Observer

Samuel Koribanič

Example

What is the expected delay?

I'll depart on time

What is the expected delay?

I'll depart on time

What is the expected delay?

I'll depart on time

What is the expected delay?

About 10 minutes

Example

Inform me about the delay

The delay has changed to _ minutes

Don't inform me anymore

Example

Inform me about the delay

The delay has changed to _ minutes

Don't inform me anymore

Observer

Subject

Subject.Attach(Observer);

Observer.Update(delay);

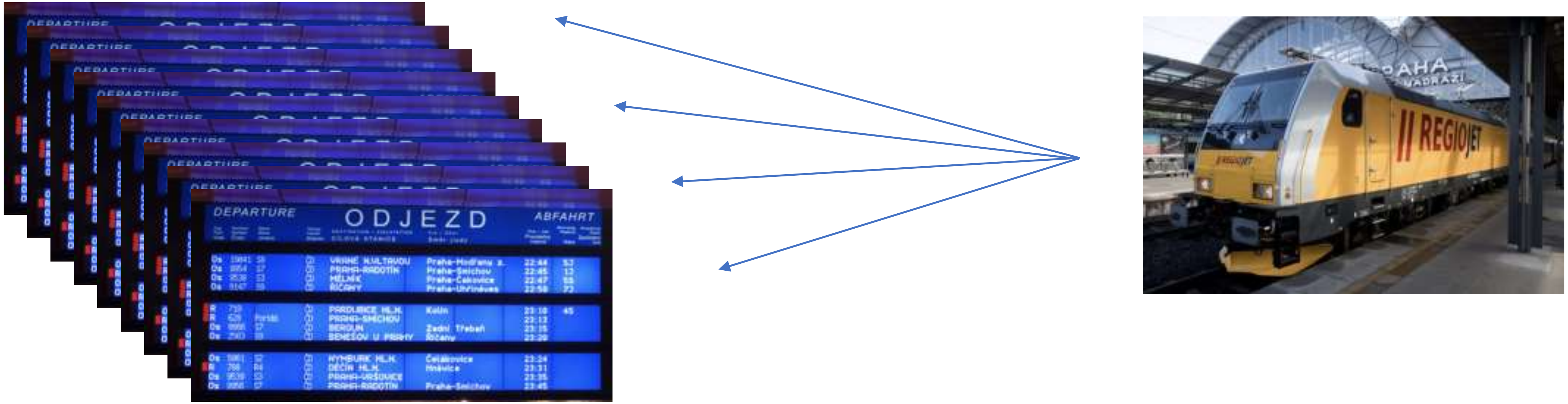
Subject.Detach(Observer);

Definition

- One(Subject / Publisher) to many(Observer/ Subscriber) dependency between objects
- When a Subject changes state, all its Observers are notified

Many Observers

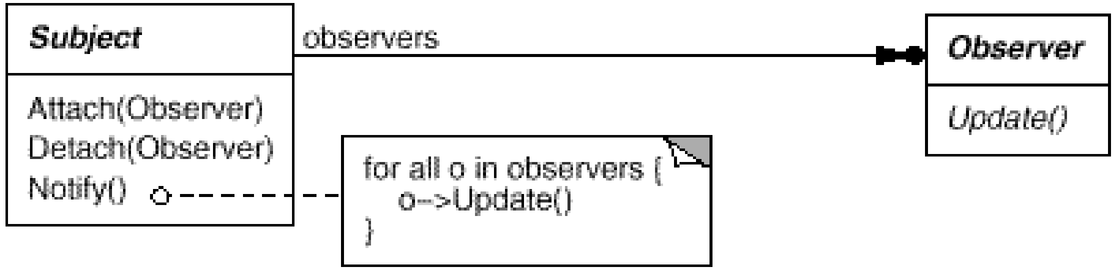
One Subject



Structure

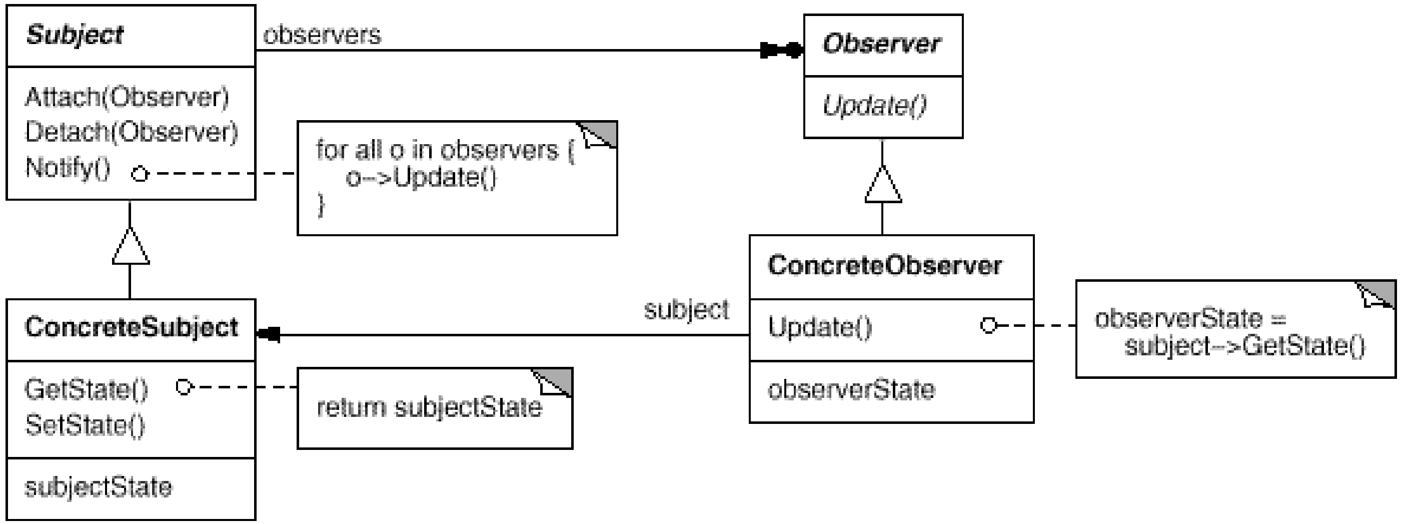
- Subject
  - Attaching and detaching interface
  - Stores references to observers

- Observer
  - Updating interface
  - It is notified when the Subject changes



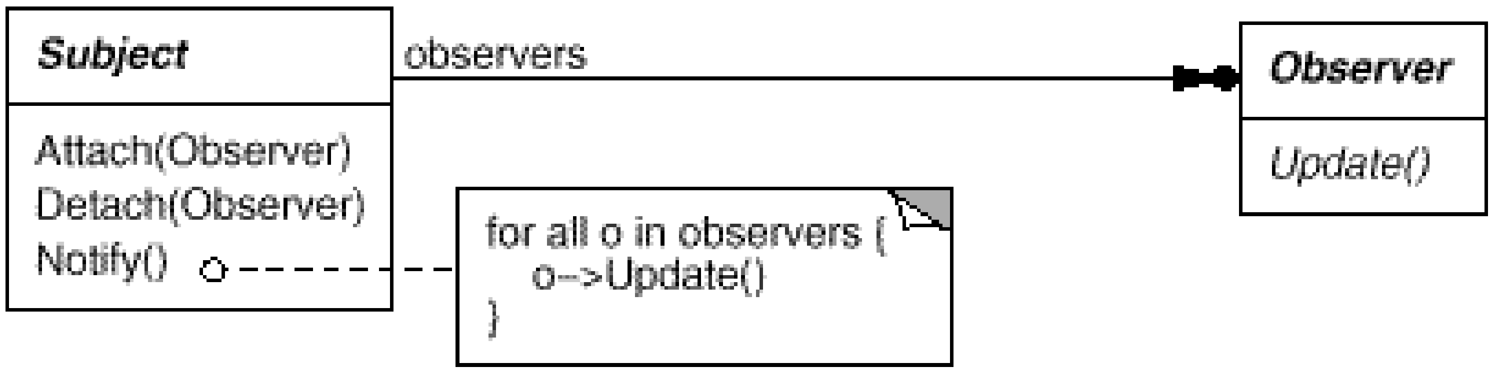
Structure

- ConcreteSubject
  - Stores a state
  - Notifies its observers
- ConcreteObserver
  - Implements updating interface
  - Has reference to a ConcreteSubject
  - Stored state should be consistent with the subject's



Consequences

- Abstract coupling between Subject and Observer
  - Subject only knows that observer implements Observer interface
- Support for broadcast communication
  - Receiver is not specified
  - Freedom to add and remove observers



Consequences

- Unexpected updates
  - operation on the subject may cause a cascade of updates to observers and their dependent objects
  - simple update protocol provides no details on what changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Observer ← Subject
Observer ← Subject
Observer ← Subject
← Subject

Implementation

- Who triggers the update?
  - Subject calls notify
    - Easy for the client
    - Potentially redundant updates

```
ConcreteSubject->setState("interest1", 100);  // calls Notify() inside
ConcreteSubject->setState("interest2", 200);  // calls Notify() inside
```

  - Client calls notify
    - Update can be triggered after a series of changes
    - Client has the responsibility to trigger the update

```
ConcreteSubject->setState("interest1", 100);
ConcreteSubject->setState("interest2", 200);
ConcreteSubject->Notify();
```

Implementation

- Making sure Subject state is self-consistent before notification
  - Can be violated by calling inherited operations

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

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

  - Use Template Method in abstract Subject classes

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

Implementation

- Specifying modifications of interest explicitly
  - To improve update efficiency

```
void Subject::Attach(Observer*, Aspect& interest);
void Observer::Update(Subject*, Aspect& interest);
```

```
void Attach(Observer* obs, std::string interested_in)
{
  attached_observers.push_back(
            std::make_pair(obs, interested_in));
}

void Notify(std::string what_changed) {
  for (const auto& obs : attached_observers)
    if(INTEREST == what_changed)
      OBSERVER->Update();
}
```

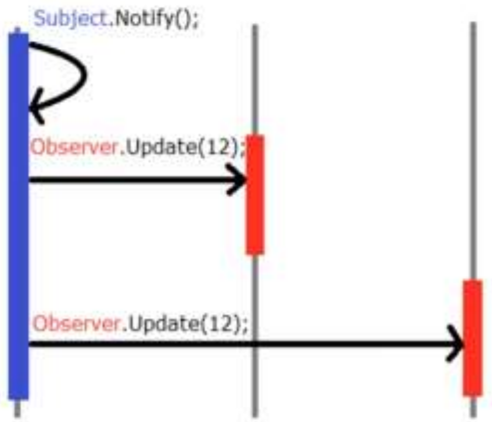
```
void Display::Update(std::string change)
{
  if (change == "temperature")
    observed_state.temperature =
              observed_subject->GetState().temperature;
  else
    return;
}
```

```
document.getElementById("myBtn").addEventListener("click", myFunction);
document.getElementById("myBtn").addEventListener("click", someOtherFunction);
```

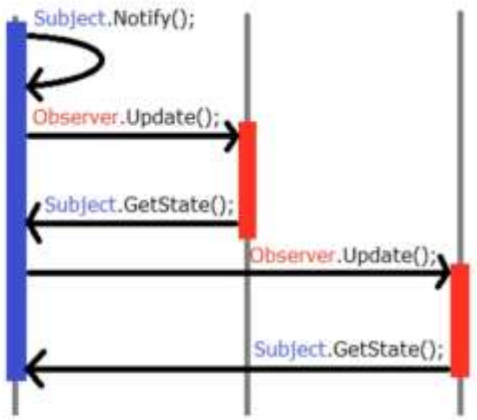
Implementation

- The push and pull models
  - Avoiding observer-specific update protocols
  - The push model assumes subjects know something about their observers' needs
  - The pull model emphasizes the subject's ignorance of its observers

Push



Pull



Implementation

- Observing more than one subject
  - // defines an updating interface for objects that should be notified of changes in a subject.

```
virtual void Update(Subject*, std::string what_changed) = 0;
```

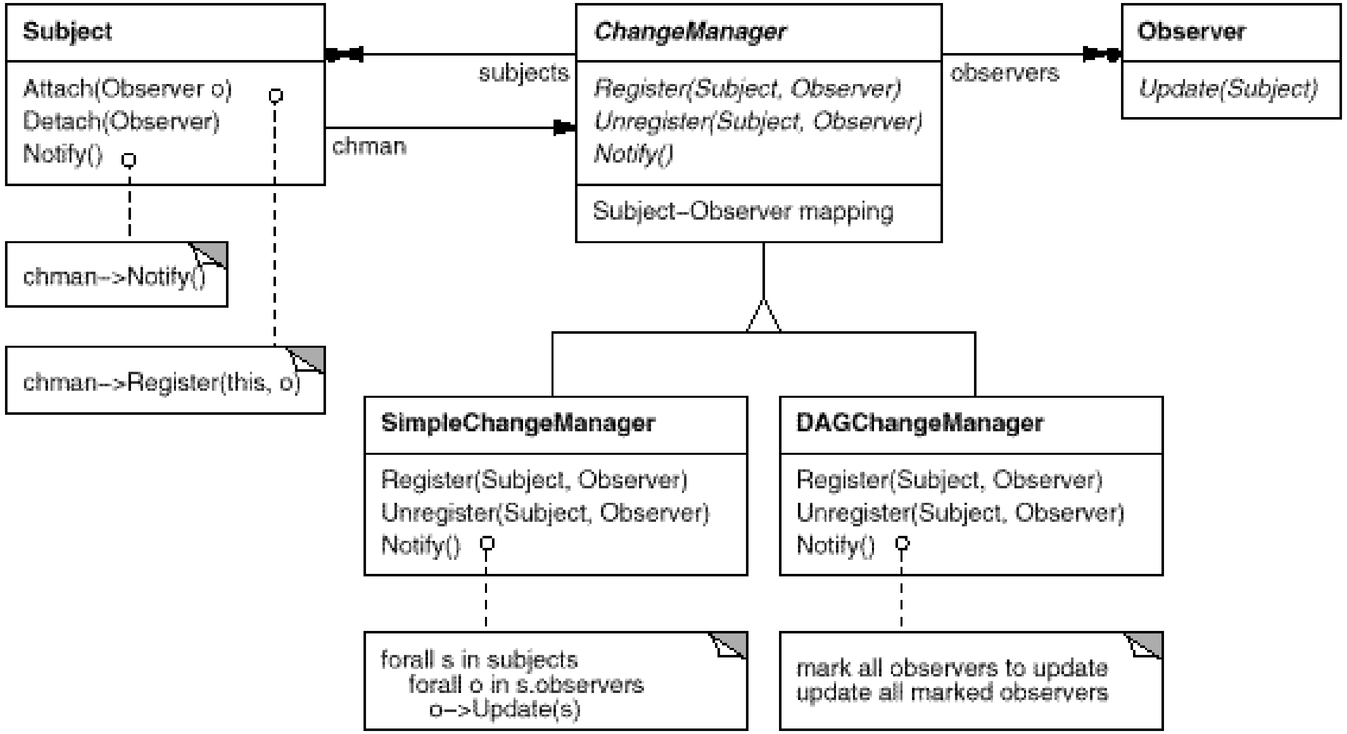
Subject

Observer

Subject

Implementation

- ChangeManager
  - Encapsulating complex update semantics
  - It maps a subject to its observers and provides an interface to maintain this mapping
  - It defines a particular update strategy
  - It updates all dependent observers at the request of a subject.



Implementation

- Mapping subjects to their observers
  - Store references explicitly in the subject or use associative look-up to maintain the subject-to-observer mapping
- Dangling references to deleted subjects
  - The observer need to notify the subject when deleted
- Cycles

  Subject → Observer

  Observer ← Subject

- Order of the updates is not guaranteed

Example of the implementation

```cpp
class Subject; // Subject declaration

class Observer {
public:
  // defines an updating interface for objects that should be notified of changes in a subject.
  virtual void Update(Subject*, std::string what_changed) = 0;
  virtual ~Observer() = default;
};

class Subject { // Subject definition
protected:
  std::list<Observer*> observers;
public:
  Subject() :observers() {}

  // knows its observers. Any number of Observer objects may observe a subject.
  void Notify(std::string what_changed) {
    for (const auto& obs:observers)
      obs->Update(this, what_changed);
  }

  // provides an interface for attaching and detaching Observer objects.
  void Attach(Observer* observer) {
    observers.emplace_back(observer);
  }
  void Detach(Observer* observer) {
    observers.remove(observer);
  }
};
```

std::string is used as the *Aspect*

Example of the implementation

ConcreteSubject

```cpp
class DelayManagement: public Subject {
public:
  DelayManagement() :subjectState() {}

  int getSpecificState(const std::string& train) {
    return subjectState[train];  // yes, this is stupid
  }

  auto getState() {
    return subjectState;
  }

  void setState(const std::string& what_changed, int delay) {
    subjectState[what_changed] = delay;
    Notify(what_changed);
  }
private:
// stores state of interest to DepartureBoard objects.
  std::unordered_map<std::string, int> subjectState;
};
```

For the simplicity of the example, Notify() is called inside the state-setting operation

Example of the implementation

ConcreteObserver

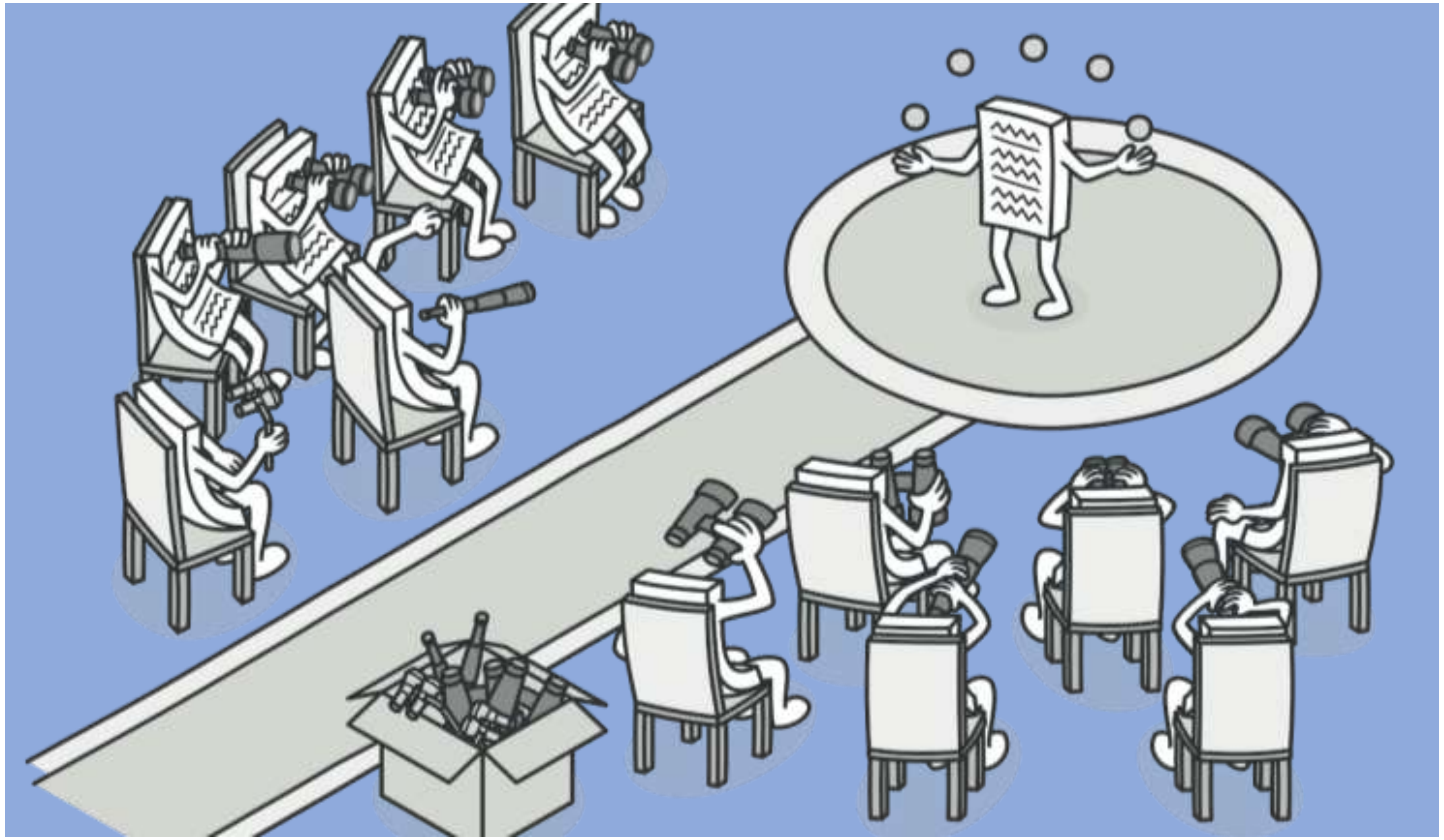
```
class DepartureBoard: public Observer {
public:
  DepartureBoard(std::shared_ptr<DelayManagement> DelayManagement_)
    :DelayManagement(DelayManagement_.get()) {
    DelayManagement->Attach(this);
    observerState = DelayManagement->getState();

  }
  // implements the Observer updating interface to keep its state consistent with the subject's.
  void Update(Subject* subject, std::string what_changed) {
    if (subject == DelayManagement)
      observerState[what_changed] = DelayManagement->getSpecificState(what_changed);
  }
  virtual ~DepartureBoard() {
    DelayManagement->Detach(this);
  }
private:
  // stores state that should stay consistent with the subject's.
  std::unordered_map<std::string, int> observerState;
  // maintains a reference to a DelayManagement object.
  DelayManagement* DelayManagement;
};
```

Known uses

- GUI
  - Event listeners
  - Multiple visualizations of the same data
  - C++ GUI library wxWidgets
- Client – Server applications
  - Group chat
  - YouTube notifications

Related patterns

- Mediator
  - Observer pattern focuses on the relationship between a subject and its observers, while the Mediator pattern focuses on interactions between multiple objects
  - The communication in the Mediator can be implemented using Observer pattern
    - The Mediator is an Observer of multiple Colleagues that act as Subjects
  - Also, the ChangeManager is a mediator between subjects and observers
- Singleton
  - ChangeManager may use this to make it unique and globally accessible



THANK YOU FOR YOUR ATTENTION