CS20006: Software Engineering
Module 06: Design Patterns

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Sources:

Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, & Vlissides

Gamma, Helm, Johnson, & Vlissides are known as Gang-of-Four (GoF)

Modern C++ Design: Generic Programming and Design Patterns Applied by Andrei Alexandrescu

Apr 09: 2021

Table of Contents

Design Pattern

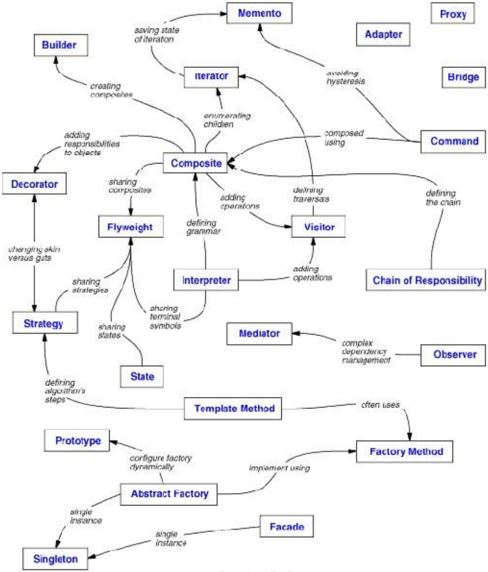
- A **Design Pattern**
  - **describes a problem**
    - *Occurring over and over again (in software engineering)*
  - **describes the solution**
    - *Sufficiently generic*
    - *Applicable in a wide variety of contexts*

Recurring Solution to a Recurring Problem

Catalogue of Design Patterns (GoF)

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Relationships of Design Patterns (GoF)



Describing a Design Pattern

- **Pattern Name and Classification**
  - The pattern's name conveys the essence of the pattern succinctly
- **Intent**
  - What does the design pattern do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?
- **Also Known As**
  - Other well-known names for the pattern
- **Motivation**
  - A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem
- **Applicability**
  - What are the situations in which the design pattern can be applied?
  - What are examples of poor designs that the pattern can address?
  - How can you recognize these situations?
- **Structure**
  - A graphical representation of the classes in the pattern UML
- **Participants**
  - The classes and/or objects participating in the design pattern and their responsibilities
- **Collaborations**
  - How the participants collaborate to carry out their responsibilities?
- **Consequences**
  - How does the pattern support its objectives?
  - What are the trade-offs and results of using the pattern?
  - What aspect of system structure does can be varied independently?
- **Implementation**
  - What pitfalls, hints, or techniques should you be aware of when implementing the pattern?
  - Are there language-specific issues?
- **Sample Code**
  - Code fragments to implement the pattern in specific language (C++ or C or Java)
- **Known Uses**
  - Examples of the pattern found in real life
- **Related Patterns**
  - What design patterns are closely related to this one?
  - What are the important differences?
  - With which other patterns should this one be used?

Describing a Design Pattern: Example of Iterator

- **Pattern Name and Classification**: Iterator
- **Intent**: Provide a way to *access* the *elements* of an *aggregate object (container) sequentially without exposing* its underlying representation
- **Also Known As**: Cursor
- **Motivation**
  - An aggregate object (list) should have a way to access its elements without exposing its internal structure
  - There is a need to traverse the list in different ways, depending on a specific task
  - Multiple traversals may be pending on the same list
  - The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object
- **Applicability**
  - to access an aggregate object's contents without exposing its internal representation
  - to support multiple traversals of aggregate objects
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)
- **Structure**: Given in Iterator section
- **Participants**
  - *Iterator* defines an interface for accessing and traversing elements
  - *ConcreteIterator* implements the Iterator, keeps track of the current position
  - *Aggregate* defines interface for Iterator
  - *ConcreteAggregate* implements the Iterator to return an instance of ConcreteIterator
- **Collaborations**: A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
- **Consequences**
  - Variety of the traversals of an aggregate
  - Iterators simplify the Aggregate interface
  - Multiple traversal on an aggregate
- **Implementation**
  - Who controls the iteration?
  - Who defines the traversal algorithm?
  - How robust is the iterator? (insert / delete)
  - Additional Iterator functionality including more operations, polymorphic iterators in C++, optional privileged access, Iterators for composites & Null iterators
- **Sample Code**: Given in Iterator section
- **Known Uses**: Iterators are common in OOP
- **Related Patterns**: Composite, Factory Method, and Memento

Pros & Cons of Design Pattern

- **Pros**
  - Help capture and disseminate expert knowledge
    - Promotes reuse and avoid mistakes
  - Provide a common vocabulary
    - Help improve communication among the developers
  - Reduce the number of design iterations:
    - Help improve the design quality and designer productivity
  - Patterns solve software structural problems attributable to:
    - Abstraction,
    - Encapsulation
    - Information hiding
    - Separation of concerns
    - Coupling and cohesion
    - Separation of interface and implementation
    - Single point of reference
    - Divide and conquer
- **Cons**
  - Design patterns do not directly lead to code reuse
  - To help select the right design pattern at the right point during a design exercise
    - At present no methodology exists

Example Design Patterns

- Iterator
- Singleton
- Factory Method
- Abstract Factory
- Visitor

Iterator Pattern

- **Pattern Name**: Iterator
- **Problem**: How to serve Patients at a Doctor's Clinic?
- **Solution**: Front-desk manages the order for patients to be called
    - By Appointment
    - By Order of Arrival
    - By Extending Gratitude
    - By Exception
- **Consequences**:
    - Patient Satisfaction
    - Clinic's Efficiency
    - Doctor's Productivity

Iterator Pattern: Intent

- Pattern Name and Classification:
  - Iterator
  - Behavioral
- Intent
  - Provide a way to <u>access</u>
  - the <u>elements</u>
  - of an <u>aggregate object (container)</u>
  - <u>sequentially</u>
  - <u>without exposing</u> its underlying representation.

ACCESS
- Read
- Write
- Read-Write

CONTAINERS
- Array
- Vector
- List
- Stack
- Queue
- Tree

SEQUENTIAL
- Forward
- Backward
- Bidirectional
- Random

Iterator Pattern: Sample Code


```
template <class Item>
class List { public: List(long size = DEFAULT_LIST_CAPACITY);
    long Count() const;
    Item& Get(long index) const; // ...
};
template <class Item>
class Iterator { public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected: Iterator();
};
template <class Item>
class ListIterator : public Iterator<Item> { public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
private: const List<Item>* _list; long _current;
};
template <class Item>
ListIterator<Item>::ListIterator (const List<Item>* aList) : _list(aList), _current(0) { }
template <class Item> void ListIterator<Item>::First() { current = 0; }
template <class Item> void ListIterator<Item>::Next() { current++; }
template <class Item> bool ListIterator<Item>::IsDone() const { return _current >= _list->Count();
}
template <class Item> Item ListIterator<Item>::CurrentItem () const {
    if (IsDone()) { throw IteratorOutOfBounds; } return _list->Get(_current);
}
```

Iterator Pattern: Sample Code


```
// Application using Iterator

void PrintEmployees (Iterator<Employee*>& i) {
for (i.First(); !i.IsDone(); i.Next()) {
    i.CurrentItem()->Print();
    }
}

List<Employee*>* employees;
// ...

ListIterator<Employee*> forward(employees);

ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);

PrintEmployees(backward);
```

Iterator Pattern: Sample Code (STL list)

Software Engineering

Partha P Das

Design Pattern

Iterator

UML Diagrams

Command

UML Diagrams

Usage

Singleton

UML Diagrams

Factory Method

Object Factory

UML Diagrams

Abstract Factory

Game

UML Diagrams

Visitor

Document Editor

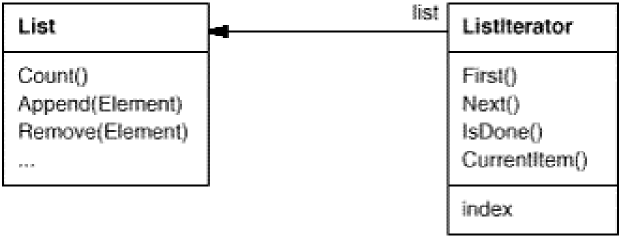
UML Diagrams

```
#include <iostream>
#include <list>

int main () { // constructing lists
    std::list<int> first;                                // empty list of ints
    std::list<int> second (4,100);                       // four ints with value 100
    std::list<int> third (second.begin(),second.end());  // iterating through second
    std::list<int> fourth (third);                       // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    std::cout << "The contents of fifth are: ";
    for (std::list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
        std::cout << *it << ' ';

    std::cout << '\n';

    return 0;
}

-----
Normal Constructor (2 Params): (1, 1)
The contents of fifth are: 16 2 77 29
```

Iterator Pattern: Sample Code (STL map)

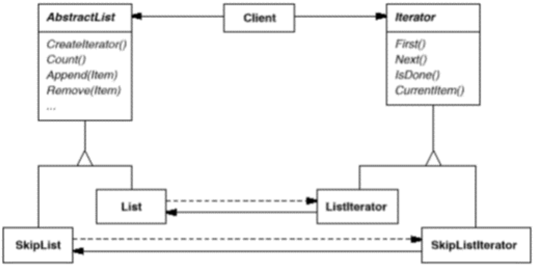

```
#include <iostream>
#include <map>

int main () { // map::begin/end
    std::map<char,int> mymap;
    std::map<char,int>::iterator it;

    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;

    // show content:
    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
        std::cout << it->first << " => " << it->second << '\n';

    return 0;
}

-----
Normal Constructor (2 Params): (1, 1)
a => 200
b => 100
c => 300
```

Iterator: Motivation



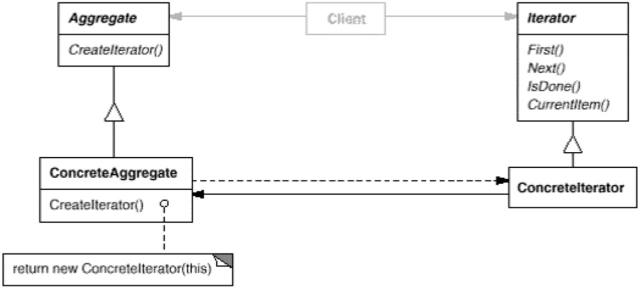
Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Iterator: Motivation



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

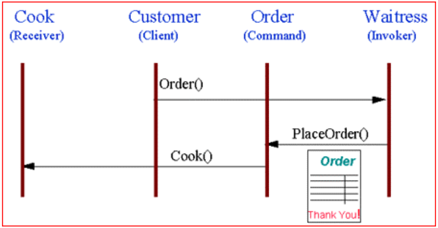
Iterator: Structure



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

"Check @ Diner" – A Command Pattern

- Customer places an Order with Waitress
- Waitress writes Order on check
- Order is queued to Cook



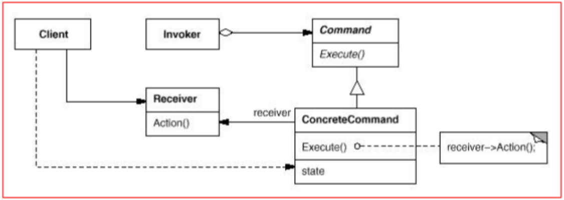
Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

"Compose & Send Mail" – A Command Pattern

- Sender composes a Mail with Compose Mail Editor (like Word) and presses "Send"
- Mail is queued on Outbox
- SendMail thread checks the connection and send the Mails from Outbox

Command

- **Command pattern's intent is to encapsulate a request in an object**



- The pattern's main piece is the *Command* class itself. Its most important purpose is to reduce the dependency between two parts of a system: the *invoker* and the *receiver*
- A typical sequence of actions is as follows:
  - The application (`Client`) creates a `ConcreteCommand` object (The dotted line), passing it enough information to carry on a task.
  - The application passes the `Command` interface of the `ConcreteCommand` object to the `Invoker`. The `Invoker` stores this interface.
  - Later, the `Invoker` decides it's time to execute the action and fires `Command`'s *`Execute`* virtual member function. The virtual call mechanism dispatches the call to the `ConcreteCommand` object. `ConcreteCommand` reaches the `Receiver` object (*the one that is to do the job*) and uses that object to perform the actual processing, such as calling its *`Action`* member function
  - Alternatively, the `ConcreteCommand` object might carry the processing all by itself. In this case, the receiver disappears

Command

- The invoker can invoke Execute at its leisure
- Most important, at runtime you can plug various actions into the invoker by replacing the Command object that the invoker holds. Two things are worth noting here
  - **Interface Separation**: The invoker is isolated from the receiver. The invoker is not aware of how the work is done
    - The invoker only calls for Execute for the Command interface it holds when certain circumstances occur
    - On the other side, the receiver itself is not necessarily aware that its Action member function was called by an invoker or otherwise
    - The invoker and receiver may be completely invisible to each other, yet communicate via Commands
    - Usually, an Application object decides the wiring between invokers and receivers
    - We can use different invokers for a given set of receivers, and we can plug different receivers into a given invoker – all without their knowing anything about each other
  - **Time Separation**: Command stores a ready-to-go processing request to be started later
    - In usual programming tasks, when we want to perform an action, we assemble an object, a member function of it, and the arguments to that member function into a call. For example:

      ```
      window.Resize(0, 0, 200, 100); // Resize the window
      ```

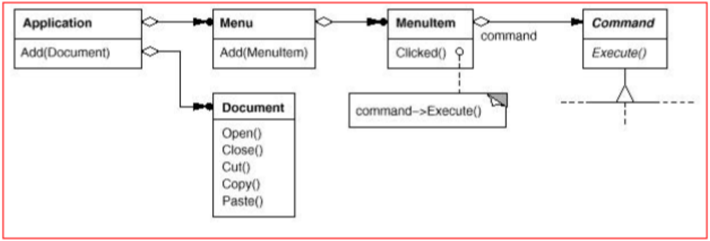
      The moment of initiating such a call is conceptually indistinguishable from the moment of gathering the elements of that call (the object, the procedure, and the arguments)
    - In the Command pattern, however, the invoker has the elements of the call, yet postpones the call itself indefinitely. The Command pattern enables delayed calls as in the following example:

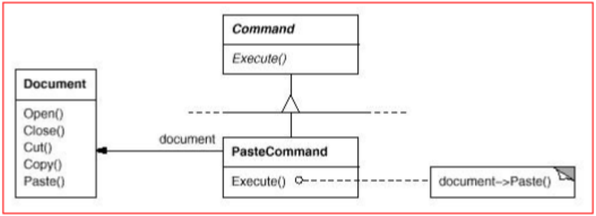
      ```
      Command resizeCmd(                              // GATHERING THE COMMAND (TASK)
      window, // Object
      &Window::Resize,     // Member function
      0, 0, 200, 100);     // Arguments
      // Later on...
      resizeCmd.Execute(); // Resize the window   // PERFORMING THE COMMAND
      ```

Command: Implementation

- From an implementation standpoint, two kinds of concrete Command classes can be identified.
  - **Forwarding Commands**: Some simply delegate the work to the receiver. All they do is call a member function for a Receiver object. They are called forwarding commands
  - **Active Commands**: Others do tasks that are more complex. They might call member functions of other objects, but they also embed logic that's beyond simple forwarding. They are called active commands
- Separating commands into active and forwarding is important for establishing the scope of a generic implementation
- Active commands cannot be canned – the code they contain is by definition application specific, but we can develop helpers for forwarding commands.
- Forwarding commands act much like pointers to functions and their C++ colleagues, functors, we call them **Generalized Functors** (Refer to *Functor* module for details.

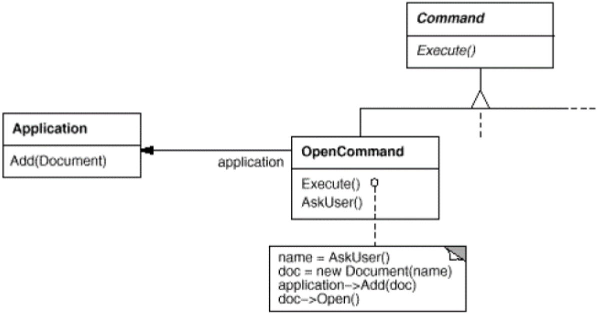
Command: Class Diagram Motivation





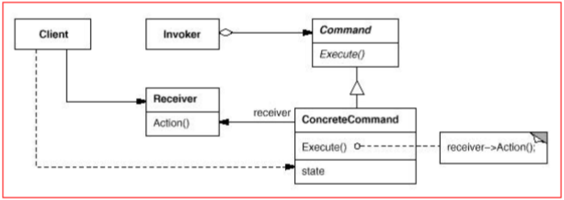
Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

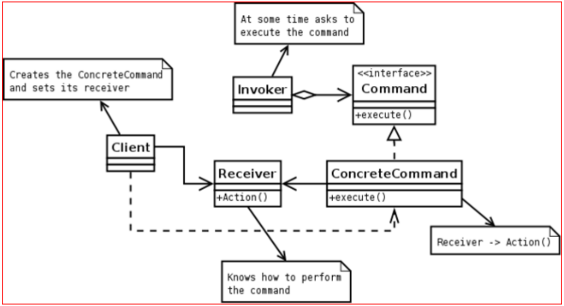
Command: Class Diagram Motivation



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

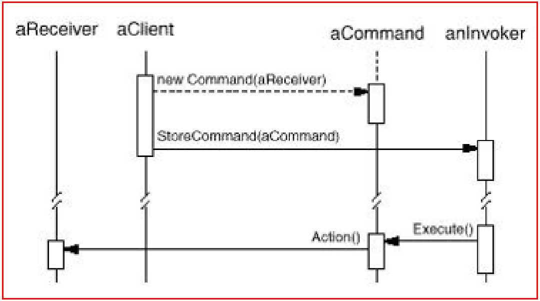
Command: Class Diagram Structure / Request-in-Object





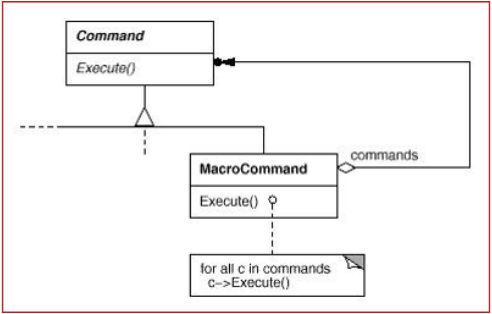
Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Command: Sequence Diagram Communication



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Command: Class Diagram
Macro / Composite Command



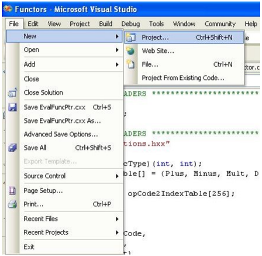
Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Sample Command Pattern Usage

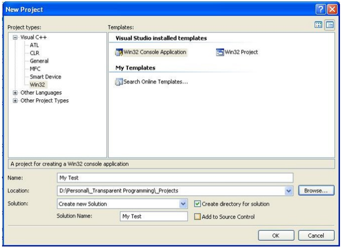
- Multi-level undo / redo
  - The program can keep a pair of stacks of commands
- Transactional behavior
  - Rollback for all-or-none operations
  - Installers / Databases
  - Two-phase commit
- Progress bars
- Wizards
  - What we see
    - Several pages of configuration for a single action
    - Action fires when the user clicks the "Finish" button
  - What we need
    - Separate user interface code from application code
    - Implement the wizard using a command object
    - The command class contains no user interface code
  - How it works
    - Created when the wizard is first displayed
    - Each wizard page stores its GUI changes in the command object
    - "Finish" simply triggers a call to execute()

Sample Command Pattern Usage

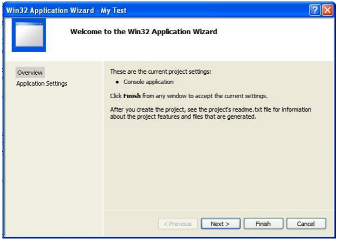
- Creating a project is MS Visual Studio



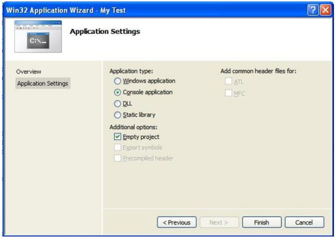
(1) Client initiates ConcreteCommand



(2) Parameter gathering



(3) Parameter gathering



(4) Execute

What is a Singleton?

- *Ensure a class only* **has** *one instance, and* **provides** *a global point of access to it* (GoF Book)
- We should use Singleton when we model types that conceptually have a unique instance in the application, such as `Keyboard`, `Display`, `PrintManager`, and `SystemClock`
  - Being able to instantiate these types more than once is unnatural at best, and often dangerous
- *A singleton is an improved global variable*
  - The improvement that Singleton brings is that *we cannot create a secondary object of the singleton's type*
  - *The Singleton object owns itself*
  - There is no special client step for creating the singleton – *the Singleton object is responsible for creating and destroying itself*
  - Managing a singleton's lifetime causes the most implementation headaches
- The Singleton design pattern is queer in that it's a strange combination:
  - *Its description is simple, yet its implementation issues are complicated*
- There is no **best** implementation of the Singleton design pattern
- Various Singleton implementations, including non-portable ones, are most appropriate depending on the problem at hand

Static Data + Static Functions != Singleton

- Can a Singleton be implemented by using static member functions and static member variables?

```
class Font { ... };
class PrinterPort { ... };
class PrintJob { ... };

class MyOnlyPrinter { public:
    static void AddPrintJob(PrintJob& newJob) {
        if (printQueue_.empty() && printingPort_.available()) {
            printingPort_.send(newJob.Data());
        }
        else { printQueue_.push(newJob); }
    }
private: // All data is static
    static std::queue<PrintJob> printQueue_;
    static PrinterPort printingPort_;
    static Font defaultFont_;
};

PrintJob somePrintJob("MyDocument.txt");
MyOnlyPrinter::AddPrintJob(somePrintJob);
```

- However, this solution has a **number of disadvantages** in some situations
  - The main problem is that static functions *cannot be virtual*, which makes it *difficult to change behavior* without opening `MyOnlyPrinter`'s code
  - A subtler problem of this approach is that it makes *initialization and cleanup difficult*. There is no central point of initialization and cleanup for `MyOnlyPrinter`'s data. Initialization and cleanup can be nontrivial tasks; for instance, `defaultFont_` can depend on the speed of `printingPort_`
- Singleton implementations therefore concentrate on *creating and managing a unique object* while *not allowing the creation of another one*

C++ Idioms for Singleton

- Most often, singletons are implemented in C++ by using some variation of the following idiom:

```
// Header file Singleton.h
class Singleton {
public:
    static Singleton* Instance() { // Unique point of access
        if (!pInstance_)
            pInstance_ = new Singleton;
        return pInstance_;
    }
... operations ...
private:
    Singleton(); // Prevents creating a new Singleton
    Singleton(const Singleton&); // Prevent creating a copy of the Singleton
    static Singleton* pInstance_; // The one and only instance
};
// Implementation file Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
```

- *All the constructors are private, user code cannot create Singletons*
- *Singleton's own member functions,* `Instance()` *in particular, are allowed to create objects*
- *The uniqueness of the Singleton object is enforced at compile time*
- **This is the essence of implementing the Singleton design pattern in C++**
- *If it's never used (no call to* `Instance()` *occurs), the Singleton object is not created*
    - The cost of this optimization is the (usually negligible) test incurred at the beginning of `Instance()`
    - The advantage of the build-on-first-request solution becomes significant if Singleton is expensive to create and seldom used

C++ Idioms for Singleton

- What is the problem with this simpler solution?

```
// Header file Singleton.h
class Singleton { public:
    static Singleton* Instance() { // Unique point of access
        return &instance_;
    }
    int DoSomething();
private: static Singleton instance_;
};
// Implementation file Singleton.cpp
Singleton Singleton::instance_;
```

- Although `instance_` *is a static member of Singleton* (just as `pInstance_` was in the previous example), there is an important difference between the two versions
- `instance_` is initialized *dynamically*, whereas `pInstance_` benefits from *static* initialization
- Compiler performs *static initialization before the very first statement* of the program is executed
- C++ does not define the *order of initialization for statically initialized objects* found in different translation units!
- Consider this code:

```
#include "Singleton.h"
int global = Singleton::Instance()->DoSomething(); // SomeFile.cpp
```

- Depending on the order of initialization that the compiler chooses for `instance_` and `global`, the call to `Singleton::Instance()` may return an object that has not been constructed yet. This means that we cannot count on `instance_` being initialized if other external objects are using it
- *The previous version too may have a similar problem if* `global` *gets initialized before* `pInstance_`:

```
#include "Singleton.h"
int global = Singleton::Instance()->DoSomething(); // SomeFile.cpp
------
Singleton* Singleton::pInstance_ = 0; // Implementation file Singleton.cpp
```

C++ Idioms for Singleton

- Canned solution:

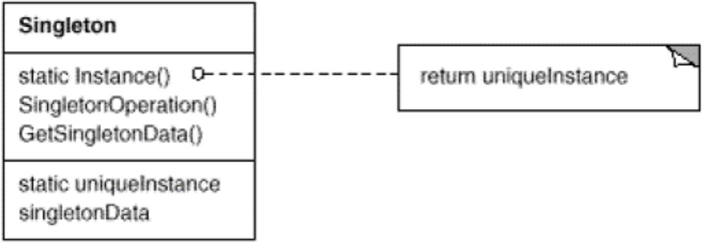
```
class Singleton {
    Singleton& Instance();  // Returning reference is better
                            // The user won't be tempted to delete
    ... operations ...
private:
    Singleton();
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);    // No assignment (though self)
                                             // should be allowed
    ~Singleton(); // Don't allow anyone to destroy
};
```

- Destroying a singleton is crucial and tricky
- Not destroying a singleton is not a *memory leak* (OS will wrap it up anyway), but can be serious *resource leak* (some database remains locked, etc.)
- The only correct way to avoid resource leaks is to delete the Singleton object during the application's shutdown
- The issue is that we have to choose the moment carefully so that no one tries to access the singleton after its destruction
- **Meyer's Singleton**: Using *local static object*

```
Singleton& Singleton::Instance() {
    static Singleton obj; // Created when used first time, destroyed by atexit() registra
    return obj;
}
```

- Other issues with singletons include:
  - *Dead Reference Problem*: Once a singleton is needed after it has been destroyed
  - *Multi-Threading Problem*: When singleton is to be used in a multi-threaded environment

Singleton: Class Diagram



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Virtual Constructor

- Holding a pointers or references to a polymorphic objects we can invoke (virtual) member functions (including the destructor) because their dynamic type is well known (although the caller might not know it)
- However, if we need to have the same flexibility in creating objects: we cannot – Constructors are always static.
- This is the paradox of *virtual constructors*
- We need virtual constructors when the information about the object to be created is inherently dynamic and cannot be used directly with C++ constructs
- Most often, polymorphic objects are created on the free store by using the new operator:

```
class Base { ... };
class Derived : public Base { ... };
class AnotherDerived : public Base { ... };
...
// Create a Derived object and assign it to a pointer to Base
Base* pB = new Derived;
```

- The issue here is the actual `Derived` type name appearing in the invocation of the new operator
- In a way, `Derived` here is much like the magic numeric constants we are advised not to use
- To create an object of the type `AnotherDerived`, we have to replace Derived with `AnotherDerived`
- We cannot make the `new` operator act dynamically: we must pass it a type known at compile time
- *Virtual member functions are dynamic: we can change their behavior without changing the call site, while each object creation is a stumbling block of statically bound, rigid code*
- *Invoking virtual functions binds the caller to the interface only (the base class), while (at least in C++) object creation binds the caller to the most derived, concrete class*

Object Factory

- Problems addressed by Object Factory:
  - For creation of objects, instead of invoking `new`, we may call a virtual function `Create` of some higher-level object to allow clients to change behavior through polymorphism
  - When we have the type knowledge, but not in a form that's expressible in C++ (for instance, say, a string containing `"Derived"`), so we actually know we have to create an object of type `Derived`, but we cannot pass a string containing a type name to `new` instead of a type name
- There are two basic cases in which object factories are needed:
  - A library needs not only to manipulate user-defined objects, but also to create them. For a framework for multiwindow document editors, we provide an abstract class `Document` from which users can derive classes such as `TextDocument` and `HTMLDocument`
  - `DocumentManager` class in the framework keeps the list of all open documents

```
class DocumentManager {
...
public:
    Document* NewDocument();
private:
    virtual Document* CreateDocument() = 0;
    std::list<Document*> listOfDocs_;
};
Document* DocumentManager::NewDocument() {  // Creating a new document is
    Document* pDoc = CreateDocument();      // tightly coupled with adding
    listOfDocs_.push_back(pDoc);            // it to DocumentManager's list
    ...                                     // of documents:
    return pDoc;
}
```

Object Factory

- The `CreateDocument` member function replaces a call to `new`
- `NewDocument` cannot use the new operator because the concrete document to be created is not known by the time `DocumentManager` is written
- To use the framework, we will derive from `DocumentManager` and override `CreateDocument`
- The GoF book calls `CreateDocument` a **factory method**
- The override is very simple and consists essentially of a call to `new`; for example:

```
Document* GraphicDocumentManager::CreateDocument() {
    return new GraphicDocument;
}
```

Object Factories in C++: Classes and Objects

- The entire issue of Object Factory lies in the fact that in

```
    Base* pB = new Derived;
```

  Derived is a class name, and we would like it to be a value, that is, an object
- So, what is a class, and what is an object?

Classes	Objects
Programmer creates	Program creates
Cannot create a new class at runtime	Cannot create an object at compile time
Created only at compile time	Created only at runtime
Don't have first-class status: Cannot copy a class, store it in a variable, or return it from a function	Has first-class status: Can copy an object, store it in a variable, or return it from a function

- Interestingly, there are languages in which classes are objects
- In those languages, we can create new classes at runtime, copy a class, store it in a variable, ...
- If C++ were such a language, we could have written code like the following:

```
// Warning-this is NOT C++
// Assumes Class is a class that's also an object

Class Read(const char* fileName);
Document* DocumentManager::OpenDocument(const char* fileName) {
    Class theClass = Read(fileName);
    Document* pDoc = new theClass;
    ...
}
```

- Such dynamic languages trade off some type safety and performance for the sake of flexibility
- As static typing is an important source of optimization, C++ took the opposite approach

Object Factories in C++: Implementation

- Consider `Shape`

```
class Shape {
    public:
    virtual void Draw() const = 0;
    virtual void Rotate(double angle) = 0;
    virtual void Zoom(double zoomFactor) = 0;
    ...
};
```

- We define a class `Drawing` that contains a complex drawing holding a collection of pointers to `Shape` such as a list, or a vector and provides operations to manipulate the drawing as a whole
- Two typical operations would be
  - *Saving a drawing as a file*. Provide a pure virtual function `Shape::Save(std:: ostream&)`:

```
class Drawing {
public:
    void Save(std::ofstream& outFile);
    void Load(std::ifstream& inFile);
    ...
};
void Drawing::Save(std::ofstream& outFile) {
    write drawing header // Unique ID of the type
    for (each element in the drawing) {
        (current element)->Save(outFile);
    }
}
```

  - *Loading a drawing from a previously saved file*: **How?**

Object Factories in C++: Implementation

- *Loading a* `Shape` *file*
  - Each Shape-derived object to saves an integral identifier at the very beginning of the file
  - Each object should have its own unique ID

```
// a unique ID for each drawing object type
namespace DrawingType { const int LINE = 1, POLYGON = 2, CIRCLE = 3 };
void Drawing::Load(std::ifstream& inFile) {
    // error handling omitted for simplicity
    while (inFile) {
        // read object type
        int drawingType;
        inFile >> drawingType;

        // create a new empty object
        Shape* pCurrentObject;
        switch (drawingType) {
            using namespace DrawingType;
            case LINE: pCurrentObject = new Line; break;
            case POLYGON: pCurrentObject = new Polygon; break;
            case CIRCLE: pCurrentObject = new Circle; break;
            default: handle error|unknown object type
        }

        // read the object's contents by invoking a virtual fn
        pCurrentObject->Read(inFile);
        add the object to the container
    }
}
```

Object Factories in C++: Implementation

- This is indeed an object factory
- It reads a type identifier from the file, creates an object of the appropriate type based on that identifier, and invokes a virtual function that loads that object from the file, and breaks the no-type-switch rule of OOP
  - Performs a switch based on a type tag
  - Collects the knowledge about all `Shape`-derived classes in the program in a single source file (`Drawing::Save` must include all headers of all possible shapes leading to high compile dependencies)
  - Hard to extend. To add a new shape `Ellipse`, to the system, we need to
    - Create the class itself
    - Add a distinct integral constant to the namespace `DrawingType`
    - Write that constant when saving an `Ellipse` object
    - Must add a label to the switch statement in `Drawing::Save`
- We would rather look for an alternate scalable solution
  - Note that the switch statement is the culprit which combines all types into one piece of code
  - So let us replace it with a function pointers table (recall discussions in Functor design)
    - We abstract every switch piece into virtual function

      ```
      Shape* CreateConcreteShape();
      ```

      to be implemented by specific `Shape` classes
    - We manage the switch by a function pointer table by a map

      ```
      std::map<int, CreateShapeCallback>
      ```

      indexed by unique IDs

      It is called a **Registry**
  - We put it together as `class ShapeFactory`

Object Factories in C++: Implementation

- Object Factory for `Shape`

```
class ShapeFactory {
    public: typedef Shape* (*CreateShapeCallback)(); // Switch function type
    private: typedef std::map<int, CreateShapeCallback> CallbackMap; // Registry type
    public:
        // Adds a shape-specific callback function to registry
        // Returns 'true' if registration was successful
        bool RegisterShape(int ShapeId, CreateShapeCallback CreateFn);

        // Removes a already registered function
        // Returns 'true' if the ShapeId was registered before
        bool UnregisterShape(int ShapeId);
        Shape* CreateShape(int ShapeId);
    private: CallbackMap callbacks_; // Registry
};
bool ShapeFactory::RegisterShape(int shapeId, CreateShapeCallback createFn) {
    return callbacks_.insert(CallbackMap::value_type(shapeId, createFn)).second;
}
bool ShapeFactory::UnregisterShape(int shapeId) { return callbacks_.erase(shapeId) == 1; }
```

- The connection of `Line` with the `ShapeFactory` is as follows

```
// Implementation module for class Line
// Create an anonymous namespace to make the function invisible from other modules
namespace {
    Shape* CreateLine() { return new Line; }
    const int LINE = 1; // The ID of class Line
    // Assume TheShapeFactory is a singleton factory
    const bool registered = TheShapeFactory::Instance().RegisterShape(LINE, CreateLine);
}
```

Object Factories in C++: Implementation

- Object Factory for `Shape`

```
Shape* ShapeFactory::CreateShape(int shapeId) {
    CallbackMap::const_iterator i =
        callbacks_.find(shapeId); // Retrieve the entry from registry
    if (i == callbacks_.end()) {
        // not found
        throw std::runtime_error("Unknown Shape ID");
    }
    return (i->second)(); // Invoke the creation function
}
```

Object Factories in C++: Implementation

- Generic Object Factory

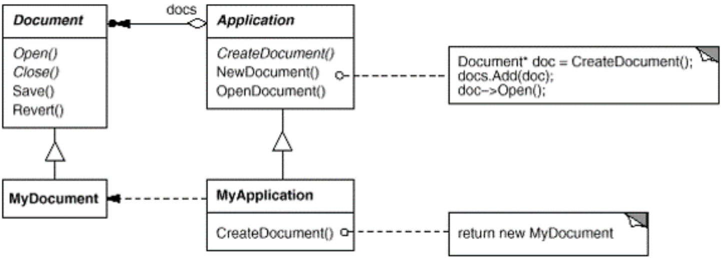
```
template<
    class AbstractProduct,      // Products inherit a base type (like, Shape)
    typename IdentifierType,    // Object that identifies the type of the concrete product
    typename ProductCreator =   // The function or functor is specialized
        AbstractProduct* (*)()> // for creating exactly one type of object
class Factory {
public:
    bool Register(const IdentifierType& id, ProductCreator creator) {
        return associations_.insert(AssocMap::value_type(id, creator)).second;
    }
    bool Unregister(const IdentifierType& id) {
        return associations_.erase(id) == 1;
    }
    AbstractProduct* CreateObject(const IdentifierType& id) {
        typename AssocMap::const_iterator i = associations_.find(id);
        if (i != associations_.end()) {
            return (i->second)();
        }
        // Handle error - how?
        // WE NEED A POLICY
            // Throw an exception?
            // Return a null pointer?
            // Terminate the program?
            // Dynamically load some library, register it on the fly, and retry the operation
    }
private: typedef std::map<IdentifierType, ProductCreator> AssocMap;
    AssocMap associations_;
};
```

Object Factories in C++: Implementation

- Generic Object Factory with error handling policy

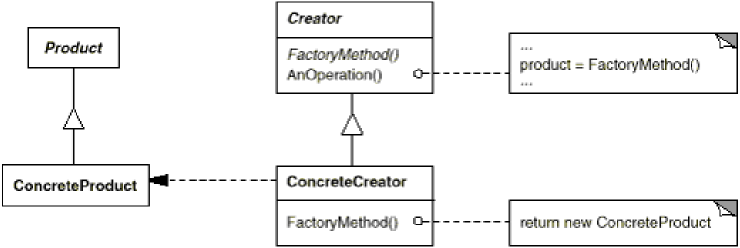
```
template <class IdentifierType, class ProductType> // Default Error Policy class
class DefaultFactoryError {
public:
    class Exception : public std::exception { // Nested class
    public:
        Exception(const IdentifierType& unknownId) : unknownId_(unknownId) { }
        virtual const char* what() { return "Unknown object type passed to Factory."; }
        const IdentifierType GetId() { return unknownId_; };
    private: IdentifierType unknownId_;
    };
protected:
    static ProductType* OnUnknownType(const IdentifierType& id) { throw Exception(id); }
};
template<class AbstractProduct, typename IdentifierType, // Factory with Error Policy, default
    typename ProductCreator = AbstractProduct* (*)(),
    template<typename, class> class FactoryErrorPolicy = DefaultFactoryError>
class Factory : public FactoryErrorPolicy<IdentifierType, AbstractProduct> { public:
    bool Register(const IdentifierType& id, ProductCreator creator) {
        return associations_.insert(AssocMap::value_type(id, creator)).second;
    }
    bool Unregister(const IdentifierType& id) { return associations_.erase(id) == 1; }
    AbstractProduct* CreateObject(const IdentifierType& id) {
        typename AssocMap::const_iterator i = associations_.find(id);
        if (i != associations_.end()) { return (i->second)(); }
        return OnUnknownType(id);
    }
private: typedef std::map<IdentifierType, ProductCreator> AssocMap;
    AssocMap associations_;
};
```

Factory Method: Class Diagram Motivation



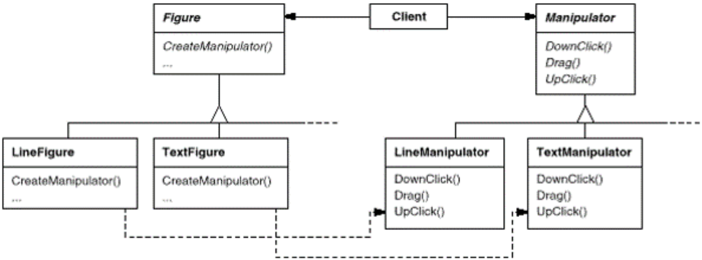
Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Factory Method: Class Diagram Structure



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

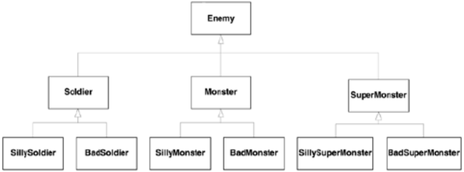
Factory Method: Class Diagram Example



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Game

- Abstract factories can be an important architectural component because they ensure that the right concrete objects are created throughout a system
- Consider designing a *find 'em and kill 'em* game, like Doom or Quake. The game has two levels:
  - **Easy Level**: This is for *beginners* where the enemy soldiers are rather dull, the monsters move like molasses, and the super-monsters are quite friendly
  - **Diehard Level**: This is for *hardcore gamers* where enemy soldiers fire three times a second and are karate pros, monsters are cunning and deadly, and really bad super-monsters appear once in a while
- A possible modeling of this would be a hierarchy with a base class `Enemy` and refined interfaces `Soldier`, `Monster`, and `SuperMonster` derived from it. Then we further derive
  - `SillySoldier`, `SillyMonster`, and `SillySuperMonster` from these interfaces (*Easy* level)
  - `BadSoldier`, `BadMonster`, and `BadSuperMonster` from these interfaces (*Diehard* level)
- This results in the following hierarchy:



- **Caveat**: In the game, an instantiation of `BadSoldier` and an instantiation of `SillyMonster` never *live* at the same time. A player plays either the easy game with `SillySoldiers`, `SillyMonsters`, and `SillySuperMonsters`, or the tough game with `BadSoldiers`, `BadMonsters`, and `BadSuperMonsters`
- *The two categories of types form two families; during the game, we always use objects in one of the two families, but we never combine them*

Game

- To enforce this *all-or-none* consistency, we gather the creation functions for all the game objects into a single interface:

```
class AbstractEnemyFactory { public:
    virtual Soldier* MakeSoldier() = 0;
    virtual Monster* MakeMonster() = 0;
    virtual SuperMonster* MakeSuperMonster() = 0;
};
```

- Then, for each play level, we implement a concrete enemy factory that creates enemies as prescribed:

```
class EasyLevelEnemyFactory : public AbstractEnemyFactory { public:
    Soldier* MakeSoldier() { return new SillySoldier; }
    Monster* MakeMonster() { return new SillyMonster; }
    SuperMonster* MakeSuperMonster() { return new SillySuperMonster; }
};
class DieHardLevelEnemyFactory : public AbstractEnemyFactory { public:
    Soldier* MakeSoldier() { return new BadSoldier; }
    Monster* MakeMonster() { return new BadMonster; }
    SuperMonster* MakeSuperMonster() { return new BadSuperMonster; }
};
```

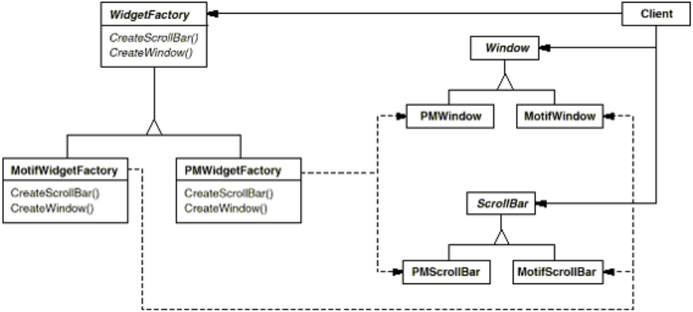
- Finally, we initialize a pointer to `AbstractEnemyFactory` with the appropriate concrete class:

```
class GameApp {
    ...
    void SelectLevel() {
        if (user chooses the Easy level) { pFactory_ = new EasyLevelEnemyFactory; }
        else { pFactory_ = new DieHardLevelEnemyFactory; }
    }
private:
    AbstractEnemyFactory* pFactory_; // Use pFactory_ to create enemies
};
```

Game: As an Abstract Factory

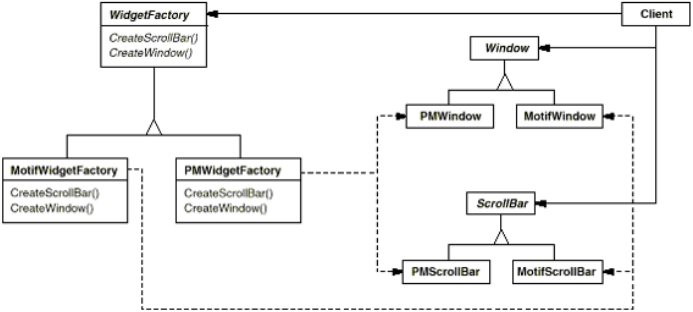
- **Advantages**
  - It keeps all the details of creating and properly matching enemies inside the two implementations of `AbstractEnemyFactory`
  - As the application uses `pFactory_` as the only object creator, consistency is enforced by design
  - *This is a typical usage of the* **Abstract Factory design** pattern
  - This design pattern prescribes collecting creation functions for families of objects in a unique interface
  - Then we must provide an implementation of that interface for each family of objects we want to create
  - The product types advertised by the abstract factory interface (`Soldier`, `Monster`, and `SuperMonster`) are called *Abstract Products*
  - The product types that the implementation actually creates (`SillySoldier`, `BadSoldier`, `SillyMonster`, and so on) are called *Concrete Products*
- **Disadvantages**
  - Abstract Factory is *type intensive*: The abstract factory base class (`AbstractEnemyFactory` in the example) must know about every abstract product that's to be created
  - Further, at least in the implementation just provided, each concrete factory class depends on the concrete products it creates
- **More Reading**: The solutions to the above problems, and the design of the generic abstract factory can be found in Alexandrescu's book

Abstract Factory: Class Diagram Motivation



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Abstract Factory: Class Diagram Structure



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Visitor Pattern

Let us consider a class hierarchy whose functionality we want to enhance. To do this, you can

- Add new classes
  - Easy
  - Derive from a leaf class and implement the needed virtual functions
  - Don't need to change or recompile any existing classes
  - It's code reuse at its best
- Add new virtual member functions
  - Difficult
  - We must add virtual member functions to the root class, and possibly to many other classes in the hierarchy
  - This is a major operation
  - It modifies the root class on which all the hierarchy and clients are dependent
  - As the ultimate result, we recompile the world

Visitor Pattern: Document Editor Example

Document Editor

- A document editor deals with document elements such as paragraphs, vector drawings, and bitmaps that are represented as classes derived from a common root, say, `DocElement`
- The document is a structured collection of pointers to `DocElements`
- We need to iterate through this structure and perform operations such as *spell checking*, *reformatting*, and *statistics gathering*
- Ideally, we should implement those operations mostly by adding code, not by modifying existing code
- Furthermore, for ease maintenance if we put all the code pertaining to, say, getting document statistics in one place
- Document statistics may include the number of *characters*, *nonblank characters*, *words*, and *images*.
- These would naturally belong to a class called `DocStats`

Visitor Pattern: Document Editor Example

```
class DocStats { unsigned int chars_, nonBlankChars_, words_, images_; ...
public:
    void AddChars(unsigned int charsToAdd) { chars_ += charsToAdd; }
    ...similarly defined AddWords, AddImages...
    void Display(); // Display the statistics to the user in a dialog box
};
class DocElement { ...
    virtual void UpdateStats(DocStats& statistics) = 0; // Helps the "Statistics" feature
};
class Paragraph: public DocElement { ... };
class RasterBitmap: public DocElement { ... };
void Paragraph::UpdateStats(DocStats& statistics) {
    statistics.AddChars(number of characters in the paragraph);
    statistics.AddWords(number of words in the paragraph);
}
void RasterBitmap::UpdateStats(DocStats& statistics) {
    statistics.AddImages(1); // A raster bitmap counts as one image and nothing else
}
void Document::DisplayStatistics() { DocStats statistics;
    for (each DocElement in the document) { element->UpdateStats(statistics); }
    statistics.Display();
}
```

- It requires `DocElement` and its derivees to have access to the `DocStats` definition. Consequently, every time you modify `DocStats`, you must recompile the whole `DocElement` hierarchy.
- The actual operations of gathering statistics are spread throughout the `UpdateStats`. A maintainer debugging or enhancing the Statistics feature must search and edit multiple files.
- This does not scale with respect to adding other similar operations. To add an operation such as *increase font size by one point*, we need to add another virtual function to `DocElement`

Visitor Pattern: Document Editor Example

- To break the dependency of DocElement on DocStats we move all operations into the DocStats class and let it figure out what to do for each concrete type
- This implies that DocStats has a member function void UpdateStats(DocElement&)
- The document then simply iterates through its elements and calls UpdateStats for each of them
- This solution effectively makes DocStats invisible to DocElement
- However, now DocStats depends on each concrete DocElement that it needs to process. *If the object hierarchy is more stable than its operations, the dependency is not very annoying*
- The problem is: *The implementation of UpdateStats has to rely on the so-called type switch*
- A type switch occurs whenever you query a polymorphic object on its concrete type and perform different operations with it depending on what that concrete type is
- DocStats::UpdateStats is bound to do such a type switch, as in the following:

```
void DocStats::UpdateStats(DocElement& elem) {
    if (Paragraph* p = dynamic_cast<Paragraph*>(&elem)) {
        chars_ += p->NumChars();
        words_ += p->NumWords();
    }
    else if (dynamic_cast<RasterBitmap*>(&elem)) { ++images_; }
    else ... add one 'if' statement for each type of object you inspect
}
```

- **Visitor Pattern** comes to use
- We need new functions to act *virtual*, but we don't want to add a new virtual function for each
- For this, we must implement a *unique bouncing virtual function* in the DocElement hierarchy, a function that *teleports* work to a different hierarchy
- DocElement hierarchy is called the *visited hierarchy*, and operations belong to a new *visitor hierarchy*
- Each implementation of the bouncing virtual function calls a *different* function in the visitor hierarchy
- The functions in the visitor hierarchy called by the bouncing function are virtual

Visitor Pattern: Document Editor Example

- First, we define an abstract class DocElementVisitor that defines an operation for each type of object in the DocElement hierarchy

```
class DocElementVisitor {
public:
    virtual void VisitParagraph(Paragraph&) = 0;
    virtual void VisitRasterBitmap(RasterBitmap&) = 0;
    ... other similar functions ...
};
```

- Next we add the bouncing virtual function, called Accept, to the class DocElement hierarchy, one that takes a DocElementVisitor& and invokes the appropriate VisitXxx function on its parameter

```
class DocElement {
public:
    virtual void Accept(DocElementVisitor&) = 0;
    ...
};
void Paragraph::Accept(DocElementVisitor& v) { v.VisitParagraph(*this); }
void RasterBitmap::Accept(DocElementVisitor& v) {  v.VisitRasterBitmap(*this); }
```

- Now here's DocStats:

```
class DocStats : public DocElementVisitor {
public:
    virtual void VisitParagraph(Paragraph& par) {
        chars_ += par.NumChars();
        words_ += par.NumWords();
    }
    virtual void VisitRasterBitmap(RasterBitmap&) { ++images_; }
    ...
};
```

Visitor Pattern: Document Editor Example

- The driver function `Document::DisplayStatistics` creates a `DocStats` object and invokes `Accept` on each `DocElement`, passing that `DocStats` object as a parameter. As the `DocStats` object visits various concrete `DocElements`, it gets nicely filled up with the appropriate data: *no type switching*!

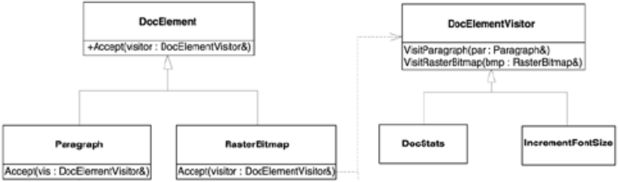
```
void Document::DisplayStatistics() {
    DocStats statistics;
    for (each DocElement in the document) { element->Accept(statistics); }
    statistics.Display();
}
```

- Let's analyze the resulting context. We have added a new hierarchy, rooted in `DocElementVisitor`
- This is a hierarchy of operations: each of its classes is actually an operation, like `DocStatsis`
- Adding a new operation becomes as easy as deriving a new class from `DocElementVisitor`
- No element of the `DocElement` hierarchy needs to be changed
- For instance, let's add a new operation, `IncrementFontSize`, as a helper in implementing an Increase Font Size hot key or toolbar button

```
class IncrementFontSize : public DocElementVisitor {
public:
    virtual void VisitParagraph(Paragraph& par) {
        par.SetFontSize(par.GetFontSize() + 1);
    }
    virtual void VisitRasterBitmap(RasterBitmap&) {
    // nothing to do
    }
    ...
};
```

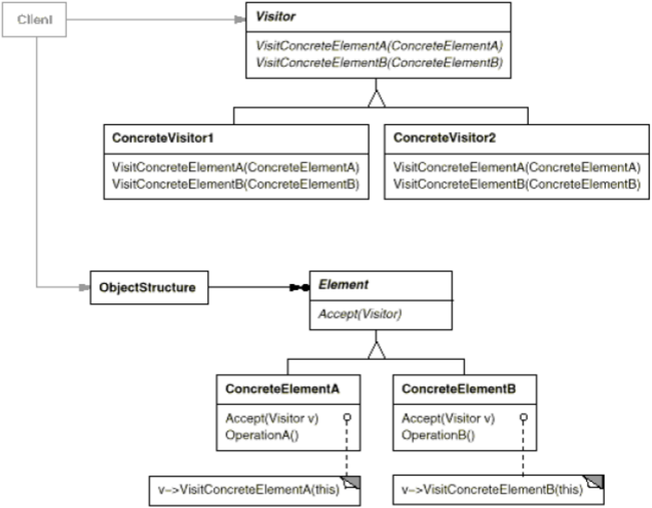
Visitor Pattern: Document Editor Example

- That's **Visitor Pattern**. No change to the DocElement class hierarchy is needed, and there is no change to the other operations
- We just add a new class. DocElement::Accept bounces IncrementFontSize objects just as well as it bounces DocStats objects



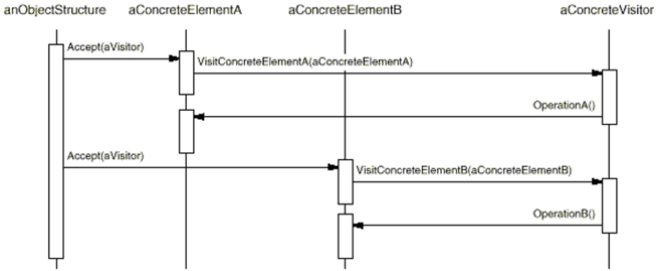
The visitor and visited hierarchies, and how operations are teleported

Visitor Pattern: Class Diagram Structure



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*

Visitor Pattern: Sequence Diagrams Communication



Sources: *Design Patterns: Elements of Reusable Object-Oriented Software*