STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW 322: Software Engineering Design VI

Structural Design Patterns
---Strategy Pattern
2020 Spring

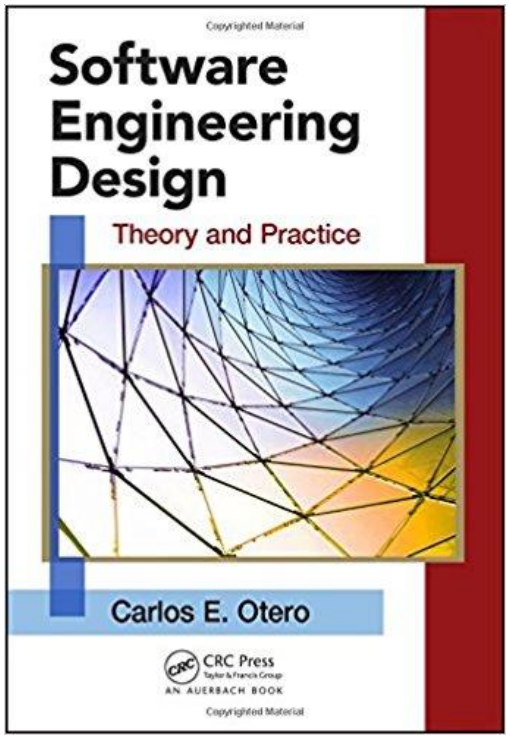
Prof. Lu Xiao
lxiao6@stevens.edu
Office Hour: Monday/Wednesday 2 to 4 pm
https://stevens.zoom.us/j/632866976
Software Engineering
School of Systems and Enterprises

Today's Topics

- Brief review of last lecture
- Behavioral design patterns:
  - Strategy pattern
- StarBuzz Application Extended
  - Decorator + Strategy
- Inheritance vs. Composition

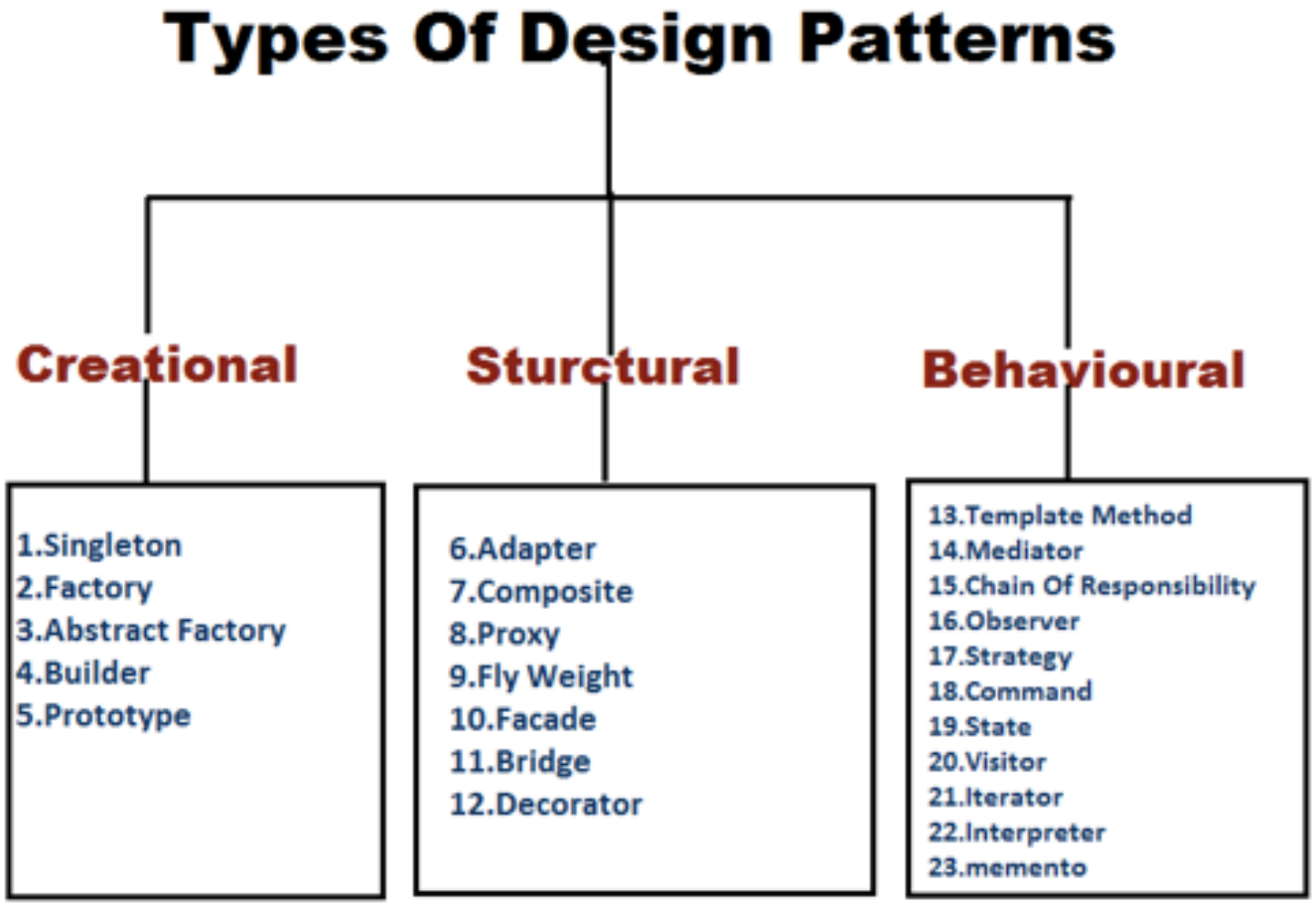
Acknowledgement

- https://sourcemaking.com/design_patterns
- **Software Engineering Design: Theory and Practice**



Design Pattern

- ***Common*** design solutions to problems that ***recur*** in different systems.



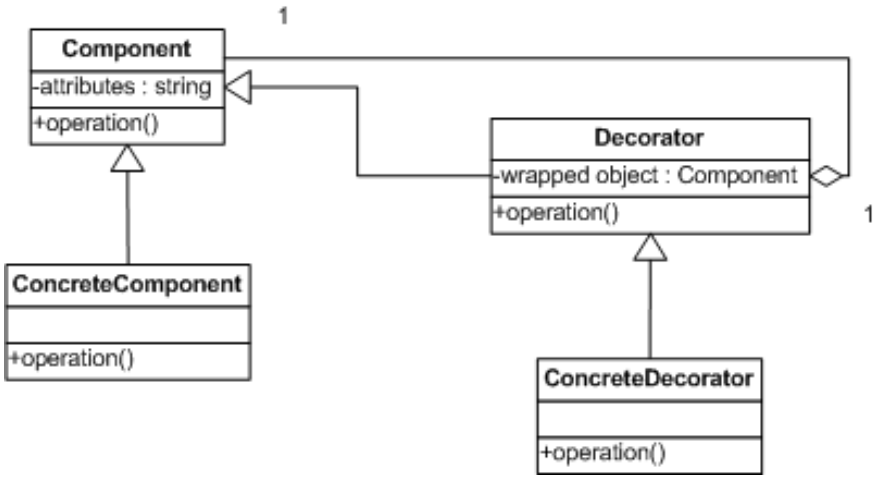
Last Lecture

- What type of design patterns we learned last time?

- Which pattern we learned last time?

Last Lecture

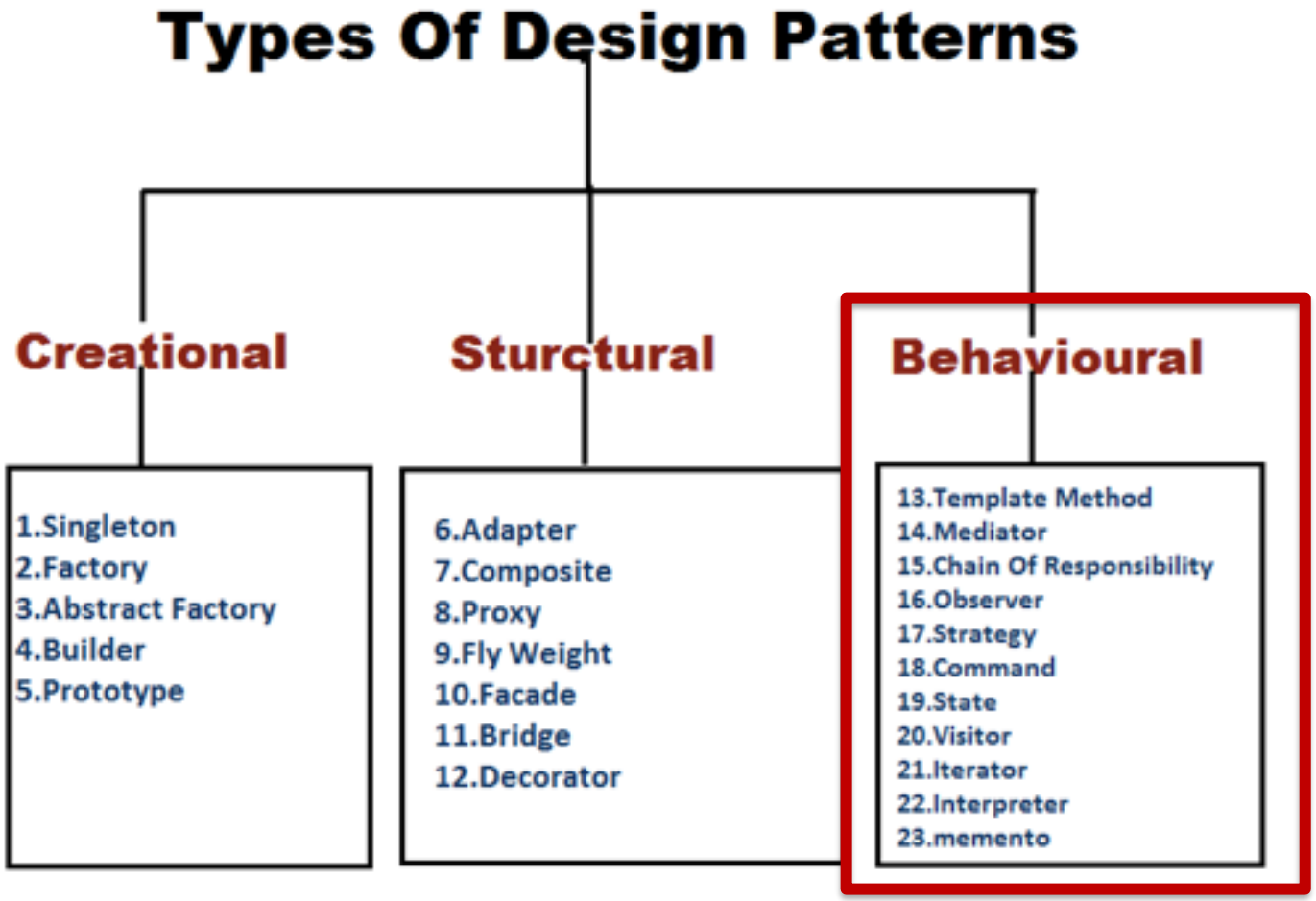
- What type of design patterns we learned last time?
  - Structural design pattern
- Which pattern we learned last time?
  - Decorator pattern:



Add responsibilities to objects dynamically

Design Pattern

- ***Common*** design solutions to problems that ***recur*** in different systems.

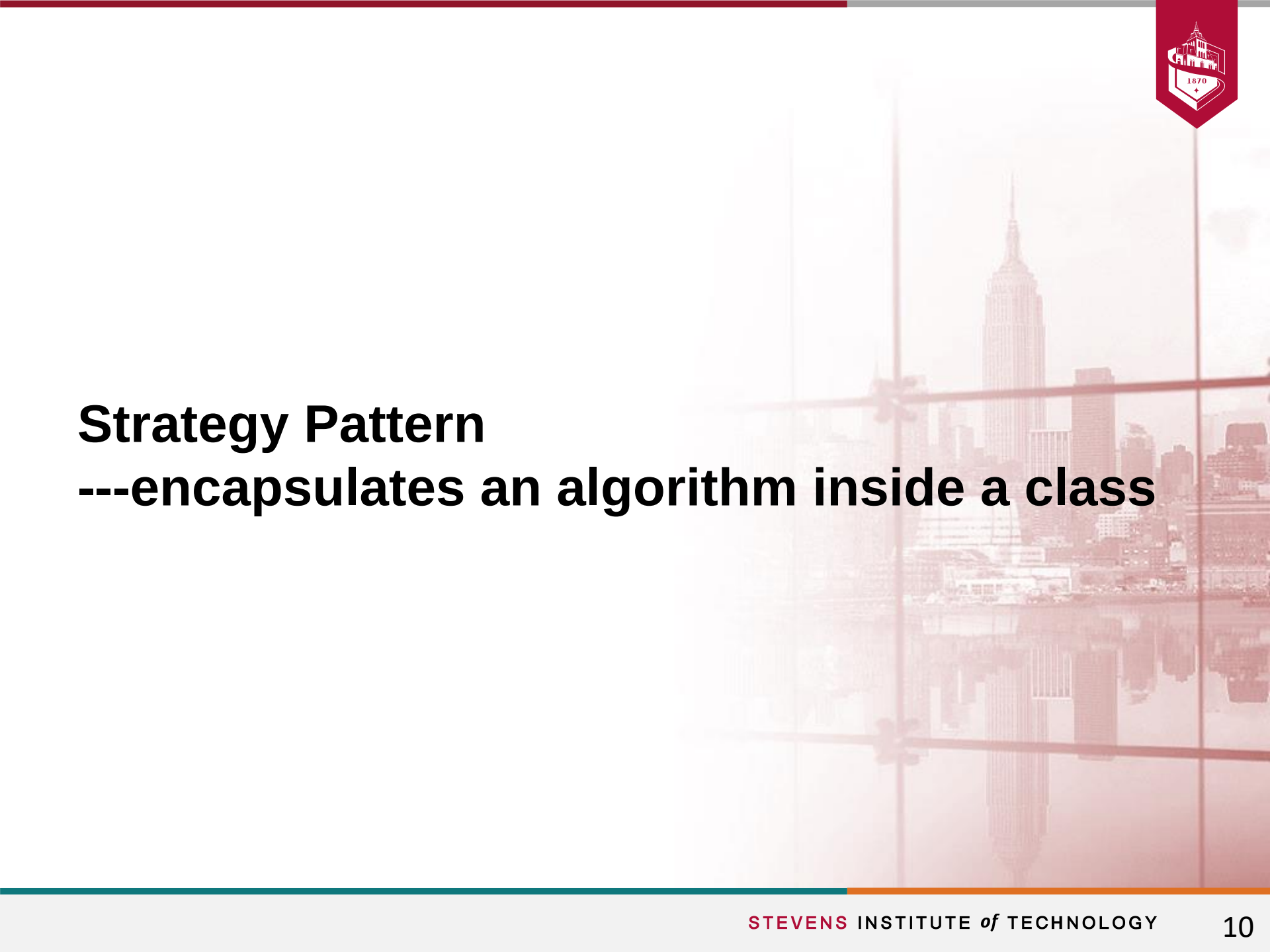


Behavioral Design Patterns

- Behavioral design patterns deal with encapsulating behavior with objects, assigning responsibility, and managing object cooperation when achieving common tasks (Gamma et. al. 1995).
- Common behavioral design patterns include:
  - Iterator: sequentially access the elements of a collection
  - Observer: a way of notifying change to a number of classes
  - Strategy: encapsulates an algorithm inside a class
  - …

Behavioral Design Patterns

- Behavioral design patterns deal with encapsulating behavior with objects, assigning responsibility, and managing object cooperation when achieving common tasks (Gamma et. al. 1995).
- Common behavioral design patterns include:
  - Iterator: sequentially access the elements of a collection
  - Observer: a way of notifying change to a number of classes
  - Strategy: encapsulates an algorithm inside a class
  - …



Strategy Pattern
---encapsulates an algorithm inside a class

Strategy Pattern

- The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them ***interchangeable***.
  - Strategy lets the algorithm vary independently from clients that use it.
  - Strategy captures the abstraction in an interface, bury implementation details in derived classes (i.e. concrete strategies).
- Rationale: We anticipate what may ***change*** and we design in a way that allows us to implement the change(s) with a small number of modifications.
  - One of the dominant principle of OO design is the "open-closed principle".

International E-Commerce System

- As a simple example, consider the development of an e-commerce system for a company, which is located in the US, but its business has an international scope.
- A **TaskController** object handles sales request and passes it to the **SalesOrder** object to process the order.

The SalesOrder Object

The functions for **SalesOrder** include the following:

- Allow for filling out the order information.
- Handle tax calculations.
- Process the order and print a sales receipt.

Some of these functions are likely to be implemented with the help of other objects. For example, **SalesOrder** might call a **SalesTicket** object that prints the **SalesOrder**.

Anticipated Point of Changes

The functions for **SalesOrder** include the following:

- Allow for filling out the order information.
- ✓ ***Handle tax calculations.***
- Process the order and print a sales receipt.

Tax calculation is country-specific, and there is potential for changes to handle taxation outside the United States.

An If-Else Approach

A Simple case – handle tax in the two countries:

```
if nation == “US”:
        // US Tax Rules here
elif nation == “Canada”:
        // Canadian Tax Rules here
```

An If-Else Approach

A Simple case – handle currency in the two countries:


```
if nation == “US”:
        // US Currency Rules here
elif nation == “Canada”:
        // Canadian Currency Rules here
```

An If-Else Approach

- Adding a third country … still a “clean” implementation

```
if nation == “US”:
        // US Tax Rules here
elif nation == “Canada”:
        // Canadian Tax Rules here
elif nation == “Germany”:
        // Germany Tax Rules here
```

```
if nation == “US”:
        // US Currency Rules here
elif nation == “Canada”:
        // Canadian Currency Rules here
elif nation == “Germany”:
        // Germany Currency Rules here
```

An If-Else Approach

You may need to add another If-Else block for handling language

```
if nation == “US”:
        // use English
elif nation == “Canada”:
        // use English
elif nation == “Germany”:
        // use German
```

An If-Else Approach

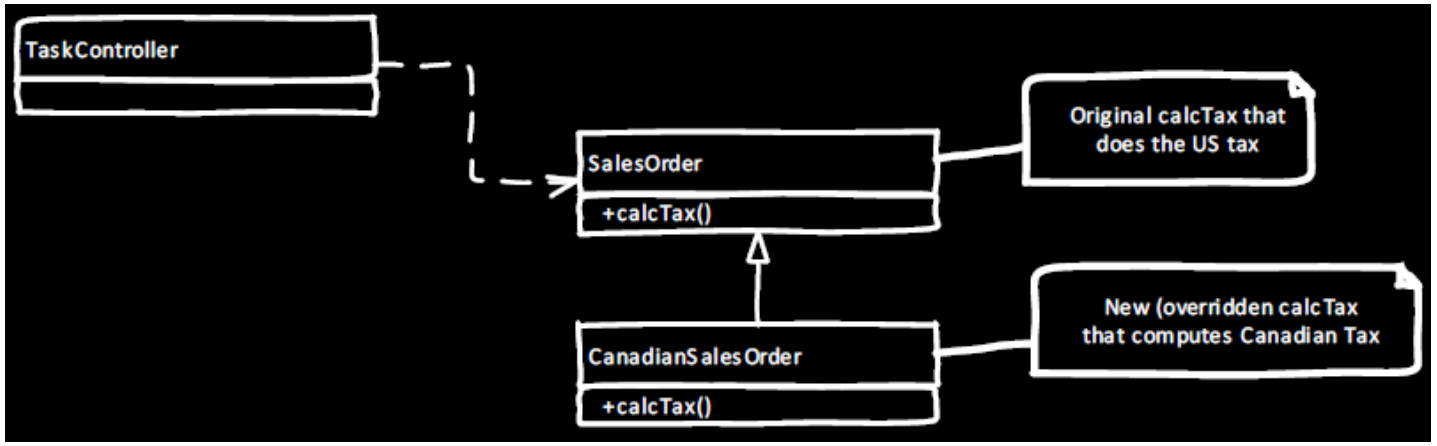
But…variations "dirty" the implementation…

```
if nation == "US":
        // use English
elif nation == "Canada":
        if inQuebec:
                // use French
        elif:
                // use English
elif nation == "Germany":
        // use German
```

Switch Creep…

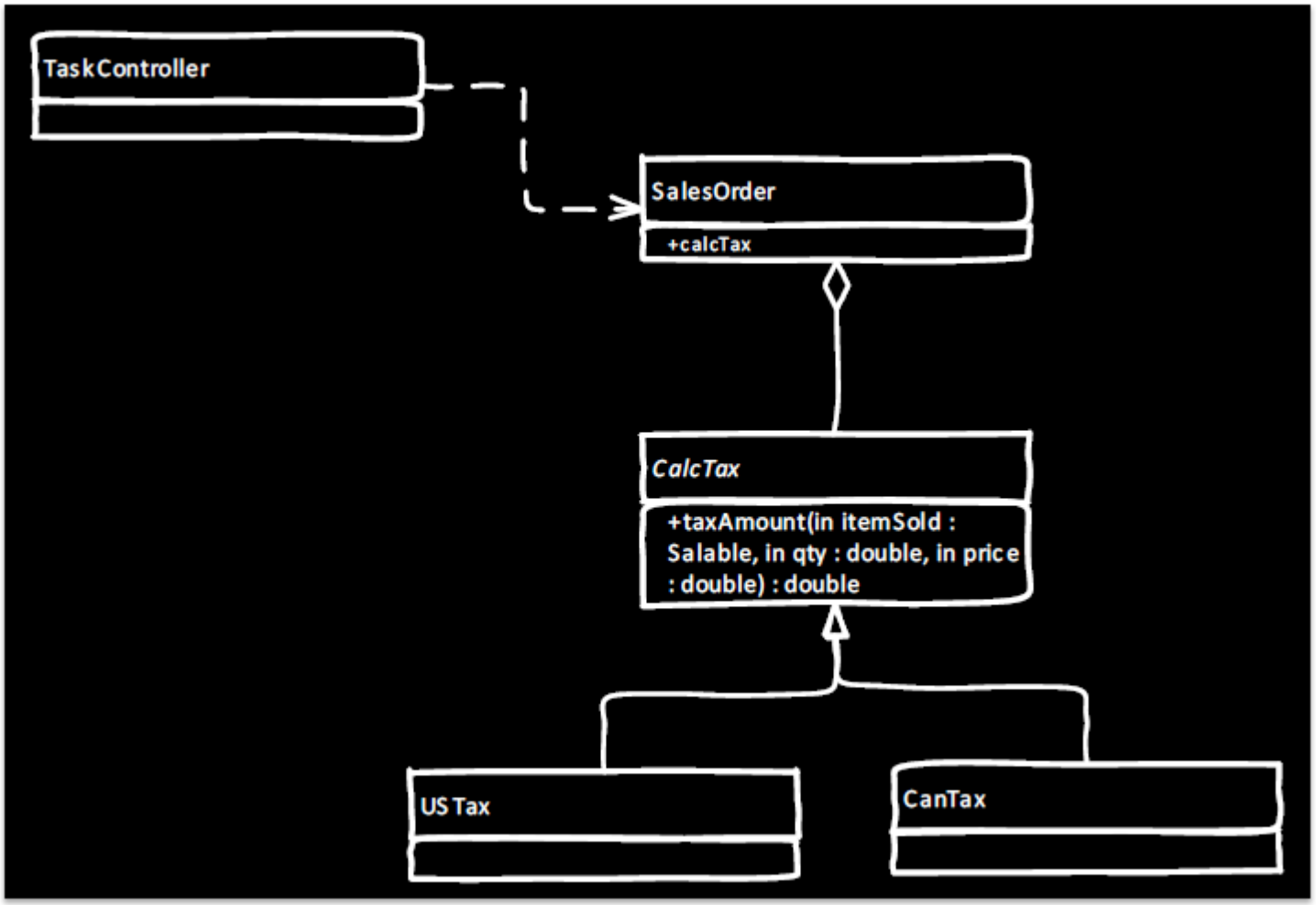
- Flow of the switches becomes harder to read.
- When new cases come in, you must find every place it can be involved.

An Inheritance Approach



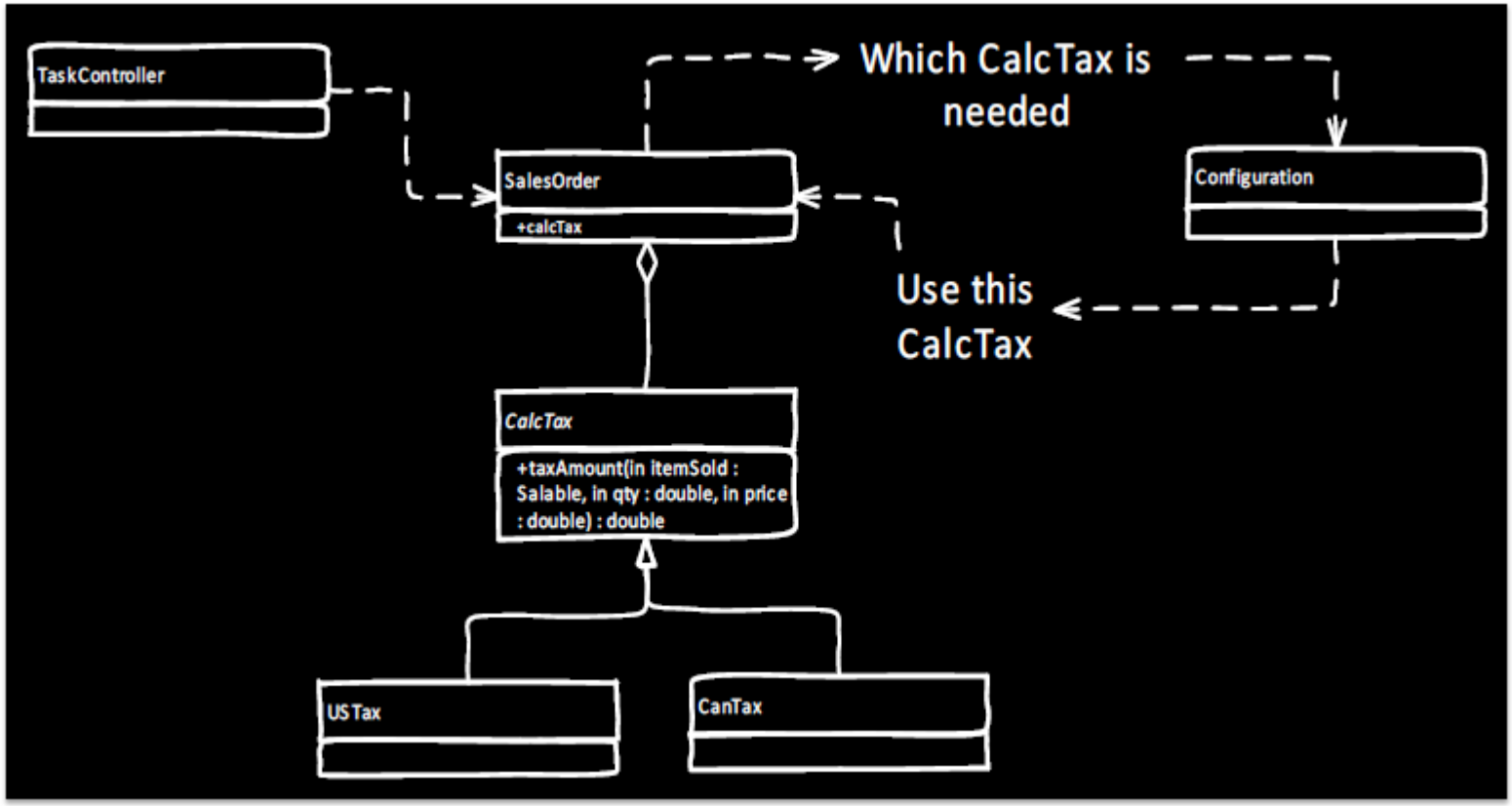
- Derive a new **CanadianSalesOrder** class from **SalesOrder**
  - Achieves reuse (although not the primary design objective).
- Overall, not a bad approach for small hierarchies, but not ideal because of potential changes.
  - What about dealing with varying date format, language, and freight rules?

A Better Approach



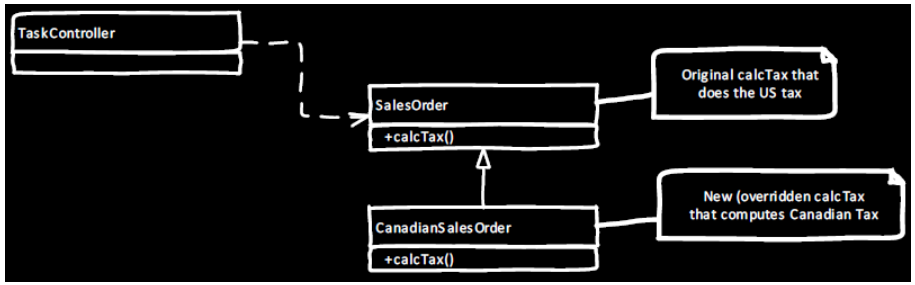
Now use **aggregation** to give the **SalesOrder** object the ability to handle Tax.

Further Shifting Responsibilities

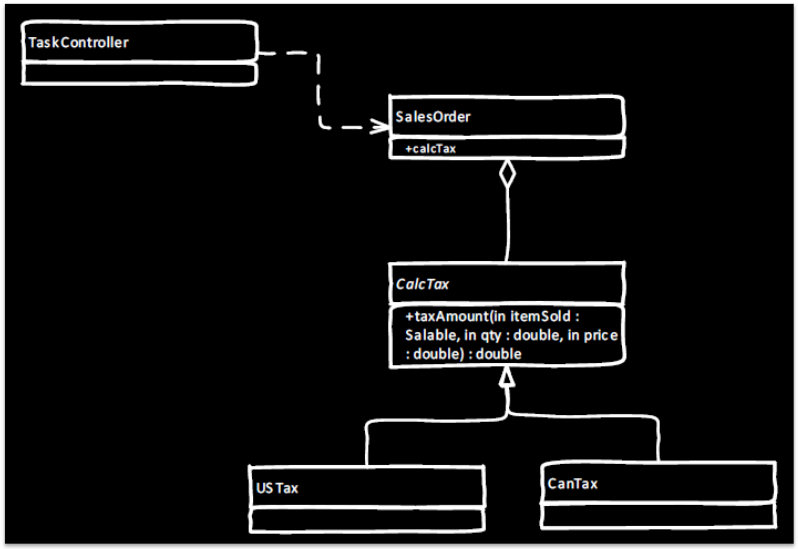


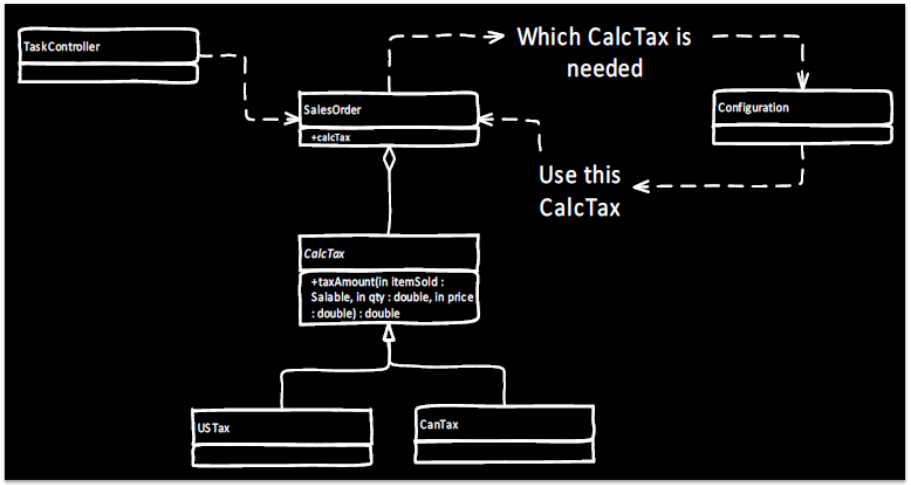
- **Configuration** is responsible of deciding which Tax Calculator to use in different circumstances
- **Separation of Concerns**

Comparing Approaches



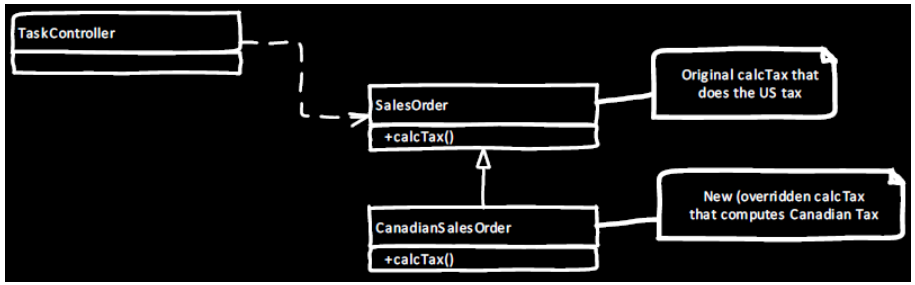
So, what is the difference?

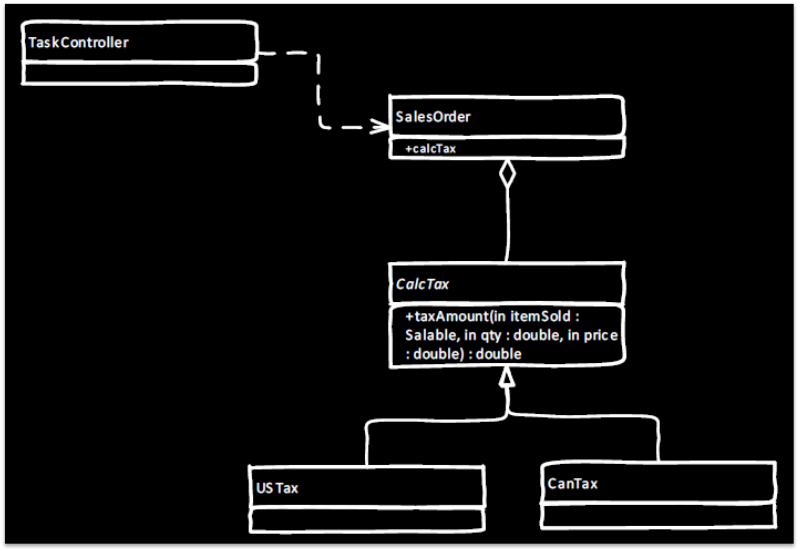


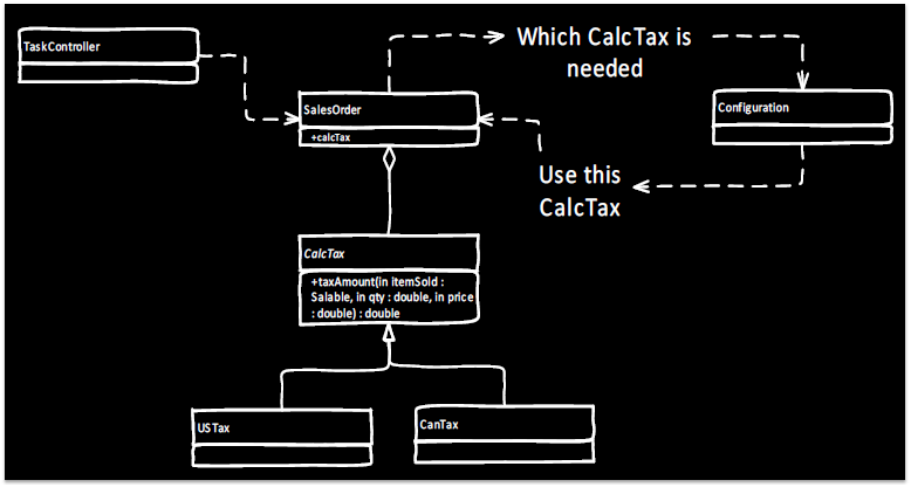


Comparing Approaches

1. Switch Creep in the SalesOrder

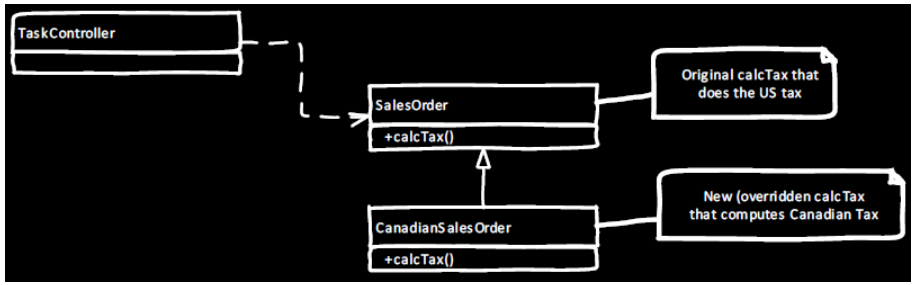




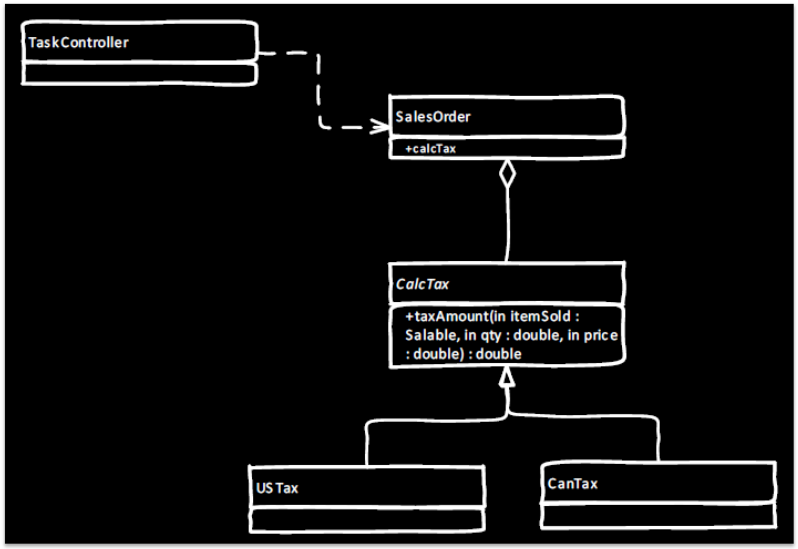


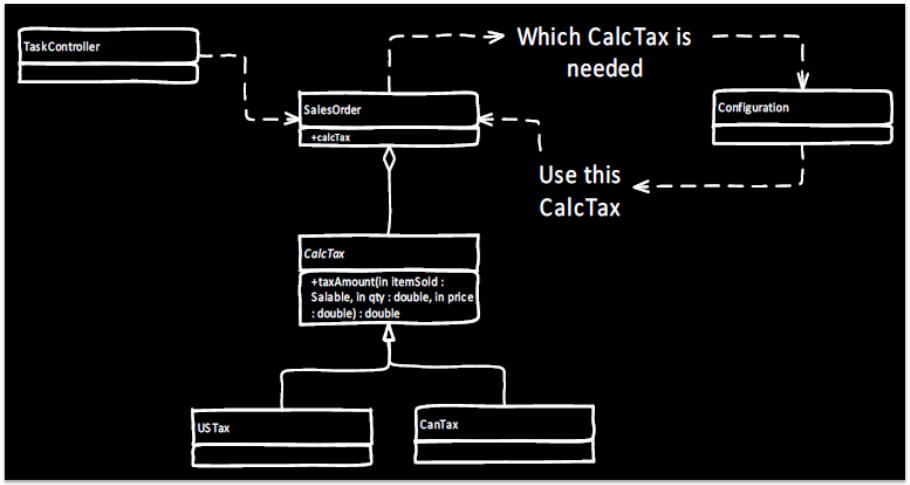
Comparing Approaches

1. Switch Creep in the SalesOrder



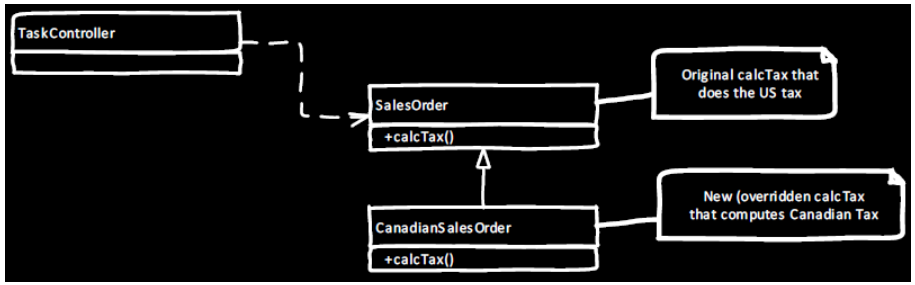
2. Switch moved to TaskController





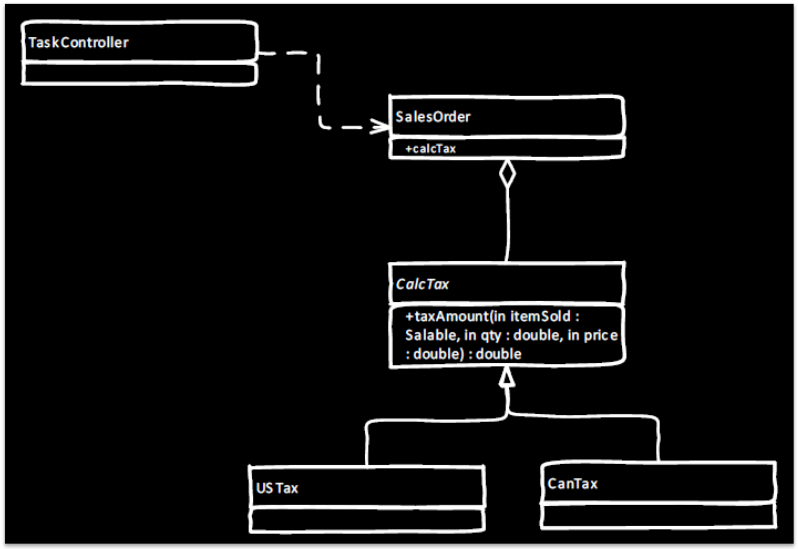
Comparing Approaches

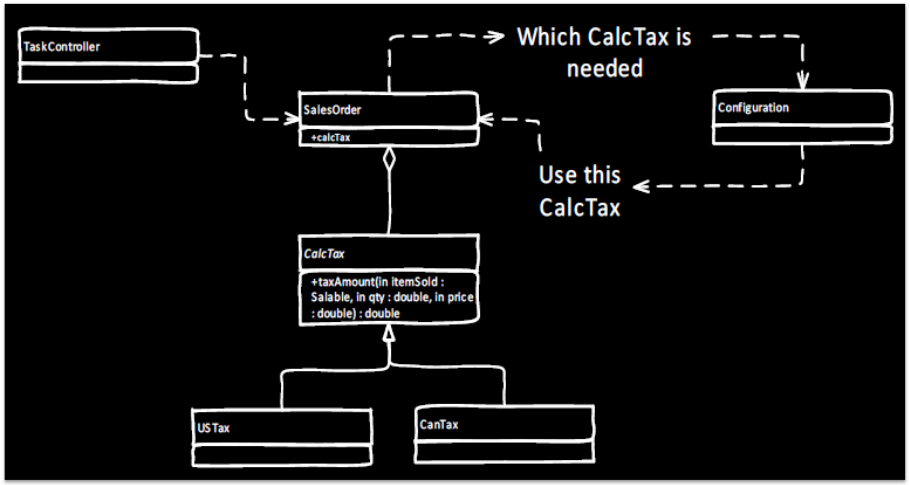
1. Switch Creep in the SalesOrder



2. Switch moved to TaskController

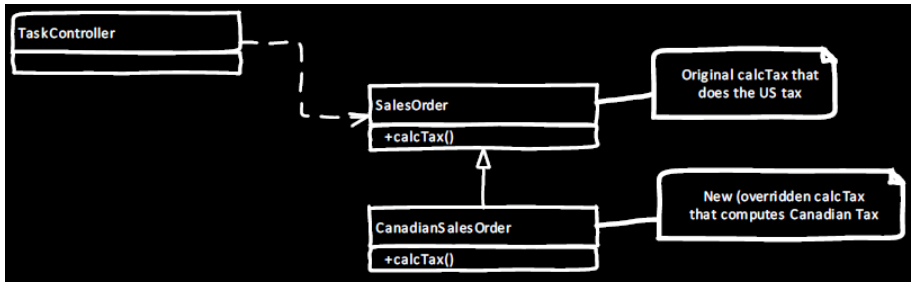
3. Tax Calculation is separated from SalesOrder, but SalesOrder still handles switch statements





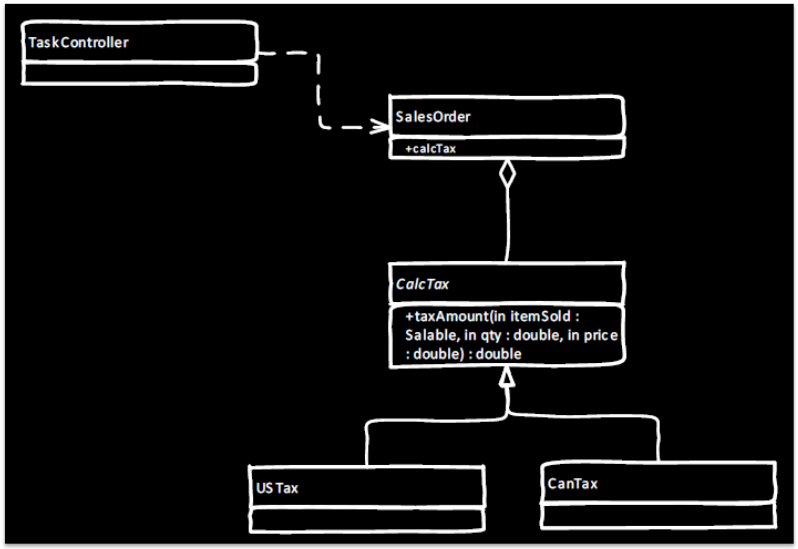
Comparing Approaches

1. Switch Creep in the SalesOrder

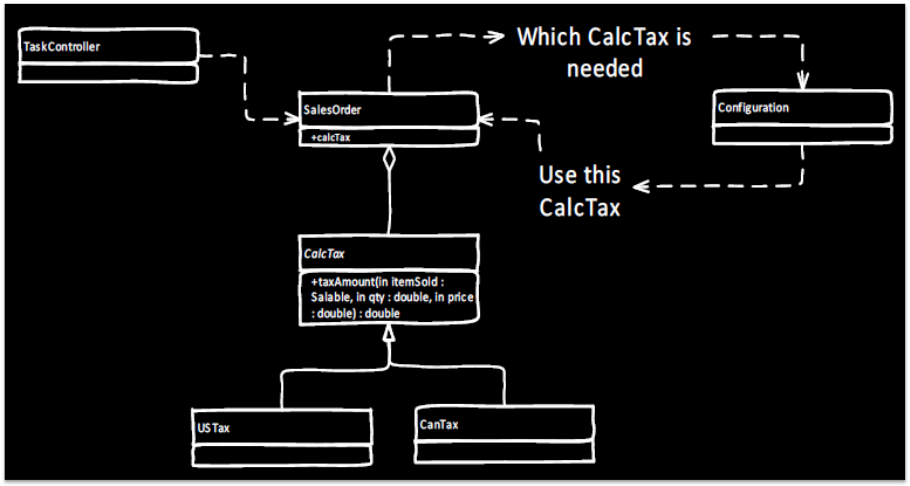


2. Switch moved to TaskController

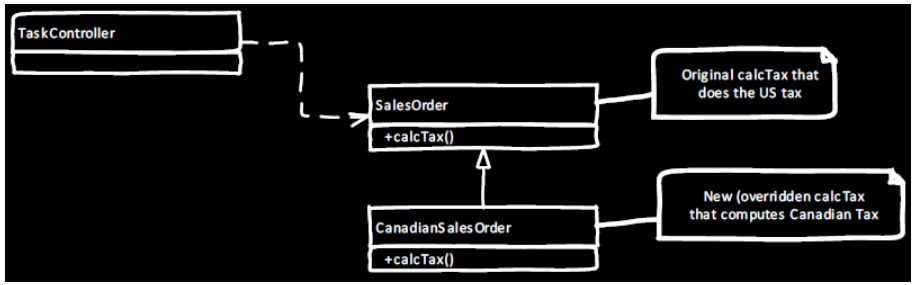
3. Tax Calculation is separated from SalesOrder, but SalesOrder still handles switch statements



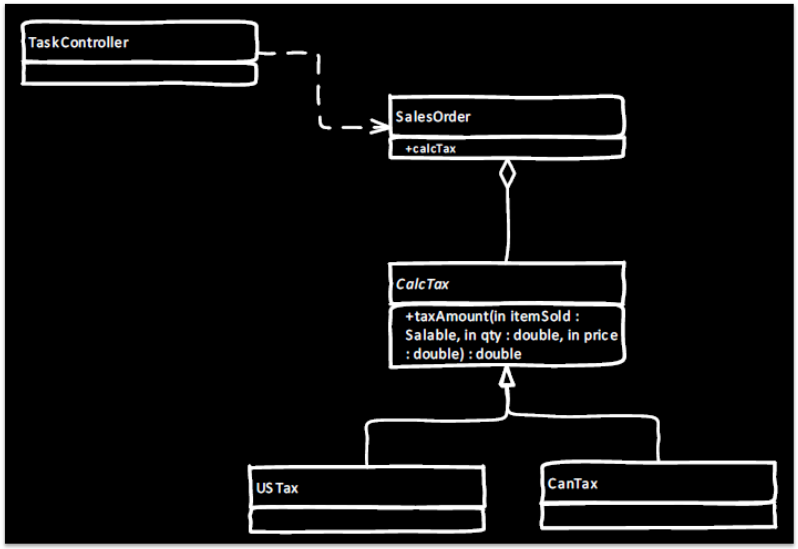
4. Tax Calculation is separated from SalesOrder, and Configuration handles switch in specific.

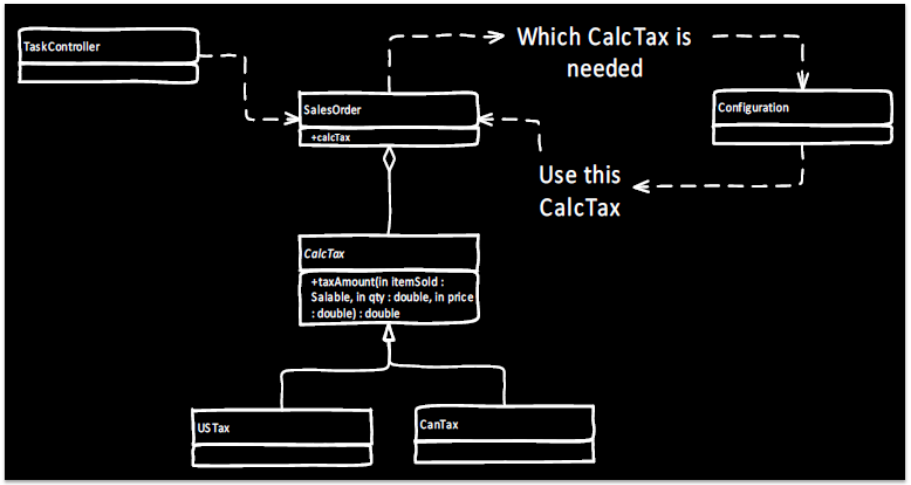


Comparing Approaches

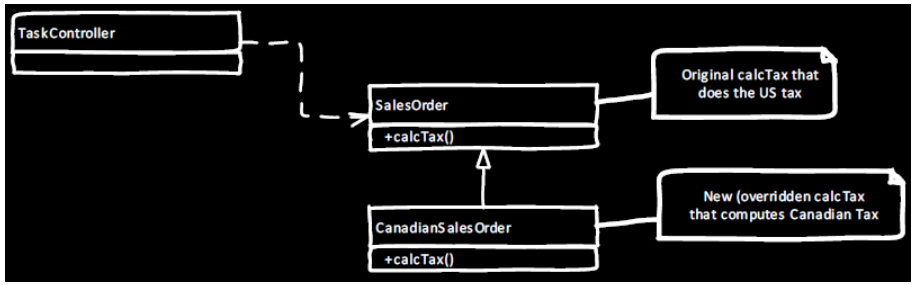


- Looking at it quickly, it appears we just pushed the problem down the chain.
- The key difference is where the hierarchy happens and the implications it has on system maintenance.



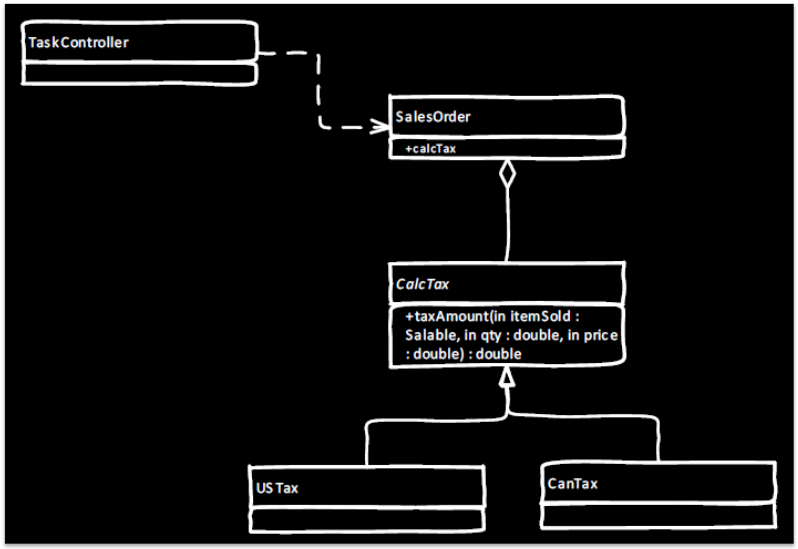


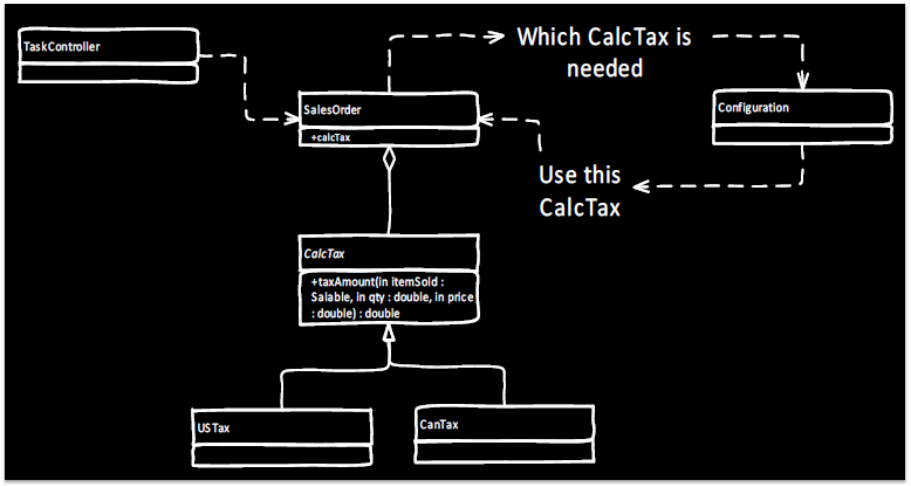
Comparing Approaches



- Looking at it quickly, it appears we just pushed the problem down the chain.
- The key difference is where the hierarchy happens and the implications it has on system maintenance.

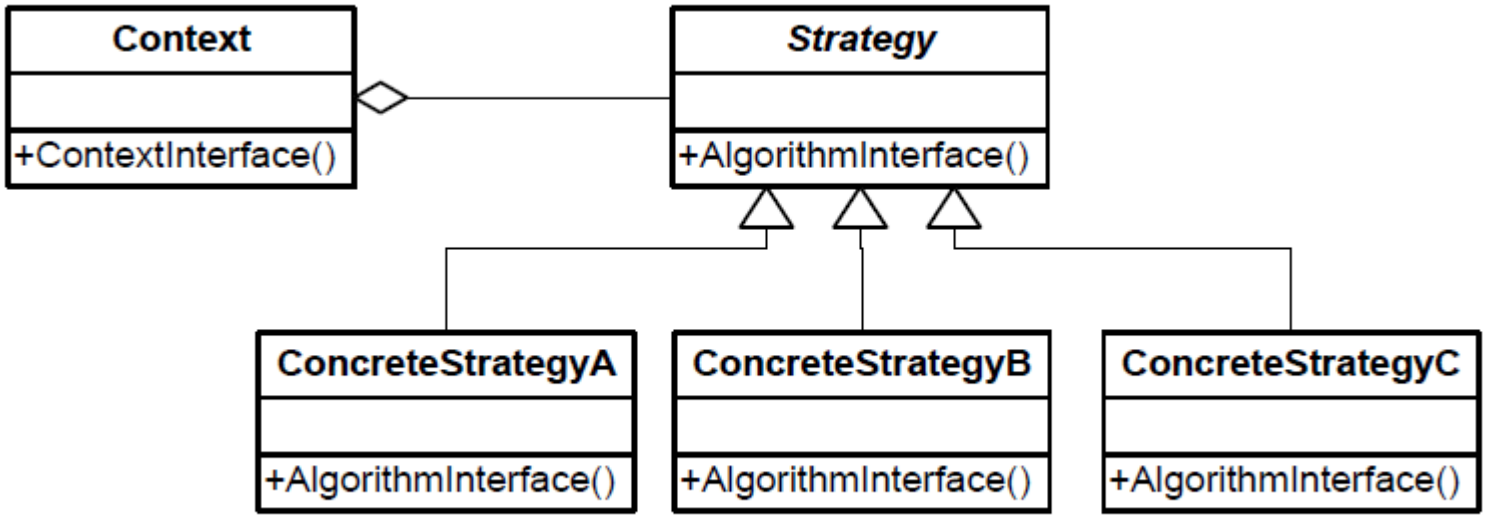
The points of changes are separated from other parts of the code!!





Meet the Strategy Pattern – Photo ID

- Encapsulate variations of an algorithm inside concrete classes.
- Variations of an algorithm are used inter-changeablely without the awareness of the direct client.



Benefits of Strategy Pattern

- Reuse: Take advantage of super-class Strategy *as a framework*
- Extendibility: Allow new strategies to be added without affecting other strategies or the clients
- Maintainability: Changes and extensions are localized and do not have ripple effects

Highly reusable + Highly maintainable

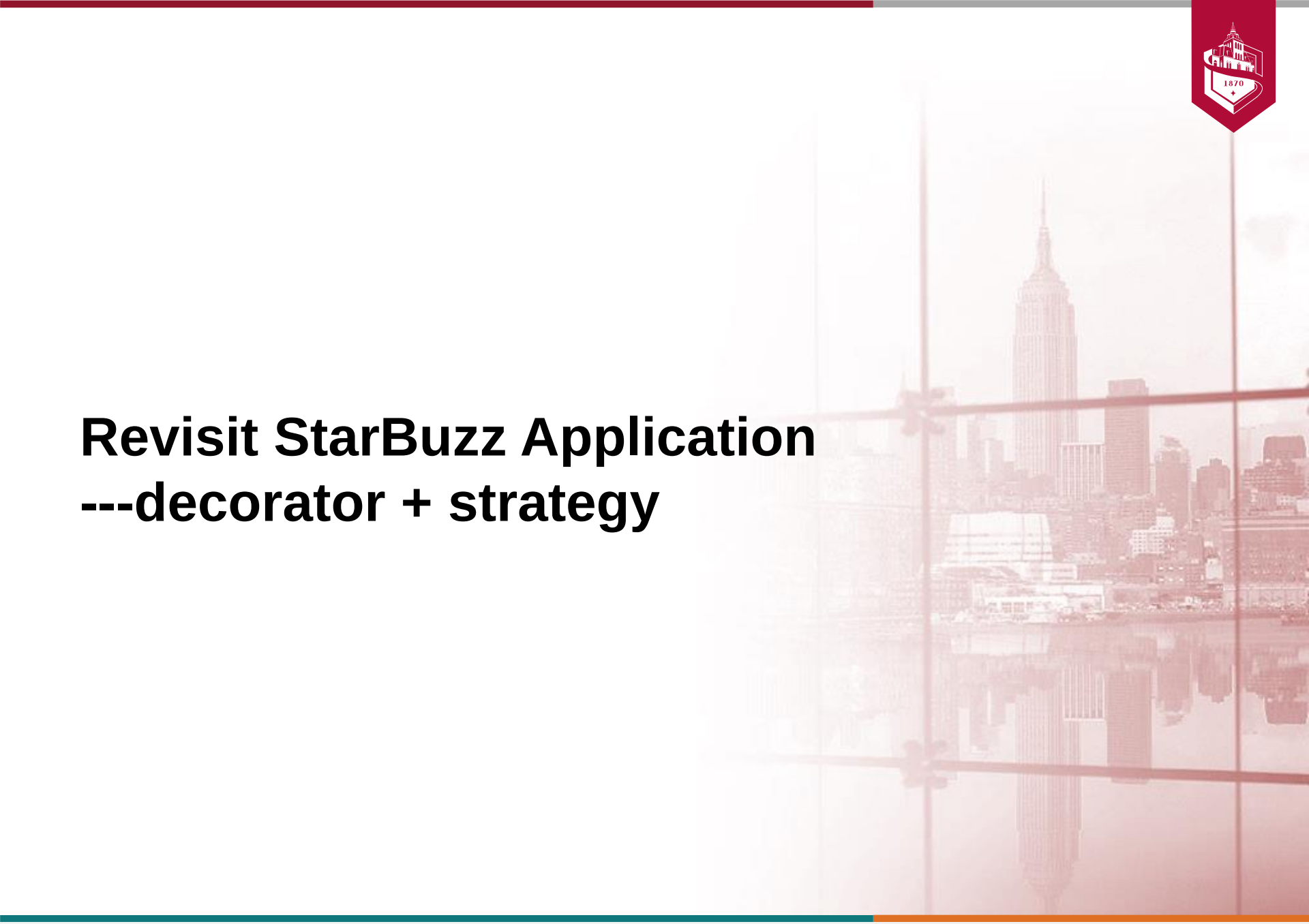
Strategy Key Features

- Intent: Enable you to use different business rules or algorithms depending on the context in which they occur.
- Problem: The selection of an algorithm that needs to be applied depends on the client making the request or the data being acted on.
- Solution: Separate the selection of the algorithm from the implementation of the algorithm.
- Consequences: Switches and/or conditionals can be eliminated. You must invoke all algorithms the same way.
- Implementation: Have the class that uses the algorithm (Context) contain an abstract class (Strategy) that has an abstract method specifying how to call the algorithm. Each derived class implements the algorithm needed.

Design Philosophy Behind Strategy

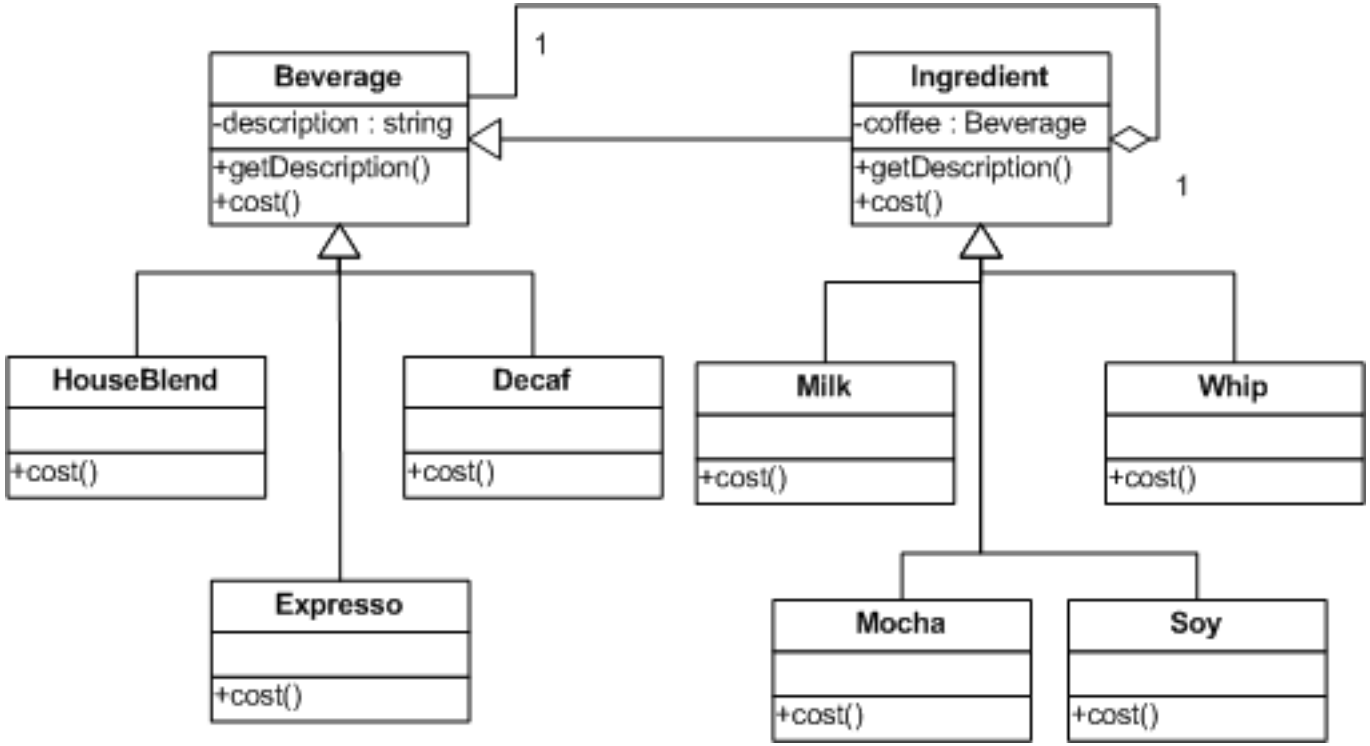
Information Hiding: Identify the aspects of your application that vary and separate them from what stays the same

- *Take what varies and "extract" it so it won't affect the rest of your code*
- *The result? Fewer unintended consequences from code changes and more flexibility in your system*



Revisit StarBuzz Application ---decorator + strategy

The StarBuzz Problem



Change!!

- The company introduces sizes for their beverages: small, medium, large
- Pricing depends on size, for beverage as well as for ingredients

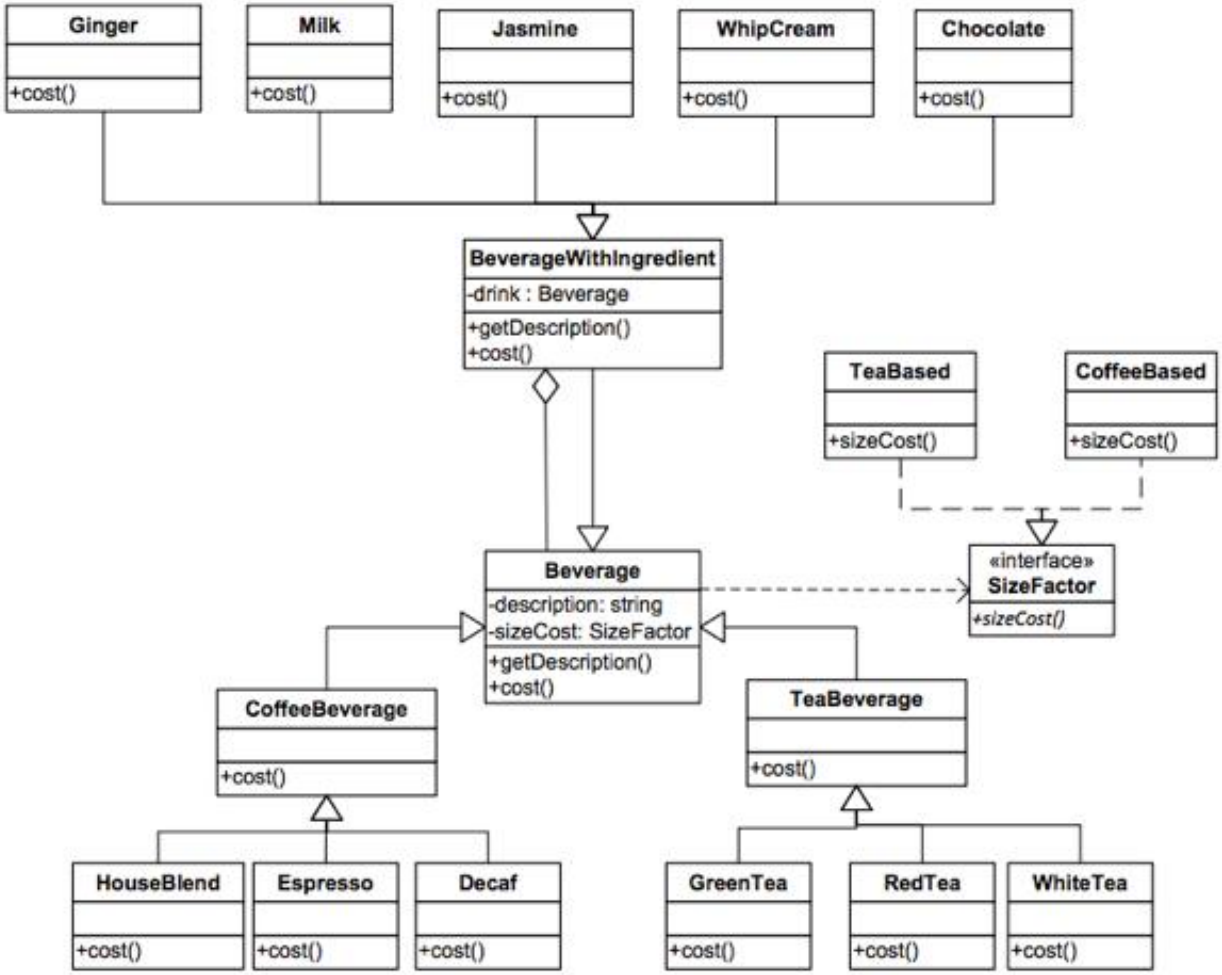
This is complicated, because…

- The company offer basic coffee beverages, including: HouseBlend, Espresso, Decaf
  - Other coffee beverages can be produced by adding the following ingredients to basic coffee beverages: Chocolate, Milk, Whip Cream.
- The company also offers Tea beverages…The basic tea beverages include: green, red, and white tea
  - Other tea beverage can be produced by adding the following ingredients: Jasmine, Ginger , Milk

Apply Strategy Pattern

- Required solution: **Decorator Pattern + Strategy Pattern**

StarBuzz Solution



Composition vs. Inheritance

- They both are ways to re-use functionality
- Inheritance:
  - Re-use functionality of parent class
  - Statically decided
  - Weakens encapsulation
- Composition:
  - Re-use functionality of objects collected at run-time
  - Invoked through the interface
  - Black-box re-use

Composition vs. Inheritance

- Composition allows more behavior flexibility
  - When possible use composition instead of inheritance
- Inheritance is still a quick way to design new components that are variants of existing ones. Over-use of inheritance creates bloated hierarchies
  - Code is more difficult to maintain
  - Unnecessary baggage for many classes
- Composition drawback: it becomes harder to understand the behavior of a program by looking only at its source code
  - Semantics of interaction are decided at run-time


thank you