Wzorce projektowe

Gniewomir Zięba 40880

Czym jest wzorzec projektowy

- Wzorce projektowe - to wielokrotnie powtórzone (powtarzające się) i pozytywnie zweryfikowane schematy rozwiązań często spotykanych problemów projektowych.
- Wzorce projektowe dotyczą architektury całej aplikacji, a nie pojedynczej klasy,

Typy wzorców projektowych

- Wzorce konstrukcyjne - opisujące proces tworzenia nowych obiektów; ich zadaniem jest tworzenie, inicjalizacja oraz konfiguracja obiektów, klas oraz innych typów danych
- Wzorce strukturalne - opisujące struktury powiązanych ze sobą obiektów
- Wzorce behawioralne - opisujące zachowanie i odpowiedzialność współpracujących ze sobą obiektów

Singleton

- Czasami ważne jest, aby mieć tylko jedną instancję dla klasy. Na przykład w systemie powinien być tylko jeden menedżer okien (tylko system plików lub tylko bufor wydruku). Zwykle singletony są używane do scentralizowanego zarządzania zasobami wewnętrznymi lub zewnętrznymi i zapewniają globalny punkt dostępu do siebie.
-
- Wzorzec singleton jest jednym z najprostszych wzorców projektowych: obejmuje tylko jedną klasę, która jest odpowiedzialna za zapewnienie, że nie ma więcej niż jednej instancji; robi to poprzez tworzenie instancji i jednocześnie zapewnia globalny punkt dostępu do tej instancji. W ten sposób klasa singleton zapewnia, że ta sama instancja może być używana z dowolnego miejsca, co zapobiega bezpośredniemu wywołaniu konstruktora singleton.

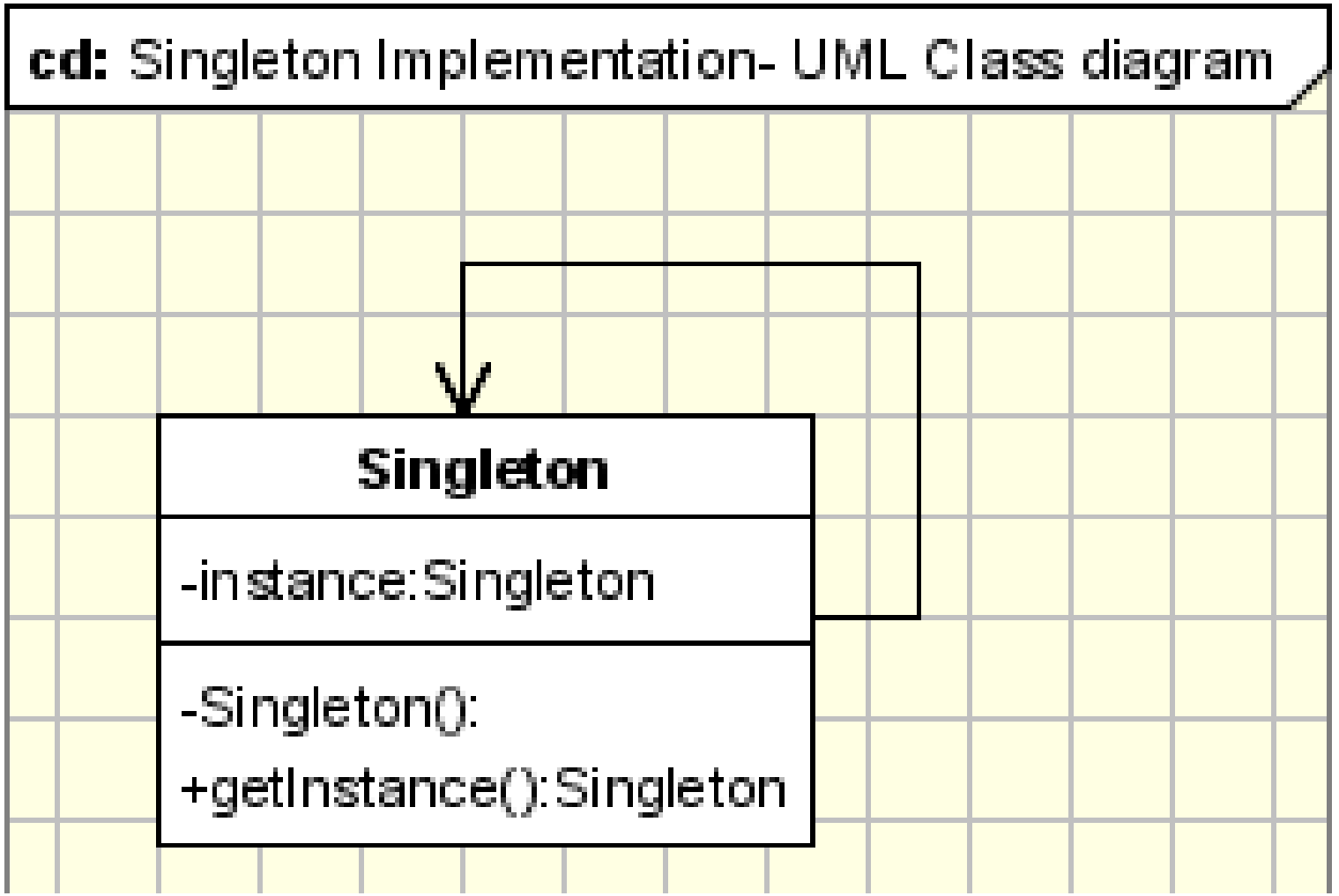

cd: Singleton Implementation- UML Class diagram
Singleton
-instance:Singleton
-Singleton():
+getInstance():Singleton

Diagram UML

Wady i zalety

+ pojedyncza zmienna nie otrzyma pamięci, dopóki właściwość lub funkcja wyznaczona do zwrócenia odwołania nie zostanie najpierw wywołana.

+ dostęp do tego samego obiektu z każdego miejsca kodu

- Odchodzi od zasady pojedynczej odpowiedzialności. Singleton musi stworzyć swoją instancję wraz z innymi funkcjami klasy.

- Singletony mogą ukrywać zależności.

Fabryka

- W programowaniu opartym na klasach wzorzec fabryki jest wzorcem kreacyjnym, który wykorzystuje metody fabryczne do rozwiązywania problemu tworzenia obiektów bez konieczności określania dokładnej klasy obiektu, który zostanie utworzony. Odbywa się to poprzez tworzenie obiektów przez wywołanie metody fabrycznej - albo określonej w interfejsie i zaimplementowaną przez klasy potomne, albo zaimplementowaną w klasie bazowej i opcjonalnie nadpisaną przez klasy pochodne - zamiast wywoływania konstruktora.

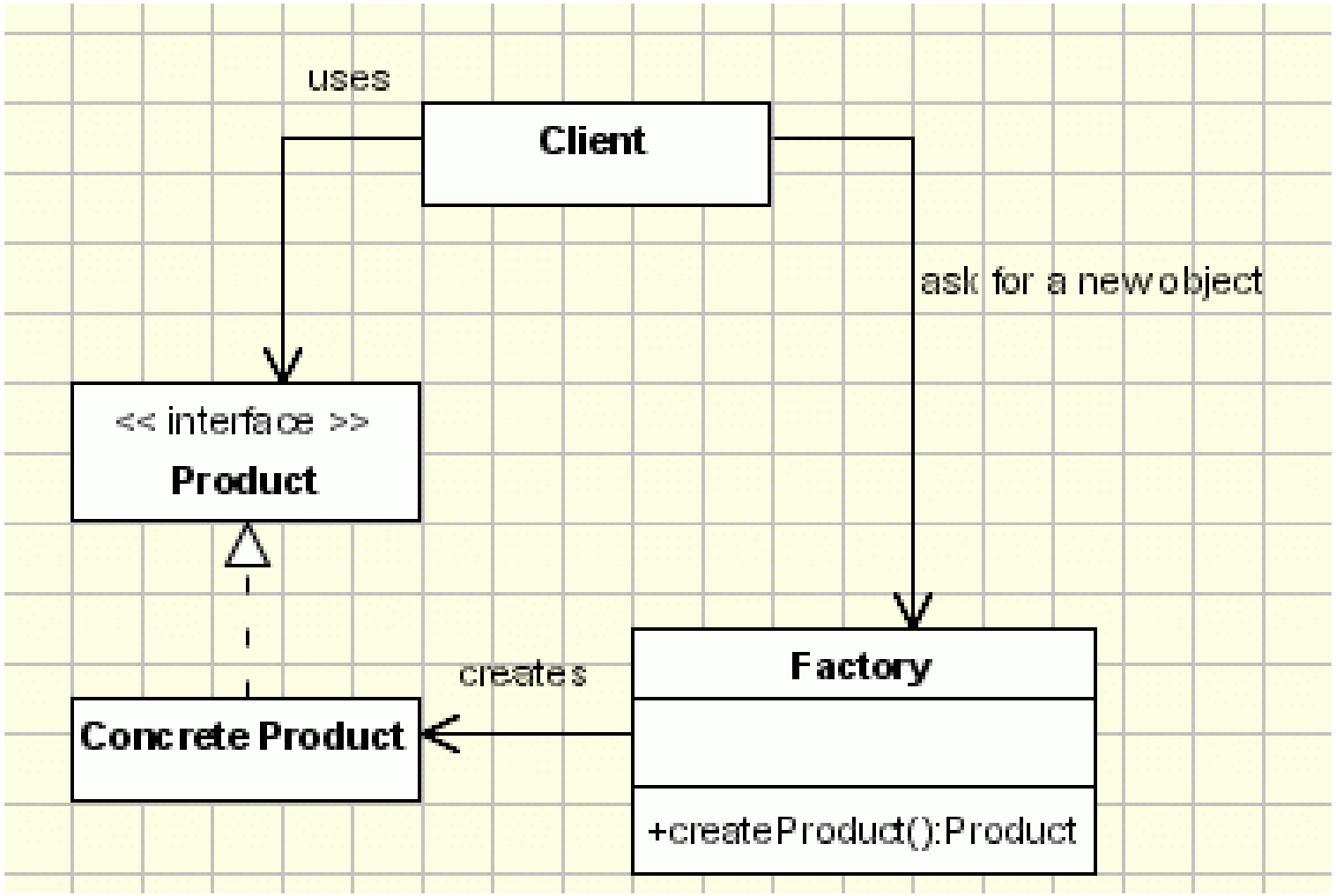

uses
Client
ask for a new object
<< interface >>
Product
creates
Factory
Concrete Product
+createProduct():Product

Diagram UML

Wady i zalety

+ Umożliwia ukrycie implementacji aplikacji (podstawowe interfejsy, które tworzą twoją aplikację)

+ Umożliwia łatwe testowanie połączenia aplikacji (czyli mockowania / zastępowania) niektórych części aplikacji, dzięki czemu można tworzyć i testować inne części

+ Umożliwia łatwiejszą zmianę projektu aplikacji

- Sprawia, że kod jest trudniejszy do odczytania, ponieważ cały kod kryje się za abstrakcją, która z kolei może ukrywać abstrakcje

Adapter

- Wzorzec adaptera dostosowuje się między klasami i obiektami. Jak każdy adapter w świecie rzeczywistym, służy on jako interfejs, pomost między dwoma obiektami. W prawdziwym świecie mamy adaptery do zasilaczy, adaptery do kart pamięci do aparatów i tak dalej.

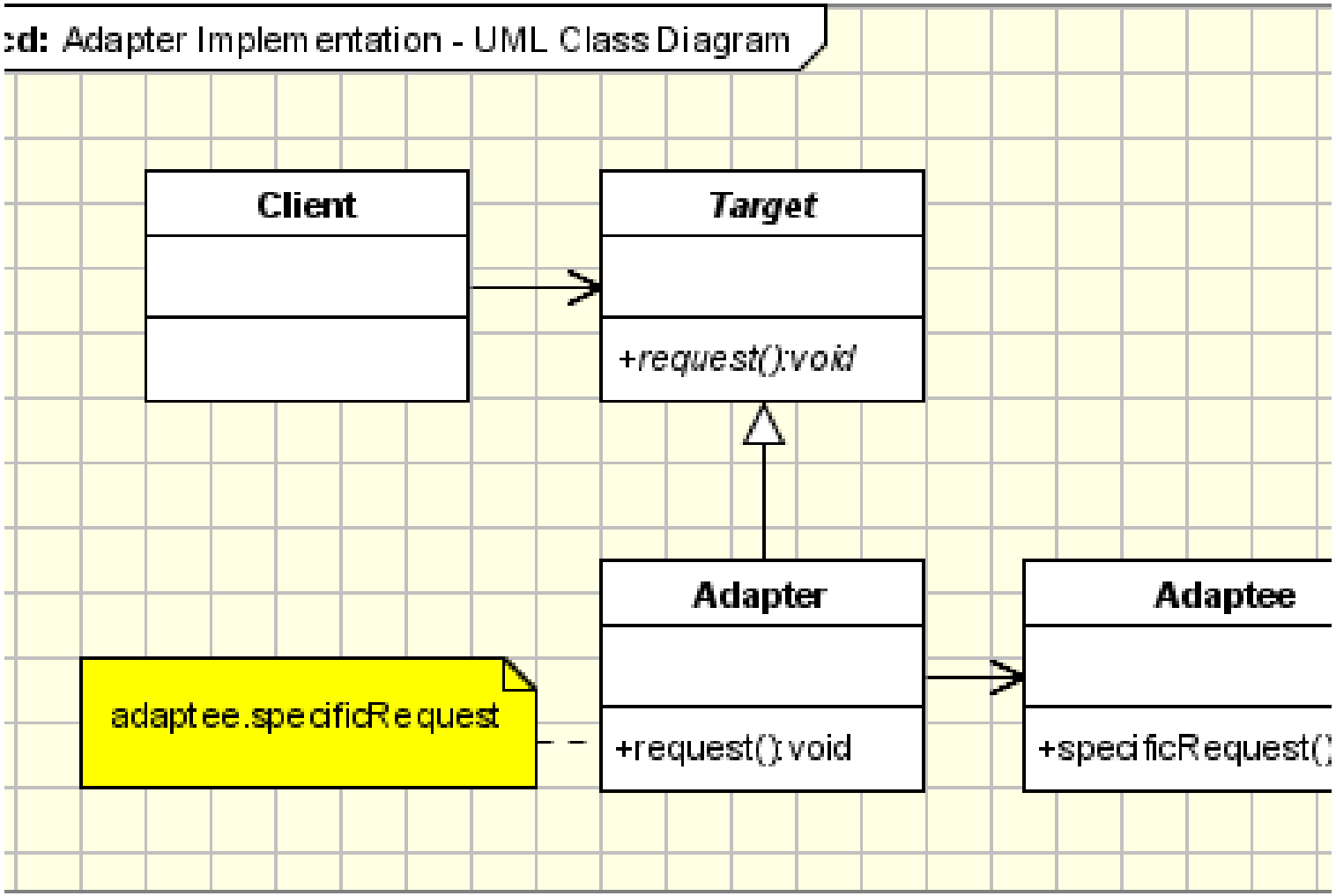

cd: Adapter Implementation - UML Class Diagram
Client
Target
+request():void
Adapter
+request():void
Adaptee
+specificRequest()
adaptee.specificRequest

Diagram UML

Wady i zalety

+ Zasada pojedynczej odpowiedzialności. Można oddzielić interfejs lub kod konwersji danych od podstawowej logiki biznesowej programu.

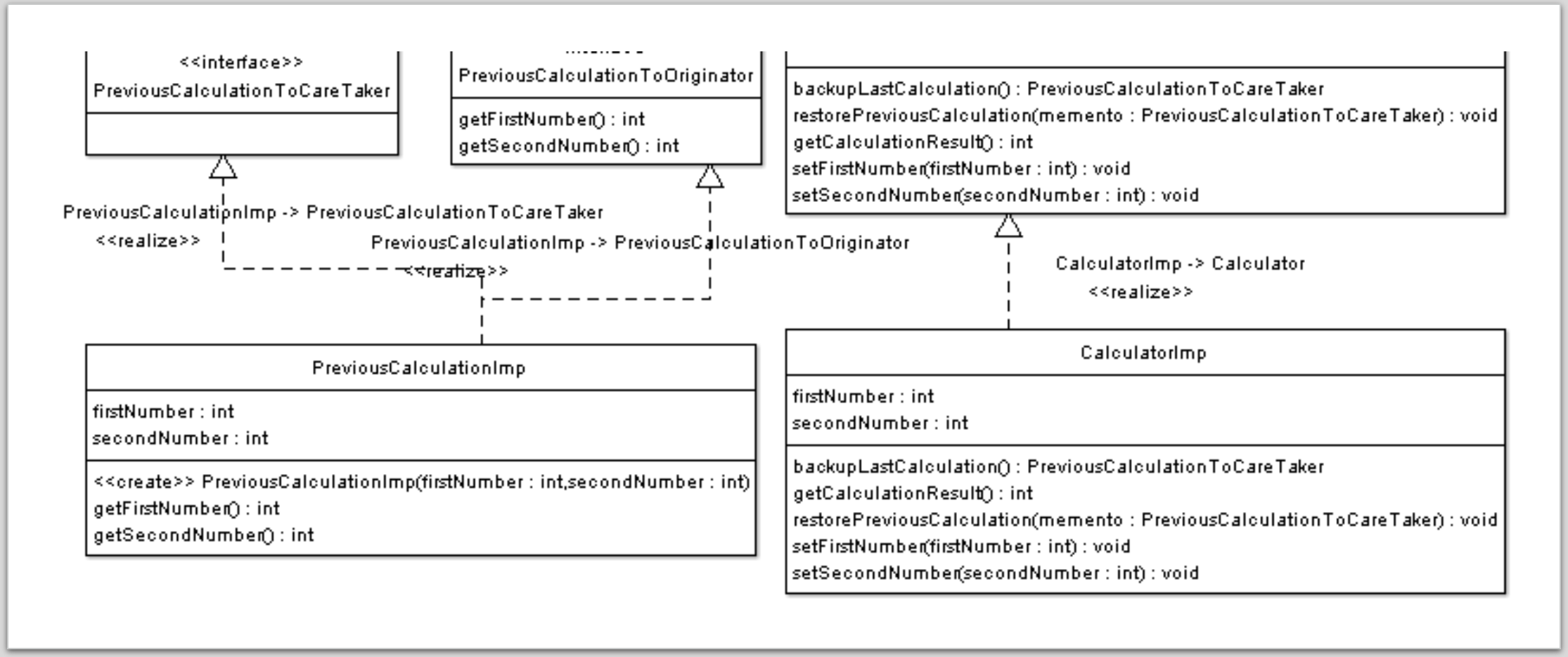
+ Zasada otwartego / zamkniętego. Do programu można wprowadzać nowe typy adapterów bez przerywania istniejącego kodu klienta, o ile współpracują one z adapterami za pośrednictwem interfejsu klienta.

- Ogólna złożoność kodu wzrasta, ponieważ trzeba wprowadzić zestaw nowych interfejsów i klas. Czasami prościej jest po prostu zmienić klasę usługi, tak aby pasowała do reszty kodu.

Memento

- Czasami w pewnym momencie konieczne jest uchwycenie stanu wewnętrznego obiektu i możliwość przywrócenia obiektu do tego stanu w późniejszym czasie. Taki przypadek jest przydatny w przypadku błędu lub awarii.

Diagram UML


<<interface>>
PreviousCalculationToCareTaker
PreviousCalculationToOriginator
getFirstNumber() : int
getSecondNumber() : int
backupLastCalculation() : PreviousCalculationToCareTaker
restorePreviousCalculation(memento : PreviousCalculationToCareTaker) : void
getCalculationResult() : int
setFirstNumber(firstNumber : int) : void
setSecondNumber(secondNumber : int) : void
PreviousCalculationImp -> PreviousCalculationToCareTaker
<<realize>>
PreviousCalculationImp -> PreviousCalculationToOriginator
<<realize>>
CalculatorImp -> Calculator
<<realize>>
PreviousCalculationImp
firstNumber : int
secondNumber : int
<<create>> PreviousCalculationImp(firstNumber : int,secondNumber : int)
getFirstNumber() : int
getSecondNumber() : int
CalculatorImp
firstNumber : int
secondNumber : int
backupLastCalculation() : PreviousCalculationToCareTaker
getCalculationResult() : int
restorePreviousCalculation(memento : PreviousCalculationToCareTaker) : void
setFirstNumber(firstNumber : int) : void
setSecondNumber(secondNumber : int) : void

Wady i zalety

+ Można tworzyć migawki stanu obiektu bez naruszania jego hermetyzacji.

+ Można uprościć kod nadawcy, pozwalając opiekunowi na zachowanie historii stanu nadawcy.

- Aplikacja może zużywać dużo pamięci RAM, jeśli klienci zbyt często tworzą pamiątki.

- Opiekunowie powinni śledzić cykl życia twórcy, aby móc niszczyć przestarzałe pamiątki.

- Większość dynamicznych języków programowania, takich jak PHP, Python i JavaScript, nie może zagwarantować, że stan w pamięci pozostanie nietknięty.

Mediator

- Aby mieć dobry projekt zorientowany obiektowo, musimy stworzyć wiele klas współdziałających ze sobą. Jeśli pewne zasady nie zostaną zastosowane, ostateczna struktura zakończy się całkowitym bałaganem, w którym każdy obiekt będzie zależał od wielu innych obiektów, aby działać. Aby uniknąć ściśle powiązanych struktur, potrzebujemy mechanizmu ułatwiającego interakcję między obiektami w taki sposób, aby obiekty nie były świadome istnienia innych obiektów.
- Weźmy przykład ekranu. Kiedy go tworzymy, dodajemy do ekranu wszelkiego rodzaju kontrolki. Ta kontrolka musi współdziałać ze wszystkimi innymi kontrolkami. Na przykład po naciśnięciu przycisku musi wiedzieć, czy dane są prawidłowe w innych kontrolkach. Jak widzieliśmy, jeśli utworzyłeś różne aplikacje przy użyciu formularzy, nie musisz modyfikować każdej klasy kontrolki za każdym razem, gdy dodajesz nową kontrolkę do formularza. Wszystkie operacje między kontrolkami są zarządzane przez samą klasę formularza. Ta klasa nazywa się mediatorem.

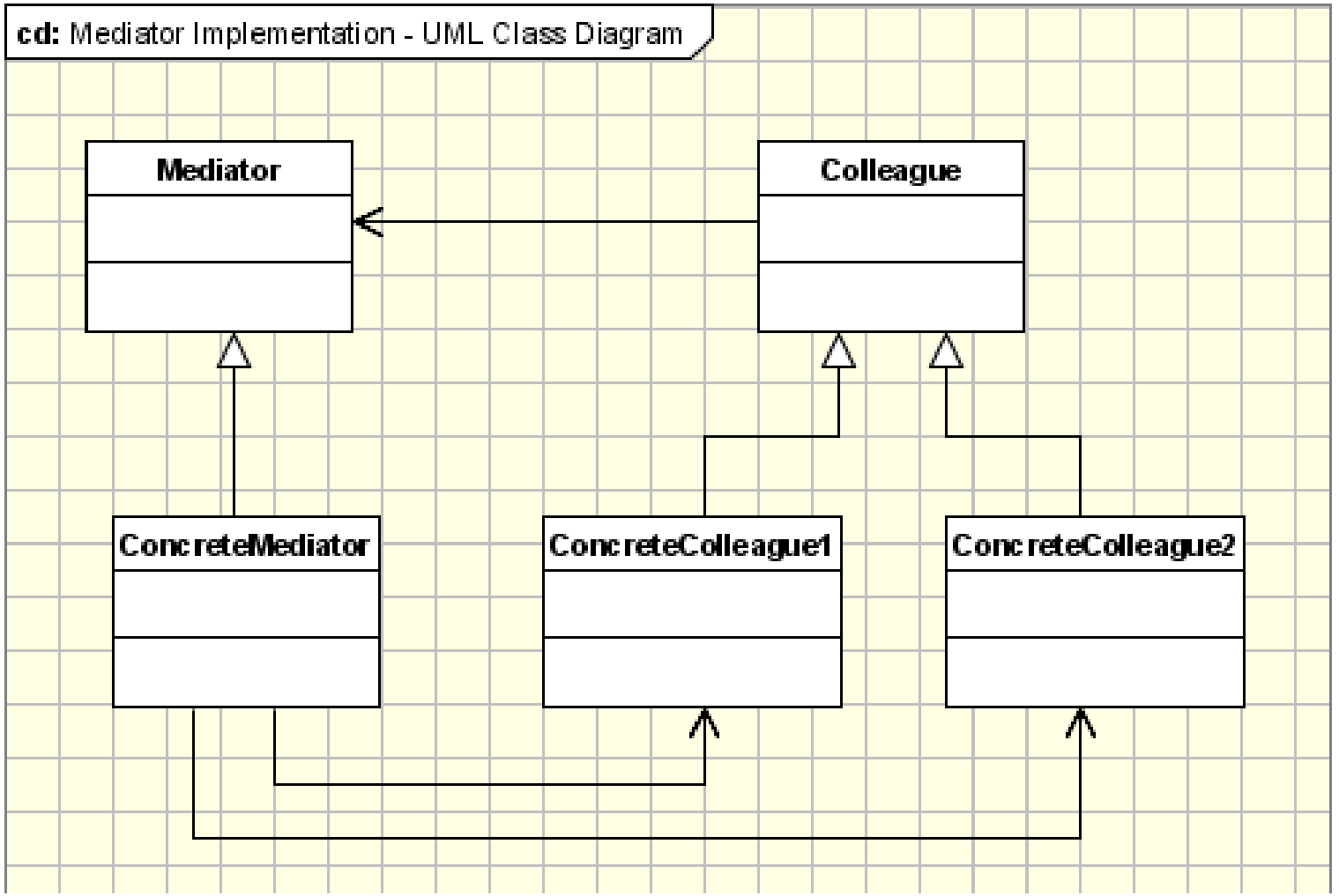

cd: Mediator Implementation - UML Class Diagram
Mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Diagram UML

Wady i zalety

+ Zasada pojedynczej odpowiedzialności. Można wyodrębnić komunikację między różnymi komponentami w jednym miejscu, ułatwiając zrozumienie i konserwację.

+ Zasada otwarcia / zamknięcia. Można wprowadzić nowych mediatorów bez konieczności zmiany rzeczywistych komponentów.

+ Można zmniejszyć powiązania między różnymi komponentami programu.

+ Można łatwiej ponownie wykorzystać poszczególne komponenty.

- Z biegiem czasu mediator może przekształcić się w obiekt nadrzędny.

Obserwator

- Nie możemy mówić o programowaniu zorientowanym obiektowo bez uwzględnienia stanu obiektów. W końcu programowanie obiektowe dotyczy obiektów i ich interakcji. Często zdarzają się przypadki, w których pewne obiekty wymagają poinformowania o zmianach, jakie zaszły w innych. Dobry projekt oznacza jak największe oddzielenie i zmniejszenie zależności. Wzorzec projektowy obserwatora może być stosowany zawsze, gdy obiekt ma być obserwowany przez jednego lub więcej obserwatorów.
- Załóżmy, że mamy system magazynowy, który dostarcza danych dla kilku typów klientów. Chcemy mieć klienta zaimplementowanego jako aplikacja internetowa, ale w niedalekiej przyszłości musimy dodać klientów na urządzenia mobilne, Palm lub Pocket PC lub mieć system powiadamiania użytkowników za pomocą alertów smsowych. Teraz łatwo jest zobaczyć, czego potrzebujemy ze wzorca obserwatorów: musimy oddzielić podmiot (serwer giełdowy) od jego obserwatorów (aplikacje klienckie) w taki sposób, aby dodanie nowego obserwatora było przezroczyste dla serwera.

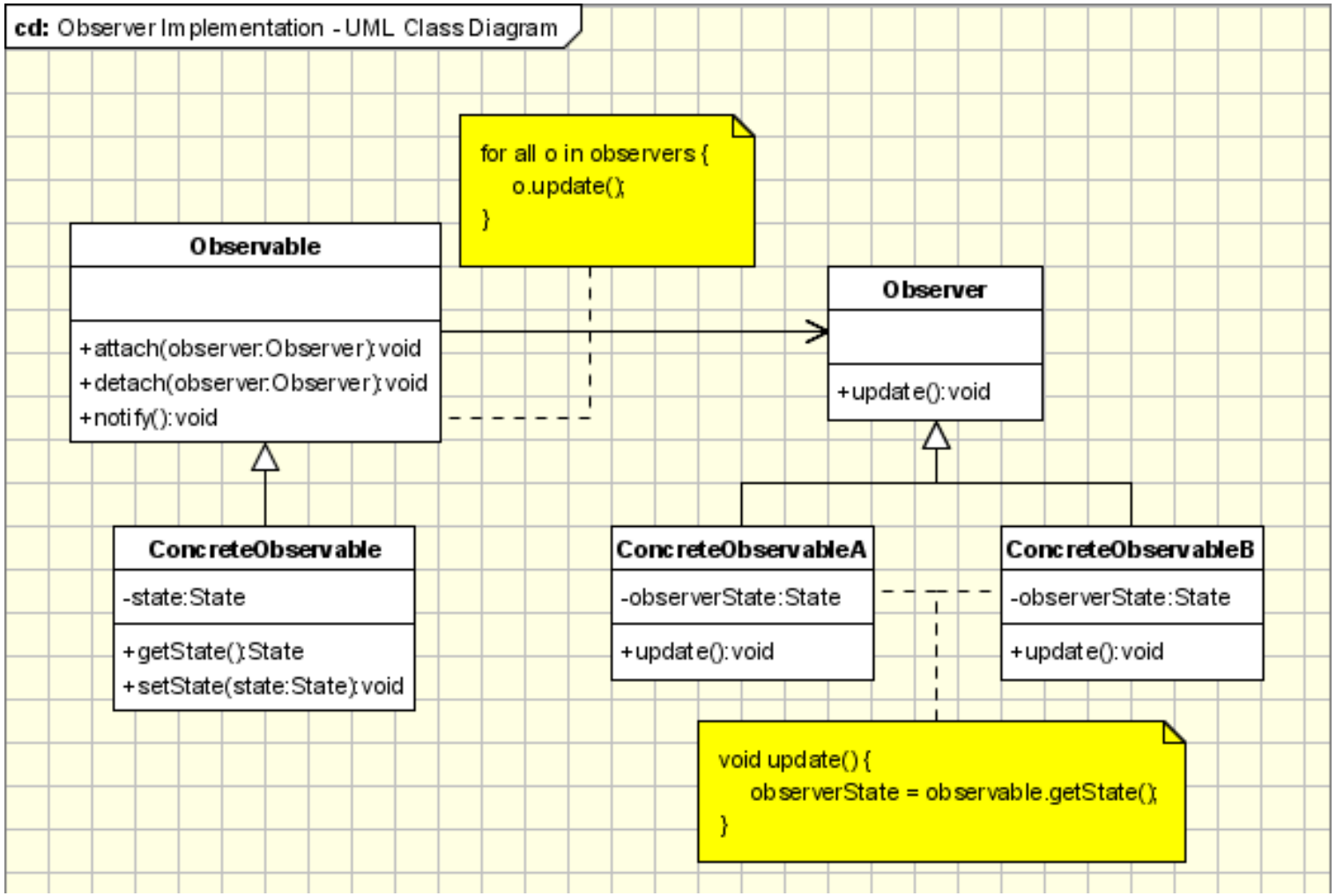

cd: Observer Implementation - UML Class Diagram
for all o in observers {
o.update();
}
Observable
+attach(observer:Observer):void
+detach(observer:Observer):void
+notify():void
Observer
+update():void
ConcreteObservable
-state:State
+getState():State
+setState(state:State):void
ConcreteObservableA
-observerState:State
+update():void
ConcreteObservableB
-observerState:State
+update():void
void update() {
observerState = observable.getState();
}

Diagram UML

Zady i walety

+ Zasada otwarcia / zamknięcia. Można wprowadzić nowe klasy subskrybentów bez konieczności zmiany kodu wydawcy (i odwrotnie, jeśli istnieje interfejs wydawcy).

+ Można ustanawiać relacje między obiektami w czasie wykonywania.

- Subskrybenci są powiadamiani w losowej kolejności.