State

Behavioural Pattern

Protected Variations (PV)

- A design principle aimed at containing (to keep under proper control) code changes after a system is implemented
  - Larman (2004) – Applying UML and Patters
- PV is usually achieved by adding a level of indirection, an interface, and using polymorphism to deal with the identified points of predicted variation.

State

- To change the behaviour of an object dynamically. It appears as if the object changes to be an object of a different class based on some internal state.

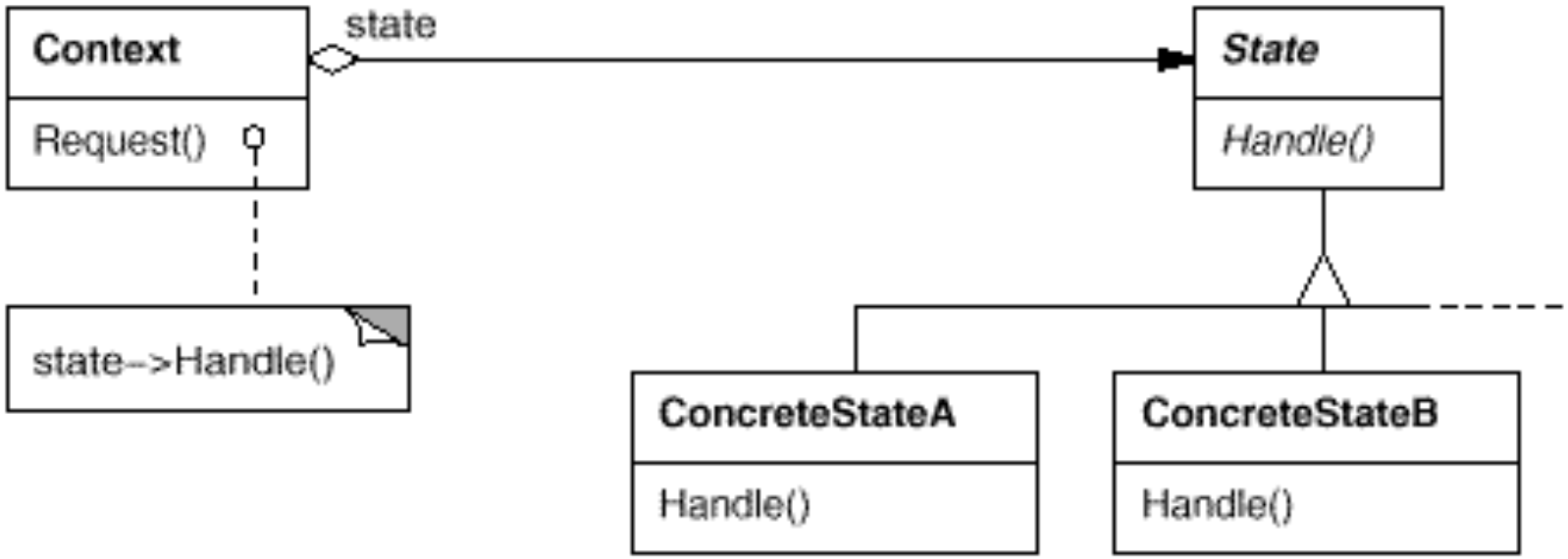
Problem solved by State

- When an object is large and has many different states dictating differing behaviours the code can become hard to maintain.

Method applied by State to solve the problem

- Avoid complex conditional statements appearing in various places by encapsulating them in classes (i.e. employ Polymorphism)

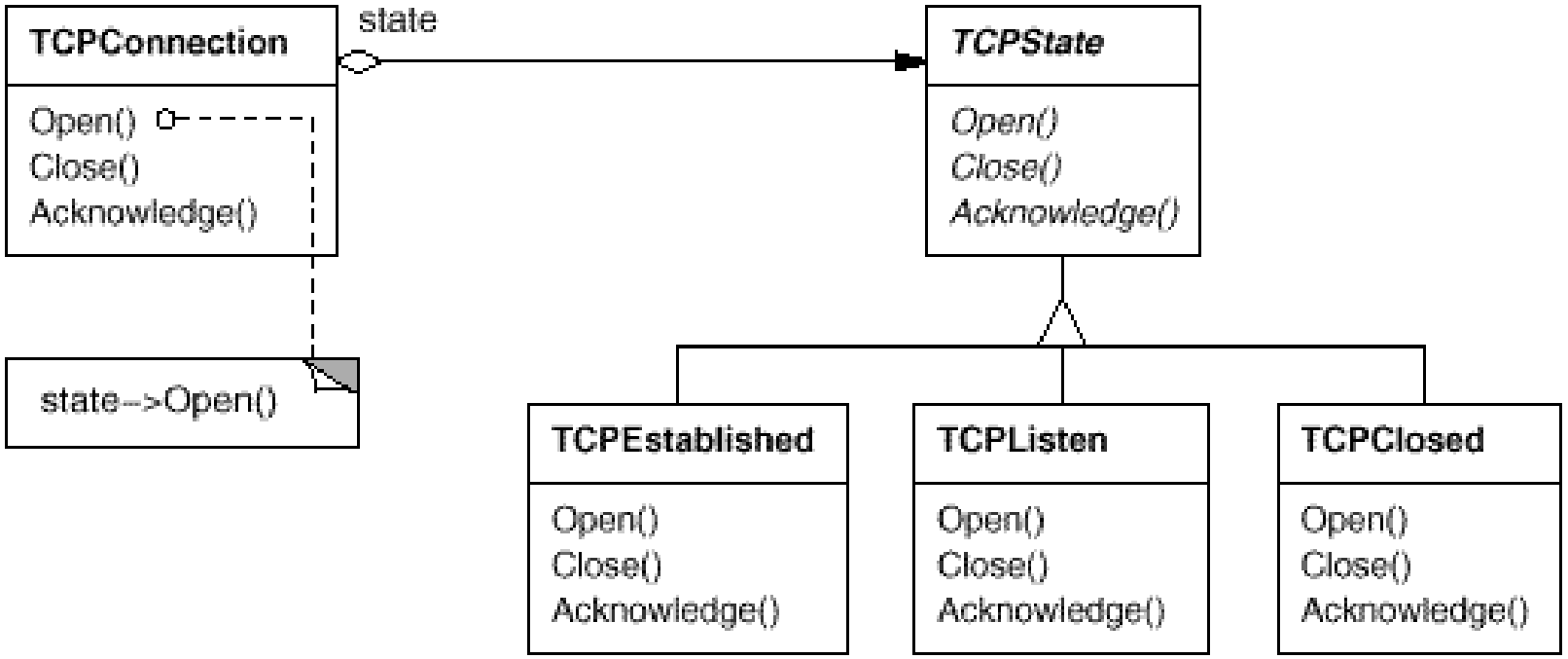
Structure


Context
Request()
state
State
Handle()
state->Handle()
ConcreteStateA
Handle()
ConcreteStateB
Handle()

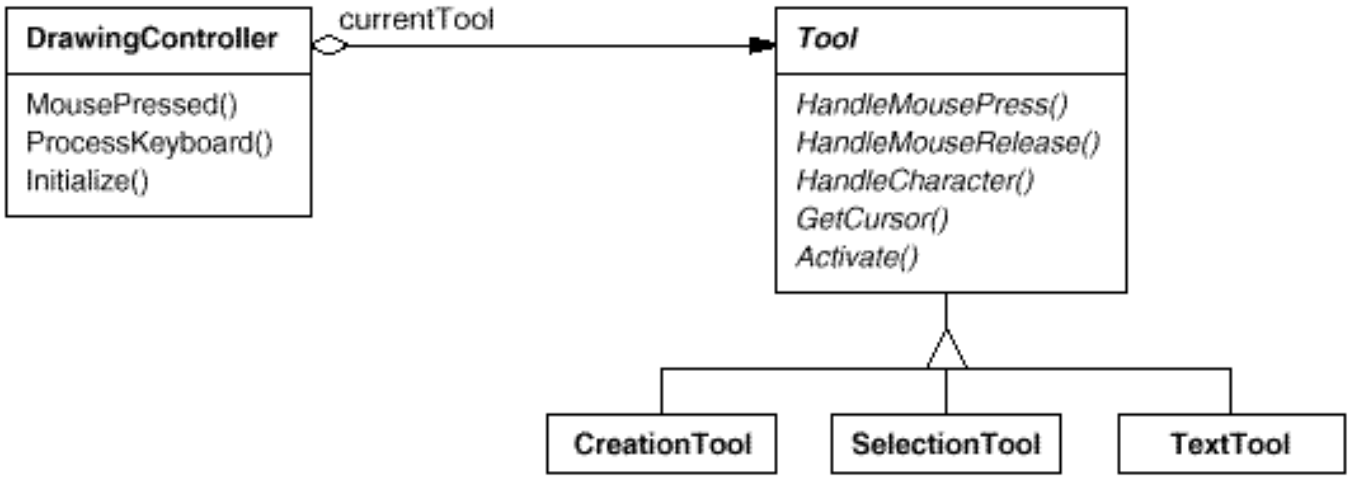
Participants

- Context
  - Delegate action to correct concrete state.
- State
  - Interface to define the functions that are dependant on state.
- Concrete State
  - Implement the state specific functions.

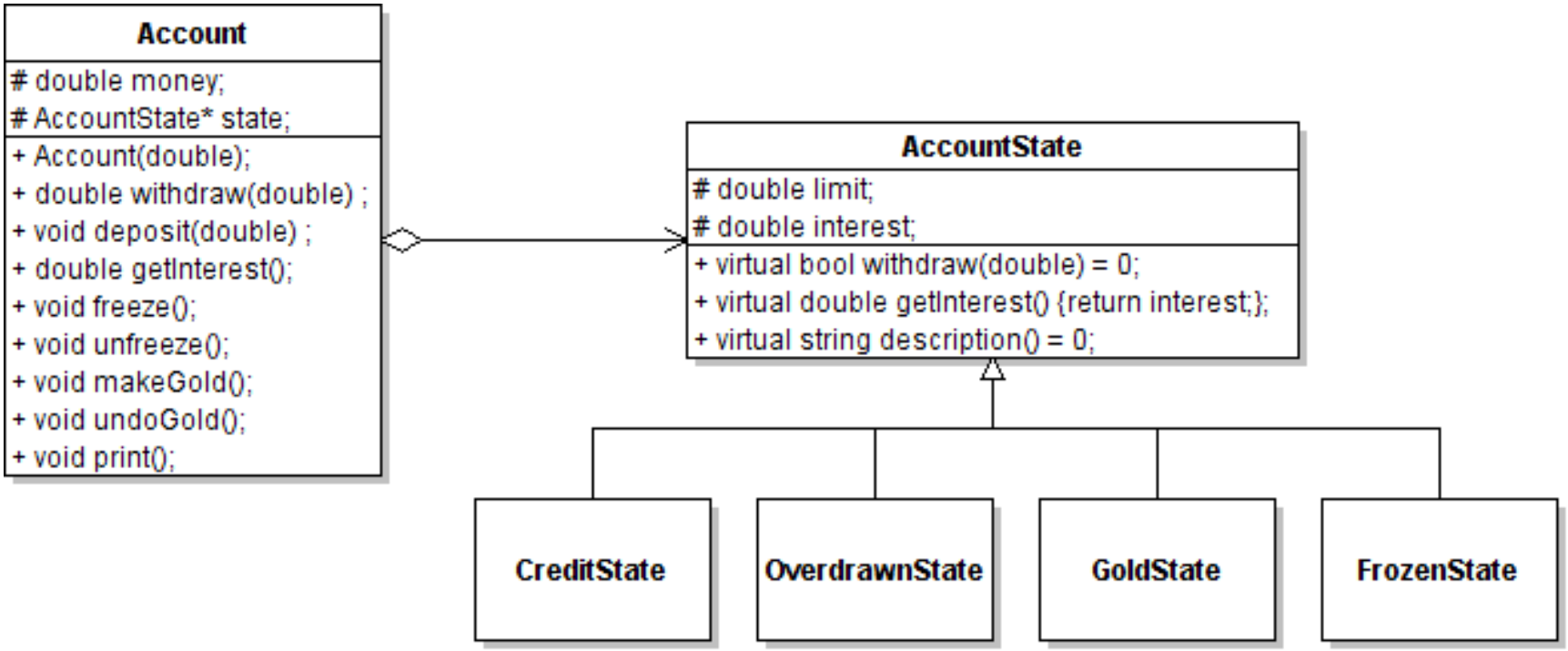
Example


TCPConnection
Open()
Close()
Acknowledge()
state
TCPState
Open()
Close()
Acknowledge()
state->Open()
TCPEstablished
Open()
Close()
Acknowledge()
TCPListen
Open()
Close()
Acknowledge()
TCPClosed
Open()
Close()
Acknowledge()

Example


DrawingController
MousePressed()
ProcessKeyboard()
Initialize()
currentTool
Tool
HandleMousePress()
HandleMouseRelease()
HandleCharacter()
GetCursor()
Activate()
CreationTool
SelectionTool
TextTool

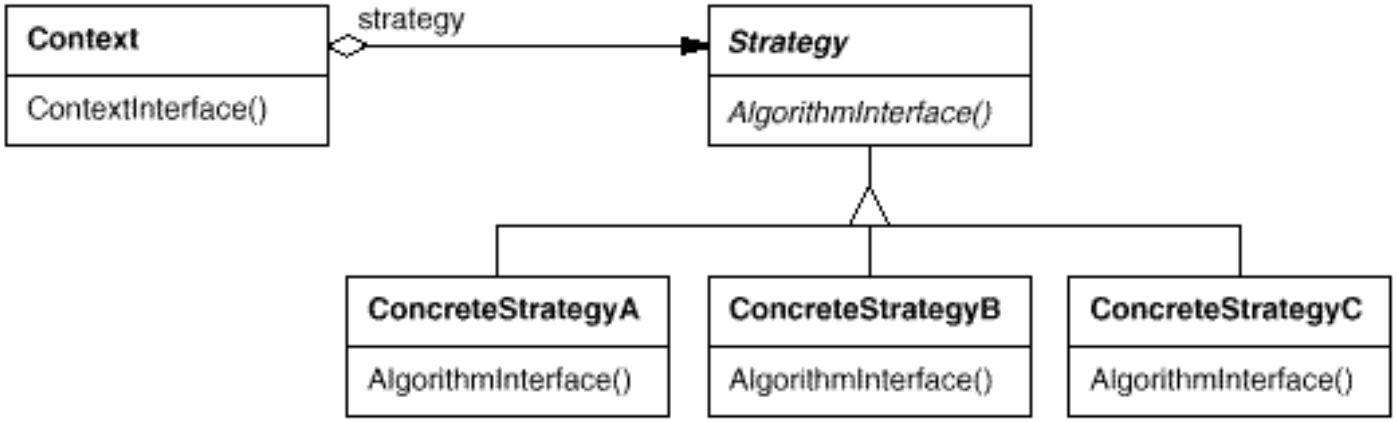
Example

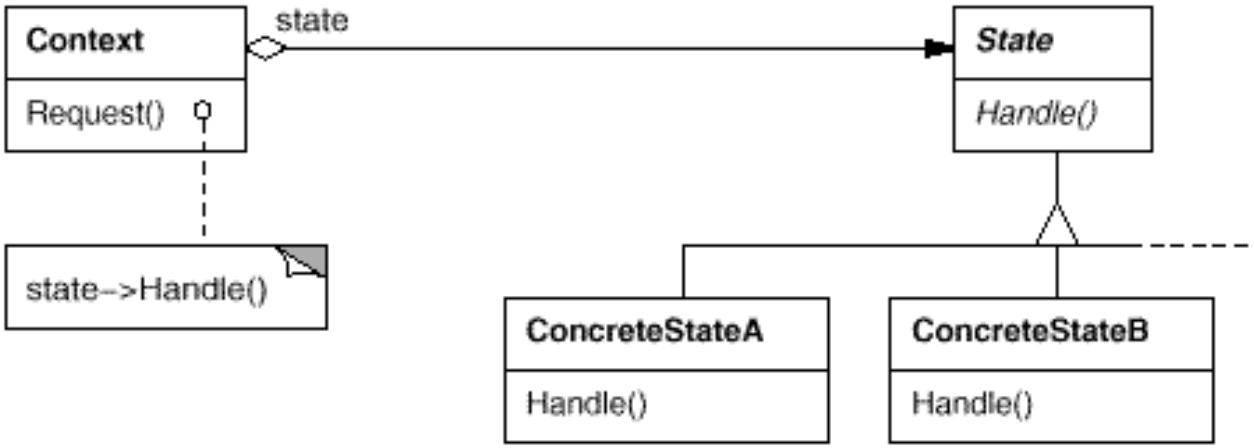

Account
double money;
AccountState* state;
+ Account(double);
+ double withdraw(double) ;
+ void deposit(double) ;
+ double getInterest();
+ void freeze();
+ void unfreeze();
+ void makeGold();
+ void undoGold();
+ void print();
AccountState
double limit;
double interest;
+ virtual bool withdraw(double) = 0;
+ virtual double getInterest() {return interest;};
+ virtual string description() = 0;
CreditState
OverdrawnState
GoldState
FrozenState

Related Patters

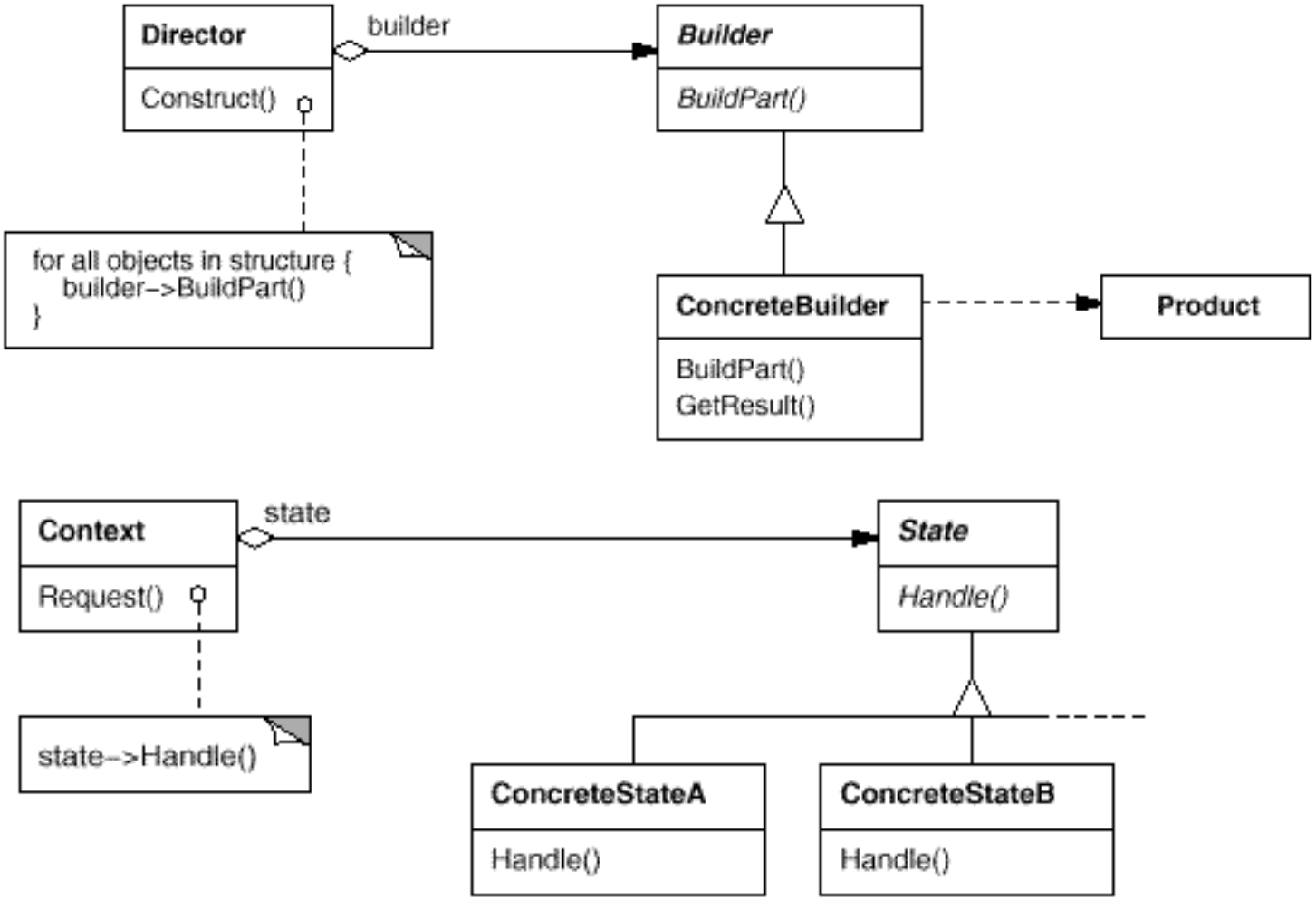
- State, Strategy and Builder has the same structure
  - Because all of them apply PV
- Sate objects are often Singletons

Strategy and State





Builder and State



Different ways to change state

- Context apply fixed criteria
  - Hard-coded in Context
- Context apply variable criteria
  - Context use a template method with variability implemented in the Concrete States
- Concrete States apply criteria
  - The application of conditions observed while executing other methods in a concrete state may trigger the change of state.

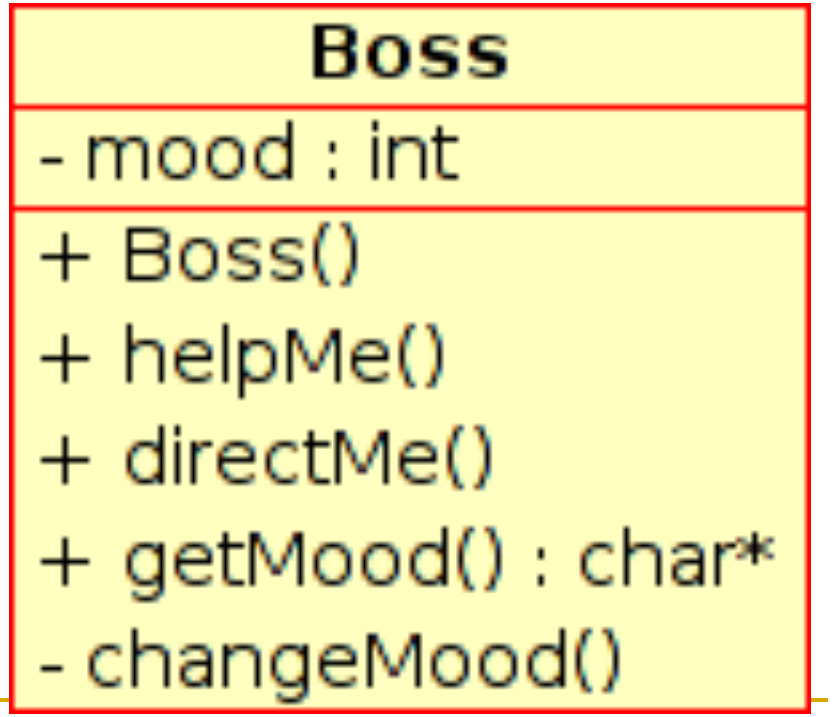
Lecture Example

- A system implementing a Boss class that implements the following actions:
  - helpMe
    - displays different strings depending on mood
  - directMe
    - displays different strings depending on mood
    - calls changeMood
  - changeMood
    - alternates between the moods
  - getMood
    - returns a string identifying the mood

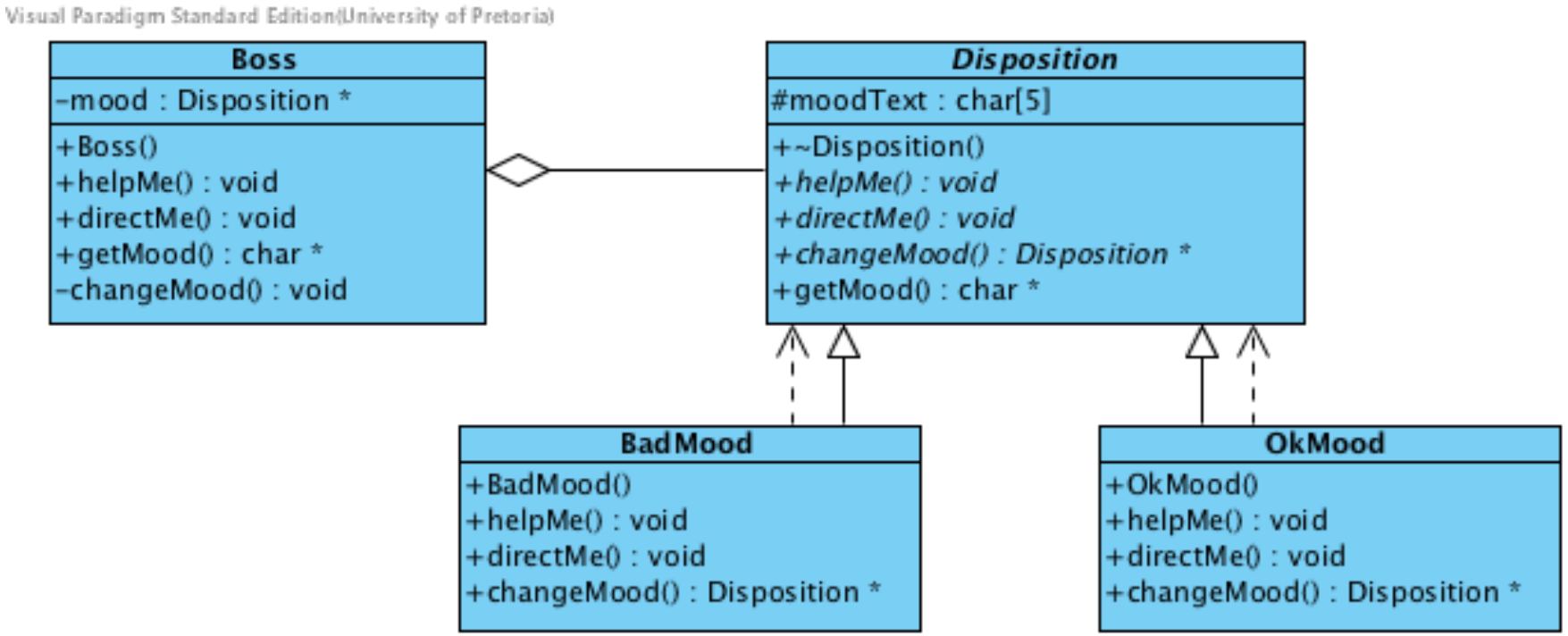
Inspired by an example by Vince Huston (http://www.vincehuston.org/dp/)

To add another mood to the system one has to:

- Define the new mood
- Redefine changeMood()
- Redefine getMood()
- Redefine helpMe()
- Redefine directMe()



Refactored Class Diagram



To Refactor the class to apply the State Pattern:

- Define an Abstract State with virtual handlers for all requests
- Change the original variable that encapsulated the state to a pointer to this Abstract State
- Define Concrete states and implement the handlers in the concrete states
- Redefine the requests to call the handlers

To add a mood after refactoring

- Define a new mood as an extension of the Abstract State (Same as other moods)
- Implement the concrete handlers

Summary

- Before refactoring
  - Changes are scattered and can give rise to complex logical structures (large switch or deeply nested if).
- Refactoring
  - Much more difficult that making the changes
- After refactoring
  - Changes are centralised
  - Logic is handled my means of polymorphism

Conclusion

- It is hard work to implement the pattern but it pays off when the system needs to be maintained.