

3. Шаблон Наблюдатель (Observer)

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

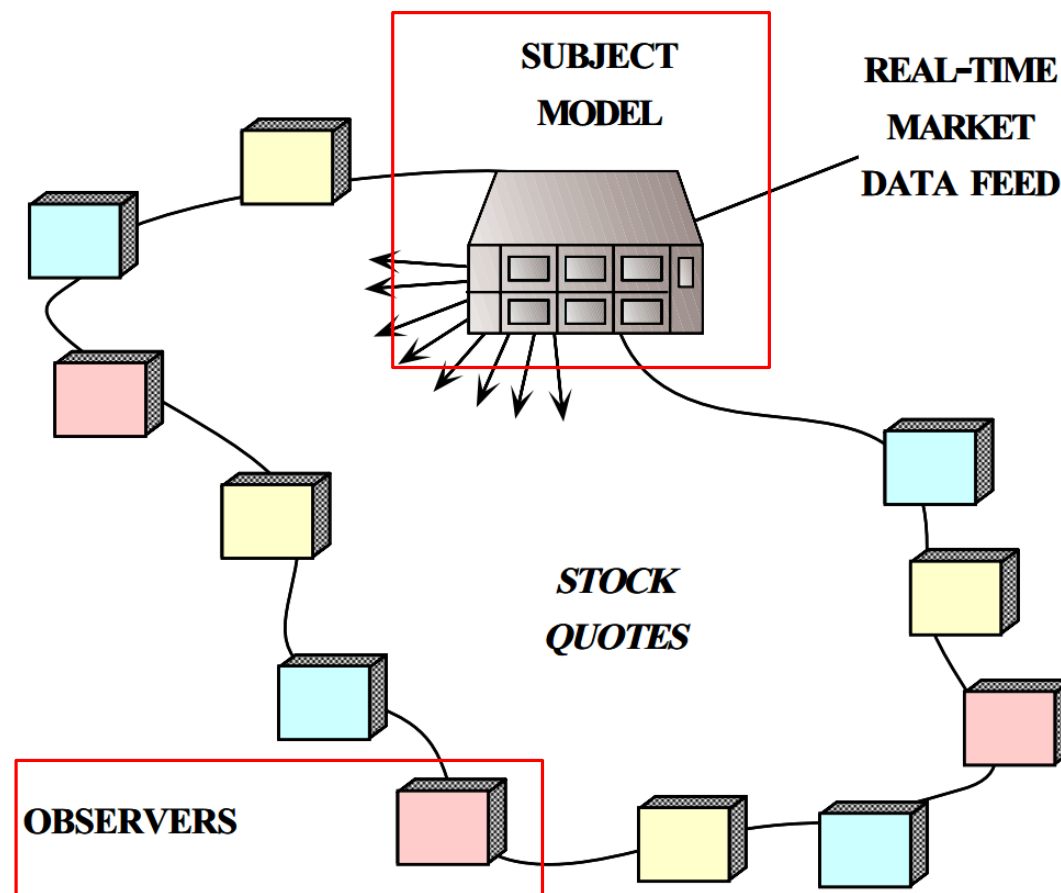
ГЛ. АС. Д-Р ЕМИЛ ДОЙЧЕВ

Общи сведения

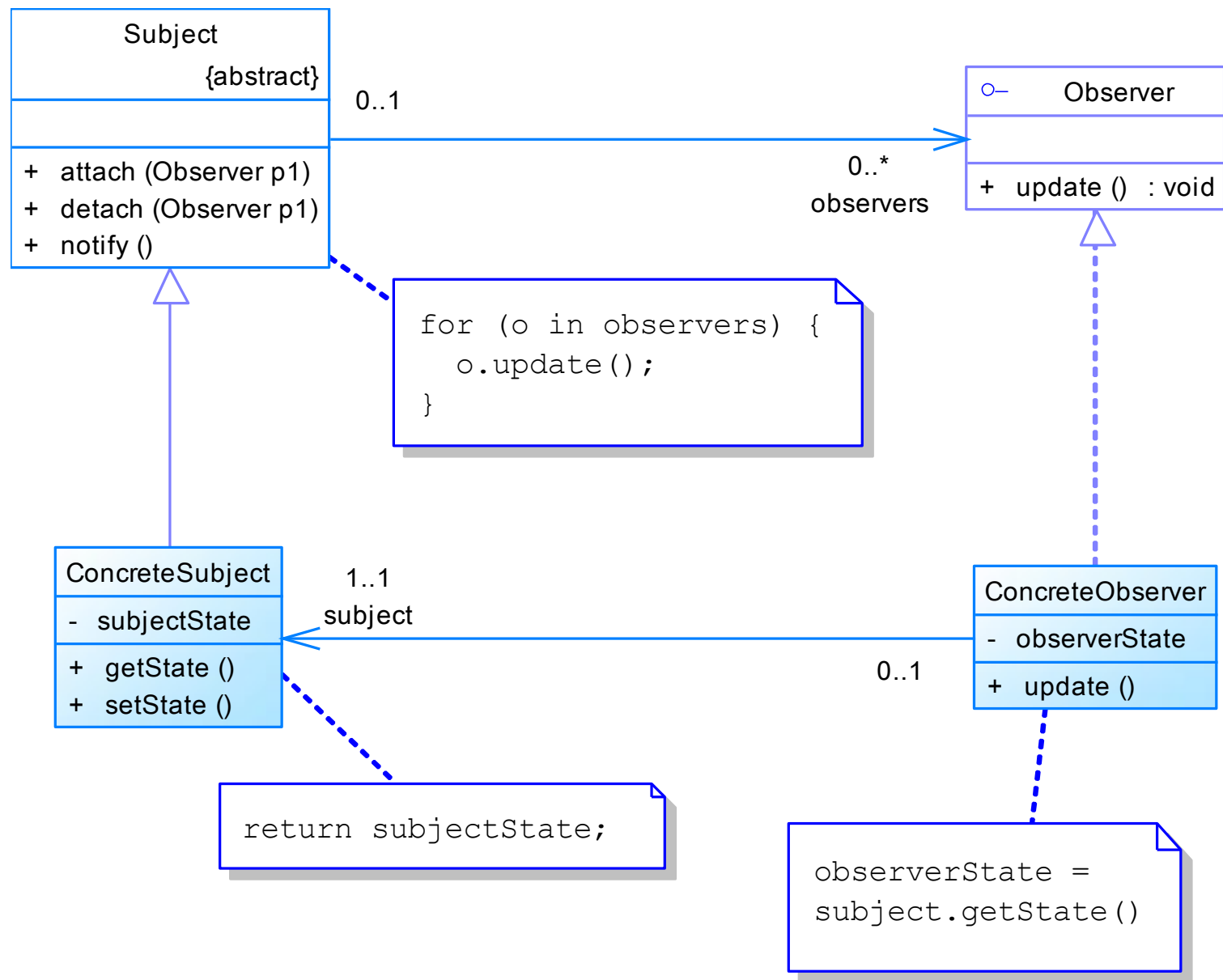
- ✓ **Вид:** Поведенчески за обекти
- ✓ **Цел:** дефинира зависимост от типа „едно към много“ между обекти, така че когато един обект промени състоянието си, всички зависими от него обекти да бъдат уведомявани и обновявани автоматично.
- ✓ **Известен и като:** Зависимост (Dependents), Публикация-Абонамент (Publish-Subscribe)

Мотивация

- ✓ **Мотивация:** Чест страничен ефект от раздробяването на една система на колекция от коопериращи се класове е нуждата от поддръжката на съгласуваност между свързаните обекти.
- ✓ **Приложимост:**
 - Когато някоя абстракция има два аспекта и единият зависи от другия.
 - Когато промяна в един обект изисква промяна и на други обекти, без да се знае броят им.
 - Когато някой обект трябва да може да известява други обекти, без да прави предположения кои са те.



Структура

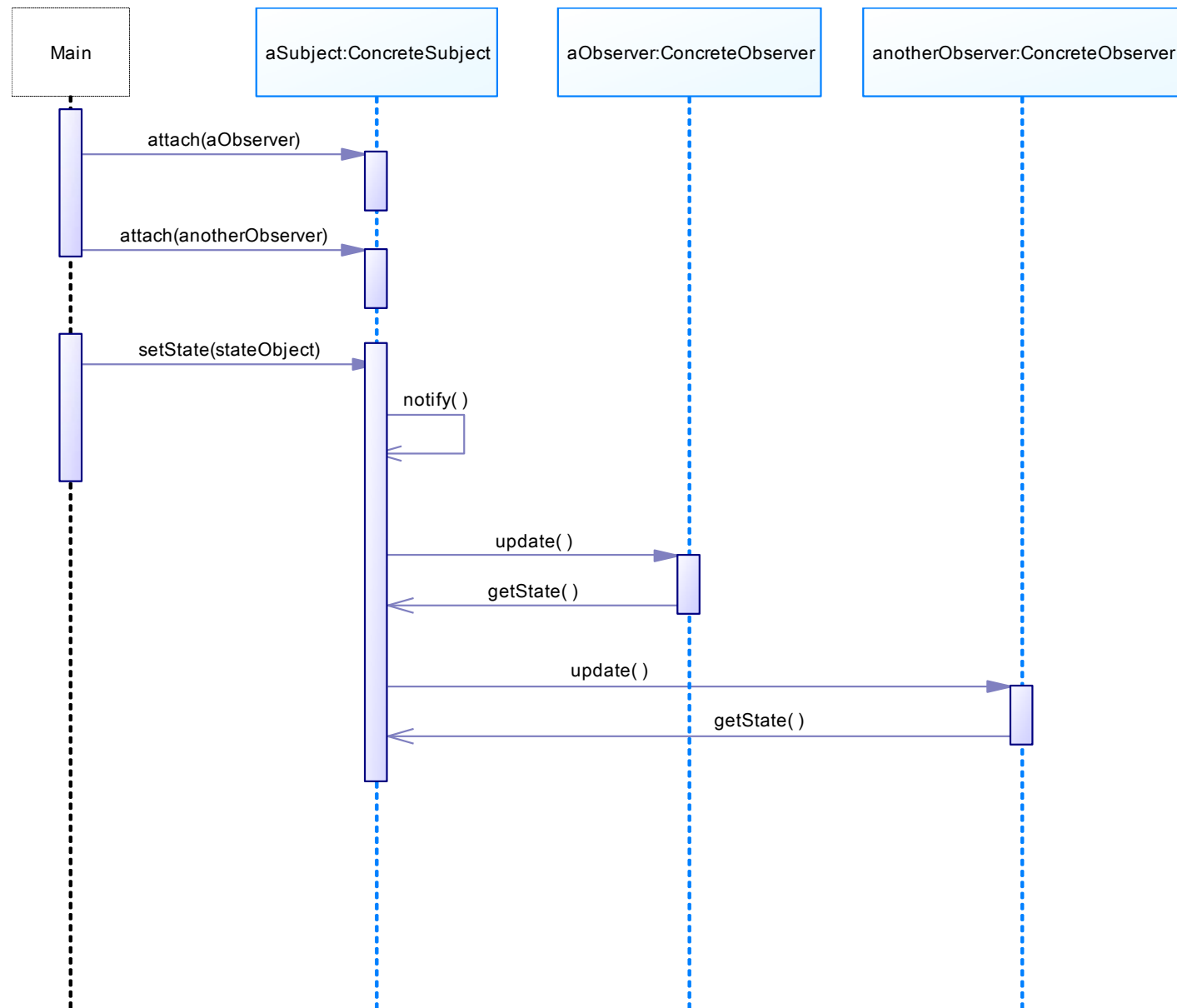


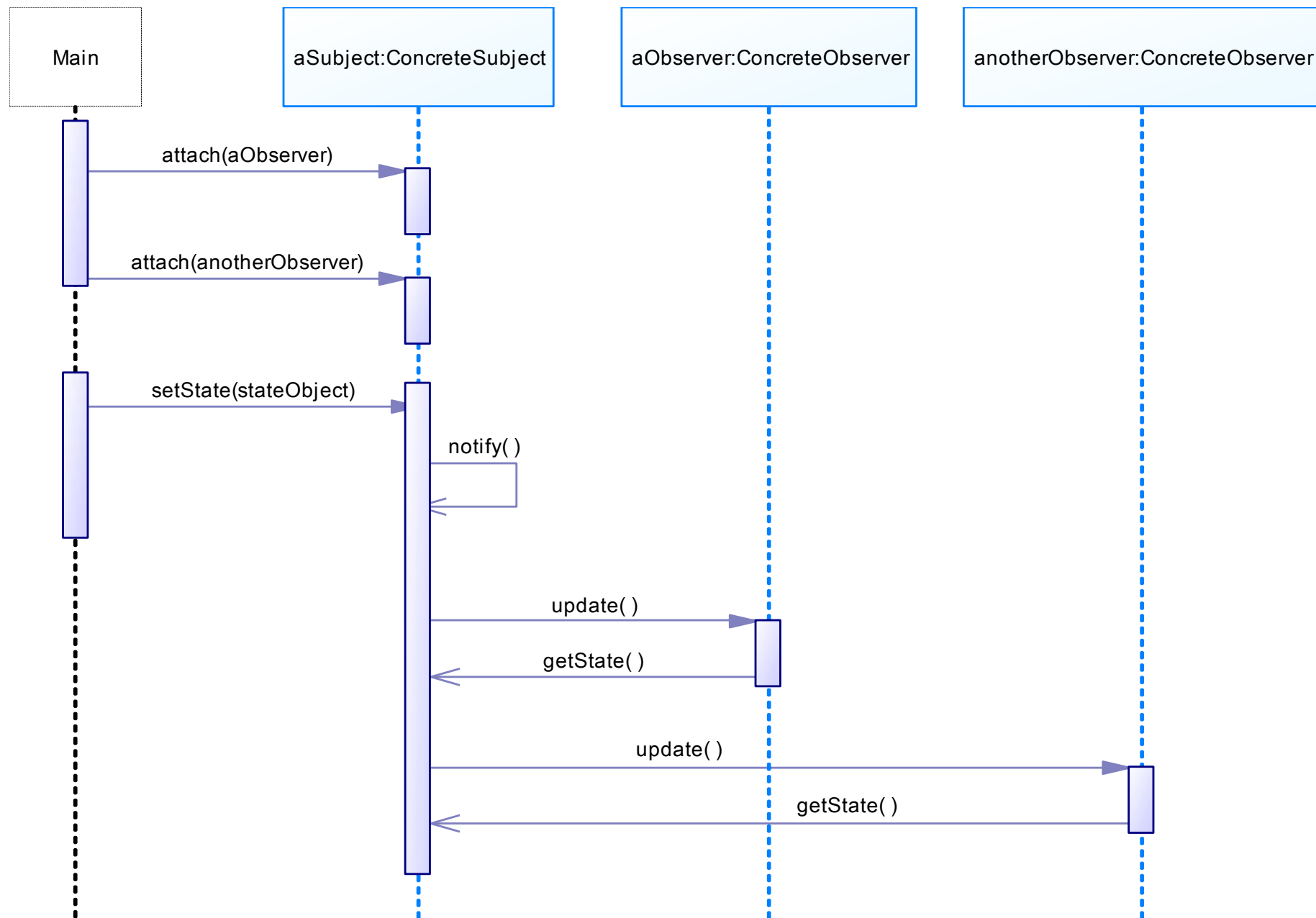
Участници

- ✓ **Subject** – знае наблюдателите си (произволен брой observer обекти могат да наблюдават); предоставя интерфейс за закачване и откачване;
- ✓ **Observer** – дефинира интерфейс за обекти, които трябва да бъдат известявани;
- ✓ **ConcreteSubject** – съхранява състояние необходимо на ConcreteObserver обекти; изпраща известие до своите наблюдатели когато състоянието му се промени.
- ✓ **ConcreteObserver** – поддържа връзка към ConcreteSubject обект; съхранява състояние, което трябва да бъде съгласувано с това на наблюдавания обект; имплементира интерфейса за обновяване на наблюдатели.

Взаимодействия

- ✓ **ConcreteSubject** известява своите наблюдатели, когато настъпи промяна, която ще направи състоянието на наблюдателите му различно от неговото.
- ✓ **ConcreteObserver** обекта може да поиска от наблюдавания обект информация за състоянието му





Следствия

✓ Предимства:

- Минимално обвързване между *Subject* и *Observer* (абстрактна връзка)
 - може да се използват многократно *наблюдаваните обекти* без да се използват *наблюдателите* им и обратното;
 - могат да бъдат добавяни *наблюдатели* без да бъде променян *наблюдавания обект*;
 - всички *subjects* „знаят“ кой ги наблюдава;
 - *наблюдаваните обекти* не е нужно да знаят конкретните класове на своите наблюдатели, а само това, че те имплементират интерфейса *Observer*, който има метод *update()*;
 - *наблюдаваните обекти* и *наблюдателите* могат да принадлежат на различни слоеве на абстракция на системата.
- Поддръжка на бродкаст комуникация.
 - *наблюдаваните обекти* информират всички абониран *наблюдатели*;
 - могат да бъдат добавяни/премахвани *наблюдатели* по всяко време

Следствия

✓ Възможни проблеми

- Вероятност от каскадни обновявания
- Простият *Observer* интерфейс изисква *наблюдателите* да открият сами какво е променено в състоянието на *наблюдавания обект*

Имплементация

- ✓ Как може *subject* да поддържа списъка с наблюдателите си?
 - Array, List, ... (всяка колекция)
- ✓ Как може един *наблюдател* да наблюдава повече от един *subject*?
 - Чрез разширяване на метода *update* на интерфейса *Observer*, така че наблюдаваният обект да може да информира *наблюдателя* за себе си
- ✓ Кой инициира обновяването?
 - *Наблюдаваният обект* винаги когато неговото състояние се промени.
 - *Клиентите*, които извършват промяната на състоянието
 - предимството, е че клиентите могат да изчакат да стартират обновяването след като направят серия от промени
 - недостатък е, че клиентите са натоварени с допълнителни задължения и могат да забравят да извикат *notify()*
- ✓ Важно е да се стартира обновяването след като състоянието на наблюдавания обект стане стабилно

Имплементация

- ✓ Колко информация трябва да изпраща *наблюдавания обект* до *наблюдателите* си?
 - модел push – цялата информация за състоянието си (недостатък – това предполага, че *наблюдаваният обект* знае нещо за нуждите на *наблюдателите* си)
 - модел pull – съвсем малко (недостатък – може да е неефективен защото *наблюдателите* трябва да разберат какво е променено без помощ от *наблюдавания обект*)
- ✓ Могат ли *наблюдателите* да се абонират за специфични събития?
 - ако могат тогава името на шаблона се променя на publish-subscribe
- ✓ Как може *наблюдател* да бъде информиран само след като няколко *наблюдавани обекта* променят състоянието си?
 - използва се междинен обект, който действа като посредник
 - *наблюдаваните обекти* информират *посредника* при промяна на състоянието си, а той от своя страна акумулира тези събития и информира *наблюдателя* при настъпване на необходимите промени във всички *наблюдавани обекти*

Известни употреби

- ✓ Smalltalk Model/View/Controller user interface framework
 - Model = Subject
 - View = Observer
 - Controller е обекта, който променя състоянието на subject
- ✓ Java AWT 1.1/Swing Event Model

Свързани шаблони

- ✓ Посредник (Mediator) – за реализиране на комплексна семантика за обновяване на *наблюдателите*.

Пример

- ✓ Реализация може да се направи „на чисто“.
- ✓ Java предоставя `Observable/Observer` класове като вградена поддръжка на шаблонът *Наблюдател*.
- ✓ Класът `java.util.Observable` е основният *Subject* клас. Всеки клас, който иска да бъде наблюдаван може да наследява този клас.
 - предоставя методи за добавяне/изтриване на наблюдатели
 - предоставя методи за известяване на всички наблюдатели
- ✓ Интерфейсът `java.util.Observer` е *Observer* интерфейса от структурата на шаблона. Той трябва да бъде имплементиран от класовете наблюдатели.

Класът java.util.Observable

Modifier and Type	Method and Description
void	<code>addObserver (Observer o)</code> Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	<code>clearChanged ()</code> Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the <code>hasChanged</code> method will now return <code>false</code> .
int	<code>countObservers ()</code> Returns the number of observers of this <code>Observable</code> object.
void	<code>deleteObserver (Observer o)</code> Deletes an observer from the set of observers of this object.
void	<code>deleteObservers ()</code> Clears the observer list so that this object no longer has any observers.
boolean	<code>hasChanged ()</code> Tests if this object has changed.
void	<code>notifyObservers ()</code> If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
void	<code>notifyObservers (Object arg)</code> If this object has changed, as indicated by the <code>hasChanged</code> method, then notify all of its observers and then call the <code>clearChanged</code> method to indicate that this object has no longer changed.
protected void	<code>setChanged ()</code> Marks this <code>Observable</code> object as having been changed; the <code>hasChanged</code> method will now return <code>true</code> .

Класът `java.util.Observable`

- ✓ Метод `notifyObservers (Object arg)` – ако обекта е променен (определя се с извикване на `hasChanged ()`) известява всички свои *наблюдатели* и след това извиква `clearChanged ()` за да укаже, че този обект няма нови промени.
 - Методът `update()` на всеки *наблюдател* се извиква с два аргумента:
 - наблюдаваният обект (т.е. `this`);
 - параметърът `arg`, който може да бъде използван за да се укаже на *наблюдателя* кой атрибут на *наблюдавания* обект е променен.
- ✓ Метод `notifyObservers ()` – също както горният метод, но в този случай `arg` е `null`.

Интерфейсът java.util.Observer

Method Detail

update

```
void update(Observable o,  
            Object arg)
```

This method is called whenever the observed object is changed. An application calls an Observable object's `notifyObservers` method to have all the object's observers notified of the change.

Parameters:

`o` - the observable object.

`arg` - an argument passed to the `notifyObservers` method.

Пример

```
/**
 * Пример за шаблона Наблюдател към дисциплината "Шаблони за проектиране"
 */
package fmi.patterns.lections.observer;

import java.util.Observable;

/**
 * Някакъв subject за наблюдаване
 */
public class ConcreteSubject extends Observable {
    private String name;
    private float price;

    public ConcreteSubject(String name, float price) {
        this.name = name;
        this.price = price;
        System.out.println("ConcreteSubject created: " + name + " at " + price);
    }

    public String getName() {
        return name;
    }

    public float getPrice() {
        return price;
    }

    public void setName(String name) {
        this.name = name;
        setChanged();
        notifyObservers(name);
    }

    public void setPrice(float price) {
        this.price = price;
        setChanged();
        notifyObservers(new Float(price));
    }
}
```

Пример

```
/**
 * Пример за шаблона Наблюдател към дисциплината "Шабини за проектиране"
 */
package fmi.patterns.lections.observer;

import java.util.Observable;
import java.util.Observer;

/**
 * Наблюдател на промяна на името
 */
public class NameObserver implements Observer {

    private String name;

    public NameObserver() {
        name = null;
        System.out.println("NameObserver created: Name is " + name);
    }

    /*
     * (non-Javadoc)
     *
     * @see java.util.Observer#update(java.util.Observable, java.lang.Object)
     */
    @Override
    public void update(Observable obj, Object arg) {
        if (arg instanceof String) {
            name = (String) arg;
            System.out.println("NameObserver: Name changed to " + name);
        } else {
            System.out.println("NameObserver: Some other change to subject!");
        }
    }
}
```

Пример

```
/**
 * Пример за шаблона Наблюдател към дисциплината "Шаблони за проектиране"
 */
package fmi.patterns.lections.observer;

import java.util.Observable;
import java.util.Observer;

/**
 * Наблюдател на промяна на цената
 */
public class PriceObserver implements Observer {
    private float price;

    public PriceObserver() {
        price = 0;
        System.out.println("PriceObserver created: Price is " + price);
    }

    /*
     * (non-Javadoc)
     * @see java.util.Observer#update(java.util.Observable, java.lang.Object)
     */
    @Override
    public void update(Observable obj, Object arg) {
        if (arg instanceof Float) {
            price = ((Float) arg).floatValue();
            System.out.println("PriceObserver: Price changed to " + price);
        } else {
            System.out.println("PriceObserver: Some other change to subject!");
        }
    }
}
```

Пример

```
/**
 * Пример за шаблона Наблюдател към дисциплината "Шабини за проектиране"
 */
package fmi.patterns.lections.observer;

/**
 * Тестова програма за ConcreteSubject, NameObserver и PriceObserver
 */
public class TestObservers {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // Създаване на Subject и Observers.
        ConcreteSubject s = new ConcreteSubject("Пуканки", 1.29f);
        NameObserver nameObs = new NameObserver();
        PriceObserver priceObs = new PriceObserver();

        // Добавяне на наблюдателите!
        s.addObserver(nameObs);
        s.addObserver(priceObs);

        // Промяна в subject.
        s.setName("Печени ядки");
        s.setPrice(4.57f);
        s.setPrice(9.22f);
        s.setName("Тиквени семки");
    }
}
```

Пример

```
ConcreteSubject created: Пуканки at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to      subject!
NameObserver: Name changed to Печени ядки
PriceObserver: Price changed to 4.57
NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22
NameObserver: Some other change to subject!
PriceObserver: Some other change to      subject!
NameObserver: Name changed to Тиквени семки
```

Край: Шаблон Наблюдател

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ