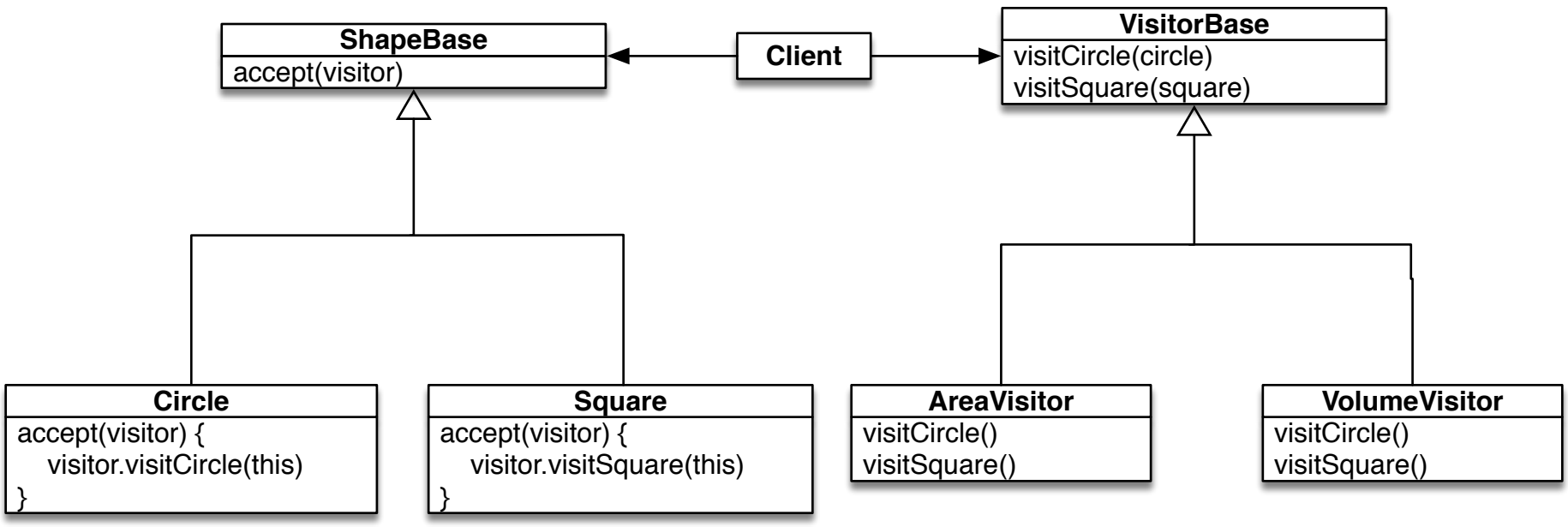
Visitor



Intent: to not obfuscate the subject with add-on (and distinct) logic that operates on it. Instead separate out these add-on logic into visitors that visit the subject and exercise their logic.

E.g. an abstract syntax tree constructed by the compiler, on which we can do semantic analysis as well as indentation.
For each AST node we could add a doSemanticAnalysis() as well as doIndentation(), and attach more functions as we extend the functionality, or we could leave node pure but accept visitors following a visitor interface, and when extending the compiler with another feature, derive a new class from visitor and operate on node structure.

Visitor allows one to keep the visited subject clean and split out add-on logic that operates on it.
For each concrete subject, there needs to be a corresponding call in visitor interface. This makes it easy to build another visitor, but more difficult to build another concrete subject: all visitors need to handle it. Thus it should be applied on fairly stable subjects on which we may want to attach additional logic.
Visitor also could potentially hurt the encapsulation of the subject: visitors may need its internals exposed to accomplish their tasks.

What I learnt when implementing visitor pattern (context: IMC take-home quiz):
- Cyclic dependency of Shape —> Visitor —> ConcreteShape —> Shape, seems only good way is forward declaration, yet I thought forward declaration is completely avoidable
- Return value of visit and accept functions: void seems fair, however, the AreaVisitor wants to get a double out. By making the return value double, the accept call is more specific to a certain kind of visitor that outputs a double
- Dupllcated symbol linker error: only templated / inlined function definitions go into header. Otherwise each translation unit with this header included will have its own definition

Yet to learn:
- The idea “double dispatching”, and a UML drawing + analysis if all interfaces are necessary