State Pattern

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

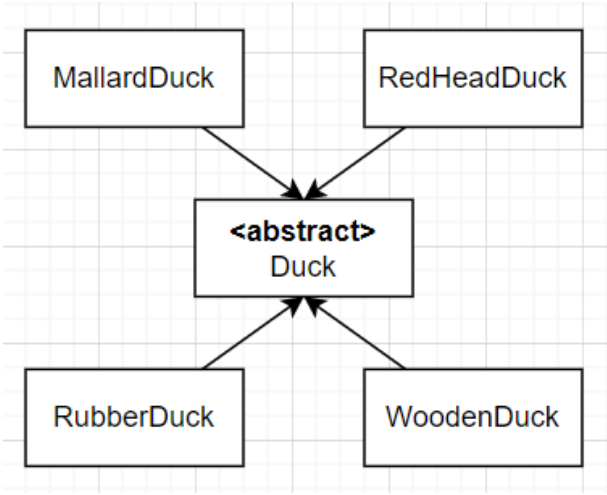
Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Thinking Back To Strategy Pattern Class

Duck Simulation Game: Setup

- Want to build a duck simulation game
  - There might be different types of ducks

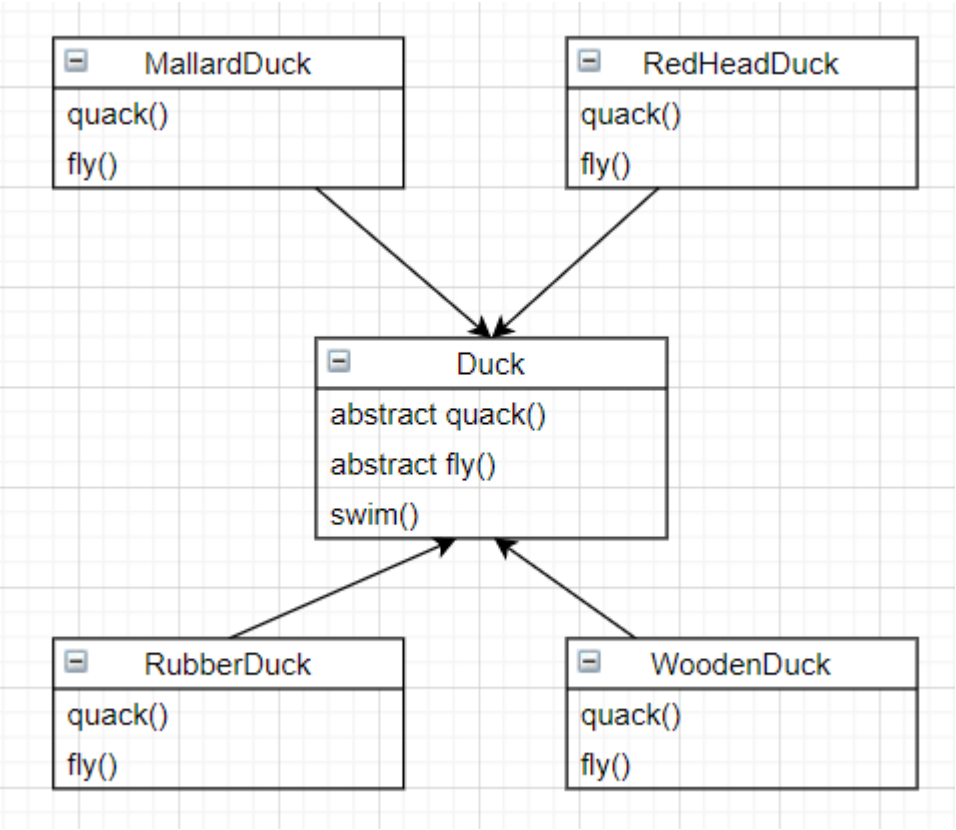


- Ducks should be able to quack, fly and swim
  - Different ducks may quack and fly *differently*

```
Duck duck = new SomeDuck();

duck.fly();
duck.swim();
duck.quack();
```

Duck Simulation Game: Problem

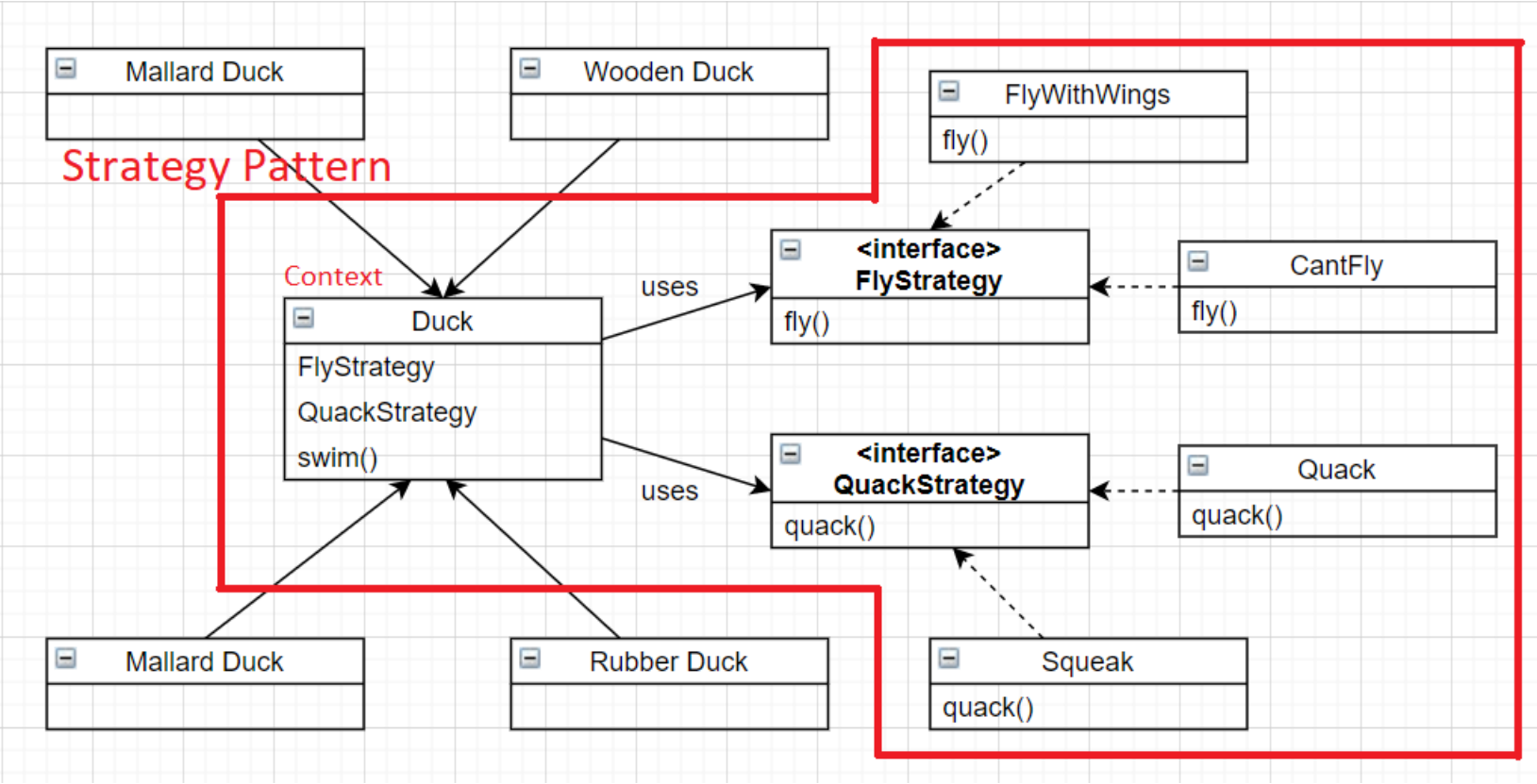
Idea: Use abstract methods



- Lots of code duplication
  - Might have to implement same functionality multiple times
- Runtime behaviour changes are not possible

Duck Simulation Game: Solution

Use strategy pattern instead!

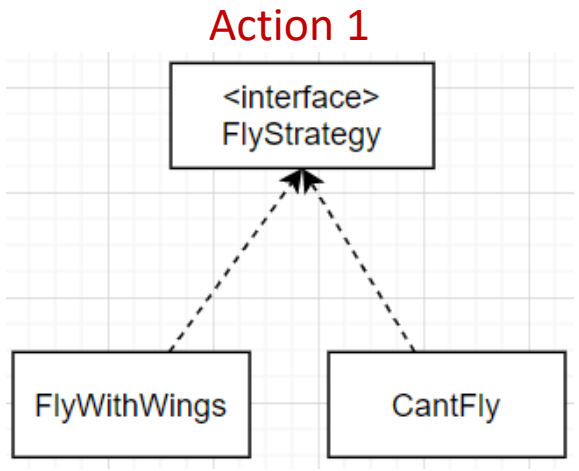


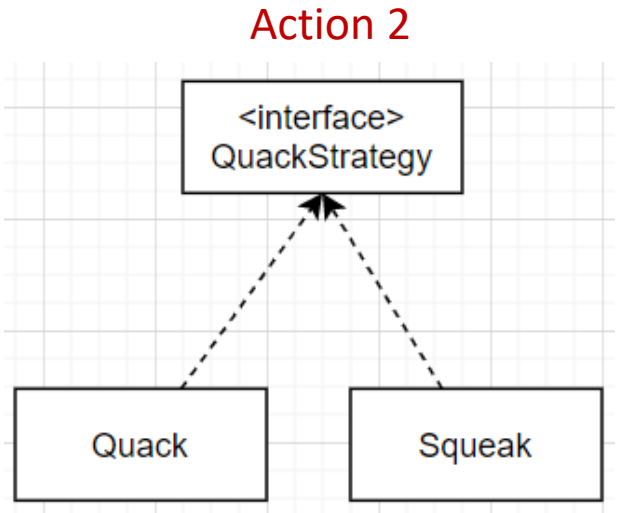
- Don’t have to write the same code twice
- Quack or fly behaviour of ducks can be changed at runtime

Strategy Pattern: Conclusion

- Strategy pattern allows context to change between different strategies defined for a single action

- Actions are in no way aware of each other





Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Extending The Duck Simulation Game

Executing Multiple Actions

```
void fly() {
  ...// gliding logic
}
```

```
void swim() {
  ...// floating logic
}
```

- Currently `duck.fly()` and `duck.swim()` can be called consecutively
  - This doesn't make sense. If duck is in air, how can it swim?? (and vice versa)

Executing Multiple Actions

```
void fly() {
  ...// gliding logic
}
```

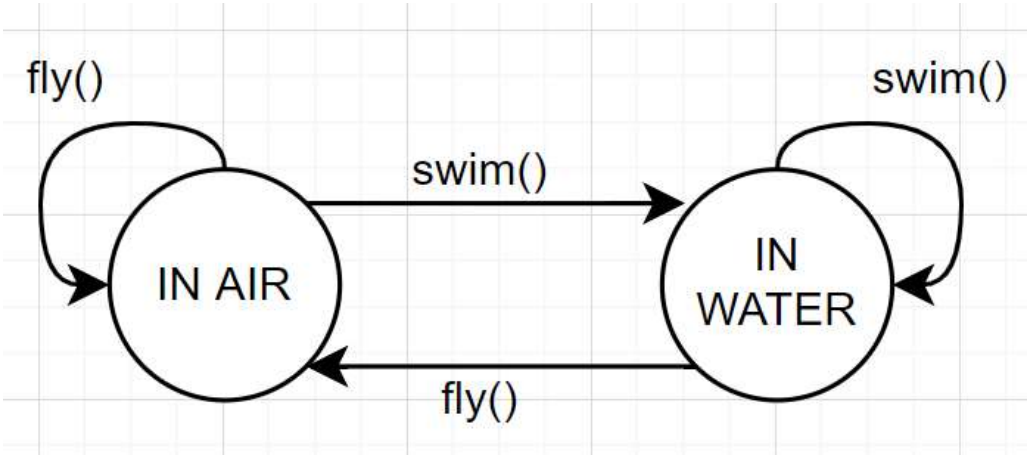
```
void swim() {
  ...// floating logic
}
```

- Currently `duck.fly()` and `duck.swim()` can be called concurrently
  - This doesn't make sense. If duck is in air, how can it swim?? (and vice versa)

- We need a way to represent the current state the duck is in.
  - And implement methods based on active state

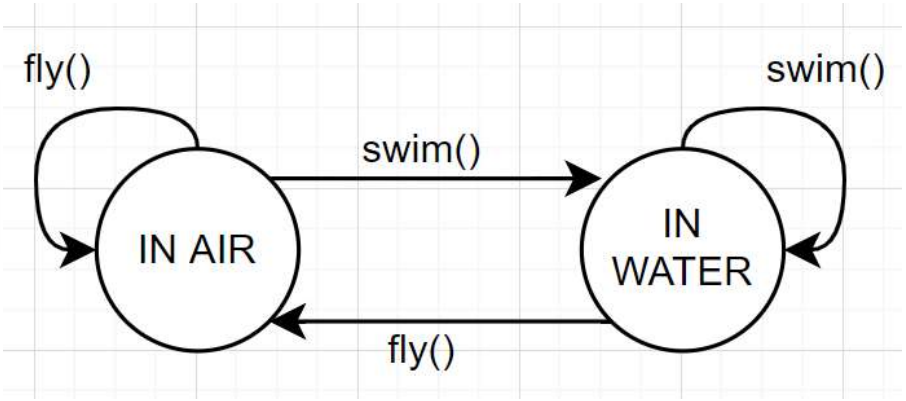
State Diagram

- Methods `fly()` and `swim()` methods should act differently based on state



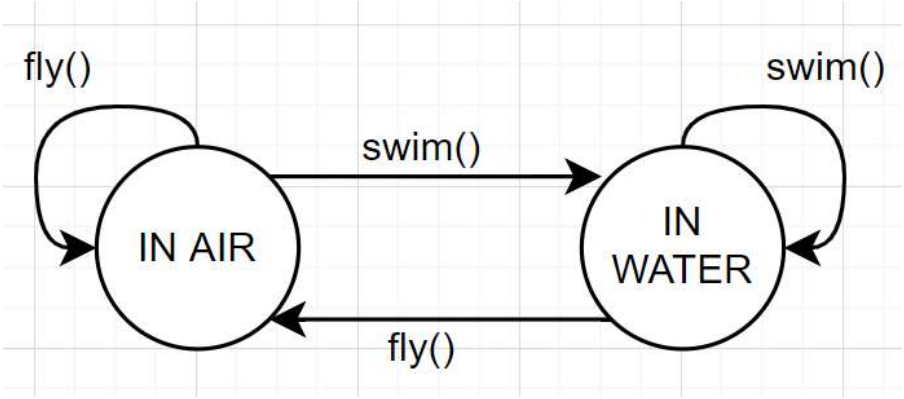
Redefining `fly()` and `swim()` Methods

```
void fly() {
  ...// gliding logic
}
```




```
void fly() {
  if (currentState == "IN AIR") {
    ...// gliding logic
  }
  else if (currentState == "IN WATER") {
    getOutOfWater();
    gainAltitude();
    currentState = "IN AIR";
  }
}
```

Redefining `fly()` and `swim()` Methods




```
void swim() {
  ...// floating logic
}
```

```
void swim() {
  if (currentState == "IN AIR") {
    loseAltitude();
    getInsideWater();
    currentState == "IN WATER";
  }
  else if (currentState == "IN WATER") {
    ...// floating logic
  }
}
```

Extending Duck Actions

Extending Duck Actions

- What if we want to add more states to duck?

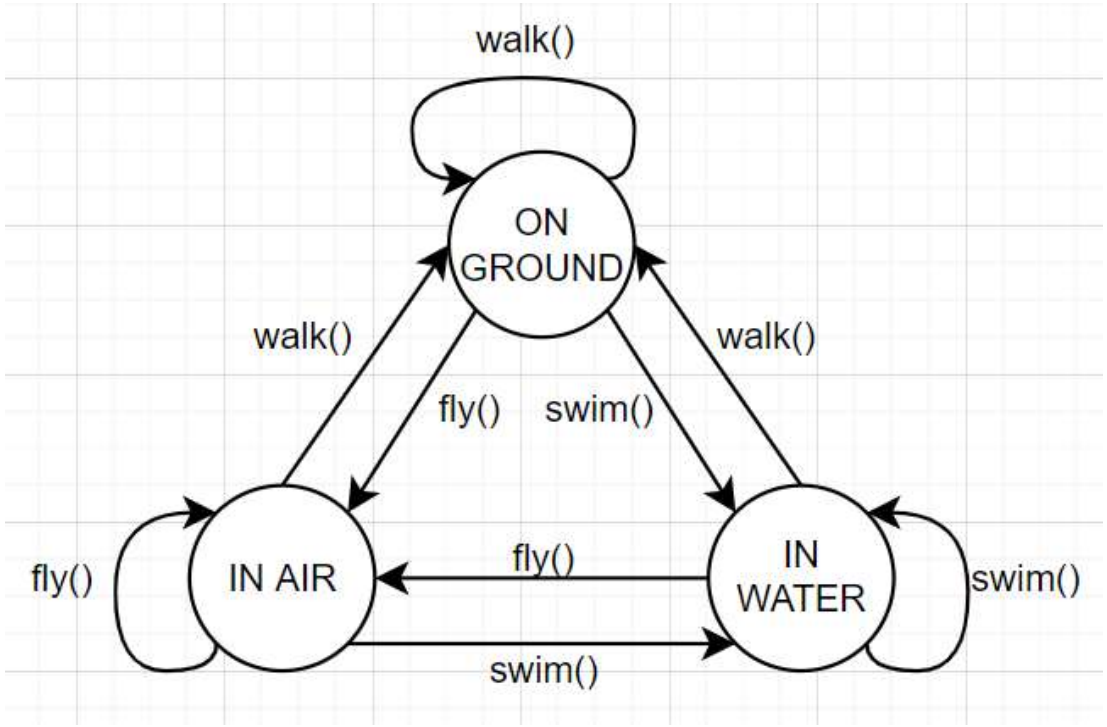
Extending Duck Actions

- What if we want to add more actions to duck?
  - Ducks can walk on ground as well!

Extending Duck Actions

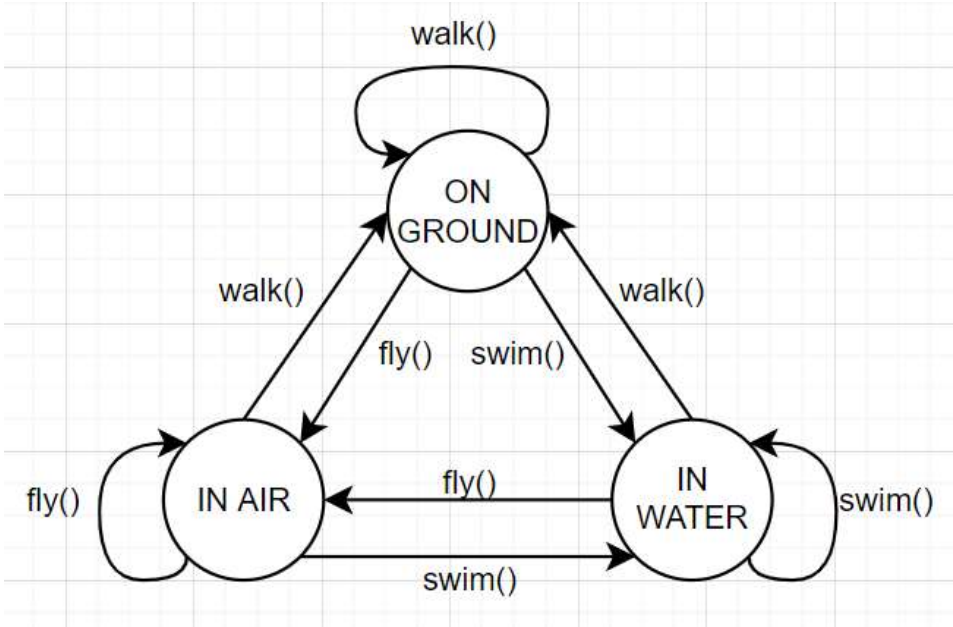
- What if we want to add more states to duck?
  - Ducks can walk on ground as well!

```
abstract class Duck() {
  String currentState;

  void fly() {...}
  void swim() {...}
  void walk() {???}
}
```



Extending Duck Actions

```
void fly() {
  if (currentState == "IN AIR") {...}
  else if (currentState == "IN WATER") {...}
  else if (currentState == "ON GROUND") {...}
}

void swim() {
  if (currentState == "IN AIR") {...}
  else if (currentState == "IN WATER") {...}
  else if (currentState == "ON GROUND") {...}
}

void walk() {
  if (currentState == "IN AIR") {...}
  else if (currentState == "IN WATER") {...}
  else if (currentState == "ON GROUND") {...}
}
```



Problems With Current Implementation

- Whenever new state added, code has to be modified
  - Encapsulate what varies!

Problems With Current Implementation

- Whenever new state added, code has to be modified
  - Encapsulate what varies!
- Context and states should be loosely coupled
  - Program to an interface not an implementation!

Constructing The State Pattern

Constructing The State Pattern

- Determine context
  - Context is the object whose states are managed by the state pattern. In our case `Duck` object.

Constructing The State Pattern (sep this)

- Determine context
  - Context is the object whose states are managed by the state pattern. In our case `Duck` object.

- Determine states
  - Determine all possible states context can take at any given time. In our case: `IN AIR` state, `IN WATER` state and `ON GROUND` state.

Constructing The State Pattern (sep this)

- Determine context
  - Context is the object whose states are managed by the state pattern. In our case `Duck` object.

- Determine states
  - Determine all possible states context can take at any given time. In our case: `IN AIR` state, `IN WATER` state and `ON GROUND` state.

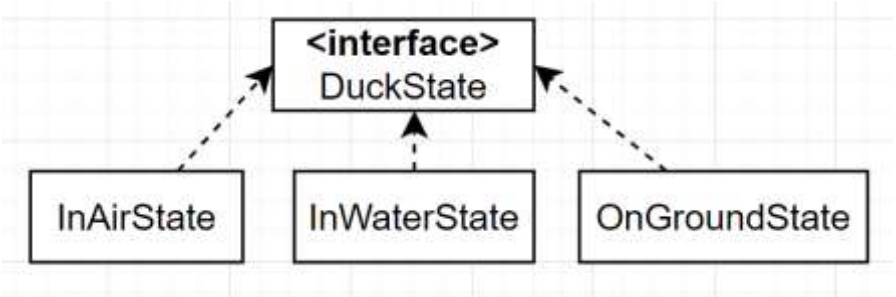
- Determine actions
  - Determine all possible actions which the context can execute. In our case: `fly()`, `swim()` or `walk()`

Constructing The State Pattern

- Context and states should be loosely coupled

Constructing The State Pattern

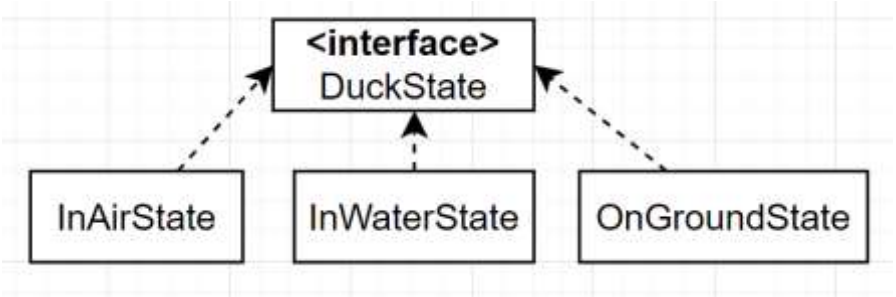
- Context and states should be loosely coupled



- When context wants to perform an action, it should not care what the current state is
  - Create a common interface for all states.

Constructing The State Pattern

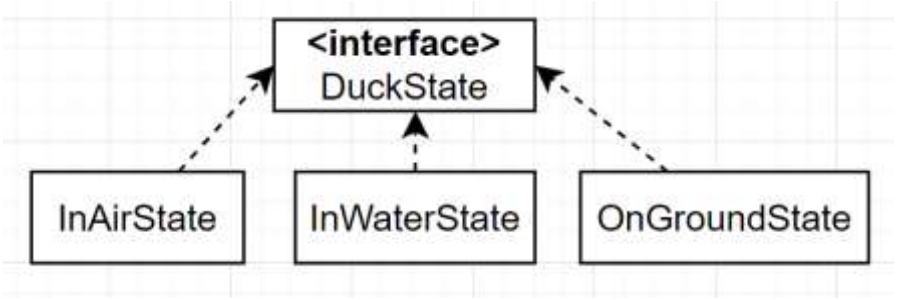
- Context and states should be loosely coupled
- When context wants to perform an action, it should not care what the current state is
  - Create a common interface for all states.
- Each state should be able to perform actions requested by context.
  - Define interface as all possible actions




```
interface DuckState {
  void fly();
  void swim();
  void walk();
}
```

State Classes

- States manage the context
  - They must have a reference to it




```
interface DuckState {
  void fly();
  void swim();
  void walk();
}
```

```
class InAirState implements DuckState {
  InAirState(Duck) {...}
  void fly() {...}
  void swim() {...}
  void walk() {...}
}

class InWaterState implements DuckState {
  InWaterState(Duck) {...}
  void fly() {...}
  void swim() {...}
  void walk() {...}
}

class OnGroundState implements DuckState {
  OnGroundState(Duck) {...}
  void fly() {...}
  void swim() {...}
  void walk() {...}
}
```

Context Class

- Have to preserve state information on each state
  - Can’t create new state object on each new state change
  - Instead create them once in context.
- Context should also keep a reference to currently active state

```
abstract class Duck {

  DuckState currentState;

  DuckState IN_AIR;
  DuckState IN_WATER;
  DuckState ON_GROUND;

  Duck() {
    IN_AIR = new InAirState(this);
    IN_WATER = new InWaterState(this);
    ON_GROUNR = new OnGroundState(this);
  }
}
```

Context Class Action Delegations

- After setting up state pattern, delegate context actions to state action

```
abstract class Duck {

  DuckState currentState;

  DuckState IN_AIR;
  DuckState IN_WATER;
  DuckState ON_GROUND;

  Duck() {
    IN_AIR = new InAirState(this);
    IN_WATER = new InWaterState(this);
    ON_GROUNR = new OnGroundState(this);
  }
```

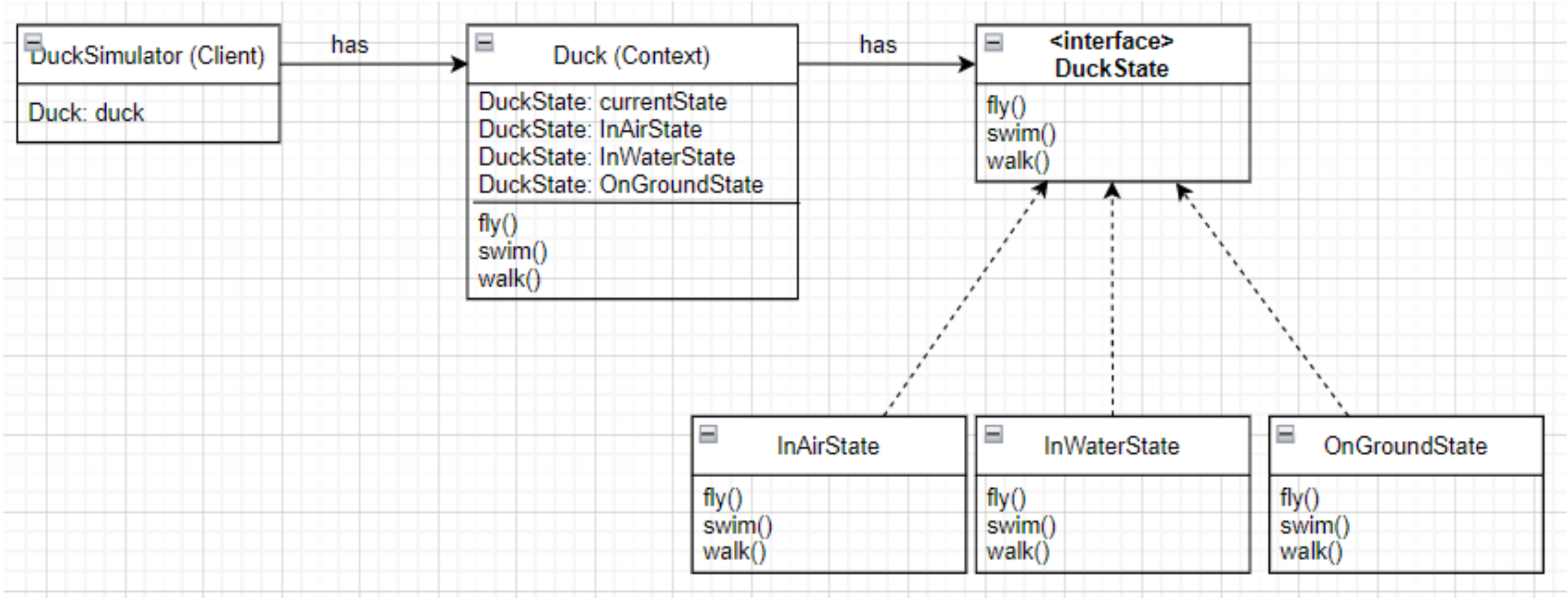
```
  void fly() {
    currentState.fly();
  }

  void swim() {
    currentState.swim();
  }

  void walk() {
    currentState.walk();
  }
}
```

Implementation Examples

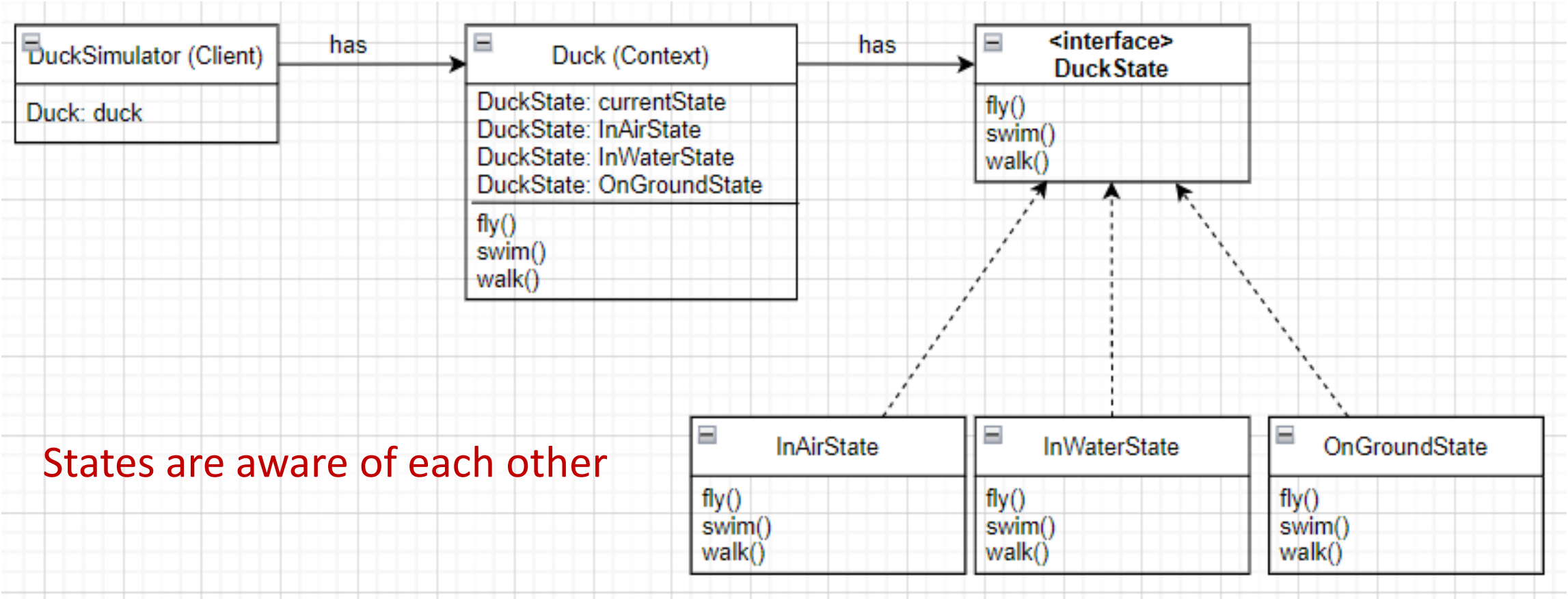
```
class InAirState implements DuckState {
  Duck duck;

  InAirState(Duck duck) {
    this.duck = duck;
  }

  fly() {
    ...// gliding logic
  }

  swim() {
    duck.loseAltitude();
    duck.getInWater();
    duck.currentState = duck.IN_WATER;
  }

  walk() {
    duck.loseAltitude();
    duck.landOnGround();
    duck.currentState = duck.ON_GROUND;
  }
}
```

```
class OnGroundState implements DuckState {
  Duck duck;

  OnGroundState(Duck duck) {
    this.duck = duck;
  }

  fly() {
    duck.takeOff();
    duck.gainAltitude();
    duck.currentState = duck.ON_GROUND;
  }

  swim() {
    duck.moveToPond();
    duck.getInWater();
    duck.currentState = duck.IN_WATER
  }

  walk() {
    ...// walking logic
  }
}
```

Overall uml diagram



Overall uml diagram

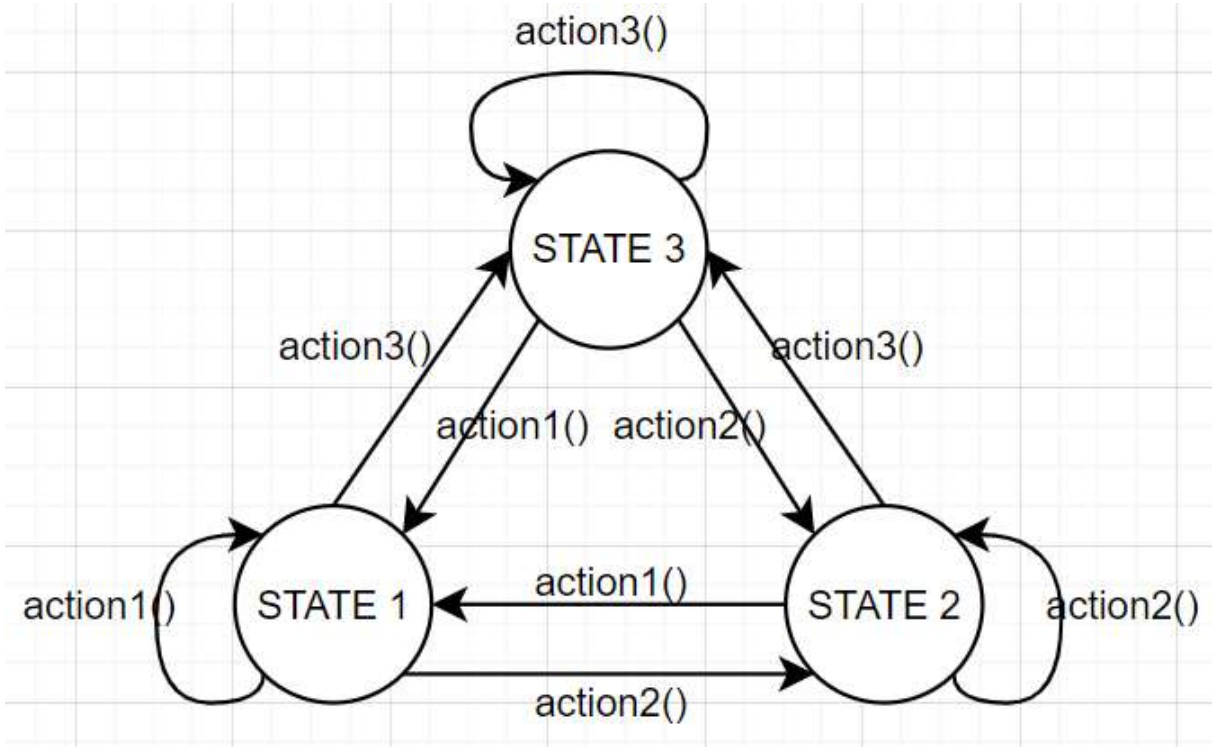


States are aware of each other

Unwanted Transitions

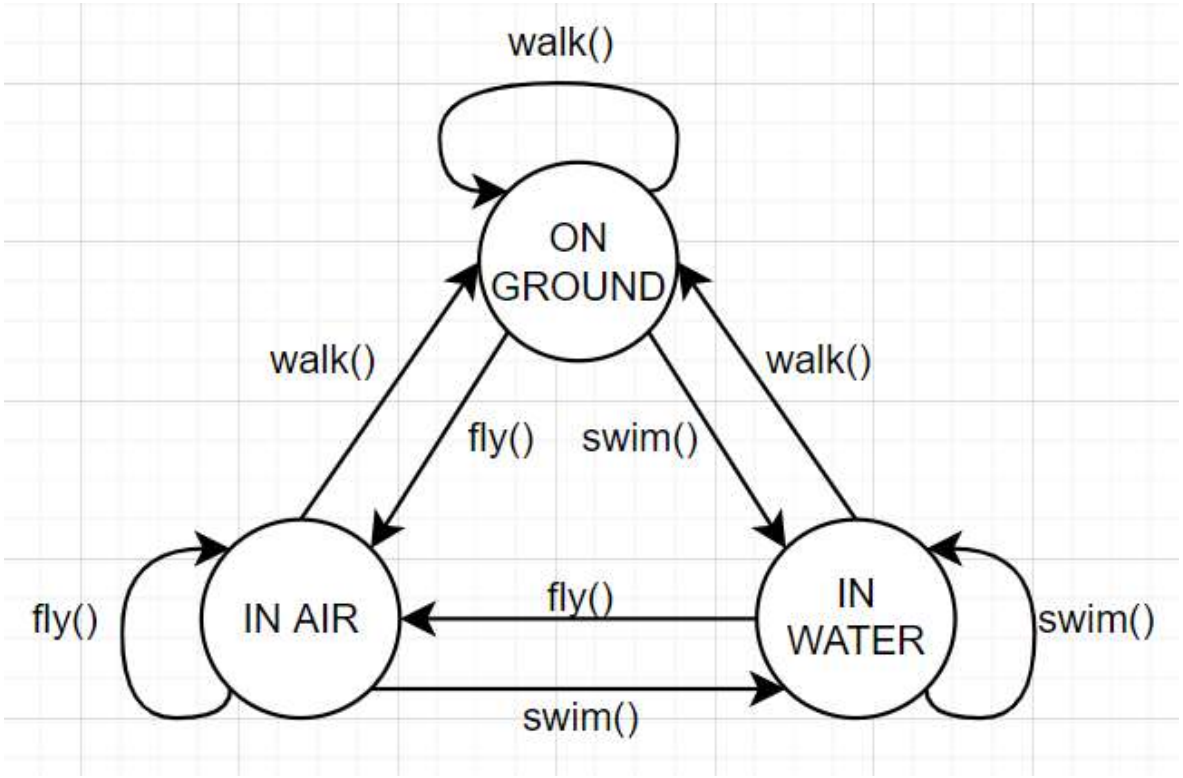
What About Unwanted Transitions?

- For previous implementation, state diagram is fully connected graph



What About Unwanted Transitions?

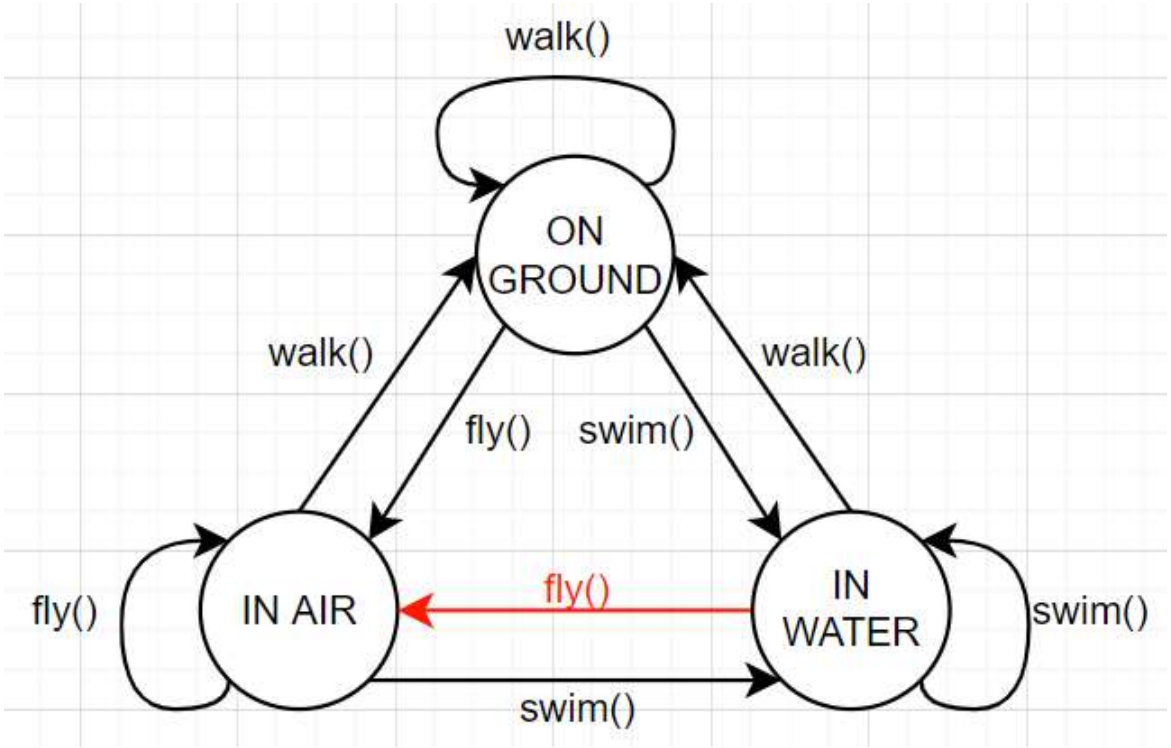
- For previous implementation, state diagram is fully connected graph
  - That is what we wanted!



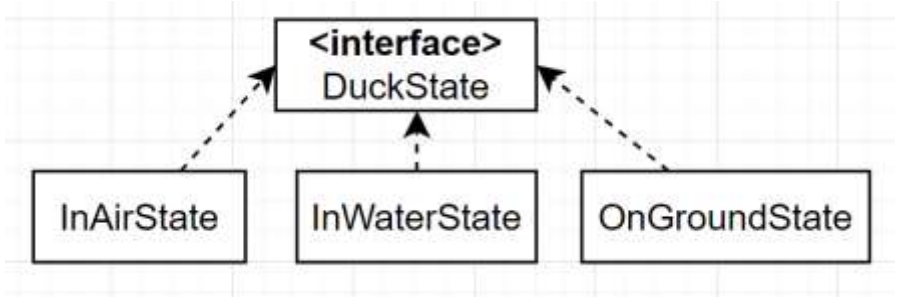
What About Unwanted Transitions?

- For previous implementation, state diagram is fully connected graph
  - That is what we wanted!

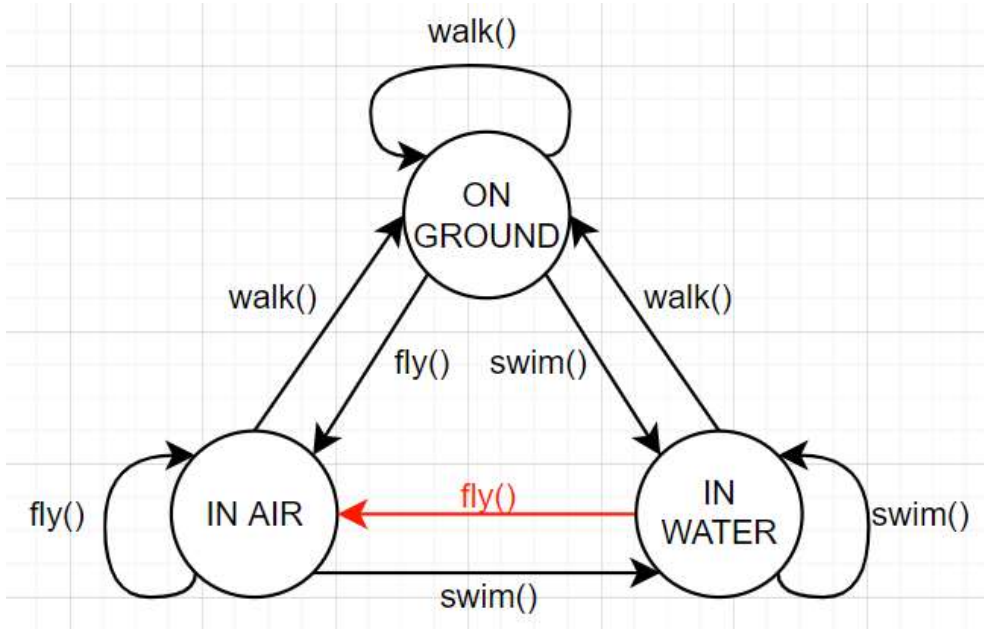
- Maybe duck shouldn't be able to directly go from IN WATER to IN AIR
  - Don't want the red transition



What About Unwanted Transitions?




```
interface DuckState {
  void fly();
  void swim();
  void walk();
}
```



- Design forces us to implement this function
  - Disadvantage of state pattern, more on next slides

What About Unwanted Transitions?

- What to do about forced function?

What About Unwanted Transitions?

- What to do about forced function?

```
class InWaterState implements DuckState {

  void fly() {
    print("WARNING: can't go from fly to swim");
  }

  void swim() {...}
  void walk() {...}
}
```

What About Unwanted Transitions?

- What to do about forced function?

```
class InWaterState implements DuckState {

  void fly() {
    print("WARNING: can't go from fly to swim");
  }

  void swim() {...}
  void walk() {...}
}
```

```
class InWaterState implements DuckState {

  void fly() throws Exception {
    throw new ImpossibleActionException();
  }

  void swim() {...}
  void walk() {...}
}
```

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Single Responsibility Principle

Single Responsibility Principle

- Single Responsibility Principle suggests each class should be responsible from single functionality.

Single Responsibility Principle

- Single Responsibility Principle suggests each class should be responsible from single functionality.

- In initial design, duck class contained:
  - low level functions: `gainAltitude()`, `loseAltitude()`, `getInWater()` etc.
  - high level functions: `fly()`, `swim()`, etc.

Single Responsibility Principle

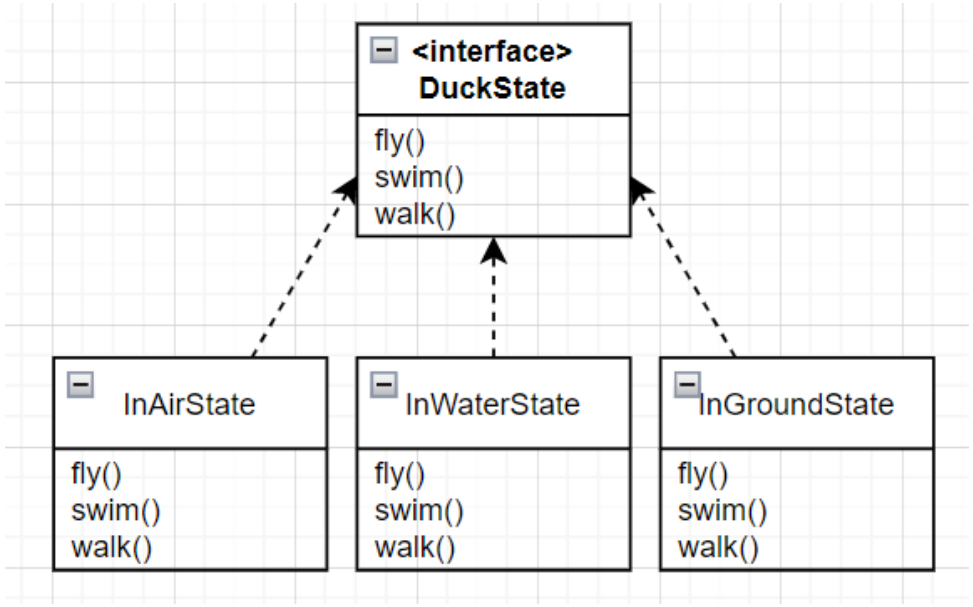
- Single Responsibility Principle suggests each class should be responsible from single functionality.

- In initial design, duck class contained:
  - low level functions: `gainAltitude()`, `loseAltitude()`, `getInWater()` etc.
  - high level functions: `fly()`, `swim()`, etc.

- This was against SRP principle
  - State pattern fixed it by encapsulating the state logic in different module

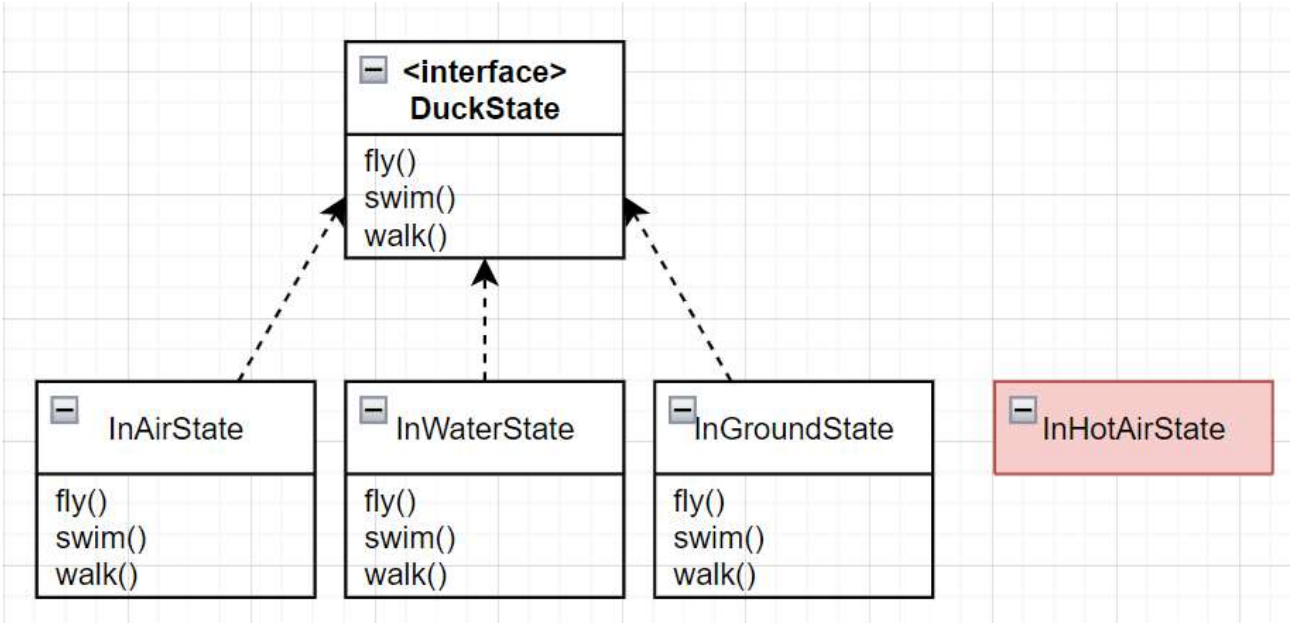
Open Closed Principle

Open Closed Principle



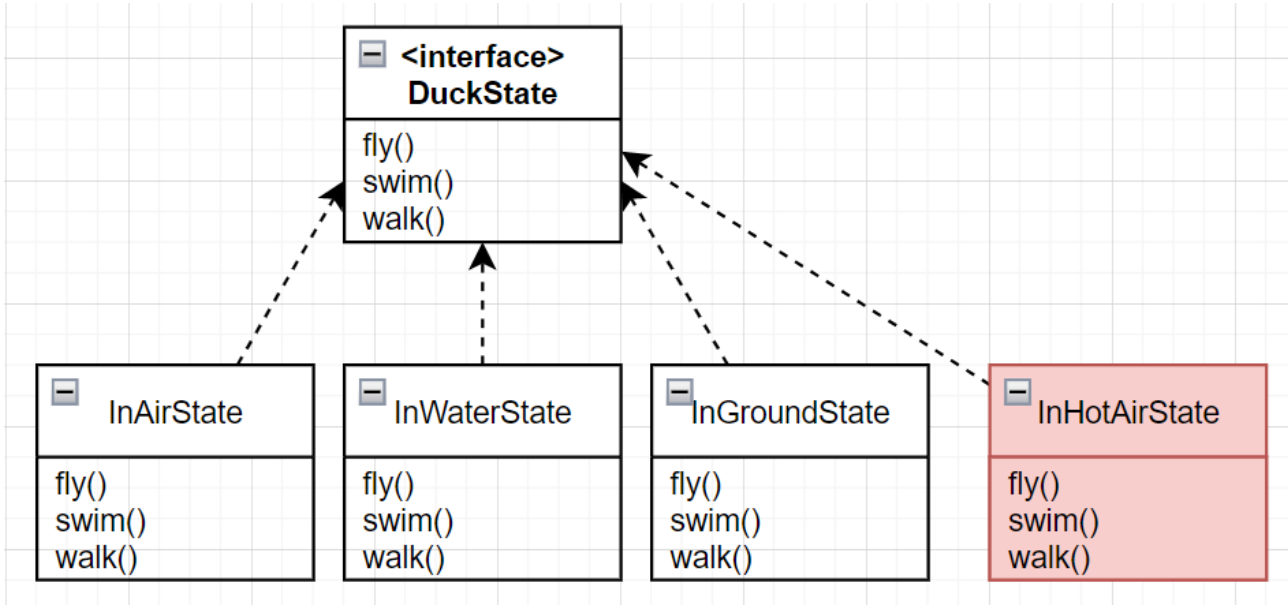
Open Closed Principle

- Extending States



Open Closed Principle

- Extending States → Just implement the interface



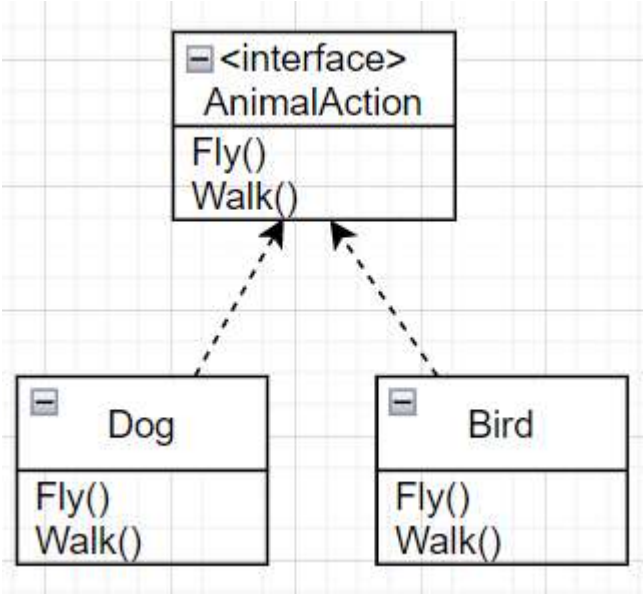
Interface Segregation Principle

Interface Segregation Principle

- What is interface segregation?

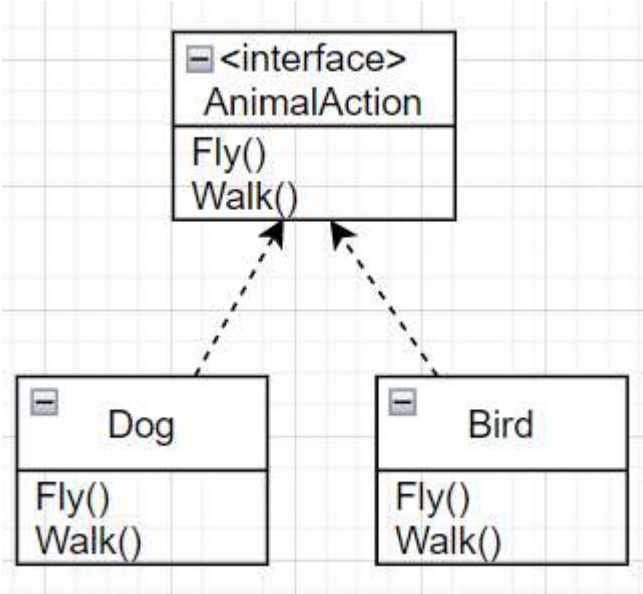
Interface Segregation Principle

- What is interface segregation?



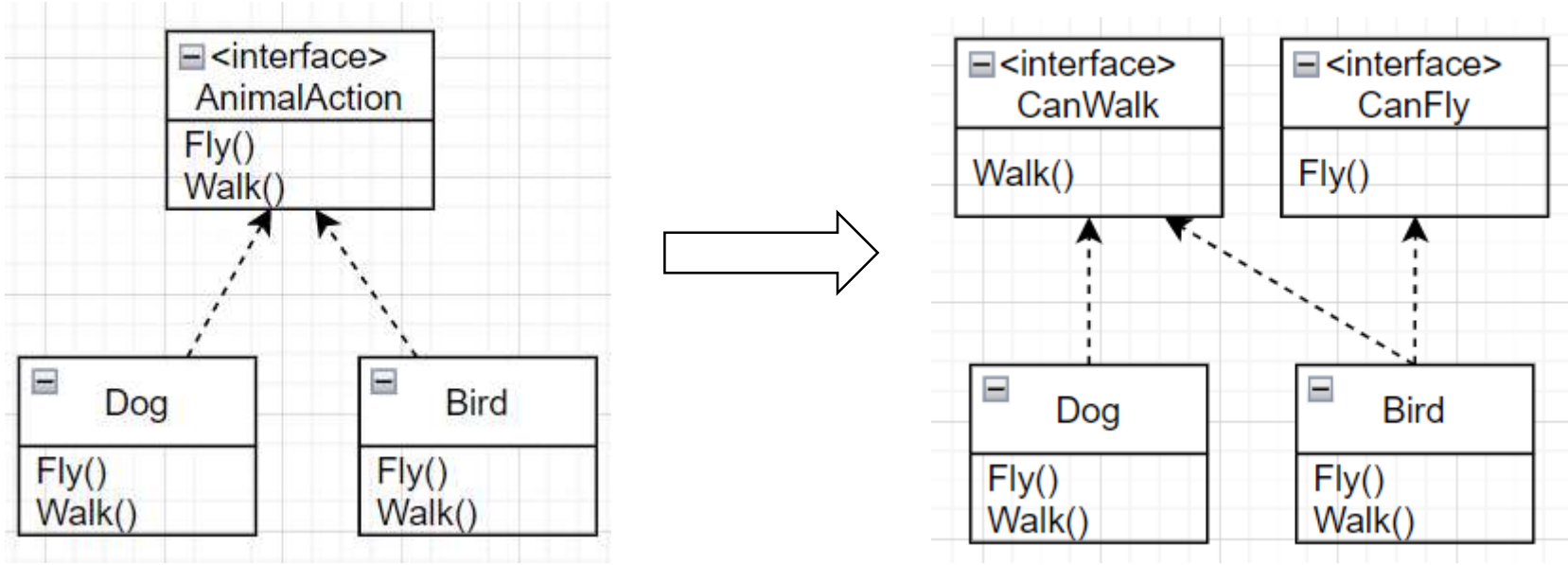
Interface Segregation Principle

- What is interface segregation?
  - Dog implements uneccessary fly()



Interface Segregation Principle

- What is interface segregation?
  - Dog implements uneccessary fly()
  - Seperate the interfaces



Interface Segregation Principle

- Does state pattern follow isp?

Interface Segregation Principle

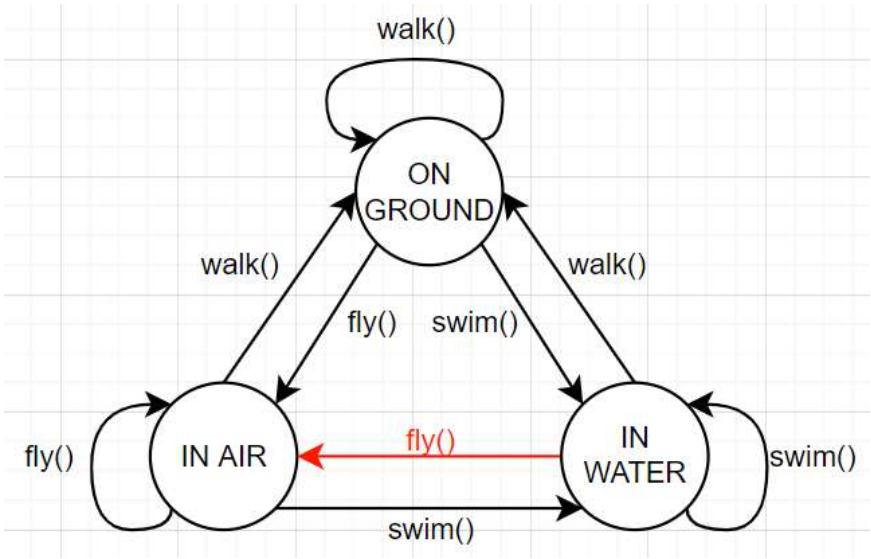
- Does state pattern follow isp?
  - Exactly the opposite…

```
interface DuckState {
  void fly();
  void swim();
  void walk();
}
```

Interface Segregation Principle

- Does state pattern follow isp?
  - Exactly the opposite…

```
interface DuckState {
  void fly();
  void swim();
  void walk();
}
```

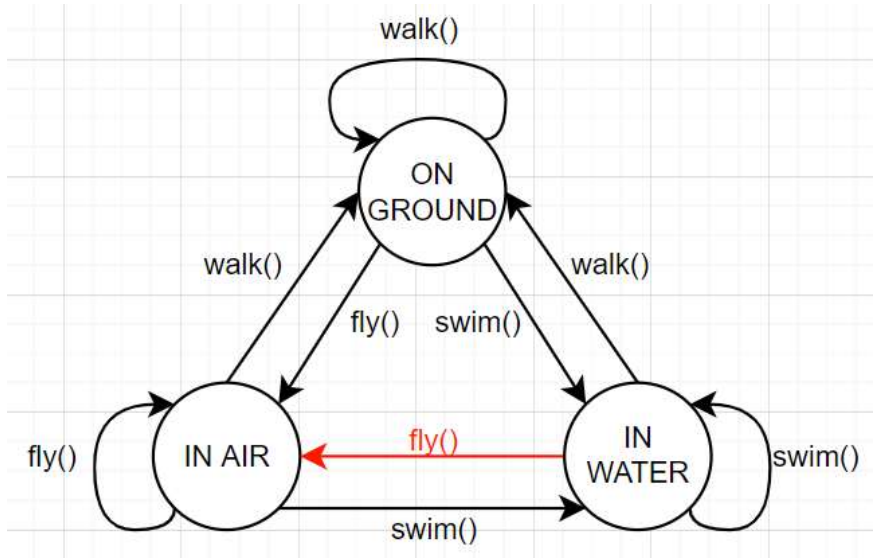


Interface Segregation Principle

- Does state pattern follow isp?
  - Exactly the opposite…

```
interface DuckState {
  void fly();
  void swim();
  void walk();
}
```

```
class InWaterState implements DuckState {

  void fly() {
    print("WARNING: can't go from fly to swim");
  }

  void swim() {...}
  void walk() {...}
}
```

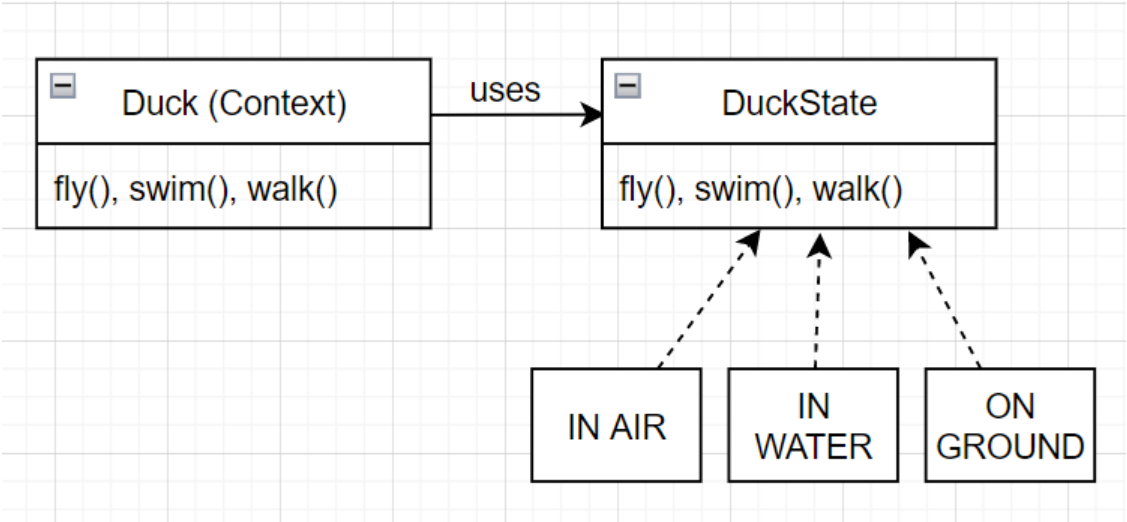


Interface Segregation Principle

- Does state pattern follow isp?
  - Exactly the opposite…
  - …but for a good reason!

Interface Segregation Principle

- Does state pattern follow isp?
  - Exactly the opposite…
  - …but for a good reason!
    - Context does not have to care for it’s state



Dependency Inversion Principle

Dependecy Inversion principle

- Overall goal of state pattern:
  - Context should perform actions (fly, swim, walk)...

Dependecy Inversion principle

- Overall goal of state pattern:
  - Context should perform actions (fly, swim, walk)...
  - …without caring for current state

Dependecy Inversion principle

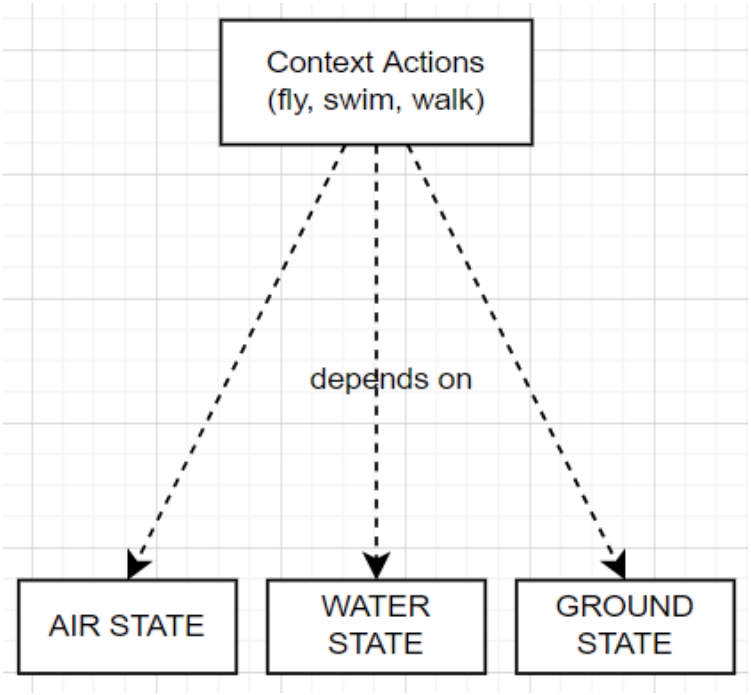
- Overall goal of state pattern:
  - Context should perform actions (fly, swim, walk)... → high level module
  - …without caring for current state → low level module

Dependecy Inversion principle

- Dependency Inversion Principle:
  - Higher level modules should not depend on lower modules

Dependecy Inversion principle

- Dependency Inversion Principle:
  - Higher level modules should not depend on lower modules



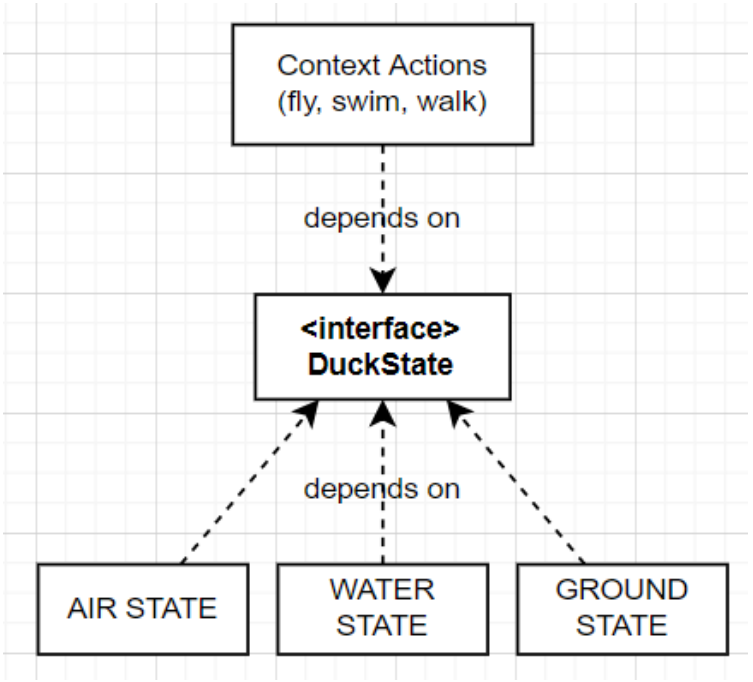

```
void fly() {
  if (currentState == "IN AIR") {...}
  else if (currentState == "IN WATER") {...}
  else if (currentState == "ON GROUND") {...}
}

void swim() {
  if (currentState == "IN AIR") {...}
  else if (currentState == "IN WATER") {...}
  else if (currentState == "ON GROUND") {...}
}

void walk() {
  if (currentState == "IN AIR") {...}
  else if (currentState == "IN WATER") {...}
  else if (currentState == "ON GROUND") {...}
}
```

Dependecy Inversion principle

- Dependency Inversion Principle:
  - Higher level modules should not depend on lower modules
  - Both should depend on abstraction

Dependecy Inversion principle

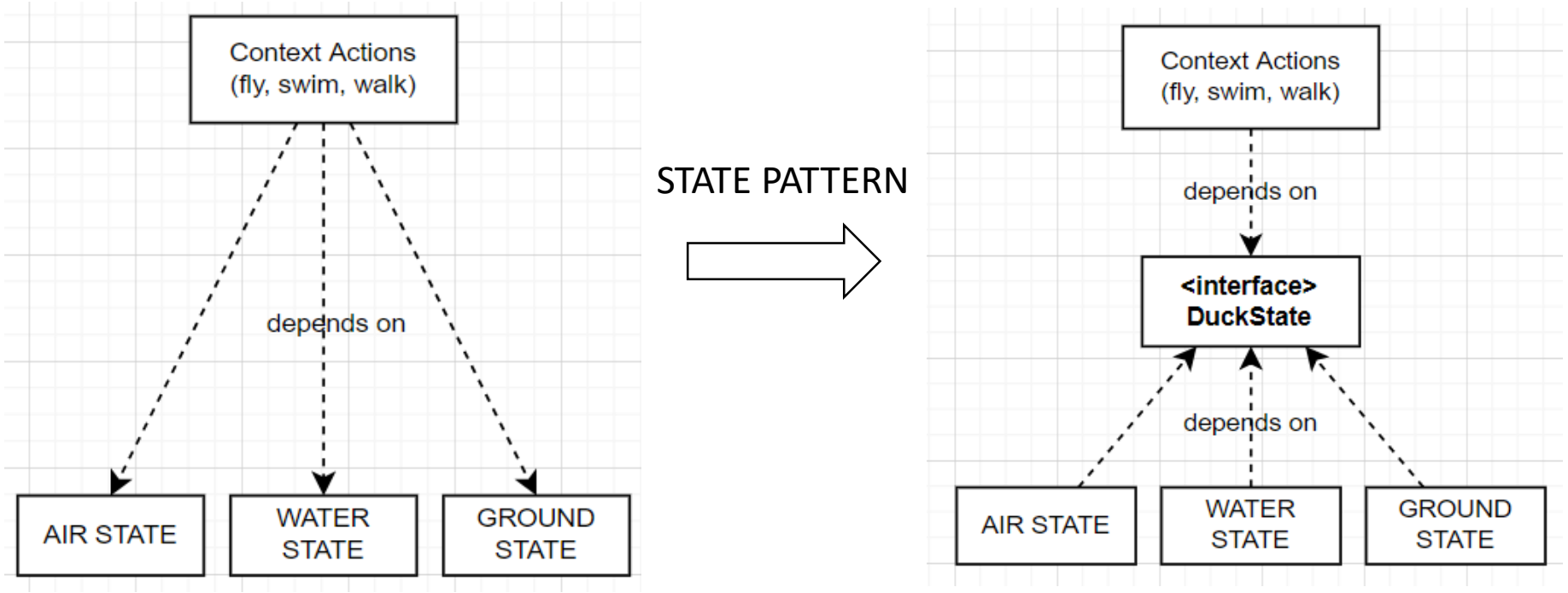
- Dependency Inversion Principle:
  - Higher level modules should not depend on lower modules
  - Both should depend on abstraction




```
class Duck {

  DuckState currentState;
  ...

  void fly() {
    currentState.fly();
  }

  void swim() {
    currentState.swim();
  }

  void walk() {
    currentState.walk();
  }

}
```

Dependecy Inversion principle

- Dependency Inversion Principle:
  - Higher level modules should not depend on lower modules
  - Both should depend on abstraction



Presentation Agenda

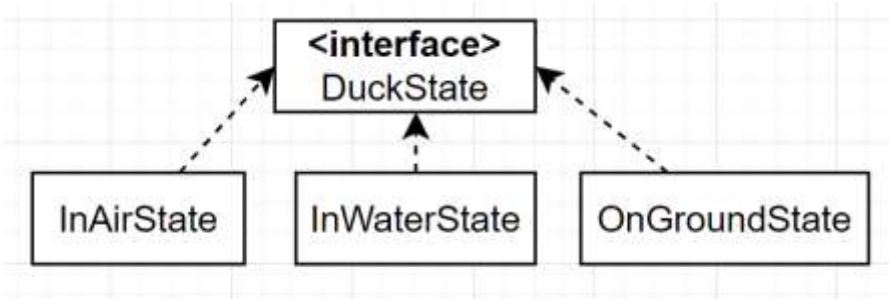
STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Advantages

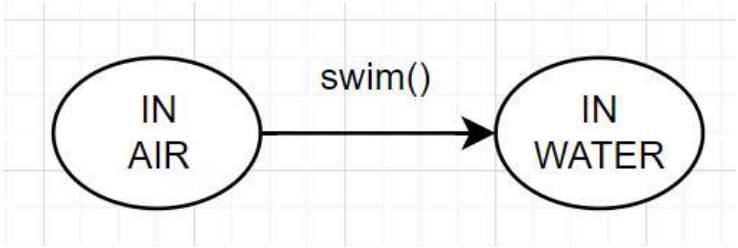
- Encapsulates the varying parts
  - Code is easier to maintain



Advantages

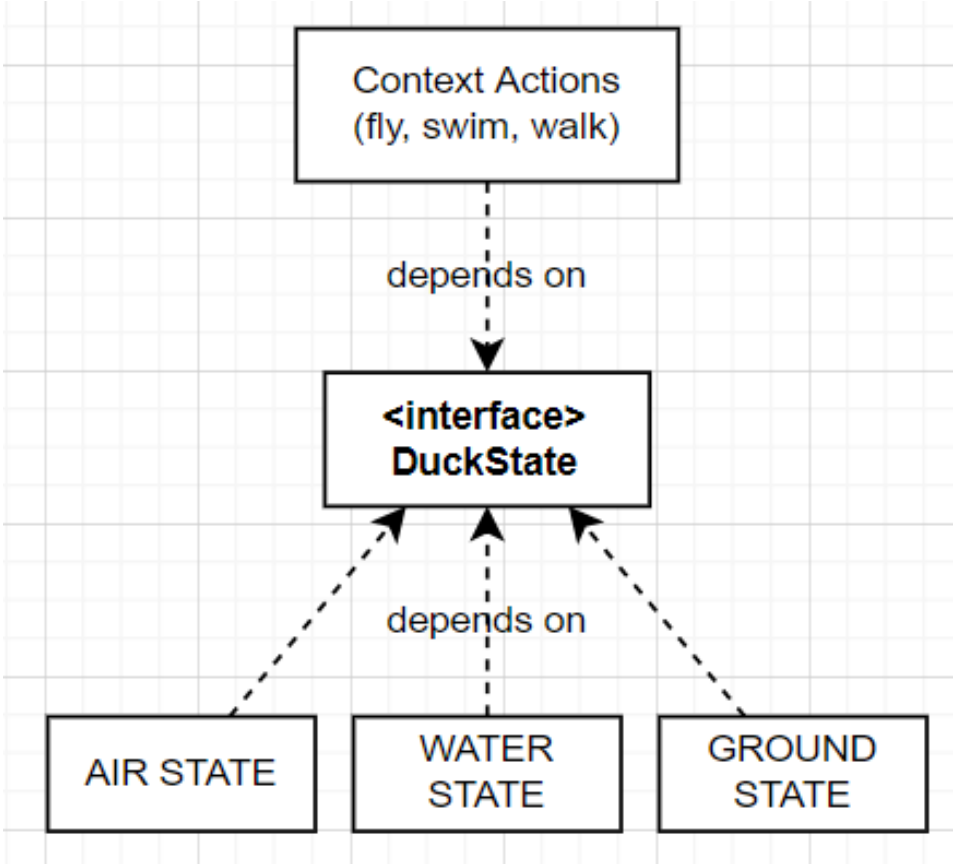
- Encapsulates the varying parts
  - Code is easier to maintain
- Implementing logic becomes intuitive

```
class InAirState {

  swim() {
    duck.loseAltitude();
    duck.getInWater();
    duck.currentState = duck.IN_WATER;
  }
}
```



Advantages

- Encapsulates the varying parts
  - Code is easier to maintain
- Implementing logic becomes intuitive
- Inverts dependencies
  - More flexible code

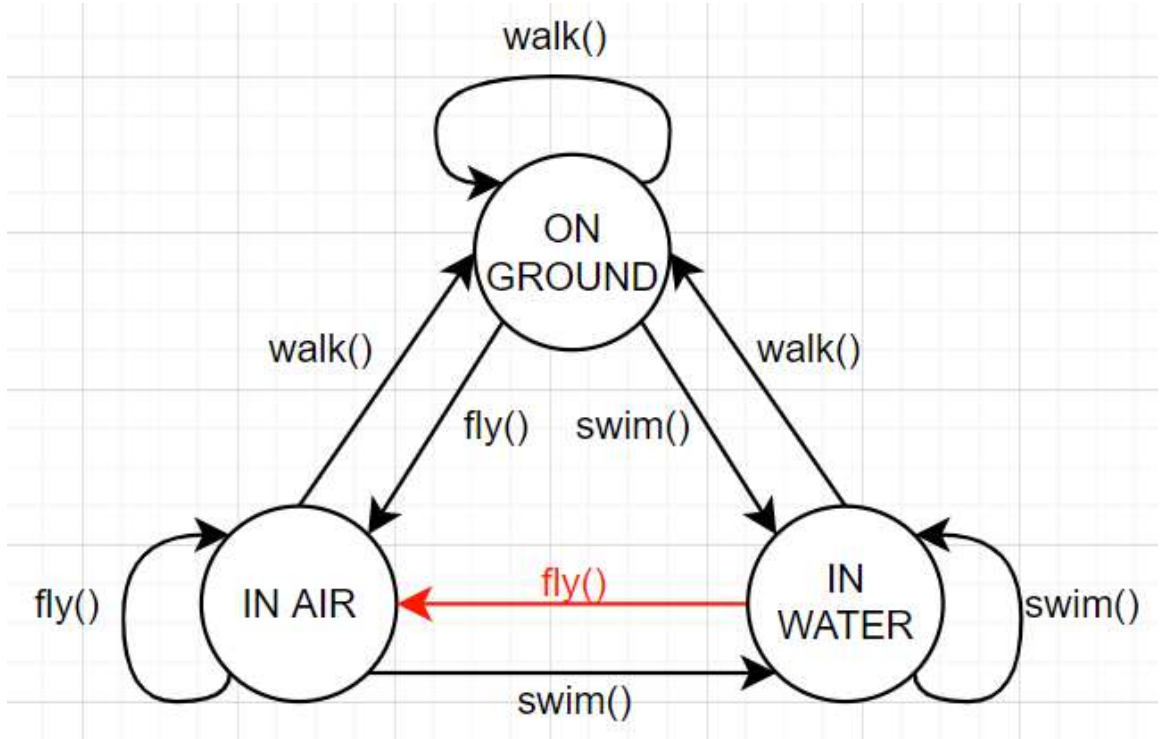


Disadvantages

- Violates interface segregation principle

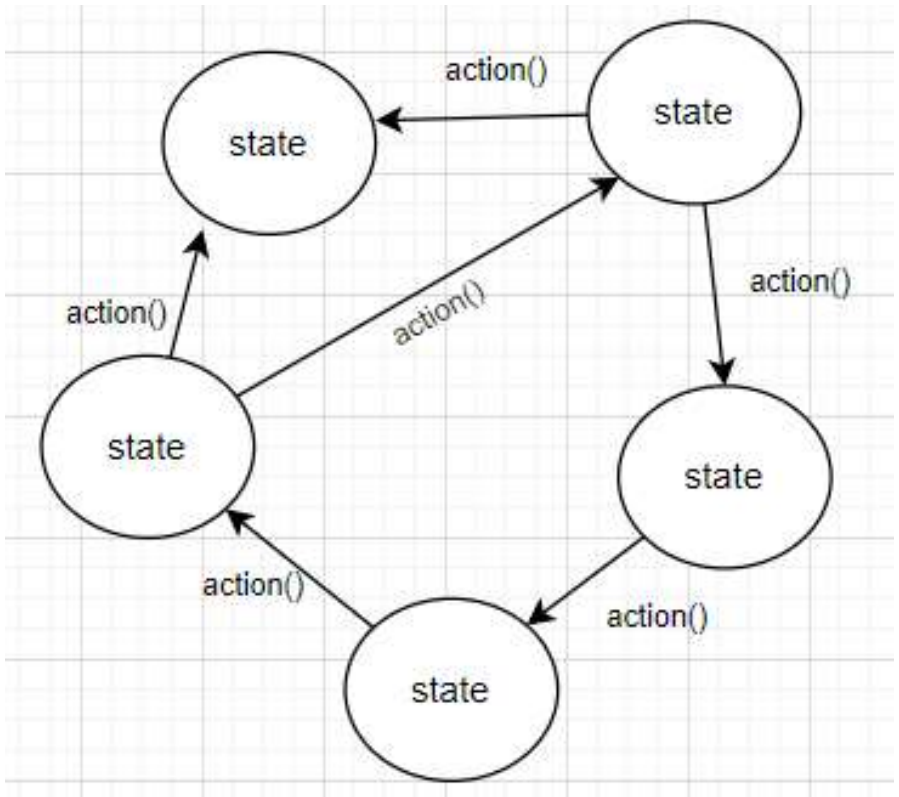
Disadvantages

- Violates interface segregation principle
  - Forces implementation
  - May cause bloated code



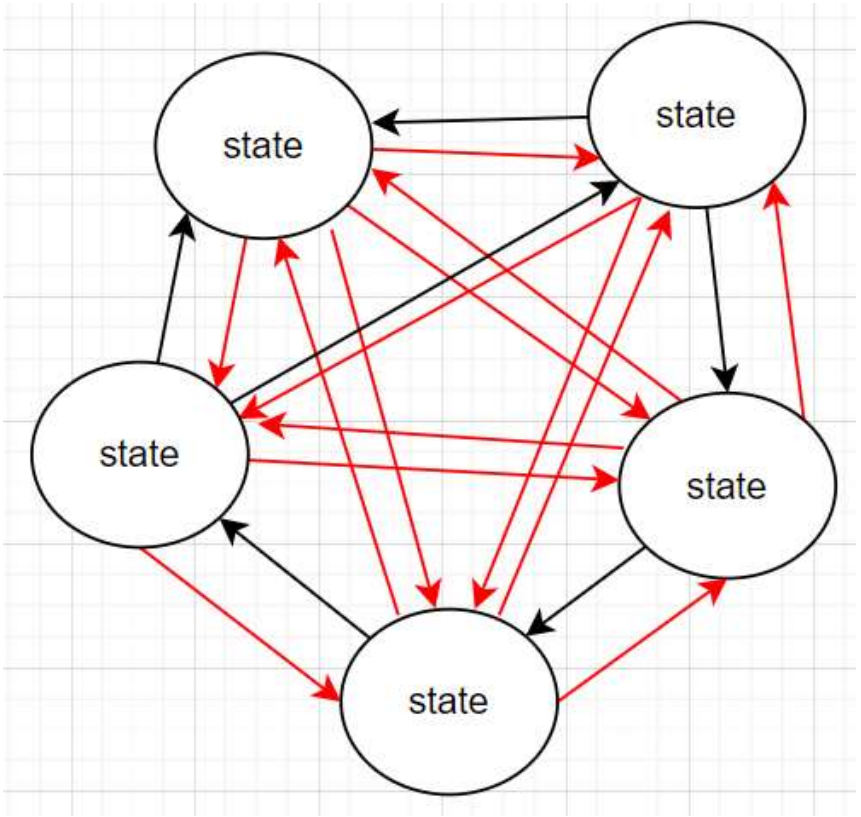
Disadvantages

- Violates interface segregation principle
  - Forces implementation
  - May cause bloated code
  - Especially for sparse graphs!



Disadvantages

- Violates interface segregation principle
  - Forces implementation
  - May cause bloated code
  - Especially for sparse graphs!



Presentation Agenda

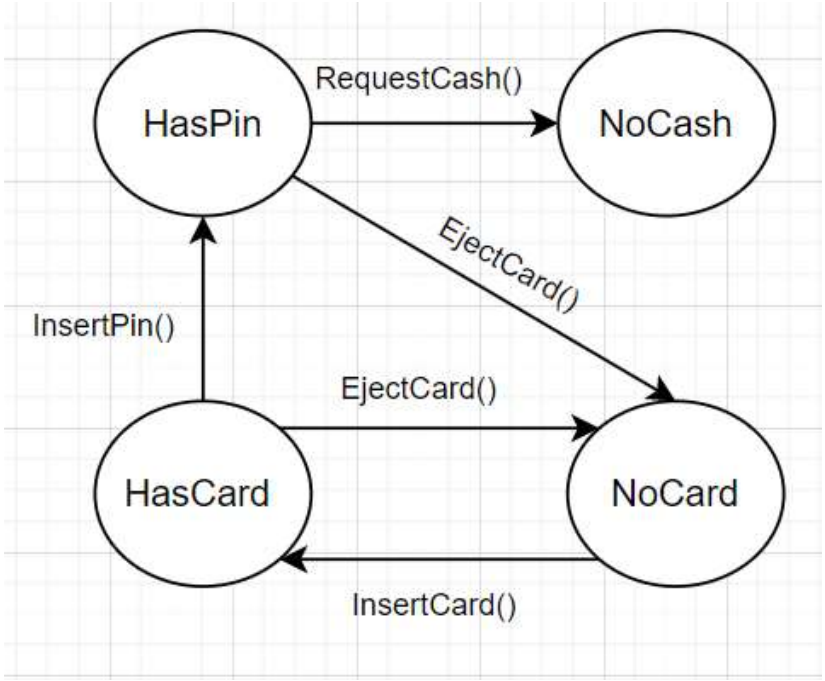
STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

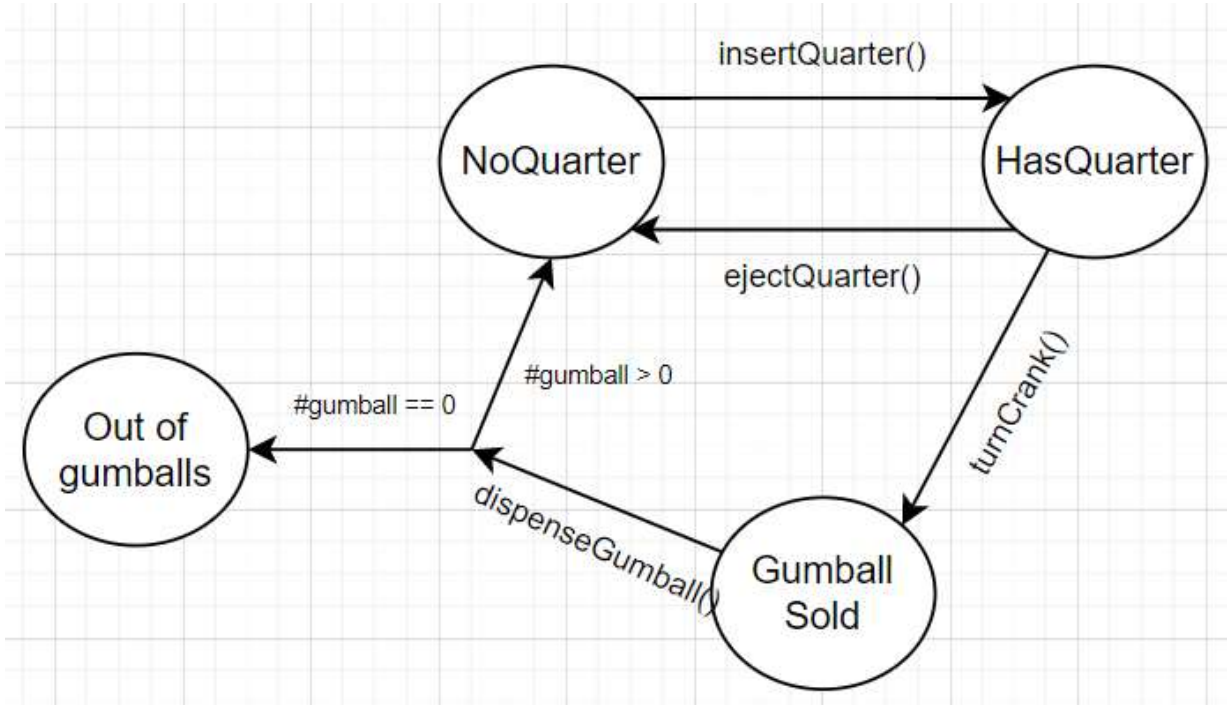
ATM Machine

- Actions: InsertCard, EjectCard, InsertPin, RequestCash
- States: HasCard, NoCard, HasPin, NoCash



Gumball Machine

- Actions: insertQuarter(), ejectQuarter(), turnCrank(), dispenseGumball()
- States: HasQuarter, NoQuarter, GumballSold, OutOfGumballs



Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

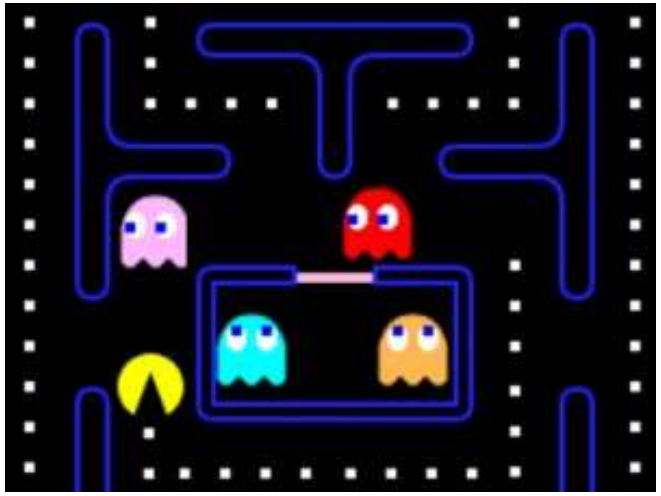
STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Concept of NPCs

- NPCs are non-playable characters in games



States of a NPC

- NPCs operate in terms of states

States of a NPC

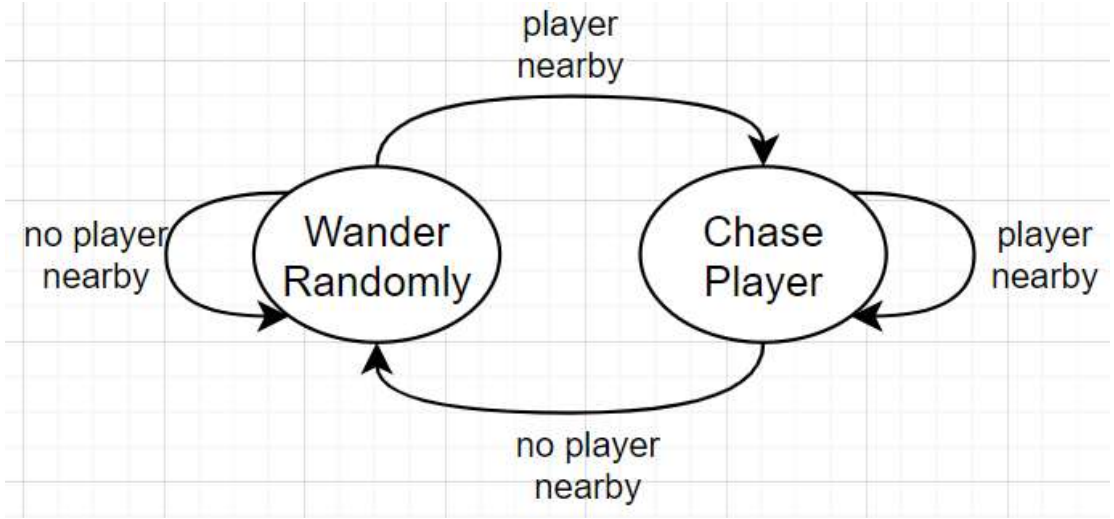
- NPCs operate in terms of states
  - If no player, wander around randomly

States of a NPC

- NPCs operate in terms of states
  - If no player, wander around randomly
  - If player nearby, chase it

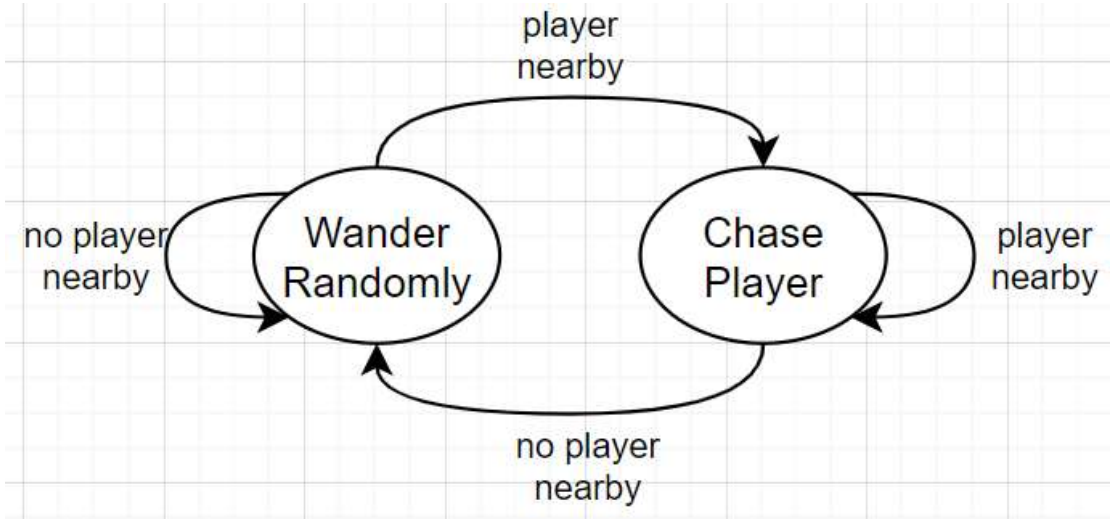
States of a NPC

- NPCs operate in terms of states
  - If no player, wander around randomly
  - If player nearby, chase it



States of a NPC

- State pattern is widely used in industry!



Presentation Agenda

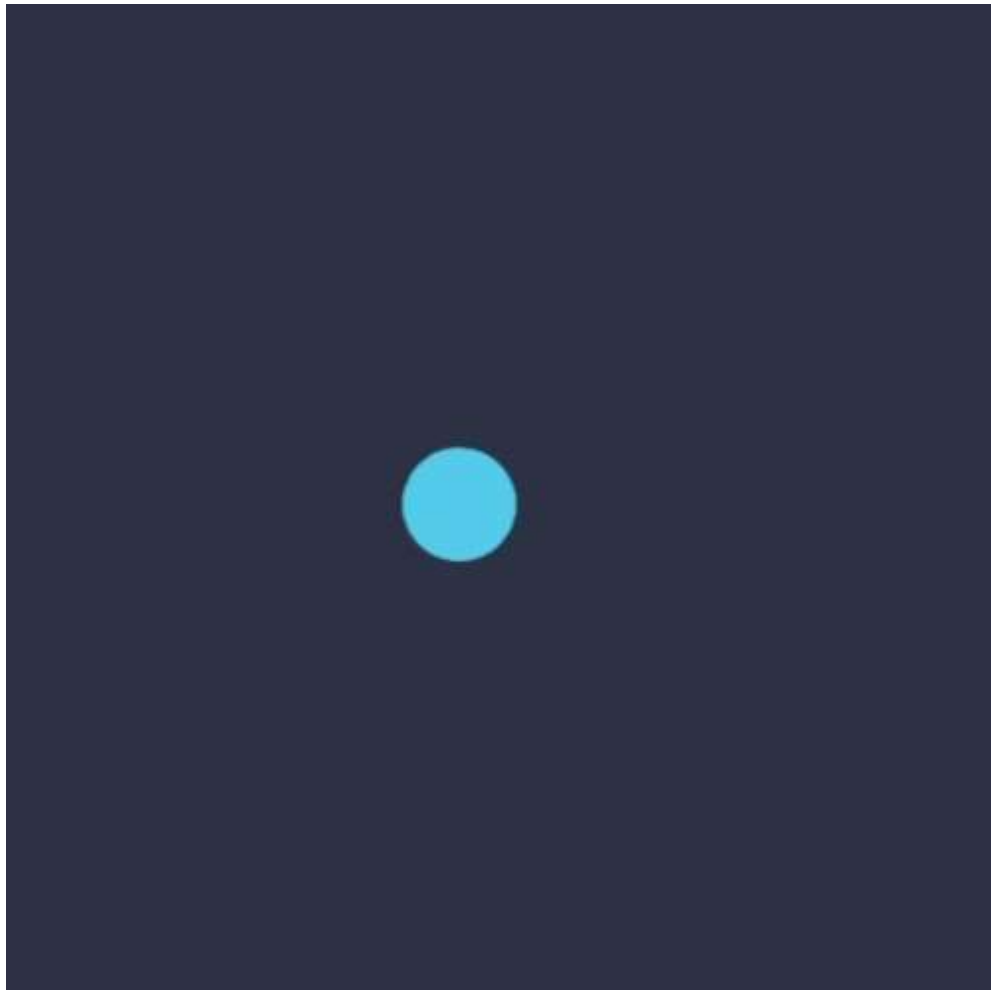
STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

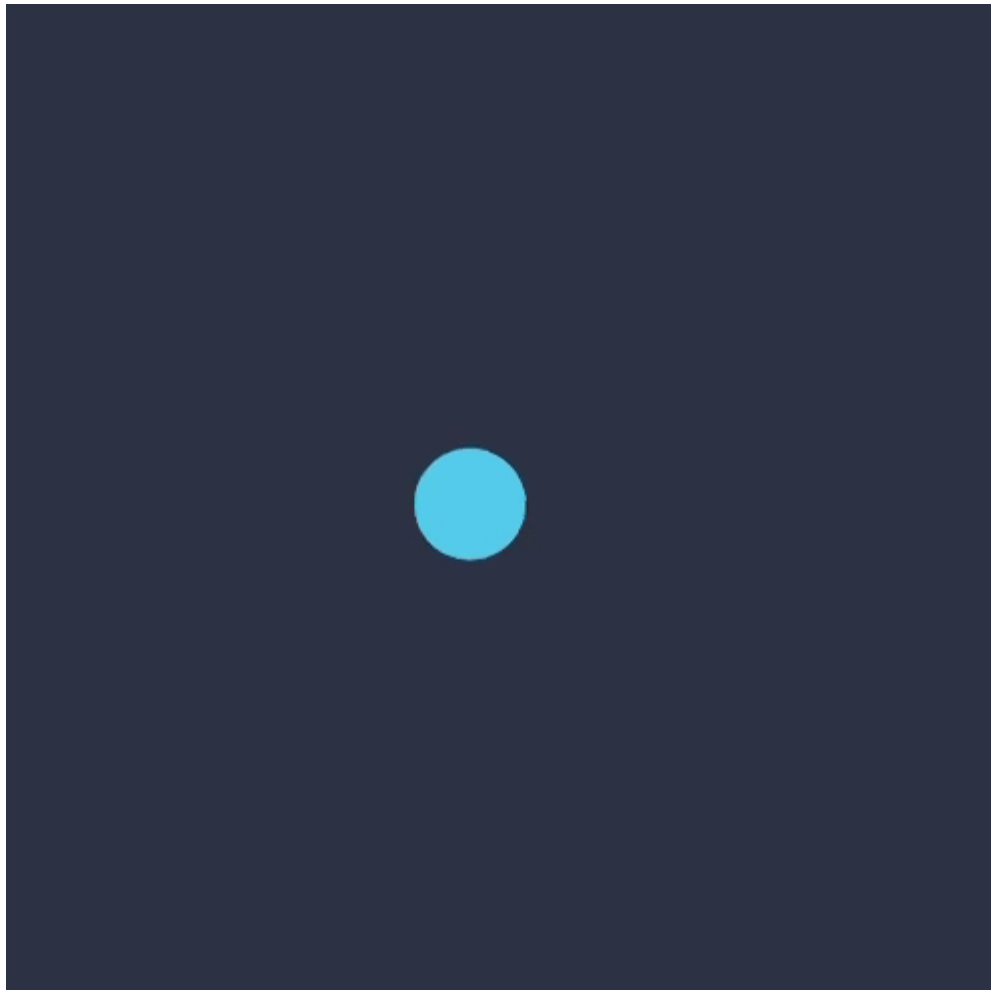
Main Idea

- Want to design an npc



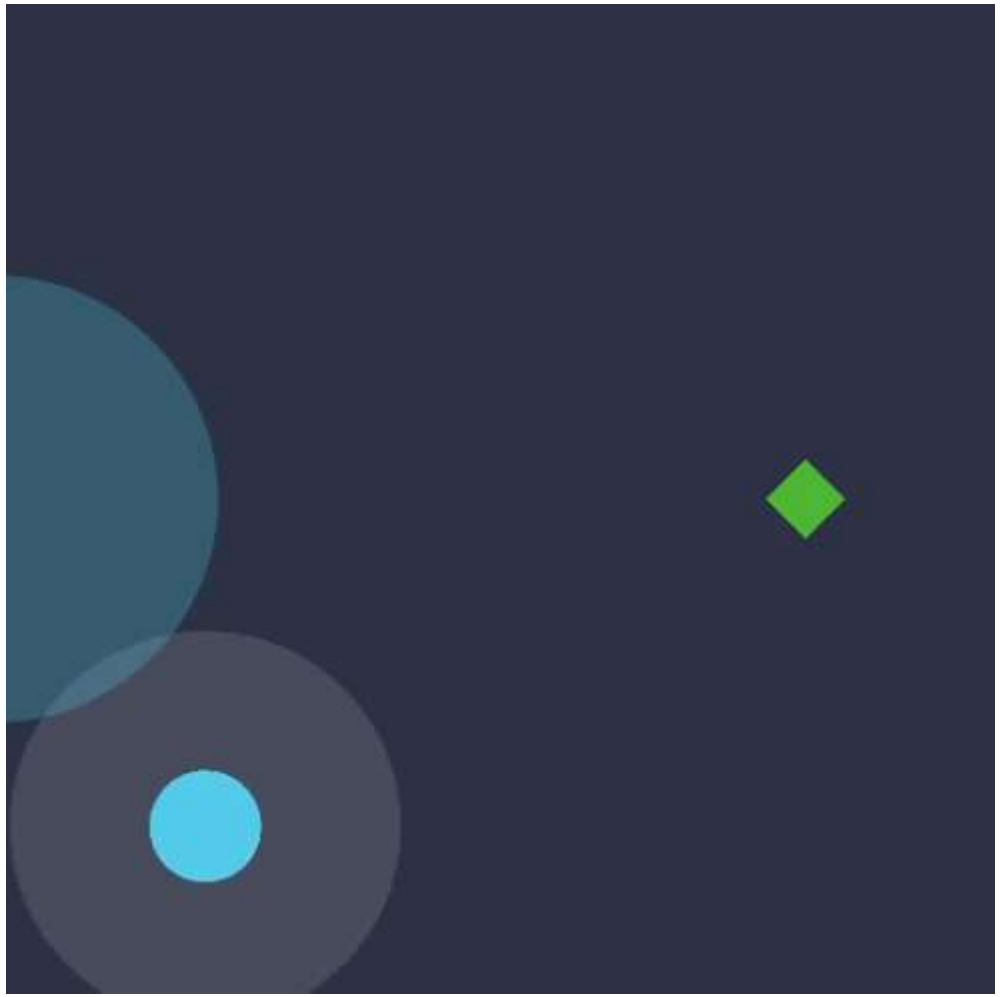
Main Idea

- Want to design an npc
  - Should wander around



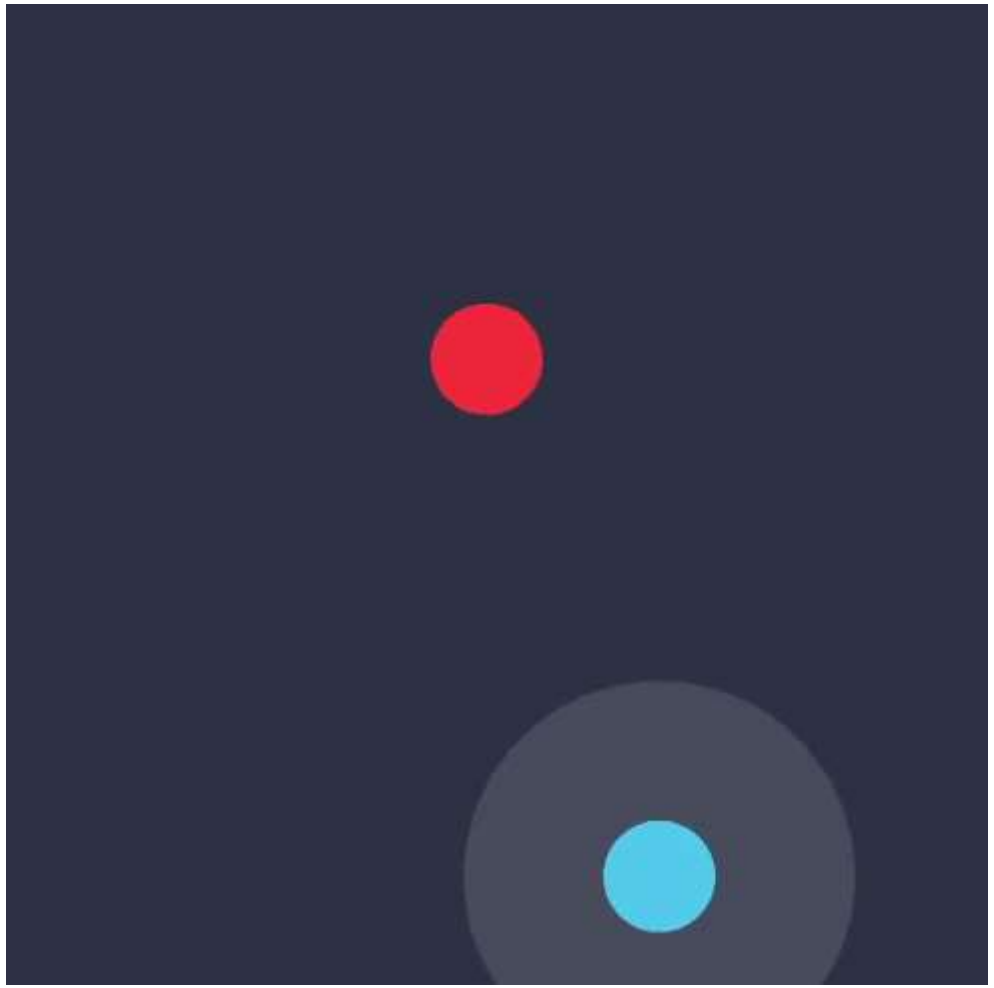
Main Idea

- Want to design an npc
  - Should wander around
  - Should collect point when in range



Main Idea

- Want to design an npc
  - Should wander around
  - Should collect point when in range
  - Should attack enemy NPC when in range
  - NPC should be able to die



Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Constructing The State Pattern

- What is the context? (whose states are being maintained?)

Constructing The State Pattern

- What is the context? (whose states are being maintained?)
  - NPC is context!

Constructing The State Pattern

- What is the context? (whose states are being maintained?)
  - NPC is context!
- What are possible states for the context?

Constructing The State Pattern

- What is the context? (whose states are being maintained?)
  - NPC is context!
- What are possible states for the context?
  - Wandering, Collecting, Attacking, Death

Constructing The State Pattern

- What is the context? (whose states are being maintained?)
  - NPC is context!
- What are possible states for the context?
  - Wandering, Collecting, Attacking, Death
- What are possible actions by the context?

Constructing The State Pattern

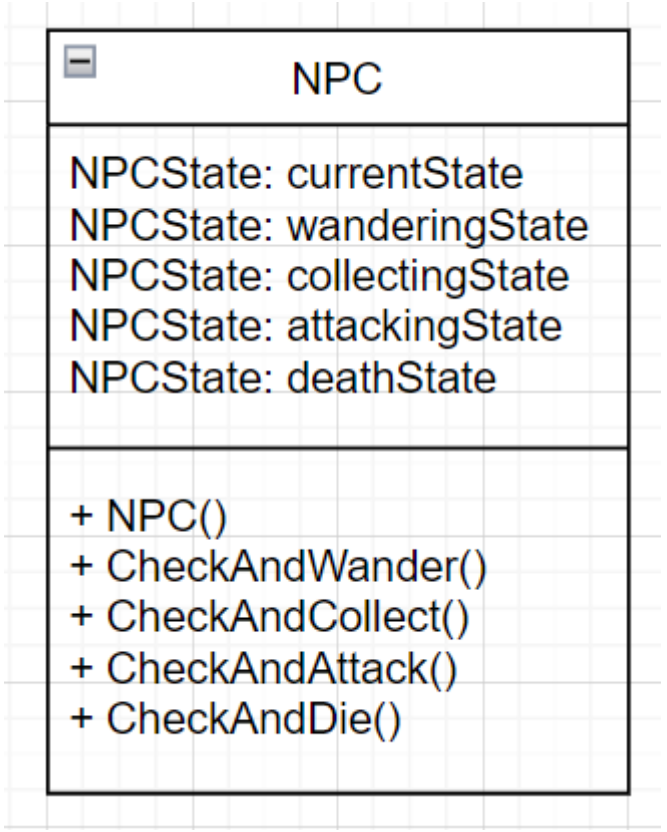
- What is the context? (whose states are being maintained?)
  - NPC is context!
- What are possible states for the context?
  - Wandering, Collecting, Attacking, Death
- What are possible actions by the context?
  - CheckAndWander, CheckAndCollect, CheckAndAttack, CheckAndDie

Constructing The State Pattern

- Context: NPC
- Possible States:
  - Wandering State
  - Collecting State
  - Attacking State
  - Death State
- Possible Actions (by context):
  - CheckAndWander
  - CheckAndCollect
  - CheckAndAttack
  - CheckAndDie

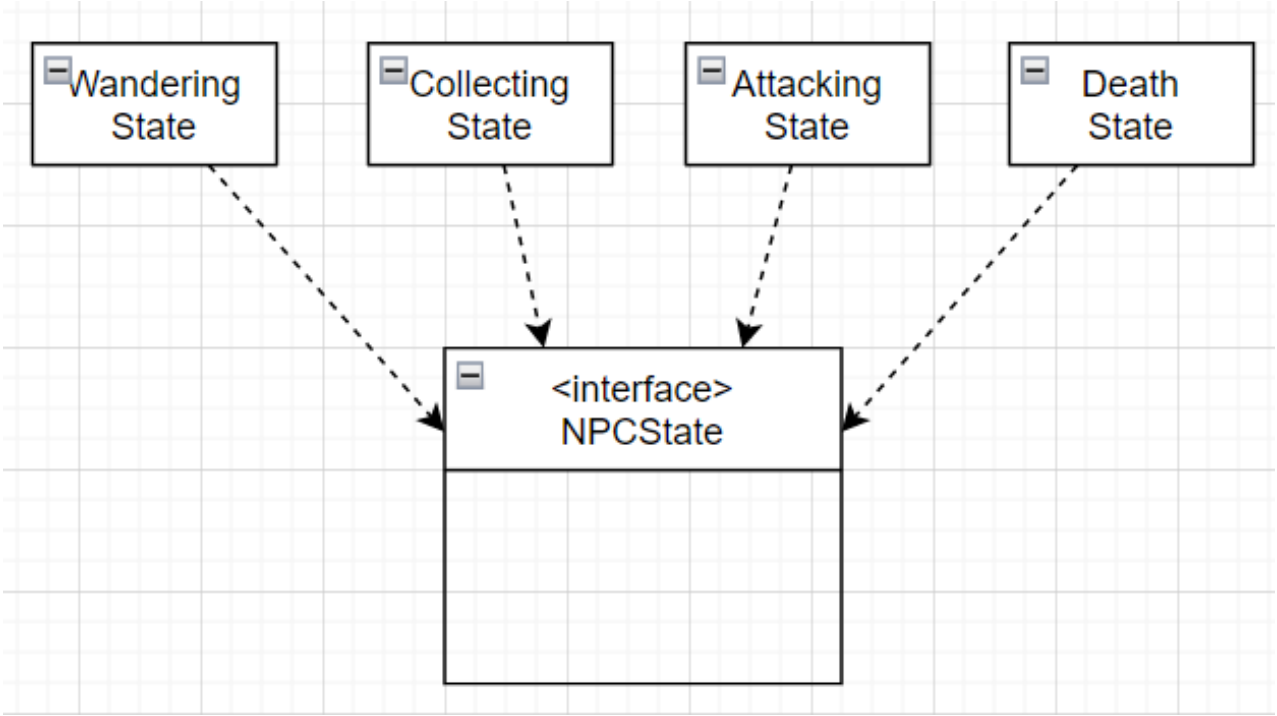
Constructing The State Pattern

- Context: NPC
- Possible States:
  - Wandering State
  - Collecting State
  - Attacking State
  - Death State
- Possible Actions (by context):
  - CheckAndWander
  - CheckAndCollect
  - CheckAndAttack
  - CheckAndDie



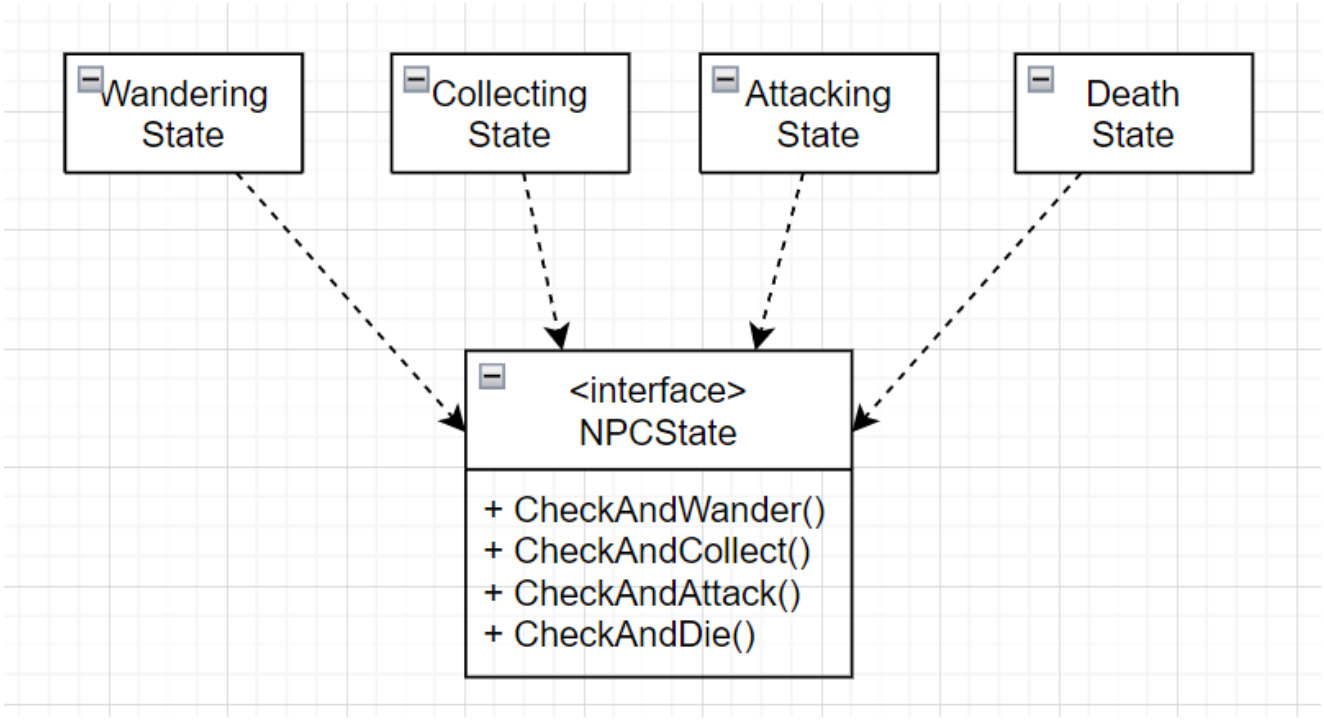
Constructing The State Pattern

- Context: NPC
- Possible States:
  - Wandering State
  - Collecting State
  - Attacking State
  - Death State



Constructing The State Pattern

- Context: NPC
- Possible States:
  - Wandering State
  - Collecting State
  - Attacking State
  - Death State
- Possible Actions (by context):
  - CheckAndWander
  - CheckAndCollect
  - CheckAndAttack
  - CheckAndDie



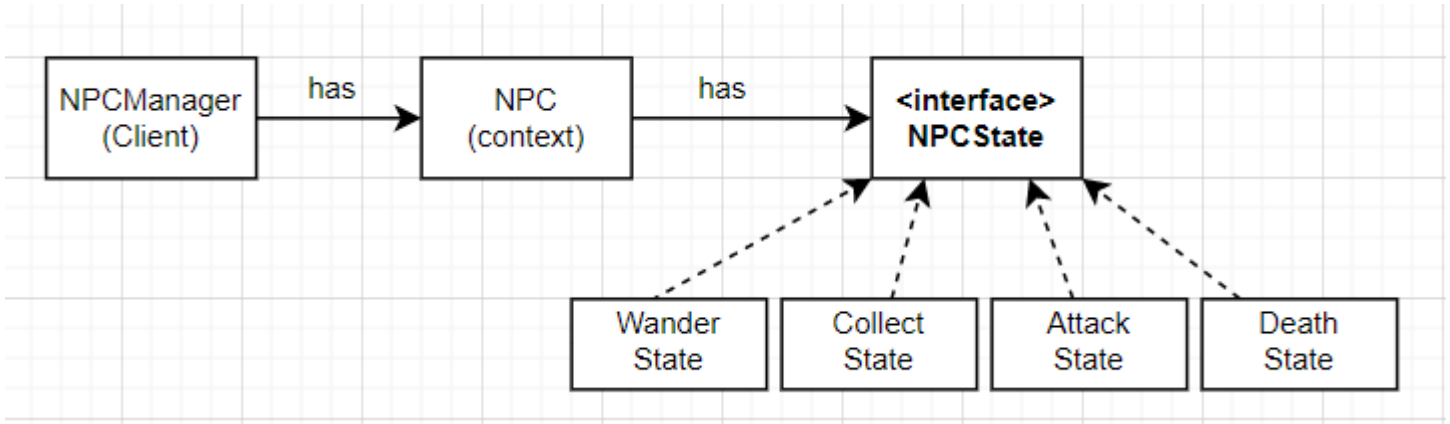
Context Action Delegations

```
NPC {
  NPCState currentState;
  NPCState wanderingState;
  NPCState collectingState;
  NPCState attackingState;
  NPCState deathState;

  NPC() {
    wanderingState = new WanderingState();
    collectingState = new CollectingState();
    attackingState = new AttackingState();
    deathState = new DeathState();

    currentState = this.wanderState;
  }
```

```
  void CheckAndWander() {
    currentState.CheckAndWander();
  }

  void CheckAndCollect() {
    currentState.CheckAndCollect();
  }

  void CheckAndAttack() {
    currentState.CheckAndAttack();
  }

  void CheckAndDie() {
    currentState.CheckAndDie();
  }
}
```

NPC manager and calls to context

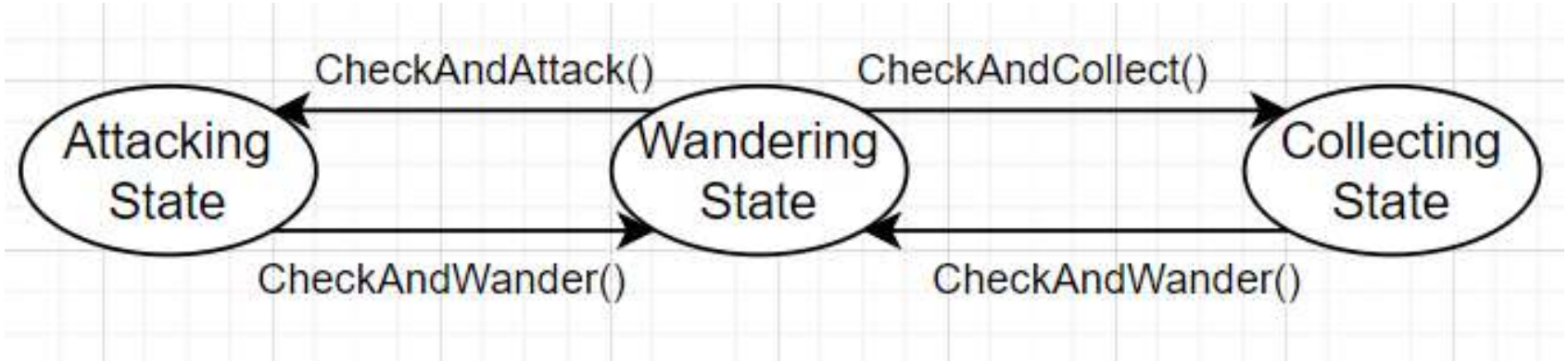
```
class NPCManager() {
  NPC npc;

  void Update() {
    npc.CheckAndDie();
    npc.CheckAndWander();
    npc.CheckAndCollect();
    npc.CheckAndAttack();
  }
}
```



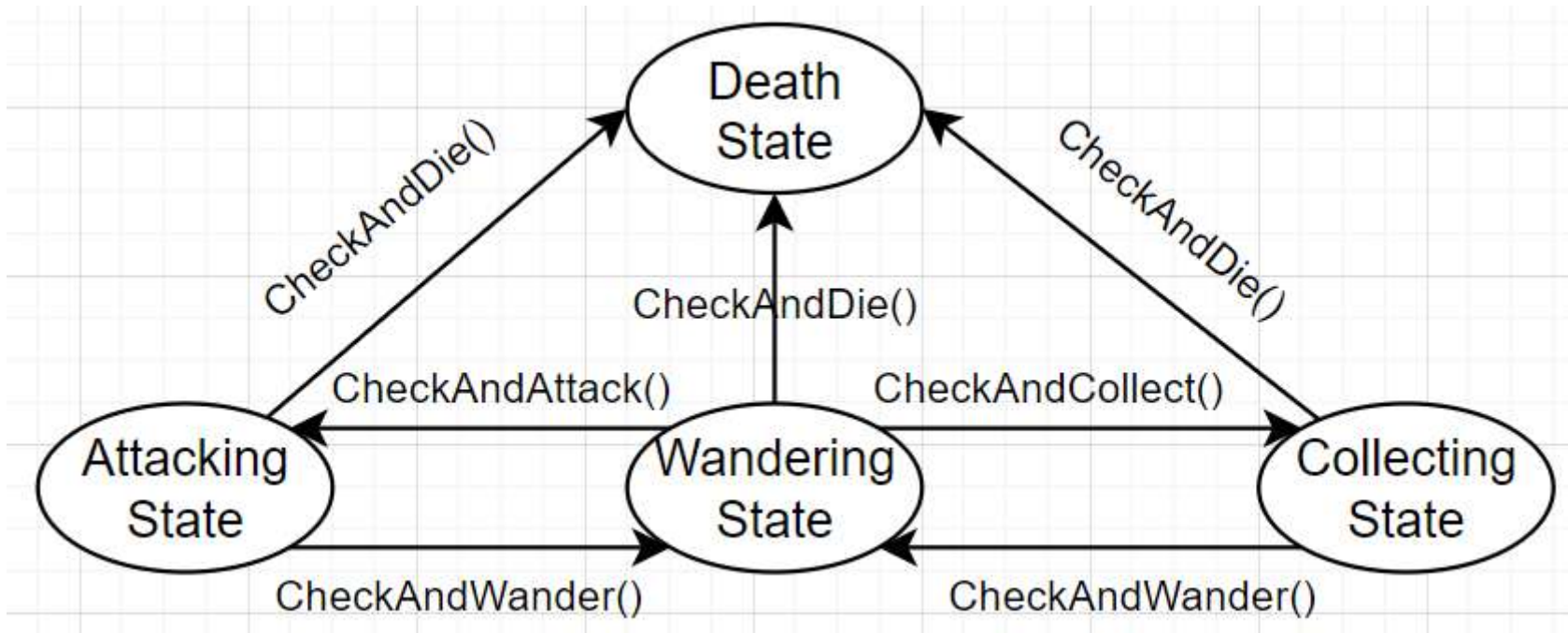
Transitions For The NPC

State Diagram

- Can’t switch between attack and collect
  - Just our choice! Could be allowed…

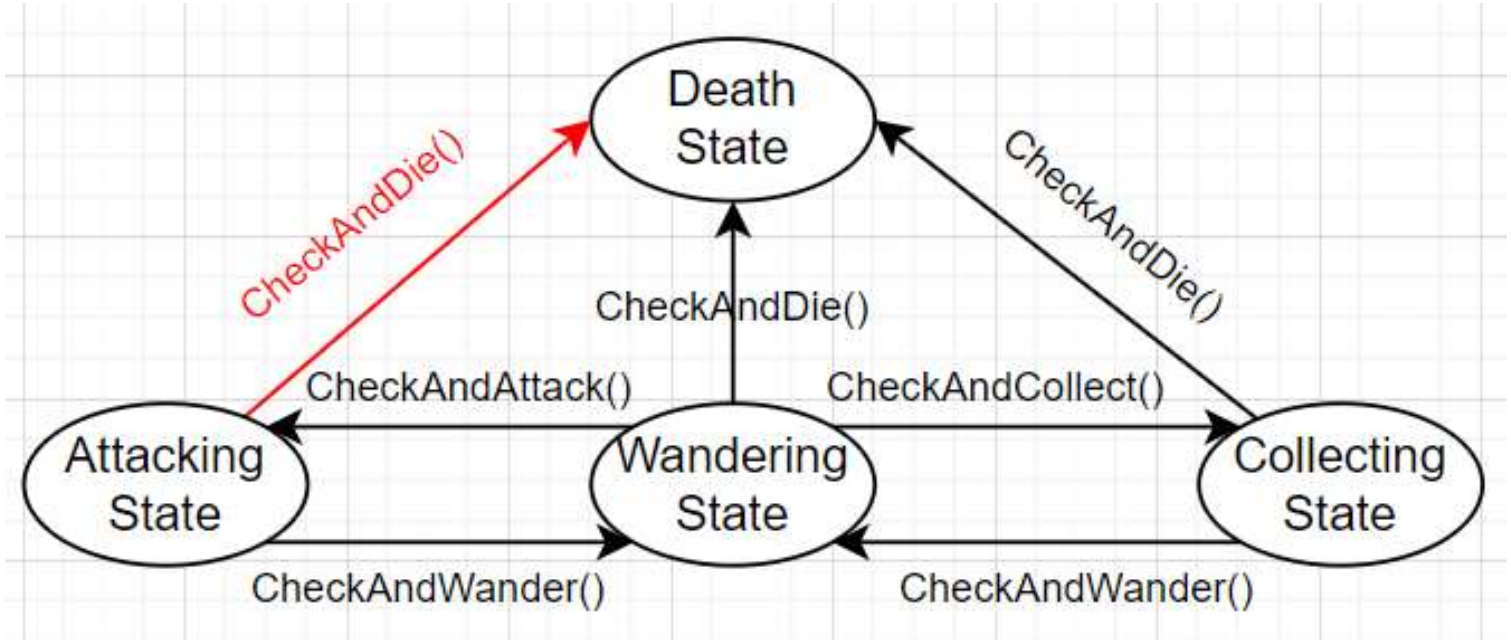


State Diagram



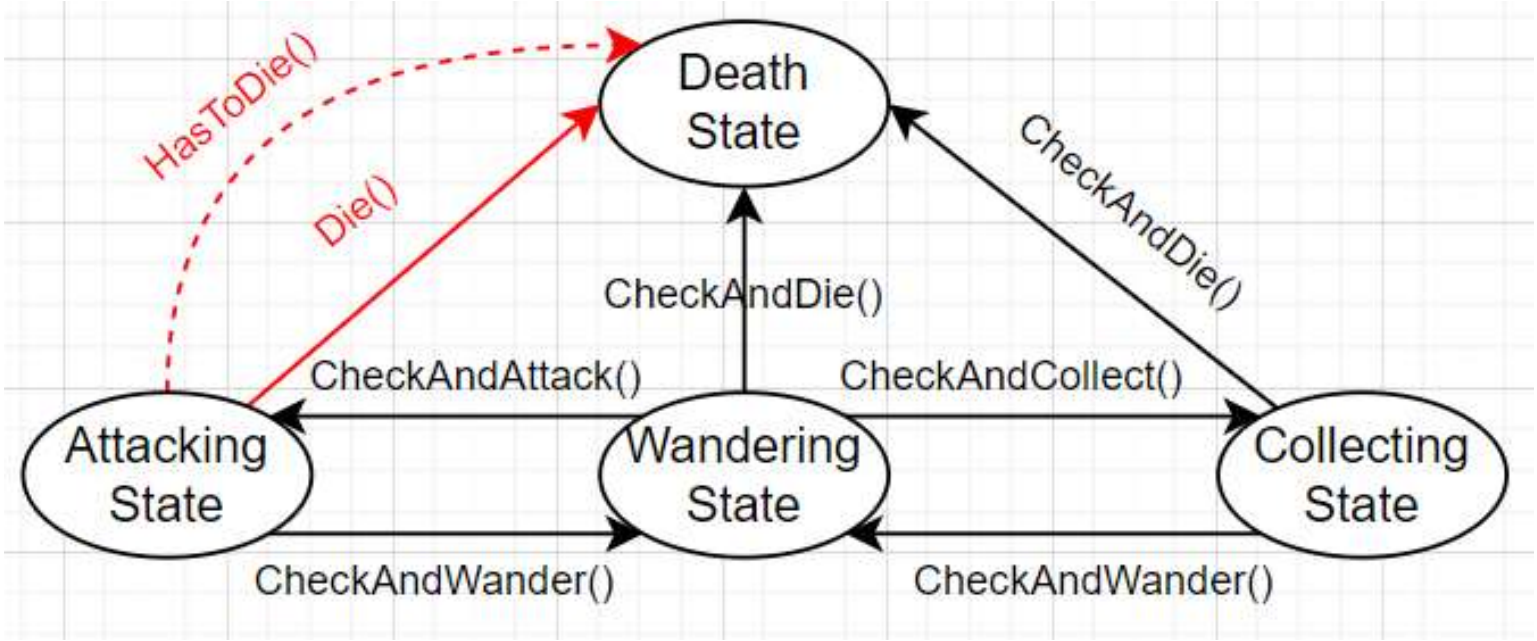
There is a problem…

- Violates single responsibility principle!



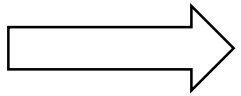
Solution

- Violates single responsibility principle!
  - Seperate to different functions



Update NPCState interface

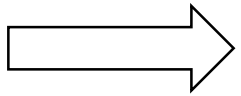
```
interface NPCState {
  void CheckAndWander();
  void CheckAndCollect();
  void CheckAndAttack();
  void CheckAndDie();
}
```



```
interface NPCState {
  bool HasToWander();
  void Wander();


  bool HasToCollect();
  void Collect()

  void HasToAttack();
  void Attack();

  void HasToDie();
  void Die();
}
```

Update NPCManager class

```
class NPCManager() {
  NPC npc;

  void Update() {
    npc.CheckAndDie();
    npc.CheckAndWander();
    npc.CheckAndCollect();
    npc.CheckAndAttack();
  }
}
```



```
class NPCManager() {
  NPC npc;

  void Update() {
    if (npc.HasToDie()) {
      npc.Die();
    }

    else if (npc.HasToWander()) {
      npc.Wander();
    }

    else if (npc.HasToCollect()) {
      npc.Collect();
    }

    else if (npc.HasToAttack()) {
      npc.Attack();
    }
  }
}
```

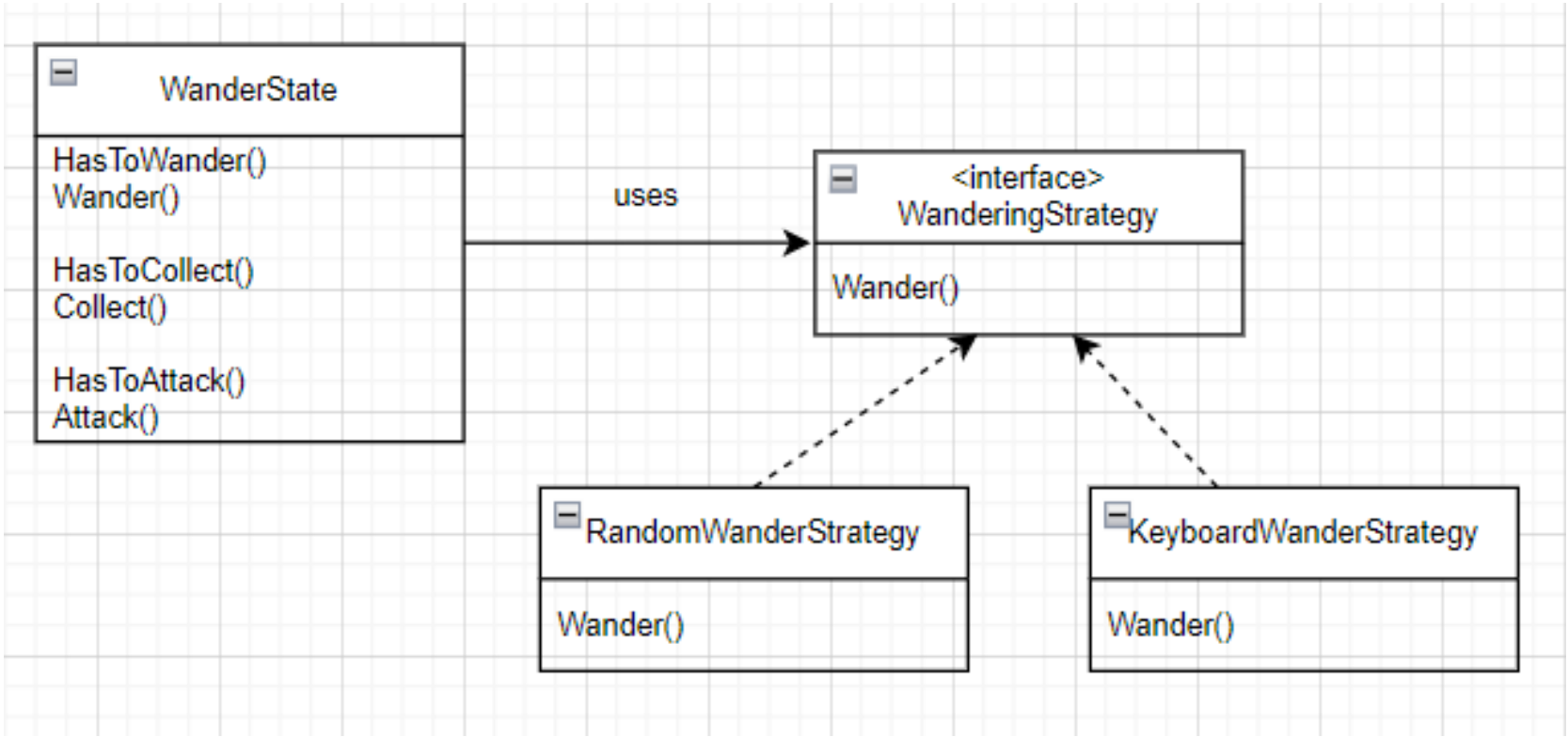
Finally, implementations

```
class WanderingState implements NPCState {
  // some internal state variables
  NPC npc;

  WanderingState(NPC npc) {this.npc = npc;}

  bool HasToWander() {...}
  void Wander() {...}

  bool HasToCollect() {...}
  void Collect() {...}

  bool HasToAttack() {...}
  void Attack() {...}

  bool HasToDie() {...}
  void Die() {...}
}
```

```
class CollectingState implements NPCState {
  // some internal state variables
  NPC npc;

  CollectingState(NPC npc) {this.npc = npc;}

  bool HasToWander() {...}
  void Wander() {...}

  bool HasToCollect() {...}
  void Collect() {...}

  bool HasToAttack() {...}
  void Attack() {...}

  bool HasToDie() {...}
  void Die() {...}
}
```

Where does the strategy pattern fit in?

Where does the strategy pattern fit in?



Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Alternative Implementations (Testing)

- If else implementation

```
class NPC {
  void Wander() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Collect() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
```

```
  void Attack() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Die() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
}
```

Alternative Implementations (Testing)

- If else implementation
  - Complicated code, hard to maintain

```
class NPC {
  void Wander() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Collect() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
```

```
  void Attack() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Die() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
}
```

Alternative Implementations (Testing)

- If else implementation
  - Complicated code, hard to maintain
  - Violates single responsibility

```
class NPC {
  void Wander() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Collect() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
```

```
  void Attack() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Die() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
}
```

Alternative Implementations (Testing)

- If else implementation
  - Complicated code, hard to maintain
  - Violates single responsibility and dependency inversion

```
class NPC {
  void Wander() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Collect() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
```

```
  void Attack() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }

  void Die() {
    if (currentState == "wander state") {...}
    else if (currentState == "collect state") {...}
    else if (currentState == "attack state") {...}
    else if (currentState == "death state") {...}
  }
}
```

Alternative Implementations (Testing)

- Strategy Pattern implementation

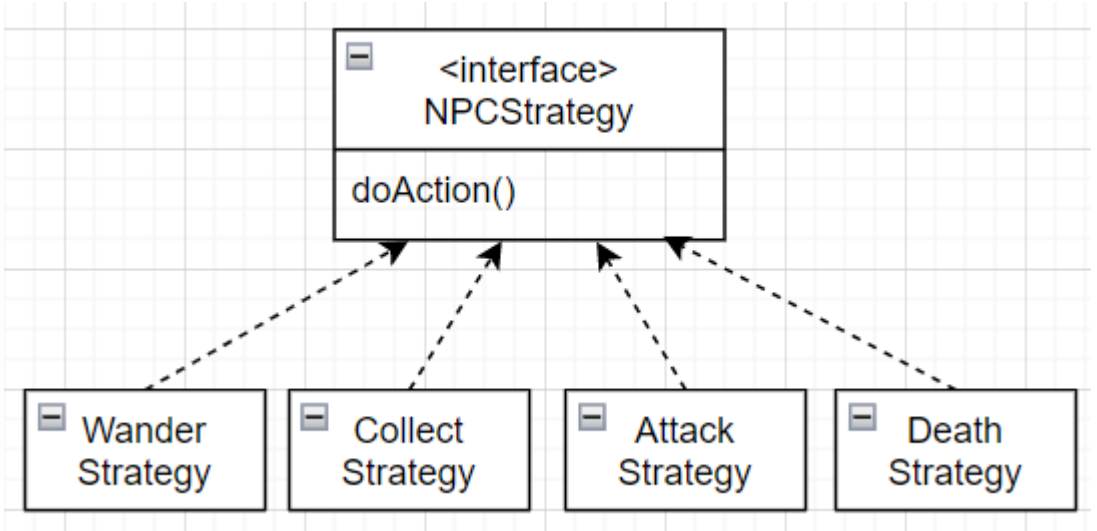
```
class NPC {
  String currentState;

  void Wander() {
    if (currentState == "wander state") {
      strategy = new WanderStrategy();
      strategy.doAction();
    }

    else if (currentState == "collect state") {
      ...// logic for switching from collect state --> wander state

    else if (currentState == "attack state") {
      ...// logic for switching from attack state --> wander state
    }

   ...// other states

  }
```

```
  void Collect() {
    if (currentState == "wander state") {
      ...// logic for switching from wander state --> collect state
    }

    else if (currentState == "collect state") {
      strategy = new CollectStrategy();
      strategy.doAction();
    }

    else if (currentState == "attack state") {
      ...// logic for switching from attack state --> collect state
    }

  ...// other states

  }
}
```

Alternative Implementations (Testing)

- Strategy Pattern implementation
  - Code is less complicated to if-else

```
class NPC {
  String currentState;

  void Wander() {
    if (currentState == "wander state") {
      strategy = new WanderStrategy();
      strategy.doAction();
    }
```



Alternative Implementations (Testing)

- Strategy Pattern implementation
  - Code is less complicated to if-else
  - Strategies are unaware of each other → Context still has to manage states

```
class NPC {
  String currentState;

  void Wander() {
    if (currentState == "wander state") {
      strategy = new WanderStrategy();
      strategy.doAction();
    }

    else if (currentState == "collect state") {
      ...// logic for switching from collect state --> wander state

    else if (currentState == "attack state") {
      ...// logic for switching from attack state --> wander state
    }

   ...// other states

  }
```

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Amazon Leadership Principles

- What are the Amazon Leadership Principles?
  - Set of principles imposed by amazon on their employees.
    - 16 principles in total, we look at 4

- Customer Obsession
- Ownership
- Think Big
- Deliver Results

Customer Obsession

Customer Obsession

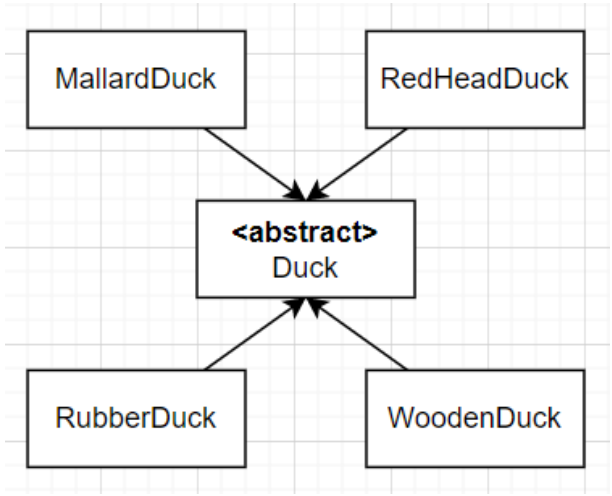
- “You have to start with the customer experience and work backwards towards the technology.”

Steve Jobs

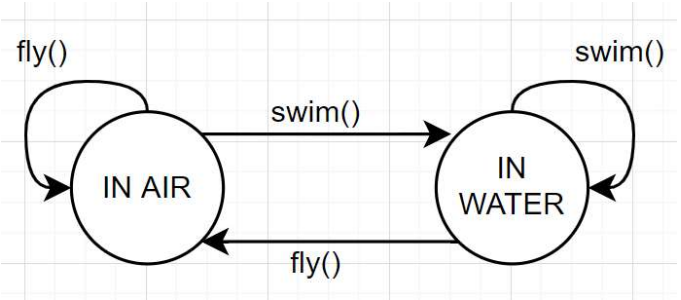
Customer Obsession

- Start with customer and work backwards, obsess over customers
  - Customer is the class!

Customer Obsession

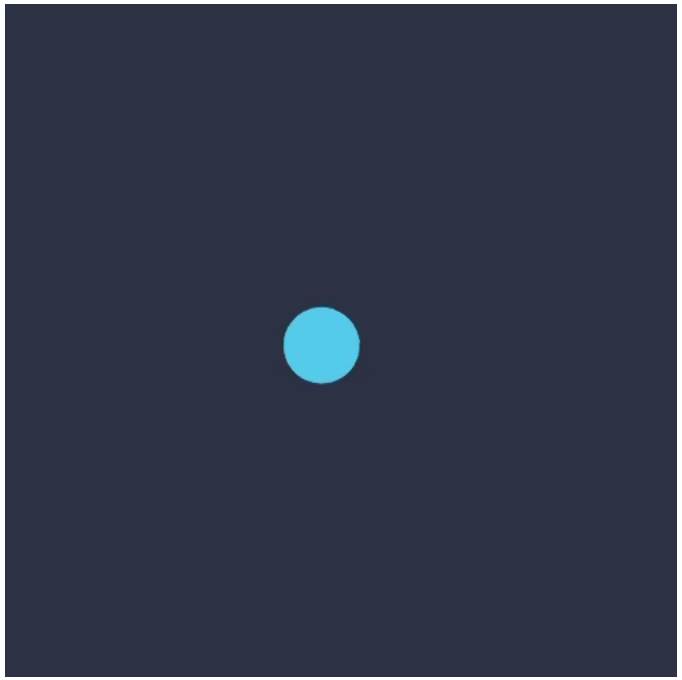


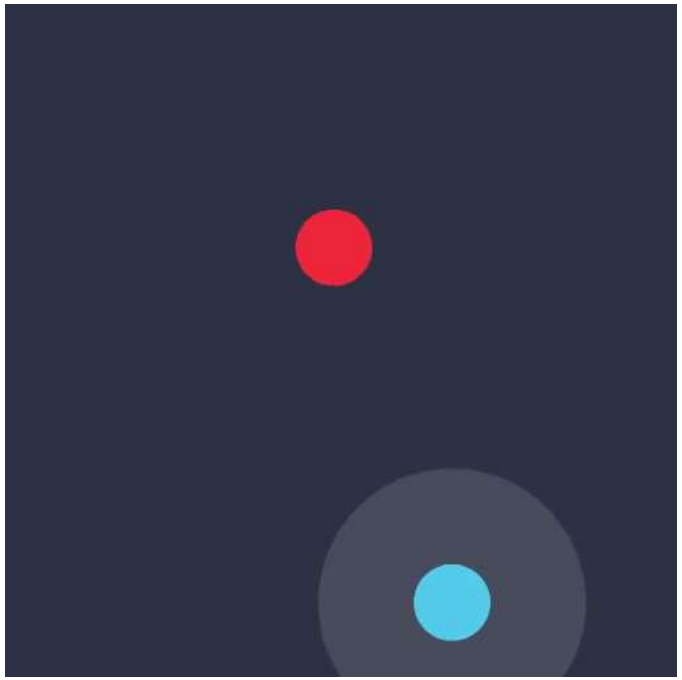

```
Duck duck = new SomeDuck();

duck.fly();
duck.swim();
duck.quack();
```

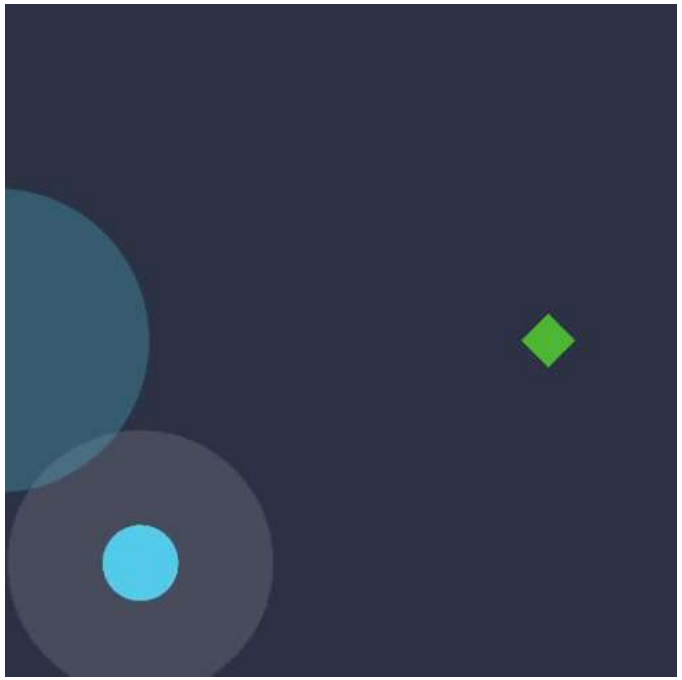


Customer Obsession

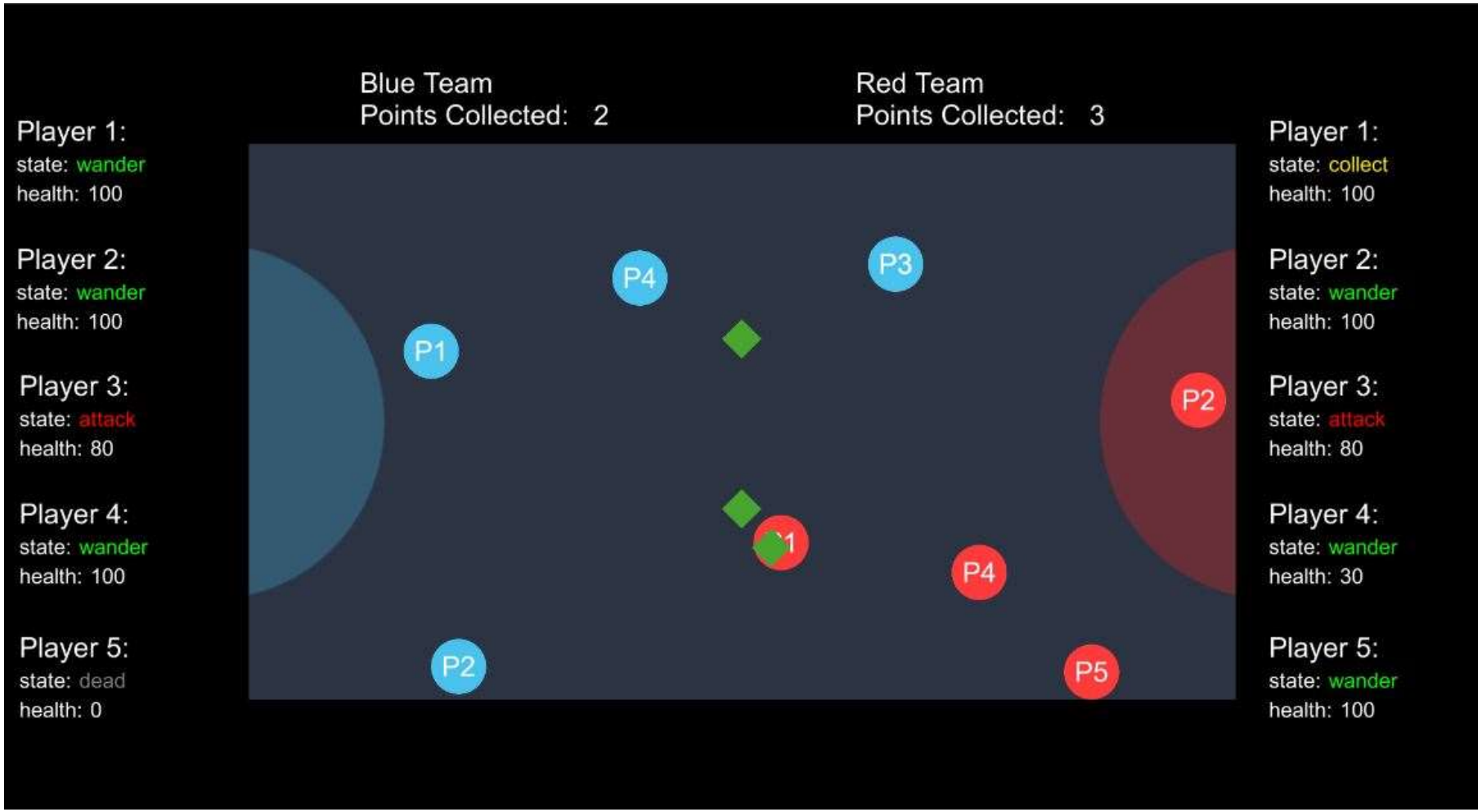
Customer Obsession







Customer Obsession



Ownership

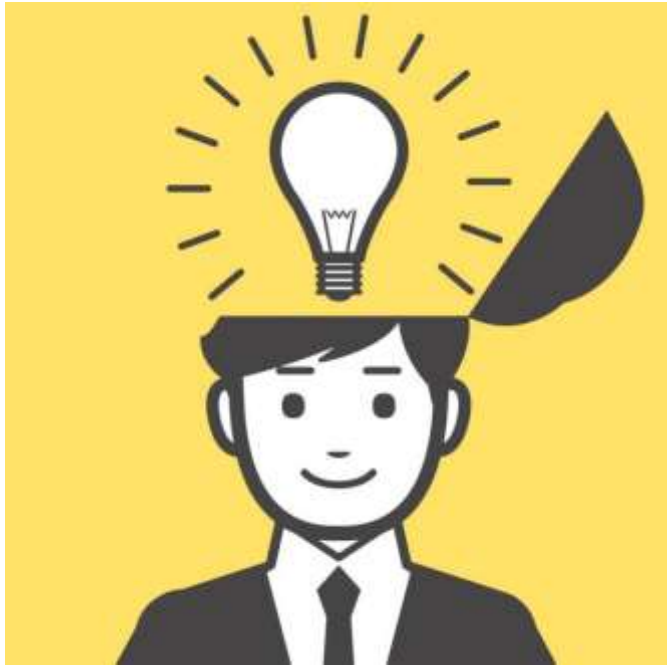
Ownership

- Act on behalf of the team, never say "not my job!"

Think Big

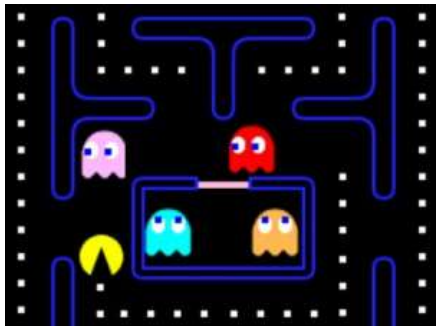
Think Big

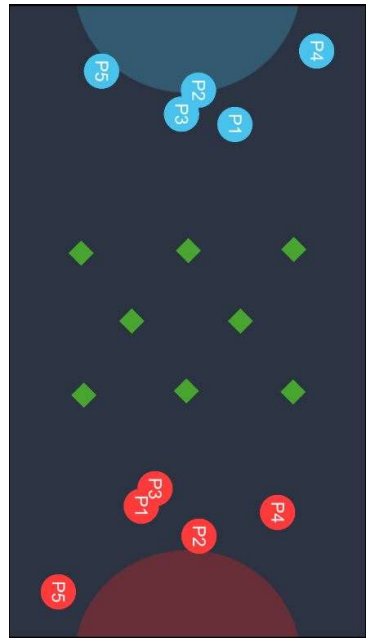
- Think of new possible ways to serve customers and act boldly



Think Big

- Think of new possible ways to serve customers
  - Instead of conceptual example, we provided a real world use case





Deliver Results

Deliver Results

- Deliver results in a timely fashion
- Despite setbacks, never settle

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

TCE Analysis

- Initially there were lot of ambiguity in the project
  - What are the states, actions and transition for NPCs?

TCE Analysis

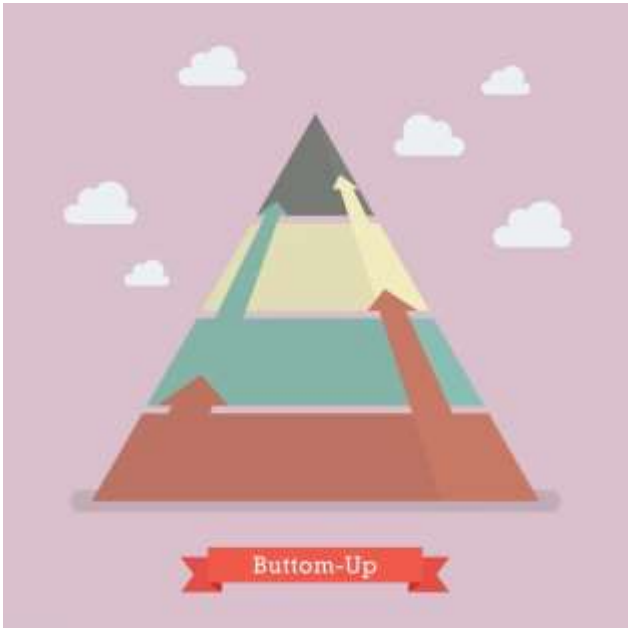
- Initially there were lot of ambiguity in the project
  - What are the states, actions and transition?

- Needed the **work together with high communication**

TCE Analysis

- Initially there were lot of ambiguity in the project
  - What are the states, actions and transition?

- Needed the **work together with high communication**
  - Bottom up governance structure
    - Everyone contributed to several modules



TCE Analysis

- As time went on project’s structure became more clear and it was easier to **decompose the work to be done into submodules**

TCE Analysis

- As time went on project's structure became more clear and it was easier to **decompose the work to be done into submodules**
  - Also, midterm's were starting → Everyone is busy with their own work, **opportunistic behaviour may arise**

TCE Analysis

- As time went on project's structure became more clear and it was easier to **decompose the work to be done into submodules**
  - Also, midterm's were starting… → Everyone is busy with their own work, **opportunistic behaviour may arise**
  - Switch to top down governance structure
    - Everyone became more focused on their own state implementation

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Presentation Agenda

STATE PATTERN THEORY	OUR APPLICATION	DEVELOPMENT PROCESS
▪ Strategy Pattern ▪ State Pattern ▪ SOLID Analysis ▪ Advantages & Disadvantages ▪ Possible Use Cases	▪ Concept of NPC ▪ Main Idea ▪ State Pattern ▪ Alternative Implementations (Testing)	▪ Amazon Leadership Principles ▪ TCE Analysis ▪ Team Work

Teamwork

- Discord and Whatsapp for communication.
- Drive and Github for share content
- At the beginning of the project, we develop together
- Project became more clear, we assigned well-defined tasks to everyone.
- We held weekly general situation assessment meetings.

The End.