Agile Design Principles: The Dependency Inversion Principle

Based on Chapter 11 of Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003.

The Age of Procedural Programming

- Although OO languages have existed since 1967, they only became popular in the late 1980s.
- Prior to that, the main units of structuring were *subroutines* (*procedures*) and, in some languages, such as Modula and Turing, *modules*
- (You can think of a *module* as a class with all fields and methods being static. I.e. classes without objects.)

Dependence in Procedural Programming

- In a procedural language, if a procedure in module C calls a method in module S, there is a dependence between C and S. In java terms

  class C { … S.f() … }

  class S { … **public static void** f() { … } … }

- Since callers are directly coupled to their servers, the callers are not reusable.
- People would make reusable subroutines but it was awkward to make reusable callers.
- Thus dependence naturally follows the direction of the calls.

Dependence in Procedural Programming

- One exception is that some languages allowed pointers to subroutines as parameters. So in C, for example, we can do the following:

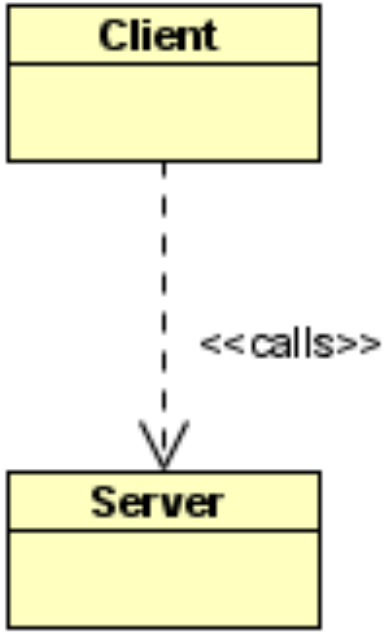
```
double integrate( double (*f)(double),
                  double low, double high, int steps ) {
    … sum += f(x) * width ; … }
```

- So integrate is a reusable caller.

Dependence in OO programming

- In OO programming, the simplest thing to do is often to have dependence follow the direction of calls:
  - **class** S { … **public void** f() { … } … } and
  - **class** C { … **void** g(S s) { … s.f() … ; } or
  - **class** C { S s = **new** S() ; … s.f() … ; } or
  - **class** C { S s ; C(S s) { **this**.s = s ; } … s.f()… } or
  - **class** C { S s ; setS(S s) { **this**.s = s ; } … s.f()… }

Dependence in OO programming



Dependence in OO programming

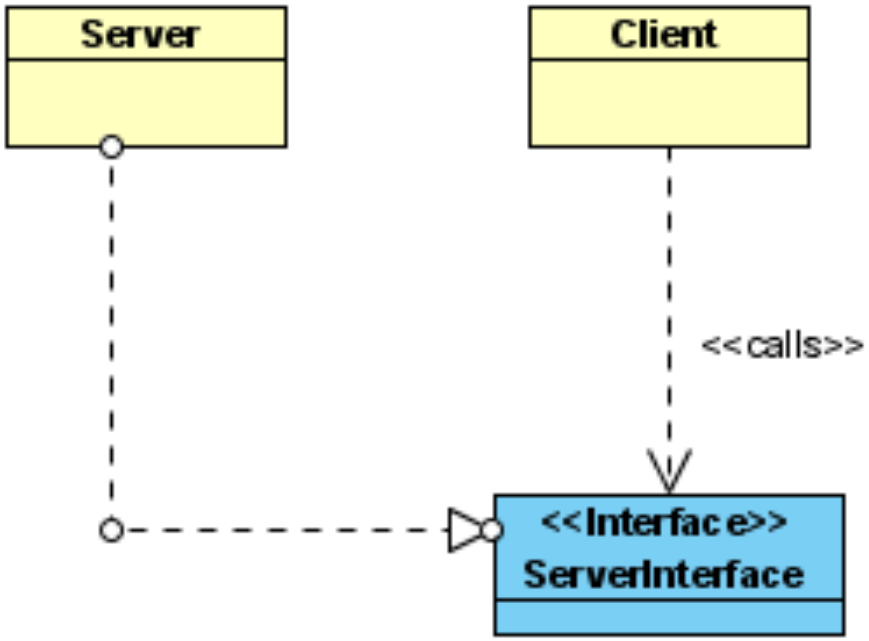
- This style
  - makes it impossible to reuse the caller independently and
  - discourages the designer from viewing the task of the client without reference to the details of what one specific server will do.
  - I.e. it discourages the separation of the concrete interface that one server happens to provide from the abstract interface that the client requires.

Dependence Inversion

- The Dependence Inversion Principle:
  a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  b. Abstractions should not depend on details. Details should depend on abstractions.

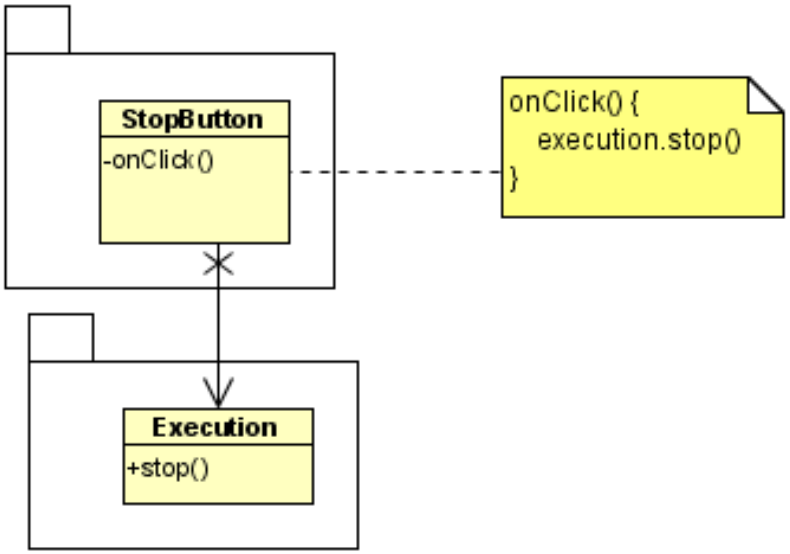
Dependence Inversion

- Our diagram looks like this



Example

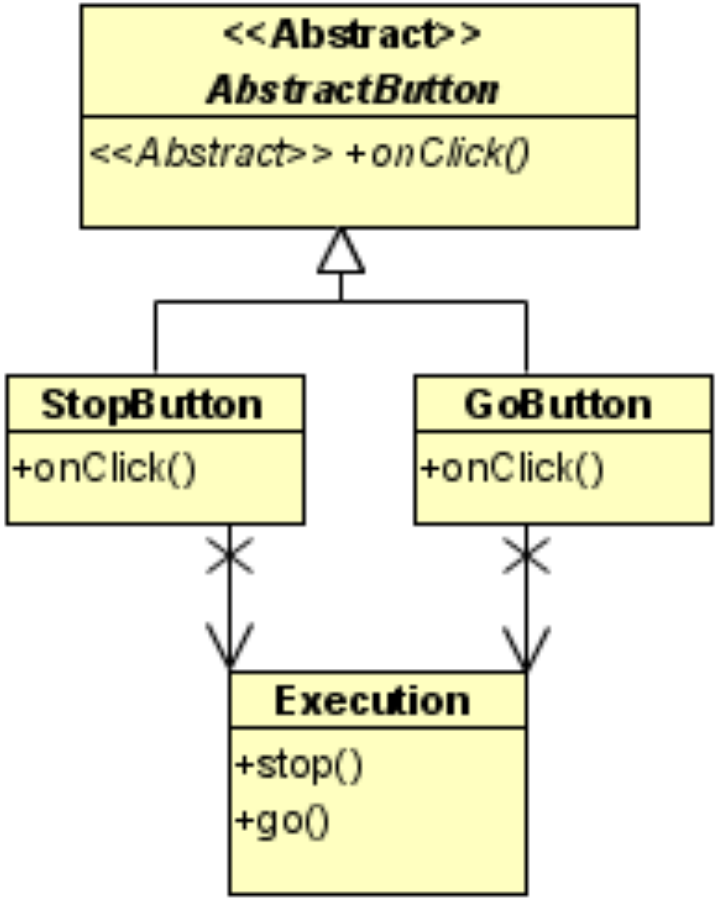
- Buttons. We need a stop button.



- Note that (a) StopButton is not reusable and (b) that the designer of StopButton is thinking only in terms of the concrete task at hand: “stop the execution”

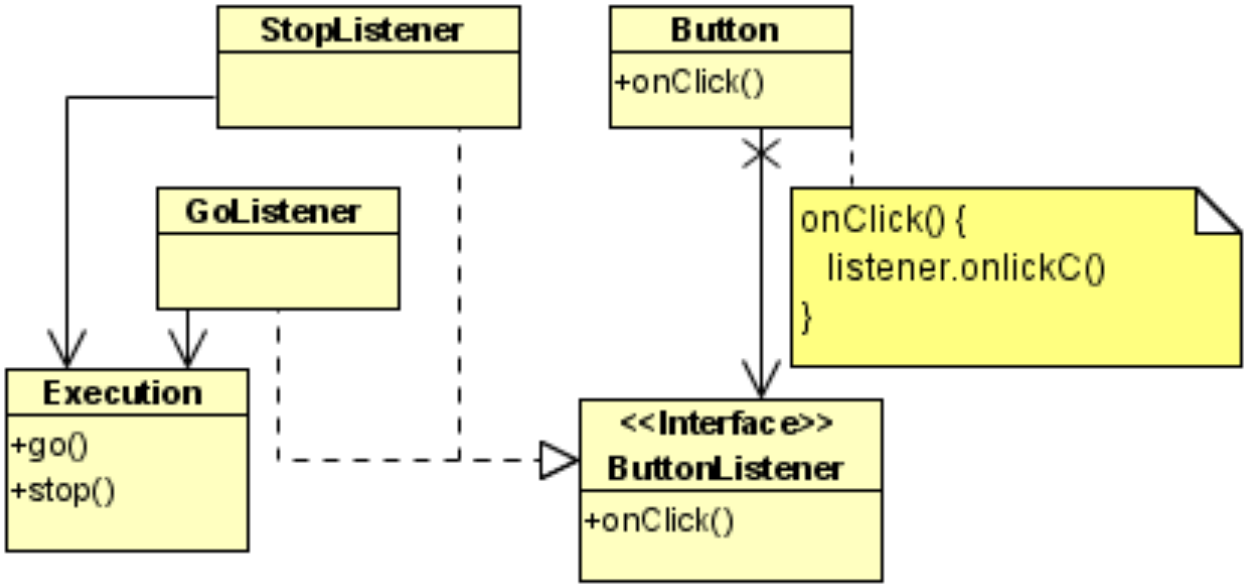
Example

- Soon we need a go buttor as well.
- We use the template method pattern.
- This is a big improvement
- But we are still thinking in terms of the concrete services provided by the "lower levels"



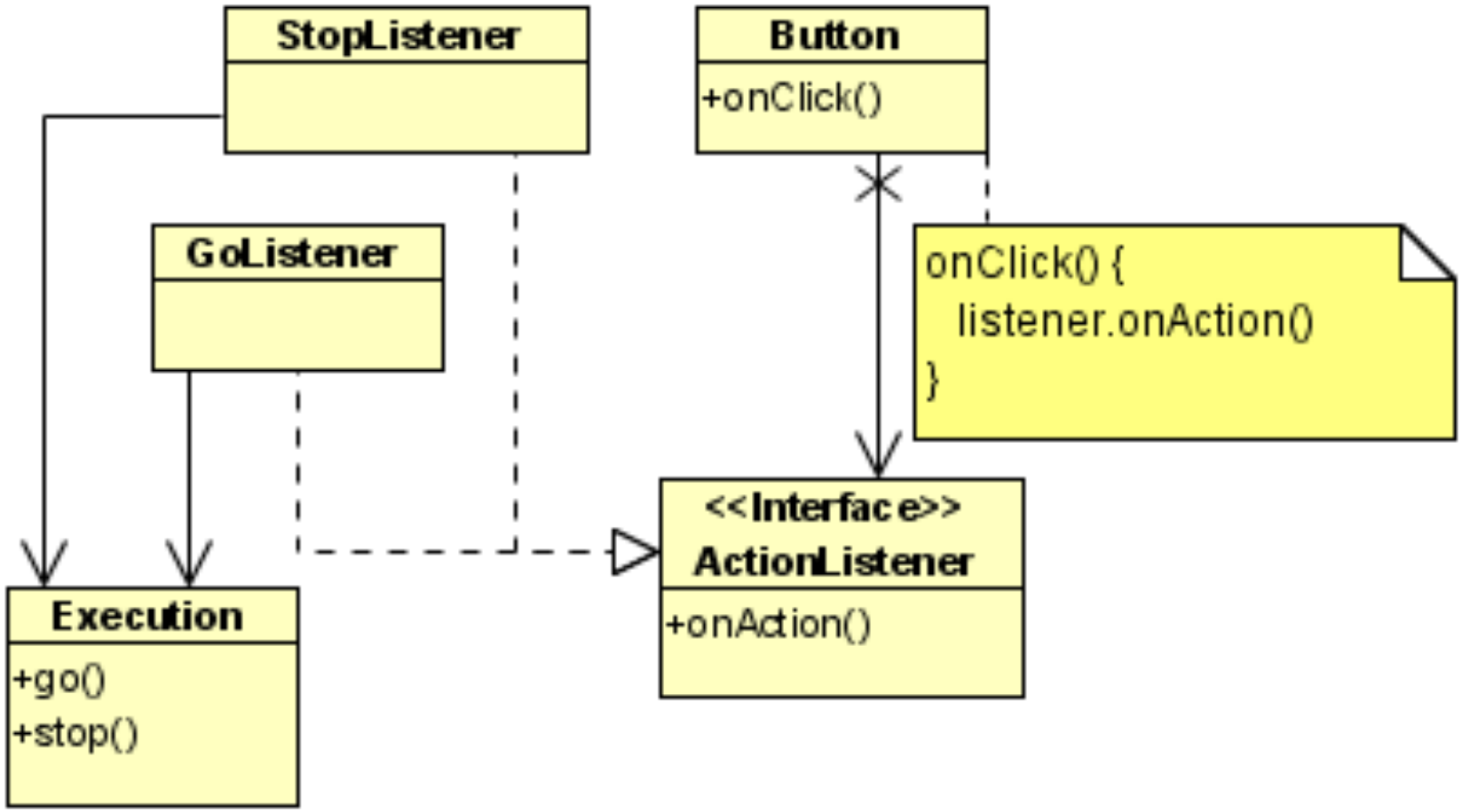
Example

- Remove all dependence



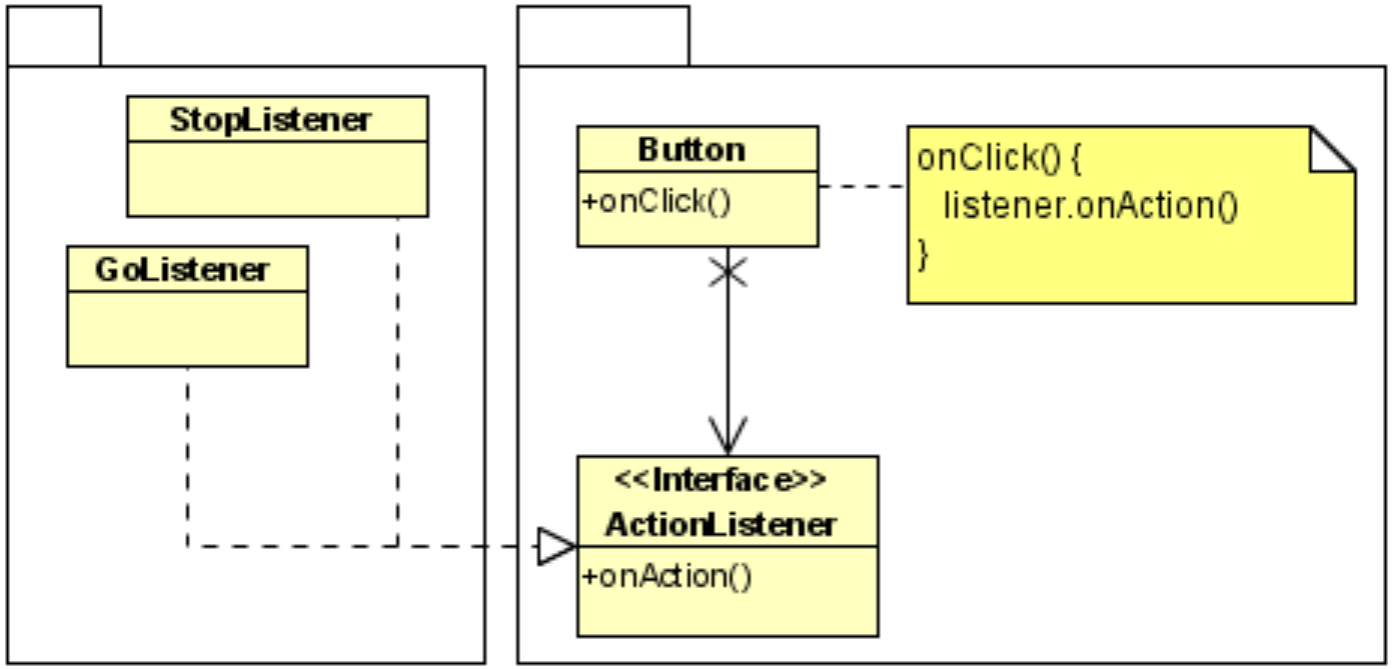
- The naming, however, is too tied to the mechanism. We still have a spiritual dependence.

Example


StopListener
Button
+onClick()
GoListener
onClick() {
listener.onAction()
}
<<Interface>>
ActionListener
+onAction()
Execution
+go()
+stop()

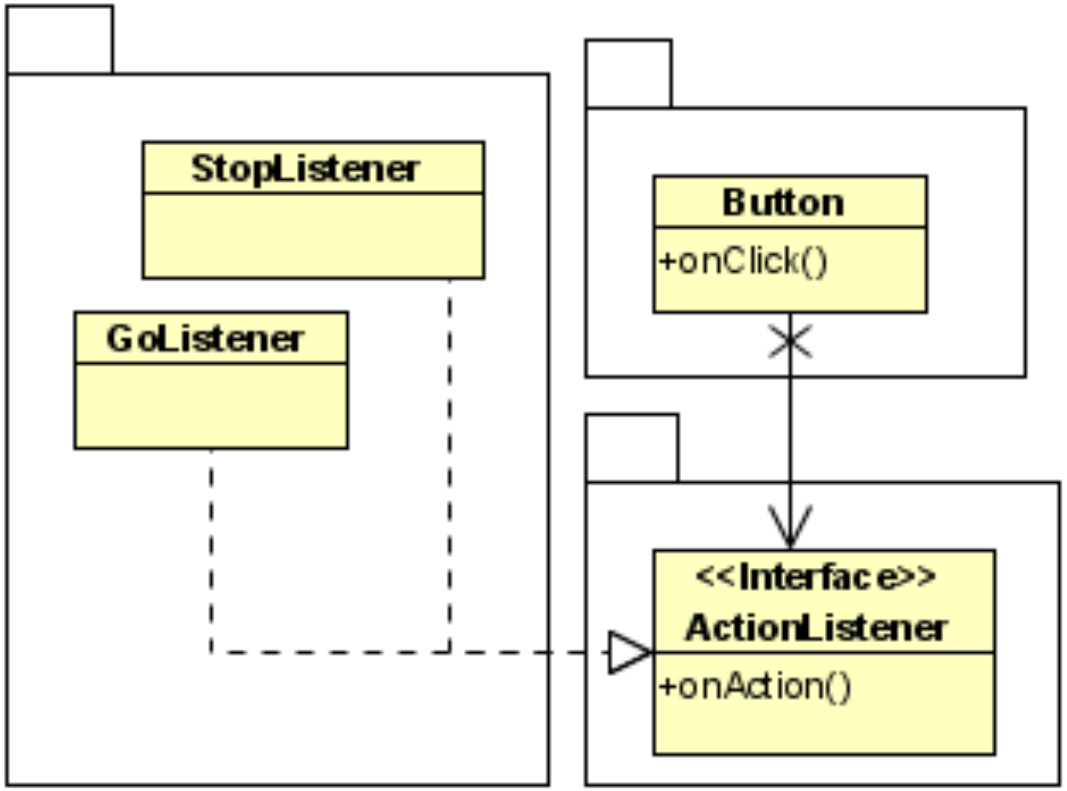
Packaging.

- How should we package these classes?
- Since we intend Button to be reusable and Button depends on ActionListener
- Note that package dependence has been inverted.



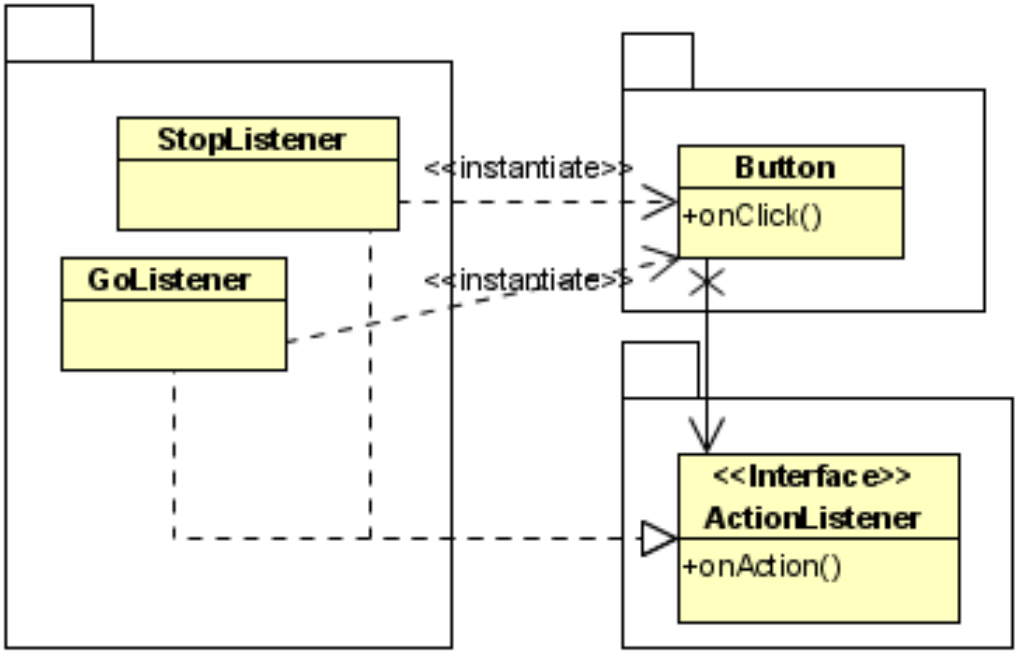
Packaging

- However our ActionListener transcends buttons, so it could be separately reused.



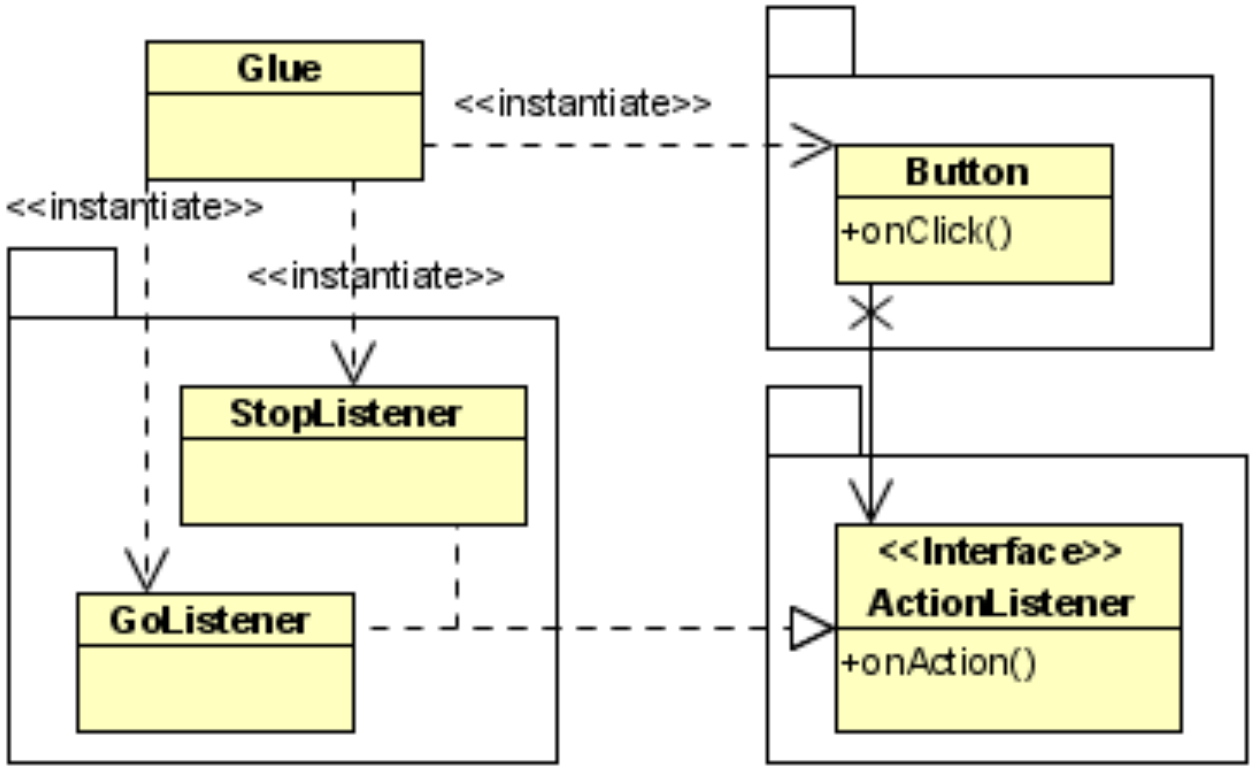
Glue layers

- When we invert dependence, the question arises of how the layers connect. E.g. where is client created.
- It could be created by the server, creating new dependence and a new kind of responsibility for the server.



Glue layer

- Alternatively we create a glue layer that plugs the parts together



Reflection: DIP and LSP

- The DIP emphasizes that there are two interfaces that a server class implements.
  a. The concrete interface that describes exactly what the class provides
  b. The abstract interface that describes what the client needs.
- This is similar to the LSP which emphasizes that a class implements
  i. The concrete interface that describes exactly what the class provides
  ii. An abstract interface that describes what all descendant classes (including self) are obligated to provide

Reflection: DIP and LSP

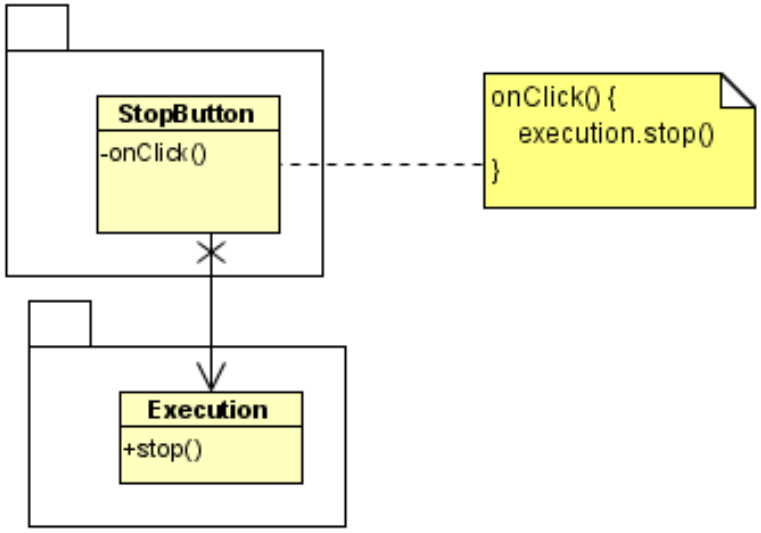
- The concrete interface is of interest to the creator of the object, as it is creating instances known to be of that specific class.
- The abstract interface is of interest to the client.
- The difference is that in the LSP we are considering one class which may be extended. Thus both interfaces must be documented in one class.
- With the DIP we are considering an <<interface>> and its realization. The abstract interface belongs to the <<interface>> while the concrete interface belongs to its realization.

Reflection: DIP and LSP

- It is generally a good idea to separate these two concerns as we have done. I.e. have two kinds of classes
  - **abstract classes** and **interfaces** exist to be extended.
  - **concrete classes** exist to be instantiated.
  - Avoid extending concrete classes.

Reflection: DIP and OCP

- The DIP supports the Open/Closed Principle.
- Consider our button example.
- Originally our button class is not open for extension, as it is coupled to one application



- After applying the DIP, the button class is open for extension, by plugging in various action listener objects.

In Summary

- The DIP makes clients reusable by abstracting the interface the client needs from a server from the server's implementation
- This protects the client's design from depending on incidental (as opposed to fundamental) aspects of its server
- Thus the DIP is good practice even when the client is not intended to be reused