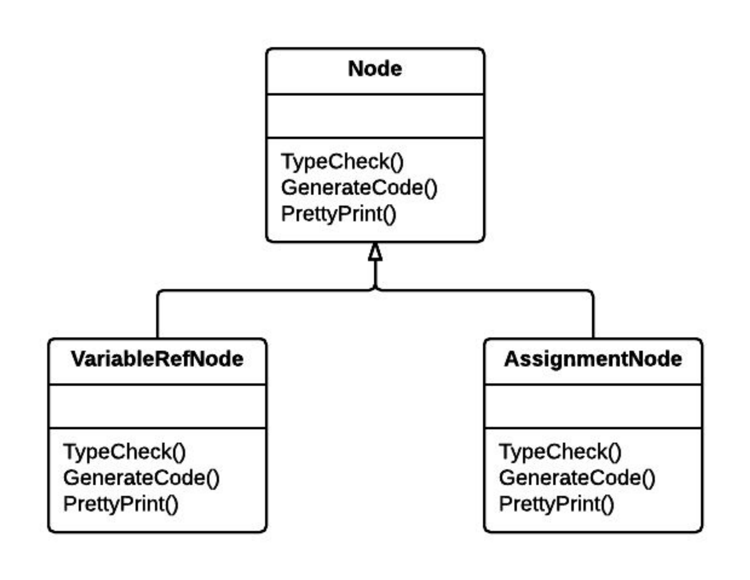
Visitor Pattern

Behavioral, Object focused

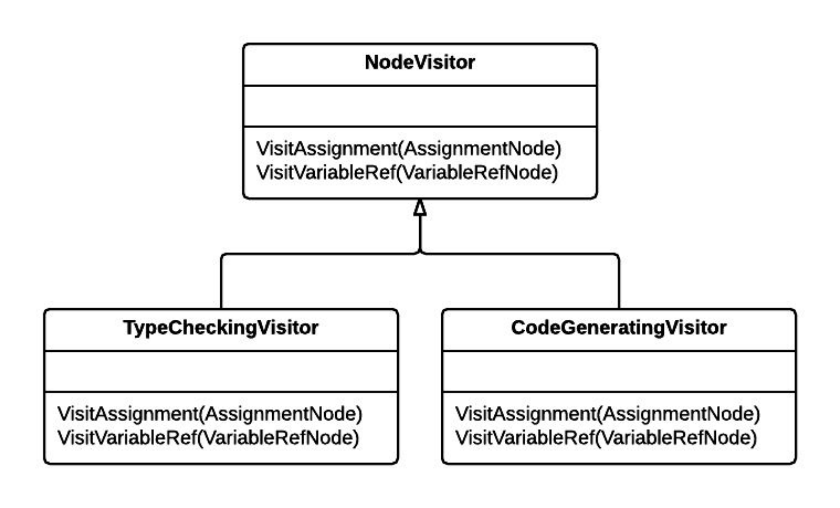
Motivation

- Consider a compiler representing a program as an abstract syntax tree (AST)
- The compiler must perform operations on the AST for “semantic” analysis
  - Type-checking
  - Code optimization
  - Flow analysis
  - Etc.
- This requires a lot of nodes with a lot of operations
- Maintaining this code base quickly becomes a nightmare

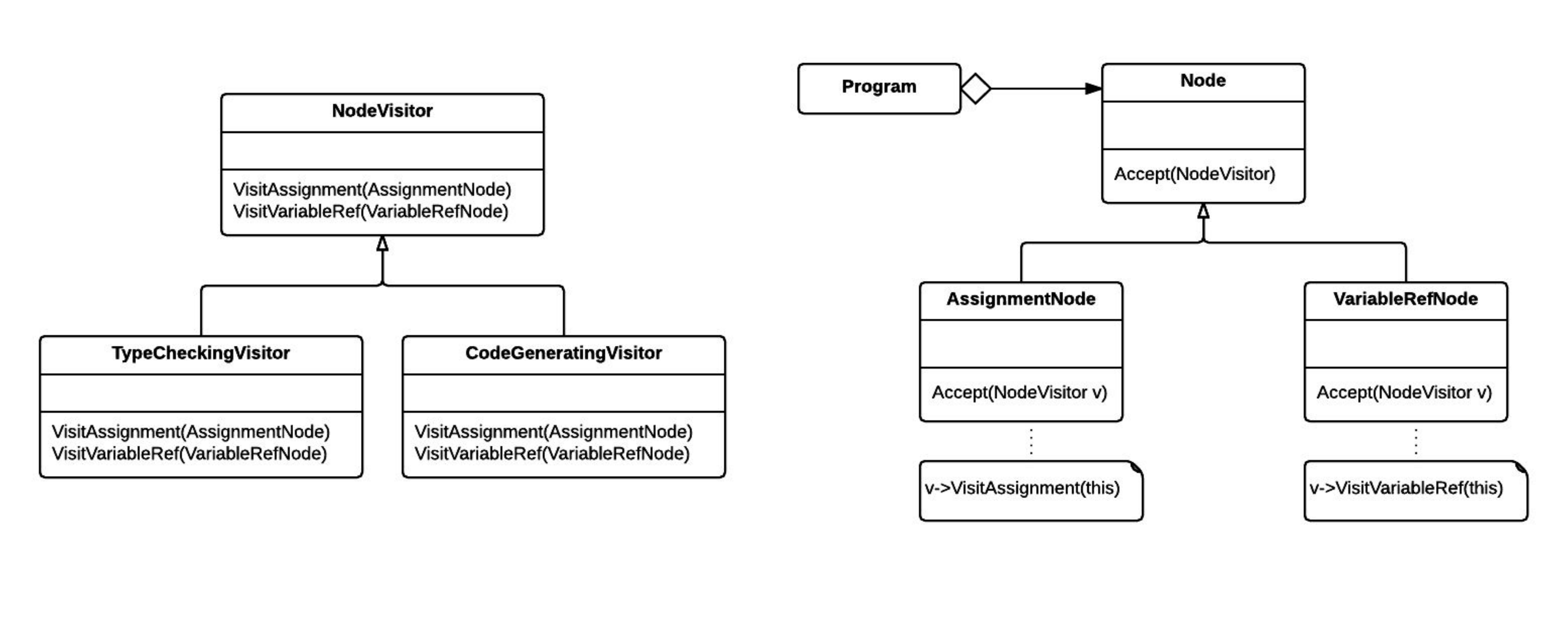


Motivation

- It would be better if new operations could be added separately and node classes were independent of the operations applied to them
- This is obtained by packaging related operations in a separate class called **Visitor**
- These **Visitor** objects are passed to elements of the AST as it is traversed


NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

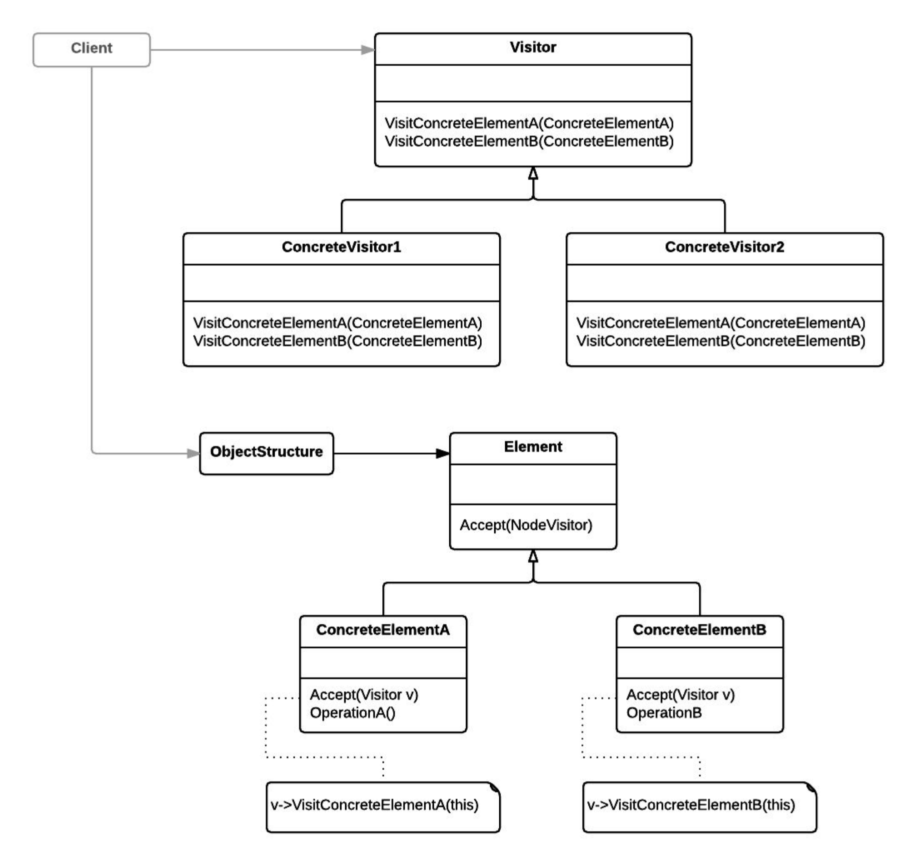
Motivation



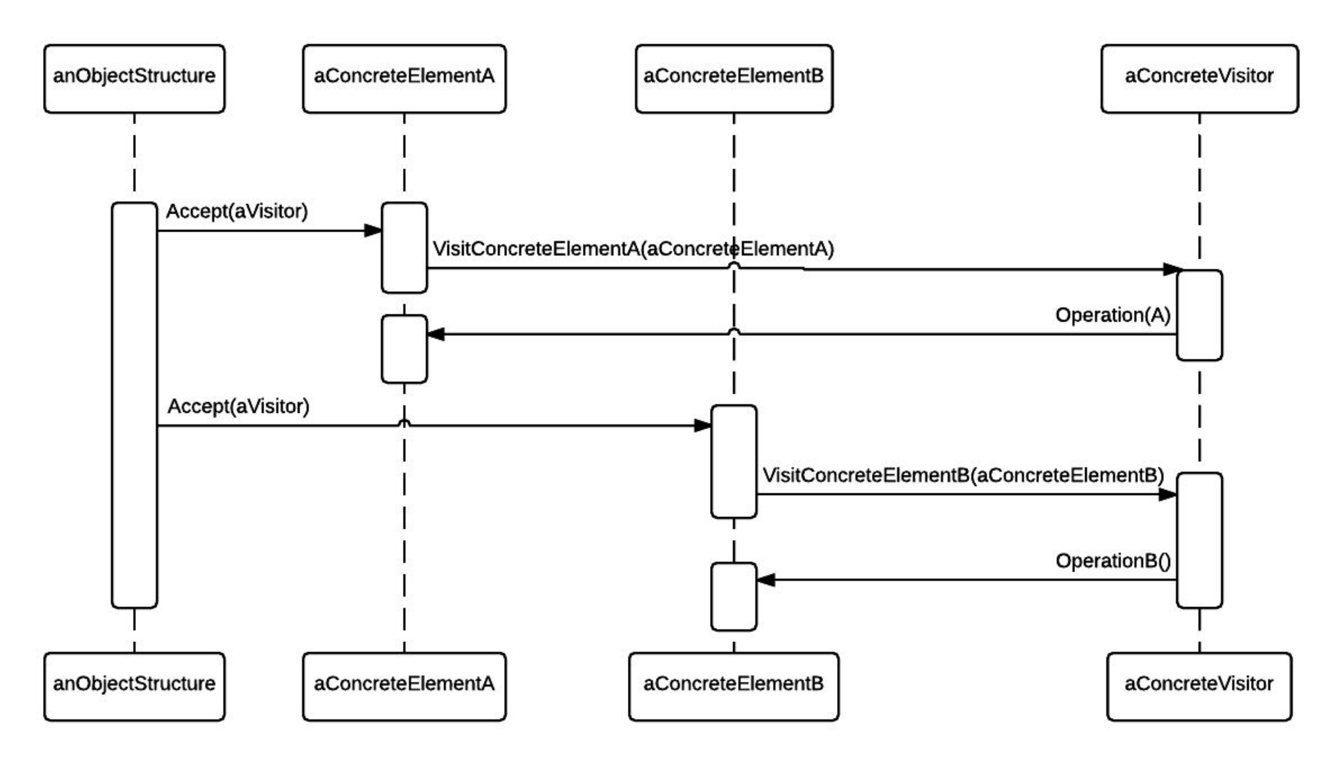
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
VariableRefNode
Accept(NodeVisitor v)
v->VisitVariableRef(this)

Structure

- **Visitor -** Declares a Visit operation for each class of ConcreteElement in the object structure
- **ConcreteVisitor -** Implements each operation declared by Visitor
- **Element -** Defines an `Accept()` operation with visitor as an argument
- **ConcreteElement -** Implements the `Accept()` operation
- **ObjectStructure -**
  - Can enumerate its elements
  - Can be a composite or collection
  - May provide high level interface


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(NodeVisitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Structure


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
Operation(A)
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()
anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor

Consequences

- **Pros:**
  - Adding new operations is easy
  - Related operations are gathered together, unrelated operations are separated
  - Visitors can visit across different class hierarchies

    ```
    Class Visitor {
         Public:
              Void VisitMyType(MyType*);
              Void VisitYourType(YourType*);
              ...
    ```

  - Visitors can accumulate state while visiting each element
- **Cons:**
  - Adding new `ConcreteElement` classes is difficult
  - Visitors assume the `ConcreteElement` interface lets them do their job, which typically breaks encapsulation

Example from “Design Patterns: Elements of Reusable Object-Oriented Software”

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Traversal vs. Traversal Actions

- The iterator provided a method of traversing the glyph structure **BUT**, it did not provide a way to check spelling and do the hyphenation

Where do we put the responsibility for analysis?

Traversal vs. Traversal Actions

- Putting the responsibility for analysis in the Iterator makes analysis an integral part of traversal, which it isn’t necessarily a part of
- We want to **distinguish** between the traversal and the analysis being performed
  - This provides more flexibility and potential for re-use

So, Where do we put the responsibility for analysis?

Analysis

- A given analysis must be able to distinguish between different kinds of glyphs
    - Spelling and hyphenation only care about visible glyphs
- We **don’t** want to put the analytical capabilities into the glyph classes themselves
- Each analysis could add one or more abstract operations, and each subclass would have to implement them with the role they play

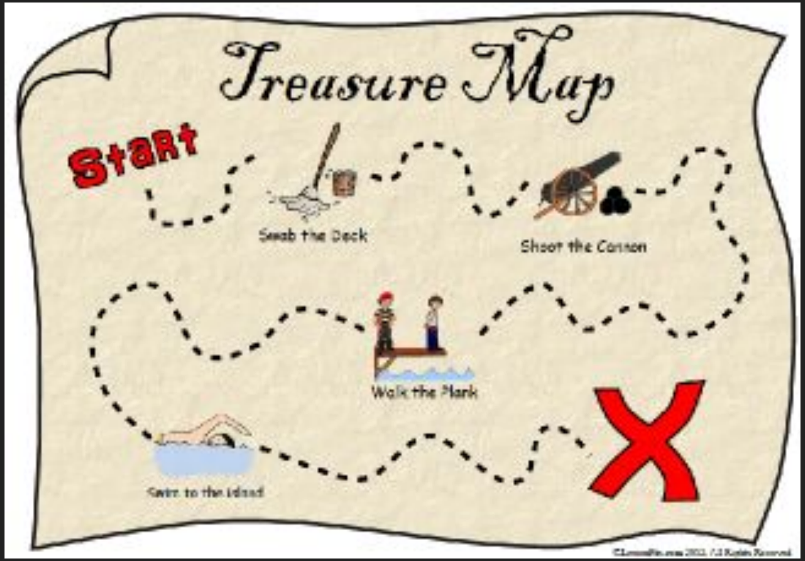
Analysis

- We then have to change every glyph class whenever we add a new analysis, even if it isn’t affected!
- This can be partially fixed by providing a default implementation in the base class, however we still need to expand the Glyph interface with each new analytical capability

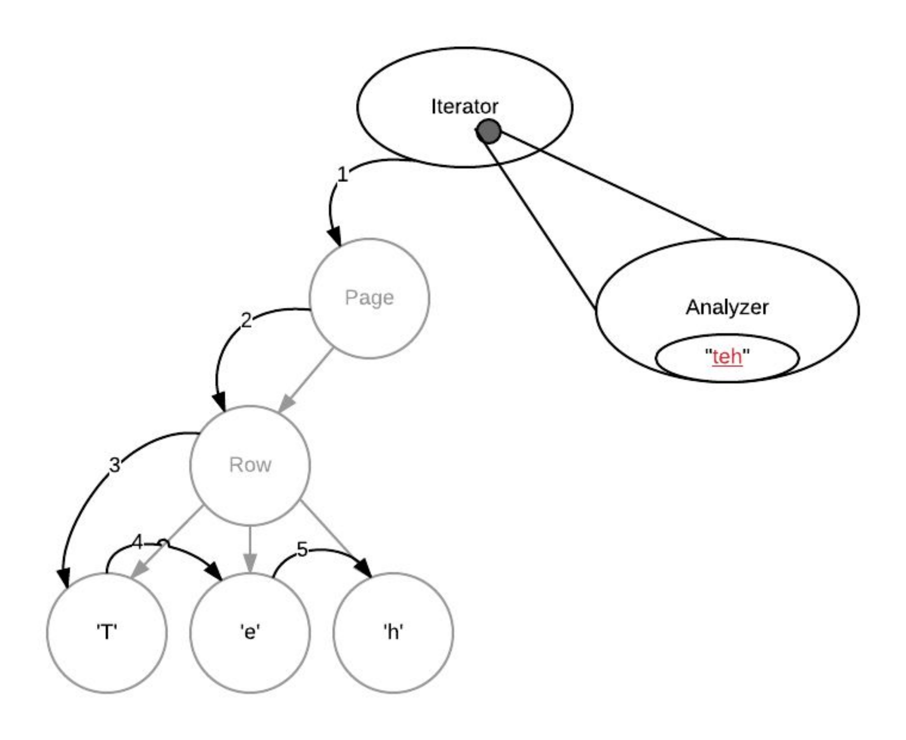
Encapsulating the Analysis

- The analysis should be encapsulated in a separate object
- This analysis object can interact with the appropriate *iterator* to perform its analysis
- The *iterator* object “carries” the glyph instance associated with it
- The **analysis** object is able to visit this instance and perform a piece of the analysis
- The **analysis** object performs a piece of the analysis at every step of the traversal, accumulating the entire **analysis**

- Pick up a “clue” **visiting** each step of the **iterators** traversal, piecing together a map to the treasure, or final **analysis**



Spell Check Example



How do we do this?

- How does the analysis object distinguish different types of glyphs?
  - If/Else trees are ugly and nobody wants to deal with them
  - Switch statements aren’t any better
  - Type checking would need to be updated every time a change was made to the glyph class hierarchy
  - Type checking is not effective in C++ and many other languages

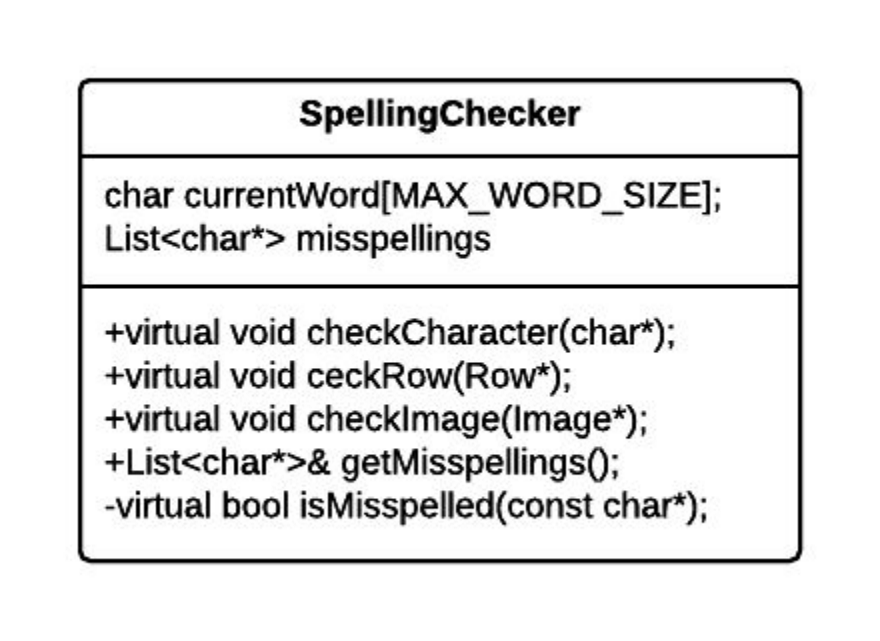
How do we do this?

- Consider the following abstract operation added to the Glyph class:
  Void `CheckMe(SpellingChecker&)`
- Every `Glyph` subclass would define `CheckMe`

```
Void GlyphSubclass::CheckMe(SpellingChecker& checker) {
    checker.CheckGlyphSubclass(this);
}
```

- Here `GlyphSubclass` would be replaced by the name of each glyph subclass
- When CheckMe is called, the specific subclass is now known

How do we do this?

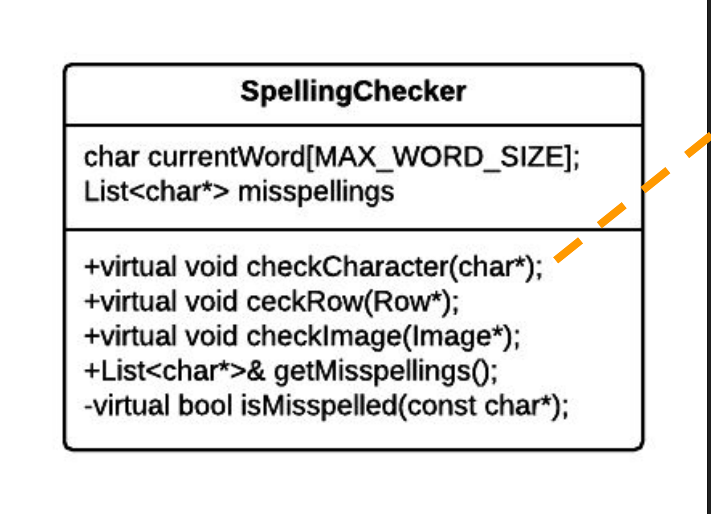


SpellingChecker

```
char currentWord[MAX_WORD_SIZE];
List<char*> misspellings
```

```
+virtual void checkCharacter(char*);
+virtual void ceckRow(Row*);
+virtual void checkImage(Image*);
+List<char*>& getMisspellings();
-virtual bool isMisspelled(const char*);
```

How do we do this?



SpellingChecker

char currentWord[MAX_WORD_SIZE];
List<char*> misspellings

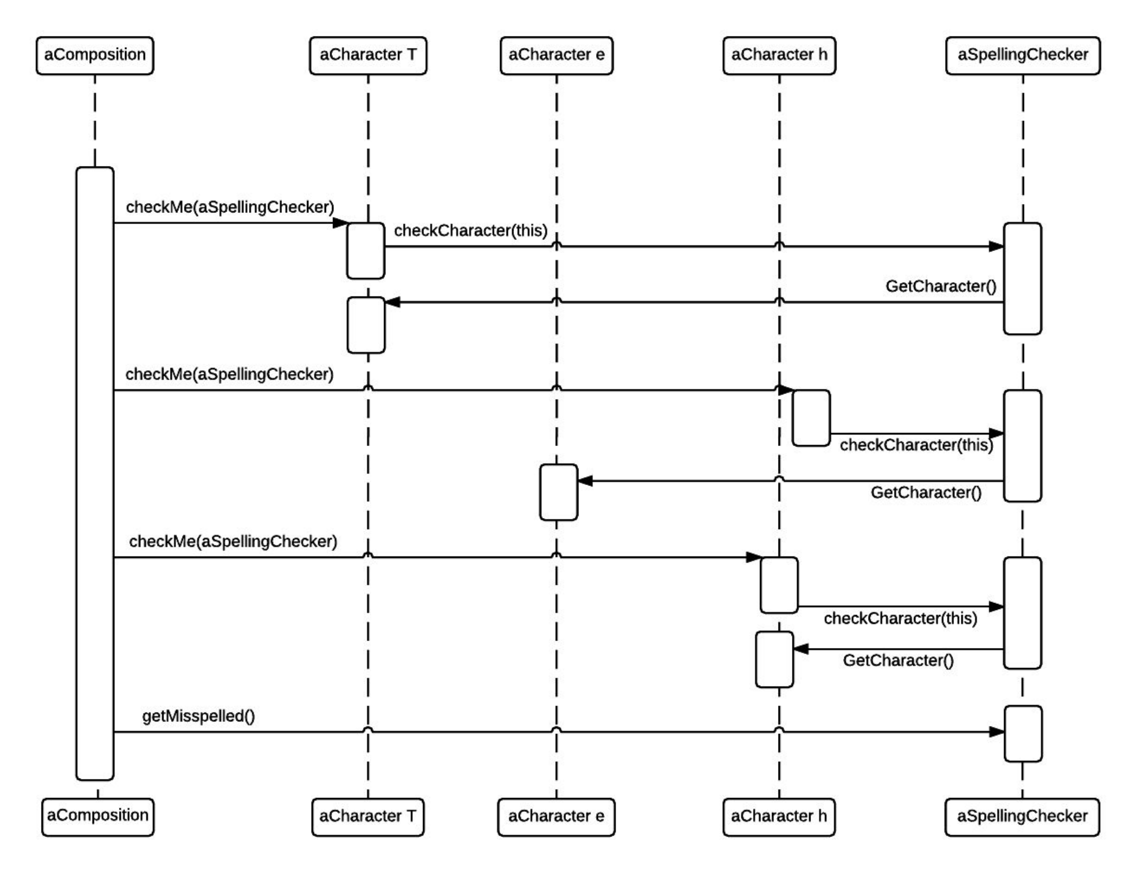
+virtual void checkCharacter(char*);
+virtual void ceckRow(Row*);
+virtual void checkImage(Image*);
+List<char*>& getMisspellings();
-virtual bool isMisspelled(const char*);

```
Void SpellingChecker::checkCharacter(char* c) {
     if(isalpha(c)) {
          //append to this->currentWord
     } else {
          // We hit a non-alphabetic character
          If (isMisspelled(this->currentWord) {
               // add to this->misspellings
          }
          // Reset the current word
          this->currentWord[0] = ‘\0’;
     }
}
```

How do we do this?

- Now we can traverse the glyph structure calling checkMe on each glyph with the SpellingChecker as an argument
- This identifies each glyph to the SpellingChecker and prompts the checker to do the next increment in the spell check

```
SpellingChecker* spellCheck;
Compotition* c;
// …
PreorderIterator iter(c);
For (iter.First(); !iter.is_done(); iter.Next()) {
    Glyph* g = iter.Current();
    g->checkMe(spellCheck);
}
```

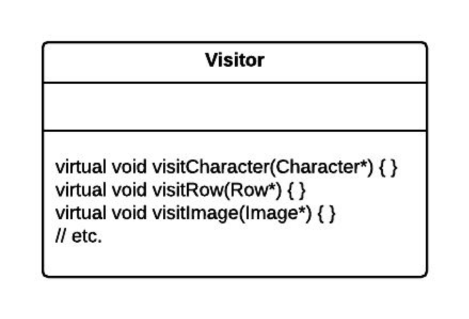

aComposition
aCharacter T
aCharacter e
aCharacter h
aSpellingChecker
checkMe(aSpellingChecker)
checkCharacter(this)
GetCharacter()
checkMe(aSpellingChecker)
checkCharacter(this)
GetCharacter()
checkMe(aSpellingChecker)
checkCharacter(this)
GetCharacter()
getMisspelled()
aComposition
aCharacter T
aCharacter e
aCharacter h
aSpellingChecker

Supporting Multiple Analyses

- We can now find spelling errors using this approach
- However, this instance is specific to spelling error analysis (why?)
- By converting it to a more generic visitor pattern, we can extend the system to allow for different types of analysis easily
- The visitor interface `accept(Visitor*)` replaces the analysis-dependent operations like `checkMe(SpellingChecker&)`
- Specific visitor functions like `checkCharacter(char*)` are replaced by functions relating to a specific class `visitCharacter(Character*)`
- Different analyses can now inherit from this base class

Visitor Class

```
class visitor {
    public:
        virtual void visitCharacter(Character*) { }
        virtual void visitRow(Row*) { }
        virtual void visitImage(Image*) { }
        // Visit functions for all Glyph classes
};
```

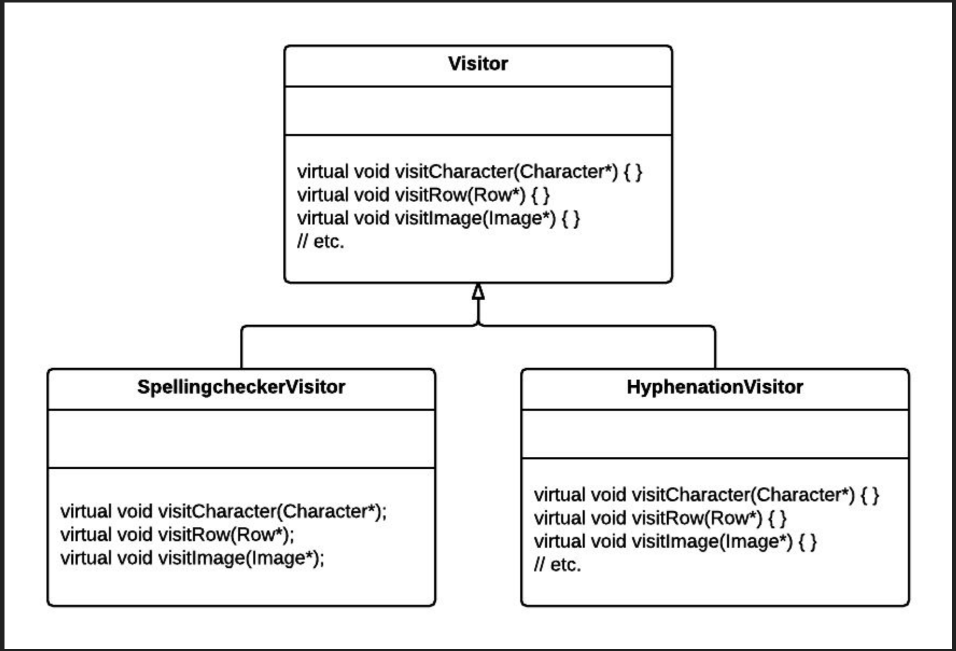


Visitor

virtual void visitCharacter(Character*) { }
virtual void visitRow(Row*) { }
virtual void visitImage(Image*) { }
// etc.

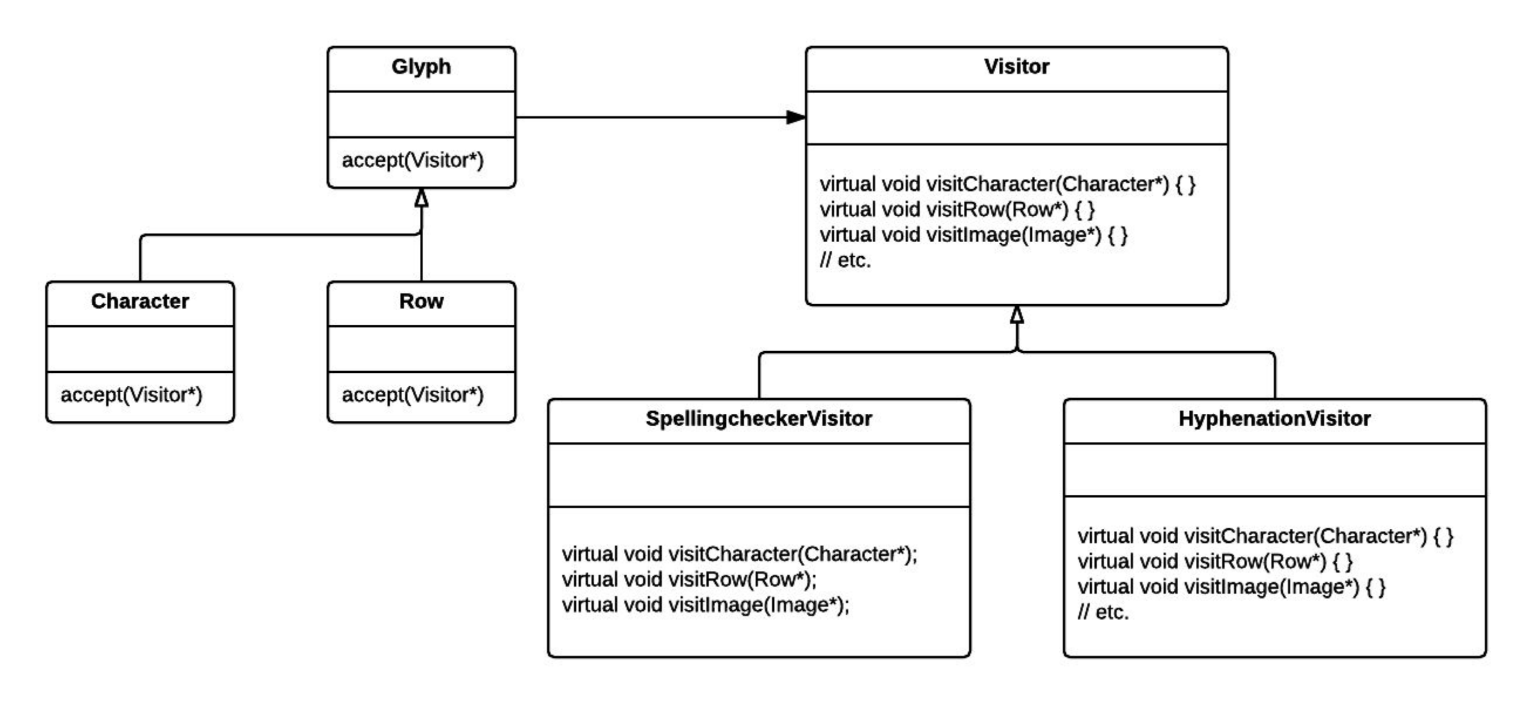
Visitor class and Concrete Subclasses

- `SpellingCheckerVisitor` is implemented the same way as before except `checkCharacter` is replaced with `visitCharacter`, etc.



Visitor
virtual void visitCharacter(Character*) { }
virtual void visitRow(Row*) { }
virtual void visitImage(Image*) { }
// etc.
SpellingcheckerVisitor
virtual void visitCharacter(Character*);
virtual void visitRow(Row*);
virtual void visitImage(Image*);
HyphenationVisitor
virtual void visitCharacter(Character*) { }
virtual void visitRow(Row*) { }
virtual void visitImage(Image*) { }
// etc.

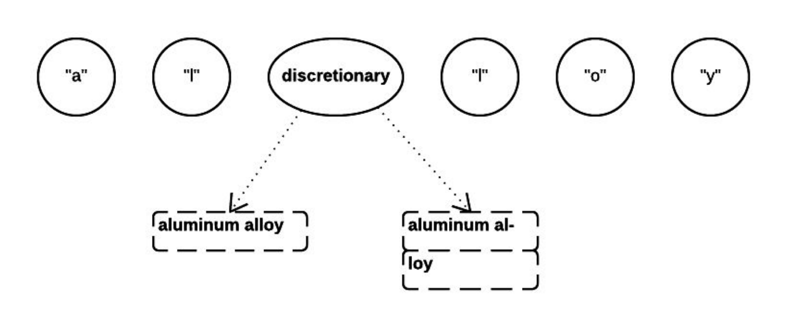
Visitor Class and Concrete Subclasses


Glyph
accept(Visitor*)
Character
accept(Visitor*)
Row
accept(Visitor*)
Visitor
virtual void visitCharacter(Character*) { }
virtual void visitRow(Row*) { }
virtual void visitImage(Image*) { }
// etc.
SpellingcheckerVisitor
virtual void visitCharacter(Character*);
virtual void visitRow(Row*);
virtual void visitImage(Image*);
HyphenationVisitor
virtual void visitCharacter(Character*) { }
virtual void visitRow(Row*) { }
virtual void visitImage(Image*) { }
// etc.

How would we design a hyphenation class as a visitor subclass?

HyphenationVisitor

- Traverse the text to accumulate characters into words
- Once an entire word has been assembled, apply an algorithm to determine potential hyphenation points
- At each hyphenation point, insert a **discretionary** glyph into the composition
- A discretionary glyph has two possible appearances, depending on whether or not it is the last character on a line, a hyphen or no appearance



Visitor Pattern Review

- The **Visitor** class and its subclasses allows an open-ended number of analyses on glyph structures without having to edit the glyph classes themselves
- Visitors can be applied to *any* object structure, not just composite structures
  - The object structure should have a traversal object (**iterator**)
- The visitor pattern is most suitable when you want to be able to do a variety of different things to objects with a stable class structure
- Adding a new kind of visitor requires no change to that class structure
- Adding a subclass to the structure requires an update to **<u>all</u>** visitor interfaces to include a `Visit` operation for that subclass