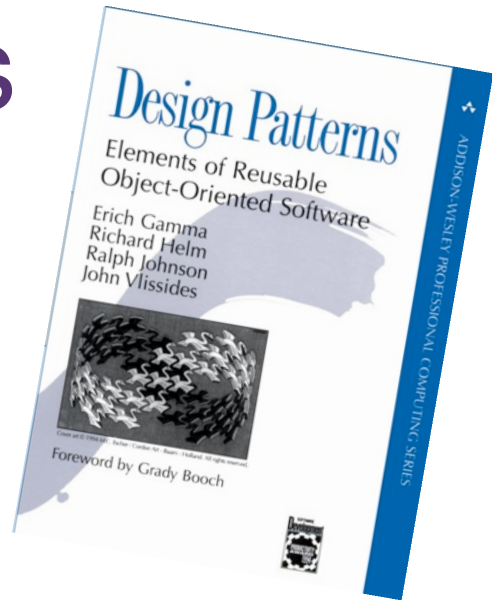
CS 420/520 — Fall 2009

Introduction to Design Patterns

Portland State University

Design Patterns

Elements of Reusable
Object-Oriented Software



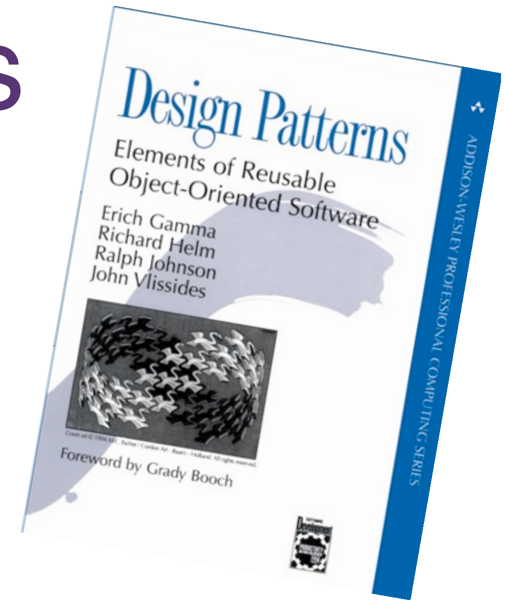
by

Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides

Addison-Wesley, 1995.

Design Patterns

Elements of Reusable Object-Oriented Software



by

The Gang of Four

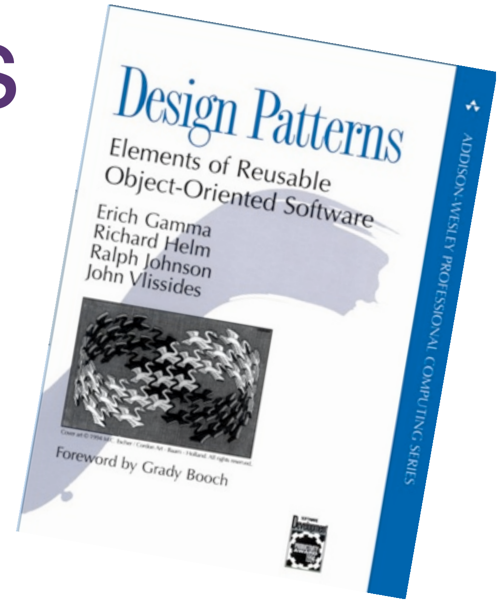
Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides

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

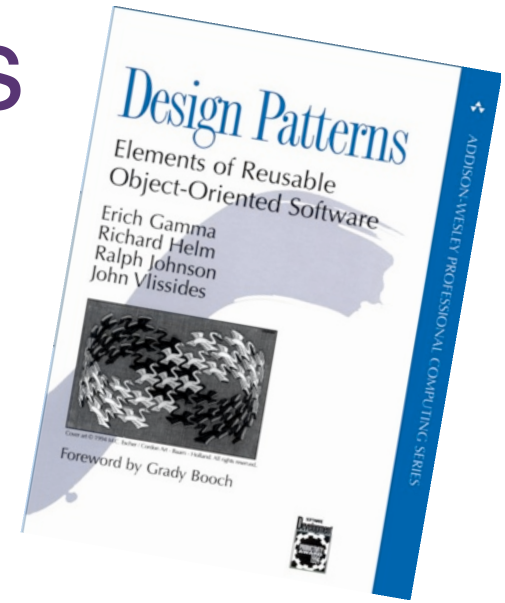
Addison-Wesley, 1995.

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

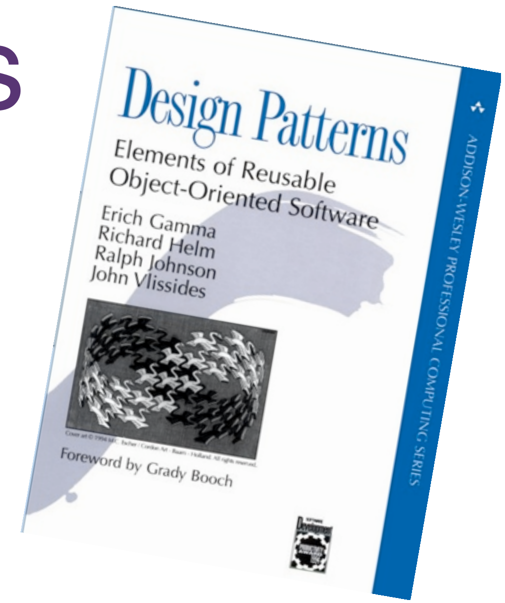
Addison-Wesley, 1995.

Portland State UNIVERSITY

original,
well-known
book introducing
design patterns

often called
the "Gang of Four"
or GoF book

Design Patterns



Elements of Reusable Object-Oriented Software

Examples presented in C++ (and Smalltalk)

by

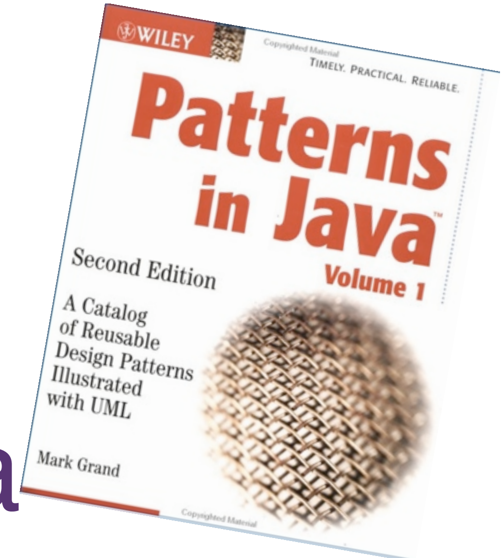
The Gang of Four

Erich Gamma, Richard Helm,
Ralph Johnson, John Vlissides

Addison-Wesley, 1995.

Portland State
UNIVERSITY

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

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.

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!

Portland State UNIVERSITY

Design Patterns in Java

by

Steven John Metsker
and William C. Wake

Portland State UNIVERSITY

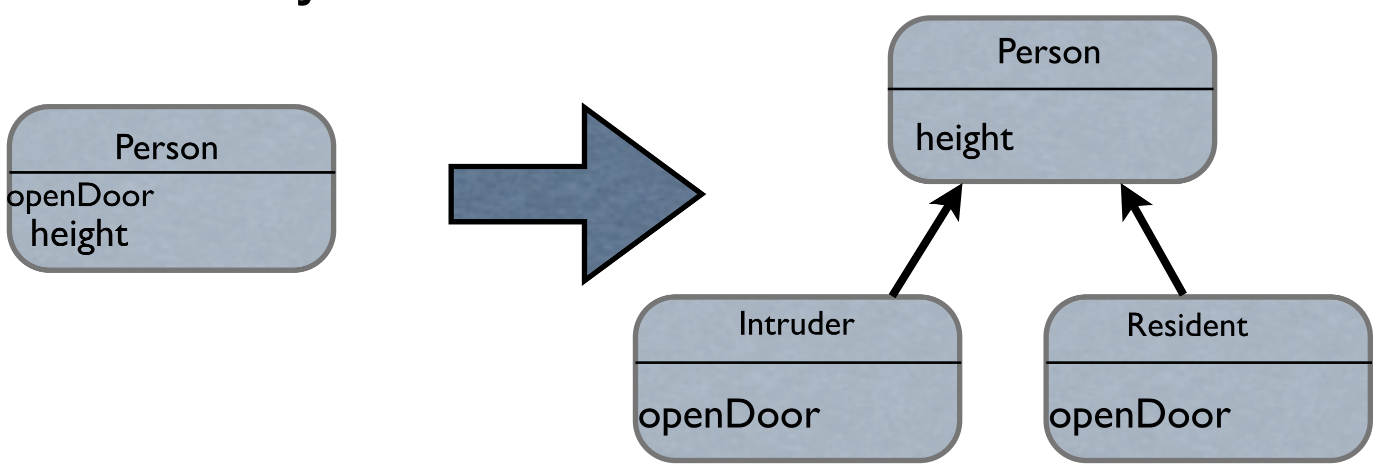
Why do patterns help in the Test–Code–Refactoring Cycle?

- 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 ?

Portland State University

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

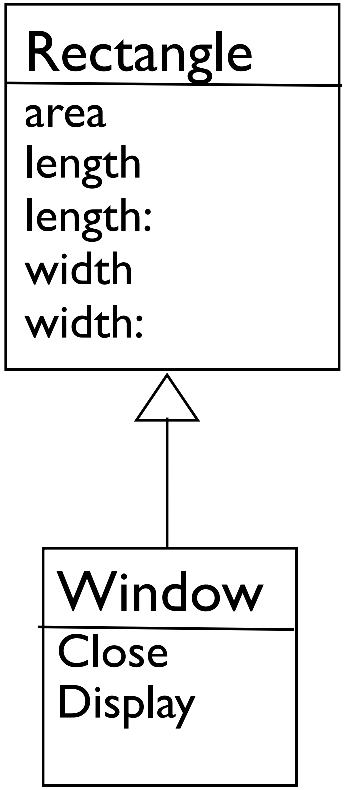
- Creating an object by specifying a class explicitly
  - CourseOffering new
- Depending on specific operations of someone else's object
  - student address line2 zipcode
- Dependence on object representations or implementations

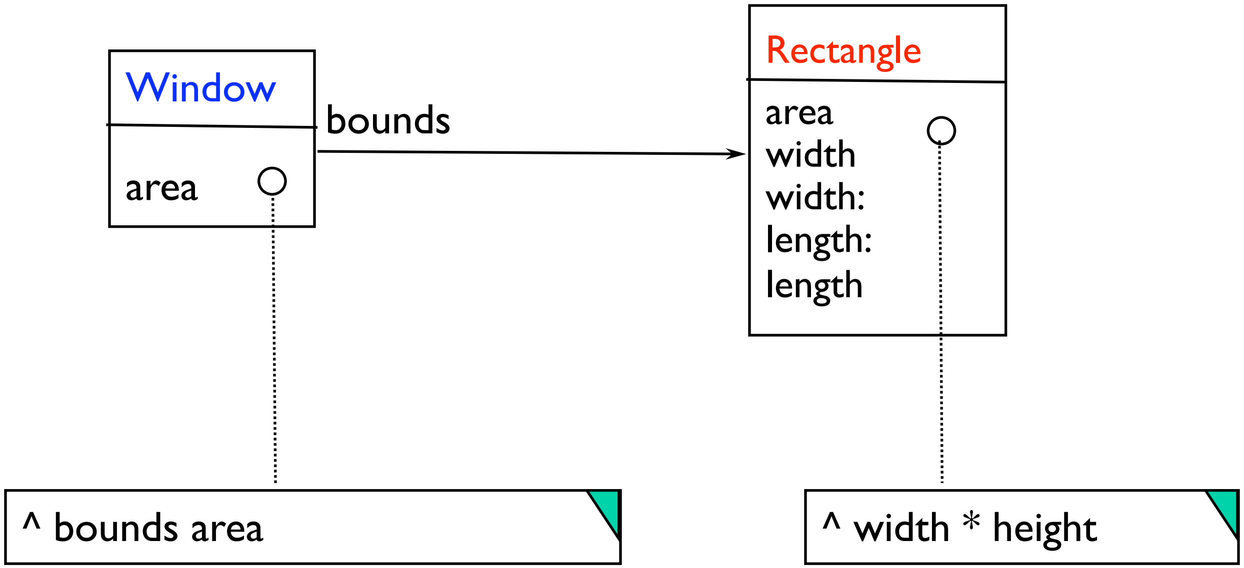
In general: information in more than one place

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

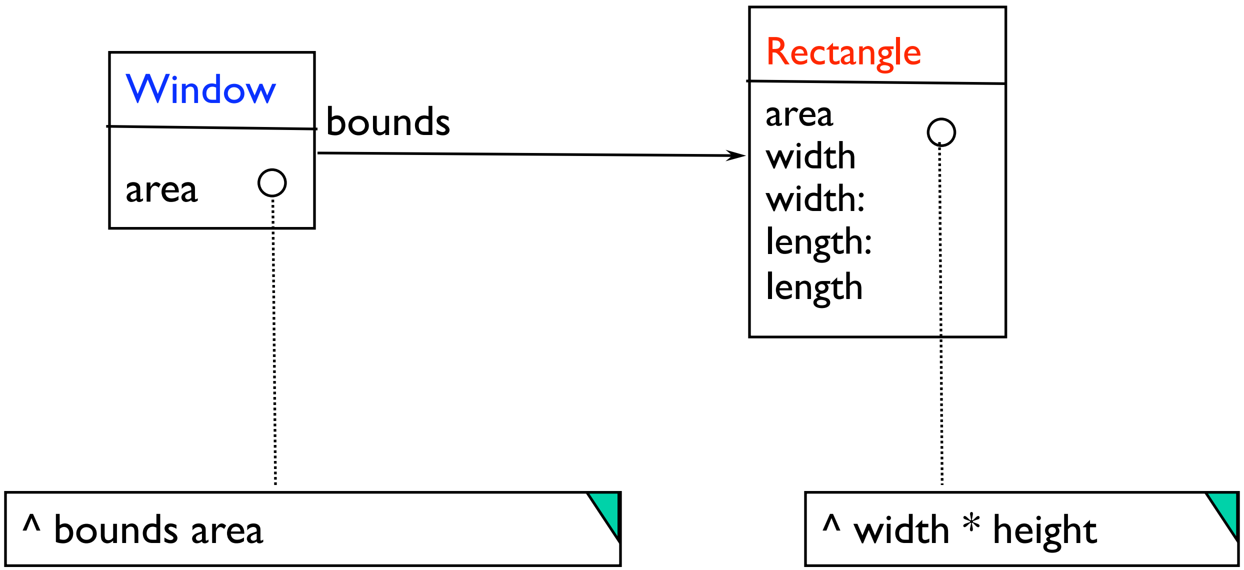




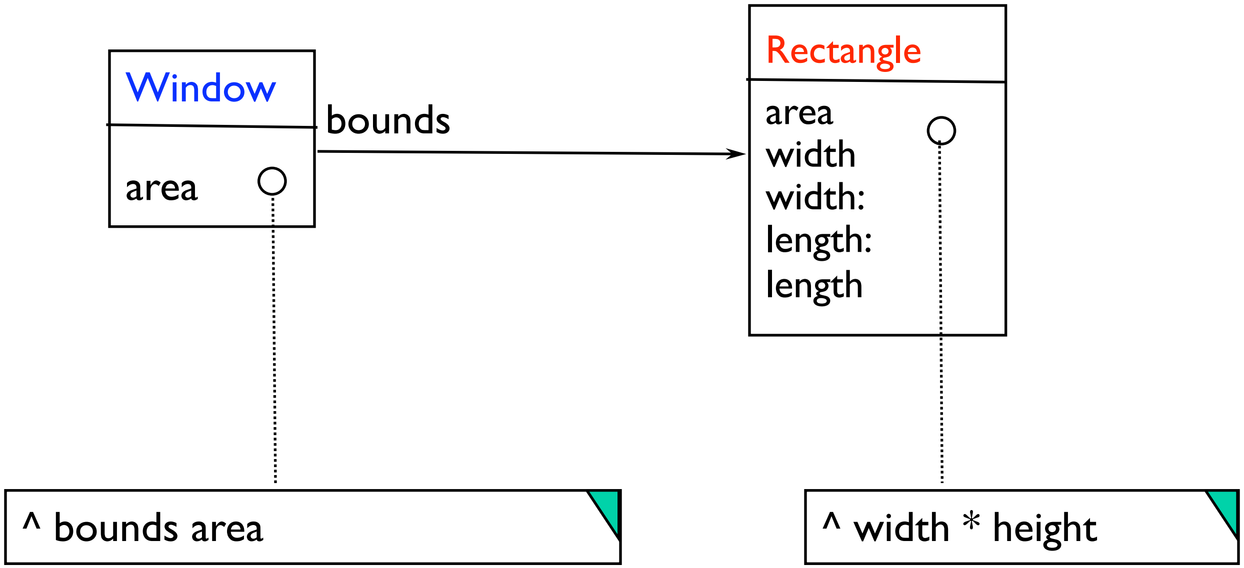
Example of delegation

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

Portland State University



Portland State University



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)

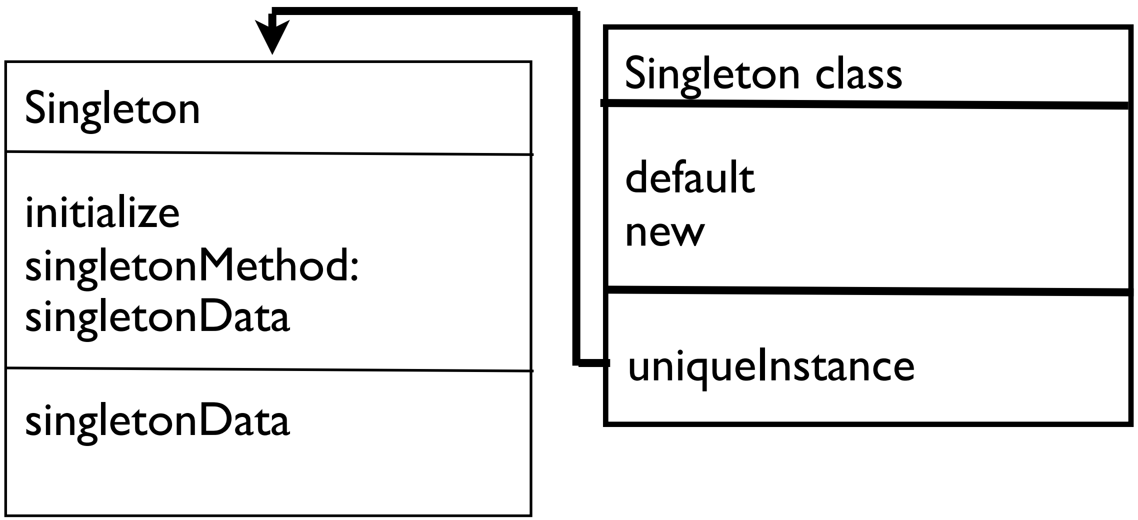
Applicability: – When the instance must be globally accessible
– Clients know that there is a single instance (or a few special instances).

Structure of the Singleton Pattern

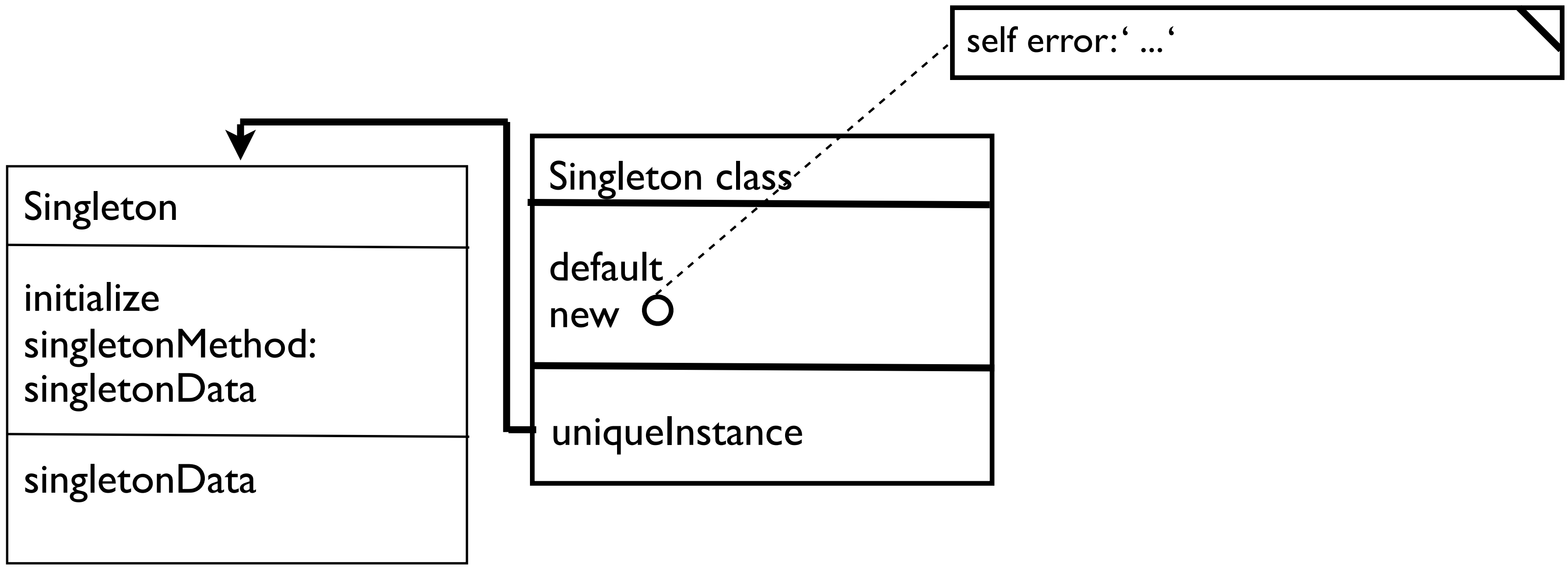
Structure of the Singleton Pattern

Singleton
initialize singletonMethod: singletonData
singletonData

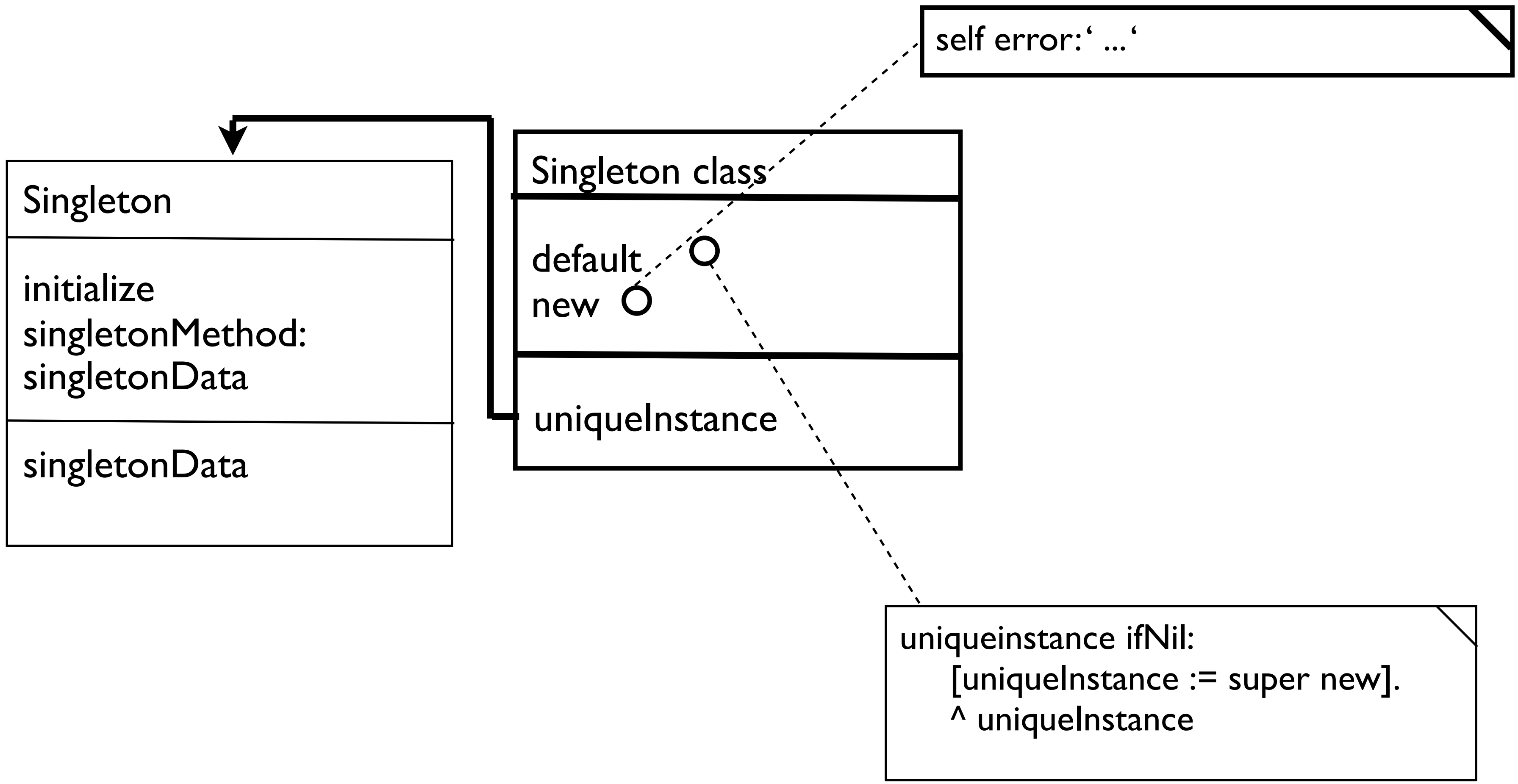
Structure of the Singleton Pattern



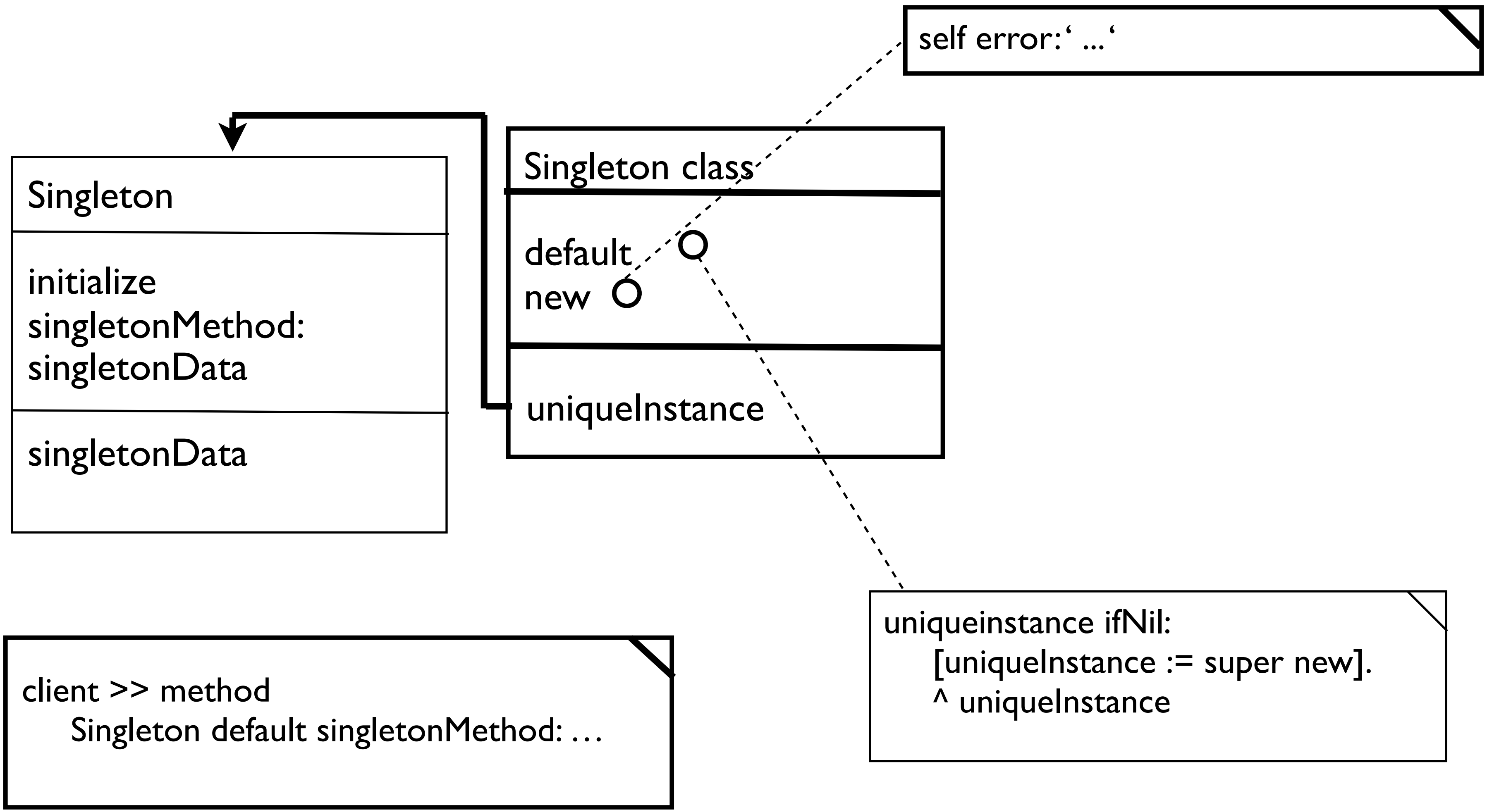
Structure of the Singleton Pattern



Structure of the Singleton Pattern



Structure of the Singleton Pattern



Portland State University

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" …

Portland State UNIVERSITY

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"
        ...
```

Portland State UNIVERSITY

Iterator

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()

Portland State University

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:

Portland State UNIVERSITY

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:

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.

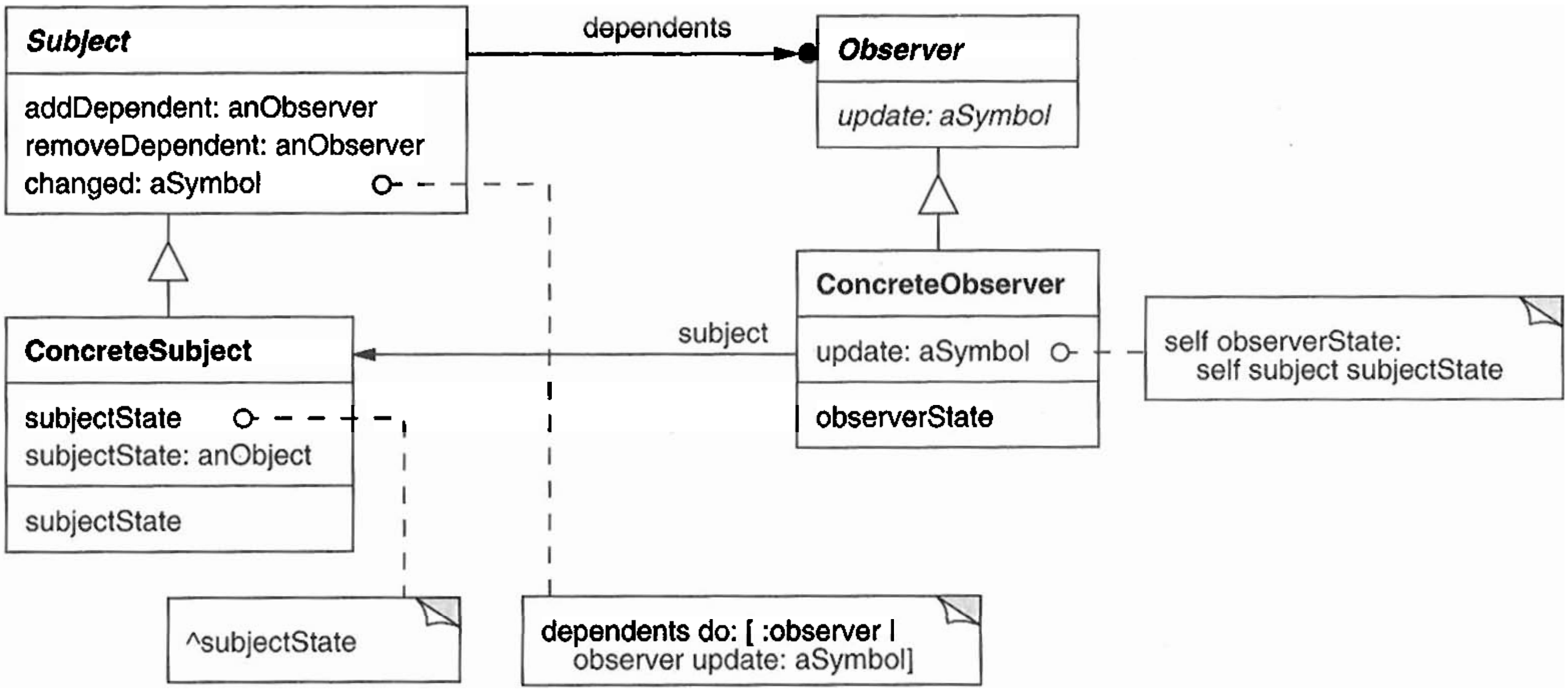


Figure from Alpert, page 305

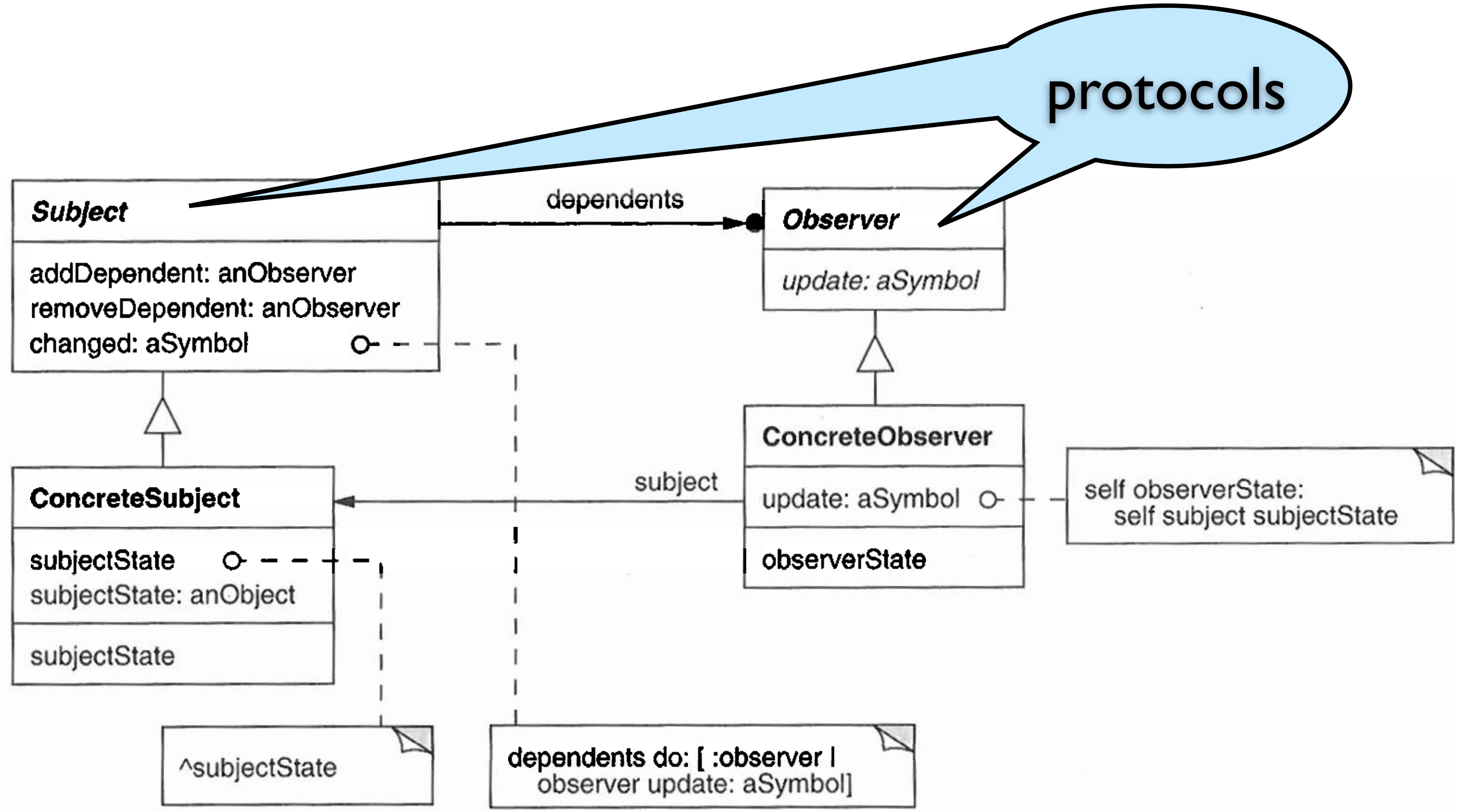


Figure from Alpert, page 305

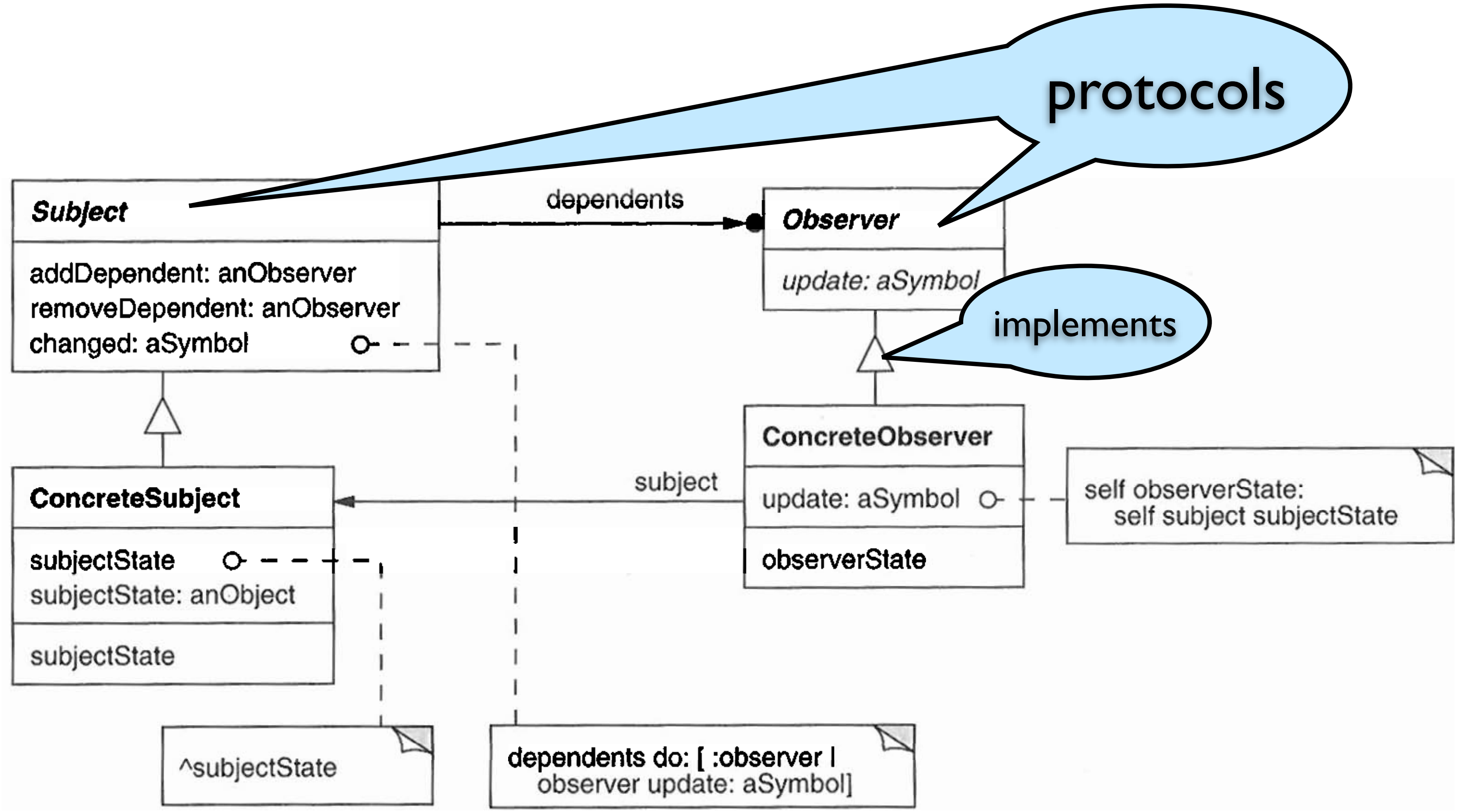


Figure from Alpert, page 305

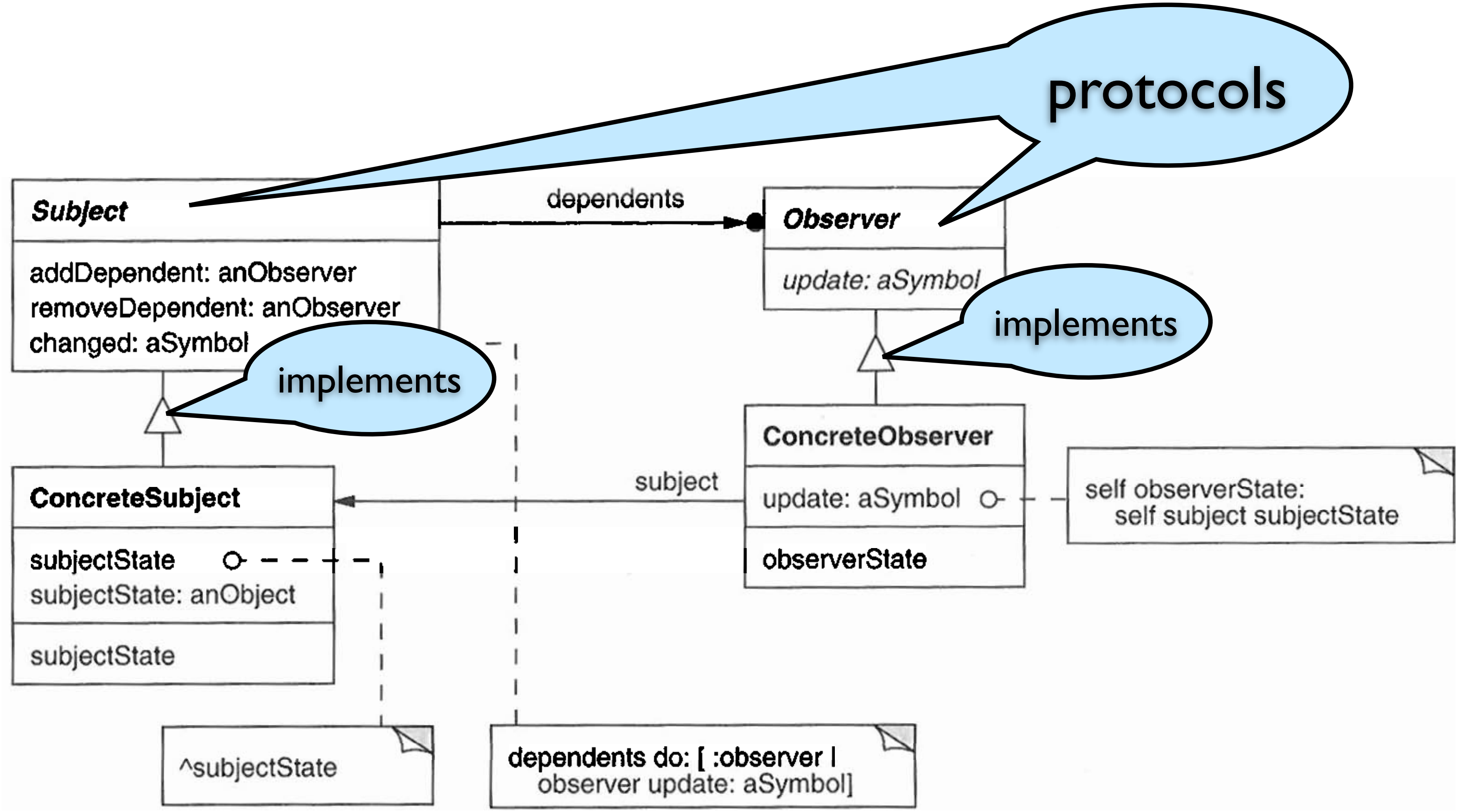


Figure from Alpert, page 305

Portland State University

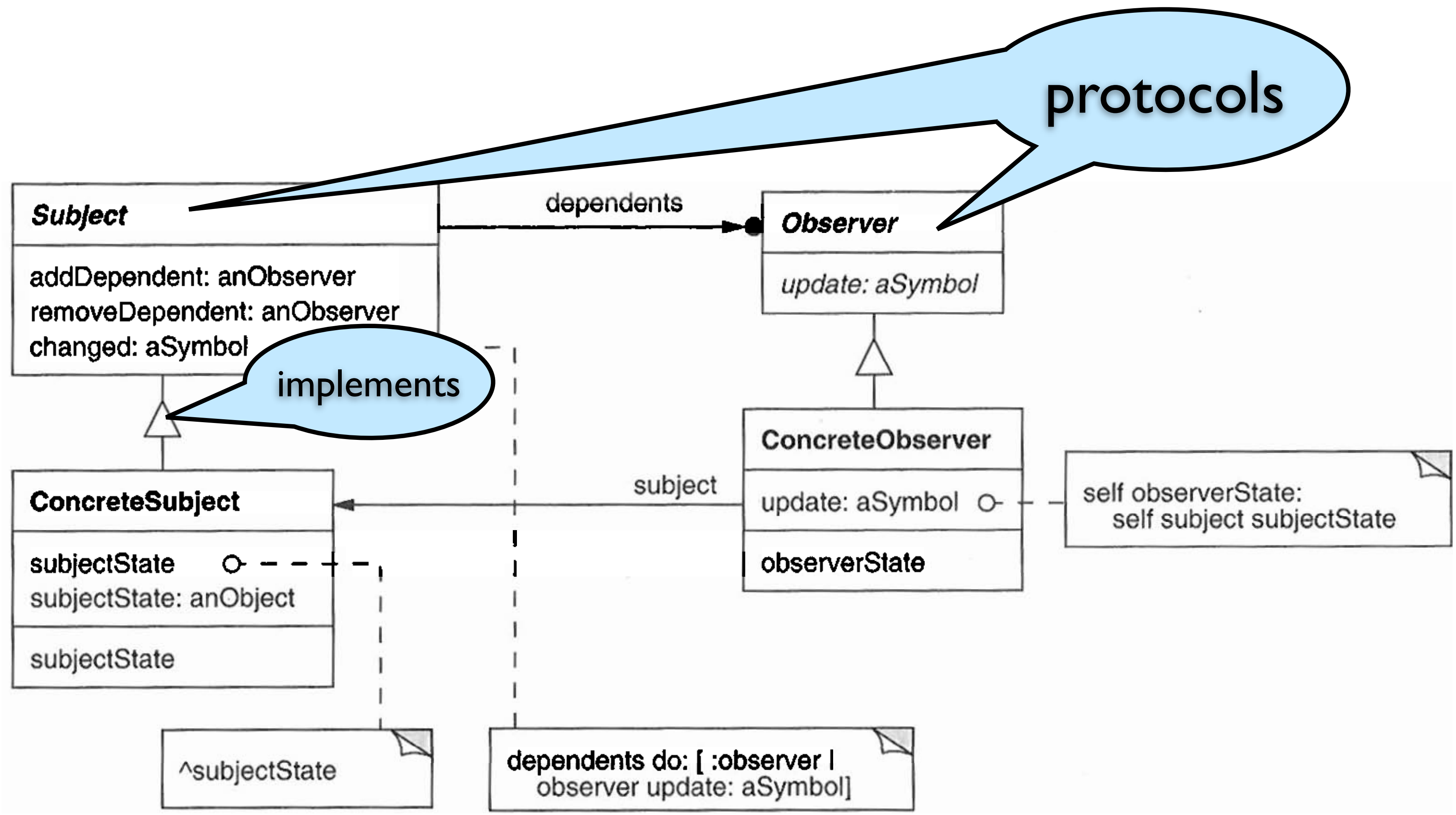


Figure from Alpert, page 305

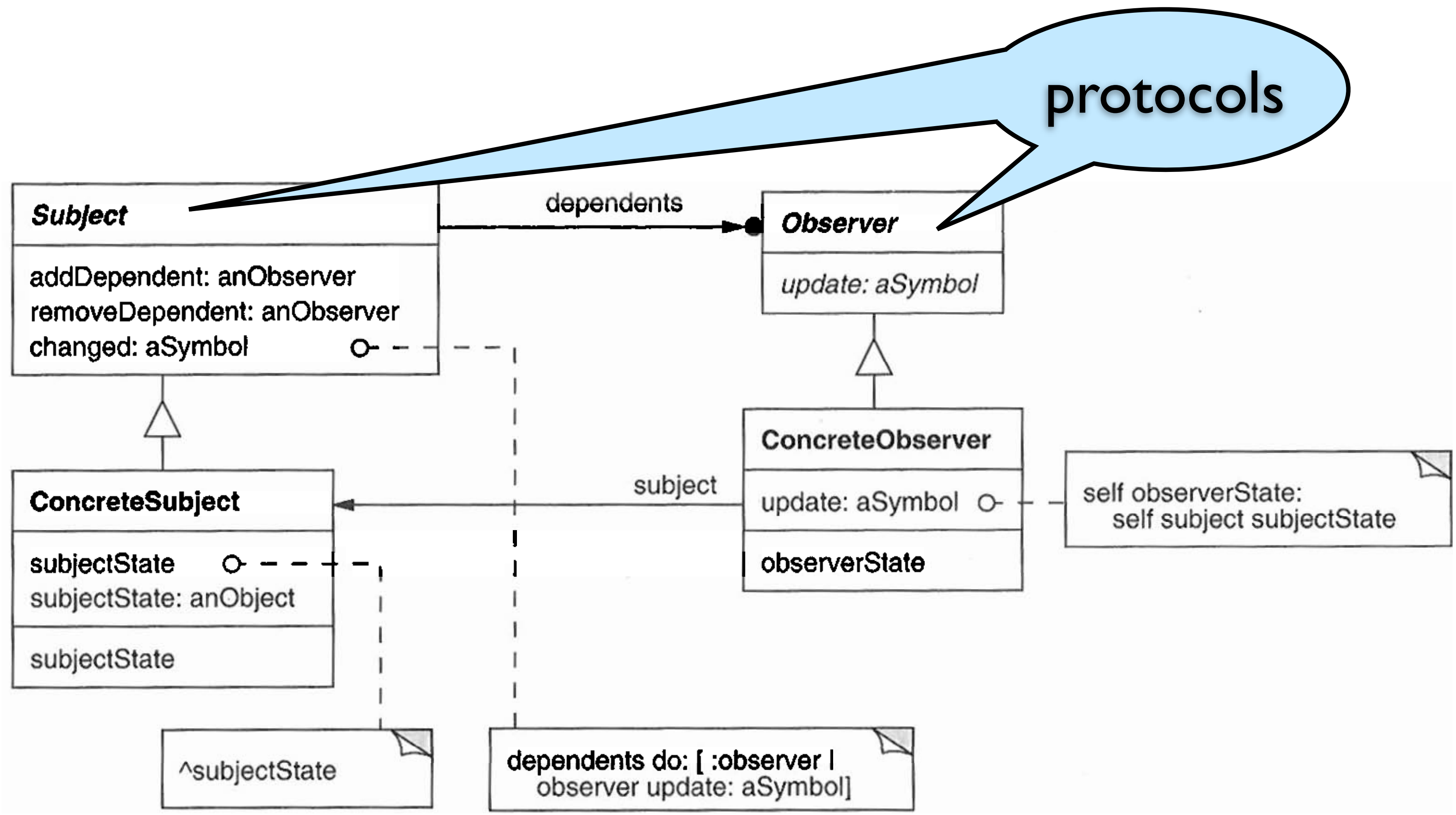


Figure from Alpert, page 305

- 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
```

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 it's 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.

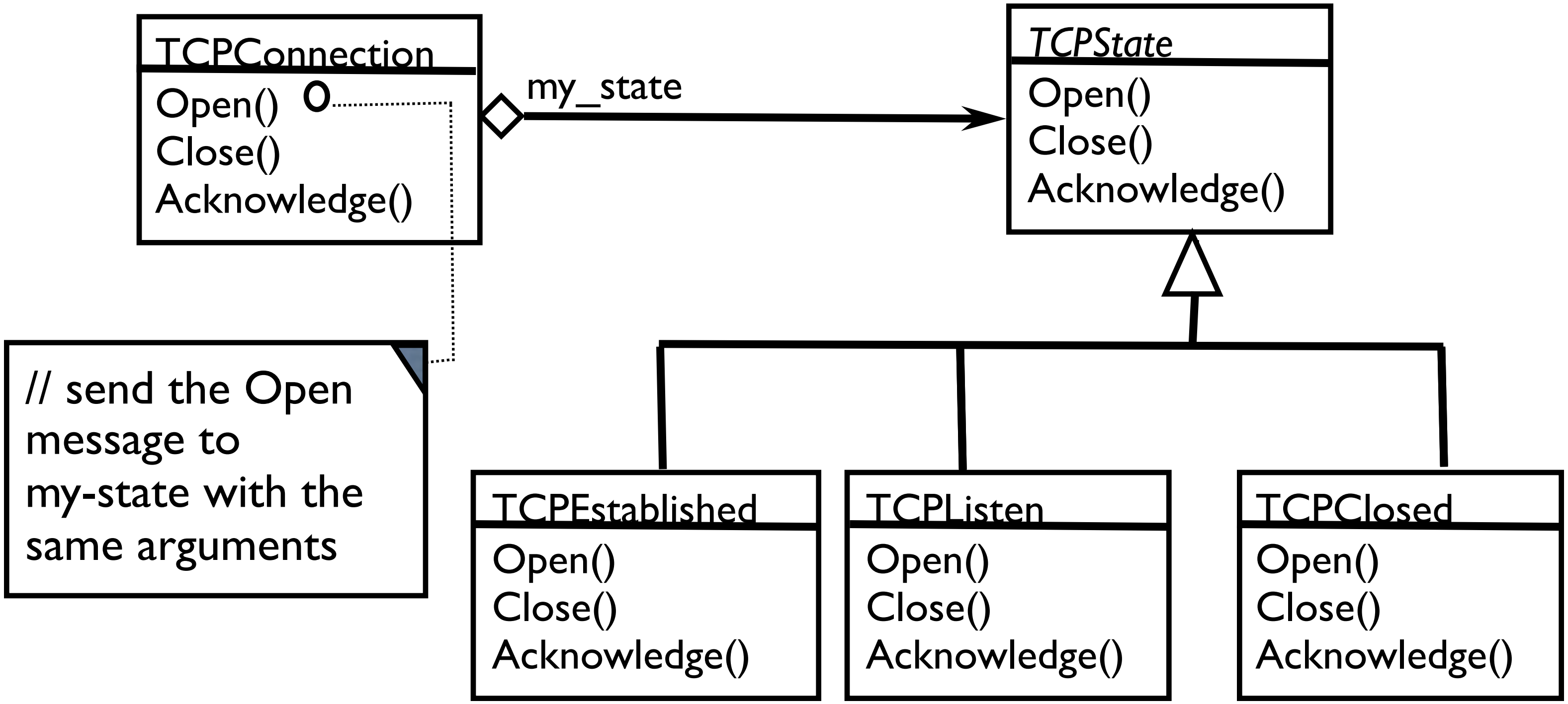
Motivation: Introduce an abstract class called State
Introduce (one instance) of each concrete class - for each state
Context has a state - delegates behavior to state
Treat object state as an object in its own right

Applicability: When an object's behavior depends on its state & it must change its state-dependent behavior at run-time
When operations have large, multi-part conditions that depend on the object's state

State Pattern: Allow an object to alter its behavior when its internal state changes (object will appear to change its class)

- Introduce an abstract class called "State"
- Use concrete subclasses to represent the possible states
- Define all operations in the abstract class State
- Then override them in the states, as appropriate
  (certain operations can be ignored...or produce errors)

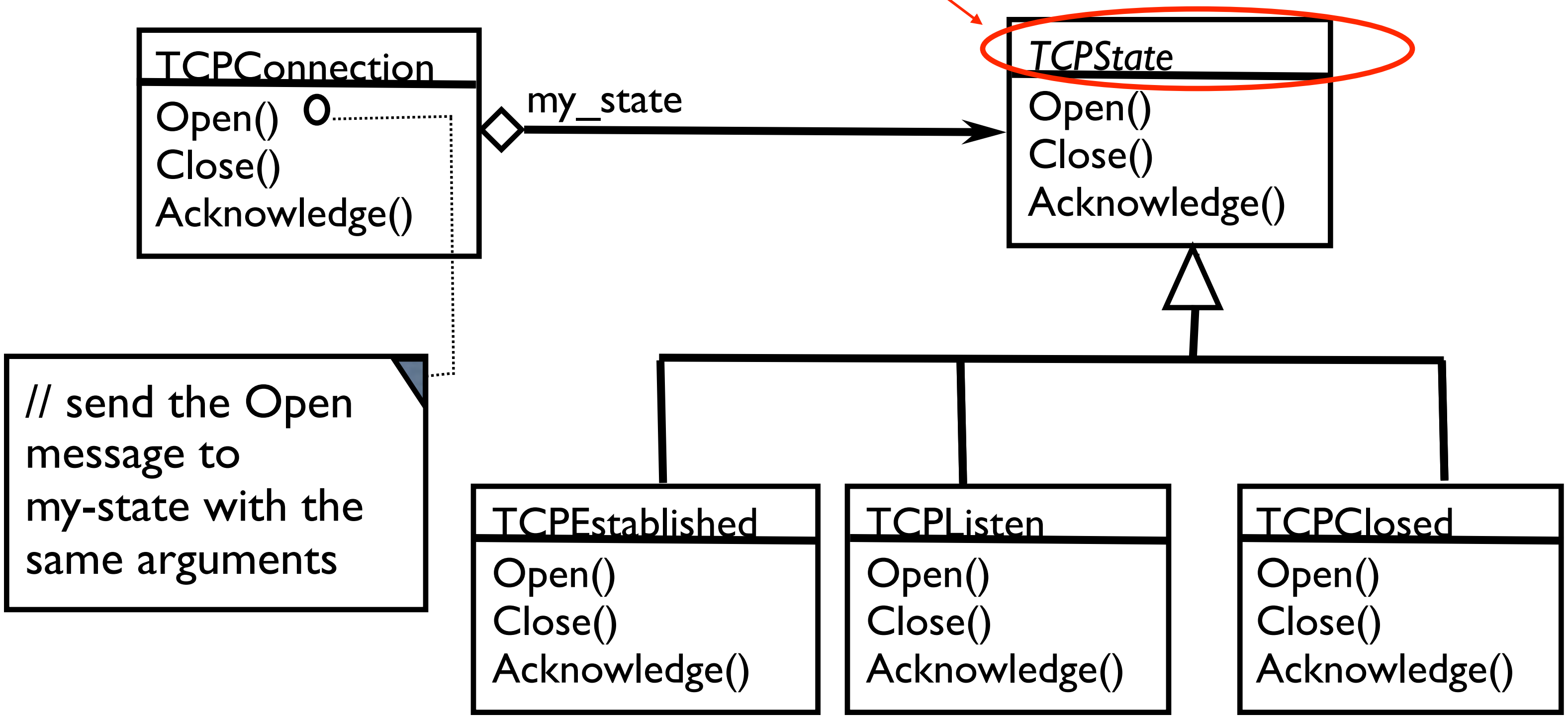
Example: Class that manages the state of a TCP/IP connection



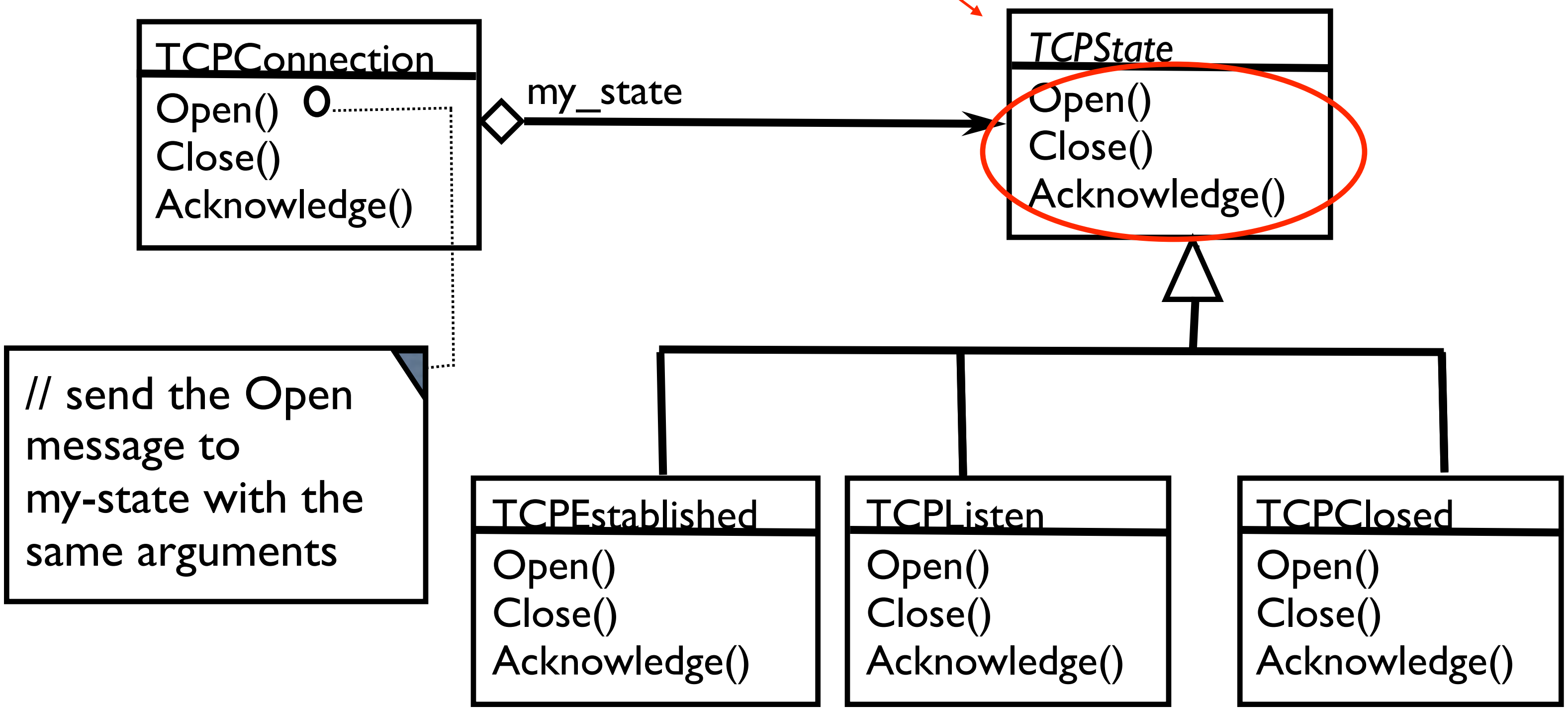
If the TCPConnection changes state, then it simply replaces the object of one state with an object of another state

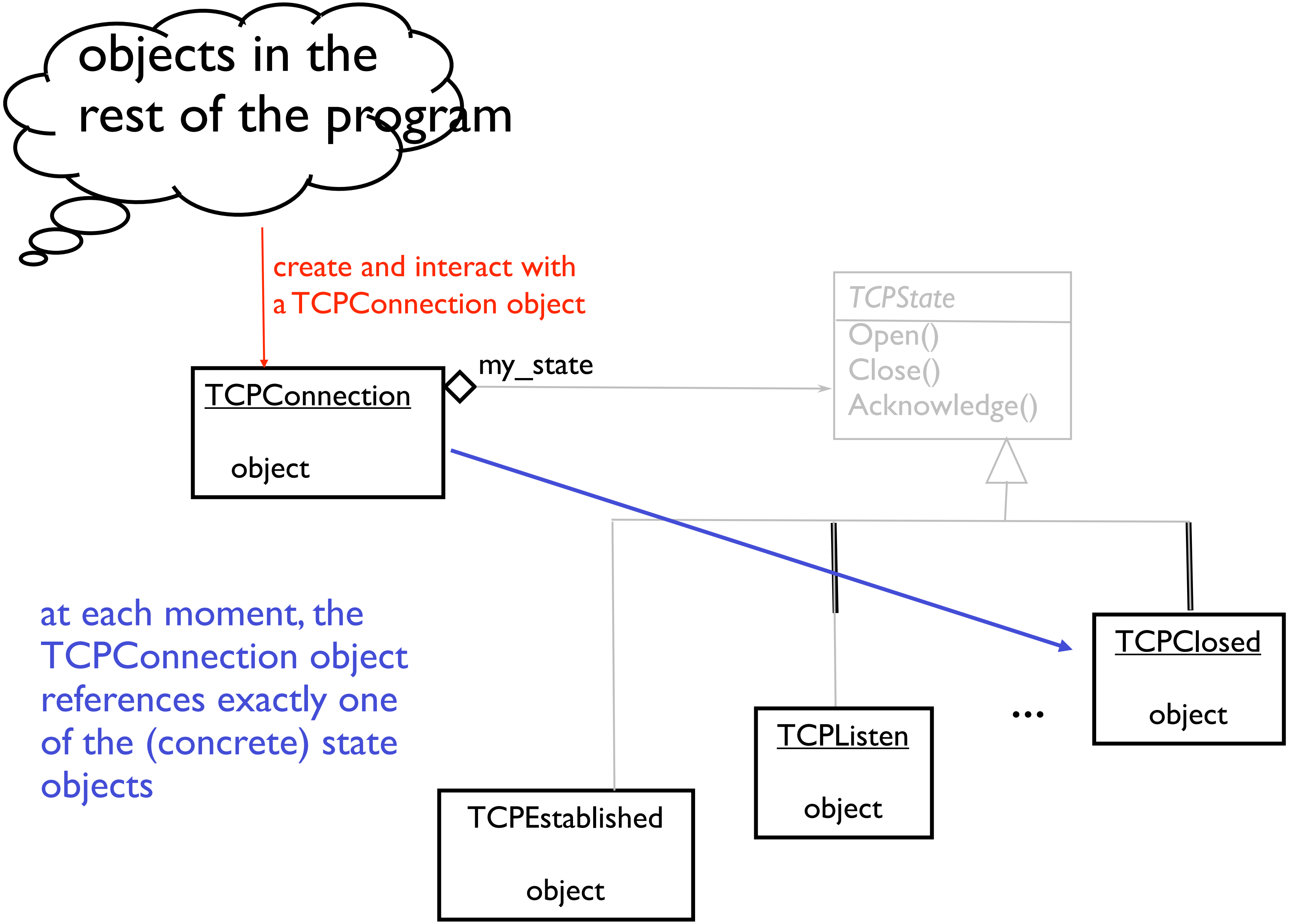
Notice that we have an abstract class here!

If we have no method bodies and no instance variables, then it could be an Interface in Java



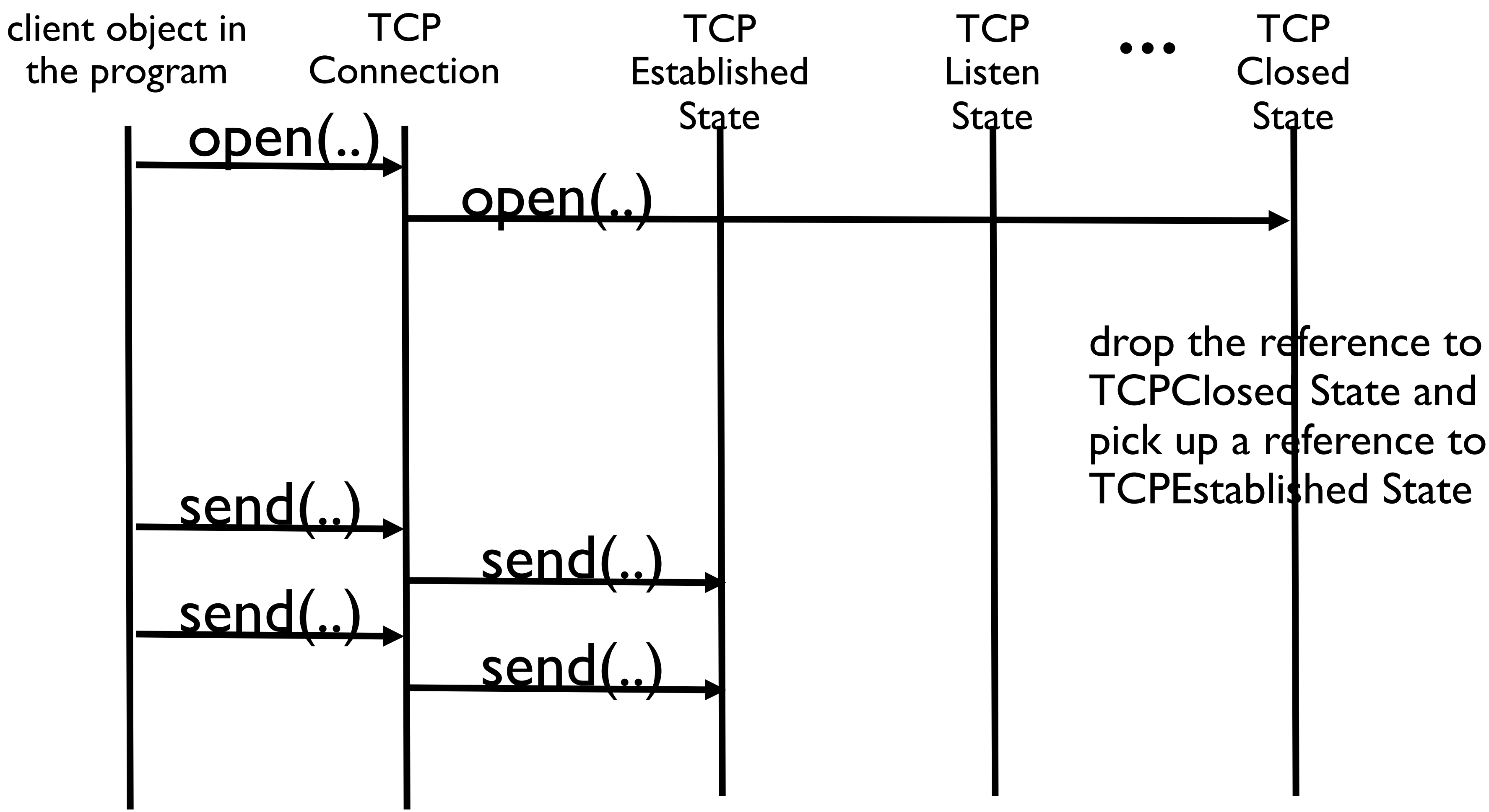
Notice that we have abstract methods here! But they could provide default behavior (usually "do nothing")





Portland State University

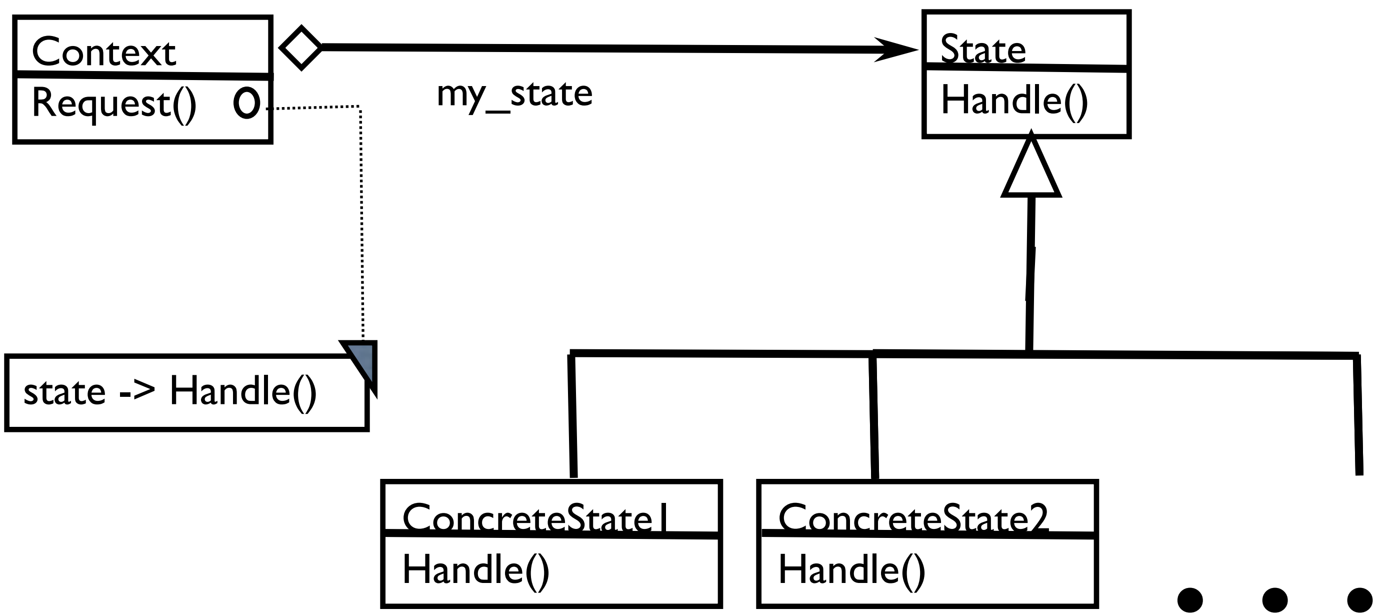
Sequence Diagram



Design Decisions for the State

- 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?

Generic Class Diagram for the State Pattern



The State Pattern

Participants:

Context (TCPConnection) - defines the interface of interest to clients

State (TCPState) - defines an interface for encapsulating the behavior for a state

ConcreteState (TCPEstablished, TCPListen, TCPClosed) - each subclass implements a behavior associated with a state

Collaborations:

Context delegates state-specific behavior to current ConcreteState object

Context may pass itself as an argument to State object (to let it access context)

Context is the primary interface for clients once states are configured, clients unaware

Either Context or ConcreteState subclasses can decide which state succeeds another & when

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 only provide state-specific behavior). All objects in the same state can "have" the same (single) state object.

Using the State Pattern

Read the consequences of the (State) pattern:

1. it localizes state-specific behavior;
   partitions the behavior for different states
2. it makes state transitions explicit
3. the state objects (the individual objects that offer the behavior for a given state) can be shared

Smalltalk Example of TCP

```
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
```

Smalltalk Example of 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)

Portland State UNIVERSITY

Smalltalk Example of 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.)

Smalltalk Example of 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
```

Smalltalk Example of 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 your state

create a new TCPEstablished state object

Portland State UNIVERSITY

Design Decisions for the

- 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

- 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

Java Example of the State

```
class TCPState {
  //Symbolic constants for events
  public static final int Open = 1;
  public static final int Send = 2;
  public static final int Close = 3;
  // etc. for all operations (events) of interest

  // Symbolic constants for states
  private final TCPClosed tcpclosedstate = new TCPClosed;
  private final TCPOpen tcpopenstate = new TCPOpen;
  private final TCPEstablished tcpestablishedstate = new

          TCPEstablished;
  private Parameters parameters;
```

Java Example of the State

```
public static TCPConnection start (Parameters p) {
  TCPConnection t = new TCPConnection();
  t. parameters = p;
  return t.TCPClosed;

public TCPConnection processEvent (int event, Parameters p) {
  // this method should never be called. it should be implemented in the
          concrete subclasses.
  throw new IllegalAccessError ();
  }

protected Boolean enter () { }
  // this method is called when this object becomes the current state.
          it returns a Boolean to indicate if the method was successful.
```

Java Example of the State Pattern (cont.)

```
private class TCPClosed extends TCPState {
  // responds to a given event.  always returns the next state to be used.

public TCPState processEvent (int event, Parameters p) {
  switch (event) {
  case Open:
          if (TCPOpen.enter (p))
                  return tcpestablishedstate;
  case Close:
          { }   // and similarly for other cases
  protected Boolean enter (Parameters p) {
          // do whatever it takes to open a TCPConnection; return Boolean.
          }
  }// class TCPClosed
}// class TCPState
```

Design Decisions for the Java

- 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 Java

- how/when are the state objects created? when the start method is invoked for TCPState (the instance variables are initialized to point to new objects) how are they addressed? in instance variables of TCPState
- 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? in the processEvent method…the case statement based on event
- is “TCPState” an interface? an abstract class? or a concrete class? TCPState is a concrete class - with instance variables & method bodies
- where will the actual methods (where the work is actually accomplished) be performed? in the concrete states? in the TCPConnection? in the “enter” method for each concrete state

C++ Example of TCP Connection

pp. 309-312, Design patterns book

```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
        TCPConnection();
        void ActiveOpen ();                void Acknowledge ();
        void PassiveOpen ();               void Synchronize ();
        void Close ();                     void Send ();
private:
        friend class TCPState;
        void ChangeState (TCPState*);
private:
        TCPState* _state        };
```

```
class TCPState {
public:
        virtual void Transmit (TCPConnection*, TCPOctetStream*);
        virtual void ActiveOpen (TCPConnection*);
        virtual void PassiveOpen (TCPConnection*);
        virtual void Close (TCPConnection*);
        virtual void Synchronize (TCPConnection*);
        virtual void Acknowledge (TCPConnection*);
        virtual void Send (TCPConnection*);
protected:
        void ChangeState (TCPConnection*, TCPState*);
};
```

```
TCPConnection::TCPConnection () {
        _state = TCPClosed::Instance();    }

void TCPConnection::ChangeState (TCPState* s) {
        _state = s;        }

void TCPConnection::ActiveOpen () {
        _state->ActiveOpen(this);   }

void TCPConnetion::PassiveOpen () {
        _state->PassiveOpen(this);   }

void TCPConnection::Close () {
        _state->Close(this);  }

void TCPConnection::Acknowledge () {
        _state->Acknowledge(this);   }
......
```

Implementation of TCPState

```
// these implementations provide the default behavior

void TCPState::Transmit(TCPConnection*, TCPOctetStream*)  { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s)
        { t->ChangeState (s);  }
```

```
class TCPEstablished : public TCPState {
public:
        static TCPState* Instance ();

        virtual void Transmit (TCPConnection*, TCPOctetStream*);
        virtual void Close (TCPConnection*);    };

class TCPListen : Public TCPState {
public:
        static TCPState* Instance ();

        virtual void Send (TCPConnection*);

class TCPClosed : Public TCPState {
public:
        static TCPState* Instance();
        virtual void ActiveOpen(TCPConnection*);
        virtual void PassiveOpen (TCPConnection*);    };
```

```
void TCPClosed::ActiveOpen (TCPConnection* t){
        // send SYN, receive SYN, ACK, etc.
        ChangeState(t, TCPEstablished::Instance());  }

void TCPClosed::PassiveOpen (TCPConnection* t) {
        ChangeState (T, TCPListen::Instance());  }

void TCPEstablished::Close (TcPConnection* t)  {
        // send FIN, receive ACK of FIN
        ChangeState(t, TCPListen::Instance());  }

void TCPEstablished::Transmit
                (TCPConnection* t, TCPOctetStream* o)  {
        t->ProcessOctet(o);   }

void TCPListen::Send (TCPConnection* t)  {
        // send SYN, receive SYN, ACK, etc.
        ChangeState (t, TCPEstablished::Instance());  }
```

How are the individual states created? referenced?

```
void TCPClosed::PassiveOpen (TCPConnection* t) {
    ChangeState (T, TCPListen::Instance()); }
```

every time a connection changes to another state, (it looks like) a new instance of the state is created!

Do we really need all of these states?