# Паттерн Observer (наблюдатель, издатель-подписчик)

## Назначение паттерна Observer

* Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.
* Паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.
* Паттерн Observer определяет часть "View" в модели Model-View-Controller (MVC) .

Паттерн Observer находит широкое применение в системах пользовательского интерфейса, в которых данные и их представления ("виды") отделены друг от друга. При изменении данных должны быть изменены все представления этих данных (например, в виде таблицы, графика и диаграммы).

## Решаемая проблема

Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).

## Обсуждение паттерна Observer

Паттерн Observer определяет объект Subject, хранящий данные (модель), а всю функциональность "представлений" делегирует слабосвязанным отдельным объектам Observer. При создании наблюдатели Observer регистрируются у объекта Subject. Когда объект Subject изменяется, он извещает об этом всех зарегистрированных наблюдателей. После этого каждый обозреватель запрашивает у объекта Subject ту часть состояния, которая необходима для отображения данных.

Такая схема позволяет динамически настраивать количество и "типы" представлений объектов.

Описанный выше протокол взаимодействия соответствует модели вытягивания (pull), когда субъект информирует наблюдателей о своем изменении, и каждый наблюдатель ответственен за "вытягивание" у Subject нужных ему данных. Существует также модель проталкивания, когда субъект Subject посылает ("проталкивает") наблюдателям детальную информацию о своем изменении.

Существует также ряд вопросов, о которых следует упомянуть, но обсуждение которых останется за рамками данной статьи:

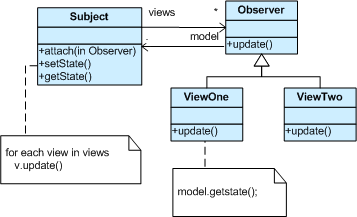
* Реализация "компрессии" извещений (посылка единственного извещения на серию последовательных изменений субъекта Subject).
* Мониторинг нескольких субъектов с помощью одного наблюдателя Observer.
* Исключение висячих ссылок у наблюдателей на удаленные субъекты. Для этого субъект должен уведомить наблюдателей о своем удалении.

Паттерн Observer впервые был применен в архитектуре Model-View-Controller языка Smalltalk, представляющей каркас для построения пользовательских интерфейсов.

## Структура паттерна Observer

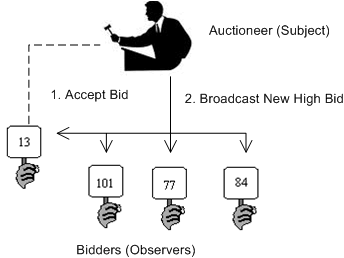
Subject представляет главную (независимую) абстракцию. Observer представляет изменяемую (зависимую) абстракцию. Субъект извещает наблюдателей о своем изменении, на что каждый наблюдатель может запросить состояние субъекта.

### UML-диаграмма классов паттерна Observer



## Пример паттерна Observer

Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически. Некоторые аукционы демонстрируют этот паттерн. Каждый участник имеет карточку с цифрами, которую он использует для обозначения предлагаемой цены (ставки). Ведущий аукциона (Subject) начинает торги и наблюдает, когда кто-нибудь поднимает карточку, предлагая новую более высокую цену. Ведущий принимает заявку, о чем тут же извещает всех участников аукциона (Observers).



## Использование паттерна Observer

1. Проведите различия между основной (или независимой) и дополнительной (или зависимой) функциональностями.
2. Смоделируйте "независимую" функциональность с помощью абстракции "субъект".
3. Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
4. Класс Subject связан только c базовым классом Observer.
5. Клиент настраивает количество и типы наблюдателей.
6. Наблюдатели регистрируются у субъекта.
7. Субъект извещает всех зарегистрированных наблюдателей.
8. Субъект может "протолкнуть" информацию в наблюдателей, или наблюдатели могут "вытянуть" необходимую им информацию от объекта Subject.

## Особенности паттерна Observer

* Паттерны [Chain of Responsibility](http://cpp-reference.ru/patterns/behavioral-patterns/chain-of-responsibility/), [Command](http://cpp-reference.ru/patterns/behavioral-patterns/command/), [Mediator](http://cpp-reference.ru/patterns/behavioral-patterns/mediator/) и Observer показывают, как можно разделить отправителей и получателей запросов с учетом своих особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей. Command определяет связь - "оправитель-получатель" с помощью подкласса. В Mediator отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне Observer связь между отправителем и получателем получается слабой, при этом число получателей может конфигурироваться во время выполнения.
* Mediator и Observer являются конкурирующими паттернами. Если Observer распределяет взаимодействие c помощью объектов "наблюдатель" и "субъект", то Mediator использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Мы обнаружили, что легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.
* Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.

## Реализация паттерна Observer

### Реализация паттерна Observer по шагам

1. Смоделируйте "независимую" функциональность с помощью абстракции "субъект".
2. Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
3. Класс Subject связан только c базовым классом Observer.
4. Наблюдатели регистрируются у субъекта.
5. Субъект извещает всех зарегистрированных наблюдателей.
6. Наблюдатели "вытягивают" необходимую им информацию от объекта Subject.
7. Клиент настраивает количество и типы наблюдателей.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | #include <iostream>  #include <vector>  using namespace std;    // 1. "Независимая" функциональность  class Subject {      // 3. Связь только базовым классом Observer      vector < class Observer \* > views;      int value;    public:      void attach(Observer \*obs) {          views.push\_back(obs);      }      void setVal(int val) {          value = val;          notify();      }      int getVal() {          return value;      }      void notify();  };    // 2. "Зависимая" функциональность  class Observer {      Subject \*model;      int denom;    public:      Observer(Subject \*mod, int div) {          model = mod;          denom = div;          // 4. Наблюдатели регистрируются у субъекта          model->attach(this);      }      virtual void update() = 0;    protected:      Subject \*getSubject() {          return model;      }      int getDivisor() {          return denom;      }  };    void Subject::notify() {    // 5. Извещение наблюдателей    for (int i = 0; i < views.size(); i++)      views[i]->update();  }    class DivObserver: public Observer {    public:      DivObserver(Subject \*mod, int div): Observer(mod, div){}      void update() {          // 6. "Вытягивание" интересующей информации          int v = getSubject()->getVal(), d = getDivisor();          cout << v << " div " << d << " is " << v/d << '\n';      }  };    class ModObserver: public Observer {    public:      ModObserver(Subject \*mod, int div): Observer(mod, div){}      void update() {          int v = getSubject()->getVal(), d = getDivisor();          cout << v << " mod " << d << " is " << v%d << '\n';      }  };    int main() {    Subject subj;    DivObserver divObs1(&subj, 4); // 7. Клиент настраивает число    DivObserver divObs2(&subj, 3); //    и типы наблюдателей    ModObserver modObs3(&subj, 3);    subj.setVal(14);  } |

Вывод программы:

|  |  |
| --- | --- |
| 1 | 14 div 4 is 3 14 div 3 is 4 14 mod 3 is 2 |

### Реализация паттерна Observer: до и после

#### До

Количество и типы "зависимых" объектов определяются классом Subject. Пользователь не имеет возможности влиять на эту конфигурацию.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | class DivObserver  {      int m\_div;    public:      DivObserver(int div)      {          m\_div = div;      }      void update(int val)      {          cout << val << " div " << m\_div               << " is " << val / m\_div << '\n';      }  };    class ModObserver  {      int m\_mod;    public:      ModObserver(int mod)      {          m\_mod = mod;      }      void update(int val)      {          cout << val << " mod " << m\_mod               << " is " << val % m\_mod << '\n';      }  };    class Subject  {      int m\_value;      DivObserver m\_div\_obj;      ModObserver m\_mod\_obj;    public:      Subject(): m\_div\_obj(4), m\_mod\_obj(3){}      void set\_value(int value)      {          m\_value = value;          notify();      }      void notify()      {          m\_div\_obj.update(m\_value);          m\_mod\_obj.update(m\_value);      }  };    int main()  {    Subject subj;    subj.set\_value(14);  } |

Вывод программы:

|  |  |
| --- | --- |
| 1 | 14 div 4 is 3 14 mod 3 is 2 |

#### После

Теперь класс Subject не связан с непосредственной настройкой числа и типов объектов Observer. Клиент установил два наблюдателя DivObserver и одного ModObserver.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | class Observer  {    public:      virtual void update(int value) = 0;  };    class Subject  {      int m\_value;      vector m\_views;    public:      void attach(Observer \*obs)      {          m\_views.push\_back(obs);      }      void set\_val(int value)      {          m\_value = value;          notify();      }      void notify()      {          for (int i = 0; i < m\_views.size(); ++i)            m\_views[i]->update(m\_value);      }  };    class DivObserver: public Observer  {      int m\_div;    public:      DivObserver(Subject \*model, int div)      {          model->attach(this);          m\_div = div;      }       /\* virtual \*/void update(int v)      {          cout << v << " div " << m\_div << " is " << v / m\_div << '\n';      }  };    class ModObserver: public Observer  {      int m\_mod;    public:      ModObserver(Subject \*model, int mod)      {          model->attach(this);          m\_mod = mod;      }       /\* virtual \*/void update(int v)      {          cout << v << " mod " << m\_mod << " is " << v % m\_mod << '\n';      }  };    int main()  {    Subject subj;    DivObserver divObs1(&subj, 4);    DivObserver divObs2(&subj, 3);    ModObserver modObs3(&subj, 3);    subj.set\_val(14);  } |

Вывод программы:

|  |  |
| --- | --- |
| 1 | 14 div 4 is 3 14 div 3 is 4 14 mod 3 is 2 |

Источник: <http://sourcemaking.com/design_patterns/observer/>