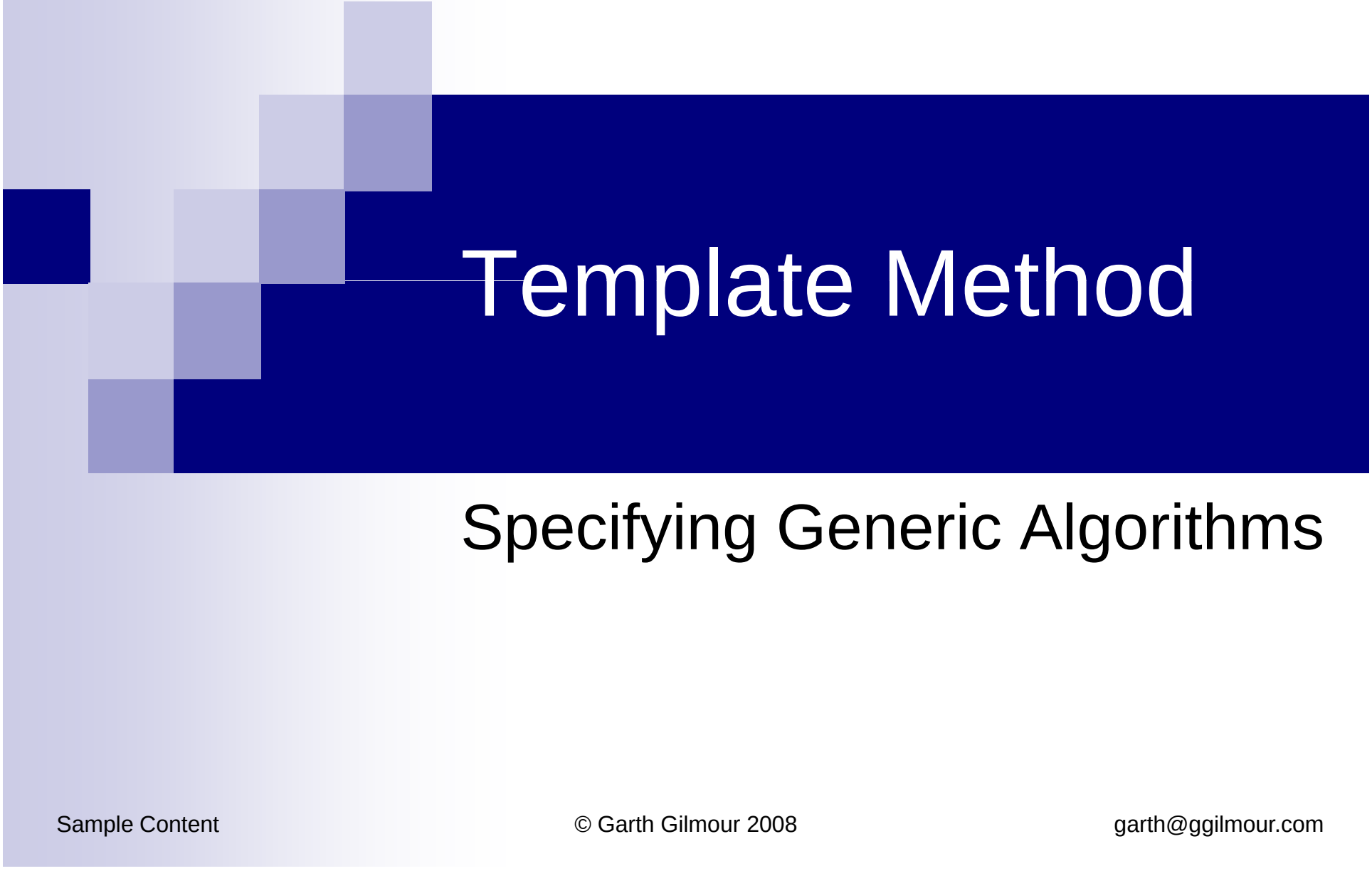


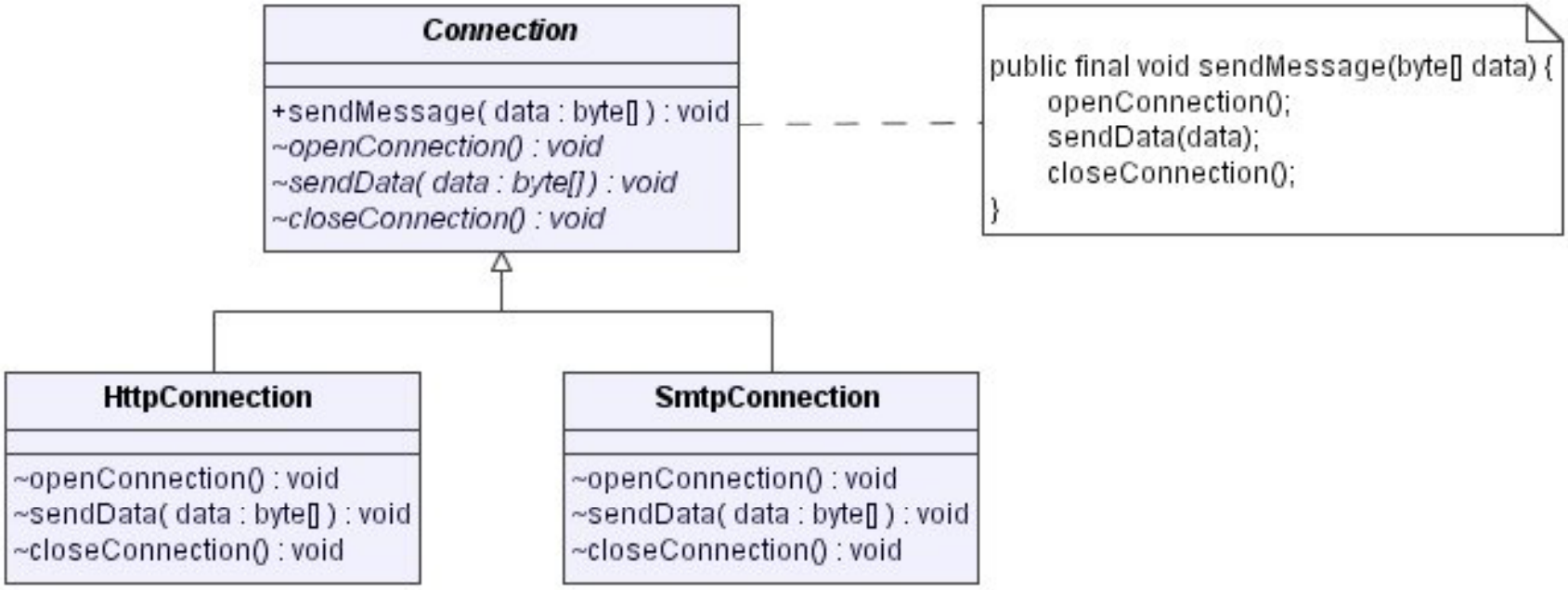
Specifying Generic Algorithms

garth@ggilmour.com

The Template Method Pattern

- Template Method is used when:
  - The steps of an algorithm are well defined
  - The individual steps have varying implementations
- To implement the pattern
  - A class hierarchy is created with a generic base class and subclasses for each implementation
  - The base class has a concrete method which calls the steps of the algorithm in the correct sequence
  - Each step is represented by a polymorphic method
    - This method should be abstract if there is no sensible default
  - The derived classes override the methods for steps as required

Implementing A Template Method



Examples of Template Method

- ‘Thread’ is a simple example of Template Method
  - You extend the base class and override the ‘run’ method
- Rendering in Swing uses a template method
  - The ‘paint’ method of ‘JComponent’ calls ‘paintBorder’, ‘paintComponent’ and ‘paintChildren’
  - To implement custom rendering override ‘paintComponent’
  - The drawing you do is buffered to reduce flicker
- Calls to Servlets also pass through a template
  - The ‘service’ method of ‘HttpServlet’ detects the type of the request and triggers the matching method
  - You derive class and implement ‘doGet’, ‘doPost’, ‘doHead’, ‘doOptions’, ‘doTrace’ or ‘doDelete’ as required

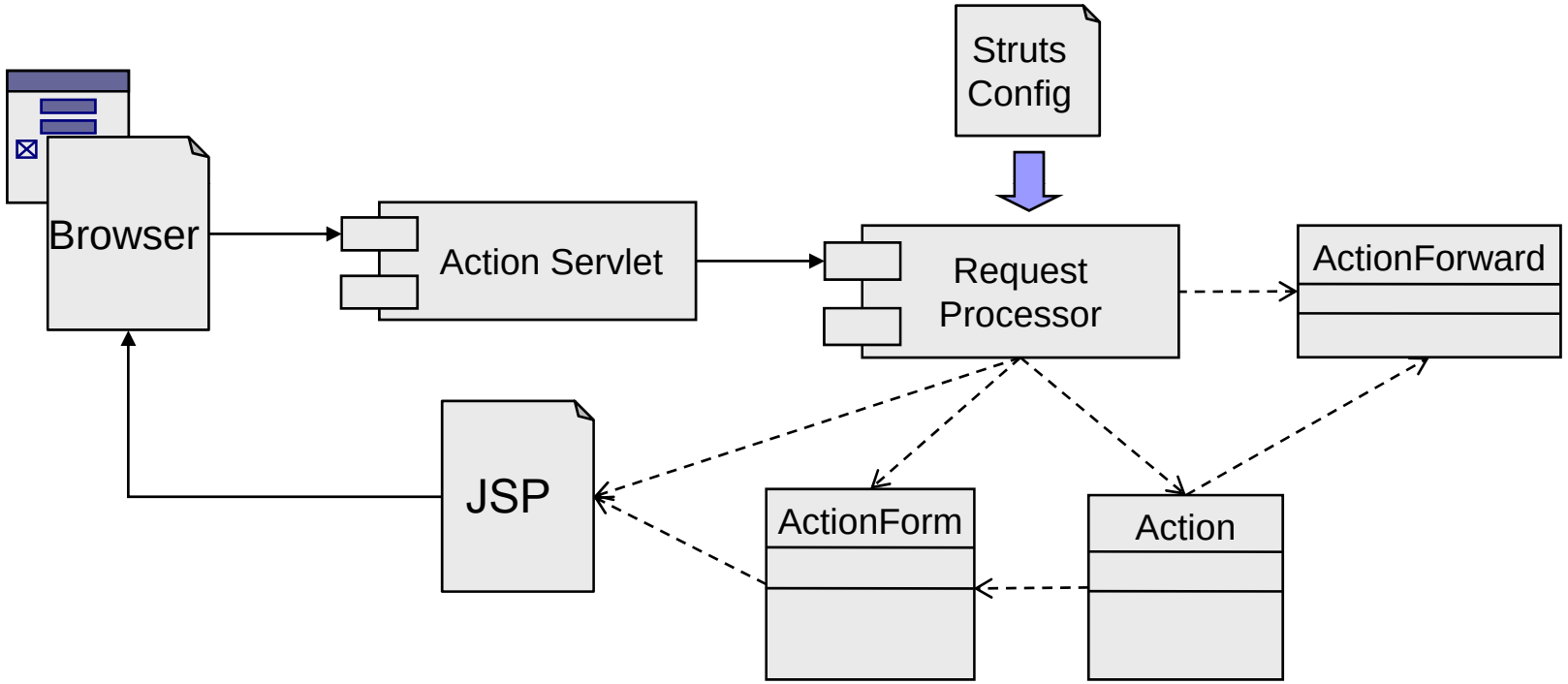
Template Method & Class Loaders

- Sometimes all the steps in the algorithm will have a default implementation in the base class
  - The pattern is used to customize the implementation
  - Normally you will only replace a single step
- This version is used to develop custom Class Loaders
  - The steps are implemented in the base class
  - Except for the ‘findClass’ method which is overridden to locate the specified class file and generate a Class object
    - The ‘defineClass’ method builds a Class object from a byte array

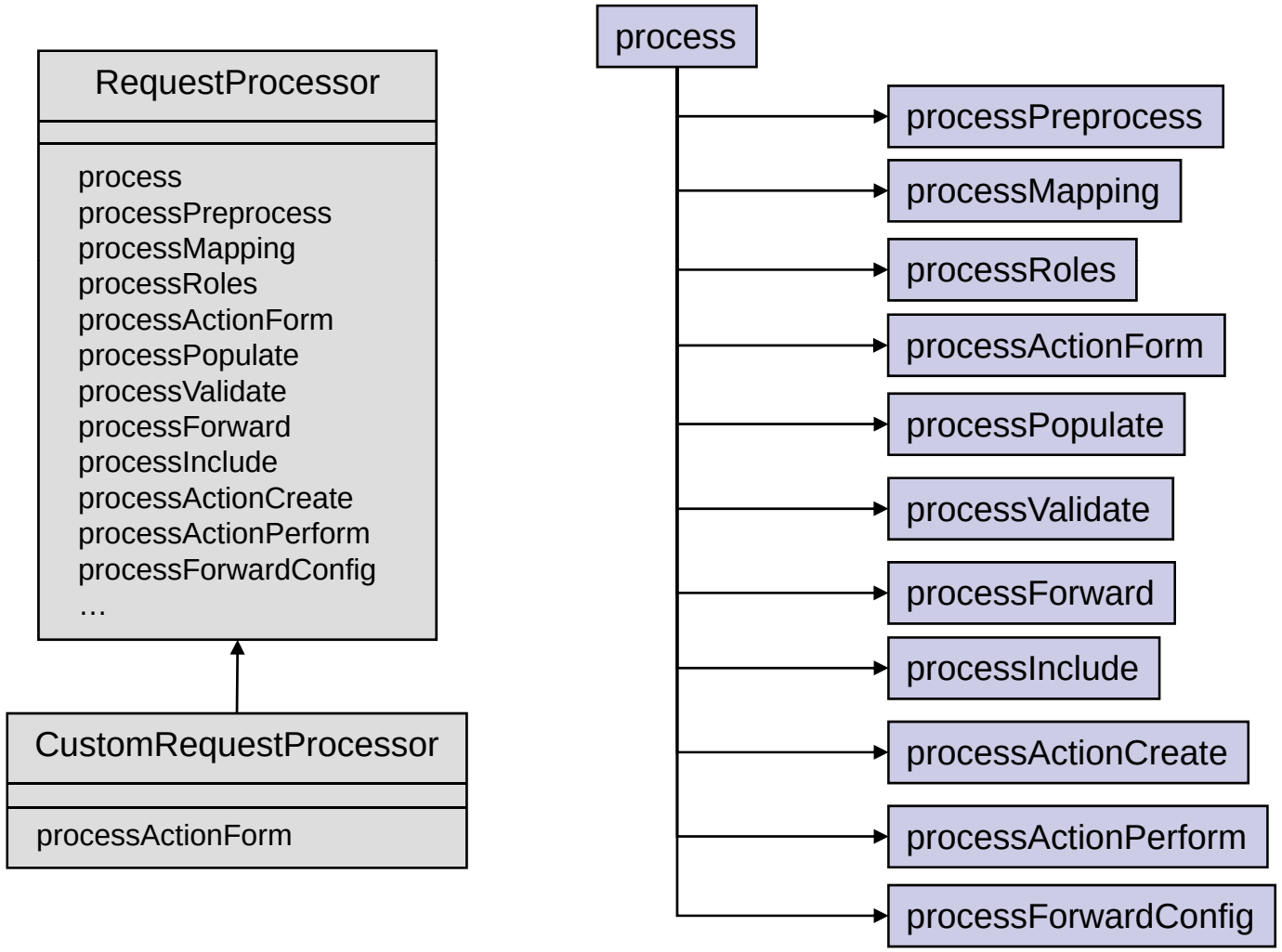
Default Template Steps and Struts

- Struts is based around Template Method
  - In V1.0 the ‘ActionServlet’ implemented the lifecycle
  - This made customizing single steps very difficult
- V1.1 introduced the ‘RequestProcessor’ class
  - The ‘process’ method manages the lifecycle
  - For each step there is a ‘processXXX’ method
    - E.g. ‘processPopulate’ takes parameters from an HTTP request and uses them to set the properties of an ‘ActionForm’ object
- Any step can be overridden to customize the lifecycle
  - The new processor is registered in the configuration file

The Struts MVC Architecture



Template Methods in Struts



Visitor Pattern

Simplifying Class Hierarchies

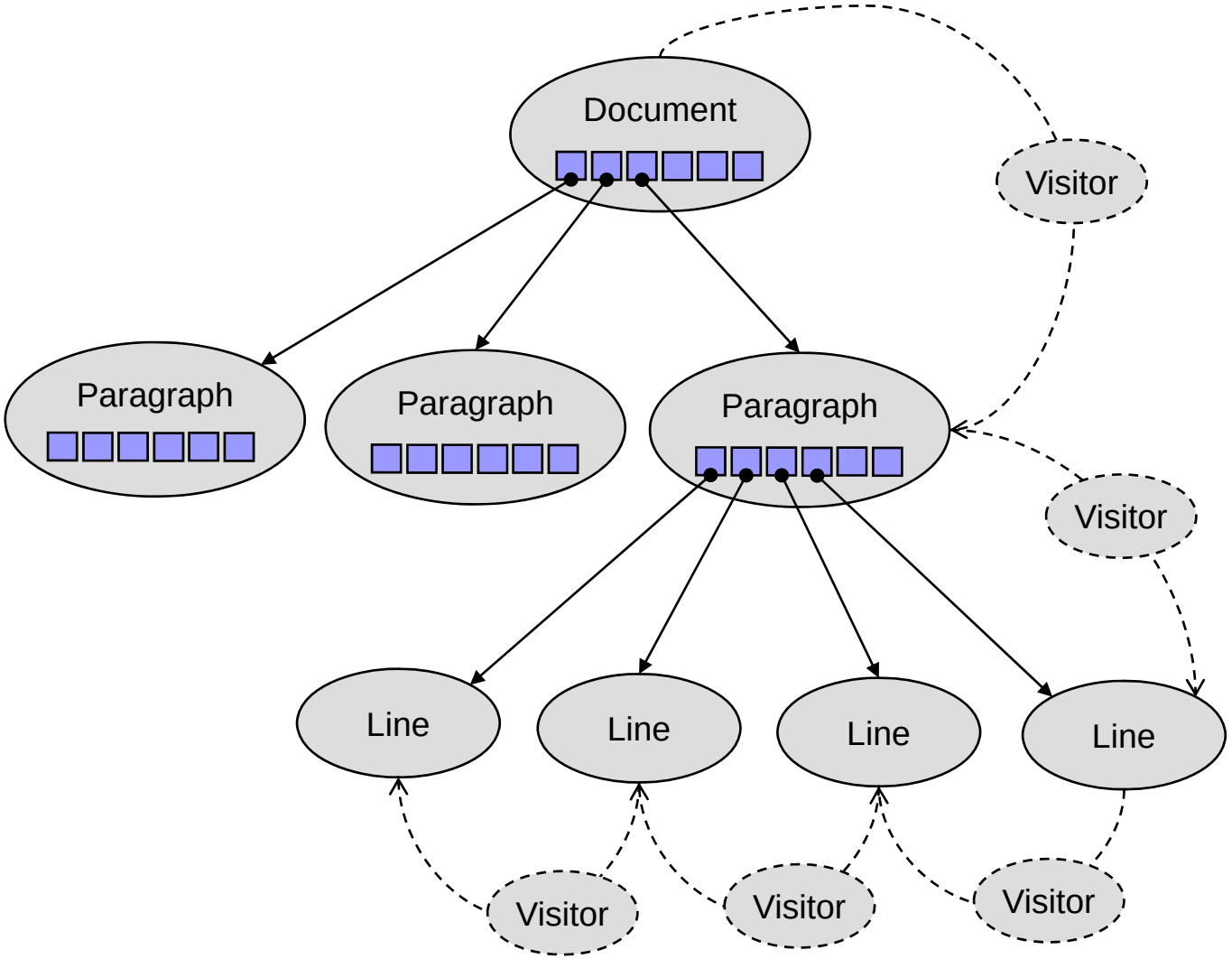
garth@ggilmour.com

The Visitor Pattern

- Visitor is used to add extra functionality to an established structure of collaborating objects
  - Such as a Document Object Model (DOM)
- Visitor should be applied when:
  - The object structure is stable and will not be modified
  - There is a need to add arbitrary functionality that requires access to the tree of objects in the structure
  - The extra functionality cannot be placed in methods because:
    - Adding all the functionality as methods would confuse the code
    - Different clients will need different subsets of the functionality
    - We must support adding functionality at any future time

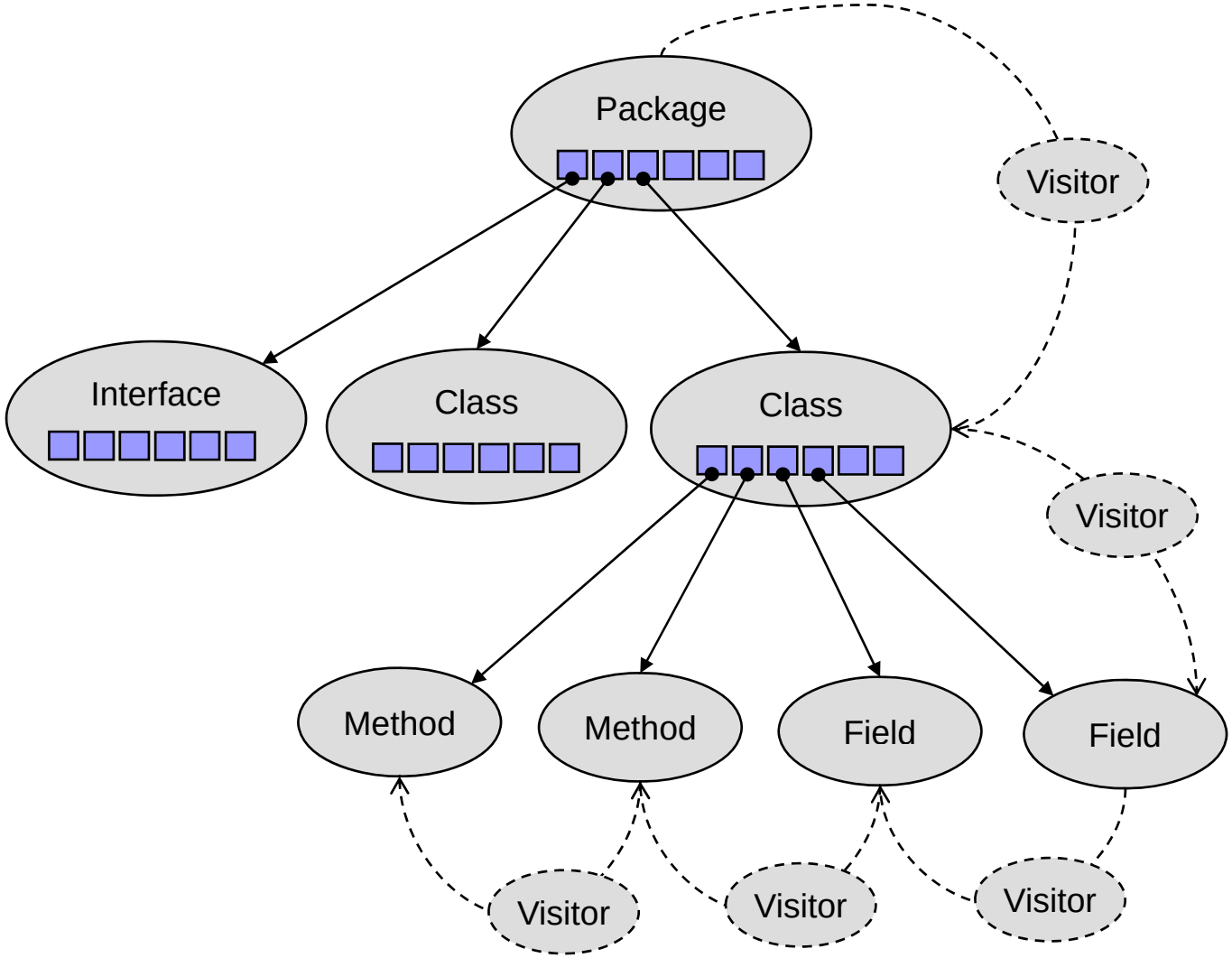
Implementing Visitor

- To implement Visitor
  - Create a hierarchy of visitors that model possible tasks
  - Create methods in the main classes to accept and use visitors
- Visitor can be difficult to implement
  - Functionality must be distributed between two classes
  - The visitor needs to access the state of the main class
  - Visitors often need a mechanism to iterate over and visit a number of classes, such as to compile a report
- The Visitor pattern is rarely used
  - But where it can be applied it is very useful



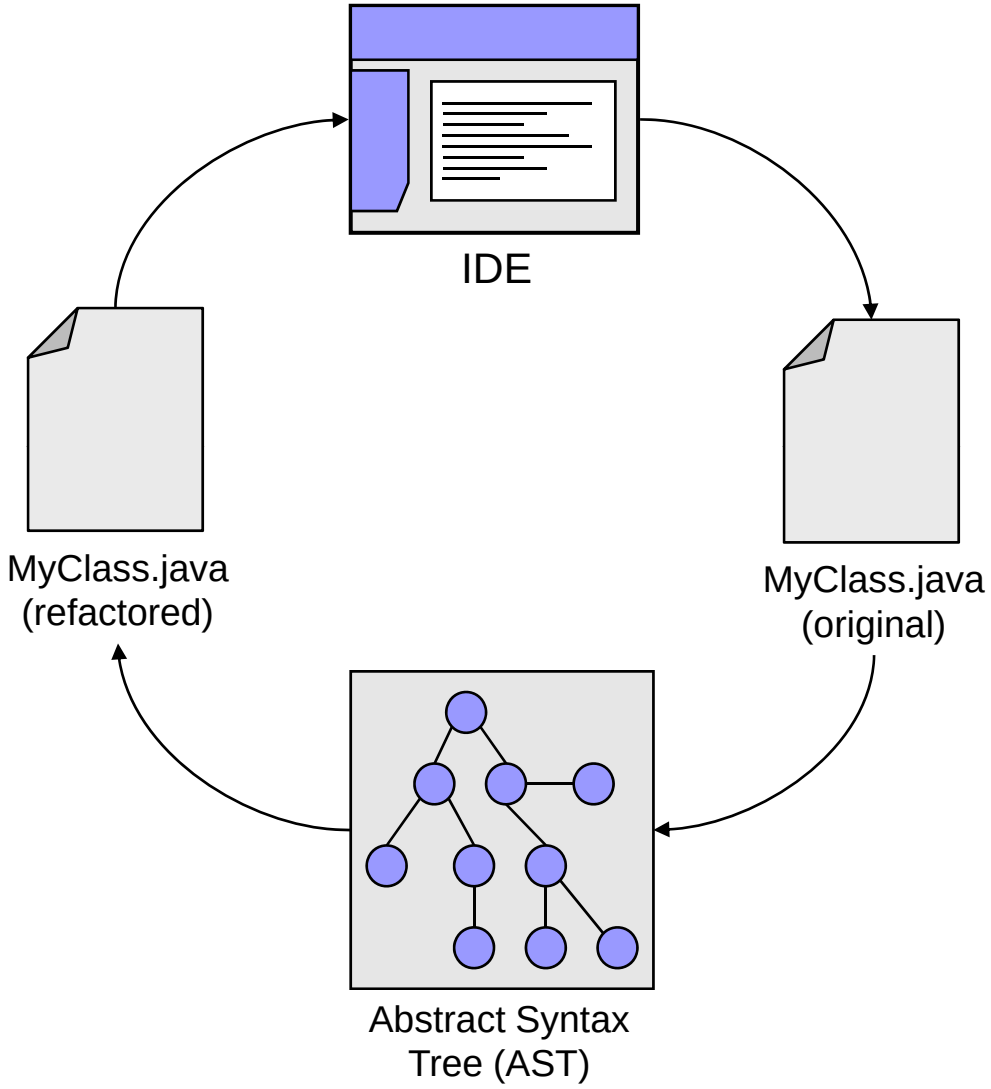
ASM Library and the Visitor Pattern

- ASM is a bytecode manipulation framework
  - It lets you process the contents of a class file
- The library is based around the Visitor Pattern
  - The ‘ClassVisitor’, ‘FieldVisitor’ and ‘MethodVisitor’ interfaces define the methods for visiting class members and bytecode
  - You could use these to map dependencies between classes, generate code quality metrics, find symbols etc…
  - If you were creating a compiler you could use it to write unit tests which validated the generated bytecode instructions
- The library also uses the Visitor Pattern in reverse
  - Unusually you can call the ‘visit’ methods in order to generate new classes and populate their methods with bytecode

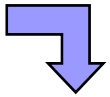


Eclipse and the Visitor Pattern

- The Eclipse IDE has its own library for representing code
  - The classes are found in ‘org.eclipse.jdt.core’
  - The ‘ICompilationUnit’ interface represents a ‘.java’ file
- Changes can be made on a ‘working copy’ of the code
  - Calling ‘getWorkingCopy’ on a compilation unit makes the copy
  - This can be merged with the original by calling ‘reconcile’’
- Visitor can be used to implement a complex refactoring
  - By passing a visitor object around the Abstract Syntax Tree
  - The ‘ASTParser’ class converts the source into an AST
    - Which can then be navigated and manipulated



```
public class MyClass {
	public MyClass(int val) {
		this.val = val;
	}
	public void setVal(int param) {
		val = param;
	}
	public int getVal() {
		return val;
	}
	private int val;
}
```



```
public class MyClass {
	public MyClass(int val) {
		this._val = val;
	}
	public void setVal(int param) {
		_val = param;
	}
	public int getVal() {
		return _val;
	}
	private int _val;
}
```

```
public class FieldModifierASTVisitor extends ASTVisitor {
	public boolean visit(VariableDeclarationFragment node) {
		if(node.getParent() instanceof FieldDeclaration) {
		   renameNode(node);
		}
		return true;
	}
	public boolean visit(FieldAccess node) {
		renameNode(node);
		return true;
	}
	private void renameNode(FieldAccess node) {
		AST ast = node.getAST();
		SimpleName oldName = node.getName();
		SimpleName newName = ast.newSimpleName("_" + oldName.getIdentifier());
		node.setName(newName);
	}
	private void renameNode(VariableDeclarationFragment node) {
		AST ast = node.getAST();
		SimpleName oldName = node.getName();
		SimpleName newName = ast.newSimpleName("_" + oldName.getIdentifier());
		node.setName(newName);
	}
}
```