

Spooky Action at a Distance



MUC++

December 14th 2021



@ciura_victor

Victor Ciura
Principal Engineer



I hate the term “Design Patterns”. It implies there are universally applicable solutions to some common code scenarios. Just codifying existing practice into some rules and blindly following them is a comfortable path, but not the optimal one. It turns out it’s not as easy as following recipes. Each situation and best associated solution is unique.

However there is value in having uniform code structure throughout a project. So this topic is not to be discarded just yet, rather it needs more careful examination.

In terms of inspectable properties of objects, what have we learned from years of OO influence from other languages and frameworks? How can we leverage these borrowed techniques in a value-oriented context? Does C++ benefit from special considerations?

I think it’s time to revisit our old friend, the Observer pattern - from “theory” to practice. I’m not going to offer The Solution, rather we’re going to examine tradeoffs for several possible implementations, in various usage scenarios from a real project.



Advanced Installer



Clang Power Tools

 **@ciura_victor**

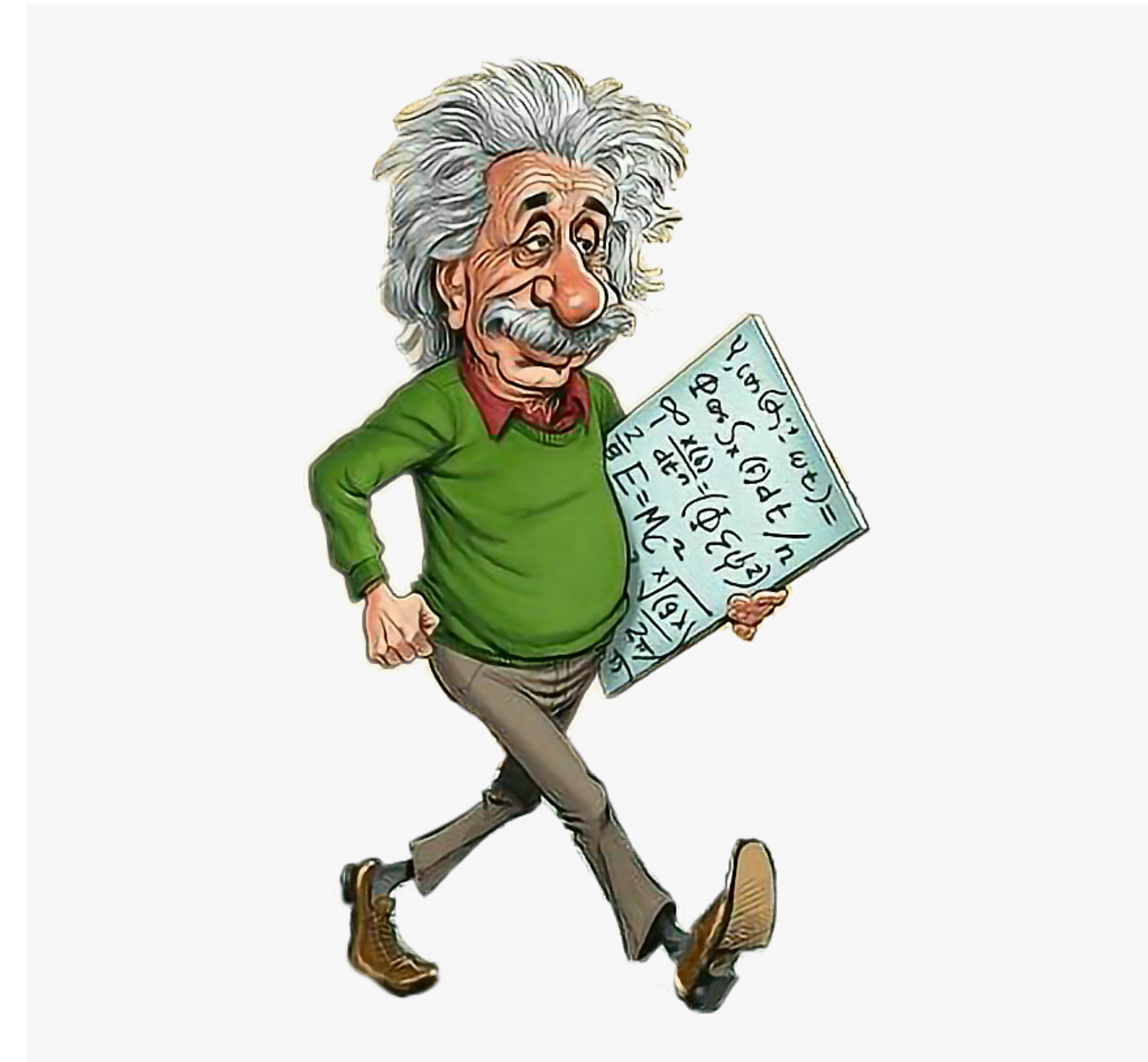
Q & A

Spooky Action at a Distance

Spooky What ?

Entangled particles

Quantum entanglement or "*spooky action at a distance*" as Albert Einstein famously called it, is the idea that the fates of tiny particles are linked to each other even if they're separated by long distances.



Revisiting Observers

Subscribe (Observer)

I hate the term “Design Patterns”

Design Patterns

It implies there are **universally** applicable solutions to some common code scenarios.

Just **codifying** existing practice into some **rules** and blindly following them is a comfortable path, but not the optimal one.

It turns out it's not as easy as following **recipes**.

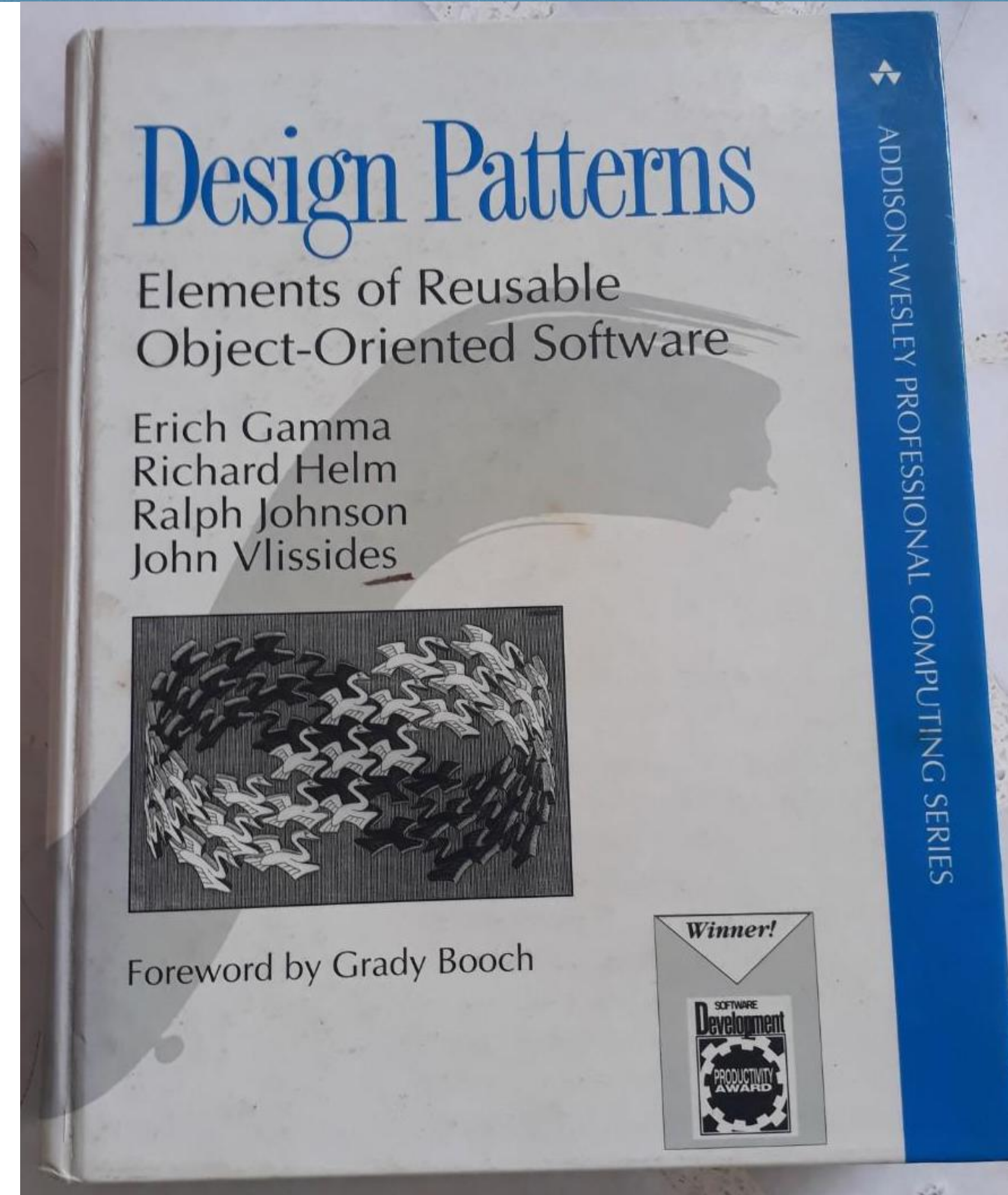
Each situation and best associated solution is **unique**.

However there is value in having **uniform** code structure throughout a project.

So this topic is not to be discarded just yet, rather it needs more **careful examination**.

A classic

Too formal & dry

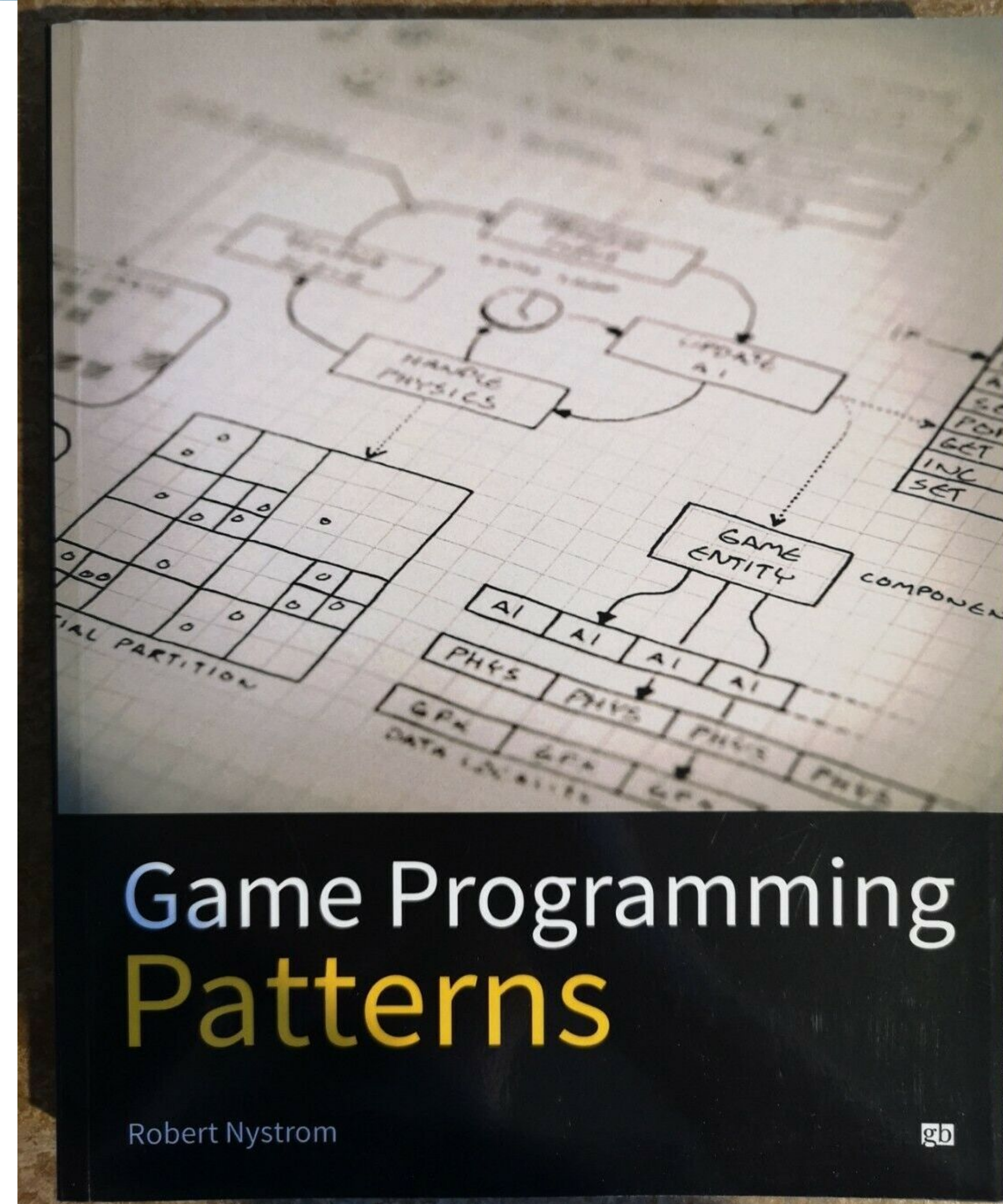


Game Programming Patterns




Bob Nystrom


gameprogrammingpatterns.com



amazon.com/Game-Programming-Patterns-Robert-Nystrom/dp/0990582906/

 **Cppcon** | The C++ Conference


Design Patterns: Facts and Misconceptions


Klaus Iglberger

- Architecture**
 - How are big entities depending on each other?
 - Design decisions that are hard to change
 - Architectural patterns
 - Examples:
 - Client-Server Architecture
 - Micro-Services
 - MVC, ...
- Design**
 - How are small entities depending on each other?
 - Design decisions that are easier to change
 - Design patterns
 - Examples:
 - GoF Patterns: Visitor, Strategy, Observer, ...
 - External Polymorphism
 - ...
- Implementation Details**
 - How is a design implemented?
 - Which features are used?
 - Implementation patterns
 - Examples:
 - new, malloc, ...
 - class vs. struct, lambda, ...
 - ...

Idioms

- NVI Idiom (Template Method Design Pattern)
- Pimpl Idiom (Bridge Design Pattern)
- Temporary-Swap Idiom
- RAII Idiom
- enable_if
- Factory Function



20 | 21 | October 24-29 | 16:12 / 50:42

Design Patterns: Facts and Misconceptions - Klaus Iglberger - CppCon 2021

Klaus Iglberger - [Design Patterns: Facts and Misconceptions](#)

Design Patterns have proven to be useful over several decades and knowledge about them is still very important to design robust, decoupled systems. However, in recent decades a lot of misconceptions have piled up, many based on [misunderstandings](#) about software design in general and Design Patterns in particular.

This purpose of this talk is to help to separate [facts](#) from [misconceptions](#).

It explains what software design is, how Design Patterns fit in, and what an [idiom](#) is.

youtube.com/watch?v=KGX6zhOWGAc

Observer Pattern

In terms of **inspectable properties** of objects:

- What have we learned from years of **OO influence** from other languages and frameworks?
- How can we leverage these borrowed techniques in a **value-oriented** context?
- Does **C++** benefit from special considerations?

Observer Pattern

Let's revisit our old friend, the **Observer** pattern - from theory to practice.

I'm not going to offer **The Solution**TM

We're going to examine **tradeoffs** for several possible implementations, in various usage scenarios from a real project.

Observer Pattern

Observers are everywhere...

Think:

- MVC
- MVVM
- Qt signal-slot mechanism
- not just GUI ↔ model, also model ↔ model



Observer Pattern

It's a show with **Actors** and **Actions**

Subject/Actor doesn't know what (type) the **Observers** are.

It just knows that they exist and **how to notify** them when certain **actions** occur.

Low Coupling

Subscription Model

Tune-in to a particular radio station



Remote Objects

Inspectable properties and remote objects

"spooky action at a distance"

```
class Widget
{
    Data mData;

public:
    void Set(const Data & d) {
        if (d != mData) {
            mData = d;
            NotifyObservers();
        }
    }
};
```

Subscription Order

Observers added in a certain order.

Do they respond in the same order?

```
class Widget
{
    ... Salient Data

    std::vector<IObserver *> mObservers;
};
```

Subscribing

```
void Widget::AddObserver(IObserver & aObserver)
{
    // too simple, right?
    mObservers.push_back(&aObserver);
}
```

Over-subscribing

Adding an observer more than once?

```
void Widget::AddObserver(IObserver & aObserver)
{
    auto found = std::find(mObservers.begin(), mObservers.end(), &aObserver);

    if (found == mObservers.end())
        mObservers.push_back(&aObserver);
}
```

Do you want to allow an observer to subscribe more than once?

Do you expect the observer to be called twice for the same event?

Over-subscribing

What about **local** reasoning?

```
void Func()
{
    obj->AddObserver(*this);
    ... // do something important
    obj->RemoveObserver(*this); // what if this obs was already added before?
}
```

Over-subscribing

What about **local** reasoning?

```
void Func()
{
    RegisterObserver obs(*this, actor); // RAII remember if we added
    ... // do something important
    // ~RegisterObserver() removes *this from observers if we added in C-tor
}
```

Over-subscribing

Signal the caller if the registration was "successful"

```
bool Widget::AddObserver(IObserver & aObserver)
{
    auto found = std::find(mObservers.begin(), mObservers.end(), &aObserver);
    if (found != mObservers.end())
        return false; // observer was already registered

    mObservers.push_back(&aObserver);
    return true;
}
```

Over-subscribing

Adding an observer more than once?

```
void Widget::AddObserver(IObserver & aObserver)
{
    mObservers.push_back(&aObserver);
}
```

We expect the observer to be called twice for the same event.

Local reasoning - restricted lifetime.



Removing an observer not in the list (already removed?)

```
void Widget::RemoveObserver(IObserver & aObserver)
{
    auto found = std::find(mObservers.begin(), mObservers.end(), &aObserver);

    if (found != mObservers.end())
        mObservers.erase(found);
}
```

For **multiple** registration scenario, what if we remove the **wrong instance**?
(sensitive to **order** of notification)

Removing **all** instances of this observer (multiple registration)

```
void Widget::RemoveObserver(IObserver & aObserver)
{
    mObservers.erase(
        std::remove(mObservers.begin(), mObservers.end(), &aObserver),
        mObservers.end() );
}
```

Removing **all** instances of this observer (multiple registration)

```
void Widget::RemoveObserver(IObserver & aObserver)
{
    std::erase(mObservers, &aObserver); // C++20 safer than erase-remove idiom
}
```

Who should be notified first?

```
void Widget::AddObserver(IObserver & aObserver)
{
    auto found = std::find(mObservers.begin(), mObservers.end(), &aObserver);

    if (found == mObservers.end())
        mObservers.insert(mObservers.begin(), &aObserver);
}
```


Do we need priority buckets?

```
class Widget
{
    ... mSalientData;

    std::vector<IObserver *> mObserversRing0;
    std::vector<IObserver *> mObserversRing1;
    std::vector<IObserver *> mObserversRing2;
    ...
};
```



Do we need priority buckets?

```
void Widget::AddObserver(IObserver & aObserver, Priority p)
{
    ...

    // what happens if an observer is registered (by mistake)
    // with different priorities?
}
```


Broadcast

Notify all registered observers, in order:

```
void Widget::NotifyObservers()  
{  
    for (auto & observer : mObservers)  
        observer->WidgetChanged(this);  
}
```

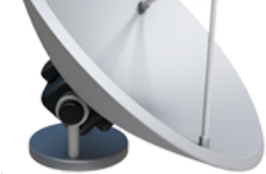


Tune-in and react to the event triggered by the actor:

```
void SomeObserver::WidgetChanged(Actor * sender)   
{  
    // react in some way to the changed object (actor)  
  
    ...  
}
```

Safe to deregister at any time?

What if an observer wants to **remove itself** after receiving a **notification**?


```
void SomeObserver::WidgetChanged(Actor * sender)   
{  
    ... // react in some way to the changed object (actor)  
    sender->RemoveObserver(*this); // WHAT?! don't care about future events  
}
```

Unsubscribe

Safe to deregister at any time?

What if an observer wants to **remove itself** after receiving a **notification**?

```
for (auto & observer : mObservers)  
    observer->WidgetChanged(this);
```

```
void SomeObserver::WidgetChanged(Actor * sender)   
{  
    ... // react in some way to the changed object (actor)  
    sender->RemoveObserver(*this); // WHAT?! don't care about future events  
}
```

```
std::erase(mObservers, &aObserver); 
```


How can we make *recursive remove* more **resilient**?

```
bool Widget::RemoveObserver(IObserver & aObserver)
{
    for(auto it = mObservers.begin(); it != mObservers.end(); ++it)
    {
        if (*it == &aObserver)
        {
            *it = nullptr; // replace observer with a sentinel
            return true;
        }
    }

    return false;
}
```

Broadcast

Notify all registered observers:

```
void Widget::NotifyObservers()  
{  
    for (auto & observer : mObservers)  
    {  
        if (observer)  
            observer->WidgetChanged(this);   
    }  
  
    std::erase(mObservers, nullptr); // deferred cleanup of removed observers  
}
```


Recursive add observer has the same problem, but it's more **rare** in practice.

Can small objects afford to have observers?

```
class SmallObject
{
    ... mSalientData;

    std::vector<IObserver *> mObservers;
};
```

Can small objects afford to have observers?

```
class SmallObject
{
    ... mSalientData;

    std::vector<IObserver *> mObservers;
};
```

What if some instances will never have a registered observer?

An **empty** `std::vector` is not tiny.

Small Objects

Small objects can be register observers lazily

```
class SmallObject
{
    ... mSalientData;

    LazyVector<IObserver *> mObservers;
};
```

We can use an indirection to "*fault-in*" the `std::vector` creation when first needed:

```
operator*()
operator->()
```

```
{
    if (mPtr == nullptr)
        mPtr = new std::vector<Type>();
    return mPtr;
}
```

Lots of Objects

What if we have lots of these small objects?

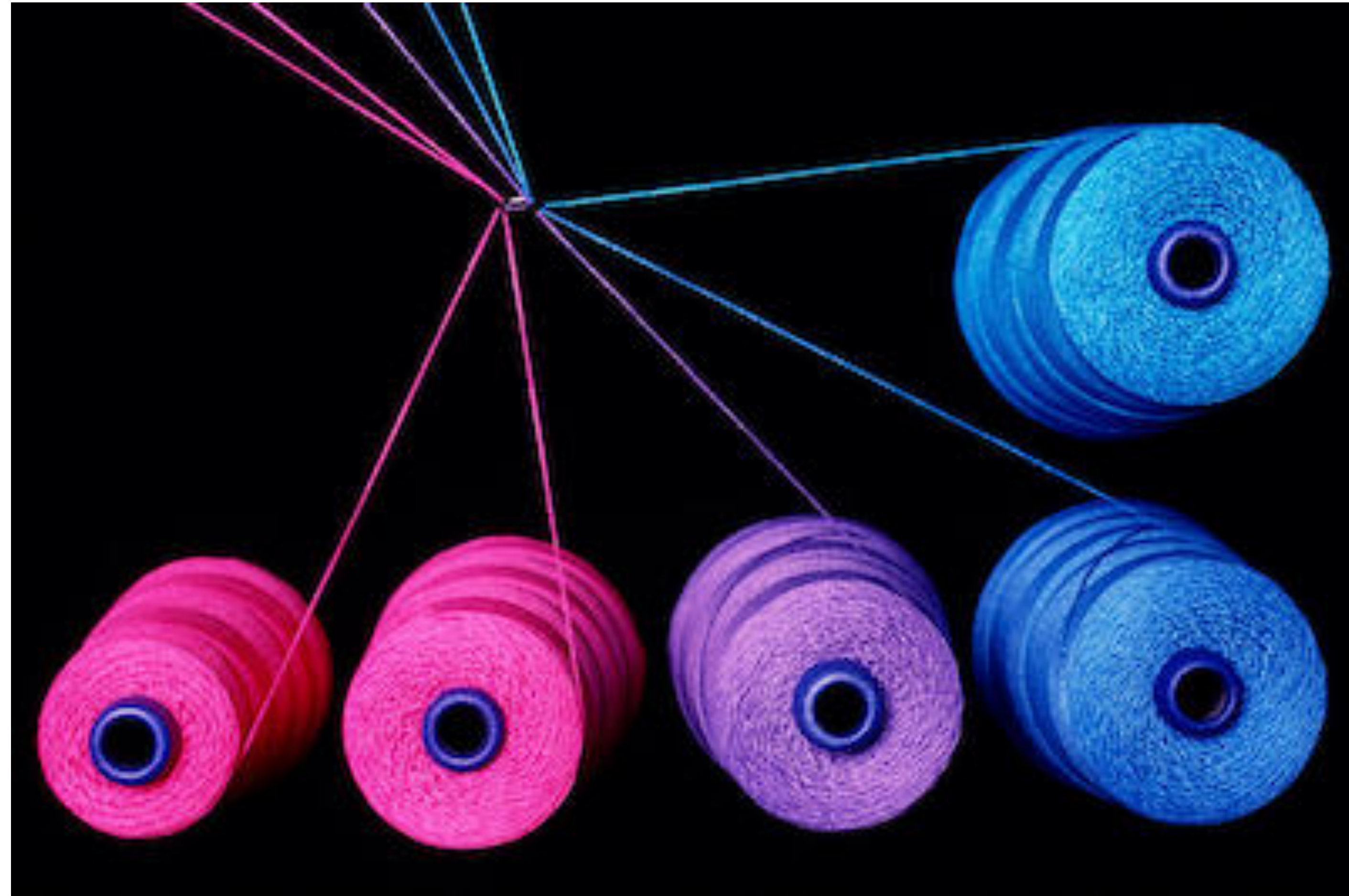
We need to use some additional **aside structure** to keep a record of all observers for each object.

```
class GlobalBottleneck
{
    // (Un)RegisterOrbserverFor(const Actor *, IObserver *);

    std::unordered_map<const Actor*, std::vector<IObserver *>> mObservers;
};
```

Threads

Multi-lane highway to... crashes 🔥



Put a `mutex` bottleneck on it !

Guard each function with a **mutex**:

- `Widget::Set()`
- `Widget::AddObserver()`
- `Widget::RemoveObserver()`
- `Widget::NotifyObservers()`

Recursive add/remove observers, bites again!

`recursive_mutex` ? 😊

Threads

```
class Widget
{
    Data mData;
    std::recursive_mutex mMtx;

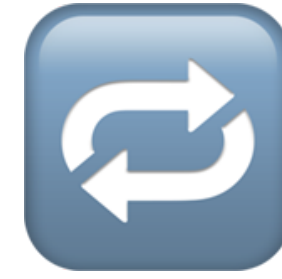
public:

    void Set(const Data & d)
    {
        std::lock_guard<recursive_mutex> lock(mMtx);

        if (d != mData) {
            mData = d;
            NotifyObservers();
        }
    }
};
```


Not bulletproof!

You can get in a dead-lock situation.



`recursive_mutex` 🙄

What about **Squaring the Circle** ?

What about **Squaring the Circle** ?

Value-oriented design in an **Object**-oriented system



Value-oriented design in an object-oriented system - Juan Pedro Bolivar Puentes [C++ on Sea 2020]

youtube.com/watch?v=SAMR5GJ_GqA

When in doubt, always make **copies**.



Threads

```
void Widget::NotifyObservers()
{
    std::vector<IObserver *> cpy;
    {
        std::lock_guard<mutex> lock(mMtx);
        cpy = mObservers;
    }

    size_t count = cpy.size();
    for (size_t i = 0; i < count; ++i) // avoid the issues with iter invalidation
    {
        if (mObservers[i])
            cpy[i]->WidgetChanged(this);
    }

    {
        std::lock_guard<mutex> lock(mMtx);
        std::erase(mObservers, nullptr); // deferred cleanup of removed observers
    }
}
```

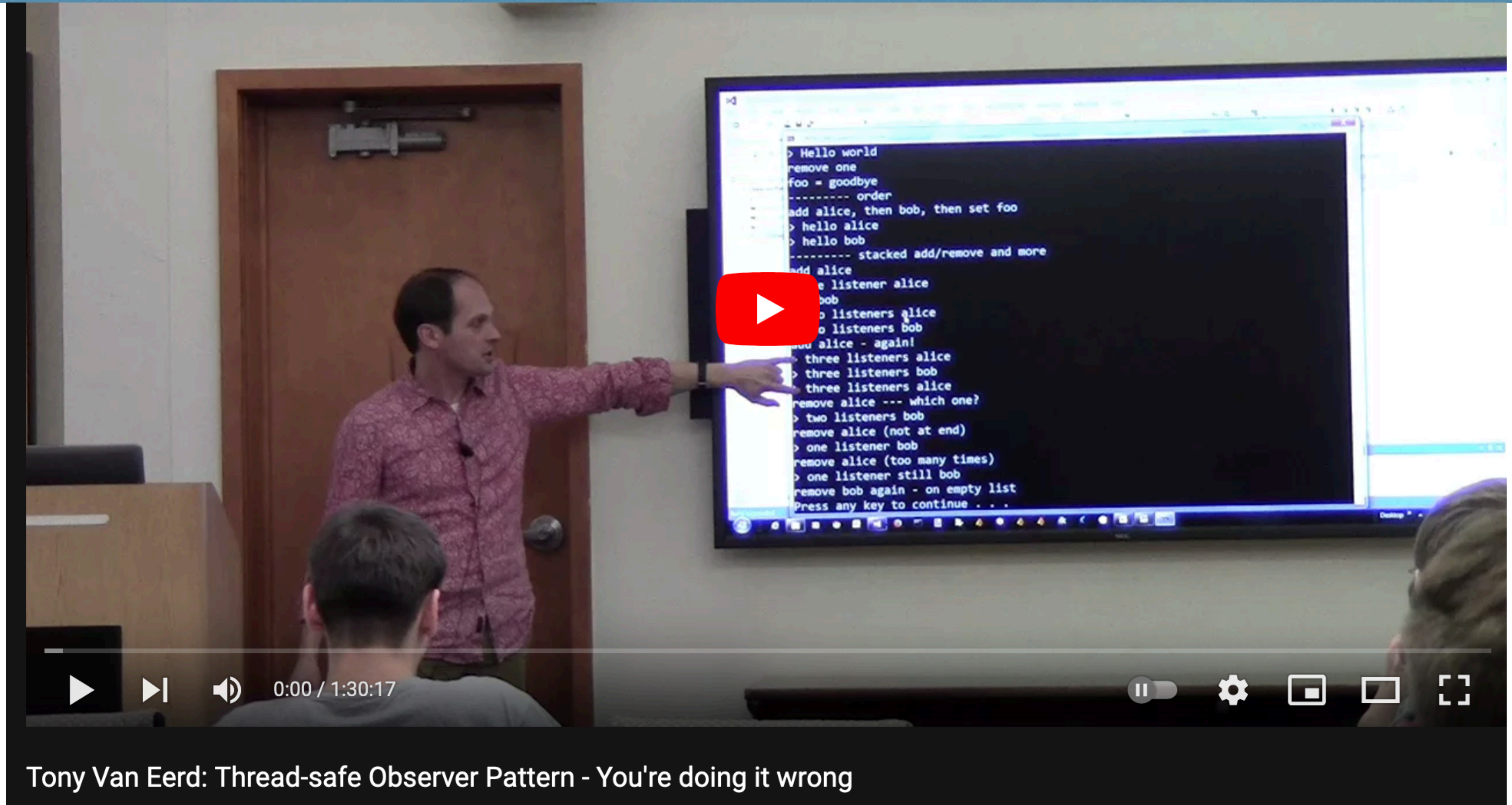


We probably need something like:

```
QObject::deleteLater()
```

In general, even if you're not using **Qt**,

I think it's very instructive to learn how UI observers are designed to work in Qt.



```
> Hello world
remove one
foo = goodbye
----- order
add alice, then bob, then set foo
> hello alice
> hello bob
----- stacked add/remove and more
add alice
add listener alice
add listener bob
> listeners alice
> listeners bob
add alice - again!
add three listeners alice
add three listeners bob
add three listeners alice
remove alice --- which one?
> two listeners bob
remove alice (not at end)
> one listener bob
remove alice (too many times)
> one listener still bob
remove bob again - on empty list
Press any key to continue . . .
```

Tony Van Eerd: Thread-safe Observer Pattern - You're doing it wrong

www.youtube.com/watch?v=RVvVQply6zc

Threads

Basically, in a multi-threaded context, it's almost impossible to implement a solid Observer pattern

In real code you can't see the **deadlocks**... until they happen.

Rule of thumb

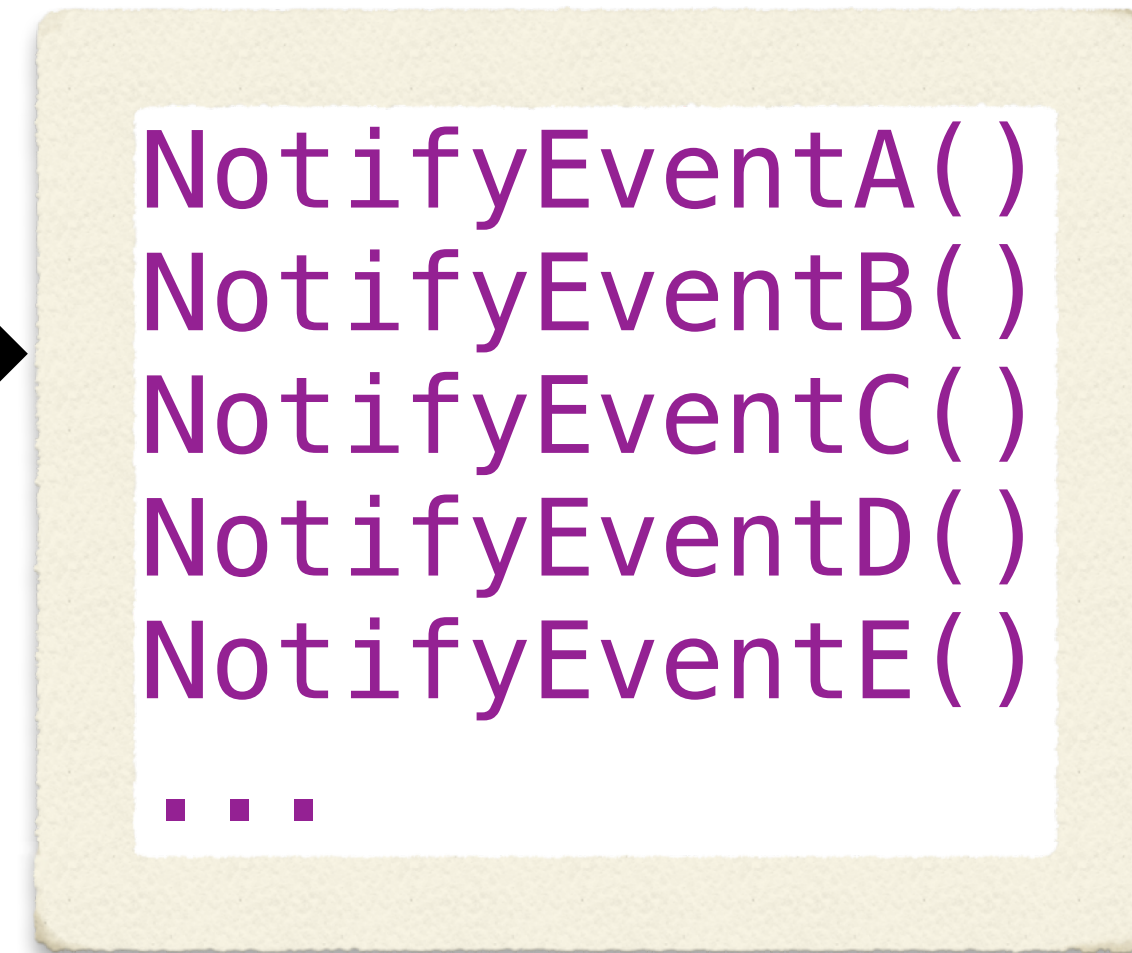
**Don't hold a lock
while calling *unknown* code.**



Intrusive

Anyway, we don't want all this mess inside our type:

- `Widget::AddObserver()`
- `Widget::RemoveObserver()`
- `Widget::NotifyObservers(.)`
- ...



And we want a generic/reusable template as a base.

```
class Widget : public Actor<Widget>  
{
```

Remote Objects

Inspectable properties and remote objects

```
class Widget : public Actor<Widget>
{
    Data mData;

public:
    void Set(const Data & d) {
        if (d != mData) {
            mData = d;
            NotifyObservers();
        }
    }
};
```



"spooky action at a distance"

Remote Observer

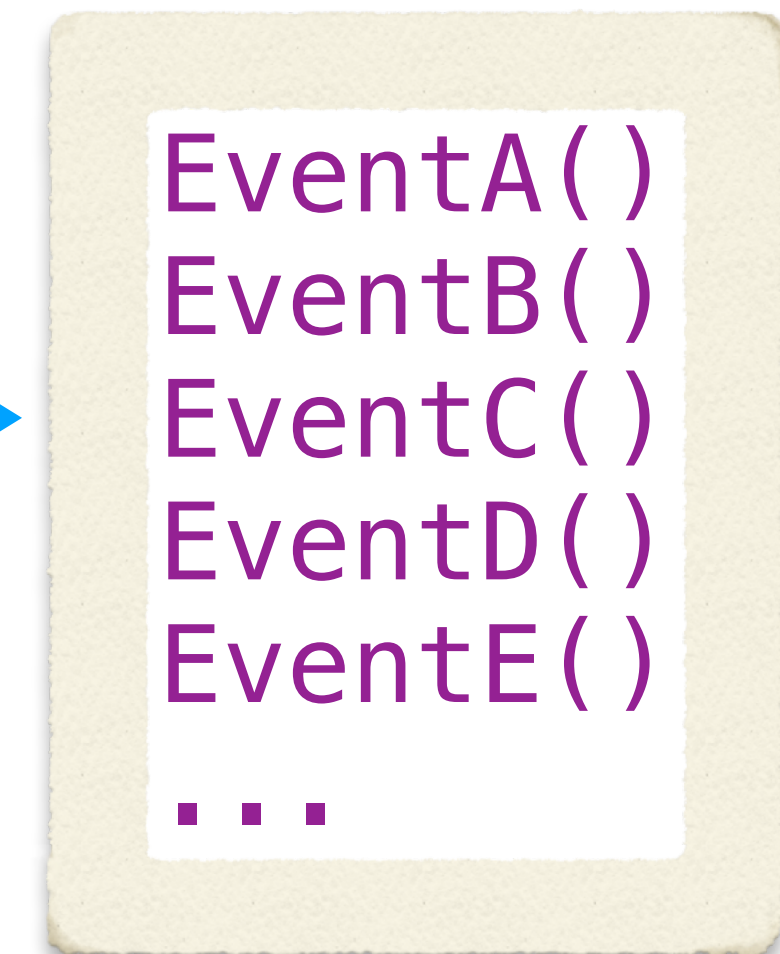


```
class RemoteObserver : public IObserver
{
    RemoteObserver() {
        mWidget->AddObserver(*this);
    }

    ~RemoteObserver() {
        mWidget->RemoveObserver(*this);
    }

    void WidgetChanged(Actor * sender) override
    {
        // react in some way to the changed object (actor)
        sender->Query??();
    }

    ...
    Actor * mWidget;
};
```



Dangling



```
class RemoteObserver : public IObserver
{
    RemoteObserver() {
        mWidget->AddObserver(*this);
    }

    ~RemoteObserver() {
        mWidget->RemoveObserver(*this);
    }

    ...
    Actor * mWidget;
};
```

Don't forget to cancel...

```
// RAII
RegisterObserver obs(*this, mWidget);
```

Optional Protocol Methods

```
class IStuffObserver
{
public:
    virtual void StuffAdded(const Stuff & stuff) = 0;
    virtual void StuffRemoved(const Stuff & stuff) = 0;
    virtual void StuffWillChange(const Stuff & stuff) = 0;
    virtual void StuffChanged(const Stuff & stuff) = 0;
    virtual void GoingToSleep(const Stuff & stuff) = 0;
    virtual void WakingUp(const Stuff & stuff) = 0;
    ...
};
```

Optional Protocol Methods


```
class StuffObserver : public IStuffObserver
{
public:
    void StuffAdded(const Stuff & stuff) override { ... }
    void StuffRemoved(const Stuff & stuff) override { ... }
    void StuffWillChange(const Stuff & stuff) override { ... }
    void StuffChanged(const Stuff & stuff) override { ... }
    void GoingToSleep(const Stuff & stuff) override { ... }
    void WakingUp(const Stuff & stuff) override { ... }
    ...
};
```

```
class Spectator : public StuffObserver
{
    void GoingToSleep(const Stuff & stuff) override
    {
        ...
    }
};
```

Observe Multiple Actors

```
class Spectator : public StuffObserver
{
    Spectator(Actor & actor)
    : mActor(actor)
    {
        mActor.AddObserver(*this);
    }
    ~Spectator()
    {
        mActor.RemoveObserver(*this);
    }

    void GoingToSleep(const Stuff & stuff) override
    {
        ...
    }
};
```



The diagram consists of a light brown rectangular box with rounded corners and a white label that says "Actor". A thick black arrow originates from the right side of this box and points diagonally down and to the left, ending at the line of code `mActor.RemoveObserver(*this);` in the destructor of the `Spectator` class.

Observe Multiple Actors

```
class Spectator : public StuffObserver
{
    Spectator(Actor & actor, Thespian & thespian)
    : mActor(actor), mThespian(thespian)
    {
        mActor.AddObserver(*this);
        mThespian.AddObserver(*this);
    }
    ~Spectator()
    {
        mActor.RemoveObserver(*this);
        mThespian.RemoveObserver(*this);
    }

    void GoingToSleep(const Stuff & stuff) override
    {
        ...
    }
};

void GoingToSleep(const Stuff & stuff) override
{
    ...
}
```



The diagram consists of two light brown rectangular boxes with rounded corners. The left box is labeled 'Actor' and the right box is labeled 'Thespian'. Two black arrows originate from the bottom of these boxes. The arrow from the 'Actor' box points to the first 'void GoingToSleep' method signature in the code. The arrow from the 'Thespian' box points to the second 'void GoingToSleep' method signature in the code.



“ There is no problem in computer science that can't be solved using another **level of indirection** ”

Observer Proxies

```
template<int ObserverIndex>
class StuffObserver
{
public:
    using TypeId = Int2Type<ObserverIndex>;

    void StuffAdded(TypeId, const Stuff & stuff) override { ... }
    void StuffRemoved(TypeId, const Stuff & stuff) override { ... }
    void StuffWillChange(TypeId, const Stuff & stuff) override { ... }
    void StuffChanged(TypeId, const Stuff & stuff) override { ... }
    void GoingToSleep(TypeId, const Stuff & stuff) override { ... }
    void WakingUp(TypeId, const Stuff & stuff) override { ... }
    ...
};
```

Observer Proxies

```
template<int ObserverIndex>
class StuffObserver
{
public:
    using TypeId = Int2Type<ObserverIndex>;

    void StuffAdded(TypeId, const Stuff & stuff) override { ... }
    void StuffRemoved(TypeId, const Stuff & stuff) override { ... }
    void StuffWillChange(TypeId, const Stuff & stuff) override { ... }
    void StuffChanged(TypeId, const Stuff & stuff) override { ... }
    void GoingToSleep(TypeId, const Stuff & stuff) override { ... }
    void WakingUp(TypeId, const Stuff & stuff) override { ... }
    ...
};
```

```
template <int v>
struct Int2Type
{
    enum {
        value = v
    };
};
```

Observer Proxies

```
template<int ObserverIndex>
class StuffObserver
{
public:
    using TypeId = Int2Type<ObserverIndex>;

    void StuffAdded(TypeId, const Stuff & stuff) override { ... }
    void StuffRemoved(TypeId, const Stuff & stuff) override { ... }
    void StuffWillChange(TypeId, const Stuff & stuff) override { ... }
    void StuffChanged(TypeId, const Stuff & stuff) override { ... }
    void GoingToSleep(TypeId, const Stuff & stuff) override { ... }
    void WakingUp(TypeId, const Stuff & stuff) override { ... }
    ...
};
```

```
template <int v>
struct Int2Type
{
    enum {
        value = v
    };
};
```

If you recognize this, you've been writing C++ for a while... (**Loki** by [A.A.](#))

Observer Proxies

```
template<typename ObserverT, int ObserverIndex>
class StuffObserverProxy : public IStuffObserver
{
public:
    using TypeId = Int2Type<ObserverIndex>;
    using ReceiverType = StuffObserver<ObserverIndex>;

    StuffObserverProxy(ObserverT & observer)
    : mObserver(observer)
    {}

    void StuffAdded(const Stuff & stuff) override {
        static_cast<ReceiverType &>(mObserver).StuffAdded(TypeId(), stuff);
    }
    void StuffRemoved(const Stuff & stuff) override {
        static_cast<ReceiverType &>(mObserver).StuffRemoved(TypeId(), stuff);
    }
    ...
    ObserverT & mObserver;
};
```

Observer Proxies

```
namespace SpectatorObserverProxies
{
    using Actor          = StuffObserver<0>;
    using ActorProxy    = StuffObserverProxy<Spectator, 0>;

    using Thespian      = StuffObserver<1>;
    using ThespianProxy = StuffObserverProxy<Spectator, 1>;
    ...
}
```

Observer Proxies


```
class Spectator : public SpectatorObserverProxies::Actor,  
                 public SpectatorObserverProxies::Thespian  
{  
public:  
  
    Spectator()  
    : mActorProxy(*this), mThespianProxy(*this)  
    {  
        mActor.AddObserver(mActorProxy);  
        mThespian.AddObserver(mThespianProxy);  
    }  
  
private:  
  
    SpectatorObserverProxies::ActorProxy    mActorProxy;  
    SpectatorObserverProxies::ThespianProxy mThespianProxy;  
};
```



Observer Proxies

```
class Spectator : public SpectatorObserverProxies::Actor,  
                 public SpectatorObserverProxies::Thespian  
{  
    void GoingToSleep(SpectatorObserverProxies::Actor::TypeId,  
                     const Stuff & stuff) override  
    {  
        ... // actor goes to sleep  
    }  
    void GoingToSleep(SpectatorObserverProxies::Thespian::TypeId,  
                     const Stuff & stuff) override  
    {  
        ... // thespian goes to sleep  
    }  
    void StuffAdded(SpectatorObserverProxies::Actor::TypeId,  
                   const Stuff & stuff) override  
    {  
        ... // actor added some new stuff  
    }  
}
```

Observer Proxies

`StuffObserver` and `StuffObserverProxy` can be  reused for any other `Spectator` type and for any observed subjects/actors, conforming to the defined `ISuffObserver` interface.

```
namespace SpectatorObserverProxies
{
    using Actor          = StuffObserver<0>;
    using ActorProxy    = StuffObserverProxy<Spectator, 0>;

    using Thespian      = StuffObserver<1>;
    using ThespianProxy = StuffObserverProxy<Spectator, 1>;
    ...
}
```

Global State

Observer **networks** form a global state.



The same reason I dislike `std::shared_ptr<>`

Pushing up the daisies

Memory management issues:

- dead subjects
- missing observers

Blissfully dangling...

Pushing up the daisies



Spooky Action at a Distance



MUC++

December 14th 2021



@ciura_victor

Victor Ciura
Principal Engineer

