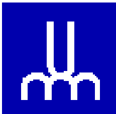
# The Visitor design pattern


Yann-Gaël Guéhéneuc

Assistant professor

guehene@iro.umontreal.ca, room 2345


Département d’informatique et de recherche opérationnelle

Université de Montréal



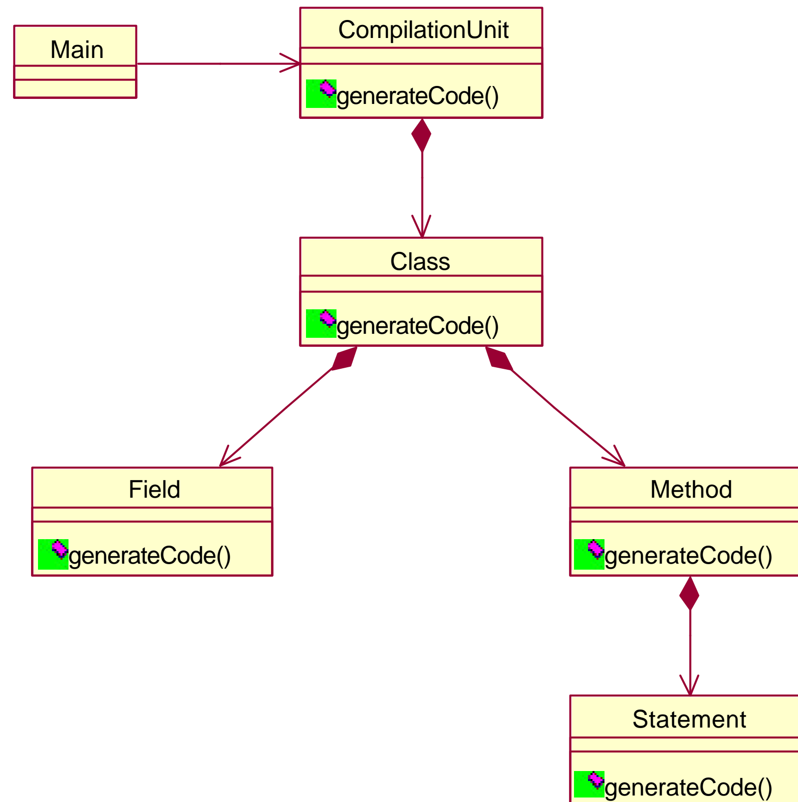
© Yann-Gaël Guéhéneuc 2003

# Running example (1/5)

- Compiler
  - Parse files to build an AST
  - Iterate over the AST
    - Bind types
    - Generate code
    - …

# Running example (2/5)

- AST



# Running example (3/5)

```
package compiler;

import java.util.Set;

public class Method {
    private Set statements;

    public void addStatement(final Statement aStatement) {
        this.statements.add(aStatement);
    }
    public void removeStatement(final Statement aStatement) {
        this.statements.remove(aStatement);
    }
}
```

```
package compiler;

public class Field {
    /* To be implemented. */
}
```

```
package compiler;

public class Statement {
    /* To be implemented. */
}
```

# Running example (4/5)

```
package compiler;

import java.util.Set;

public class Class {
    private String name;
    private Set methods;
    private Set fields;

    public String getName() {
        return this.name;
    }
    public void addMethod(final Method aMethod) {
        this.methods.add(aMethod);
    }
    public void removeMethod(final Method aMethod) {
        this.methods.remove(aMethod);
    }
    public void addField(final Method aField) {
        this.fields.add(aField);
    }
    public void removeField(final Field aField) {
        this.fields.remove(aField);
    }
}
```

# Running example (5/5)

```
package compiler;

import java.util.Iterator;
import java.util.Set;

public class CompilationUnit {
    private Set classes;

    public void addClass(final Class aClass) {
        this.classes.add(aClass);
    }
    public void removeClass(final Class aClass) {
        this.classes.remove(aClass);
    }
    public Class getClass(final String aName) {
        final Iterator iterator = this.classes.iterator();
        while (iterator.hasNext()) {
            final Class aClass = (Class) iterator.next();
            if (aClass.getName().equals(aName)) {
                return aClass;
            }
        }
        return null;
    }
}
```

# Naïve implementation (1/7)

- How to generate microcode for
  - Microsoft Windows operating system
  - Intel Pentium processor

# Naïve implementation (1/7)

- How to generate microcode for
  - Microsoft Windows operating system
  - Intel Pentium processor

Add a `generateCode()` method in each class

# Naïve implementation (2/7)

```java
public class Method {
          …
    public String generateCode() {
        String generatedCode = "";
        /* Do something at the beginning. */
        final Iterator iterator = this.statements.iterator();
        while (iterator.hasNext()) {
            final Statement aStatement = (Statement) iterator.next();
            generatedCode += aStatement.generateCode();
        }
        /* Do something at the end. */
        return generatedCode;
    }
}
```

```java
public class Field {
          …
    public String generateCode() {
        String generatedCode = "";
        /* Do something. */
        return generatedCode;
    }
}
```

```java
public class Statement {
          …
    public String generateCode() {
        String generatedCode = "";
        /* Do something. */
        return generatedCode;
    }
}
```

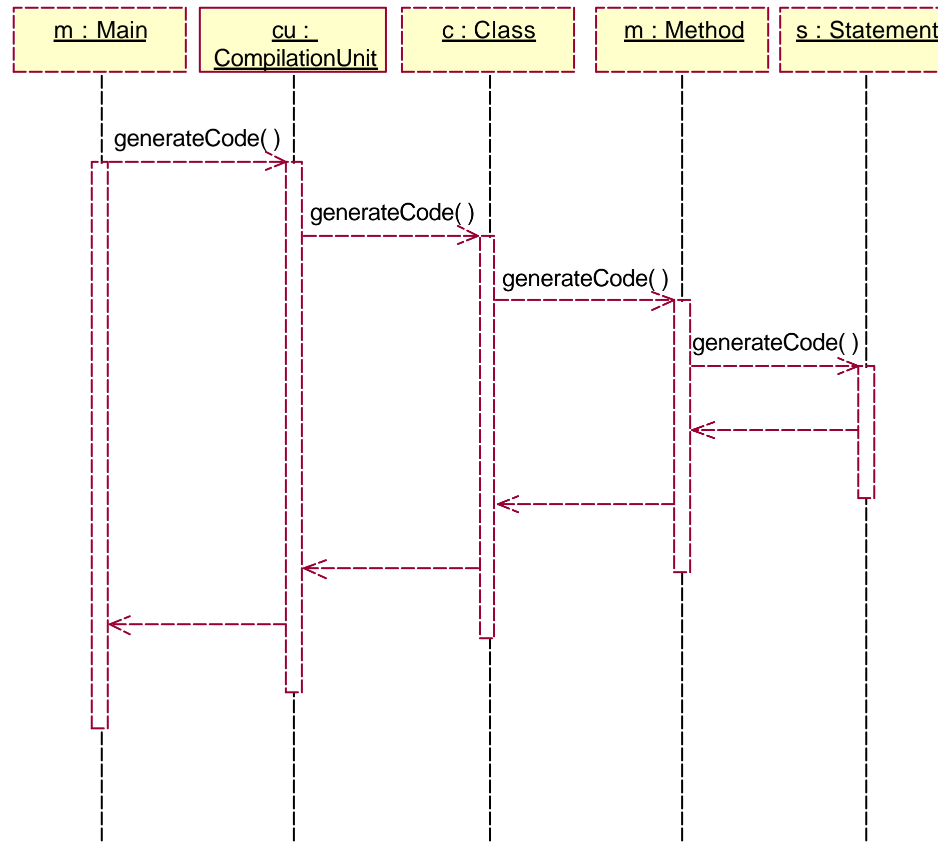
# Naïve implementation (3/7)

```
public class Class {
        …
    public String generateCode() {
        String generatedCode = "";
        /* Do something at the beginning. */
        final Iterator iteratorOnFields = this.fields.iterator();
        while (iteratorOnFields.hasNext()) {
            final Field aField = (Field) iteratorOnFields.next();
            generatedCode += aField.generateCode();
        }
        final Iterator iteratorOnMethods = this.methods.iterator();
        while (iteratorOnMethods.hasNext()) {
            final Method aMethod = (Method) iteratorOnMethods.next();
            generatedCode += aMethod.generateCode();
        }
        /* Do something at the end. */
        return generatedCode;
    }
}
```

# Naïve implementation (4/7)

```
public class CompilationUnit {
        …
    public String generateCode() {
        String generatedCode = "";
        /* Do something at the beginning. */
        final Iterator iterator = this.classes.iterator();
        while (iterator.hasNext()) {
            final Class aClass = (Class) iterator.next();
            generatedCode += aClass.generateCode();
        }
        /* Do something at the end. */
        return generatedCode;
    }
}
```

# Naïve implementation (5/7)



# Naïve implementation (6/7)

- Limitations of the naïve implementation
  - What about generating code for
    - Linux on PowerPC?
    - Linux on Motorola 68060?
    - OS/400 on AS/400?

# Naïve implementation (6/7)

- Limitations of the naïve implementation
  - What about generating code for
    - Linux on PowerPC?
    - Linux on Motorola 68060?
    - OS/400 on AS/400?

Combinatorial explosion of `generateCodeForXXX()` methods in each class

# Naïve implementation (7/7)

- Requirements
  - Decouple the data structure
    - The AST
  - From algorithms on the data structure
    - The `generateCodeForXXX()` method
    - And others, including type binding!

# Naïve implementation (7/7)

- Requirements
  - Decouple the data structure
    - The AST
  - From algorithms on the data structure
    - The `generateCodeForXXX()` method
    - And others, including type binding!

The Visitor design pattern guides you to do that!

*“Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”*

[Gamma *et al.*]

# The Visitor design pattern (2/11)

- Name : Visitor
- Intent : *"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."*

# The Visitor design pattern (3/11)

- Motivation : *“Consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined. It will also need to generate code.”*

- Motivation (cont'd) : *"[…] It will be confusing to have type-checking code mixed with pretty-printing code or flow analysis code. […] It would be better if each new operation could be added separately, and the node classes were independent of the operations that apply to them."*

- Motivation (cont'd) : *"We can have both by packaging related operations from each class in a separate object, called a visitor, and passing it to elements of the abstract syntax tree as it's traversed."*

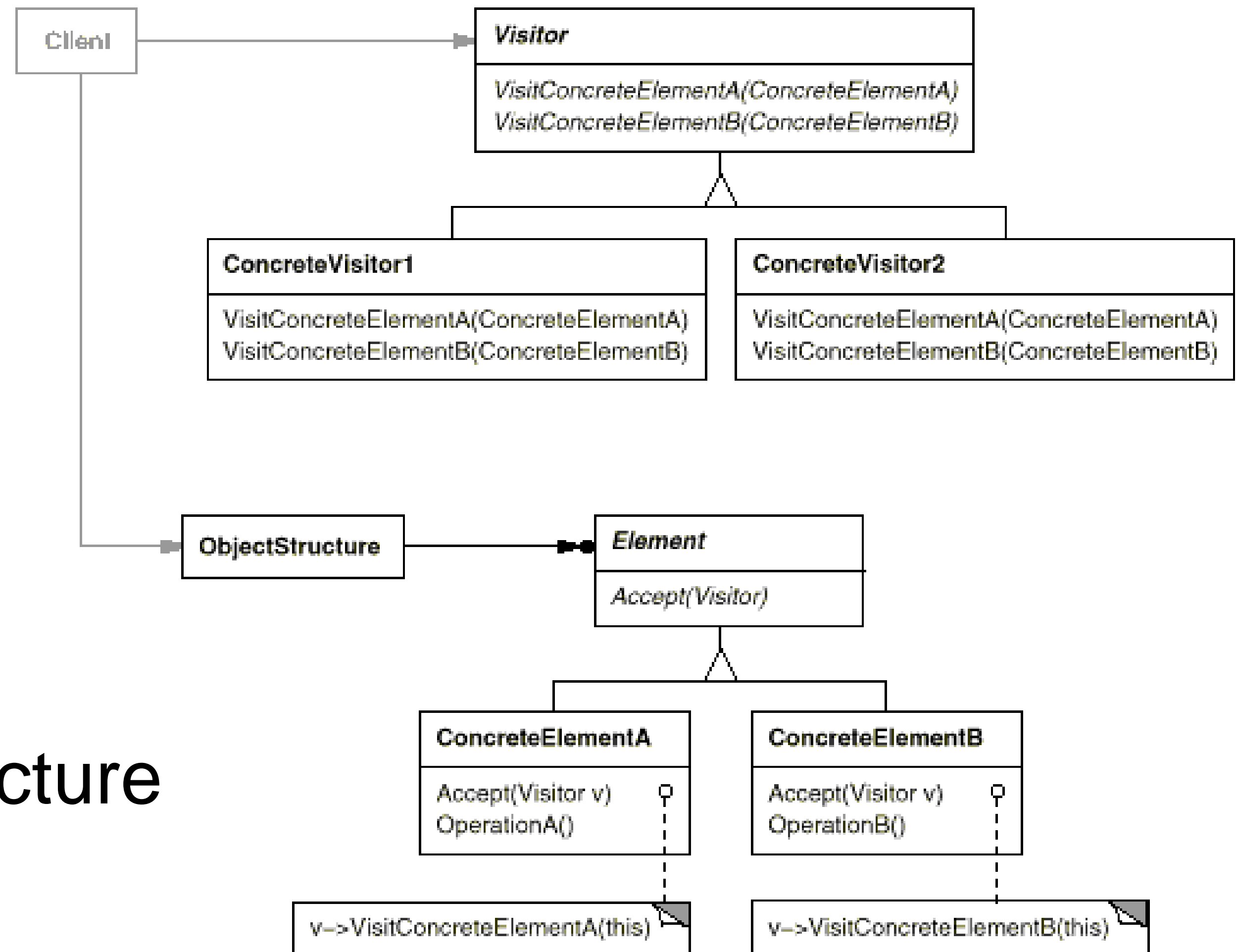
# The Visitor design pattern (6/11)

- Motivation (cont'd) : *"When an element accepts the visitor, it sends a request to the visitor that encodes the element's class. It also includes the element as an argument. The visitor will then execute the operation for that element—the operation that used to be in the class of the element."*

# The Visitor design pattern (7/11)

- Applicability : Use the pattern when
  - An object structure contains many classes of objects with differing interfaces…
  - Many distinct and unrelated operations need to be performed on objects in an object structure…
  - The classes defining the object structure rarely change, but you often want to define new operations over the structure…

# The Visitor design pattern (8/11)



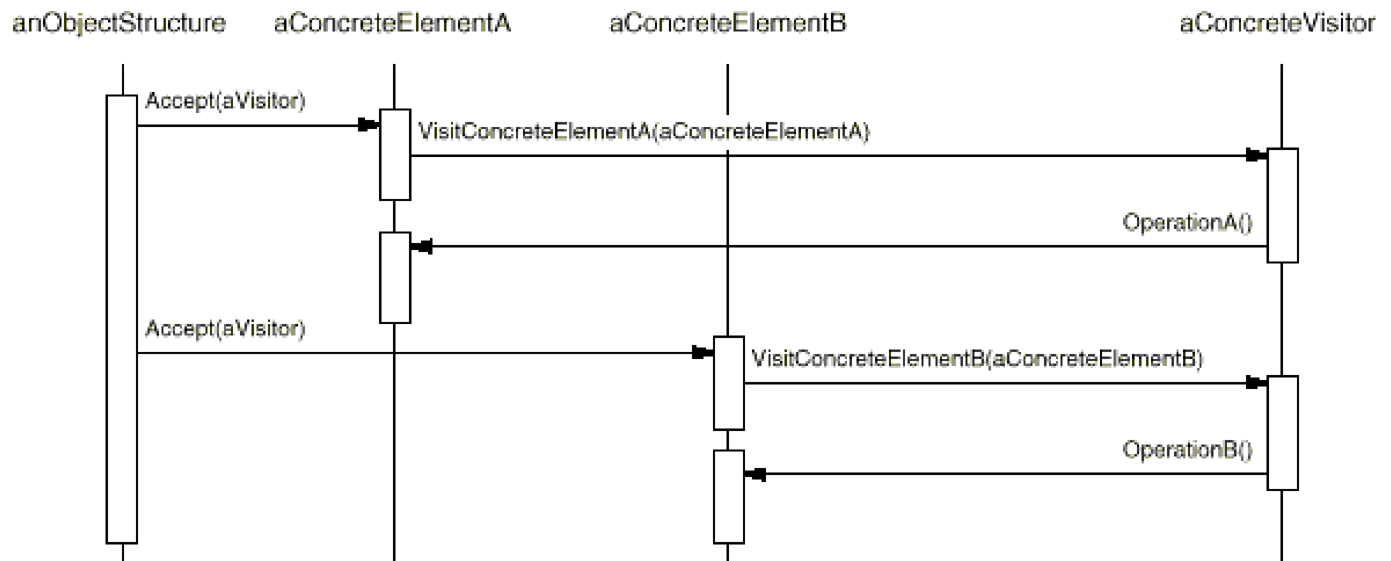

- Structure

# The Visitor design pattern (9/11)

- Participants
  - Visitor (`NodeVisitor`)
    - Declares a Visit operation for each class…
  - ConcreteVisitor (`TypeCheckingVisitor`)
    - Implements each Visit…
  - Element (`Node`)
    - Defines an Accept operation…
  - ConcreteElement (`AssignmentNode`)
    - Implements Accept …
  - ObjectStructure (`Program`)
    - Can enumerate its elements
    - May provide a high-level interface to allow the visitor to visit its elements
    - May either be a composite (see Composite) or a collection

# The Visitor design pattern (10/11)

- Collaborations



- Consequences : …
- Implementation : …
- Sample code : …
- Known uses
  - …
  - PADL ☺
- Related patterns : Composite and Interpreter

# Better implementation (1/6)

```java
package compiler.visitor;

import compiler.Class;
import compiler.CompilationUnit;
import compiler.Field;
import compiler.Method;
import compiler.Statement;

public interface Visitor {
    void open(final Class aClass);
    void open(final CompilationUnit aCompilationUnit);
    void open(final Method aMethod);
    void close(final Class aClass);
    void close(final CompilationUnit aCompilationUnit);
    void close(final Method aMethod);
    void visit(final Field aField);
    void visit(final Statement aStatement);
}
```

# Better implementation (2/6)

```
public class Method {
        …
    public void accept(final Visitor aVisitor) {
        aVisitor.open(this);
        final Iterator iterator = this.statements.iterator();
        while (iterator.hasNext()) {
            final Statement aStatement = (Statement) iterator.next();
            aStatement.accept(aVisitor);
        }
        aVisitor.close(this);
    }
}
```

```
public class Field {
        …
    public void accept(final Visitor aVisitor) {
        aVisitor.visit(this);
    }
}
```

```
public class Statement {
        …
    public void accept(final Visitor aVisitor) {
        aVisitor.visit(this);
    }
}
```

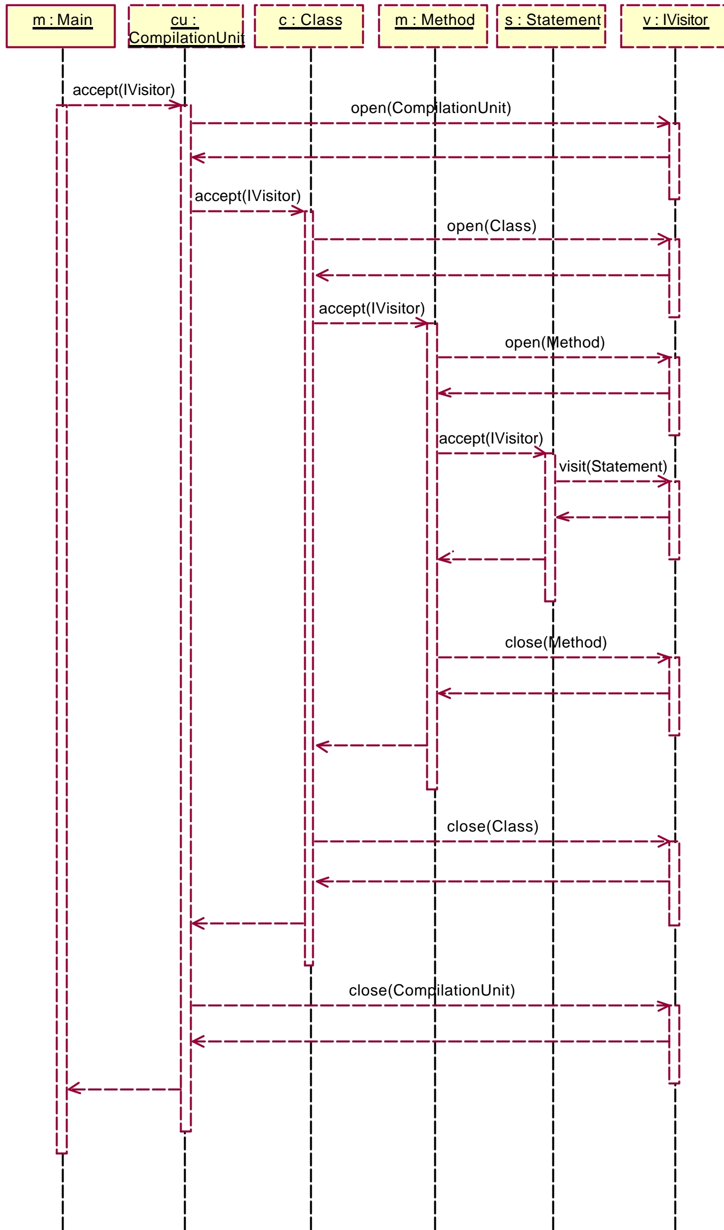
# Better implementation (3/6)

```java
public class Class {
            …
    public void accept(final Visitor aVisitor) {
        aVisitor.open(this);
        final Iterator iteratorOnFields = this.fields.iterator();
        while (iteratorOnFields.hasNext()) {
            final Field aField = (Field) iteratorOnFields.next();
            aField.accept(aVisitor);
        }
        final Iterator iteratorOnMethods = this.methods.iterator();
        while (iteratorOnMethods.hasNext()) {
            final Method aMethod = (Method) iteratorOnMethods.next();
            aMethod.accept(aVisitor);
        }
        aVisitor.close(this);
    }
}
```

# Better implementation (4/6)

```java
public class CompilationUnit {
          …
    public void accept(final Visitor aVisitor) {
        aVisitor.open(this);
        final Iterator iterator = this.classes.iterator();
        while (iterator.hasNext()) {
            final Class aClass = (Class) iterator.next();
            aVisitor.accept(aClass);
        }
        aVisitor.close(this);
    }
}
```

# Better implementation (5/6)



- By using the visitor design pattern
  - Decouple data structure and algorithms
  - Allow the addition of new algorithms without changing the data structure

# Better implementation (6/6)

- By using the visitor design pattern
  - Decouple data structure and algorithms
  - Allow the addition of new algorithms without changing the data structure

Much better, clearer implementation!

- The Visitor design pattern is useful anywhere you have
  - A data (stable) structure
  - Algorithms (infinite) on that data structure

- Design patterns provided *good* solution to recurrent problems!