State

State

- behavioral design pattern that lets an object alter its behavior when its internal state changes
- a lot of applications have object that realizes some "logic states" (aka Finite State Machine)
- "orders", "machines", "buttons", car (engine), invoice, employee - we are living in a stateful world

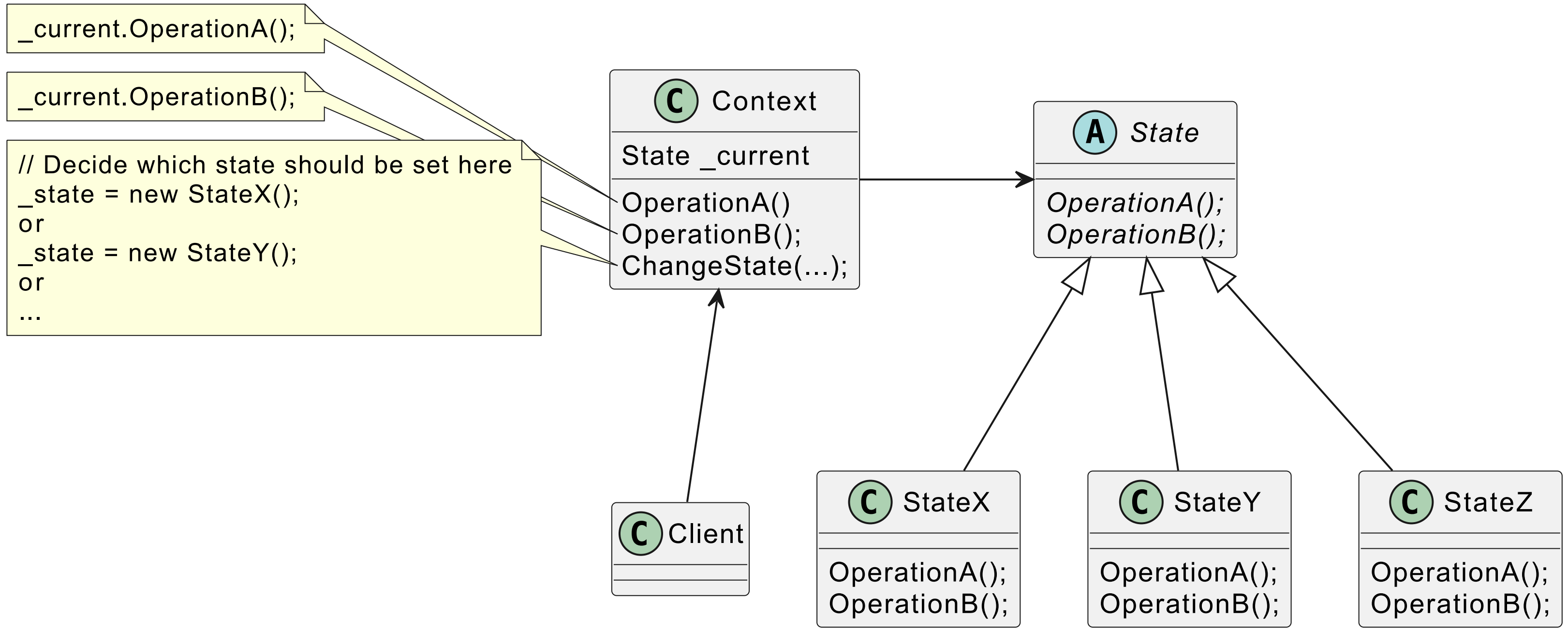
```
public void InsertCard()
{
   switch (_currentState)
   {
      case MACHINE_STATE.INITIAL:
         _currentState = MACHINE_STATE.CARD_INSERTED;
         break;
      case MACHINE_STATE.CARD_INSERTED:
      case MACHINE_STATE.PIN_ENTERED:
      case MACHINE_STATE.CASH_WITHDRAWN:
         throw new InvalidOperationException("Card already inserted");
      default:
         throw new ArgumentOutOfRangeException();
   }
}
public void EnterPin(Pin pin)
{
   switch (_currentState)
   {
      case MACHINE_STATE.INITIAL:
         throw new InvalidOperationException("No card inserted");
      case MACHINE_STATE.CARD_INSERTED:
         if (pin != 1234) throw new InvalidOperationException("incorect pin");
         _currentState = MACHINE_STATE.PIN_ENTERED;
         break;
      case MACHINE_STATE.PIN_ENTERED:
      case MACHINE_STATE.CASH_WITHDRAWN:
         throw new InvalidOperationException("Pin already entered");
      default:
         throw new ArgumentOutOfRangeException();
   }
}
```

State

- this pattern **delegates** state handling to specific *State* types
- kind of "Strategy pattern" where "strategies" (states) know about each other and create transitions between them

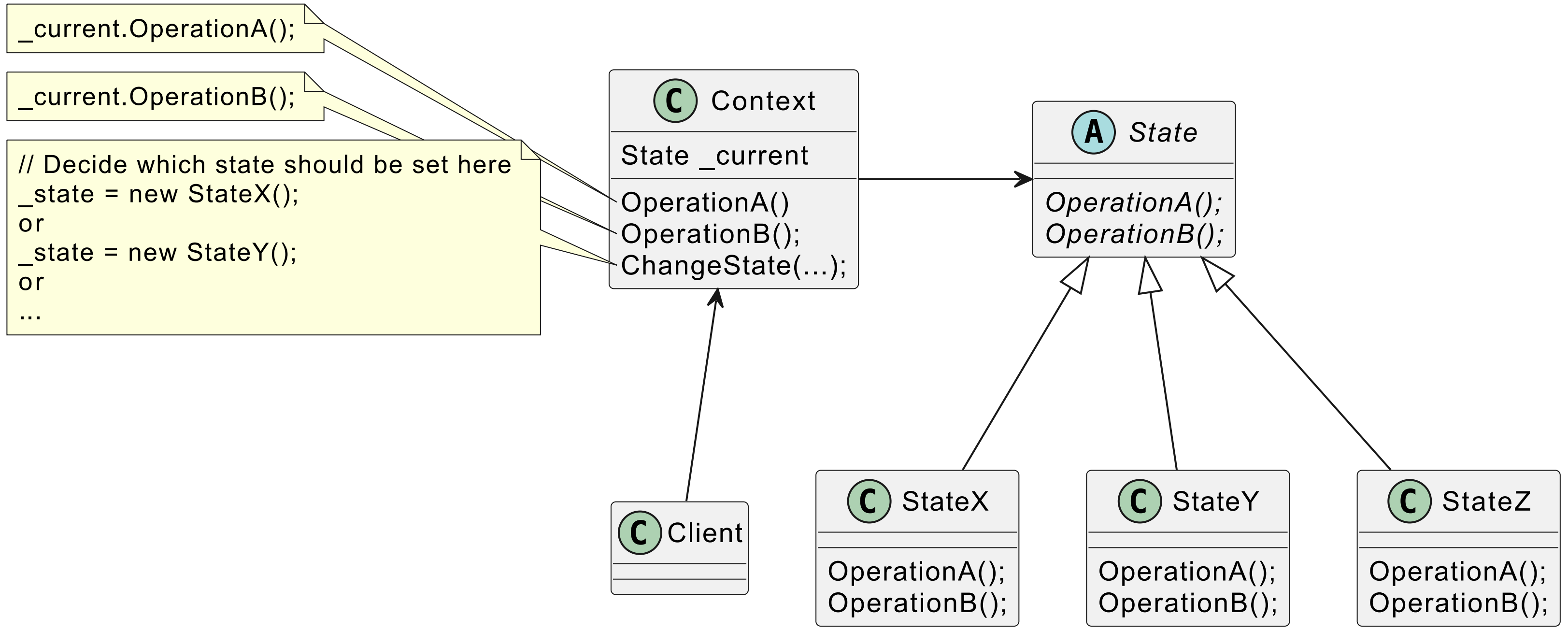
State

- this pattern **delegates** state handling to specific *State* types
- kind of "Strategy pattern" where "strategies" (states) know about each other and create transitions between them



State

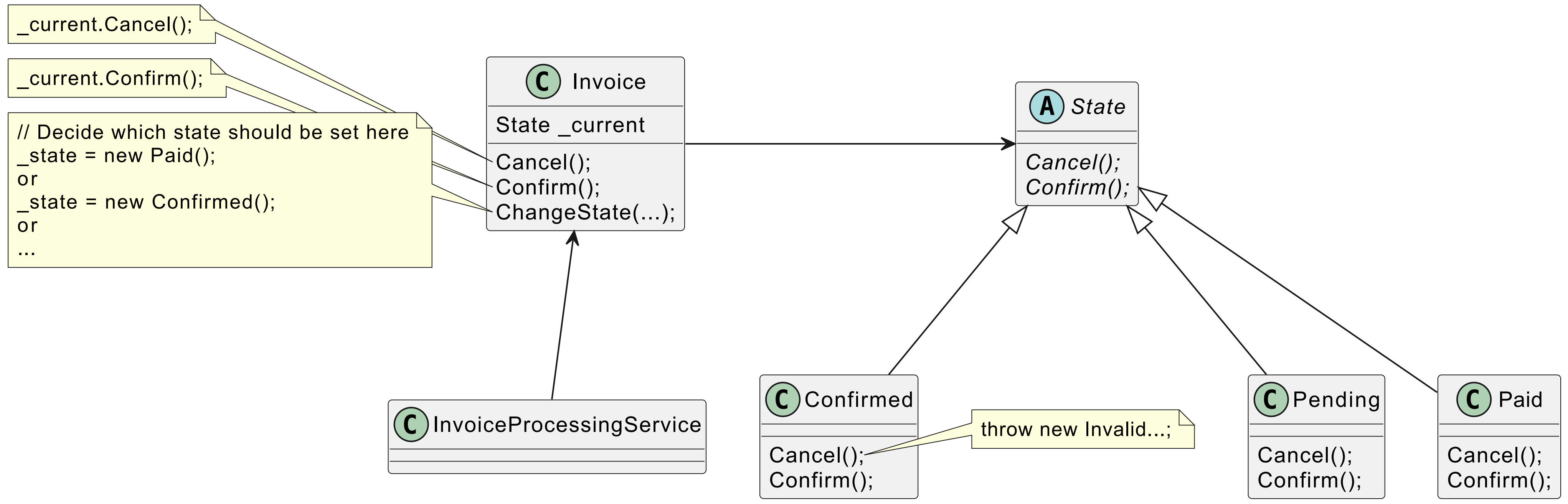
- this pattern **delegates** state handling to specific *State* types
- kind of "Strategy pattern" where "strategies" (states) know about each other and create transitions between them



- context - interface for clients, facade for operations, holds state
- who changes "state" - context or specific state?
- states created *ad-hoc* or at once (or maybe from a pool?).

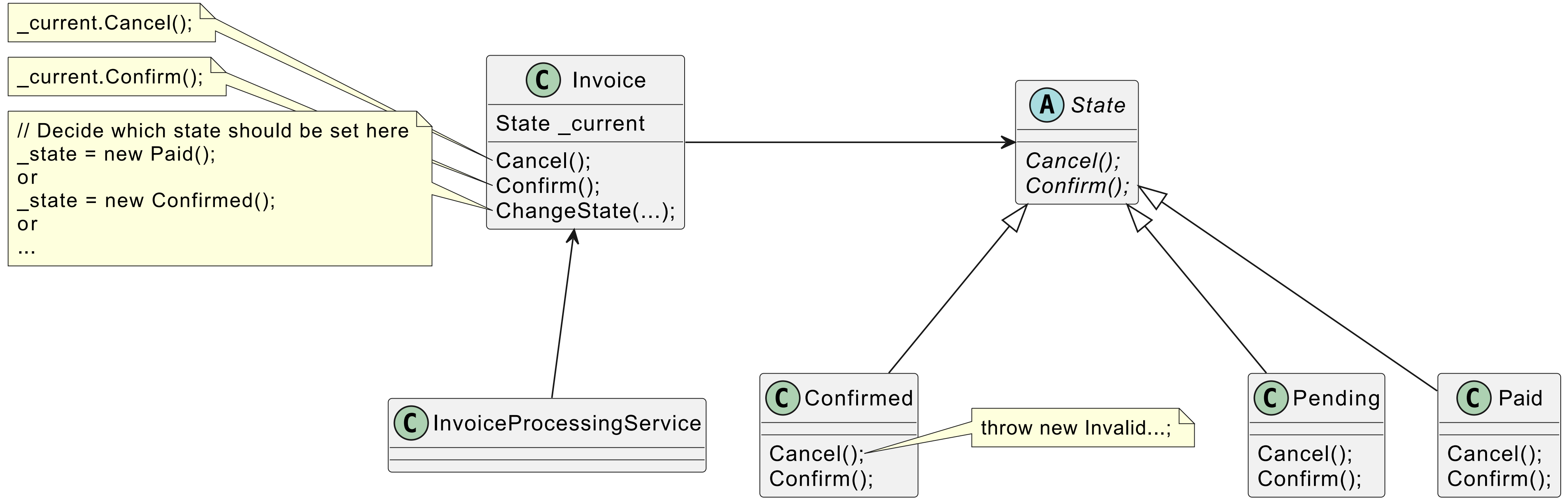
State

- this pattern **delegates** state handling to specific *State* types
- kind of "Strategy pattern" where "strategies" (states) know about each other and create transitions between them



State

- this pattern **delegates** state handling to specific *State* types
- kind of "Strategy pattern" where "strategies" (states) know about each other and create transitions between them



- context (**`Invoice`**) - interface for client (**`InvoiceProcessingService`**), facade for operations, holds state
- who changes "state" - probably **`InvoiceProcessingService`**

State - who changes "state"

- transitions rather fixed and unchanging - in context.
- transitions rather flexible - in specific states

State

- use when:
  - you have an object that behaves differently depending on its current state
  - the number of states is **big**
  - and/or the state-specific code **changes** frequently

State - implementation details

State - implementation details

- what is *Context* in your code? That may be the easiest part.

State - implementation details

- what is *Context* in your code? That may be the easiest part.
- what's *State interface*? Like in *Strategy*, it may be not trivial
    - what operations are "state-specific"?
    - how much we will violate "L"? 👀

State - implementation details

- what is *Context* in your code? That may be the easiest part.
- what's *State interface*? Like in *Strategy*, it may be not trivial
  - what operations are "state-specific"?
  - how much we will violate "L"? 👀
- how *State*s access *Context* data? Again, similar to *Strategy pattern*, we can:
  - make such data/method public 😬
  - nest the state classes in the context class (or at least its base class) 👀

State - implementation details

- what is *Context* in your code? That may be the easiest part.
- what's *State interface*? Like in *Strategy*, it may be not trivial
  - what operations are "state-specific"?
  - how much we will violate "L"? 👀
- how *State*s access *Context* data? Again, similar to *Strategy pattern*, we can:
  - make such data/method public 😬
  - nest the state classes in the context class (or at least its base class) 👀
- who implements **`changeState`** logic/validation?
  - "centralized" version - our big swtich
  - "decentralized" version - *concrete state* objects:

Przykłady:

- _Intro - smutny switch

Przykłady:

- _Intro - smutny switch
- PreMain - wydzielony stan

Przykłady:

- _Intro - smutny switch
- PreMain - wydzielony stan
- Main - lepsza enkapsulacja (nested base state)

Przykłady:

- _Intro - smutny switch
- PreMain - wydzielony stan
- Main - lepsza enkapsulacja (nested base state)
- Decentralized - stany samodzielnie określają przejścia (dodana odpowiednia operacja w kontekście - `bookState` property)

Przykłady:

- _Intro - smutny switch
- PreMain - wydzielony stan
- Main - lepsza enkapsulacja (nested base state)
- Decentralized - stany samodzielnie określają przejścia (dodana odpowiednia operacja w kontekście - **`bookState`** property)

Ogólnie: stany mogą być statyczne/singletony (jeśli nie mają danych) i tylko operować na kontekście jako argumencie