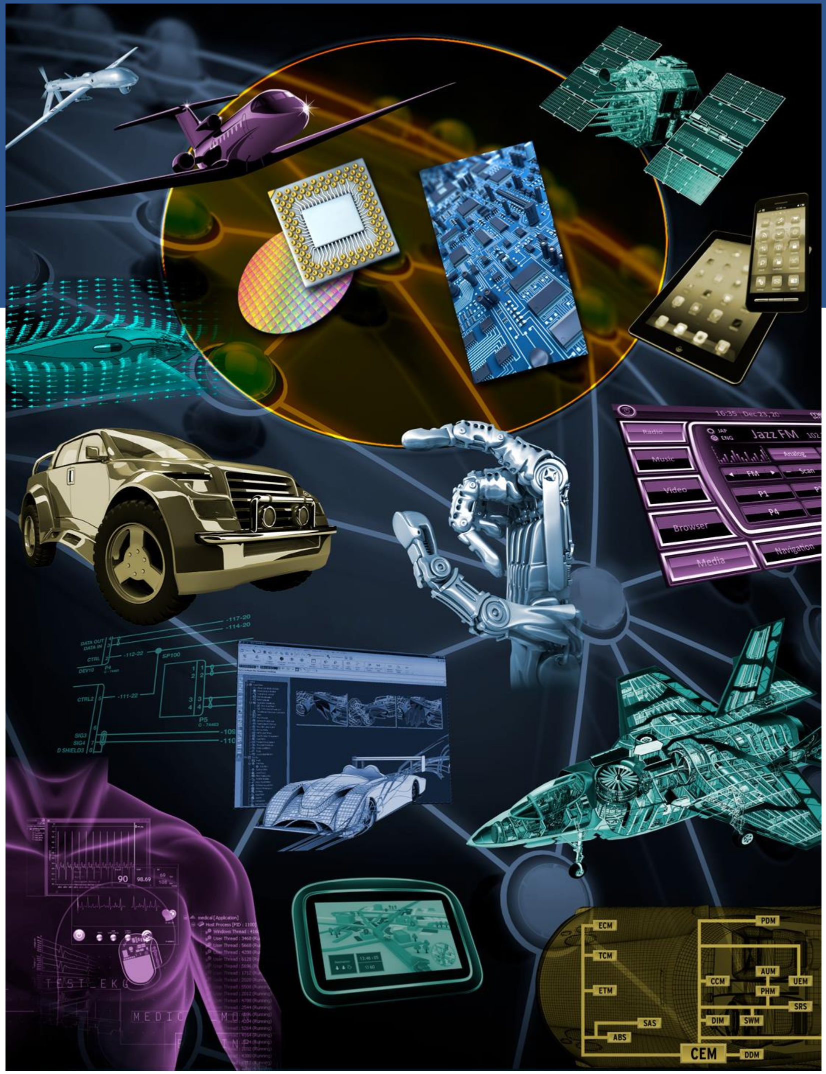


C++ Design Patterns: From C++03 to C++17

Fedor G Pikus

Chief Scientist,

Design2Silicon Division

CPPCon 2019

Mentor®

A Siemens Business

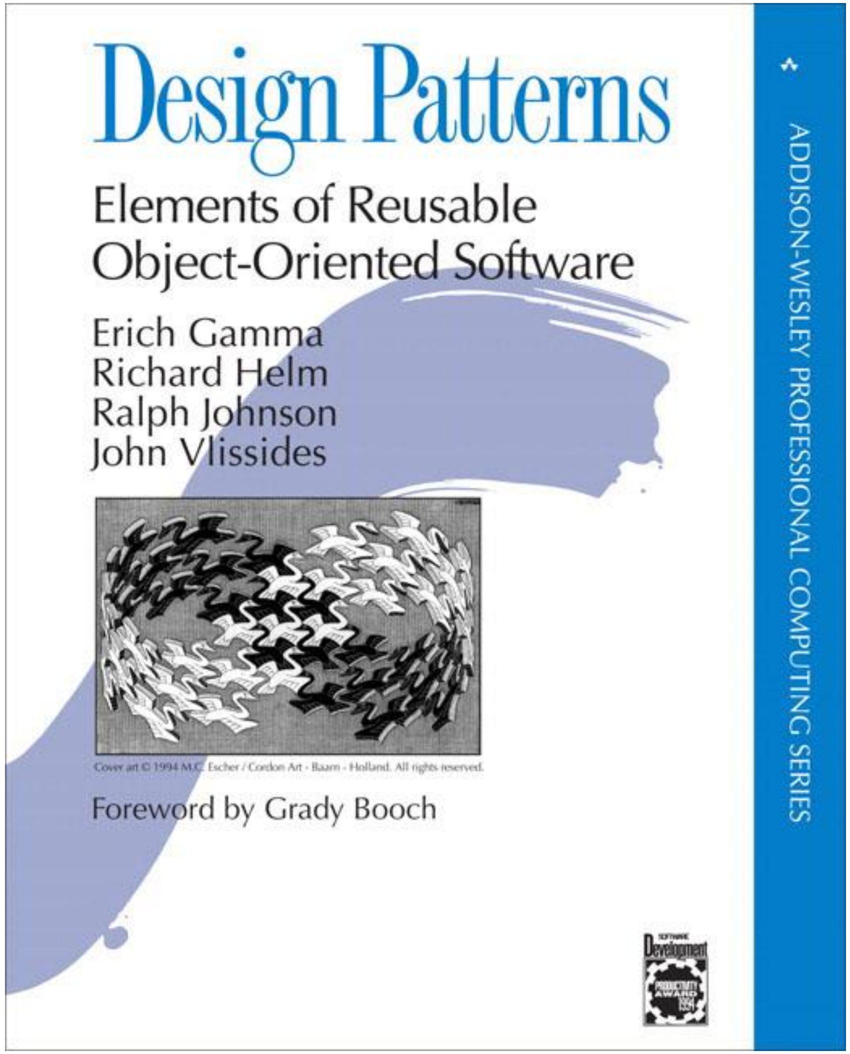
DESIGN PATTERNS

What are Design Patterns?

- [Software] Design pattern is a repeatable, commonly recognized and understood solution to a design problem commonly occurring in software engineering
  - Design problem
  - Commonly occurring
  - Widely accepted solution
  - Known advantages and trade-offs
- Patterns are "design templates", guidelines for design
- Patterns are compact expressive vocabulary elements
  - Like allusions in speech, they reference commonly known large concepts

What are Design Patterns?

- The "gang of four" (1995) book largely defined the way we think about patterns
- Concept is borrowed from architecture
  - — Christopher Alexander, 1977
    - – Concept is borrowed from the archetypes
- Software design patterns: Beck and Cunningham, 1987
- Patterns are not static, they evolve
  - — Not limited to the "original 23 patterns"
  - — Not limited to OOP patterns either
  - — Generic programming has patterns too



Should I Always Use Patterns?

- Patterns are very general principles
  - There is always a counter-example where the rigorous application of the rule is worse than breaking it
- A general principle is a "default" rule
  - A guideline that should be followed in absence of a good reason not to
- The majority of everyday work is "not special" and the result is better if this principle is followed
  - The majority of the exceptions (where an idiosyncratic solution is superior) don't gain enough to justify the effort – follow the default rule anyway
- There are new problems never seen before, or new constraints
- There isn't a design pattern for every challenge

Do Design Patterns Depend on the Language?

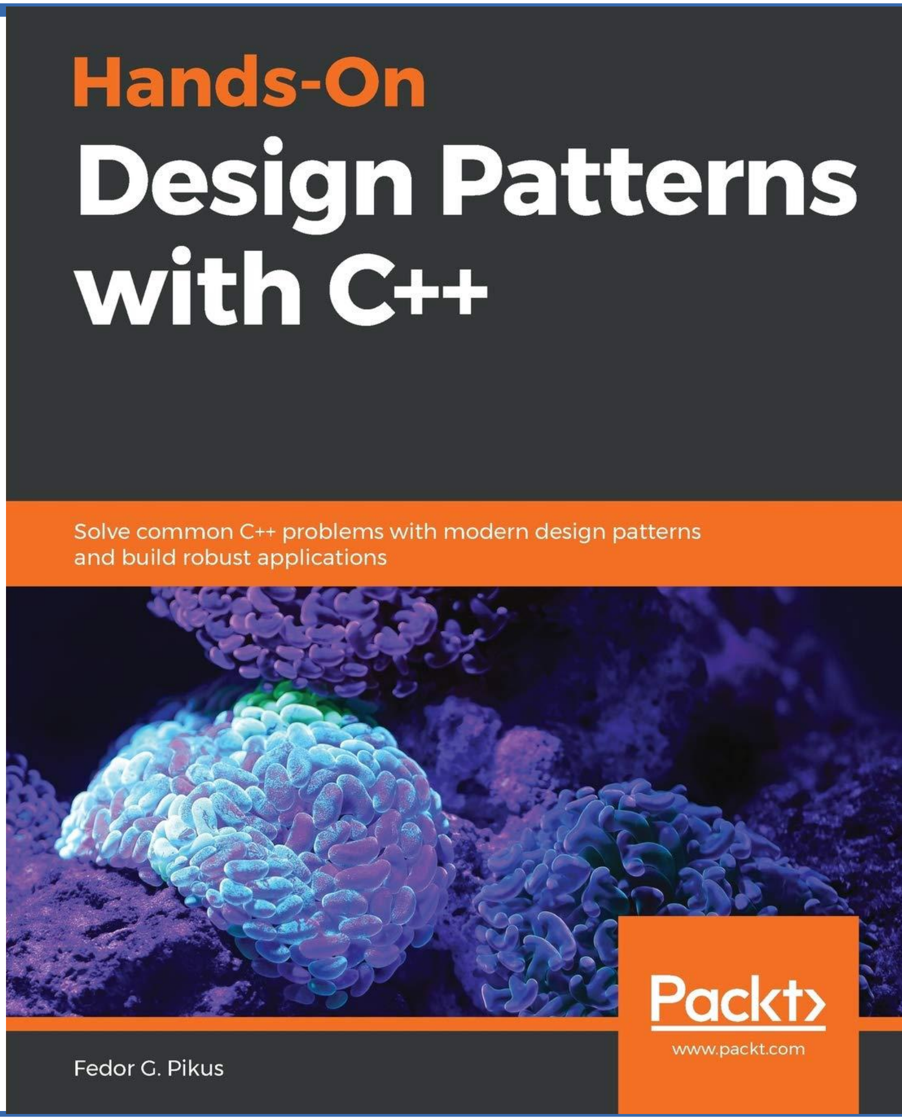
- Design patterns apply to software design and transcend language
  - In practice, some languages are preferred for certain problems
  - Some languages are more likely to create certain problems
- Some languages offer unique variations on a pattern
  - Strategy pattern: selecting an algorithm for a particular behavior aspect at run time (also known as policy pattern)
  - In C++, more commonly used as policy-based design (compile-time strategy pattern)
  - A pattern could be so hard on a particular language that it's not practical
- Contrast with language-specific idioms
  - Often exist to work around specific problems or deficiencies in a language
  - Change, appear, or disappear as language evolves

Does Language Development Affect Pattern Use?

- Not the same question as "do patterns depend on language?"
- Some patterns are easier to use in a particular language
  - Often, the same overall problem can be solved using multiple designs
- Some patterns just map perfectly to a language feature
  - Null object pattern – std::optional (C++17), Maybe (Haskell)
- Language development may change the ease of use balance between patterns
  - Some patterns become "easy/convenient enough" to use widely
- There is friction in the use of the patterns
  - "Small stuff" matters in practice
- A "tipping point" may prompt a different design approach

C++ Evolution and Patterns

- C++ features that significantly reduced the "friction" for using many design patterns:
  - C++14: universal references, variadic templates, lambdas, SFINAE, auto function return types
  - C++17: constructor templates and deduction rules, fold expressions, lambda overloads
- Patterns in this talk are mostly examples to illustrate the effect of C++ evolution on design decision



C++ Evolution and Patterns

- C++ features that significantly reduced the "friction" for using many design patterns:
  - C++14: universal references, variadic templates, lambdas, SFINAE, auto function return types
  - C++17: constructor templates and deduction rules, fold expressions, lambda overloads
- Patterns in this talk are mostly examples to illustrate the effect of C++ evolution on design decision

exercise for the reader

BUILDER

Builder

- The Builder is a pattern used to create (build) objects
- Builder is often used to construct complex objects that are initialized in multiple stages
- Generally, the Builder separates the creation of the object from its representation
  - Builder is a separate class (each class C has its own C_Builder)
  - Separate builder class is also a disadvantage

Builder in C++03

```
class HTMLElement {
    friend class HTMLBuilder;
    std::string name_;
    std::string text_;
    std::vector<HTMLElement> children_;
    public:
    HTMLElement(const std::string& name, const std::string& text)
        : name_(name), text_(text) {}
};
```

- Many different ways to write builders

Fluent Builder in C++03

- Many different ways to write builders

```
HTMLElement el = HTMLBuilder("ul").
    add_child("li", "item 1").
    add_child("li", "item 2");
```

- Method chaining

Fluent Builder in C++03

- Many different ways to write builders

```
HTMLElement el = HTMLBuilder("ul").
    add_child("li", "item 1").
    add_child("li", "item 2");
```

- Method chaining
  — Also used to create named arguments to C++ functions:

```
void Fly(double speed, double distance, double angle);
Fly(5, 180, 2.2);
```

Fluent Builder in C++03

- Many different ways to write builders

```
HTMLElement el = HTMLBuilder("ul").
    add_child("li", "item 1").
    add_child("li", "item 2");
```

- Method chaining
  - Also used to create named arguments to C++ functions:

```
void Fly(double speed, double distance, double angle);
Fly(5, 180, 2.2);
```

error-prone
Old ~~boring~~ way

```
Fly(FlyParams().Speed(5).Distance(2.2).Angle(180));
```

New cool way

Fluent Builder in C++03

```
class HTMLBuilder {
    HTMLElement root_;
    public:
    explicit HTMLBuilder(const string& name,
        const std::string& text = string()) : root_(name, text) {}
    HTMLBuilder& add_child(const std::string& name,
                           const std::string& text) {
        root_.children_.push_back(HTMLElement(name, text));
        return *this;
    }
    operator HTMLElement() const { return root_; }
};
```

Fluent Builder in C++11

```
class HTMLBuilder {
    HTMLBuilder& add_child(const std::string& name,
                           const std::string& text) {
        root_.children_.emplace_back(name, text));
        return *this;
    }
    operator HTMLElement() const { return root_; }
};
```

- Usual C++11 enhancements
  - Maybe C++11 for-loop to iterate over children

A Very C++11 Builder

```
std::cout << UL{
    LI{"item 1"},
    LI{"item 2",
        UL{LI{"sub-item 2.1"},
           LI{"sub-item 2.2"}
        }
    }
};
```

- No separate builder class

A Very C++11 Builder

```
class HTMLElement {
    std::string name_;
    std::string text_;
    std::vector<HTMLElement> children_;
    public:
    HTMLElement(const std::string& name, const std::string& text)
        : name_(name), text_(text) {}
    HTMLElement(const std::string& name, const std::string& text,
         std::vector<HTMLElement>&& children)
        : name_(name), text_(text), children_(std::move(children)) {}
}; // No more friends (in fact, no HTMLBuilder class)
```

A Very C++11 Builder

- Concrete classes define grammar
- Builder is mostly auto-generated

```
struct UL : public HTMLElement {
    UL() : HTMLElement("ul", "") {}
    UL(std::initializer_list<HTMLElement> children) :
      HTMLElement("ul", "", children) {};
};
```

A Very C++11 Builder

```
struct LI : public HTMLElement {
    explicit LI(const std::string& text) : HTMLElement("li", text) {}
    LI(const string& text, std::initializer_list<HTMLElement> children)
        : HTMLElement("li", text, children) {}
};
```

A Very C++11 Builder

```
struct LI : public HTMLElement {
    explicit LI(const std::string& text) : HTMLElement("li", text) {}
/* Don't I wish…
    LI(const string& text, std::initializer_list<HTMLElement> children)
        : HTMLElement("li", text, children) {} */
    template <typename ... Children>
    LI(const std::string& text, const Children& ... children)
        : HTMLElement("li", text,
            std::initializer_list<HTMLElement>{children ... }) {}
};
```

Does Language Development Affect Pattern Use?

- C++03 code wasn't particularly bad or annoying…
  — C++11 allows for some improvements (ease of use and efficiency)
- C++11 parameter packs can be used as code generators
  — New twist on the old pattern

VISITOR

Visitor

- The Visitor is a pattern that separates the algorithm from the object structure which is the data for this algorithm.
- Visitor adds new operations to the class hierarchy without modifying the classes themselves
- Open/Closed principle of the software design: a class should be closed for modifications but open for extensions
  - Interface should remain stable under maintenance
  - New functionality can be added to satisfy new requirements
- Useful for public APIs that must be extended by the clients

Visitor – Technical Viewpoint

- Visitor is double dispatch
- Single dispatch:

```
class B {
        virtual void f() = 0;
};
class D1 : public B {
        void f() { … do D1 stuff … };
};
class D2 : public B {
        void f() { … do D2 stuff … };
};
B* b = … D1 or D2 …;
b->f();
```

Depends on real type of *b

- The two viewpoints describe the same pattern

Visitor – Technical Viewpoint

- Visitor is double dispatch
- Double dispatch:

```
class B1 {                                  class B2 { … }
      virtual void f(B2*) = 0;
};
class D1A : public B1 { … }; class D1B : public B1 { … };
class D2A : public B2 { … }; class D2B : public B2 { … };
B1* b1 = … D1A or D1B …;
B2* b2 = … D2A or D2B …;
b1->f(b2);
```

Depends on real types of *b1, *b2

Why Visitor?

- Public APIs or other cases when changing source is not possible
- A way to keep decision-making decentralized
- Example: serialization
  - Each class knows how to serialize itself
  - There is serialization to disk, buffer, socket, other destination
  - One option is a huge central function with a case for every combination of class and destination (not the only option)
  - Visitor alternative: double dispatch based on class and destination

Classic Vistor (C++03)

- Class hierarchy:

```
class Cat; class Dog;
```

- New operations:

```
class FeedingVisitor {
        void visit(Cat* c); void visit(Dog* d);
};
```

- Client code:

```
Cat c("orange"); FeedingVisitor fv;
c.accept(fv);
```

Double dispatch

Classic C++ Visitor (C++03)

- Class hierarchy:

```
class Pet {
        std::string color_;
        public:
        Pet(const std::string& color) : color_(color) {}
        const std::string& color() const { return color_; }
        virtual void accept(PetVisitor& v) = 0;
};
class Cat : public Pet { void accept(PetVisitor& v) { v.visit(this); }  };
class Dog : public Pet { … };
Pet* p; p->accept(pv);
```

Depends on the Pet p
and the Visitor v

Classic C++ Visitor (C++03)

- Visitors (new operations):

```
class PetVisitor {
    virtual void visit(Cat* c) = 0;
    virtual void visit(Dog* d) = 0;
};
class FeedingVisitor : public PetVisitor {
    void visit(Cat* c) override {
        cout << "Feed tuna to the " << c->color() << " cat" << endl; }
    void visit(Dog* d) override {
        cout << "Feed steak to the " << d->color() << " dog" << endl; }
};
```

Classic C++ Visitor (C++03)

- Client code:

```
FeedingVisitor fv;
PlayingVisitor pv;
WalkingVisitor wv;
Pet* c = new Cat(“orange”);
Pet* d = new Dog(“brown”);
c->accept(pv);
d->accept(wv);
```

Why Visitor? And Why Not?

- (+) New operations can be added without modifying the hierarchy
  - After the classes are made visitable, once
- (+) Impossible to forget to implement an option
  - If the implementation for a class and a visitor type is missing, the code will not compile (pure virtual not overridden)
- (−) Once a class is added, all visitors must be updated
  - Visitor is recommended for “stable hierarchies”
- (−) Visitor does not have privileged access, sacrifices encapsulation
- (−) Visitor functions can take additional arguments and return values, but must be the same types for all visitors
  - Arguments are usually passed to visitors directly

Visitor in Modern C++

- Mostly cleaner and easier to maintain
- Hierarchy has boilerplate visitation code:

```
class Cat : public Pet {
    void accept(PetVisitor& v) { v.visit(this); }      // Cannot move to Pet
};
```

- Visitor classes must be declared, have some boilerplate:

```
class FeedingVisitor : public PetVisitor {
    void visit(Cat* c) override {
        cout << "Feed tuna to the " << c->color() << " cat" << endl; }
};
```

Visitor in Modern C++

- Class hierarchy:

```
class Pet { … }; // Same as before
template <typename Derived> class Visitable : public Pet {
    using Pet::Pet;
    void accept(PetVisitor& v) { v.visit(static_cast<Derived*>(this)); }
};
class Cat : public Visitable<Cat> {  // Pet is still the base class
    using Visitable<Cat>::Visitable;
    … class-specific code …
};
```

Write once per hierarchy

Boilerplate generator

- Almost CRTP but not quite

Visitor in Modern C++

- Visitor and client code:

```
auto v(lambda_visitor<PetVisitor>(
    [](Cat* c) { cout << "Let the " << c->color() << " cat out" << endl; },
    [](Dog* d) { cout << “Walk the " << d->color() << " dog“ << endl; }));
Pet* p = …;
p->accept(v);
```

- There is the small matter of implementation
- lambda_visitor<> is written only once (ever)
- PetVisitor is per hierarchy but is auto-generated

Visitor in Modern C++ - Implementation

- PetVisitor is the base class for all visitors in the hierarchy
  - It's essentially a typelist of all visited classes
  - It needs to be updated when a class is added, but in one place only!

```
template <typename ... Types> class Visitor;          // List of classes
template <typename T> class Visitor<T> { virtual void visit(T* t) = 0; };
template <typename T, typename ... Types>
class Visitor<T, Types ...> : public Visitor<Types ...> { // Recursive
   using Visitor<Types ...>::visit;
   virtual void visit(T* t) = 0;
};
using PetVisitor = Visitor<class Cat, class Dog>;
```

Forward declaration

Write once (ever)

Visitor in Modern C++ - Implementation

- Visitor relies on overload resolution

```
class FeedingVisitor : public PetVisitor {
    void visit(Cat* c) override;
    void visit(Dog* d) override;
};
```

- Lambda visitor uses lambda expressions instead of functions
- Lambda visitor relies on overload resolution of lambda expressions

Visitor in Modern C++ - Implementation

- Visitor relies on overload resolution

```
class FeedingVisitor : public PetVisitor {
    void visit(Cat* c) override;
    void visit(Dog* d) override;
};
```

- Lambda visitor uses lambda expressions instead of functions
- Lambda visitor relies on overload resolution of lambda expressions
- There is no overload resolution of lambda expressions

Divertimento – lambda overload resolution

- The idea is to create a class with overloaded operator():

```
template <typename ... F> struct overload_set : public F ... {
    overload_set(F&& ... f) : F(std::forward<F>(f)) ... {}
    using F::operator() ...; // C++17
};
```

- Also needs a helper function:

```
template <typename ... F> auto overload(F&& ... f) {
    return overload_set<F ...>(std::forward<F>(f) ...);
}
```

Divertimento – lambda overload resolution

- Use of the overload set:

```
auto l = overload(
    [](int i) { std::cout << "i=" << i << std::endl; },
    [](double d) { std::cout << "d=" << d << std::endl; }
);
l(5);
l(double(5));
l(float(5));
```

- Can be done in C++14 but does not handle ambiguous overloads as well

Lambda overload resolution in style

- C++17 all the way:

```
template <typename ... F> struct overload_set : public F ... {
    using F::operator() ...;
};
template <typename ... F> overload_set(F&& ... f) ->
    overload_set<F ...>;
```

- That’s not a helper function!
  - No function body, no return
- It’s a deduction guide
  - Creates fictional constructors and template deduction rules for them

Lambda overload resolution in style

- Use of the [fancy] overload set:

```
auto l = overload_set{
    [](int i) { std::cout << "i=" << i << std::endl; },
    [](double d) { std::cout << "d=" << d << std::endl; },
};
l(5);
l(double(5));
l(float(5));
```

- Exactly the same as before as far as the client code is concerned

Visitor in Modern C++ - Implementation

- Visitor relies on overload resolution

```
class FeedingVisitor : public PetVisitor {
    void visit(Cat* c) override;
    void visit(Dog* d) override;
};
```

- Lambda visitor uses lambda expressions instead of functions
- Lambda visitor relies on overload resolution of lambda expressions
  - We have it!

Visitor in Modern C++ - Implementation

- We need to iterate over the typelist hidden in PetVisitor
  — We can construct the lambda overload set along the way

```
template <typename Base, typename... > class LambdaVisitor;
```

- Primary template definition, never used

```
template <typename Base, typename T1, typename ... T,
                                      typename F1, typename ... F>
class LambdaVisitor<Base, Visitor<T1, T ...>, F1, F ...>;
```

- Specialization, uses two parameter packs: visitable types T and lambda expressions F

Visitor in Modern C++ - Implementation

- We need to iterate over the typelist hidden in PetVisitor
  — We can construct the lambda overload set along the way

```
template <class Base, class... > class LambdaVisitor;
template <…> class LambdaVisitor<…> :
  private F1, public LambdaVisitor<Base, Visitor<T ...>, F ...> {
    LambdaVisitor(F1&& f1, F&& ... f)
      : F1(std::move(f1)),
        LambdaVisitor<Base, Visitor<T ...>, F ...>(std::forward<F>(f) ...) {}
  void visit(T1* t) override { return F1::operator()(t); }
};
```

- Recursion must end somewhere

Visitor in Modern C++ - Implementation

- Recursion termination – last type in the list:

```
template <typename Base, typename T1, typename F1>
class LambdaVisitor<Base, Visitor<T1>, F1> : private F1, public Base
{
    LambdaVisitor(F1&& f1) : F1(std::move(f1)) {}
    void visit(T1* t) override { return F1::operator()(t); }
};
```

- Only the last class in the inheritance chain inherits from base
  — Base is the PetVisitor class, contains the list of visitable types

Visitor in Modern C++

From the earlier slide

```
using PetVisitor = Visitor<class Cat, class Dog>;
class Pet { … }; // Same as before
template <typename Derived> class Visitable : public Pet {
    using Pet::Pet;
    void accept(PetVisitor& v) { v.visit(static_cast<Derived*>(this)); }
};
class Cat : public Visitable<Cat> { using Visitable<Cat>::Visitable; };
auto v(lambda_visitor<PetVisitor>(
    [](Cat* c) { cout << "Let the " << c->color() << " cat out" << endl; },
    [](Dog* d) { cout << “Walk the " << d->color() << " dog“ << endl; }));
Pet* p = …; p->accept(v);
```

Write once per hierarchy

A Very C++17 Visitor

- C++17 has std::variant – alternative to polymorphism
- C++17 has std::visit – seems like a bold hint

A Very C++17 Visitor

- C++17 has std::variant and std::visit – we can build a visitor

```
using Pets = std::variant<class Cat, class Dog>;
template <typename Visitor, typename Pet>
void do_visit(const Visitor& v, const Pet& p) {
    std::visit(v, Pets{p});
}
class Cat {
    Cat(const std::string& color) : color_(color) {}
    const std::string& color() const { return color_; }
};
class Dog { … also has color() … };
```

Forward declarations

No visitation interface

Common base not required

A Very C++17 Visitor

- C++17 visitor – the client code

Lambda overload

```
auto pv = overloaded {
    [](const Cat& c) { std::cout << “Drive " << c.color() << " cat nuts“
        “ with the laser pointer” << std::endl; },
    [](const Dog& d) { std::cout << "Play fetch with the " << d.color() <<
        " dog" << std::endl; },
};
Cat c("orange");
Dog d("brown");
do_visit(pv, c); // std::visit(pv, Pets{c});
do_visit(pv, d);
```

A Very C++17 Visitor

- C++17 visitor – the client code

```
template <typename Pet> void walk(const Pet& p) {
    auto v = overloaded {
        [](const Cat& c) { std::cout << "Let the " << c.color() << " cat out“
            << std::endl; },
        [](const Dog& d) { std::cout << "Take the " << d.color() <<
            " dog for a walk" << std::endl; },
    };
    std::visit(v, Pets{p});
}
Cat c("orange"); Dog d("brown");
walk(c); walk(d);
```

A Very C++17 Visitor

- std::variant is used instead of object polymorphism
  - No need for a single hierarchy
- Visitation is not routed through the accept() method

A Very C++17 Visitor

- std::variant is used instead of object polymorphism
  - No need for a single hierarchy
- Visitation is not routed through the accept() method
- Harder to write composable visitors:

```
class Family {
    Cat cat_; Dog dog_;
    void accept(PetVisitor& v) { cat_.accept(v); dog_.accept(v); }
};
```

- Serialization/deserialization visitors often use this pattern

Does Language Development Affect Pattern Use?

- C++03 supports the classic OOP visitor (also acyclic visitor)
  - Fair amount of copy-paste boilerplate
  - C++11 removes most of the boilerplate
- C++14 allows visiting lambda expressions
  - With some limitations on overloading (less important for Visitor)
  - Made slightly more compact in C++17
- Definitely much less friction in newer C++ versions
  - Nothing truly radical, but you have to compare with your alternatives
  - Visitor may become easier than the alternative
- C++17 allows visiting variants instead of class hierarchies
  - Has advantages and tradeoffs

SCOPEGUARD

Exception Handling

```
class Record { ... };
class Database {
    void insert(const Record& r);
};
```

- To the caller, insert() appears to be a transaction
- It is reasonable to expect transactional behavior
  - insert() either succeeds and inserts the record, or fails and nothing happens to the database (exception is thrown)

~~Exception~~ Error Handling

```
class Record { ... };
class Database {
    int insert(const Record& r);
};
```

- To the caller, insert() appears to be a transaction
- It is reasonable to expect transactional behavior
  - insert() either succeeds and inserts the record, or fails and nothing happens to the database (exception is thrown)
- It’s not about exception handling but error handling

Transactions Are Hard

```
class Database {
    class Storage { ... }; // Disk storage
    Storage S;
    class Index { ... }; // Memory index
    Index I;
    void insert(const Record& r) {
        S.insert(r);
        I.insert(r);
};
```

- The implementation does not guarantee the atomic transaction

Transactions Are Hard

```
class Database {
    class Storage { ... }; // Disk storage
    Storage S;
    class Index { ... }; // Memory index
    Index I;
    void insert(const Record& r) {
        S.insert(r);    <- Fails (throws exception)
        I.insert(r);
};
```

- Nothing happens if the first step fails – so far so good

Transactions Are Hard

```
class Database {
    class Storage { ... }; // Disk storage
    Storage S;
    class Index { ... }; // Memory index
    Index I;
    void insert(const Record& r) {
        S.insert(r);    // Already done!
        I.insert(r);    // Fails (throws exception)
};
```

- Storage is altered if the second step fails – database corrupted

Transactions Are Easy

```
void Database::insert(const Record& r) {
    S.insert(r);
    try { I.insert(r);}
    catch (...) {
        S.undo();
        throw; // Rethrow
    }
};
```

Assume undo is possible

- Either all necessary changed are done, or none of them
  - The invariant of the database is maintained
- Exceptions are not required, error codes are handled the same way

Transactions Are Hard

- Undo is hard to do, but easier if you have a point of no return
  — insert() or undo() must be followed by finalize()

```
void Database::insert(const Record& r) {
    S.insert(r);
    try { I.insert(r); }
    catch (...) {
        S.undo();
        S.finalize();
        throw;
    }
    S.finalize();
}
```

It's going to be worse if there are more steps

A Three-Step Transaction

```
if (action1() == SUCCESS) {
    if (action2() == SUCCESS) {
        if (action3() == FAIL) {
            rollback2();
            rollback1();
        }
        cleanup2();
    } else {
        rollback1();
    }
    cleanup1();
}
```

- Ugly
- Requires copy-paste
- Gets worse with more steps
- Not composable:
  - The solution for N steps is not the solution for N-1 steps with some more code added
  - To add a step, inner code has to be modified

There Is a Pattern For That

- Resource Acquisition is Initialization (RAII)

```
class StorageFinalizer {
    StorageFinalizer(Storage& S) : S_(S) {}
    ~StorageFinalizer() { S_.finalize(); }
    Storage& S_;
};
```

- In this case, more like Resource Release is Destruction
  — finalize() happens whenever the finalizer is destroyed

There Is a Pattern For That

- Resource Acquisition is Initialization (RAII)

```
class StorageFinalizer { … };
void Database::insert(const Record& r) {
    S.insert(r);
    StorageFinalizer SF(S);
    try { I.insert(r); }
    catch (...) {
        S.undo();
        throw;
    }
}
```

- Better, but only so much
  - finalize() is hidden and automated
  - undo() is not

A Three-Step Transaction

```
action1();
Cleanup1 c1; // RAII for cleanup1()
try { action2();
    Cleanup2 c2;
    try { action3(); }
    catch (...) {
        rollback2();
        throw;
    } catch (...) {
        rollback1();
    }
}
```

- Still not composable:
  — rollbackN() is inserted into the innermost scope

A Three-Step Transaction

```
action1();
Cleanup1 c1;
try { action2();
    Cleanup2 c2;
    try { action3(); }
    catch (...) {
        rollback2();
        throw;
    } catch (...) {
        rollback1();
    }
}
```

- Still not composable:
  - rollbackN() is inserted into the innermost scope
- But consider the success (cleanup) path by itself

We’re Onto Something here

- Cleanup path is perfectly composable:

```
action1();
Cleanup1 c1;
action2();
Cleanup2 c2;
…
actionN();
CleanupN cN;
```

- Cleanup path is hidden in RAII objects
  - Rollback path is explicit

There Is a Pattern For That

```
class StorageGuard {
    StorageGuard(Storage& S) : S_(S), commit_(false) {}
    ~StorageGuard() { if (!commit_) S_.undo(); }
    void commit() noexcept { commit_ = true; }
    Storage& S_;
    bool commit_;
};
```

- StorageGuard is similar to StorageFinalizer
  - Except the destructor action is conditional
  - Cleanup always happens, rollback happens only on failure

There Is a Pattern For That

```
void Database::insert(const Record& r) {
    S.insert(r);
    StorageFinalizer SF(S);     // Arm cleanup action
    StorageGuard SG(S);         // Arm rollback action (hope to fail?)
    I.insert(r);
    SG.commit();                // Disarm rollback if we didn't fail
}
```

- No try-catch blocks!
- Catch exceptions only to prevent them from propagating
  - Never to execute exception-only code
- Declarative programming (state your intent, and magic happens)

Problems with RAII

```
class StorageGuard {
    StorageGuard(Storage& S) : S_(S), commit_(false) {}
    ~StorageGuard() { if (!commit_) S_.undo(); }
    void commit() noexcept { commit_ = true; }
    Storage& S_;
    bool commit_;
};
```

- RAII classes have to be written for every task
- RAII classes have boilerplate code to capture external variables
- RAII classes are not trivial to write correctly (this one is wrong)

Problems with RAII

```
class StorageGuard {
    StorageGuard(Storage& S) : S_(S), commit_(false) {}
    ~StorageGuard() { if (!commit_) S_.undo(); }
    void commit() noexcept { commit_ = true; }
    Storage& S_;
    bool commit_;
    StorageGuard(const StorageGuard&) = delete;
    StorageGuard& operator=(const StorageGuard&) = delete;
};
```

- RAII classes are not trivial to write correctly
  - Really bad things happen if RAII classes are copied

The ScopeGuard

- The ScopeGuard pattern has two elements:
  - The optimal approach to the cleanup/rollback problem – we have already seen it (applies to any deferred action)
  - The recommended implementation – we are about to see it
- This is what we want:

```
S.insert(r);
ScopeGuard SF(finalize, S);
ScopeGuard SG(undo, S); // Or something like this
I.insert(r);
SG.commit();
```

The ScopeGuard

- The ScopeGuard pattern has two elements:
  - The optimal approach to the cleanup/rollback problem – we have already seen it (applies to any deferred action)
  - The recommended implementation – we are about to see it
- This is what we really want:

```
S.insert(r);
ScopeGuardFail SG(undo, S); // Or something like this
ScopeGuardSuccess SF(finalize, S);
I.insert(r);
// No explicit SG.commit() – committed if no exceptions thrown
```

ScopeGuard in C++03 – Client code

```
S.insert(r);
const ScopeGuardImplBase& SG = MakeGuard(undo, S);
const ScopeGuardImplBase& SF = MakeGuard(finalize, S);
I.insert(r);
SG.commit();
```

- Explicit commit
- MakeGuard helper function
  - Fixed number of arguments (one object, one function)
- Cheating alert: this implementation calls a non-member function

```
void undo(Storage& S) { S.undo(); }
```

ScopeGuard in C++03 - Implementation

```
template <typename Func, typename Arg>
ScopeGuardImpl<Func, Arg> MakeGuard(const Func& f, Arg& arg) {
    return ScopeGuardImpl<Func, Arg>(f, arg);
}
```

- Cheating alert: this implementation calls a non-member function

```
void undo(Storage& S) { S.undo(); }
```

- Member function guard is possible but the syntax is even more verbose

ScopeGuard in C++03 - Implementation

```
template <typename Func, typename Arg>
class ScopeGuardImpl : public ScopeGuardImplBase {
    public:
    ScopeGuardImpl(const Func& f, Arg& arg) : func_(f), arg_(arg) {}
    ~ScopeGuardImpl() { if (!commit_) func_(arg_); }
    private:
    const Func& func_;
    Arg& arg_;
};
```

ScopeGuard in C++03 - Implementation

```
class ScopeGuardImplBase {
  public:
  ScopeGuardImplBase() : commit_(false) {}
  void commit() const throw() { commit_ = true; }
  protected:
  ScopeGuardImplBase(const ScopeGuardImplBase& other)
    : commit_(other.commit_) { other.commit(); }
  ~ScopeGuardImplBase() {}
  mutable bool commit_;
  private:
  ScopeGuardImplBase& operator=(const ScopeGuardImplBase&);
};
```

ScopeGuard in C++1X

- C++11: real move constructor, variadic templates
- C++14: return type deduction in functions
- C++17: template type deduction in constructors
- It's still ugly and somewhat limiting, there is a better way:
- Lambda expressions!
  - ScopeGuard pattern is radically changed by C++11 and again C++17

ScopeGuard in C++11/14

- Instead of writing RAII classes we can use lambda expressions:

```
S.insert(r);
auto SF = MakeGuard([&] { S.finalize(); });
auto SG = MakeGuard([&] () { S.undo(); });      // () is optional
I.insert(r);
SG.commit();                                    // Still explicit commit
```

- Automatic variable capture (S)
- Any cleanup/rollback code, no limitations on the number of arguments or types of functions to call

ScopeGuard in C++17

- Instead of writing RAII classes we can use lambda expressions:

```
S.insert(r)
ScopeGuard SF([&] { S.finalize(); });
ScopeGuard SG([&] { S.undo(); });
I.insert(r);
SG.commit();                    // Optional
```

- No MakeGuard function
- Automatic success/failure detection is possible
  - Only if failure means exception and success means return (any return)

ScopeGuard in C++1x - Implementation

```
class ScopeGuardBase {
    public:
    ScopeGuardBase() : commit_(false) {}
    void commit() noexcept { commit_ = true; } // Not const now
    protected:
    ScopeGuardBase(ScopeGuardBase&& other)  // Real move ctor!
        : commit_(other.commit_) { other.commit(); }
    ~ScopeGuardBase() {}
    bool commit_;                   // Not mutable anymore
    ScopeGuardBase& operator=(const ScopeGuardBase&) = delete;
};
```

ScopeGuard in C++1x - Implementation

```
template <class Func> class ScopeGuard : public ScopeGuardBase {
public:
    ScopeGuard(Func&& func) : func_(func) {}
    ScopeGuard(const Func& func) : func_(func) {}
   ~ScopeGuard() { if (!commit_) func_(); }
   ScopeGuard(ScopeGuard&& other)
      : ScopeGuardBase(std::move(other)), func_(other.func_) {}
   private:
   Func func_;
};
```

ScopeGuard in C++11/14 - Implementation

```
template <typename Func>
ScopeGuard<Func> MakeGuard(Func&& func) {
    return ScopeGuard<Func>(std::forward<Func>(func));
}
```

- In C++17 the factory function is not needed
  - Constructors deduce template parameters

ScopeGuard in C++1X

```
action1();
ScopeGuard  cleanup1([&] { … });
ScopeGuard  rollback1([&] { … });
action2();
ScopeGuard  cleanup2([&] { … });
ScopeGuard  rollback2([&] { … });
action3();
rollback1.commit();
rollback2.commit();
```

- Composable ScopeGuard – just add actionN(), commit/rollback guards, and commit() at the end

ScopeGuard and Exceptions

- In C++, only one exception can propagate at any time
  — Not "only one exception can be throw at any time"

```
void exception_handler(…) { // Called if exception X is thrown
    try { … throw 1; } catch ( int ) { };
}  // Only exception X is propagating - OK
```

- This presents problems for ScopeGuard:
  — Rollback (undo) must not throw (if action fails and undo fails, then what?)
  — If rollback throws, we either die, or ignore the exception (shielded guard)
- Commit also must not throw, otherwise false rollback will run
  — This part is easy, it's just a flag set

ScopeGuard and Exceptions

```
{
    S.insert(r);
    ScopeGuard SF([&] { S.finalize(); });
    ScopeGuard SG([&] { S.undo(); });
    I.insert(r);
    SG.commit();   // Why?!
}  // run finalize(), maybe run undo()
```

Do we know how we got here?

- std::uncaught_exception() – return true iff exception is currently propagating
  - Almost enough to auto-detect commit

ScopeGuard and Exceptions

```
void exception_handler( … ) {        // Called if exception was thrown
    S.insert(r);
    ScopeGuard SF([&] { S.finalize(); });
    ScopeGuard SG([&] { S.undo(); });
    I.insert(r);
    //SG.commit(); // We have std::uncaught_exception(), ha!
}  // run finalize(), maybe run undo()
```

Exception is already propagating here

Nothing fails here, should commit

Exception is still propagating here

- std::uncaught_exception() – return true iff exception is currently propagating
  - If exception was already propagating when a ScopeGuard was armed, rollback will occur even when guarded actions succeeded

ScopeGuard and Exceptions in C++17

```
void exception_handler( … ) {        // Called if exception was thrown
    S.insert(r);
    ScopeGuard SF([&] { S.finalize(); });
    ScopeGuardFail SG([&] { S.undo(); });
    l.insert(r);
}  // run finalize(), maybe run undo()
```

N exceptions are already propagating here

Nothing fails here, should commit

Still N exceptions – success!

- std::uncaught_exceptions() – return the count of exceptions currently propagating
  - If new exception was thrown since ScopeGuard was armed, it's a failure

ScopeGuard and Exceptions in C++17

```
class UncaughtExceptionDetector {
    const int c_;
    public:
    UncaughtExceptionDetector() : c_(std::uncaught_exceptions()) {}
    operator bool() const noexcept {
        return std::uncaught_exceptions() > c_;
    }
};
```

- Helper class to count exceptions

ScopeGuard and Exceptions in C++17

```
template <typename Func> class ScopeGuardFail {
    Func func_;
    UncaughtExceptionDetector detector_;
    public:
    ScopeGuardFail(Func&& func) : func_(func) {}
    ScopeGuardFail(const Func& func) : func_(func) {}
    ~ScopeGuardFail() { if (detector_) func_(); }
    ScopeGuardFail(ScopeGuardFail&& other) : func_(other.func_) {}
};
```

- Scope guard for rollback and other failure handling actions

ScopeGuard and Exceptions in C++17

```
template <typename Func> class ScopeGuard {
    Func func_;
    public:
    ScopeGuard(Func&& func) : func_(func) {}
    ScopeGuard(const Func& func) : func_(func) {}
    ~ScopeGuard() { func_(); }
    ScopeGuard(ScopeGuard&& other) : func_(other.func_) {}
};
```

- Scope guard for cleanup and other actions that always happen

ScopeGuard and Exceptions in C++17

```
{
    S.insert(r);
    ScopeGuard SF([&] { S.finalize(); });          // Always happens
    ScopeGuardFail SG([&] { S.undo(); });         // If I.insert() throws
    I.insert(r);
}
```

- Composable – just add action and guards
- Client code is as simple as it gets
  - But only if success == exception
- Would this be enough to make you change your error handling to use exceptions only?

Does Language Development Affect Pattern Use?

- C++03 has just enough clever hacks to implement a mostly working ScopeGuard
  — No major problems but a lot of minor annoyances (i.e. friction)
- C++14 removes most of the friction by using lambda expressions
- C++17 supports automatic detection of success or failure on exit
  — Only if exceptions are used for any failure
- The pattern is much easier to use, may influence design decisions

STRATEGY

Strategy Pattern

- Enables run-time selection of a specific algorithm for a particular behavior
- Also known as the Policy Pattern
- In C++ is mostly used at compile-time

POLICY-BASED DESIGN

Policy-Based Design

template <class T, class DeletePolicy, class CopyPolicy,
  class MovePolicy, class DebugPolicy> class SuperSmartPtr { … };

- Each policy controls the specific behavior aspect
- Each policy can have a default
  - Changing 15th policy requires repeating all preceding defaults

Policy-Based Design

template <class T, class DeletePolicy, class CopyPolicy,
class MovePolicy, class DebugPolicy> class SuperSmartPtr { … };

- Each policy controls the specific behavior aspect
- Each policy can have a default
  - Changing 15th policy requires repeating all preceding defaults
- Wait, what?! 15th? – policy customization aspects tend to increase
- In practice, everyone needs a small subset of policy options
  - Typical death spiral of policy-based designs: number of policies grows until everything has a policy, then nobody uses policy-based types because of all the policies they don't need but have to specify
- Solution is aliases for policy types

Policy Aliases in C++03

```
template <class T, class DebugPolicy> class MyPtr :
    public SuperSmartPtr<T, ArrayDeleter, NoCopy,
                                    MoveOK, DebugPolicy> {
    MyPtr(T* p) …;
    MyPtr(const MyPtr&) … and MyPtr&& … and more
};
```

- All constructors must be repeated in all derived classes
  - Seems trivial, in practice may be enough friction to make alternatives preferable

Policy Aliases in C++1X

```
template <class T, class DebugPolicy> class MyPtr :
    public SuperSmartPtr<T, ArrayDeleter, NoCopy,
                                        MoveOK, DebugPolicy> {
    Using SuperSmartPtr<… all template args …>::SuperSmartPtr;
};
```

- All constructors can be “resurrected” at once
- Template arguments must be repeated

Policy Aliases in C++1X

```
template <class T, class DebugPolicy>
using MyPtr = SuperSmartPtr<T, ArrayDeleter, NoCopy,
                            MoveOK, DebugPolicy;
```

- Nothing to repeat, very compact
- But no C++17 constructor argument deduction for template aliases

```
MyPtr p(new A, VeryVerboseLogger());   // Does not work
```

Does Language Development Affect Pattern Use?

- Policy-based design works in C++03
  - Trade-offs and drawbacks are known but not always avoided
  - In practice, policy-based design tend to evolve toward unmanageable complexity
  - Sometimes simple commonly used classes are reimplemented even when they are really a particular case of a 15-parameter policy template
- C++14/17 enhancements are not major but practically significant
  - Some of the drawbacks become less severe
  - Complexity is easier to manage

EVOLUTION OF LANGUAGE AND OF DESIGN PATTERNS

Do Design Patterns Depend on the Language? Does Language Development Affect Pattern Use?

- Design patterns apply to software design and transcend language
- Design pattern drawbacks/trade-offs are more language-dependent
- Drawbacks are not to be analyzed in abstract
  - You have a problem, you need a solution, you'll have to choose one
  - Each solution has some drawbacks and trade-offs
- Language development helps some patterns more than others
  - There is always friction in the use of the patterns (price of complexity)
  - "Small stuff" done many times every day matters in practice
- A "tipping point" in the balance of different patterns may lead to a completely different design

Mentor®

A Siemens Business

www.mentor.com