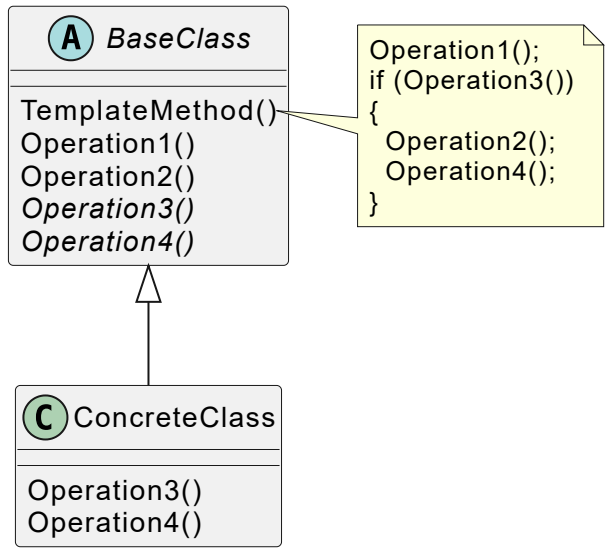
Template Method

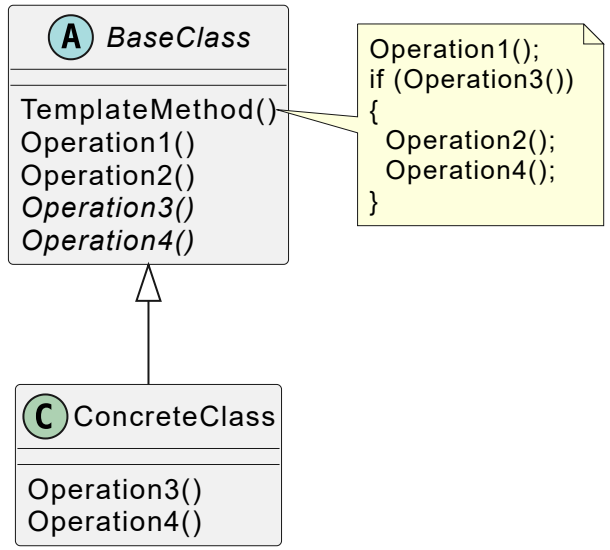
Template Method

- define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- lets subclasses redefine certain steps of an algorithm without changing the algorithm structure



Template Method

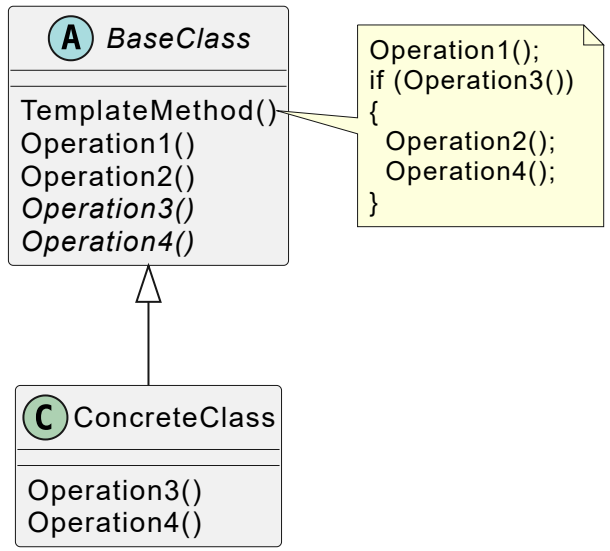
- define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- lets subclasses redefine certain steps of an algorithm without changing the algorithm structure



- not all steps needs to be virtual/abstract, but concrete in abstract class

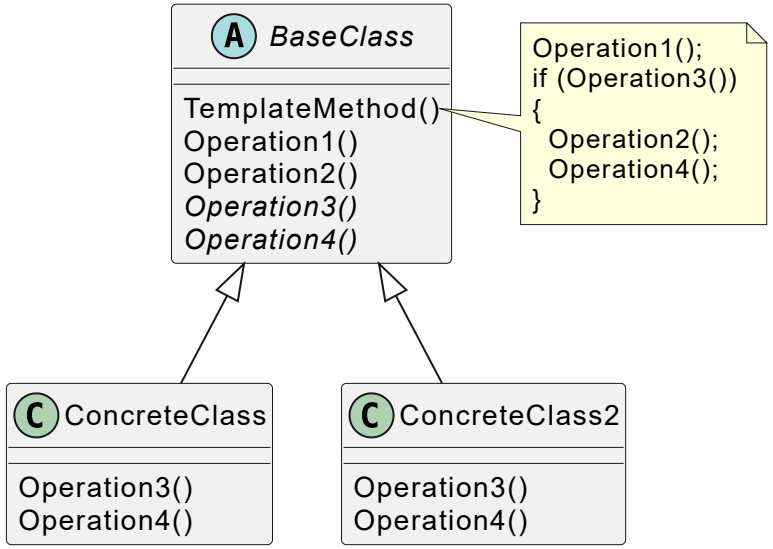
Template Method

- define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- lets subclasses redefine certain steps of an algorithm without changing the algorithm structure



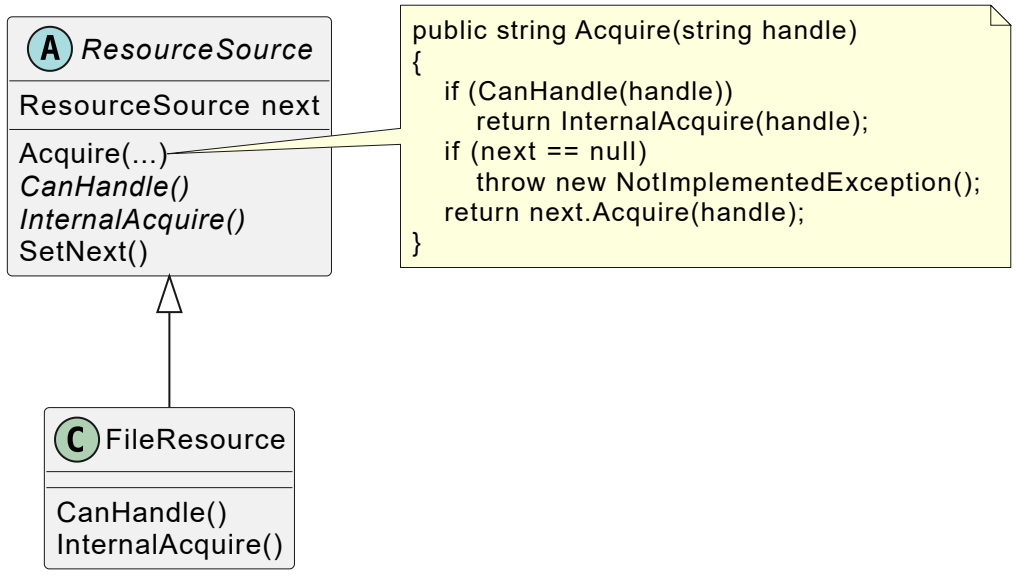
- not all steps needs to be virtual/abstract, but concrete in abstract class
- two types of operations in abstract class:
  - primitive/abstract operations - the ones that abstract could not define
  - *hook* operations (aka *callbacks*)

Template Method

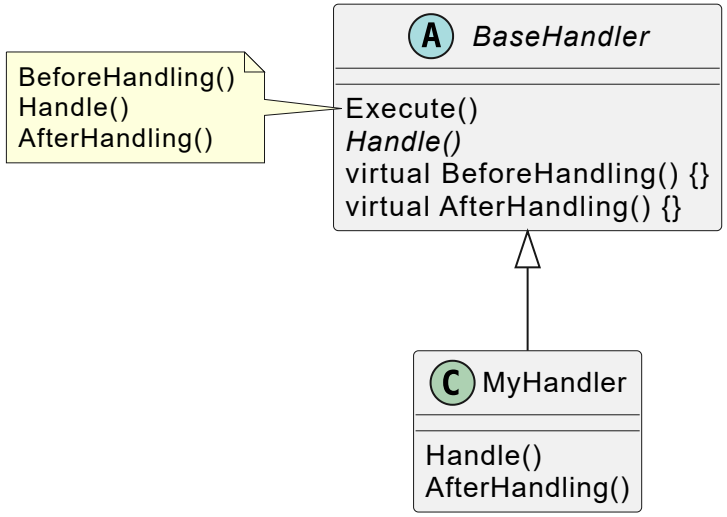


Template Method - example #1

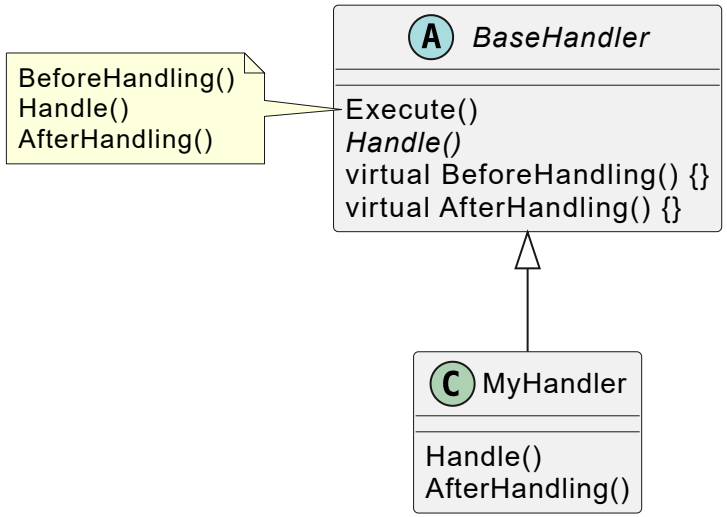
Chain of Responsibility base class "handle" method may be a *Template Method*:



Template Method - example #2



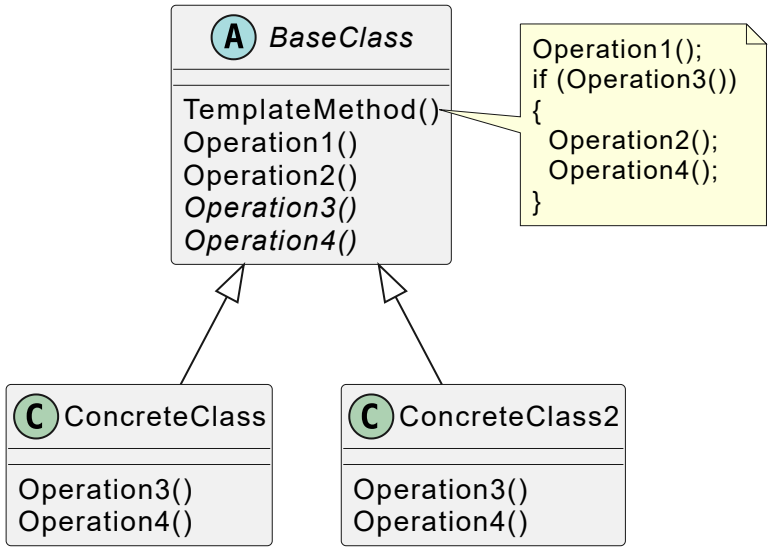
Template Method - example #2

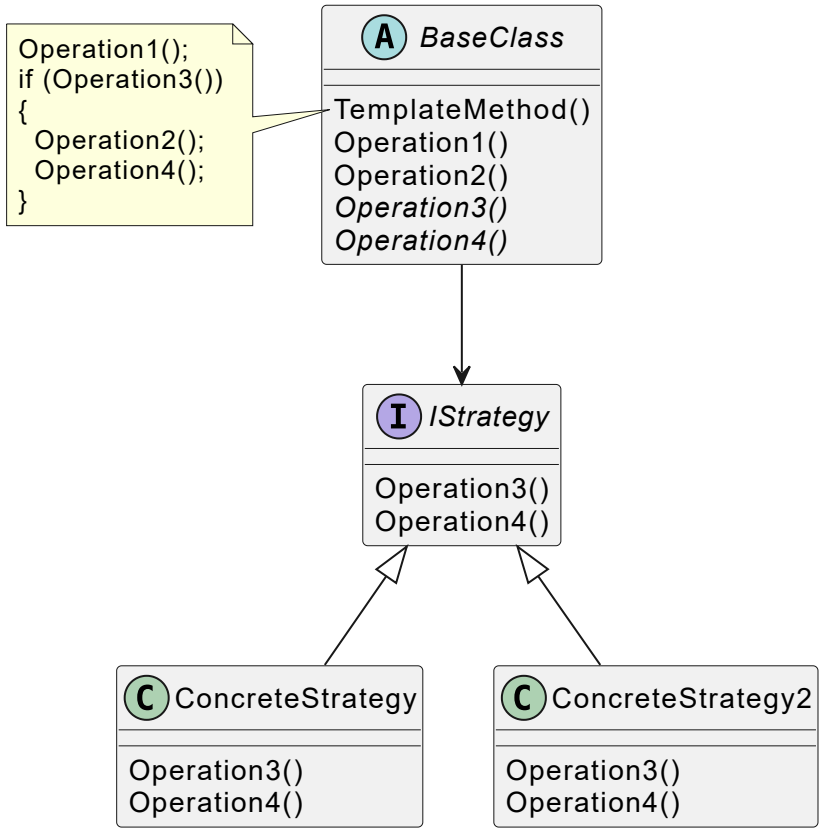



```
// // ASP.NET R.I.P. 💀
public class MyModule : IHttpModule
{
    public void Init(HttpApplication application)
    {
        application.BeginRequest += (new EventHandler(this.Application_BeginRequest));
        application.EndRequest += (new EventHandler(this.Application_EndRequest));
    }
```

Template Method

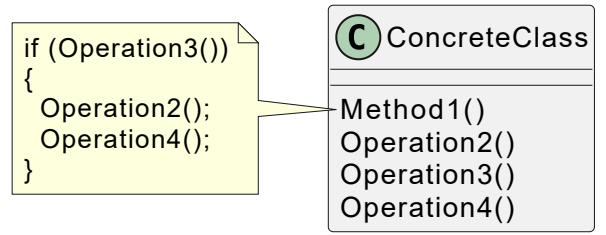
Quite opposite to the good practice - uses inheritance over composition - typically we's probably inject *Strategy* for those virtual steps

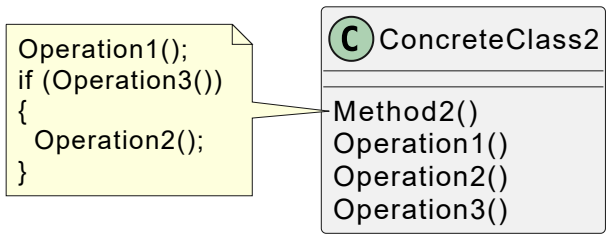


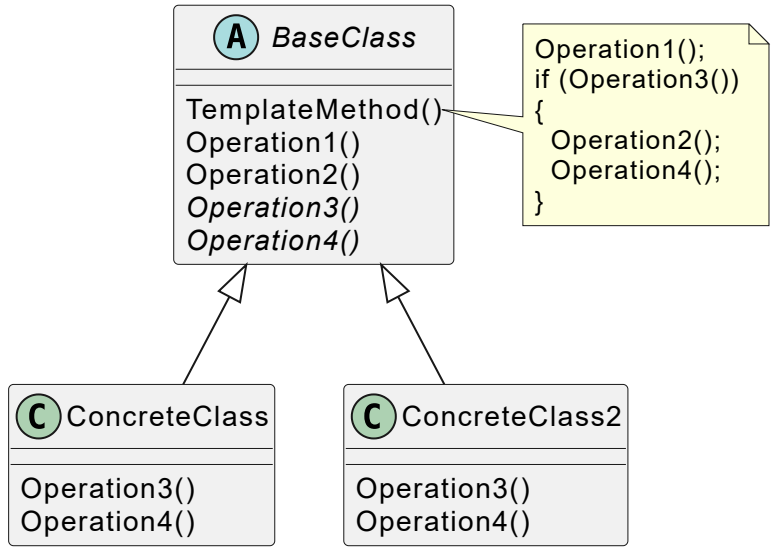


Template Method

Can be tempting to use for DRY deduplication (👀)







Template Method

```
internal class FeatureBehaviour : ICardBehaviour
{
 ...
 public void ApplyFor(GameEngine gameEngine,
     PlayerSide playerSide,
     Card playerCardState)
 {
   var gameState = gameEngine.Game;
   var player = gameState.GetPlayerState(playerSide);

   if (!player.CanPlayFeatureType(_featureType))
     throw new InvalidGameCommandException(...);
   player.FeaturePoints += _feaurePoints;
 }
}
```

```
internal class PerformanceBehaviour : ICardBehaviour
{
 ...
 public void ApplyFor(GameEngine gameEngine,
     PlayerSide playerSide,
     Card cardState)
 {
   var gameState = gameEngine.Game;
   var player = gameState.GetPlayerState(playerSide);

   player.Memory += _memory;
   player.Ticks += _ticks;
 }
}
```