CSC 335: Object-Oriented Programming and Design

Pattern-Oriented Design

by Rick Mercer based on the GoF book and

Design Patterns Explained

A New Perspective on Object-Oriented Design

Alan Shalloway, James R. Trott

Addison Wesley ISBN 0-201-71594-5

Using Patterns to Design

- There are 23 Object-Oriented design patterns cataloged in the GoF book--we've considered 10 so far (Fall 09)
  - Iterator, Observer, Strategy, Composite, Singleton, Flyweight, Command, Template, Chain of Responsibility, Decorator
- We'll use some patterns to help design a system
  - The new case study is in electronic retailing over the internet (*An Ecommerce system)*
  - Several design decisions will be aided by knowledge of existing design patterns
    - at a fairly high level of abstraction

Plan too much, plan ahead, or don't plan at all?

- Development of software systems can suffer from analysis paralysis: attempt to consider all possible changes in the future
- At other times developers jump to code too quickly
  - there is tremendous pressure to deliver, not maintain
- Life's three certainties *for software developers*
  - Death, Taxes, and Changes in Requirements
- There is a middle ground for planning for change

How will change occur

- First, anticipate that changes will occur
- Consider *where* they will change, rather than the exact nature of the changes
- These issues will come up in the Ecommerce case study

What is variable in the design?

- Consider what is variable in your design
  - Instead of focusing on what might force a change to your design
    - Consider what you might want to change
    - Encapsulate the concept that varies
      - this is a theme of many design patterns
- Hopefully there are long term benefits without a lot of extra work up front

OO Design Patterns Used

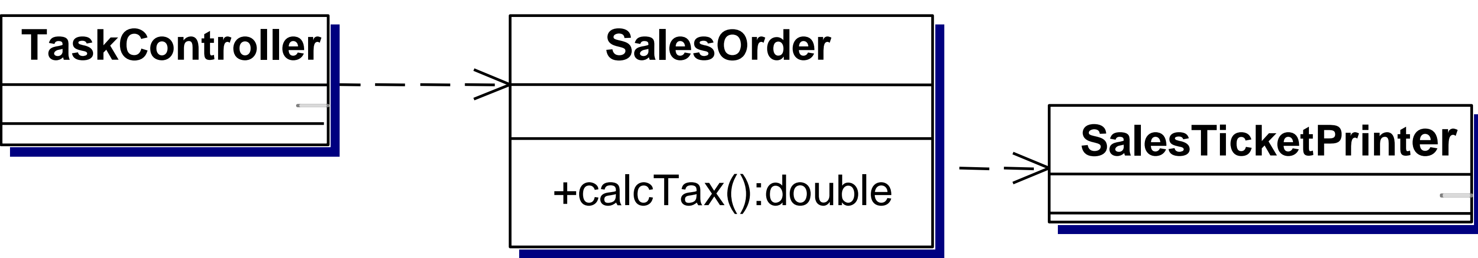
- In the upcoming case study, these design patterns will help make for a system that is good design
  - Strategy
  - Singleton
  - Decorator
  - Observer
- We've considered all four

An Ecommerce System

- There is a TaskController object that handles sales requests over the internet
- When the sales order is requested, the controller delegates to a SalesOrder object

Assign Responsibilities

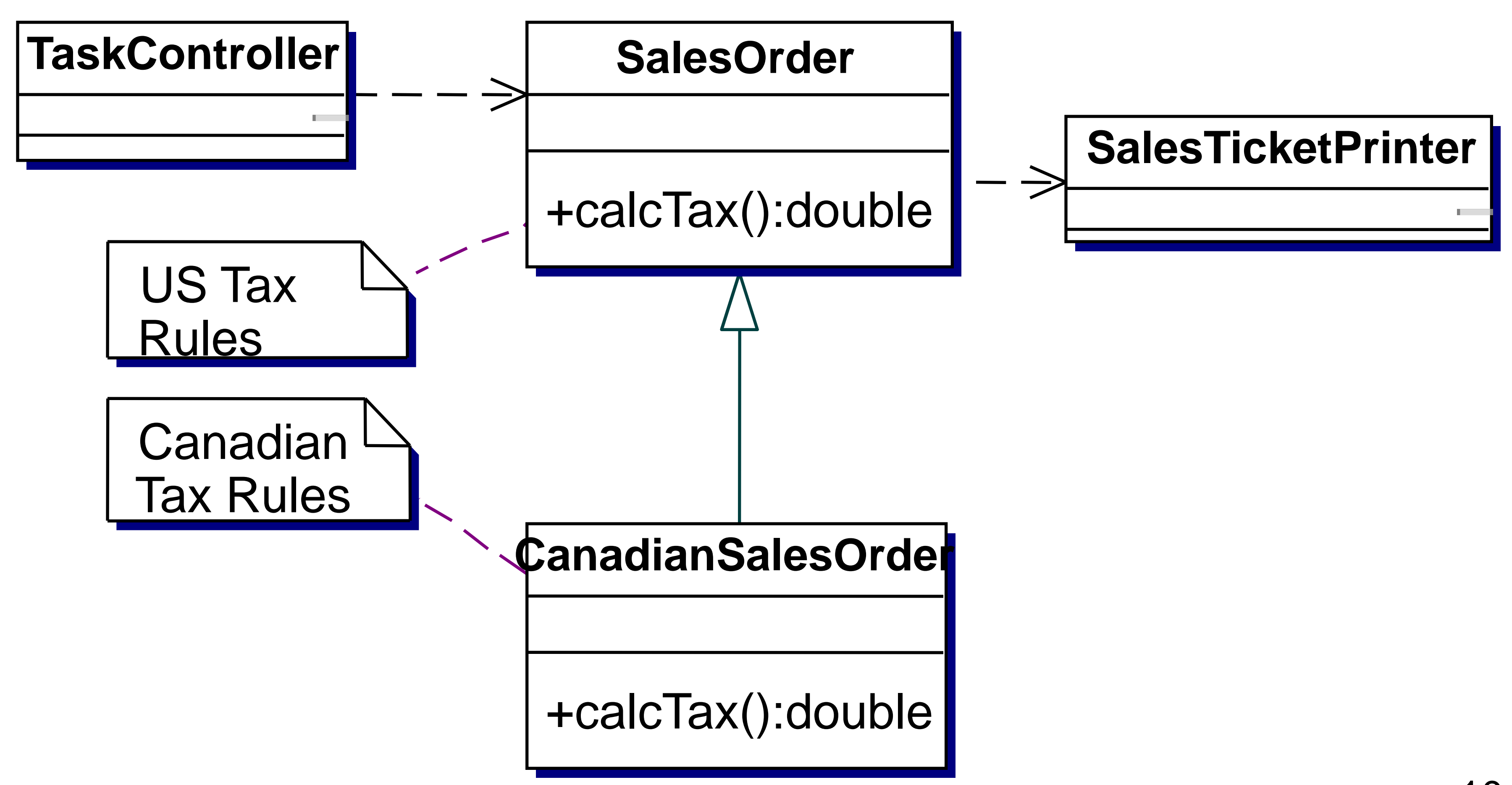
- SalesOrder responsibilities:
  - Allow users to make an order using GUI input
  - Process the order
  - Print a sales receipt



Changing Requirements

- Start charging taxes on order from customers
- need to add rules for taxation, but how?
  - modify existing SalesOrder to handle U.S. taxes
  - extend the existing SalesOrder object and modify the tax rules so it applies to the new country
    - This is an inheritance solution

Subclassing Solution



Favor Composition Over Inheritance

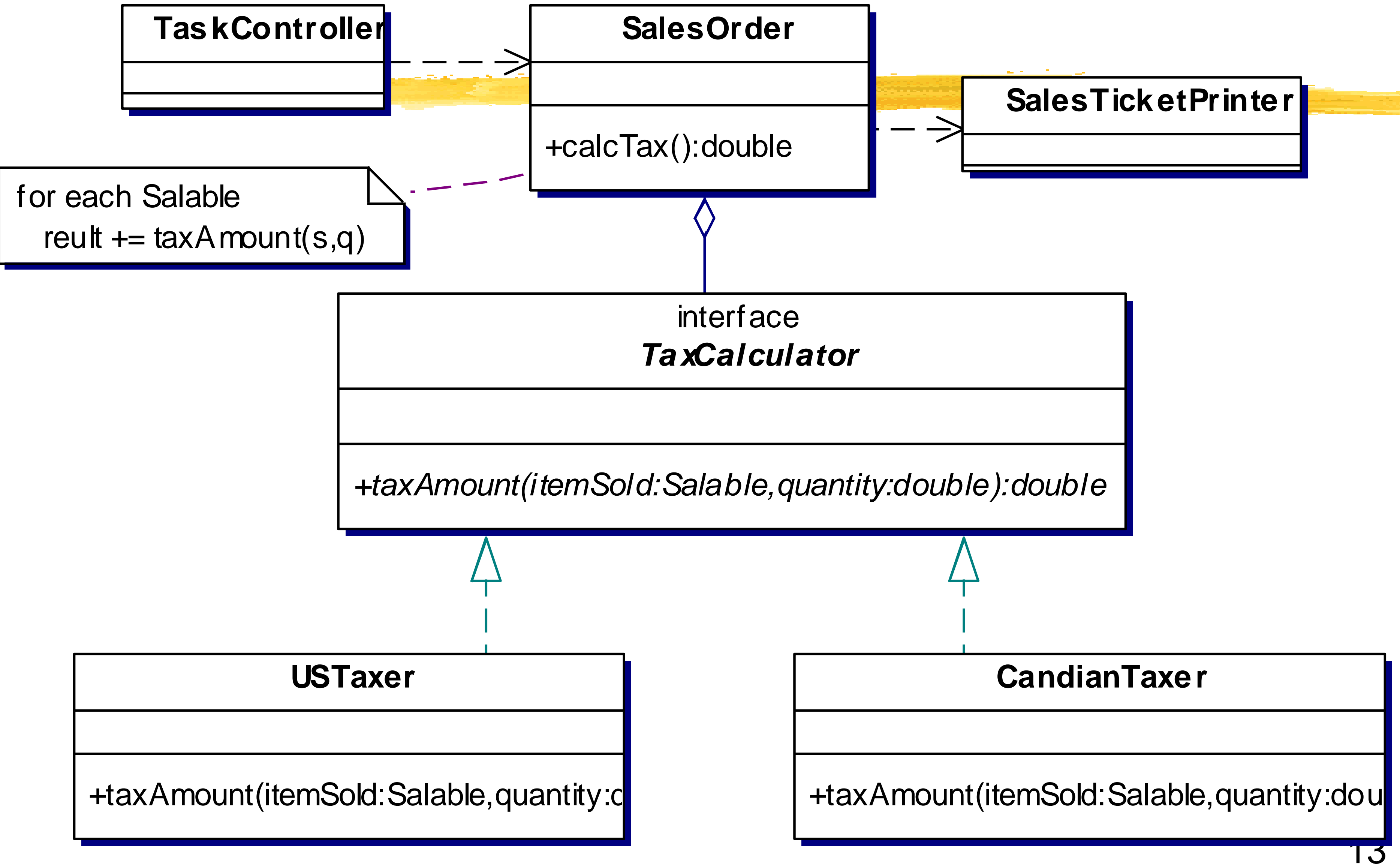
- Design pattern theme of composition over inheritance is ignored in previous design
- Here is a different approach
  - consider what is variable in the design
  - encapsulate the concept the varies
- Accept the fact that tax rules vary country to country and state to state and county to county, and sometimes city to city (like in Arizona) *and they change*

Alternate Designs

- Or use an abstract class with an abstract double calcTax() and many classes in a hierarchy
- Or design an interface to be implemented by different classes using different algorithms
  - Then have SalesOrder contain a reference to the correct object (composition over inheritance)

```
public interface TaxCalculator {
  // A Salable object knows price and how it is taxed
  public double taxAmount(Salable itemSold,
                          double quantity);
}
```

A Better Design with Strategy



Why does Strategy make this design better?

- Better Cohesion (hangs together)
  - sales tax details are in its own class
- Easy to add tax rules from different countries
- Easier to shift responsibilities
  - In the first design where CanadianSalesOrder extends USSalesOrder, only TaskController is able to determine which type of sales order to use
  - With Strategy, either TaskController or SalesOrder could set the TaxCalculator

Determine What Varies

- What Varies?
  - The business rules for taxation
- Current design handles variations at least as well as the other design design
- Current design will handle future variations as well
- A family of tax calculation algorithms have been encapsulated as objects, they are interchangeable,
  - Strategy pattern applied in an Ecommerce system

Using the Strategy Pattern

- What happens when EnglishTaxer is added
  - In England, old-age pensioners are not required to pay taxes on sales items
- How can this be handled?
  1) Pass age of the Customer to TaxCalculator object
  2) Be more general and pass a Customer object
  3) Be even more general and pass a reference to the SalesOrder object (this) to the TaxCalculator and let that EnglishStrategy object ask SalesOrder for customer age (post some html to the client)

Is this change bad?

- To handle this new requirement, SalesOrder and TaxCalculator have to be modified
  - But the change is small and certainly doable
  - Not likely to cause a new problem
- If a Strategy needs more information, pass the information to the object as an argument
  - Some objects may ignore the extra parameter
- Strategy can be applied anywhere you hear this
  - "At different times, different rules apply"

Singleton Pattern

- **Singleton** Ensure a class only has one instance and provide a global point of access to it
- The singleton pattern works by having a special method that is used to instantiate the object
  - when called, the method checks to see if the object has already been instantiated
  - it returns the singleton if instantiated or constructs a new one if this is the first call to get the instance
  - to guarantee this, have a private constructor

Using Singleton

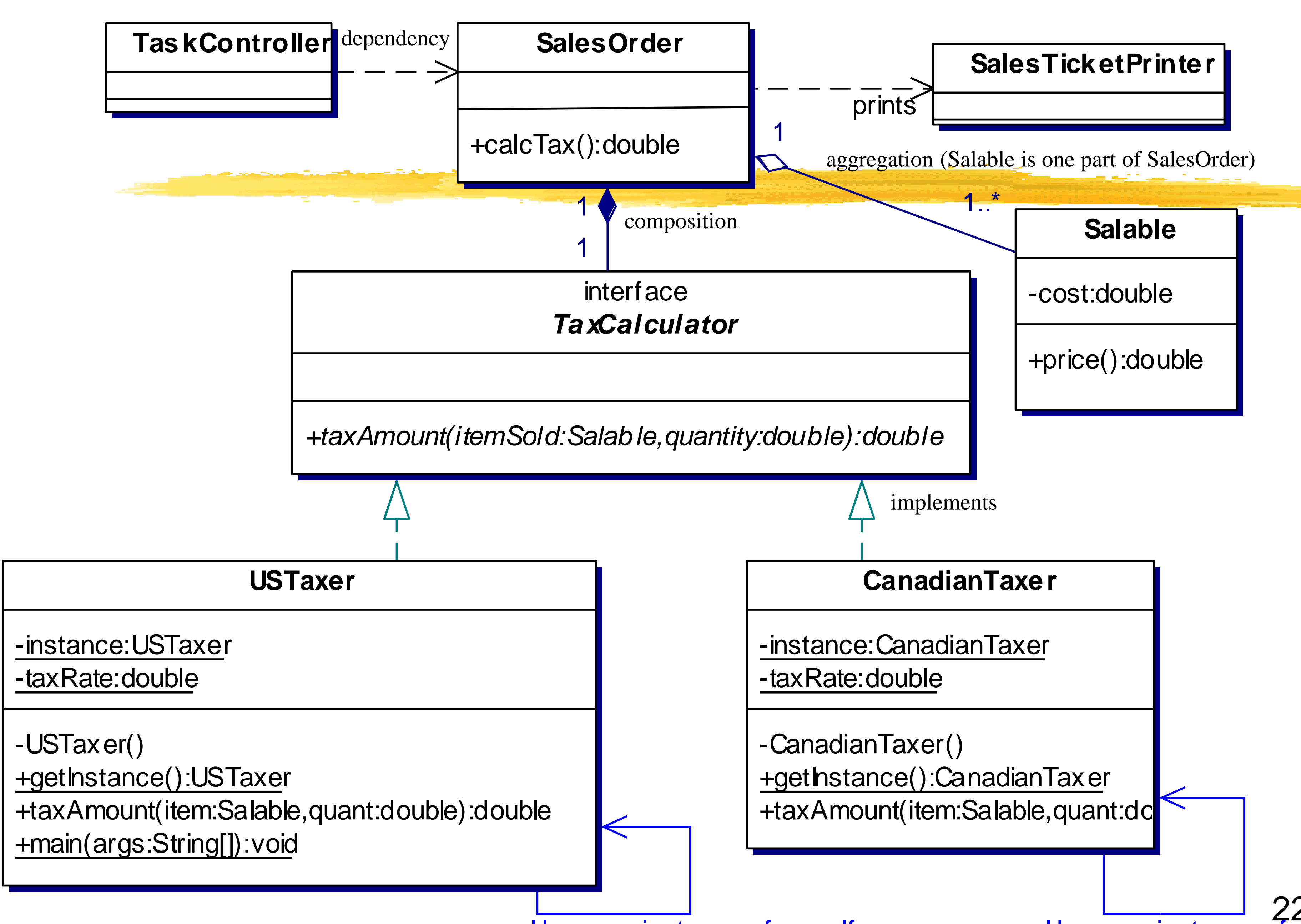
- TaxCalculators are currently encapsulated as Strategy objects
  - How many USTaxer objects are required in this system? How many CanadianTaxers?
- Forces:
  - The same object is being used over and over again
  - More efficient to avoid instantiating them and throwing them away again and again
  - Doing all at once could be slow to start up
    - Could instantiate these objects as needed

Only want one when needed

- Don’t need more than one instance of each TaxCalculator class
- Solution:
  - Let Strategy objects handle the instantiation
  - Let there be only one instance
  - Don’t concern clients (SalesOrder) over this detail
  - In other words, use the Singleton design pattern

USTaxer is now a Singleton

```
public class USTaxer implements TaxCalculator {
  private static USTaxer instance;  // Only one
  private static double taxRate;

  private USTaxer() {
    taxRate = 0.06;  // greatly simplified
  }

  public static USTaxer getInstance() {
    if (instance == null)
      instance = new USTaxer();
    return instance;
  }

  public double taxAmount(Salable item, double quan) {
    return 0; // TODO: Implement tax algorithm
  }
}
```


TaskController
dependency
SalesOrder
+calcTax():double
SalesTicketPrinter
prints
1
aggregation (Salable is one part of SalesOrder)
1..*
Salable
-cost:double
+price():double
1
composition
1
interface
TaxCalculator
+taxAmount(itemSold:Salable,quantity:double):double
implements
USTaxer
-instance:USTaxer
-taxRate:double
-USTaxer()
+getInstance():USTaxer
+taxAmount(item:Salable,quant:double):double
+main(args:String[]):void
CanadianTaxer
-instance:CanadianTaxer
-taxRate:double
-CanadianTaxer()
+getInstance():CanadianTaxer
+taxAmount(item:Salable,quant:do
I have an instance of myself

Aggregation vs. Composition

Definitions from the Unified Modeling Language Guide

- **Aggregation** A special form of association that specifies a whole/part relationship between the aggregate (the whole) and a component (the part)
  - When a class has an instance variable
- **Composition** A form of aggregation with strong ownership. Once a component is created, its lives and dies with its whole
  - A TaxCalculator object is only necessary with a SalesOrder *not used elsewhere*

Other Patterns applied

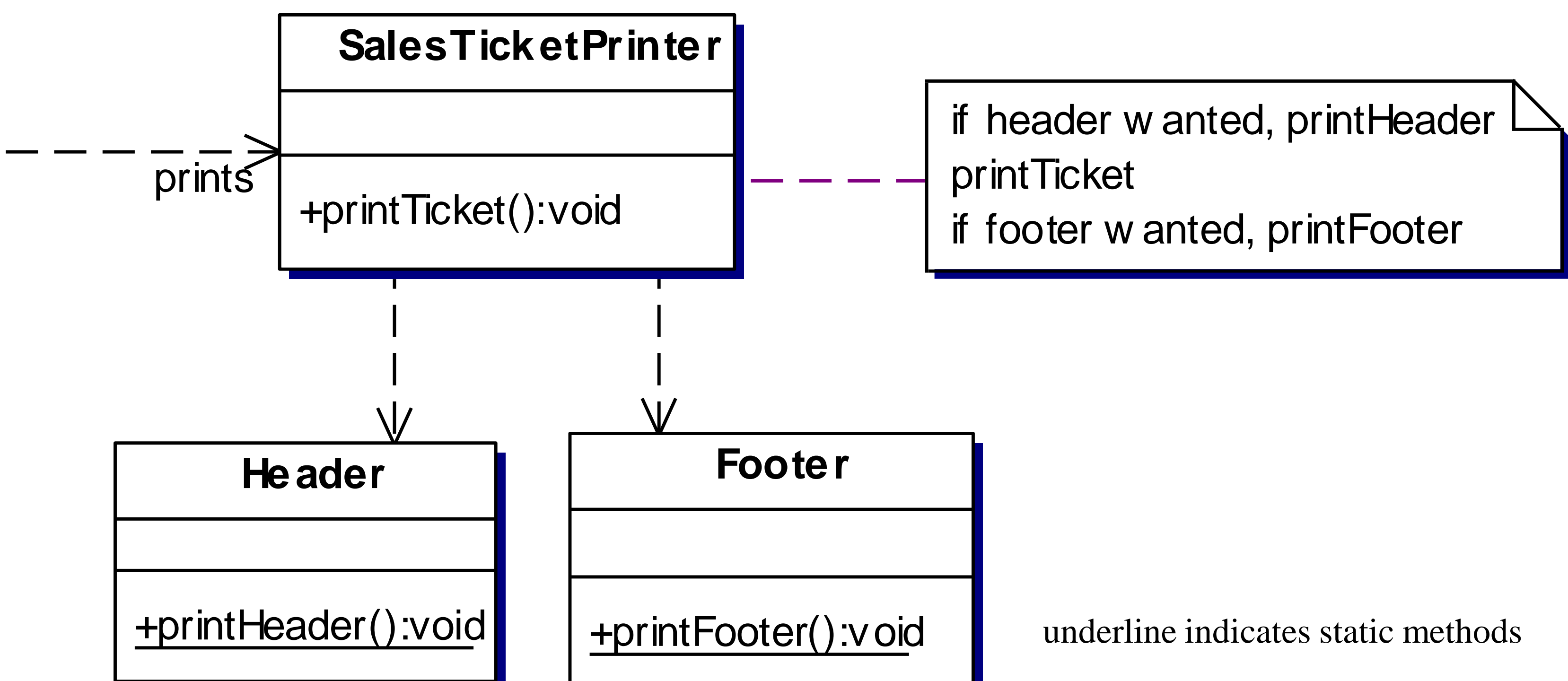
- In the Ecommerce system, we will now
  - “Decorate” a SalesTicket and
  - “Observe” a Customer

Decorate SalesTicketPrinter

- Assume the SalesTicketPrinter currently creates an html sales receipt *Airline Ticket*
- New Requirement: Add header with company name, add footer that is an advertisement, during the holidays add holiday relevant header(s) and footer(s), we're not sure how many
- One solution
  - Place control in SalesTicketPrinter
    - Then you need flags to control what header(s) get printed

One Solution

- This works well if there are few header and footer options *or perhaps just add a few private helper methods*

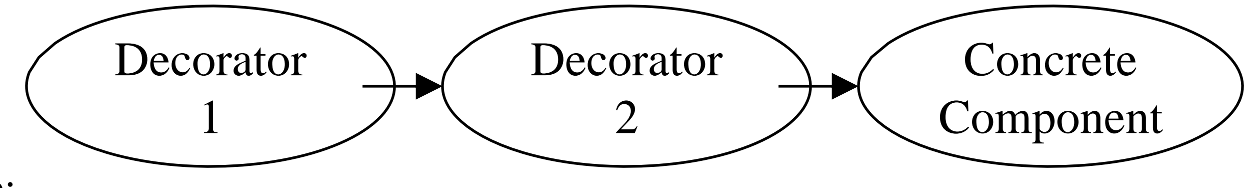


Strategy Pattern?

- If there are many types of headers and footers, with only one being printed each time, use Strategy
- If there are more than one header and footer, and the ordering changes, and the number of combinations grows,
  - use the Decorator design pattern to chain together the desired functionality in the correct order needed

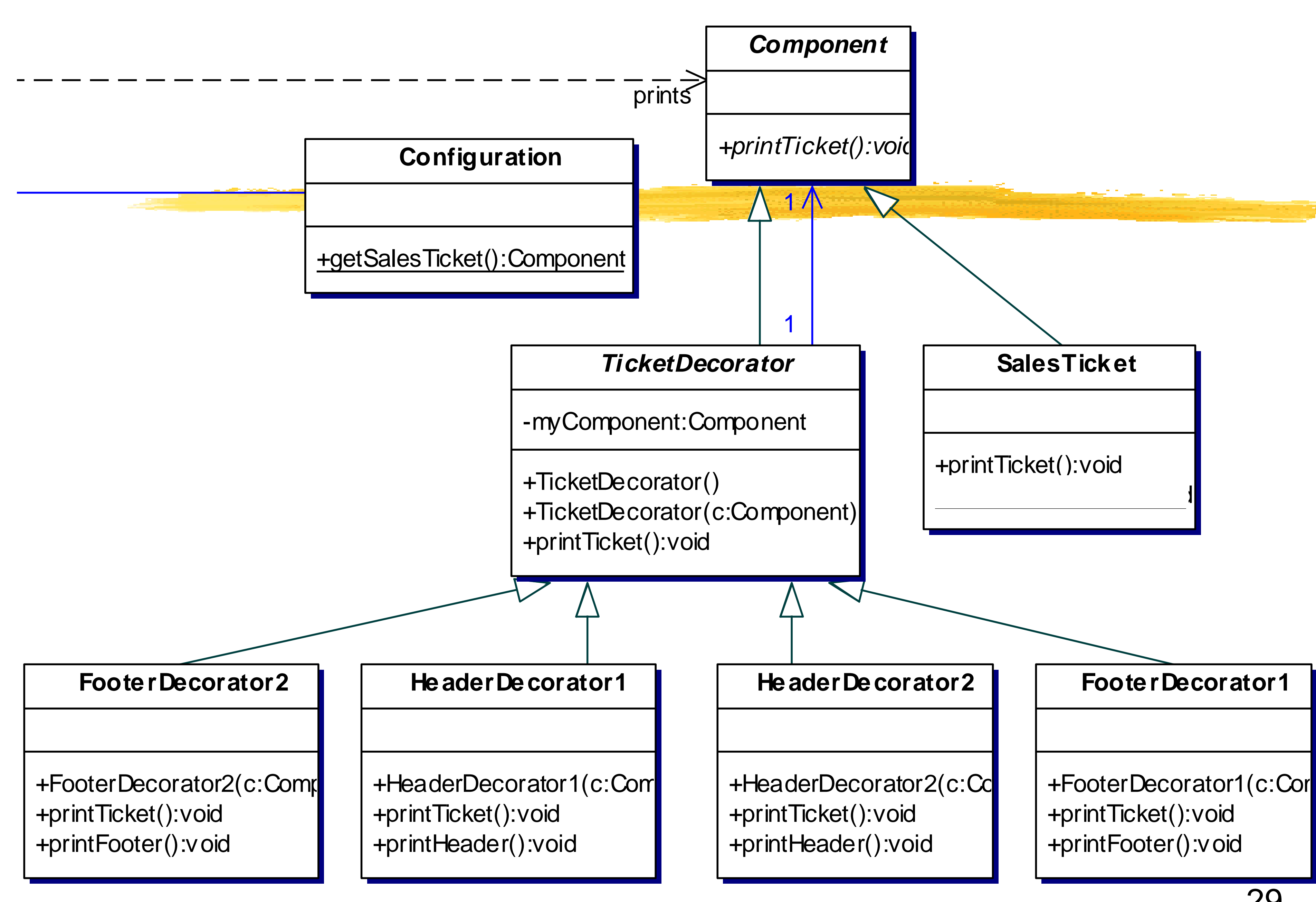
Decorator Again

- Decorator *summary repeated* Attach additional Responsibilities to an object dynamically
- GoF book states: Decorators provide a flexible alternative to subclassing for functionality
- Start chain with decorators, end with original object



Example:

```
keyboard = new BufferedReader(
              new InputStreamReader(
                   System.in));
```


Component
prints
+printTicket():void
Configuration
+getSalesTicket():Component
1
1
TicketDecorator
-myComponent:Component
+TicketDecorator()
+TicketDecorator(c:Component)
+printTicket():void
SalesTicket
+printTicket():void
FooterDecorator2
+FooterDecorator2(c:Comp
+printTicket():void
+printFooter():void
HeaderDecorator1
+HeaderDecorator1(c:Com
+printTicket():void
+printHeader():void
HeaderDecorator2
+HeaderDecorator2(c:Co
+printTicket():void
+printHeader():void
FooterDecorator1
+FooterDecorator1(c:Co
+printTicket():void
+printFooter():void

A Simple SalesTicket

```
abstract class Component {
  abstract public void printTicket();
}

// Instances of this class are sales tickets
// that may be decorated
class SalesTicket extends Component {
  @Override
  public void printTicket() {
    // Hard coded here, but simpler than
    // adding a new Customer class . . .
    System.out.println("Customer: Kim");
    System.out.println("The sales ticket itself");
    System.out.println("Total: $123.45");
  }
}
```

TicketDecorator

```
abstract class TicketDecorator extends Component {
  private Component myComponent;

  public TicketDecorator() {
    myComponent = null;
  }

  public TicketDecorator(Component c) {
    myComponent = c;
  }

  @Override
  public void printTicket() {
    if (myComponent != null)
      myComponent.printTicket();
  }
}
```

A Header Decorator

```
class HeaderDecorator1 extends TicketDecorator {
  public HeaderDecorator1(Component c) {
    super(c);
  }

  @Override
  public void printTicket() {
    this.printHeader();
    super.printTicket();
  }

  public void printHeader() {
    System.out.println("@@ Header One @@");
  }
}
```

A Footer Decorator

```
class FooterDecorator1 extends TicketDecorator {
  public FooterDecorator1(Component c) {
    super(c);
  }

  @Override
  public void printTicket() {
    super.printTicket();
    this.printFooter();
  }

  public void printFooter() {
    System.out.println("%% FOOTER one %%");
  }
}
```

A Client

```
public class Client {

  public static void main(String[] args) {
    Component myST = Configuration.getSalesTicket();
    myST.printTicket();
  }
}
```

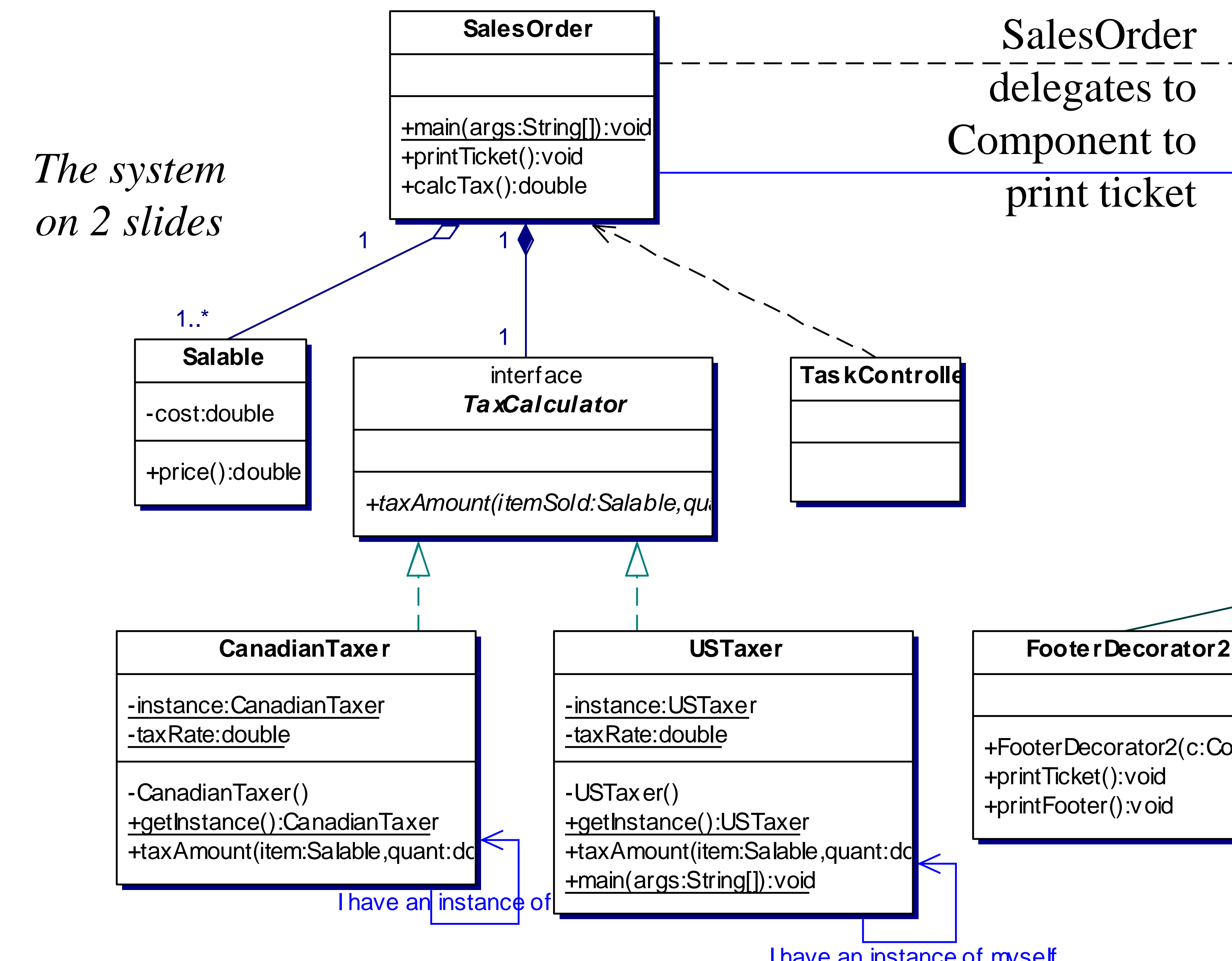
Simple Configuration

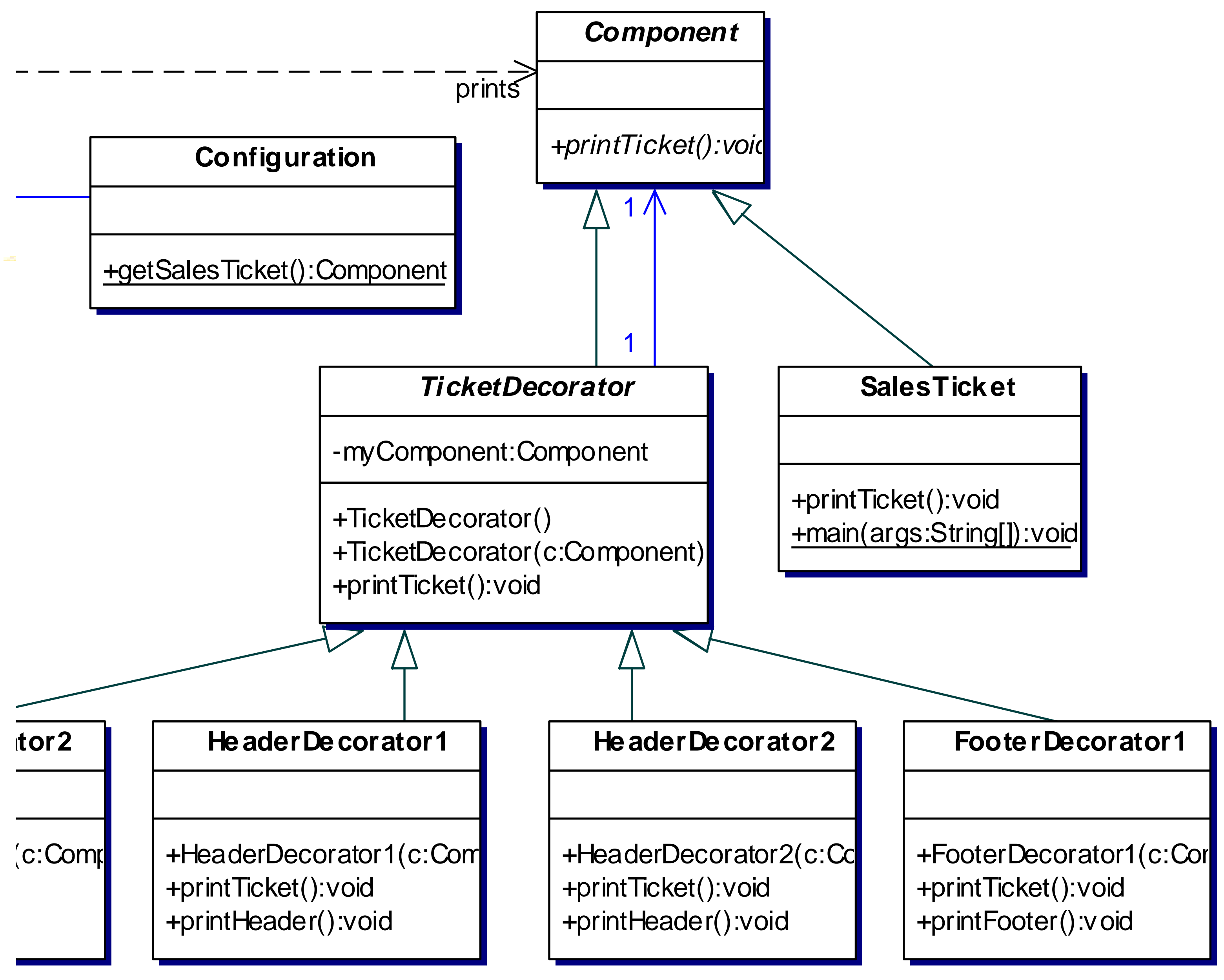
```
// This method determines how to decorate SalesTicket
class Configuration {

  public static Component getSalesTicket() {
    // Return a decorated SalesTicket
    return
        new HeaderDecorator1(
          new HeaderDecorator2(
             new FooterDecorator2(
               new FooterDecorator1(
                     new SalesTicket()
                                  )
                                )
                              )
                            )
                         ;
  }
}
```

Output with Current Configuration

Output:

```
@@ Header One @@
>> Header Two <<
Customer: Bob
The sales ticket itself
Total: $123.45
%% FOOTER one %%
## FOOTER two ##
```

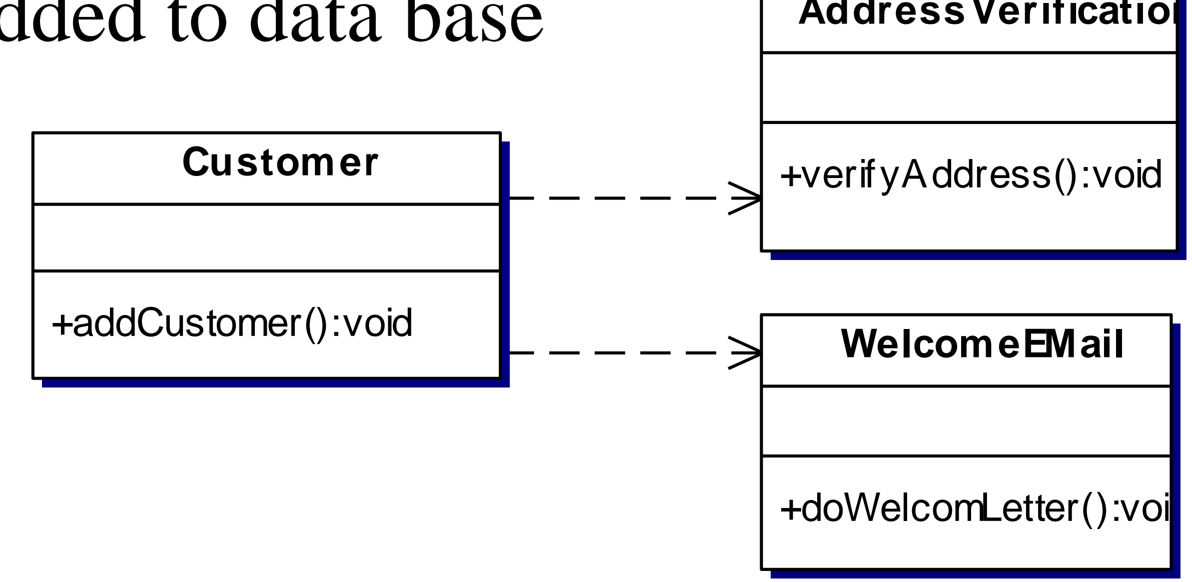

SalesOrder
+main(args:String[]):void
+printTicket():void
+calcTax():double
SalesOrder delegates to Component to print ticket
The system on 2 slides
1
1
1..*
1
Salable
-cost:double
+price():double
interface
TaxCalculator
+taxAmount(itemSold:Salable,qu
TaskControlle
CanadianTaxer
-instance:CanadianTaxer
-taxRate:double
-CanadianTaxer()
+getInstance():CanadianTaxer
+taxAmount(item:Salable,quant:do
I have an instance of
USTaxer
-instance:USTaxer
-taxRate:double
-USTaxer()
+getInstance():USTaxer
+taxAmount(item:Salable,quant:do
+main(args:String[]):void
I have an instance of myself
FooterDecorator2
+FooterDecorator2(c:Co
+printTicket():void
+printFooter():void



Component
prints
+printTicket():void
Configuration
+getSalesTicket():Component
1
1
TicketDecorator
-myComponent:Component
+TicketDecorator()
+TicketDecorator(c:Component)
+printTicket():void
SalesTicket
+printTicket():void
+main(args:String[]):void
tor2
(c:Comp
HeaderDecorator1
+HeaderDecorator1(c:Com
+printTicket():void
+printHeader():void
HeaderDecorator2
+HeaderDecorator2(c:Co
+printTicket():void
+printHeader():void
FooterDecorator1
+FooterDecorator1(c:Co
+printTicket():void
+printFooter():void

Observe Customer

- New Requirements: Send an email to a new customer and verify the customer's address with the post office
- If this was it, hard code Customer behavior when being added to data base



Or Use Observer

- With additional behaviors (such as send advertisements via snail mail), there may be a changing list of objects that need notification that a new customer is being added
- These objects will have different interfaces
  - SendEmail, SendCouponsViaSnailMail, SellPrivateInformationToTelemarketers, ...
- Next up: change two objects into "Observers"

Observer

- Have Customer extend Observable
- Have all of the objects that need notification implement Observer (all have the update method)
- Have some configurer add the correct observers to the Customer object with addObservers
- Have the addCustomer method send the message notifyObservers

Design with Observer