Design Patterns

- The Strategy Pattern
- The Factory Method
- Generics
- The Abstract Factory Pattern
- **The State Pattern**
- The Observer Pattern
- The Adapter Pattern
- The Composite Pattern
- The Iterator Pattern
- The Builder Pattern
- Fallen Patterns
  - The Singleton Pattern
  - The Visitor Pattern

Motivating Example

- Suppose we're developing a video game…
- We have the following "states":
  - Main menu
  - Character Select
  - In-game
- Each state has dramatically different behavior!
- Also applies to simulations and many servers

The Basics

Every game and simulation has a *main loop*

The main loop runs multiple times per second to maintain the game:

- Check for button presses
- Update the interface
- Process animations
- Move objects / process physics
- *etc.*

Structured Approach

- Giant switch-case or if-then statement inside of main loop:

```
void update() {
    if( currentState == MAIN_MENU ) {
        drawMainMenu();
        checkMenuButtonClicked();
        …
    }
    else if( currentState == IN_GAME ) {
        drawCharacters();
        drawInterface();
        getCurrentMove();
        …
    }
}
```

What's Wrong?

- What's wrong with having a single switch-case to perform updates?
  - The states should be totally independent, but they exist in the same function
  - States will share data, data should be isolated
  - What if States have sub-states? This gets nasty
  - What if we need multiple instances of a state?
    - This could violate DRY (Don't Repeat Yourself)

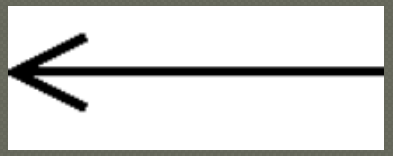
Solution

Give each state its own class.

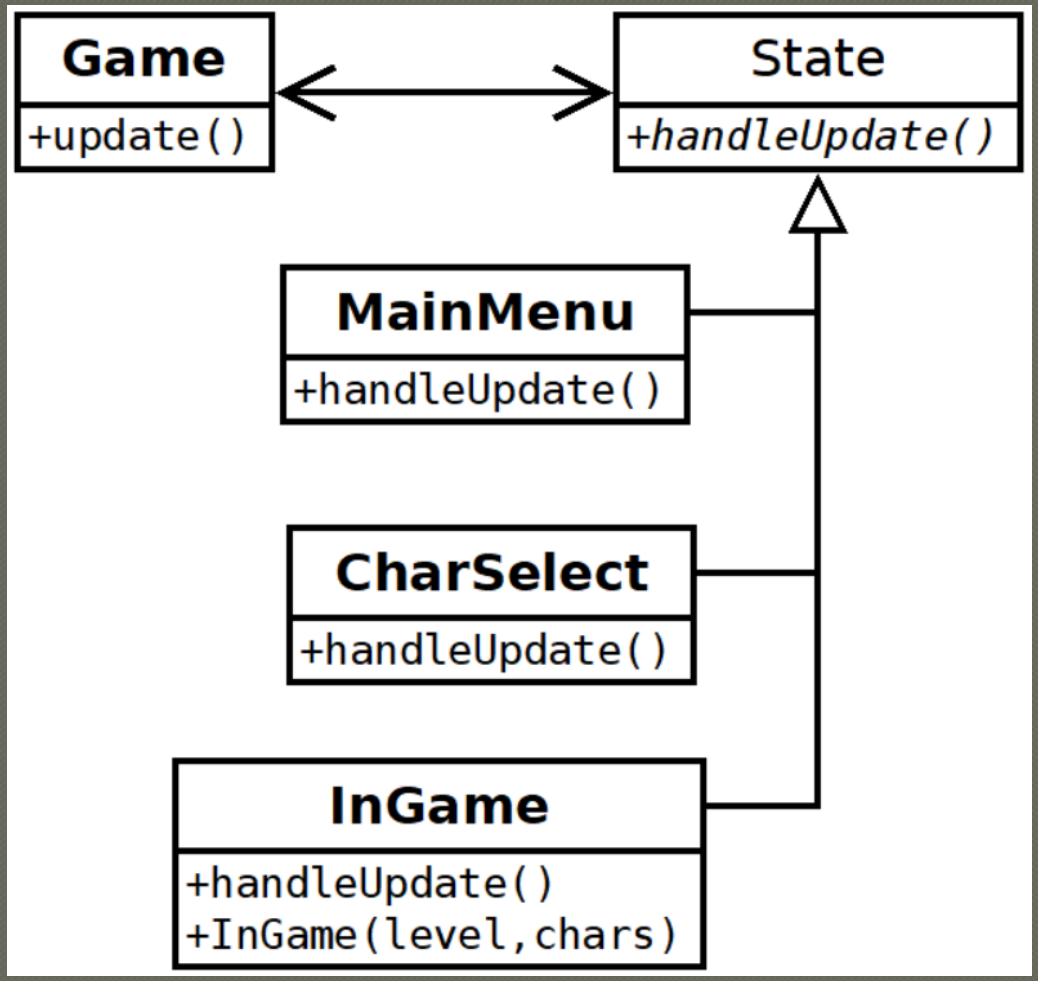
Have them inherit/implement a base state.

A Note on Delegation

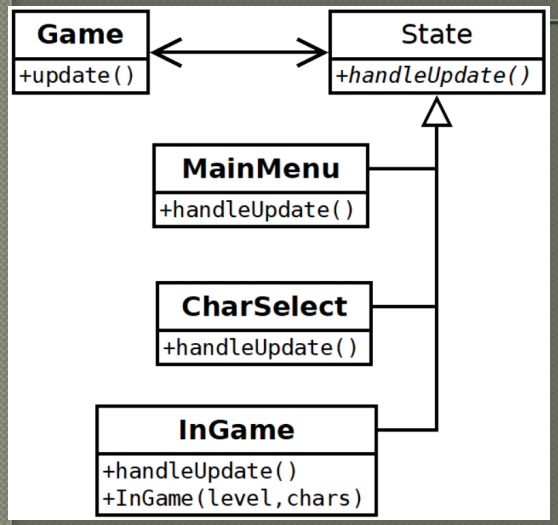
- Delegation allows an object to delegate one or more tasks to a helper object
- Uses the Association connector



Game State


Game
+update()
State
+handleUpdate()
MainMenu
+handleUpdate()
CharSelect
+handleUpdate()
InGame
+handleUpdate()
+InGame(level,chars)

Some Notes



- Delegation is two way in this particular example
  - States often need information from the Game to perform their duties
  - Only delegate both ways if the State actually needs information from the Game
- States often have constructors
  - InGame takes information from CharSelect to create the game scenario
- If a State does not have state of its own, only one instance of it is needed

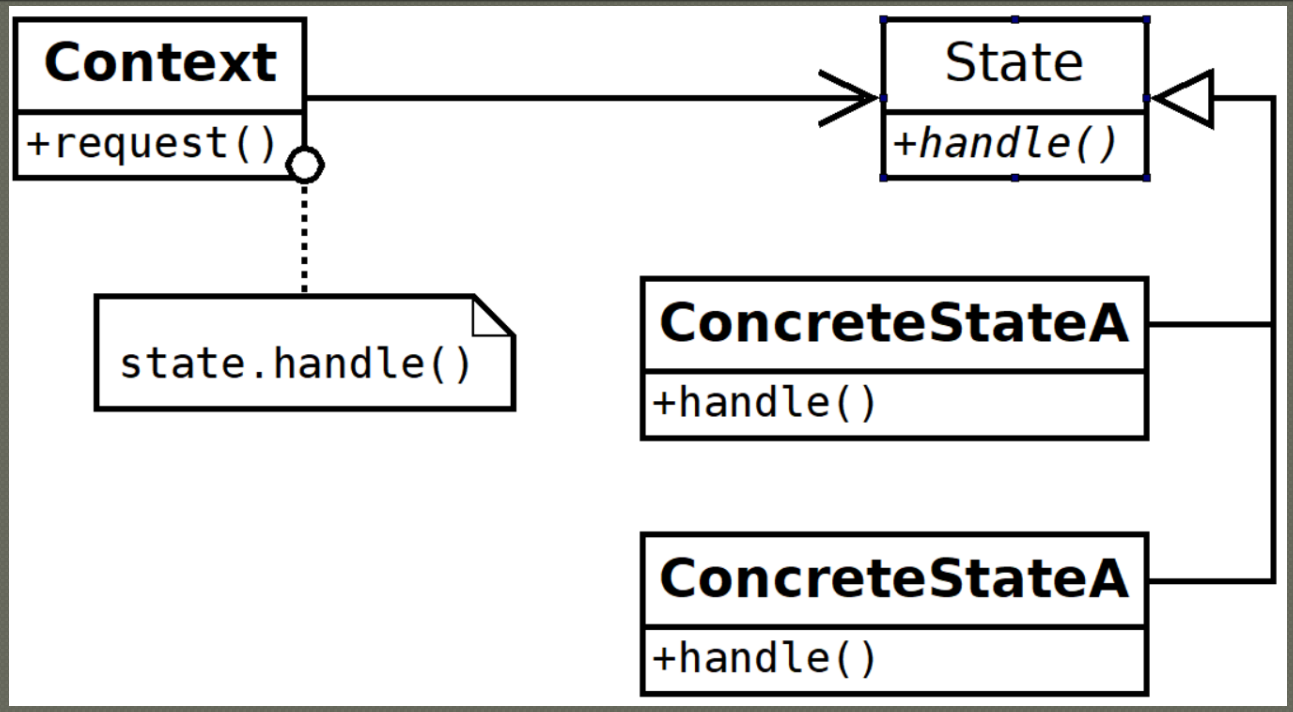
Implementation

```
class Game {
    //starting state
    State state = new MainMenu();

    void update() {
        //some of these methods may use
        //fields we're omitting…
        updateAudioBuffer();
        drawGraphics();
        //use the state
        state = state.handleUpdate();

    }
}

abstract class State {
    protected Game game;
    abstract void handleUpdate();
    State(Game g) { game = g; }
}
```

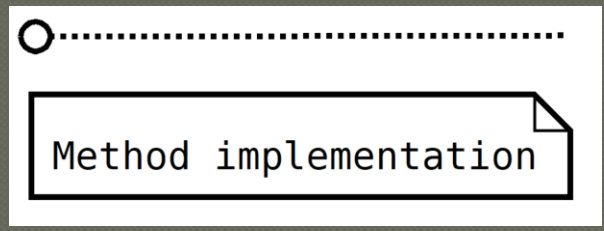
Implementation

```
class MainMenu extends State {
      State handleUpdate() {
          if( getSelectedMenuItem() == CHAR_SELECT)
              return new CharSelect(game, … );
      }
}

class CharSelect extends State {
    State handleUpdate() {
          if( selectedChar() )
              return new InGame(
                          …, getSelectedChar());
    }
}
```

Generic Structure



- New Notation
  - Not an official UML notation
  - Used for our explanation purposes only



Notice Something Familiar?

- The graph is essentially the same as Strategy
- But there are some key differences:
  - States can refer to their context
  - States can often replace themselves
  - State tends to be an abstract class, where Strategy is usually an interface

More Differences

- Strategy often has sub-classes of Context
- More than one strategy per context is common
  - More than one state per context is extremely rare
- Strategies almost never refer to their Context

Another Example

- Suppose we're making some kind of server
  - The server listens for connections
  - Connections involve some ongoing communication between client and server
  - To handle many clients, servers delegate processing to Procs
  - We can't pause while waiting or the server is vulnerable to denial of service
  - If a client hangs, we want to drop the connection

The Structured Approach

- Store a list of Procs, each of which has a connection and a state enum
- We call a function "update(proc)" which contains a giant `if` statement to determine what to do if the connection is "IN_PROGRESS", "WAITING", or "TIMED_OUT"
- When the connection is timed out, we kill it

The OO Approach

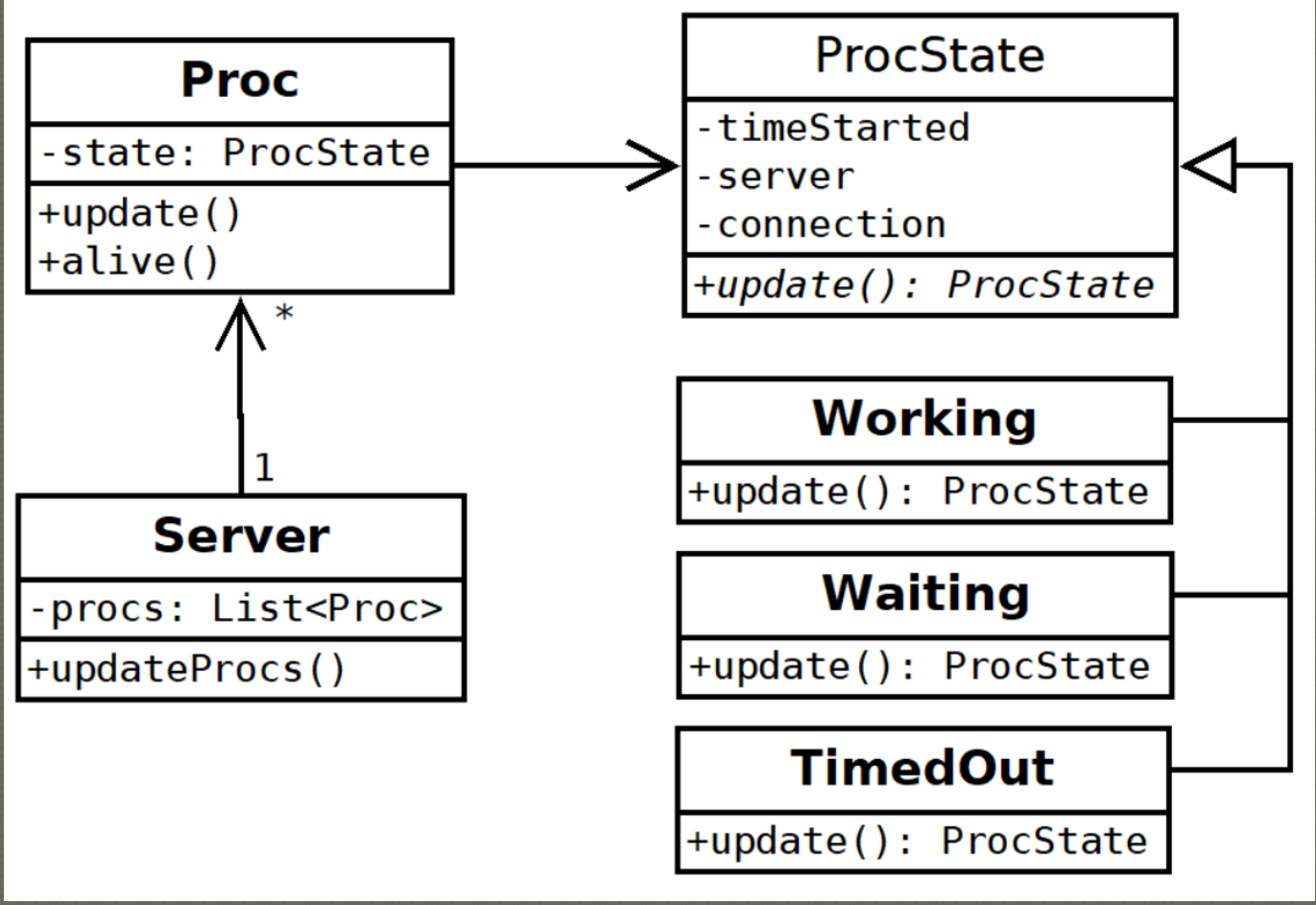
Represent each state as a class

Each state does the correct thing when "update" is called.

When a proc has the state "TimedOut", it is killed.

No giant if statement, only a small if statement for TimedOut

Server Design



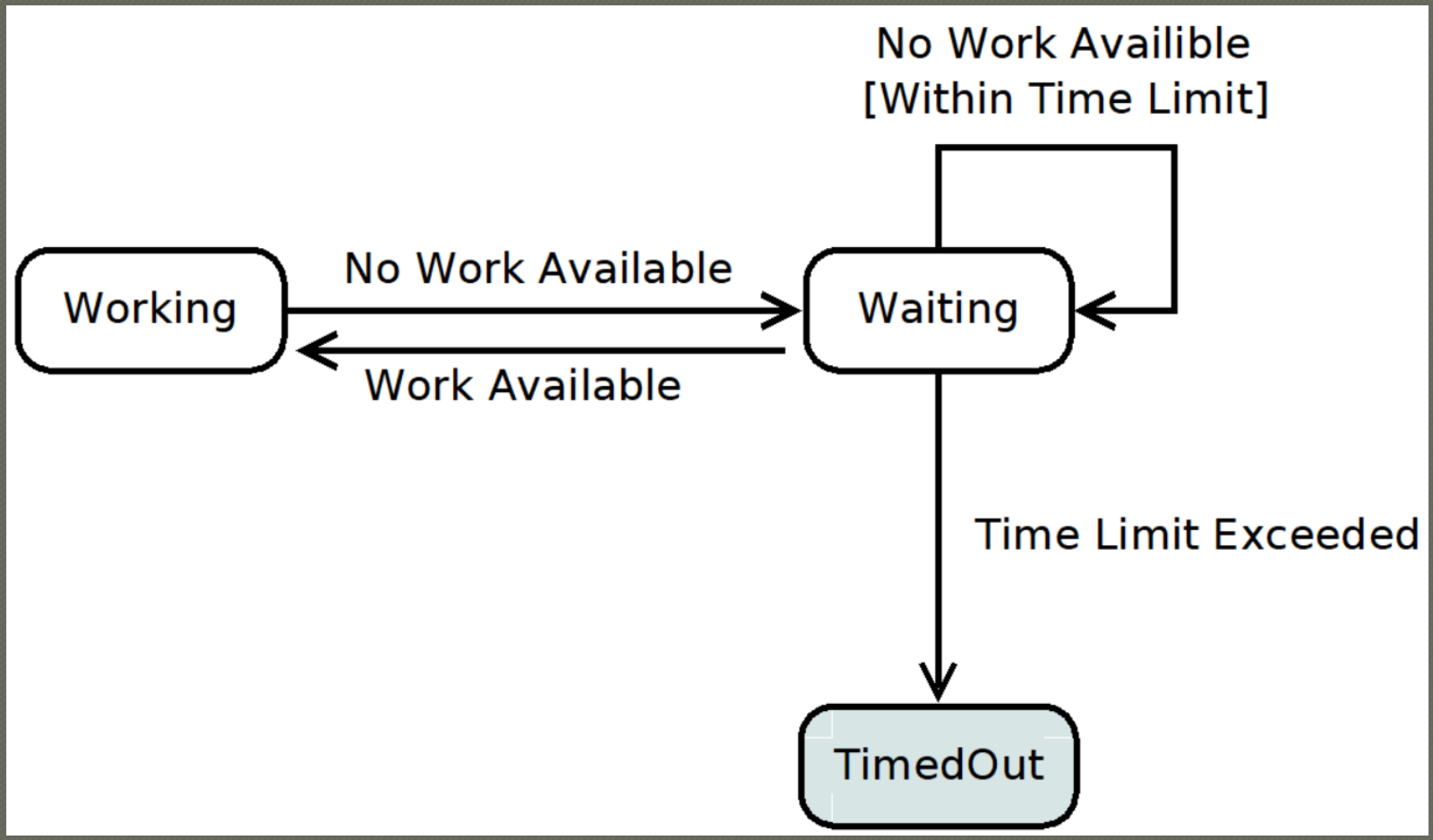
The Server

```
class Server {
    private List<Proc> procs = new …;

    public void updateProcs() {
        //somewhat inefficient way to remove dead procs
        List<Proc> aliveProcs = new …;

        foreach(Proc proc : procs ) {
            proc.update();

            if( proc.alive() )
                aliveProcs.add( proc );
        }

        procs = aliveProcs;

    }
}
```

The Context

```
class Proc {
     private ProcState state;

     public void update() {
          state = state.update();
     }

     public void alive() {
          return state instanceof TimedOut;
    }
}
```

State Diagram



The Concrete States: Working

```
class Working extends ProcState {
    public ProcState update() {
        if( stuffToProcess() )
            processStuff();
            return new Working();
        else
            return new Waiting();
    }

    void processStuff() { … }
}
```

The Concrete States: Waiting

```
class Waiting extends ProcState {
     public void update() {
          if( timeElapsed() > TIME_MAX )
               return new TimedOut();
          else if( stuffToProcess() )
               return new Working();
          else
               return new Waiting();
     }
}
```

The Concrete States: TimedOut

```
class TimedOut extends ProcState {
    public ProcState update() {
        return new TimedOut();
    }
}
```

States with Sub-States

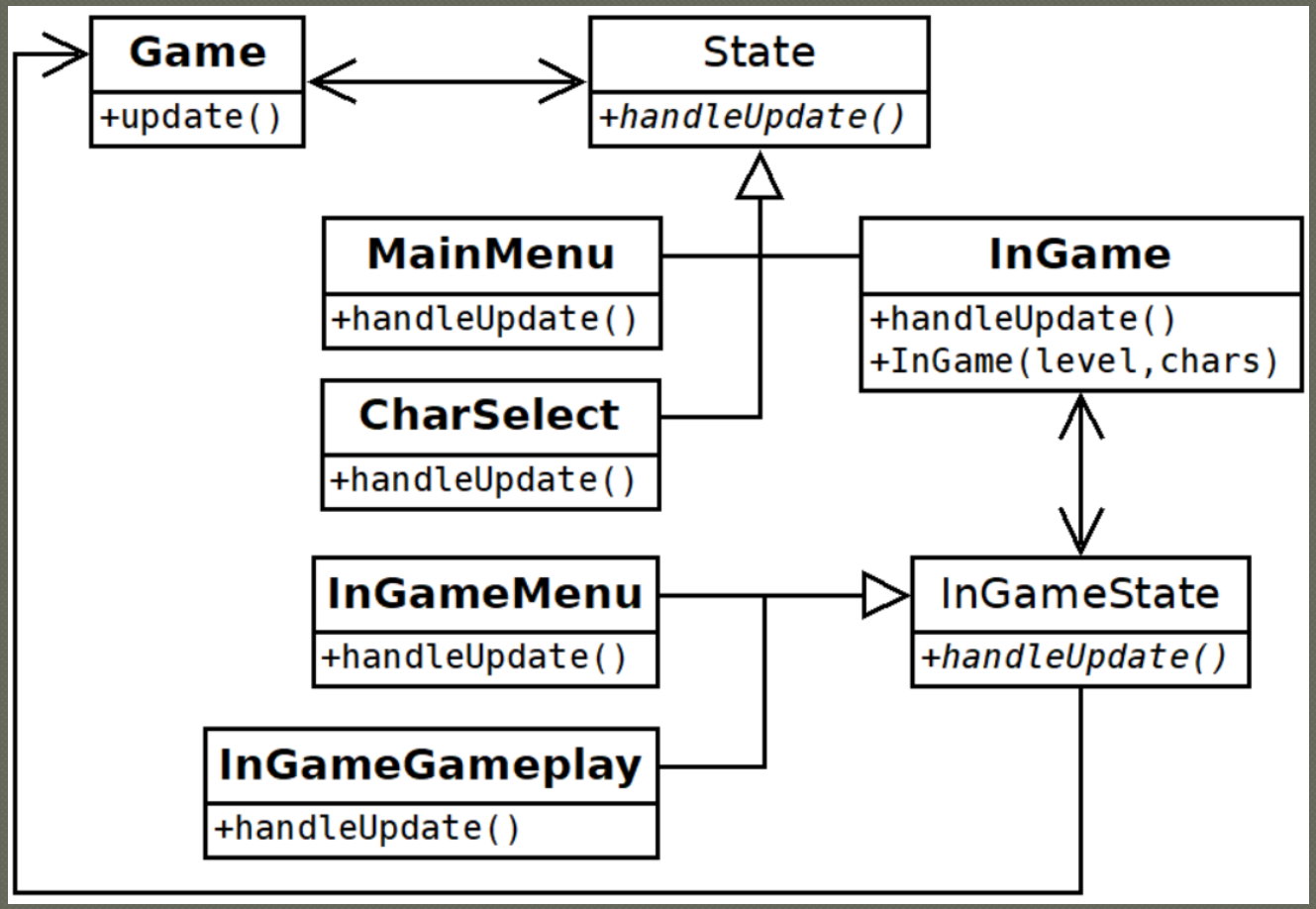
- A more complex scenario: back to the game
- What about the in-game menu?

Design Requirements

1. Additional State for InGameMenu can't be accidentally switched to by other states

2. InGameMenu can lead back to MainMenu and CharacterSelect

Design


Game
+update()
State
+handleUpdate()
MainMenu
+handleUpdate()
InGame
+handleUpdate()
+InGame(level,chars)
CharSelect
+handleUpdate()
InGameMenu
+handleUpdate()
InGameState
+handleUpdate()
InGameGameplay
+handleUpdate()

Changes

Now InGame has its own private state

We can't accidentally assign an InGameState to a regular State variable

Don't need a nested if statement in InGame

Strategy vs State

- Similar diagrams, easy to confuse
- Strategies:
  - Created outside the object
    - Given to it with constructor or as a function parameter
  - Handle a single, specific thing (like grasping or navigation)
  - Can be multiple strategies simultaneously in one object
  - Usually don't change while the object is live
- States:
  - Handle everything an object does
  - Sometimes hold a reference to their context
  - Often the state is responsible for changing to a new state or returning one
  - Generally only one

You Know the Drill

- Apply the state pattern to your project