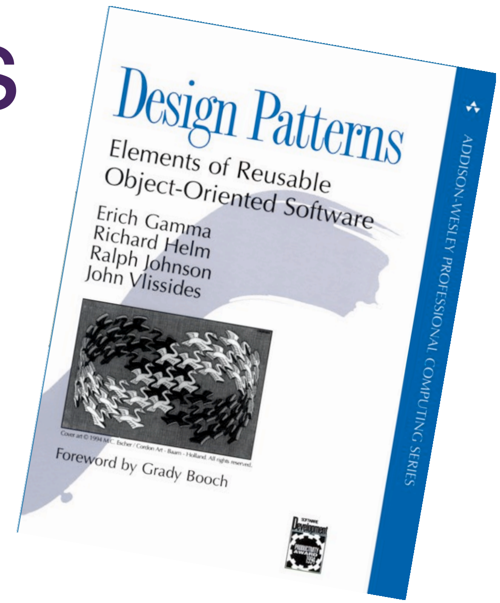
Introduction to Design Patterns

Design Patterns

Elements of Reusable Object-Oriented Software



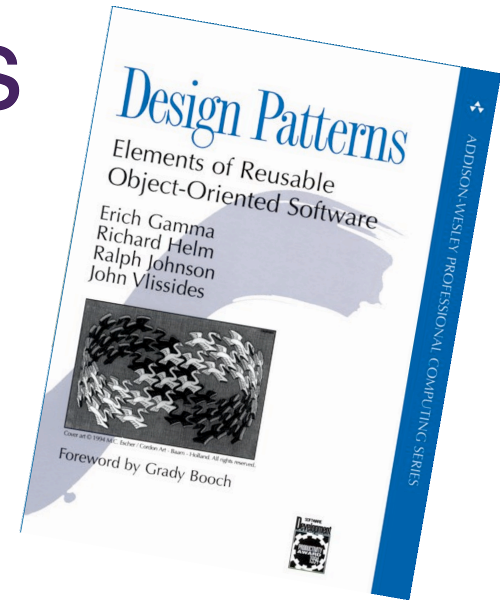
by

Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides

Addison-Wesley, 1995.

Design Patterns

Elements of Reusable Object-Oriented Software



by

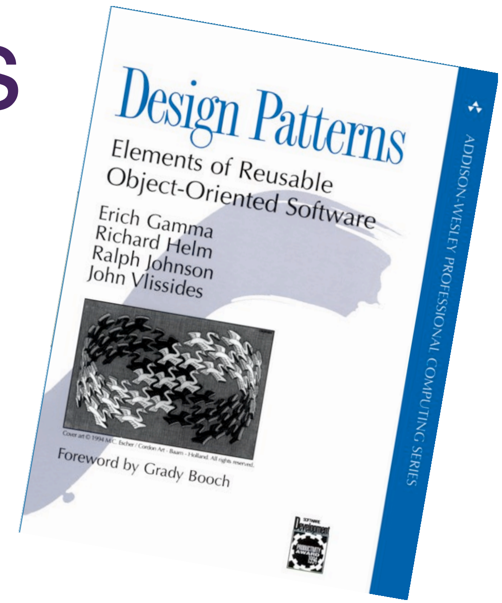
Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides

The Gang of Four

Addison-Wesley, 1995.

often called
the "Gang of Four"
or GoF book

Design Patterns



Elements of Reusable Object-Oriented Software

by

The Gang of Four

Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides

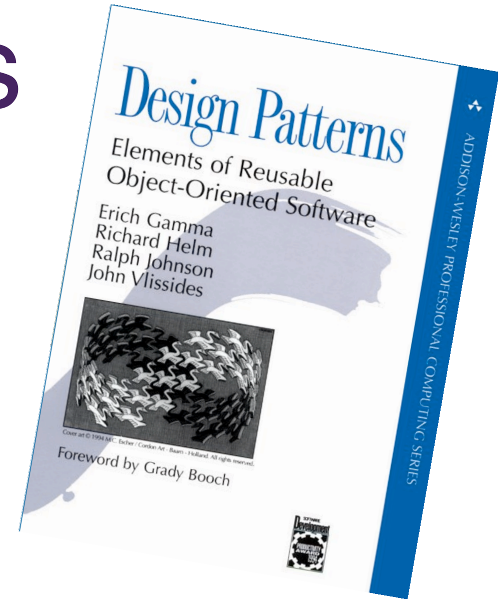
Addison-Wesley, 1995.

original, well-known book introducing design patterns

often called the "Gang of Four" or GoF book

Design Patterns

Elements of Reusable Object-Oriented Software



by

The Gang of Four

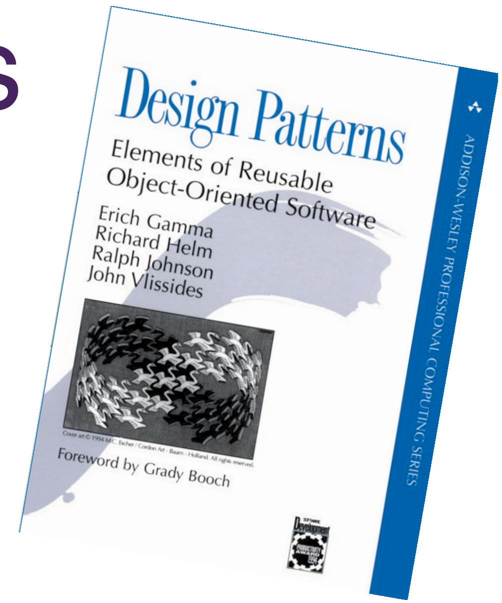
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Addison-Wesley, 1995.

original, well-known book introducing design patterns

often called the "Gang of Four" or GoF book

Design Patterns



Examples presented in C++ (and Smalltalk)

Elements of Reusable Object-Oriented Software

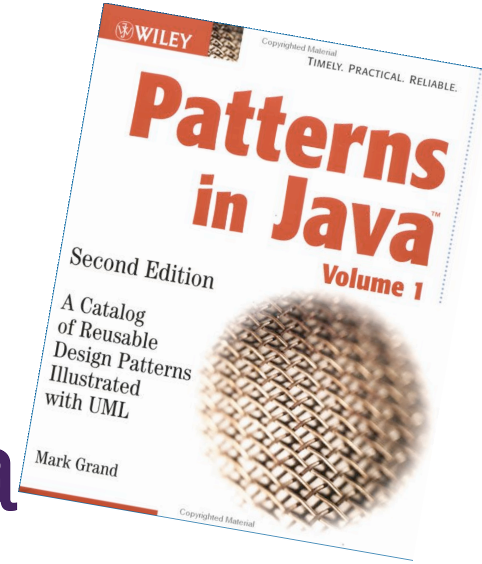
by

The Gang of Four

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Addison-Wesley, 1995.

same
design patterns
as the GoF
but with a little
bit of refactoring



Patterns in Java Volume 1

A Catalog of Reusable Design Patterns Illustrated with UML

by
Mark Grand

Wiley, 1998.

Not highly recommended

another resource…
follows GoF
book format

The Design Patterns Smalltalk Companion

by
Sherman R. Alpert, Kyle Brown, Bobby Woolf
Foreword by Kent Beck

Addison-Wesley, 1998.

Portland State University

another resource…
follows GoF
book format

The Design Patterns Smalltalk Companion

by
Sherman R. Alpert, Kyle Brown, Bobby Woolf
Foreword by Kent Beck

Addison-Wesley, 1998.

A great book!

Design Patterns in Java

by

Steven John Metsker
and William C. Wake

Portland State University

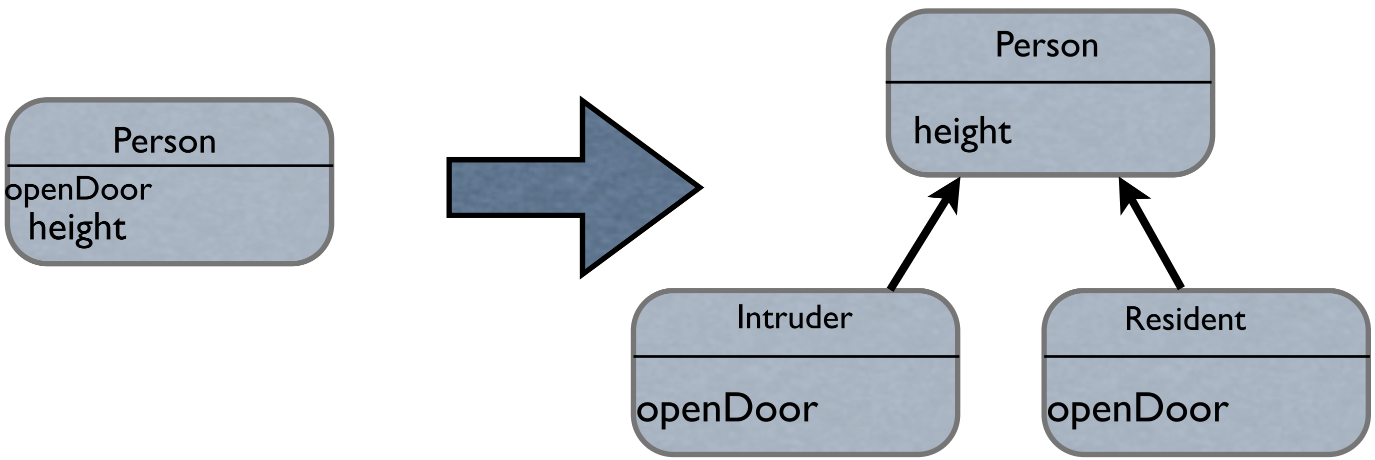
Why do patterns help in the Test–Code–Refactoring Cycle?

- When you are faced with a problem for which you don't have an obvious solution:
  - Design patterns may give you a design solution
    - that you can use "off the shelf", or
    - that you can adapt
  - Design patterns give you an implementation of that solution in your current language
  - Design patterns save you from having to think!
- Don't use a design pattern if you don't have a problem!

Revisit **Problem** from Monday…

```
Person ›› openDoor
    self isIntruder ifTrue: [ … ].
    self isResident ifTrue: [ … ].
    …
```

- On Monday I told you to refactor the class hierarchy:



Portland State University

How many occurrences of

```
Person ›› openDoor
    self isIntruder ifTrue: [ … ].
    self isResident ifTrue: [ … ].
    …
```

How many occurrences of

```
Person ›› openDoor
    self isIntruder ifTrue: [ … ].
    self isResident ifTrue: [ … ].
    …
```

are needed to prompt this refactoring?

0 ?

1 ?

2 ?

3 ?

Use patterns pro-actively?

- Hot Spots and Cold Spots
  - Rebecca Wirfs-Brock and others recommend that you identify which of your Classes are hot spot cards and which are cold spot cards

    hot = responsibilities very likely to change
    cold = responsibilities not very likely to change

- Hot spots are candidates for patterns!

Common Causes of Redesign

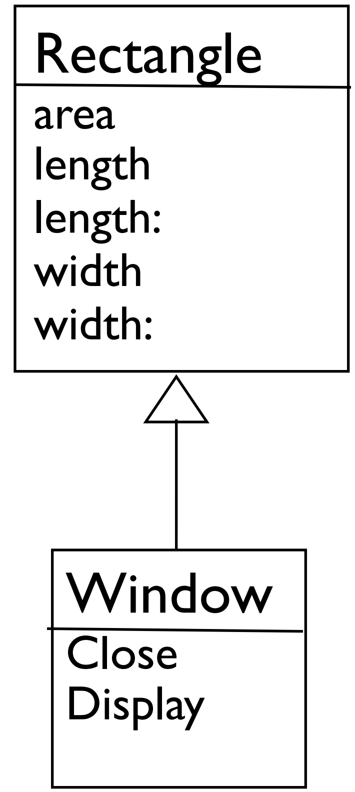
- Creating an object by specifying a class explicitly
  - CourseOffering new
- Depending on specific operations of someone else's object
  - student address line2 zipcode
- Dependence on object representations or implementations

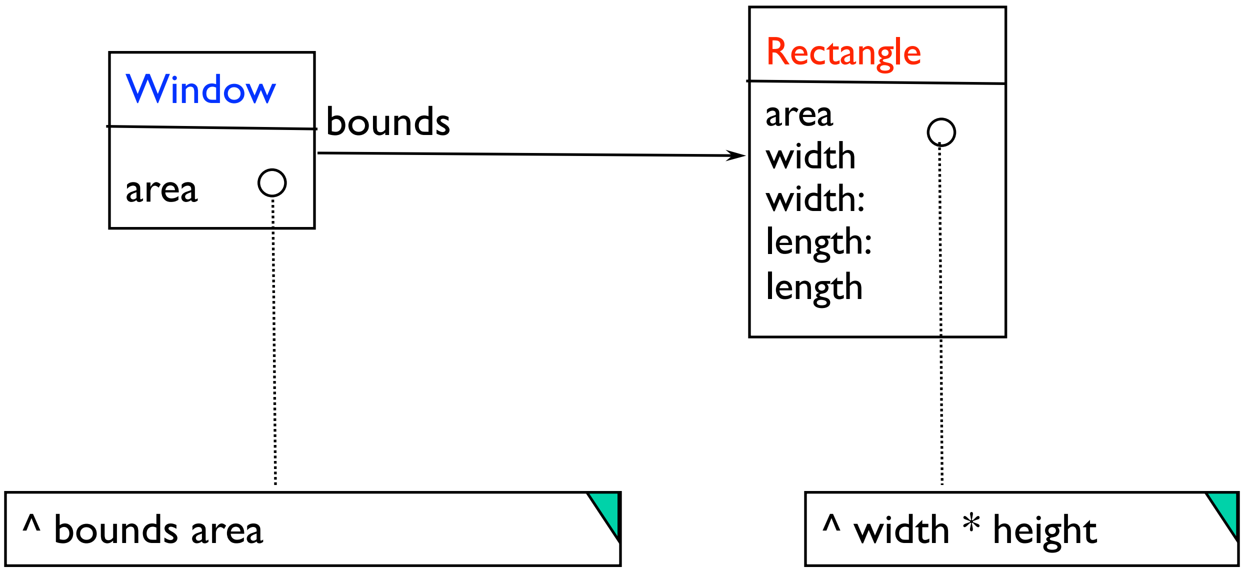
In general: information in more than one place

Advice from the Gang of Four

- Program to an interface, not an implementation
  - depend on the behavior of another object, not on its class
- Favor object composition (delegation) over class inheritance
- Encapsulate the concept that varies
  - once you know that it varies

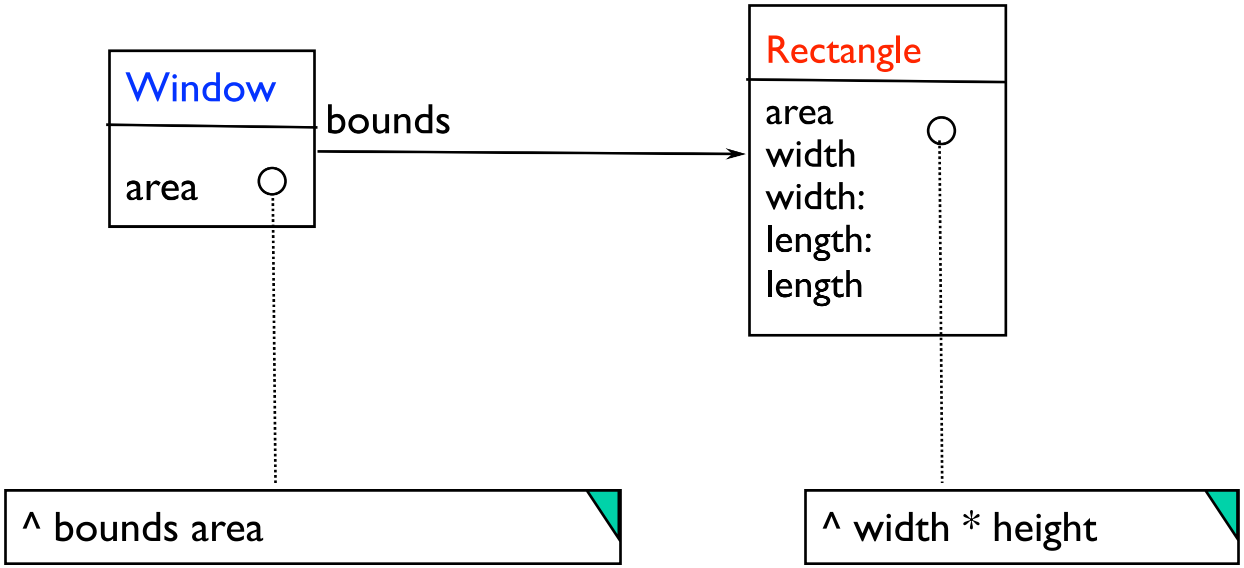
Misuse of Inheritance

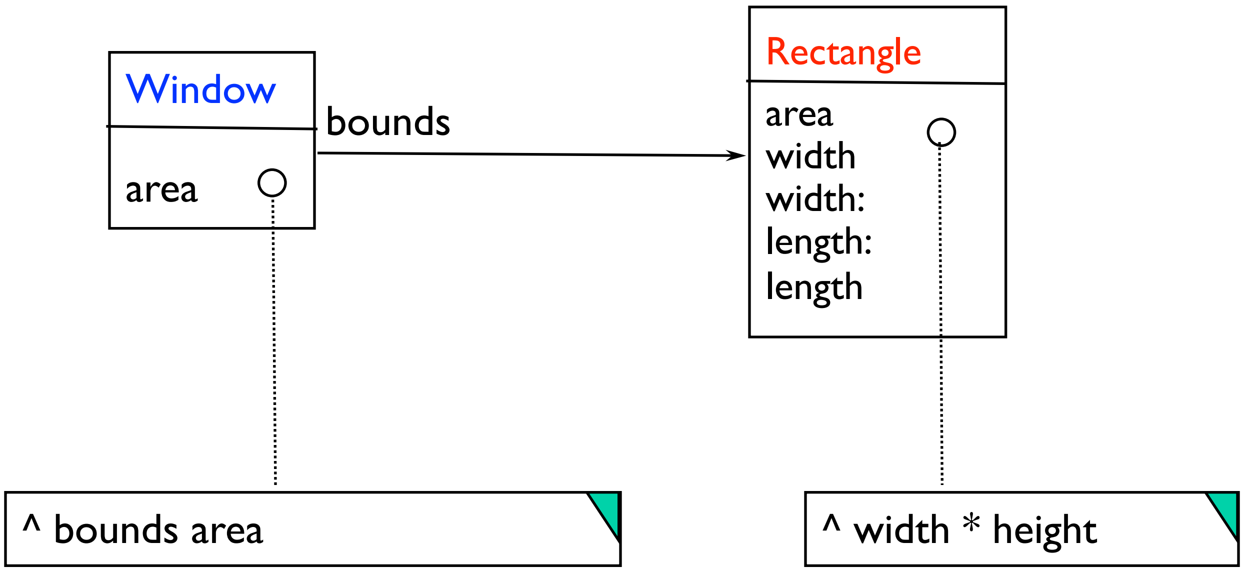




Example of delegation

Now we have two objects:
a Window object and a Rectangle object


Window
area
bounds
Rectangle
area
width
width:
length:
length
^ bounds area
^ width * height



Let a window HAVE a rectangle (as a bounding box) rather than BE a rectangle (through inheritance)

If bounding "box" becomes a polygon...then Window would just HAVE a polygon

Design Patterns provide ...

- abstractions for reasoning about designs
- a common design vocabulary
- a documentation and learning aid
- the experience of experts,
  - *e.g.*, to identify helper objects
- easier transition from design to implementation

A pattern has four essential elements:

- pattern name — to describe a design problem, it's solution, and consequences
- problem — to describe when to apply the pattern.

  it may include a list of conditions that must be true to apply the pattern
- solution — to describe the elements that make up the design, their relationships, responsibilities, and collaborations
- consequences — the results and trade-offs of applying the pattern

Design Patterns Categorized

Purpose

Scope

	Creational	**Structural**	**Behavioral**
class	factory method	adapter	interpreter template method
object	abstract factory builder prototype singleton	adapter bridge composite decorator façade flyweight proxy	chain of responsibility command iterator mediator memento observer state strategy visitor

The Singleton Pattern

The Singleton Pattern

Intent:

– Ensure that a class has a small fixed number of instances (typically, a single instance).
– Provide a global point of access to the instances

Motivation:

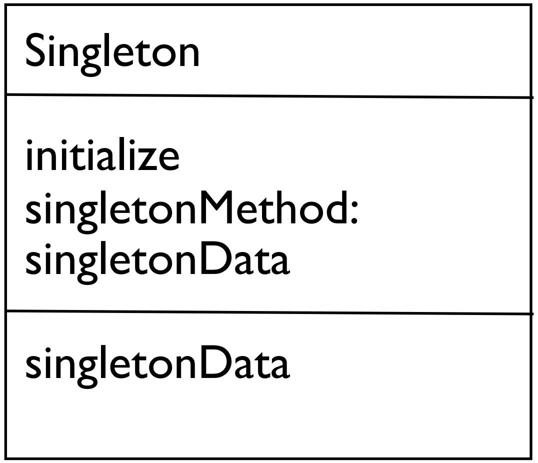
– Make sure that no other instances are created.
– Make the class responsible for keeping track of its instance(s)

Applicability:

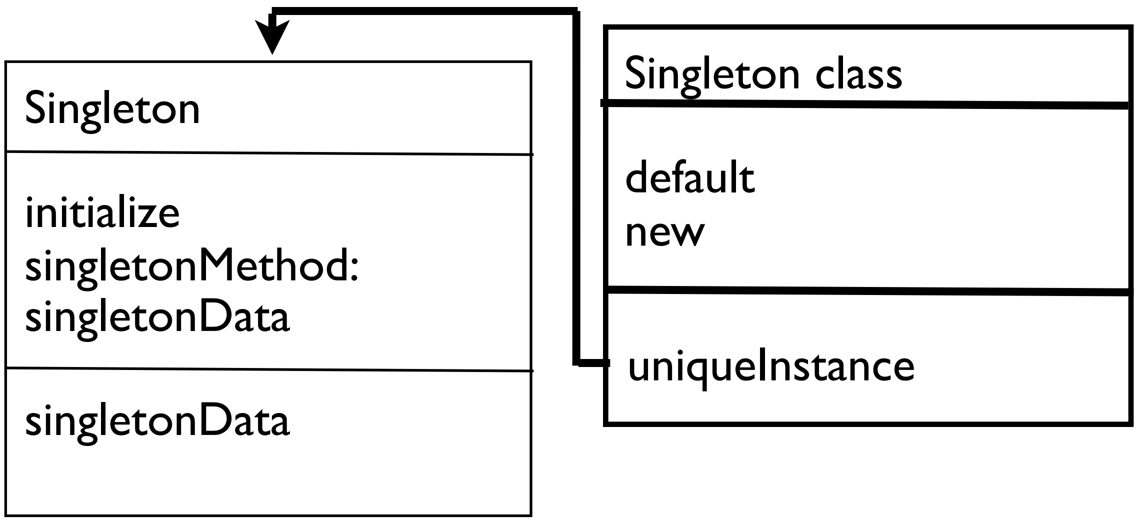
– When the instance must be globally accessible
– Clients know that there is a single instance (or a few special instances).

Structure of the Singleton Pattern

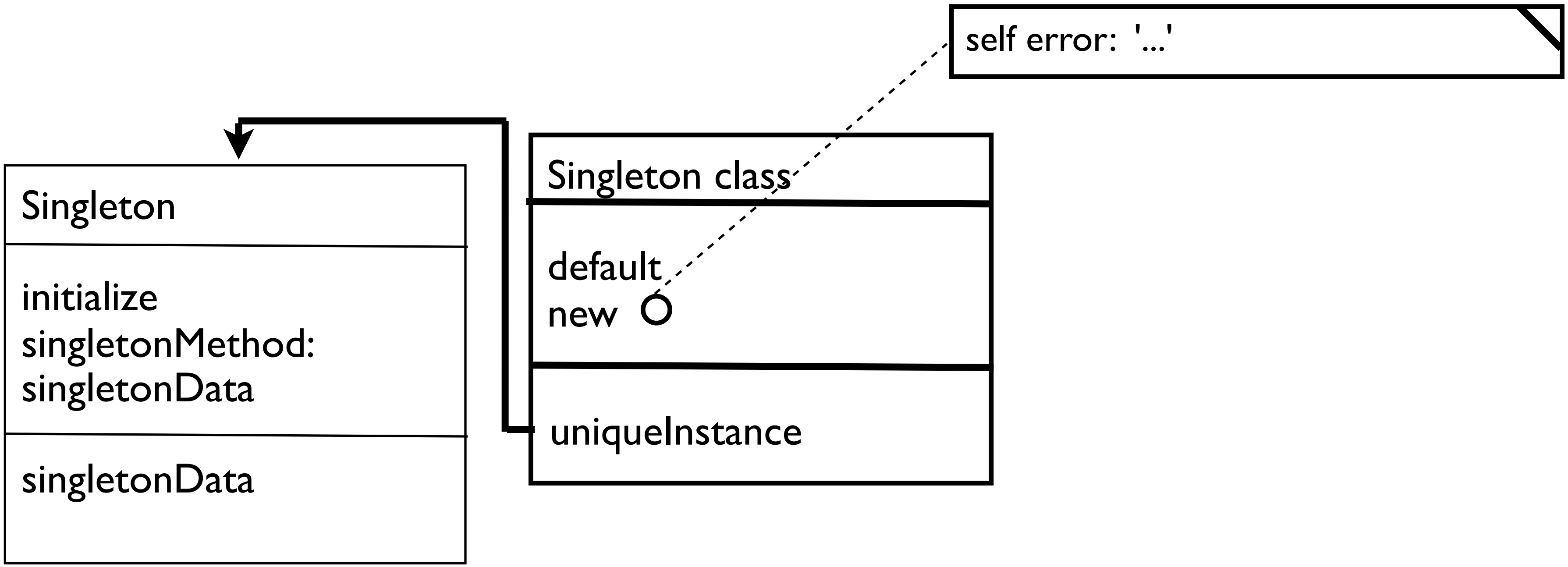
Structure of the Singleton Pattern



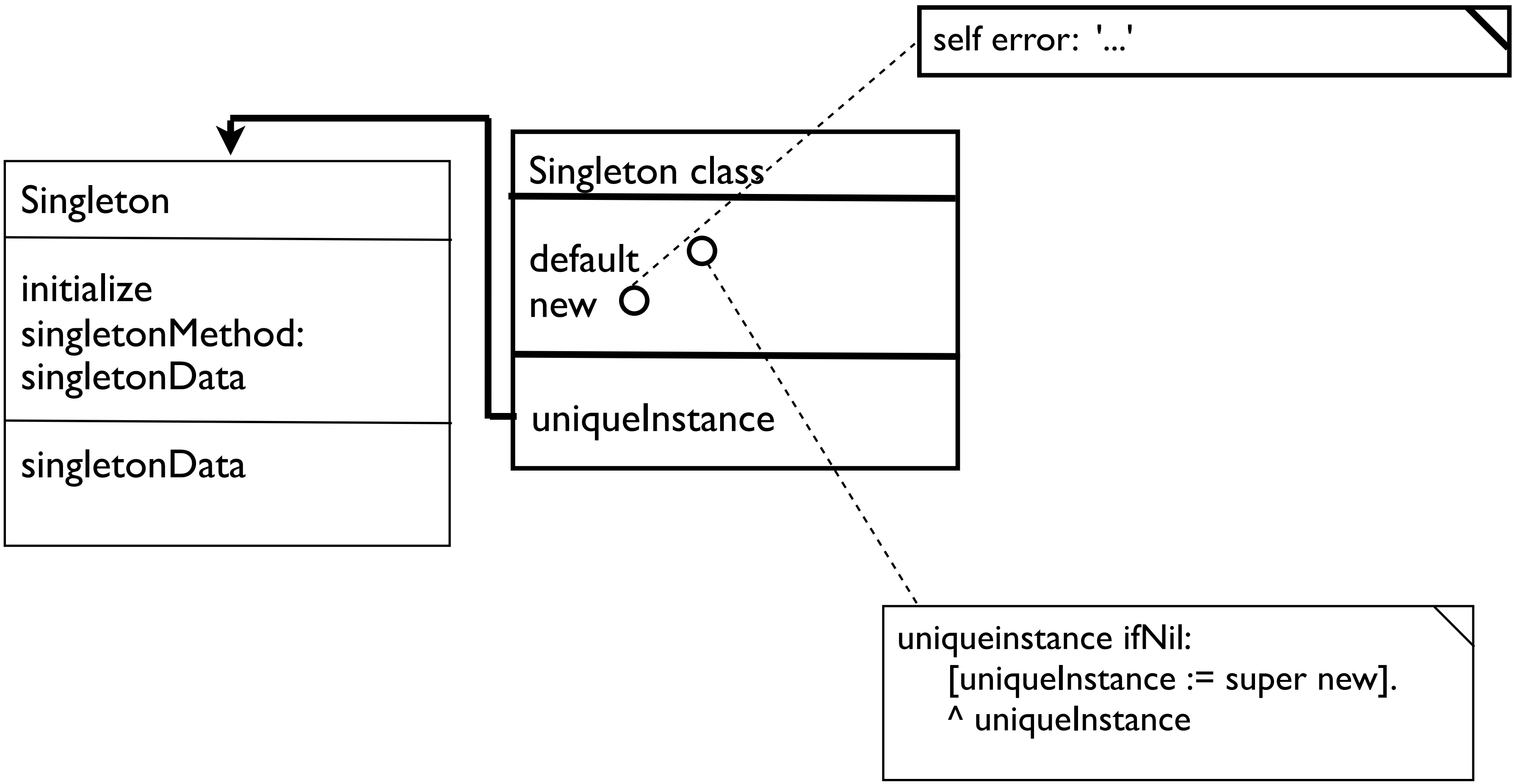
Structure of the Singleton Pattern



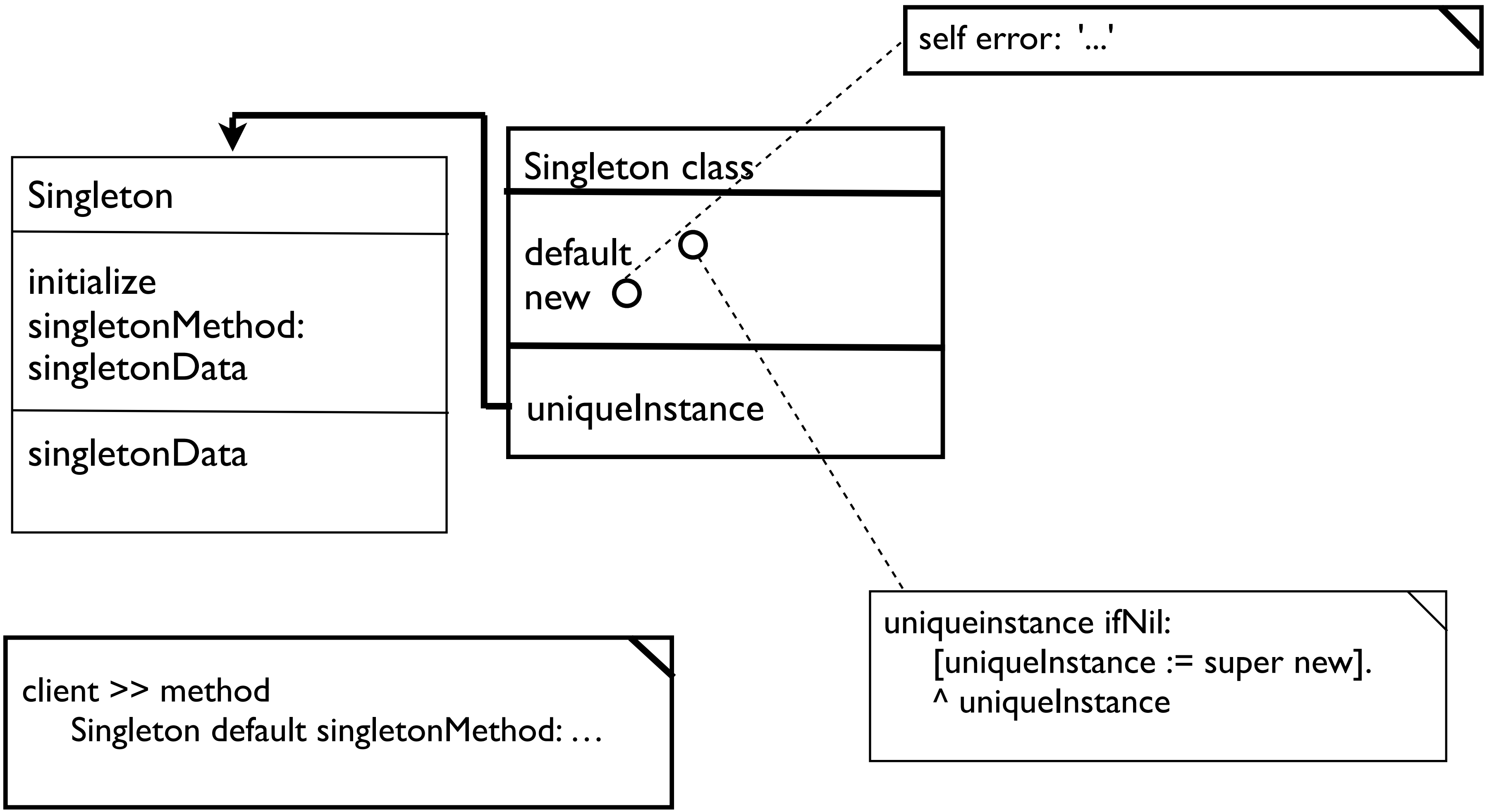
Structure of the Singleton Pattern



Structure of the Singleton Pattern



Structure of the Singleton Pattern



The Singleton Pattern

Participants: Singleton class
- defines a *default* method
  - is responsible for creating its own unique instance and maintaining a reference to it
- overrides "new"

Singleton
- the unique instance
- overrides "initialize"
- defines application-specific behavior

Collaborations: Clients access singleton sole through Singleton class's *default* method
- may also be called "current", "instance", "uniqueInstance" …

The Singleton Pattern

Consequences: Controlled access to instance(s)

Reduced name space (no need for global variable)

Singleton class could have subclasses
similar but distinct singletons

pattern be adapted to limit to a specific number of instances

Smalltalk Implementation

In Smalltalk, the method that returns the unique instance is implemented as a class method on the Singleton class. The new method is overridden.

uniqueInstance is a *class instance variable*, so that if this class is ever subclassed, each subclass will have its own uniqueInstance.

```
Object subclass: #Singleton
              instanceVariableNames: ''
              classVariableNames: ''
              poolDictionaries: ''

       Singleton class
              instanceVariableNames: 'uniqueInstance'
```

The Singleton Pattern: Implementation

```
Singleton class>>new
        "Override the inherited #new to ensure that there
        is never more than one instance of me."
        self error: 'Class ', self name,
        ' is a singleton; use "', self name,
        ' default" to get its unique instance'

Singleton class>>default
        "Return the unique instance of this class; if it hasn’t
         yet been created, do so now."

        ^ uniqueInstance ifNil: [uniqueInstance := super new]

Singleton>>initialize
        "initialize me"
        ...
```

Proxy

The Proxy Pattern

Intent: provide a surrogate or placeholder for another object (the RealSubject), to provide or control access to the object

Motivation: The RealSubject might be on disk, or on a remote computer. It might be expensive or undesirable to reify it in the Smalltalk image.

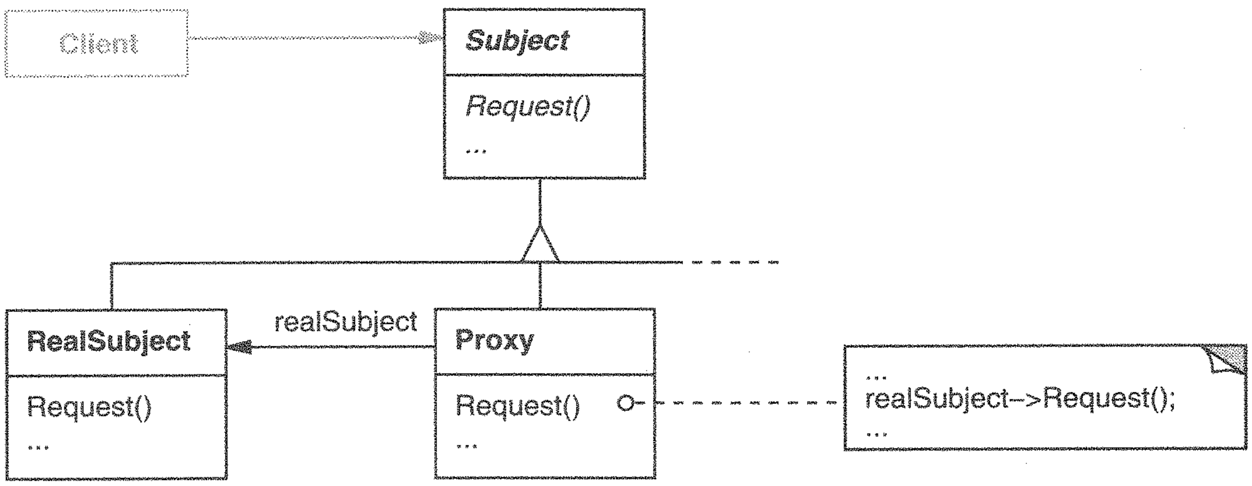
The RealSubject might understand some messages that the client is not authorized to send.

The Proxy Pattern

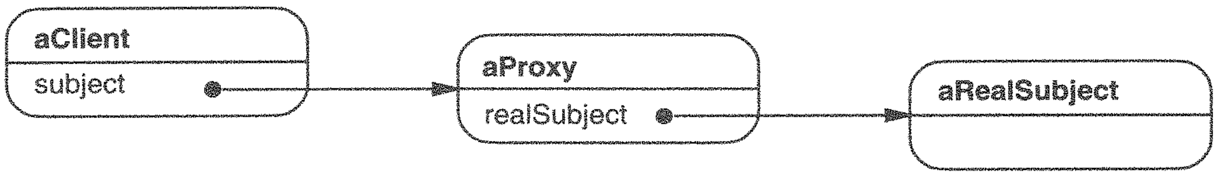
Solution The Proxy object forwards some or all of the messages that it receives to the RealSubject. It might encapsulate a network protocol, a disk access protocol, or a protection policy.

The Proxy and the RealSubject understand the same protocol, i.e., they have the same conceptual type. They have different implementations (different Classes).

Proxy Structure

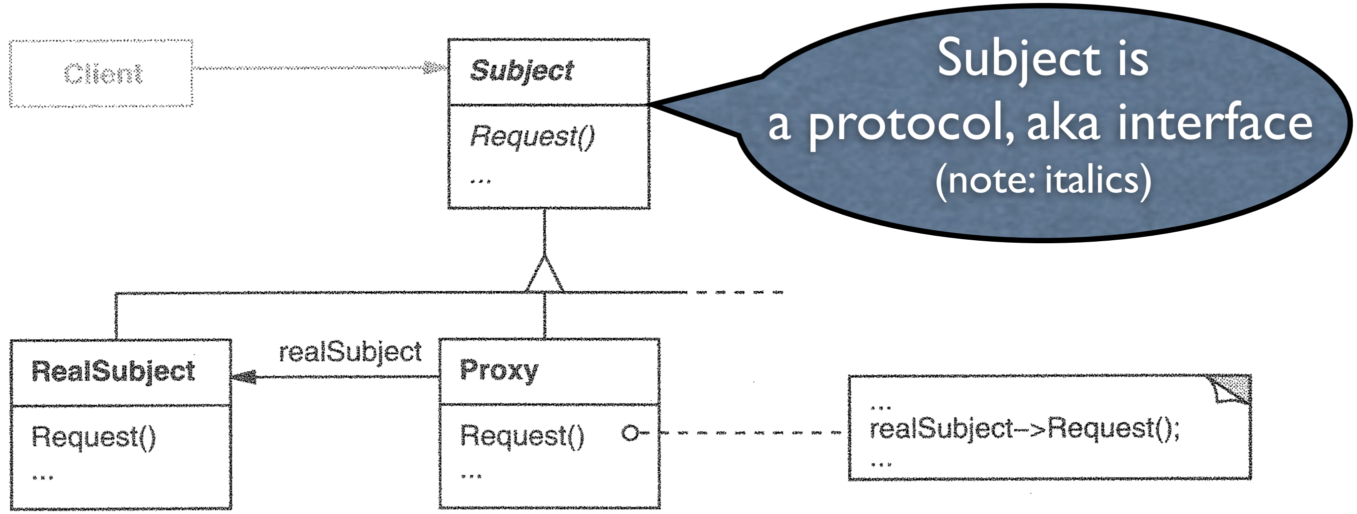


Here's a possible object diagram of a proxy structure at run-time:

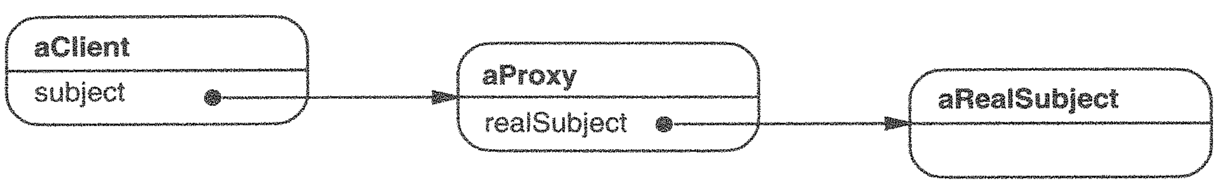


Portland State University

Proxy Structure



Here's a possible object diagram of a proxy structure at run-time:



Implementation

Two approaches

1. Implement all of Subject's methods in Proxy. A few of them will actually have code in Proxy, but most will just forward the message to RealSubject

```
aMessage: parameter
    ^ realSubject aMessage: parameter
```

2. Use dynamic binding. Don't write code for all of the messages that will be forwarded; instead, override the method for doesNotUnderstand:. For this to work, the Proxy should not inherit methods from its superclass.

ProtoObject

In Pharo, ProtoObject is the superclass of Object. It implements the methods that all objects really, really, really need to support.

Object methodDict size ==> 340.
ProtoObject methodDict size ==> 38.

If you use approach 2 to implement a Proxy, then it should subclass ProtoObject

Avoiding Forwarding

- Forwarding each message adds overhead. Interpreting messages in a doesNotUnderstand: method adds more overhead.
- Instead, the first message trapped by doesNotUnderstand: can replace the proxy by the realSubject.

```
doesNotUnderstand: aMessage
    realSubjectProtocol includes: aMessage ifFalse: […].
    self become: realSubject.
    ^aMessage sendTo: self
```

Example use: RemoteString

- In Visualworks Smalltalk, the RemoteString objects are proxies for text stored on the disk, such as a class comment or other piece of text in a file (such as the *sources* file).
- The actual String is created by the proxy on demand by reading the file.
- This is done by the doesNotUnderstand: method in the Proxy.

Iterator

- Iterator defines an interface for sequencing through the objects in a collection.
  - This interface is independent of the details of the kind of collection and its implementation.
- This pattern is applicable to any language

External Iterators

- In languages without closures, we are forced to use external iterators, *e.g.*, in Java:
  - aCollection.iterator() answers an iterator.
  - the programmer must explicitly manipulate the iterator with a loop using hasNext() and next()

Java test

- Given a collection of integers, answer a similar collection containing their squares:

your answer here ...

Internal Iterators

- Languages with closures provide a better way of writing iterators
- Internal Iterators encapsulate the loop itself, and the next and hasNext operations in a single method
- Examples: do:, collect:, inject:into:
  - look at the enumerating protocol in Collection

doing: Iterators for effect

For every (or most) elements in the collection, do some action

do: do:separatedBy: do:without:

- for keyedCollections

  associationsDo: keysDo: valuesDo:

- for SequenceableCollections

  withIndexDo: reverseDo: allButFirstDo:

mapping: create a new collection

- Create a new collection of the same kind as the old one, with elements in one-to-one correspondence
- For every element in the collection, create a new element for the result.

  collect: collect:thenDo: collect:thenSelect:

- for SequenceableCollections

  collect:from:to: withIndexcollect:

selecting: filtering a collection

- Create a new collection of the same kind as the old one, with a subset of its elements
- For every element in the collection, apply a filter.
- Examples:

select:	reject:
select:thenDo:	reject:thenDo:

partial do

- It's OK to return from the block that is the argument of a do:

```
coll do: [ :each | each matches: pattern ifTrue: [^ each]].
^ default
```

  - but consider using one of the "electing" iterators first!

```
coll detect: [ :each | each matches: pattern]
    ifNone: [default]
```

electing: picking an element

Choose a particular element that matches some criterion

- Criterion might be fixed:
  - max: min:
- or programmable:
  - detect: detect:ifNone:

Summarizing: answering a single value

- Answer a single value that tells the client something about the collection
  - allSatisfy: anySatisfy: detectMin: detectMax: detectSum:
  - sum inject: into:

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.

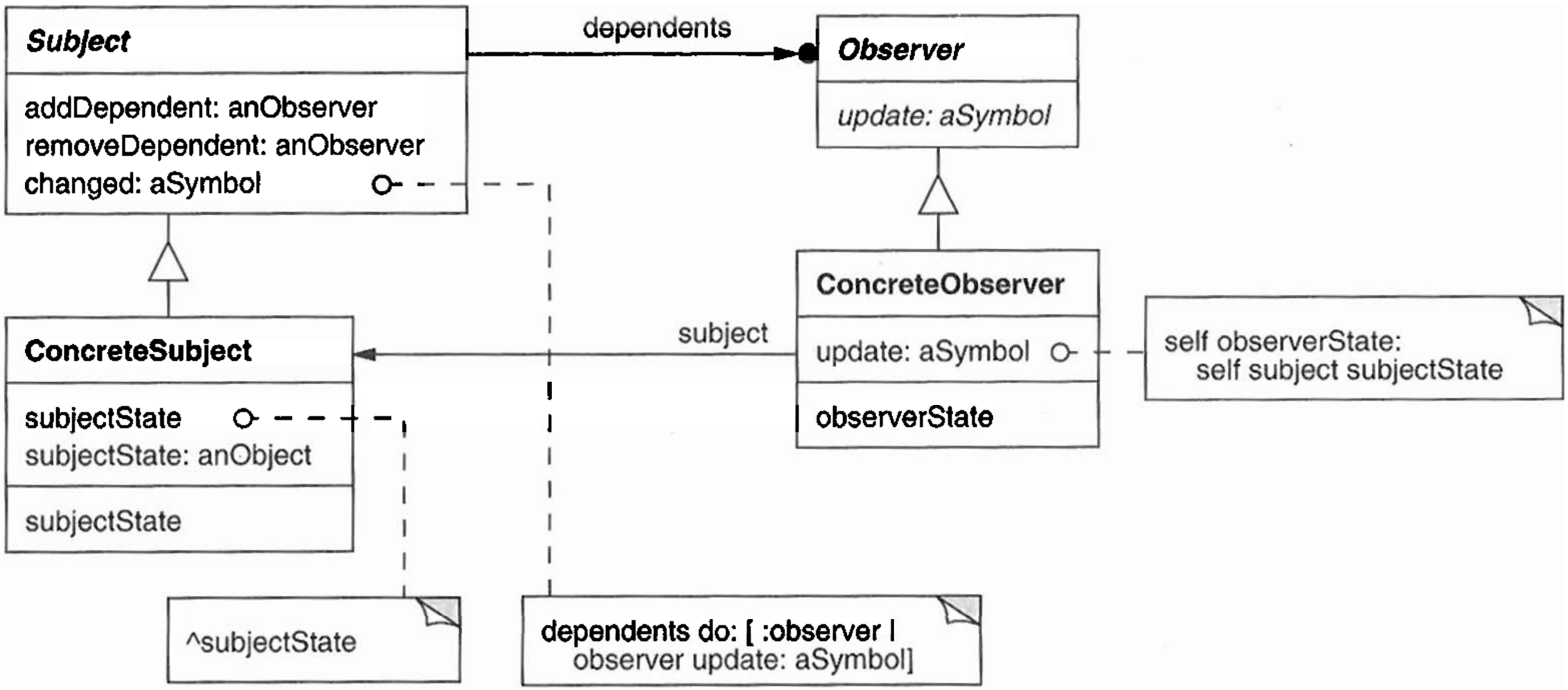


Figure from Alpert, page 305

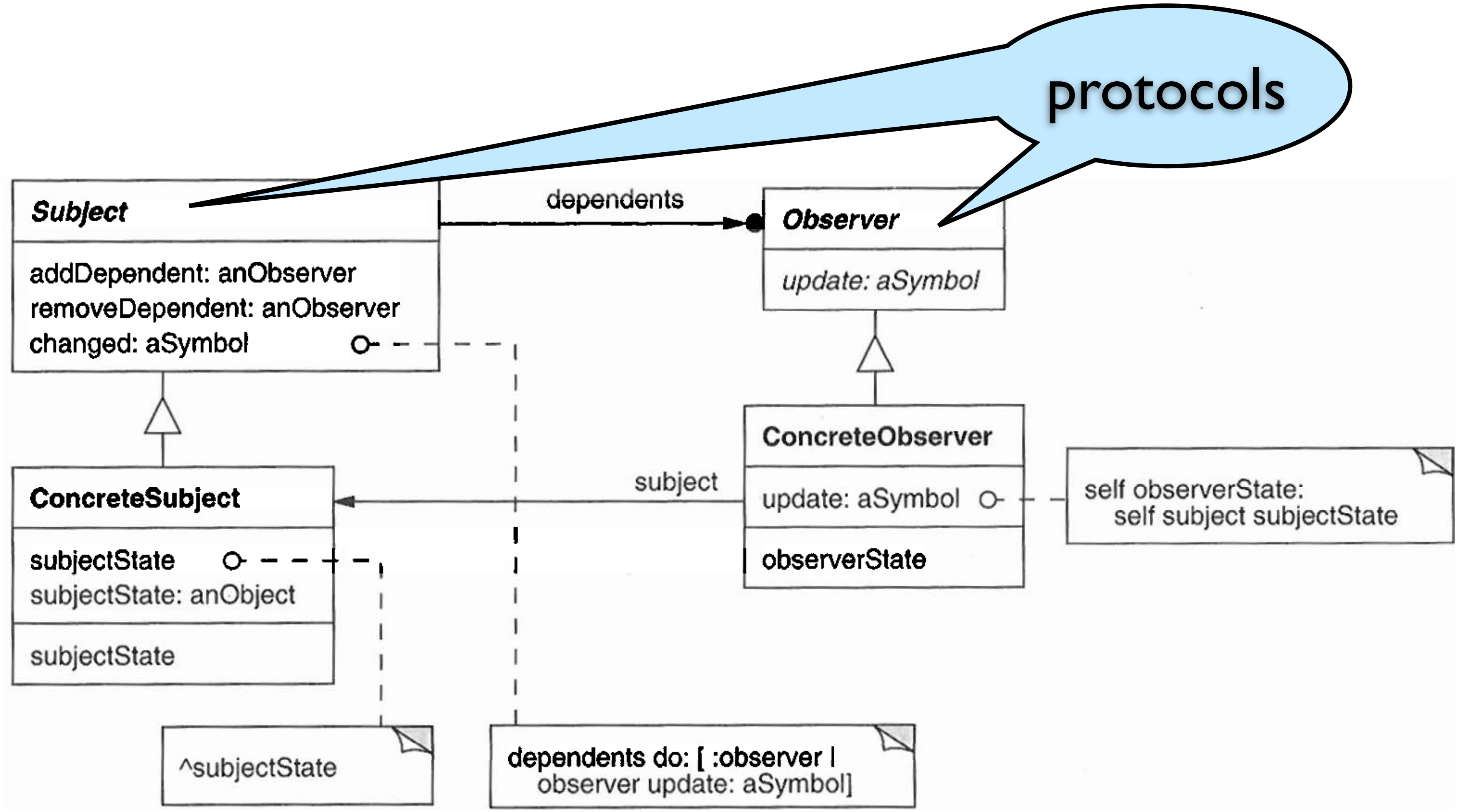


Figure from Alpert, page 305

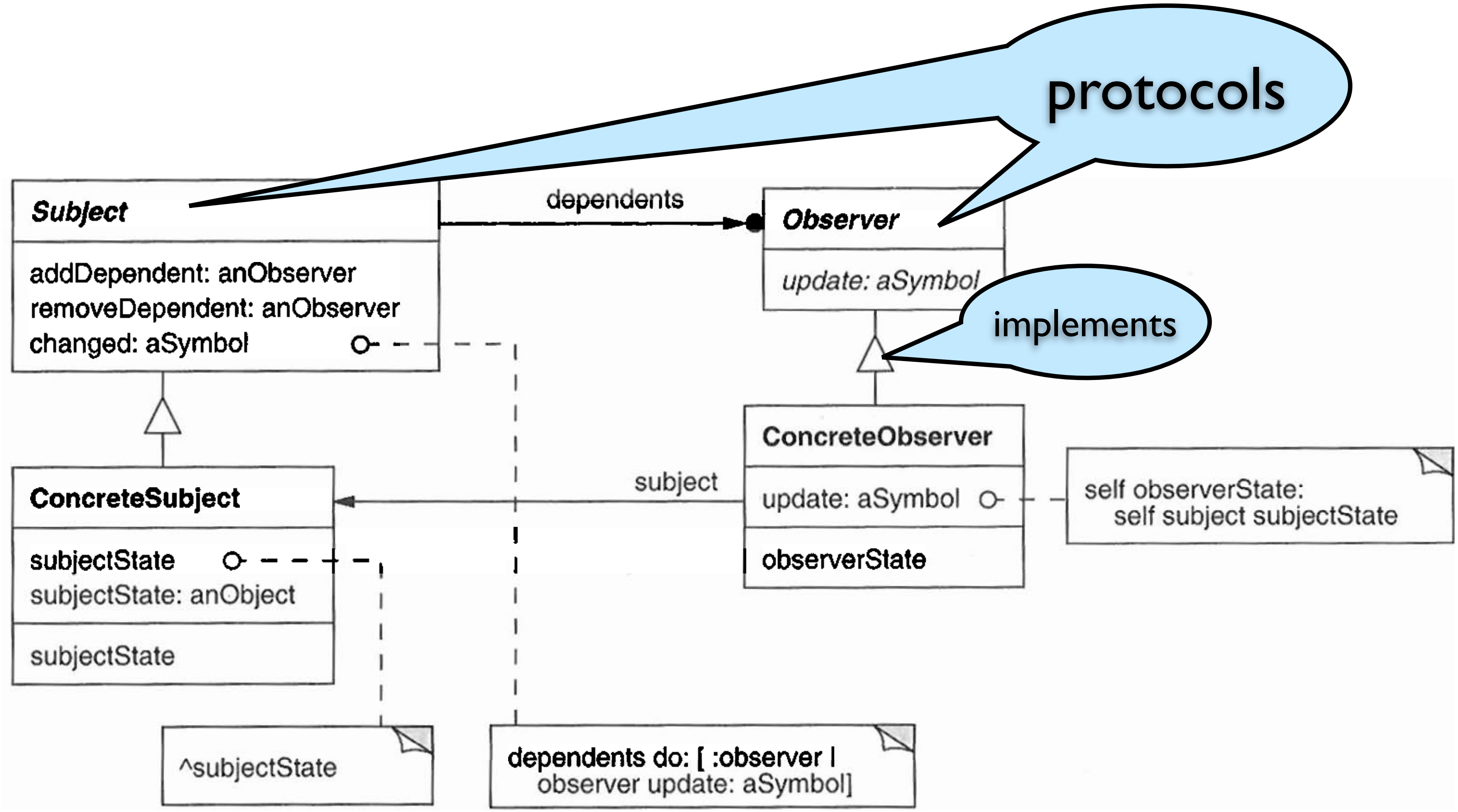


Figure from Alpert, page 305

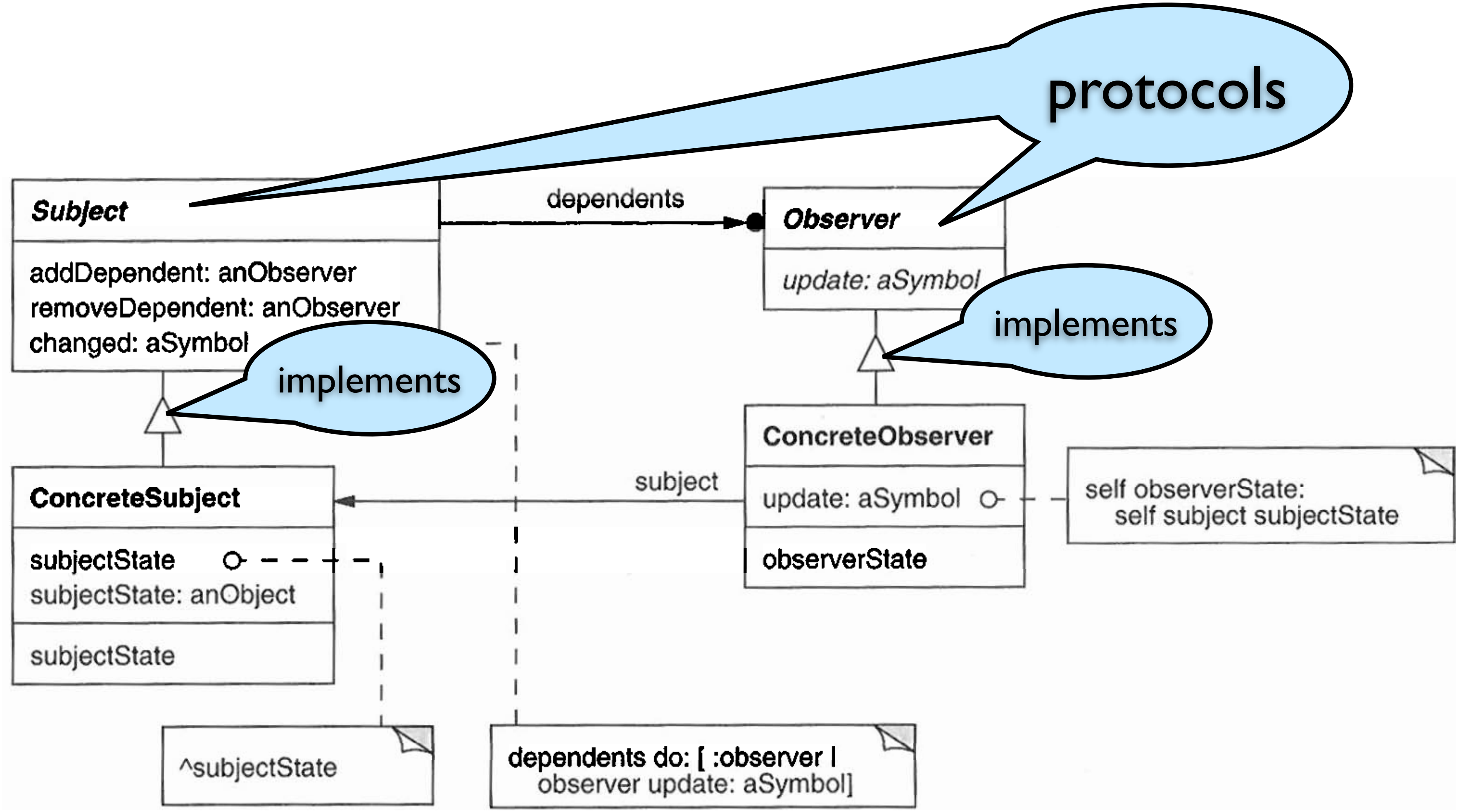


Figure from Alpert, page 305

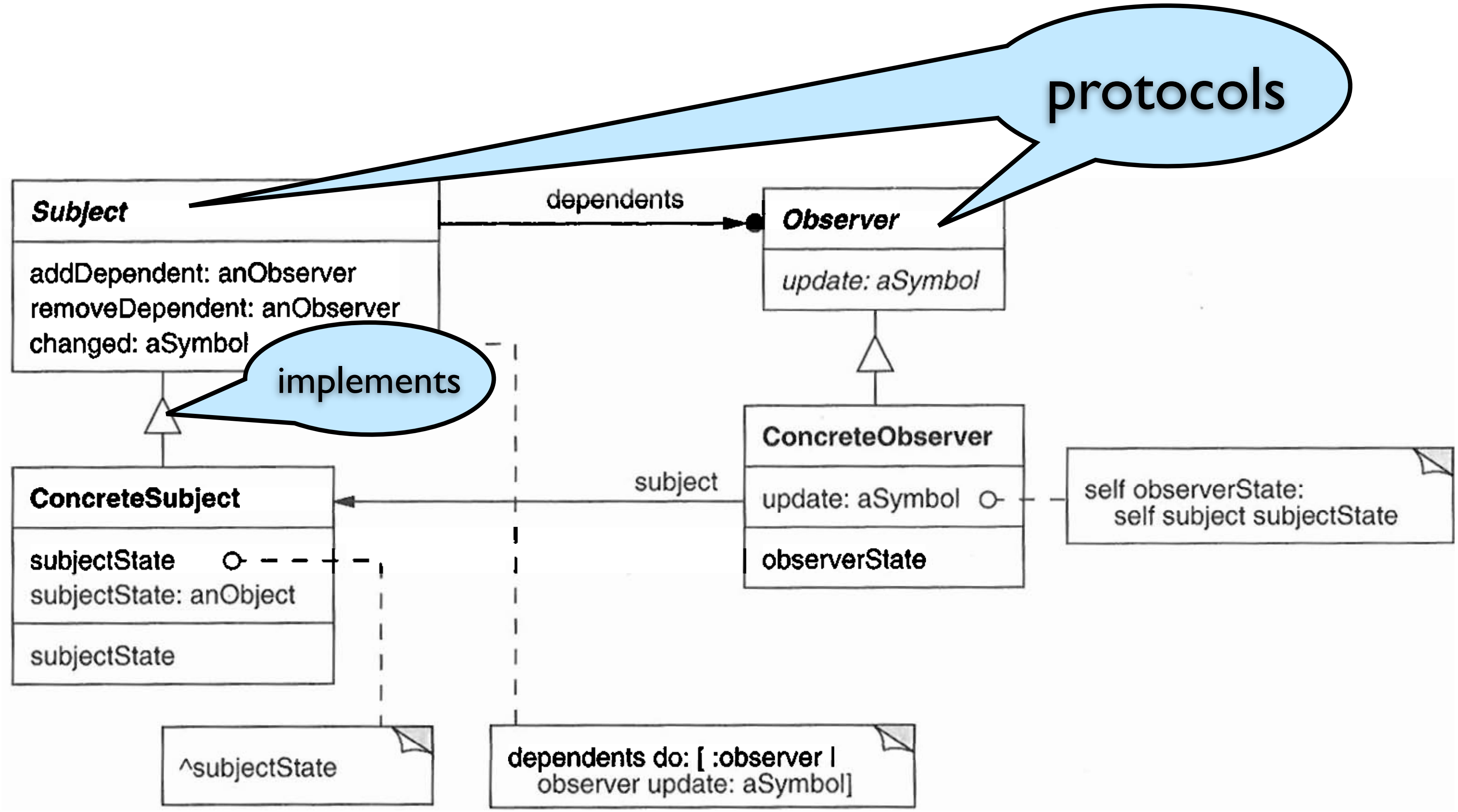


Figure from Alpert, page 305

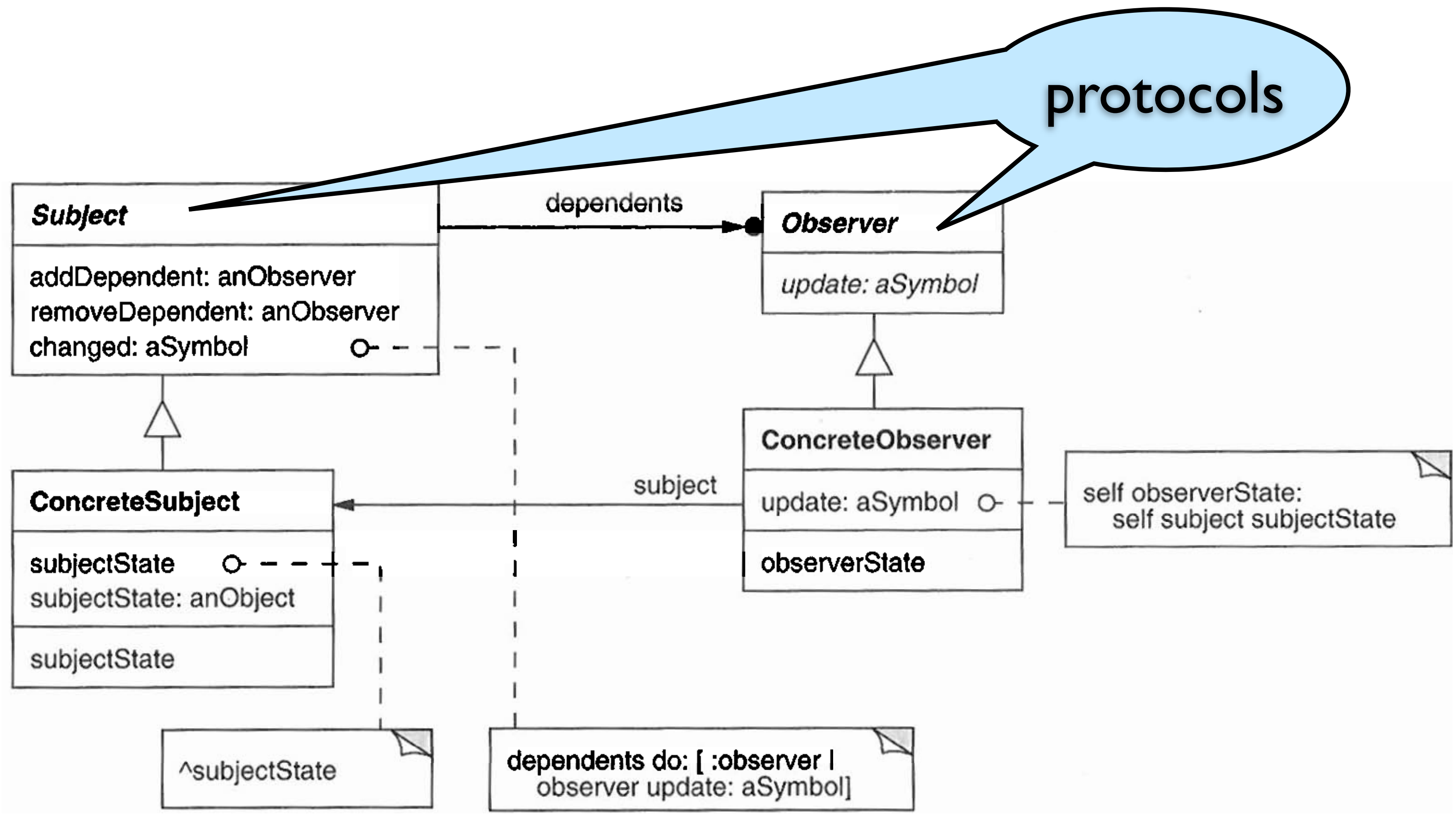


Figure from Alpert, page 305

- 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
```

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

The State Pattern

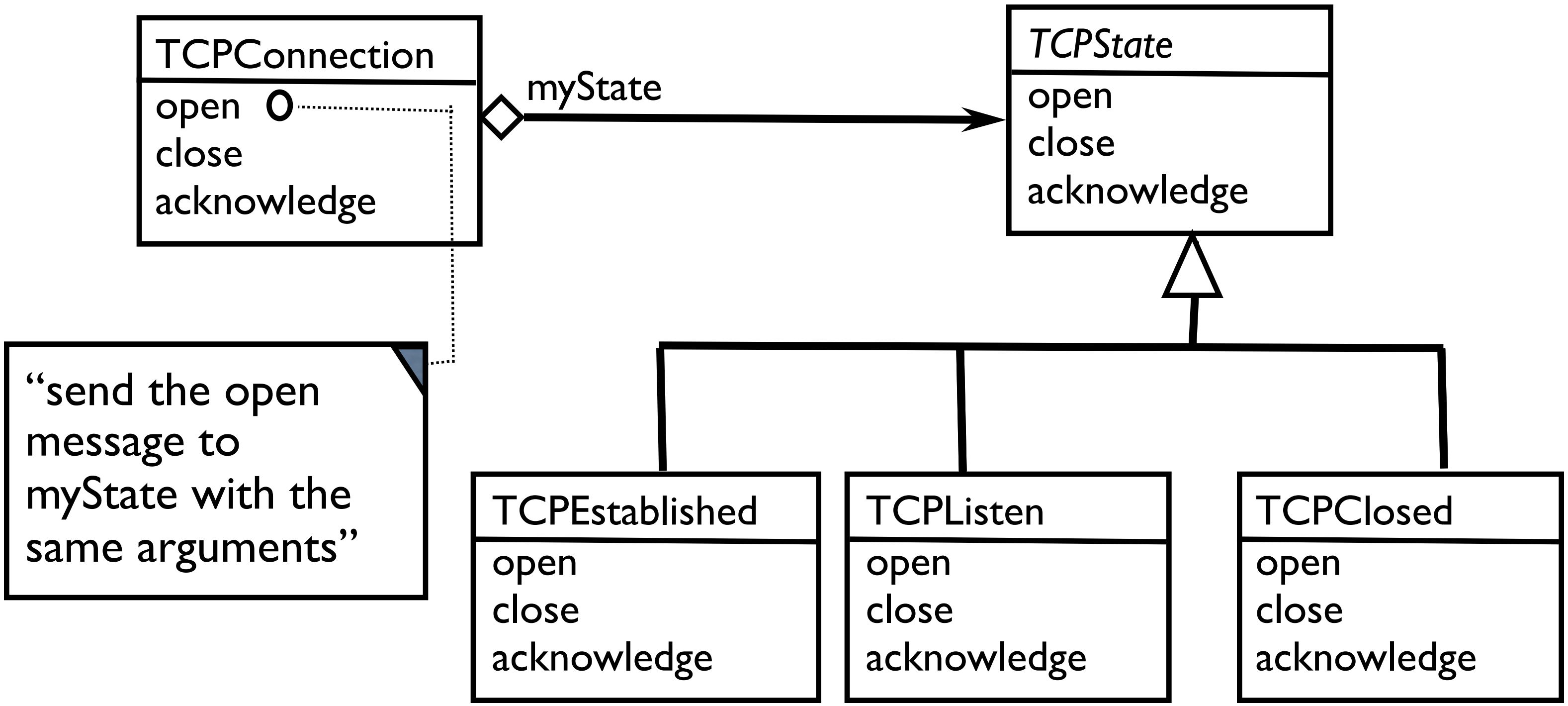
The State Pattern

Intent: Allow an object to alter its behavior when its internal state changes.
Object appears to change class.

Context: An object's behavior depends on its state, and it must change its state-dependent behavior at run-time

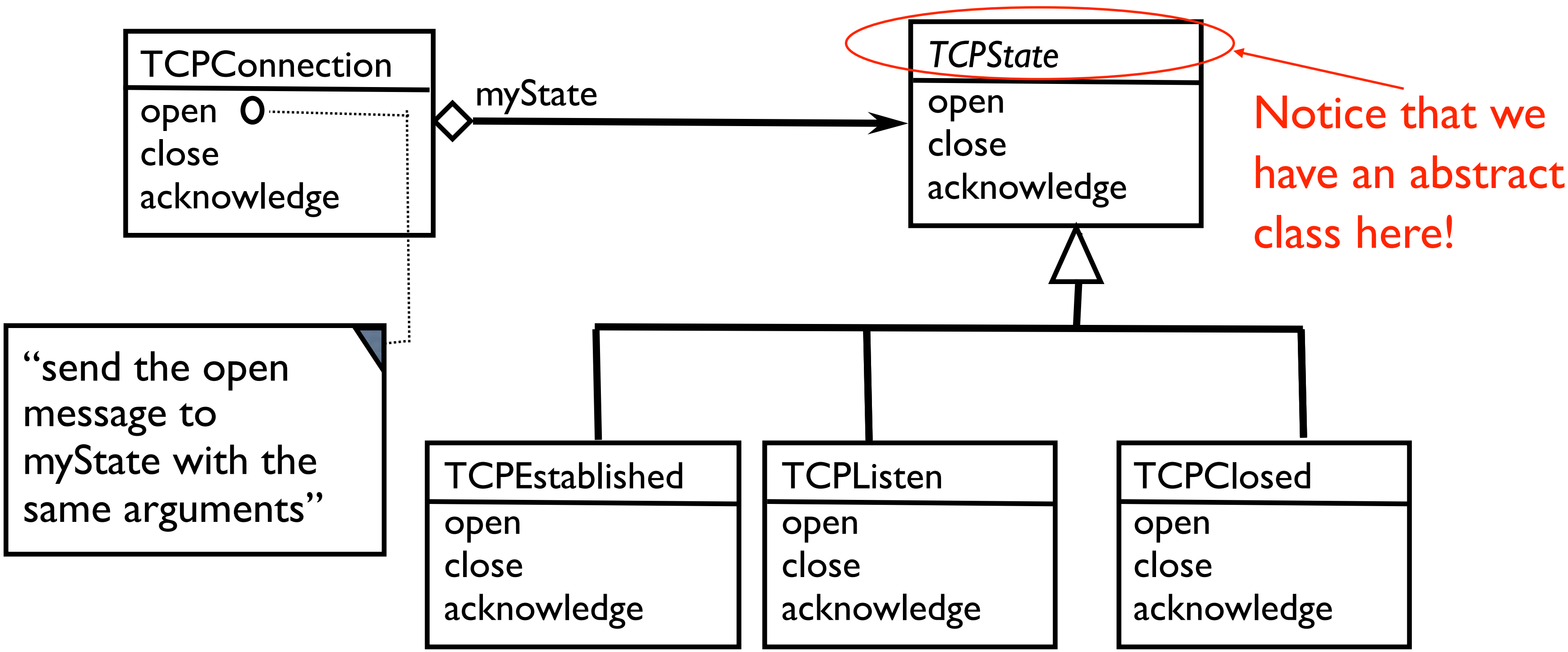
Problem: Operations have large, multi-part conditions that depend on the object's state

Example: Class that manages the state of a TCP/IP connection



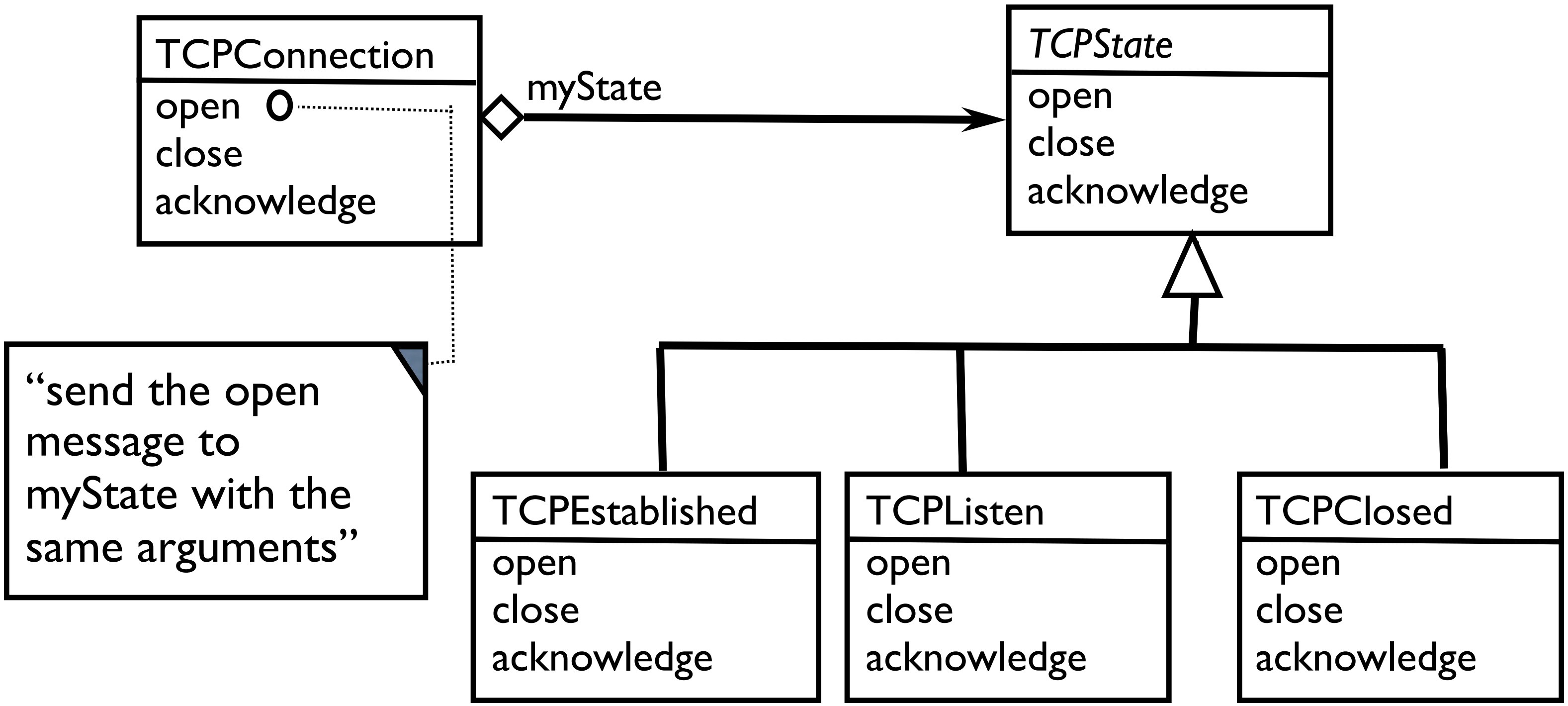
When the TCPConnection changes state, it simply replaces the myState object with an object of another state

Example: Class that manages the state of a TCP/IP connection



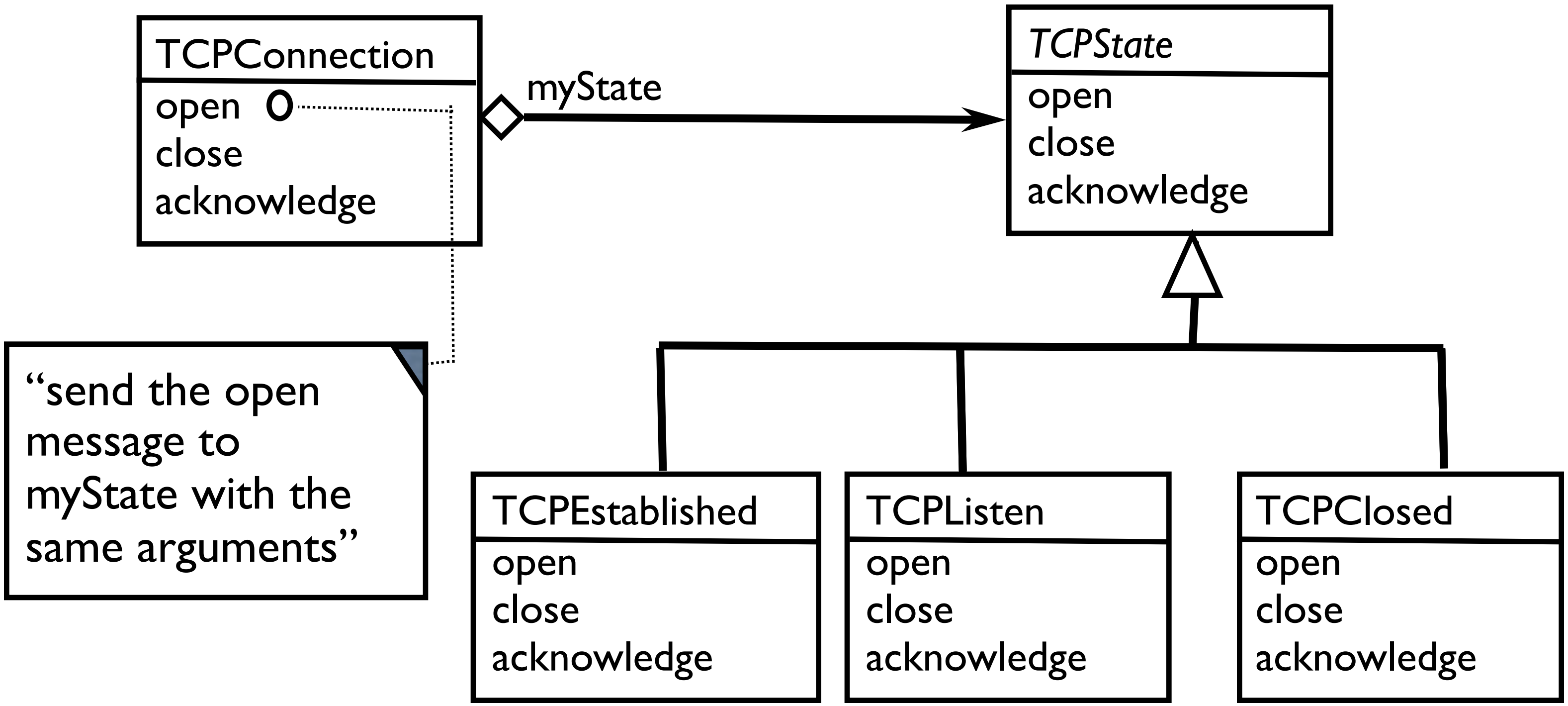
When the TCPConnection changes state, it simply replaces the myState object with an object of another state

Example: Class that manages the state of a TCP/IP connection



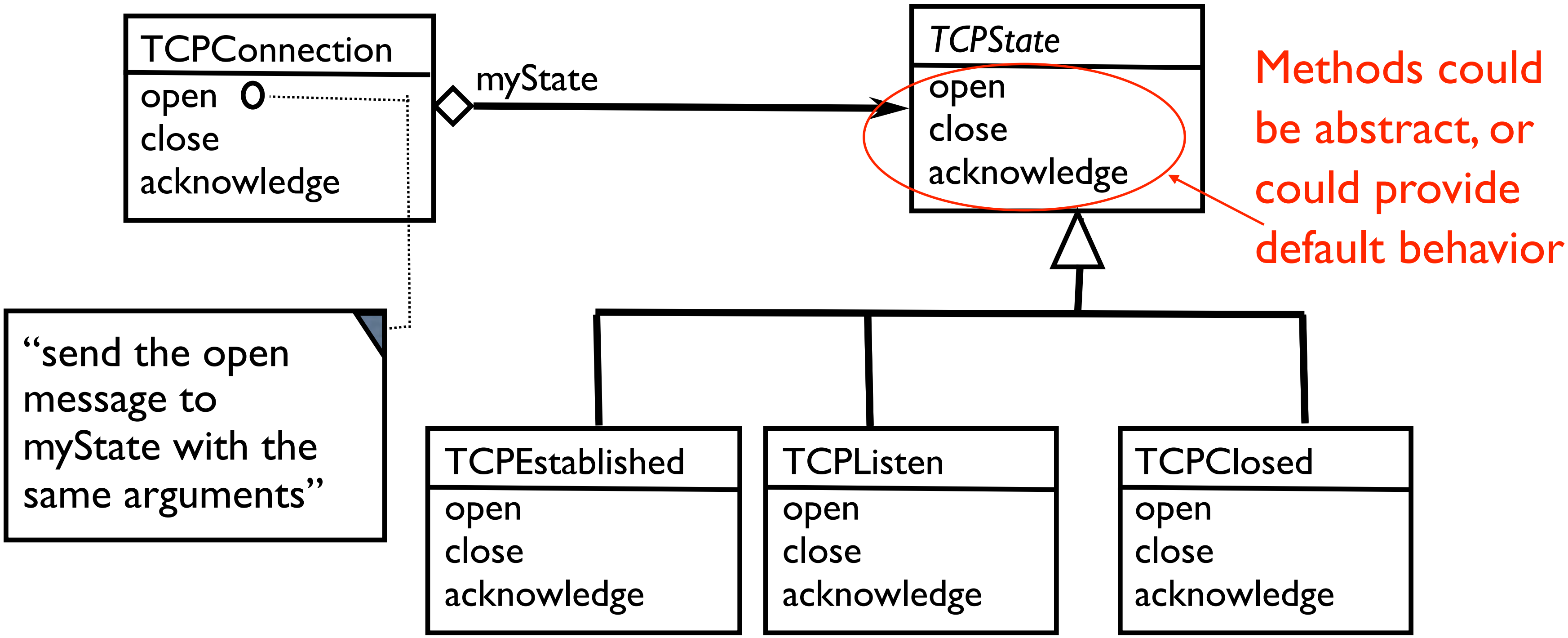
When the TCPConnection changes state, it simply replaces the myState object with an object of another state

Example: Class that manages the state of a TCP/IP connection



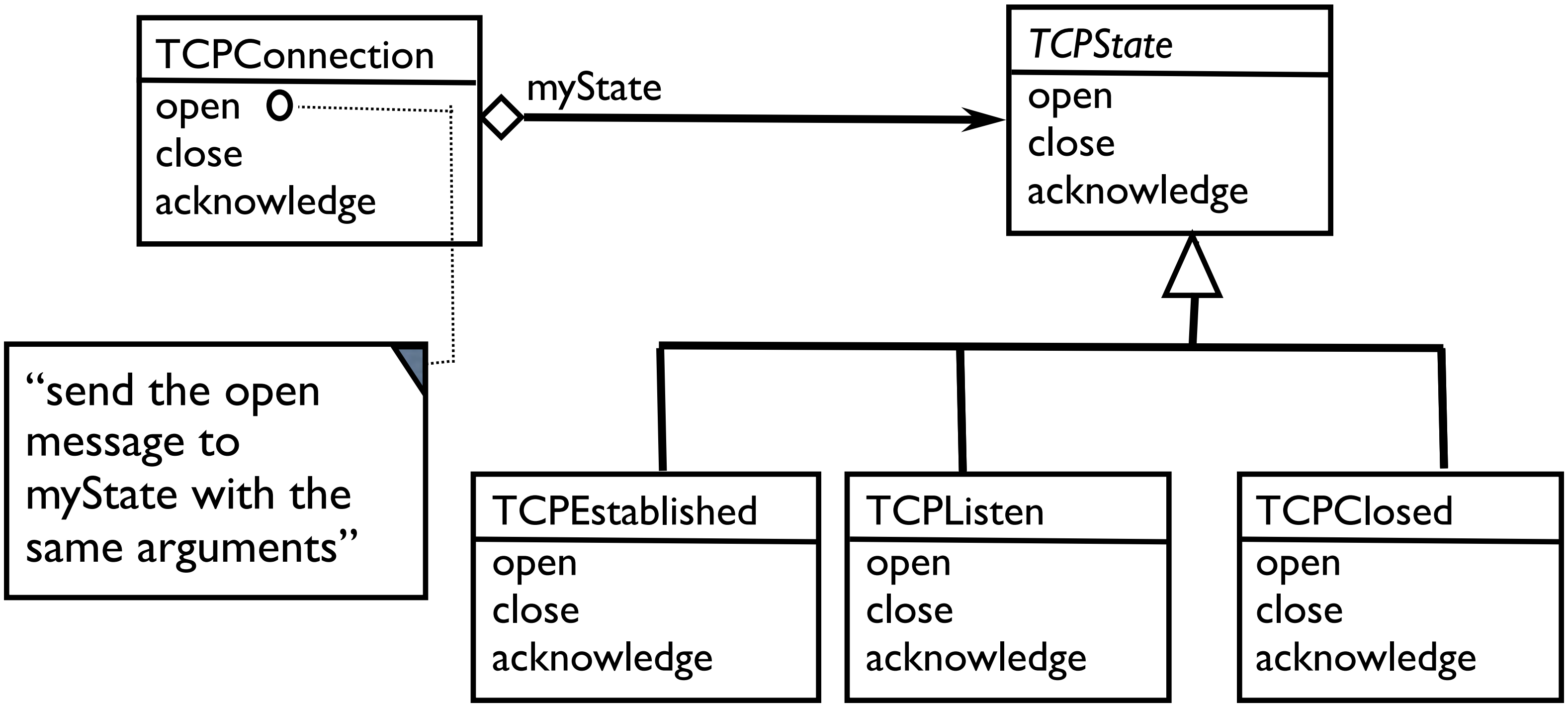
When the TCPConnection changes state, it simply replaces the myState object with an object of another state

Example: Class that manages the state of a TCP/IP connection

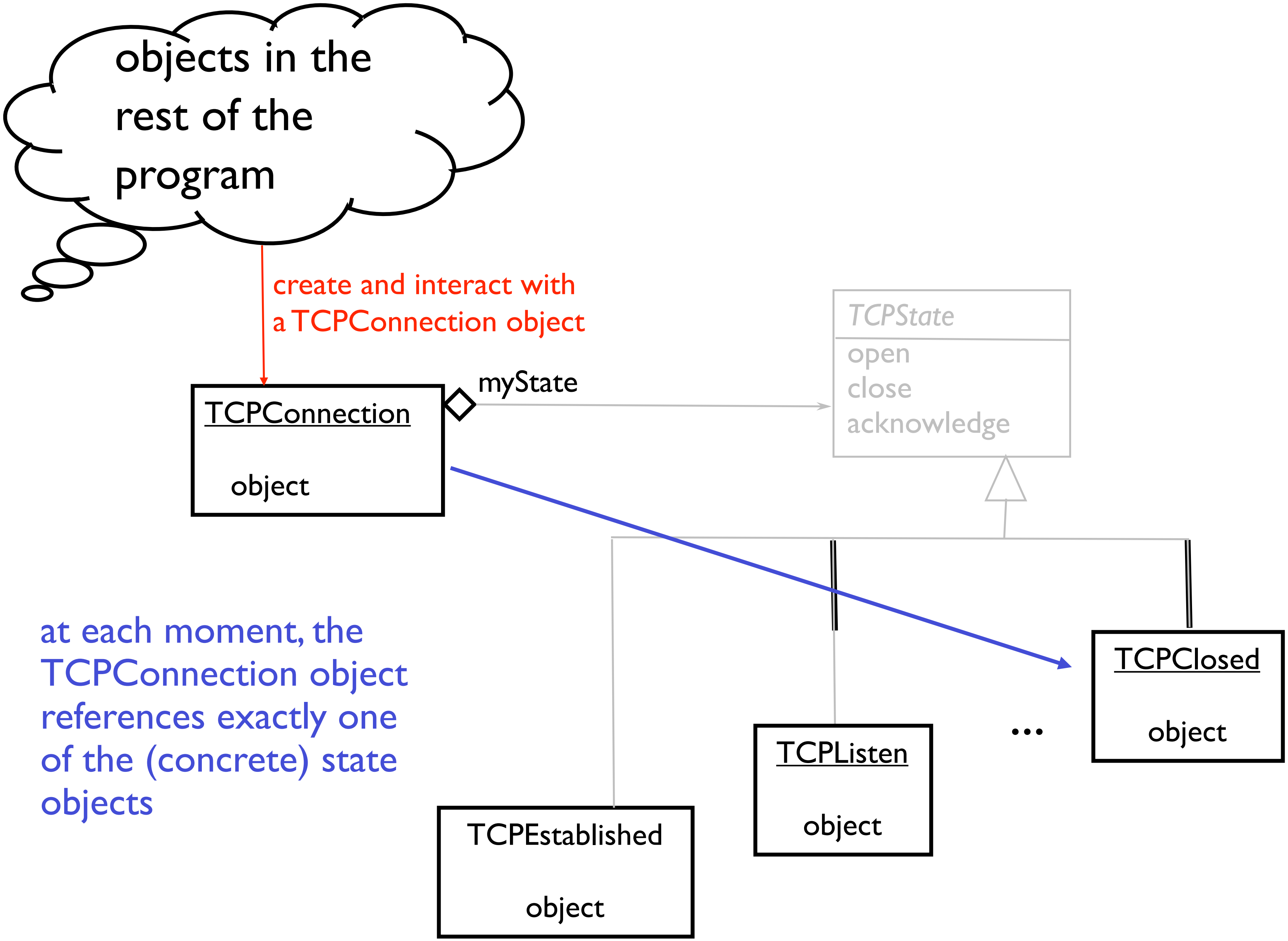


When the TCPConnection changes state, it simply replaces the myState object with an object of another state

Example: Class that manages the state of a TCP/IP connection

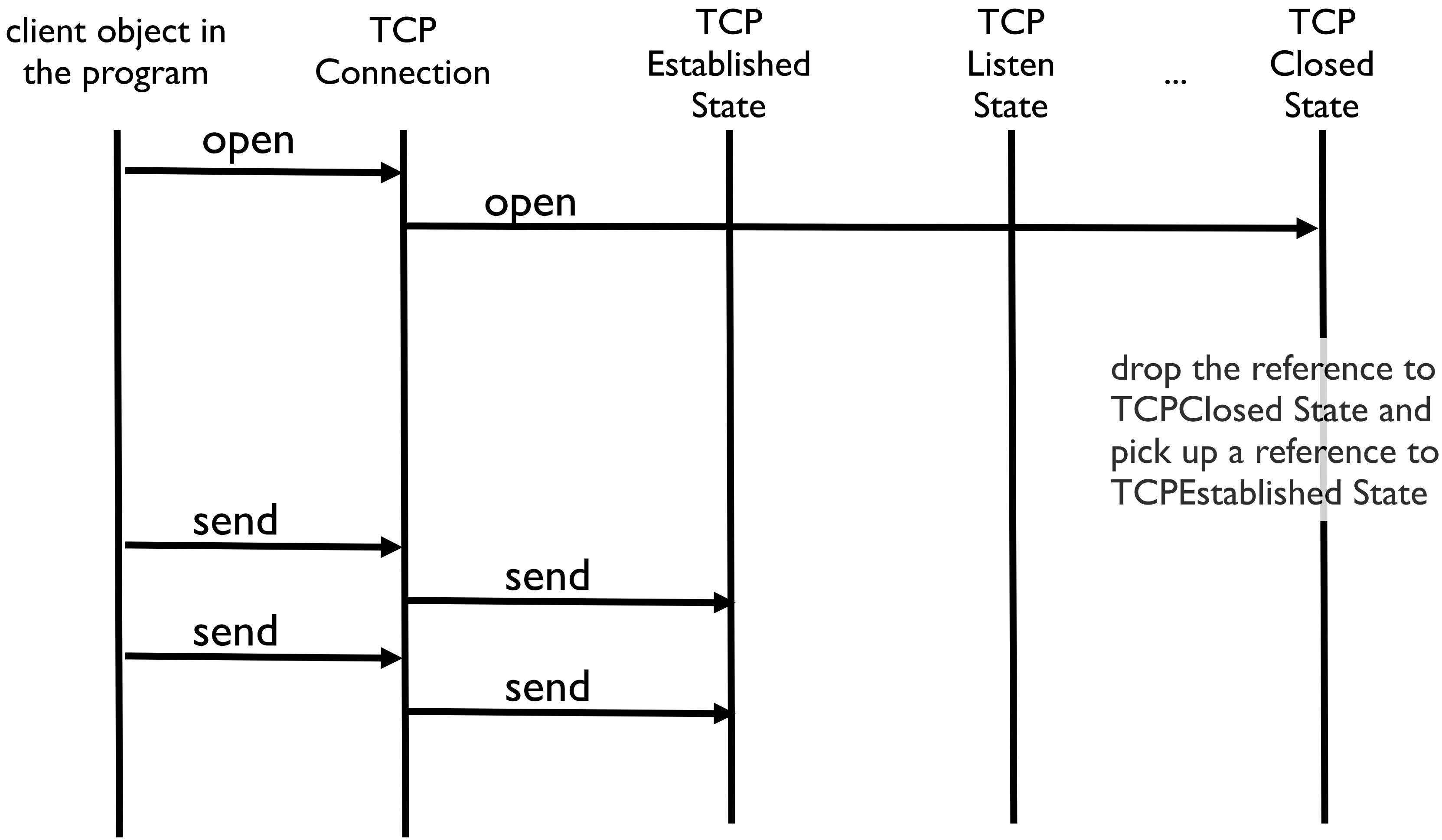


When the TCPConnection changes state, it simply replaces the myState object with an object of another state



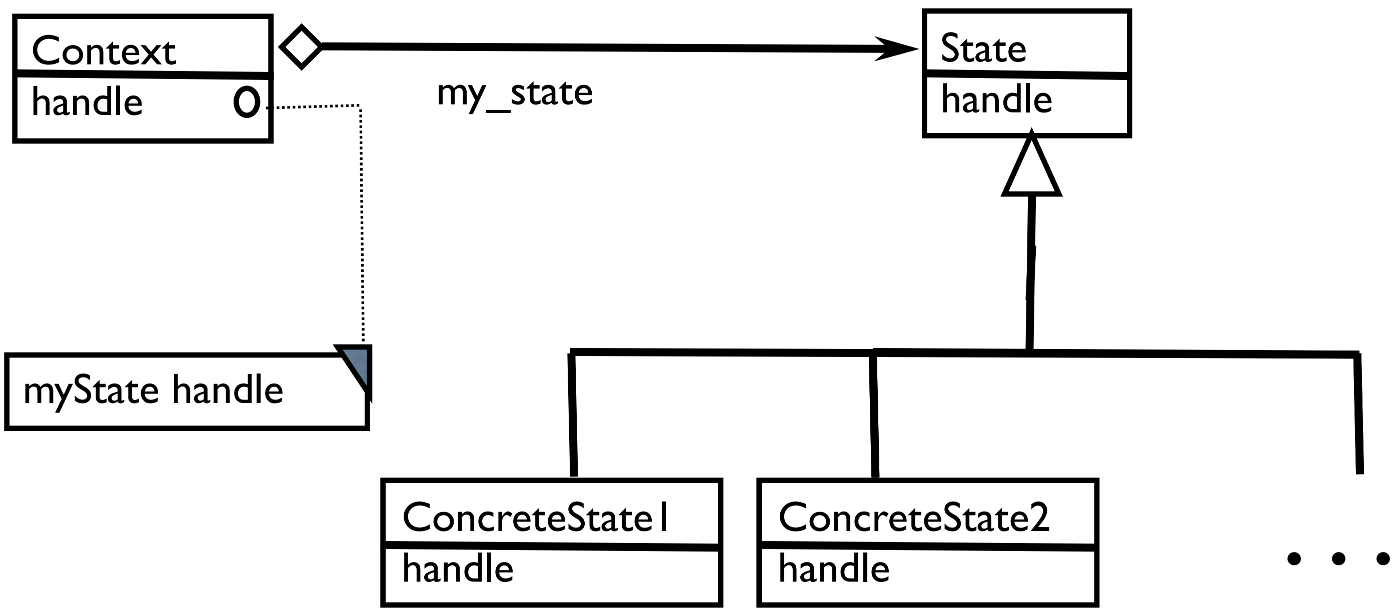
Portland State University

Sequence Diagram



Portland State University

Generic Class Diagram for the State Pattern



The State Pattern

Consequences: Localizes state-specific behavior & partitions behavior for different states. New states & transitions can be added easily.

Makes state transitions explicit. The context must "have" a different state.

State objects can be shared if they provide only behavior and have no instance variables of their own. All context objects in the same state can then share the same (singleton) state object.

Smalltalk Example of TCP Connection

The Design Patterns Smalltalk Companion by Alpert et al.

```
Object subclass: #TCPConnection
  instanceVariableNames: ‘state’
  classVariableNames: ‘’
  poolDictionaries: ‘’

Object subclass: #TCPState
  instanceVariableNames: ‘’
  classVariableNames: ‘’
  poolDictionaries: ‘’

TCPConnection>>activeOpen
  “delegate the open message to the current state.”
  self state activeOpen: self
```

TCP Connection (cont.)

```
Object subclass: #TCPConnection
  instanceVariableNames: 'state'
  classVariableNames: ''
  poolDictionaries: ''

Object subclass: #TCPState
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

TCPConnection>>activeOpen
  "delegate the open message to the current state."
  self state activeOpen: self
```

send it the *activeOpen* message (with self as an argument)

TCP Connection (cont.)

```
TCPState>>activeOpen: aTCPConnection
  “Don’t implement an open method….expect the concrete subclasses to”
  self subclassResponsibility
```

and do the same thing for all other messages for TCPState
(that is, TCPState is an abstract class)

```
TCPState subclass: #TCPEstablished
  instanceVariableNames: ‘’
  classVariableNames: ‘’
  poolDictionaries: ‘’
```

and do the same thing for all other concrete states that you need
(TCPListen state, TCPClosed state, etc.)

TCP Connection (cont.)

```
TCPEstablishedState>>activeOpen: aTCPConnection
  “Do nothing….the connection is already open”
  ^self

TCPClosedState >>activeOpen: aTCPConnection
  “do the open….invoke the “establishConnection method of TCPConnection”
  ^aTCPConnection establishConnection

TCPConnection>>establishConnection
  “Do the work to establish a connection.  Then change state.”
  self state: TCPEstablishedState new
```

TCP Connection (cont.)

```
TCPEstablishedState>>activeOpen: aTCPConnection
  “Do nothing….the connection is already open”
  ^self

TCPClosedState >>activeOpen: aTCPConnection
  “do the open….invoke the “establishConnection method of TCPConnection”
  ^aTCPConnection establishConnection

TCPConnection>>establishConnection
  “Do the work to establish a connection.  Then change state.”
  self state: TCPEstablishedState new
```

create a new TCPEstablished-State object

Portland State UNIVERSITY

TCP Connection (cont.)

```
TCPEstablishedState>>activeOpen: aTCPConnection
  “Do nothing….the connection is already open”
  ^self

TCPClosedState >>activeOpen: aTCPConnection
  “do the open….invoke the “establishConnection method of TCPConnection”
  ^aTCPConnection establishConnection

TCPConnection>>establishConnection
  “Do the work to establish a connection.  Then change state.”
  self state: TCPEstablishedState new
```

send the state: message to self to change my state

create a new TCPEstablished-State object

Portland State UNIVERSITY

Design Decisions for the TCP example

- how/when are the state objects created? how are they addressed?

- are the state objects shared?
- who is responsible for making the state transitions? methods in the concrete states? or methods in the TCPConnection objects?

- is “TCPState” an interface? an abstract class? or a concrete class?

- where will the actual methods (where the work is actually accomplished) be performed? in the concrete states? in the TCPConnection?

Design Decisions for the TCP example

- how/when are the state objects created? every time we make a state transition! How are they addressed? returned by new operator
- are the state objects shared? no
- who is responsible for making the state transitions? methods in the concrete states? or methods in the TCPConnection objects? state transistions are made in TCPConnection (within the methods that actually perform the valid operations)
- is “TCPState” an interface? an abstract class? or a concrete class?
  TCPState is an abstract class (Smalltalk doesn’t support interfaces)
- where will the actual methods (where the work is actually accomplished) be performed? in the concrete states? in the TCPConnection? in methods of the TCPConnection

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects?
- Are the state objects shared?
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared?
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? No
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? No
  They are created anew in each context
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.

- Are the state objects shared? No

  They are created anew in each context
- How and when are the state objects created?
  How are they addressed? Through instance variables of the context

- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?

- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?

- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? No
- How and when are the state objects created? They are created anew in each context
  How are they addressed? Through instance variables of the context
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects? The state objects
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? No
- How and when are the state objects created? They are created anew in each context
  How are they addressed? Through instance variables of the context
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects? The state objects
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class? They inherit from an abstract class
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? *Let's leave them in the context object.*

- Are the state objects shared? *No*

  They are created anew in each context

- How and when are the state objects created?
  How are they addressed? *Through instance variables of the context*

- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects? *The state objects*

- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class? *They inherit from an abstract class*

- Where will the the work actually be accomplished? In the concrete states? Or in the Context object? *In the concrete states*

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects?
- Are the state objects shared?
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? *Let's leave them in the context object.*
- Are the state objects shared?
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? Let's make them singletons
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? Let's make them singletons
  They are created once on initialization
- How and when are the state objects created?
  How are they addressed?
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.

- Are the state objects shared? Let's make them singletons

  They are created once on initialization

- How and when are the state objects created?
  How are they addressed? Through methods of the singleton class

- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects?

- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?

- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? Let's make them singletons
They are created once on initialization
- How and when are the state objects created?
How are they addressed? Through methods of the singleton class
- Who is responsible for making the state transitions?
Methods in the context object?
Or methods in the state objects? The state objects
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class?
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.

- Are the state objects shared? Let's make them singletons

  They are created once on initialization
- How and when are the state objects created?
  How are they addressed? Through methods of the singleton class

- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects? The state objects

- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class? They inherit from an abstract class

- Where will the the work actually be accomplished? In the concrete states? Or in the Context object?

Design Decisions for the Reflex Tester

- Where are the instance variables? In the context object, or in the state objects? Let's leave them in the context object.
- Are the state objects shared? Let's make them singletons
- How and when are the state objects created? They are created once on initialization
  How are they addressed? Through methods of the singleton class
- Who is responsible for making the state transitions?
  Methods in the context object?
  Or methods in the state objects? The state objects
- Do the state objects implement an interface? Or inherit from an abstract class? Or a concrete class? They inherit from an abstract class
- Where will the the work actually be accomplished? In the concrete states? Or in the Context object? In the concrete states

Summary

- Consider the state pattern when several methods choose their behavior based on the value of an instance variable
- You can introduce patterns using refactoring
- Take baby steps: code a little; test a little; repeat

- Don’t expect to go in a straight line
  - You will make changes to enable a refactoring that you will later undo
    - example: introducing the three instance variables to address the three possible states
  - You will make design decisions that turn out to be inconvenient
  - We are not yet done:
    - example: the random number generator is accessed only from ReflexStateIdle, so could be moved there
    - but then ReflexStateIdle would no longer be a Singleton