Wzorce projektowe

Cel zajęć

Celem zajęć jest ugruntowanie wiedzy o wybranych wzorcach projektowych i pokazanie ich użycia w rzeczywistym kodzie – standardowej bibliotece Javy oraz implementacja jednego wybranego wzorca. W szczególności chcielibyśmy abyś po wykonaniu tych ćwiczeń zapamiętał(a) że:

1. **Wzorce projektowe są obecne w rzeczywistym, „dużym" kodzie jak i w prostym API**
2. **Nazwy klas w użyciu nie muszą i najczęście nie odpowiadają nazwom klas z ideowego obrazu wzorca** – ważna jest struktura / dynamika, która stanowi istotę danego wzorca (chociaż oczywiście użycie nazewnictwa ze wzorca projektowego może ułatwić komunikację).

Z racji koncepcyjnego charakteru zajęć można je wykonywać **w parach**.

Zadanie 1

Spróbujmy sprawdzić jak implementacja wzorca projektowego może wyglądać w rzeczywistym kodzie. W tym przypadku zajrzymy do biblioteki standardowej języka Java. Implementacja pierwszego wzorca, którego będziemy poszukiwać jest prawie książkowa. Zacznijmy od pobrania projekty **Patterns** z Moodle'a, jego rozpakowania i otwarcia w środowisku IntelliJ. W pliku pom.xml jest ustawiona wersja Java 11, w przypadku gdy używasz starszej wersji ustaw odpowiednio properties maven.compiler.source i maven.compiler.target.

Twoje poszukiwania powinny rozpocząć się od pakietu **put.io.patterns.searchfor.streams**. Jest tam klasa **StreamsRunner**, która zawiera kawałek wykonywalnego kodu. W uproszczeniu w metodzie **main** zostaje uruchomiony kod z metody **run**. Natomiast, aby uruchomić program potrzebujesz wywołać go z parametrem będącym ścieżką do jakiegoś pliku.

Możesz podejrzeć i wyedytować konfigurację uruchomieniową: Run→Edit Configurations. W polu **Program arguments** wpisz ścieżkę do jakiegoś pliku - np. do pom.xml (uwaga: może być już ustawiona domyślna). Po uruchomieniu na ekranie pojawią się informacje na temat tego pliku.

Jeśli przyjrzysz się dobrze metodzie run zobaczysz odwołania do trzech klas: **InputStream**, **FileInputStream, CheckedInputStream**. Twoje zadanie polega na:

- ***przejrzeniu hierarchii tych klas*** - w IntelliJ możesz przeglądać hierarchię klas na wiele sposób, dwie szybkie metody: 1) ctrl+klik (cmd+click) na nazwie klasy w edytorze lub wywołaniu jej metody otworzy nową zakładkę edytora z jej źródłem; 2) użycie widoku hierarchii - ustaw kursor na nazwie klasy w edytorze i wybierz z menu Navigate→Type Hierarchy - przy użyciu tego widoku łatwo dotrzesz do nadklas i podklas klasy, która Cię interesuje.
- ***uzupełnienie otrzymanego diagramu klas*** - swoje obserwacje nanieś na dostarczony diagram klas do uzupełnienia (dostępny na Moodle'u). Może nanieść je w formie adnotacji do PDFa jeśli posiadasz odpowiednie narzędzie lub opisać w pliku tekstowym co powinno zostać zapisane w polach oznaczonych kolejnymi liczbami.
- ***ustalenie jaki wzorzec projektowy tworzą*** - pamiętaj, że każdy wzorzec ma określony cel i od poszukiwania tego celu rozpocznij. Jeśli skupisz się tylko na strukturze masz dużą szansę na popełnieniu błędu. Zatem zastanów się co autor kodu chciał osiągnąć tworząc daną strukturę w kodzie obiektowym?
- ***określenie, która klasa pełni jaką rolę zdefiniowaną we wzorcu projektowym*** - użyj nazw z definicji wzorca (np. ze slajdów wykładowych) dla poszczególnych klas w jego implementacji, którą analizujesz. Jeśli nie jesteś tego w stanie zrobić najprawdopodobniej nie jest to ten wzorzec.

Zadanie 2

Na chwilę porzućmy zabawę w Sherlocka Holmesa i spróbujmy zaimplementować jakiś wzorzec. Zajrzyjmy zatem do **pakietu put.io.patterns.implement**.

Najbardziej istotną dla nas klasą w pakiecie jest w tej chwili klasa **SystemMonitor**. Potrafi ona badać stan zasobów systemowych (pamięci, CPU oraz urządzenia USB).

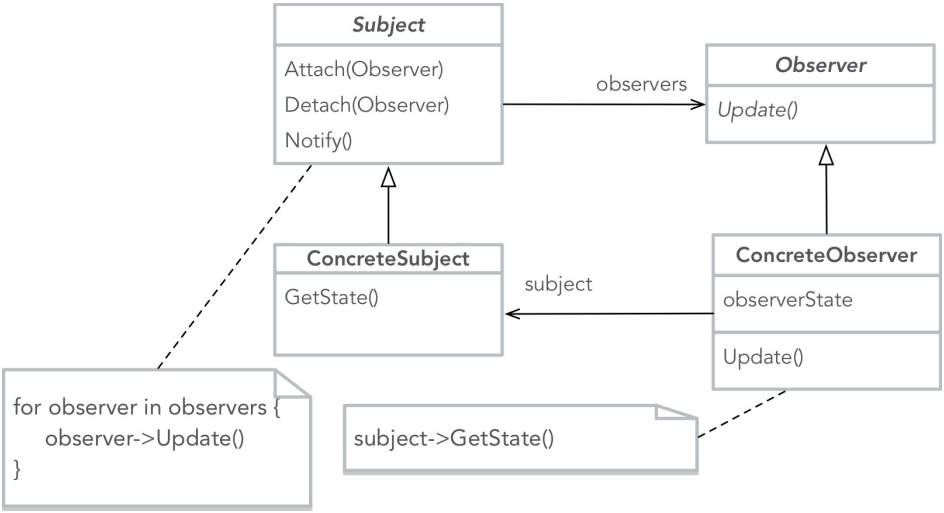
W pakiecie znajduje się też klasa **MonitorRunner**. Klasa zawiera metodę **main**, którą będziemy uruchamiać nasz monitor systemu (w tej chwili w pętli nieskończonej będzie badać stan systemu z interwałem 5000 ms).

W tej chwili w metodzie **probe** klasy **SystemMonitor** dzieją się trzy rzeczy:

1. Zostaje odczytany aktualny stan systemu.
2. Zostaje wyświetlona informacja o stanie systemu.
3. Zostają aktywowane pewne reguły w zależności od spełnienia pewnych warunków (oczywiście rzeczywiste akcje udajemy wypisując na ekran informacje, że miały miejsce).

Nie wiem czy już to zauważyłeś(aś), ale klasa **SystemMonitor** wydaje się być przeciążona odpowiedzialnościami: monitoruje, raportuje, przeciwdziała… Zdecydowanie nie jest to dobrze napisany kod obiektowy. Pomyśl tylko jak przetestować te poszczególne zachowania? Spróbujmy temu zaradzić. Jak myślisz jaki wzorzec by się tu przydał?

Jednym z takich wzorców, który pomógłby rozwiązać nasze problemy byłby Obserwator (znany też jako Listener). Poniżej znajduje się ideowy diagram klas dla tego wzorca (na wykładzie mówiliśmy o nim nieco więcej). Analizując diagram klas zastanów się jak można rozbić klasę **SystemMonitor**, aby zostawić jej tylko jedną odpowiedzialność = **odczytywanie informacji o systemie i udostępnianie tej informacji innym** (w końcu od tego pochodzi jej nazwa).



Zapewne widzisz już, że **SystemMonitor** mógłby pełnić rolę obiektu obserwowanego, czyli *Subject.* Umożliwiłby on zapisywanie się *obserwatorom* na powiadomienia o zmianie stanu systemu. *Obserwatory* byłyby powiadamiane po każdym odczycie stanu systemu i do każdego z nich należałaby odpowiednia reakcja na otrzymane informacje.

Jeśli czujesz się na siłach i masz ochotę na wyzwanie możesz spróbować zaimplementować wzorzec bez poniższych wskazówek.

Zacznijmy zatem od zdefiniowania wspólnego interfejsu dla wszystkich obserwatorów. Nazwijmy go **SystemStateObserver**. Obserwator będzie miał jedną metodę **update**. Każdy obserwator implementujący ten interfejs właśnie w tej metodzie będzie wykonywał operacje w odpowiedzi na zmianę stanu systemu:

```
public interface SystemStateObserver{
  public void update(SystemMonitor monitor);
}
```

Implementując metodę **update** możemy przekazać instancję klasy **SystemMonitor** jako parametr wywołania, a wtedy konkretny obserwator musiałby sobie ten stan z niej pobrać:

```
public void update(SystemMonitor monitor){
  monitor.getLastSystemState()
  ...
}
```

Moglibyśmy odejść też trochę od książkowego wzorca i przekazać w tej metodzie “uchwyt” bezpośrednio do aktualnego stanu:

```
public update(SystemState newState);
```

Mamy już dobry punkt zaczepienia. Teraz przygotujmy naszą klasę obserwowaną (Subject), czyli **SystemMonitor** na możliwość przyjęcia obserwatorów. Dokonajmy modyfikacji klasy **SystemMonitor** (oczywiście zaimplementuj obie metody):

```
public class SystemMonitor{
  private List<SystemStateObserver> observers = new ArrayList<SystemStateObserver>();
  ...
  public void addSystemStateObserver(SystemStateObserver observer){
    ...
  }
  public void removeSystemStateObserver(SystemStateObserver observer){
  ...
  }
...
}
```

Czas zaimplementować pierwszego obserwatora. Będzie on oczywiście osobną klasą implementującą interfejs **SystemStateObserver**:

- **SystemInfoObserver** - wypisze raport na konsole o stanie zasobów (implementacje przenieśmy bezpośrednio z **SystemMonitor→probe**)

Musimy jeszcze dodać metodę informującą wszystkie obserwatory o tym, że stan się zmienił. Będzie to metoda w klasie naszego “Subject” czyli **SystemMonitor**:

```
public void notifyObservers(){
  for(SystemStateObserver observer : this.observers){
    observer.update(this);
  }
}
```

I oczywiście w metodzie **probe** zaraz po zaczytaniu parametrów systemu należy tę metodę wywołać:

```
public void probe(){
  ...
  lastSystemState = new SystemState(cpuLoad, cpuTemp, memory, usbDevices);
  notifyObservers();
}
```

Mamy już zatem po jednym z klocków układanki. Spróbujmy uruchomić to jako całość. Przejdźmy do klasy **MonitorRunner** i zmieńmy metodę **main** dodając kod tworzący instancję nowego obserwatora i zapisujący ją na zmiany w monitorze :

```
public static void main(String args[]){
  // tworzymy monitor
  SystemMonitor monitor = new SystemMonitor();

  // tworzymy obserwatora i dodajemy do monitora
  SystemStateObserver infObserver =  new SystemInfoObserver();
  monitor.addSystemStateObserver(infObserver);

  while (true) {
      monitor.probe();
      try {
          Thread.sleep(5000);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
  }
}
```

Czas zaimplementować dodatkowe obserwatory - ponownie każdy będzie osobną klasą implementującą interfejs **SystemStateObserver** wykorzystując istniejący kod przeniesiony z metody **probe** w **SystemMonitor** (jeśli odpowiedni kod się tam znajduje):

- **SystemGarbageCollectorObserver** - uruchomi garbage collector jak jest mało pamięci.
- **SystemCoolerObserver** - uruchomi chłodzenie jak procesor się przegrzewa
- **USBDeviceObserver** - to nowość spróbujmy zrobić obserwatora, który wypisze komunikat tylko jeśli zwiększyła się lub zmniejszyła liczba urządzeń USB (obserwator musi pamiętać ostatnio otrzymane stane od SystemMonitorów, które obserwuje).

Brawo! Udało się zaimplementować pierwszy wzorzec. Dzięki zastosowaniu wzorca obserwator odchudziliśmy klasę SystemMonitor z odpowiedzialności. Otwarliśmy też drogę do dodawania nowej funkcjonalności, która będzie związana z reakcją na zmiany stanu zasobów systemu. Zauważ także, że o

wiele łatwiej byłoby nam teraz przetestować czy zachowania w odpowiedzi na zmianę stanu systemu są poprawne ponieważ możemy testować każdy z obserwatorów w oderwaniu od SystemMonitora.

Zadanie 3

Wróćmy do szukania wzorców w kodzie Javy. Tym razem przejdźmy do pakietu **put.io.patterns.searchfor.sierpinski**. Program znajduje się w klasi **SierpinskiRunner**.

Konfiguracja SierpinskiRunner ma domyślnie ustawioną głębokość 3. Aby zmienić głębokość należy zmienić parametr wywołania: Run→Edit Configurations i w polu **Program arguments** wybrać wartość w rozsądnych granicach (2-4).

Oczywiście nie wyświetlanie dywanu Sierpińskiego jest dla nas istotne w tym miejscu. Podobnie jak przy zadaniu 1 dokonaj eksploracji hierarchii klas. Przejrzyj też implementacje wskazanych metod. Tym razem poszukujemy dwóch wzorców. Implementacja wzorca nr 2 jest dość książkowa, natomiast wzorzec 1 ma już widoczne różnice, które w tym wypadku do pewnego stopnia wynikają z uwarunkowań historycznych rozwoju Javy (wzorzec przenika przez dwie “biblioteki” - starszą i “niżej” poziomową AWT i późniejszą “wyżej” poziomową zwaną Swing). Pamiętaj żeby określać rolę w zależności od wzorca (w dostarczonym PDFie z diagramami miejsce wystąpienia dwóch wzorców jest jawnie oddzielone).