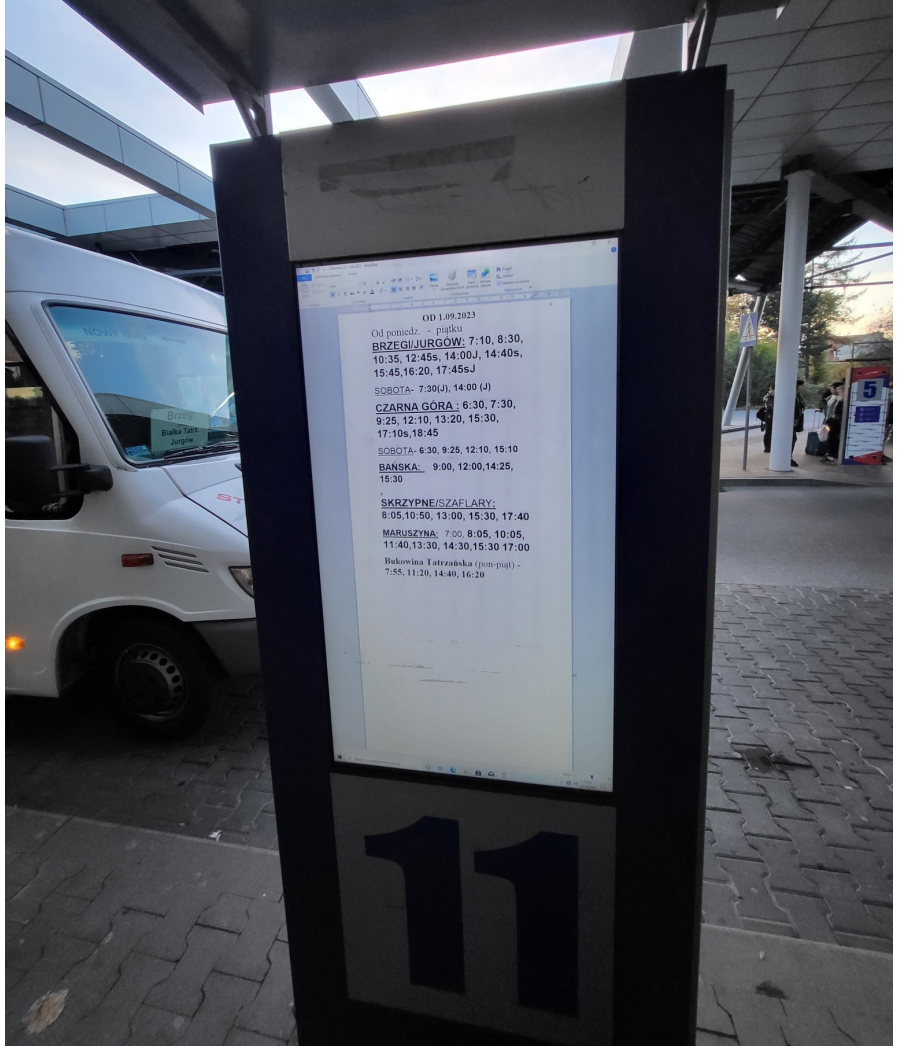
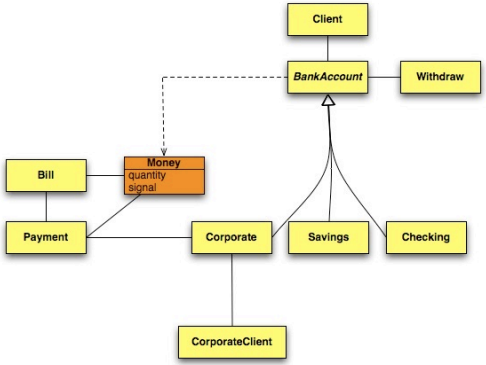
Interfejsy graficzne

Wzorce projektowe

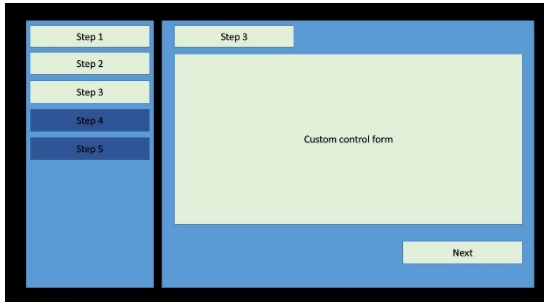

OD 1.09.2023
Od poniedz. - piątku
BRZEGI/JURGÓW: 7:10, 8:30, 10:35, 12:45s, 14:00J, 14:40s, 15:45,16:20, 17:45sJ
SOBOTA- 7:30(J), 14:00 (J)
CZARNA GÓRA : 6:30, 7:30, 9:25, 12:10, 13:20, 15:30, 17:10s,18:45
SOBOTA- 6:30, 9:25, 12:10, 15:10
BAŃSKA: 9:00, 12:00,14:25, 15:30
SKRZYPNE/SZAFLARY: 8:05,10:50, 13:00, 15:30, 17:40
MARUSZYNA: 7:00, 8:05, 10:05, 11:40,13:30, 14:30,15:30 17:00
Bukowina Tatrzańska (pon-piąt) - 7:55, 11:20, 14:40, 16:20
11

Synchronizacja danych w systemie

Repozytorium danych

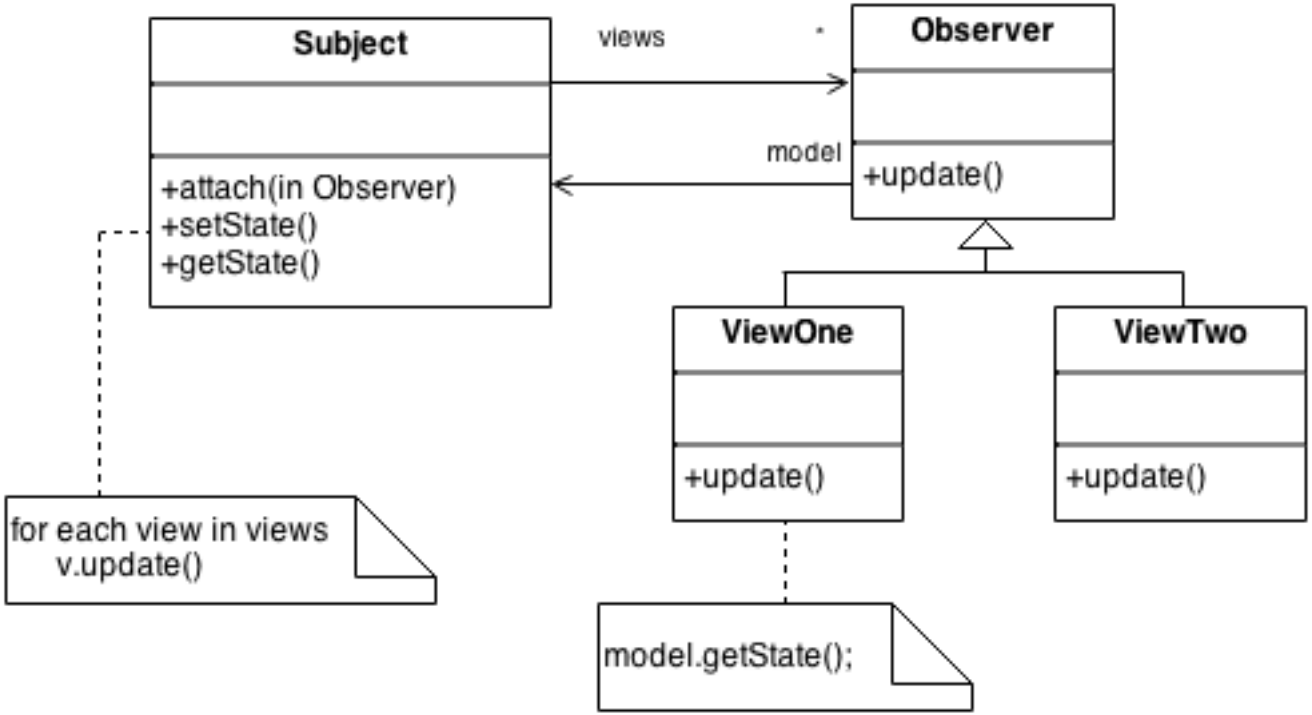


Model danych w pamięci systemu



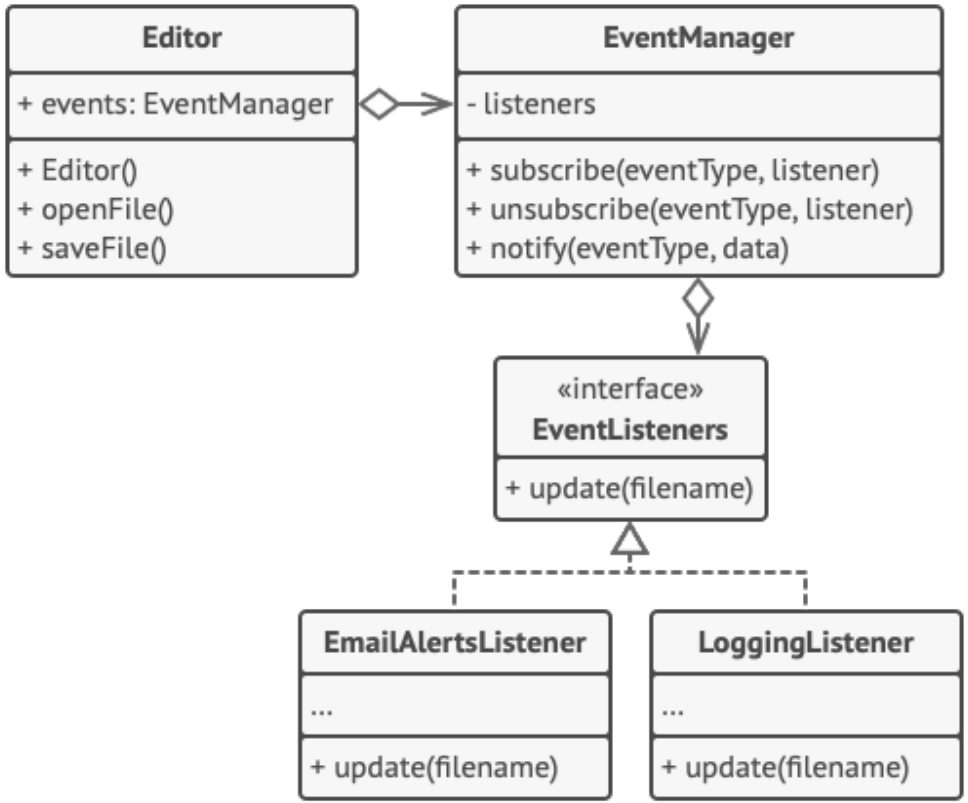
Model odpowiedzialny za wyświetlenie danych

Wzorzec Obserwator



Źródło: https://sourcemaking.com/design_patterns/observer

Obserwator - przykład



Źródło: https://refactoring.guru/pl/design-patterns/observer

Interaktywne GUI – reagowanie na zdarzenia

- Bezpośrednia reakcja na zdarzenia

```
Button red = new Button("Red");
red.setOnAction(event -> message.setTextFill(Color.RED));
```

  - Pełna kontrola nad działaniem użytkownika
  - Wiele możliwych zdarzeń – skomplikowana obsługa, redundancja kodu

Wiązanie właściwości

- Data bindings – wiązanie atrybutów obiektów

```
public class Greeting {
  private StringProperty text = new SimpleStringProperty("");
  public final StringProperty textProperty() { return text; }
  public final void setText(String newValue) { text.set(newValue); }
  public final String getText() { return text.get(); }
}
```

  - Enkapsulacja atrybutów obiektu w obiekt właściwości, odpowiedzialny m.in. za wysyłanie notyfikacji o zmianach wartości atrybuty

Wiązanie właściwości – rozszerzenie obserwatora

- Interfejs Property (JavaFX)

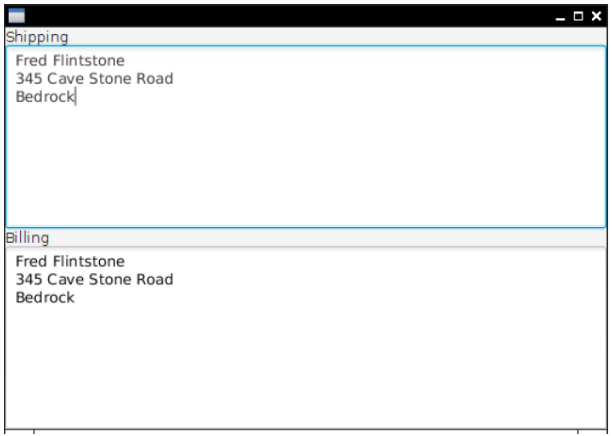
Modifier and Type	Method	Description
void	`bind(ObservableValue<? extends T> observable)`	Create a unidirection binding for this `Property`.
void	`bindBidirectional(Property<T> other)`	Create a bidirectional binding between this `Property` and another one.
boolean	`isBound()`	Can be used to check, if a `Property` is bound.
void	`unbind()`	Remove the unidirectional binding for this `Property`.
void	`unbindBidirectional(Property<T> other)`	Remove a bidirectional binding between this `Property` and another one.

Interaktywne GUI

- Wiązanie właściwości kontrolek z elementami modelu…

```
textInputControl.text().bindBidirectional(greetings.textProperty());
```

- … lub innymi komponentami graficznymi




```
billing.textProperty()
  .bind(shipping.textProperty());
```

Model-View-Controller

Separated Presentation → Rozdzielenie pomiędzy obiektami dziedzinowymi oraz obiektami związanymi z interfejsem graficznym

- Obiekty dziedzinowe nie powinny być świadome ich „reprezentacji” graficznej
- Możliwość stworzenia różnych interfejsów dla tego samego modelu
- Możliwość testowania zachowania obiektów dziedzinowych

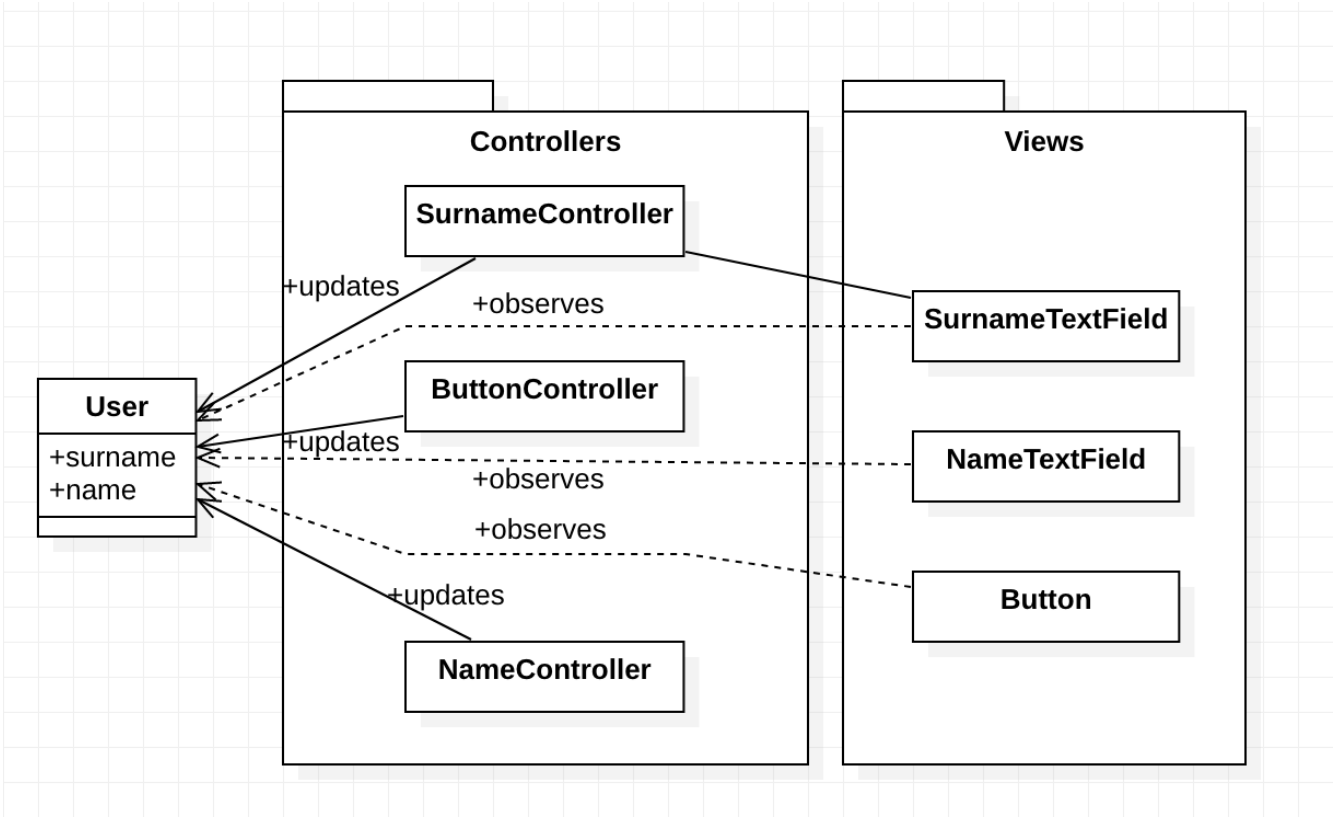
Model-View-Controller - początki

- Najczęściej „cytowany” wzorzec, jednocześnie często „interpretowany” na różne sposoby
- Oryginalna wersja wzorca powstała w latach 70-80 XX wieku (SmallTalk)
- Pierwsze próby budowy interfejsu graficznego

Observer synchronization

- idea zapoczątkowana przez wzorzec MVC
- widoki i kontrolery obserwują model. Kiedy model się zmienia, odpowiednie widoki aktualizują informacje wyświetlane na ekranie. Widoki pobierają odpowiednie informacje z modelu (obserwując poszczególne elementy modelu).
- kontroler nie aktualizuje zatem widoku, widok reaguje na zmiany w modelu.

Classic MVC


Controllers
Views
SurnameController
+updates
+observes
SurnameTextField
ButtonController
User
+surname
+name
+updates
NameTextField
+observes
+observes
Button
+updates
NameController

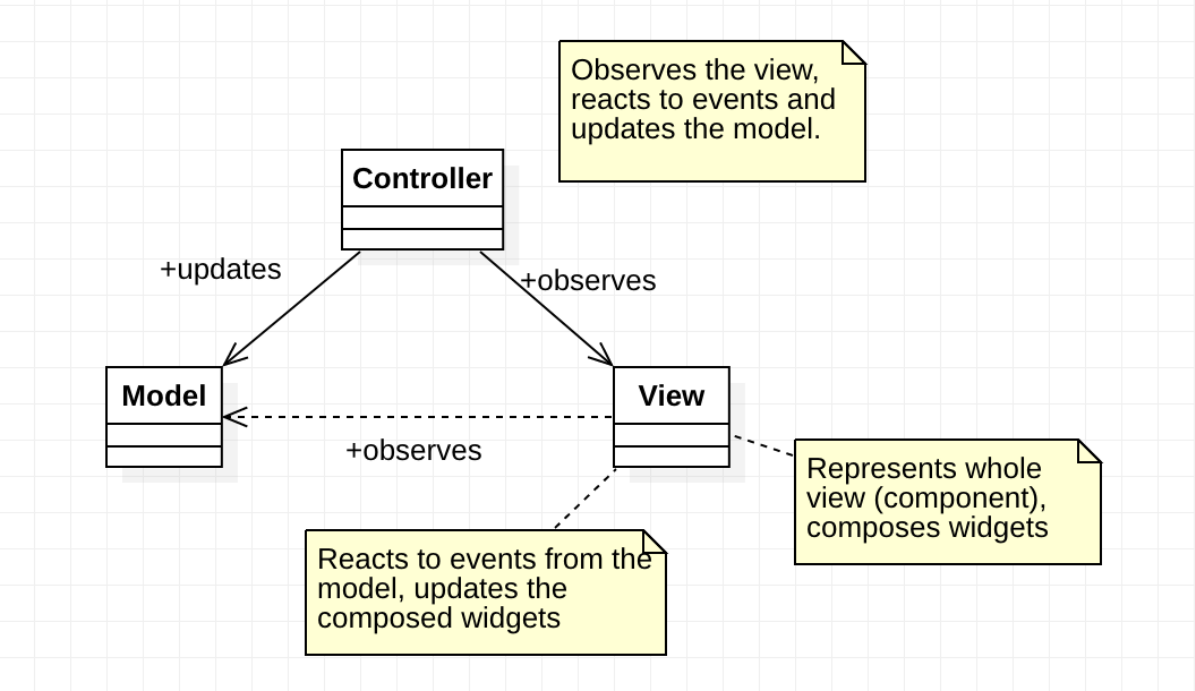
Classic MVC

- Model – obiekty dziedzinowe
- Controller – odczytuje zdarzenia wykonywane przez użytkownika na skojarzonym elemencie interfejsu (kontrolce)
- View – reprezentuje kontrolkę wyświetlaną na ekranie

Para view-controller tworzona jest dla każdego elementu wyświetlanego na ekranie.

Kontrolery współpracują ze sobą (poprzez reagowanie na zdarzenia), aby odpowiednio odczytać zachowanie użytkownika.

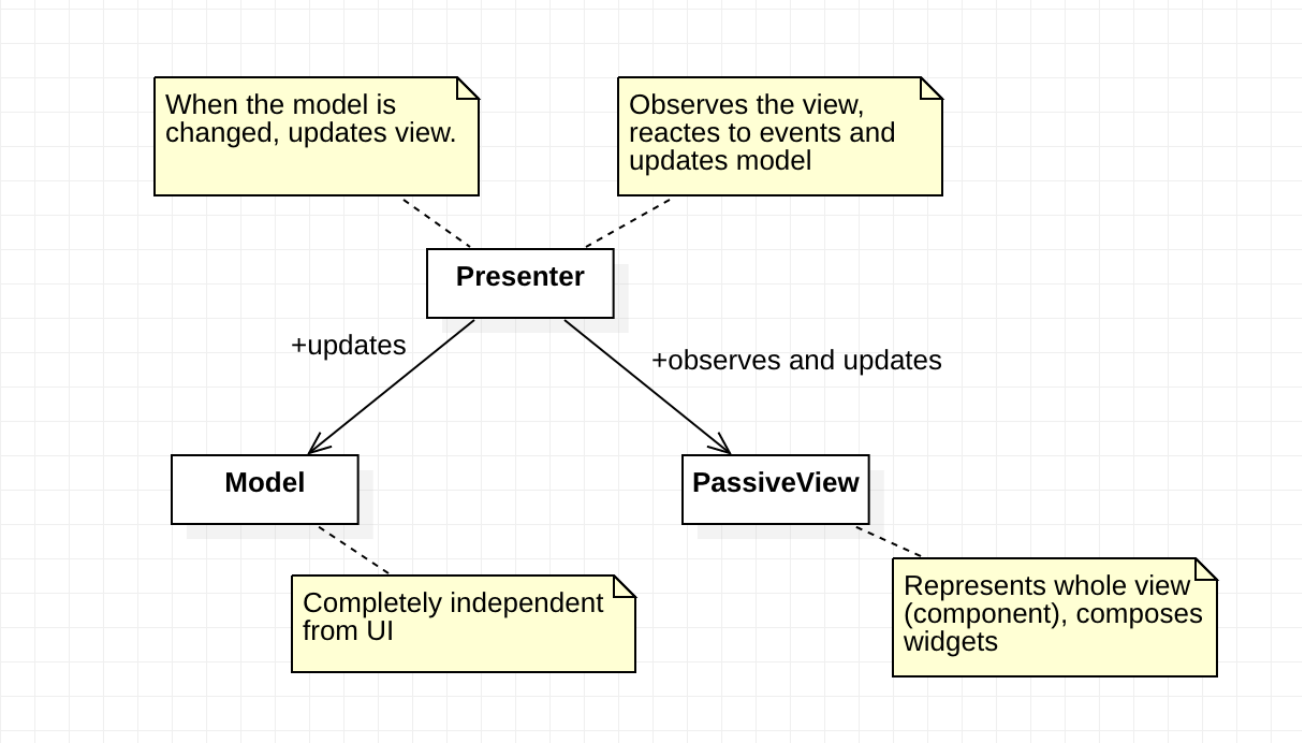
Model-View-Controller (aktualnie)



Model-View-Controller

- **Model** - obiekty dziedzinowe
- **View -** wyświetla rezultaty działania systemu oraz dostarcza interakcje z użytkownikiem, stanowi cały okno/komponent graficzny (komponuje kontrolki)
- **Controller** pośredniczy pomiędzy widokiem i modelem, interpretuje akcje generowane przez widok i powoduje odpowiednie modyfikacje modelu; jeden dla widoku (okna/komponentu)

Model-View-Presenter (PV)



Model-View-Presenter

- Model – obiekty dziedzinowe (a'la MVC)
- View – struktura kontrolek przedstawiająca np. ekran aplikacji, stronę kreatora; brak separacji na widok-kontroler poszczególnych kontrolek
- Presenter – reaguje na zdarzenia otrzymywane do elementów widoku, wykonuje akcje na modelu (np. przy użyciu komend)

MVP Passive View

- Najbardziej popularna wersja MVP
- Łatwość testowania modelu dziedzinowego
  - Kompletna niezależność modelu do widoku
  - Brak mechanizmów odpowiedzialnych za automatyczną synchronizację modelu i widoku (takich jak właściwości JavaFX)

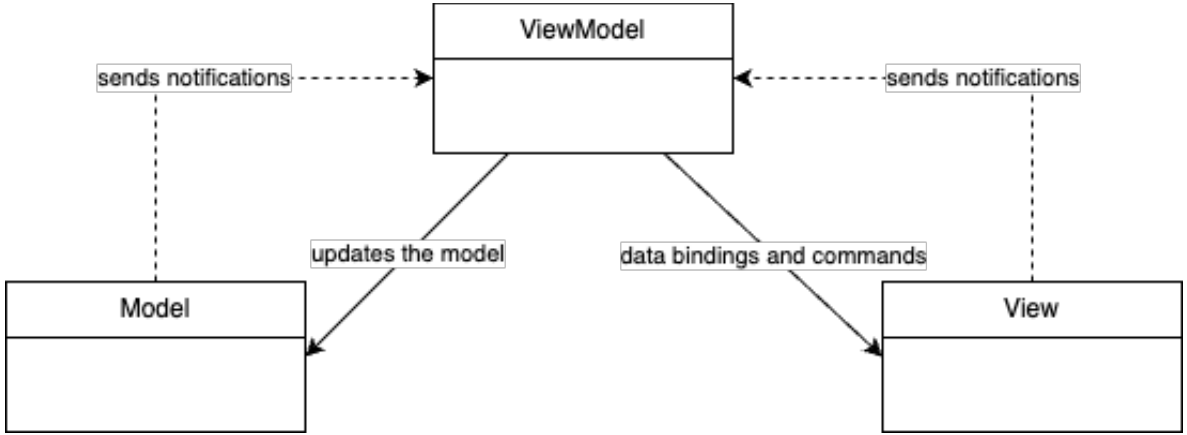
Zasada działania MVP

- Interakcje użytkownika przekazywane są z kontrolek do presentera.
- Presenter interpretuje otrzymane zdarzenia i zleca wykonanie odpowiednich zmian w modelu dziedzinowym.
- W zależności od wariantu MVP, widoki aktualizowane są z wykorzystaniem modelu Observer’a lub bezpośrednio przez presenter, który często wykonuje dodatkowe operacje.

Wariacje MVP

- Presenter nie manipuluje widokiem; widok renderowany jest tylko i wyłącznie na podstawie stanu modelu (M. Potel)
- Presenter manipuluje elementami widoku w sytuacji, gdy nie da się tego zrobić za pomocą (deklaratywnego) widoku (Bower & McGlashan, Supervising Controller)
- Presenter wykonuje wszystkie zmiany w widoku (Passive View) – nieujęty w oryginalnej definicji MVP

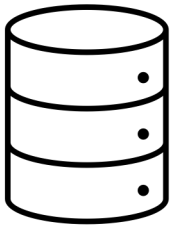
Model-View-ViewModel

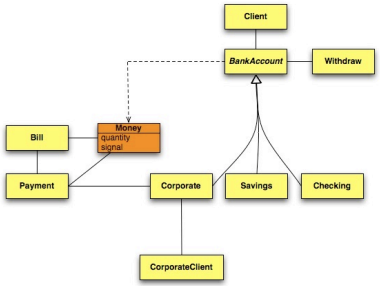


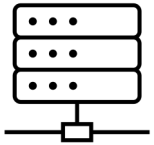
Model-View-ViewModel

- Model – obiekty dziedzinowe, zawierają logikę biznesową (a'la MVC, MVP)
- View – komponenty interfejsu graficznego przedstawiające np. ekran aplikacji, stronę kreatora;
- ViewModel – model danych dedykowany dla danego widoku (model UI) wraz komponentami obsługującymi zdarzenia i aktualizującymi model dziedzinowy (np. za pomocą usług)

Synchronizacja danych w systemie







Repozytorium danych

Model danych w pamięci systemu

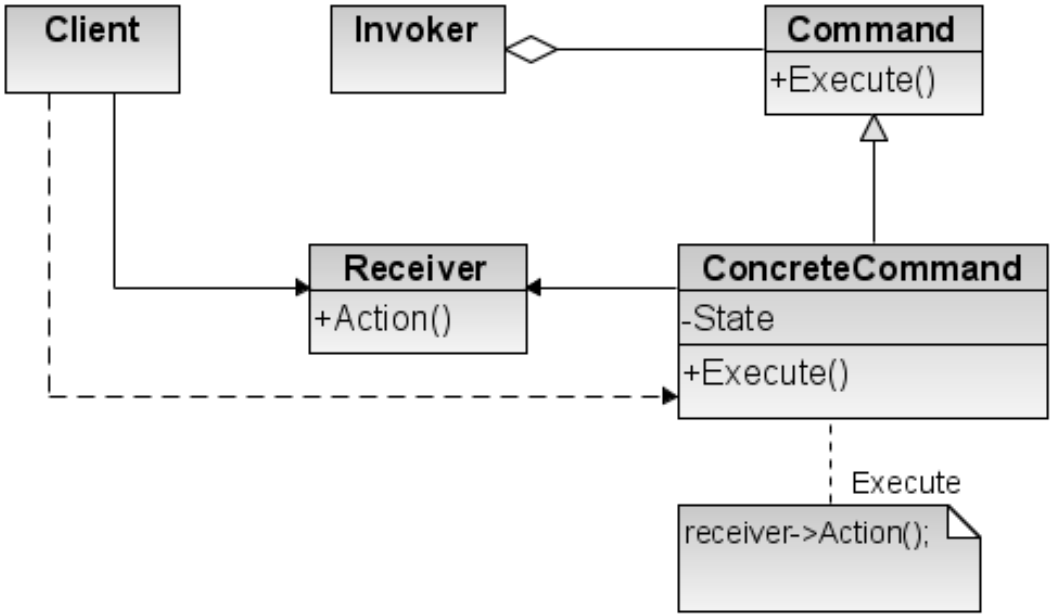
Usługa REST

Model odpowiedzialny za wyświetlenie danych

Polecenie (Command)

- Enkapsulacja operacji (polecenia) do osobnego obiektu
- Spójny interfejs dla wszystkich poleceń
- Możliwość odseparowania czasu przygotowania (konfiguracji) polecenia od czasu jego wykonania

Struktura wzorca Polecenie



Polecenie (Command)

- Enkapsulacja operacji (polecenia) do osobnego obiektu
- Wykonanie wszystkich operacji przez jeden obiekt
- Możliwości:
  - cofanie (i powtarzanie) wykonanych akcji,
  - logowanie wszystkich akcji,
  - kolejkowanie i zarządzanie wykonaniem akcji.

Źródła

- Martin Fowler, GUI Architectures, http://martinfowler.com/eaaDev/uiArchs.html
- Mike Potel, MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java, http://www.wildcrest.com/Potel/Portfolio/mvp.pdf
- Andy Bower, Blair McGlashan, Twisting the triad, The evolution of the Dolphin Smalltalk MVP application framework, http://www.object-arts.com/downloads/papers/TwistingTheTriad.PDF
- Command Design Pattern, https://sourcemaking.com/design_patterns/command