helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku

Autorzy: Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides
Tłumaczenie: Tomasz Walczak
ISBN: 978-83-246-2662-5
Tytuł oryginału: Design Patterns: Elements of Reusable Object-Oriented Software
Format: 180×235, stron: 376

Naucz się wykorzystywać wzorce projektowe i ułatw sobie pracę!

- Jak wykorzystać projekty, które już wcześniej okazały się dobre?
- Jak stworzyć elastyczny projekt obiektowy?
- Jak sprawnie rozwiązywać typowe problemy projektowe?

Projektowanie oprogramowania obiektowego nie jest łatwe, a przy założeniu, że powinno ono nadawać się do wielokrotnego użytku, staje się naprawdę skomplikowane.
Aby stworzyć dobry projekt, najlepiej skorzystać ze sprawdzonych i efektywnych rozwiązań, które wcześniej były już stosowane. W tej książce znajdziesz właśnie najlepsze doświadczenia z obszaru programowania obiektowego, zapisane w formie wzorców projektowych gotowych do natychmiastowego użycia!

W książce „Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku" opisano, czym są wzorce projektowe, a także w jaki sposób pomagają one projektować oprogramowanie obiektowe. Podręcznik zawiera studia przypadków, pozwalające poznać metody stosowania wzorców w praktyce. Zamieszczono tu również katalog wzorców projektowych, podzielony na trzy kategorie: wzorce konstrukcyjne, strukturalne i operacyjne. Dzięki temu przewodnikowi nauczysz się skutecznie korzystać z wzorców projektowych, ulepszać dokumentację i usprawniać konserwację istniejących systemów. Krótko mówiąc, poznasz najlepsze sposoby sprawnego opracowywania niezawodnego projektu.

- Wzorce projektowe w architekturze MVC
- Katalog wzorców projektowych
- Projektowanie edytora dokumentów
- Wzorce konstrukcyjne, strukturalne i operacyjne
- Dziedziczenie klas i interfejsów
- Określanie implementacji obiektów
- Obsługa wielu standardów wyglądu i działania
- Zastosowanie mechanizmów powtórnego wykorzystania rozwiązania

Wykorzystaj zestaw konkretnych narzędzi do programowania obiektowego!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2010

SPIS TREŚCI

Przedmowa **9**

Wstęp **11**

Przewodnik dla Czytelników **13**

Rozdział 1. Wprowadzenie **15**
1.1. Czym jest wzorzec projektowy? 16
1.2. Wzorce projektowe w architekturze MVC w języku Smalltalk 18
1.3. Opisywanie wzorców projektowych 20
1.4. Katalog wzorców projektowych 22
1.5. Struktura katalogu 24
1.6. Jak wzorce pomagają rozwiązać problemy projektowe? 26
1.7. Jak wybrać wzorzec projektowy? 42
1.8. Jak stosować wzorce projektowe? 43

Rozdział 2. Studium przypadku — projektowanie edytora dokumentów **45**
2.1. Problemy projektowe 45
2.2. Struktura dokumentu 47
2.3. Formatowanie 52
2.4. Ozdabianie interfejsu użytkownika 55
2.5. Obsługa wielu standardów wyglądu i działania 59
2.6. Obsługa wielu systemów okienkowych 63
2.7. Działania użytkowników 69
2.8. Sprawdzanie pisowni i podział słów 74
2.9. Podsumowanie 86

Rozdział 3. Wzorce konstrukcyjne **87**
BUDOWNICZY (BUILDER) 92
FABRYKA ABSTRAKCYJNA (ABSTRACT FACTORY) 101
METODA WYTWÓRCZA 110
PROTOTYP (PROTOTYPE) 120
SINGLETON (SINGLETON) 130

Omówienie wzorców konstrukcyjnych 137

Rozdział 4. Wzorce strukturalne ... 139
ADAPTER (ADAPTER) ... 141
DEKORATOR (DECORATOR) ... 152
FASADA (FACADE) ... 161
KOMPOZYT (COMPOSITE) ... 170
MOST (BRIDGE) ... 181
PEŁNOMOCNIK (PROXY) ... 191
PYŁEK (FLYWEIGHT) ... 201

Omówienie wzorców strukturalnych ... 213

Rozdział 5. Wzorce operacyjne ... 215
INTERPRETER (INTERPRETER) ... 217
ITERATOR (ITERATOR) ... 230
ŁAŃCUCH ZOBOWIĄZAŃ (CHAIN OF RESPONSIBILITY) ... 244
MEDIATOR (MEDIATOR) ... 254
METODA SZABLONOWA (TEMPLATE METHOD) ... 264
OBSERWATOR (OBSERVER) ... 269
ODWIEDZAJĄCY (VISITOR) ... 280
PAMIĄTKA (MEMENTO) ... 294
POLECENIE (COMMAND) ... 302
STAN (STATE) ... 312
STRATEGIA (STRATEGY) ... 321

Omówienie wzorców operacyjnych ... 330

Rozdział 6. Podsumowanie ... 335
6.1. Czego można oczekiwać od wzorców projektowych? ... 335
6.2. Krótka historia ... 339
6.3. Społeczność związana ze wzorcami ... 340
6.4. Zaproszenie ... 342
6.5. Słowo na zakończenie ... 342

Dodatek A Słowniczek ... 343

Dodatek B Przewodnik po notacji ... 347
B.1. Diagram klas ... 347
B.2. Diagram obiektów ... 349
B.3. Diagram interakcji ... 350

Dodatek C Klasy podstawowe ... 351
C.1. List ... 351
C.2. Iterator ... 354
C.3. ListIterator ... 354
C.4. Point ... 355
C.5. Rect ... 355

Bibliografia ... 357

Skorowidz ... 363

ROZDZIAŁ 3.

Wzorce konstrukcyjne

Konstrukcyjne wzorce projektowe pozwalają ująć w abstrakcyjnej formie proces tworzenia egzemplarzy klas. Pomagają zachować niezależność systemu od sposobu tworzenia, składania i reprezentowania obiektów. Klasowe wzorce konstrukcyjne są oparte na dziedziczeniu i służą do modyfikowania klas, których egzemplarze są tworzone. W obiektowych wzorcach konstrukcyjnych tworzenie egzemplarzy jest delegowane do innego obiektu.

Wzorce konstrukcyjne zyskują na znaczeniu wraz z coraz częstszym zastępowaniem w systemach dziedziczenia klas składaniem obiektów. Powoduje to, że programiści kładą mniejszy nacisk na trwałe zapisywanie w kodzie określonego zestawu zachowań, a większy — na definiowanie mniejszego zbioru podstawowych działań, które można połączyć w dowolną liczbę bardziej złożonych zachowań. Dlatego tworzenie obiektów o określonych zachowaniach wymaga czegoś więcej niż prostego utworzenia egzemplarza klasy.

We wzorcach z tego rozdziału powtarzają się dwa motywy. Po pierwsze, wszystkie te wzorce kapsułkują informacje o tym, z których klas konkretnych korzysta system. Po drugie, ukrywają proces tworzenia i składania egzemplarzy tych klas. System zna tylko interfejsy obiektów zdefiniowane w klasach abstrakcyjnych. Oznacza to, że wzorce konstrukcyjne dają dużą elastyczność w zakresie tego, *co* jest tworzone, *kto* to robi, *jak* przebiega ten proces i *kiedy* ma miejsce. Umożliwiają skonfigurowanie systemu z obiektami-produktami o bardzo zróżnicowanych strukturach i funkcjach. Konfigurowanie może przebiegać statycznie (w czasie kompilacji) lub dynamicznie (w czasie wykonywania programu).

Niektóre wzorce konstrukcyjne są dla siebie konkurencją. Na przykład w niektórych warunkach można z pożytkiem zastosować zarówno wzorzec Prototyp (s. 120), jak i Fabryka abstrakcyjna (s. 101). W innych przypadkach wzorce się uzupełniają. We wzorcu Budowniczy (s. 92) można wykorzystać jeden z pozostałych wzorców do określenia, które komponenty zostaną zbudowane, a do zaimplementowania wzorca Prototyp (s. 120) można użyć wzorca Singleton (s. 130).

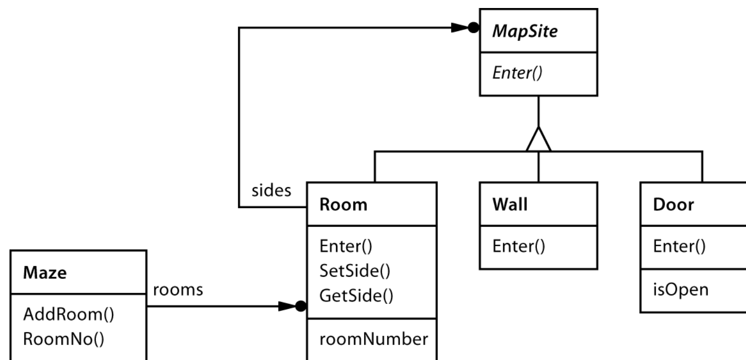
Ponieważ wzorce konstrukcyjne są mocno powiązane ze sobą, przeanalizujemy całą ich piątkę razem, aby podkreślić podobieństwa i różnice między nimi. Wykorzystamy też jeden przykład do zilustrowania implementacji tych wzorców — tworzenie labiryntu na potrzeby gry komputerowej. Labirynt i gra będą nieco odmienne w poszczególnych wzorcach. Czasem celem gry będzie po prostu znalezienie wyjścia z labiryntu. W tej wersji gracz prawdopodobnie będzie

widział tylko lokalny fragment labiryntu. Czasem w labiryntach trzeba będzie rozwiązać problemy i poradzić sobie z zagrożeniami. W tych odmianach można udostępnić mapę zbadanego już fragmentu labiryntu.

Pominiemy wiele szczegółów dotyczących tego, co może znajdować się w labiryncie i czy gra jest jedno-, czy wieloosobowa. Zamiast tego skoncentrujemy się na tworzeniu labiryntów. Labirynt definiujemy jako zbiór pomieszczeń. Każde z nich ma informacje o sąsiadach. Mogą to być następne pokoje, ściana lub drzwi do innego pomieszczenia.

Klasy `Room`, `Door` i `Wall` reprezentują komponenty labiryntu używane we wszystkich przykładach. Definiujemy tylko fragmenty tych klas potrzebne do utworzenia labiryntu. Ignorujemy graczy, operacje wyświetlania labiryntu i poruszania się po nim oraz inne ważne funkcje nieistotne przy generowaniu labiryntów.

Poniższy diagram ilustruje relacje między wspomnianymi klasami:



Każde pomieszczenie ma cztery strony. W implementacji w języku C++ do określania stron północnej, południowej, wschodniej i zachodniej służy typ wyliczeniowy `Direction`:

```
enum Direction {North, South, East, West};
```

W implementacji w języku Smalltalk kierunki te są reprezentowane za pomocą odpowiednich symboli.

`MapSite` to klasa abstrakcyjna wspólna dla wszystkich komponentów labiryntu. Aby uprościć przykład, zdefiniowaliśmy w niej tylko jedną operację — `Enter`. Jej działanie zależy od tego, gdzie gracz wchodzi. Jeśli jest to pomieszczenie, zmienia się lokalizacja gracza. Jeżeli są to drzwi, mogą zajść dwa zdarzenia — jeśli są otwarte, gracz przejdzie do następnego pokoju, a o zamknięte drzwi użytkownik rozbije sobie nos.

```
class MapSite {
public:
  virtual void Enter() = 0;
};
```

`Enter` to prosty podstawowy element bardziej złożonych operacji gry. Na przykład jeśli gracz znajduje się w pomieszczeniu i zechce pójść na wschód, gra może ustalić, który obiekt `MapSite` znajduje się w tym kierunku, i wywołać operację `Enter` tego obiektu. Operacja `Enter` specyficzna

dla podklasy określi, czy gracz zmienił lokalizację czy rozbił sobie nos. W prawdziwej grze operacja `Enter` mogłaby przyjmować jako argument obiekt reprezentujący poruszającego się gracza.

`Room` to podklasa konkretna klasy `MapSite` określająca kluczowe relacje między komponentami labiryntu. Przechowuje referencje do innych obiektów `MapSite` i numer pomieszczenia (numery te służą do identyfikowania pokojów w labiryncie).

```
class Room : public MapSite {
public:
  Room(int roomNo);

  MapSite* GetSide(Direction) const;
  void SetSide(Direction, MapSite*);

  virtual void Enter();

private:
  MapSite* _sides[4];
  int _roomNumber;
};
```

Poniższe klasy reprezentują ścianę i drzwi umieszczone po dowolnej stronie pomieszczenia.

```
class Wall : public MapSite {
public:
  Wall();

  virtual void Enter();
};

class Door : public Mapsite {
public:
  Door(Room* = 0, Room* = 0);

  virtual void Enter();
  Room* OtherSideFrom(Room*);

private:
  Room* _room1;
  Room* _room2;
  bool _isOpen;
};
```

Potrzebne są informacje nie tylko o częściach labiryntu. Zdefiniujemy też klasę `Maze` reprezentującą kolekcję pomieszczeń. Klasa ta udostępnia operację `RoomNo`, która znajduje określony pokój po otrzymaniu jego numeru.

```
class Mase {
public:
  Maze();

  void AddRoom(Room*);
```

```
    Room* RoomNo(int) const;
  private:
    // ...
  };
```

Operacja `RoomNo` może znajdować pomieszczenia za pomocą wyszukiwania liniowego, tablicy haszującej lub prostej tablicy. Nie będziemy jednak zajmować się takimi szczegółami. Zamiast tego skoncentrujmy się na tym, jak określić komponenty obiektu `Maze`.

Następną klasą, jaką zdefiniujemy, jest `MazeGame`. Służy ona do tworzenia labiryntu. Prostym sposobem na wykonanie tego zadania jest użycie serii operacji dodających komponenty do labiryntu i łączących je. Na przykład poniższa funkcja składowa utworzy labirynt składający się z dwóch pomieszczeń rozdzielonych drzwiami:

```
Maze* MazeGame::CreateMaze () {
  Maze* aMaze = new Maze;
  Room* r1 = new Room(1);
  Room* r2 = new Room(2);
  Door* theDoor = new Door(r1, r2);

  aMaze->AddRoom(r1);
  aMaze->AddRoom(r2);

  r1->SetSide(North, new Wall);
  r1->SetSide(East, theDoor);
  r1->SetSide(South, new Wall);
  r1->SetSide(West, new Wall);

  r2->SetSide(North, new Wall);
  r2->SetSide(East, new Wall);
  r2->SetSide(South, new Wall);
  r2->SetSide(West, theDoor);

  return aMaze;
}
```

Funkcja ta jest stosunkowo skomplikowana, jeśli weźmiemy pod uwagę, że jedyne, co robi, to tworzy labirynt składający się z dwóch pomieszczeń. Można łatwo wymyślić sposób na uproszczenie tej funkcji. Na przykład konstruktor klasy `Room` mógłby inicjować pokój przez przypisanie ścian do jego stron. Jednak to rozwiązanie powoduje jedynie przeniesienie kodu w inne miejsce. Prawdziwy problem związany z tą funkcją składową nie jest związany z jej rozmiarem, ale z *brakiem elastyczności*. Powoduje ona zapisanie na stałe układu labiryntu. Zmiana tego układu wymaga zmodyfikowania omawianej funkcji składowej. Można to zrobić albo przez jej przesłonięcie (co oznacza ponowną implementację całego kodu), albo przez zmodyfikowanie jej fragmentów (to podejście jest narażone na błędy i nie sprzyja ponownemu wykorzystaniu rozwiązania).

Wzorce konstrukcyjne pokazują, jak zwiększyć *elastyczność* projektu. Nie zawsze oznacza to zmniejszenie samego projektu. Wzorce te przede wszystkim ułatwiają modyfikowanie klas definiujących komponenty labiryntu.

Załóżmy, że chcemy powtórnie wykorzystać układ labiryntu w nowej grze obejmującej (między innymi) magiczne labirynty. Potrzebne będą w niej nowe rodzaje komponentów, takie jak `DoorNeedingSpell` (drzwi, które można zamknąć i następnie otworzyć tylko za pomocą czaru) i `EnchantedRoom` (pokój z niezwykłymi przedmiotami, na przykład magicznymi kluczami lub czarami). Jak można w łatwy sposób zmodyfikować operację `CrateMaze`, aby tworzyła labirynty z obiektami nowych klas?

W tym przypadku największa przeszkoda związana jest z zapisaniem na stałe klas, których egzemplarze tworzy opisywana operacja. Wzorce konstrukcyjne udostępniają różne sposoby usuwania bezpośrednich referencji do klas konkretnych z kodu, w którym trzeba tworzyć egzemplarze takich klas:

- Jeśli operacja `CreateMaze` przy tworzeniu potrzebnych pomieszczeń, ścian i drzwi wywołuje funkcje wirtualne zamiast konstruktora, można zmienić klasy, których egzemplarze powstają, przez utworzenie podklasy klasy `MazeGame` i ponowne zdefiniowanie funkcji wirtualnych. To rozwiązanie to przykład zastosowania wzorca Metoda wytwórcza (s. 110).
- Jeśli operacja `CreateMaze` otrzymuje jako parametr obiekt, którego używa do tworzenia pomieszczeń, ścian i drzwi, można zmienić klasy tych komponentów przez przekazanie nowych parametrów. Jest to przykład zastosowania wzorca Fabryka abstrakcyjna (s. 101).
- Jeśli operacja `CreateMaze` otrzymuje obiekt, który potrafi utworzyć cały nowy labirynt za pomocą operacji dodawania pomieszczeń, drzwi i ścian, można zastosować dziedziczenie do zmodyfikowania fragmentów labiryntu lub sposobu jego powstawania. W ten sposób działa wzorzec Budowniczy (s. 92).
- Jeśli operacja `CreateMaze` jest sparametryzowana za pomocą różnych prototypowych obiektów reprezentujących pomieszczenia, drzwi i ściany, które kopiuje i dodaje do labiryntu, można zmienić układ labiryntu przez zastąpienie danych obiektów prototypowych innymi. Jest to przykład zastosowania wzorca Prototyp (s. 120).

Ostatni wzorzec konstrukcyjny, Singleton (s. 130), pozwala zagwarantować, że w grze powstanie tylko jeden labirynt, a wszystkie obiekty gry będą mogły z niego korzystać (bez uciekania się do stosowania zmiennych lub funkcji globalnych). Wzorzec ten ułatwia też rozbudowywanie lub zastępowanie labiryntów bez modyfikowania istniejącego kodu.

BUDOWNICZY (BUILDER) *obiektowy, konstrukcyjny*

PRZEZNACZENIE

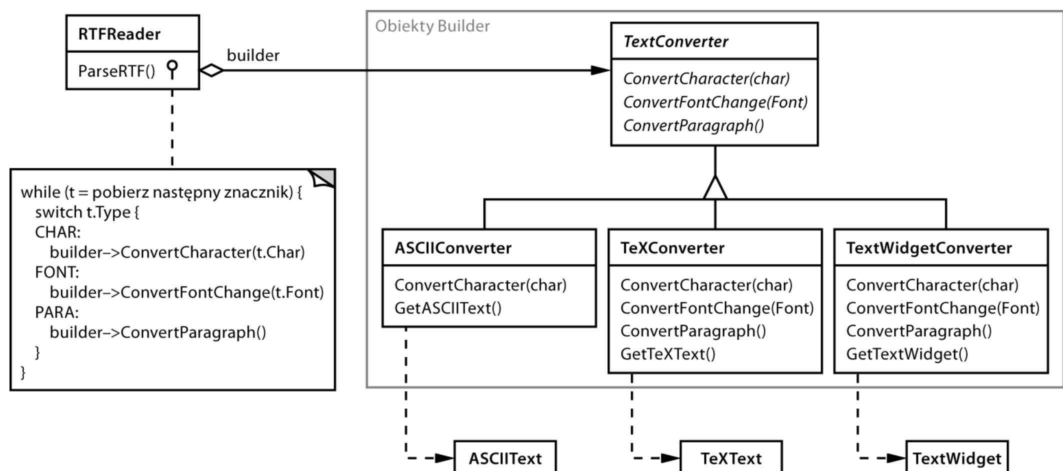
Oddziela tworzenie złożonego obiektu od jego reprezentacji, dzięki czemu ten sam proces konstrukcji może prowadzić do powstawania różnych reprezentacji.

UZASADNIENIE

Czytnik dokumentów w formacie RTF (ang. *Rich Text Format*) powinien móc przekształcać takie dokumenty na wiele formatów tekstowych. Takie narzędzie mogłoby przeprowadzać konwersję dokumentów RTF na zwykły tekst w formacie ASCII lub na widget tekstowy, który można interaktywnie edytować. Jednak problem polega na tym, że liczba możliwych przekształceń jest nieokreślona. Dlatego należy zachować możliwość łatwego dodawania nowych metod konwersji bez konieczności modyfikowania czytnika.

Rozwiązanie polega na skonfigurowaniu klasy `RTFReader` za pomocą obiektu `TextConverter` przekształcającego dokumenty RTF na inną reprezentację tekstową. Klasa `RTFReader` w czasie analizowania dokumentu RTF korzysta z obiektu `TextConverter` do przeprowadzania konwersji. Kiedy klasa `RTFReader` wykryje znacznik formatu RTF (w postaci zwykłego tekstu lub słowa sterującego z tego formatu), przekaże do obiektu `TextConverter` żądanie przekształcenia znacznika. Obiekty `TextConverter` odpowiadają zarówno za przeprowadzanie konwersji danych, jak i zapisywanie znacznika w określonym formacie.

Podklasy klasy `TextConverter` są wyspecjalizowane pod kątem różnych konwersji i formatów. Na przykład klasa `ASCIIConverter` ignoruje żądania związane z konwersją elementów innych niż zwykły tekst. Z kolei klasa `TeXConverter` obejmuje implementację operacji obsługujących wszystkie żądania, co umożliwia utworzenie reprezentacji w formacie TEX, uwzględniającej wszystkie informacje na temat stylu tekstu. Klasa `TextWidgetConverter` generuje złożony obiekt interfejsu użytkownika umożliwiający oglądanie i edytowanie tekstu.



Każda klasa konwertująca przyjmuje mechanizm tworzenia i składania obiektów złożonych oraz ukrywa go za abstrakcyjnym interfejsem. Konwerter jest oddzielony od czytnika odpowiadającego za analizowanie dokumentów RTF.

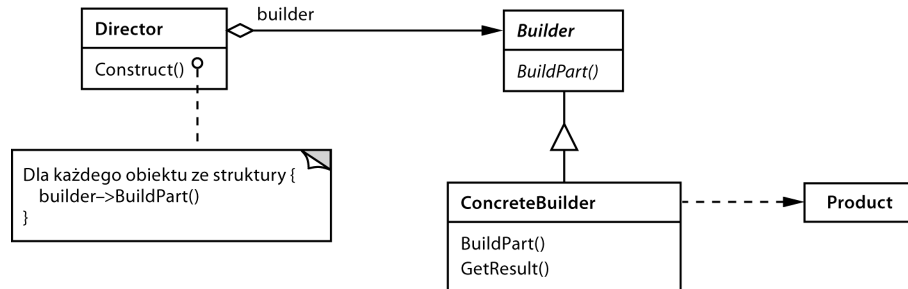
Wzorzec Budowniczy ujmuje wszystkie te relacje. W tym wzorcu każda klasa konwertująca nosi nazwę **builder** (czyli budowniczy), a klasa czytnika to **director** (czyli kierownik). Zastosowanie wzorca Budowniczy w przytoczonym przykładzie powoduje oddzielenie algorytmu interpretującego format tekstowy (czyli parsera dokumentów RTF) od procesu tworzenia i reprezentowania przekształconego dokumentu. Umożliwia to powtórne wykorzystanie algorytmu analizującego z klasy RTFReader do przygotowania innych reprezentacji tekstu z dokumentów RTF. Aby to osiągnąć, wystarczy skonfigurować klasę RTFReader za pomocą innej podklasy klasy TextConverter.

WARUNKI STOSOWANIA

Wzorca Budowniczy należy używać w następujących sytuacjach:

- Jeśli algorytm tworzenia obiektu złożonego powinien być niezależny od składników tego obiektu i sposobu ich łączenia.
- Kiedy proces konstrukcji musi umożliwiać tworzenie różnych reprezentacji generowanego obiektu.

STRUKTURA



ELEMENTY

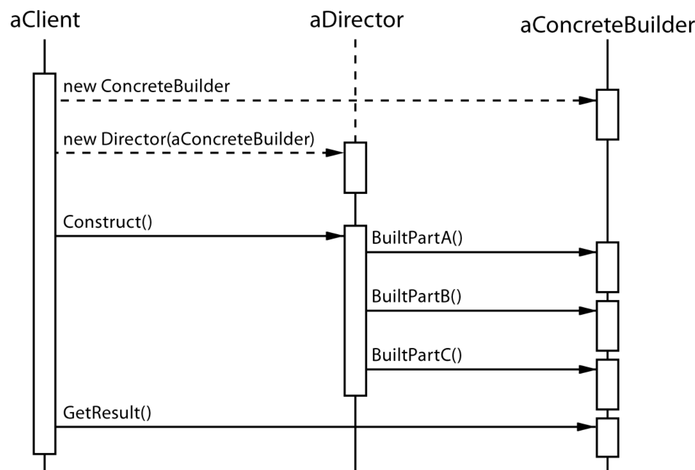
- **Builder** (TextConverter), czyli budowniczy:
  - określa interfejs abstrakcyjny do tworzenia składników obiektu Product.
- **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter), czyli budowniczy konkretny:
  - tworzy i łączy składniki produktu w implementacji interfejsu klasy Builder;
  - definiuje i śledzi generowane reprezentacje;
  - udostępnia interfejs do pobierania produktów (na przykład operacje GetASCIIText i GetTextWidget).

- **`Director`** (`RTFReader`), czyli kierownik:
  - tworzy obiekt za pomocą interfejsu klasy `Builder`.
- **`Product`** (`ASCIIText`, `TeXText`, `TextWidget`):
  - reprezentuje generowany obiekt złożony; klasa `ConcreteBuilder` tworzy wewnętrzną reprezentację produktu i definiuje proces jej składania;
  - obejmuje klasy definiujące składowe elementy obiektu, w tym interfejsy do łączenia składowych w ostateczną postać obiektu.

WSPÓŁDZIAŁANIE

- Klient tworzy obiekt `Director` i konfiguruje go za pomocą odpowiedniego obiektu `Builder`.
- Kiedy potrzebne jest utworzenie części produktu, obiekt `Director` wysyła powiadomienie do obiektu `Builder`.
- Obiekt `Builder` obsługuje żądania od obiektu `Director` i dodaje części do produktu.
- Klient pobiera produkt od obiektu `Builder`.

Poniższy diagram interakcji pokazuje, w jaki sposób klasy `Builder` i `Director` współdziałają z klientem.



KONSEKWENCJE

Oto kluczowe konsekwencje zastosowania wzorca Budowniczy:

1. *Możliwość modyfikowania wewnętrznej reprezentacji produktu*. Obiekt `Builder` udostępnia obiektowi `Director` interfejs abstrakcyjny do tworzenia produktu. Interfejs ten umożliwia obiektowi `Builder` ukrycie reprezentacji i wewnętrznej struktury produktu, a także sposobu jego składania. Ponieważ do tworzenia produktu służy interfejs abstrakcyjny, zmiana wewnętrznej reprezentacji produktu wymaga jedynie zdefiniowania obiektu `Builder` nowego rodzaju.

2. *Odizolowanie reprezentacji od kodu służącego do tworzenia produktu*. Wzorzec Budowniczy pomaga zwiększyć modularność, ponieważ kapsułkuje sposób tworzenia i reprezentowania obiektu złożonego. Klienty nie potrzebują żadnych informacji o klasach definiujących wewnętrzną strukturę produktu, ponieważ klasy te nie występują w interfejsie obiektu `Builder`.

   Każdy obiekt `ConcreteBuilder` obejmuje cały kod potrzebny do tworzenia i składania produktów określonego rodzaju. Kod ten wystarczy napisać raz. Następnie można wielokrotnie wykorzystać go w różnych obiektach `Director` do utworzenia wielu odmian obiektu `Product` za pomocą tych samych składników. W przykładzie dotyczącym dokumentów RTF moglibyśmy zdefiniować czytnik dokumentów o formacie innym niż RTF, na przykład klasę `SGMLReader`, i użyć tych samych podklas klasy `TextConverter` do wygenerowania reprezentacji dokumentów SGML w postaci obiektów `ASCIIText`, `TeXText` i `TextWidget`.

3. *Większa kontrola nad procesem tworzenia*. Wzorzec Budowniczy — w odróżnieniu od wzorców konstrukcyjnych tworzących produkty w jednym etapie — polega na generowaniu ich krok po kroku pod kontrolą obiektu `Director`. Dopiero po ukończeniu produktu obiekt `Director` odbiera go od obiektu `Builder`. Dlatego interfejs klasy `Builder` w większym stopniu niż inne wzorce konstrukcyjne odzwierciedla proces tworzenia produktów. Zapewnia to pełniejszą kontrolę nad tym procesem, a tym samym i wewnętrzną strukturą gotowego produktu.

IMPLEMENTACJA

Zwykle w implementacji znajduje się klasa abstrakcyjna `Builder` obejmująca definicję operacji dla każdego komponentu, którego utworzenia może zażądać obiekt `Director`. Domyślnie operacje te nie wykonują żadnych działań. W klasie `ConcreteBuilder` przesłonięte są operacje komponentów, które klasa ta ma generować.

Oto inne związane z implementacją kwestie, które należy rozważyć:

1. *Interfejs do składania i tworzenia obiektów*. Obiekty `Builder` tworzą produkty krok po kroku. Dlatego interfejs klasy `Builder` musi być wystarczająco ogólny, aby umożliwiał konstruowanie produktów każdego rodzaju przez konkretne podklasy klasy `Builder`.

   Kluczowa kwestia projektowa dotyczy modelu procesu tworzenia i składania obiektów. Zwykle wystarczający jest model, w którym efekty zgłoszenia żądania konstrukcji są po prostu dołączane do produktu. W przykładzie związanym z dokumentami RTF obiekt `Builder` przekształca i dołącza następny znacznik do wcześniej skonwertowanego tekstu.

   Jednak czasem potrzebny jest dostęp do wcześniej utworzonych części produktu. W przykładzie dotyczącym labiryntów, który prezentujemy w punkcie Przykładowy kod, interfejs klasy `MazeBuilder` umożliwia dodanie drzwi między istniejącymi pomieszczeniami. Następnym przykładem, w którym jest to potrzebne, są budowane od dołu do góry struktury drzewiaste, takie jak drzewa składni. Wtedy obiekt `Builder` zwraca węzły podrzędne obiektowi `Director`, który następnie przekazuje je ponownie do obiektu `Builder`, aby ten utworzył węzły nadrzędne.

2. *Dlaczego nie istnieje klasa abstrakcyjna produktów?* W typowych warunkach produkty tworzone przez obiekty `ConcreteBuilder` mają tak odmienną reprezentację, że udostępnienie wspólnej klasy nadrzędnej dla różnych produktów przynosi niewielkie korzyści. W przykładzie dotyczącym dokumentów RTF obiekty `ASCIIText` i `TextWidget` prawdopodobnie nie będą miały wspólnego interfejsu ani też go nie potrzebują. Ponieważ klienty zwykle konfigurują obiekt `Director` za pomocą odpowiedniego obiektu `ConcreteBuilder`, klient potrafi określić, która podklasa konkretna klasy `Builder` jest używana, i na tej podstawie obsługuje dostępne produkty.
3. *Zastosowanie pustych metod domyślnych w klasie `Builder`*. W języku C++ metody służące do tworzenia obiektów celowo nie są deklarowane jako czysto wirtualne funkcje składowe. W zamian definiuje się je jako puste metody, dzięki czemu w klientach trzeba przesłonić tylko potrzebne operacje.

PRZYKŁADOWY KOD

Zdefiniujmy nową wersję funkcji składowej `CreateMaze` (s. 90). Będzie ona przyjmować jako argument obiekt budujący klasy `MazeBuilder`.

Klasa `MazeBuilder` definiuje poniższy interfejs służący do tworzenia labiryntów:

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }
    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};
```

Ten interfejs pozwala utworzyć trzy elementy: (1) labirynt, (2) pomieszczenia o określonym numerze i (3) drzwi między ponumerowanymi pokojami. Operacja `GetMaze` zwraca labirynt klientowi. W podklasach klasy `MazeBuilder` należy ją przesłonić, aby zwracały one generowany przez siebie labirynt.

Wszystkie związane z budowaniem labiryntu operacje klasy `MazeBuilder` domyślnie nie wykonują żadnych działań. Jednak nie są zadeklarowane jako czysto wirtualne, dzięki czemu w klasach pochodnych wystarczy przesłonić tylko potrzebne metody.

Po utworzeniu interfejsu klasy `MazeBuilder` można zmodyfikować funkcję składową `CreateMaze`, aby przyjmowała jako parametr obiekt tej klasy:

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

Porównajmy tę wersję operacji `CreateMaze` z jej pierwowzorem. Warto zauważyć, w jaki sposób w budowniczym ukryto wewnętrzną reprezentację labiryntu — czyli klasy z definicjami pomieszczeń, drzwi i ścian — i jak elementy te są składane w gotowy labirynt. Można się domyślić, że istnieją klasy reprezentujące pomieszczenia i drzwi, jednak w kodzie nie ma wskazówek dotyczących klasy związanej ze ścianami. Ułatwia to zmianę reprezentacji labiryntu, ponieważ nie trzeba modyfikować kodu żadnego z klientów używających klasy `MazeBuilder`.

Wzorzec Budowniczy — podobnie jak inne wzorce konstrukcyjne — kapsułkuje tworzenie obiektów. Tutaj służy do tego interfejs zdefiniowany w klasie `MazeBuilder`. Oznacza to, że możemy wielokrotnie wykorzystać tę klasę do tworzenia labiryntów różnego rodzaju. Przykładem na to jest operacja `CreateComplexMaze`:

```
Maze* MazeGame::CreateComplexMaze (MazeBuilder& builder) {
    builder.BuildRoom(1);
    // ...
    builder.BuildRoom(1001);

    return builder.GetMaze();
}
```

Warto zauważyć, że klasa `MazeBuilder` nie tworzy labiryntu. Służy ona głównie do definiowania interfejsu do generowania labiryntów. Puste implementacje znajdują się w niej dla wygody programisty, natomiast potrzebne działania wykonują podklasy klasy `MazeBuilder`.

Podklasa `StandardMazeBuilder` to implementacja służąca do tworzenia prostych labiryntów. Zapisuje ona budowany labirynt w zmiennej `_currentMaze`.

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);

    virtual Maze* GetMaze();
private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

`CommonWall` to operacja narzędziowa określająca kierunek standardowej ściany pomiędzy dwoma pomieszczeniami.

Konstruktor `StandardMazeBuilder` po prostu inicjuje zmienną `_currentMaze`.

```
StandardMazeBuilder::StandardMazeBuilder () {
    _currentMaze = 0;
}
```

Operacja `BuildMaze` tworzy egzemplarz klasy `Maze`, który pozostałe operacje składają i ostatecznie zwracają do klienta (za to odpowiada operacja `GetMaze`).

```
void StandardMazeBuilder::BuildMaze () {
    _currentMaze = new Maze;
}

Maze* StandardMazeBuilder::GetMaze () {
    return _currentMaze;
}
```

Operacja `BuildRoom` tworzy pomieszczenie i ściany wokół niego.

```
void StandardMazeBuilder::BuildRoom (int n) {
    if (!_currentMaze->RoomNo(n)) {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);

        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

Aby utworzyć drzwi między dwoma pomieszczeniami, obiekt `StandardMazeBuilder` wyszukuje w labiryncie odpowiednie pokoje i łączącą je ścianę.

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);
    Door* d = new Door(r1, r2);

    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

Klienty mogą teraz użyć do utworzenia labiryntu operacji `CreateMaze` wraz z obiektem `StandardMazeBuilder`.

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

Moglibyśmy umieścić wszystkie operacje klasy `StandardMazeBuilder` w klasie `Maze` i pozwolić każdemu obiektowi `Maze, aby samodzielnie utworzył swój egzemplarz`. Jednak zmniejszenie klasy `Maze` sprawia, że łatwiej będzie ją zrozumieć i zmodyfikować, a wyodrębnienie z niej klasy `StandardMazeBuilder` nie jest trudne. Co jednak najważniejsze, rozdzielenie tych klas pozwala utworzyć różnorodne obiekty z rodziny `MazeBuilder`, z których każdy używa innych klas do generowania pomieszczeń, ścian i drzwi.

`CountingMazeBuilder` to bardziej wymyślna podklasa klasy `MazeBuilder`. Budowniczowie tego typu w ogóle nie tworzą labiryntów, a jedynie zliczają utworzone komponenty różnych rodzajów.

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();

    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);

    void GetCounts(int&, int&) const;
private:
    int _doors;
    int _rooms;
};
```

Konstruktor inicjuje liczniki, a przesłonięte operacje klasy `MazeBuilder` w odpowiedni sposób powiększają ich wartość.

```
CountingMazeBuilder::CountingMazeBuilder () {
    _rooms = _doors = 0;
}

void CountingMazeBuilder::BuildRoom (int) {
    _rooms++;
}

void CountingMazeBuilder::BuildDoor (int, int) {
    _doors++;
}

void CountingMazeBuilder::GetCounts (
    int& rooms, int& doors
) const {
    rooms = _rooms;
    doors = _doors;
}
```

Klient może korzystać z klasy `CountingMazeBuilder` w następujący sposób:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);

cout << "Liczba pomieszczeń w labiryncie to "
     << rooms << ", a liczba drzwi wynosi "
     << doors << "." « endl;
```

ZNANE ZASTOSOWANIA

Aplikacja do konwersji dokumentów RTF pochodzi z platformy ET++ [WGM88]. Jej część służąca do obsługi tekstu wykorzystuje budowniczego do przetwarzania tekstu zapisanego w formacie RTF.

Wzorzec Budowniczy jest często stosowany w języku Smalltalk-80 [Par90]:

- Klasa `Parser` w podsystemie odpowiedzialnym za kompilację pełni funkcję kierownika i przyjmuje jako argument obiekt `ProgramNodeBuilder`. Obiekt `Parser` za każdym razem, kiedy rozpozna daną konstrukcję składniową, wysyła do powiązanego z nim obiektu `ProgramNodeBuilder` powiadomienie. Kiedy parser kończy działanie, żąda od budowniczego utworzenia drzewa składni i przekazuje je klientowi.
- `ClassBuilder` to budowniczy, którego klasy używają do tworzenia swoich podklas. W tym przypadku klasa jest zarówno kierownikiem, jak i produktem.
- `ByteCodeStream` to budowniczy, który tworzy skompilowaną metodę w postaci tablicy bajtów. Klasa `ByteCodeStream` to przykład niestandardowego zastosowania wzorca Budowniczy, ponieważ generowany przez nią obiekt złożony jest kodowany jako tablica bajtów, a nie jako zwykły obiekt języka Smalltalk. Jednak interfejs klasy `ByteCodeStream` jest typowy dla budowniczych i łatwo można zastąpić tę klasą inną, reprezentującą programy jako obiekty składowe.

Platforma Service Configurator wchodząca w skład środowiska Adaptive Communications Environment korzysta z budowniczych do tworzenia komponentów usług sieciowych dołączanych do serwera w czasie jego działania [SS94]. Komponenty te są opisane w języku konfiguracyjnym analizowanym przez parser LALR(1). Akcje semantyczne parsera powodują wykonanie operacji na budowniczym, który dodaje informacje do komponentu usługowego. W tym przykładzie parser pełni funkcję kierownika.

POWIĄZANE WZORCE

Fabryka abstrakcyjna (s. 101) przypomina wzorzec Budowniczy, ponieważ też może służyć do tworzenia obiektów złożonych. Główna różnica między nimi polega na tym, że wzorzec Budowniczy opisuje przede wszystkim tworzenie obiektów złożonych krok po kroku. We wzorcu Fabryka abstrakcyjna nacisk położony jest na rodziny obiektów-produktów (zarówno prostych, jak i złożonych). Budowniczy zwraca produkt w ostatnim kroku, natomiast we wzorcu Fabryka abstrakcyjna produkt jest udostępniany natychmiast.

Budowniczy często służy do tworzenia kompozytów (s. 170).