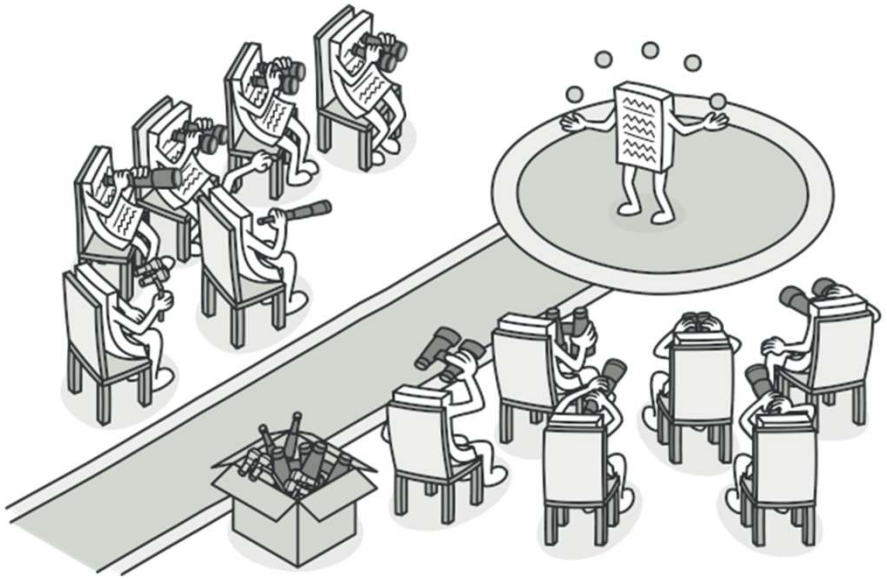
Спостерігач

або ж Видавець-Підписник, Слухач, Observer

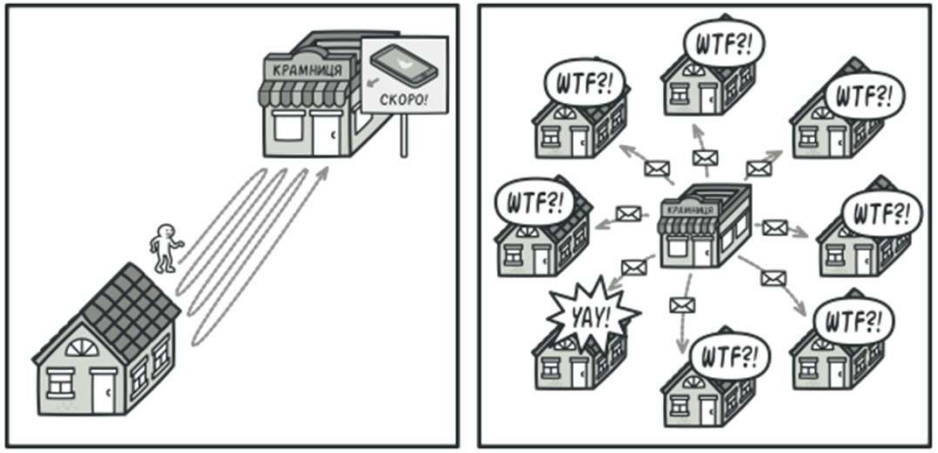


https://refactoring.guru/images/patterns/content/observer/observer.png

Аналогія з реального світу

Передплата та доставка газет.

https://refactoring.guru/images/patterns/content/observer/observer-comic-2-uk.png

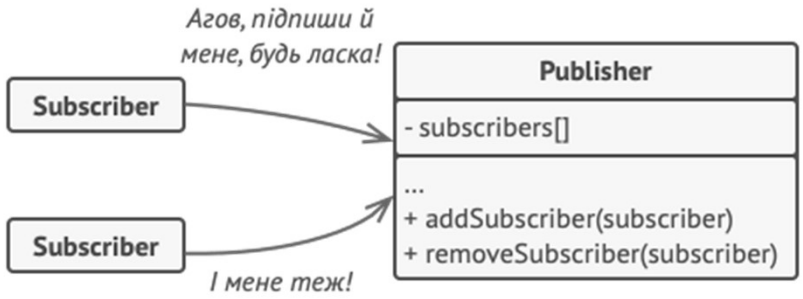


Постійне відвідування магазину чи спам?

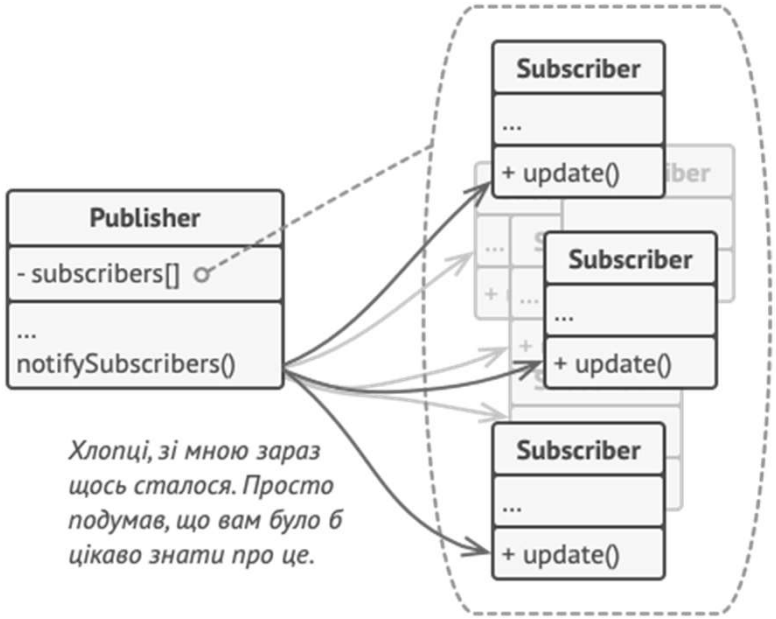
https://refactoring.guru/images/patterns/content/observer/observer-comic-1-uk.png

Структура

Subject (або Publisher) – тримає стан, про зміну якого повідомляє усіх спостерігачів.
Subject також надає інтерфейс для підписки на його розсилку.

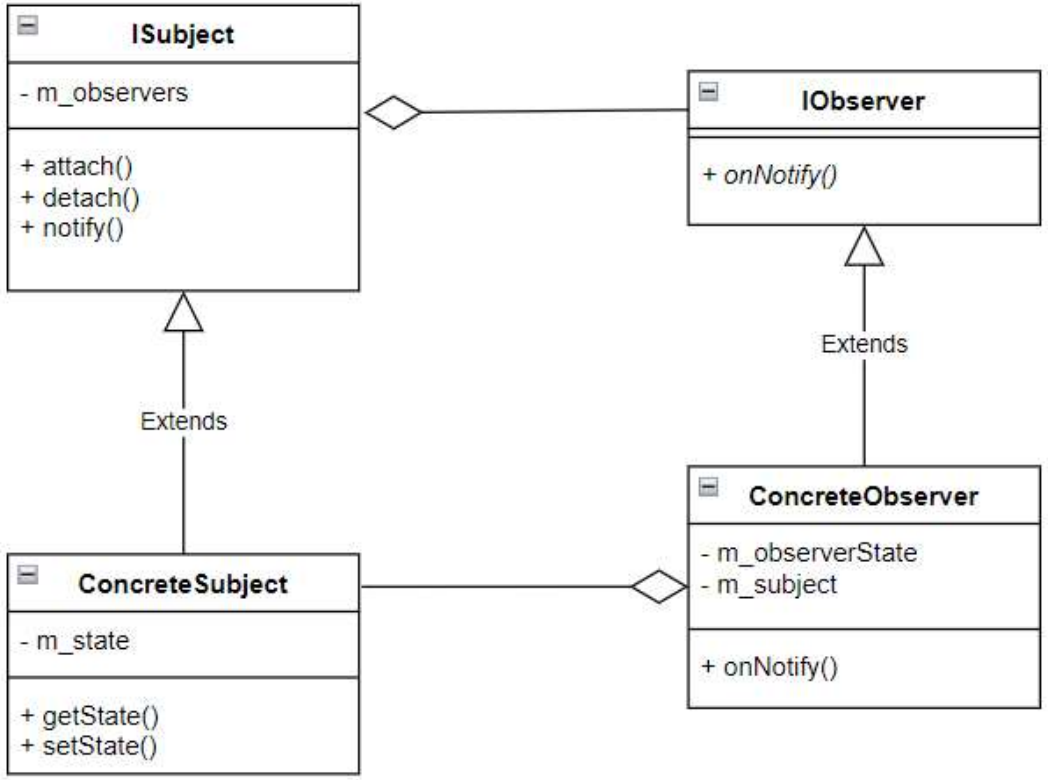


Observer (або Subscriber) – клас, який буде підписуватись на повідомлення. Він матиме метод, який викликатиме Subject під час зміни свого стану.



Реалізація 1 - GoF

Проект: observer-gof



Реалізація 1 - GoF

```cpp
// ---- <IObserver.h> ----------------------------------------

class IObserver {
public:
    virtual ~IObserver() {}

    virtual void onNotify() = 0;
};
```

Надаємо спільний інтерфейс

```cpp
// ---- <ConcreteObserver.h> ----------------------------------------

#include "IObserver.h"
#include "ConcreteSubject.h"
#include <iostream>

class ConcreteObserver : public IObserver {
public:
    ConcreteObserver(ConcreteSubject& subject)
        : m_subject(subject), m_observerState(subject.getState()) {
        m_subject.attach(observer: this);
    }

    void onNotify() {
        m_observerState = m_subject.getState();
        std::cout << "Notified and updated state to "
            << m_observerState << "\n";
    }

private:
    ConcreteSubject& m_subject;
    int m_observerState;
};
```

У кожного спостерігача своя реакція на подію!

Знаємо про ConcreteSubject і за потреби використовуємо його інтерфейс

Реалізація 1 - GoF

```cpp
// ---- <ISubject.h> ----------------------------

#include "IObserver.h"
#include <unordered_set>

class ISubject {
public:
    void attach(IObserver* observer) {
        if (observer != nullptr) {
            m_observers.insert(observer);
        }
    }

    void detach(IObserver* observer) {
        m_observers.erase(observer);
    }

    void notify() {
        for (auto observer : m_observers) {
            observer->onNotify();
        }
    }

private:
    std::unordered_set<IObserver*> m_observers;
};
```

Методи для динамічної реєстрації спостерігачів

Повідомляємо усіх спостерігачів про зміну стану

Надаємо інтерфейс для читання стану

```cpp
// ---- <ConcreteSubject.h> ----------------------

#include "ISubject.h"

class ConcreteSubject : public ISubject {
public:
    ConcreteSubject() : m_state(0) {}

    void setState(int i) {
        m_state = i;
        notify();
    }

    int getState() {
        return m_state;
    }

private:
    int m_state;
};
```

GoF - Наслідки

Абстрактний зв'язок між видавцем і спостерігачем.

- Видавець має лише список усіх спостерігачів
- Немає жорсткої прив’язки до реалізації спостерігача
- Можливість комунікації між різними рівнями

Підтримка “broadcast” зв’язку

- Немає потреби вказувати отримувача повідомлення
- Спостерігач сам вирішує:
  - До яких видавців підписуватись
  - Як на них реагувати

Несподівані оновлення

- Найменша зміна стану видавця може призвести до великого ланцюга оновлень у спостерігачів
- Цей простий протокол не надає спостерігачу жодної інформації про те, що саме змінилось

Проблеми (нюанси) реалізації

- Зберігання спостерігачів
- Спостерігання більш ніж за 1 видавцем
- Хто запускає оновлення?
- Підвислі посилання та memory-safe рішення
- Забезпечення інваріанту перед оновленням (Template Method)
- Push/pull моделі
- Типізація сповіщень
- Інкапсуляція семантики оновлення
- Поєднання інтерфейсів
- Поєднання інтерфейсів Observer та Subject
- CRTP + std::function реалізація
- Багатопоточність

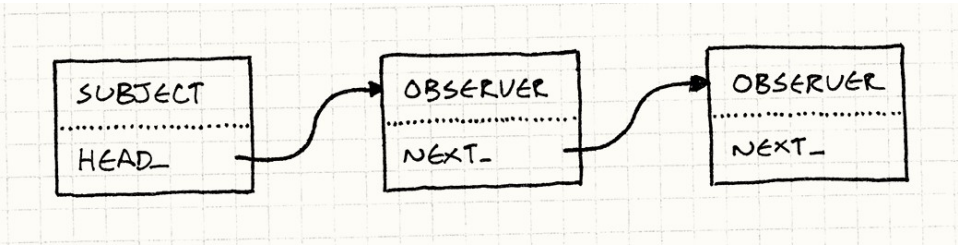
Зберігання спостерігачів

Найпростіший варіант – динамічна колекція (або хеш-таблиця)

- Може бути дорого через динамічне виділення пам’яті

Linked List!

- Зберігаємо наступного по списку у самому спостерігачі




```
void Subject::notify(const Entity& entity, Event event)
{
  Observer* observer = head_;
  while (observer != NULL)
  {
    observer->onNotify(entity, event);
    observer = observer->next_;
  }
}
```

Спостерігач при сповіщенні може повертати, який вказує, чи треба зупинити оновлення.
Майже **Chain of Responsibility**!

Спостерігання більш ніж за 1 видавцем

- Спостерігач не знатиме, від кого прийшла зміна
- Рішення – додати відсилку на видавця як параметр onNotify()

```
class Observer {
public:
    void onNotify(Subject& subject) {
        // we know what subject to examine
    }
};


class Subject {
public:
    // Subject
    void notify() {
        for (auto Observer * o : observers) {
            o->onNotify(& subject: *this);
        }
    }
private:
    std::unordered_set<Observer*> observers;
};
```

Хто запускає оновлення?

А. Видавець запускає оновлення при зміні стану

- Зручно, але може бути неефективно при послідовній зміні стану

```
// ---- <ConcreteSubject.h> ------------------

#include "ISubject.h"

class ConcreteSubject : public ISubject {
public:
    ConcreteSubject() : m_state(0) {}

    void setState(int i) {
        m_state = i;
        notify();
    }

    int getState() {
        return m_state;
    }

private:
    int m_state;
};
```

Б. Клієнти запускають оновлення у потрібний час

- Менше оновлень – ефективніше
- Клієнт відповідальний – можливо більше багів

```
class SomeConcreteSubject : public ISubject {
public:
    void changeState1();
    void changeState2();
    void changeState3();
};


// client code
int main() {
    SomeConcreteSubject subject;
    subject.changeState1();
    subject.changeState2();
    subject.changeState3();
    subject.notify();
}
```

«Підвислі» посилання

Видалення видавця не має ламати спостерігачів

- Спостерігач житиме у зелених окулярах, не знаючи про видалення видавця
- Якщо він агрегує видавця, ризикуємо багами
- → Сповіщати про видалення через notify() ☺

Видалення спостерігача не має ламати видавців

- Видалятись автоматично у деструкторі спостерігача
- → Smart pointers ☺

«Підвислі» посилання - рішення

Проект: observer-memorysafe

```
class ISubject {
public:
  virtual ~ISubject() {}

  virtual void addObserver(std::weak_ptr<IObserver> &&observer);
  virtual void removeObserver(const std::weak_ptr<IObserver> &observer);

  virtual void notify() {
    // go through list of observers
    for (auto std::list<std...r>>::iterator it = mObservers.begin(); it != mObservers.end();) {
      if (it->expired()) {
        // delete if it doesn't exist
        it = mObservers.erase( _Where: it);
      } else {
        // otherwise notify!
        it->lock()->onNotify();
        it++;
      }
    }
  }

private:
  std::list<std::weak_ptr<IObserver>> mObservers;
};
```

std::weak_ptr – тому що Subject не володіє спостерігачами.

«Підвислі» посилання - рішення

```cpp
void ISubject::addObserver(std::weak_ptr<IObserver> &&observer) {
  if (observer.lock()) {
    std::cout << "Adding " << observer.lock()->getName() << "\n";
    mObservers.push_front(_Val: std::move(observer));
  }
}

void ISubject::removeObserver(const std::weak_ptr<IObserver> &observer) {
  if (observer.lock()) {
    std::cout << "Removing " << observer.lock()->getName() << "\n";
    mObservers.remove_if(_Pred: [observer](const std::weak_ptr<IObserver> &ptr) {
      return !observer.owner_before(_Right: ptr) && !ptr.owner_before(_Right: observer);
    });
  }
}
```

«Підвислі» посилання - рішення

```
virtual void notify() {
  // go through list of observers
  for (auto std::list<std...r>>::iterator it = mObservers.begin(); it != mObservers.end();) {
    if (it->expired()) {
      // delete if it doesn't exist
      it = mObservers.erase( _Where: it);
    } else {
      // otherwise notify!
      it->lock()->onNotify();
      it++;
    }
  }
}
```

Перед сповіщенням маємо перевіряти, чи є кого сповіщати :)

«Підвислі» посилання - рішення

```
int main() {
  ISubject subject;
  std::shared_ptr<IObserver> observer1 =
      std::make_shared<LoggingObserver>(_Args: "observer1");
  std::shared_ptr<IObserver> observer2 =
      std::make_shared<SoundObserver>(_Args: "observer2");

  subject.addObserver(observer1);
  subject.addObserver(observer2);

  {
    std::shared_ptr<IObserver> observer3 =
        std::make_shared<PhysicsObserver>(_Args: "observer3 (i will be gone)");
    subject.addObserver(observer3);
    subject.notify();
  }

  subject.notify();

  subject.removeObserver(observer1);

  subject.notify();

  return 0;
}
```

Microsoft Visual Studio Debug Console

```
Adding observer1
Adding observer2
Adding observer3 (i will be gone)
observer3 (i will be gone) - physics!
observer2 - sound!
observer1 - logging!
observer2 - sound!
observer1 - logging!
Removing observer1
observer2 - sound!
```

«Підвислі» посилання - рішення

```
int main() {
  ISubject subject;
  std::shared_ptr<IObserver> observer1 =
      std::make_shared<LoggingObserver>(_Args: "observer1");
  std::shared_ptr<IObserver> observer2 =
      std::make_shared<SoundObserver>(_Args: "observer2");

  subject.addObserver(observer1);
  subject.addObserver(observer2);

  {
    std::shared_ptr<IObserver> observer3 =
        std::make_shared<PhysicsObserver>(_Args: "observer3 (i will be gone)");
    subject.addObserver(observer3);
    subject.notify();
  }

  subject.notify();

  subject.removeObserver(observer1);

  subject.notify();

  return 0;
}
```

Microsoft Visual Studio Debug Console

```
Adding observer1
Adding observer2
Adding observer3 (i will be gone)
observer3 (i will be gone) - physics!
observer2 - sound!
observer1 - logging!
observer2 - sound!
observer1 - logging!
Removing observer1
observer2 - sound!
```

«Підвислі» посилання - рішення

```
int main() {
  ISubject subject;
  std::shared_ptr<IObserver> observer1 =
      std::make_shared<LoggingObserver>(_Args: "observer1");
  std::shared_ptr<IObserver> observer2 =
      std::make_shared<SoundObserver>(_Args: "observer2");

  subject.addObserver(observer1);
  subject.addObserver(observer2);

  {
    std::shared_ptr<IObserver> observer3 =
        std::make_shared<PhysicsObserver>(_Args: "observer3 (i will be gone)");
    subject.addObserver(observer3);
    subject.notify();
  }

  subject.notify();

  subject.removeObserver(observer1);

  subject.notify();

  return 0;
}
```

Microsoft Visual Studio Debug Console

```
Adding observer1
Adding observer2
Adding observer3 (i will be gone)
observer3 (i will be gone) - physics!
observer2 - sound!
observer1 - logging!
observer2 - sound!
observer1 - logging!
Removing observer1
observer2 - sound!
```

observer3 перестає існувати за блоком

«Підвислі» посилання - рішення

```
int main() {
  ISubject subject;
  std::shared_ptr<IObserver> observer1 =
      std::make_shared<LoggingObserver>(_Args: "observer1");
  std::shared_ptr<IObserver> observer2 =
      std::make_shared<SoundObserver>(_Args: "observer2");

  subject.addObserver(observer1);
  subject.addObserver(observer2);

  {
    std::shared_ptr<IObserver> observer3 =
        std::make_shared<PhysicsObserver>(_Args: "observer3 (i will be gone)");
    subject.addObserver(observer3);
    subject.notify();
  }

  subject.notify();

  subject.removeObserver(observer1);

  subject.notify();

  return 0;
}
```

Microsoft Visual Studio Debug Console

```
Adding observer1
Adding observer2
Adding observer3 (i will be gone)
observer3 (i will be gone) - physics!
observer2 - sound!
observer1 - logging!
observer2 - sound!
observer1 - logging!
Removing observer1
observer2 - sound!
```

Забезпечення інваріанту перед сповіщенням

```
void MySubject::Operation(int newValue) {
    BaseClassSubject::Operation(newValue);
    // trigger notification

    _myInstVar += newValue;
    // update subclass state (too late!)
}
```

Змінюємо стан предка, він запускає сповіщення

Змінюємо свій стан, але усі вже сповіщені ☹

Рішення:

Template Method:
для оновлення стану у нащадках

```
void Text::Cut(TextRange r) {
    ReplaceRange(r);        // redefined in subclasses
    Notify();
}
```

Власне, сповіщення

Моделі Push та Pull

Push

Детальна інформація у сповіщенні

```
template<typename MessageInfo>
class Observer {
public:
    virtual ~Observer() {}
    virtual void onNotify(MessageInfo) = 0;
};
```

Pull

Мінімальна інформація у сповіщенні

```
class Observer {
private:
    Subject* m_subject;
public:
    virtual ~Observer() {}
    virtual void onNotify() {
        // look for changes
        m_subject->getState();
    }
};
```

або

```
class Observer {
public:
    virtual ~Observer() {}
    virtual void onNotify(Subject& subject) {
        // look for changes
        subject.getState();
    }
};
```

Типізація сповіщень

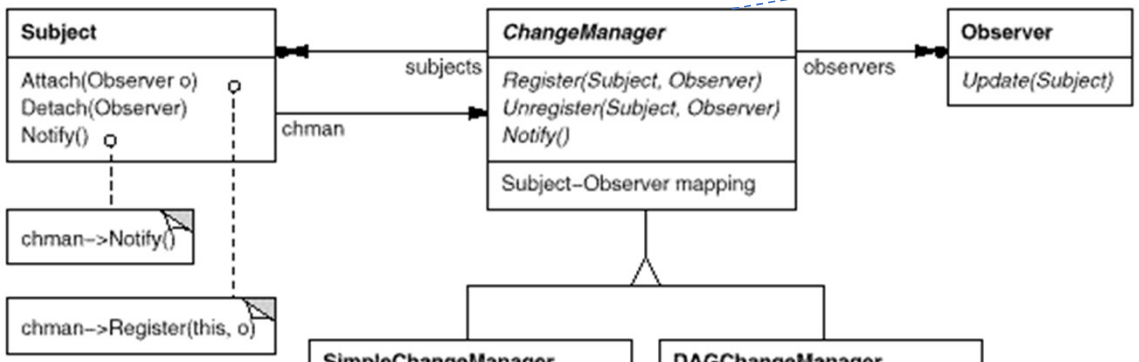
```
template <typename Aspect>
class Subject {
public:
    void attach(Observer<Aspect>*, Aspect&);
};

template <typename Aspect>
class Observer {
public:
    virtual void onNotify(Subject*, Aspect&) = 0;
};
```

Тепер ми можемо підписуватись лише на деякий тип сповіщень

Шаблон спостерігача має бути інстанційований тим же параметром, що і видавця

Інкапсуляція складної семантики оновлення



ChangeManager – **Mediator:**

- Містить мапу Subject-Observers
- Надає інтерфейс для реєстрації та оповіщення спостерігачів
- визначає певну стратегію оновлення
- оновлює всіх залежних спостерігачів за запитом видавця

Зазвичай нам потрібен один такий менеджер, тому – **Singleton**!

Нащадки визначають, як саме будуть оновлюватись спостерігачі

Поєднання інтерфейсів Observer та Subject

- Спостерігач тепер може виступати у ролі видавця
- Вирішує проблему, коли мова не підтримує множинне спадкування

CRTP - Curiously recurring template pattern

Статичний поліморфізм [ред. | ред. код]

```
#include <iostream>
using namespace std;

template<typename Derived>
struct Base
{
    void foo()
    {
        static_cast<Derived*>(this)->foo();
    }
};
```

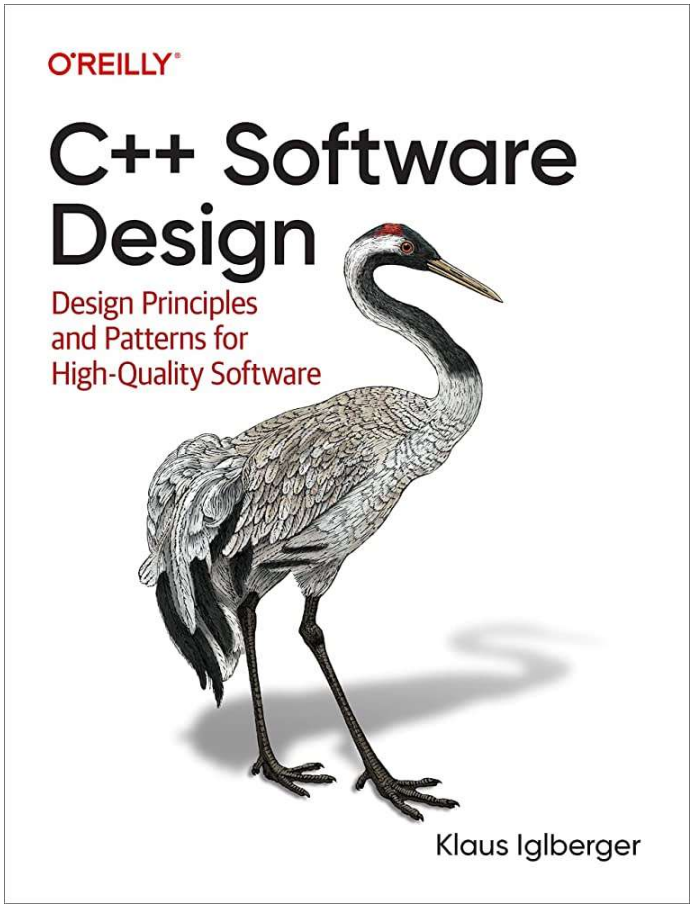
```
struct A : Base<A>
{
    void foo()
    {
        cout << "A::foo();" << endl;
    }
};

struct B : Base<B>
{
    void foo()
    {
        cout << "B::foo();" << endl;
    }
};
```

Wikipedia:

Дивно рекурсивний шаблон (англ. curiously recurring template pattern (CRTP)) — це підхід в мові програмування C++, в якому клас X є похідним від шаблону класу, інстанційованого із використанням самого X як шаблонного аргументу. Ім'я цього підходу було винайдене Джимом Копліном, який розглянув його в одному з найперших шаблонних кодів на C++.

Modern C++: CRTP + std::function



Modern C++: CRTP + std::function

Код: https://github.com/igl42/cpp_software_design Проект: observer-modern-cpp

```cpp
// ---- <Observer.h> ----------------------------------------------------
#include <functional>
#include <exception>

template<class Subject, typename StateTag>
class Observer
{
public:
    using NotifyCallback = std::function<void(const Subject&, StateTag)>;

    explicit Observer(NotifyCallback onUpdate)
        : m_onNotify(std::move(onUpdate)) {
        if (!m_onNotify) {
            throw std::exception("You need to provide non-empty function.");
        }
    }

    void onNotify(const Subject& subject, StateTag property)
    {
        m_onNotify(subject, property);
    }

private:
    NotifyCallback m_onNotify;
};
```

Хочемо отримувати тип повідомлення від конкретного видавця (**pull model**)

Використовуємо лямбди для зручності (не потрібно щоразу створювати новий клас)

Викликаємо лямбду у onNotify()

Modern C++: CRTP + std::function

Код: https://github.com/igl42/cpp_software_design

```cpp
// ---- <Subject.h> ----------------------------------------------------
#include "Observer.h"
#include <list>

template<class Derived, typename StateTag>
class Subject {
public:

    void attach(Observer<Derived, StateTag>* observer) {
        m_observers.push_back(observer);
    }

    void detach(Observer<Derived, StateTag>* observer) {
        m_observers.remove(observer);
    }

    void notify(StateTag property) {
        for (auto o : m_observers) {
            o->onNotify(*static_cast<Derived*>(this), property);
        }
    }

private:
    std::list<Observer<Derived, StateTag>*> m_observers;
};
```

CRTP! Визначаємо, якого конкретного видавця передавати у onNotify(…)

Modern C++: CRTP + std::function

Код: https://github.com/igl42/cpp_software_design

```cpp
// ---- <Person.h> ----------------------------------------------------
#include "Subject.h"
#include "Observer.h"
#include <string>

enum class StateChange
{
    forenameChanged,
    surnameChanged,
    addressChanged
};

class Person : public Subject<Person, StateChange> {
public:
    using PersonObserver = Observer<Person, StateChange>;

    explicit Person(std::string forename, std::string surname)

    // ...
```

CRTP! Перевикористання реалізації видавця

Створюємо спостерігача, використовуючи лямбду

Стягуємо поточний стан видавця

```cpp
PersonObserver fornameObserver([](const Person& person, StateChange state) {
        if (state == StateChange::forenameChanged) {
            std::cout << "[forename] Forename changed: " << person.forename() << "\n";
        }
    }
);
```

Багатопоточність

Які проблеми потенційно виникають?

- Атомічне додавання/видалення спостерігачів
- Взаємодія з видавцем під час оповіщення (виконання методу notify())
- Оновлення стану спостерігачів через pull model
- І т. д. …

Tony Van Eerd - Well-known speaker, Member of C++ ISO Standards Committee

Thread-safe Observer Pattern - You're doing it wrong

- Розглядаються загальні проблеми у реалізаціях взірця
- Thread-safe реалізація

Багатопоточність Проект: observer-threadsafe

Код: https://github.com/boostcon/cppnow_presentations_2016/tree/master/01_wednesday/thread_safe_observer_pattern_youre_doing_it_wrong

```
void set(std::string str) {
    LockGuard lock(mu);
    if (s != str) {
        s = str;
        notifyListeners();
    }
}

std::string get() {
    LockGuard lock(mu);  return s;
}
```

Блокування на зміні стану

Запускаємо сповіщення

```
void addListener(Listener* listener) {
    LockGuard lock(mu);
    listeners.push_back(_Val: listener);
}

bool removeListener(Listener* listener) {
    LockGuard lock(mu);
    for (auto std::vector<S... *>::iterator it = listeners.end(); it != listeners.begin(); )
    {
        it--;
        if (*it == listener) {
            listeners.erase(_Where: it);
            return true;
        }
    }
    return false;
}
```

Блокування на додаванні/видаленні спостерігачів

Багатопоточність

Код: https://github.com/boostcon/cppnow_presentations_2016/tree/master/01_wednesday/thread_safe_observer_pattern_youre_doing_it_wrong

```
void notifyListeners() {
    LockGuard lock(mu);
    for (auto Subject::Listener * listener : listeners)
        listener->informSubjectChanged( s: get());
}

std::vector<Listener* > listeners;
```

Блокування перед сповіщенням

Передаємо стан (push model)

Додаткове блокування на стороні спостерігача (можна уникнути з неблокуючим контейнером)

Зазвичай спостерігач триматиме історію оновлень у виді черги

```
    void informSubjectChanged(std::string s)
    {
        {
            std::lock_guard<std::mutex> lock(mu);
            // push onto a queue usually
            info = s;
            newData = true;
        }
        condvar.notify_all();
    }

private:
    // info can be a queue of events
    std::string info;
```

Багатопоточність

Код: https://github.com/boostcon/cppnow_presentations_2016/tree/master/01_wednesday/thread_safe_observer_pattern_youre_doing_it_wrong

```
void informSubjectChanged(std::string s)
{
    {
        std::lock_guard<std::mutex> lock(mu);
        // push onto a queue usually
        info = s;
        newData = true;
    }
    condvar.notify_all();
}
```

Перевіряємо один раз, чи немає змін

```
void check_for_change()
{
    std::unique_lock<std::mutex> lock(mu);
    if (newData) {
        newData = false;
        std::string temp = info;
        lock.unlock();
        onNotify( s: temp);
    }
}
```

Багатопоточність

Код: https://github.com/boostcon/cppnow_presentations_2016/tree/master/01_wednesday/thread_safe_observer_pattern_youre_doing_it_wrong

Виділяємо потік на оновлення змін

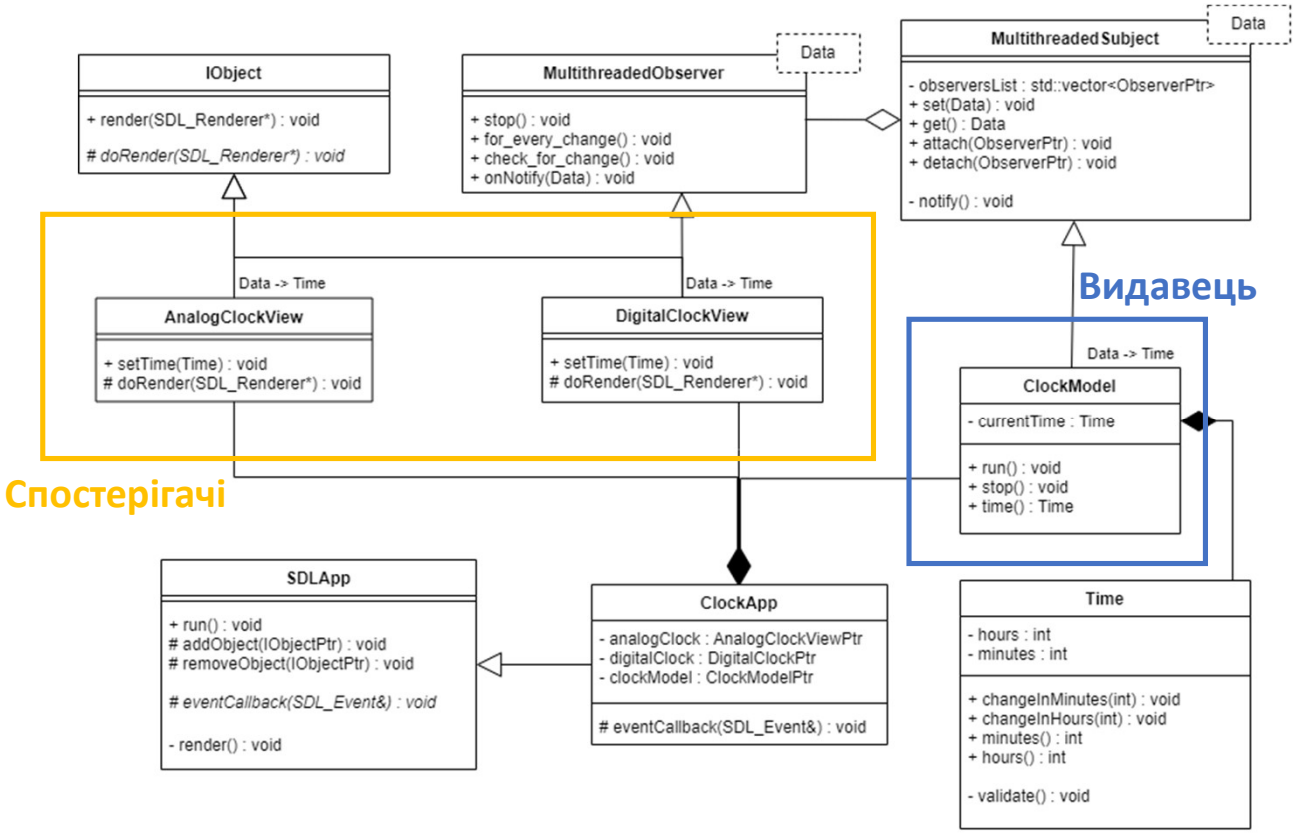
```
void informSubjectChanged(std::string s)
{
    {
        std::lock_guard<std::mutex> lock(mu);
        // push onto a queue usually
        info = s;
        newData = true;
    }
    condvar.notify_all();
}
```

```
void stop() {
    keepGoing = false; condvar.notify_all();
}

void for_every_change()
{
    keepGoing = true;
    while (keepGoing)
    {
        std::unique_lock<std::mutex> lock(mu);
        condvar.wait(& _Lck: lock, _Pred: [&] { return newData || !keepGoing; });
        newData = false; // no more data
        std::string temp = info;
        lock.unlock();
        onNotify(s: temp);
    }
}
```

Проект

Проект: observer-gui

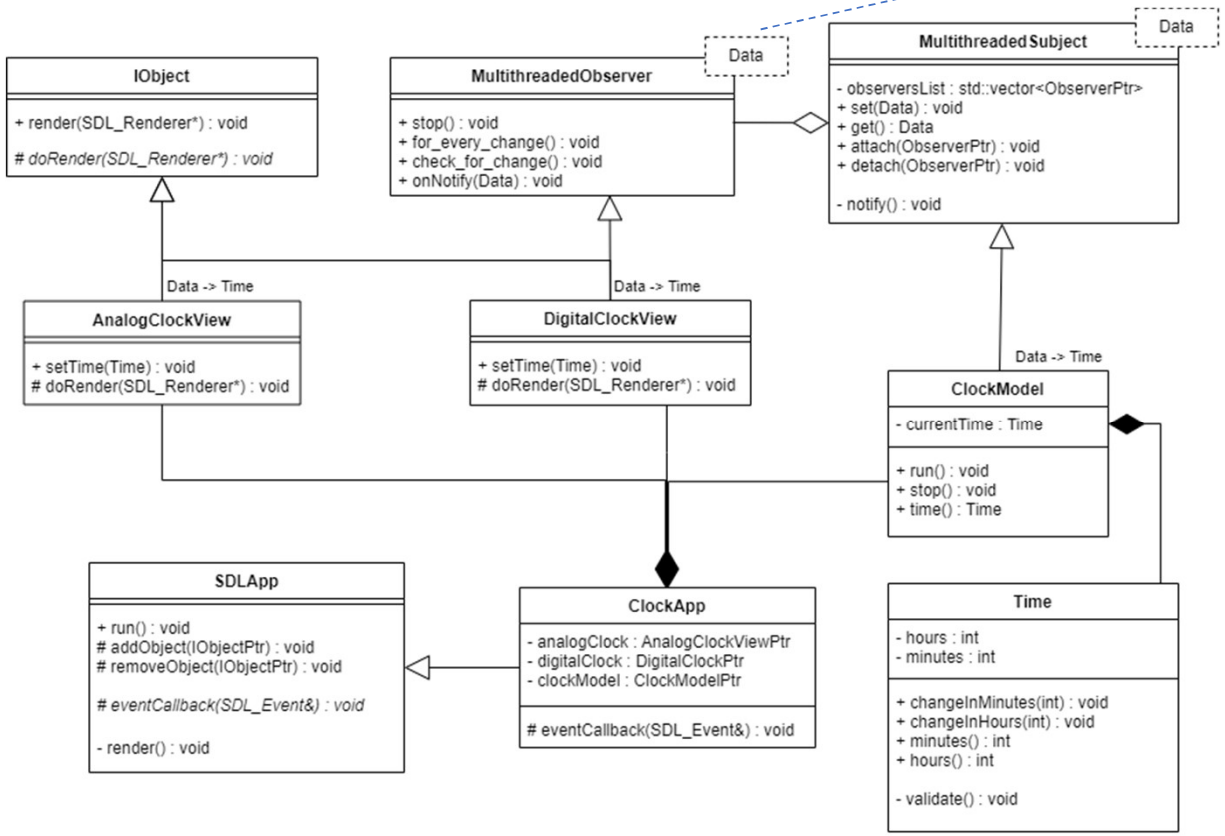


На початку ініціалізуємо усі елементи та реєструємо усі елементи на оновлення моделі

```
m_clockModel->attach(observer: m_digitalClock);
m_clockModel->attach(observer: m_analogClock);

std::thread timerThread([&]() { m_clockModel->run(); });
timerThread.detach();
```

Проект



Ці класи – поєднання технік CRTP та багатопоточної

Модель виконання:
UI thread:
- перевірка подій
- рендер

Timer thread:
- Оновлення моделі

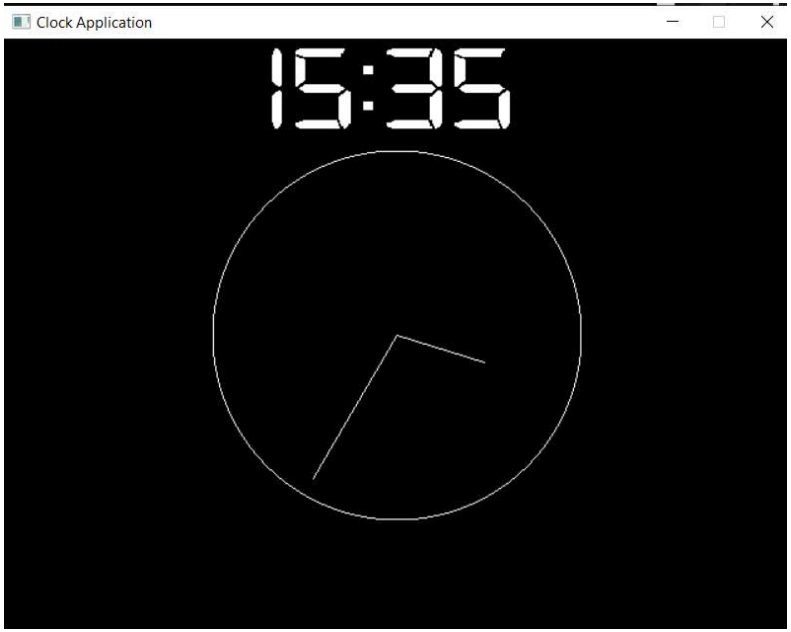
```
void run() {
    while (m_running) {
        checkEvents();
        render();

        SDL_Delay(ms: 1000 / FPS);
    }
}
```

SDLApp.h

Проект

Для наочності елементи на екрані динамічно від’єднуються від моделі при натисканні кнопок 1 та 2.




```
void processKeyPressed(SDL_Scancode& key) {
    if (key == SDL_SCANCODE_1) {
        if (m_digitalAttached) {
            m_clockModel->detach( observer: m_digitalClock);
            std::cout << "Digital clock detached!\n";
        }
        else {
            m_clockModel->attach( observer: m_digitalClock);
            std::cout << "Digital clock attached!\n";
        }
        m_digitalAttached = !m_digitalAttached;
    } else if (key == SDL_SCANCODE_2) {
        if (m_analogAttached) {
            m_clockModel->detach( observer: m_analogClock);
            std::cout << "Analog clock detached!\n";
        }
        else {
            m_clockModel->attach( observer: m_analogClock);
            std::cout << "Analog clock attached!\n";
        }
        m_analogAttached = !m_analogAttached;
    }
}
```

ClockApp.h

Застосування Спостерігача

Використовуйте цей взірець, коли:

Абстракція роздільна на два аспекти

- Один аспект залежить від іншого
- Інкапсуляція цих аспектів в окремі класи дозволяє варіювати та первикористовувати їх незалежно

Зміна у одному об’єкті вимагає зміни багатьох інших

- Ви заздалегідь не знаєте, скільки об’єктів треба оновити
- Ви хочете динамічно їх реєструвати на оновлення під час виконання програми

Об’єкт повинен мати можливість сповіщати інші об’єкти

- Ви не хочете, щоб вони були сильно зв’язні

MVC & MVVM

Переваги/Недоліки Спостерігача

Open/Closed принцип

- Можна додавати зміни у похідних класах, не змінюючи вже написаний код

Динамічний зв’язок

- Можна реєструвати та видаляти спостерігачів у run-time

Inversion of Control + гнучкість

- Можемо визначити будь-який тип повідомлень
- Є вибір серед різних моделей взаємодії між спостерігачем та видавцем

Випадковий порядок оповіщення

- Залежить від реалізації

Важкість у реалізації

- Прості варіанти хоч і гнучкі, але не дають гарантій працездатності

Зв’язок з іншими взірцями

Chain of Responsibility

- При реалізації зберігання спостерігачів як зв’язний список

Template Method

- Коли треба застосувати зміни у похідному класі і лише після цього запустити оновлення

Mediator

- Реалізація через ChangeManager
- Різниця з Observer:
  - Mediator – інкапсулює зв’язки, Observer – визначає динамічний зв’язок один до багатьох

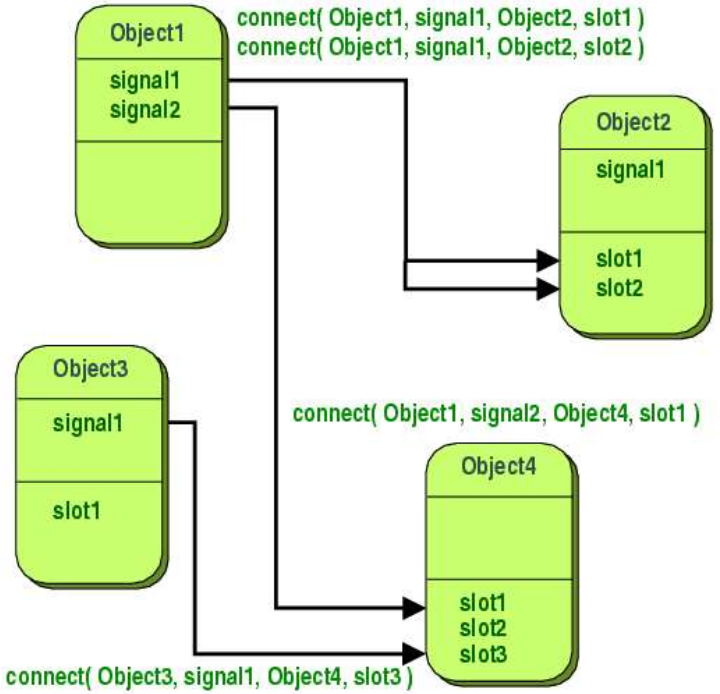
Singleton

- ChangeManager зазвичай буде єдиним на всю програму

Known uses

Signals and slots

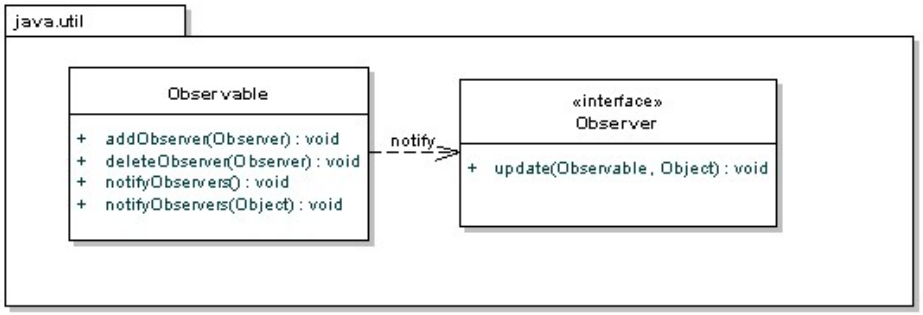
slot = observer
signal = observable / subject



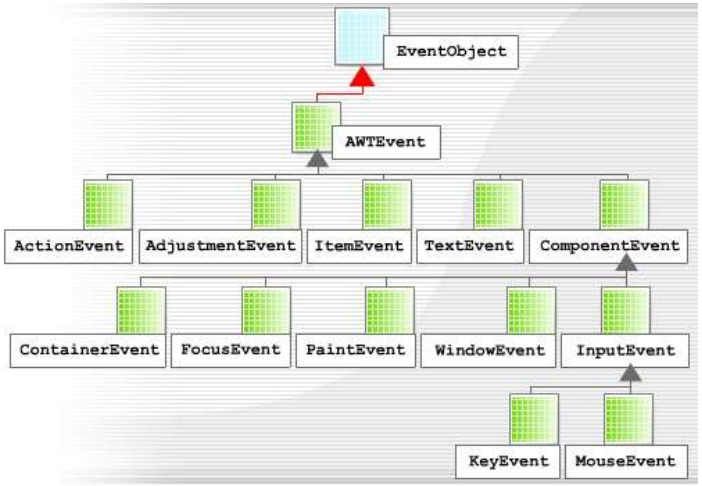
Порівняння: https://www.elpauer.org/stuff/a_deeper_look_at_signals_and_slots.pdf

Known uses

java.util.Observer



java.awt.event – асинхронний Observer



Known uses

LiveData – Android JetPack

- ViewModel надає об’єкт LiveData
- View додає до нього спостерігачів

- LiveData сповіщає, коли значення всередині змінюється
- Framework сам слідкує за життям об’єктів