Practical Design Patterns in the C++ Context

Patterns After the Fact

Legal Stuff

2

Legal Stuff

- If you have not attended this course, you may possess these notes provided you acquired them directly from Saks & Associates, or:
  - You have acquired them, either directly or indirectly, from someone who has (1) attended the course, or (2) paid to attend it at a conference, or (3) licensed the material from Saks & Associates.
  - The person from whom you acquired the notes no longer possesses any copies.
- If you would like permission to make additional copies of these notes, contact Saks & Associates.

3

About the Author

Steve Dewhurst is the cofounder and president of Semantics Consulting, Inc. He is the author of *C++ Common Knowledge* (Addison-Wesley, 2005), *C++ Gotchas* (Addison-Wesley, 2003), and the co-author of *Programming in C++* (Prentice Hall, 2nd edition 1995). He has also written numerous technical articles on C++ programming techniques and compiler design.

Steve has taught extensively in both university and commercial settings. He is a frequent and highly-rated speaker at industry conferences such as *Software Development* and *Embedded Systems*. He was the C++ training series adviser for Technology Exchange Company (Addison-Wesley).

Steve has mentored and consulted for projects in areas such as compiler design, embedded telecommunications, e-commerce, and derivative securities trading.

4

About the Author

As a Member of Technical Staff at AT&T Bell Laboratories, Steve worked with Bjarne Stroustrup, the designer of C++, on the first public release of the C++ language and cfront compiler. He later served as the lead designer and implementer of AT&T's first non-cfront C++ compiler. As a compiler architect at Glockenspiel, Ltd., he designed and implemented a second C++ compiler.

Steve is an advisory board member for the on-line publication *The C++ Source*. He has also been a contributing editor for *The C/C++ Users Journal*, a member of the editorial board of and columnist for *The C++ Report*, and co-founder and editorial board member of *The C++ Journal*.

5

Contents

6

Some Design Pattern Concepts

7

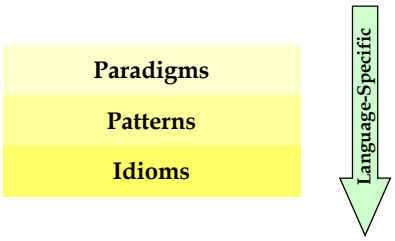
Roadmap

- What are design patterns?
- Pattern elements and description
- Where patterns come from
- Patterns culture
- Finding the right pattern

8

What Is A Design Pattern?

- A recurring architectural theme.
- Provides a solution to a common design problem within a particular context, and describes the consequences of this solution.
- A pattern is more than a simple technical description of a technique.



9

A Pattern Definition

"...descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

GOF, **Design Patterns**

10

Essential Pattern Elements

- Every pattern description must contain the following:

✓ *Name*
  - an *unambiguous* handle by which we refer to the pattern

✓ *Problem*
  - the problem that the pattern addresses

✓ *Solution*
  - the technique for solving the problem

✓ *Consequences*
  - how the context is changed, for better or worse, after application of the technique

- There are several common pattern forms.
  - The "gang of four" used a very ornate description.
  - We'll use a simplified form of GOF to introduce each pattern.

11

Factory Method

Pattern name: very important!

- Category
  - Class Creational

Pattern category: idiosyncratic, of limited use, but conventional.

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Variances
  - Subclass of object that is instantiated.

What aspects of my design will be made more flexible, and in what way?

- Redesign Problems Solved
  - Creating an object by specifying a class explicitly.
- See Also
  - Abstract Factory
  - Template Method
  - Prototype

Earliest appearance of "code smell" concept, hint at "refactoring into patterns."

Can be useful advice, if this pattern doesn't apply in this context. Also, may indicate presence of a "compound pattern."

12

Patterns and Design

- Patterns are composable "micro-architectures."
- Patterns describe variances over their structure.
- Many programmers write only a "single program." Patterns can help.

13

Where Patterns Originate, Patterns Culture

- Patterns are not invented, they are "mined."
- A pattern requires the "rule of three."
- Pattern discovery is difficult, time-consuming, and worthwhile.
- If you have a great idea, it's not a pattern, it's a great idea. (It might be a protopattern.)
- A pattern author is not claiming to have invented the technique.
- The patterns community supports "aggressive disregard for originality."

✓ *Simple patterns are important.*

14

How to Find the Right Pattern

- The *intent* of a pattern is often more important than its structure.
- Consider how you expect the architecture to be extended over time; what are the likely "variances."
- Pattern catalogs don't help much.
- Internalizing a set of patterns works.

15

Patterns and Frameworks

- What frameworks are
- Customizing frameworks: derivation, callbacks, and the Hollywood principle, "Don't call us, we'll call you!"
- Substitutability
- Hierarchy design: base classes and contracts
- Moving conditional code to the hierarchy
- Maintaining framework-oriented code
- Planning ahead: patterns and variances; code for today, design for tomorrow

16

Some SOLID Principles, Briefly

- ***S***ingle Responsibility Principle
- ***O***pen-Closed Principle
- ***L***iskov Substitution Principle
- ***I***nterface Segregation Principle
- ***D***ependency Inversion Principle

17

Single Responsibility Principle

- Bob Martin: "A class should have only one reason to change."
- A "responsibility" is a "reason to change."
- A common effect of violating SRP is excessive coupling.
  - A stock class that has a "print" function couples the stock to (for example) iostream.
  - However, the stock class may be used to model pricing algorithms with no need to print.
  - Nevertheless, the modeling application must include iostream.
- Some good advice from Martin: "It is not wise to apply SRP, or any other principle for that matter, if there is no symptom."

18

Open-Closed Principle

- Bertrand Meyer: Software entities should be open for extension, but closed for modification.
- This is one of the guiding principles in framework design.
- Addressed explicitly in pattern description of a pattern's variances.
- The "closed for modification" is context-dependent, and may indicate that no source code may be changed, that no source code may be recompiled, etc.

19

Liskov Substitution Principle

- Barbara Liskov: "Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T."
- Two quotes from Cline and Lomow (C++ FAQs) that succinctly express the motivation behind substitutability:
  - "A derived class shouldn't shock users of the base class."
  - A derived class should "require no more, promise no less" than its base class.
- More generally, note that a polymorphic object will be manipulated at various times through different type interfaces. Its behavior must be consistent no matter which interface is used to manipulate it.

20

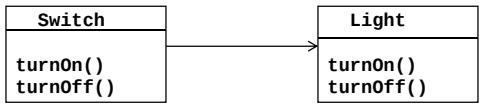
Interface Segregation Principle

- Bob Martin: "Clients should not be forced to depend upon interfaces they do not use."
- "Make find-grained interfaces that are client-specific."
- Clients apply forces on interfaces; separate clients mean separate interfaces.
- A common effect of violation of ISP are wide base class interfaces.
- Another effect is that particular parts of the interface are used by few clients.

21

Abstract Server "Pattern"

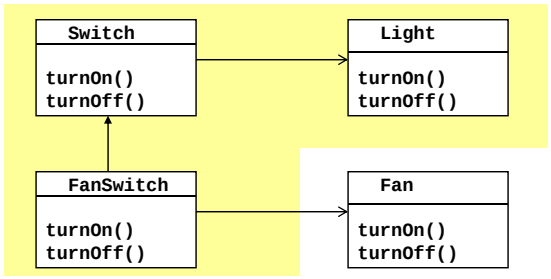
- This is really more of a design guideline than a pattern…
- The motivating issue concerns the ownership of an interface.
- Consider a switch control that manipulates a light.



22

Abstract Server

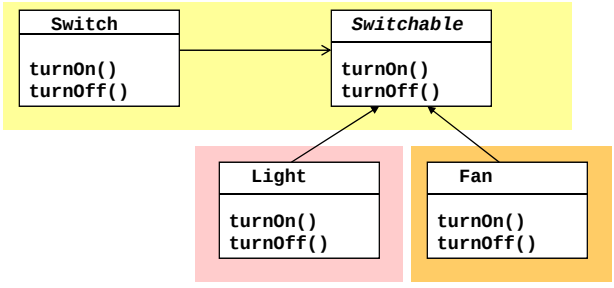
- This design makes it hard to reuse Switch, and impossible to reuse it without dragging Light along. They're coupled.



23

Abstract Server Design Implications

- A better design would be to allow a Switch to manipulate anything that's Switchable.



- Switch now depends on the abstract Switchable interface, not the concrete Light.

24

Dependency Inversion Principle

- Bob Martin: "Depend upon abstractions, not on concretions."
- Interfaces belong to the client of the interface, not to the derived classes.
- Implication: Package the interface with the client of the interface.

✓ *The derived classes are ignorant of each other.*

✓ *The interface is ignorant of the derived classes.*

✓ *Ignorance is bliss.*

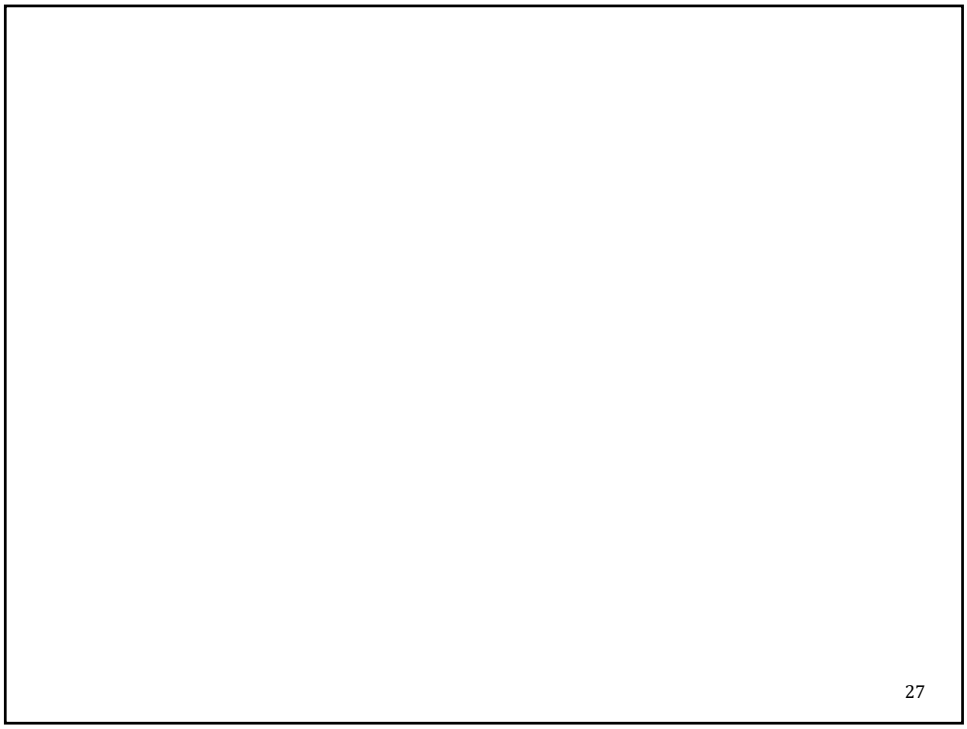
25

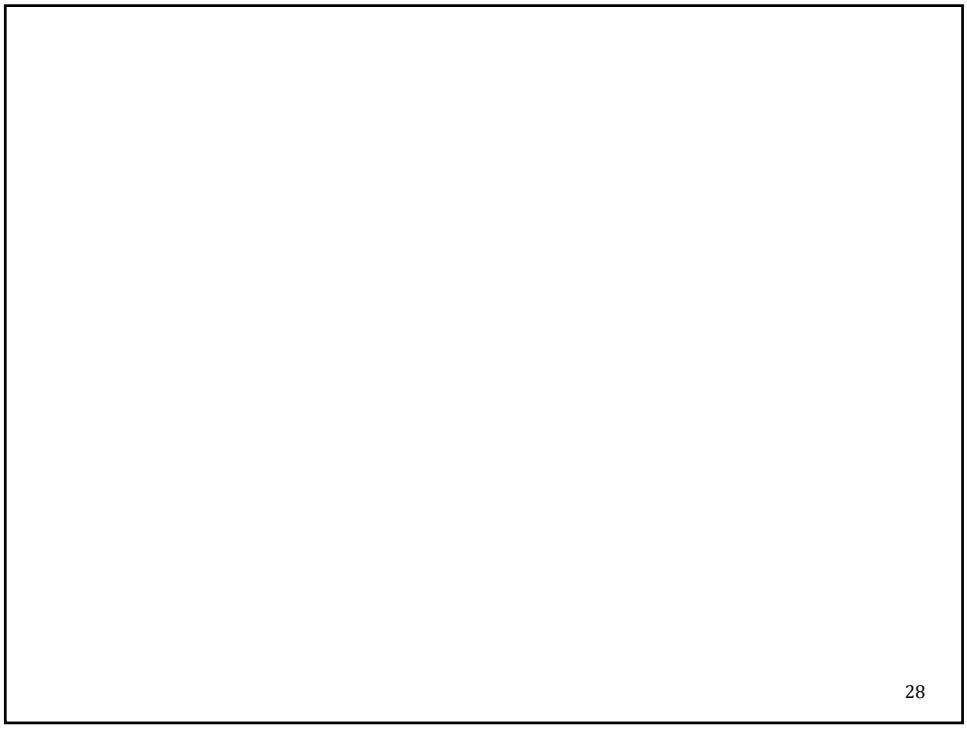
But Seriously…

"The most important thing is that you take the pattern seriously. There is no point at all in using the pattern if you only give lip service to it."

Christopher Alexander, *The Timeless Way of Building*

26





Patterns and Hierarchy Design

29

Bossy Bases

- Well-designed base classes tell derived classes how they may customize or extend the base class.

```
class Base {
public:
    virtual ~Base(); // I'm a polymorphic base class
    virtual bool verify() const = 0; // you must do this
    virtual void doit(); // do it your way or mine
    const char *id() const; // do it exactly this way
    void jump(); // here's how you must jump, but...
protected:
    virtual double doHowHigh() = 0; // ...you have some
    virtual int doHowManyTimes();   // limited freedom
};
```

30

Polymorphic Base Classes

- A public, virtual destructor means a class is a polymorphic base class.

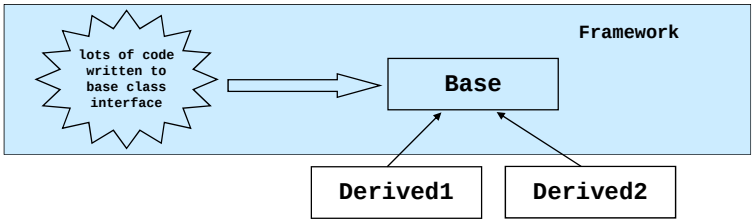
✓ *Polymorphic base classes have public, virtual destructors.*

- A non-virtual member function is an invariant over the hierarchy.
- A virtual function is a required interface with a default implementation. The derived class must verify that the default implementation is appropriate or supply its own.
- A pure virtual function is a required interface. A concrete derived class must supply an appropriate implementation.
- A Template Method fixes the overall structure of a base class member function, but allows or requires derived classes to customize individual steps.

31

The Contract

- A base class establishes a contract between generic code written to the contract and derived classes that implement the contract.



- The generic code knows nothing about the derived classes.
- The generic code may have been compiled long before the derived classes existed.
- The authors of the generic code and base class may have no knowledge of or control over the derived classes.
- The contract provided by the base class is what allows the derived classes and generic code to work together.

32

Abstract and Concrete Classes

- An abstract class cannot be used to instantiate an object.
- Polymorphic base classes represent abstract concepts, and should therefore be abstract.
- Polymorphic code should be written to a base class's interface, without making the assumption that it is dealing precisely with a base class.
- Concrete base classes may give rise to low-level problems.
  - Slicing!
  - Containers of base class objects.

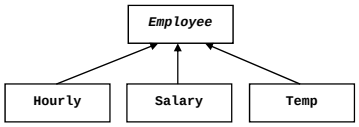
✓ *Class hierarchies should be designed with abstract base classes and concrete leaves.*

- This means your base classes can (and generally should) have protected copy operations.

33

Where Do Hierarchies Come From?

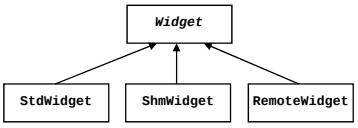
- We may recognize a hierarchy from the top, through specialization.
  - "Our application deals with employees." "What kind of employees are there?" "The usual: hourly, salaried, and probably some others in the future."
- We may recognize a hierarchy from the bottom, through abstraction.
  - "I've got a class table, a function table, and a global table, and they all have different implementations." "Have they got anything in common?" "Well, they all behave like symbol tables."



34

Where Do Hierarchies Come From?

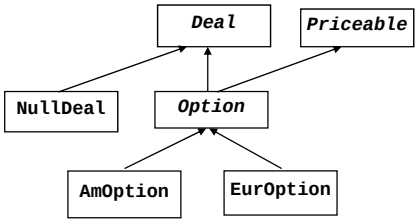
- We may recognize a hierarchy late in development, from implementation issues.
  - "I've got a Widget object that may be in my local memory, in shared memory, or on another node in the network. I'm getting pretty tired of special casing every time I want to access a Widget." "Don't."



35

The Meaning of Polymorphism

- Consider a type of financial option, `AmOption`. It is simultaneously an `AmOption`, an `Option`, a `Deal`, and a `Priceable`.
- This means it can respond to messages sent to any of its four interfaces.



- This means that an `AmOption` can leverage generic code written to any of its base classes' interfaces.
- Our hierarchy design heuristics tell us how to craft class hierarchies to make this possible.

36

Polymorphic Objects

- An object should exhibit the same behavior no matter which of its interfaces is used to manipulate it.

```
void process(Option *option) {
    option->price();          // some behavior
}

AmOption *amop = new AmOption;
process(amop);                // exhibit some behavior
amop->price();                // should be same behavior!
```

- Polymorphic objects are manipulated through several interfaces.
- Their behavior must be consistent no matter which interface is used.

37

Type-Based Conditionals

- We don't switch on type codes in object-oriented programs.

```
void process(Employee *e) {
    switch (e->type()) {       // evil code!
    case SALARY:
        fireSalary(e); break;
    case HOURLY:
        fireHourly(e); break;
    case TEMP:
        fireTemp(e); break;
    default:
        throw UnknownEmployeeType();
}
```

38

Type-Based Conditionals

- The polymorphic approach is more appropriate.

```
void process(Employee *e)
    { e->fire(); }
```

- The advantages are enormous:
  - It's simpler.
  - It doesn't have to be to be recompiled as new types are added.
  - It is impossible to have type-based runtime errors.
  - It's probably faster and smaller!

✓ *Implement type-based decisions with dynamic binding, not with conditional control structures.*

39

Avoiding Decisions with Dynamic Binding

- One way to avoid making an incorrect decision is not to make a decision.
- Many conditional constructs can be "encoded" in a class hierarchy.
- We effectively convert conditional code into type-based code.

✓ *Convert conditional control structures into type-based decisions where appropriate.*

40

Broadcast Failure

- Type-based conditional code affects more than just the function in which it appears.
- Type-based conditional code can fail.

```
void process(Employee *e) {
    switch (e->type()) {       // evil code!
    ~~~
    default:
        throw UnknownEmployeeType();
}
```

- This possibility of failure will affect all users of the function:
  - users will have to check for success/failure, or (sometimes)
  - users will not check and be buggy.

41

Cosmic Hierarchies

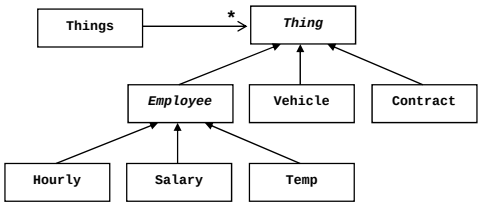
- Overly-inclusive hierarchies are generally bad design.
- Such hierarchies tend to give rise to "containers of anything."
- This gives rise to conditional code that is particularly inefficient, hard to maintain, and prone to error.

✓ *Avoid cosmic hierarchies.*

42

A Cosmic Hierarchy

- In "containers of anything," type information is lost, and must be recovered through conditional code.
  - *Type-based conditional code.*
- "Ok, thing, I'm going to process you. Are you a vehicle?" "No." "All right, are you a contract?" "Nope." "Well, perhaps you're an employee?" "Wrong again." "I give up!"



43

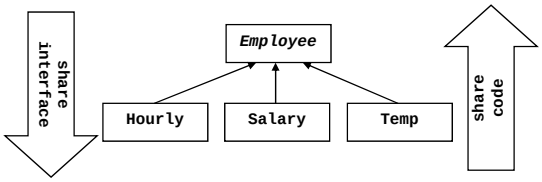
```
void process(Thing *a) {
    if (Vehicle *v = dynamic_cast<Vehicle *>(a))
        v->drive();
    else if (Contract *c = dynamic_cast<Contract *>(a))
        c->enforce();
    else if (Employee *e = dynamic_cast<Employee *>(a))
        e->fire();
    else
        throw UnknownThing(a);
}

void doThings(list<Thing *> &things) {
    for (auto i : things) {
        try { process(i); }
        catch (UnknownThing &ut) { /* ??? */ }
}
```

44

Hierarchies and Reuse

- Class hierarchies promote reuse in two ways.
  - code sharing
  - interface sharing
- We get code sharing by putting common derived class implementations in base classes. This is good.



- We get interface sharing by writing substitutable derived classes. This is better.
- Interface sharing is more important than code sharing. Don't sacrifice the base class interface in order to share code.

45

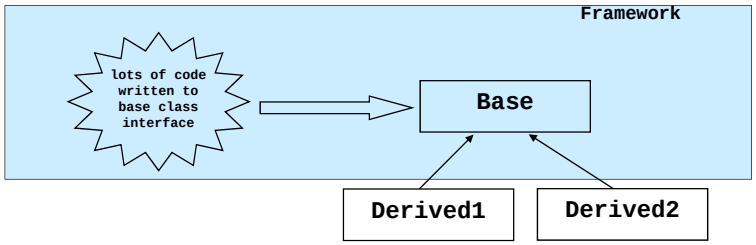
The Contract

- A base class establishes a contract between polymorphic code written to the contract and derived classes that implement the contract.
- The polymorphic code knows nothing about the derived classes.
- The polymorphic code may have been compiled long before the derived classes existed.
- The authors of the polymorphic code and base class may have no knowledge of or control over the derived classes.
- The contract provided by the base class is what allows the derived classes and polymorphic code to work together.

46

The Contract

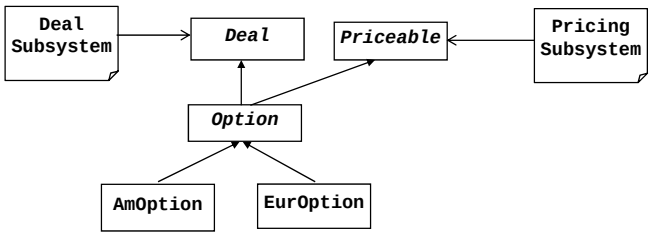
- Base class design is about writing clear contracts.
- Derived class design is about fulfilling base class contracts.
- The base class is ignorant of its derived classes.



47

Contracts and Leveraging Generic Code

- A base class specifies a contract.
  - generic code is written to the base class interface
  - derived classes customize the generic code by being substitutable for the base class

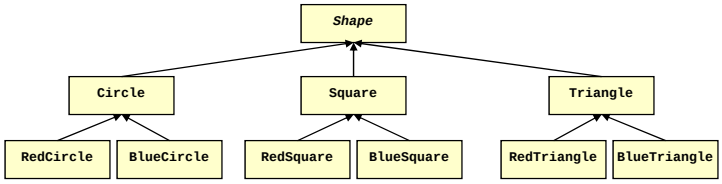


- The greatest reuse is achieved by leveraging entire subsystems with substitutable derived classes.

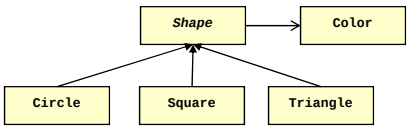
48

Exponentially Expanding Hierarchies

- A common error among new OO designers is to overuse inheritance.



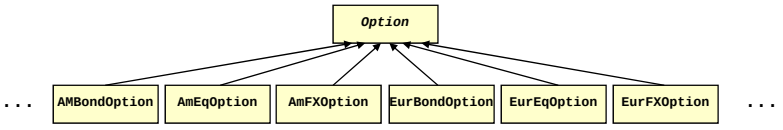
- Composition, or composition of simpler hierarchies, is usually a better choice.



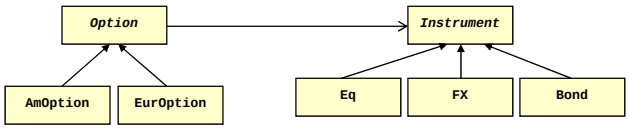
49

Wide or Deep Hierarchies

- A very wide or very deep inheritance hierarchy usually indicates a design flaw.
- A hierarchy that exhibits "exponential expansion" during maintenance usually indicates a design flaw.



✓ *Prefer composition of simple hierarchies to monolithic hierarchies.*

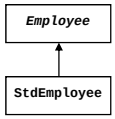


50

Base Classes and Non-Base Classes Differ

- Client code treats base classes very differently from standalone classes.
- Standalone classes that later become base classes wreak havoc on using code. Start a potential base class off as an abstract base class.
- Classes that are part of a hierarchy and standalone classes come from different planets.

✓ *Degenerate hierarchies are your friends.*



51

Framework-Oriented Design

- Every significant application has variations, either in "space" or in time.
- Framework-oriented design deals well with these issues.
  - supports the open/closed principle (Meyer)
  - "build for today, design for tomorrow" (Goldfedder)
- It is often a mistake to simply write an application.
  - every significant application should be designed as a framework.
  - patterns help a lot here, but be wary of patternitis.
- Observation: the existence of a design pattern can affect a hierarchy design even if the pattern is not explicitly used in the hierarchy.

52

Some Patterns Are A Priori

- Some patterns must be "designed in" from the start to allow easy extension of the framework.
- Examples include Visitor/Acyclic Visitor/Private Interface.
- Other common examples are Prototype and Factory Method.
- All these patterns require that a specific interface be part of the base class.

53

Some Patterns Can Be Slipped In

- Application of some patterns can "repurpose" an existing base class interface.
- For example, an application of Template Method, State, Strategy, or Bridge can be added without changing a base class *interface*.
- However, application will likely require change to the *implementation* of a base class member function.

54

Some Patterns Have "Needs"

- Other patterns may seem not to have an effect on an existing hierarchy, and can be added without changing or recompiling existing code.
- For example, Composite and Decorator may be used to extend a hierarchy without change or recompilation.
- However, these patterns apply well only if the root of the hierarchy is a relatively simple interface class.
  - Any implementation will be inherited by the derived class that implements the pattern, and may cause incorrect behavior or expense.
  - A wide interface will also be inherited, and will make it difficult to implement a substitutable derived class.

55

Observations

- Design patterns are often selected for "force resolution," but knowledge of their existence is also a force on the structure of a hierarchy.
- Composition of simple parts is simpler than a monolithic design, but can represent a more complex structure.
- Designs that promote ignorance and a single point of change are good.
- Code the minimum, but design toward the future.
- Maximum flexibility is not a goal, reasonable flexibility is.
- Idioms are useful only if they are both generally used and sometimes disregarded.
- There is no substitute for thoughtful abstraction and careful design. There are no cookbooks for OOD.

56

Unforeseen Extension

57

Roadmap

- Adapter
- External Polymorphism

58

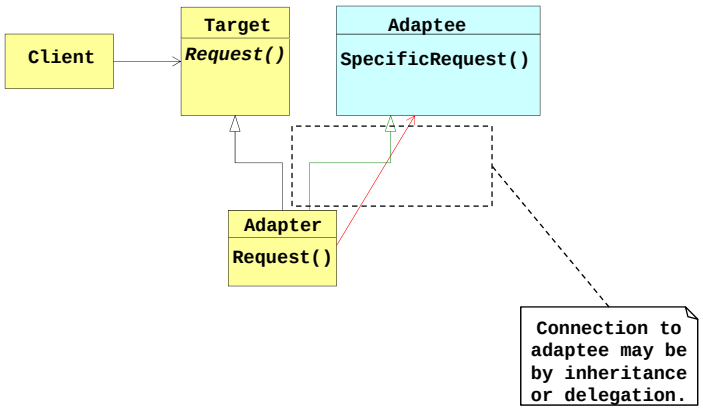
Adapter

- Category
  - Class Structural, Object Structural
- Intent
  - Convert the interface of a class (or hierarchy) into another that clients expect.
- Variances
  - Interface to an object.
- Redesign Problems Solved
  - Inability to alter classes conveniently.
  - Import of third party interfaces.
- See Also
  - Façade
  - Bridge
  - External Polymorphism

59

Adapter Structure

- An Adapter adapts a "third party" interface to be in conformance with a client's expected interface.



60

Sensors

- Here's a polymorphic sensor base class.

```
class Sensor {
public:
    virtual ~Sensor();
    virtual bool status() const = 0;
};
```

- There are lots of concrete sensors.

```
class DoorSensor final : public Sensor {
    bool status() const override
        { return the_door_is_open(); }
};
```

61

Third-Party Sensors

- We may want to include sensors in our hierarchy that don't support the `Sensor` interface.

```
struct XDeviceDriver { // a poorly-designed class
    bool getState() const;
protected:
    void reset();
private:
    volatile unsigned register_;
};

struct YDeviceDriver { // a polymorphic base class
    virtual ~YDeviceDriver();
    virtual int level() const = 0;
};
```

62

A Class Adapter

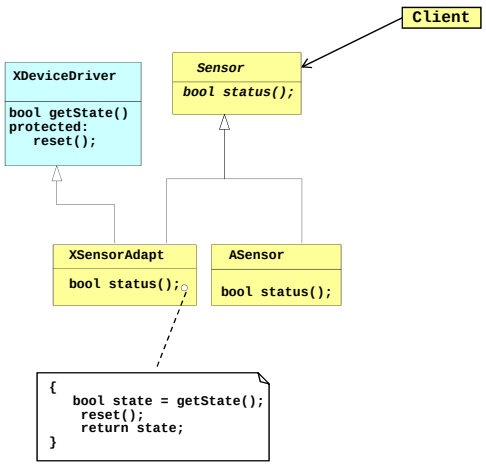
- The first third-party sensor is just poorly-designed.
- But we don't have access to the source code.
- We'll have to apply a Class Adapter.

```
class XSensorAdapt final : public Sensor,
                           private XDeviceDriver {
    bool status() const override {
        bool state = getState();
        const_cast<XSensorAdapt *>(this)->reset();
        return state;
    }
};
```

- The use of private inheritance indicates an "is-implemented-in-terms-of" relationship, not an is-a relationship.

63

Class Adapter



64

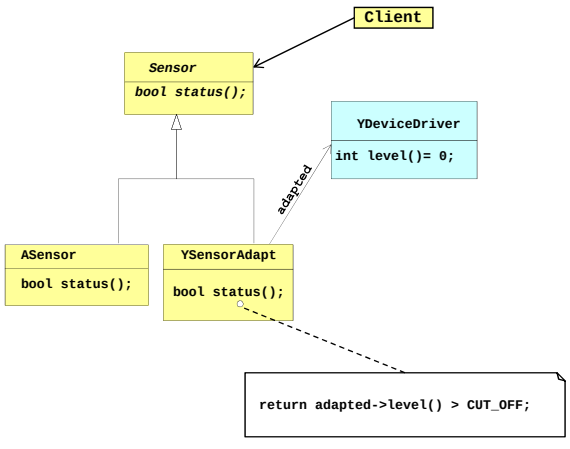
An Object Adapter

- The other third-party sensors form a well-designed hierarchy.
- We don't want to adapt each concrete sensor individually.
- We'll choose to apply an Object Adapter.

```
class YSensorAdapt final : public Sensor {
public:
    YSensorAdapt(YDeviceDriver const *to_adapt)
        : adapted_(to_adapt) {}
    ~YSensorAdapt() {}
private:
    bool status() const override
        { return adapted_->get_level() > CUT_OFF; }
    YDeviceDriver const *adapted_;
};
```

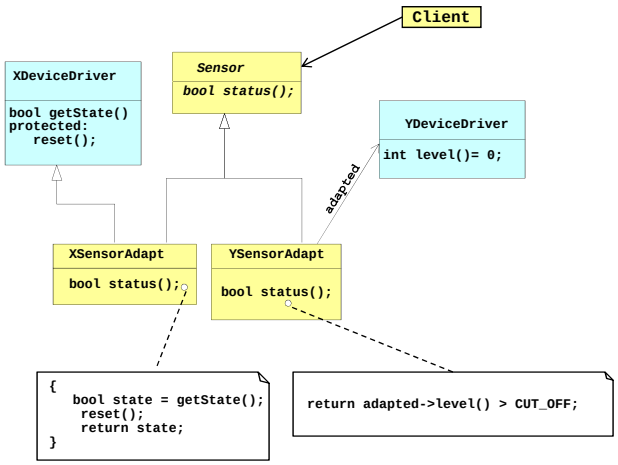
65

Object Adapter



66

Adapters



67

Ownership Issues

- In this Object Adapter, we do not own the adaptee.
- We have to be concerned that the actual owner of the adaptee agrees with us! (If the adaptee is a memory-mapped device, this implementation may be appropriate.)

```
class YSensorAdapt final : public Sensor {
public:
    YSensorAdapt(YDeviceDriver const *to_adapt)
        : adapted_(to_adapt) {} // dangling pointer?
    ~YSensorAdapt() {}          // memory leak in caller?
private:
    bool status() const override
        { return adapted_->get_level() > CUT_OFF; }
    YDeviceDriver const *adapted_;
};
```

68

Ownership Issues

- An Object Adapter makes you consider ownership issues.
- In the previous implementation, we didn't own the adaptee.
- Here we do, by making a copy: "I don't know what kind of sensor this is, but I want another one just like it!" Prototype.

```
class YSensorAdapt final : public Sensor {
public:
    YSensorAdapt(YDeviceDriver const *to_adapt)
        : adapted_(to_adapt->clone()) {}
    ~YSensorAdapt()
        { delete adapted_;} // memory leak in caller?
private:
    bool status() const override
        { return adapted_->get_level() > CUT_OFF; }
    YDeviceDriver const *adapted_;
};
```

69

Ownership Issues

- Here we choose not to copy, but to *transfer* ownership explicitly.

```
class YSensorAdapt final : public Sensor {
public:
    YSensorAdapt(unique_ptr<YDeviceDriver> to_adapt)
        : adapted_(to_adapt) {}
    ~YSensorAdapt() {}
private:
    bool status() const override
        { return adapted_->get_level() > CUT_OFF; }
    unique_ptr<YDeviceDriver> adapted_;
};
```

- YSensorAdapt is movable, but not copyable.

70

Ownership Issues

- Here we choose not to copy, but to *share* ownership explicitly.

```
class YSensorAdapt final : public Sensor {
public:
    YSensorAdapt(shared_ptr<YDeviceDriver> to_adapt)
        : adapted_(to_adapt) {}
    ~YSensorAdapt() {}
private:
    bool status() const override
        { return adapted_->get_level() > CUT_OFF; }
    shared_ptr<YDeviceDriver> adapted_;
};
```

71

Class vs. Object Adapter

- A Class Adapter uses multiple inheritance.
  - Typically public for the interface, private for the adaptee.
  - The Adapter is-a interface, and is-implemented-in-terms-of the adaptee.
  - Can override adaptee's behavior.
- An Object Adapter uses composition.
  - Lets a single Adapter adapt an entire hierarchy or sub-hierarchy.
  - Harder to customize behavior of the adaptee.

72

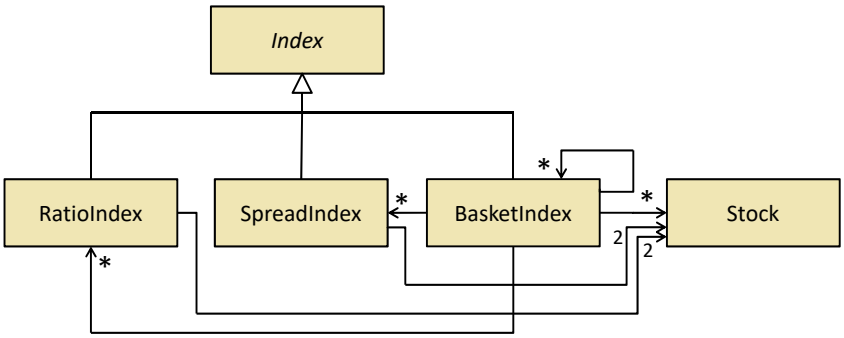
Example: Stock Indexes

- Suppose we have to model different types of stock indexes.
  - The value of a *ratio index* is the quotient of two weighted stock prices.
  - The value of a *spread index* is the difference of two weighted stock prices.
  - The value of an *intraday basket index* is the sum of an unbounded number of weighted stock indexes and individual stocks.

73

Complex, Restricted Relationships

- A straightforward translation of these requirements results in an unwieldy design.



- …and what happens if we add another type of index to the hierarchy?

74

Pricing a Basket, Version 1

```
class Index {
public:
    ~~~
    virtual double value() const = 0;
    ~~~
};
class BasketIndex : public Index {
    ~~~
    double value() const override;
private:
    // for simplicity, weights are all 1.0
    vector<SpreadIndex const *> spr_;
    vector<RatioIndex const *> rat_;
    vector<BasketIndex const *> bas_;
    vector<Stock const *> sto_;
};
```

75

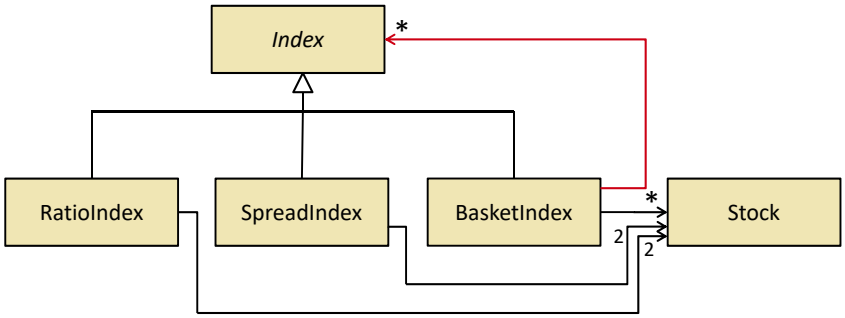
Pricing a Basket, Version 1

```
double BasketIndex::value() const {
    double val = 0.0;
    for (auto spread : spr_)
        val += spread->value();
    for (auto ratio : rat_)
        val += ratio->value();
    for (auto basket : bas_)
        val += basket->value();
    for (auto stock : sto_)
        val += stock->price();
    return val;
}
```

76

Extract Composite

- If we apply a Composite, the design is simplified considerably.



77

Pricing a Basket, Version 2

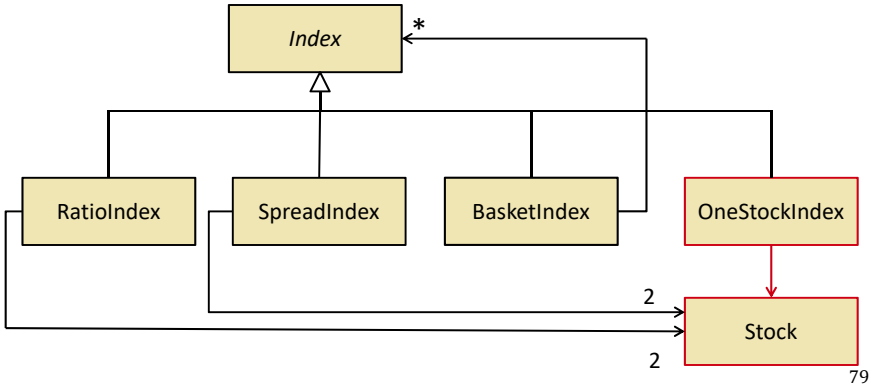
```
class BasketIndex : public Index {
    ~~~
    double value() const override;
private:
    vector<Index const *> ind_;
    vector<Stock const *> sto_;
};

double BasketIndex::value() const {
    double val = 0.0;
    for (auto index : ind_)
        val += index->value();
    for (auto stock = sto_)
        val += stock->price();
    return val;
}
```

78

Extract Object Adapter

- We apply an Object Adapter to allow us to treat the (atomic) `Stock` as an `Index`.
- This allows us to apply Composite more generally.
- …and just look at how ignorant `BasketIndex` is now!



79

Object Adapter Pattern

- An Object Adapter allows us to treat a `Stock` as an `Index` by adapting its interface to make it look like an `Index`.

```
class OneStockIndex final : public Index {
public:
    ~~~
    double value() const override
        { return sto_->price(); }
    ~~~
private:
    Stock const *sto_;
}
```

80

Pricing a Basket, Version 3

```
class BasketIndex final : public Index {
    ~~~
    double value() const override;
private:
    vector<Index const *> ind_;
};

double BasketIndex::value() const {
    double val = 0.0;
    for (auto index : ind_)
        val += index->value();
    return val;
}
```

81

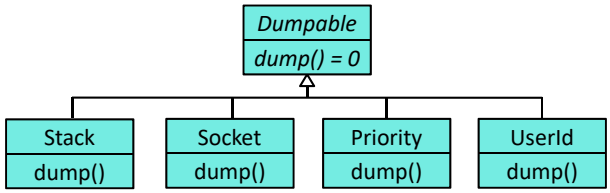
External Polymorphism

- Category
  - Object Structural
- Intent
  - Transparently extend concrete types with polymorphic behavior.
- Variances
  - Interface to a set of unrelated object types.
- Redesign Problems Solved
  - Inability to alter classes conveniently.
  - Import of third party interfaces.
- See Also
  - Façade
  - Bridge
  - Adapter
  - Visitor

82

What We Want

- Consider the problem of dumping out information about a variety of objects of different types.
- It's easy if the types share a common polymorphic base class.

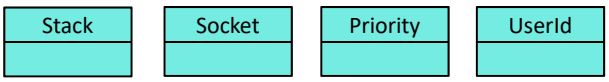



```
void dumpEm(list<Dumpable *> const &toDump) {
    for (auto d = toDump)
        d->dump();
}
```

83

Polymorphism Among Unrelated Types

- It's hard if the types have no relationship whatsoever.



84

Wide Interfaces

- Even if we have an existing hierarchy, it may not be advisable to encumber all users of the hierarchy with an interface that applies only to a single user.

```
class Component {
Public:
    virtual ~Component();
    virtual void generally_useful_op1() = 0;
    virtual void generally_useful_op2() = 0;
    virtual void dump() const = 0;
};
```

- Additionally, we may not have permission to change the base class interface.

85

A Capability Interface Class

- Another approach is to design a polymorphic base class for the ad hoc capability.

```
class Dumpable {
public:
    Dumpable() = default;
    virtual ~Dumpable() = default;
    Dumpable(Dumpable const &) = delete;
    Dumpable &operator =(Dumpable const &) = delete;
    virtual void dump() const = 0;
};
```

- Anything that is-a Dumpable can be dumped.

86

Inheriting a Capability

- However, direct application of this approach requires that each "dumpable" class be modified.

```
class Socket : public Dumpable ~~~
```

- Alternatively…

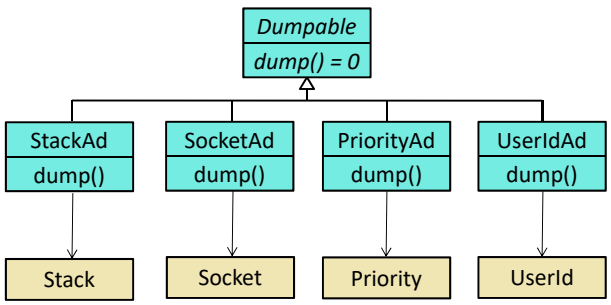
```
class Socket : public Component, public Dumpable ~~~
```

- This is a form of "interface pollution," and is a violation of the Interface Segregation Principle.

87

Polymorphism Among Unrelated Types

- A better approach is typically to create a polymorphic hierarchy "off to the side" without affecting the types that will be dumped.
- The individual object types are adapted to the *ad hoc* interface.
- Each concrete type in the hierarchy is an Object Adapter.



- This is the basis of External Polymorphism.

88

Hand-Coded Adapters

- The most straightforward approach might be to write a separate adapter for each type we want to dump.

```
class DumpableSocket final : public Dumpable {
public:
    DumpableSocket(Socket const *s)
        : to_dump_ (s) {}
    void dump() const override {
        // dump the socket…
    }
private:
    Socket const *to_dump_;
};
```

89

Family of Object Adapters

- Alternatively, we could use a class template to help write the Object Adapters.

```
template <typename T>
class ConcreteDumpable final : public Dumpable {
public:
    ConcreteDumpable(T const *t)
        : to_dump_ (t) {}
    void dump() const override
        { do_dump(to_dump_); }
private:
    T const *to_dump_;
};
```

- All types are dumped by calling a `do_dump` function.

90

Overloading

- Anything that already has a `dump` operation (with the correct meaning!) is handled by a function template.

```
template <typename T>
inline void do_dump(T const *t) { t->dump(); }
```

- Other types have to specify how they can be dumped.

```
inline void do_dump(INET_Addr const *t) { ~~~ }
inline void do_dump(SockStream const *t) { ~~~ }
inline void do_dump(SockAcceptor const *t) { ~~~ }
```

- In the case of an exact match, the non-template function is preferred to the function template.

91

You've Got To Have Helpers

- We'll use the Helper Function idiom to perform template argument deduction and specialize `ConcreteDumpable` for us.

```
template <typename T>
Dumpable *make_dumpable(T const &t)
    { return new ConcreteDumpable<T>(&t); }
```

- Alternatively:

```
typedef unique_ptr<Dumpable const> UD;

template <typename T>
UD make_dumpable(T const &t)
    { return UD(new ConcreteDumpable<T>(&t)); }
```

92

```
class DumpableCollection : public vector<UD> {
public:
    void dump() const {
        for_each(begin(), end(),
                 [](UD const &d){d->dump();});
    }
};
void dump_stuff() {
    DumpableCollection c;
    INET_Addr aInetAddr;
    SockStream aSockStream;
    SockAcceptor aSockAcceptor;
    c.push_back(make_dumpable(aSockStream));
    c.push_back(make_dumpable(aSockAcceptor));
    c.push_back(make_dumpable(aInetAddr));
    c.dump();
}
```

93

Polymorphic Built-Ins

- Note that External Polymorphism may also be used to wrap polymorphic behavior around non-polymorphic types.

```
int a = 12;
UD d = make_dumpable(a);
d->dump();
```

- For example, `INET_Addr` is probably an integer.

94

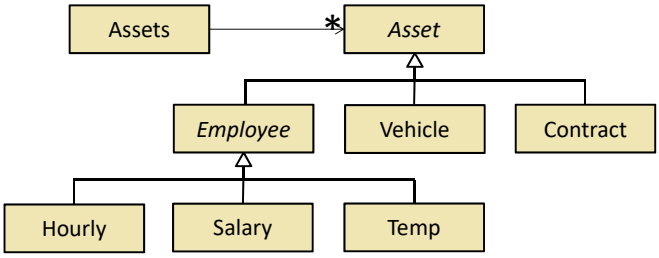
When to use External Polymorphism?

- External Polymorphism is a "special-purpose tool."
- External Polymorphism allows addition of polymorphic behavior without modifying a class's layout.
  - Consider a memory-mapped device.
- External Polymorphism circumvents the need for establishing monolithic hierarchies for *ad hoc* code.
  - Employees are Assets, but so are Contracts and Bonds. Should they share a base class?
  - Reduction of wide base class interfaces.
  - Easy to remove an interface from code that does not use it.
- External Polymorphism allows addition of polymorphic behavior to built-in and other non-polymorphic objects.
  - Polymorphic `FILE` *s, `int`s, and `double`s!

95

Causes of Cosmic Hierarchies

- Recall our "container of anything."
- A common cause of such monolithic hierarchies is an *ad hoc* need to treat disparate types of objects in a polymorphic manner.

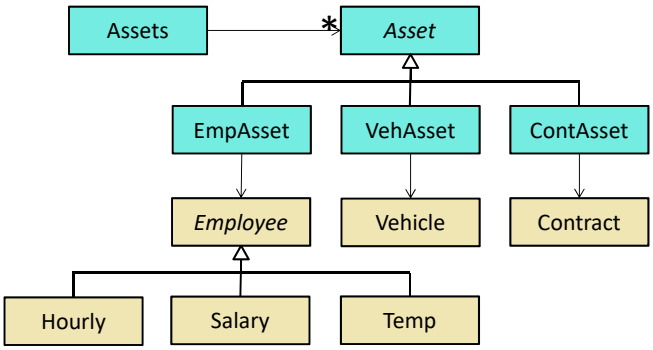


- Often, the *ad hoc* need disappears in time, leaving the cosmic hierarchy in place to wreak havoc.

96

Ad Hoc Polymorphism

- Use of External Polymorphism makes the *ad hoc* nature of the hierarchy explicit.
- If the *ad hoc* need passes, the application of External Polymorphism can be removed without affecting other code.



97

Causes of Wide Interfaces

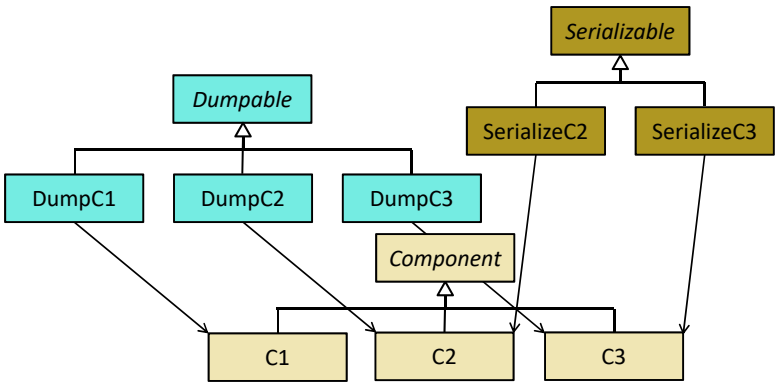
- The presence of many special-purpose interfaces can cause a polymorphic base class to have an inappropriately wide interface.

```
class Component {
Public:
    virtual ~Component();
    virtual void generally_useful_op1() = 0;
    virtual void generally_useful_op2() = 0;
    virtual void dump() const = 0; // for the 2% who dump
    virtual string serialize() const = 0; // 1% serialize
    // ad infinitium...
};
```

98

Managing Special-Purpose Interfaces

- External Polymorphism can be used to supply special-purpose interfaces in such a way that most users are not encumbered.
- Note that with External Polymorphism not all derived components have to support a particular interface.



99

A Cloneable Interface

- Consider a java-like cloneable interface.

```
class Cloneable {
public:
    virtual ~Cloneable() {}
    virtual Cloneable *clone() const = 0;
};
```

- We can use External Polymorphism to apply cloneability to arbitrary types.
  - We can clone across hierarchies.
  - We can clone built-in types.

100

Plusses and Minuses

- How do you feel about the following usage?

```
template <typename T>
void f(T const *p) {
    static_assert(std::is_polymorphic<T>::value,
                  "must have virtual function!");
    if (Cloneable *to_clone
        = dynamic_cast<Cloneable *>(p)) {
        T *p2 = static_cast<T *>(to_clone->clone());
    }
    ~~~
}
```

- Is the dynamic_cast acceptable in this situation?
- Is the static_cast guaranteed to be safe?

101

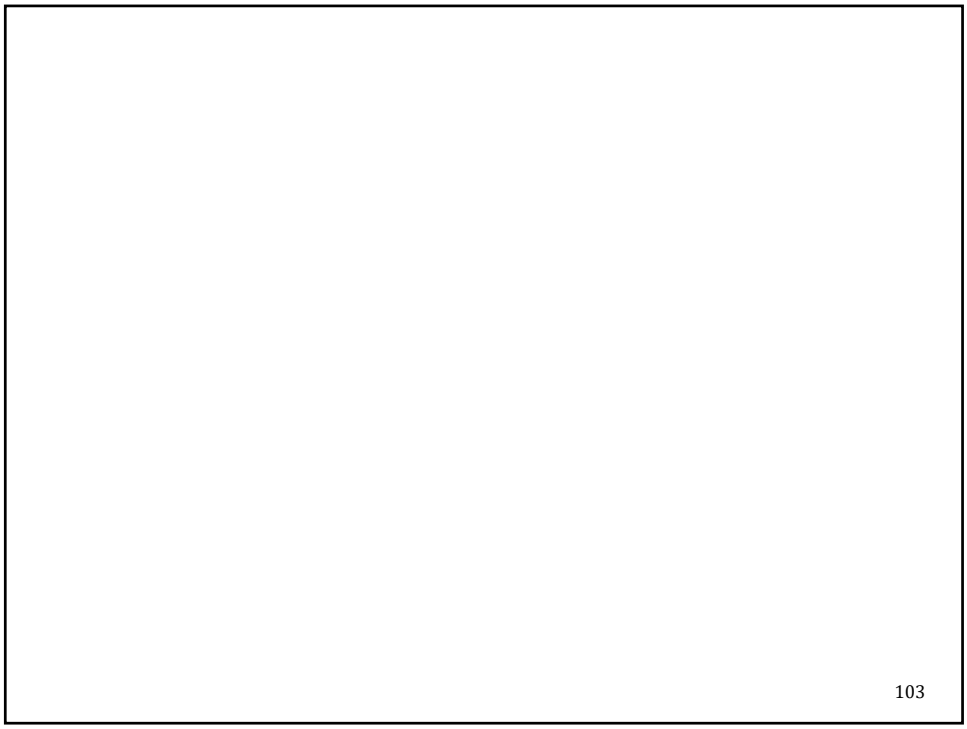
Static vs. Dynamic

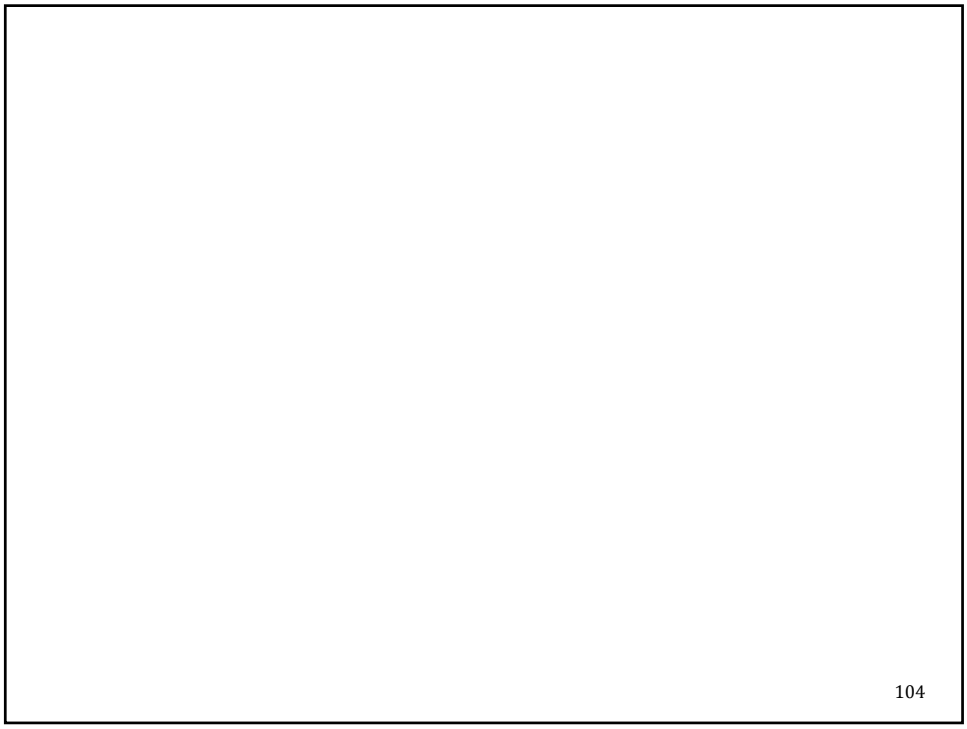
- How do you feel about the following usage?

```
template <typename T>
void f(T const *p) {
    if (std::is_base_of<Cloneable, T>::value) {
        Cloneable *c
            = static_cast<Cloneable *>(p)->clone();
        T *p2 = static_cast<T *>(c);
    }
    ~~~
}
```

- Does this code make sense?
- Does it (always) work?

102





Planned Extension

105

Roadmap

- Visitor
- Private Interface
- Acyclic Visitor

106

Visitors

- Even though it's Eric Gamma's least favorite pattern, Visitor and its variants are occasionally useful.
- Their underlying techniques have served as inspiration for other protopatterns.
- They're also fun and somewhat challenging.
- We'll look at three variants of Visitor:
  - GOF Visitor
  - Acyclic Visitor
  - Ad Hoc Visitor

107

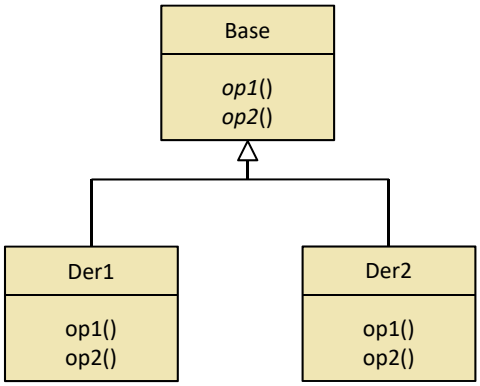
Visitor

- Category
  - Object Behavioral
- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Variances
  - Operations that can be applied to objects without changing their classes.
- Redesign Problems Solved
  - Algorithmic dependencies.
  - Inability to alter classes conveniently.
- See Also: Strategy, Decorator, Acyclic Visitor

108

Visitor Motivation

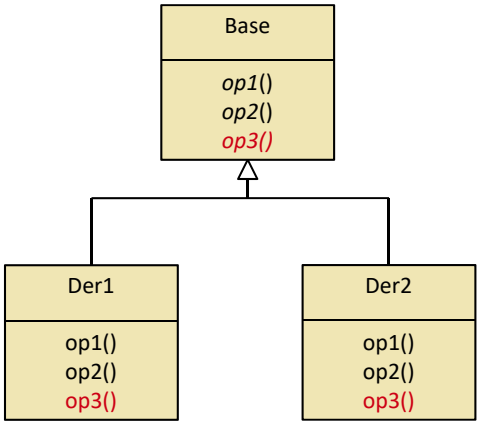
- Conventionally, operations common to all classes in the hierarchy are declared in the base class and overridden, where required, in each derived class.



109

Visitor Motivation

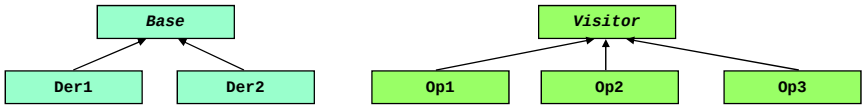
- However, adding a new operation is fairly difficult, since every class in the hierarchy must be modified.



110

Effect of Applying The Visitor Pattern

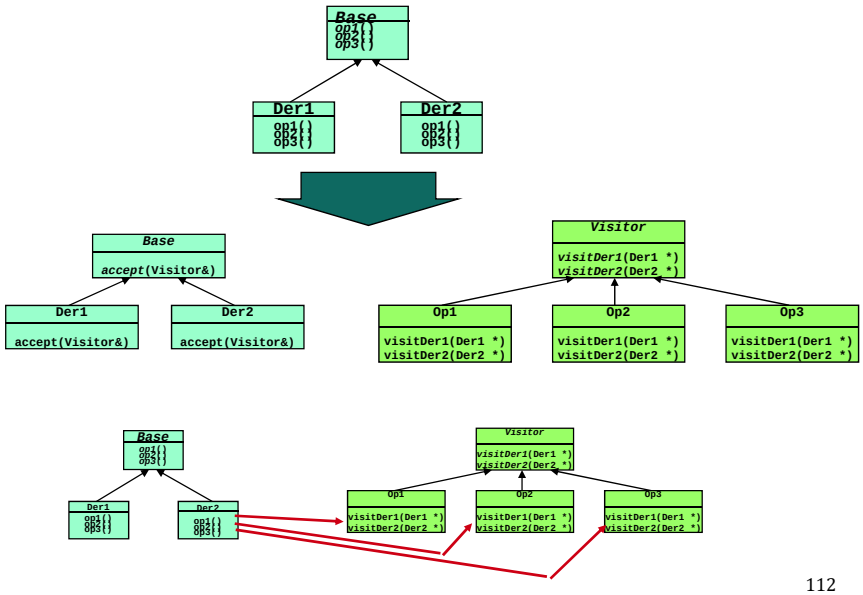
- Alternatively, the operations can be separated from the classes on which they operate, producing a dual, non-parallel type/action structure.



- New operations may effectively be added to the Base hierarchy without changing it.
- The visitor pattern shows how to do this.
- Visitor may often be used in place of RTTI provided the hierarchy designer makes provision for it in advance.

111

Transformation of Hierarchy



112

Double Dispatch, Step 1

- The idea is to select the function to call based on the types of both the `Base` and the `Visitor`.

```
Base *bp = getNextBase();    // assume a ptr to a Der1
Visitor &v = getOperation(); // assume an Op1
bp->accept(v);    // perform op1 on Der1
```

- The first call, to accept, determines the type of the `Base` object.

113

Double Dispatch, Step 2

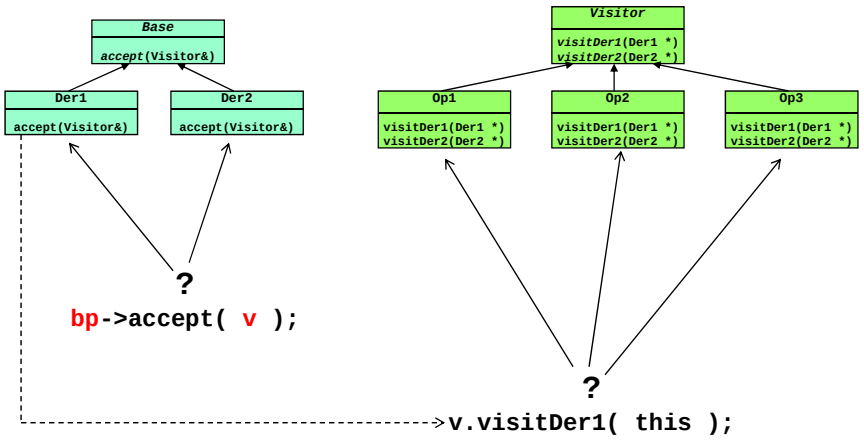
- The implementation of accept determines the type of the `Visitor` by calling the appropriate virtual member of the `Visitor` abstract base.

```
void Der1::accept(Visitor &v) {
    v.visitDer1( this );
}
```

- The effect is to select the appropriate object/operation pair through a sequence of two virtual calls: double dispatch.

114

Double Dispatch



115

A Simple Design

- A simple design…

```
class Employee {
public:
    Employee(const Name &name, const Address &address);
    virtual ~Employee();
    ~~~
private:
    Name name_;
    Address address_;
    list<Role *> roles_;
};
```

116

Adding Type-Based Behavior

- We'd like to be able to right-size these assets…
- But it's necessary to do so without changing or recompiling released source.

```
// Oops! Forgot we have to right-size these assets!
void terminate(Employee *);
void terminate(SalaryEmployee *);
void terminate(HourlyEmployee *);
```

117

Abuse of RTTI

- "The syntactic elegance of C with the efficiency of Smalltalk."

```
void terminate(Employee *e) {
    if (HourlyEmployee *h
            = dynamic_cast<HourlyEmployee *>(e))
        terminate(h);
    else if (SalaryEmployee *s
            = dynamic_cast<SalaryEmployee *>(e))
        terminate(s);
    else
        throw UnknownEmployeeType();
}
```

118

RTTI as a Weapon

- You mean you want this code to be maintainable?
- You mean you mind runtime type errors?
- However, we were able to add a new operation to the hierarchy without changing, or recompiling, the code.
- But would you actually do this?

✓ *Yes, if you were forced to...*

119

Avoid Type-Based Conditionals

- Don't ask an object personal questions. Just tell it to get to work.

```
class Employee {
public:
    Employee(const Name &name, const Address &address);
    virtual ~Employee();
    virtual bool isPayday() const = 0;
    virtual void pay() = 0;
    virtual void terminate() = 0;
private:
    ~~~
};
~~~
Employee *e = getEmployeeOfSomeSort();
e->terminate(); // direct order, no personal questions
```

120

Interfaces

- Consider the following interface classes.

```
struct Saveable {   // persistence interface
    virtual void save() const = 0;
};

struct Priceable {    // pricing interface
    virtual double price() const = 0;
};
```

- These interfaces specify capabilities.
- For example, anything that is-a `Priceable` can be priced.

121

Attaching Interfaces

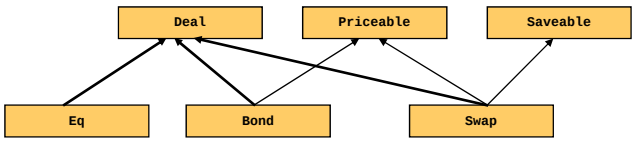
- These interface classes would commonly be used to attach interfaces to classes in a multiple inheritance hierarchy.

```
class Deal {
public:
    virtual bool validate() const = 0;
    ~~~
};
class Swap final
    : public Deal, public Priceable, public Saveable {
public:
    bool validate() const override;
    double price() const override;
    void save() const override;
    ~~~
};
```

122

Extended Is-A

- A `Bond` is a `Deal` that is also `Priceable`.
- A `Swap` is a `Deal` that is also `Priceable` and `Savable`.



123

dynamic_cast as the Downcast of Doom

- Consider again adding a new operation on deals without changing or recompiling the hierarchy.
- Naive code might simply ask the obvious questions.

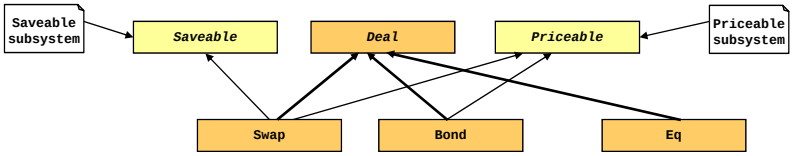
```
void processDeal(Deal *d) {
    d->validate();
    if(Bond *b = dynamic_cast<Bond *>(d))
        b->price();
    else if(Swap *s = dynamic_cast<Swap *>(d)) {
        s->price();
        s->save();
    }
}
```

- This code is very fragile, slow, and hard to maintain.

124

Abstract Interfaces as Capabilities

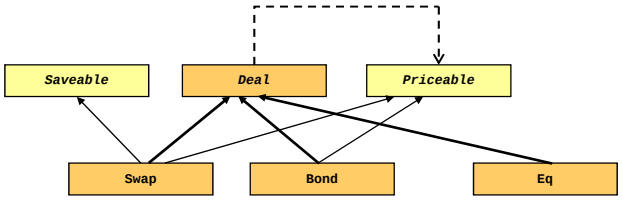
- A common organization in OOD is to attach "capabilities" to classes in a hierarchy through multiple inheritance.
- The interface classes represent a set of operations; a capability.



125

Cross Casting

- It's possible to use `dynamic_cast` to cast between unrelated types.
- If the complete object's type is derived from the target of the cast, the cast will succeed.
- (Well, it could also fail if the cast is ambiguous…)



126

Capability Queries by Cross Casting

- A successful cross-cast to the interface type indicates that the unknown concrete class has that capability.

```
Deal *dp = getNextDeal();
Priceable *pp
    = dynamic_cast<Priceable *>(dp); // is it priceable?
if (pp) {
    // yup, price it
}
else {
    // nope.
}
```

127

Capability Queries as a Stopgap

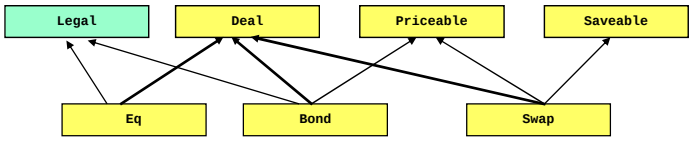
- A `dynamic_cast` can be used to ask if a particular deal, referred to through a base class interface, may be priced and/or is persistent.

```
void processDeal(Deal *d) {
    d->validate();
    if (Priceable *p = dynamic_cast<Priceable *>(d))
        p->price();
    if (Saveable *s = dynamic_cast<Saveable *>(d))
        s->save();
}
```

128

Capability Queries as a Stopgap

- This code is somewhat less fragile.
- It's still slow and dangerous.
- Cascading capability queries are not a good base for a design; they are a hack or a stopgap.



129

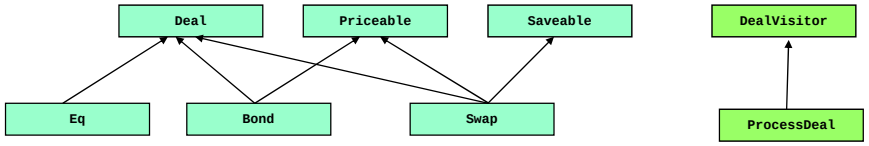
A Better Design, But Changes Base

```
class Deal {
public:
    virtual bool validate() const = 0;
    virtual void process() = 0;
    ~~~
};
class Swap final
    : public Deal, public Priceable, public Saveable {
public:
    bool validate() const override;
    void price() override;
    void save() override;
    void process() override
        { validate(); price(); save(); }
};
```

130

A Deal Visitor Tries to Help

- If the hierarchy doesn't change often, we can add operations and avoid RTTI through application of Visitor.



131

A Deal Visitor

```
class Deal {
public:
    virtual bool validate() const = 0;
    virtual void accept(DealVisitor &) = 0;
    ~~~
};

class DealVisitor {
public:
    virtual ~DealVisitor() {}
    virtual void visitBond(Bond *) = 0;
    virtual void visitSwap(Swap *) = 0;
    virtual void visitEq(Eq *) = 0;
};
```

132

"Process" Deal Visitor

```
class ProcessDeal final : public DealVisitor {
    void visitBond(Bond *d) override {
        d->validate();
        d->price();
    }
    void visitSwap(Swap *d) override {
        d->validate();
        d->price();
        d->save();
    }
    void visitEq(Eq *d) override
        { d->validate(); }
};
~~~
void Swap::accept(DealVisitor &v)
    { v.visitSwap(this); }
```

133

Simpler and Faster

```
void processDeal(Deal *d) { // before
    d->validate();
    if(Bond *b = dynamic_cast<Bond *>(d))
        b->price();
    else if(Swap *s = dynamic_cast<Swap *>(d)) {
        s->price();
        s->save();
    }
}

void processDeal(Deal *d) { // after
    ProcessDeal process;
    d->accept(process);
}
```

134

Non-Overloaded Visit Functions

- So far, we've used differently-named "visit" functions in the Visitor implementation.

```
class DealVisitor {
public:
    virtual ~DealVisitor() {}
    virtual void visitBond(Bond *) {}
    virtual void visitSwap(Swap *) {}
    virtual void visitEq(Eq *) {}
};
```

135

Non-Overloaded Visit Functions

- There are some advantages to using non-overloaded functions in the Visitor hierarchy.
- If the base class functions have default implementations, new derived Visitor types may not have to override all base class visit functions.

```
class CancelDeal : public DealVisitor {
public:
    void visitSwap(Swap *);
};
```

- This allows a derived Visitor class to override just one visit function without hiding all of them.

136

Non-Overloaded Visit Functions

- If there is sufficient commonality among the "default" implementations, we can use a catch-all.

```
class DealVisitor {
public:
    virtual void visitDeal(Deal *d);   // catch-all
    virtual void visitBond(Bond *b)
        { visitDeal(b); }              // virtual call
    virtual void visitEq(Eq *e)
        { DealVisitor::visitDeal(e); } // non-virtual
    ~~~
};
```

137

Catch-All Choices

- Another design choice is whether the catch-all may be overridden.

```
class CancelDeal : public DealVisitor {
public:
    void visitDeal(Bond *); // use for Bond
    void visitSwap(Swap *); // use for Swap
                            // use base catch-all for Eq
};
```

- This choice depends on both the virtualness of the catch-all as well as how the catch-all is called.

138

Pure Catch-All Visit Functions

- Be aware of the behavior of pure virtual functions!

```
class DealVisitor {
public:
    virtual void visitDeal(Deal *d) = 0; // catch-all
    virtual void visitEq(Eq *e)
        { DealVisitor::visitDeal(e); }   // non-virtual!
    ~~~
};
```

- Note that you can define a pure virtual function, but it may be called only in a non-virtual manner (as above).

```
void DealVisitor::visitDeal(Deal *) {}
```

139

Statically Safe, But Inflexible

- Note that this catch-all *does not* allow unrecognized deal types to be visited.
- If the deal hierarchy is extended with a new deal type, there will be a compile-time error...

```
class NewDeal : public Deal {
    ~~~
    void accept(DealVisitor &v) {
        v.visitNewDeal(this); // error! no such function
    }
};
```

- ...until the `DealVisitor` hierarchy is maintained.
- This is often a good thing; static errors are better than runtime.

140

Overloaded Visit Functions

- However, it's more typical to use overloading to distinguish visit operations.

```
class DealVisitor {
public:
    virtual ~DealVisitor();
    virtual void visit(Deal *);     // catch-all
    virtual void visit(Bond *) = 0;
    virtual void visit(Swap *) = 0;
    virtual void visit(Eq *) = 0;
};
```

- Note that this catch-all *does* allow unrecognized deal types to be visited.

141

Details: Overloaded Visit Functions

- One advantage of overloading may be seen in the implementation of the accept operation.

```
void NewDeal::accept(DealVisitor &v)
    { v.visit(this); } // could be call to catch-all
```

- All the `accept` implementations look identical.
- Without overloading, each `accept` must identify its visitor callback precisely.

```
void NewDeal::accept(DealVisitor &v)
    { v.visitNewDeal(this); } // always to visitNewDeal
```

142

Overloading and Automatic Maintenance

- Maintenance of the `accept` function is now automatic, and tracks changes to the `Visitor` hierarchy.

```
class DealVisitor {
public:
    virtual void visit(Deal *);       // bind here before
    virtual void visit(Bond *) = 0;
    ~~~
    virtual void visit(NewDeal *) = 0; // bind here after
};
```

- On recompilation, `accept` will bind to the new deal-specific visit function without source code change.

143

Overloaded Visit Functions and Catch-alls

- Another advantage to overloaded `visit` is that it allows easy insertion of a default action, or catch-all, for unrecognized deal types.

```
void NewDeal::accept(DealVisitor &v)
    { v.visit(this); }   // calls base visit(Deal *)
```

- The catch-all action can be as simple or complex as required.

```
void DealVisitor::visit(Deal *)
    { throw BadVisit("operation on unknown deal type"); }
```

144

Overloaded Visit Functions and Catch-alls

- The catch-all can be customized to the individual visitor type.

```
class CancelDeal : public DealVisitor {
    void visit(Deal *);
    void visit(Swap *);
};
```

- Note that the catch-all can know the type of the Visitor, but does not know the type of the visited object.

```
void CancelDeal::visit(Deal *) {
     throw BadVisit("cancelation of unknown deal type");
}
```

145

Overriding and Hiding Issues

- When using overloaded `visit` functions, it's possible to break polymorphism:

```
class CancelDeal : public DealVisitor {
public:
    void visit(Swap *);
};
```

- In this case, `CancelDeal::visit` overrides only `DealVisitor::visit(Swap *)`, but hides all versions of `visit` in `DealVisitor`.
- As a result, a user of `CancelDeal` may get different runtime behavior depending on the interface used to access the `visit` function.

146

Dealing with Hiding Issues

- One way to improve things is to force users to use only the base class interface.

```
class CancelDeal : public DealVisitor {
private:
    void visit(Swap *);
};
```

- In this case, `CancelDeal::visit` still has the same overriding and hiding semantics, but users may not access the function through the derived class interface.

147

Dealing with Hiding Issues

- Another approach is to import all the base class visit functions with a using declaration.

```
class CancelDeal : public DealVisitor {
public:
    using DealVisitor::visit;
    void visit(Swap *);
};
```

148

Process, Re-Visited

```
class ProcessDeal : public DealVisitor {
    void visit(Deal *);     // override catch-all
    void visit(Bond *);
    void visit(Swap *);
    void visit(Eq *);
    void visit(NewDeal *nd); // doesn't override...yet
};

void ProcessDeal::visit(Deal *d) {
    // temporary, ad hoc code
    if (NewDeal *nd = dynamic_cast<NewDeal *>(d))
        visit(nd);
    else
        DealVisitor::visit(d); // forward to base
}
```

149

Visiting Multiple Hierarchies

- Note that a Visitor may visit objects from different hierarchies.

```
class InstVisitor {
public:
    virtual ~InstVisitor();
    virtual void visit(Deal *);    // catch-all Deals
    virtual void visit(Bond *) = 0;
    virtual void visit(Eq *) = 0;
    ~~~
    virtual void visit(Option *);  // catch-all Options
    virtual void visit(AmOption *) = 0;
    virtual void visit(EurOption *) = 0;
    ~~~
};
```

150

Example: A Clone Visitor

- The Prototype pattern is often useful, but is an *a priori* pattern that must be designed in from the start.
- It may be that a late-breaking requirement requires the cloning of Deals.

```
class Deal {
public:
    virtual Deal *clone() const = 0; // we'd like to...
    ~~~                              // ...but can't!
};
class Bond final : public Deal, public Priceable {
public:
    Bond *clone() const override
        { return new Bond(*this); }
    ~~~
};
```

151

A Clone Visitor

- Fortunately, we do have the hooks for a `Deal` Visitor.

```
class CloneVisitor final : public DealVisitor {
    unique_ptr<Deal> clone_;
    void visit(Deal *) override // catch-all
        { clone_ = nullptr; }
    void visit(Eq *e) override
        { clone_.reset(new Eq(*e)); }
    void visit(Bond *b) override
        { clone_.reset(new Bond(*b)); }
    void visit(Swap *s) override
        { clone_.reset(new Swap(*s)); }
public:
    CloneVisitor() : clone_ (nullptr) {}
    unique_ptr<Deal> get() const
        { return (unique_ptr<Deal>&&)(clone_); }
};
```

152

Using the Clone Visitor

- Usage is straightforward:

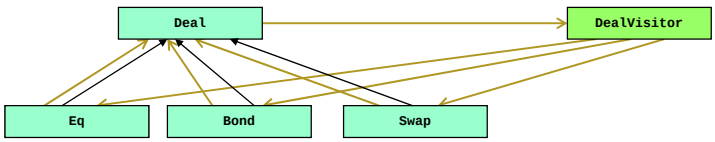
```
CloneVisitor cloner;
auto d = getMeAnotherDeal();
d->accept(cloner);
auto clone = cloner.get();
```

- The interface is less than ideal.
  - We've lost the covariant return, among other things.
- However, we were able to add Prototype-like functionality without resorting to type-based conditional code.

153

Visitor Circular Dependencies

- Visitor introduces a cycle of dependencies.

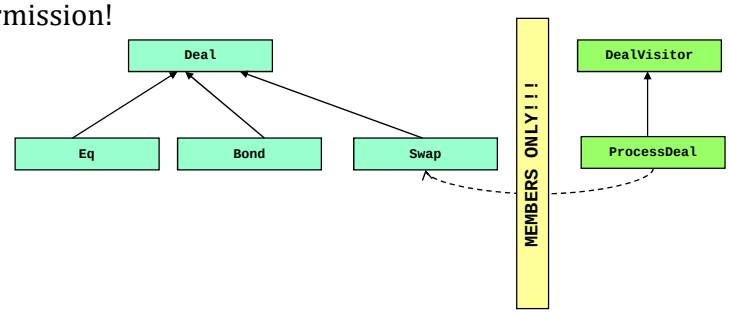


- Adding a new deal type, or changing the deal or visitor base classes affects the entire structure.
- This is often desirable; we usually want errors at compile time, not runtime!

154

Access Difficulties

- Visitor exhibits a recurring problem with C++ access protection.
- In our earlier example, the visitors "called back" on public members of the deals they were visiting.
- However, the visitors may want to access private or protected members of the class they're visiting, but they don't have permission!



- The same issue arises with the Strategy and State patterns.

155

Private Interface

- Category
  - Class Structural
- Intent
  - Provide a mechanism that allows specific classes to use a non-public subset of a class interface without inadvertently increasing the visibility of any hidden member variables or member functions.
- Variances
  - Extension of friendship in a hierarchically-based, controlled fashion.
- Redesign Problems Solved
  - Implementation dependencies.
  - Inability to extend friendship without rewriting code.
- See Also
  - Various hacks
  - Java package level protection

156

Private Interface

- The Private Interface pattern allows a class to export a portion of its private implementation in a structured way.
- A class can export a portion of its implementation by deriving from an interface class that exports access to the implementation.

157

Private Interface Example

- Suppose our `Eq` deal type has some private implementation that it wants to share with its visitors.

```
class Eq : public Deal {
    ~~~
private:
    T1 secret1_;
    T2 secret2_;
    T3 secret3_;
};
```

- It wants to share only a portion of its private implementation, and only with its visitors.

158

Private Interface Example

- We'll define an appropriate "view" interface class.

```
class EqVisitorView {
public:
    virtual ~EqVisitorView() {}
    virtual T1 secret1() const = 0;
    virtual T3 secret3() const = 0;
};
```

159

Visitors and Views

- We'll augment the Visitor interface to expect not only an object to visit, but a means to get that object's private view.

```
class DealVisitor {
public:
    virtual ~DealVisitor() {}
    virtual void visit(Bond *, BondVisitorView *) = 0;
    virtual void visit(Swap *, SwapVisitorView *) = 0;
    virtual void visit(Eq *, EqVisitorView *) = 0;
};
```

160

Implementing Private Interface

- The Eq class exports a portion of its implementation by deriving from the interface class.

```
class Eq final : public Deal, private EqVisitorView {
    ~~~
private:
    T1 secret1() const override
        { return secret1_; }
    T3 secret3() const override
        { return secret3_; }
    T1 secret1_;
    T2 secret2_;
    T3 secret3_;
};
```

161

Implementing Private Interface

- Notice that the interface is private!
- There is no predefined conversion of an `Eq` to an `EqVisitorView`.
- …except in a member of `Eq` itself. Like `Eq::accept`.

```
void Eq::accept(DealVisitor &v) {
    v.visit(this, this); // implicit conversion
}
```

- Note that there is an implicit conversion from a derived class to a private (or protected) base class within a member function of the derived class.
  - In other contexts, a `static_cast` or similar is required.

162

Safe, Private Access

- Use of the Private Interface is straightforward.

```
class ProcessDeal final : public DealVisitor {
    void visit(Bond *d, BondVisitorView *v) override;
    void visit(Swap *d, SwapVisitorView *v) override;
    void visit(Eq *d, EqVisitorView *v) override {
        ~~~ // access v->secret1() and v->secret3()
    }
};
```

- Only `Eq` is permitted to perform the conversion that gives access.

```
void Eq::accept(DealVisitor &v)
    { v.visit(this, this); } // implicit conversion
```

163

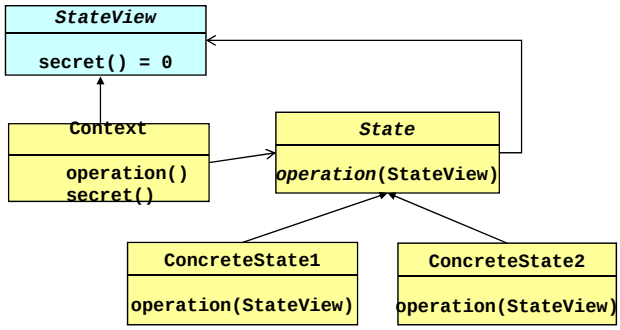
Drawbacks

- Unfortunately, Private Interface does have overhead that direct access does not.
  - Use of multiple inheritance will (in this case) result in larger objects.
  - Indirect access to private members through virtual functions is more costly than direct access.
- Through casting, it is possible to circumvent the safety this approach provides.
  - As always, the cast is the "insert bug here" operator.
  - As Stroustrup says, "C++ can prevent accident, but it cannot prevent fraud."

164

Another Example

- The State pattern could also make use of Private Interface.



- Notice a somewhat different structure in this application of Private Inheritance.
- We're duplicating part of the public interface of `Context`. `State` can remain ignorant of `Context`, and work entirely through `StateView`.

165

State View

```
class StateView {
public:
    virtual ~StateView();
    virtual void somePublicMember() = 0;
    virtual int secret() const = 0;
};
class Context final : private StateView {
public:
    void operation();
    void somePublicMember() override;
    ~~~
private:
    State *state_;
    int secret() const { return secret_; }
    int secret_;
};
```

166

State With Private Interface

```
class State {
public:
    virtual void operation(StateView *) = 0;
};

void Context::operation() {
    state_->operation(this); // conversion to StateView *
}
```

167

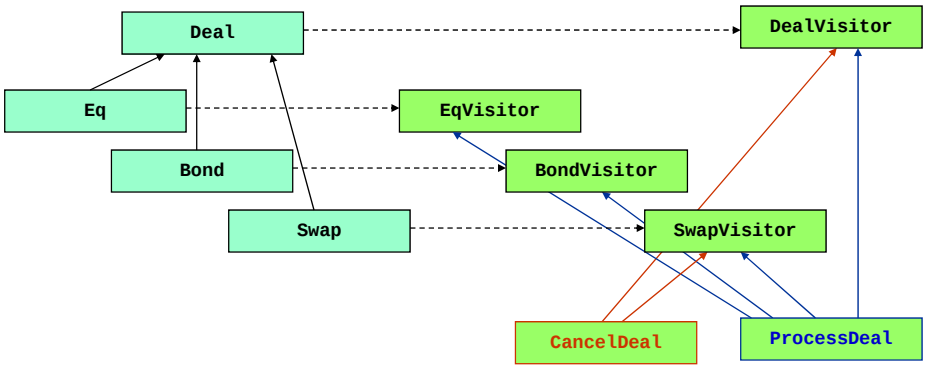
Acyclic Visitor

- Category
  - Object Behavioral
- Intent
  - Allow new functions to be added to existing class hierarchies without affecting those hierarchies, and without creating the troublesome dependency cycles that are inherent to the GOF Visitor pattern.
- Variances
  - Operations that can be applied to objects without changing their classes.
- Redesign Problems Solved
  - Algorithmic dependencies.
  - Inability to alter classes conveniently.
- See Also: Strategy, Decorator, Visitor

168

An Acyclic Visitor

- An acyclic visitor uses a little runtime type information and a lot of multiple inheritance to achieve more flexibility.



169

Acyclic Visitor Interfaces

- The `DealVisitor` class is now just a placeholder for cross-casting.

```
class DealVisitor {
public:
    virtual ~DealVisitor() = default;
};
```

- Note that `dynamic_cast` requires that the "from" type be a polymorphic type (that it have at least one virtual function).

170

Acyclic Visitor Interfaces

- Each specific type of visitor is concerned only with its corresponding deal class.

```
class BondVisitor {
public:
    virtual ~BondVisitor();
    virtual void visit(Bond *) = 0;
};
```

- These are capability interface classes.
- For example, `BondVisitor` specifies that a substitutable derived class has the ability to visit a `Bond`.

171

A Concrete Acyclic Visitor

- A concrete acyclic visitor type decides which deal types it will handle.

```
class ProcessDeal final
    : public DealVisitor,  // a deal visitor that handles
      public SwapVisitor,  // swaps,
      public BondVisitor,  // bonds, and
      public EqVisitor {   // equities
public:
    void visit(Swap *) override;
    void visit(Bond *) override;
    void visit(Eq *) override;
};
```

172

Another Concrete Acyclic Visitor

- A concrete acyclic visitor type may select a subset of types it is willing to handle.

```
class CancelDeal final
    : public DealVisitor,  // a deal visitor that handles
      public SwapVisitor { // swaps only
public:
    void visit(Swap *) override;
};
```

- More to the point, an Acyclic Visitor written to an earlier version of a hierarchy will still function as the hierarchy is augmented with new derived classes.

173

Acyclic Accept

- The acyclic accept operation uses RTTI to see if the operation applies to the deal type.
- This is a (reasonable!) capability query.

```
void Bond::accept(DealVisitor &v) {
    if (BondVisitor *bv           // can v deal with me?
            = dynamic_cast<BondVisitor *>(&v))
        bv->visit(this);          // yes!
    else
        // No?  OK, I'll deal with it...
        throw DoesNotUnderstand(typeid(v).name());
}
```

- The `Bond` is asking the `DealVisitor`, "Do you know about `Bonds`?"

174

Using an Acyclic Visitor

- The visitor is applied to the `Deal` object as in plain visitor.

```
auto &dv = getCurrentOperation();
auto d = getCurrentDeal();
d->accept(dv);
```

175

Using an Acyclic Visitor

- The calling sequence must be prepared to deal with runtime type-based errors.

```
try {
    auto &dv = getCurrentOperation();
    auto d = getCurrentDeal();
    d->accept(dv);
}
catch (DoesNotUnderstand &huh) {
    ~~~
}
```

176

Acyclic Catch-Alls

- The catch-all for Acyclic Visitor occurs on failure of the capability query.

```
void Swap::accept(DealVisitor &v) {
    if (SwapVisitor *sv
            = dynamic_cast<SwapVisitor *>(&v))
        // OK, this visitor knows about Swap
        sv->visit(this);
    else {
        // this visitor doesn't know about Swap...
        // this is the Acyclic Visitor catch-all
    }
}
```

177

Differing Information in Catch-Alls

- In the case of Acyclic Visitor, each visited type has its own catch-all.
  - We know the type that is being visited, but not the type of the Visitor.
- In the case of the GOF Visitor, the catch-all was in the Visitor.
  - We know the type of Visitor, but not what is being visited.

178

Warning!

- Visitor and Acyclic Visitor are too much fun!
- As a result, they tend to be overused.
- Other patterns that fall into the overused category are
  - Singleton
  - Decorator
- Please remember that pattern application should arise spontaneously from the context.
- A good designer will not try to force-fit a pattern into a context that does not call for it.

179

180