Modern C++ Design

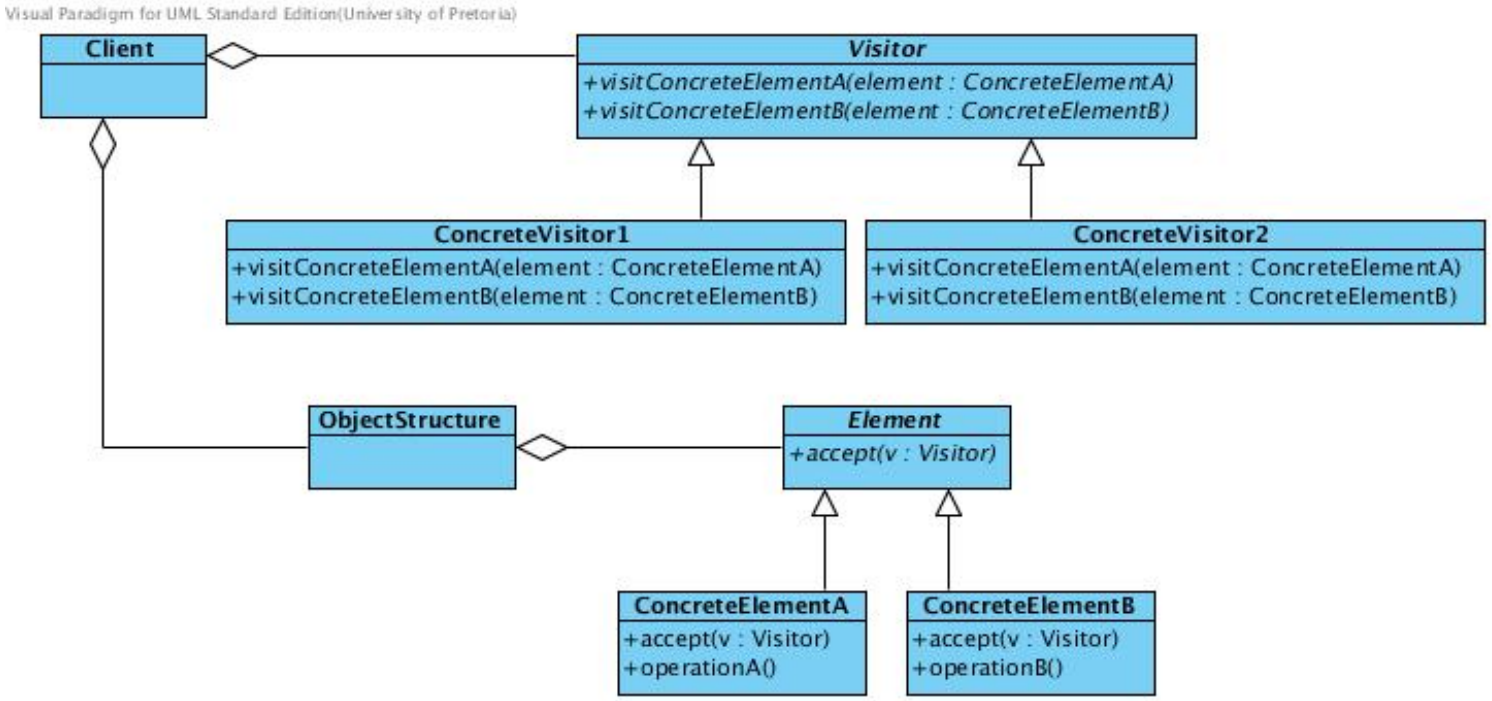
Chapter 10: Visitor

Introduction

Visitor gives a lot of flexibility at the expense of disabling features that designers take for granted.

- it enables addition of behaviour (new `ConcreteVisitor`) without adding code to where the behaviour is be be applied or recompiling other classes in the Visitor hierarchy or the clients.
- a leaf class cannot be added to the `Element` hierarchy without recompiling the Visitor hierarchy and all its clients.

Visitor works well with stable hierarchies and with heavy processing needs




```
returnType ConcreteVisitor1::visit(
                        ConcreteElementA element) {
  element.operationA();
}
returnType ConcreteElementA::accept(Visitor v) {
  v.visit(this);
}
```

Goal of the Chapter

'Craft' a dependable version of Visitor, leaving as little burden on the application programmer as possible.

After reading the chapter, you should:

- understand how the Visitor works
- know when to and when not to use the Visitor pattern
- understand the drawbacks of the GoF implementation and know how to overcome these
- understand a generic implementation of the Visitor and be able to apply it to problems requiring a Visitor pattern.

Visitor Basics

In general adding new functionality to a hierarchy can be done in two ways:

- By adding a new class
- By adding a new virtual member function

Adding a new class is easy - no need to recompile existing classes. Good code reuse.

Adding a new virtual function is difficult - to make use of polymorphic properties you may need to add a virtual function in the base class and possibly other classes. This requires a recompilation of the 'world' to take place.

This is in contrast to what the visitor allows....

Consider a Document Editor example

- Document elements, such as paragraphs, vector drawings, bitmaps etc. are represented as classes and derive from a common root, `DocElement`
- A document is a structured collection of pointers to `DocElement`s.
- You wish to iterate through the document structure and perform operations such as spellchecking, reformatting, gathering of document statistics...
- This type of operation requires code to be added, not modified. The addition of code in one place will ease code maintenance.

- Consider the following Document Statistics class:

```
class DocStats {
  ...
  public:
    void AddChars(unsigned int charsToAdd) {
      chars_ += charsToAdd;
    }
    // Similarly for AddWords, AddImages ...

    // Display the statistics to the user in
    // a dialog box
    void Display();
  private:
    unsigned int chars_, nonBlankChars_,
                 words_, images_;
};
```

Using a classic OO approach to gather the stats will result in a virtual function in **DocElement** that deals with gathering stats

```
class DocElement {
  ...
  // This member function helps with the
  // "Statistics" feature
  virtual void UpdateStats(DocStats& statistics)=0;
};
```

Each concrete document element defines the function in its own way.

```
class Paragraph :public DocElement {
 ...
  void UpdateStats(DocStats& statistics) {
   statistics.AddChars(number of chars
                       in the paragraph);
   statistics.AddWords(number of words
                       in the paragraph);
  }
 ... }
class RasterBitMap :public DocElement {
 ...
 void UpdateStats(DocStats& statistics) {
   statistics.AddImages(1);
 }
 ... }
```

The 'driver' function will be defined as follows:

```
void Document::DisplayStatistics() {
 DocStats statistics;
 foreach (DocElement in the document as element) {
   element->UpdateStats(statistics);
 }
 statistics.Display();
}
```

Disadvantages of this approach:

- `DocElement` and all classes deriving from it must have access to the `DocStats` definition. Every change to `DocStats` will result in the `DocElement` hierarchy needing to be recompiled.
- The operations for gathering the stats are spread throughout the `UpdateStats` implementations. Maintenance of the stats feature requires that multiple classes/files need to be considered.
- Adding additional operations that relate to stats gathering requires a virtual function to be added to `DocElement`. This implies adding the function to all classes that derive from `DocElement` as well.

Breaking the dependency of `DocElement` on `DocStats` is necessary.

1. Move all operations into `DocStats`.
   - `DocStats` must then figure out what to do with each concrete type.
   - `DocStats` therefore requires a member function `void UpdateStats(DocElement&)`.
   - Document now iterates through its elements and calls `UpdateStats` for each of them.
   - `DocStats` is now invisible to `DocElement`, however `DocStats` now depends on each `DocElement` that it needs to process.
   - `UpdateStats` now relies on a *type switch* - call a polymorphic object on its concrete type and perform different operations with it depending on that concrete type.

```
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 ...
}
```

– This code is hard to understand, hard to extend, and hard to maintain.

– The order in which the *dynamic casts* are done are of utmost importance - Refer to Chapter 8

2. Make use of the visitor pattern
   - you require new functions to act as virtual without adding a new virtual function for each operation
   - make use of what is referred to as a *unique bouncing virtual function* in the `DocElement` hierarchy. This function 'teleports' work to another hierarchy. The `DocElement` hierarchy is referred to as the *visited* hierarchy and the operations belong to the new *visitor* hierarchy.
   - "Each implementation of the bouncing virtual function calls a *different* function in the visitor hierarchy.", thereby selecting the visited types. "The functions in the visitor hierarchy called by the bouncing function are virtual.", thereby selecting the operations.

The `DocElementVisitor` class

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

Add the bouncing virtual function (`Accept`) to the `DocElement` hierarchy. It takes a `DocElementVisitor&` and invokes the appropriate `VisitXxxx` function.

```
class DocElement {
  public:
    virtual void Accept(DocElementVisitor&) = 0;
    ...
};
```

```
void Paragraph::Accept(DocElementVisitor& v) {
  v.VisitParagraph(*this);
}

void RasterBitmap::Accept(DocElementVisitor& v) {
  v.VisitRasterBitmap(*this);
}
```

`DocStats` now changes

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

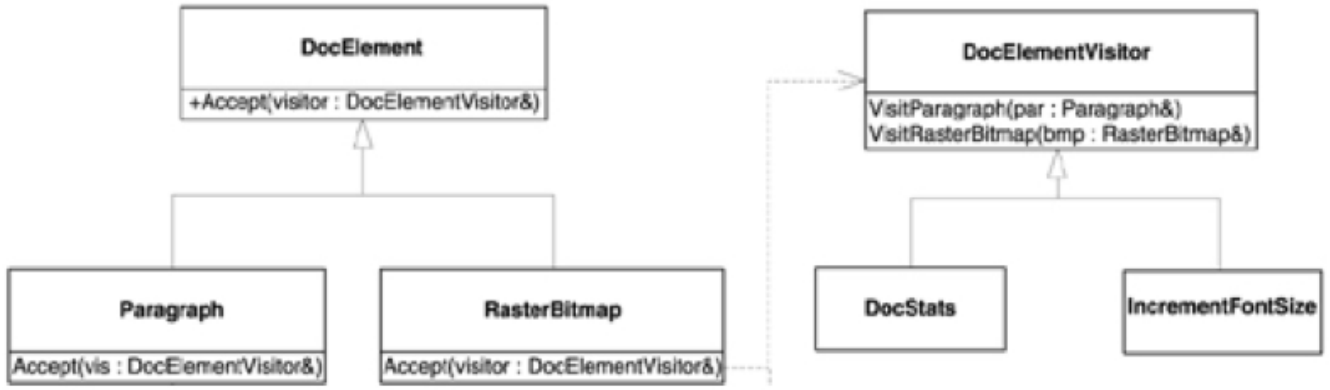
```
    virtual void VisitRasterBitmap(RasterBitmap&) {
        ++images;
    }
    ...
};
```

The 'driver' function changes as follows:

```
void Document::DisplayStatistics() {
 DocStats statistics;
 foreach (DocElement in the document as element) {
   // element->UpdateStats(statistics);
   element->Accept(statistics)
 }
 statistics.Display();
}
```

Adding a visitor, for example to increase font size is easy!

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



For each document element type, the `DocElementVisitor` defined a virtual function with the element type as part of the function name. This can be simplified by letting the compiler determine the overloaded function to call.

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

The `Accept` functions for each of the document elements also simplifies.

```
void Paragraph::Accept(DocElementVisitor& v) {
  // v.VisitParagraph(*this);
  v.Visit(*this);  }
```

```
void RasterBitmap::Accept(DocElementVisitor& v) {
  //v.VisitRasterBitmap(*this);
  v.Visit(*this);  }
```

NOTE: the static type of `*this` in `Paragraph::Accept` is `Paragraph&`, and similarly for `RasterBitmap`. This means these functions are quite different and cannot be factored out.

What-if there is no matching Visit function for the type received and you do not want a compile-time error! Can provide a catch-all as the last resort.

```
class DocElementVisitor {
  public:
    // virtual void VisitParagraph(Paragraph&) = 0
    virtual void Visit(Paragraph&) = 0;
    // virtual void VisitRasterBitmap(RasterBitmap
    virtual void Visit(RasterBitmap&) = 0;
```

```
    ...
    virtual void Visit(DocElement&) = 0;
};
```

The Acyclic Visitor

The previous example exhibits cyclic dependencies... `DocElement` needs `DocElementVisitor`, and `DocElementVisitor` needs all of `DocElement`'s hierarchy

Dividing the classes into files needs to be done with care

```
// File DocElementVisitor.h
class DocElement;
class Paragraph;
class RasterBitmap;
  ... forward declarations for all
      DocElement derivatives

class DocElementVisitor {
  public:
    virtual void Visit(Paragraph&) = 0;
    ...
};
```

```
// File DocElement.h
class DocElementVisitor;

class DocElement {
  public:
    virtual void Accept(DocElementVisitor&) = 0;
    ...
};
```

Adding new classes to the `DocElement` hierarchy becomes complex - something we are not meant to do as the premise was that this hierarchy remains stable. But it may be necessary at a stage to do so, for example adding the document element `VectorGraphic`.

- Add a new forward declaration for `VectorGraphic` in `DocElementVisitor.h`
- Add a new pure overload to `DocElementVisitor`

```
class DocElementVisistor {
   ...
   virtual void Visit(VectorGraphic&) = 0;
};
```

- Every concrete visitor must implement `Visit(VectorGraphic&)` as a do-nothing as opposed to a pure-virtual function
- Implement `Accept` in the `VectorGraphic` class
- Recompile all the `DocElement` and `DocElementVisitor` hierarchies.

Removing cycles

- In 1996, Robert Martin developed a variation of the Visitor pattern to remove cycles.
- Develop a base class for the visitor hierarchy that only carries type information
- The visited hierarchy's `Accept` function accepts a reference to this base class visitor and applies a *dynamic cast* to detect the matching visitor
- A match results in a jump from the visited hierarchy to the visitor hierarchy

How is this implemented?

- Define the archetypical base class, the strawman

```
class DocElementVisitor {
  public:
    virtual ~DocElementVisitor() {}
};
```

– The virtual destructor does 2 things: (i) gives `DocElementVisitor` RTTI capabilities; and (ii) it ensures correct polymorphic destruction of `DocElementVisitor` objects.

- `DocElement` class remains that same
- for each derived class of `DocElement` a visiting class is defined with one function `VisitXxxx`

```
class ParagraphVisitor {
  public:
    virtual void VisitParagraph(Paragraph&) = 0;
};
```

- `Paragraph::Accept` changes to

```
void Paragraph::Accept(DocElementVisitor& v) {
  if (ParagraphVisitor* p =
          dynamic_cast<ParagraphVisitor*>(&v)) {
      p->VisitParagraph(*this);
  } else {
```

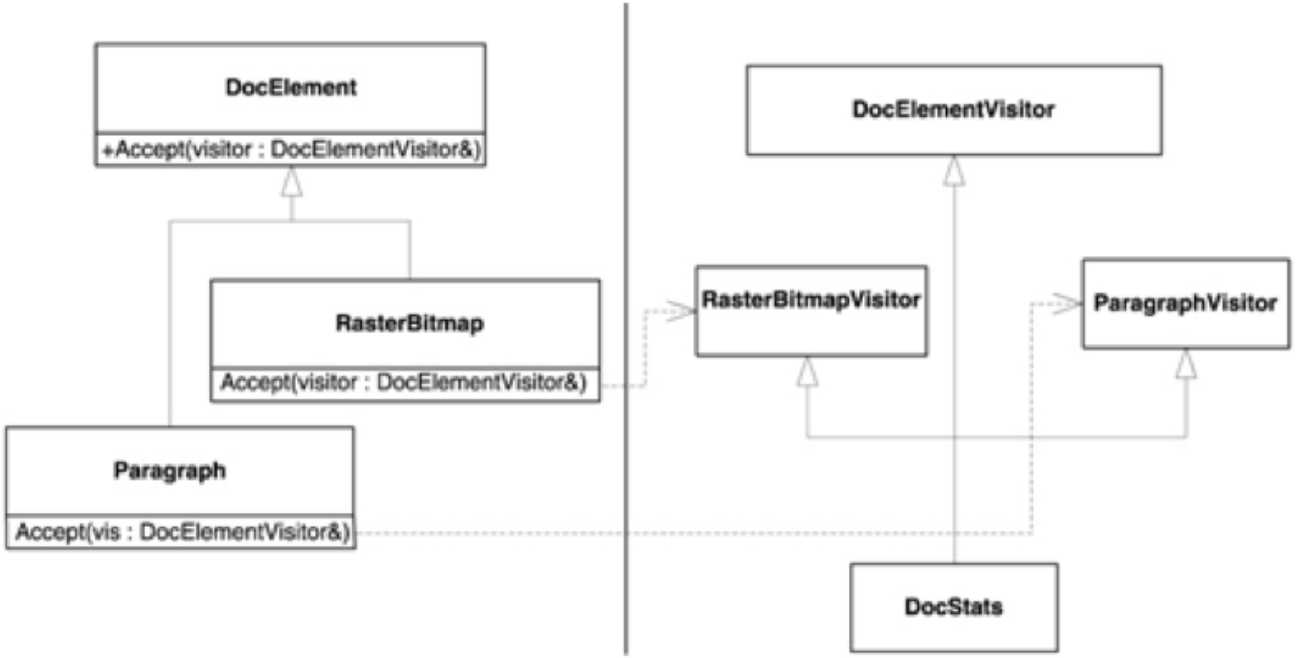
```
        // call a catch-all function
    }
}
```

- Other document element classes define similar implementations of `Accept`.
- A concrete visitor now derives from `DocElementVisitor` and all archetypical visitors of all classes that is of interest to it.

```
class DocStats :
            public DocElementVisitor, // required
            public ParagraphVisitor,
            public RasterBitmapVistor {
  public:
   virtual void VisitParagraph(Paragraph& par)
     chars_ += par.NumChars();
     words_ += par.NumWords();
   }
```

```
        virtual void VisitRasterBitmap(RasterBitmap&)
            ++images;
        }
        ...
};
```

- The resulting class structure is given by



Cycles have been removed, but parallel hierarchies need to be maintained.

A Generic Implementation of Visitor

Divide the implementation into 2 units

- Visitable classes - the hierarchy to be visited (add operations to)
- Visitor classes - where the actual operations are implemented

Refer to Table 10.1 for Component names

Table 10.1. Component Names

Name	*Belongs To*	*Represents*
`BaseVisitable`	Library	The root of all hierarchies that can be visited
`Visitable`	Library	A mix-in class that confers visitable properties to a class in the visitable hierarchy
`DocElement`	Application	A sample class—the root of the hierarchy we want to make visitable
`Paragraph,RasterBitmap`	Application	Two sample visitable concrete classes, derived from `DocElement`
`Accept`	Library and application	The member function that's called for visiting the visitable hierarchy
`BaseVisitor`	Library	The root of the visitor hierarchy
`Visitor`	Library	A mix-in class that confers visitor properties to a class in the visitor hierarchy
`Statistics`	Application	A sample visitor concrete class
`Visit`	Library and application	The member function that is called back by the visitable elements, for its visitors

Begin by defining the *Visitor* hierarchy - provide a base class and classes that define the `Visit` operation for the given type

```
class BaseVisitor {
  public:
    virtual ~BaseVisitor() {}
};

template <class T, typename R = void>
class Visitor {
  public:
    typedef R ReturnType;
    virtual ReturnType Visit(T&) = 0;
};
```

To visit the hierarchy you derive your visitor from `BaseVisitor` and all the `Visitor` instantiations as types you want to visit

```
class SomeVisitor :
  public BaseVisitor, // required
  public Visitor<Paragraph>,
  public Visitor<RasterBirmap> {
  public:
    void Visit(Paragraph&);
    void Visit(RasterBitmap&);
};
```

Define the *Visitable* hierarchy

- Define the base for the hierarchy

```
template <typename R = void>
class BaseVisitable {
  public:
    typedef R ReturnType;
    virtual ~BaseVisitable() {}
    virtual ReturnType Accept(BaseVisitor&) = 0;
};
```

- Need to find a way to implement `Accept` in the framework. This is done with macros - the client must insert `DEFINE_VISITABLE()` in each class of the visitable hierarchy. The decision is made in the name of *efficiency*

```
#define DEFINE_VISITABLE() \
 virtual ReturnType Accept(BaseVisitor& guest) \
 { return AcceptImpl(*this, guest); }
```

  - the macro defines `Accept` to forward to another function `AcceptImpl`
  - `AcceptImpl` is parameterised with the type of `*this`
  - `AcceptImpl` therefore gains access to the static type.
  - class `BaseVisitable` changes to implement `AcceptImpl`

```
template <typename R = void>
class BaseVisitable {
  public:
    typedef R ReturnType;
    virtual ~BaseVisitable() {}
    virtual ReturnType Accept(BaseVisitor&)=0;
  protected:
    // give access only to the hierarchy
    template <class T>
    static ReturnType AcceptImpl(T& visited,
                                 BaseVisitor& guest) {
      // Apply the acyclic visitor
      if (Visitor<T>* p =
          dynamic_cast<Visitor<T>*>(&guest)) {
        return p->Visit(visited);
      }
```

```
        return ReturnType();
    } };
```

- The design of the *Visitor* and *Visitable* is given by:

```
// Visitor part
class BaseVisitor
{
public:
    virtual ~BaseVisitor() {}
};
template <class T, typename R = void>
class Visitor
{
public:
typedef R ReturnType; // Available for clients
    virtual ReturnType Visit(T&) = 0;
};
```

```
// Visitable part
template <typename R = void>
class BaseVisitable
{
public:
    typedef R ReturnType;
    virtual ~BaseVisitable() {}
    virtual R Accept(BaseVisitor&) = 0;
protected:
template <class T>
static ReturnType AcceptImpl(T& visited,
                             BaseVisitor& guest) {
        // Apply the Acyclic Visitor
        if (Visitor<T>* p =
            dynamic_cast<Visitor<T>*>(&guest)) {
            return p->Visit(visited);
```

```
        }
        return ReturnType();
    }
};
#define DEFINE_VISITABLE() \
  virtual ReturnType Accept(BaseVisitor& guest)\
  { return AcceptImpl(*this, guest); }
```

- Using the library

```
class DocElement : public BaseVisitable<> {
public:
   DEFINE_VISITABLE()
};
class Paragraph : public DocElement {
public:
   DEFINE_VISITABLE()
};
```

```
class MyConcreteVisitor :
public BaseVisitor, // required
public Visitor<DocElement>,// visits DocElements
public Visitor<Paragraph> // visits Paragraphs {
  public:
    void Visit(DocElement&) {
       std::cout << "Visit(DocElement&) \n";
    }
    void Visit(Paragraph&) {
       std::cout << "Visit(Paragraph&) \n";
    }
};
```

```
int main() {
  MyConcreteVisitor visitor;
  Paragraph par;
  DocElement* d = &par;
                        // "hide" the static type of
                        // 'par' d->Accept(visitor);
}
```

This implementation is adequate for most situations. However, for applications where speed is a requirement, the `dynamic_cast` may be inhibitive.

And the Cyclic Visitor?

This is also referred to as the GoF Visitor

The bottom line is, we have a collection of types we want to visit. Therefore, pass a typelist to a Cyclic Visitor

```
// Forward declarations needed by the typelist
class DocElement;
class Paragraph;
class RasterBitmap;

// Visits DocElement, Paragraph and RasterBitmap
typedef CycleVisitor <
  void, // return type
  TYPELIST_3(DocElement, Paragraph, RasterBitmap)
> MyVisitor;
```

Follow a similar technique as was applied in Abstract Factory

```
template <typename R, class TList>
class CyclicVisitor : public GenScatterHierarchy<
    Tlist,
    Private::VisitorBinder<R>::Result> {
  typedef R ReturnType;
  template <class Visited>
  ReturnType Visit(Visited& host) {
    Visitor<Visisted>& subObj = *this;
    return subObj.Visit(host);
  }
};
```

- `CyclicVisitor` inherits a `Visitor<T>` for each type `T` in the typelist `TList`

The visitable hierarchy is set up by

```
typedef CycleVisitor <
  void, // return type
  TYPELIST_3(DocElement, Paragraph, RasterBitmap)
> MyVisitor;

class DocElement {
  public:
    virtual void Accept(MyVisitor&) = 0;
};

class Paragraph : public DocElement {
  public:
    DEFINE_CYCLIC_VISITABLE(MyVisitor);
};
```

Putting it all together?

Defining the catch-all function

- For the cyclic visitor, this is easy. In the typelist add the class `DocElement` at the head. This will result in `Visit(DocElement&)` to be defined which will be called when trying to visit an unknown type.
- With the acyclic visitor, instead of *visiting an unknown class by a known visitor*, a *unknown visitor visits a known class*. This is as a result of how the `BaseVisitable::AcceptImpl` is defined. It will return a default value of `Returntype()`. Require a policy to define the catch-all behaviour. `BaseVisitable` changes as follows:

```
template <
    typename R = void,
    template <typename, class>
        class CatchAll = DefaultCatchAll >
class BaseVisitable
```

```
{
... as above ...
template <class T>
static ReturnType AcceptImpl(T& visited,
       BaseVisitor& guest)  {
  if (Visitor<T>* p =
        dynamic_cast<Visitor<T>*>(&guest))
       {
          return p->Visit(visited);
      }
// Changed
return CatchAll<R, T>::OnUnknownVisitor(visited,
                                        guest);
} };
```

```
template <class R, class Visited>
struct DefaultCatchAll
{
static R OnUnknownVisitor(Visited&, BaseVisitor&)
{
// Here's the action that will be taken when
// there is a mismatch between a Visitor and
// a Visited. The stock behaviour is to return
// a default value
  return R();
} };
```