SOLID Principles

- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Open/Closed Principle

- Classes should be
  - open for enhancements, while being
  - closed for modifications.

Benefits

- Easier maintenance
- Easier unit testing
- Reduced QE testing requirements
- Fewer branches in source-control

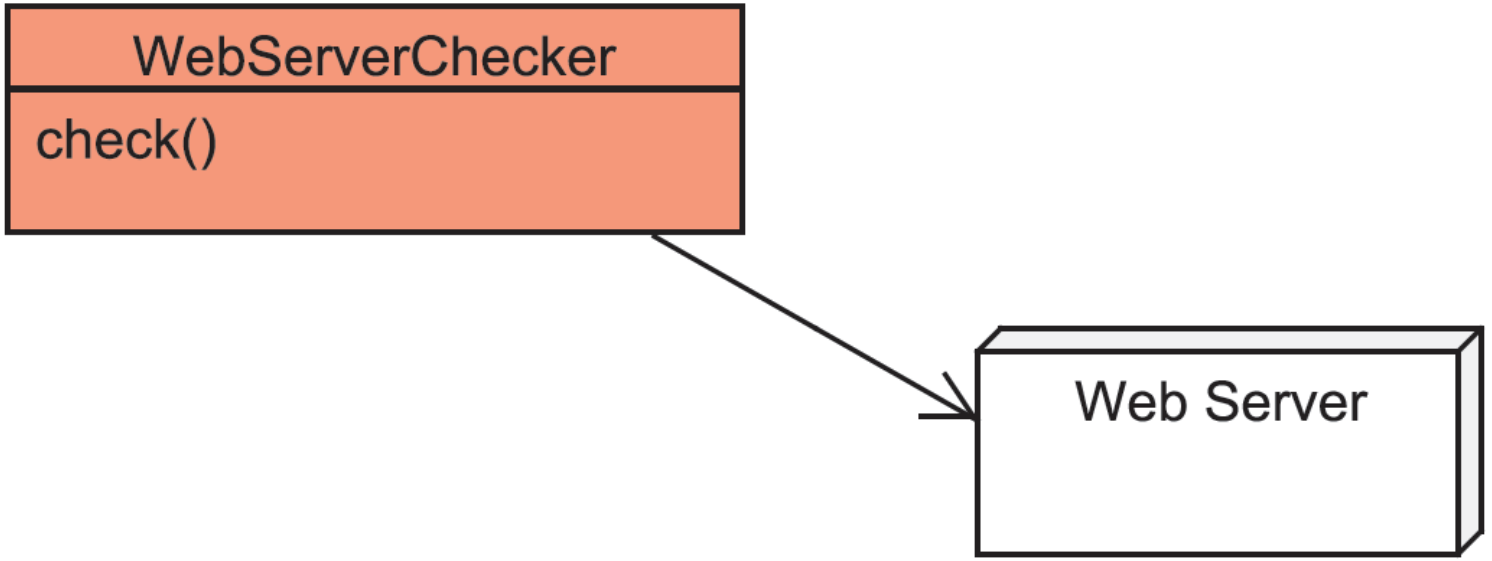
Patterns

- Abstract Server
- Bridge
- Composite
- State
- Template Method
- Command
- Factory
- Strategy

WebServerChecker Code

```
public class WebServerChecker
{
	private boolean unattended;
	public void check()
	{
		try
		{
			URLConnection web = new URL("http://10.204.166.23/TestPage.html").openConnection();
			web.setConnectTimeout(3000);
			web.connect();
			if (unattended)
				System.out.println("attended");
		}
		catch (Throwable e)
		{
			if (unattended)
				System.out.println("still unattended");
			else
			{
				System.out.println("unattended");
				unattended = true;
			}
		}
	}
}
```

WebServerChecker UML



WebServerChecker Problems

- Two algorithms in one
- Can't re-use alarm algorithm for other checks
- Must copy/paste to re-use
- Difficult to fix bugs
- Depends on web server so QE testing is harder
- Can't unit test

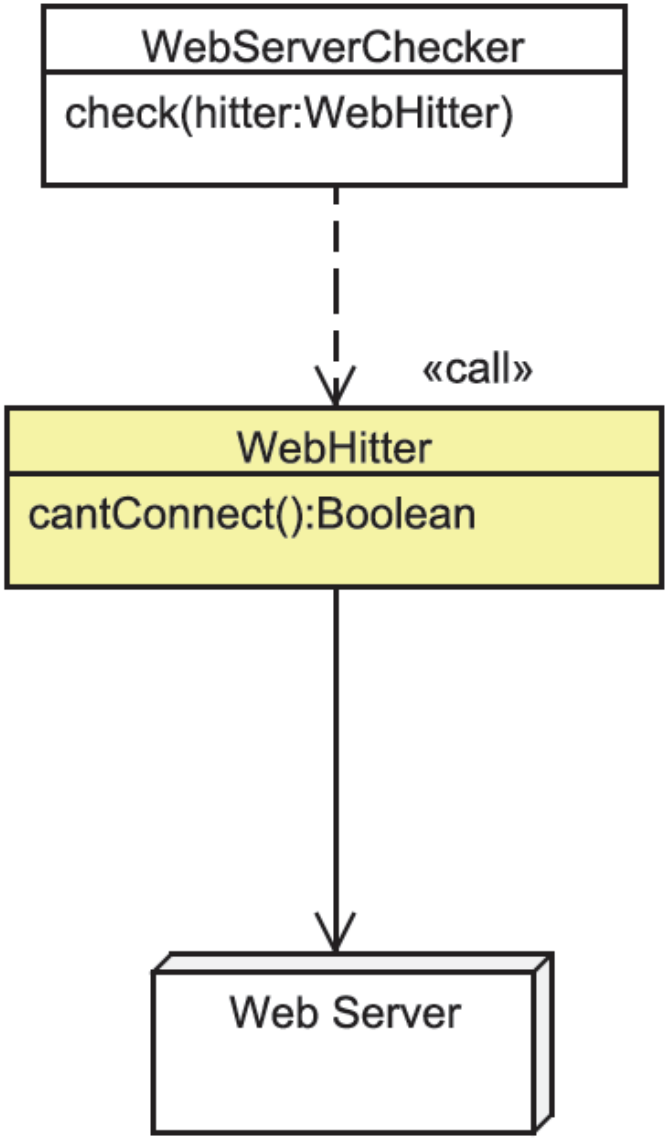
WebHitter Code

```
public class WebServerChecker
{
	private boolean unattended;
	public void check(WebHitter hitter)
	{
		final boolean alarm = hitter.cantConnect();

		if (alarm)
			if (unattended)
				System.out.println("still unattended");
			else
			{
				System.out.println("unattended");
				unattended = true;
			}
		else
			if (unattended)
				System.out.println("attended");
	}
}

public class WebHitter
{
	public boolean cantConnect()
	{
		try
		{
			URLConnection web = new URL("http://10.204.166.23/TestPage.html").openConnection();
			web.setConnectTimeout(3000);
			web.connect();
			return false;
		}
		catch (Throwable e)
		{
			return true;
		}
	}
}
```

WebHitter UML



WebHitter--Better

- Better; algorithms are separate
- But still depends on web server
- Still cannot unit test
- Still cannot re-use
- How do we decouple the alarm algorithm from the “checking” class it needs to use?

Abstract Server Pattern

- Algorithms should depend on abstractions with minimal interfaces, instead of specific implementations

Abstract Server Code

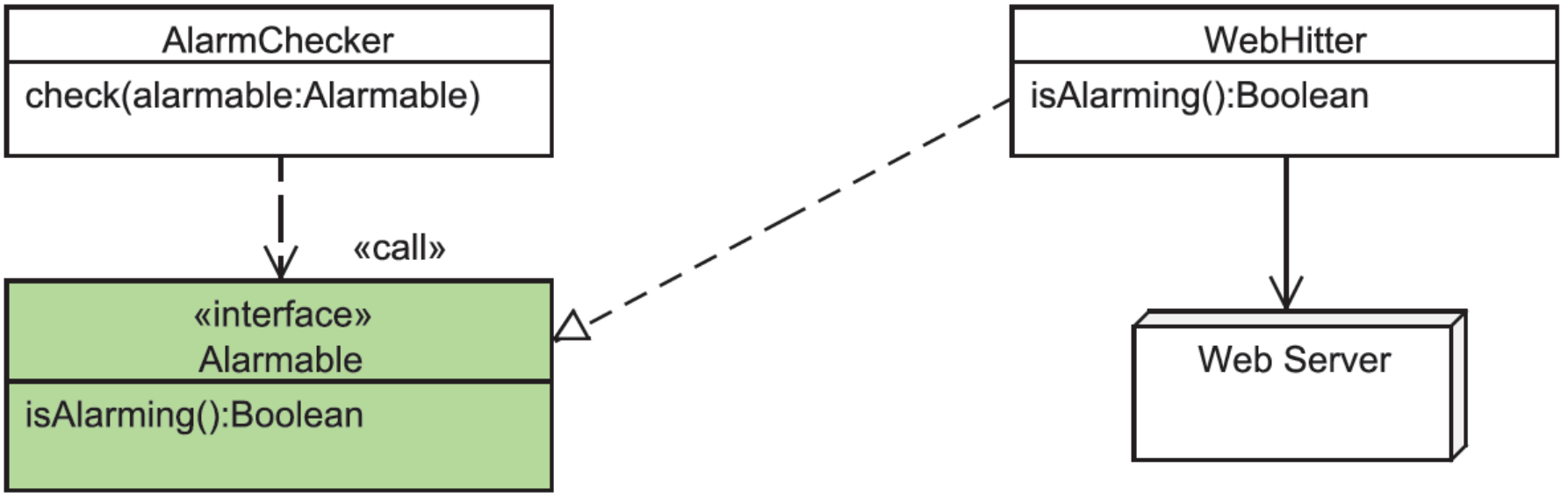
```
public class AlarmChecker
{
	private boolean unattended;
	public void check(Alarmable alarmable)
	{
		final boolean alarm = alarmable.isAlarming();

		if (alarm)
			if (unattended)
				System.out.println("still unattended");
			else
			{
				System.out.println("unattended");
				unattended = true;
			}
		else
			if (unattended)
				System.out.println("attended");
	}
}
```

```
public interface Alarmable
{
	boolean isAlarming();
}
```

```
public class WebHitter implements Alarmable
{
	@Override
	public boolean isAlarming()
	{
		try
		{
			URLConnection web = new URL("http://10.204.166.23/TestPage.html").openConnection();
			web.setConnectTimeout(3000);
			web.connect();
			return false;
		}
		catch (Throwable e)
		{
			return true;
		}
	}
}
```

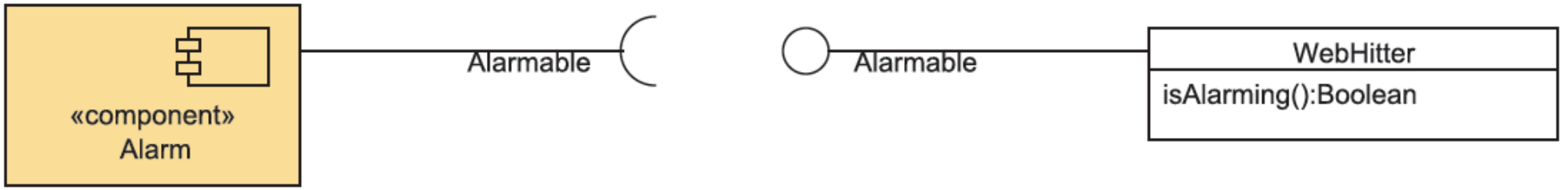
Abstract Server UML



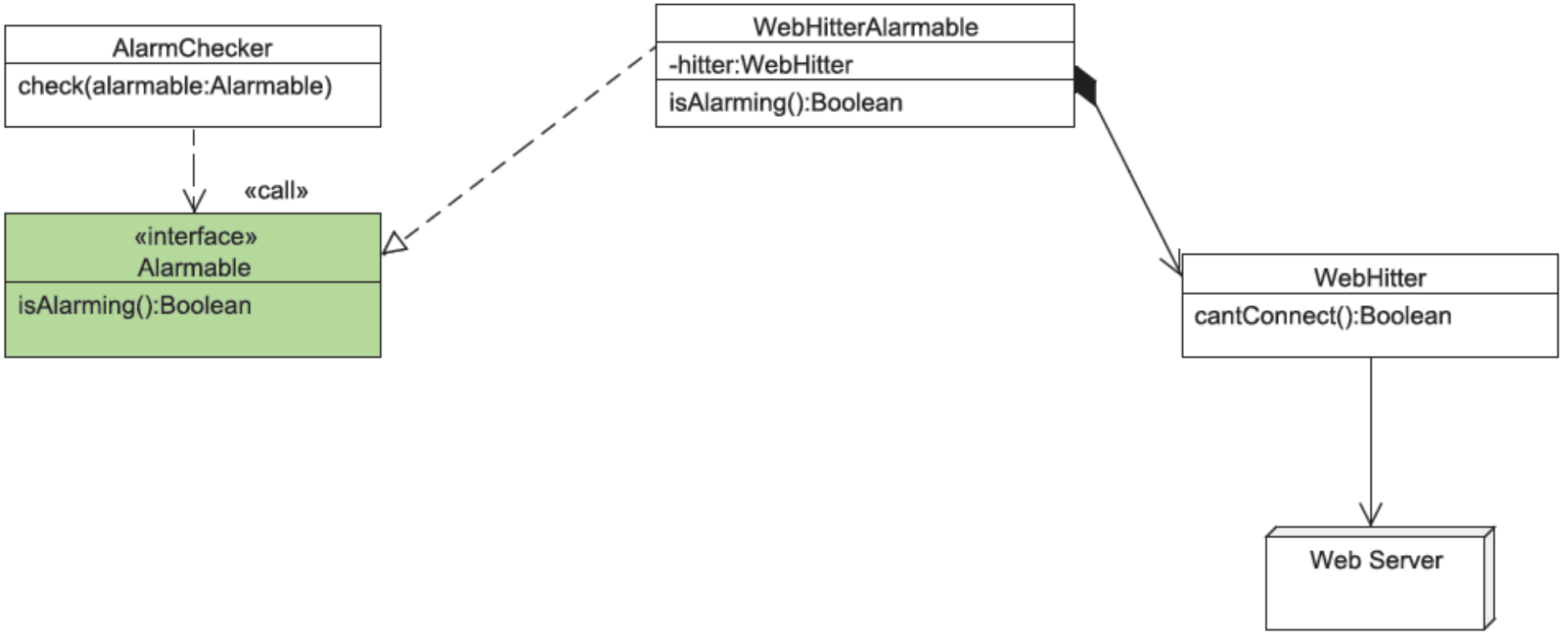
Abstract Server Pattern

- AlarmChecker depends only on Alarmable
- Alarmable is an abstraction (has no impl.)
- AlarmChecker can be unit tested
- Doesn't need web server
- Can be re-used for other checkers
- AlarmChecker doesn't require modification
- So doesn't require re-testing
- Alarm is a “component”
- Find the bug (It's easy to fix)

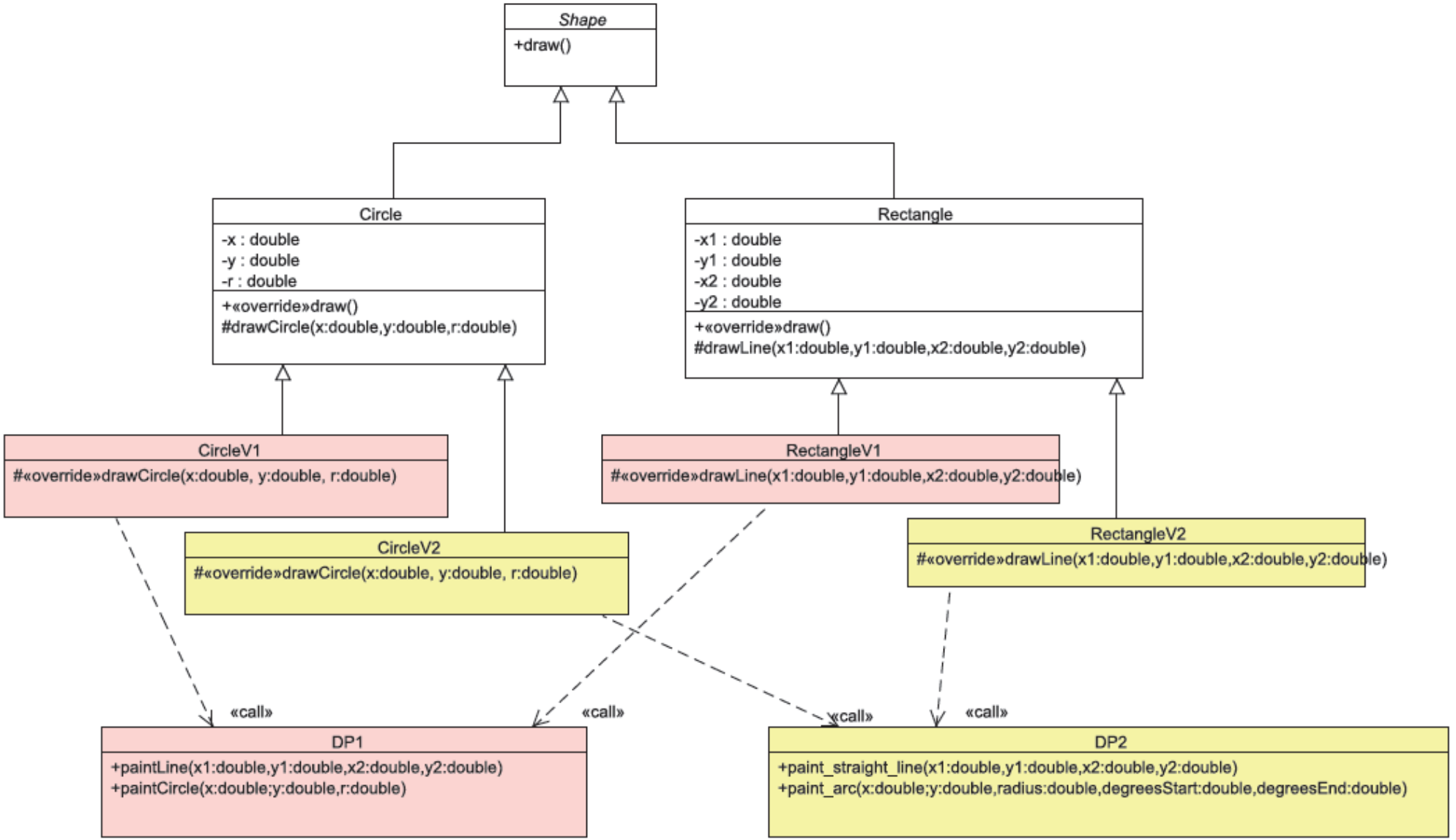
Abstract Server UML (Alt.)



Adapted Server



Problem: Degrees of Freedom



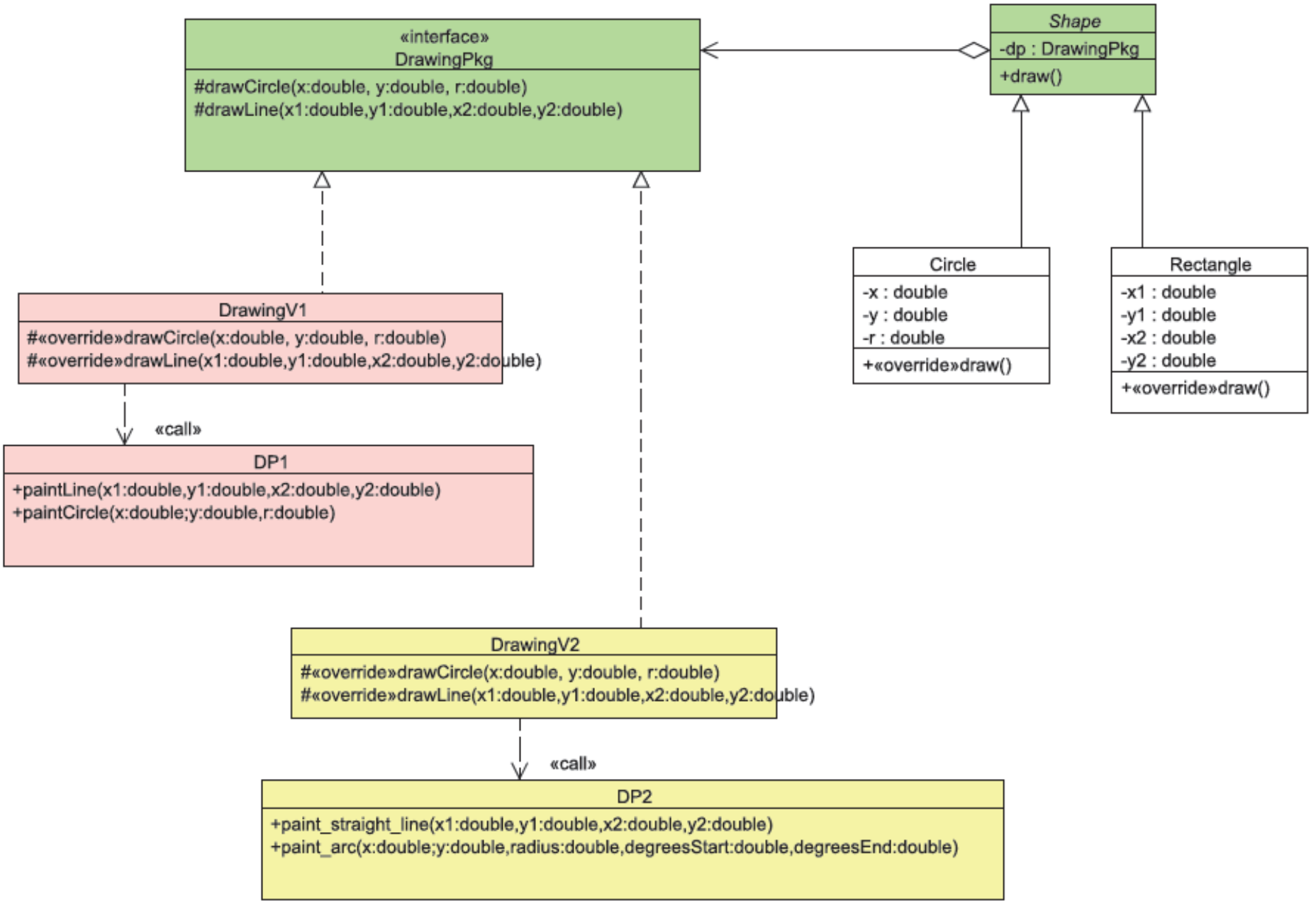
Problem

- Two degrees of freedom:
  - Shapes
  - Drawing packages
- n*m classes
- Too hard to add new shape or drawing package
- Not open for enhancements

Bridge Pattern

- Decouple an abstraction from its implementation so the two can vary independently (Gamma, et al.)
- Abstraction: shapes
- Implementation: how to draw them
- Decouple into two hierarchies

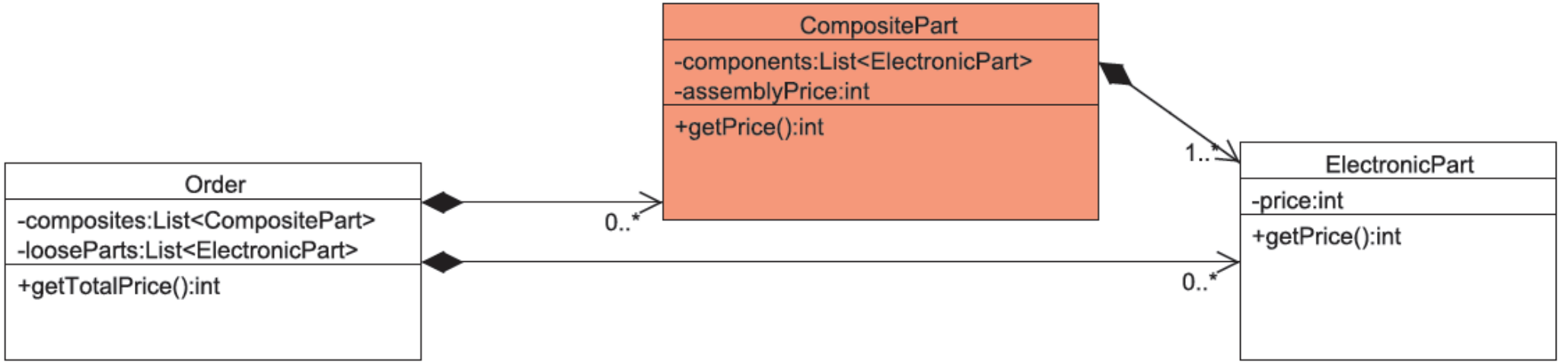
Bridge UML


«interface»
DrawingPkg
#drawCircle(x:double, y:double, r:double)
#drawLine(x1:double,y1:double,x2:double,y2:double)
Shape
-dp : DrawingPkg
+draw()
DrawingV1
#«override»drawCircle(x:double, y:double, r:double)
#«override»drawLine(x1:double,y1:double,x2:double,y2:double)
«call»
DP1
+paintLine(x1:double,y1:double,x2:double,y2:double)
+paintCircle(x:double;y:double,r:double)
Circle
-x : double
-y : double
-r : double
+«override»draw()
Rectangle
-x1 : double
-y1 : double
-x2 : double
-y2 : double
+«override»draw()
DrawingV2
#«override»drawCircle(x:double, y:double, r:double)
#«override»drawLine(x1:double,y1:double,x2:double,y2:double)
«call»
DP2
+paint_straight_line(x1:double,y1:double,x2:double,y2:double)
+paint_arc(x:double;y:double,radius:double,degreesStart:double,degreesEnd:double)

Bridge Pattern

- New hierarchy for drawing packages
- Eliminates “explosion” of classes
- Add new drawing pkg without affecting shapes
- Add new shape without affecting drawing pkgs
- Isolated changes: less testing

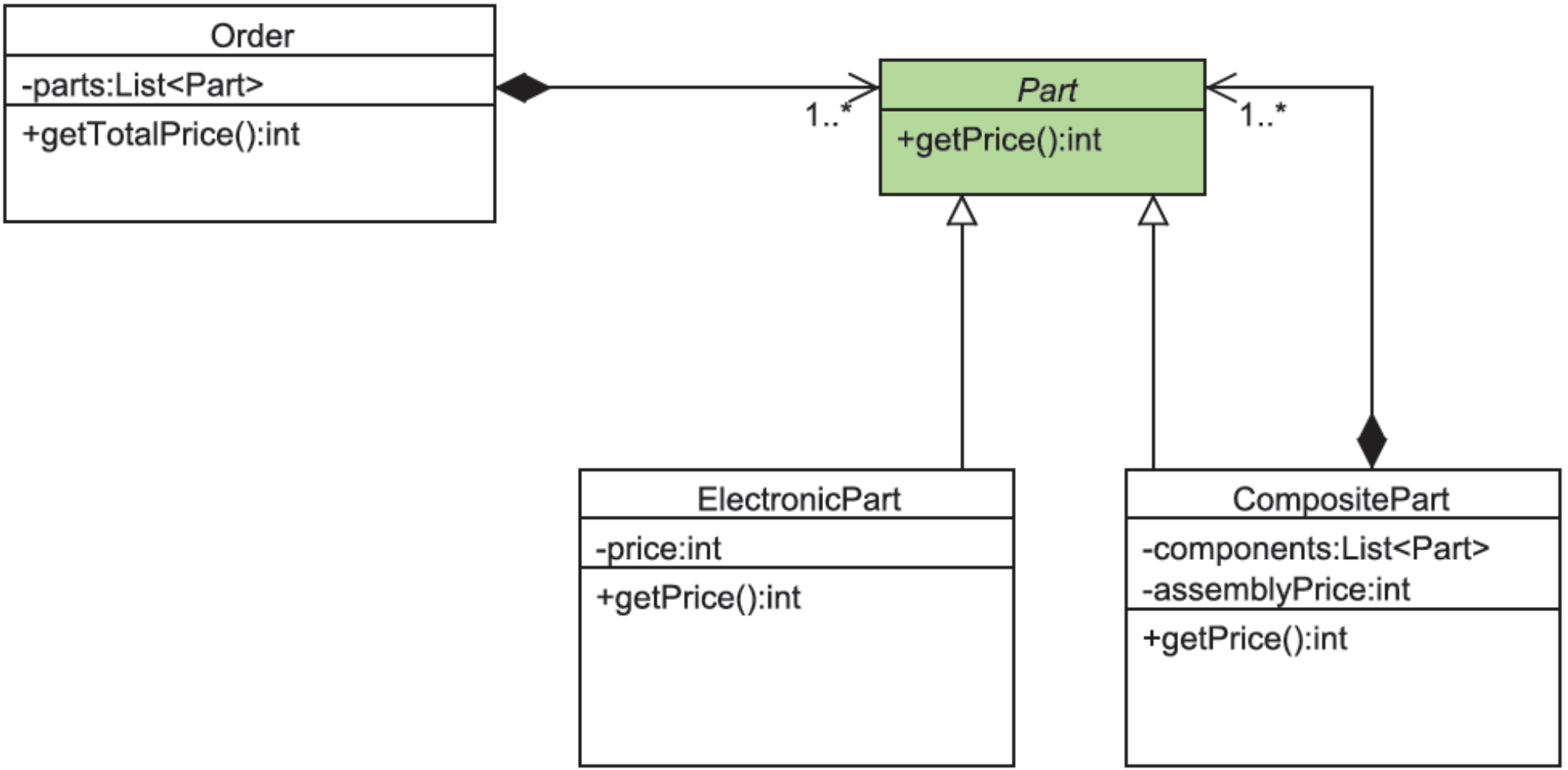
Composite Problem



Composite Problem

- Composite item treated differently than parts
- Forces clients to use them differently
- Doesn't isolate them from changes
- Doesn't allow composites of composites

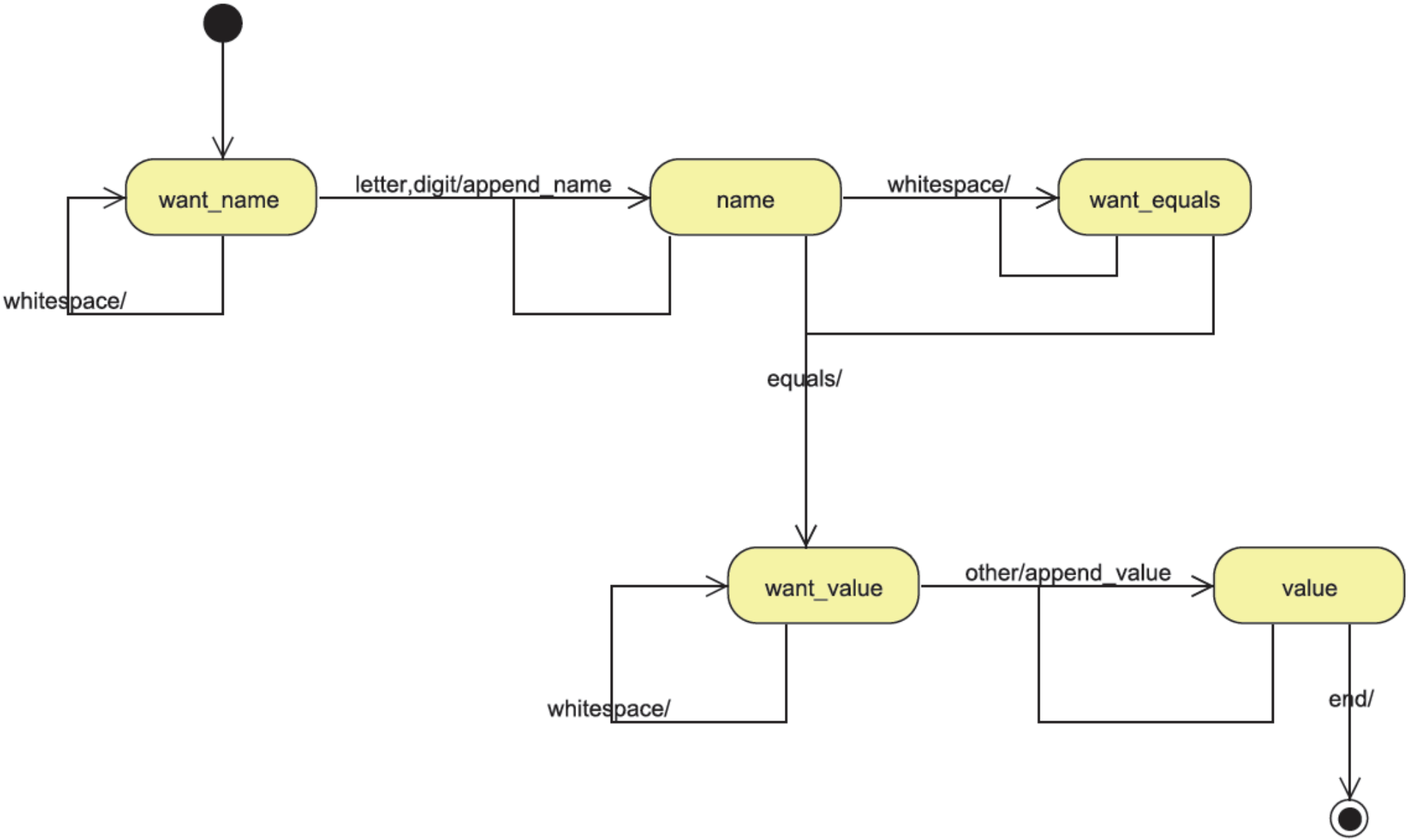
Composite Solution



Composite Pattern

- Composite item treated same as individuals
- Insulates client from differences
- Allows composites of composites

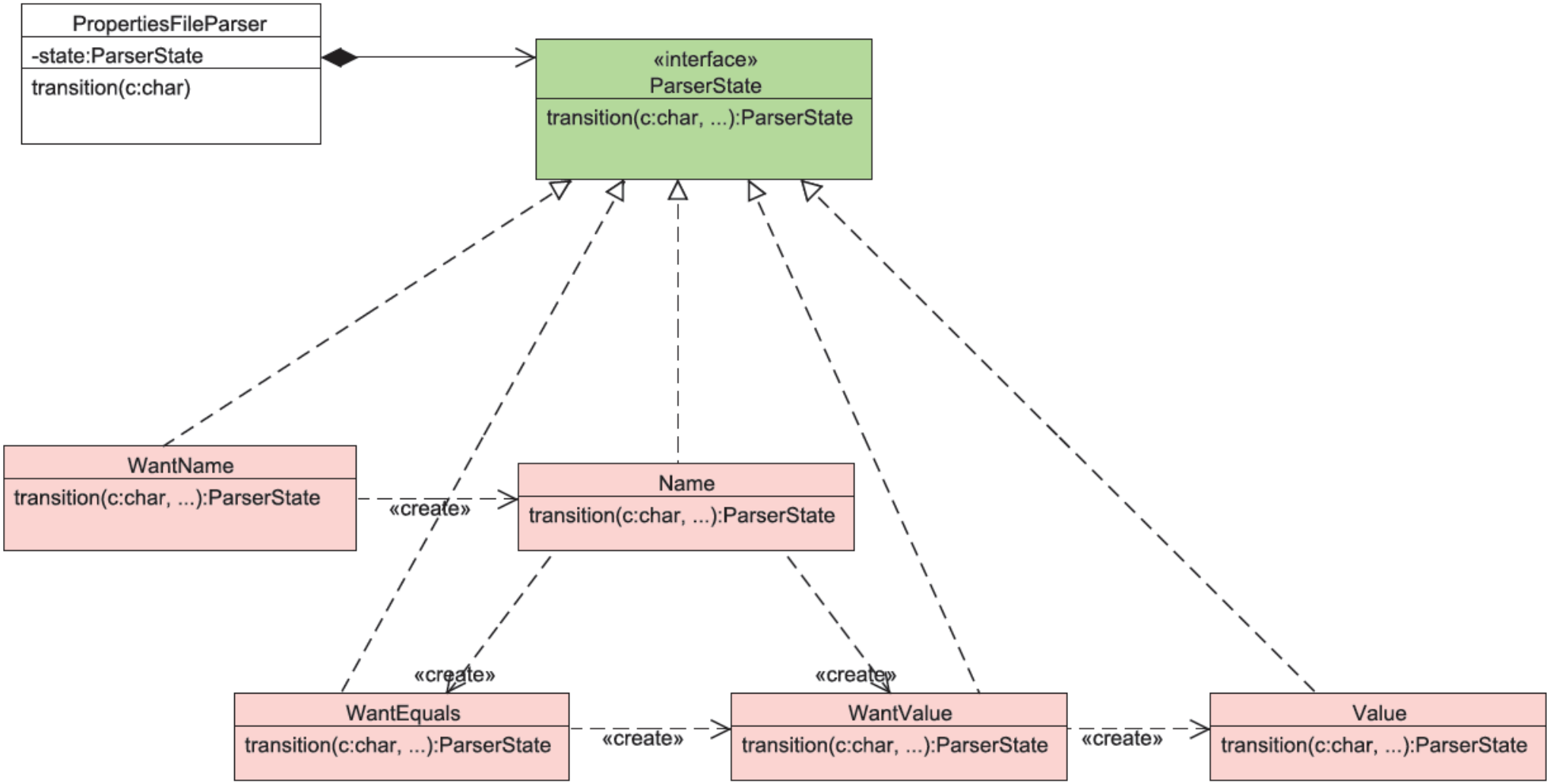
Parsing a Properties File



Traditional FSM

- Typically a big switch statement
- Individual states are not isolated from changes to other states
- Duplicate checks
- Hard to read

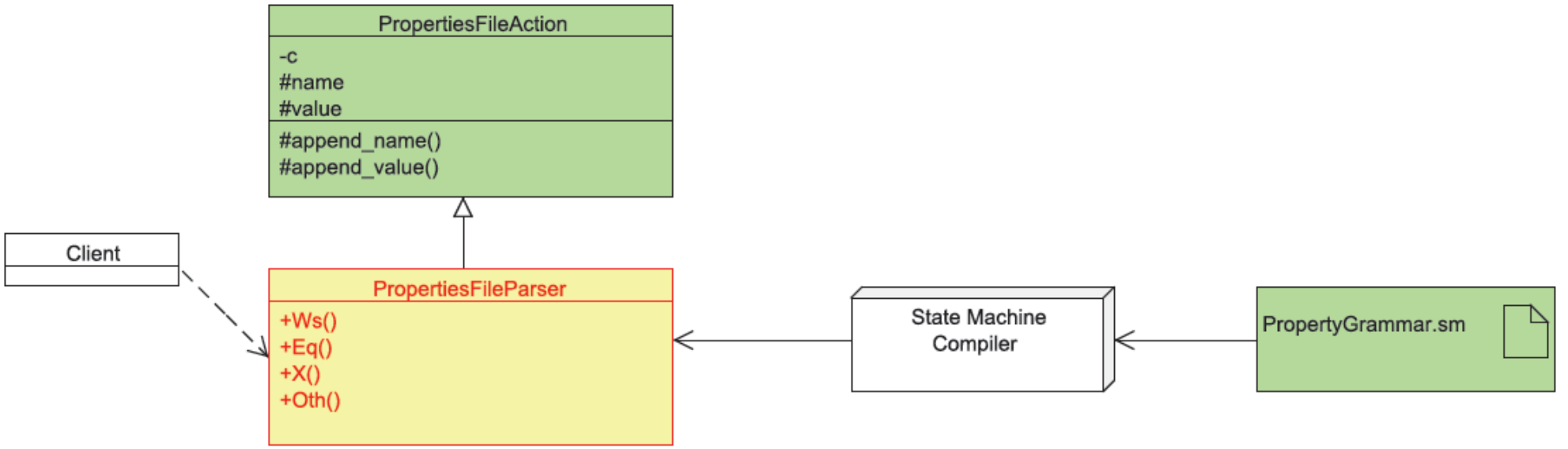
State Pattern--Better



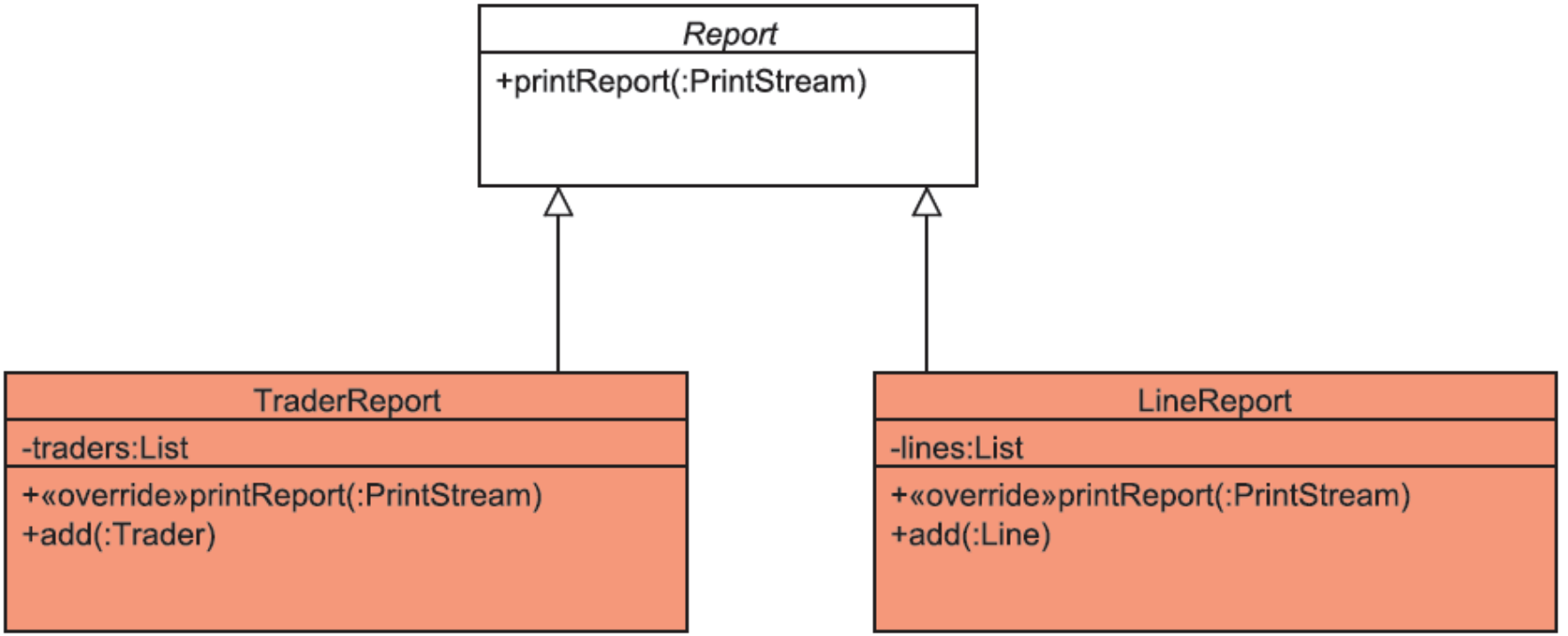
State Machine Compiler--Best

- Robert Martin, Object Mentor
- Removes “boilerplate” code
- Grammar file → SMC → Parser.java
- Changes are usually trivial
- Allows inheritance of states (super-states)

State Machine Compiler UML



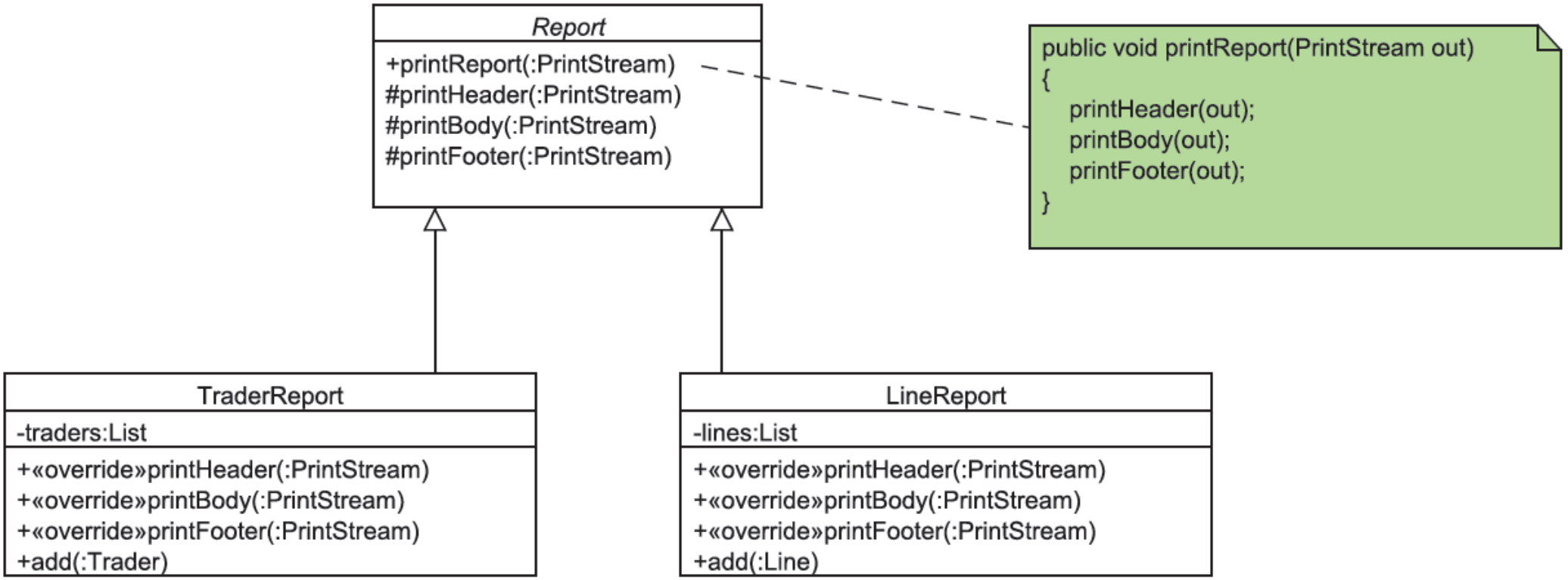
No Template--Problem



No Template

- Cannot change algorithm without touching all implementations
- Hard to read code
- Cannot test overall algorithm independently of each implementation

Template Method



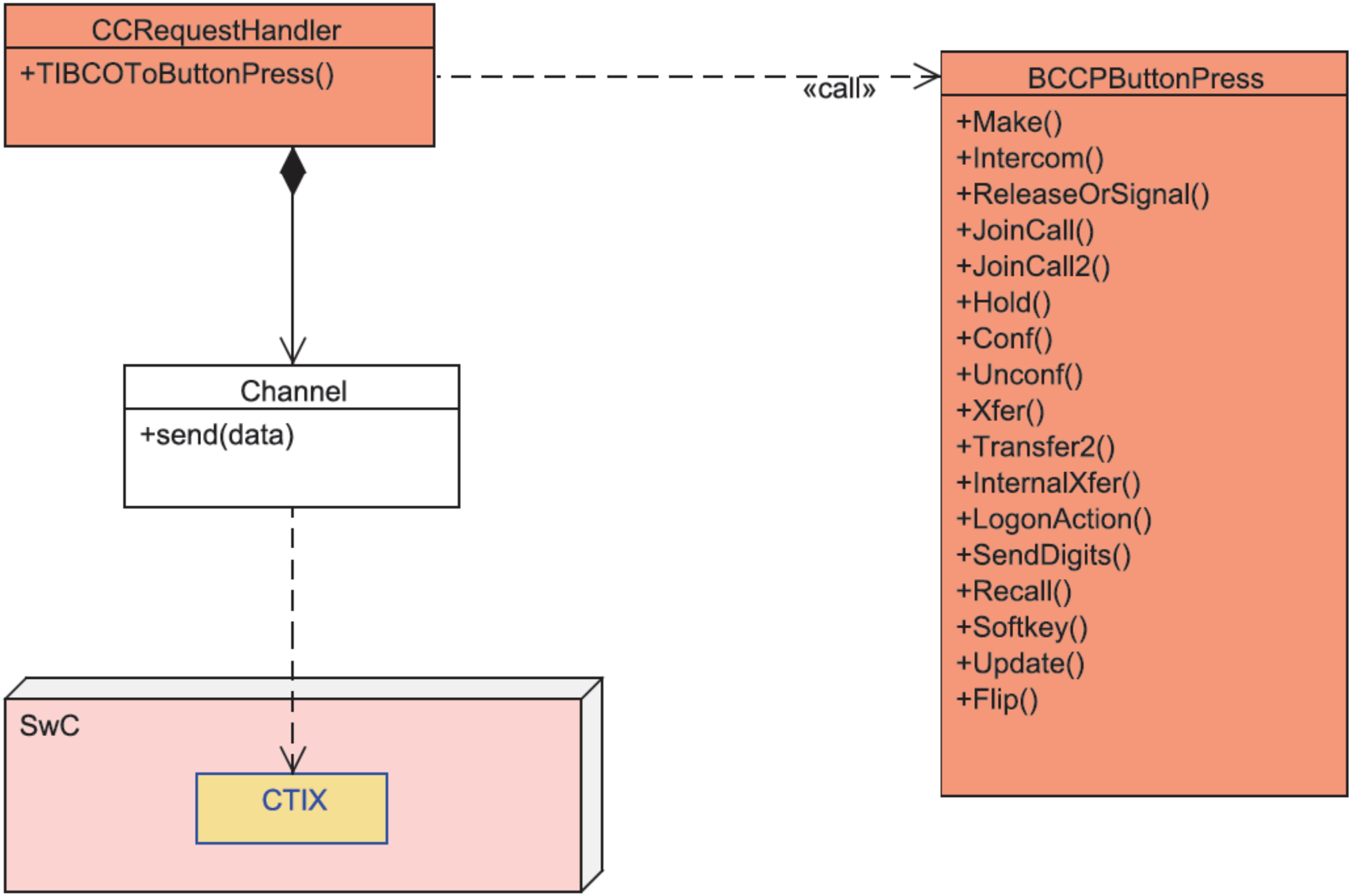
Template Method

- Search for the algorithm
- Sometimes it's hidden in the comments!
- Make it explicit in a “template” method
- Template method can be easily tested
- New subclasses can't break the template algorithm

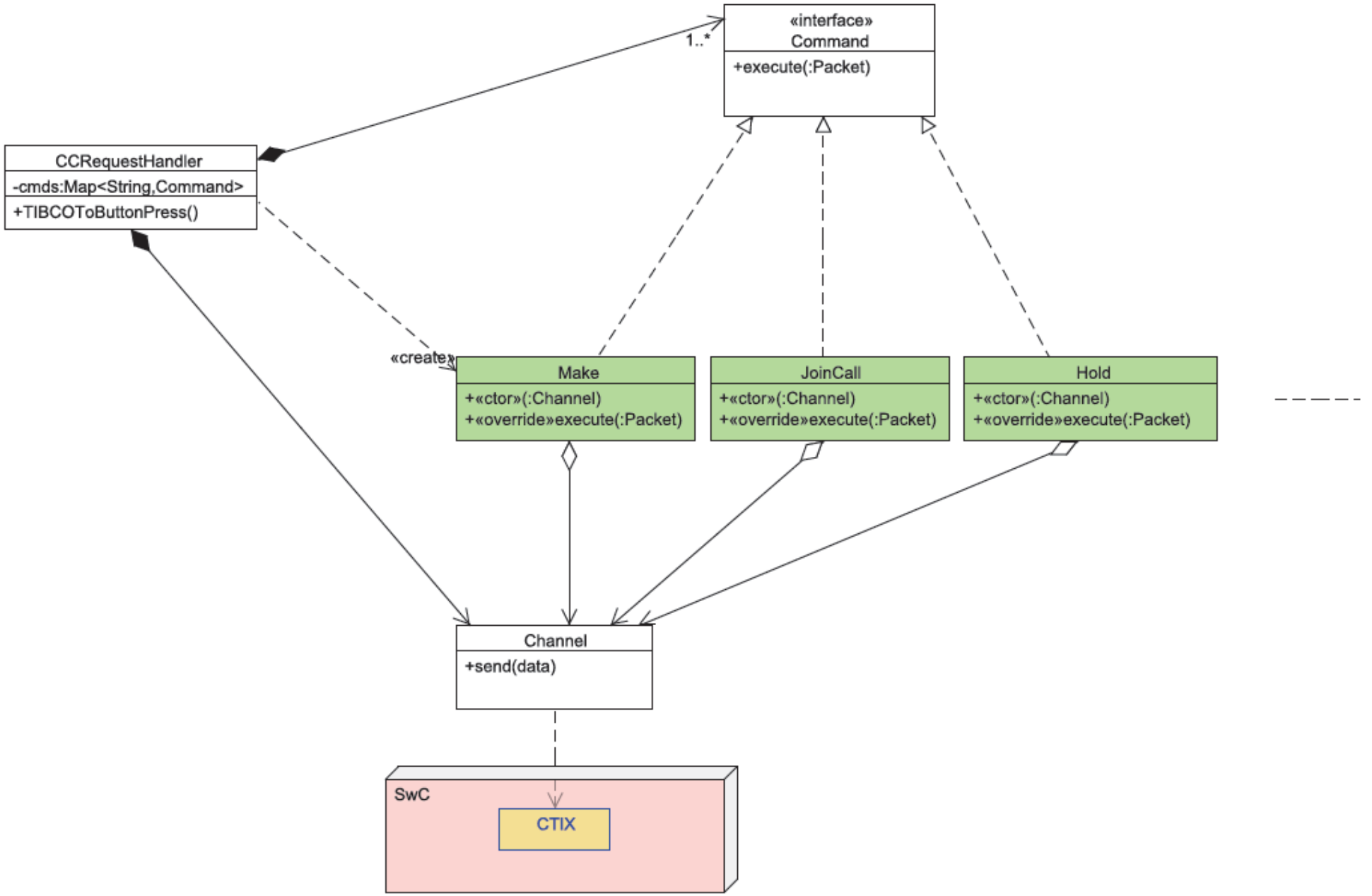
Command Handler

- Trade Central Server—Call Control
- Sends commands from clients to CTIX
- Dispatch algorithm is a big if-then-else
- It works...
- But enhancements need to made very carefully
- To add a new command, must modify TIBCOToButtonPress and BCCPButtonPress
- Not too bad, but we can do better

Command Handler



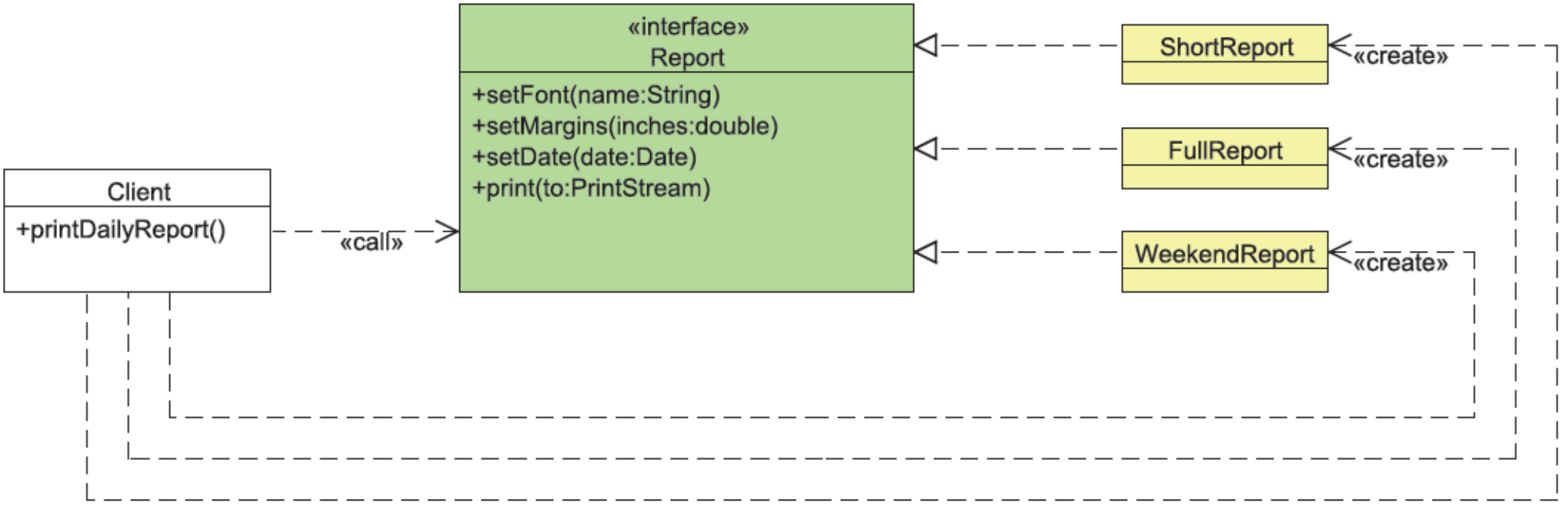
Command Pattern UML



Command Pattern

- Put each command handler into its own class
- To add a new command, just add a new class, add one entry to map in CCRequestHandler
- Don't need to modify existing commands
- Don't need to modify TIBCOToButtonPress

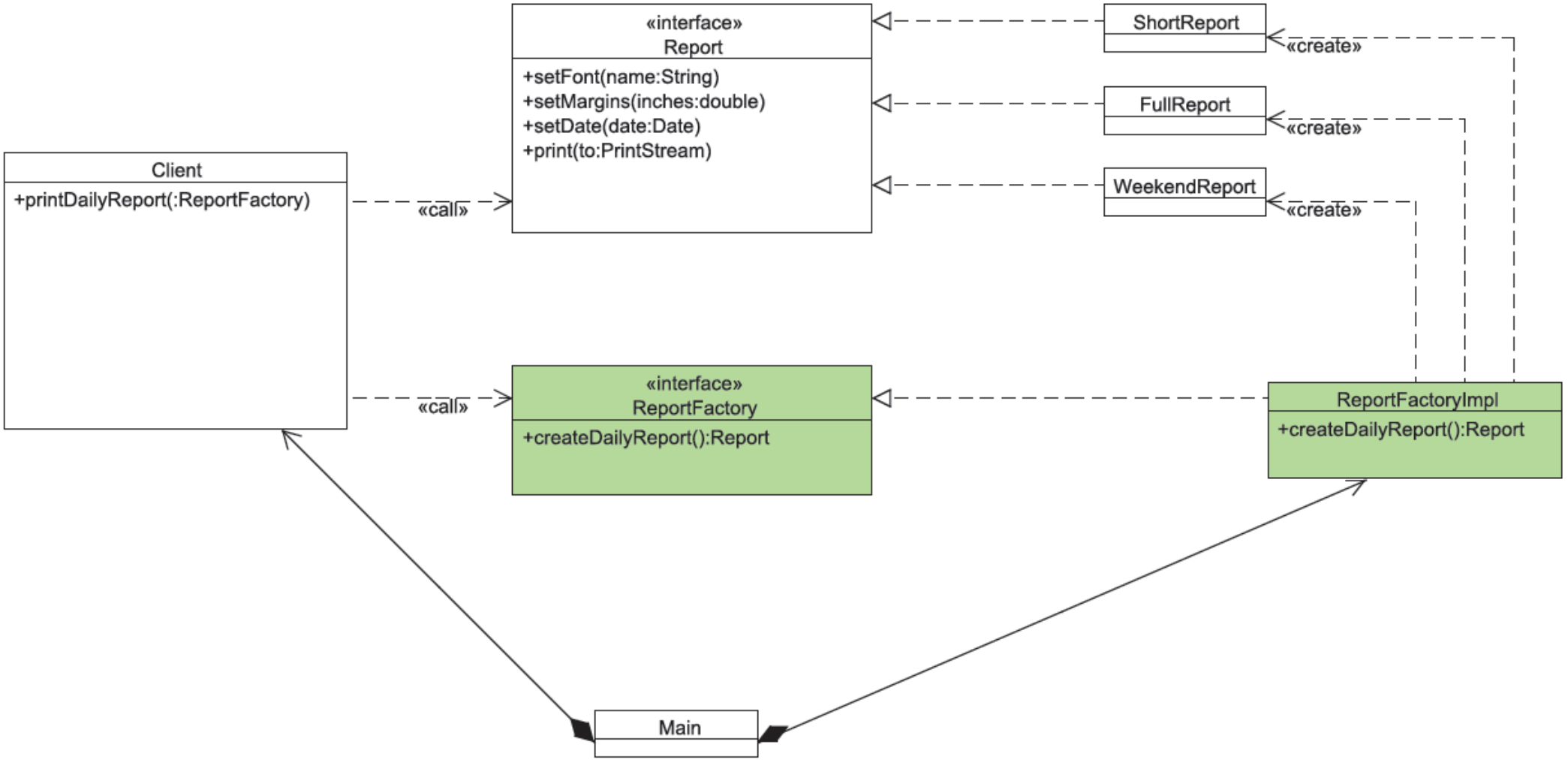
Creating Concrete Classes



Need a Factory?

- Client depends ONLY on Report interface
- (Well... almost)
- Client needs to CREATE concrete classes

Factory Pattern



Factory Pattern

- Client depends only on abstractions
- Client is totally isolated from concrete classes
- (Concrete classes have to be somewhere...)
- At least we can isolate that place

Which Strategy?

- Break out (potential) multiple algorithms (“strategies”) into their own classes.
- Allows switching algorithms to track changing requirements
- Could even allow switching at runtime

Strategy Pattern