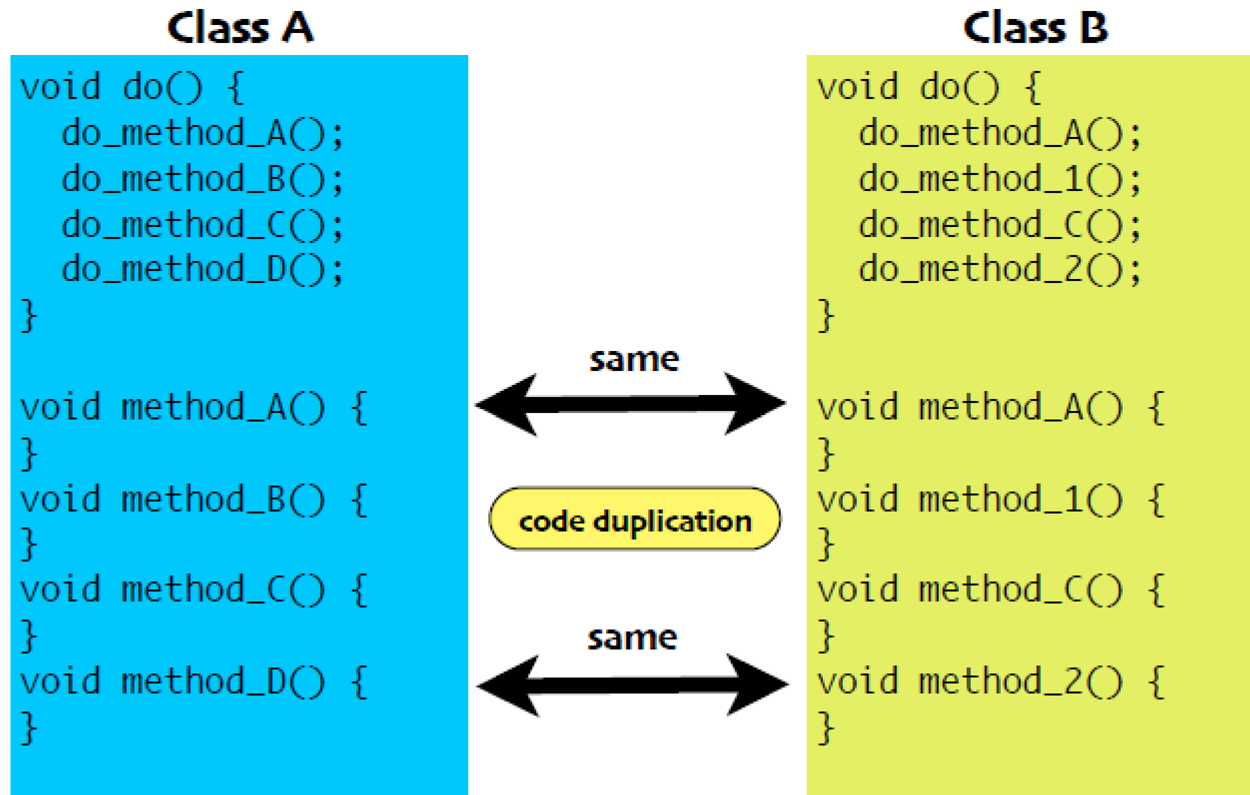
Template Method

- ★ Structural Patterns
  - » strategy
  - » adapter
  - » façade
- ★ Behavioral Patterns
  - » observer
  - » decorator
  - » command
  - » template method
- ★ Creational Patterns
  - » factory method
  - » abstract factory
  - » singleton

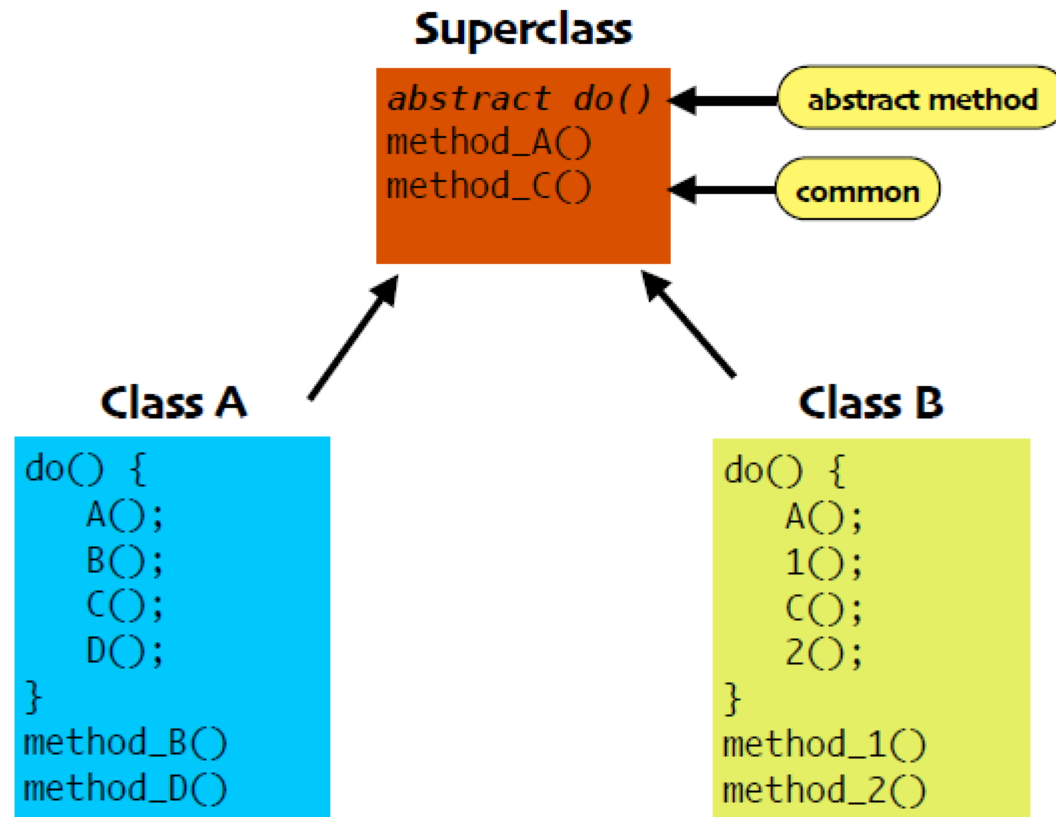
Existing Problem with Code

Duplicated code is difficult to change, maintain, or extend

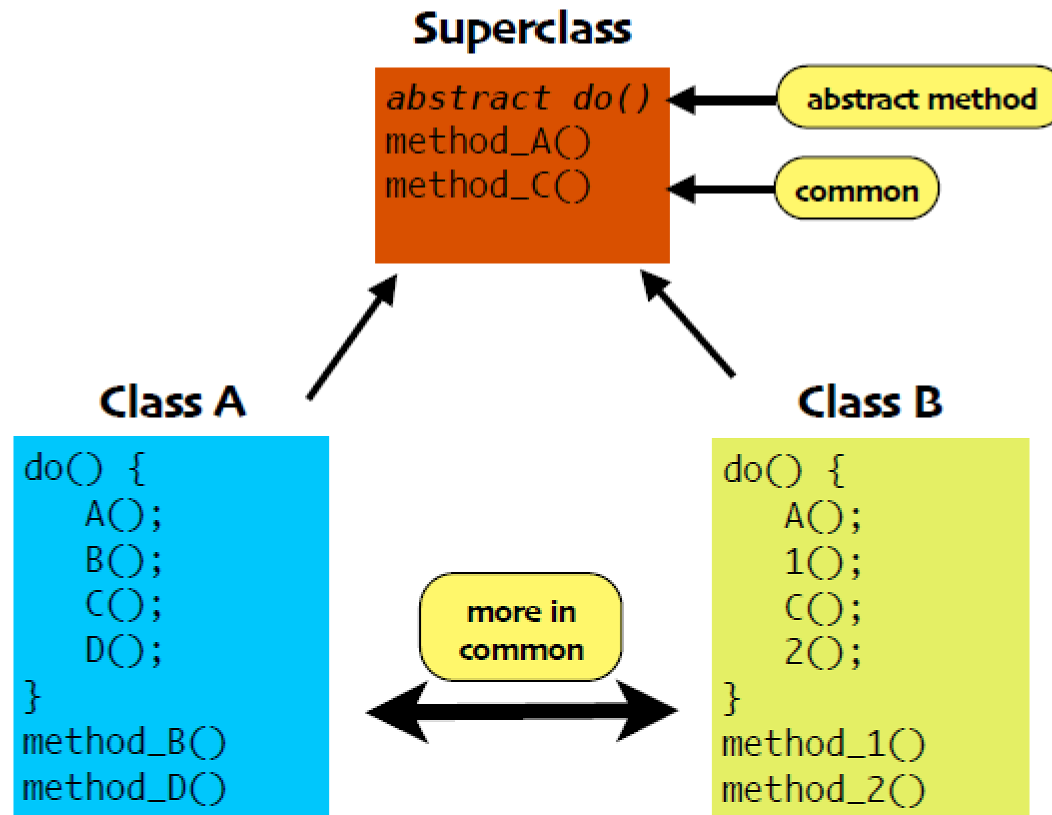
Example



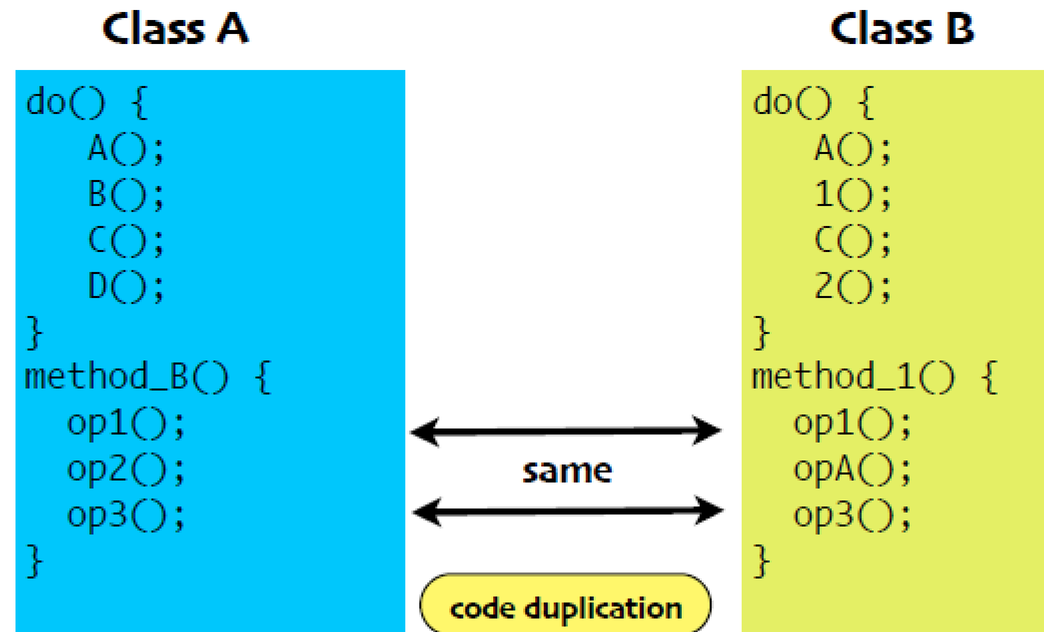
Remove Redundancy



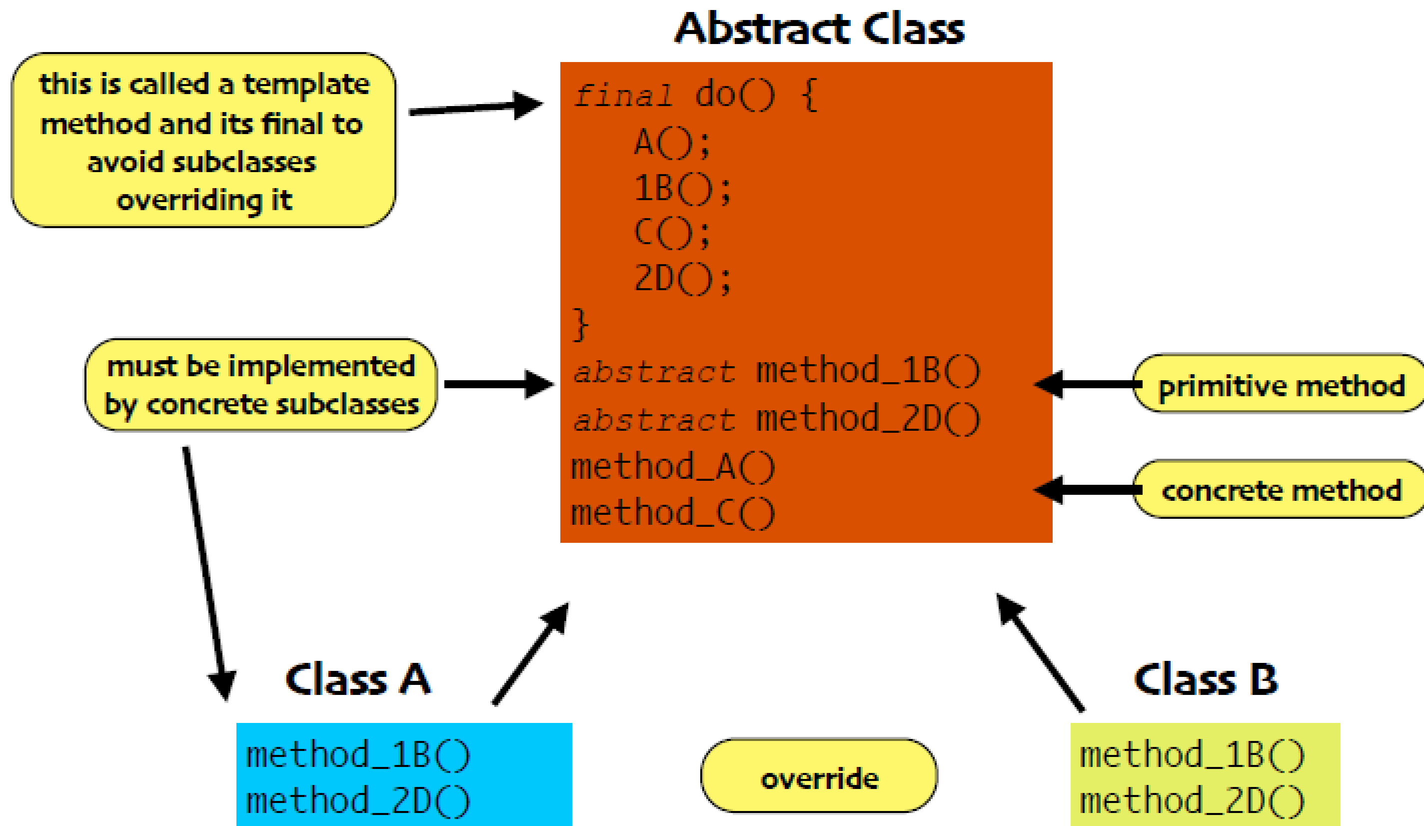
Remove Redundancy



Remove Redundancy



Define Template Method



Template Method Definition

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses
- Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- Note the method itself is typically declared final so that it cannot be modified by subclasses
- This pattern may be the most common pattern used!

From Wikipedia

```
abstract class AbstractClass {
    /* A template method : */
    final void TemplateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
      doSomething();
    }
}
```

generic

must be implemented by concrete subclasses

generic

Add a Hook (from HFDP)

```
abstract class AbstractClass {
    /* A template method : */
    final void TemplateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
     }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
      doSomething();
    }
    void hook() { }
}
```

useful for logging or whatever

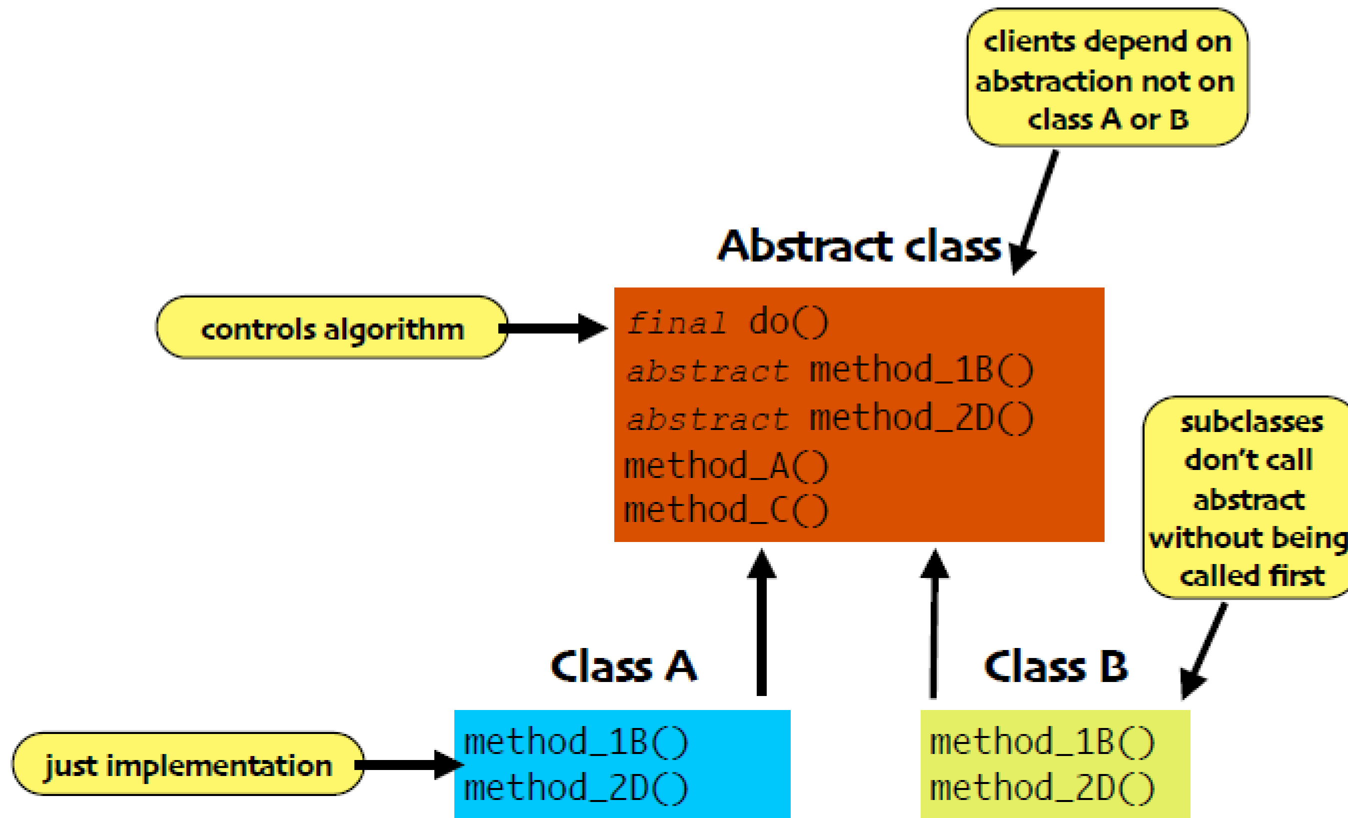
stub

in the concrete class you could do a test to decide what the hook should do

Hooks

- Concept is used often in APIs – Java API paint method for drawing is a hook
- Applets have hooks (repaint, start, stop, destroy, etc.)
- Junit TestCase (setUp, tearDown)
- Hook is an empty method or provides some default behavior in super class. Designed to by overridden **if deemed necessary** in subclasses.

Desing Principle: Depend on Abstraction



OO Design Principle

The Hollywood Principle: Don't call us, we'll call you

- A low-level component never calls a high-level component directly
- Low-level components can participate in a computation, but the high-level components control when and how
- Low-level components are sub-classes that provide implementations for abstract behaviors defined in super-class (high level)

Example from Wikipedia

```
/**
 * An abstract class that is
 * common to several games in
 * which players play against
 * the others, but only one is
 * playing at a given time.
 */

abstract class Game {

    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    /* A template method : */
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;        }
        printWinner();    }}
```

Example from Wikipedia

```
//Now we can extend this class in order
//to implement actual games:

class Monopoly extends Game {

    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        // Initialize money
    }
    void makePlay(int player) {
        // Process one turn of player    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules    }
    void printWinner() {
        // Display who won    }
    /* Specific declarations for the Monopoly game. */

    // ...}
```

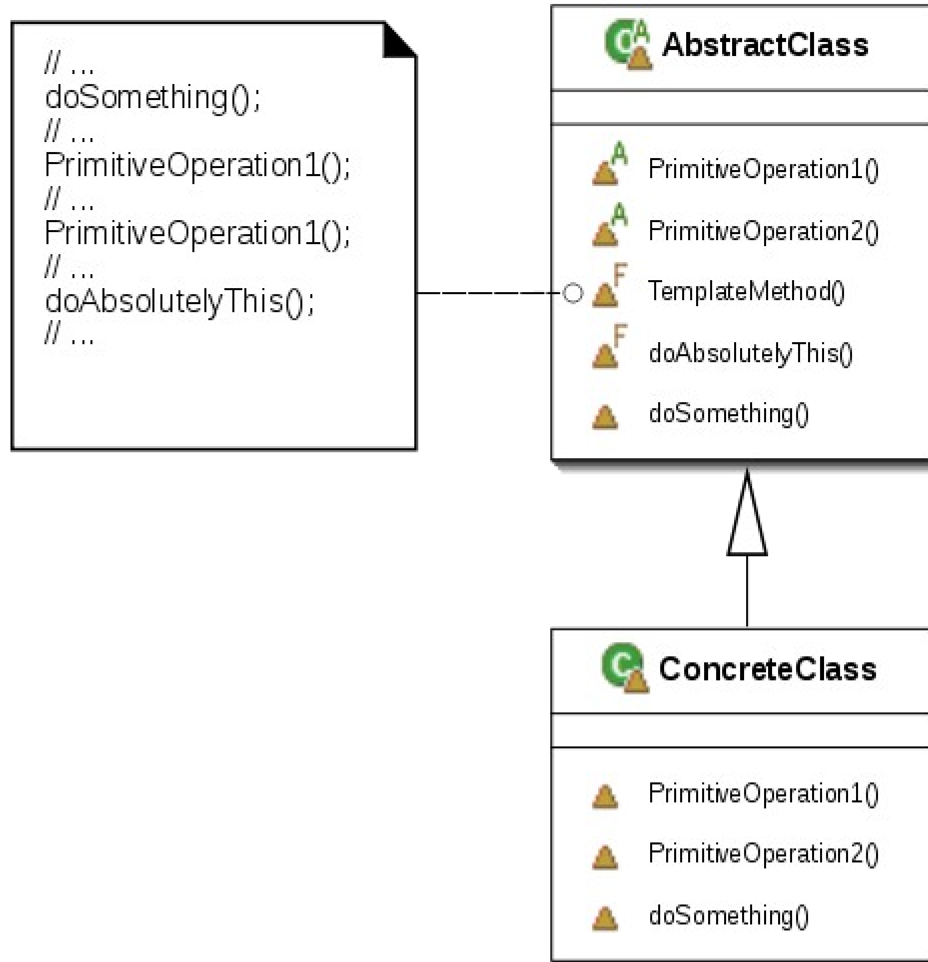
Example from Wikipedia

```
class Chess extends Game {

   /* Implementation of necessary concrete methods */
   void initializeGame() {
      // Initialize players
      // Put the pieces on the board
   }
   void makePlay(int player) {
      // Process a turn for the player
   }
   boolean endOfGame() {
      // Return true if in Checkmate or
      // Stalemate has been reached
   }
   void printWinner() {
      // Display the winning player
   }
   /* Specific declarations for the chess game. */

   // ...}
```

UML



Template Method Usage

- Let subclasses implement (through method overriding) behavior that can vary
- Avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in each of the subclasses.
- Control at what point(s) subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.
- There are many real world examples of this pattern in action, but not all follow the pattern 'by the book'

Template and Strategy

- Both encapsulate algorithms
- Template Method accomplishes its work via inheritance
- Strategy uses composition
- Which provides more flexibility?
- Side note: the Factory Method we know is a specialization of the Template Method :-)