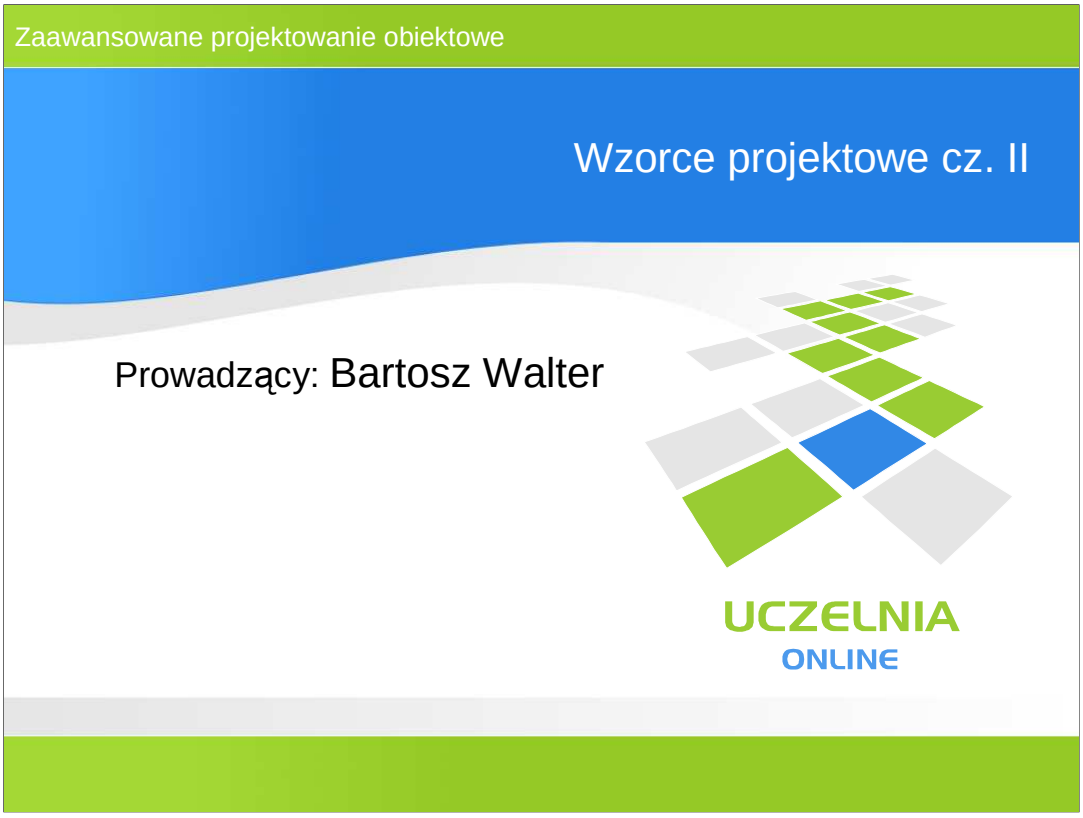
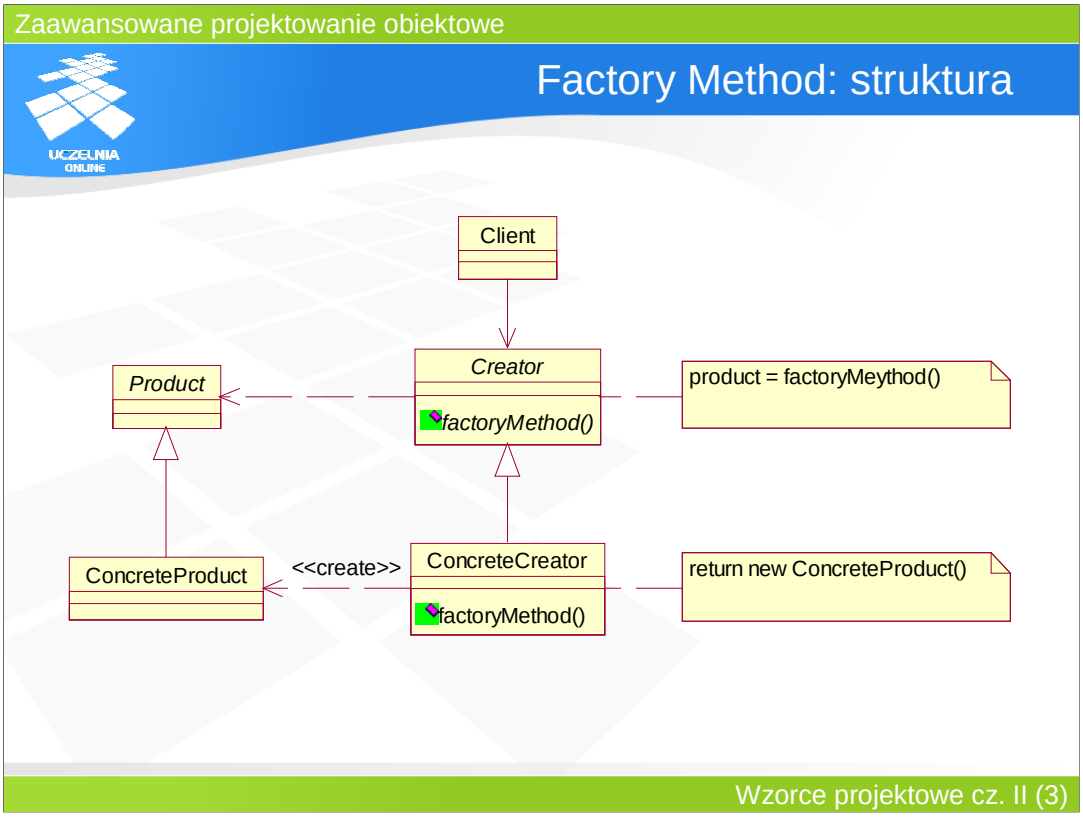

Zaawansowane projektowanie obiektowe
Wzorce projektowe cz. II
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE

Factory Method: cel

- Zdefiniowanie interfejsu do tworzenia obiektów
- Umożliwienie przekazania odpowiedzialności za tworzenie obiektów do podklas
- Umożliwienie wyboru klasy i konstruktora użytego do utworzenia obiektu

E. Gamma et al. (1995)

Wzorzec Factory Method jest podstawowym wzorcem kreacyjnym. Jego celem jest zastąpienie prostych wywołań konstruktora dedykowanym interfejsem (metodą), która przejmie odpowiedzialność za tworzenie i ew. inicjację obiektu danej klasy. Podobnie jak w przypadku wzorca Singleton (który jest specjalizowaną wersją Factory Method, ograniczoną do tworzenia jednego obiektu), istnieje możliwość hermetyzacji wewnątrz tej metody sposobu wyboru klasy obiektu spośród jej podklas oraz użytego konstruktora.



Klient odwołuje się do dwóch interfejsów (lub klas abstrakcyjnych): Creator, zawierającej metodę tworzącą produkty, i Product, reprezentującą obiekty tworzone przez Factory Method. Oba interfejsy posiadają implementacje powiązane parami: obiekt klasy ConcreteCreator o przeciążonej metodzie *factoryMethod()* tworzy instancję obiektu ConcreteProduct. Dzięki temu, podczas zmiany obiektu Creator, jednocześnie zmieniany jest tworzony produkt. Warto zwrócić na dualizm hierarchii klas produktu i kreatora, który pozwala na abstrakcyjne traktowanie całego procesu tworzenia obiektów za pomocą wymiennych producentów, co jest niemożliwe przy bezpośrednim użyciu konstruktora.

Z punktu widzenia klienta metoda *factoryMethod()* jest równoważna pod względem funkcjonalnym z konstruktorem: jej wywołanie powoduje utworzenie obiektu żądanego typu. Warto zwrócić uwagę, że wzorzec ten pozwala także na dodatkowy stopień swobody: bezpośrednie wywołanie konstruktora przez klienta zawsze powoduje utworzenie obiektu konkretnej klasy (konstruktory nie są metodami polimorficznymi), natomiast użycie wzorca Factory Method pozwala metodzie tworzącej obiekty na wybór klasy obiektu i sposobu jego tworzenia.

Factory Method: uczestnicy

- **Product**
  - definiuje interfejs obiektów tworzonych przez *Factory Method*
- **Concrete Product**
  - specyficzny produkt tworzony przez *Factory Method*
- **Creator**
  - definiuje interfejs do tworzenia obiektów typu *Product*
- **Concrete Creator**
  - tworzy obiekt typu *Concrete Product*

Product reprezentuje wszystkie obiekty, jakie są tworzone przez metodę *factoryMethod().* Często jest to grupa klas posiadająca wspólną nadklasę lub zwykły interfejs z implementującymi go klasami. Klient jest powiązany z produktami właśnie poprzez ten interfejs.

Tworzeniem produktów zajmują się obiekty o interfejsie Creator. W podstawowej postaci wzorca Interfejs ten także jest jedyną informacją dotyczącą typu, jaką posiada klient. Użycie odpowiedniej klasy ConcreteCreator determinuje klasę i właściwości produktu, jaki zostanie utworzony.

W innej wersji tego wzorca Creator jest klasą, której statyczna metoda *factoryMethod()* dokonuje selekcji produktów na podstawie przekazanych jej parametrów.

Najważniejszym efektem użycia wzorca jest przeniesienie odpowiedzialności za tworzenie obiektów klasy Product z klienta na obiekt klasy Creator. Dzięki temu klient może założyć, że za każdym razem, gdy wywoła metodę *factoryMethod(),* otrzyma instancję klasy gotową do użycia.

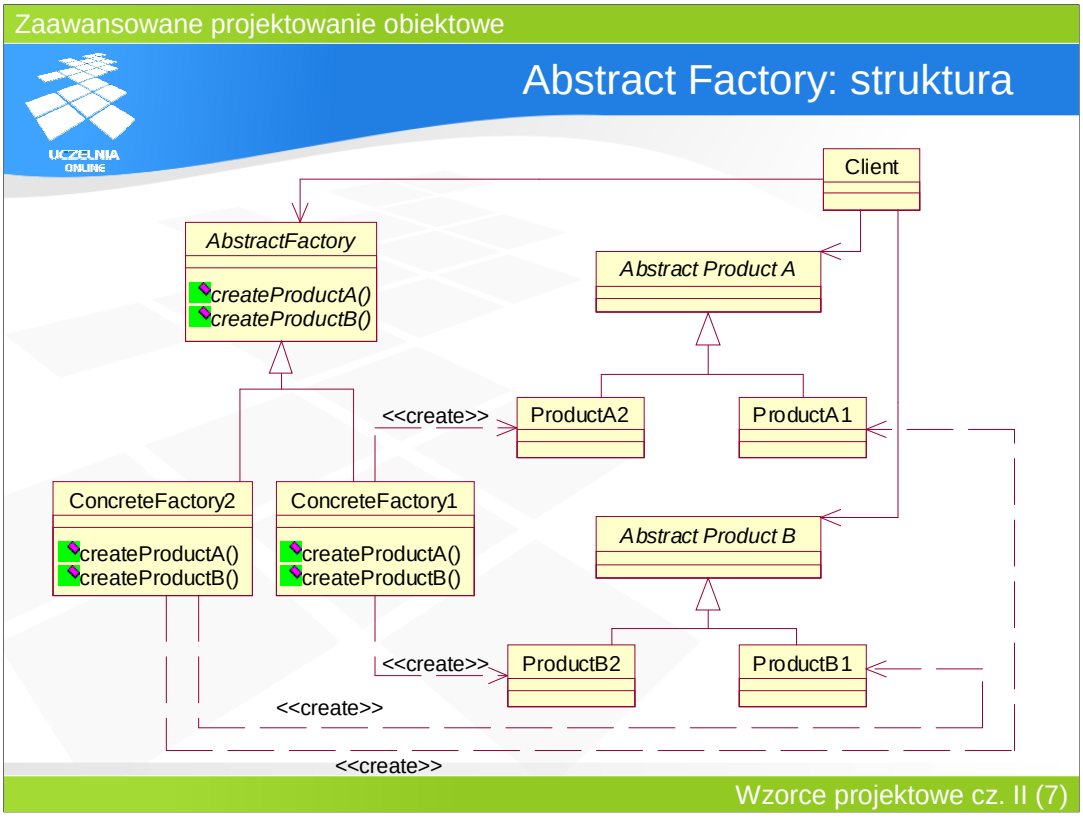
Ponadto wzorzec umożliwia tworzenie nie tylko instancji jednej klasy, ale całych ich hierarchii, z możliwością wyboru klasy i użytego konstruktora. Bezpośrednie wywołanie konstruktora nie daje takiej możliwości.

Abstract Factory: cel

- Stworzenie interfejsu do tworzenia grup powiązanych ze sobą produktów
- Rozszerzenie *Factory Method* na grupy produktów

E. Gamma et al. (1995)

Wzorzec Abstract Factory jest rozszerzeniem koncepcji znanej z Factory Method na całą rodzinę produktów, której zmiana na inną powinna odbywać się w postaci jednego kroku. Celem wzorca jest zdefiniowanie interfejsu do tworzenia takich rodzin obiektów, korzystając (podobnie jak w przypadku poprzedniego wzorca) z dedykowanych klas zajmujących się produkcją obiektów.



Klient, analogicznie do rozwiązania stosowanego w Factory Method, odwołuje się do klasy AbstractFactory służącej do tworzenia grupy różnych produktów. Każdy produkt jest tworzony przez osobną metodę, która sama stosuje wzorzec Factory Method. AbstractFactory jest klasą abstrakcyjną, tzn. nie definiuje, w jaki sposób mają być tworzone odpowiednie produkty; deklaruje jedynie obecność odpowiedzialnych za to metod. Tworzeniem konkretnych produktów zajmują się jej implementacje, w których każda z metod tworzy odpowiedni obiekt należący do typu danego produktu.

Klient postrzega instancje produktów wyłącznie poprzez zdefiniowane interfejsy AbstractProdyct. Poniżej warstwy abstrakcji znajdują się implementacje tych interfejsów (np. ProductA1 i ProductA2) które są tworzone właśnie przez obiekty ConcreteFactory.

Klient, wybierając odpowiednią fabrykę ConcreteFactory, decyduje jednocześnie o wyborze całej rodziny Produktów. To pozwala zmieniać całe rodziny produktów w prosty sposób – posługując się inną implementacją fabryki.

Abstract Factory: uczestnicy

- **Abstract Factory**
  - definiuje interfejs do tworzenia obiektów *Abstract Product*
- **Concrete Factory**
  - tworzy obiekty *Concrete Product* należące do jednej grupy
- **Abstract Product**
  - deklaruje interfejs obiektów *Product*
- **Concrete Product**
  - definiuje obiekt *Product*

Interfejs AbstractFactory definiuje osobne metody typu *factoryMethod()* dla każdego typu produktu, jaki ma tworzyć. Produkty nie są w żaden sposób ze sobą związane. Dopiero obiekt ConcreteFactory, będący implementacją klasy AbstractFactory, określa, jakie konkretne klasy zostaną użyte do konstrukcji produktów.

Klasy produktów również są widziane jako interfejsy AbstractProduct. Dzięki temu zmiana rodziny produktów jest przezroczysta z punktu widzenia klienta.

Zastosowanie tego wzorca pozwala w łatwy sposób zmieniać całe rodziny produktów, zmieniając tylko ich fabrykę. Ponadto, struktura wzorca pozwala łatwo wydzielić warstwę abstrakcji i implementacji, i to zarówno w przypadku fabryki, jak i produktu. Szczegóły implementacyjne obu typów klas są więc niewidoczne dla klienta, co przyczynia się do większej elastyczności systemu.

Dodawanie kolejnych rodzin produktów wiąże się z koniecznością zaimplementowania także nowej fabryki, która będzie dostarczać wspomniane produkty. W ten sposób obiekt fabryki i tworzone przez niego obiekty są związane ze sobą i tworzą hermetyczną całość.

Należy jednak pamiętać, że dodanie do wzorca kolejnego typu Product jest utrudnione, ponieważ wymaga modyfikacji wszystkich istniejących dotychczas fabryk. Dlatego wzorzec ten stosuje się w sytuacjach, w których zestaw produktów jest zamknięty.

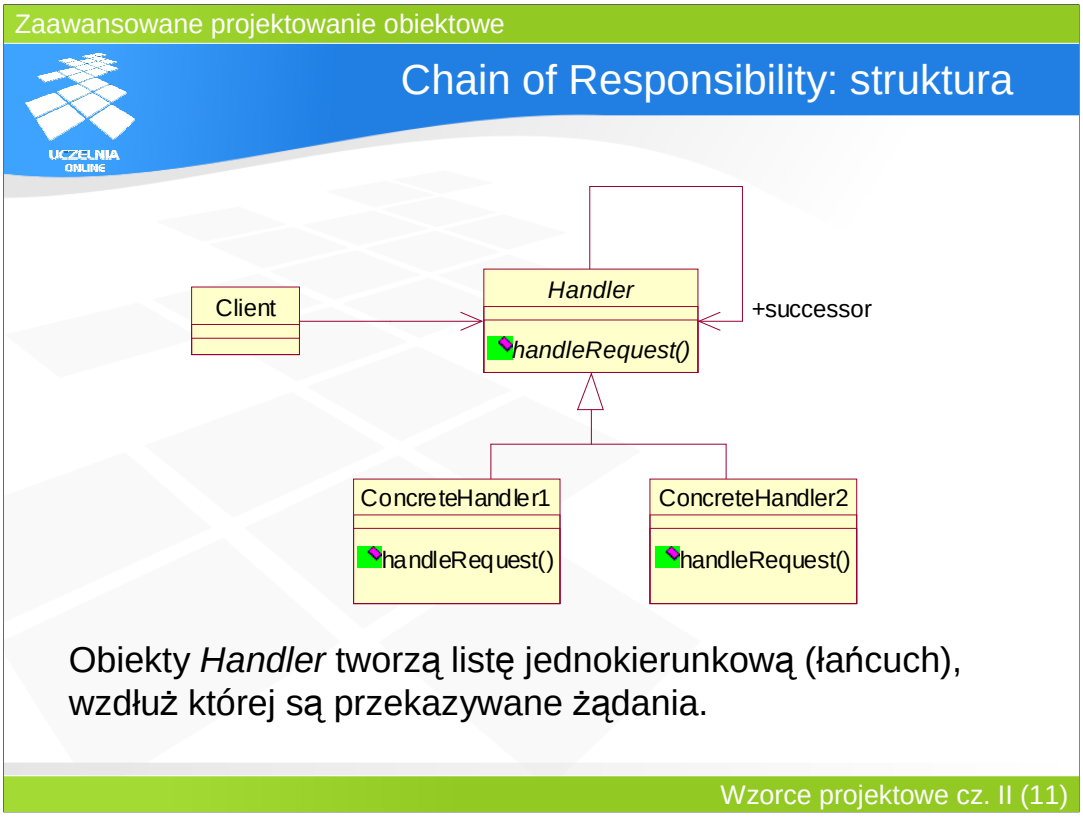
Wzorzec ten jest stosowany m.in. w bibliotece Java Swing do reprezentacji tzw. skórek (czyli mechanizmu umożliwiającego szybką zmianę wyglądu interfejsu użytkownika). Wszystkie implementacje okienek, przycisków, list i innych elementów GUI są produkowane przez wybraną fabrykę. Zmiana implementacji tej fabryki oznacza jednoczeną modyfikację zawartości ekranu.

Chain of Responsibility: cel

- Usunięcie powiązania pomiędzy nadawcą i odbiorcą żądania
- Umożliwienie wielu obiektom obsługi żądania

E. Gamma et al. (1995)

Wzorzec Chain of Responsibility jest strukturą, która definiuje łańcuch obiektów będących potencjalnymi odbiorcami i wykonawcami żądań klienta. Dzięki temu wiele obiektów ma możliwość obsługi żądania, a powiązania pomiędzy nadawcą i odbiorcą (jak i poszczególnymi potencjalnymi odbiorcami) stają się znacznie osłabione lub zostają usunięte.



Struktura tego wzorca jest bardzo prosta: obiekty typu Handler są powiązane ze sobą w postaci jednokierunkowej kolejki (albo łańcucha). Nadchodzące od klienta żądanie jest przekazywane wzdłuż tego łańcucha, gdzie każdy obiekt typu Handler ma szansę na ich obsłużenie. Co ważne, obiekty typu Handler są od siebie niezależne, tzn. nie wiedzą o sobie nic (poza abstrakcyjnym wskazaniem na obiekt następnika).

Chain of Responsibility: uczestnicy

- **Handler**
  - definiuje interfejs do obsługi żądań
- **Concrete Handler**
  - obsługuje jeden rodzaj żądania, pozostałe przekazuje do następnika w łańcuchu
  - posiada referencję typu *Handler* do następnika
- **Client**
  - inicjuje przetwarzanie, przekazując żądanie do pierwszego obiektu *Handler* w łańcuchu

Handler definiuje interfejs obsługi żądań. Zwykle jest to jedna metoda, która realizuje prosty algorytm: jeżeli dany obiekt ConcreteHandler jest w stanie obsłużyć żądanie, to obsługuje je; w przeciwnym wypadku (bądź w sytuacji, gdy wiele obiektów typu Handler może obsłużyć jedno żądanie) – przekazuje je do swojego następnika w łańcuchu. Charakterystyczna dla wzorca jest dowolna konfigurowalność łańcucha: żaden jego element nie musi posiadać wiedzy o rodzaju żądań obsługiwanych przez kolejne elementy, dlatego zmiany w jego strukturze nie mają wpływu na zachowanie.

Zadaniem klienta przy takiej strukturze jest przekazanie żądania pierwszemu elementowi łańcucha, który następnie dalej obsługuje żądanie.

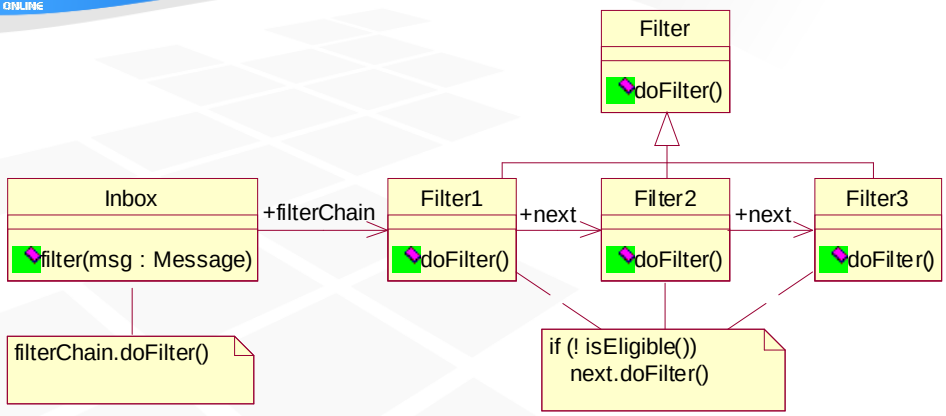
Zaletą tego wzorca jest znaczne ograniczenie powiązań pomiędzy klientem i każdym z obiektów Handler. Klient, przekazując żądanie, nie wie, który z obiektów Handler będzie je w rzeczywistości obsługiwał. Poszczególne ogniwa łańcucha są zorganizowane w postaci prostej kolejki jednokierunkowej, a ich wiedza o sobie nawzajem ogranicza się do abstrakcyjnego typu ogniwa. Nie znają swoich zadań ani klas, jakie implementują.

Taka struktura pozwala elastycznie przydzielać odpowiedzialność do poszczególnych ogniw: każdy z nich zajmuje się obsługą żądań jednego typu, a rozszerzenie łańcucha o kolejne elementy nie wpływa na sposób przetwarzania przez niego żądań. To z kolei przyczynia się do łatwiejszego testowania każdego ogniwa łańcucha z osobna: wystarczy zweryfikować, czy poprawnie obsługuje on żądania jednego typu.

Wadą takiej konstrukcji łańcucha jest brak gwarancji obsługi żądania: kolejne ogniwa mogą zrezygnować z zajęcia się nim. Co więcej, informacja o tym fakcie nie jest przekazywana klientowi. W tym celu stosuje się rozmaite rozwiązania pośrednie: umieszczając informację o obsłudze wewnątrz żądania (wówczas brak takiej informacji oznacza jego nieobsłużenie) lub zmieniając nieco strukturę przetwarzania.

Ponadto, błąd w implementacji filtra może skutkować nieprzekazaniem sterowania do następnika i przerwaniem łańcucha. Aby zminimalizować to ryzyko, w niektórych implementacjach klasa bazowa Filter posiada zaimplementowany na stałe mechanizm przekazywania sterowania do następnika, a programiście udostępniona jest tylko metoda dokonująca faktycznej obsługi żądania.

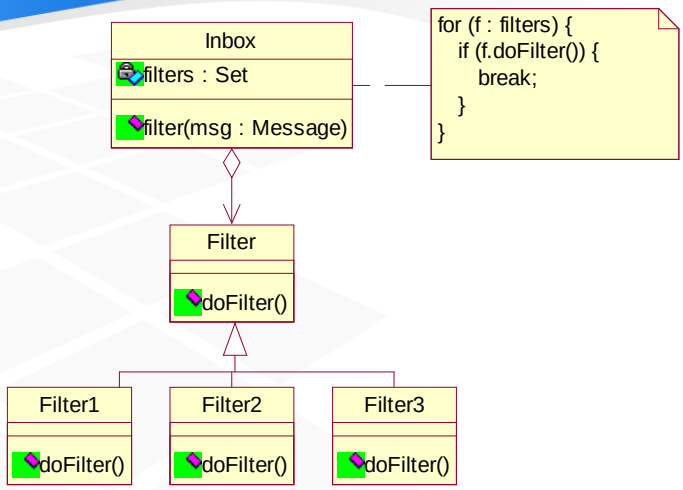
Chain of Responsibility: przykład 1



Obiekt *Inbox* wywołuje pierwszy obiekt *Filter* w łańcuchu. Kolejne filtry przekazują sobie sterowanie

Prostym przykładem tego wzorca jest np. mechanizm filtrów obecnych w większości klientów poczty elektronicznej. Wiadomość przychodząca do foldera Inbox jest przesyłana przez łańcuch zdefiniowanych przez użytkownika filtrów: każdy z nich może dokonać pewnej akcji na wiadomości, polegającej na przeniesieniu jej do innego foldera, zmianie jej priorytetu czy usunięciu jej. Zasada działania filltrów w takim systemie została przedstawiona na poprzednich slajdach każdy podejmuje decyzję (poprzez wywołanie metody *isEligible()*), czy konkretna wiadomość powinna być przez niego obsłużona, i przekazuje sterowanie dalej.

Chain of Responsibility: przykład 2



Obiekt *Inbox* wywołuje kolejno obiekty *Filter*. Nie występuje bezpośrenie przekazywanie sterowania z jednego filtra do drugiego.

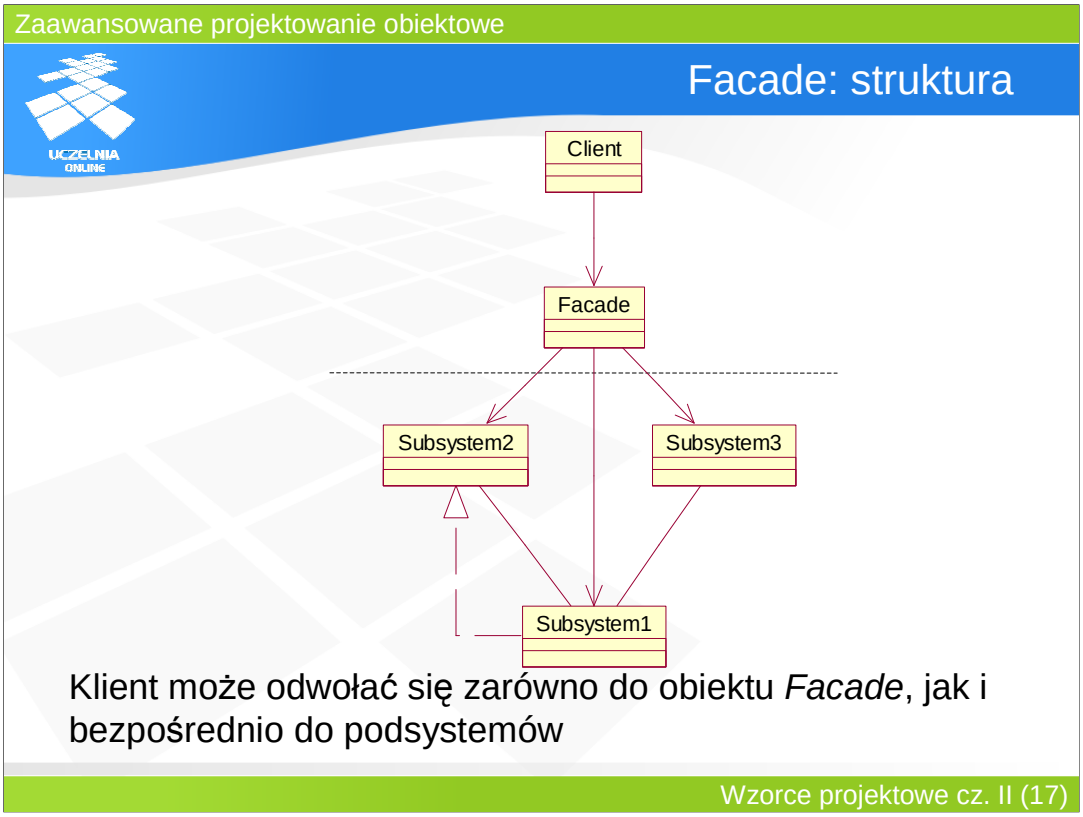
Z uwagi na wymienione wcześniej niedogodności, przede wszystkim możliwość przerwania łańcucha sterowania, możliwa jest także inna struktura przetwarzania, która nie posiada już topologii łańcucha. W tym rozwiązaniu pojawia się nowa rola: zarządcy, który posiada referencje do wszystkich filtrów. Zarządca (w tym przypadku jest nim także obiekt Inbox) wywołuje po kolei wszystkie filtry, które obsługują daną wiadomość lub nie. Jednak dzięki temu, że filtry nie przekazują sobie bezpośrednio sterowania, nie ma możliwości przerwania łańcucha, a ponadto informacja o nieobsłużeniu żądania może być w łatwy sposób przedstawiona klientowi przez zarządcę.

Facade: cel

- Dostarczenie jednorodnego interfejsu wyższego poziomu do zbioru różnych interfejsów w systemie
- Ukrycie złożoności podsystemów przed klientem

E. Gamma et al. (1995)

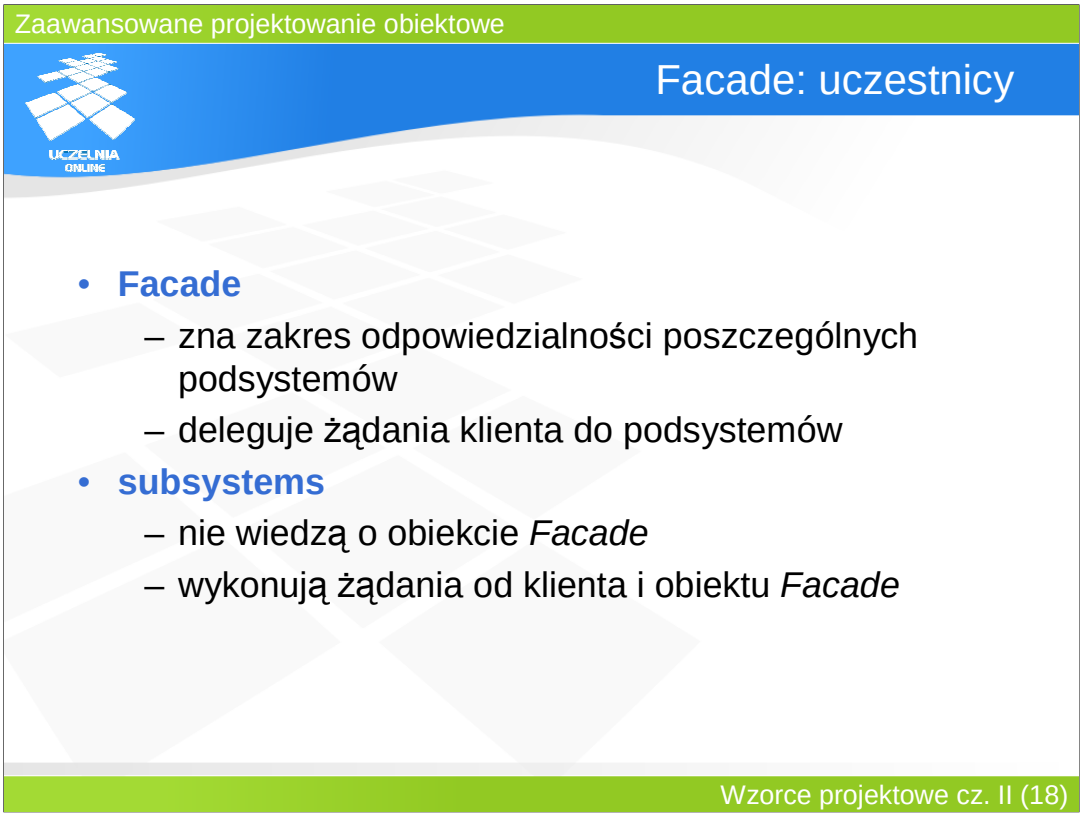
Wzorzec Facade jest prostym wzorcem strukturalnym, który ma na celu stworzenie alternatywnego interfejsu dostępu do grupy podsystemów. Dzięki temu klient jest odseparowany od ich złożoności i ma możliwość wyboru między skomplikowanym interfejsem natywnym (ale za to o pełniejszej funkcjonalności) oraz uproszczonym interfejsem dostarczonym przez wzorzec (realizującym najpotrzebniejszą i najczęściej wykorzystywaną funkcjonalność).



W skład wzorca wchodzi klasa (lub kilka klas), stanowiących fasadę grupy podsystemów. Fasada stanowi zatem dodatkową warstwę abstrakcji w dostępie do tych podsystemów i pozwala w łatwiejszy sposób posługiwać się nimi. Należy zwrócić uwagę, że stworzenie obiektu upraszczającego protokół komunikacji z podsystemami zwykle oznacza, że jego funkcjonalność będzie niepełna i ograniczona jedynie do najpopularniejszych operacji. W praktyce takie rozwiązanie jest jednak całkowicie akceptowalne.

Podsystemy nie muszą posiadać wiedzy o klasie Facade, natomiast ona musi znać ich strukturę i przeznaczenie. Żądania przesyłane przez klienta fasadzie są przez nią delegowane do odpowiednich podsystemów.

Fasada pod względem funkcjonalnym spełnia podobne zadanie co Proxy – pośredniczy w wywoływaniu operacji na faktycznym wykonawcy usług, jednak w odróżnieniu od niego, pozwala także na bezpośrednie odwołania do podsystemów. Klient zatem ma wybór dotyczący sposobu obsługi żądań.



We wzorcu uczestniczą obiekty Facade i podsystemy realizujące żądania klienta. Obiekt Facade zna strukturę i powiązania pomiędzy podsystemami, wie także, jak się nimi posługiwać w celu osiągnięcia określonego efektu. W ten sposób problem złożoności podsystemów, ich konfiguracji i interfejsów, który byłby przerzucony na klienta, jest hermetyzowany w postaci fasady.

Podsystemy nie są w żaden sposób modyfikowane przez zastosowanie wzorca: ich wiedza i zakres odpowiedzialności nie zmienia się. Wykonują one polecenia zlecanie albo bezpośrednio przez klienta, bądź przez fasadę.

Facade: konsekwencje

- Odseparowanie klienta od podsystemów
  - łatwiejsze korzystanie z podsystemów
  - niższe koszty pielęgnacji podsystemów
  - możliwość wymiany/rozbudowy podsystemów
- Elastyczny dostęp do podsystemów
  - klient może odwołać się do obiektu *Facade* lub bezpośrednio do podsystemów

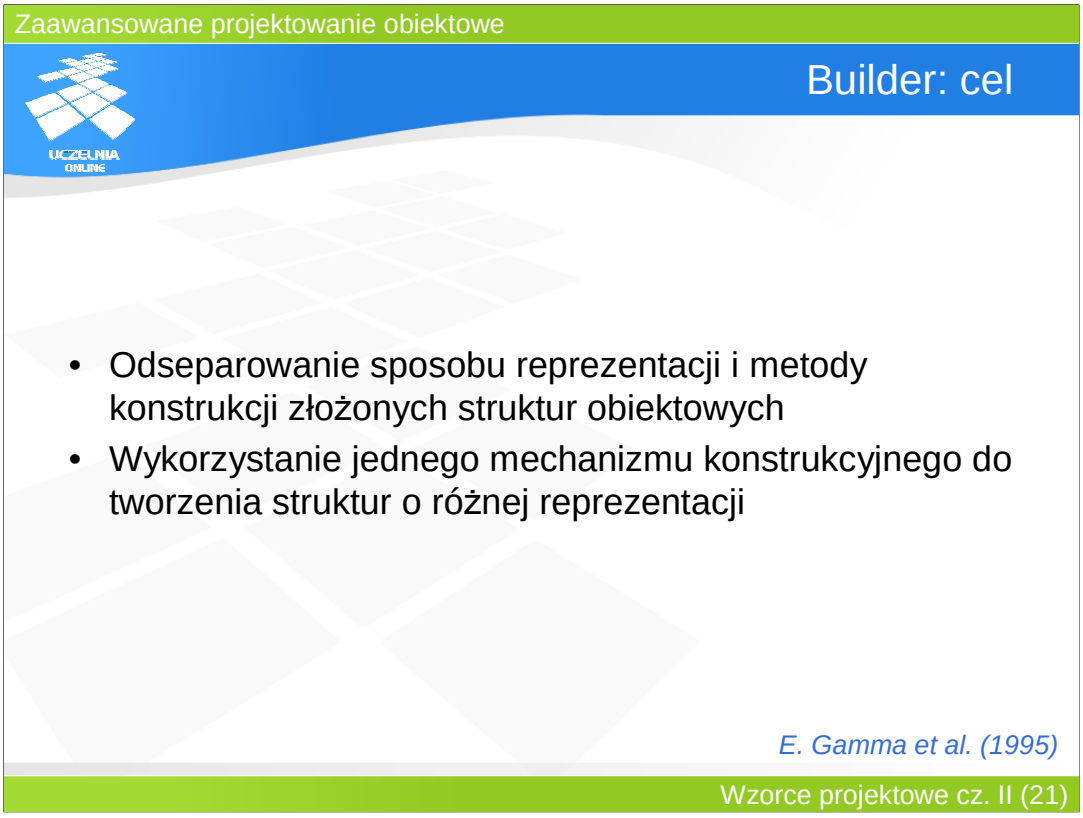
Wzorzec ten przede wszystkim ułatwia korzystanie z podsystemów: programista, wywołując odpowiednie metody fasady, nie musi znać szczegółów interfejsu podsystemu, ponieważ komunikacją z nim zajmie się fasada. Zmiany w podsystemach, ich wymiana lub rozbudowa są zatem niewidoczne dla klienta, co obniża koszty ich pielęgnacji. Z drugiej strony, klient ma nadal możliwość wyboru sposobu obsługi żądania pomiędzy fasadą i bezpośrednim skorzystaniem z podsystemów.

Facade: przykład

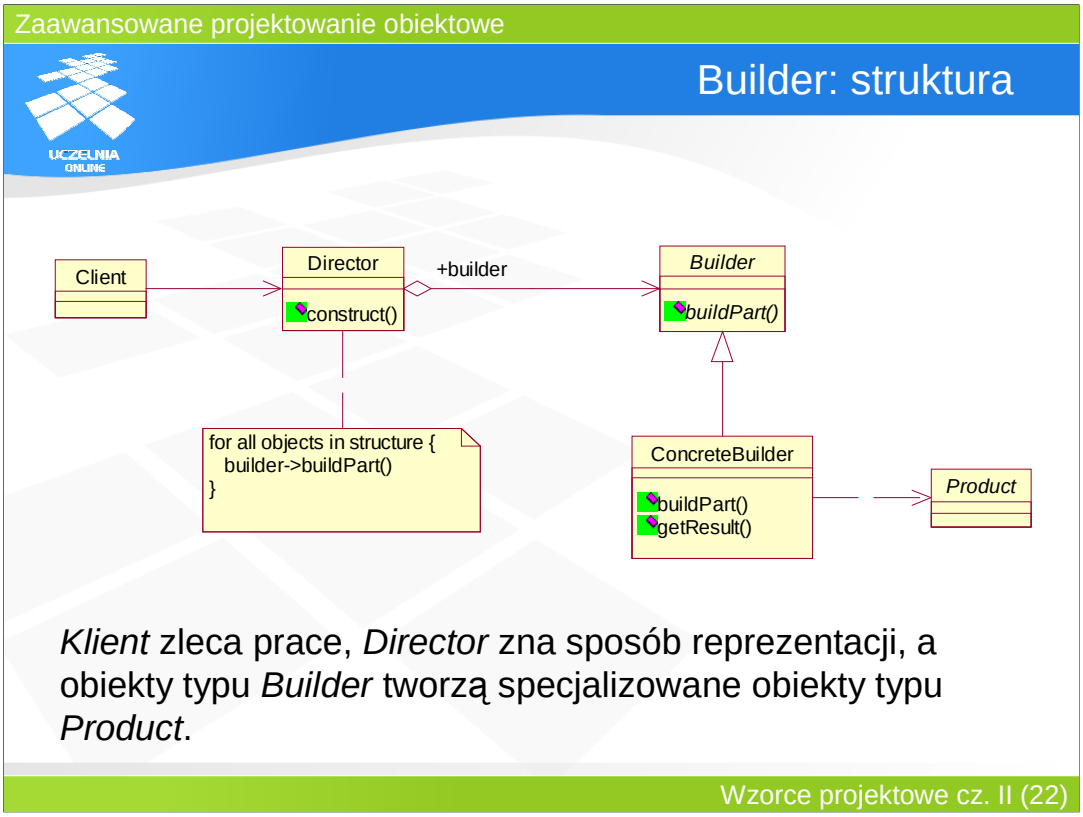
```
public class Email {                                        // facade
   MimeMessage msg = null;                                  // podsystem 1
   Session session = Session.getInstance(null, props); // podsystem 2

   public Email(String subject, String text) {
      msg = new MimeMessage(session);
      msg.setFrom(DEFAULT_FROM);
      msg.setSubject(subject);
      msg.setText(text, "UTF-8");
   }

   public void sendTo(String[] to) {
      msg.setRecipients(Message.RecipientType.TO, convert(to));
      Transport transport = session.getTransport("smtp");
      transport.sendMessage(msg, msg.getAllRecipients()
   }

   public void sendTo(String[] to, String[] cc) {
      msg.setRecipients(Message.RecipientType.TO, convert(to));
      msg.setRecipients(Message.RecipientType.CC, convert(Cc));
      Transport transport = session.getTransport("smtp");
      transport.sendMessage(msg, msg.getAllRecipients()
   }
}
```


W tym przykładzie klasa Email stanowi fasadę dla protokołu SMTP zaimplementowanego w postaci biblioteki Java Activation Framework. Ustalanie parametrów służących do stworzenia i wysłania wiadomości, ich konwersja do właściwych typów, interpretacja ich znaczenia są dość skomplikowane, dlatego dla najprostszych zastosowań zostały zdefiniowane metody fasady.

Użytkownik ma możliwość bezpośredniego posłużenia się podsystemami MimeMessage i Session, albo skorzystać z klasy Email.



Builder jest wzorcem strukturalnym i służy do tworzenia złożonych struktur obiektowych. Jego celem jest oddzielenie sposobu reprezentacji tych struktur od mechanizmu ich konstrukcji. Pozwala to także wykorzystać te same mechanizmy konstrukcyjne do tworzenia różnych struktur.



Struktura tego wzorca bardzo przypomina podział ról na budowie. Klient odpowiada za zlecenie wykonania prac. Odbiorcą jego zlecenia jest kierownik budowy (Director), który posiada projekt budowlany (algorytm realizacji struktury). Kierownik zna i dysponuje specjalistami od różnych zadań (reprezentowanymi przez klasy implementujące interfejs Builder). Każdy z fachowców, będący swego rodzaju wzorcem Factory, potrafi wykonywać produkty jednego rodzaju i przekazywać je kierownikowi. On, na podstawie projektu, składa elementy stworzone przez fachowców i konstruuje strukturę, a następnie przekazuje ją klientowi.

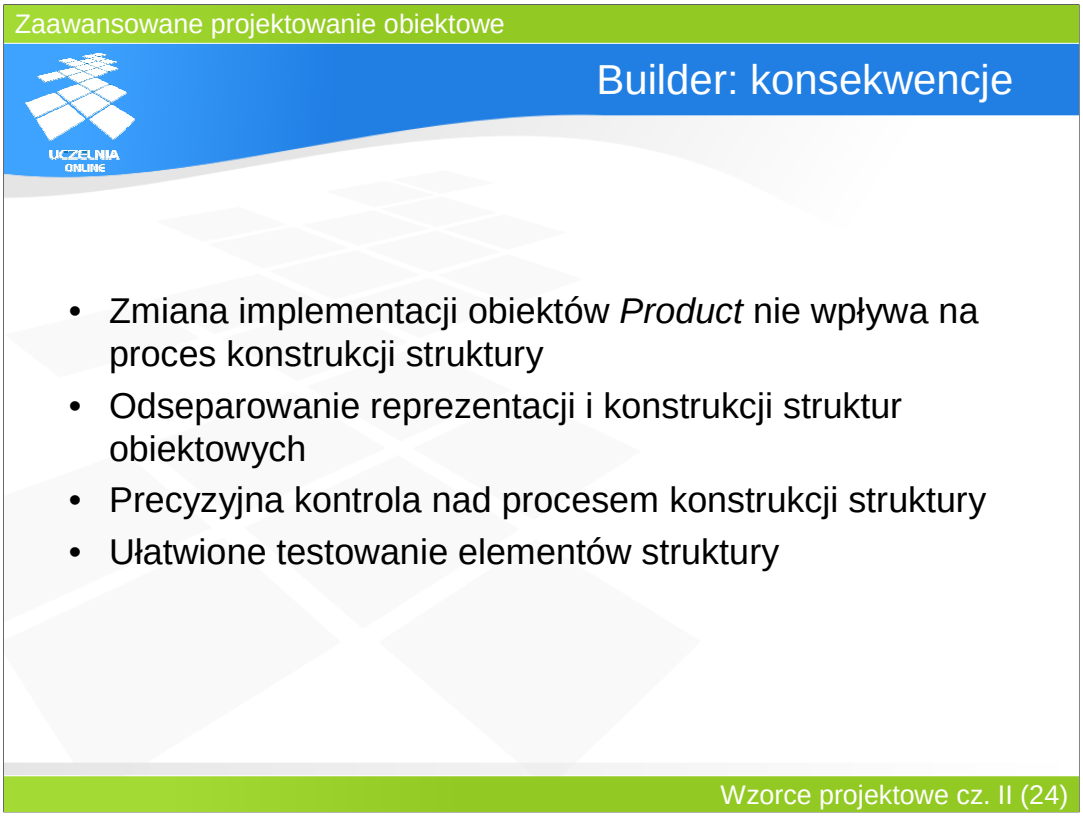
Builder: uczestnicy

- **Builder**
  - definiuje interfejs do tworzenia obiektów typu *Product*
- **Concrete Builder**
  - tworzy specjalizowany obiekt typu *Product*
- **Director**
  - zna sposób realizacji struktury i jej algorytm
  - zarządza grupą obiektów *Builder* i podzleca im wykonanie obiektów *Product*
- **Product**
  - reprezentuje element składowy struktury
  - posiada interfejs umożliwiający łączenie z innymi obiektami *Product*

We wzorcu występuje bardzo wyraźny podział na warstwy różniące się zakresem odpowiedzialności: obiekt Director odpowiada za zarządzanie obiektami typu Builder i zlecanie im prac; nie zajmuje się on jednak bezpośrednią realizacją zadań. Zarządzanie tymi obiektami wymaga, aby znał ich zakres odpowiedzialności, a zatem powiązania między nim a obiektami są dość silne. Ponadto zna on algorytm i sposób reprezentacji docelowej struktury danych, i na tej podstawie zleca prace.

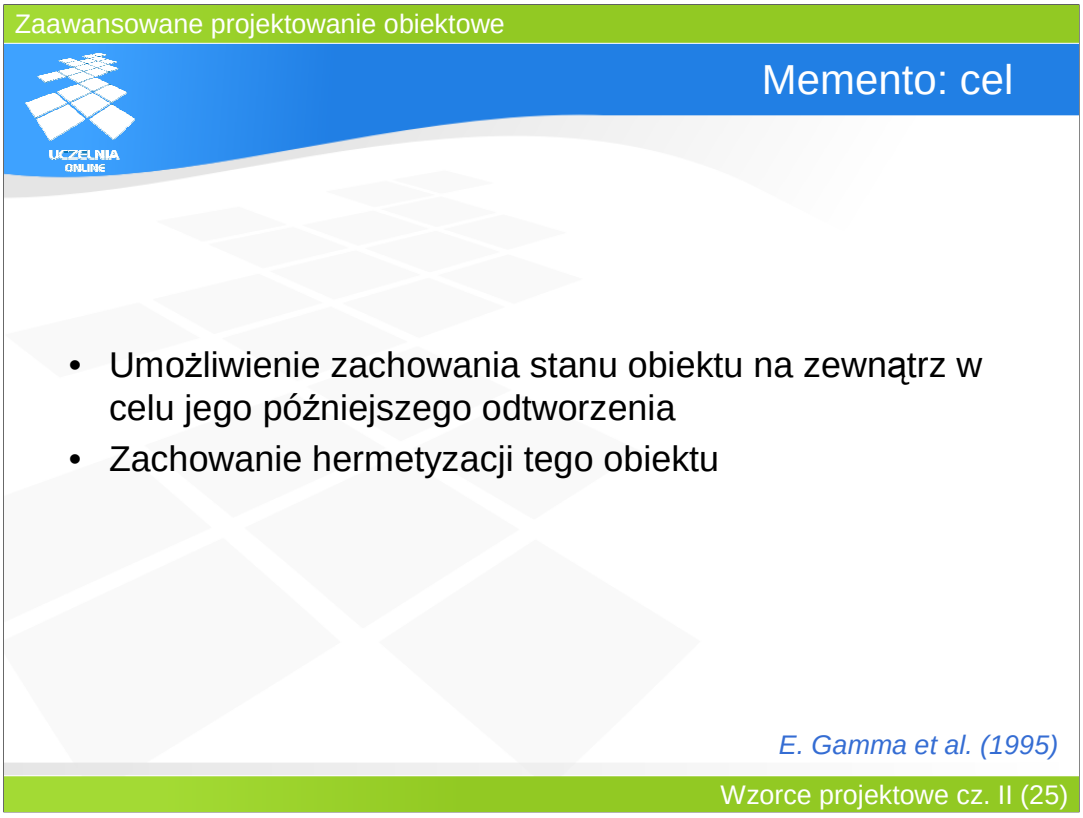
Obiekty Builder potrafią wytwarzać produkty: każdy ConcreteBuilder jest związany z produktem, który umie wyprodukować, natomiast nie zajmują się ich kompozycją ani rodzajem struktury. W ten sposób obiekty te mogą być wykorzystane do tworzenia różnych struktur, w zależności od potrzeb.

Wszystkie obiekty typu Product posiadają wspólny interfejs, definiujący metody pozwalające łączyć te obiekty w struktury.



Wzorzec ten, dzięki przejrzystemu i jednoznacznemu podziałowi odpowiedzialności pomiędzy poszczególne obiekty, zapewnia, że zmiana sposobu implementacji obiektów Product nie wpływa na sam proces konstrukcji. Podobnie, zmiana procesu konstrukcji nie wymaga zmian w implementacji elementów. Istnieje możliwość tworzenia wielu różnych struktur obiektowych bez modyfikacji pozostałych obiektów uczestniczących we wzorcu.

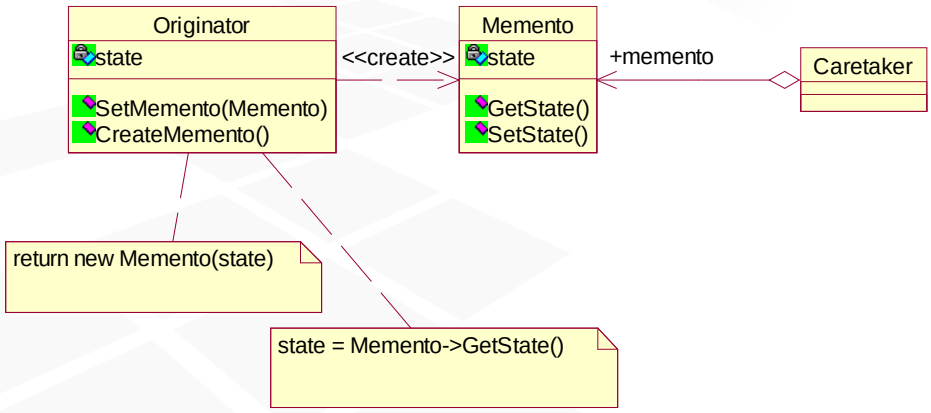
Taki podział zwiększa też kontrolę nad procesem konstrukcji struktury, a także umożliwia łatwe testowanie poszczególnych elementów.



Wzorzec Memento umożliwia zapamiętywanie, przechowywanie i odtwarzanie wewnętrznego stanu obiektu. Potrzeba taka często pojawia się w większości aplikacji.

Istotą wzorca jest jednak nie zarządzanie samym stanem, ale zapewnienie sposobu bezpiecznego dostępu do niego.

Memento: struktura



Originator zapisuje i odtwarza swój stan w postaci obiektu *Memento*. Obiekt *Caretaker* przechowuje obiekty *Memento*, ale nie ma dostępu do ich danych

Obiektem, którego stan należy przechować, jest Originator. Posiada on metody służące do utworzenia migawki stanu (*createMemento()*) oraz jej odczytania w celu przywrócenia wcześniejszego stanu (*setMemento()*). Obiekty-migawki stanu (Memento) przechowują stan obiektu Originator w postaci niezależnych instancji obiektu. Obiekty Memento posiadają metody *getState()* i *setState(),* służące do odczytania i zapisania stanu wewnątrz niego. Zarządzaniem kolejnymi migawkami stanu zajmuje się dedykowany obiekt Caretaker.

Jednak istotą wzorca nie jest sama możliwość tworzenia migawek stanu, ale zapewnienie im właściwego poziomu bezpieczeństwa. Wzorzec Memento pozwala na dostęp do stanu zapisanego w migawce wyłącznie jego właścicielowi, czyli obiektowi Originator, natomiast inne obiekty (w tym Caretaker) mogą tylko odwoływać się do całych obiektów, a metody *setState()* i *getState()* są dla nich niewidoczne.

Memento: uczestnicy

- **Memento**
  - przechowuje zapisany stan obiektu *Originator*
  - uniemożliwia dostęp do tego stanu obiektowi *Caretaker*
- **Originator**
  - tworzy obiekt *Memento* ze swoim aktualnym stanem
  - odtwarza stan na podstawie obiektu *Memento*
- **Caretaker**
  - przechowuje obiekty *Memento*
  - nie ma dostępu do ich zawartości

Szczególną rolę we wzorcu odgrywają dwie klasy: Originator, który jest twórcą i właścicielem wszystkich migawek stanu, oraz Memento, której obiekty przechowują stan Originatora.

Obiekt Originator musi posiadać możliwość utworzenia obiektu Memento oraz odczytania jego zawartości w celu przywrócenia na tej podstawie poprzedniego stanu. Memento przechowuje stan obiektu Originator zapisany w dowolnym momencie; pozwala też na dostęp do niego obiektowi Originator, natomiast uniemożliwia operacje na migawce wszelkim innym obiektom. Przykładem jest obiekt Caretaker, który zarządza utworzonymi migawkami, natomiast nie ma dostępu do ich zawartości.

Memento: konsekwencje

- Zachowanie hermetyzacji obiektu *Memento*
- Uproszczenie obiektu *Originator*
  - odpowiedzialność za zapis stanu przeniesiona na *Memento*
- Podwójny interfejs obiektu Memento
  - wąski: dla obiektu *Caretaker*
  - szeroki: dla obiektu *Originator*
- Potencjalny wzrost złożoności pamięciowej
  - stan może być obszerny
  - *Caretaker* nie zna tego rozmiaru i nie może optymalizować sposobu zarządzania nim

W ten sposób obiekt Memento posiada dwa logiczne interfejsy: szeroki, umożliwiający pełen dostęp do ich zawartości, przeznaczony wyłącznie dla obiektu Originator, oraz wąski, w praktyce blokujący dostęp do większości metod, przeznaczony dla pozostałych obiektów, w tym obiektu Caretaker.

Takie rozwiązanie przede wszystkim zachowuje hermetyczność obiektu Memento, ale również upraszcza obiekt Originator i zmniejsza jego zakres odpowiedzialności. Nie musi już on zajmować się w żaden sposób przechowywaniem migawek stanu, usuwaniem ich etc; Funkcje te zostały wydzielone do obiektów Memento i Caretaker.

Pełna hermetyzacja obiektów Memento w stosunku do klasy Caretaker ma także pewne wady: stan przechowywany w tych obiektach może mieć znaczny rozmiar, i zarządzanie nim może wymagać optymalizacji, stosowania heurystycznych algorytmów usuwania niektórych migawek etc. Niestety, ponieważ obiekt Caretaker nie może stwierdzić rozmiaru migawki, nie może również podjąć skutecznego działania w tym kierunku.

Memento: implementacja

Realizacja podwójnego interfejsu wymaga wsparcia ze strony języka programowania

- **Java**: klasy wewnętrzne

  Klasa wewnętrzna jest znana tylko swojej klasie zewnętrznej

- **C++:** klasy zaprzyjaźnione

  Klasy zaprzyjaźnione są uprzywilejowane w dostępie do swoich składowych niepublicznych

Struktura wzorca jest dość oczywista i nie wymaga komentarza. Warto jednak zastanowić się nad sposobem zapewnienia zróżnicowanego dostępu do obiektów Memento dla dwóch różnych klas. Implementacja wzorca w znacznym stopniu zależy od możliwości oferowanych przez język programowania. W przypadku języka C++ może to być zaprzyjaźnienie klas, które pozwala wybranym klasom odwoływać się do swoich składowych jak do elementów prywatnych. Niestety, istnieje grupa języków nie posiadających takich możliwości. W języku Java możliwym rozwiązaniem jest zastosowanie klasy wewnętrznej do reprezentacji obiektu Memento. W ten sposób jedynie jej klasa zewnętrzna posiada dostęp do jej składowych niepublicznych, natomiast inne klasy nie mają o niej żadnej wiedzy.

Memento: przykład

```
public class Account {
   private int balance = 0;

   public void credit(int amount) {
      balance += amount;
   }
   public void debit(int amount) {
      balance -= amount;
   }
   public void setMemento(Memento memento) {
      memento.restoreState();
   }
   public Memento createMemento() {
      Memento mementoToReturn = new Memento();
      mementoToReturn.setState();

      return mementoToReturn;
   }
```

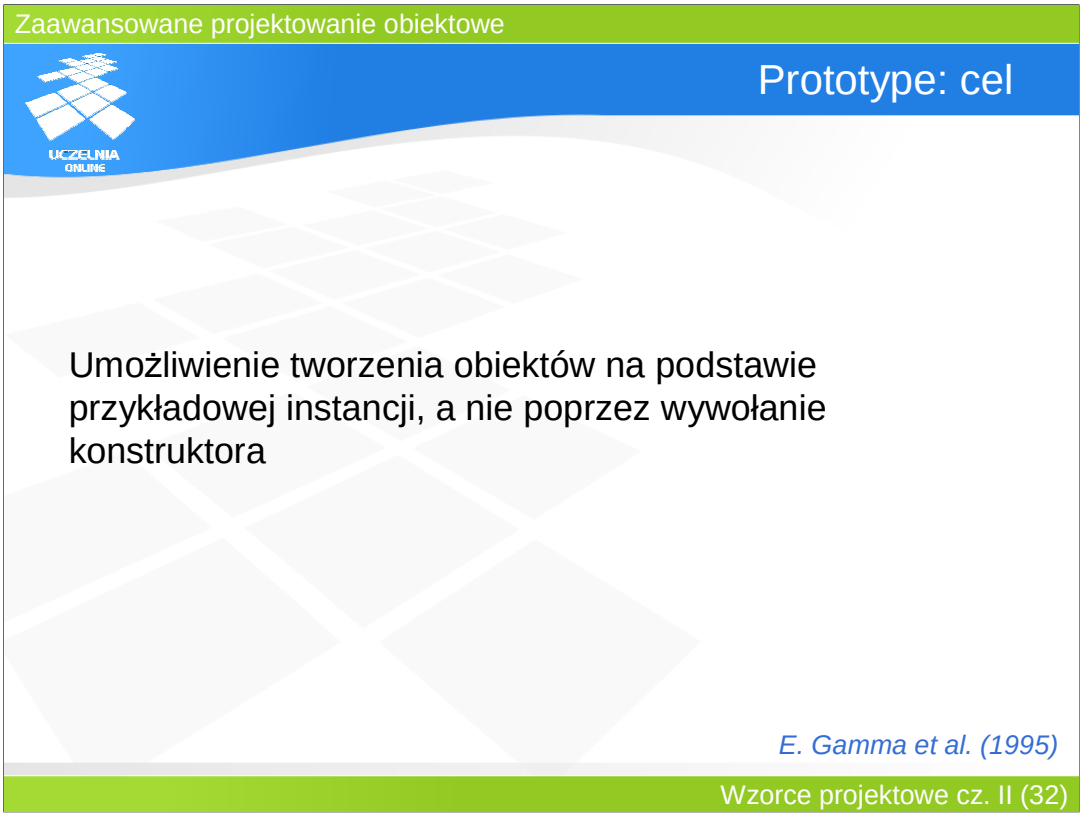

Jak przykład rozważmy klasę Account, której stanem jest zmienna *balance* reprezentująca saldo przechowywane na rachunku bankowym. Poza metodami biznesowymi *credit()* i *debit()* klasa ta posiada metodę *setMemento(),* służącą do odtworzenia stanu na podstawie migawki, oraz *createMemento(),* tworzącą nową migawkę.

Memento: przykład cd.

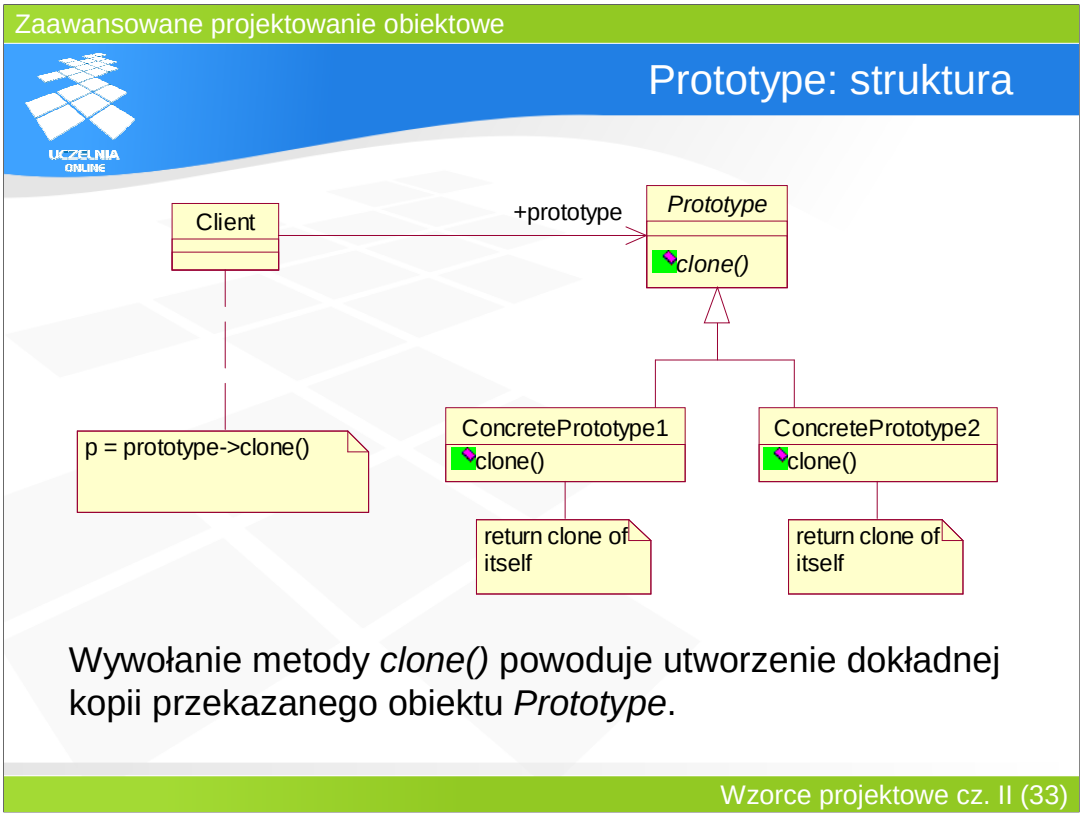
```
public class Account {
   // continued...
   class Memento {
      int mementoBalance = 0;

      private void setState() {
         mementoBalance = balance;
      }
      private void restoreState() {
         balance = mementoBalance;
      }
   }
}
```

Prywatna klasa wewnętrzna pozwala osiągnąć efekt podwójnego interfejsu: tylko klasa zewnętrzna może odwoływać się do jej stanu

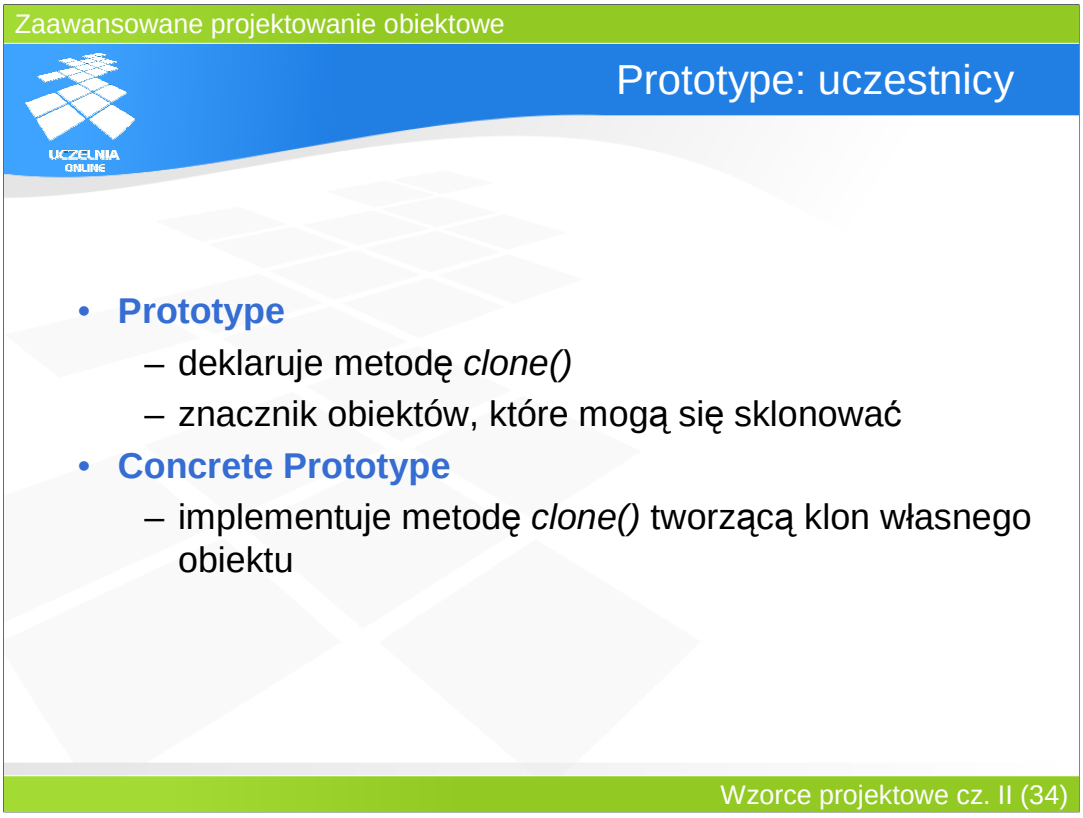
Wewnątrz klasy Account jest zdefiniowana klasa Memento, posiadająca pole mementoBalance, służące do przechowania wartości salda w danym momencie. Metody *setState()* oraz *restoreState()* są widoczne jedynie dla jej nadklasy, natomiast inne obiekty nie mają do nich dostępu. Rolę obiektu Caretaker może pełnić dowolna zmienna typu Account.Memento, która przechowuje instancję migawki. W ten sposób założenia dotyczące podwójnego interfejsu zostały spełnione.



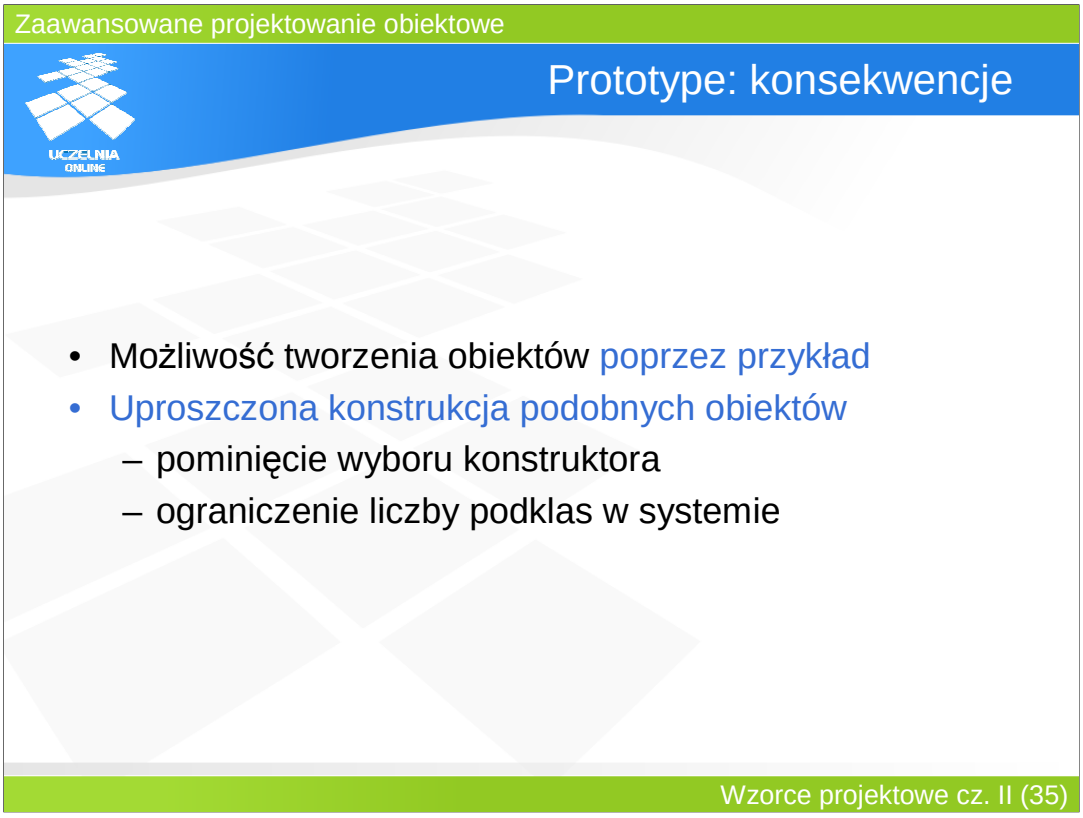
Wzorzec Prototype należy do grupy wzorców kreacyjnych, jednak sposób tworzenia przez niego obiektów jest zupełnie inny niż w przypadku innych rozwiązań z tej grupy, np. Factory Method czy Singletona. Celem jego stosowania jest tworzenie nowych obiektów poprzez klonowanie już istniejącego wzorcowego obiektu.



Obiekt poddający się klonowaniu, Prototype, posiada metodę *clone()*. Metoda ta jest implementowana we wszystkich jego obiektach potomnych w ten sposób, że tworzy ona dokładną kopię bieżącego obiektu. Jedyna różnica pomiędzy oryginałem i klonem polega na odrębnej tożsamości obiektu (w większości języków tożsamość jest rozstrzygana na podstawie referencji do tego obiektu). Klient, żądając utworzenia kopii obiektu Prototype, wywołuje w istniejącej instancji tego obiektu metodę *clone()*, która zwraca jego klon.



We wzorcu uczestniczy właściwie tylko jedna klasa: Prototype, która posiada możliwość sklonowania obiektów własnej klasy poprzez wywołanie metody *clone()*.



Najważniejszą konsekwencją zastosowania tego wzorca jest całkowita zmiana sposobu tworzenia obiektów. Typowy sposób polega na podaniu wprost klasy i konstruktora użytego do stworzenia instancji obiektu. Jednak nawet w przypadku wzorca Factory Method oznacza to ograniczenie producenta w zakresie typów obiektów, jakie może stworzyć.

Ta niedogodność nie występuje we wzorcu Prototype: dowolny obiekt, jeżeli tylko posiada możliwość sklonowania się, może utworzyć nowy obiekt identyczny ze sobą. Zatem metoda służąca do produkcji obiektów przyjmowałaby jako parametr instancję obiektu do sklonowania, ignorując jego rzeczywistą klasę, i zwracała jego kopię.

Dzięki temu możliwe jest uproszczone tworzenie serii obiektów identycznych lub jedynie nieznacznie różniących się od siebie.

Prototype: przykład

```
public class Employee implements Cloneable {
   private String name = null;

   public Employee(String name) {
       this.name = name;
   }
   public Object clone() throws CloneNotSupportedException {
       // tutaj: specyficzne operacje związane z klonowaniem
       return super.clone();
   }
}
```

```
Employee emp = new Employee("John Smith");
Employee emp2 = (Employee) emp.clone();
assertEquals(emp, emp2);
```


W języku Java wzorzec ten jest zaimplementowany bezpośrednio w maszynie wirtualnej. Każdy obiekt posiada metodę *clone()*, a co za tym idzie – potencjalną możliwość klonowania siebie. Jednak aby skorzystać z tej możliwości, konieczne jest zaimplementowanie w wybranej klasie interfejsu *Cloneable*. Interfejs ten nie definiuje żadnych metod, a jedynie pełni rolę znacznika, wskazującego, że dana klasa posiada uprawnienie do klonowania samej siebie. Próba wywołania tej metody bez zaimplementowania interfejsu powoduje zgłoszenie wyjątku.

Domyślnie wywołanie metody *clone()* powoduje utworzenie tzw. płytkiej kopii obiektu, tzn. obiekty zależne są kopiowane jako referencje, a nie jako obiekty. Płytka kopia jest bezpieczna, ponieważ nie powoduje rekurencyjnego alokowania znacznych obszarów pamięci. Jeżeli istnieje potrzeba realizacji tzw. głębokiej kopii, zadanie jej zaimplementowania leży po stronie programisty.

State/Strategy: cel

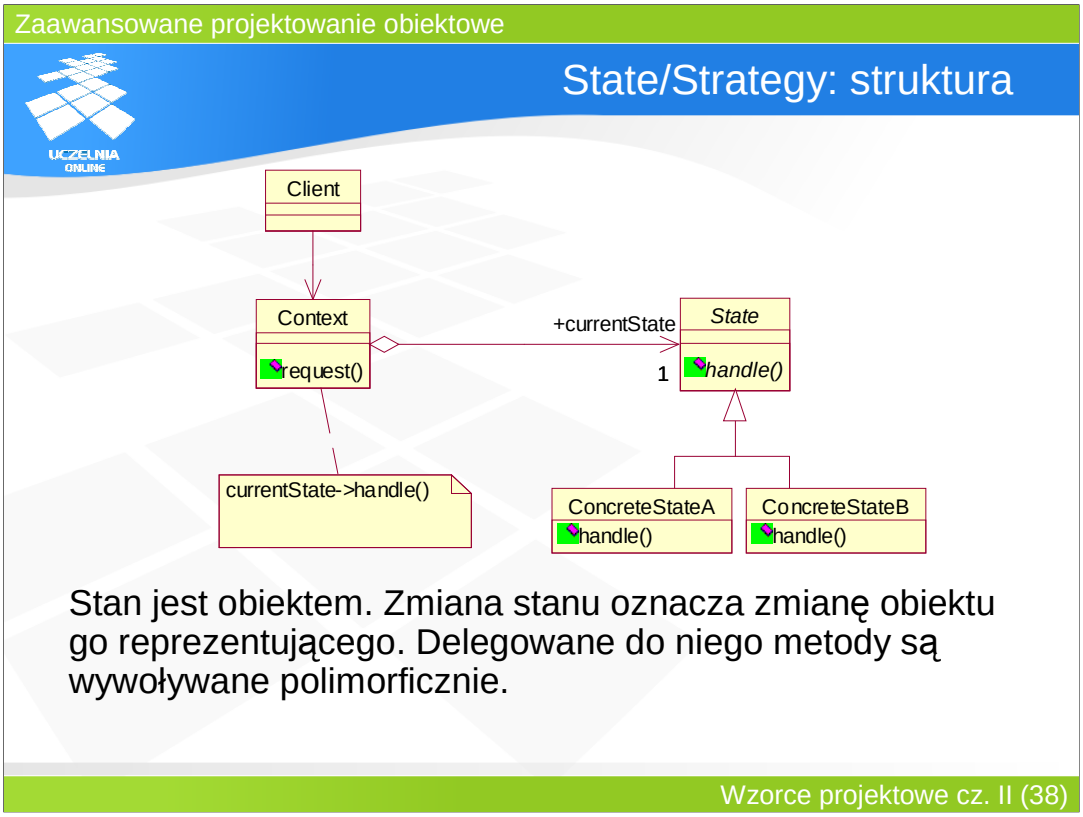
- **State**
  - umożliwienie zmiany zachowania obiektu w momencie zmiany jego stanu
  - pozorna zmiana klasy obiektu
- **Strategy**
  - umożliwienie zmiany algorytmu realizacji pewnej funkcji
  - algorytmy są wymienne

E. Gamma et al. (1995)

Wzorce State i Strategy zostaną omówione wspólnie, ponieważ mają identyczną strukturę i zbliżone cele. Dotyczą one funkcjonalnej zmiany zachowania obiektu w trakcie wykonywania programu. W ten sposób pozornie obiekt ten zmienia klasę, do której należy.

W przypadku wzorca State celem jest zmiana zachowania obiektu w zależności od stanu, w jakim obiekt się znajduje.

Wzorzec Strategy służy do modelowania algorytmu realizacji pewnej czynności, który może zostać określony i zmieniony w trakcie wykonywania programu.



W przypadku wzorca State centralnym obiektem jest Context. Jego metody wywoływane przez klientów delegują żądania do skojarzonego z nim relacją kompozycji obiektu typu State, reprezentującego jego stan. Metody obiektu State są polimorficzne, czyli wraz ze zmianą tego obiektu zmienia się też ich funkcjonalność. W ten sposób, gdy zachodzi zmiana skojarzonego z obiektem Context obiektu State, zmieniają się też zachowanie metod kontekstu. Pozornie zatem obiekt Context zmienia klasę, do której należy.

Wzorzec Strategy stosuje podobne rozwiązanie, tylko na nieco większą skalę. Obiekt Context realizuje pewien algorytm, którego poszczególne kroki mogą zmieniać się w zależności od wyboru konkretnego algorytmu. Z obiektem tym skojarzony jest (także za pomocą kompozycji) obiekt algorytmu, którego metody implementują zmieniające się kroki. Zmiana obiektu algorytmu powoduje zmianę zachowania obiektu Context.

W obu przypadkach najważniejszą zaletą jest możliwość zmiany skojarzonego obiektu (stanu lub algorytmu) w trakcie działania programu, bez potrzeby jego rekompilacji.

State/Strategy: uczestnicy

- **Context**
  - posiada referencję do obiektu reprezentującego bieżący stan
- **State**
  - definiuje interfejs pozwalający hermetyzować zachowanie związane z każdym stanem
- **Concrete State**
  - definiuje własne metody implementujące zachowanie specyficzne dla tego stanu

Obiekt Context posiada referencję do obiektu typu State, wskazującą na bieżący stan. W obiekcie State zdefiniowane są wszystkie metody, których zachowanie zależy od stanu obiektu Context.

State/Strategy: konsekwencje

- Podział zachowania obiektu wg stanów
  - kod związany ze jednym stanem jest zapisany w jednym obiekcie
- zmiana stanu jest realizowana przez zmianę obiektu stanu na inny
- ochrona przed stanem niespójnym
- możliwość współdzielenia obiektów *State*
  - obiekty *State* zwykle definiują tylko zachowanie
  - obiekty *State* zwykle są bezstanowe

Zastosowanie wzorca pozwala modyfikować zachowanie obiektów tak jakby zmieniała się ich klasa – i to jest najważniejszy cel i konsekwencja tego wzorca. Istnieje natomiast grupa efektów pośrednich, ale o dość interesujących właściwościach. Hermetyzacja stanu w postaci niezależnych klas pozwala na jednorazową, niepodzielną zmianę tego stanu, bez wprowadzania stanów niespójnych czy nieoznaczonych. Jeżeli obiekty State nie przechowują informacji (w większości przypadków może ona być zapamiętana w obiekcie Context, ponieważ ona nie ulega zmianie), a jedynie definiują zachowanie, wówczas – paradoksalnie – obiekty te, reprezentujące stan, są bezstanowe i mogą być współdzielone między wiele obiektów Context.

State: przykład

```
public class Account {
   private int balance = 0;
   private String owner = null;
   private boolean isOpen = false;

   public Account(String owner, int balance) {
      this.owner = owner;
      this.balance = balance;
      this.isOpen = true;
   }
   public void credit(int amount) {
      if (isOpen) {
         balance += amount;
      } else {
         alert("Konto nieaktywne!");
      }
   }
}
```


Przykładem ponownie będzie rachunek bankowy. Tym razem, obok stanu biznesowego, przechowującego informacje związane z rachunkiem (saldo, właściciel), posiada on także zmienną *isOpen*, określającą, czy rachunek jest aktywny, czy nie. Zmienna ta ma wpływ na działanie niektórych metod biznesowych: wykonanie operacji *credit()* nie jest możliwe, jeżeli zmienna *isOpen* ma wartość *false*.

State: przykład cd.

```
public interface AccountState {
    public void credit(Account acc, int amount);
}

public class AccountOpen implements AccountState {
    public void credit(Account acc, int amount) {
        acc.balance += amount;
    }
}

public class AccountClosed implements AccountState {
    public void credit(Account acc, int amount) {
        alert("The account is closed!");
    }
}
```

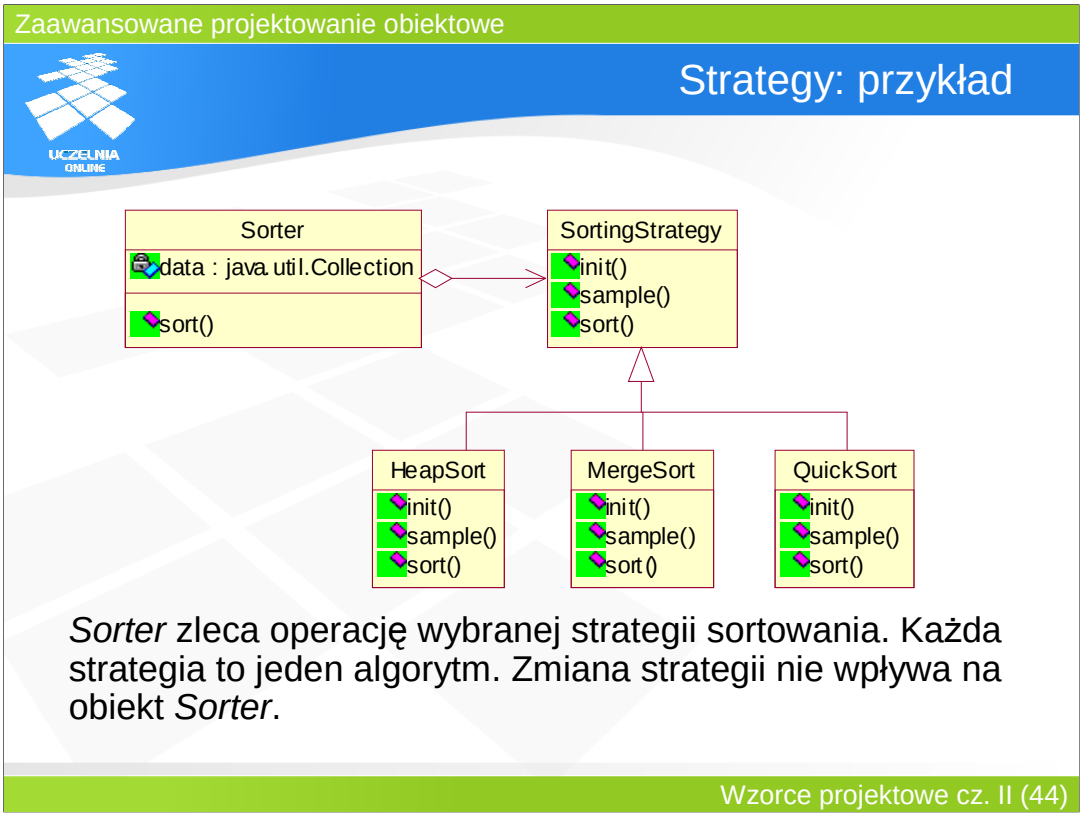

Aby zastosować wzorzec State w tym przypadku, należy zdefiniować interfejs AccountState oraz klasy reprezentujące stan aktywności i nieaktywności rachunku. Ten interfejs i implementujące go klasy posiadają metodę *credit()*, której zachowanie jest różne w zależności od klasy: AccountOpen realizuje tę metodę bezwarunkowo, natomiast AccountClosed – również bezwarunkowo ją blokuje.

State: przykład cd.

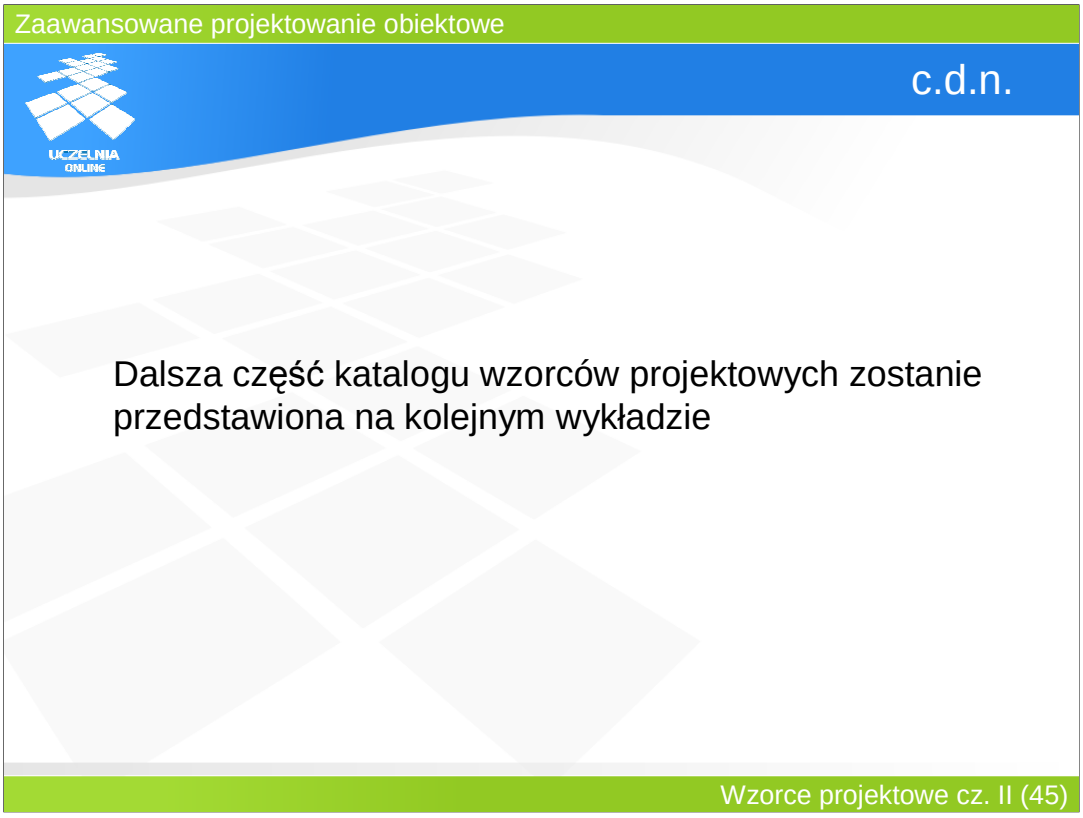
```
public class Account {
   private int balance = 0;
   private String owner = null;
   private AccountState state = null;

   public Account(String owner, int balance) {
      this.owner = owner;
      this.balance = balance;
      this.state = new AccountOpen();
   }
   public void credit(int amount) {
      this.state.credit(this, amount); // delegacja
   }
   public void close() {
      this.state = new AccountClosed();
   }
}
```


W klasie Account pole *isOpen* jest zastąpione poprzez referencję typu AccountState wskazującą na obiekt reprezentujący bieżący stan, przy czym domyślnym stanem początkowym jest stan aktywności (AccountOpen). Metoda *credit()* w klasie Account jest delegowana do obiektu stanu, dzięki czemu zmiana tego obiektu spowoduje inną obsługę tego komunikatu.

Metoda *close()* powoduje zmianę bieżącego obiektu stanu na AccountClosed – od tego momentu metoda *credit()* jest zablokowana.



Drugi przykład dotyczy wzorca Strategy. Klasa Sorter wykonuje sortowanie wewnętrznej kolekcji. Ponieważ istnieją różne algorytmy sortowania, dlatego realizacja metody *sort()* jest delegowana do aktywnego algorytmu, stanowiącego implementację klasy SortingStrategy. Metody tej klasy to kroki algorytmu. Każdy algorytm może realizować je w charakterystyczny dla siebie sposób. Zmiana algorytmu sortowania jest realizowana wyłącznie przez zmianę obiektu reprezentującego ten algorytm: jest przezroczysta z punktu widzenia obiektu Sorter.



Ostatnia część katalogu przekształceń refaktoryzacyjnych, zostanie przedstawiona podczas kolejnego wykładu.