

Design Patterns in the XXI Century

Dmitri Nesteruk

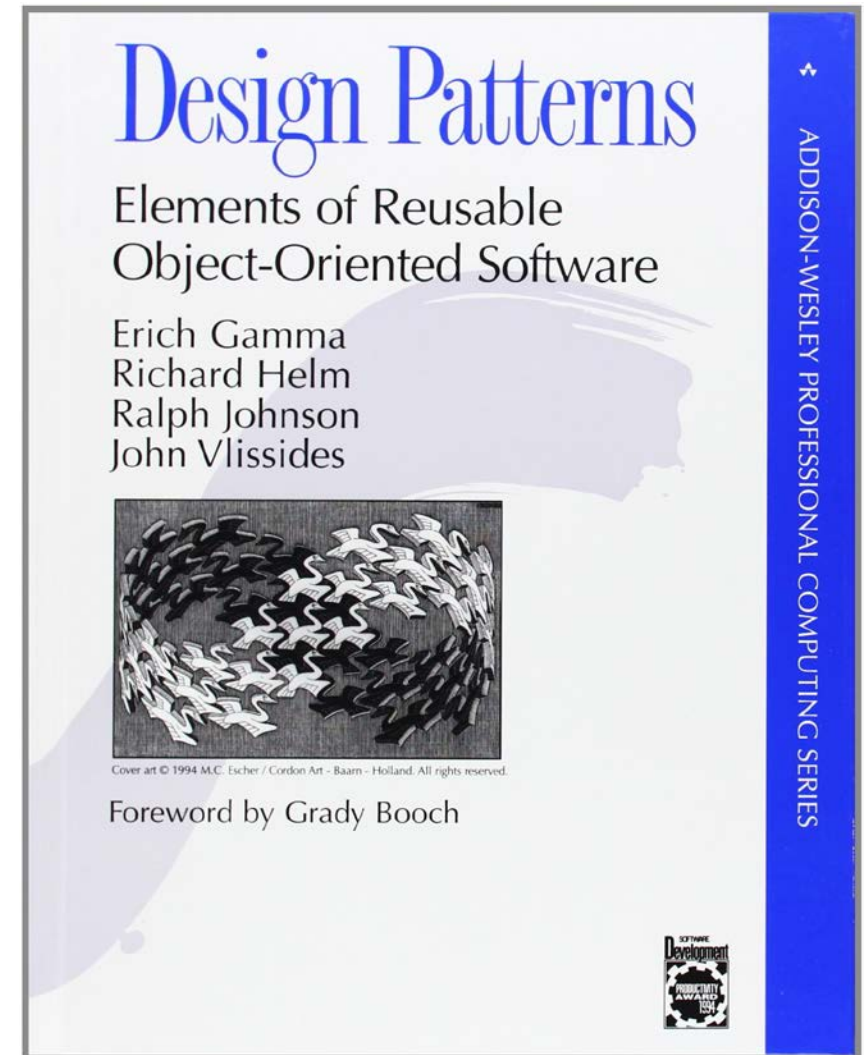
dmitrinesteruk@gmail.com

@dnesteruk

©Dmitri Nesteruk. Presented at C++ Now 2016

Brief Recap

- 1994 book by “Gang of Four”
- Examples in C++ and Smalltalk
- Patterns became “sacred knowledge”
- Translated to many (all?) other OOP languages
- Internalized in some languages
 - E.g., Observer = event in C#
 - But not in C++



This and other images shamelessly stolen from the interwebs

What's In This Talk?

- Examples of patterns and approaches in OOP design
 - In C++
- Adapter
- Composite
- Specification pattern/OCP
- Fluent and Groovy-style builders
- Maybe monad

Disclaimers

- Suboptimal code (pass by value etc.)
- Namespaces mostly imported
- May include non-standard code
 - I.e., code that compiles only in MSVC 😊

Adapter

STL String Complaints

- Making a string is easy
`string s{"hello world"};`
- Getting its constituent parts is not
`vector<string> words;`
`boost::split(words, s, boost::is_any_of(" "));`
- Instead I would prefer
`auto parts = s.split(" ");`
- It should work with "hello world"
- Maybe some other goodies, e.g.
 - Hide `size()`
 - Have `length()` as a property, not a function

Basic Adapter

```
class String {  
    string s;  
public:  
    String(const string &s) : s{ s } { }  
};
```

Implement Split

```
class String {
    string s;
public:
    String(const string &s) : s{ s } { }
    vector<string> split(string input)
    {
        vector<string> result;
        boost::split(result, s,
            boost::is_any_of(input), boost::token_compress_on);
        return result;
    }
};
```


Length Proxying

```
class String {  
    string s;  
public:  
    String(const string &s) : s{ s } { }  
    vector<string> split(string input);  
    size_t get_length() const { return s.length(); }  
};
```

Length Proxying

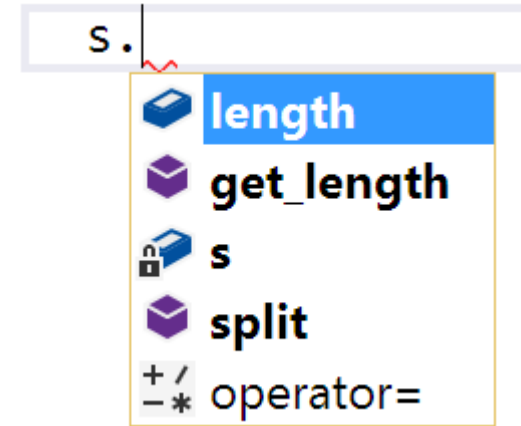
```
class String {
    string s;
public:
    String(const string &s) : s{ s } { }
    vector<string> split(string input);
    size_t get_length() const { return s.length(); }

    // not portable, works in at least MSVC+Intel
    __declspec(property(get = get_length)) size_t length;
};
```

String Wrapper Usage

```
auto words = s.split(" ");  
for (auto& word : words)  
    cout << word << endl;
```

```
String s{ "hello world" };  
cout << "string has " << s.length  
    << " characters" << endl;
```



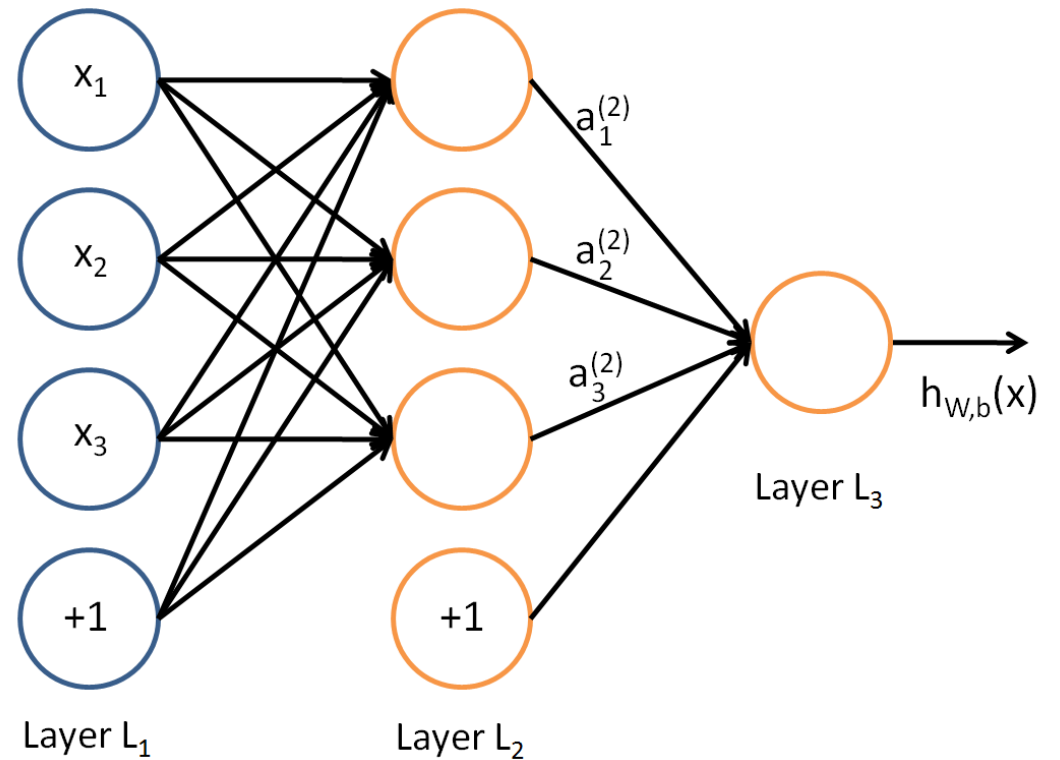
Adapter Summary

- Aggregate objects (or keep a reference)
- Can aggregate more than one
 - E.g., string and formatting
- Replicate the APIs you want (e.g., length)
 - No tools for this (yet)
- Miss out on the APIs you don't need
- Add your own features :)

Composite

Scenario

- Neurons connect to other neurons
- Neuron *layers* are collections of neurons
- These two need to be connectable



Scenario

```
struct Neuron
{
    vector<Neuron*> in, out;
    unsigned int id;

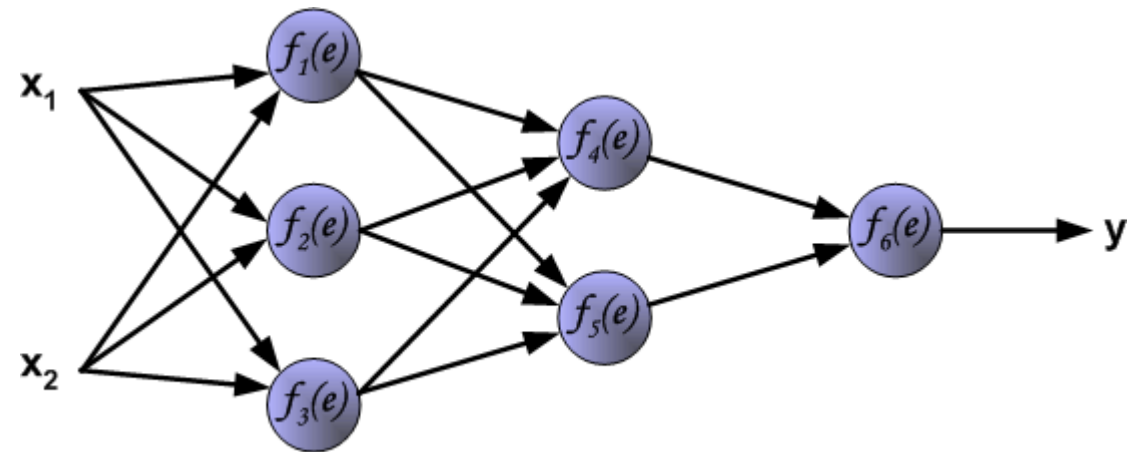
    Neuron()
    {
        static int id = 1;
        this->id = id++;
    }
}
```

Scenario

```
struct NeuronLayer : vector<Neuron>
{
    NeuronLayer(int count)
    {
        while (count --> 0)
            emplace_back(Neuron{});
    }
}
```


State Space Explosion

- `void connect_to(Neuron& other)`
{
 `out.push_back(&other);`
 `other.in.push_back(this);`
}
- Unfortunately, we need 4 functions
 - Neuron to Neuron
 - Neuron to NeuronLayer
 - NeuronLayer to Neuron
 - NeuronLayer to NeuronLayer



One Member Function Solution?

- Simple: treat Neuron as NeuronLayer of size 1
 - Not strictly correct
 - Does not take into account other concepts (e.g., NeuronRing)
- Better: expose a single Neuron in an iterable fashion
- Other programming languages have interfaces for iteration
 - E.g., C# `IEnumerable<T>`
 - `yield` keyword
- C++ does duck typing
 - Expects `begin/end` pair
- One function solution not possible, but...

Generic Connection Function

```
struct Neuron
{
    // Neuron to Neuron (as before)
    template<> void connect_to<Neuron>(Neuron& other) { ... }
    // general
    template <typename T> void connect_to(T& other)
    {
        for (Neuron& to : other)
            connect_to(to);
    }
};
```

Generic Connection Function

```
struct NeuronLayer : vector<Neuron>
{
    ...
    template <typename T> void connect_to(T& other)
    {
        for (Neuron& from : *this)
            for (Neuron& to : other)
                from.connect_to(to);
    }
};
```

How to Iterate on a Single Value?

```
struct Neuron
{
    ...
    Neuron* begin() { return this; }
    Neuron* end()   { return this + 1; }
};
```

API Usage

```
Neuron n, n2;
```

```
NeuronLayer n1{3}, n12{5};
```

```
n.connect_to(n2); // Neuron::connect_to<Neuron>
```

```
n.connect_to(n1); // Neuron::connect_to<T = NL>
```

```
n1.connect_to(n); // what we did begin()/end() for
```

```
n1.connect_to(n12); // NL::connect_to<T = NL>
```

Specification Pattern and the OCP

Open-Closed Principle

- Open for extension, closed for modification
- Bad: jumping into old code to change a stable, functioning system
- Good: making things generic enough to be externally extensible
- Example: product filtering

Scenario

```
enum class Color { Red, Green, Blue };  
enum class Size { Small, Medium, Large };
```

```
struct Product  
{  
    std::string name;  
    Color color;  
    Size size;  
};
```

Filtering Products

```
struct ProductFilter
{
    typedef std::vector<Product*> Items;
    static Items by_color(Items items, Color color)
    {
        Items result;
        for (auto& i : items)
            if (i->color == color)
                result.push_back(i);
        return result;
    }
}
```

Filtering Products

```
struct ProductFilter
{
    typedef std::vector<Product*> Items;
    static Items by_color(Items items, Color color) { ... }
    static Items by_size(Items items, Size size)
    {
        Items result;
        for (auto& i : items)
            if (i->size == size)
                result.push_back(i);
        return result;
    }
}
```

Filtering Products

```
struct ProductFilter
{
    typedef std::vector<Product*> Items;
    static Items by_color(Items items, Color color) { ... }
    static Items by_size(Items items, Size size) { ... }
    static Items by_color_and_size(Items items, Size size, Color color)
    {
        Items result;
        for (auto& i : items)
            if (i->size == size && i->color == color)
                result.push_back(i);
        return result;
    }
}
```

Violating OCP

- Keep having to rewrite existing code
 - Assumes it is even possible (i.e. you have access to source code)
- Not flexible enough (what about other criteria?)
- Filtering by X or Y or X&Y requires 3 functions
 - More complexity -> state space explosion
- Specification pattern to the rescue!

ISpecification and IFilter

```
template <typename T> struct ISpecification
{
    virtual bool is_satisfied(T* item) = 0;
};

template <typename T> struct IFilter
{
    virtual std::vector<T*> filter(
        std::vector<T*> items,
        ISpecification<T>& spec) = 0;
};
```

A Better Filter

```
struct ProductFilter : IFilter<Product>
{
    typedef std::vector<Product*> Products;
    Products filter(
        Products items,
        ISpecification<Product>& spec) override
    {
        Products result;
        for (auto& p : items)
            if (spec.is_satisfied(p))
                result.push_back(p);
        return result;
    }
};
```

Making Specifications

```
struct ColorSpecification : ISpecification<Product>
{
    Color color;
    explicit ColorSpecification(const Color color)
        : color{color} { }

    bool is_satisfied(Product* item) override {
        return item->color == color;
    }
}; // same for SizeSpecification
```


Improved Filter Usage

```
Product apple{ "Apple", Color::Green, Size::Small };  
Product tree { "Tree", Color::Green, Size::Large };  
Product house{ "House", Color::Blue, Size::Large };
```

```
std::vector<Product*> all{ &apple, &tree, &house };
```

```
ProductFilter pf;  
ColorSpecification green(Color::Green);
```

```
auto green_things = pf.filter(all, green);  
for (auto& x : green_things)  
    std::cout << x->name << " is green" << std::endl;
```

Filtering on 2..N criteria

- How to filter by size **and** color?
- We don't want a SizeAndColorSpecification
 - State space explosion
- Create combinators
 - A specification which *combines* two other specifications
 - E.g., AndSpecification

AndSpecification Combinator

```
template <typename T> struct AndSpecification : ISpecification<T>
{
    ISpecification<T>& first;
    ISpecification<T>& second;

    AndSpecification(ISpecification<T>& first,
                    ISpecification<T>& second)
        : first{first}, second{second} { }

    bool is_satisfied(T* item) override
    {
        return first.is_satisfied(item) && second.is_satisfied(item);
    }
};
```

Filtering by Size AND Color

```
ProductFilter pf;  
ColorSpecification green(Color::Green);  
SizeSpecification big(Size::Large);  
AndSpecification<Product> green_and_big{ big, green };  
  
auto big_green_things = pf.filter(all, green_and_big);  
for (auto& x : big_green_things)  
    std::cout << x->name << " is big and green" << std::endl;
```

Specification Summary

- Simple filtering solution is
 - Too difficult to maintain, violates OCP
 - Not flexible enough
- Abstract away the specification interface
 - `bool is_satisfied_by(T something)`
- Abstract away the idea of filtering
 - Input items + specification \rightarrow set of filtered items
- Create combinators (e.g., `AndSpecification`) for combining multiple specifications

Fluent and Groovy-Style Builders

Scenario

- Consider the construction of structured data
 - E.g., an HTML web page
- Structured and formalized
- Rules (e.g., P cannot contain another P)
- Can we provide an API for building these?

Building a Simple HTML List

```
// <ul><li>hello</li><li>world</li></ul>
string words[] = { "hello", "world" };
ostringstream oss;
oss << "<ul>";
for (auto w : words)
    oss << "  <li>" << w << "</li>";
oss << "</ul>";
printf(oss.str().c_str());
```


HtmlElement

```
struct HtmlElement
{
    string name;
    string text;
    vector<HtmlElement> elements;
    const size_t indent_size = 2;

    string str(int indent = 0) const; // pretty-print
}
```

Html Builder (non-fluent)

```
struct HtmlBuilder
{
    HtmlElement root;
    HtmlBuilder(string root_name) { root.name = root_name; }
    void add_child(string child_name, string child_text)
    {
        HtmlElement e{ child_name, child_text };
        root.elements.emplace_back(e);
    }
    string str() { return root.str(); }
}
```

Html Builder (non-fluent)

```
HtmlBuilder builder{"ul"};  
builder.add_child("li", "hello")  
builder.add_child("li", "world");  
cout << builder.str() << endl;
```

Making It Fluent

```
struct HtmlBuilder
{
    HtmlElement root;
    HtmlBuilder(string root_name) { root.name = root_name; }
    HtmlBuilder& add_child(string child_name, string child_text)
    {
        HtmlElement e{ child_name, child_text };
        root.elements.emplace_back(e);
        return *this; // or return HtmlBuilder* this
    }
    string str() { return root.str(); }
}
```

Html Builder

```
HtmlBuilder builder{"ul"};  
builder.add_child("li", "hello").add_child("li", "world");  
cout << builder.str() << endl;
```

Associate Builder & Object Being Built

```
struct HtmlElement
{
    static HtmlBuilder build(string root_name)
    {
        return HtmlBuilder{root_name};
    }
};

// usage:
HtmlElement::build("ul")
    .add_child_2("li", "hello").add_child_2("li", "world");
```

Groovy-Style Builders

- Express the *structure* of the HTML in code
- No visible function calls
- ```
UL {
 LI {"hello"},
 LI {"world"}
}
```
- Possible in C++ using uniform initialization

# Tag (= HTML Element)

```
struct Tag
{
 string name;
 string text;
 vector<Tag> children;
 vector<pair<string, string>> attributes;
protected:
 Tag(const std::string& name, const std::string& text)
 : name{name}, text{text} { }
 Tag(const std::string& name, const std::vector<Tag>& children)
 : name{name}, children{children} { }
}
```



# Paragraph

```
struct P : Tag
{
 explicit P(const std::string& text)
 : Tag{"p", text}
 {
 }
 P(std::initializer_list<Tag> children)
 : Tag("p", children)
 {
 }
};
```

# Image

```
struct IMG : Tag
{
 explicit IMG(const std::string& url)
 : Tag{"img", ""}
 {
 attributes.emplace_back(make_pair("src", url));
 }
};
```

# Example Usage

```
std::cout <<
```

```
 P {
```

```
 IMG {"http://pokemon.com/pikachu.png"}
```

```
 }
```

```
<< std::endl;
```

# Facet Builders

- An HTML element has different facets
  - Attributes, inner elements, CSS definitions, etc.
- A Person class might have different facets
  - Address
  - Employment information
- Thus, an object might necessitate *several* builders

# Personal/Work Information

```
class Person
{
 // address
 std::string street_address, post_code, city;

 // employment
 std::string company_name, position;
 int annual_income = 0;

 Person() {} // private!
}
```

# Person Builder (Exposes Facet Builders)

```
class PersonBuilder
{
 Person p;
protected:
 Person& person;
 explicit PersonBuilder(Person& person)
 : person{ person } { }
public:
 PersonBuilder() : person{p} { }
 operator Person() { return std::move(person); }

 // builder facets
 PersonAddressBuilder lives();
 PersonJobBuilder works();
}
```

# Person Builder (Exposes Facet Builders)

```
class PersonBuilder
{
 Person p;
protected:
 Person& person;
 explicit PersonBuilder(Person& person)
 : person{ person } { }
public:
 PersonBuilder() : person{p} { }
 operator Person() { return std::move(person); }

 // builder facets
 PersonAddressBuilder lives();
 PersonJobBuilder works();
}
```

# Person Builder Facet Functions

```
PersonAddressBuilder PersonBuilder::lives()
{
 return PersonAddressBuilder{ person };
}
```

```
PersonJobBuilder PersonBuilder::works()
{
 return PersonJobBuilder{ person };
}
```



# Person Address Builder

```
class PersonAddressBuilder : public PersonBuilder
{
 typedef PersonAddressBuilder Self;
public:
 explicit PersonAddressBuilder(Person& person)
 : PersonBuilder{ person } { }
 Self& at(std::string street_address)
 {
 person.street_address = street_address;
 return *this;
 }
 Self& with_postcode(std::string post_code);
 Self& in(std::string city);
};
```

# Person Job Builder

```
class PersonJobBuilder : public PersonBuilder
{
 typedef PersonJobBuilder Self;
public:
 explicit PersonJobBuilder(Person& person)
 : PersonBuilder{ person } { }

 Self& at(std::string company_name);
 Self& as_a(std::string position);
 Self& earning(int annual_income);
};
```

# Back to Person

```
class Person
{
 // fields
public:
 static PersonBuilder create();

 friend class PersonBuilder;
 friend class PersonAddressBuilder;
 friend class PersonJobBuilder;
};
```

# Final Person Builder Usage

```
Person p = Person::create()
 .lives().at("123 London Road")
 .with_postcode("SW1 1GB")
 .in("London")
 .works().at("PragmaSoft")
 .as_a("Consultant")
 .earning(10e6);
```

Maybe Monad

# Presence or Absence

- Different ways of expressing absence of value
- Default-initialized value
  - `string s; // there is no 'null string'`
- Null value
  - `Address* address;`
- Not-yet-initialized smart pointer
  - `shared_ptr<Address> address`
- Idiomatic
  - `boost::optional`

# Monads

- Design patterns in functional programming
- First-class function support
- Related concepts
  - Algebraic data types
  - Pattern matching
- Implementable *to some degree* in C++
  - Functional objects/lambda

# Scenario

```
struct Address
{
 char* house_name = nullptr;
}
```

```
struct Person
{
 Address* address = nullptr;
}
```



# Print House Name, If Any

```
void print_house_name(Person* p)
{
 if (p != nullptr &&
 p->address != nullptr &&
 p->address->house_name != nullptr)
 {
 cout << p->address->house_name << endl;
 }
}
```

# Maybe Monad

- Encapsulate the 'drill down' aspect of code
- Construct a `Maybe<T>` which keeps context
- Context: pointer to evaluated element
  - person -> address -> name
- While context is non-null, we drill down
- If context is `nullptr`, propagation does not happen
- All instrumented using lambdas

# Maybe<T>

```
template <typename T>
struct Maybe {
 T* context;
 Maybe(T *context) : context(context) { }
};
```

// but, given Person\* p, we cannot make a 'new Maybe(p)'

```
template <typename T> Maybe<T> maybe(T* context)
{
 return Maybe<T>(context);
}
```

# Usage So Far

```
void print_house_name(Person* p)
{
 maybe(p). // now drill down :)
}
```

# Maybe::With

```
template <typename T> struct Maybe
{
 ...
 template <typename TFunc>
 auto With(TFunc evaluator)
 {
 return context != nullptr
 ? maybe(evaluator(context))
 : nullptr;
 };
}
```

# Usage So Far

```
void print_house_name(Person* p)
{
 maybe(p) // now drill down :)
 .With([](auto x) { return x->address; })
 .With([](auto x) { return x->house_name; })
 . // print here (if context is not null)
}
```

# Maybe::Do

```
template <typename TFunc>
auto Do(TFunc action)
{
 // if context is OK, perform action on it
 if (context != nullptr) action(context);

 // no context transition, so...
 return *this;
}
```

# How It Works

- `print_house_name(nullptr)`
  - Context is null from the outset and continues to be null
  - **Nothing happens** in the entire evaluation chain
- `Person p; print_house_name(&p);`
  - Context is Person, but since Address is null, it becomes null henceforth
- `Person p;`  
`p->Address = new Address;`  
`p->Address->HouseName = "My Castle";`  
`print_house_name(&p);`
  - Everything works and we get our printout



# Maybe Monad Summary

- Example is not specific to `nullptr`
  - E.g., replace pointers with `boost::optional`
- Default-initialized types are harder
  - If `s.length() == 0`, has it been initialized?
- Monads are difficult due to lack of functional support
  - `[](auto x) { return f(x); }` instead of `x => f(x)` as in C#
  - No implicits (e.g. Kotlin's `it`, Rust's `$0`)

# That's It!

- Questions?
- Design Patterns in C++ courses on Pluralsight
- Book will happen (eventually)
- dmitrinesteruk /at/ gmail.com
- @dnesteruk