Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

WZORCE PROJEKTOWE

Elementy oprogramowania obiektowego wielokrotnego użytku

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?

Helion

Tytuł oryginału: Design Patterns: Elements of Reusable Object-Oriented Software

Tłumaczenie: Tomasz Walczak
Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-8609-9

Authorized translation from the English language edition, entitled: Design Patterns, First Edition, ISBN 0201633612, by Erich Gamma; and Richard Helm, published by Pearson Education, Inc, publishing as Addison-Wesley Professional; Copyright © 1995 by Addison-Wesley.

Polish language edition published by Helion S.A.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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).

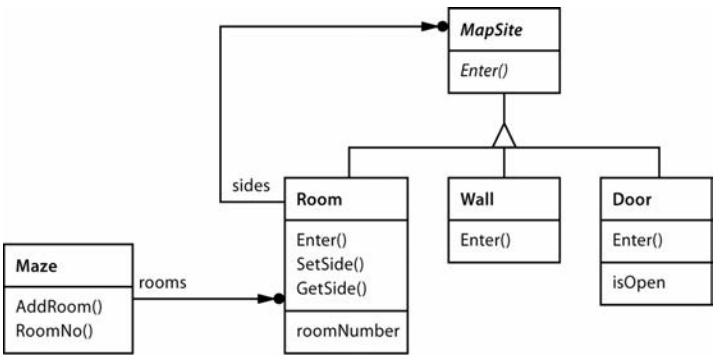
Ponieważ wzorce konstrukcyjne są mocno powiązane ze sobą, przeanalizujemy całą ich piątkę razem, aby podkreślić podobieństwa i różnice między nimi. Wykorzystamy też jeden przykład do zilustrowania implementacji tych wzorców — tworzenie labiryntu na potrzeby gry komputerowej. Labirynt i gra będą nieco 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

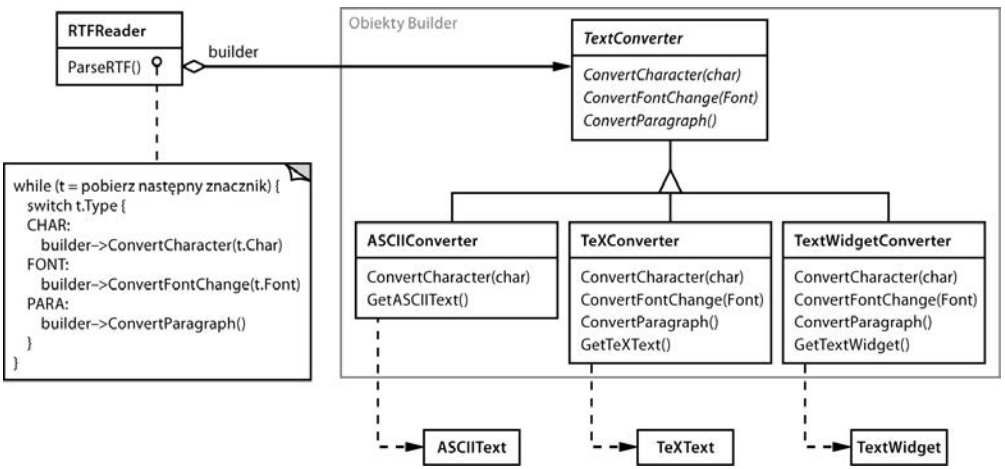
Oddziela tworzenie złożonego obiektu od jego reprezentacji, dzięki czemu ten sam proces konstrukcji może prowadzić do powstawania różnych reprezentacji.

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.

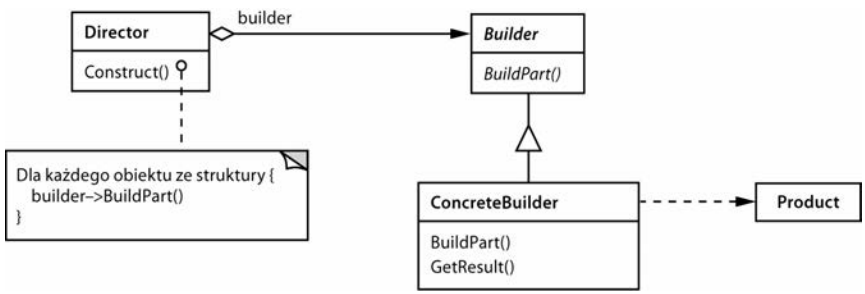
Wzorzec Budowniczy ujmuje wszystkie te relacje. W tym wzorcu każda klasa konwertująca nosi nazwę **`builder`** (czyli budowniczy), a klasa czytnika to **`director`** (czyli kierownik). Zastosowanie wzorca Budowniczy w przytoczonym przykładzie powoduje oddzielenie algorytmu interpretującego format tekstowy (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

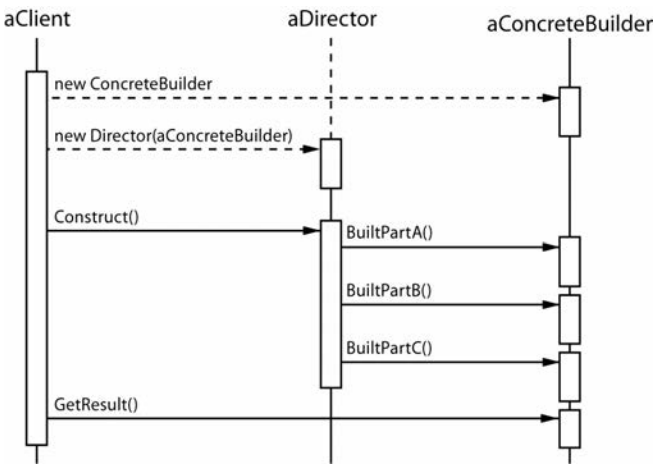
- **`Builder`** (`TextConverter`), czyli budowniczy:
  - określa interfejs abstrakcyjny do tworzenia składników obiektu `Product`.
- **`ConcreteBuilder`** (`ASCIIConverter`, `TeXConverter`, `TextWidgetConverter`), czyli budowniczy konkretny:
  - tworzy i łączy składniki produktu w implementacji interfejsu 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).

FABRYKA ABSTRAKCYJNA (ABSTRACT FACTORY) *obiektowy, konstrukcyjny*

PRZEZNACZENIE

Udostępnia interfejs do tworzenia rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez określania ich klas konkretnych.

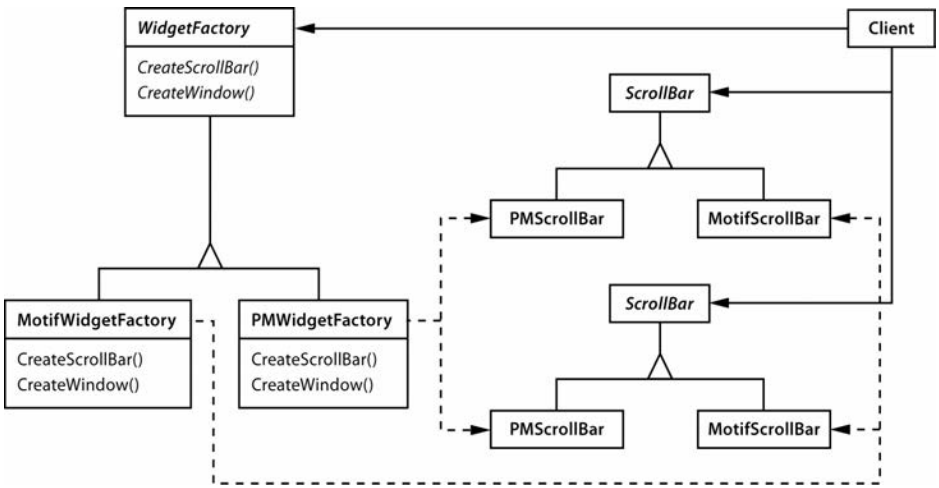
INNE NAZWY

Zestaw (ang. *kit*).

UZASADNIENIE

Zastanówmy się nad pakietem narzędziowym do tworzenia interfejsów użytkownika, obsługującym różne standardy wyglądu i działania (na przykład Motif i Presentation Manager). Poszczególne standardy wyznaczają różny wygląd i inne zachowanie widgetów interfejsu użytkownika, takich jak paski przewijania, okna i przyciski. Aby aplikacja była przenośna między różnymi standardami, nie należy zapisywać w niej na stałe określonego wyglądu i sposobu działania widgetów. Tworzenie określających te aspekty egzemplarzy klas w różnych miejscach aplikacji utrudnia późniejszą zmianę jej wyglądu i zachowania.

Możemy rozwiązać ten problem przez zdefiniowanie klasy abstrakcyjnej `WidgetFactory` i zadeklarowanie w niej interfejsu do tworzenia podstawowych widgetów. Należy przygotować też klasy abstrakcyjne dla poszczególnych rodzajów widgetów oraz podklasy konkretne z implementacją określonych standardów wyglądu i działania. Interfejs klasy `WidgetFactory` obejmuje operacje, które zwracają nowe obiekty dla klas abstrakcyjnych reprezentujących poszczególne widgety. Klienty wywołują te operacje, aby otrzymać egzemplarze widgetów, ale nie wiedzą, której klasy konkretnej używają. Dlatego klienty pozostają niezależne od stosowanego standardu wyglądu i działania.



Dla każdego standardu wyglądu i działania istnieje podklasa konkretna klasy `WidgetFactory`. W każdej takiej podklasie zaimplementowane są operacje do tworzenia widgetów odpowiednich dla danego standardu. Na przykład operacja `CreateScrollBar` klasy `MotifWidgetFactory` tworzy i zwraca egzemplarz paska przewijania zgodnego ze standardem Motif, natomiast odpowiadająca jej operacja klasy `PMWidgetFactory` tworzy pasek przewijania dla standardu Presentation Manager. Klienty tworzą widgety wyłącznie za pośrednictwem interfejsu klasy `WidgetFactory` i nie znają klas z implementacjami widgetów dla określonych standardów wyglądu i działania. Oznacza to, że klienty muszą być zgodne tylko z interfejsem klasy abstrakcyjnej, a nie z konkretnymi klasami konkretnymi.

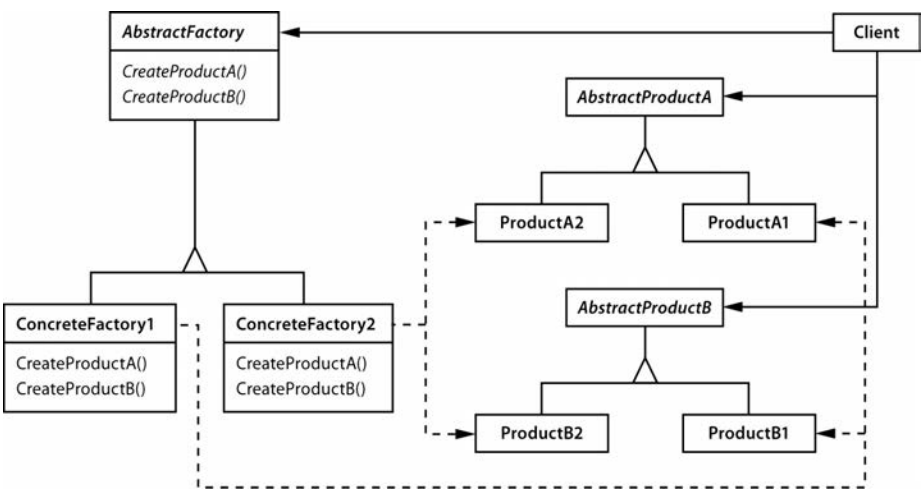
Klasa `WidgetFactory` wymusza ponadto zależności między klasami konkretnymi widgetów. Paska przewijania standardu Motif należy używać wraz z przyciskiem i edytorem tekstu zgodnymi z tym standardem. Ograniczenie to jest wymuszane automatycznie (jest to skutek zastosowania klasy `MotifWidgetFactory`).

WARUNKI STOSOWANIA

Wzorzec Fabryka abstrakcyjna należy stosować w następujących warunkach:

- Kiedy system powinien być niezależny od sposobu tworzenia, składania i reprezentowania jego produktów.
- Jeśli system należy skonfigurować za pomocą jednej z wielu rodzin produktów.
- Jeżeli powiązane obiekty-produkty z jednej rodziny są zaprojektowane do wspólnego użytku i trzeba wymusić jednoczesne korzystanie z tych obiektów.
- Kiedy programista chce udostępnić klasę biblioteczną produktów i ujawnić jedynie ich interfejsy, a nie implementacje.

STRUKTURA



ELEMENTY

- AbstractFactory (`WidgetFactory`), czyli fabryka abstrakcyjna:
  - obejmuje deklarację interfejsu z operacjami tworzącymi produkty abstrakcyjne.
- ConcreteFactory **(**`MotifWidgetFactory`**,** `PMWidgetFactory`**)**, czyli fabryka konkretna:
  - obejmuje implementację operacji tworzących produkty konkretne.
- AbstractProduct **(**`Window`**,** `ScrollBar`**)**, czyli produkt abstrakcyjny:
  - obejmuje deklarację interfejs dla produktów określonego typu.
- ConcreteProduct **(**`MotifWindow`**,** `MotifScrollBar`**)**, czyli produkt konkretny:
  - definiuje obiekt-produkt tworzony przez odpowiadającą mu fabrykę konkretną;
  - obejmuje implementację interfejsu klasy `AbstractProduct`**.**
- Client**:**
  - korzysta jedynie z interfejsów zadeklarowanych w klasach `AbstractFactory` i `Abstract` ↪`Product`.

WSPÓŁDZIAŁANIE

- W czasie wykonywania programu powstaje zwykle jeden egzemplarz klasy `ConcreteFactory`. Ta fabryka konkretna tworzy obiekty-produkty o określonej implementacji. Aby wygenerować różne obiekty-produkty, klienty muszą użyć odmiennych fabryk konkretnych.
- Klasa `AbstractFactory` przekazuje tworzenie obiektów-produktów do swojej podklasy `ConcreteFactory`.

KONSEKWENCJE

Wzorzec Fabryka abstrakcyjna ma następujące zalety i wady:

1. *Izoluje klasy konkretne*. Wzorzec Fabryka abstrakcyjna pomaga kontrolować klasy obiektów tworzonych przez aplikację. Ponieważ fabryka kapsułkuje zadanie i proces tworzenia obiektów-produktów, izoluje klienty od klas zawierających implementację. Klienty manipulują egzemplarzami tych klas za pośrednictwem interfejsów abstrakcyjnych. Nazwy klas produktów są odizolowane w implementacji fabryki konkretnej i nie pojawiają się w kodzie klienckim.
2. *Ułatwia zastępowanie rodzin produktów*. Klasa fabryki konkretnej pojawia się w aplikacji tylko raz — w miejscu tworzenia jej egzemplarza. Dlatego łatwo jest zmienić fabrykę konkretną wykorzystywaną przez aplikację. Aby użyć w programie innego zestawu produktów, wystarczy podać inną fabrykę konkretną. Ponieważ fabryka abstrakcyjna tworzy kompletną rodzinę produktów, jednocześnie zmieniana jest cała taka rodzina. W przykładowym interfejsie użytkownika można zastąpić widgety standardu Motif widgetami standardu Presentation Manager w prosty sposób — przez podmianę odpowiednich obiektów-fabryk i odtworzenie interfejsu.

3. *Ułatwia zachowanie spójności między produktami*. Jeśli obiekty-produkty z danej rodziny są zaprojektowane tak, aby używać ich razem, ważne jest, aby aplikacja w danym momencie korzystała z obiektów z tylko jednej rodziny. Klasa `AbstractFactory` pozwala w łatwy sposób wymusić to ograniczenie.
4. *Utrudnia dodawanie obsługi produktów nowego rodzaju*. Rozszerzanie fabryk abstrakcyjnych w celu tworzenia produktów nowego typu nie jest proste. Wynika to z tego, że w interfejsie klasy `AbstractFactory` na stałe zapisany jest zestaw produktów, które można utworzyć. Aby dodać obsługę produktów nowego rodzaju, trzeba rozszerzyć interfejs fabryki, co wymaga zmodyfikowania klasy `AbstractFactory` i wszystkich jej podklas. Jedno z rozwiązań tego problemu omawiamy w punkcie Implementacja.

IMPLEMENTACJA

Oto kilka technik przydatnych przy implementowaniu wzorca Fabryka abstrakcyjna.

1. *Fabryki jako singletony*. W aplikacji zwykle potrzebny jest tylko jeden egzemplarz klasy `ConcreteFactory` na każdą rodzinę produktów. Dlatego zazwyczaj najlepiej jest implementować takie klasy zgodnie ze wzorcem Singleton (s. 130).
2. *Tworzenie produktów*. Klasa `AbstractFactory` obejmuje jedynie deklarację *interfejsu* do tworzenia produktów. To podklasy `ConcreteProduct` odpowiadają za ich generowanie. Najczęściej definiowana jest w tym celu metoda wytwórcza (zobacz wzorzec Metoda wytwórcza, s. 110) dla każdego produktu. W fabryce konkretnej generowane produkty są określane przez przesłonięcie metody fabrycznej dla każdego z tych produktów. Choć taka implementacja jest prosta, wymaga przygotowania dla każdej rodziny produktów nowej podklasy konkretnej reprezentującej fabrykę, nawet jeśli różnice między poszczególnymi rodzinami są niewielkie.

   Jeśli aplikacja może obejmować wiele rodzin produktów, fabrykę konkretną można zaimplementować za pomocą wzorca Prototyp (s. 120). Fabryka konkretna jest wtedy inicjowana za pomocą prototypowego egzemplarza każdego produktu z rodziny i tworzy nowe produkty przez klonowanie ich prototypów. Podejście oparte na wzorcu Prototyp pozwala wyeliminować konieczność tworzenia dla każdej rodziny produktów nowej klasy konkretnej reprezentującej fabrykę.

   Oto sposób na zaimplementowanie fabryki opartej na wzorcu Prototyp w języku Smalltalk. Fabryka konkretna przechowuje klonowane prototypy w słowniku o nazwie `partCatalog`. Metoda `make:` pobiera prototyp i klonuje go:

   ```
   make: partName
       ^ (partCatalog at: partName) copy
   ```

   Fabryka konkretna obejmuje metodę do dodawania elementów do katalogu:

   ```
   addPart: partTemplate named: partName
       partCatalog at: partName put: partTemplate
   ```

   Prototypy są dodawane do fabryki przez wskazanie ich za pomocą symbolu:

   ```
   aFactory addPart: aPrototype named: #ACMEWidget
   ```

W językach, w których klasy są traktowane jak standardowe obiekty (na przykład w językach Smalltalk i Objective C), można zastosować pewną odmianę podejścia opartego na wzorcu Prototyp. W tych językach klasy można uznać za uproszczone fabryki tworzące produkty tylko jednego rodzaju. W tworzącej produkty fabryce konkretnej można przypisać do zmiennych *klasy* (podobnie jak prototypy). Te klasy będą tworzyć nowe egzemplarze na rzecz fabryki konkretnej. Aby zdefiniować nową fabrykę, należy zainicjować egzemplarz fabryki konkretnej za pomocą *klas* produktów, zamiast tworzyć podklasę. To podejście pozwala wykorzystać specyficzne cechy języków, natomiast podstawowe rozwiązanie oparte na wzorcu Prototyp jest niezależne od języka.

Wersja oparta na klasach — podobnie jak opisane właśnie fabryki oparte na wzorcu Prototyp napisane w języku Smalltalk — ma jedną zmienną egzemplarza (`partCatalog`). Jest to słownik, którego kluczami są nazwy poszczególnych elementów. Zmienna `partCatalog` nie przechowuje przeznaczonych do sklonowania prototypów, ale klasy produktów. Nowa wersja metody `make:` wygląda tak:

```
make: partName
    ^ (partCatalog at: partName) new
```

3. *Definiowanie rozszerzalnych fabryk.* W klasie `AbstractFactory` zwykle zdefiniowane są różne operacje dla wszystkich rodzajów produktów generowanych przez tę klasę. Rodzaje produktów są określone w sygnaturach operacji. Dodanie produktu nowego rodzaju wymaga zmodyfikowania interfejsu klasy `AbstractFactory` i wszystkich klas od niego zależnych.

   Elastyczniejszy (choć mniej bezpieczny) projekt wymaga dodania parametru do operacji tworzących obiekty. Ten parametr określa rodzaj generowanego obiektu. Jako parametru można użyć identyfikatora klasy, liczby całkowitej, łańcucha znaków lub dowolnego innego elementu identyfikującego rodzaj produktu. W tym podejściu klasa `AbstractFactory` potrzebuje jedynie pojedynczej operacji `Make` z parametrem określającym rodzaj tworzonego obiektu. Tej techniki użyliśmy w omówionych wcześniej fabrykach abstrakcyjnych opartych na wzorcu Prototyp lub klasie.

   Tę wersję łatwiej jest stosować w językach z dynamiczną kontrolą typów (takich jak Smalltalk) niż w językach ze statyczną kontrolą typów (na przykład C++). W języku C++ rozwiązania tego można użyć tylko wtedy, jeśli wszystkie obiekty mają tę samą abstrakcyjną klasę bazową lub gdy klient, który zażądał produktów, może bezpiecznie przekształcić ich typ na właściwy. W punkcie Implementacja poświęconym wzorcowi Metoda wytwórcza (s. 110) pokazujemy, jak zaimplementować takie sparametryzowane operacje w języku C++.

   Jednak nawet kiedy przekształcanie na właściwy typ nie jest konieczne, pozostaje do rozwiązania pewien problem — wszystkie produkty przekazywane do klienta mają *ten sam* abstrakcyjny interfejs określony przez zwracany typ. Dlatego klient nie może rozróżnić klas produktów ani dokonywać bezpiecznych założeń na ich temat. Jeśli klient musi wykonać operacje specyficzne dla podklasy, nie będzie mógł uzyskać dostępu do nich za pośrednictwem abstrakcyjnego interfejsu. Choć klient może przeprowadzić rzutowanie w dół (na przykład za pomocą instrukcji `dynamic_cast` w języku C++), nie zawsze jest to wykonalne lub bezpieczne, ponieważ operacja ta może zakończyć się niepowodzeniem. Jest to typowy koszt utworzenia wysoce elastycznego i rozszerzalnego interfejsu.

PRZYKŁADOWY KOD

Zastosujmy wzorzec Fabryka abstrakcyjna do utworzenia labiryntów opisanych w początkowej części rozdziału.

Klasa `MazeFactory` służy do tworzenia elementów labiryntów — pomieszczeń, ścian i drzwi między pokojami. Można użyć jej w programie, który wczytuje plany labiryntów z pliku i tworzy odpowiednie labirynty. Ponadto można wykorzystać ją w aplikacji generującej labirynty w sposób losowy. Programy, które tworzą labirynty, przyjmują obiekt `MazeFactory` jako argument, dzięki czemu programista może określić generowane pomieszczenia, ściany i drzwi.

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Przypomnijmy, że funkcja składowa `CreateMaze` (s. 90) tworzy mały labirynt składający się z dwóch pomieszczeń i drzwi między nimi. W tej funkcji nazwy klas zapisane są na stałe, co utrudnia generowanie labiryntów o różnych elementach.

Oto wersja operacji `CreateMaze`, w której rozwiązaliśmy ten problem przez zastosowanie obiektu `MazeFactory` jako parametru:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

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

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());

    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);

    return aMaze;
}
```

Możemy utworzyć klasę `EnchantedMazeFactory` (fabrykę magicznych labiryntów) jako podklasę klasy `MazeFactory`. Klasa `EnchantedMazeFactory` powinna przesłaniać kilka funkcji składowych i zwracać różne podklasy klas `Room`, `Wall` itd.

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

Teraz załóżmy, że chcemy utworzyć grę z labiryntem, w której w pomieszczeniach mogą znajdować się bomby. Jeśli bomba wybuchnie, uszkodzi co najmniej ściany. Możemy dodać podklasę klasy `Room` służącą do rejestrowania, czy w pokoju znajduje się bomba i czy już wybuchła. Potrzebna będzie też podklasa klasy `Wall` do śledzenia uszkodzeń ścian. Nazwijmy te klasy `RoomWithABomb` i `BombedWall`.

Ostatnia klasa, którą zdefiniujemy, to `BombedMazeFactory`. Jest to podklasa klasy `MazeFactory` gwarantująca, że ściany to obiekty `BombedWall`, a pomieszczenia to obiekty `RoomWithABomb`. W klasie `BombedMazeFactory` trzeba przesłonić tylko dwie funkcje:

```
Wall* BombedMazeFactory::MakeWall () const {
  return new BombedWall;
}

Room* BombedMazeFactory::MakeRoom(int n) const {
    return new RoomWithABomb(n);
}
```

Aby zbudować prosty labirynt zawierający bomby, wystarczy wywołać operację `CreateMaze` i przekazać do niej obiekt klasy `BombedMazeFactory`.

```
MazeGame game;
BombedMazeFactory factory;

game.CreateMaze(factory);
```

Operacja `CreateMaze` może przyjmować także egzemplarz klasy `EnchantedMazeFactory`, jeśli ma utworzyć magiczny labirynt.

Zauważmy, że klasa `MazeFactory` jest jedynie kolekcją metod wytwórczych. Jest to najczęściej stosowany sposób implementowania wzorca Fabryka abstrakcyjna. Ponadto warto zwrócić uwagę na to, że klasa `MazeFactory` nie jest abstrakcyjna. Dlatego pełni jednocześnie funkcje klas `AbstractFactory` *oraz* `ConcreteFactory`. Jest to następna często używana implementacja w prostych zastosowaniach wzorca Fabryka abstrakcyjna. Ponieważ `MazeFactory` to klasa konkretna składająca się w całości z metod wytwórczych, łatwo jest utworzyć nową klasę tego rodzaju przez utworzenie podklasy i przesłonięcie operacji, które trzeba zmodyfikować.

W operacji `CreateMaze` wykorzystaliśmy operację `SetSide` obiektów `Room` do określenia stron w tych obiektach. Jeśli operacja `CreateMaze` tworzy pomieszczenia za pomocą klasy `BombedMazeFactory`, labirynt będzie składał się z obiektów `RoomWithABomb` ze stronami typu `BombedWall`. Jeśli obiekt `RoomWithABomb` będzie musiał uzyskać dostęp do specyficznej dla podklasy składowej obiektu `BombedWall`, konieczne będzie zrzutowanie referencji do ścian z typu `Wall*` na `BombedWall*`. To rzutowanie w dół jest bezpieczne, jeśli argument *rzeczywiście* ma typ `BombedWall`. Jest to pewne, jeżeli ściany są zbudowane wyłącznie za pomocą klasy `BombedMazeFactory`.

Języki z dynamiczną kontrolą typu, na przykład Smalltalk, oczywiście nie wymagają rzutowania w dół, jednak mogą generować błędy czasu wykonania, jeśli natrafią na obiekt `Wall` w miejscu, gdzie oczekują *podklasy* klasy `Wall`. Wykorzystanie przy tworzeniu ścian wzorca Fabryka abstrakcyjna pomaga zapobiec podobnym błędom czasu wykonania, ponieważ mamy wtedy pewność, że program utworzy ściany określonego typu.

Rozważmy wersję klasy `MazeFactory` w języku Smalltalk. Obejmuje ona jedną operację `make`, która przyjmuje jako parametr rodzaj generowanego obiektu. Ponadto fabryka konkretna przechowuje klasy tworzonych produktów.

Najpierw należy napisać odpowiednik operacji `CreateMaze` w języku Smalltalk:

```
CreateMaze: aFactory
    | room1 room2 aDoor |
    room1 := (aFactory make: #room) number: 1.
    room2 := (aFactory make: #room) number: 2.
    aDoor := (aFactory make: #door) from: room1 to: room2.
    room1 atSide: #tnorth put: (aFactory make: #wall).
    room1 atSide: #east put: aDoor.
    room1 atSide: #tsouth put: (aFactory make: #wall).
    room1 atSide: #twest put: (aFactory make: #wall).
    room2 atSide: #north put: (aFactory make: #wall).
    room2 atSide: #east put: (aFactory make: #wall).
    room2 atSide: #tsouth put: (aFactory make: #wall).
    room2 atSide: #twest put: aDoor.
    ^ Maze new addRoom: room1; addRoom: room2; yourself
```

Klasa `MazeFactory` — jak opisaliśmy to w punkcie Implementacja — wymaga tylko jednej zmiennej egzemplarza, `partCatalog`, aby udostępnić katalog, którego kluczami są klasy komponentów labiryntu. Przypomnijmy też, jak zaimplementowaliśmy metodę `make:`.

```
make: partName
    ^ (partCatalog at: partName) new
```

Teraz można utworzyć obiekt `MazeFactory` i wykorzystać go do zaimplementowania operacji `createMaze`. Do utworzenia fabryki posłuży metoda `createMazeFactory` klasy `MazeGame`.

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: Room named: #room;
        addPart: Door named: #door;
        yourself)
```

Obiekty `BombedMazeFactory` i `EnchantedMazeFactory` są tworzone przez powiązanie różnych klas z odpowiednimi kluczami. Na przykład obiekt `EnchantedMazeFactory` można utworzyć tak:

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: EnchantedRoom named: #room;
        addPart: DoorNeedingSpell named: #door;
        yourself)
```

ZNANE ZASTOSOWANIA

W pakiecie InterViews do określania klas `AbstractFactory` służy przyrostek `Kit` [Lin92]. Pakiet ten obejmuje definicje fabryk abstrakcyjnych `WidgetKit` i `DialogKit` generujących obiekty interfejsu użytkownika specyficzne dla danego standardu wyglądu i działania. Pakiet InterViews obejmuje też klasę `LayoutKit`, która w zależności od wybranego układu tworzy różne obiekty złożone. Na przykład układ poziomy może wymagać zastosowania odmiennych obiektów złożonych w zależności od orientacji dokumentu (pionowej lub poziomej).

W platformie ET++ [WGM88] wzorzec Fabryka abstrakcyjna zastosowano do zapewnienia przenośności rozwiązań między różnymi systemami okienkowymi (na przykład X Windows i SunView). Abstrakcyjna klasa bazowa `WindowSystem` definiuje interfejs do tworzenia obiektów reprezentujących zasoby systemów okienkowych (interfejs ten obejmuje na przykład operacje `MakeWindow`, `MakeFont`, `MakeColor` itd.). W podklasach konkretnych interfejs ten jest zaimplementowany dla określonych systemów okienkowych. W czasie wykonywania programu platforma ET++ tworzy egzemplarz podklasy konkretnej klasy `WindowSystem`, a egzemplarz ten generuje obiekty konkretne reprezentujące zasoby systemowe.

POWIĄZANE WZORCE

Fabryki abstrakcyjne często są implementowane za pomocą metod wytwórczych (Metoda wytwórcza, s. 110), jednak można wykorzystać do tego także wzorzec Prototyp (s. 120).

Fabryki konkretne są często singletonami (Singleton, s. 130).

METODA WYTWÓRCZA (FACTORY METHOD) *klasowy, konstrukcyjny*

PRZEZNACZENIE

Określa interfejs do tworzenia obiektów, przy czym umożliwia podklasom wyznaczenie klasy danego obiektu. Metoda wytwórcza umożliwia klasom przekazanie procesu tworzenia egzemplarzy podklasom.

INNE NAZWY

Konstruktor wirtualny (ang. *virtual constructor*).

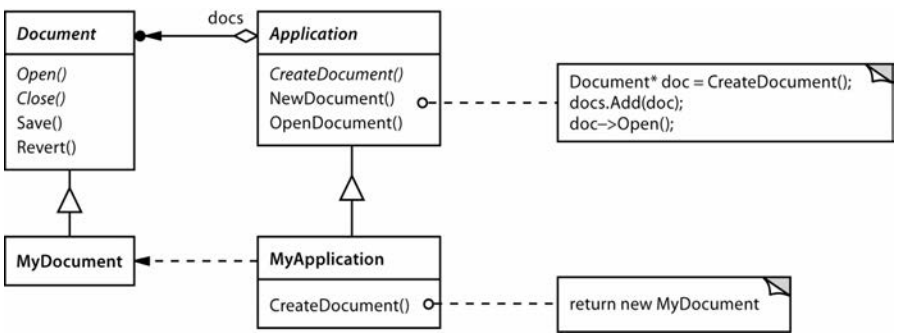
UZASADNIENIE

W platformach klasy abstrakcyjne służą do definiowania i podtrzymywania relacji między obiektami. Platforma często odpowiada także za tworzenie obiektów.

Zastanówmy się nad platformą dla aplikacji potrafiących wyświetlać wiele dokumentów. Dwie kluczowe abstrakcje w tej platformie to klasy `Application` i `Document`. Obie te klasy są abstrakcyjne, a w klientach trzeba utworzyć ich podklasy i umieścić tam implementacje specyficzne dla aplikacji. Aby utworzyć aplikację do rysowania, należy zdefiniować klasy `Drawing ↪Application` i `DrawingDocument`. Klasa `Application` odpowiada za zarządzanie obiektami `Document` i tworzy je na żądanie (na przykład kiedy użytkownik wybierze z menu opcję *Otwórz* lub *Nowy*).

Ponieważ określona podklasa klasy `Document`, której egzemplarz należy utworzyć, jest specyficzna dla aplikacji, w klasie `Application` nie można z góry ustalić rodzaju tej podklasy. Klasa `Application` potrafi jedynie określić, *kiedy* należy utworzyć nowy dokument, a nie *jakiego rodzaju* powinien on być. Stawia nas to przed dylematem — platforma musi tworzyć egzemplarze klas, ale ma informacje tylko o klasach abstrakcyjnych, których egzemplarzy wygenerować nie może.

Rozwiązaniem jest zastosowanie wzorca Metoda wytwórcza. Pozwala on zakapsułkować informacje o tym, którą podklasę klasy `Document` należy utworzyć, i zapisać te dane poza platformą.



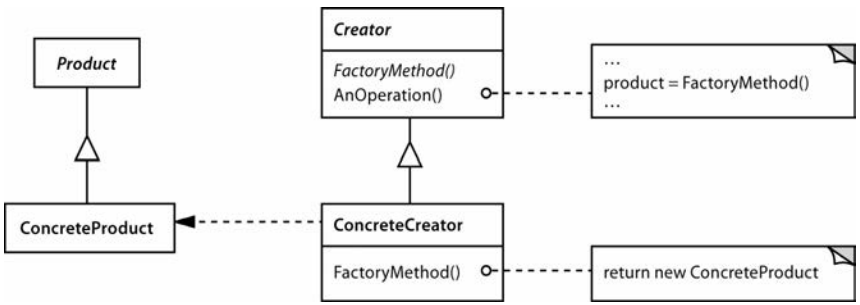
W podklasach klasy `Application` należy przedefiniować operację `CreateDocument` klasy `Application`, tak aby nowa wersja operacji zwracała odpowiednią podklasę klasy `Document`. Egzemplarz podklasy klasy `Application` może następnie generować specyficzne dla aplikacji egzemplarze klasy `Document` bez znajomości ich klasy. Operację `CreateDocument` nazywamy **metodą wytwórczą**, ponieważ odpowiada za wytwarzanie obiektów.

WARUNKI STOSOWANIA

Wzorca Metoda wytwórcza należy używać w następujących warunkach:

- Kiedy w danej klasie nie można z góry ustalić klasy obiektów, które trzeba utworzyć.
- Jeśli programista chce, aby to podklasy danej klasy określały tworzone przez nią obiekty.
- Jeżeli klasy delegują zadania do jednej z kilku podklas pomocniczych, a programista chce zapisać w określonym miejscu informacje o tym, która z tych podklas jest delegatem.

STRUKTURA



ELEMENTY

- **`Product`** (`Document`), czyli produkt:
  - definiuje interfejs obiektów generowanych przez metodę wytwórczą.
- **`ConcreteProduct`** (`MyDocument`), czyli produkt konkretny:
  - obejmuje implementację interfejsu klasy `Product`.
- **`Creator`** (`Application`), czyli wytwórca:
  - obejmuje deklarację metody wytwórczej zwracającej obiekty typu `Product`; w obiekcie `Creator` można też zdefiniować implementację domyślną metody fabrycznej, zwracającą domyślny obiekt `CreateProduct`;
  - może wywoływać metodę wytwórczą w celu wygenerowania obiektu `Product`.
- **`ConcreteCreator`** (`MyApplication`), czyli wytwórca konkretny:
  - przesłania metodę wytwórczą, tak aby zwracała egzemplarz klasy `ConcreteProduct`.

WSPÓŁDZIAŁANIE

- Klasa `Creator` działa na podstawie założenia, że w jej podklasach zdefiniowana jest metoda wytwórcza zwracająca egzemplarz odpowiedniej klasy `ConcreteProduct`.

KONSEKWENCJE

Metoda wytwórcza eliminuje konieczność wiązania klas specyficznych dla aplikacji z kodem. W kodzie używany jest tylko interfejs klasy `Product`, dlatego działać w nim będzie dowolna zdefiniowana przez użytkownika klasa `ConcreteProduct`.

Potencjalną wadą metody wytwórczej jest to, że klienty czasem muszą tworzyć podklasy klasy `Creator` tylko w celu wygenerowania określonego obiektu `ConcreteProduct`. Nie ma nic złego w tworzeniu podklas, jeśli w kliencie i tak trzeba dodać takie podklasy dla klasy `Creator`. Jednak jeżeli jest inaczej, w kliencie trzeba wprowadzić dodatkowe zmiany.

Oto dwie następne konsekwencje zastosowania wzorca Metoda wytwórcza:

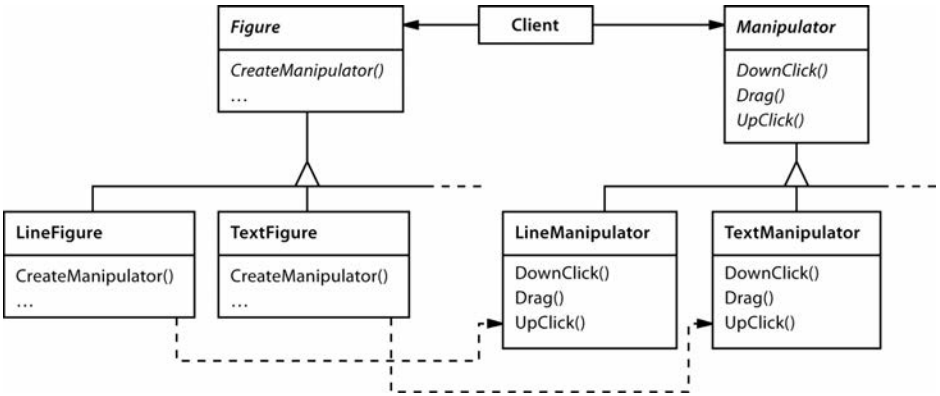
1. *Zapewnienie punktów zaczepienia dla podklas*. Tworzenie obiektów w klasie za pomocą metody wytwórczej zawsze daje większą elastyczność niż bezpośrednie ich generowanie. Wzorzec Metoda wytwórcza zapewnia punkty zaczepienia dla podklas na potrzeby tworzenia wzbogaconej wersji obiektu.

   W przykładzie dotyczącym klasy `Document` moglibyśmy zdefiniować w niej metodę wytwórczą o nazwie `CreateFileDialog` generującą domyślny obiekt okna dialogowego służący do otwierania istniejących dokumentów. W podklasie klasy `Document` można zdefiniować specyficzne dla aplikacji okno dialogowe przez przesłonięcie wspomnianej metody wytwórczej. W tym przykładzie metoda wytwórcza nie jest abstrakcyjna, ponieważ udostępnia przydatną implementację domyślną.

2. *Połączenie równoległych hierarchii klas*. W przykładach omówionych do tej pory metodę fabryczną wywołują tylko obiekty `Creator`. Jednak nie zawsze musi tak być. Metoda wytwórcza może okazać się przydatna także dla klientów (zwłaszcza w systemach z równoległymi hierarchiami klas).

   Równoległe hierarchie klas powstają, kiedy klasa deleguje część zadań do odrębnej klasy. Rozważmy figury graficzne, którymi można interaktywnie manipulować — rozciągać je, przenosić lub rotować za pomocą myszy. Zaimplementowanie takich interakcji nie zawsze jest łatwe. Często wymaga to zapisywania i aktualizowania informacji o stanie zmian w danym momencie. Ten stan jest niezbędny tylko w czasie manipulowania elementem, dlatego nie trzeba go przechowywać w obiekcie reprezentującym figurę. Ponadto poszczególne figury zachowują się inaczej w czasie manipulowania nimi przez użytkownika. Na przykład rozciąganie odcinka może doprowadzić do przesunięcia jego punktu końcowego, a rozciąganie tekstu — do zmiany wysokości interlinii.

   Z uwagi na te ograniczenia lepiej jest użyć odrębnego obiektu `Manipulator` i zaimplementować w nim obsługę interakcji oraz przechowywać tam stan specyficzny dla manipulacji. Poszczególne figury będą korzystać z różnych podklas klasy `Manipulator` do obsługi określonych interakcji. Ostateczna hierarchia klasy `Manipulator` jest równoległa (przynajmniej w części) do hierarchii klasy `Figure`.



Klasa `Figure` udostępnia metodę wytwórczą `CreateManipulator`. Umożliwia ona klientom tworzenie obiektów `Manipulator` odpowiadających podklasom klasy `Figure`. W tych podklasach omawiana metoda jest przesłonięta, tak aby zwracała egzemplarz odpowiedniej dla nich podklasy klasy `Manipulator`. Inna możliwość to zaimplementowanie w klasie `Figure` metody `CreateManipulator` w taki sposób, żeby zwracała domyślny egzemplarz klasy `Manipulator`. Wtedy podklasy klasy `Figure` mogą odziedziczyć tę domyślną implementację. Takie podklasy nie potrzebują powiązanych z nimi podklas klasy `Manipulator`, dlatego hierarchie są tylko częściowo równoległe.

Zauważmy, w jaki sposób metoda wytwórcza łączy dwie hierarchie klas. Obejmuje ona informacje o tym, które klasy są powiązane ze sobą.

IMPLEMENTACJA

Przy stosowaniu wzorca Metoda wytwórcza należy uwzględnić następujące zagadnienia:

1. *Dwie główne odmiany*. Dwa podstawowe warianty wzorca Metoda wytwórcza to: (1) utworzenie klasy `Creator` jako klasy abstrakcyjnej i pominięcie w niej implementacji zadeklarowanej metody wytwórczej oraz (2) utworzenie klasy `Creator` jako klasy konkretnej i umieszczenie w niej domyślnej implementacji metody wytwórczej. Można też utworzyć klasę abstrakcyjną z definicją implementacji domyślnej, jednak jest to rzadziej stosowane rozwiązanie.

   W pierwszym przypadku w podklasie *trzeba* zdefiniować implementację, ponieważ nie istnieje przydatna implementacja domyślna. Pozwala to rozwiązać problem tworzenia egzemplarzy nieprzewidzianych klas. W drugiej sytuacji umieszczenie metody wytwórczej w konkretnej klasie `Creator` służy przede wszystkim zwiększeniu elastyczności. Podejście to jest zgodne z następującą zasadą: „Twórz obiekty za pomocą odrębnej operacji, aby można przesłonić sposób ich generowania w podklasach". Ta reguła gwarantuje, że projektanci podklas będą mogli w razie potrzeby zmienić klasę obiektów generowanych przez klasę nadrzędną.

2. *Sparametryzowane metody wytwórcze*. Inna odmiana wzorca umożliwia metodom wytwórczym generowanie produktów *wielu* rodzajów. Metoda wytwórcza przyjmuje wtedy parametr określający rodzaj generowanego obiektu. Wszystkie obiekty tworzone przez taką

metodę wytwórczą będą miały wspólny interfejs klasy `Product`. W przykładzie dotyczącym klasy `Document` klasa `Application` może obsługiwać różne rodzaje obiektów `Document`. Aby określić specyficzny typ dokumentu, należy przekazać do operacji `CreateDocument` dodatkowy parametr.

W platformie Unidraw [VL90] (służy ona do tworzenia aplikacji z funkcją edycji w trybie graficznym) podejście to zastosowano do odtwarzania obiektów zapisanych na dysku. Platforma ta obejmuje definicję klasy `Creator` z metodą wytwórczą `Create` przyjmującą jako argument identyfikator klasy. Ten identyfikator określa klasę, której egzemplarz należy utworzyć. Kiedy platforma zapisuje obiekt na dysku, najpierw rejestruje identyfikator klasy, a następnie zmienne egzemplarza. W czasie odtwarzania obiektu najpierw wczytuje identyfikator klasy.

Po wczytaniu identyfikatora klasy platforma wywołuje operację `Create` i przekazuje do niej identyfikator jako parametr. Operacja `Create` wyszukuje konstruktor odpowiedniej klasy i wykorzystuje go do utworzenia egzemplarza danej klasy. W ostatnim kroku `Create` wywołuje operację `Read` obiektu, co powoduje wczytanie pozostałych informacji z dysku i zainicjowanie zmiennych egzemplarza.

Sparametryzowana metoda wytwórcza ma następującą ogólną postać (`MyProduct` i `YourProduct` są tu podklasami klasy `Product`).

```
class Creator {
public:
    virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
    if (id == MINE) return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // Powtarzane dla pozostałych produktów.

    return 0;
}
```

Przesłonięcie sparametryzowanej metody wytwórczej pozwala łatwo i wybiórczo rozszerzać lub modyfikować produkty tworzone przez klasę `Creator`. Można wprowadzić nowe identyfikatory dla produktów nowego rodzaju lub powiązać istniejące identyfikatory z innymi produktami.

Na przykład w podklasie `MyCreator` można zastąpić miejscami klasy `MyProduct` i `YourProduct` oraz dodać obsługę nowej podklasy `TheirProduct`.

```
Product* MyCreator::Create (ProductId id) {
    if (id == YOURS) return new MyProduct;
    if (id == MINE) return new YourProduct;
        // Uwaga — identyfikatory YOURS i MINE zamieniono miejscami.

    if (id == THEIRS) return new TheirProduct;

    return Creator::Create(id); // Wywoływana, jeśli żaden z warunków nie jest spełniony.
}
```

Zauważmy, że ostatnim zadaniem wykonywanym przez tę operację jest wywołanie operacji `Create` z klasy nadrzędnej. Dzieje się tak, ponieważ operacja `MyCreator::Create` obsługuje w specyficzny sposób (inaczej niż klasa nadrzędna) jedynie identyfikatory `YOUR`, `MINE` i `THEIRS`. Inne klasy nie są tu uwzględniane. Dlatego klasa `MyCreator` *rozszerza* listę tworzonych produktów i przekazuje zadanie generowania większości z nich klasie nadrzędnej.

3. *Warianty i problemy specyficzne dla języka*. Z poszczególnymi językami programowania związane są inne ciekawe odmiany i zastrzeżenia.

   W programach w języku Smalltalk często używana jest metoda zwracająca klasę, której egzemplarz należy utworzyć. W metodzie wytwórczej w klasie `Creator` można wykorzystać tę wartość do utworzenia produktu, a klasa `ConcreteCreator` może przechowywać, a nawet obliczać tę wartość. W efekcie określanie typu tworzonego egzemplarza podklasy klasy `ConcreteProduct` ma miejsce jeszcze później.

   W napisanej w języku Smalltalk wersji przykładu dotyczącego klasy `Document` można w klasie `Application` zdefiniować metodę `documentClass`. Metoda ta powinna zwracać odpowiednią klasę `Document`, której egzemplarz należy utworzyć. Implementacja metody `documentClass` w klasie `MyApplication` zwraca klasę `MyDocument`. Dlatego w klasie `Application` należy umieścić następujący kod:

   ```
   clientMethod
       document := self documentClass new.

   documentClass
       self subclassResponsibility
   ```

   Klasa `MyApplication` obejmuje poniższy kod:

   ```
   documentClass
       ^ MyDocument
   ```

   Ten fragment zwraca do klasy `Application` klasę `MyDocument`, której egzemplarz należy utworzyć.

   Jeszcze elastyczniejsze rozwiązanie, zbliżone do sparametryzowanych metod wytwórczych, polega na przechowywaniu klasy tworzonych obiektów w zmiennej statycznej klasy `Application`. Pozwala to uniknąć tworzenia podklasy klasy `Application` w celu zmodyfikowania produktu.

   Metody wytwórcze w języku C++ zawsze są funkcjami wirtualnymi (często czysto wirtualnymi). Należy jednak zachować ostrożność i nie wywoływać metod wytwórczych w konstruktorze klasy `Creator`, ponieważ metoda wytwórcza klasy `ConcreteCreator` nie będzie wtedy jeszcze dostępna.

   Można uniknąć tego problemu dzięki zachowaniu staranności i korzystaniu z produktów wyłącznie za pośrednictwem akcesora tworzącego dany produkt na żądanie. W konstruktorze zamiast generować konkretny produkt, należy zainicjować go za pomocą wartości 0. Do zwrócenia produktu posłuży akcesor. Najpierw jednak sprawdzi, czy produkt istnieje, a jeśli nie — utworzy go. Ta technika jest czasem nazywana **leniwym inicjowaniem**. Poniższy kod ilustruje typową implementację tego rozwiązania.

```
class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};

Product* Creator::GetProduct () {
    if (_product = = 0 ) {
        _product = CreateProduct();
    }
    return _product;
}
```

4. *Wykorzystanie szablonów w celu uniknięcia tworzenia podklas*. Wspomnieliśmy już, że następnym potencjalnym problemem związanym z metodami wytwórczymi jest to, iż czasem trzeba utworzyć podklasę tylko w celu utworzenia odpowiednich obiektów `Product`. Inny sposób na poradzenie sobie z tą niedogodnością w języku C++ polega na udostępnieniu szablonu podklasy klasy `Creator` sparametryzowanego za pomocą klasy `Product`.

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

Dzięki temu szablonowi klient może podać samą klasę produktu — tworzenie podklasy klasy `Creator` nie jest konieczne.

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

5. *Konwencje nazewnicze*. Dobrym zwyczajem jest stosowanie konwencji nazewniczych wyraźnie wskazujących na zastosowanie metod wytwórczych. Na przykład w platformie MacApp [App89] (służy ona do tworzenia aplikacji na komputery Macintosh) operacja abstrakcyjna definiująca metodą wytwórczą zawsze deklarowana jest w postaci `Class* DoMakeClass()`, gdzie `Class` to nazwa klasy produktu.

PRZYKŁADOWY KOD

Funkcja `CreateMaze` (s. 90) tworzy i zwraca labirynt. Jeden ze związanych z nią problemów polega na tym, że zapisano w niej na stałe klasy labiryntu, pomieszczeń, drzwi i ścian. Zastosujemy metodę wytwórczą, aby umożliwić zmodyfikowanie tych komponentów w podklasach.

Najpierw zdefiniujmy metody wytwórcze w klasie `MazeGame`. Posłużą one do tworzenia obiektów reprezentujących labirynt, pomieszczenie, ścianę i drzwi.

```
class MazeGame {
public:
    Maze* CreateMaze();

// Metody wytwórcze:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Każda metoda wytwórcza zwraca komponent określonego rodzaju. Klasa `MazeGame` udostępnia implementację domyślną zwracającą labirynt, pomieszczenia, ściany i drzwi najprostszego rodzaju.

Teraz można zmodyfikować operację `CreateMaze` z wykorzystaniem metod wytwórczych.

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom (1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

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

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

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

    return aMaze;
}
```

W różnych grach można utworzyć podklasy klasy `MazeGame`, aby dodać wyspecjalizowane części labiryntu. W tych podklasach można przedefiniować niektóre (lub wszystkie) metody wytwórcze w celu określenia odmian produktów. Na przykład w klasie `BombedMazeGame` można umieścić nowe definicje produktów `Room` i `Wall`, tak aby metody zwracały wersje specyficzne dla labiryntu z bombami.

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

Podklasę `EnchantedMazeGame` można zdefiniować w następujący sposób:

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

ZNANE ZASTOSOWANIA

Metody wytwórcze są powszechnie stosowane w pakietach narzędziowych i platformach. Wcześniejszy przykład dotyczący dokumentu ilustruje typowe zastosowanie wzorca w platformach MacApp i ET++ [WGM88]. Przykład opisujący manipulatory pochodzi z platformy Unidraw.

Klasa `View` w architekturze MVC języka Smalltalk-80 obejmuje metodę `defaultController`. Tworzy ona kontroler, dlatego można traktować ją jak metodę wytwórczą [Par90]. Jednak w podklasach klasy `View` klasa ich kontrolera domyślnego jest określana za pomocą metody `defaultControllerClass` (zwraca ona klasę, której egzemplarz tworzy metoda `default↪Controller`). Dlatego prawdziwą metodą wytwórczą, czyli tą, którą należy przesłonić w podklasach, jest `defaultControllerClass`.

Bardziej wymyślny przykład z języka Smalltalk-80 to metoda wytwórcza `parserClass` zdefiniowana w klasie `Behavior` (jest to nadklasa wszystkich obiektów reprezentujących klasy). To rozwiązanie umożliwia klasom wykorzystanie niestandardowego parsera do analizy ich kodu źródłowego. W kliencie można na przykład zdefiniować klasę `SQLParser` do analizowania kodu źródłowego klasy z zagnieżdżonymi instrukcjami w języku SQL. Metoda `parserClass`

zaimplementowana w klasie `Behavior` zwraca standardową klasę `Parser` języka Smalltalk. W klasie z zagnieżdżonymi instrukcjami w języku SQL należy przesłonić tę metodę (jako metodę statyczną), tak aby zwracała klasę `SQLParser`.

W systemie Orbix ORB firmy IONA Technologies [ION94] wzorzec Metoda wytwórcza wykorzystano do generowania pełnomocników odpowiedniego typu (zobacz wzorzec Pełnomocnik, s. 191) w odpowiedzi na żądanie referencji do zdalnego obiektu. Metoda wytwórcza sprawia, że można łatwo zastąpić domyślnego pełnomocnika inną wersją, na przykład używającą pamięci podręcznej po stronie klienta.

POWIĄZANE WZORCE

Wzorzec Fabryka abstrakcyjna (s. 101) jest często implementowany za pomocą metod wytwórczych. Przykład w punkcie „Uzasadnienie" w opisie wzorca Fabryka abstrakcyjna ilustruje także zastosowania wzorca Metoda wytwórcza.

Metody wytwórcze zwykle wywołuje się w metodach szablonowych (s. 264). We wcześniejszym przykładzie dotyczącym dokumentu `NewDocument` to metoda szablonowa.

Prototypy (s. 120) nie wymagają tworzenia podklas klasy `Creator`, jednak często konieczne jest wtedy umieszczenie operacji `Initialize` w klasie `Product`. W klasie `Creator` operacja `Initialize` służy do inicjowania obiektu. Metoda wytwórcza nie wymaga stosowania takich operacji.

PROTOTYP (PROTOTYPE)

obiektowy, konstrukcyjny

PRZEZNACZENIE

Określa na podstawie prototypowego egzemplarza rodzaje tworzonych obiektów i generuje nowe obiekty przez kopiowanie tego prototypu.

UZASADNIENIE

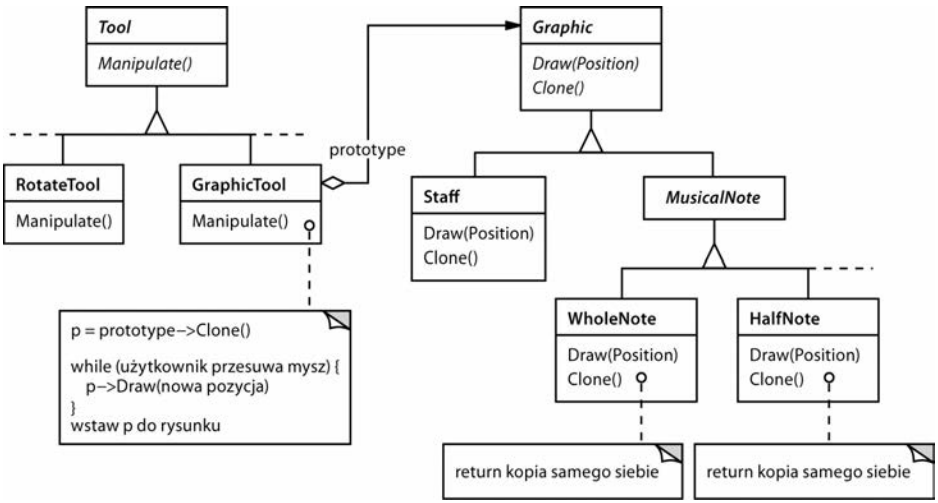
Można zbudować edytor partytur przez dostosowanie ogólnej platformy do tworzenia edytorów graficznych i dodanie nowych obiektów reprezentujących nuty, pauzy i pięciolinie. Platforma do tworzenia edytorów może udostępniać paletę narzędzi do dodawania takich obiektów muzycznych do partytury. W palecie mogą znaleźć się też narzędzia do zaznaczania i przenoszenia tych obiektów oraz manipulowania nimi. Użytkownik mógłby na przykład kliknąć narzędzie związane z ćwierćnutami i użyć go do dodania ćwierćnut do partytury. Mógłby też wykorzystać narzędzie do przenoszenia, aby przesunąć nutę w górę lub w dół na pięciolinii i zmienić w ten sposób wysokość dźwięku.

Załóżmy, że omawiana platforma udostępnia klasę abstrakcyjną `Graphic` reprezentującą komponenty graficzne, takie jak nuty i pięciolinie. Ponadto obejmuje klasę abstrakcyjną `Tool`, która służy do definiowania w palecie narzędzi podobnych do tych omówionych wcześniej. W platformie należy ponadto zdefiniować podklasę `GraphicTool` reprezentującą narzędzia do tworzenia egzemplarzy obiektów graficznych i dodawania ich do dokumentu.

Jednak klasa `GraphicTool` może sprawić problemy projektantowi platformy. Klasy reprezentujące nuty i pięciolinie są specyficzne dla aplikacji, natomiast klasa `GraphicTool` należy do platformy, dlatego nie potrafi tworzyć egzemplarzy klas muzycznych dodawanych do partytury. Moglibyśmy przygotować podklasę klasy `GraphicTool` dla obiektu muzycznego każdego rodzaju, jednak prowadzi to do utworzenia wielu podklas różniących się jedynie rodzajem generowanego obiektu. Wiemy, że składanie obiektów to elastyczna alternatywa dla tworzenia podklas. Pozostaje jednak pytanie, jak wykorzystać tę technikę w platformie, aby sparametryzować egzemplarze klasy `GraphicTool` za pomocą *klasy* z rodziny `Graphic`, której egzemplarz należy utworzyć.

Rozwiązanie polega na tworzeniu przez klasę `GraphicTool` nowego obiektu `Graphic` przez kopiowanie (lub klonowanie) egzemplarza podklasy klasy `Graphic`. Taki egzemplarz nazywamy **prototypem**. Klasa `GraphicTool` jest sparametryzowana za pomocą prototypu, który powinna sklonować i dodać do dokumentu. Jeśli wszystkie podklasy klasy `Graphic` obsługują operację `Clone`, klasa `GraphicTool` może sklonować dowolny obiekt z rodziny klas `Graphic`.

Dlatego w edytorze utworów muzycznych każde narzędzie do tworzenia obiektów muzycznych jest egzemplarzem klasy `GraphicTool` zainicjowanym za pomocą innego prototypu. Każdy egzemplarz klasy `GraphicTool` tworzy obiekt muzyczny przez klonowanie jego prototypu, a następnie dodaje kopię do partytury.



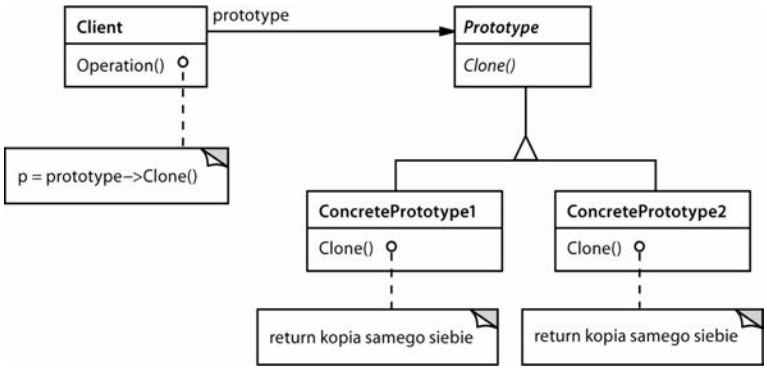
Możemy użyć wzorca Prototyp, aby jeszcze bardziej zmniejszyć liczbę klas. Odrębne klasy reprezentują całe nuty i półnuty, choć prawdopodobnie nie jest to konieczne. Zamiast tego można użyć egzemplarzy tej samej klasy zainicjowanych za pomocą różnych bitmap i czasu trwania dźwięku. Narzędzie do tworzenia całych nut będzie obiektem `GraphicTool`, którego prototyp to obiekt `MusicalNote` zainicjowany w taki sposób, aby reprezentował całą nutę. Pozwala to znacznie ograniczyć liczbę klas w systemie, a ponadto ułatwia dodawanie do edytora nut nowego rodzaju.

WARUNKI STOSOWANIA

Ze wzorca Prototyp należy korzystać, jeśli system powinien być niezależny od sposobu tworzenia, składania i reprezentowania produktów *oraz*

- klasy tworzonych egzemplarzy są określane w czasie wykonywania programu (na przykład przez dynamiczne wczytywanie) *lub*
- programista chce uniknąć tworzenia hierarchii klas fabryk odpowiadającej hierarchii klas produktów *lub*
- egzemplarze klasy mogą przyjmować jeden z niewielu stanów; wygodniejsze może wtedy okazać się dodanie odpowiedniej liczby prototypów i klonowanie ich zamiast ręcznego tworzenia egzemplarzy klasy (za każdym razem z właściwym stanem).

STRUKTURA



ELEMENTY

- **`Prototype`** (`Graphic`):
  - obejmuje deklarację interfejsu do klonowania.
- **`ConcretePrototype`** (`Staff`, `WholeNote`, `HalfNote`):
  - obejmuje implementację procesu klonowania.
- **`Client`** (`GraphicTool`)
  - tworzy nowy obiekt przez zażądanie od prototypu sklonowania się.

WSPÓŁDZIAŁANIE

- Klient żąda od prototypu, aby ten się sklonował.

KONSEKWENCJE

Konsekwencje zastosowania Prototypu w dużej części pokrywają się ze skutkami użycia wzorców Fabryka abstrakcyjna (s. 101) i Budowniczy (s. 92). Wzorzec ten ukrywa klasy konkretne produktów przed klientem, zmniejszając w ten sposób liczbę nazw znanych klientom. Ponadto wzorce te umożliwiają klientowi korzystanie z klas specyficznych dla aplikacji bez konieczności modyfikowania go.

Poniżej wymieniamy dodatkowe korzyści płynące z zastosowania wzorca Prototyp:

1. *Możliwość dodawania i usuwania produktów w czasie wykonywania programu*. Prototypy umożliwiają dołączenie do systemu nowej klasy konkretnej produktu przez samo zarejestrowanie prototypowego egzemplarza w kliencie. Zapewnia to nieco większą elastyczność niż inne wzorce konstrukcyjne, ponieważ klient może instalować i usuwać prototypy w czasie wykonywania programu.
2. *Możliwość określania nowych obiektów przez zmianę wartości*. Wysoce dynamiczne systemy umożliwiają definiowanie nowych zachowań przez składanie obiektów — na przykład za pomocą określania wartości zmiennych — a nie przez definiowanie nowych klas. Nowe

rodzaje obiektów można definiować przez tworzenie egzemplarzy istniejących klas i rejestrowanie tych egzemplarzy jako prototypów obiektów klienta. Klient może wykorzystać nowe zachowanie przez oddelegowanie zadania do prototypu.

Projekt tego rodzaju umożliwia użytkownikom definiowanie nowych „klas" bez programowania. Klonowanie prototypu przypomina w swej istocie tworzenie egzemplarza klasy. Wzorzec Prototyp pozwala znacznie zmniejszyć liczbę klas potrzebnych w systemie. W edytorze utworów muzycznych jedna klasa `GraphicTool` może utworzyć nieskończoną liczbę różnorodnych obiektów muzycznych.

3. *Możliwość określania nowych obiektów przez modyfikowanie struktury*. Wiele aplikacji tworzy obiekt z mniej i bardziej złożonych części. Na przykład w edytorach do projektowania obwodów elektrycznych można budować takie struktury z podukładów[1]. Dla wygody aplikacje tego typu często umożliwiają tworzenie egzemplarzy złożonych struktur zdefiniowanych przez użytkownika, na przykład w celu wielokrotnego wykorzystania specyficznego podukładu.

   Wzorzec Prototyp obsługuje także to rozwiązanie. Wystarczy dodać określony podukład jako prototyp do palety dostępnych składników układów. Jeśli w składanym obiekcie obwodu operacja `Clone` jest zaimplementowana z wykorzystaniem głębokiego kopiowania, układy o różnych strukturach można tworzyć jako prototypy.

4. *Zmniejszenie liczby podklas*. Wzorzec Metoda wytwórcza (s. 110) często powoduje utworzenie hierarchii klasy `Creator` odpowiadającej hierarchii klasy produktu. Wzorzec Prototyp pozwala sklonować prototyp, zamiast żądać od metody wytwórczej utworzenia nowego obiektu. Dlatego hierarchia klasy `Creator` w ogóle nie jest potrzebna. Ta zaleta dotyczy głównie języków podobnych do C++, w których klasy nie są traktowane jak standardowe obiekty. W językach, które obsługują obiekty reprezentujące klasy (na przykład w językach Smalltalk i Objective C), te korzyści są mniejsze, ponieważ zawsze można użyć takiego obiektu do tworzenia klas. Obiekty reprezentujące klasy działają w tych językach jak prototypy.

5. *Możliwość dynamicznego konfigurowania aplikacji za pomocą klas*. Niektóre środowiska uruchomieniowe umożliwiają dynamiczne wczytywanie klas do aplikacji. Wzorzec Prototyp to klucz do wykorzystania takich mechanizmów w językach podobnych do C++.

   Aplikacja, w której programista chce tworzyć egzemplarze dynamicznie wczytywanej klasy, nie będzie mogła statycznie wskazać jej konstruktora. Zamiast tego środowisko uruchomieniowe automatycznie utworzy egzemplarz każdej klasy w czasie jej wczytywania i zarejestruje ten egzemplarz za pomocą menedżera prototypów (zobacz punkt „Implementacja"). Następnie aplikacja może zażądać od menedżera prototypów egzemplarzy nowo wczytanych klas, które początkowo nie były dołączone do programu. Rozwiązanie to wykorzystano w systemie uruchomieniowym platformy do tworzenia aplikacji ET++ [EGM88].

Główną wadą wzorca Prototyp jest to, że w każdej podklasie klasy `Prototype` trzeba zaimplementować operację `Clone`. Może to sprawiać problemy. Dodanie tej operacji jest trudne, jeśli dane klasy już istnieją. Zaimplementowanie operacji `Clone` może okazać się skomplikowane także wtedy, jeżeli w tych klasach używane są obiekty nieobsługujące kopiowania lub mające referencje cykliczne.

[1] Takie aplikacje ilustrują zastosowanie wzorców Kompozyt (s. 170) i Dekorator (s. 152).

IMPLEMENTACJA

Wzorzec Prototyp jest szczególnie przydatny w językach ze statyczną kontrolą typów, na przykład w C++, gdzie klasy nie są obiektami i w czasie wykonywania programu dostępnych jest niewiele informacji o typie (lub w ogóle ich brak). Wzorzec ten ma mniejsze znaczenie w takich językach, jak Smalltalk lub Objective C, ponieważ udostępniają one do tworzenia egzemplarzy każdej klasy strukturę o możliwościach prototypu (chodzi tu o obiekt klasy). Omawiany wzorzec wbudowano w języki oparte na prototypach, takie jak Self [US87], w których tworzenie obiektów zawsze odbywa się przez klonowanie prototypu.

W czasie implementowania prototypów należy rozważyć następujące kwestie:

1. *Korzystanie z menedżera prototypów*. Jeśli liczba prototypów w systemie nie jest stała (ponieważ można je dynamicznie tworzyć i usuwać), należy przechowywać rejestr dostępnych prototypów. Klienty nie będą wtedy samodzielnie zarządzać prototypami, ale mogą zapisywać je w archiwum i stamtąd pobierać. Klient przed sklonowaniem prototypu musi wtedy zażądać udostępnienia go przez rejestr. To archiwum nazywamy **menedżerem prototypów**.

   Menedżer prototypów to struktura asocjacyjna zwracająca prototyp pasujący do podanego klucza. Udostępnia operacje do rejestrowania prototypów za pomocą klucza i wyrejestrowywania ich. Klienty mogą w czasie wykonywania programu modyfikować rejestr, a nawet przeglądać go. Umożliwia to rozszerzanie i sprawdzanie zawartości systemu bez konieczności pisania kodu.

2. *Implementowanie operacji* `Clone`. Najtrudniejszym aspektem stosowania wzorca Prototyp jest właściwe zaimplementowanie operacji `Clone`. Jest to szczególnie skomplikowane, jeśli struktury obiektu obejmują referencje cykliczne.

   Większość języków udostępnia pewne mechanizmy do klonowania obiektów. Na przykład język Smalltalk obejmuje implementację metody `copy` dziedziczoną we wszystkich podklasach klasy `Object`. Język C++ udostępnia konstruktor kopiujący. Jednak mechanizmy te nie rozwiązują problemu płytkiego i głębokiego kopiowania [GR83] związanego z tym, czy klonowanie obiektu spowoduje skopiowanie zmiennych egzemplarza, czy klon i oryginał będą jedynie współużytkować te zmienne.

   Płytka kopia jest prosta i zwykle wystarczająca. W języku Smalltalk jest ona tworzona domyślnie. Domyślny konstruktor kopiujący w języku C++ przeprowadza kopiowanie poszczególnych składowych, co oznacza, że wskaźniki będą współużytkowane przez kopię i oryginał. Jednak klonowanie prototypów o złożonych strukturach zwykle wymaga utworzenia głębokiej kopii, ponieważ klon i oryginał muszą być niezależne od siebie. Dlatego trzeba zagwarantować, że komponenty klonu to kopie komponentów prototypu. Klonowanie wymaga zadecydowania, co (jeśli cokolwiek) będzie współużytkowane.

   Jeżeli obiekty w systemie udostępniają operacje `Save` i `Load`, można je wykorzystać do utworzenia domyślnej implementacji operacji `Clone`, która po prostu zapisuje obiekt i natychmiast ponownie go wczytuje. Operacja `Save` zapisuje obiekt do bufora w pamięci, a operacja `Load` tworzy duplikat przez odtworzenie danego obiektu na podstawie danych z bufora.

3. *Inicjowanie klonów*. Choć niektóre klienty bez problemów korzystają z klonu w jego pierwotnej postaci, w innych pożądane jest zainicjowanie części lub całości wewnętrznego stanu klonu wybranymi wartościami. Zwykle wartości tych nie można przekazać do operacji `Clone`, ponieważ ich liczba jest inna w zależności od klasy prototypu. Niektóre prototypy wymagają wielu parametrów inicjujących, a inne wcale ich nie potrzebują. Przekazywanie parametrów do operacji `Clone` narusza jednolity interfejs klonowania.

   Może się zdarzyć, że w klasach prototypów zdefiniowane są operacje do ustawiania lub modyfikowania kluczowych składników stanu. Jeśli tak jest, klienty mogą wywołać te operacje bezpośrednio po klonowaniu. W przeciwnym razie konieczne może być wprowadzenie operacji `Initialize` (zobacz punkt „Przykładowy kod"). Powinna ona przyjmować parametr inicjujący i na jego podstawie ustawiać wewnętrzny stan klonu. Należy zachować szczególną ostrożność przy korzystaniu z operacji `Clone` przeprowadzających głębokie kopiowanie. Utworzone przez nie kopie czasem trzeba usunąć (albo bezpośrednio, albo w operacji `Initialize`) przed ich ponownym zainicjowaniem.

PRZYKŁADOWY KOD

Zdefiniujemy tu podklasę `MazePrototypeFactory` klasy `MazeFactory` (s. 106). Do inicjowania klasy `MazePrototypeFactory` posłużą prototypy obiektów, które ma ona utworzyć, dlatego nie trzeba będzie tworzyć jej podklasy tylko po to, aby zmienić generowane w niej ściany lub pomieszczenia.

W klasie `MazePrototypeFactory` wzbogaciliśmy interfejs klasy `MazeFactory` o konstruktor przyjmujący argumenty w postaci prototypów:

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

Nowy konstruktor jedynie inicjuje prototypy:

```
MazePrototypeFactory::MazePrototypeFactory (
    Maze* m, Wall* w, Room* r, Door* d
) {
    _prototypeMaze = m;
    _prototypeWall = w;
    _prototypeRoom = r;
    _prototypeDoor = d;
}
```

Funkcje składowe służące do tworzenia ścian, pomieszczeń i drzwi wyglądają podobnie. Każda z nich klonuje prototyp, a następnie go inicjuje. Oto definicje funkcji `MakeWall` i `MakeDoor`:

```
Wall* MazePrototypeFactory::MakeWall () const {
    return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
    Door* door = _prototypeDoor->Clone();
    door->Initialize(r1, r2);
    return door;
}
```

Klasę `MazePrototypeFactory` możemy wykorzystać do tworzenia prototypowego (domyślnego) labiryntu przez zainicjowanie jej za pomocą prototypów podstawowych komponentów labiryntu.

```
MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

Aby zmienić rodzaj labiryntu, należy zainicjować klasę `MazePrototypeFactory` za pomocą innego zestawu prototypów. Poniższe wywołanie tworzy labirynt z obiektami `BombedDoor` i `RoomWithABomb`.

```
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
) ;
```

Obiekt, który można zastosować jako prototyp (na przykład egzemplarz klasy `Wall`), musi obsługiwać operację `Clone`. Musi też posiadać konstruktor kopiujący potrzeby do klonowania. Czasem potrzebna jest ponadto odrębna operacja do ponownego inicjowania wewnętrznego stanu. Dodajmy do klasy `Door` operację `Initialize`, aby umożliwić klientom inicjowanie pomieszczeń klonu.

Warto porównać poniższą definicję klasy `Door` do kodu ze strony 89.

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&);

    virtual void Initialize(Room*, Room*);
    virtual Door* Clone() const;
    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};
```

```
Door::Door (const Door& other) {
    _room1 = other._room1;
    _room2 = other._room2;
}

void Door::Initialize (Room* r1, Room* r2) {
    _room1 = r1;
    _room2 = r2;
}

Door* Door::Clone () const {
    return new Door(*this);
}
```

W podklasie `BombedWall` trzeba przesłonić operację `Clone` i zaimplementować odpowiedni konstruktor kopiujący.

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&);

    virtual Wall* Clone() const;
    bool HasBomb();
private:
    bool _bomb;
};

BombedWall::BombedWall (const BombedWall& other) : Wall(other) {
    _bomb = other._bomb;
}

Wall* BombedWall::Clone () const {
    return new BombedWall(*this);
}
```

Choć operacja `BombedWall::Clone` zwraca wskaźnik `Wall*`, w implementacji tej operacji zwracany jest wskaźnik do nowego egzemplarza podklasy — `BombedWall*`. Zdefiniowaliśmy operację `Clone` w klasie bazowej w ten sposób, aby zagwarantować, że klienty klonujące prototyp nie będą potrzebowały informacji o podklasach konkretnych. W klientach nigdy nie powinno być konieczne rzutowanie wartości zwróconej przez operację `Clone` w dół (do pożądanego typu).

W języku Smalltalk do sklonowania dowolnego obiektu z rodziny `MapSite` można powtórnie wykorzystać standardową metodę `copy` odziedziczoną po klasie `Object`. Klasy `MazeFactory` można użyć do wytwarzania potrzebnych prototypów. Na przykład aby utworzyć pomieszczenie, należy podać nazwę `#room`. Klasa `MazeFactory` obejmuje słownik z odwzorowaniami nazw na prototypy. Metoda `make:` w tej klasie wygląda tak:

```
make: partName
    ^ (partCatalog at: partName) copy
```

Po utworzeniu odpowiednich metod do inicjowania klasy `MazeFactory` za pomocą prototypów można zbudować prosty labirynt przy użyciu poniższego kodu:

```
CreateMaze
    on: (MazeFactory new
        with: Door new named: #door;
        with: Wall new named: #wall;
        with: Room new named: #room;
        yourself)
```

W tym rozwiązaniu definicja metody statycznej `on:` dla metody `CreateMaze` może wyglądać tak:

```
on: aFactory
    | room1 room2 |
    room1 := (aFactory make: #room) location: 1@1.
    room2 := (aFactory make: #room) location: 2@1.
    door := (aFactory make: #door) from: room1 to: room2.

    room1
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: door;
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: (aFactory make: #wall).
    room1
        atSide: #north put: (aFactory make: #wall);
        atSide: #east put: (aFactory make: #wall);
        atSide: #south put: (aFactory make: #wall);
        atSide: #west put: door.
    ^ Maze new
        addRoom: room1;
        addRoom: room2;
        yourself
```

ZNANE ZASTOSOWANIA

Prawdopodobnie pierwszym przykładem zastosowania wzorca Prototyp był system Sketchpad Ivana Sutherlanda [Sut63]. Pierwszym powszechnie znanym przypadkiem wykorzystania tego wzorca w języku obiektowym był system ThingLab. Umożliwiał on użytkownikom tworzenie kompozytów, a następnie korzystanie z nich jak z prototypów przez zainstalowanie ich w bibliotece obiektów wielokrotnego użytku [Bor81]. Goldberg i Robson wspomnieli o prototypach jako o wzorcu [GR83], jednak dużo pełniejszy opis przedstawił Coplien [Cop92]. Wyjaśnił on idiomy języka C++ związane ze wzorcem Prototyp oraz przedstawił wiele przykładów i wersji tego wzorca.

Etgdb to oparty na platformie ET++ fronton debugerów zapewniający interfejs graficzny dla różnych debugerów działających z poziomu wiersza poleceń. Dla każdego debugera istnieje odpowiednia podklasa `DebuggerAdaptor`. Na przykład klasa `GdbAdaptor` przystosowuje narzędzie etgdb do składni debugera GNU gdb, natomiast klasa `SunDbxAdaptor` robi to samo na potrzeby debugera dgx firmy Sun. W etgdb nie ma zapisanego na stałe zestawu klas `Debugger ↪Adaptor`. Zamiast tego narzędzie wczytuje nazwę adaptera ze zmiennej środowiskowej,

wyszukuje w tabeli globalnej prototyp o określonej nazwie, a następnie klonuje go. Do etgdb można dodawać obsługę nowych debugerów przez powiązanie ich z klasą `DebuggerAdaptor` specyficzną dla danego debugera.

„Biblioteka technik interakcji" w aplikacji Mode Composer obejmuje prototypy obiektów obsługujących różne metody interakcji [Sha90]. Każdą technikę utworzoną przez ten program można wykorzystać jako prototyp przez umieszczenie jej w bibliotece. Wzorzec Prototyp umożliwia obsługę w aplikacji Mode Composer nieograniczonego zbioru technik interakcji.

Opisany wcześniej przykład dotyczący edytora utworów muzycznych oparliśmy na platformie do edycji graficznej Unidraw [VL90].

POWIĄZANE WZORCE

Prototyp i Fabryka abstrakcyjna (s. 101) to pod niektórymi względami „konkurencyjne" wzorce (zagadnienie to omawiamy w końcowej części rozdziału). Można ich jednak używać wspólnie. Fabryka abstrakcyjna może przechowywać zestaw prototypów stosowanych do klonowania i zwracania obiektów-produktów.

Zastosowanie wzorca Prototyp może być korzystne także w tych projektach, w których w wielu miejscach wykorzystano wzorce Kompozyt (s. 170) i Dekorator (s. 152).

SINGLETON (SINGLETON)

obiektowy, konstrukcyjny

PRZEZNACZENIE

Gwarantuje, że klasa będzie miała tylko jeden egzemplarz, i zapewnia globalny dostęp do niego.

UZASADNIENIE

W przypadku niektórych klas ważne jest, aby miały one tylko jeden egzemplarz. Choć w systemie może działać wiele drukarek, powinien znajdować się w nim tylko jeden program buforujący drukowania. Potrzebny jest tylko jeden system plików i menedżer okien, filtr cyfrowy powinien mieć tylko jeden konwerter analogowy-cyfrowy, a system rozliczeniowy powinien być przeznaczony do obsługi tylko jednej firmy.

Jak można zagwarantować, że klasa będzie miała tylko jeden łatwo dostępny egzemplarz? Zmienna globalna zapewnia dostęp do obiektu, jednak pozostawia możliwość utworzenia wielu obiektów.

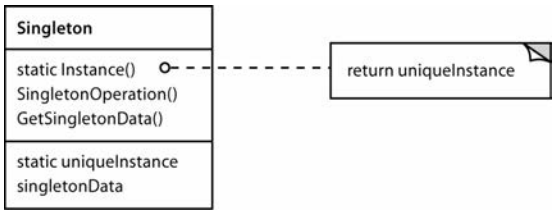
Lepsze rozwiązanie polega na przydzieleniu klasie zadania śledzenia swojego jedynego egzemplarza. Klasa może zagwarantować (przez przechwytywanie żądań utworzenia nowych obiektów), że nie powstanie żaden inny jej egzemplarz, a także może umożliwiać dostęp do jedynego egzemplarza. Tak właśnie działa wzorzec Singleton.

WARUNKI STOSOWANIA

Wzorca Singleton należy używać w następujących warunkach:

- Jeśli musi istnieć dokładnie jeden egzemplarz klasy dostępny klientom w znanym miejscu.
- Kiedy potrzebna jest możliwość rozszerzania jedynego egzemplarza przez tworzenie podklas, a klienty powinny móc korzystać ze wzbogaconego egzemplarza bez konieczności wprowadzania zmian w ich kodzie.

STRUKTURA



ELEMENTY

- **`Singleton`:**
  - definiuje operację `Instance` umożliwiającą klientom dostęp do niepowtarzalnego egzemplarza klasy; `Instance` to operacja statyczna (czyli metoda statyczna w języku Smalltalk lub statyczna funkcja składowa w języku C++);
  - może odpowiadać za tworzenie własnego niepowtarzalnego egzemplarza.

WSPÓŁDZIAŁANIE

- Klienty mogą uzyskać dostęp do egzemplarza klasy `Singleton` wyłącznie poprzez operację `Instance` z tej klasy.

KONSEKWENCJE

Wzorzec Singleton zapewnia kilka korzyści:

1. *Zapewnia kontrolę dostępu do jedynego egzemplarza*. Ponieważ klasa `Singleton` kapsułkuje swój jedyny egzemplarz, można w niej ściśle kontrolować, w jaki sposób i kiedy klienty mogą uzyskać do niego dostęp.
2. *Pozwala zmniejszyć przestrzeń nazw*. Wzorzec Singleton jest ulepszeniem w porównaniu do zmiennych globalnych. Pozwala uniknąć zaśmiecania przestrzeni nazw zmiennymi globalnymi przechowującymi jedyne egzemplarze.
3. *Umożliwia dopracowywanie operacji i reprezentacji*. Można tworzyć podklasy klasy `Singleton`, a ponadto łatwo jest skonfigurować aplikację za pomocą egzemplarza takiej rozszerzonej klasy. Potrzebną do tego klasę można podać w czasie wykonywania programu.
4. *Umożliwia określenie dowolnego limitu liczby egzemplarzy*. Omawiany wzorzec umożliwia łatwą zmianę podejścia i zezwolenie na tworzenie więcej niż jednego egzemplarza klasy `Singleton`. Ponadto to samo rozwiązanie można zastosować do kontrolowania liczby egzemplarzy używanych w aplikacji. Trzeba wtedy zmodyfikować jedynie operację, która zapewnia dostęp do egzemplarza klasy `Singleton`.
5. *Jest bardziej elastyczny od operacji statycznych*. Inny sposób na opakowanie funkcji singletonu polega na wykorzystaniu operacji statycznych (czyli statycznych funkcji składowych w języku C++ lub metod statycznych w języku Smalltalk). Jednak obie te techniki utrudniają zmianę projektu tak, aby umożliwić tworzenie więcej niż jednego egzemplarza klasy. Ponadto statyczne funkcje składowe w języku C++ nigdy nie są wirtualne, dlatego w podklasach nie można przesłonić ich w sposób polimorficzny.

IMPLEMENTACJA

Oto kwestie związane z implementacją, które należy rozważyć przy stosowaniu wzorca Singleton:

1. *Zapewnianie niepowtarzalności egzemplarza*. We wzorcu Singleton jedyny egzemplarz jest zwykłym egzemplarzem klasy, jednak jest ona napisana tak, aby można utworzyć tylko ten egzemplarz. Standardowe rozwiązanie polega na ukryciu operacji tworzącej egzemplarz w operacji statycznej (czyli statycznej funkcji składowej lub metodzie statycznej), która

gwarantuje, że może powstać tylko jeden egzemplarz danej klasy. Ta operacja ma dostęp do zmiennej przechowującej ów egzemplarz, a zanim zwróci jej wartość, upewnia się, że zmienna została zainicjowania za pomocą niepowtarzalnego egzemplarza. To podejście gwarantuje, że singleton zostanie utworzony i zainicjowany przed jego pierwszym użyciem.

W języku C++ opisaną operację statyczną można zdefiniować jako statyczną funkcję składową `Instance` klasy `Singleton`. W klasie tej należy też zdefiniować statyczną zmienną składową `_instance` zawierającą wskaźnik do niepowtarzalnego egzemplarza.

Deklaracja klasy `Singleton` wygląda tak:

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

A oto jej implementacja:

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Klienty mogą uzyskać dostęp do singletonu wyłącznie poprzez funkcję składową `Instance`. Zmienna `_instance` jest inicjowana wartością 0, a statyczna funkcja składowa `Instance` zwraca wartość tej zmiennej (przy czym jeśli wartość ta wynosi 0, funkcja inicjuje zmienną za pomocą niepowtarzalnego egzemplarza). W funkcji `Instance` zastosowano leniwe inicjowanie. Wartość, którą zwraca ta funkcja, nie jest tworzona ani zapisywana do momentu pierwszego wywołania funkcji.

Warto zauważyć, że konstruktor jest chroniony. Jeśli klient spróbuje bezpośrednio utworzyć egzemplarz klasy `Singleton`, w czasie kompilacji pojawi się komunikat o błędzie. Gwarantuje to, że powstanie tylko jeden egzemplarz tej klasy.

Ponadto z uwagi na to, że zmienna `_instance` to wskaźnik do obiektu `Singleton`, funkcja składowa `Instance` może przypisać do tej zmiennej wskaźnik do obiektu podklasy klasy `Singleton`. Przykład takiego rozwiązania przedstawiamy w punkcie „Przykładowy kod”.

Warto zwrócić uwagę na jeszcze jeden aspekt przedstawionej implementacji w języku C++. Nie wystarczy zdefiniować singletonu jako obiektu globalnego lub statycznego i zdać się na inicjowanie automatyczne. Wynika to z trzech powodów:

a) Nie można zagwarantować, że zadeklarowany zostanie tylko jeden egzemplarz obiektu statycznego.

b) Możemy nie mieć informacji wystarczających do utworzenia egzemplarza każdego singletonu w czasie statycznego inicjowania. Niezbędne mogą być wartości obliczane w późniejszej fazie działania programu.

c) W języku C++ kolejność wywoływania konstruktorów obiektów globalnych z różnych jednostek translacji nie jest określona [ES90]. Oznacza to, że między singletonami nie mogą występować zależności. Jeśli warunek ten nie będzie spełniony, z pewnością pojawią się błędy.

Dodatkową (choć niewielką) wadą podejścia opartego na obiektach globalnych lub statycznych jest to, że trzeba utworzyć wszystkie singletony niezależnie od tego, czy są używane czy nie. Zastosowanie statycznej funkcji składowej pozwala uniknąć wszystkich tych problemów.

W języku Smalltalk funkcja zwracająca niepowtarzalny egzemplarz jest implementowana jako metoda statyczna w klasie `Singleton`. Aby zagwarantować, że powstanie tylko jeden jej egzemplarz, należy przesłonić operację `new`. Utworzona w ten sposób klasa `Singleton` może obejmować dwie poniższe metody statyczne (`SoleInstance` to zmienna statyczna używana wyłącznie w tym miejscu):

```
new
    self error: 'Nie można utworzyć nowego obiektu'

default
    Solelnstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

2. *Tworzenie podklas klasy `Singleton`*. Największy problem jest związany nie tyle z definiowaniem podklas, ile z instalowaniem niepowtarzalnych egzemplarzy, aby klienty mogły z nich korzystać. Rozwiązanie sprowadza się do tego, że zmienną wskazującą na egzemplarz singletonu trzeba zainicjować za pomocą egzemplarza odpowiedniej podklasy. Najprostsza technika, która to umożliwia, polega na określeniu potrzebnego singletonu w operacji `Instance` klasy `Singleton`. W przykładzie w punkcie „Przykładowy kod" pokazaliśmy, jak zastosować tę technikę za pomocą zmiennych środowiskowych.

   Inny sposób wybierania podklasy klasy `Singleton` polega na umieszczeniu implementacji operacji `Instance` poza klasą nadrzędną (na przykład `MazeFactory`) — w podklasie. Umożliwia to programiście języka C++ ustalenie klasy singletonu w czasie konsolidacji (na przykład przez dołączenie pliku wynikowego zawierającego inną implementację), jednak ukrywa tę klasę przed klientami korzystającymi z singletonu.

   W podejściu opartym na dołączaniu klasa singletonu jest ustalana w czasie konsolidacji, co utrudnia wskazanie takiej klasy w czasie wykonywania programu. Wykorzystanie instrukcji warunkowych do określania podklasy to elastyczniejsze rozwiązanie, jednak powoduje zapisanie na stałe zestawu dostępnych klas singletonów. Żadne z tych podejść nie zapewnia wystarczającej elastyczności w każdych warunkach.

   Elastyczniejsze podejście polega na zastosowaniu **rejestru singletonów**. Zamiast definiować zestaw dostępnych klas singletonów w operacji `Instance`, można nakazać takim klasom rejestrowanie egzemplarzy singletonów w znanym rejestrze za pomocą nazw.

Rejestr łączy nazwy w postaci łańcuchów znaków z singletonami. Kiedy operacja `Instance` potrzebuje singletonu, korzysta z rejestru, żądając singletonu przez podanie jego nazwy. Rejestr wyszukuje wtedy odpowiedni singleton (jeśli taki istnieje) i zwraca go. W tym rozwiązaniu nie trzeba zapisywać w operacji `Instance` wszystkich dostępnych klas lub egzemplarzy singletonów. Wystarczy we wszystkich klasach singletonów umieścić wspólny interfejs z operacjami do obsługi rejestru:

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

Operacja `Register` rejestruje egzemplarz singletonu pod podaną nazwą. Aby nie komplikować rejestru, umieścimy w nim listę obiektów `NameSingletonPair`. Każdy taki obiekt odwzorowuje nazwę na singleton. Operacja `Lookup` wyszukuje singleton na podstawie jego nazwy. Zakładamy, że nazwę potrzebnego singletonu określa zmienna środowiskowa.

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // Użytkownik lub środowisko podaje tę wartość w czasie uruchamiania programu.

        _instance = Lookup(singletonName);
        // Operacja Lookup zwraca 0, jeśli podany singleton nie istnieje.
    }
    return _instance;
}
```

W którym miejscu klasy singletonów się rejestrują? Jedną z możliwości jest użycie do tego konstruktora. Na przykład w podklasie `MySingleton` można wykonać następującą operację:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

Konstruktor oczywiście zostanie wywołany dopiero w momencie tworzenia egzemplarza danej klasy. W ten sposób wracamy do problemu, który wzorzec Singleton miał rozwiązać. W języku C++ można sobie z nim poradzić przez zdefiniowanie statycznego egzemplarza klasy `MySingleton`. Możemy na przykład dodać poniższy kod:

```
static MySingleton theSingleton;
```

Należy go umieścić w pliku z implementacją klasy `MySingleton`.

W tym rozwiązaniu klasa `Singleton` nie odpowiada za tworzenie singletonu. Teraz jej głównym zadaniem jest udostępnianie w systemie wybranego obiektu singletonu. Podejście oparte na statycznym obiekcie nadal ma pewną wadę — trzeba utworzyć egzemplarze wszystkich możliwych podklas klasy `Singleton`, ponieważ w przeciwnym razie singletony nie zostaną zarejestrowane.

PRZYKŁADOWY KOD

Załóżmy, że zdefiniowaliśmy klasę do tworzenia labiryntów, `MazeFactory`, w sposób opisany na stronie 106. Klasa ta określa interfejs do tworzenia różnych części labiryntu. W podklasach można umieścić nowe definicje operacji, aby zwracały egzemplarze wyspecjalizowanych klas produktów (na przykład obiekty `BombedWall` zamiast `Wall`).

Ważne jest to, że w aplikacji Labirynt potrzebny jest tylko jeden egzemplarz klasy `MazeFactory`. Powinien być on dostępny w kodzie tworzącym poszczególne części labiryntu. W uzyskaniu tego efektu pomoże wzorzec Singleton. Przez utworzenie klasy `MazeFactory` jako singletonu można bez uciekania się do korzystania ze zmiennych globalnych sprawić, że obiekt reprezentujący labirynt będzie globalnie dostępny.

Dla uproszczenia przyjmijmy, że nie będziemy tworzyć podklas klasy `MazeFactory` (inną możliwość rozważymy za chwilę). W języku C++ można zdefiniować ją jako klasę typu singleton przez dodanie operacji statycznej `Instance` i zmiennej składowej `_instance` do przechowywania jedynego egzemplarza. Ponadto trzeba zadeklarować konstruktor jako chroniony, aby zapobiec przypadkowemu utworzeniu większej liczby egzemplarzy.

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // Tu należy umieścić istniejący interfejs.
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

A oto implementacja tej klasy:

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0 ) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Teraz rozważmy inny przypadek. Załóżmy, że klasa `MazeFactory` ma podklasy, a aplikacja musi wybrać jedną z nich. Rodzaj labiryntu wskażemy za pomocą zmiennej środowiskowej i dodamy kod, który utworzy egzemplarz odpowiedniej podklasy klasy `MazeFactory` na podstawie wartości tej zmiennej. Dobrym miejscem na ten kod jest operacja `Instance`, ponieważ tworzy ona egzemplarz klasy `MazeFactory`:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0 ) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
```

```
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // Inne dostępne podklasy.

        } else { // Klasa domyślna.
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Warto zauważyć, że po zdefiniowaniu każdej nowej podklasy klasy `MazeFactory` operację `Instance` trzeba zmodyfikować. W tej aplikacji nie sprawia to problemu, jednak może okazać się to trudne w przypadku fabryk abstrakcyjnych zdefiniowanych w platformie.

Możliwym rozwiązaniem jest zastosowanie podejścia opartego na rejestrze, które opisaliśmy w punkcie Implementacja. Dynamiczne dołączanie może być przydatne także w tym przypadku, ponieważ sprawia, że aplikacja nie musi wczytywać nieużywanych podklas.

ZNANE ZASTOSOWANIA

Przykładem zastosowania wzorca Singleton w języku Smalltalk-80 [Par90] jest zbiór zmian w kodzie — `ChangeSet current`. Bardziej wyrafinowanym przykładem jest relacja między klasami i ich **metaklasami**. Metaklasa to klasa reprezentująca klasę. Każda metaklasa ma jeden egzemplarz. Metaklasy nie mają nazw (są określane pośrednio poprzez ich jedyne egzemplarze), jednak śledzą swój jedyny egzemplarz i w standardowych warunkach nie tworzą innych.

W pakiecie narzędziowym InterViews [LCI+92] (służy on do tworzenia interfejsów użytkownika) wzorzec Singleton wykorzystano do zapewnienia dostępu do niepowtarzalnych egzemplarzy klas `Session` i `WidgetKit` (a także innych). W klasie `Session` zdefiniowano główną pętlę obsługi zdarzeń aplikacji. Klasa ta służy też do przechowywania bazy danych z preferencjami stylistycznymi użytkownika i zarządzania połączeniami z wyświetlaczami. `WidgetKit` to Fabryka abstrakcyjna (s. 101) do definiowania wyglądu i działania widgetów interfejsu użytkownika. Operacja `WidgetKit::instance()` określa specyficzną podklasę klasy `WidgetKit`, której egzemplarz jest tworzony na podstawie zmiennej środowiskowej zdefiniowanej w klasie `Session`. Podobna operacja w klasie `Session` określa, czy obsługiwane są wyświetlacze czarno-białe czy kolorowe, i zgodnie z tym konfiguruje jedyny egzemplarz klasy `Session`.

POWIĄZANE WZORCE

Za pomocą wzorca Singleton można zaimplementować wiele innych wzorców. Zobacz wzorce Fabryka abstrakcyjna (s. 101), Budowniczy (s. 92) i Prototyp (s. 120).

OMÓWIENIE WZORCÓW KONSTRUKCYJNYCH

Istnieją dwa standardowe sposoby na sparametryzowanie systemu za pomocą klas tworzonych w nim obiektów. Jedna możliwość to utworzenie podklasy klasy tworzącej obiekty. W ten sposób działa wzorzec Metoda wytwórcza (s. 110). Podstawową wadą tego podejścia jest konieczność tworzenia nowej podklasy tylko po to, aby zmienić klasę produktu. Czasem z jednej modyfikacji tego rodzaju wynika inna. Na przykład jeśli obiekt tworzący produkt sam jest generowany przez metodę fabryczną, trzeba przesłonić także tę metodę.

Inny sposób parametryzowania systemów jest w większym stopniu oparty na składaniu obiektów. Należy zdefiniować obiekt odpowiedzialny za określanie klasy obiektów-produktów i użyć go jako parametru systemu. Jest to kluczowy aspekt działania wzorców Fabryka abstrakcyjna (s. 101), Budowniczy (s. 92) i Prototyp (s. 120). Wszystkie trzy dotyczą tworzenia nowego „obiektu wytwórczego", którego zadaniem jest tworzenie obiektów-produktów. We wzorcu Fabryka abstrakcyjna obiekt wytwórczy generuje obiekty kilku klas. We wzorcu Budowniczy obiekt wytwórczy stopniowo buduje złożony produkt za pomocą odpowiednio złożonego protokołu. We wzorcu Prototyp obiekt wytwórczy generuje produkt przez kopiowanie obiektu prototypowego. W tym przypadku obiekt wytwórczy i prototyp to ten sam obiekt, ponieważ prototyp odpowiada za zwrócenie produktu.

Rozważmy platformę do tworzenia edytorów graficznych omówioną w opisie wzorca Prototyp. Jest kilka sposobów na sparametryzowanie klasy `GraphicTool` za pomocą klasy produktu:

- Przy stosowaniu wzorca Metoda wytwórcza dla każdej podklasy klasy `Graphic` zostanie utworzona podklasa klasy `GraphicTool`. W klasie `GraphicTool` należy umieścić operację `NewGraphic` i przedefiniować ją w każdej podklasie klasy `GraphicTool`.
- Przy stosowaniu wzorca Fabryka abstrakcyjna powstanie hierarchia klas z rodziny `Graphic ↪Factory` (po jednej dla każdej podklasy klasy `Graphic`). Każda fabryka będzie tworzyć tylko jeden produkt. Klasa `CircleFactory` będzie generować obiekty `Circle`, klasa `LineFactory` będzie tworzyć obiekty `Line` itd. Klasę `GraphicTool` należy sparametryzować za pomocą fabryki do tworzenia obiektów `Graphic` odpowiedniego rodzaju.
- Przy stosowaniu wzorca Prototyp każda podklasa klasy `Graphic` będzie obejmować implementację operacji `Clone`, a klasa `GraphicTool` zostanie sparametryzowana prototypem klasy `Graphic`, której egzemplarz ma tworzyć.

To, który wzorzec jest najbardziej przydatny, zależy od wielu czynników. W platformie do tworzenia edytorów graficznych początkowo najprościej jest użyć wzorca Metoda wytwórcza. Łatwo jest zdefiniować nową podklasę klasy `GraphicTool`, a egzemplarze tej klasy są tworzone tylko po zdefiniowaniu palety narzędzi. Główną wadą tego rozwiązania jest duża liczba podklas klasy `GraphicTool` i to, że ich zadania są bardzo ograniczone.

Zastosowanie wzorca Fabryka abstrakcyjna nie jest istotnym usprawnieniem, ponieważ wymaga utworzenia równie dużej hierarchii klasy `GraphicsFactory`. Wzorzec ten warto zastosować zamiast Metody wytwórczej tylko wtedy, jeśli hierarchia klasy `GraphicsFactory` już istnieje, ponieważ albo została automatycznie udostępniona przez kompilator (jak ma to miejsce w językach Smalltalk i Objective C), albo jest potrzebna w innej części systemu.

Ogólnie prawdopodobnie najlepszym wzorcem na potrzeby platformy do tworzenia edytorów graficznych jest Prototyp, ponieważ wymaga jedynie zaimplementowania operacji `Clone` w każdej klasie `Graphics`. Pozwala to zmniejszyć liczbę klas, a operację `Clone` można wykorzystać także do innych celów oprócz tworzenia egzemplarzy (na przykład do obsługi operacji *Powiel* w menu).

Metoda wytwórcza zwiększa możliwość dostosowania projektu do własnych potrzeb, a przy tym sprawia, że jest on w niewielkim tylko stopniu bardziej skomplikowany. Inne wzorce projektowe wymagają tworzenia nowych klas, natomiast przy stosowaniu Metody wytwórczej wystarczy dodać nową operację. Programiści często korzystają z tego wzorca jako standardowego sposobu tworzenia obiektów, jednak nie jest to konieczne, jeśli klasa, której egzemplarze powstają, nigdy się nie zmienia, lub jeżeli generowanie obiektów odbywa się w operacji łatwej do przesłonięcia w podklasach (na przykład w operacji odpowiedzialnej za inicjowanie).

Projekty oparte na wzorcach Fabryka abstrakcyjna, Prototyp lub Budowniczy są jeszcze elastyczniejsze od tych, w których zastosowano wzorzec Metoda wytwórcza, jednak dzieje się to kosztem wyższej złożoności. Często projektant początkowo korzysta ze wzorca Metoda wytwórcza, a następnie — kiedy odkryje, że potrzebna jest większa elastyczność — zmienia projekt przez zastosowanie innych wzorców konstrukcyjnych. Znajomość wielu wzorców projektowych zapewnia większy wybór w czasie analizowania różnych kryteriów projektowych.

Skorowidz

_instance, 132, 135

A

Abstract Factory, 101
AbstractClass, 266
AbstractExpression, 219
AbstractFactory, 103, 109
Abstraction, 183
AbstractList, 231
AbstractProduct, 103
abstrakcje, 18, 26
abstrakcyjne ujęcie procesu tworzenia obiektów, 59
action, 302
ActionCallback, 310
Ada, 35
Adaptee, 143
Adapter, 22, 141, 143, 213
  Adaptee, 143
  Adapter, 143
  Client, 143
  elementy, 143
  implementacja, 145, 147
  konsekwencje stosowania, 144
  powiązane wzorce, 151
  przykładowy kod, 147
  struktura, 143
  Target, 143
  uzasadnienie, 141
  warunki stosowania, 142
  współdziałanie, 143
  zastosowanie, 150
adapter dołączalny, 144, 146
adapter dwukierunkowy, 145
adapter klasowy, 139, 142, 144
adapter obiektowy, 143, 144, 149
adapter parametryzowany, 147
Adaptive Communications Environment, 100
AddressTranslation, 168
Aggregate, 232, 233
agregacja, 36
akcja, 302
Alexander, 340
Alexander Christopher, 16
algorytmy, 38
  algorytm formatowania, 53
  algorytm przechodzenia, 233
AlternationExpression, 219, 223
Anderson Bruce, 341
API, 37
aplikacje, 39
aplikacje graficzne, 170
AppKit, 190
Application, 110, 264
ApplicationWindow, 65
architektura MVC, 18, 118, 179
  kontroler, 20
  model, 19
  reagowanie widoku na działania użytkownika, 20
  widok, 19
  zagnieżdżanie widoków, 19
ArrayCompositor, 322
ArrayIterator, 77
ASCII, 92

ASCII7Stream, 160
ASCIIConverter, 92
automatyczne przekazywanie, 249

B

Beck Kent, 342
Bedrock, 156
BinaryExpression, 180
black-box reuse, 32
BNF, 221
BooleanExp, 225
Border, 56
BorderDecorator, 153
BorderedComposition, 56
brak elastyczności, 90
Bridge, 181
BTree, 208
Budowniczy, 22, 92, 138
  Builder, 93
  ConcreteBuilder, 93
  Director, 93, 94
  elementy, 93
  implementacja, 95
  konsekwencje stosowania, 94
  powiązane wzorce, 100
  Product, 94
  przykładowy kod, 96
  relacje, 93
  Smalltalk-80, 100
  uzasadnienie, 92
  warunki stosowania, 93
  współdziałanie, 94
  zastosowanie, 100
Builder, 92, 93
Bytecode, 164
BytecodeStream, 100, 161, 164

C

C++, 18
  dziedziczenie, 30
  kontrola dostępu, 267
  przeciążanie operatora dostępu do składowych, 195
  szablony, 35
  this, 33
Caretaker, 296
CASE, 337
centralizacja sterowania, 258
Chain of Responsibility, 244
ChangeManager, 262, 274, 275
Choices, 168
CISCscheduler, 328
ClassBuilder, 100
Client, 103, 122, 143, 172, 205, 220, 246, 305
Clone, 124
CLOS, 18
cofanie działań, 294, 304, 306
Colleague, 257
Command, 71, 72, 253, 302, 305
Component, 154, 156, 172, 174, 179, 253
ComponentView, 253
Composite, 170, 172, 179
CompositeView, 19
Composition, 53, 326, 327
Compositor, 53, 326
CompressingStream, 160
ConcreteAggregate, 232
ConcreteBuilder, 93
ConcreteClass, 266
ConcreteCommand, 305
ConcreteComponent, 154
ConcreteCreator, 111, 115
ConcreteDecorator, 154, 155
ConcreteElement, 283, 284, 288
ConcreteFactory, 103
ConcreteFlyweight, 204
ConcreteHandler, 246
ConcreteImplementor, 183
ConcreteIterator, 232
ConcreteMediator, 257
ConcreteObserver, 271
ConcreteProduct, 103, 111
ConcretePrototype, 122
ConcreteState, 313
ConcreteStrategy, 323
ConcreteSubject, 271
ConcreteVisitor, 283, 288
ConstraintSolver, 294, 298
ConstraintSolverMemento, 298
Context, 220, 313, 323
Controller, 18, 20
CreateIterator, 231
CreateMaze(), 117
Creator, 111
CSolver, 300
cursor, 230
cykl życia oprogramowania obiektowego, 337
czytnik dokumentów RTF, 92

D

DebuggerAdaptor, 128
DebuggingGlyph, 159
decentralizacja logiki zmian stanów, 315
Decorator, 152, 154
defaultController(), 118
defaultControllerClass(), 118
definiowanie rozszerzalnych fabryk, 105
dekorator, 152
Dekorator, 22, 58, 152, 213
  Component, 154
  ConcreteComponent, 154
  ConcreteDecorator, 154
  Decorator, 154
  elementy, 154
  implementacja, 155
  konsekwencje stosowania, 155
  powiązane wzorce, 160
  przykładowy kod, 157
  uzasadnienie, 152
  warunki stosowania, 154
  współdziałanie, 154
  zastosowanie, 159
delegat, 33, 146
delegowanie, 33, 34
dependents, 269
DialogDirector, 259
DialogKit, 109
DialogWindow, 65
Director, 93, 94
Display, 268
dobre projekty, 15
Document, 110, 264
dodawanie operacji, 284
dodawanie produktów w czasie wykonywania programu, 122
doesNotUnderstand, 196, 199, 249
dokumentowanie, 336
dokumentowanie platformy, 41
dokumenty RTF, 92
Domain, 168
dostęp do rozproszonych informacji, 75
double dispatch, 287
DrawingApplication, 110
DrawingController, 320
DrawingDocument, 110
drzewo składni abstrakcyjnej, 221, 280
dwukrotna dyspozycja, 287
Dylan, 300
dynamiczna konfiguracja aplikacji, 123
dyspozycja
  dwukrotna, 287
  jednokrotna, 287
  wielokrotna, 287
dziedziczenie, 27, 29, 215
  C++, 30
  Eiffel, 30
  interfejsy, 30
  klasy, 30, 31
  klasy abstrakcyjne, 29
  klasy nadrzędne, 29
  podklasy, 29
  przesłanianie metod, 29
  Smalltalk, 30
  stosowanie, 31
  typy sparametryzowane, 35
dziedziczenie dynamiczne, 316

E

edytor dokumentów, 45
  ApplicationWindow, 65
  Border, 56
  BorderedComposition, 56
  cofanie operacji, 72
  Command, 71, 72
  Composition, 53
  Compositor, 53
  Dekorator, 58
  DialogWindow, 65
  dostęp do rozproszonych informacji, 75
  działania użytkowników, 69
  Fabryka abstrakcyjna, 62
  fabryki, 60
  formatowanie, 46, 52
  glify-widgety, 60
  Glyph, 50, 59, 76
  GUIFactory, 60
  historia poleceń, 73
  IconWindow, 65
  interfejs użytkownika, 46
  Iterator, 77
  kapsułkowanie algorytmu formatowania, 52
  kapsułkowanie analiz, 81
  kapsułkowanie dostępu do danych, 75
  kapsułkowanie zależności implementacyjnych, 64
  kapsułkowanie żądania, 70

edytor dokumentów
klasy produktów, 60
Kompozyt, 52
konfiguracja obiektów WINDOW, 68
MenuItem, 70
MonoGlyph, 56
MotifFactory, 60
MotifScrollBar, 59, 60
obsługa wielu standardów wyglądu i działania, 59
obsługa wielu systemów okienkowych, 63
operacje, 69
ozdabianie interfejsu użytkownika, 46, 55
PMScrollBar, 59
podział słów, 74
Polecenie, 74
powtarzanie operacji, 72
problemy projektowe, 45
produkty, 61
przechodzenie po elementach, 80
Rectangle, 51
ScrollBar, 59
Scroller, 57
SpellingChecker, 81, 82, 83
sprawdzanie pisowni, 74
Strategia, 55
struktura dokumentu, 45, 47
Visitor, 84
Window, 51, 64, 66
WindowImp, 66
edytor graficzny, 141
egzemplarz klasy, 28
Eiffel, 35
elastyczność projektu, 90
Element, 283
Encapsulator, 200
Equipment, 289
ET++, 118, 253
ET++SwapsManager, 328
Etgdb, 128
Execute, 302, 303

F

Fabryka abstrakcyjna, 22, 62, 63, 101, 137
AbstractFactory, 103
AbstractProduct, 103
Client, 103
ConcreteFactory, 103
ConcreteProduct, 103
elementy, 103
implementacja, 104
konsekwencje stosowania, 103
powiązane wzorce, 109
przykładowy kod, 106
uzasadnienie, 101
warunki stosowania, 102
współdziałanie, 103
zastosowanie, 109
Facade, 161, 163, 167
Factory Method, 110
Fasada, 22, 161
elementy, 163
Facade, 163
implementacja, 164
klasy podsystemu, 163
konsekwencje stosowania, 163
powiązane wzorce, 169
przykładowy kod, 164
uzasadnienie, 161
warunki stosowania, 162
współdziałanie, 163
zastosowania, 167
FileStream, 159
FileSystemInterface, 168
Flyweight, 201, 204
FlyweightFactory, 205, 206, 210
FontDialogDirector, 255
Foote Brian, 337
format danych
ASCII, 92
RTF, 92
framework, 40
funkcje zwrotne, 304
funktory, 311

G

GenerateCode, 287
Glyph, 50, 76, 206
GlyphContext, 208
głębokie kopiowanie, 124
gra, 88
gramatyka, 217, 220
Graphic, 120, 197
GraphicTool, 120, 137
GUIFactory, 60

H

Handle, 189
handle/body, 181
HandleHelp, 245
Handler, 246, 250, 252
HelpHandler, 245, 250
historia poleceń, 73, 307
HotDraw, 320

I

IconWindow, 65, 182
ImageProxy, 192, 198
implementacja gramatyki, 220
implementacja obiektu, 28
Implementor, 183, 184
inicjowanie klonów, 125
Initialize, 125
InspectClass, 167
InspectObject, 167
instrukcje warunkowe, 323
Instrument, 328
inteligentne referencje, 193, 194
inteligentne wskaźniki, 193
interfejs użytkownika, 101
interfejsy, 27
  dziedziczenie, 30
Interpret, 221
Interpreter, 22, 217
  AbstractExpression, 219
  Client, 220
  Context, 220
  elementy, 219
  implementacja, 221
  konsekwencje stosowania, 220
  NonterminalExpression, 220
  powiązane wzorce, 229
  przykładowy kod, 222
  TerminalExpression, 220
  uzasadnienie, 217
  warunki stosowania, 219
  współdziałanie, 220
  zastosowanie, 228
InterViews, 109, 136, 150, 159, 278
InvalidateRect, 253
InventoryVisitor, 291
Invoker, 305
IRIS Inventor, 292
iteracja polimorficzna, 231
IterationState, 300
Iterator, 22, 77, 80, 230, 234, 330
  Aggregate, 232, 233
  ConcreteAggregate, 232
  ConcreteIterator, 232
  elementy, 232
  implementacja, 233
  Iterator, 232
  konsekwencje stosowania, 233
  powiązane wzorce, 243
  przykładowy kod, 236
  uzasadnienie, 230
  warunki stosowania, 232
  współdziałanie, 232
  zastosowanie, 243
iteratory, 77, 230
  aktywny, 233
  pasywny, 233
  polimorficzny, 234
  pusty, 235
  wewnętrzny, 233, 240
  zewnętrzny, 233
izolowanie klas konkretnych, 103

J

jawne referencje do elementu nadrzędnego, 173
język programowania, 18
  C++, 18
  Dylan, 300
  Smalltalk-80, 18

K

kapsułkowanie, 26
  algorytm formatowania, 52
  dostęp do danych, 75
  komunikacja między obiektami, 331
  zależności implementacyjne, 64
  złożona semantyka aktualizacji, 274
  zmiany, 330
  żądania, 70
katalog wzorców projektowych, 22
kategoria wzorca, 20
KernelProxy, 192
kit, 101
klasowe wzorce operacyjne, 215
klasowe wzorce strukturalne, 139

klasy, 28
  dziedziczenie, 29, 30
  klasy abstrakcyjne, 29, 50
  klasy konkretne, 29
  klasy mieszane, 29
  klasy nadrzędne, 29
klient, 26
klonowanie obiektów, 120, 124
Knuth Donald, 341
kod wielokrotnego użytku, 15
kolejkowanie żądania, 304
kompilator, 228, 280
komponenty, 56
Kompozyt, 22, 52, 170, 213
  Client, 172
  Component, 172
  Composite, 172
  elementy, 172
  implementacja, 173
  konsekwencje stosowania, 173
  Leaf, 172
  powiązane wzorce, 180
  przykładowy kod, 177
  uzasadnienie, 170
  warunki stosowania, 171
  współdziałanie, 172
  zastosowanie, 179
kompresja danych ze strumienia, 159
komunikacja, 331
komunikaty, 26
konsekwencje stosowania wzorca projektowego, 17
konserwowanie złożonych gramatyk, 221
konsolidowanie, 337
Konstruktor wirtualny, 110
kontrola dostępu do jedynego egzemplarza, 131
kontrola dostępu do obiektu, 191
kontroler, 18, 20
konwencje nazewnicze, 116, 267
konwersja dokumentów RTF, 100
kopiowanie przy zapisie, 194
kursor, 230, 233

L

labirynt, 88, 106
LALR, 100
Layout, 211, 212
Leaf, 172
leniwe inicjowanie, 115
libg++, 189
limit liczby egzemplarzy, 131
List, 35, 236
lista rozwijana, 254
ListIterator, 77, 230, 240
LiteralExpression, 219
Look, 212
luźne powiązanie, 38

Ł

Łańcuch, 333
łańcuch następników, 247
Łańcuch zobowiązań, 22, 244
  Client, 246
  ConcreteHandler, 246
  elementy, 246
  Handler, 246
  implementacja, 247
  konsekwencje stosowania, 247
  powiązane wzorce, 253
  przykładowy kod, 250
  uzasadnienie, 244
  warunki stosowania, 246
  współdziałanie, 247
  zastosowanie, 252
łączenie następników, 248

M

MacApp, 116, 118, 156
MacroCommand, 306, 307, 309
MapSite, 127
MazeFactory, 106
mechanizm powtórnego wykorzystania
  rozwiązania, 32
Mediator, 23, 254, 330, 333
  ConcreteMediator, 257
  elementy, 257
  implementacja, 258
  konsekwencje stosowania, 258
  Mediator, 257
  powiązane wzorce, 263
  przykładowy kod, 259
  uzasadnienie, 254
  warunki stosowania, 256
  współdziałanie, 257
  zastosowanie, 261

Memento, 294, 296
MemoryObject, 168
MemoryObjectCache, 168
MemoryStream, 159
menedżer prototypów, 124
MenubarLayout, 211
MenuItem, 70
metaklasy, 136
Metoda szablonowa, 23, 264
  AbstractClass, 266
  ConcreteClass, 266
  elementy, 266
  implementacja, 267
  konsekwencje stosowania, 266
  powiązane wzorce, 268
  przykładowy kod, 267
  uzasadnienie, 264
  warunki stosowania, 265
  współdziałanie, 266
  zastosowanie, 268
Metoda wytwórcza, 20, 23, 110, 137
  ConcreteCreator, 111
  ConcreteProduct, 111
  Creator, 111
  elementy, 111
  implementacja, 113
  konsekwencje stosowania, 112
  powiązane wzorce, 119
  Product, 111
  przykładowy kod, 117
  uzasadnienie, 110
  warunki stosowania, 111
  współdziałanie, 112
  zastosowanie, 118
metodologie projektowania obiektowego, 26
metody, 26
metody projektowania obiektowego, 337
mixin class, 29
Mode Composer, 129
model, 18
Model, 18, 278
model publikuj-subskrybuj, 270
model wyciągania, 274
model wypychania, 274
Model/View/Controller, 18
MonoGlyph, 56
Most, 23, 69, 181, 213
  Abstraction, 183
  ConcreteImplementor, 183
  elementy, 183
  implementacja, 184
  Implementor, 183
  konsekwencje stosowania, 184
  powiązane wzorce, 190
  przykładowy kod, 185
  RefinedAbstraction, 183
  uzasadnienie, 181
  warunki stosowania, 182
  współdziałanie, 184
  zastosowanie, 189
Motif, 60, 101
MotifFactory, 60
MotifScrollBar, 59, 60
MotifWidgetFactory, 102
MVC, 18, 278

N

nadtypy, 27
Nakładka, 141, 152
nazwa wzorca, 17, 20
nazwy klas abstrakcyjnych, 29
niejawny odbiorca żądań, 245
nieoczekiwane aktualizacje, 272
niepowtarzalność egzemplarza, 131
niewidoczna otoczka, 56
niezawodny iterator, 234
NodeVisitor, 281
NonterminalExpression, 220
notacja BNF, 221
notacja OMT, 21, 28
NullIterator, 78, 235
NXImage, 190
NXImagRep, 190
NXProxy, 192, 200

O

obiekt fasadowy, 161
obiektowe wzorce operacyjne, 215
obiektowy język programowania, 18
obiekty, 26
  interfejsy, 27
  poziom szczegółowości, 27
  składanie, 33
obiekty jako argumenty, 330
obiekty stanów, 312
obiekty zależne, 269

obiekty złożone, 19
objects for states, 312
ObjectStructure, 283
ObjectWindows, 243, 329
Observer, 269, 271, 272
Obserwator, 23, 269, 331, 332
  aspekty obiektów Subject, 274
  ConcreteObserver, 271
  ConcreteSubject, 271
  elementy, 270
  implementacja, 272
  konsekwencje stosowania, 272
  Observer, 271
  obserwator, 270
  podmiot, 270
  powiązane wzorce, 279
  przykładowy kod, 276
  publikuj-subskrybuj, 270
  Subject, 270
  uzasadnienie, 269
  warunki stosowania, 270
  współdziałanie, 271
  zastosowanie, 278
obsługa cofania działań, 294
obsługa rozsyłania grupowego komunikatów, 272
obsługa wielu standardów wyglądu i działania, 59
obsługa wielu systemów okienkowych, 63
oddzielanie interfejsu od implementacji, 184
oddzielanie nadawców od odbiorców, 332
odporność na zmiany, 37
Odwiedzający, 23, 85, 280
  ConcreteElement, 283, 284
  ConcreteVisitor, 283
  Element, 283
  elementy, 283
  implementacja, 285
  konsekwencje stosowania, 284
  ObjectStructure, 283
  powiązane wzorce, 293
  przykładowy kod, 288
  uzasadnienie, 280
  Visitor, 283
  warunki stosowania, 282
  współdziałanie, 283
  zastosowanie, 292
ograniczanie tworzenia podklas, 258
określanie implementacji obiektów, 28
określanie nowych obiektów
  przez modyfikowanie struktury, 123
  przez zmianę wartości, 122
określanie poziomu szczegółowości obiektu, 27
OMT, 21, 28
OpenCommand, 307
OpenDocument, 265
operacje, 26
  abstrakcyjne operacje, 29
operator dostępu do składowych, 195
opis problemu, 17
opis wzorców projektowych, 20
oprogramowanie obiektowe, 335
Orbix ORB, 119
Originator, 296
otoczka, 56
otwarte powtórne wykorzystanie, 32
ozdabianie interfejsu użytkownika, 55

P

pakiety narzędziowe, 39
Pamiątka, 23, 294
  Caretaker, 296
  elementy, 296
  implementacja, 297
  konsekwencje stosowania, 297
  Memento, 296
  Originator, 296
  powiązane wzorce, 301
  przykładowy kod, 298
  uzasadnienie, 294
  warunki stosowania, 295
  współdziałanie, 296
  zastosowanie, 300
  źródło, 295
pamięć podręczna, 176
Pane, 262
parametry, 35
parametryzacja obiektów, 304
parametryzacja systemu, 137
Parser, 100, 161, 164
parserClass(), 118
PassivityWrapper, 159
PasteCommand, 307
Pełnomocnik, 23, 191, 214
  elementy, 193
  implementacja, 195
  konsekwencje stosowania, 194
  powiązane wzorce, 200
  Proxy, 193
  przykładowy kod, 197

RealSubject, 194
Subject, 194
uzasadnienie, 191
warunki stosowania, 192
współdziałanie, 194
zastosowanie, 200
pełnomocnik wirtualny, 192, 194, 200
pełnomocnik zabezpieczający, 194
platforma, 40
pluggable adapter, 144
PluggableAdapter, 150
płytka kopia, 124
PMIconWindow, 181
PMScrollBar, 59
PMWindow, 181
podklasy, 29, 38, 123
podklasy klasy Singleton, 133
podsystem kompilujący, 161
podtypy, 27
podział słów, 74
podział strumienia tekstu na wiersze, 321
Polecenie, 23, 74, 302, 332
Client, 305
Command, 305
ConcreteCommand, 305
elementy, 305
implementacja, 306
Invoker, 305
konsekwencje stosowania, 306
powiązane wzorce, 311
przykładowy kod, 307
Receiver, 305
uzasadnienie, 302
warunki stosowania, 304
współdziałanie, 305
zastosowania, 310
policy, 321
polimorfizm, 27
polityka, 321
połączenie równoległych hierarchii klas, 112
połączenie sieciowe, 312
TCP, 316
pomijanie klasy abstrakcyjnej, 155
porządkowanie wzorców, 25
PostorderIterator, 77
pośrednik
wirtualny, 194, 197
zabezpieczający, 192
zdalny, 194
powtarzanie operacji, 306
powtórne wykorzystanie, 32, 37
kod, 39, 266
projekt, 40
poziom szczegółowości obiektu, 27
PreorderIterator, 77
Presentation Manager, 101, 181
PricingVisitor, 290
problem, 17
problemy specyficzne dla języka, 115
Product, 94, 111
produkty, 61
ProgrammingEnvironment, 167
ProgramNode, 161
ProgramNodeBuilder, 161
ProgramNodeEnumerator, 292
programowanie pod kątem interfejsu, 31
programy obiektowe, 26
projekt obiektowy, 15
projekt platformy, 41
projektowanie
aplikacje, 40
edytor dokumentów, 45
obiektowe, 26, 31, 33
oprogramowanie obiektowe, 15
pod kątem zmian, 37
prototyp, 120
Prototyp, 23, 104, 105, 120, 138
Client, 122
ConcretePrototype, 122
elementy, 122
implementacja, 124
konsekwencje zastosowania, 122
powiązane wzorce, 129
Prototype, 122
przykładowy kod, 125
uzasadnienie, 120
warunki stosowania, 121
zastosowanie, 128
Prototype, 120, 122
Proxy, 191, 193
przeciążanie operatora dostępu do składowych, 195
przesłanianie metod, 29
przestrzeń nazw, 131
przestrzeń wzorców projektowych, 24
przyrostowe zmiany, 298
publikuj-subskrybuj, 269, 270
publish-subscribe, 269
punkty zaczepienia, 266
podklasy, 112

Pyłek, 23, 201
  Client, 205
  ConcreteFlyweight, 204
  elementy, 204
  Flyweight, 204
  FlyweightFactory, 205
  implementacja, 206
  konsekwencje stosowania, 205
  powiązane wzorce, 212
  przykładowy kod, 206
  UnsharedConcreteFlyweight, 205
  uzasadnienie, 201
  warunki stosowania, 203
  współdziałanie, 205
  zastosowanie, 211

Q

QOCA, 145, 228
Queue, 243

R

RApp, 329
ReadStream, 243
RealSubject, 194
Receiver, 305
Rectangle, 51
refaktoryzacja, 337
RefinedAbstraction, 183
RegisterAllocator, 328
RegisterTransfer, 180
RegularExpression, 217, 224
rejestr singletonów, 133
rejestrowanie zmian, 304
rekurencyjne składanie, 48
relacje między wzorcami projektowymi, 25
RepetitionExpression, 219, 223
reprezentowanie żądań, 248
Request, 248
Rich Text Format, 92
RISCscheduler, 328
rozmiar obiektu, 191
rozsyłanie grupowe komunikatów, 272
rozszerzanie gramatyki, 220
rozwiązanie, 17
rozwijanie, 337
RTF, 92
RTFReader, 92
RTL, 180
RTL System, 328
RTLExpression, 180

S

Scanner, 161, 164
ScrollBar, 59
ScrollbarLayout, 211
ScrollDecorator, 153
Scroller, 57
self, 33
Self, 124, 316
SequenceExpression, 223
Service Configurator, 100
Session, 136
SimpleCompositor, 321
singleton, 130
Singleton, 23, 91, 104, 131
  elementy, 131
  implementacja, 131
  konsekwencje stosowania, 131
  powiązane wzorce, 136
  przykładowy kod, 135
  uzasadnienie, 130
  warunki stosowania, 130
  zastosowanie, 136
Sketchpad, 128
składanie obiektów, 32, 33, 56, 137, 149, 215
składanie rekurencyjne, 48
słownictwo projektowe, 336
Smalltalk-80, 18
  automatyczne przekazywanie, 249
  Budowniczy, 100
  dziedziczenie, 30
  self, 33
sparametryzowane metody wytwórcze, 113
SPECTalk, 228
SpellingChecker, 81, 82, 83
społeczność związana ze wzorcami, 340
spójność między produktami, 104
SSA, 180
Stan, 23, 312
  ConcreteState, 313
  Context, 313
  elementy, 313
  implementacja, 315
  konsekwencje stosowania, 314
  powiązane wzorce, 320

przykładowy kod, 316
State, 313
uzasadnienie, 312
warunki stosowania, 313
współdziałanie, 313
zastosowania, 319
stan wewnętrzny, 201
stan zewnętrzny, 201
State, 312, 313, 330
statyczne dziedziczenie, 155
stosowanie wzorca projektowego, 43
Strategia, 20, 23, 55, 321
ConcreteStrategy, 323
Context, 323
elementy, 322
implementacja, 324
konsekwencje stosowania, 323
powiązane wzorce, 329
przykładowy kod, 326
Strategy, 322
uzasadnienie, 321
warunki stosowania, 322
współdziałanie, 323
zastosowanie, 328
Strategy, 321, 322, 330
StreamDecorator, 160
strukturalne wzorce obiektowe, 139
struktury projektowe, 18
strumienie, 159
Subject, 194, 270, 272
Substytut, 191
SunDbxAdaptor, 128
SunWindowPort, 189
surrogate, 191
sygnatura, 27
system odporny na zmiany, 37
system pomocy w graficznym interfejsie użytkownika, 244
szablony, 35, 116, 307

Ś

ścisłe powiązanie, 38
środowiska okienkowe, 63

T

TableAdaptor, 151
tablice, 315
Target, 143
TCP, 316
TCPConnection, 312, 316
Template method, 264
TerminalExpression, 220
TeXCompositor, 322
TextConverter, 92
ThingLab, 128
THINK, 310
this, 33
token, 294
Tool, 320
transaction, 302
transakcja, 302
TransientWindow, 182
transparent enclosure, 56
TreeDisplay, 146
tworzenie
egzemplarze klasy, 28
podtypy, 30
produkty, 104
prototypy, 337
TypeCheck, 281
typy danych, 27
typy generyczne, 35
typy sparametryzowane, 35

U

Uchwyt/ciało, 181
ukrywanie szczegółów implementacji przed klientami, 184
Unidraw, 114, 145, 300, 310, 320
UnsharedConcreteFlyweight, 205
uporządkowanie elementów podrzędnych, 176
usuwanie produktów w czasie wykonywania programu, 122
usuwanie stanu zewnętrznego, 206

V

Validator, 329
ValueModel, 150
View, 18, 179
ViewManager, 262
virtual constructor, 110
Visitor, 84, 280, 283, 330
VisualComponent, 153
VObject, 179

W

wewnętrzna reprezentacja produktu, 94
white-box reuse, 32
wiązanie dynamiczne, 27
WidgetFactory, 101, 102
WidgetKit, 109, 136
widok, 18
wielodziedziczenie, 143, 148, 185
wielokrotny użytek, 15
Window, 51, 64, 66, 181, 182, 185
WindowImp, 66, 182, 185
WindowPort, 189
wiszące referencje do usuniętych podmiotów, 273
wrapper, 141, 152
współużytkowanie komponentów, 173
współużytkowanie symboli końcowych, 221
wyrażenia regularne, 217, 222
WYSIWYG, 45
wyszukiwanie liniowe, 90
wyszukiwanie wzorców, 42
wywoływanie żądania, 304
wzorce konstrukcyjne, 24, 87
  Abstract Factory, 101
  Budowniczy, 92
  Builder, 92
  Fabryka abstrakcyjna, 101
  Metoda wytwórcza, 110
  Prototyp, 120
  Prototype, 120
  Singleton, 91, 130
wzorce operacyjne, 24, 215
  Chain of Responsibility, 244
  Command, 302
  Interpreter, 217
  Iterator, 230
  kapsułkowanie zmian, 330
  Łańcuch zobowiązań, 244
  Mediator, 254
  Memento, 294
  Metoda szablonowa, 264
  obiekty jako argumenty, 330
  Observer, 269
  Obserwator, 269
  oddzielanie nadawców od odbiorców, 332
  Odwiedzający, 280
  Pamiątka, 294
  Polecenie, 302
  Stan, 312
  State, 312
  Strategia, 321
  Strategy, 321
  Template method, 264
  Visitor, 280
  wzorce klasowe, 215
  wzorce obiektowe, 215
wzorce projektowe, 16, 17
  abstrakcje, 26
  Adapter, 22, 141
  architektura MVC, 18
  Budowniczy, 22, 92
  Dekorator, 22, 58, 152
  elementy, 17
  Fabryka abstrakcyjna, 22, 62, 63, 101
  Fasada, 22, 161
  graficzna reprezentacja klas, 21
  implementacja, 21
  Interpreter, 22, 217
  Iterator, 22, 80, 230
  katalog, 22
  kategoria, 20
  Kompozyt, 22, 52, 170
  konsekwencje, 17, 21
  Łańcuch zobowiązań, 22, 244
  Mediator, 23, 254
  Metoda szablonowa, 23, 264
  Metoda wytwórcza, 23, 110
  Most, 23, 69, 181
  nazwa, 17, 20
  Obserwator, 23, 269
  Odwiedzający, 23, 85, 280
  opis, 17, 20
  Pamiątka, 23, 294
  Pełnomocnik, 23, 191
  Polecenie, 23, 74, 302
  Prototyp, 23, 104, 120
  przeznaczenie, 20
  Pyłek, 23, 201
  relacje między wzorcami, 25
  rozwiązanie, 17
  Singleton, 23, 130
  Stan, 23, 312
  stosowanie, 43
  Strategia, 23, 55, 321
  struktura, 21
  warunki stosowania, 21
  współdziałanie, 21
  wybór, 42
  zasięg, 24

wzorce strukturalne, 24, 139
Adapter, 141
Bridge, 181
Composite, 170
Decorator, 152
Dekorator, 152
Facade, 161
Fasada, 161
Flyweight, 201
klasowe wzorce, 139
Kompozyt, 170
Most, 181
obiektowe wzorce, 139
Pełnomocnik, 191
Proxy, 191
Pyłek, 201

X

X Window, 181
XIconWindow, 181
XWindow, 181
XWindowPort, 189

Y

YieldCurve, 328

Z, Ź, Ż

zagnieżdżanie widoków, 19
zależność od algorytmów, 38
zależność od platformy sprzętowej lub programowej, 37
zależność od reprezentacji lub implementacji obiektu, 38
zależność od specyficznych operacji, 37
zamknięte powtórne wykorzystanie, 32
zapisywanie przyrostowych zmian, 298
zarządzanie pamięcią wirtualną, 168
zarządzanie współużytkowanymi obiektami, 206
zasięg wzorca, 24
zdalny pełnomocnik, 192
Zestaw, 101
zliczanie referencji, 193
zmiany, 37, 304, 330
zmiany stanu, 314
zmienne egzemplarza, 28
zmniejszanie złożoności systemu, 161
znacznik, 294
znajomość obiektów, 36
związki między strukturami czasu wykonywania programu i strukturami czasu kompilacji, 36
źródło pamiątki, 295
żądanie, 26, 248


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

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!

dr Erich Gamma jest dyrektorem technicznym w Software Technology Center of Object Technology International w Zurychu (Szwajcaria).

dr Richard Helm jest członkiem zespołu Object Technology Practice Group w IBM Consulting Group w Sydney (Australia).

dr Ralph Johnson jest pracownikiem naukowym na wydziale nauk komputerowych Uniwersytetu Illinois w Urbana-Champaign.

dr John Vlissides prowadzi badania w Thomas J. Watson Research Center firmy IBM w Hawthorne w stanie Nowy Jork.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-8609-9

Cena: 79,00 zł