

Паттерн Observer

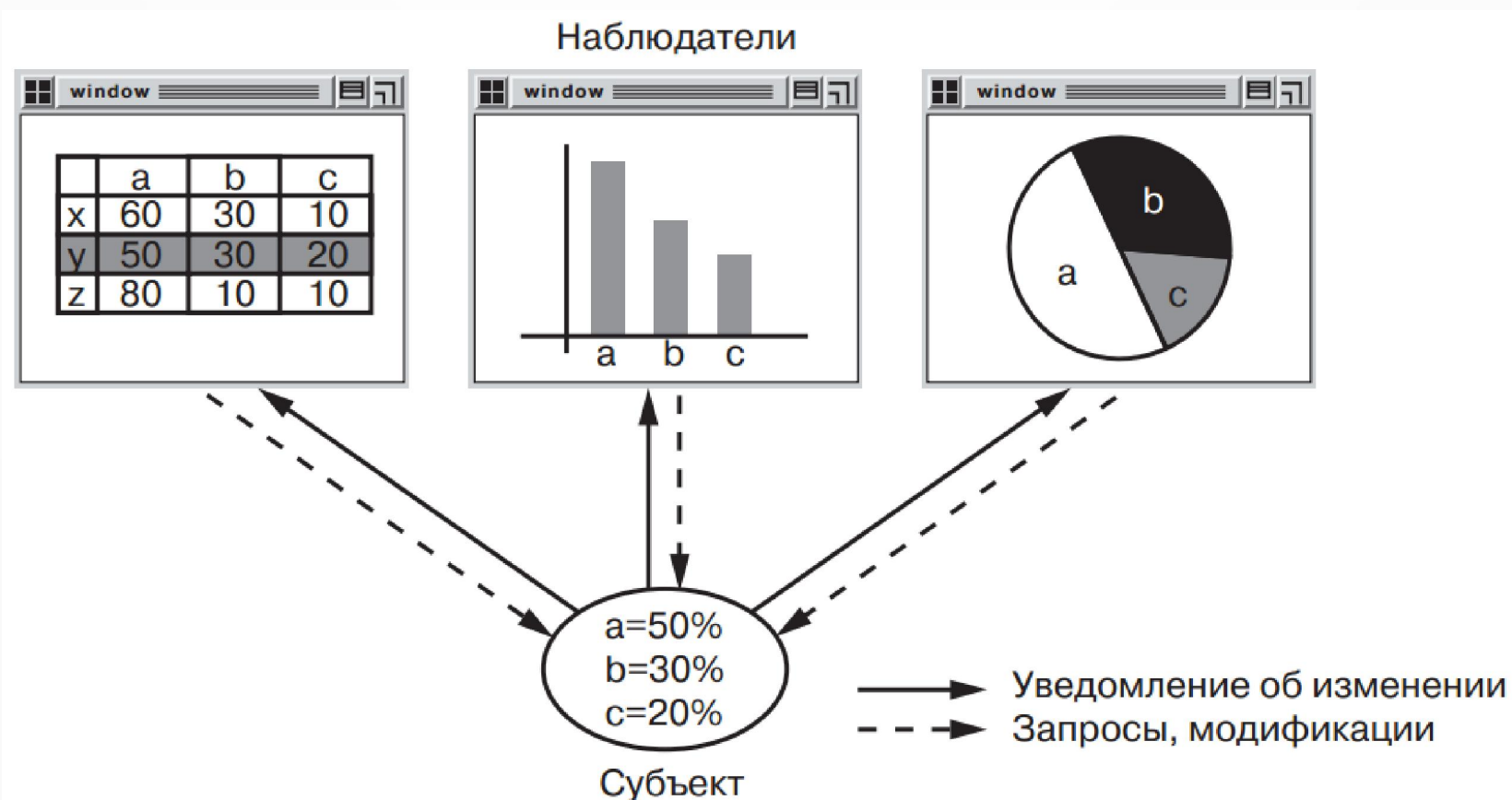
Выполнил:
студент 1 курса магистратуры
05182м группы
Василевский А.В.

Описание

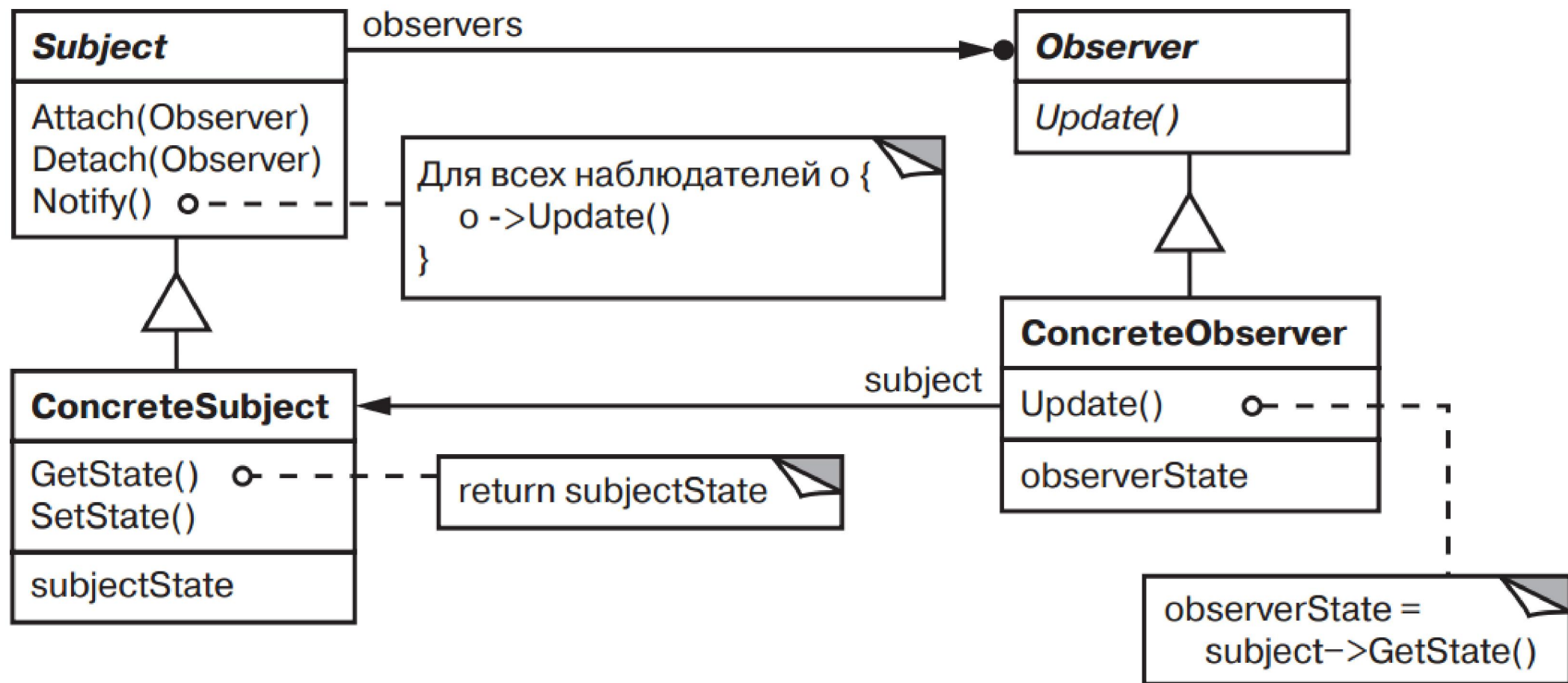
- Известен как **Observer (Наблюдатель), Dependents (Подчиненные), Publish-Subscribe (Издатель-Подписчик)**.
- Поведенческий паттерн.
- Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

Мотивация

- Автоматическое обновление графических виджетов при изменении их разделяемой **модели**.



Структура



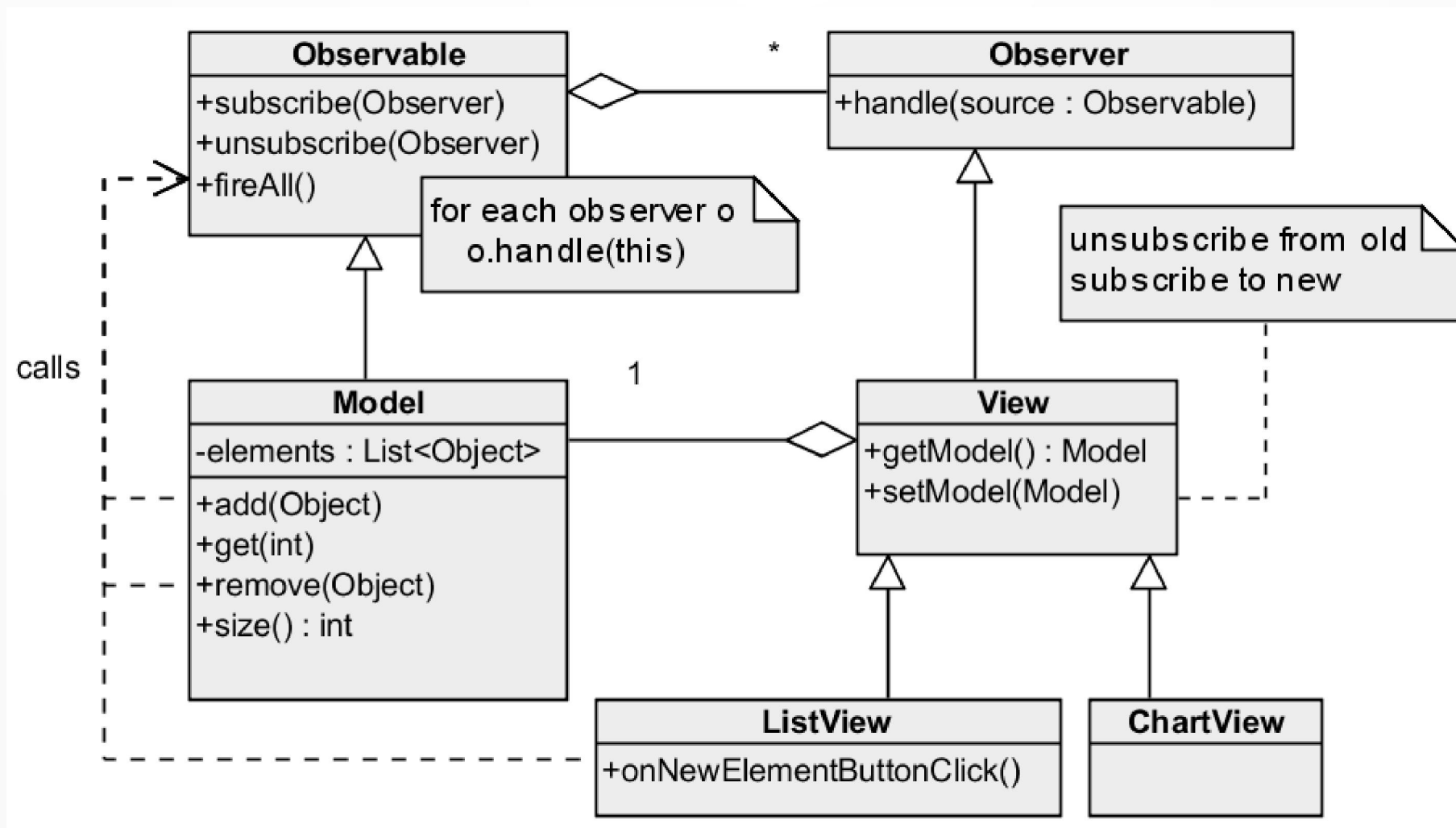
Участники

- **Subject** – субъект:
 - располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей;
 - предоставляет интерфейс для присоединения и отделения наблюдателей;
- **Observer** – наблюдатель:
 - определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта;
- **ConcreteSubject / ConcreteObserver** – конкретные реализации субъекта и наблюдателя

Достоинства и недостатки

- *+ Абстрактная связанность субъекта и наблюдателя.*
Субъект более низкого уровня абстракции может уведомлять наблюдателей на верхних уровнях абстракции.
- *+ Поддержка широковещательных коммуникаций.*
Уведомление автоматически поступает всем подписавшимся на него объектам.
- *– Неожиданные обновления.* Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта.

Реализация



Реализация

```
interface Observer {
    void handle(Observable source);
}

interface Observable {
    void subscribe(Observer o);
    void unsubscribe(Observer o);
    void fireAll();
}

abstract class View implements Observer {

    private Model model;

    public Model getModel() {
        return this.model;
    }

    public void setModel(Model m) {
        if (this.model != null) {
            this.model.unsubscribe(this);
        }
        this.model = m;
        this.model.subscribe(this);
        // ensure initial update
        this.handle(model);
    }
}
```

```
class Model implements Observable {
```

```
    private final List<Observer> observers;
    private final List<Object> elements;
    public Model() {
        this.observers = new ArrayList<>();
        this.elements = new ArrayList<>();
    }
    public void add(Object o) {
        this.elements.add(o);
        fireAll();
    }
    public void remove(Object o) {
        this.elements.remove(o);
        fireAll();
    }
    public Object get(int i) {
        return this.elements.get(i);
    }
    public int size() {
        return this.elements.size();
    }
    // Observable
    @Override public void subscribe(Observer o) {
        this.observers.add(o);
    }
    @Override public void unsubscribe(Observer o) {
        this.observers.remove(o);
    }
    @Override public void fireAll() {
        for (Observer o : observers) {
            o.handle(this);
        }
    }
}
```



Реализация

```
class ListView extends View {
    @Override
    public void handle(Observable source) {
        System.out.println("TODO: actual update of list view");
    }
    public void onNewElementButtonClick() {
        Object newElement = JOptionPane.showInputDialog(
            "Enter new element");
        getModel().add(newElement);
    }
}

class ChartView extends View {
    @Override
    public void handle(Observable source) {
        System.out.println("TODO: actual update of chart view");
    }
}
```

```
Model model = new Model();
model.add("item1");
model.add("item2");
```

```
ListView v1 = new ListView();
ChartView v2 = new ChartView();
```

```
v1.setModel(model);
v2.setModel(model);
```

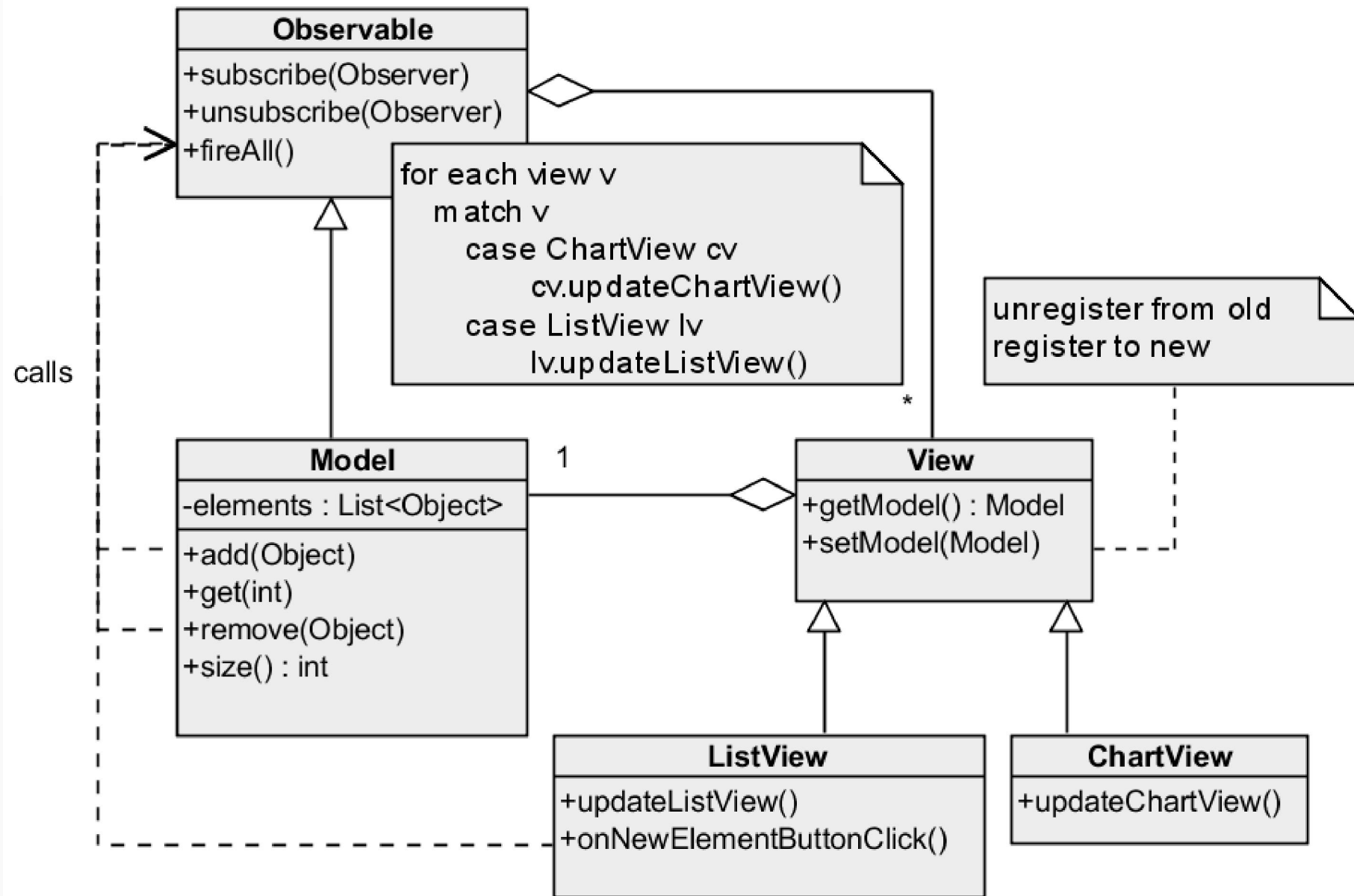
```
model.add("item3");
```

```
// TODO: actual update of list view
// TODO: actual update of chart view
```

```
v1.onNewElementButtonClick();
```

```
// TODO: actual update of list view
// TODO: actual update of chart view
```


Без Observer



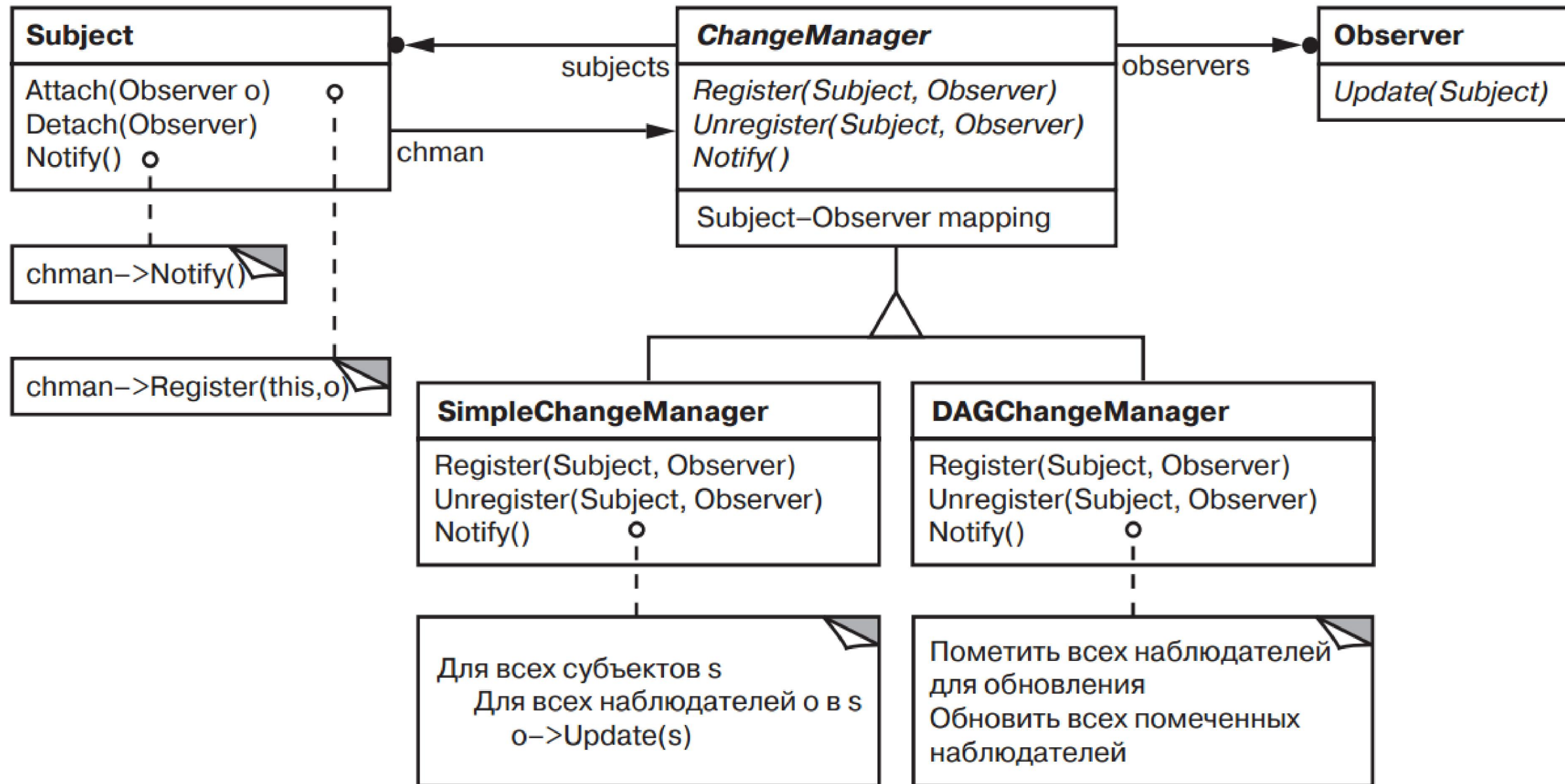
Без Observer

```
class ListView extends View {  
  
    @Override  
    public void setModel(Model m) {  
        super.setModel(m);  
        updateListView();  
    }  
    public void updateListView() {  
        System.out.println("TODO: actual update of list view");  
    }  
    public void onNewElementButtonClick() {  
        Object newElement = JOptionPane.showInputDialog(  
            "Enter new element");  
        getModel().add(newElement);  
    }  
}  
  
class ChartView extends View {  
  
    @Override  
    public void setModel(Model m) {  
        super.setModel(m);  
        updateChartView();  
    }  
    public void updateChartView() {  
        System.out.println("TODO: actual update of chart view");  
    }  
}
```

```
class Model {  
  
    private final List<View> views;  
    private final List<Object> elements;  
    public Model() {  
        this.views = new ArrayList<>();  
        this.elements = new ArrayList<>();  
    }  
    public void add(Object o) {  
        this.elements.add(o);  
        fireAll();  
    }  
    public void remove(Object o) {  
        this.elements.remove(o);  
        fireAll();  
    }  
    public Object get(int i) {  
        return this.elements.get(i);  
    }  
    public int size() {  
        return this.elements.size();  
    }  
    public void register(View o) {  
        this.views.add(o);  
    }  
    public void unregister(View o) {  
        this.views.remove(o);  
    }  
    public void fireAll() {  
        for (View o : views) {  
            if (o instanceof ListView)  
                ((ListView) o).updateListView();  
            if (o instanceof ChartView)  
                ((ChartView) o).updateChartView();  
        }  
    }  
}
```



ChangeManager



Родственные паттерны

- **Посредник (Mediator)**: класс **ChangeManager** действует как посредник между субъектами и наблюдателями, инкапсулируя сложную семантику обновления.
- **Одиночка (Singleton)**: класс **ChangeManager** может воспользоваться паттерном одиночка, чтобы гарантировать уникальность и глобальную доступность менеджера изменений.

Использование

- Практически все MV-паттерны (MVC, MVP, MVVM).
- Практически все современные GUI-фреймворки.
 - Swing/AWT (java): class *AWTEvent*, interface *XXXListener* (Mouse, Key, Component, etc.), *Component#addXXXListener(XXXListener l)*.
 - WPF/WinForms (C#): связка event+delegate. Также можно подписаться на события.
 - Qt (C++): концепция signals & slots. Добавление наблюдателя методом `connect(signal, slot)`.

Литература

- [1] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2015. — 368 с.: ил. — (Серия «Библиотека программиста»).