State Pattern

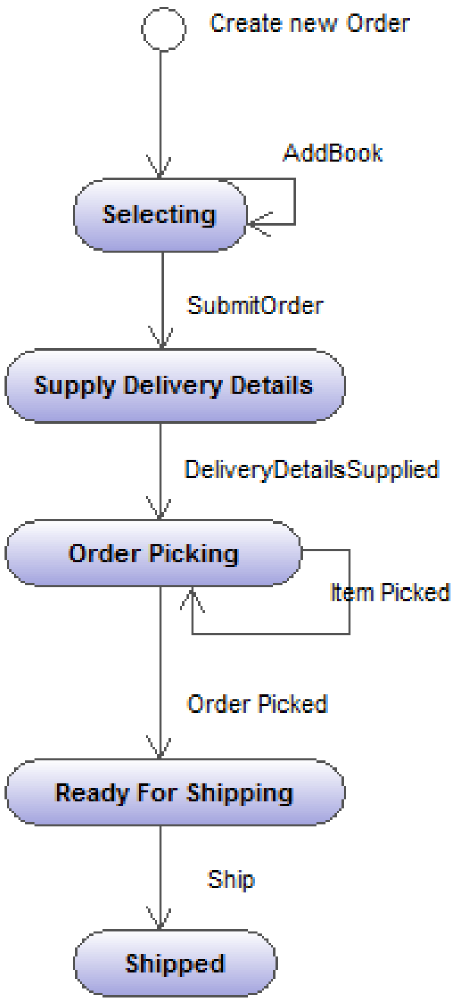
The need for state awareness

- Objects can behave differently over time
  - TCP Connection Object
    - Allows Open only when not connected
    - Allows Read/Write only when connected
    - Allows Close only when connected
  - Vending Machine
    - Select Item
      - If sufficient credit then vend item
      - Else display amount required
  - Smart Client
    - If connected, get up to date information
    - Else use local cache

Online book store

- ACME Corp wish to have an online book store
  - An Order type is created to represent an order as it passes through the business process
  - The ordering process goes through a series of steps triggered by events
    - Select books
    - Set delivery details
    - Items are picked
    - Order is shipped
  - The Order type is responsible for ensuring the business process is followed.

Order Process, State Machine



Initial implementation thoughts

- Need to ensure operations can only be called in appropriate state

```
public class SimpleOrder  {
  enum OrderStates {
     SELECTING, SUPPLYING_DELIVERY_INFO,
     BEING_PICKED, ALL_PICKED, SHIPPED
  };
  private OrderStates state = OrderStates.SELECTING;

  public void AddBook(string book) {
    if (state == OrderStates.SELECTING) {
        books.Add(book);
        Console.WriteLine("{0} added to order", book);
    }
    else
        throw new InvalidOperationException("AddBook");
   }
   // More operations..
```

ACME Corp. innovates

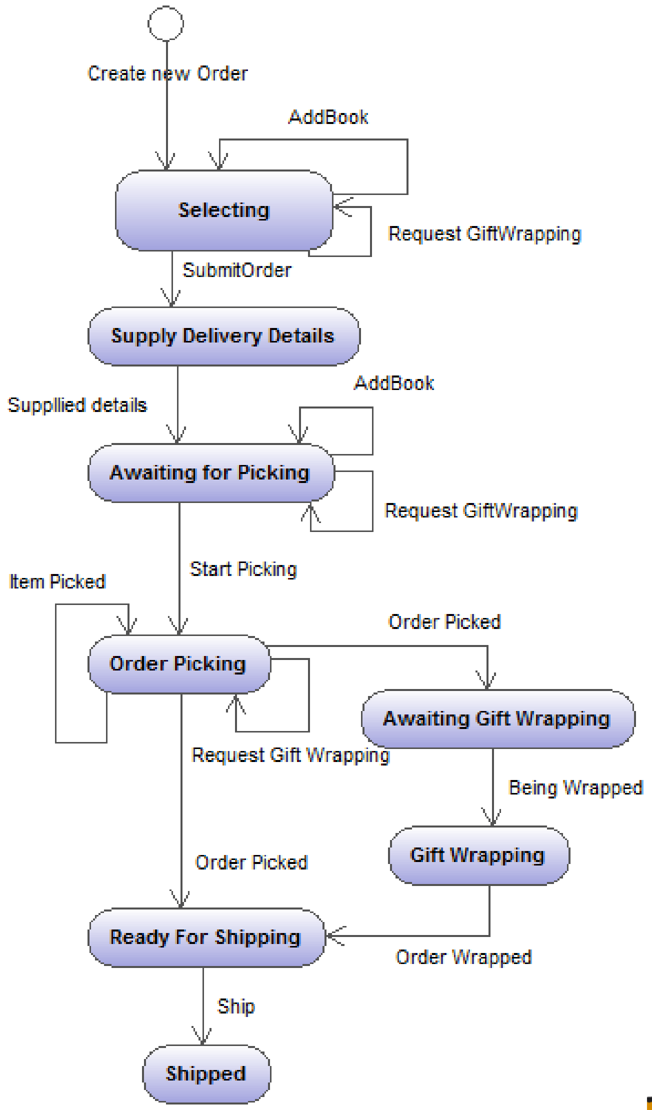
- The CEO has seen what other book companies offer
  - Add to existing order up to the point the order is actually picked
  - Gift wrapping service, selectable any time prior to shipping

Ordering process enhanced

- New States
  - Awaiting Picking
  - Awaiting Gift Wrapping
  - Gift Wrapping

•New Operations
  - Request Gift Wrapping
  - Start Picking
  - Being Wrapped
  - Order Wrapped

- A little more involved...



Dealing with the enhancements

- The AddBook Operation now needs to know about additional states
- WARNING...We are modifying existing code that has been working

```
public void AddBook(string book) {
 if ((State == OrderStates.SELECTING) ||
     (State == OrderStates.WAITING_FOR_PICKING )) {
         books.Add(book);
 }
 else {
   throw new InvalidOperationException();
 }
}
```

Messy code

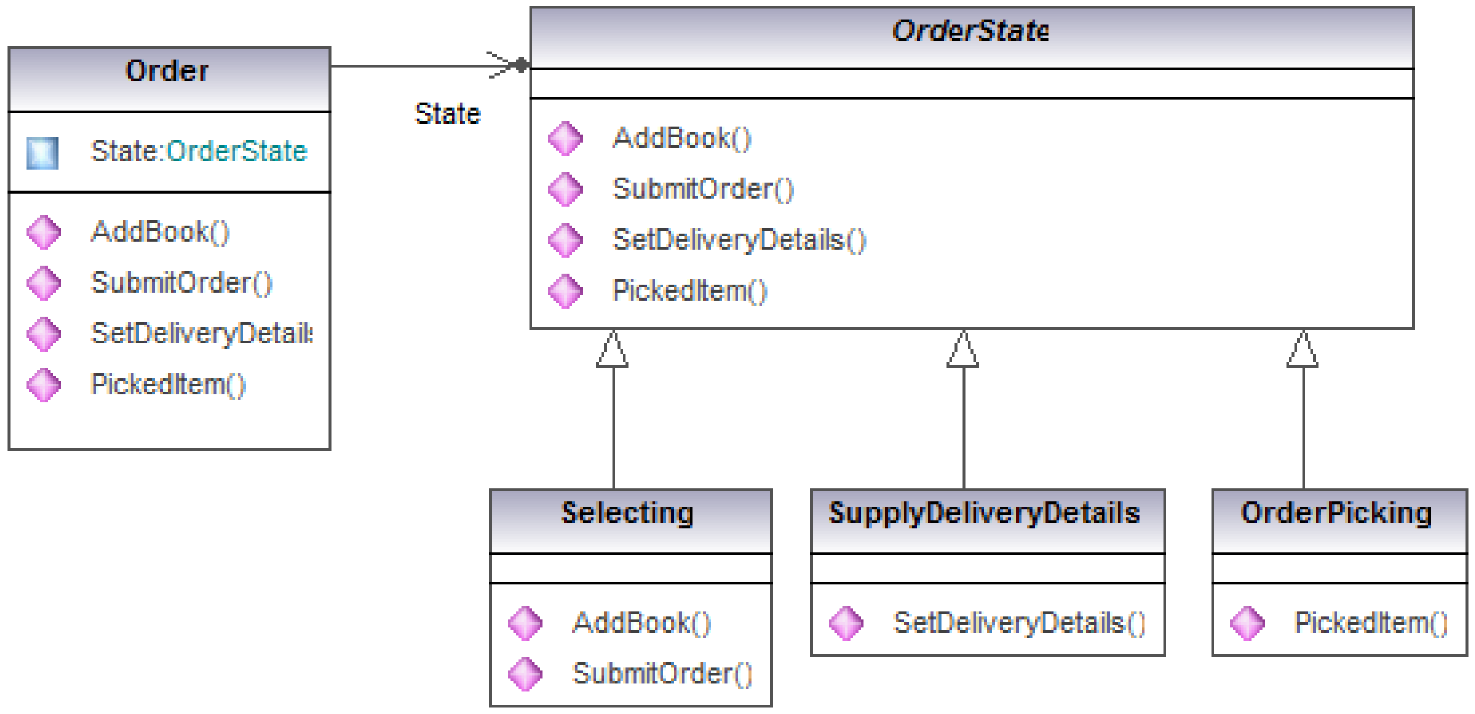
- Operation validation is now becoming more complex
  - Not clear which operations are supported by a given state
- More evolved solution
  - going to lead to bugs
  - and difficult to maintain

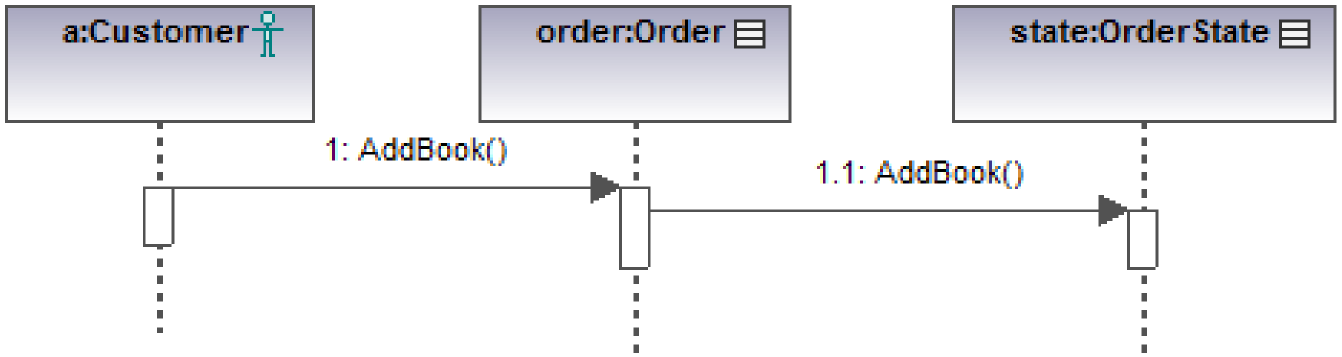
```
 public void PleaseGiftWrap() {
  if ((State == OrderStates.SELECTING) ||
      (State == OrderStates.BEING_PICKED) ||
      (State == OrderStates.WAITING_FOR_PICKING) ||
      (State == OrderStates.SUPPLYING_DELIVERY_INFO)) {
    toGiftWrap = true;
  }
 else {
   throw new InvalidOperationException();
 }
}
```

Introducing the State Pattern

- Need to re-factor to make it easy to maintain
  - Localise the behaviour of each state
    - So that changes to one state don't effect another
  - Implement each state as its own class
  - Have the Order object delegate behaviour to the current state object

Separation and Delegation of behaviour





Re-factored order class

- Order object creates instances of each of the states
- All state information kept inside the order

```
public class Order {
 private List<string> items = new List<string>;

 // Possible states of the order
 private OrderState selectingBooksState = new SelectingOrderState();
 // .... More states

 private OrderState state; // current state of the order

 public Order() {
    State = selectingBooksState;
 }

 public void AddBook(string item) {
    state.AddBook(item); // Delegates to the state object
 }
```

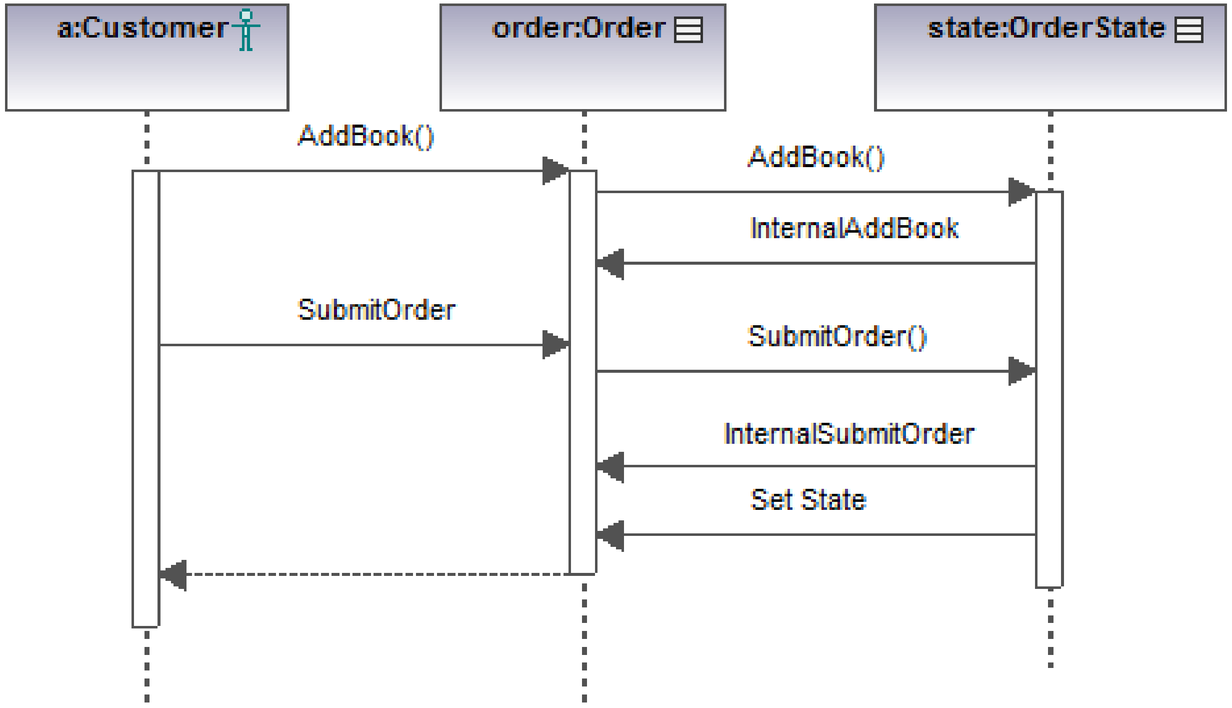
Selecting books state

- States derive from common base
- Override supported operations

```
public class SelectingOrderState : OrderState {
  public override void AddBook(string book)   {
     // Adds book item to order
  }

  public override void SubmitOrder()   {
     // Submit the order...
     // Change state, but how ?
  }

   public override void PleaseGiftWrap() {
     // Gift Wrap
   }
}
```

Updating state

- The state classes need to have the ability to update state
  - Options
    - Make the order class have additional public methods
    - Make the state types inner classes
      - Use partial class to place states into own files



Selecting books state

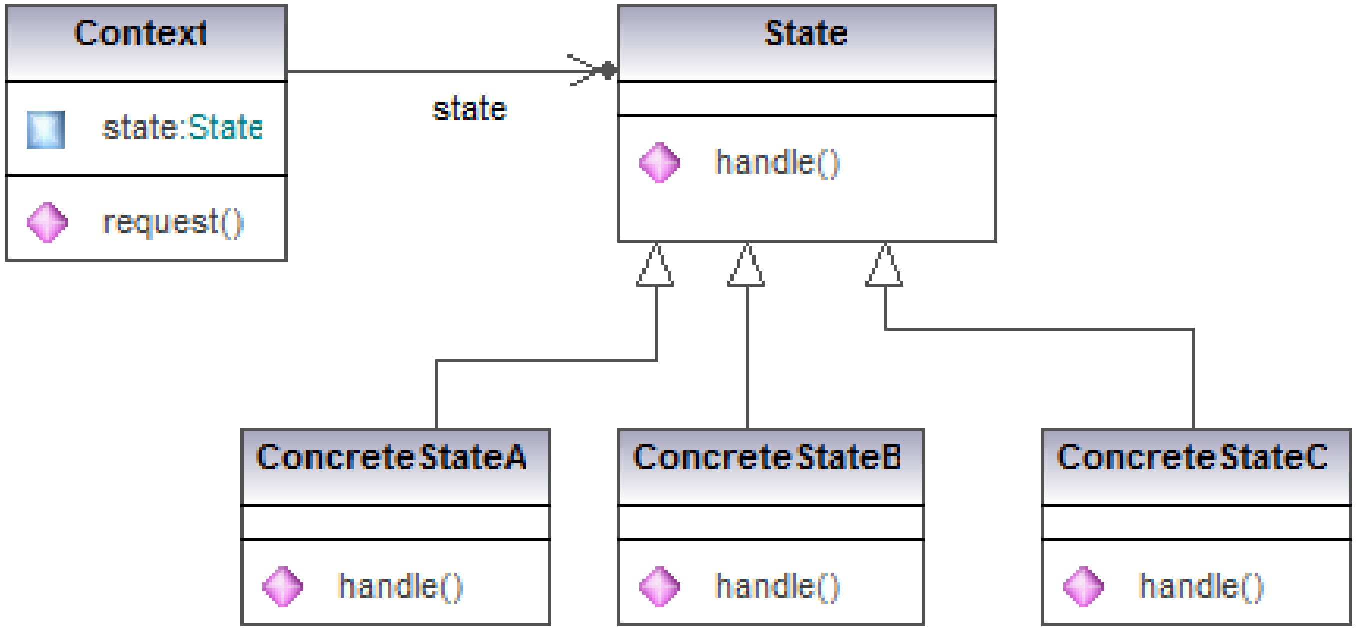
```
public partial class Order {
  public class SelectingOrderState : OrderState {
     private Order order;

     public SelectingOrderState(Order order) {
        this.order = order;
     }
     public override void AddBook(string book) {
       order.InternalAddBook(book);
     }
     public override void SubmitOrder() {
        order.InternalSubmitOrder();

        this.order.State = SetDeliveryDetailsState;
     }
     public override void PleaseGiftWrap()  {
        order.InternalPleaseGiftWrap();
     }

  }
}
```

Further Enhancements

- Creating state objects for each context can be inefficient, consider using a Singleton for each state
  - Will require instance to be passed to state for each call
- Often useful to know when a state is entered/exited
  - Add additional virtual methods to state class for this

Summary

- State Pattern
  - Removes the need for state based if/then/else logic
  - Placed a state set of behaviours in it own class
  - Allowed the addition of new states with out effecting existing working states


Context
state:State
request()
state
State
handle()
ConcreteStateA
handle()
ConcreteStateB
handle()
ConcreteStateC
handle()