; }
}
class TreeDirector {
     private TreeBuilder builder;
     public TreeDirector(TreeBuilder b) { builder = b; }
     public void construct() {
          Node root = builder.addNode(null, "root");
          builder.addNode(root, "A");
          builder.addNode(root, "B");
          builder.addNode(root, "C");
}    }
```

2. dlaczego nie ma abstrakcyjnej klasy produktu? – gdyż konkretne produkty różnią się tak bardzo, że nic byśmy nie zyskali dając im wspólny interfejs; ponieważ to klient konfiguruje directora konkretnym builderem, wie on który builder jest w użyciu i jak się nim zająć poprawnie;

```
class WindowBuilderGUI extends WindowBuilder {
     public void addButton( /* ... */ ) { /* ... */ }
     public void addLabel( /* ... */ ) { /* ... */ }
     public JFrame getProduct() { /* ... */ }
}
class WindowBuilderImage extends WindowBuilder {
     public void addButton( /* ... */ ) { /* ... */ }
     public void addLabel( /* ... */ ) { /* ... */ }
     public Image getProduct() { /* ... */ }
}
```

3. domyślnie puste metody w builderze (a nie abstrakcyjne) – aby klient mógł przesłonić tylko te operacje, których potrzebuje;

Powiązania:

- Abstract Factory również może konstruować złożone produkty, ale Builder skupia się na ich tworzeniu krok-po-kroku, zaś Abstract Factory na wydzieleniu osobnych rodzin produktów. Builder zwraca produkt dopiero w ostatnim kroku produkcji.
- Builder zazwyczaj zajmuje się tworzeniem Kompozytów (Composite).