SWEN 262
Engineering of Software Subsystems

State Pattern

Superb Plumber Siblings™

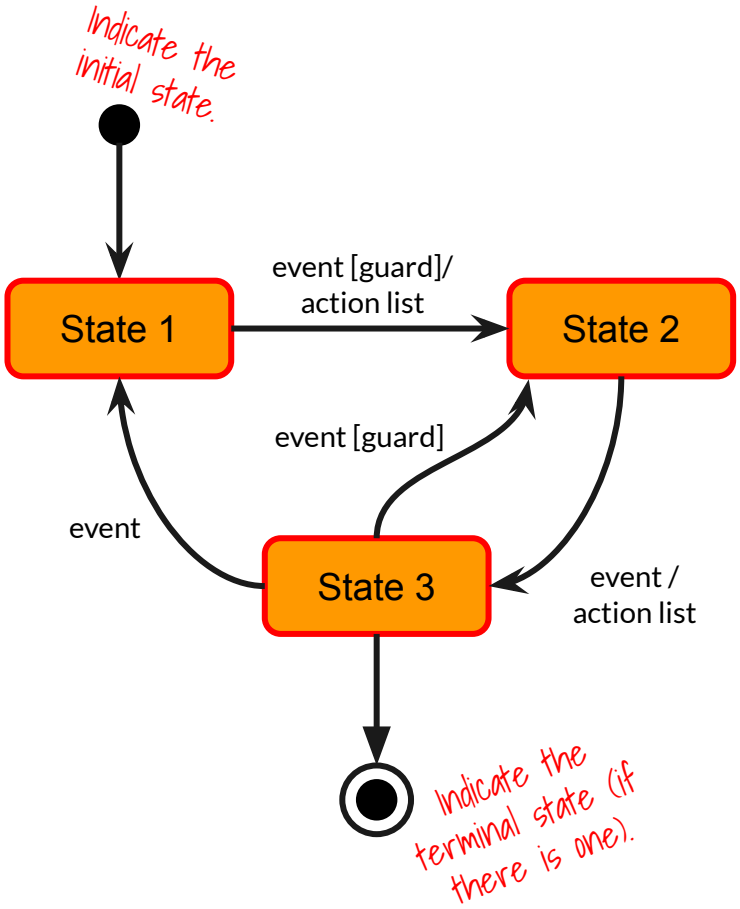
1. In Superb Plumber Siblings™, the player avatar starts in its ***small form***.
   a. *Contact with an enemy results in losing a life (character death).*
   b. *Acquiring a mushroom awards 1000 points and changes the player avatar to its **superb form**.*
   c. *Acquiring a fire flower awards 1000 points and changes the player avatar to its **shooter form**.*
2. While in ***superb form***.
   a. *Contact with an enemy changes the player avatar to its **small form**.*
   b. *Acquiring a mushroom awards 1000 points.*
   c. *Acquiring a fire flower awards 1000 points and changes the player avatar to its **shooter form**.*
3. While in ***shooter form***.
   a. *Contact with an enemy changes the player avatar to its **small form**.*
   b. *Pressing the fire button shoots a fireball.*
   c. *Acquiring a mushroom awards 1000 points.*
   d. *Acquiring a fire flower awards 1000 points.*

There are a number of ways that the different ***states*** of the player avatar may be handled...

State Diagrams

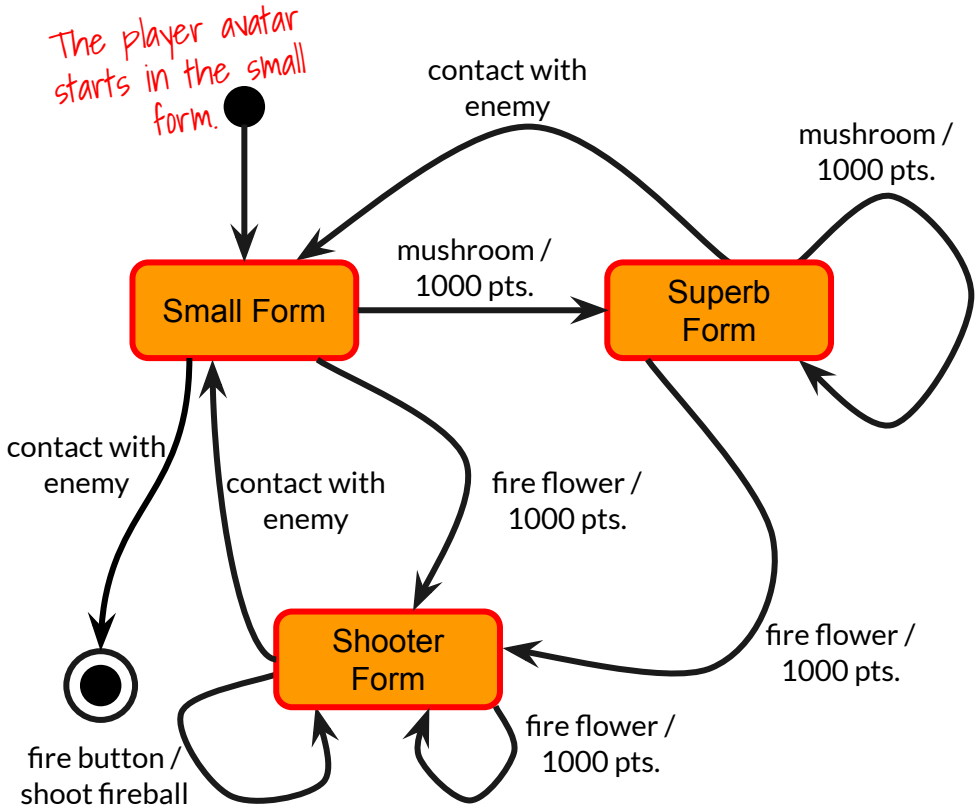
- But first, it can be helpful to create a ***state diagram*** of the system.
  - We will use ***UML statechart*** notation.

- A state diagram includes:
  - Each of the possible ***states*** that the system may be in at any given time.
  - The ***events*** that cause the system to transition from one state to another.
    - Each event may define a ***guard*** condition that must be true for the transition to occur. Guard conditions must be mutually exclusive.
    - Each event may define an ***action list*** that is executed before the transition occurs.
- A state diagram defines a ***finite state machine***.
  - The system exists in exactly one state at a time.
  - In each state the system behaves differently.
    - Each and every method in an object may behave differently depending on the current state.



Superb Plumber Siblings™ State Diagram

- As mentioned previously, the player avatar may exist in one of three potential ***states***.
  - Small Form
  - Superb Form
  - Shooter Form
- Different ***events*** may cause transitions from one state to another.
  - Some events include an ***action list***, e.g. awarding 1000 points.
  - There are no ***guard*** conditions in this diagram.
- The same or similar events are handled differently depending on which state the system is in, e.g.
  - Contacting an enemy in ***small form*** loses a life.
  - Contacting an enemy in ***superb form*** transitions to small form.



Conditionals

```
public void enemyContacted() {
  switch(currentState) {
    case SMALL_FORM:
      loseALife();
      break;
    case SUPERB_FORM:
    case SHOOTER_FORM:
      currentState = SMALL_FORM;
      shrinkAvatar();
      break;
  }
}
```

Every method on the player avatar changes behavior depending on the current state using a conditional to determine which code to execute, e.g. what to do when the player contacts an enemy.

Q: What are the major drawbacks of this approach?

A: One massive class that contains all of the behavior for all of the possible states is not very cohesive.

Adding new states (like invincibility!) requires modifying all of the state-dependent methods (OCP).

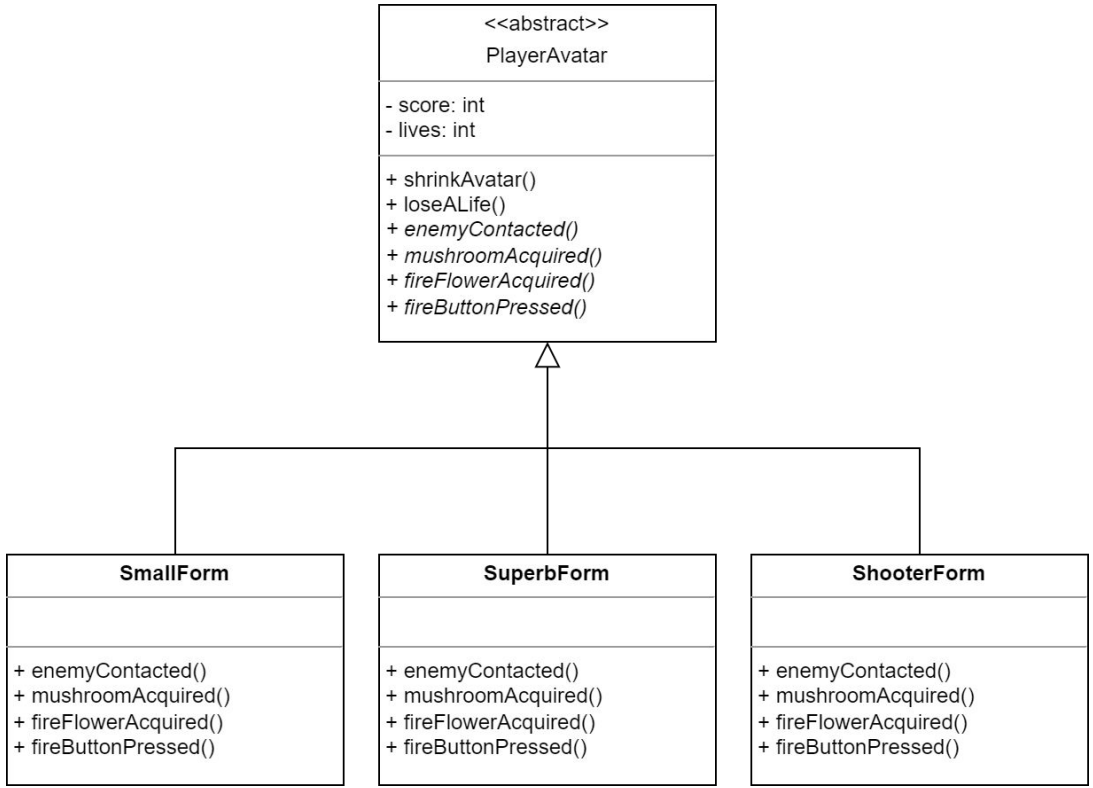
All possible states and behavior are mixed together. This makes it possible for the object to be in an *inconsistent* state where some methods execute as though the system is in one state and others execute as though it is in a different state.

Subclassing

A: Create an abstract class for the *player avatar*, and a different subclass for each possible state.

But what decides which to instantiate? What controls the transitions from one instance to another when an event occurs?

How is player state (like score and remaining lives) copied from one to the other?



How About a State Interface?

```
interface Form {
  void handleEnemy();
  void handleMushroom();
  void handleFireFlower();
  void handleFireButton();
}
```

Define an interface that represents the ***state*** that the player avatar may be in.

It should define a method for each of the behaviors that changes based on the current state.

A Context

Create a ***context*** that represents the player avatar.

The ***context*** contains all of the state required for the player avatar including an instance of the ***state*** interface that represents the current state.

The ***context*** also provides a (usually protected) method used to change the current ***state***.

In each method that changes behavior depending on the current state, the context ***delegates*** to the current state.

```
public class PlayerAvatar {
  private Form currentForm;
  int score;
  int lives;

  void setForm(Form form) {
    currentForm = form;
  }

  public void enemyContacted() {
    currentForm.handleEnemy();
  }

  public void mushroomAcquired() {
   currentForm.handleMushroom();
  }

  // and so on...
}
```

Concrete States

Create a ***concrete state*** for each of the possible states that the player avatar may be in.

In this example, the concrete state has a reference to the ***context***...

...which is used to change the state and/or call methods on the context.

Some of the methods on the concrete state may cause the context to transition to another state.

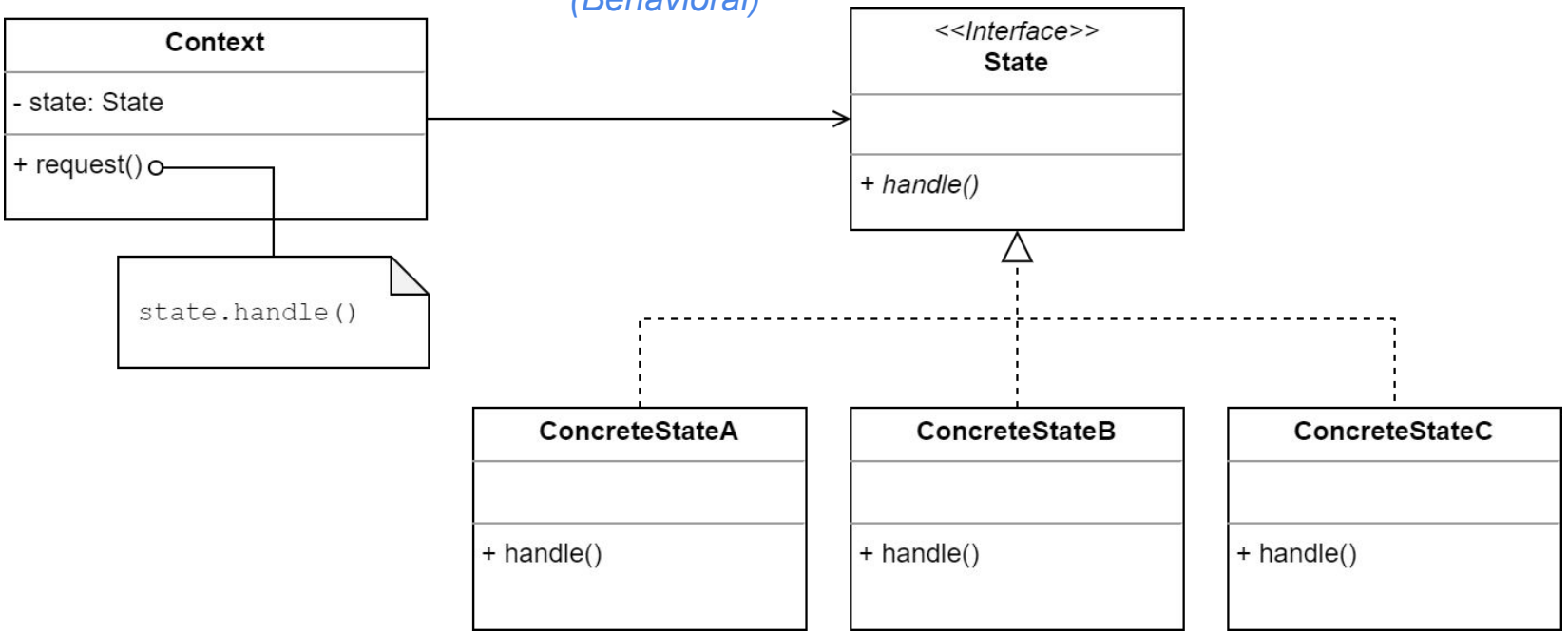
Additional concrete states may be added in the future (e.g. *Invincible Form*).

```
class SmallForm implements Form {
  private final PlayerAvatar avatar;

  SmallForm(PlayerAvatar avatar) {
    this.avatar = avatar;
  }

  public void handleEnemy() {
    avatar.loseALife();
  }

  public void handleMushroom() {
    avatar.score += 1000;
    avatar.grow();

    avatar.setForm(new SuperbForm(avatar));
  }
  // and so on...
}
```

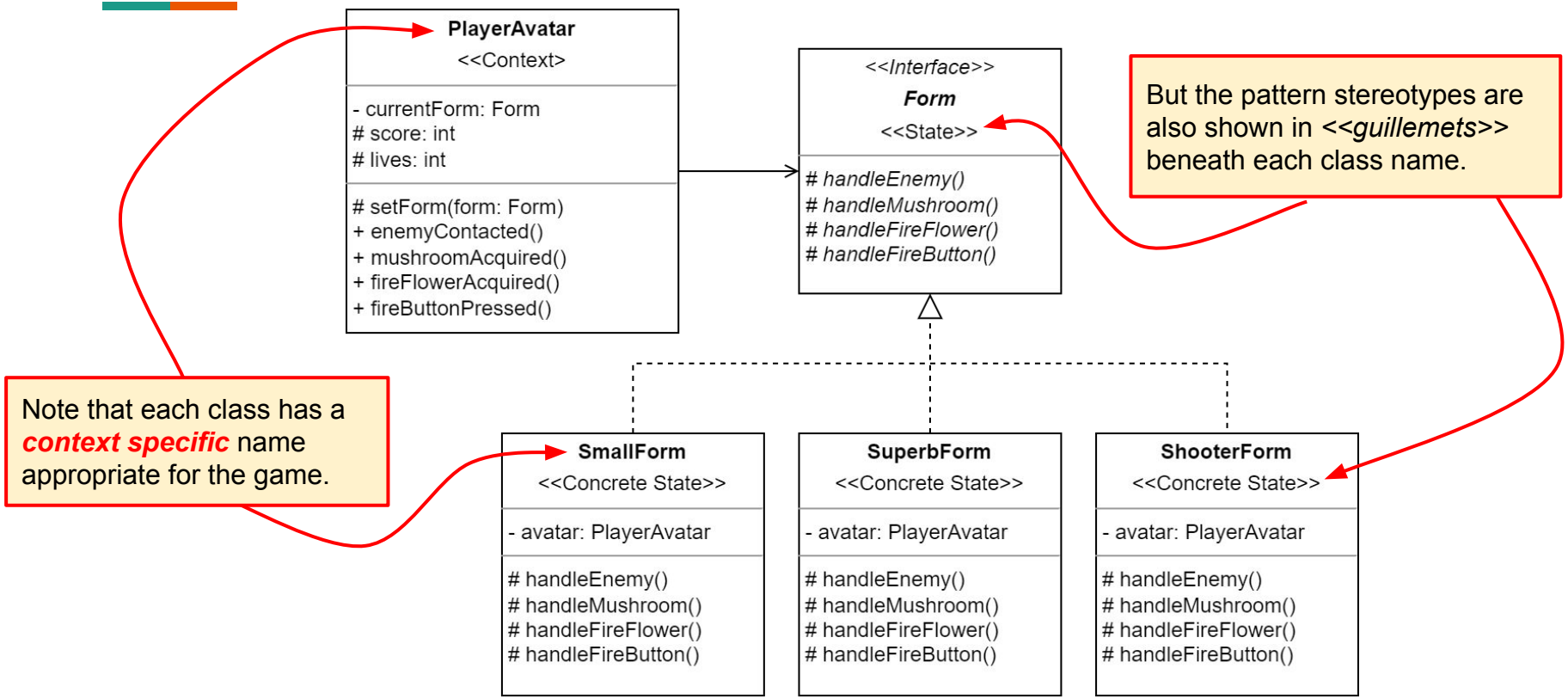
GoF State Structure Diagram

Intent: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

(Behavioral)

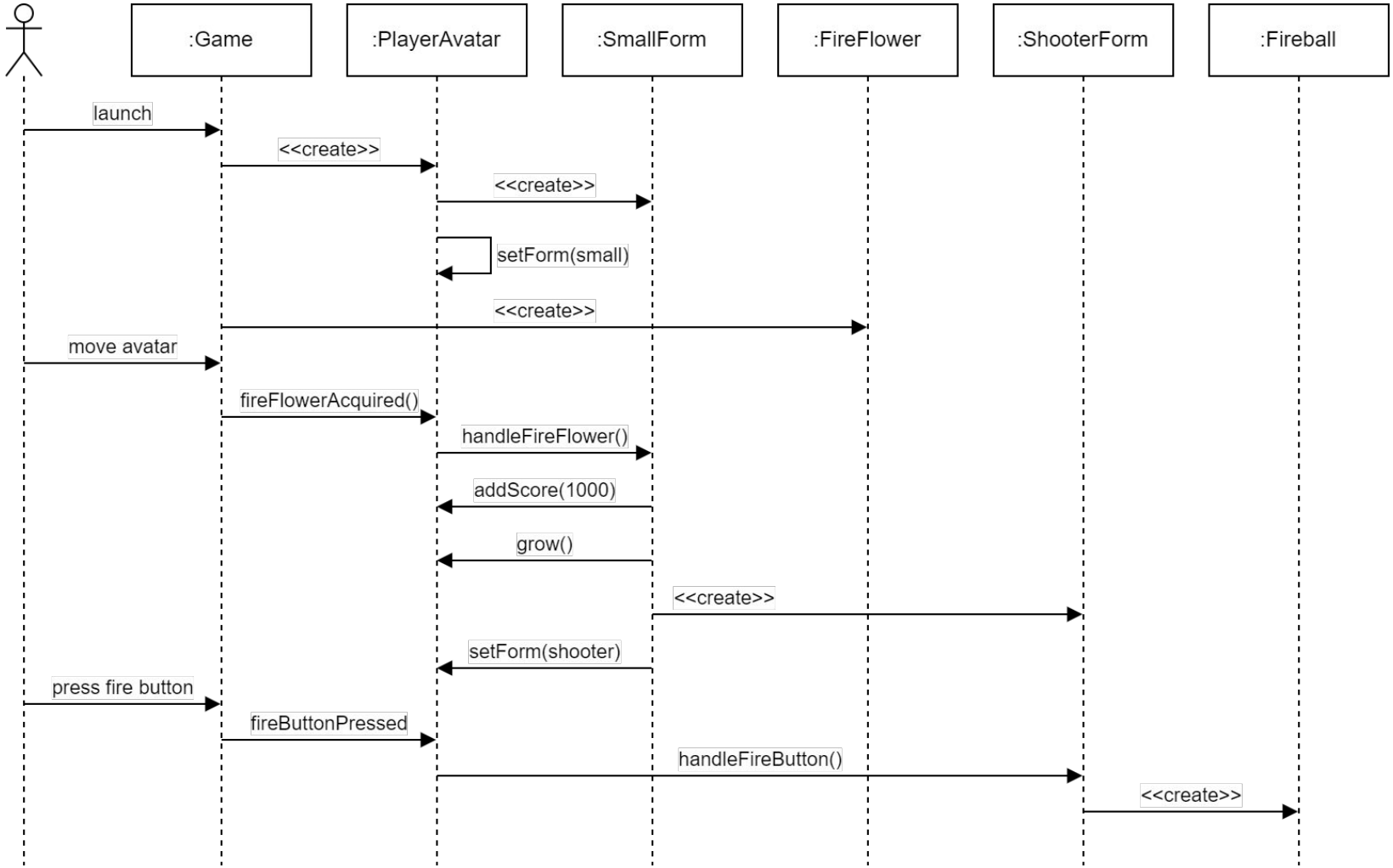


Superb Plumber Siblings™ State Design



GoF Pattern Card

Name: *Player Avatar Subsystem*		GoF Pattern: *State*
Participants		
Class	Role in Pattern	Participant's Contribution in the context of the application
PlayerAvatar	*Context*	*Represents the player avatar in the system. Keeps track of the score, remaining lives, etc. Delegates state-based behavior to its current state.*
Form	*State*	*Defines the set of behaviors that change based on the current state of the player avatar. These include handling enemy contact, acquiring a mushroom, acquiring a fire flower, and pressing the fire button.*
SmallForm	*ConcreteState*	*A player avatar form that represents the default state. Loses a life on enemy contact, awards 1000 points and transitions to superb form when a mushroom is acquired, and awards 1000 points and transitions to shooter form when a fire flower is acquired. The fire button does nothing.*
SuperbForm	*ConcreteState*	*The player avatar form after a small form acquires a mushroom. Enemy contact transitions to the small form. Acquiring a mushroom awards 1000 points. Acquiring a fire flower awards 1000 points and transitions to the shooter form. Pressing the fire button does nothing.*
ShooterForm	*ConcreteState*	*The player avatar form after small or superb forms acquire a fire flower. Enemy contact transitions to the small form. Acquiring a mushroom or fire flower awards 1000 points. Pressing the fire shoots a fireball.*
Deviations from the standard pattern: *None*		
Requirements being covered: *1. Small form, 2. Superb form, 3. Shooter form*		


:Game
:PlayerAvatar
:SmallForm
:FireFlower
:ShooterForm
:Fireball
launch
<<create>>
<<create>>
setForm(small)
<<create>>
move avatar
fireFlowerAcquired()
handleFireFlower()
addScore(1000)
grow()
<<create>>
setForm(shooter)
press fire button
fireButtonPressed
handleFireButton()
<<create>>

State

The intent of the State pattern does not identify the key indicator for the pattern.

- If the behavior of an object is defined by a finite state machine, then the State pattern is appropriate.
  - *No state machine = no need for this pattern.*
- The intent refers to the *internal* state of the object, and object state is not usually visible to external clients.
- Object behavior is defined by state machines more often than you might realize.
  - The state machine is often implicit rather than explicit: it emerges from the program logic.
  - It is often obfuscated in convoluted logic.
- Using a finite state machine to define behavior is a *very* important skill to develop.
  - If nothing else, it helps provide a clear description of expected behavior.

The State and Strategy patterns appear to have identical structures. The UML diagrams are nearly identical.

Both include a ***context*** that relies on some ***interface*** (the ***state*** or ***strategy***) to which it delegates some/all of its behavior.

Changing the interface from one object to another will cause the context to change its behavior.

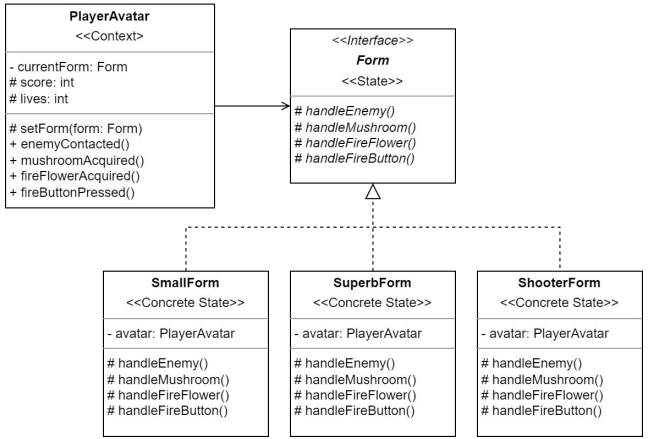
However, in the ***strategy*** pattern, some external ***client*** controls the change from one strategy to another.

But in the ***state*** pattern, the state of the context only changes based on its current state and some triggering event. External clients ***do not*** directly control the transition.

State

There are many issues to consider when implementing a State pattern.

- How do you define state changes?
  - *Naive approach - explicitly code all state changes.*
  - *Transition table - defines inputs, outputs, and transitions for all states.*
- Who is responsible for state changes?
  - *Concrete States - Easier to extend. Do the states have the information needed?*
    - *States become coupled. Why?*
  - *Context - Easier to understand.*
    - *Context potentially becomes coupled with all of the states. Why?*
- When are Concrete State objects constructed?
  - *Create all states when the Context is created.*
  - *Create/destroy states as needed.*
  - *Can Concrete States be shared by multiple context objects?*
- Some kind of event mechanism is needed.



In the Player Avatar Subsystem the ***states*** control the transitions in the ***context***, and the transitions are explicitly coded.

Q: How does this affect the encapsulation in the system? What is the best way to handle this in a language like Java?

State

There are several *consequences* to implementing the strategy pattern:

- *State-specific behavior is partitioned into separate objects.*
- *State-transitions become explicit and the context is protected from inconsistent internal states.*
- *Managing the transition between states can be messy or tricky.*
- *The context may be a "state holder."*
- *Class Explosion (lots of states).*