Design Patterns and Symbol Table Generation Lab 3

Killian Smith

March 3, 2017

Misc.

- Grading time line
- Lab session next Friday?

Review

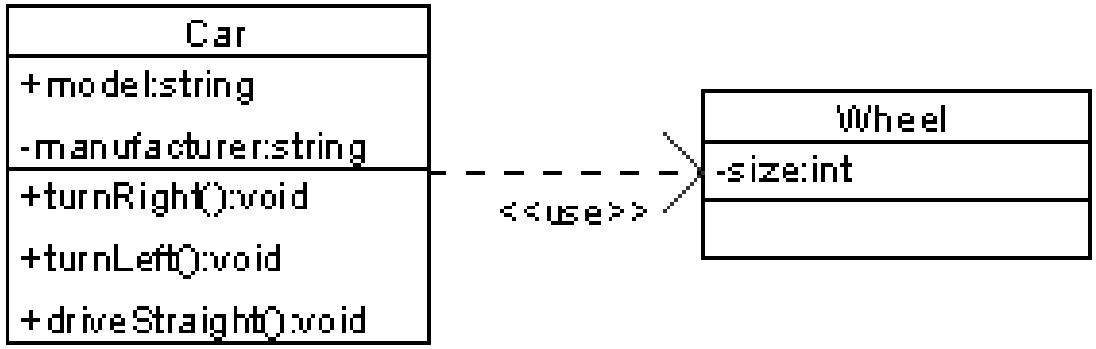
Design Patterns in SE

- General and reusable solutions
- Abstract common patterns dealing with program structure, behavior, concurrency, or object creation.
- Fairly OO centric (something to take into consideration)
- Solutions given in design patterns are usually regarded as "best practice" (when implemented correctly)
- Popularized with the "Gang of Four" book

Review (cont)

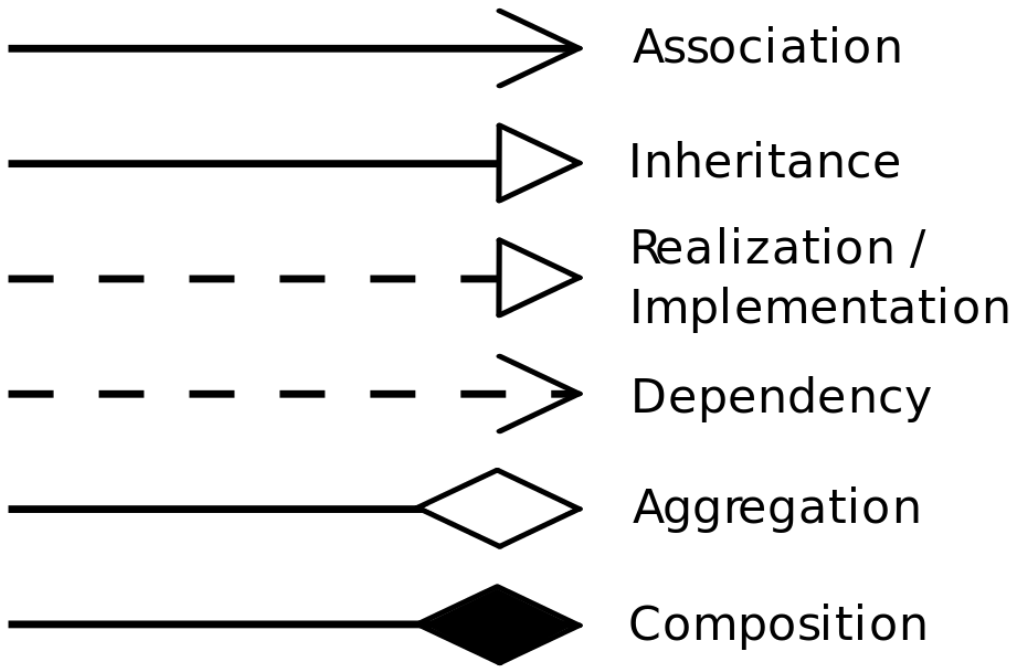
UML Class Diagrams

- A visual representation of objects and their interactions
- Useful for conveying ideas without concrete code

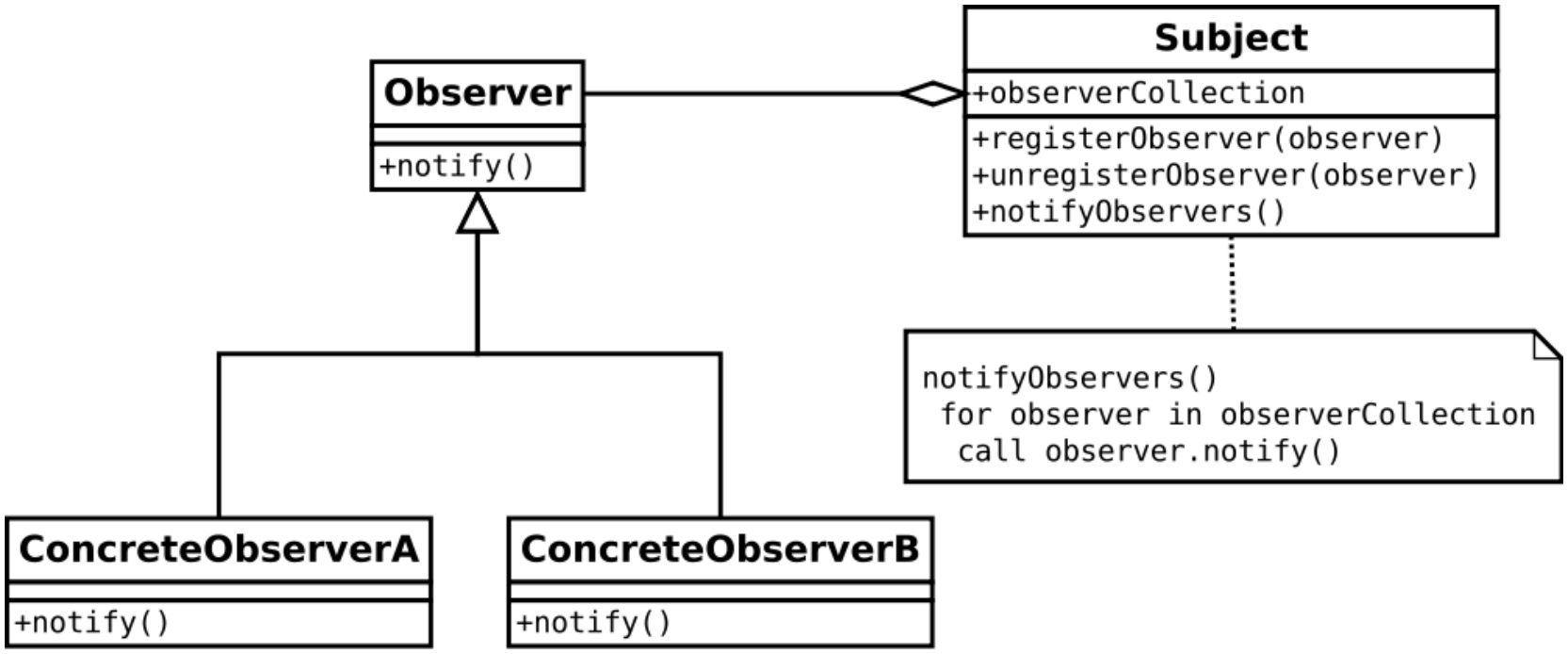


Images are from Wikipedia design pattern pages

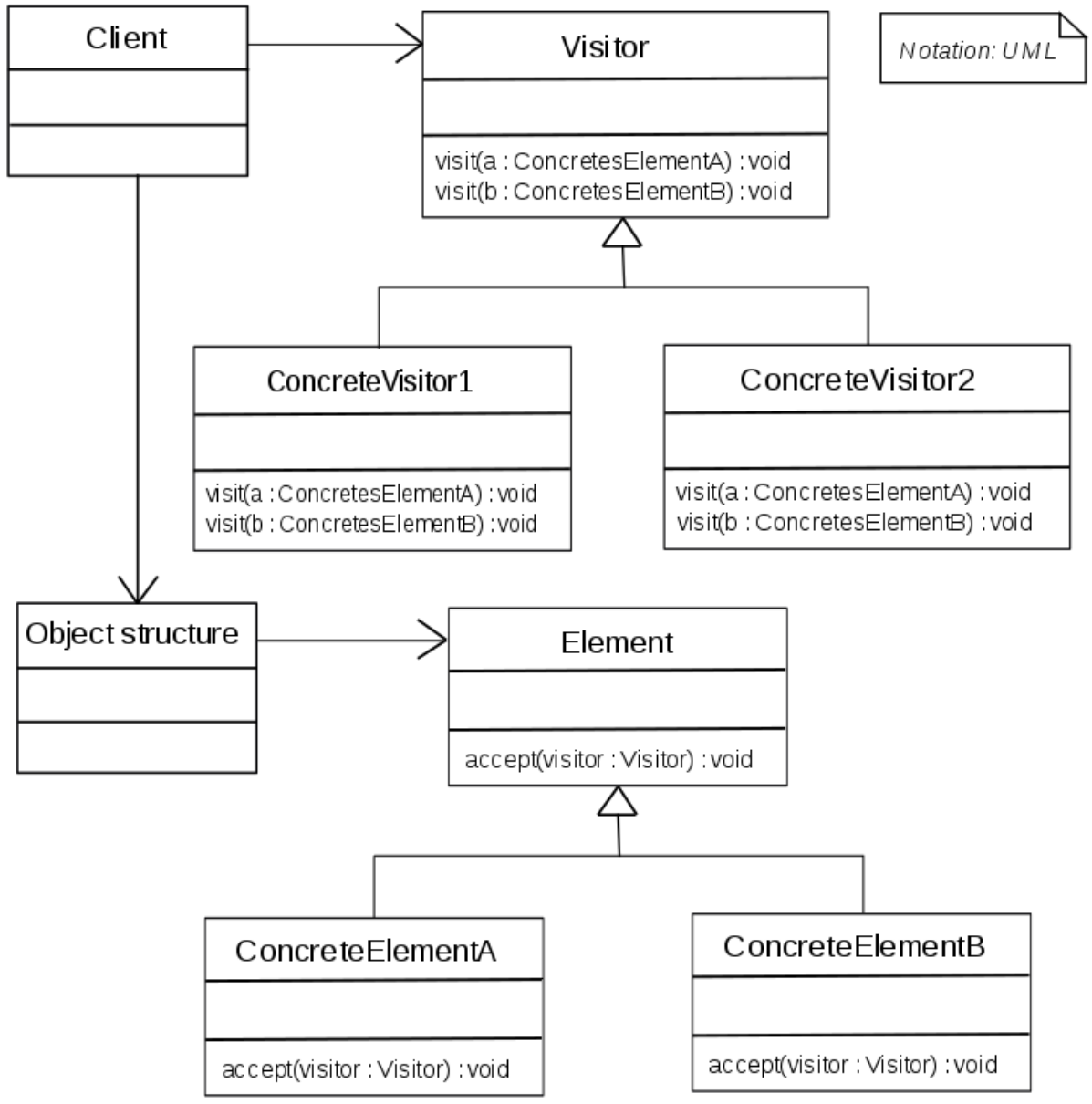
Relations


Association
Inheritance
Realization /
Implementation
Dependency
Aggregation
Composition

Observer Pattern



Visitor Pattern



Using ANTLR Listener

```
LittleParser parser =
    new LittleParser(
        new CommonTokenStream(
            new LittleLexer(
                new ANTLRFileStream(args[0]))));

Listener listener = new Listener();
new ParseTreeWalker().walk(listener,
                           parser.program());

SymbolTable s = listener.getSymbolTable();

prettyPrint(s);
```

Using ANTLR Listener (cont)

```
class Listener extends LittleBaseListener {
    // initialize a symbol table object here
    // may want to have a stack as well...

    @Override
    public void enterFunc_decl(LittleParser.
        Func_declContext ctx){
        // operate on symbol table here
    }

    @Override
    public void exitFunc_decl(LittleParser.
        Func_declContext ctx){
        // operate on symbol table here
    }

    // additional rules and/or helper methods here...

}
```

Using ANTLR Listener (cont)

- This is the default method for taking actions on parse tree traversals
- There are no return values for the listener actions
- State and intermediate structures are handled internally

Using ANTLR Visitor

- Need to run ANTLR with the *visitor* option
- Driver class will have a similar structure to the *listener* example
- Visitor instances must have a return value in ANTLR
  - This is different from the *Listener*, which performs actions without returning.
  - Can do in-place code transformations during translation phase, meaning an intermediate structure is not needed.

Other Approaches

Depending on language, the following may be useful:

- Manual tree traversal, and apply a function to certain nodes/sub-trees
- Pattern matching on sub-trees

Symbol Table

This structure is what will be used to store static information about the source program. Some things to keep in mind:

- Scope must be preserved.
  - Nested symbol table design
  - Use unique identifiers in an additional field. Need to keep track of all parent tables.
- Entries for symbols must contain enough information that the compiler can adequately represent them internally.
  - Need to know the size of the object
  - Need to know arguments for functions

Symbol Table (cont)

Depending on language, the following may be useful:

- Hash tables / dictionaries
  - Fast and easy to use if the language implements them for you
  - May require a stack to keep track of current symbol table
- Tree structure
  - Relatively fast and easy to implement
  - Does not require a stack
- List of lists
  - Relatively fast and easy to implement
  - Does not require a stack
  - Note: This is equivalent to a tree.

Lab 3

Goal is to *completely* generate the symbol table for a source program

- Need to modify Driver to traverse tree
- Need to extend base Listener or Visitor, or make your own actions
- Need to return and pretty-print the Symbol Table
- Test on the provided Step3 files
- The grading script is also provided

Questions or Comments?