Critical code analysis with Observer pattern

• Adam Badura
• 15-11-2016
Agenda

1. Brief introduction to the pattern
2. Analysis of typical implementations
3. Conclusions from the implementations
4. Sneak peek at more complex aspects
5. Summary
The Observer pattern
Comes under many names

1. Observer/Observable (GOF, Java)
2. Listener & Events (Java)
3. Publish-Subscribe
4. Signals/Slots (Boost, Qt)
The Observer pattern
Gang of Four approach
The Observer pattern
Varies in implementations

- Single-threaded / Multithreaded
- Synchronous / Asynchronous
- Local / Distributed
- Inheritance / Composition
The Observer pattern
Varies in features

• Attaching and detaching
  – Attaching more than once
The Observer pattern
Varies in features

• Attaching and detaching
  – Attaching more than once
  – Detaching based on token, identity, ...
The Observer pattern
Varies in features

• Attaching and detaching
  – Attaching more than once
  – Detaching based on token, identity, ...
  – Dealing with destroyed observers
The Observer pattern
Varies in features

• Attaching and detaching
  – Attaching more than once
  – Detaching based on token, identity, ...
  – Dealing with destroyed observers

• Notifying
  – Order
The Observer pattern
Varies in features

• Attaching and detaching
  – Attaching more than once
  – Detaching based on token, identity, ...
  – Dealing with destroyed observers

• Notifying
  – Order
  – Observers attached while notifying
The Observer pattern
Varies in features

• Attaching and detaching
  – Attaching more than once
  – Detaching based on token, identity, ...
  – Dealing with destroyed observers

• Notifying
  – Order
  – Observers attached while notifying

• Update
  – With source
The Observer pattern
Varies in features

• Attaching and detaching
  – Attaching more than once
  – Detaching based on token, identity, ...
  – Dealing with destroyed observers

• Notifying
  – Order
  – Observers attached while notifying

• Update
  – With source
  – With additional data
Implementation – 1st attempt
Classes

class observable {
public:
  virtual ~observable() = default;
  void attach(observer* o);
  void detach(observer* o);
protected:
  void notify();
private:
  std::vector<observer*> obss;
};
class observer {
public:
  virtual ~observer() = default;
  virtual void update() = 0;
};
Implementation – 1st attempt

Methods

```
void notify(){
  std::for_each(
    obss.cbegin(), obss.cend(),
    [](observer* o){
      o->update();
    });
}

void attach(observer* o){
  obss.push_back(o);
}

void detach(observer* o){
  obss.erase(
    std::find(
      obss.cbegin(), obss.cend(), o
    )
  );
}
```
Implementation – 1st attempt

The bug

```cpp
void update() {
  some_observable.attach(any_observer);
}
```
Implementation – 2nd attempt

Methods

```cpp
void notify() {
    auto const temp = obss;
    std::for_each(temp.cbegin(), temp.cend(), [] (observer* o) {
        o->update();
    });
}
```

Prevent iterators invalidation while in `notify` by iterating over a local copy.

*attach* and *detach* no longer affect the container that is being iterated!
Implementation – 2nd attempt

The bug

```cpp
void update()
{
    some_observable.detach(next_observer);
}
```

1. `notify()`
2. `update()`
3. `detach(next_observer)`
4. `erase from obs` (not shown)
5. `...still somewhere in std::for_each...`
6. `++it`
7. `All is fine since it was the obs rather than local temp that was changed.`
8. `update()`

However, `next_observer` was already detached and did not expect a call to `update`.
Implementation – 3rd attempt

Classes

class observable {
public:
    virtual ~observable() = default;
    void attach(observer* o);
    void detach(observer* o);
    void notify();
private:
    std::vector<observer*> obss;
    std::vector<observer*> temp;
};

Then why not just have the temp as member variable and erase from it in detach but not add to it in attach?

This way temp will not reallocate and invalidate our iterators.
Implementation – 3rd attempt

Methods

```cpp
void notify()
{
    temp = obss;
    std::for_each(
        temp.cbegin(), temp.cend(),
        [](observer* o)
        {
            o->update();
        });
}
```

```cpp
void detach(observer* o)
{
    auto const obss_it = std::find(
        obss.cbegin(), obss.cend(), o);
    auto const remove_index = obss_it - obss.cbegin();
    obss.erase(obss_it);
    temp.erase(
        temp.cbegin() + remove_index);
}
```
Implementation – 3rd attempt
The bug

```cpp
void update() {
    some_observable.detach(any_observer);
}
```
Implementation – 4th attempt

Methods

```cpp
void notify()
{
    for (decltype (obss.size()) i = 0; i < obss.size(); ++i)
    {
        obss[i]->update();
    }
}
```

Since iterators keep making problems let’s give up on them and use good old indexes.

No more `temp`, no more iterators, just indexes. They are never invalidated!

You can't go wrong with that – or can you?
Implementation – 4th attempt

The bug

```c
void update() {
  some.observable.detach(
    previous_observer);
}
```

Diagram:

- `notify()`: Notify the observer.
- `update()`: Update function.
- `detach(previous_observer)`:
  - After detach, `i` no longer points to `our_observer`:
    - Instead, it points to `observer` just after it.
    - Incrementing `i` will skip the observer from its update call.

```
...still somewhere in for loop...

1 notify()
2 update()
3
4 ++i
5 update()
6 detach(previous_observer)
7 erase from obs
8
9
10 ++i
...still somewhere in for loop...
```
Implementation – 5th attempt

Classes

class observable {

public:
  virtual ~observable() = default;
  void attach(observer* o);
  void detach(observer* o);
  void notify();

private:
  using vector_type = std::vector<observer*>;
  vector_type obss;
  vector_type::size_type notify_index;
};
Implementation – 5th attempt
Methods

```cpp
void notify()
{
  for (notify_index = 0;
       notify_index < obss.size();
       ++notify_index)
  {
    obss[notify_index]->update();
  }
}

void detach(observer* o)
{
  auto const it = std::find(obss.cbegin(), obss.cend(), o);
  auto const remove_index = it - obss.cbegin();
  obss.erase(it);
  if (notify_index >= remove_index)
  {
    --notify_index;
  }
}
```
Implementation – 5th attempt
Conclusions

• It finally works!
Implementation – 5th attempt
Conclusions

• It finally works!
• It would work with iterators as well
Implementation – 5th attempt

Conclusions

• It finally works!
• It would work with iterators as well

• list or set/multiset instead of vector would not change much
Implementation – 5th attempt

Conclusions

• It finally works!
• It would work with iterators as well
• list or set/multiset instead of vector would not change much
• Properties
  – Observers may attach multiple times
Implementation – 5th attempt
Conclusions

• It finally works!
• It would work with iterators as well
• list or set/multiset instead of vector would not change much

• Properties
  – Observers may attach multiple times
  – Detaching based on identity
Implementation – 5th attempt

Conclusions

• It finally works!
• It would work with iterators as well
• *list* or *set/multiset* instead of *vector* would not change much

• Properties
  – Observers may attach multiple times
  – Detaching based on identity
  – With no way to distinguish those multiple attachments
Implementation – 5\textsuperscript{th} attempt

Conclusions

• It finally works!
• It would work with iterators as well
• \texttt{list} or \texttt{set/multiset} instead of \texttt{vector} would not change much
• Properties
  – Observers may attach multiple times
  – Detaching based on identity
  – With no way to distinguish those multiple attachments
  – Observers added while notifying are updated as well
Implementation – 5th attempt

Conclusions

• It finally works!
• It would work with iterators as well
• list or set/multiset instead of vector would not change much
• Properties
  – Observers may attach multiple times
  – Detaching based on identity
  – With no way to distinguish those multiple attachments
  – Observers added while notifying are updated as well
• Quasi-multithreading
Other aspects

• Destruction of observable while notifying
Other aspects

- Destruction of **observable** while notifying
  - Boost.Signals2 uses pimpl-like approach
Other aspects

• Destruction of **observable** while notifying
  – Boost.Signals2 uses pimpl-like approach
  – Qt Signals requires `deleteLater`
Other aspects

• Destruction of **observable** while notifying
  – Boost.Signals2 uses pimpl-like approach
  – Qt Signals requires **deleteLater**

• Asynchronous updates
Other aspects

- Destruction of observable while notifying
  - Boost.Signals2 uses pimpl-like approach
  - Qt Signals requires `deleteLater`

- Asynchronous updates
  - Boost.Signals2 doesn’t support it
Other aspects

• Destruction of observable while notifying
  – Boost.Signals2 uses pimpl-like approach
  – Qt Signals requires `deleteLater`

• Asynchronous updates
  – Boost.Signals2 doesn’t support it
  – Qt Signals use event loops
Other aspects

• Destruction of **observable** while notifying
  – Boost.Signals2 uses pimpl-like approach
  – Qt Signals requires `deleteLater`

• Asynchronous updates
  – Boost.Signals2 doesn’t support it
  – Qt Signals use event loops

• Multithreading
Other aspects – Multithreading

By the optimist

/**
 * @todo Add thread safety.
 */

class observable
Other aspects – Multithreading
By the realist

/**
 * @warning Not thread safe!
 */

class observable
Best implementation

Avoid reinventing the wheel!
Summary

- Beware of quasi-multithreading
- Changing iterators to indexes doesn’t help much
- Neither does changing containers
- Avoid reinventing the wheel
Questions & Answers