Design Patterns 2

Tuesday, November 13

Announcements

Sprint 4 overview

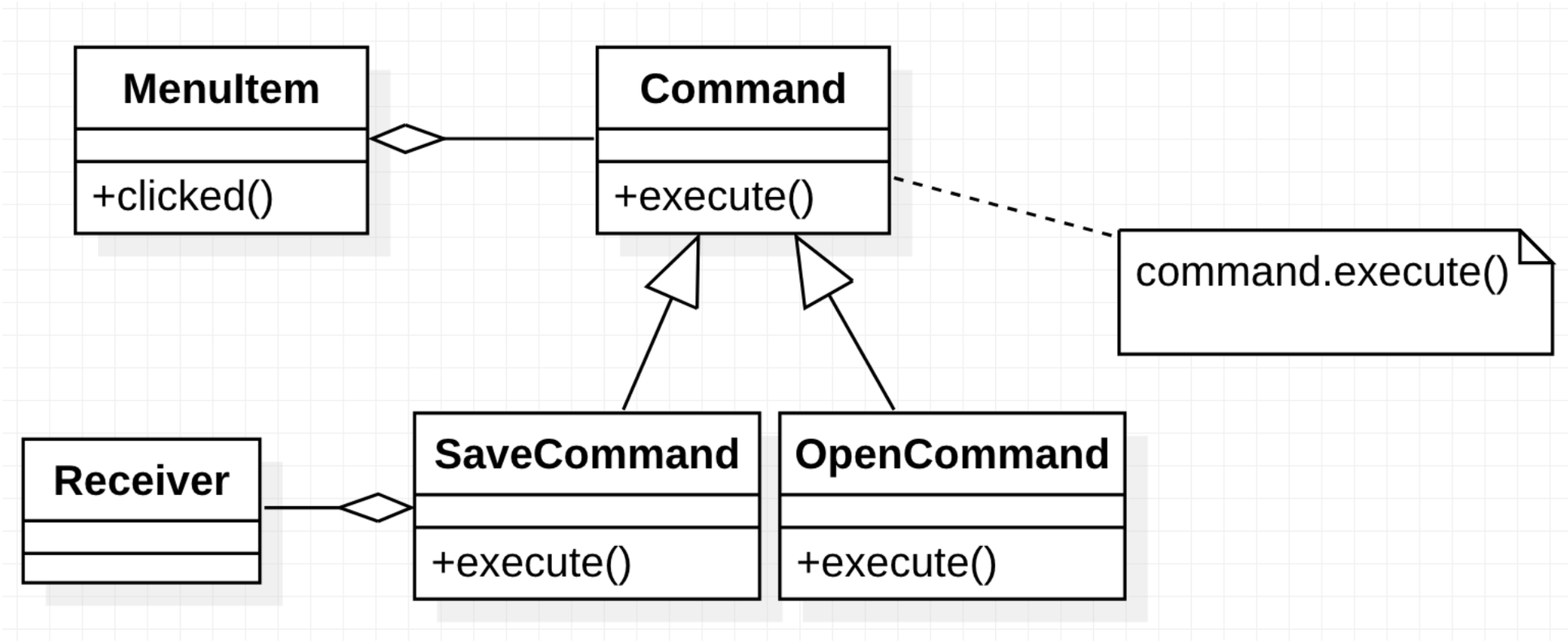
Tomorrow, short intro from the InnovationX center

Command

Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Motivation: When you need to issue a request, without knowing anything about the receiver.

Command



Command

The caller does not know which command subclass it uses

Command subclasses stores the receiver of the request, and perform operations.

Consequences

It allows you to **parameterize** objects with an action to perform. It's the **callback** of the OO world.

Allows for **undo and redo.** The command needs to support an `unexecute` operation.

Allows for a **transaction log.**

Undo and redo

Hysteresis can be a problem in ensuring a reliable and semantics preserving undo/redo mechanism.

Sometimes, the commands need to store extra info, to guarantee that object are restored to their original state.

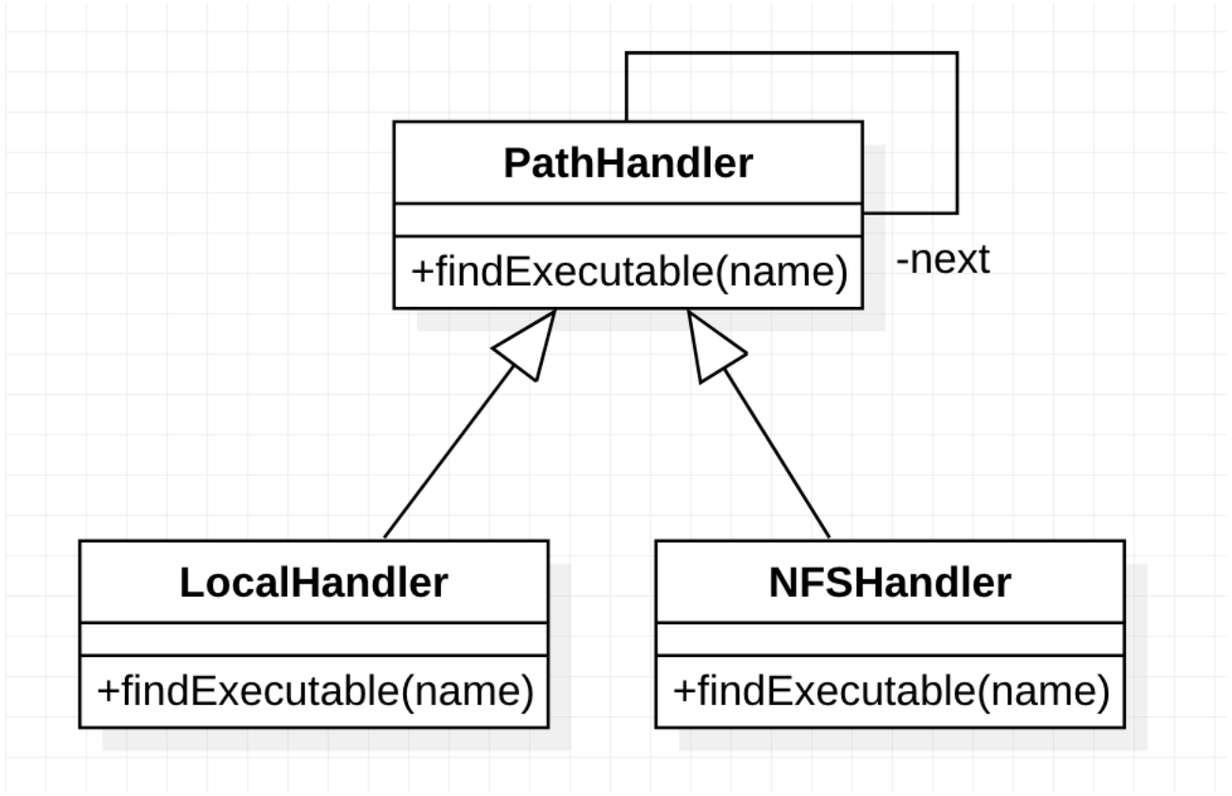
When using a history-list, you might need to create a new command object for each invocation.

Chain of Responsibility

Intent: Avoid coupling the sender of a request with its receiver by giving more than one object a chance to handle the request.

Motivation: Find the right executable to call, from a list of valid paths ($PATH).

Chain of Responsibility



```
public class PathHandler {
 private PathHandler next;
 public File findExecutable(String name) {
   return next.findExecutable(name)
  }
 }
}
```

```
public class LocalHandler extends PH {
 public File findExecutable(String name) {
  if (canHandle()) {…}
  else super.findExecutable(name);
 }
}
```

```
public class PathHandler {
 private PathHandler next;
 public File findExecutable(String name) {
   return next.findExecutable(name)
  }
 }
}
```

```
public class Local
 public File findE
  if (canHandle()) {…}
  else super.findExecutable(name);
 }
}
```

The default implementations asks **the next handler** in the chain to fulfill the request

```
public class PathHandler {
 private PathHandler next;
 public File findExecutable(String name) {
   return next.findExecutable(name)
 }
```

Each client handles the request, if they can. Otherwise, it asks the next handler in the chain.

```
public class LocalHandler extends PH {
 public File findExecutable(String name) {
  if (canHandle()) {…}
  else super.findExecutable(name);
 }
}
```

```
public class PathHandler {
 private PathHandler next;
 public File findExecutable(String name) {
   return next.findExecutable(name)
  }
 }
}
```

```
public class LocalHandler extends PH {
 public File findExecutable(String name) {
  if (canHandle()) {…}
  else super.findExecutable(name);
 }
}
```

Chain of Responsibility

You can easily **extend** the event handling.

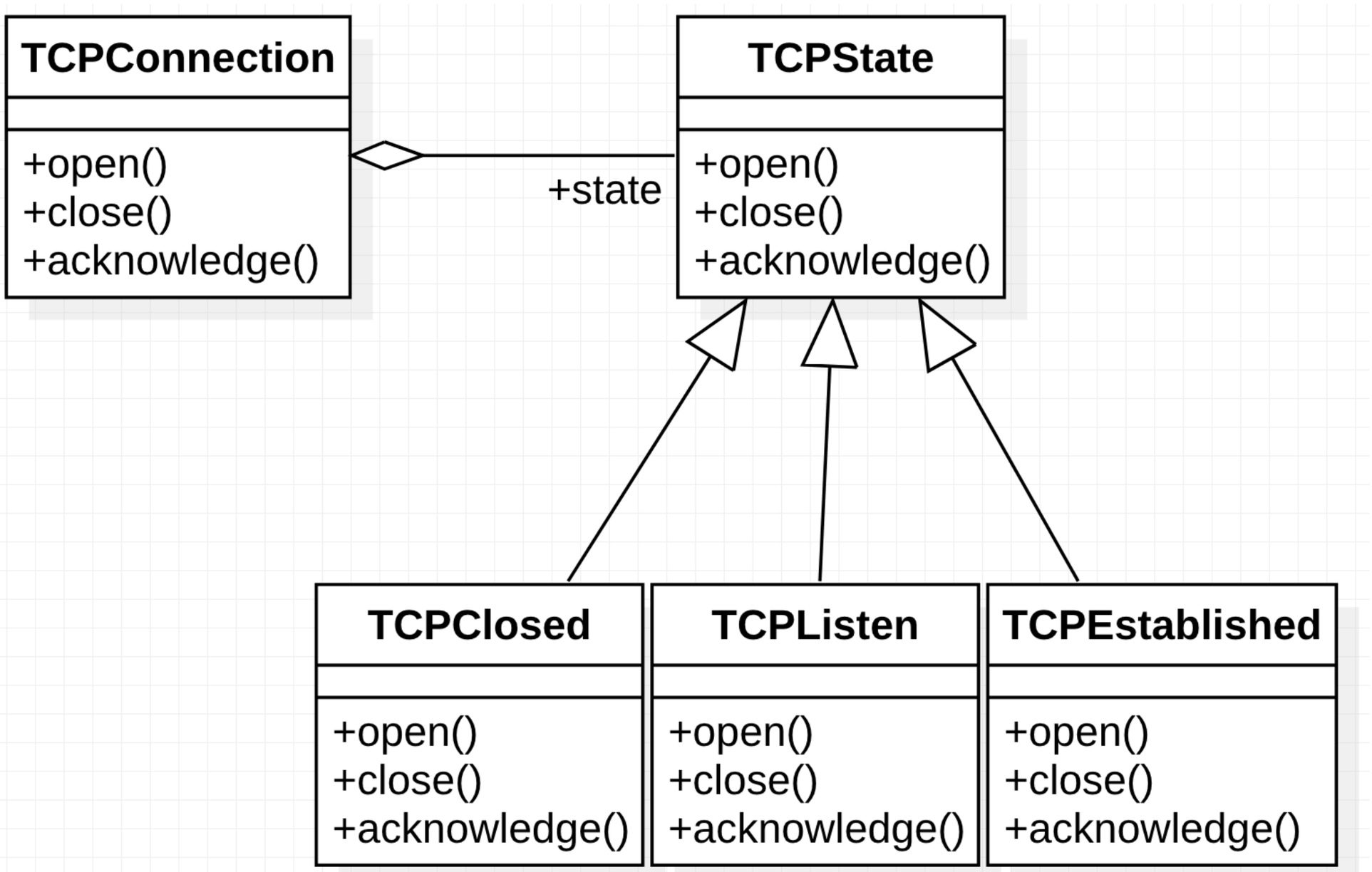
The event handling logic is **dynamic.**

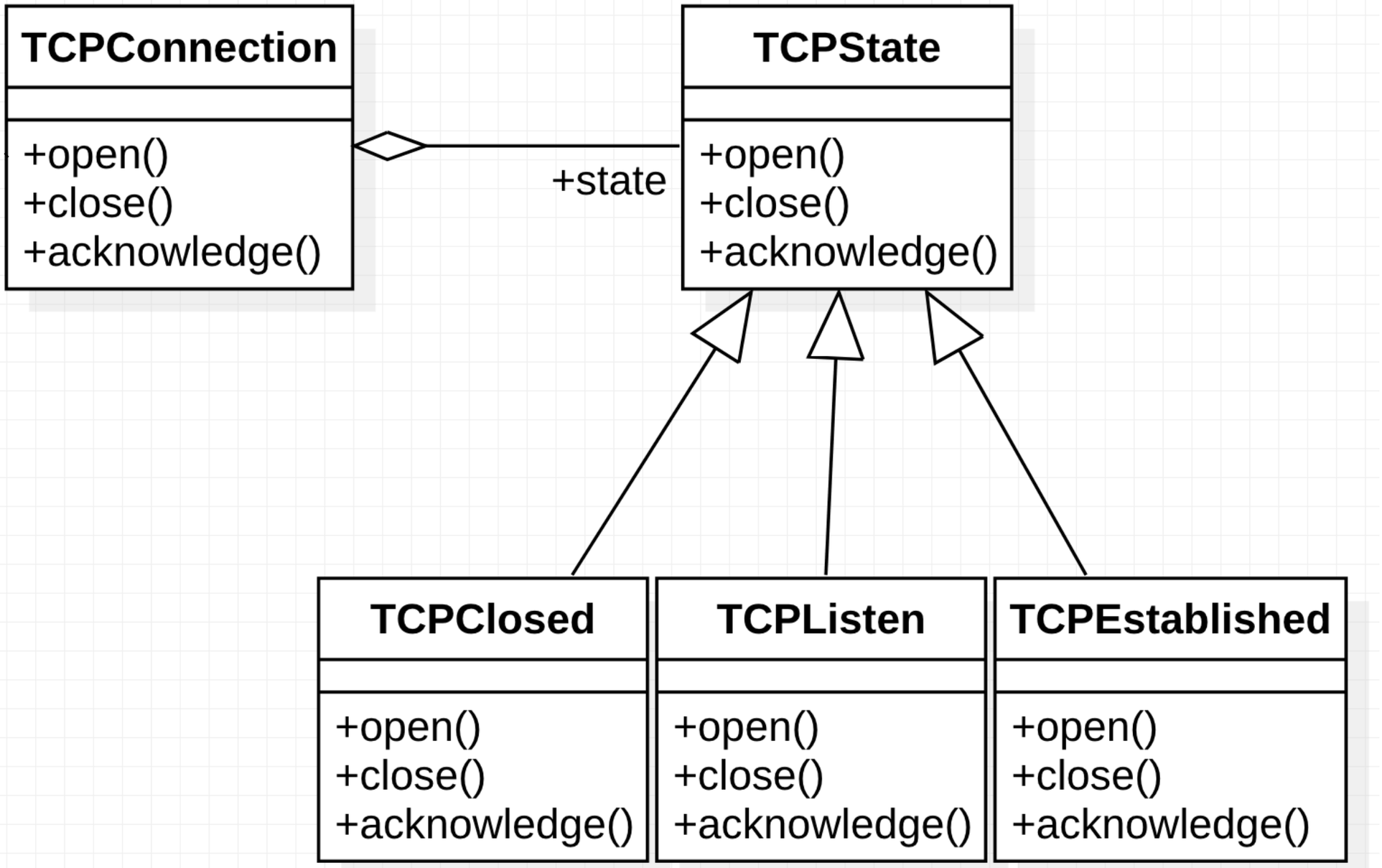
Receipt of the event it not guaranteed. If you want the request to be handled, you need to properly configure the chain.

State

Motivation: Allow an object to change its behavior when its state changes.

Intent: Consider a TCP connection. It has 3 states: Established, Listening and Closed. The effect of an **open** request depends on the connection state.






```
public void open() {
 state.open()
}
```

State

The TCPConnection is sometimes referred to as a **context.** The context passes information, or itself, to the state object, so it can fulfill its responsibility.

The context serves as the primary interface for the clients.

Handling state changes

All state related behavior is encapsulated in the state object.

This avoids complicated conditionals, and state variables.

State changes can be decided by the context, or by the state classes themselves.

Consequences

It **localizes** state specific behavior, and transition logic between states

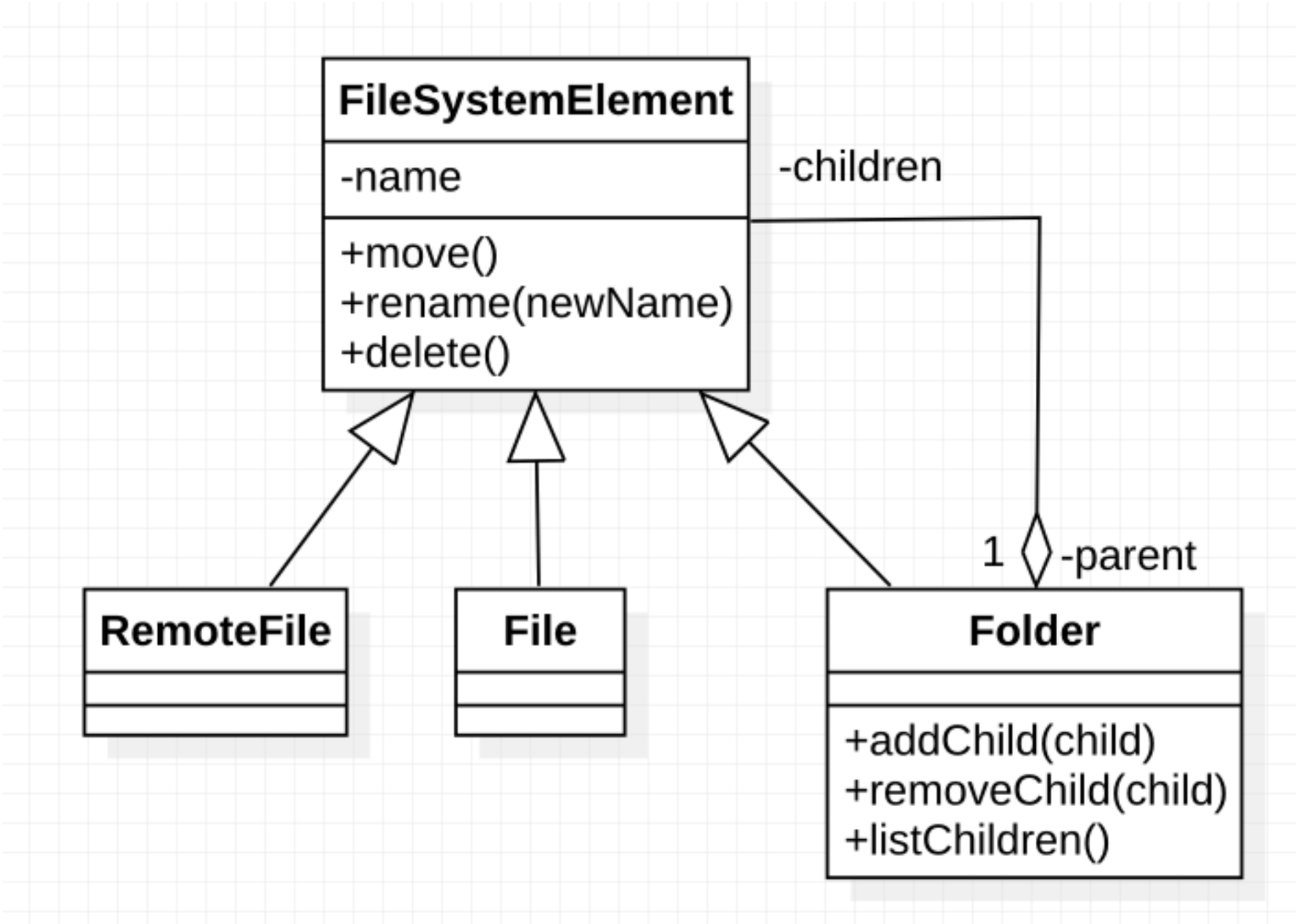
State transitions are **explicit.**

State objects can be **shared.**

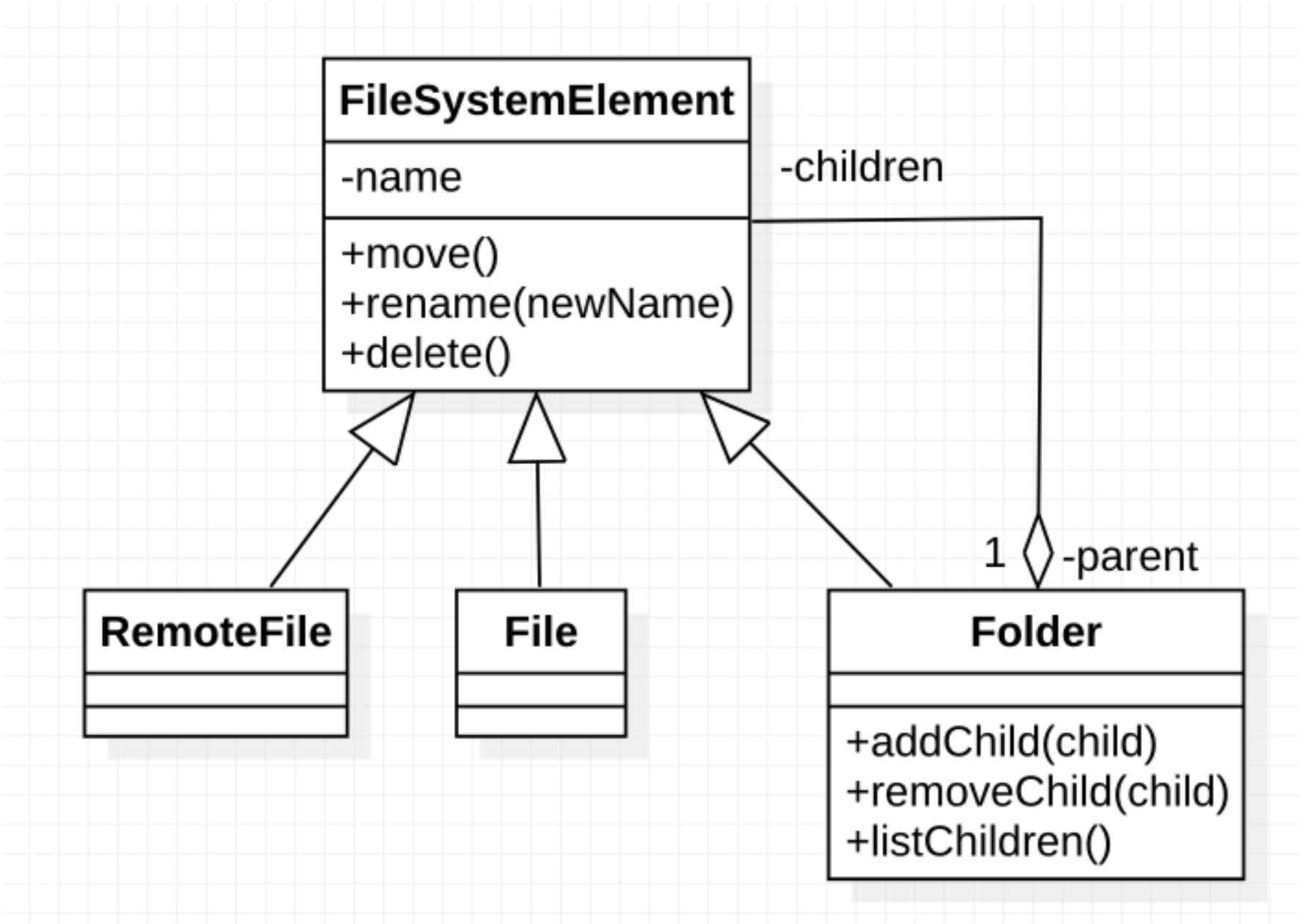
Visitor

Intent: Represent an operation to be performed on the elements of an object structure. You can define new operations without changing the classes of the elements on which they operate.

Motivation: When trying to operate over elements of a Composite structure. If you need to support many operations, that differ depending on the concrete node type.



What if I want to add extra operations to this structure?



Visitor

Adding operations to the Composite classes can **violate SRP;** suddenly the classes are doing a lot more work.

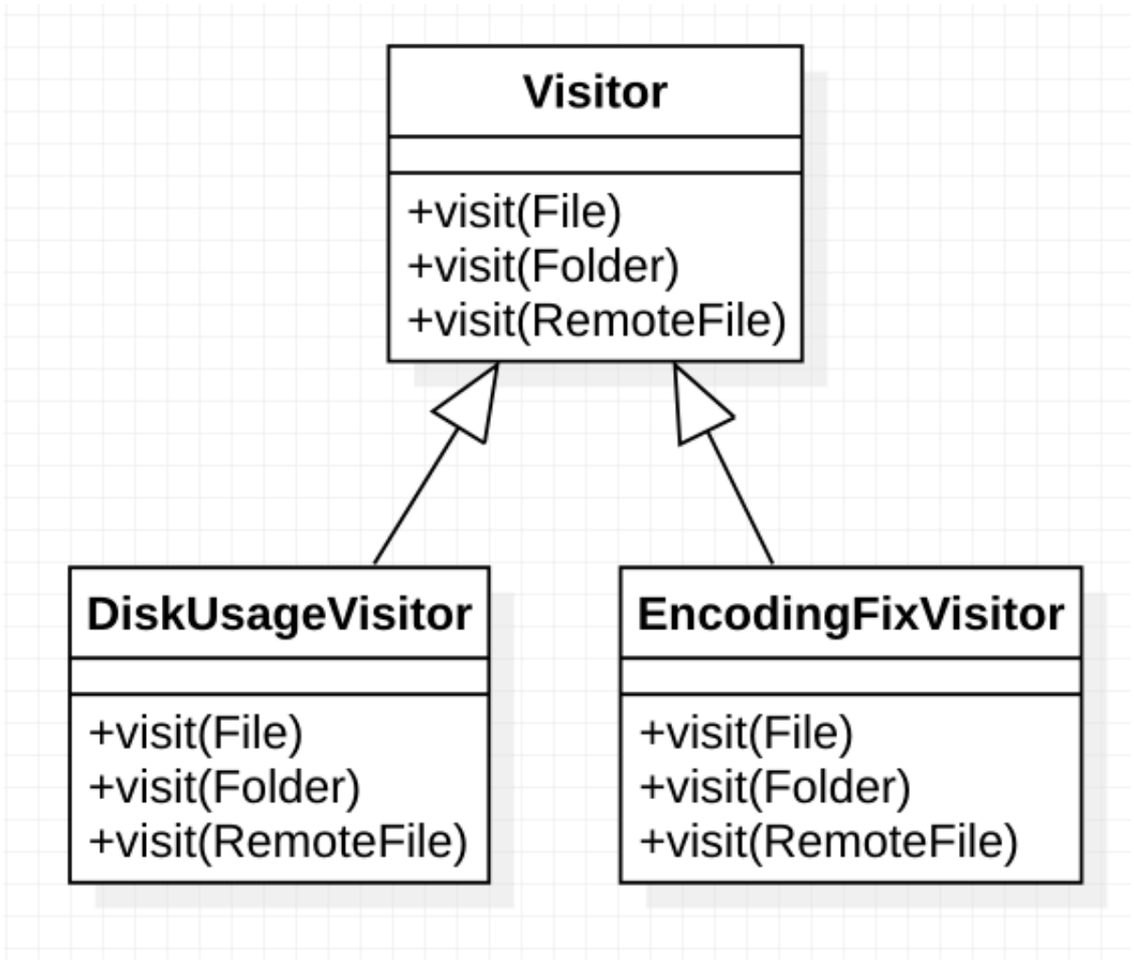
If adding operations is common, changing the classes **breaks OCP.**

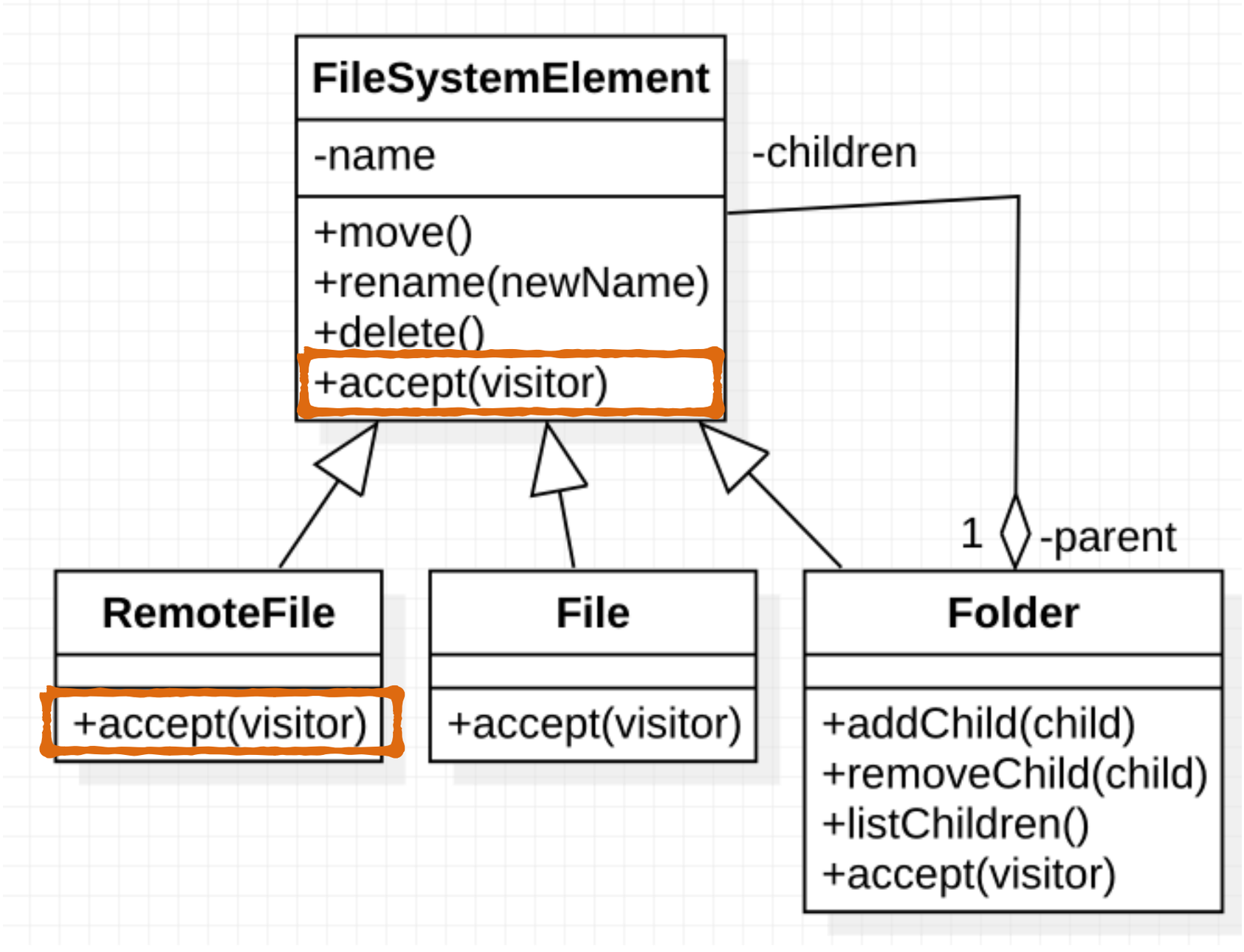
Visitor

Can we add operations, simply by **adding** a new class?

Visitor

Can we add operations, simply by **adding** a new class?





Using visitor

A client that uses the Visitor pattern must create a ConcreteVisitor (e.g. `DiskUsageVisitor`) and traverse the object structure, visiting each node.

When an element is visited, it calls the `visit` operation that corresponds to its class, and passes itself as an argument.

```
public class ConcVisitor extends Visitor {
 public void visit(File f) {…}
 public void visit(Folder f) {…}
 public void visit(RemoteFile f) {…}
}
```

```
FileSystemElement el = …;
Visitor v = new ConcVisitor();
el.accept(v);
// get the results from the visitor
```

```
public class ConcVisitor extends Visitor {
 public void visit(File f) {…}
 public void visit(Folder f) {…}
 public void visit(RemoteFile f) {…}
}
```

```
FileSystemElement el =
Visitor v = new ConcVi
el.accept(v);
// get the results fro
```

The client must implement a Visitor class for the operation they wish to perform

```
public class ConcVisitor extends Visitor {
 public void visit(File f) {…}
 public void visit(Folder f) {…}
 public void visit(RemoteFile f) {…}
}
```

```
FileSystemElement el = …;
Visitor v = new ConcVisitor();
el.accept(v);
// get the results from the visitor
```

Call `accept` on the root of your object structure.

```
public class ConcVisitor extends Visitor {
 public void visit(File f) {…}
 public void visit(Folder f) {…}
 public void visit(RemoteFile f) {…}
}
```

```
FileSystemElement el = …;
Visitor v = new ConcVisitor();
el.accept(v);
// get the results from the visitor
```

```
public class File extends FSE {
 public void accept(Visitor v) {
  v.visit(this);
 }
}
```

```
public class Folder extends FSE {
 List<FSE> children;
 public void accept(Visitor v) {
  for (FSE child : children){
   child.accept(v);
 }
 v.visit(this);
}
```

```java
public class File extends FSE {
 public void accept(Visitor v) {
  v.visit(this);
 }
}
```

```java
public class Folder
 List<FSE> children;
 public void accept(
  for (FSE child : children){
   child.accept(v);
 }
 v.visit(this);
}
```

Each element passes itself to the visitor. Polymorphism will guarantee that the right method is called.

```
public class File extends FSE {
 public void accept(Visitor v) {
  v.visit(this);
 }
}
```

For composite elements, you need to make sure that each child gets to accept the visitor.

```
ends FSE {

itor v) {
  for (FSE child : children){
   child.accept(v);
  }
 v.visit(this);
}
```

```
public class File extends FSE {
 public void accept(Visitor v) {
  v.visit(this);
 }
}
```

```
public class Folder extends FSE {
 List<FSE> children;
 public void accept(Visitor v) {
  for (FSE child : children){
   child.accept(v);
 }
 v.visit(this);
}
```

Consequences

Consequences

Visitor makes **adding new operations easy.**

Visitor makes **adding new elements hard.**

Visitors can **accumulate state.** You don't need to pass it along through method call, or have fields or global variables to keep track of it

Visitor **breaks encapsulation.**

Closing remarks

Design patterns form a vocabulary

Do you need to use all of them?

Don't use design patterns just because you can. Use them when you have a specific problem to solve.