The Observer Pattern

Context

- You have partitioned your program into separate objects

Problem

- A set of objects — the Observers — need to know when the state of another object — the *Observed Object* a.k.a. the *Subject* — changes.
- The Subject should be unaware of who its observers are, and, indeed, whether it is being observed at all.

Solution

- Define a one-to-many relation between the *subject* and a set of *dependent* objects (the *observers*).
- The dependents register themselves with the subject.
- When the subject changes state, it notifies all of its dependents of the change.

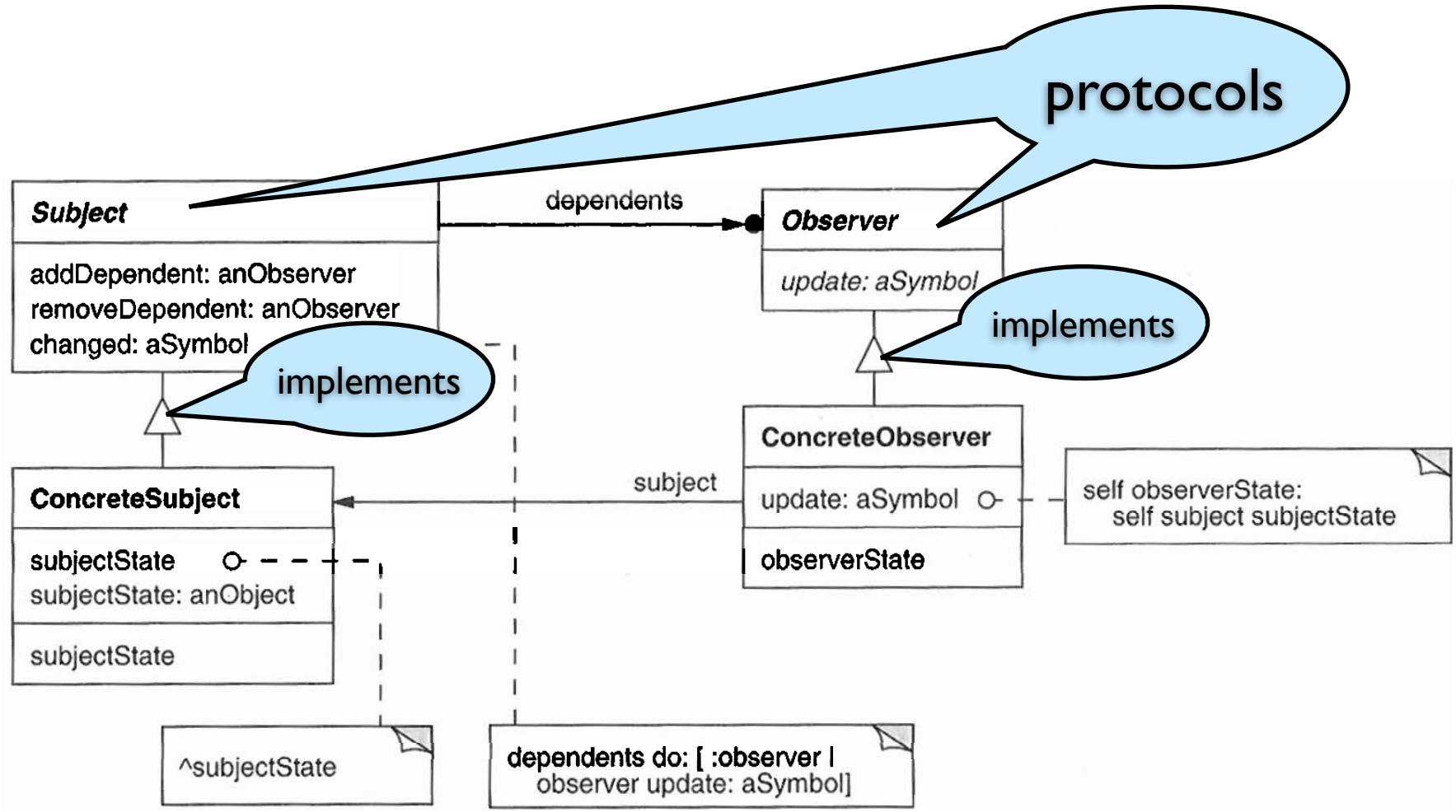

Figure from Alpert, page 305

- O-O solutions break the problem into small pieces — objects
  - + Each object is easy to implement and maintain
  - + Objects can be re-combined in many ways to solve a variety of problems
  - – Many simple behaviors will require the collaboration of multiple objects
  - – Unless the collaboration is “at arms length”, the benefits of the separation will be lost.
- The observer patterns implements this “arms length” collaboration
  - it’s key to the successful use of objects

Two Protocols

- The subject protocol
  - Used by the subject when its state changes
- The observer protocol
  - Used to tell the observer about a change in the subject
- *Both* implemented in class Object
  - So every Smalltalk object can be a subject, or an observer, or both.

Pharo Implementation

Subject messages	self changed self changed: anAspectSymbol self changed: anAspectSymbol with: aParameter
Dependent messages	aDependent update: mySubject aDependent update: anAspectSymbol aDependent update: anAspectSymbol with: aParameter

Managing dependencies

Subject messages	aSubject addDependent: aDependent aSubject removeDependent: aDependent

- Dependents are stored in a collection, accessed through the message myDependents
- In class Object, the collection is stored in a global dictionary, keyed by the identity of the subject:

```
myDependents: aCollectionOrNil
        aCollectionOrNil
            ifNil: [DependentsFields removeKey: self ifAbsent: []]
            ifNotNil: [DependentsFields at: self put: aCollectionOrNil]
```

- In class Model, the collection is an instance variable:

```
myDependents: aCollectionOrNil
        dependents := aCollectionOrNil
```

Explicit Interest

Context:

- The subject's state requires significant calculation — too costly to perform unless it is of interest to some observer

Problem:

- How can the subject know whether to calculate its new state?

Solution

- Have the observers declare an *Explicit Interest* in the subject
- observers must retract their interest when appropriate

Explicit Interest *vs.* Observer

Intent:

- Explicit interest is an optimization hint; can always be ignored
- Observer is necessary for correctness; the subject has the *responsibility* to notify its observers

Architecture

- Explicit interest does not change the application architecture
- Observer does

Who and What

- Explicit interest says *what* is interesting, but not *who* cares about it
- Observer says *who* cares, but not *what* they care about.

Further Reading

- The Explicit Interest pattern is described by Vainsencher and Black in the paper "*A Pattern Language for Extensible Program Representation*", Transactions on Pattern Languages of Programming, Springer LNCS 5770