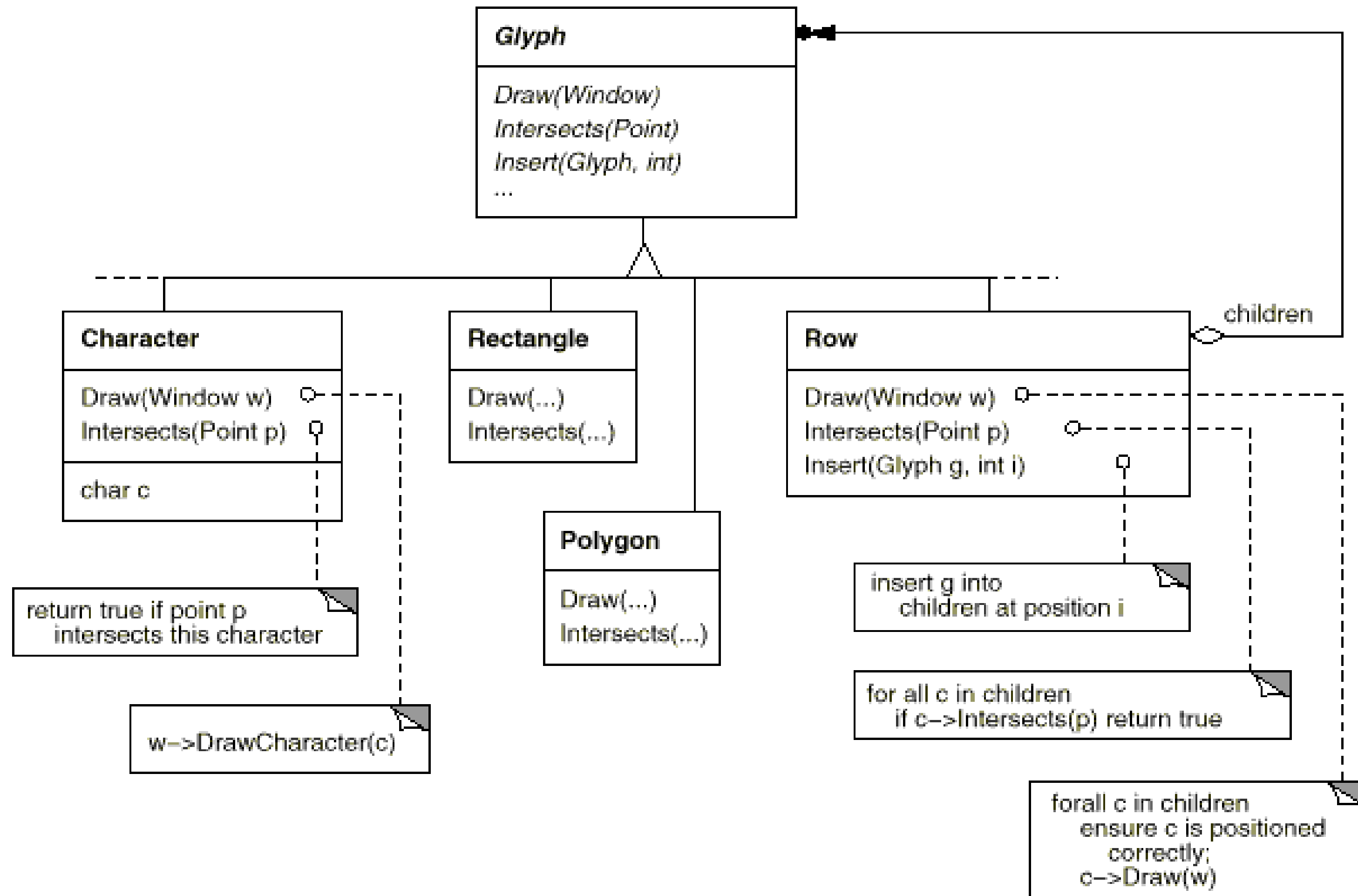
Design Issue #3

Spell Check and Hyphenation

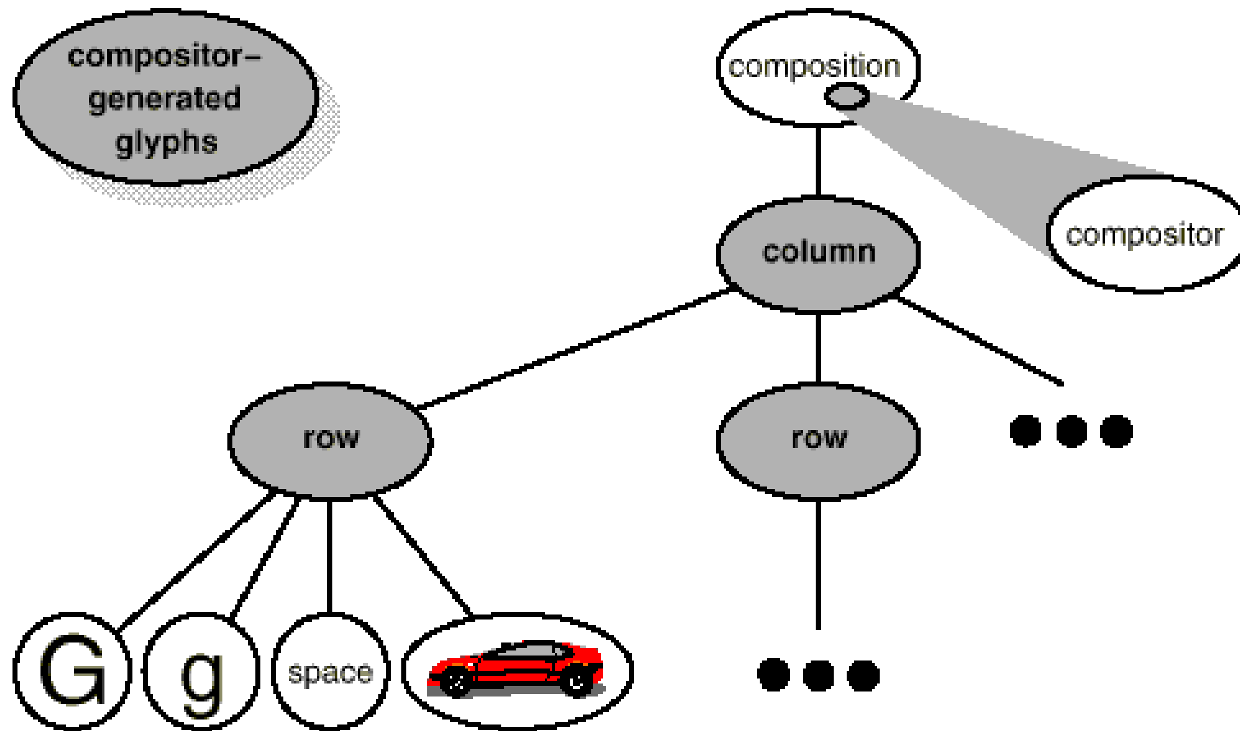
- Remember our (partial) class diagram



Design Issue #3

Spell Check and Hyphenation

- Remember our (partial) object structure



Design Issue #3

Spell Check and Hyphenation

- Similar constraints to formatting
- Need to support multiple algorithms
- We may want to add
  - search
  - grammar check
  - word count
  - Thesaurus
  - Text to speech
- This is **too much** for any single pattern…
- There are actually two parts
  - (1) Access the information
  - (2) Do the analysis

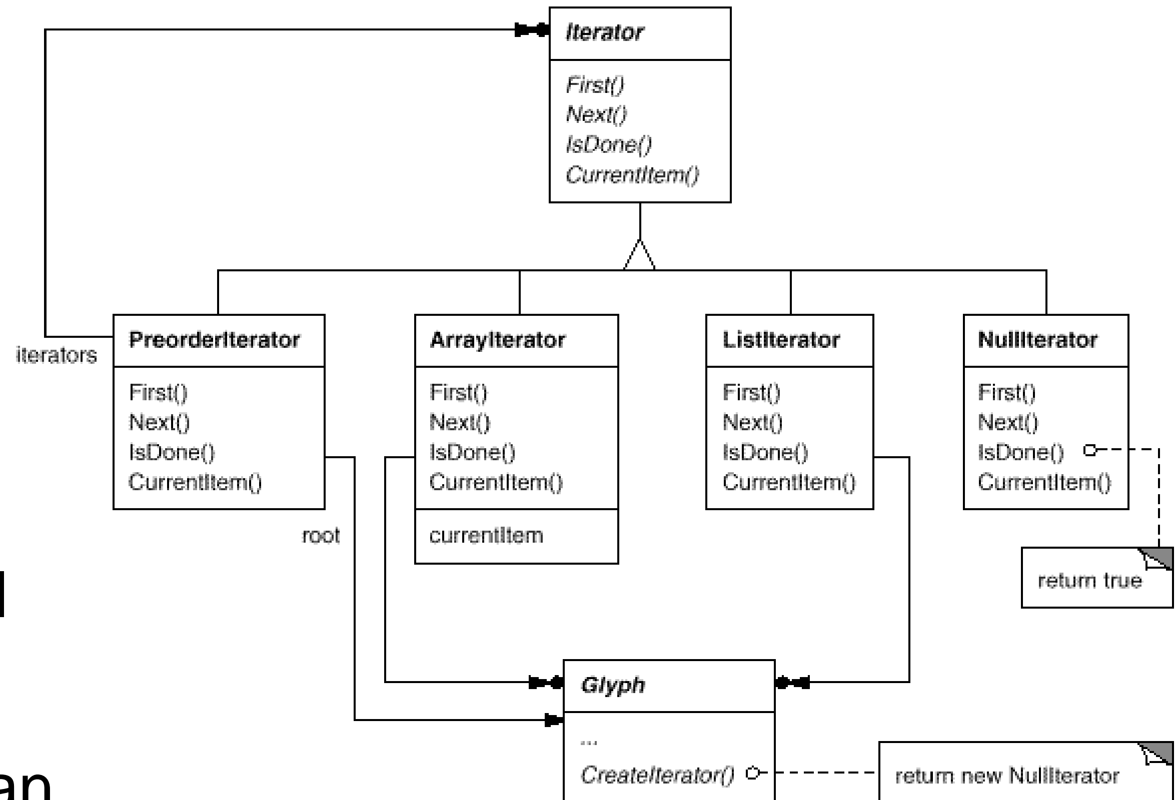
Design Issue #3

Spell Check and Hyphenation – Accessing the Information

- We can encapsulate access and traversal using the **Iterator** pattern

Reasons for separation:

- Don't 'pollute' Gyph interface with traversal operations
- Multiple traversal strategies
- Support more than one traversal at a time



Design Issue #3

Spell Check and Hyphenation – Accessing the Information

- Sample code illustrating the usage of an iterator to do our analysis

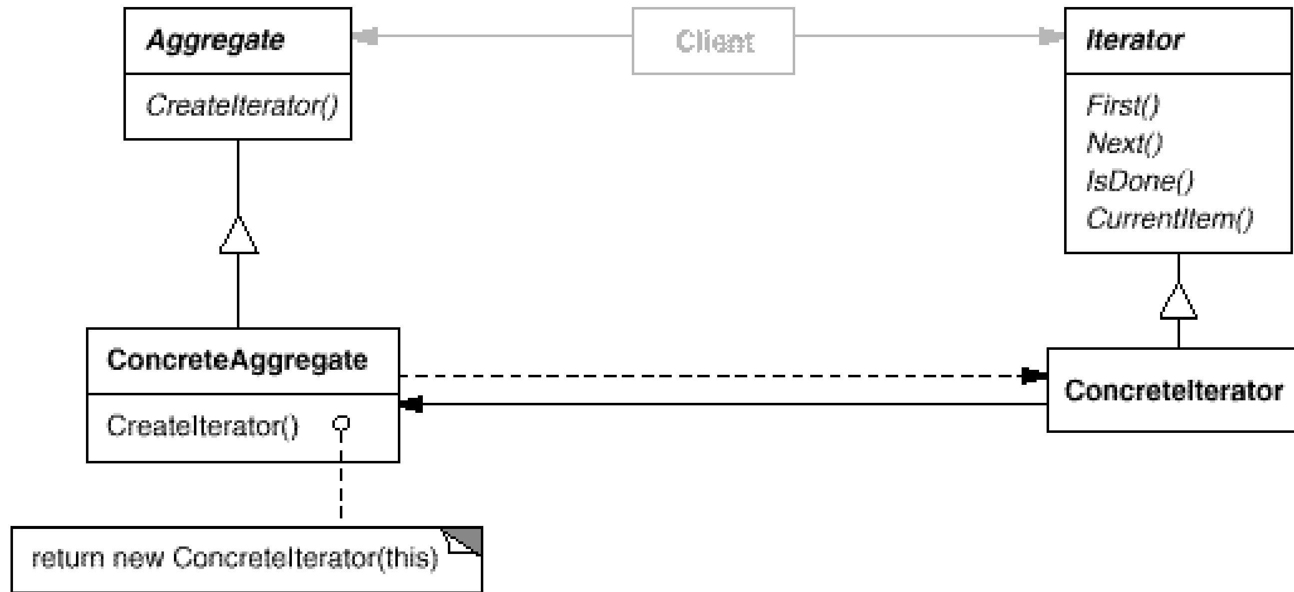
```
Glyph* root;
Iterator* i = root->createPreOrderIterator();
for (i->First(); !i->IsDone(); i->Next())
{
     Glyph* current = i->GetCurrent();
     // do some analysis
}
```

- Examples of iterators: an int* is an iterator for int[] type.

Design Issue #3

Spell Check and Hyphenation – Accessing the Information

- **Iterator** pattern



- Types of iterators
  - External iterator vs Internal iterator
  - Robust iterator
  - Null iterator
  - Polymorphic iterator

Design Issue #3

Spell Check and Hyphenation – The Analysis

- We don’t want our analysis in our iterator
  - Iterators can be reused
- We don’t want analysis in our Glyph class
  - Every time we add a new type of analysis… we have to change our glyph classes
- Therefore
  - Analysis gets its own class(es)
  - It will use the appropriate iterator
  - Analyzer class may need to accumulate data during analysis process

Design Issue #3

Spell Check and Hyphenation – The Analysis

Team exercise: Implement word counting

Design Issue #3

Spell Check and Hyphenation – The Analysis

```
class SpellingChecker
{
public:
   void Check(Glyph* glyph);
};

void SpellingChecker::Check (Glyph* glyph)
{
   Character* c;
   Row* r;
   Image* i;

  if (c = dynamic_cast<Character*>(glyph))
   {
         // analyze the character
   }
   else if (r = dynamic_cast<Row*>(glyph))
   {
         // prepare to analyze r's children
   }
   else if (i = dynamic_cast<Image*>(glyph))
   {
         // do nothing
   }
}
```

```
Glyph* root;
SpellingChecker checker;
Iterator* i = root->createPreOrderIterator();
for (i->First();!i->IsDone(); i->Next())
{
     Glyph* current = i->GetCurrent();
     checker.Check(current);
}
```

This is a start… but not what we want!

Design Issue #3

Spell Check and Hyphenation – The Analysis

- Why don’t we want this?
  - Difficult to extend: each time a new Glyph is introduced, one needs to change SpellingCheck::Check
  - Error prone: missing one type of Glyph
  - Violates OCP and SRP principles
  - Usually, the usage of `dynamic_cast` denotes poor OO modeling
- We want a better solution…

Design Issue #3

Spell Check and Hyphenation – The Analysis

- …we will use the **Visitor** pattern

```
class Visitor
{
  public:
          virtual void visitCharacter(Character*) { }
          virtual void visitRow(Row*) { }
          virtual void visitImage(Image*) { }
          // ... and so forth
};
```

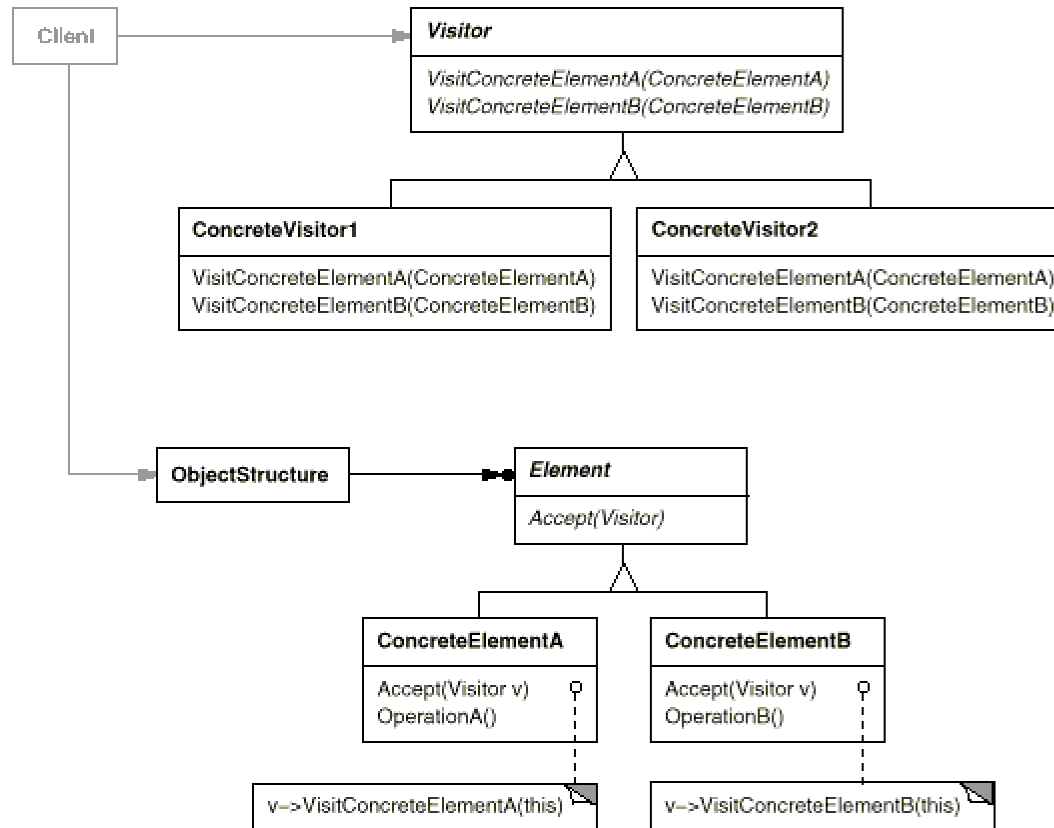
- Then, we specialize this superclass into
  - SpellCheckingVisitor
  - HyphenationVisitor
  - and so on…

- There is a little impact on Glyph hierarchy that need to be changed in order to accept visitors:
  - within Glyph we define an abstract operation

    `void accept(Visitor& visitor)`
  - Character class implements it by calling `visitor.visitCharacter(this)`
  - Row class implements it by calling `visitor.visitRow(this)`

Design Issue #3

Spell Check and Hyphenation – The Analysis



Design Issue #3

Spell Check and Hyphenation – The Analysis

Team exercise: Re-implement word counting using Visitor pattern

Design Issue #3

Spell Check and Hyphenation – The Analysis

```
Class Glyph {
     // other declarations. . .
     virtual void accept(Visitor* ) = 0;
};

class Character : public Glyph {
     // other declarations. . .
     void accept(Visitor* v) {
          v->visitCharacter(this);
     }
};

class Row : public Glyph {
     // other declarations. . .
     void accept(Visitor* v) {
          v->visitRow(this);
     }
};

class Image : public Glyph {
     // other declarations. . .
     void accept(Visitor* v) {
          v->visitImage(this);
     }
};
```

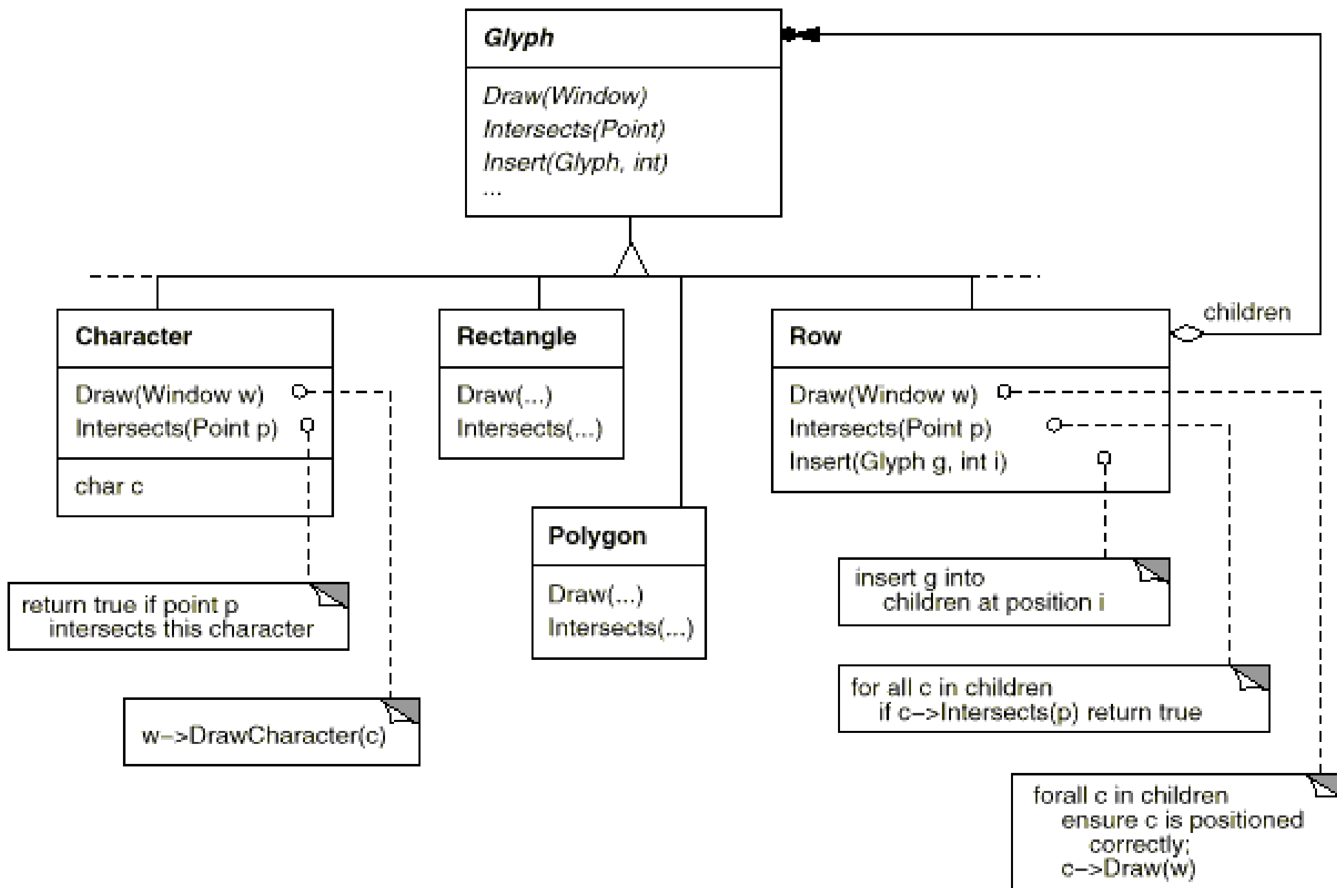
```
Glyph* root;
SpellCheckerVisitor checker;
Iterator* i = root->createPreOrderIterator();
for (i->First();!i->IsDone(); i->Next())
{
      Glyph* current = i->GetCurrent();
      current->accept(&checker);
}
```

```
class Visitor {
public:
     virtual void visitCharacter(Character*) {
     }
     virtual void visitRow(Row*) {
     }
     virtual void visitImage(Image*) {
     }
};

class SpellCheckerVisitor : public Visitor{
public:
     virtual void visitCharacter(Character*) {
           // analyze the character
     }

      virtual void visitRow(Row*) {
           // prepare to analyze r's children
      }
 };
```

This is a what we need!

Design Issue #4
Document persistence

- How to implement Open / Save document operations so that:



1. New formats (PDF, XML, RTF etc.) are easily supported
2. Decoupling from document structure

Design Issue #4
Document persistence

- Need different approaches for Open vs. Save because the context is completely different:
  - Save: the document fully exist in memory and we need to implement an operation that process it (similar to Spell Check, or Hyphenation)
  - Open: the document DOES NOT exist in memory and need to be constructed from a stream of bytes

Design Issue #4
Document persistence

Team exercise: Design Save and Open operations

Design Issue #4
Document persistence : Save

- Key patterns (similar to Spell Check and alike) are:
  - Iterator
  - Visitor

```
class Visitor {
public:
     virtual void visitCharacter(Character*) {
     }
     virtual void visitRow(Row*) {
     }
     virtual void visitImage(Image*) {
     }
};

class XMLSaveVisitor : public Visitor{
     File* fileXML;
public:
     XMLSaveVisitor(const char* fileName);
     virtual void visitCharacter(Character*);
     virtual void visitRow(Row*);
     virtual void visitImage(Image*)
     virtual ~XMLSaveVisitor();
 };
```

```
XMLSaveVisitor::XMLSaveVisitor(const char* filename)
{
     fileXML = new File(filename, “rw”);
}

Void XMLSaveVisitor::visitCharacter(Character* c)
{
     fileXML->write(“<char>”+c+”</char>”);
}

Void XMLSaveVisitor::visitImage(Image* img)
{
     fileXML->write(“<image width=”+img->getWidth()+
                    “ height=“+img->getHeight());
     fileXML->write(“ url=“+img->getURL());
     fileXML->write(”></image>”);
}
```

Design Issue #4
Document persistence : Open

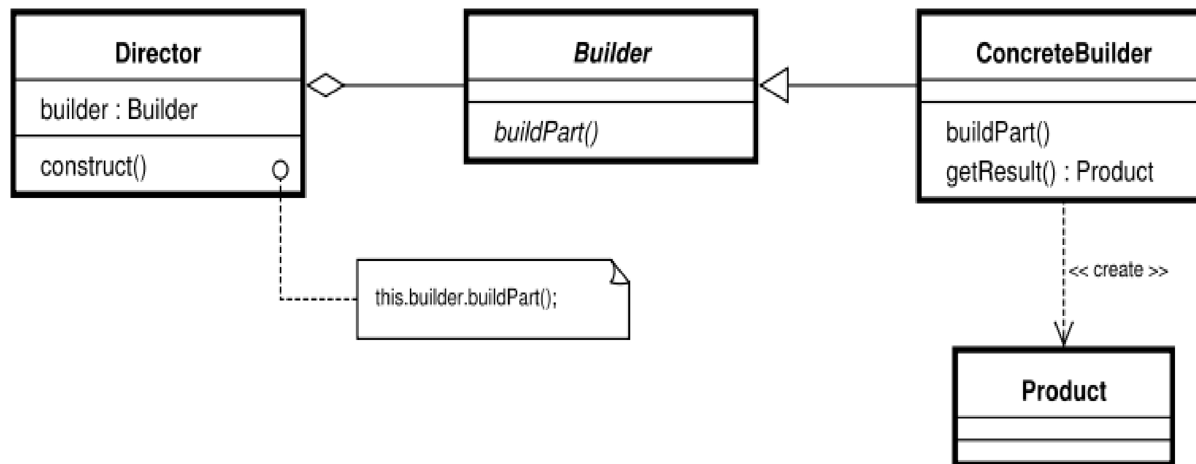
- It's different (no object yet), what we need is:
  - To be able to build step-by-step a complex structure of objects;
  - The process of constructing the object(s) should be a generic one, easy to adapt to different source streams

Design Issue #4
Document persistence : Open

- It’s different (no object yet), what we need is:
  - To be able to build step-by-step a complex structure of objects;
  - The process of constructing the object(s) should be a generic one, easy to adapt to different source streams
- This is exactly what **BUILDER** pattern does!

Design Issue #4
Document persistence : Builder

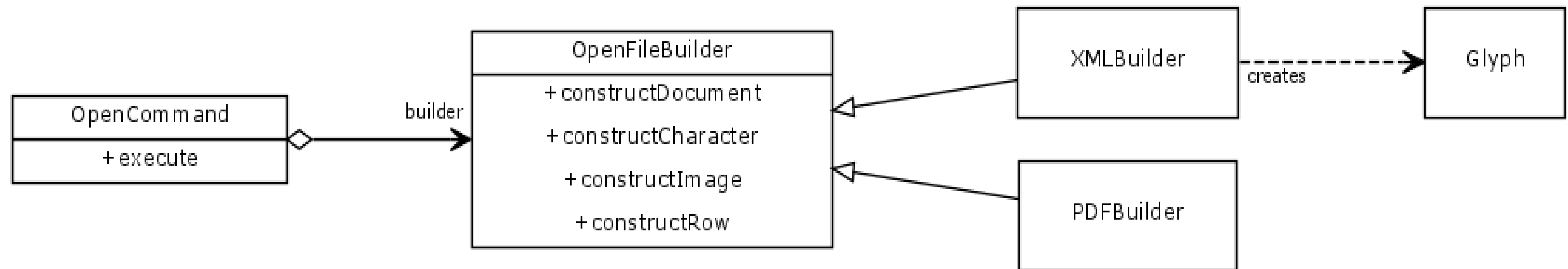
- Applicability of Builder pattern:
  - Decoupling the algorithm to create a complex object from its parts and the relationships between them
  - The building process must allow various representations for the object under construction
  - Avoid proliferation of constructors; enhance code clarity



Design Issue #4
Document persistence : Open

- Builder gives us a hand:
  - OpenCommand class acts as Director
  - The “products” are represented by the different Glyph types (Image, Character, Row etc)
  - OpenFileBuilder is the Builder interface in charge with handling different types of glyphs; it has a constructXXX member function for each Glyph type
  - For each import format supported (such as XML, PDF, RTF etc.) a new ConcreteBuilder is implemented

Design Issue #4
Document persistence : Open




```
// Paste this in yUML.me
[OpenCommand|+execute]<>-builder->[OpenFileBuilder]
[OpenFileBuilder|+constructDocument;+constructCharacter;+constructImage;+constructRow]
[OpenFileBuilder]^[XMLBuilder]
[OpenFileBuilder]^[PDFBuilder]
[XMLBuilder]creates -.->[Glyph]
```

Design Issue #4
Document persistence : Open

```
Void OpenCommand::execute() {
     // pop-up a window to select the file
     // if no file selected, or format incompatible,
     // exit

     OpenFileBuilder* builder =
          OpenFileBuilderFactory.getBuilder(filename);
     Glyph* document = builder->constructDocument();

     // register document in the opened documents
     // collection
}

XMLBuilder::XMLBuilder(const char* filename) {
     XMLFile* file = new XMLFile(filename, “r”);
}
```

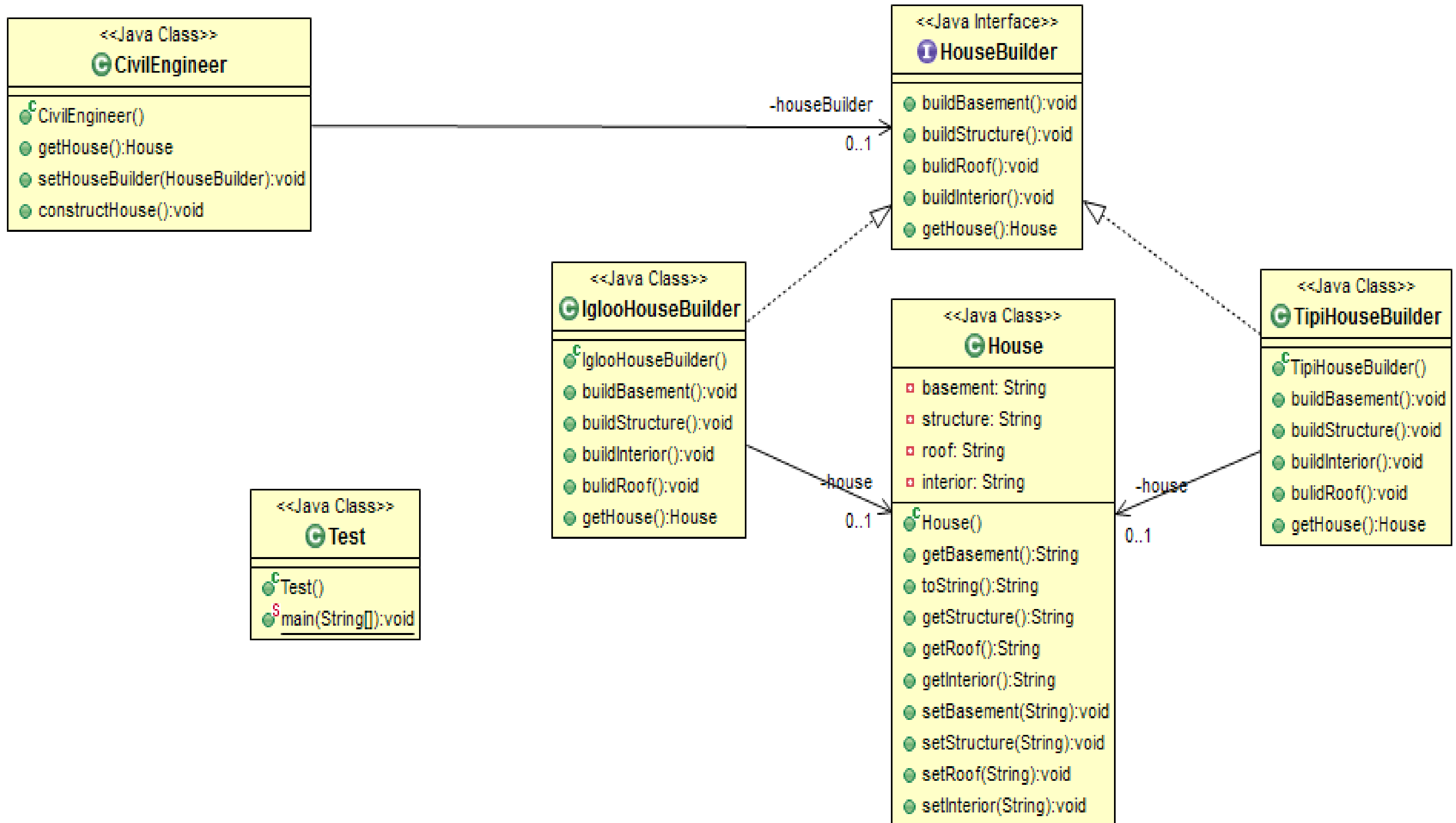
What other pattern is used in this code snippet?

```
Glyph* XMLBuilder::constructDocument() {
     XMLElement e;
     Glyph* g = NULL, doc = NULL;

     file->first();
     while(!file->isDone()) {
        e = file->currentItem();
        switch(e.getType()) {
              case DOCUMENT:
              doc = new Document();
              break;
           case IMAGE:
              g = constructImage(e);
              break;
           case CHARACTER:
              g = contructCharacter(e);
              break;
           // etc.
        }
        if(g!=NULL && doc!=NULL)
              doc->insertGlyph(g);
        file->next();
     }
}


Image* XMLBuilder::constructImage(XMLElement& e) {
     Image* img = new Image(e.getAttribute(“width”),
                               e.getAttribute(“height”));
    img->loadFile(e.getAttribute(“url”));
     return img;
}
```

Design Issue #4

Document persistence : Builder

Examples: let’s build a house…



Design Issue #4
Document persistence : Builder

- Builder may use other creational patterns to build its parts;

- It is common that builders itself are implemented as Singleton

- While Builder constructs a single object in a step-by-step process, the Abstract Factory constructs a family of related objects

- The complex object built using Builder is often represented using Composite design pattern.

Design Issue #4

Document persistence : Builder

- Allows the variation of internal representation of a product
- Decouples the code for representing the object and the code to construct it
- Employs good control and customization of
- Easy to add new ConcreteBuilders.