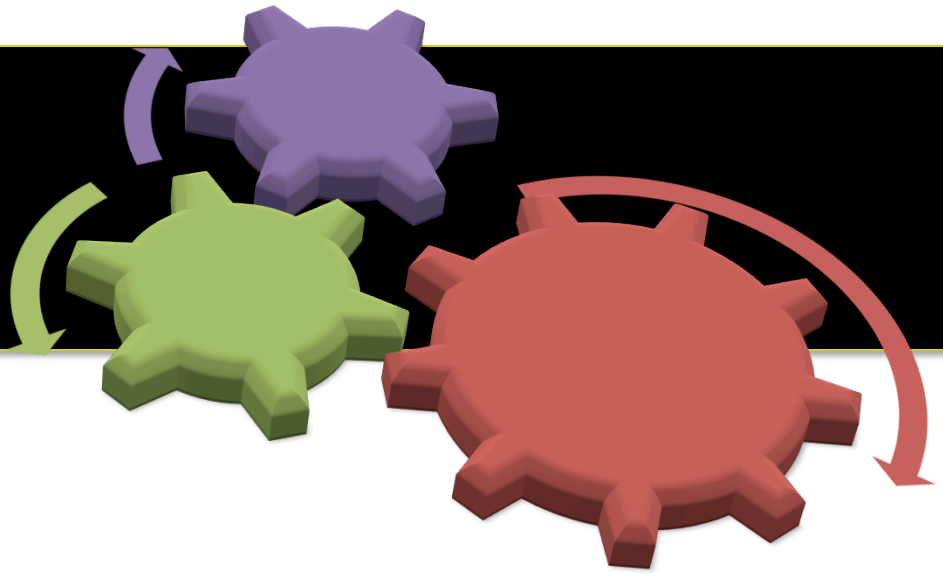
Design Patterns

CSCI-4448 - Boese

University of Colorado **Boulder**

Overview



What is a Design Pattern?

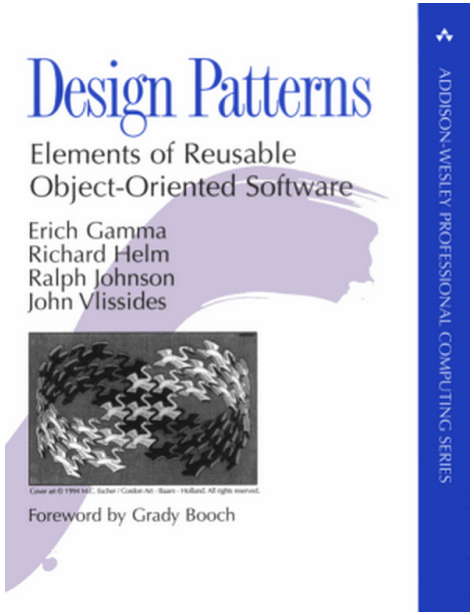
"A solution to a problem in a context."

- Patterns are best-practice solutions for recurring problems in a particular context.
- Can be thought of as:
  Describing the relationships between entities (objects) in our problem domain.
- It is an art form.

Design Patterns

Where do design patterns come from?

- Comes from work of architect Christopher Alexander
  - Studied ways to improve process of designing buildings and urban areas
- GoF
  - *"discovered, not invented"*
- Now used heavily
- Frameworks
  - Techniques that use both design patterns and code



Why?

- Jobs…

Developer **Jobs** in **Boulder, Colorado** - Information Technology **Jobs**
information-technology.thingamajob.com › ... › Colorado
Jan 23, 2014 - View and apply for Developer jobs in **Boulder**, **Colorado** at thingamajob.com. **...** Developer **Job Description**: Our client is looking for ... Knowledgeable of object-oriented principles and development patterns like **MVC** - Portfolio **...**

PHP Developer- **Jobs**, **Employment** in **Boulder, CO** | Indeed.com
www.indeed.com/**jobs**?q=PHP...l=**Boulder**,+**CO** Indeed.com
Jan 3, 2014 - 144 PHP Developer- Jobs available in **Boulder**, **CO** on Indeed.com. one search. all jobs. **... Job Description**: LAMP Developer U.S. Citizen and... a talented LAMP Developer who can contribute **...** Must have **MVC** Experience.

.NET Developer
National Computing Group - Denver, CO, 80202 - Posted 7 days ago
.NET Developer Development Engineer SUMMARY Develop and support general .NET based applications using the Microsoft technology stack. Essential Skills - 5 Years C# .NET General Experience. Demonstrate mastery of .NET development, process and technology. (Win forms, C#, IIS) - 4 Years MVC Development of web applications - 4 Years WCF/Web Service development - 4 years ASP.NET development - 4 Years J

Killer **Android** Developer
Blackstone Technology Group - Denver, CO
with any **MVC** frameworks such as Spring **MVC**, Struts, other.. Experience with Tomcat ... mailtorwillardbstonetech.com Thank you for your attention and interest. **android**...
30+ days ago from Dice

Qualifications

Required Qualifications:

- 2-5 years of related ex

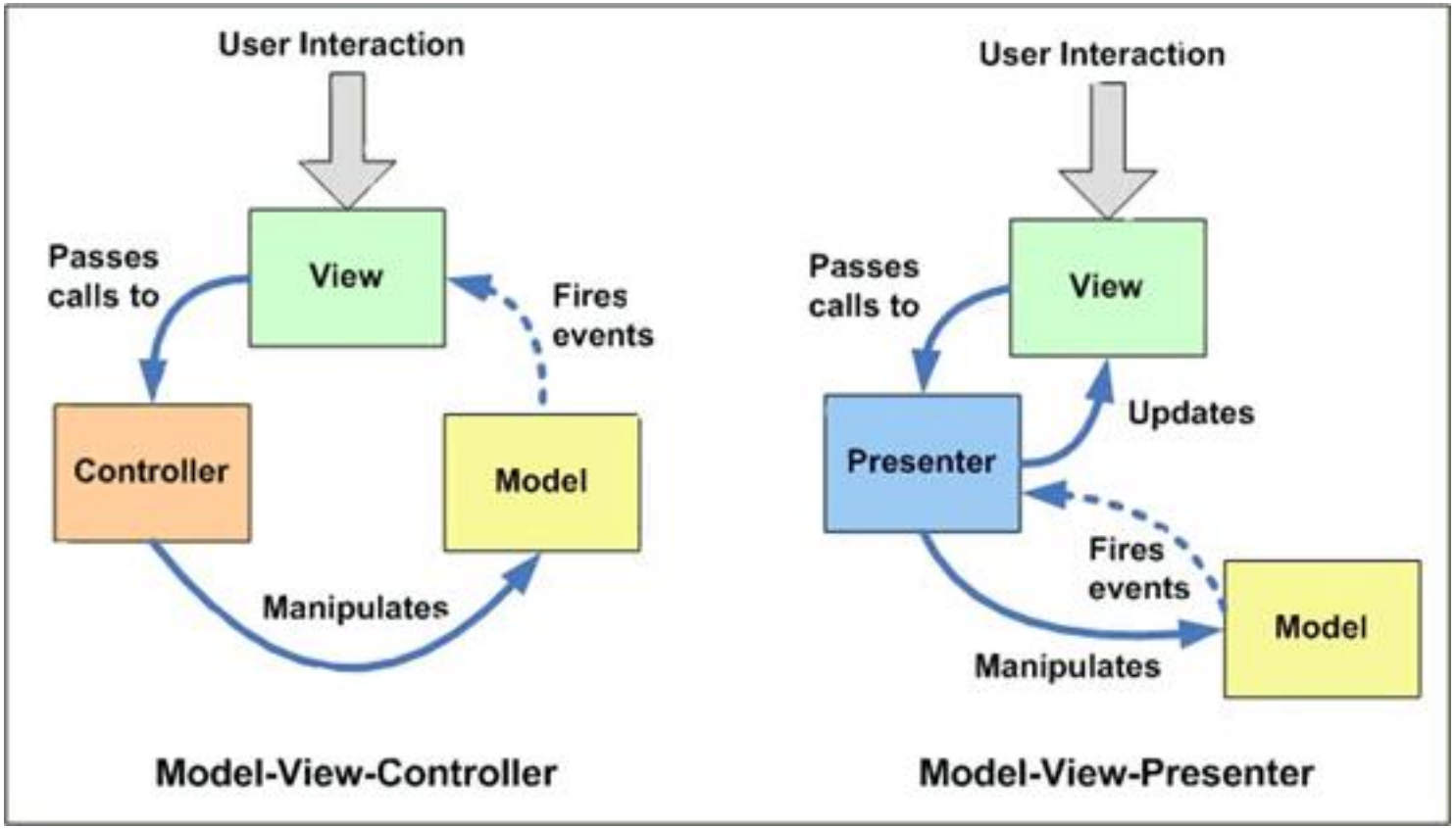
- Thorough working knowledge and experience with one or more of the following: Google Web Toolkit, Oracle, Dependency Injection, Web Services (SOAP and REST), Interfaces (XML/WSDL), Struts, Hibernate, Spring, MVC

Frameworks

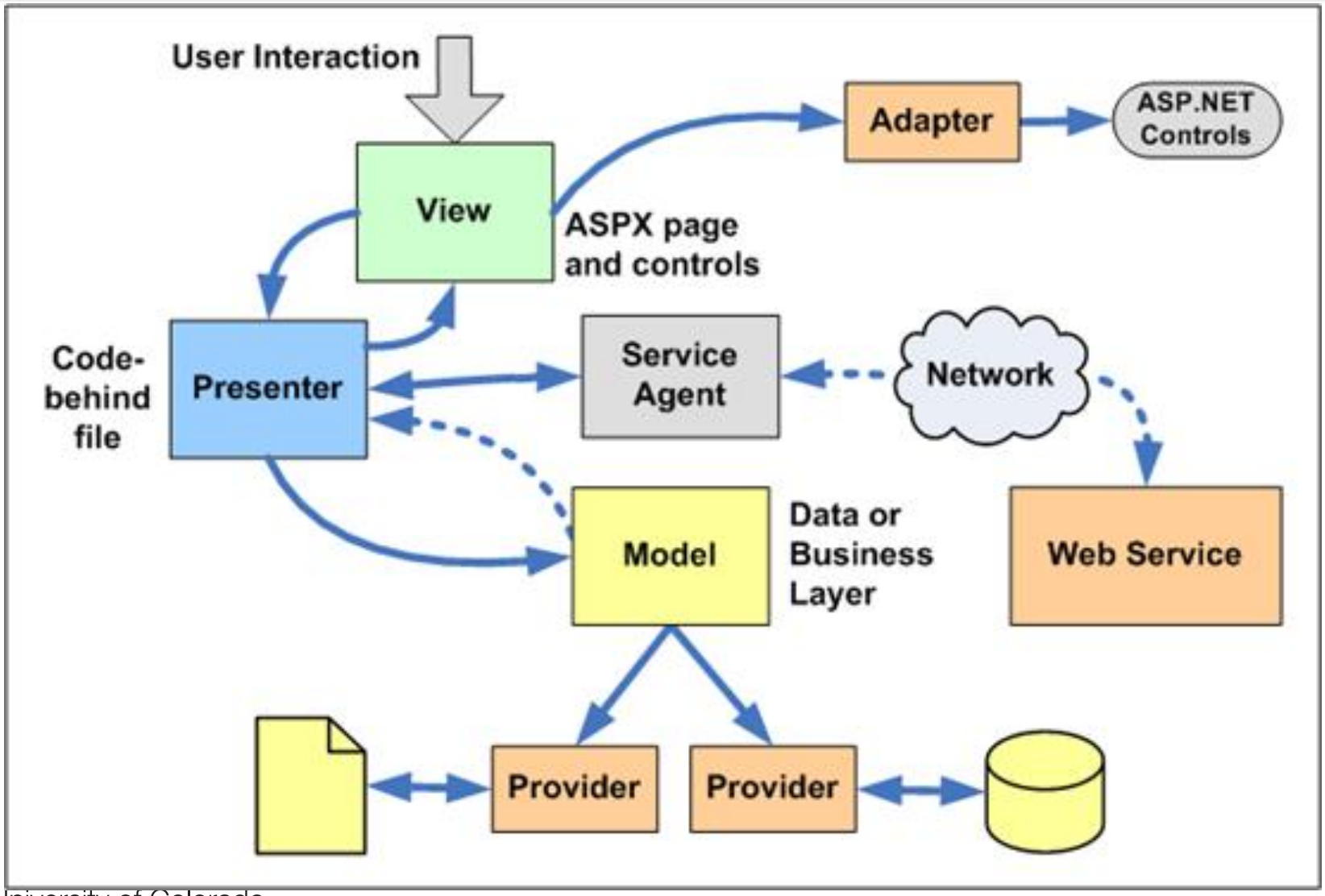
Framework	Language	Score
• ASP.NET	C#	97
• Flask	Python	90
• CodeIgniter	PHP	89
• Zend	PHP	88
• Ruby on Rails	Ruby	87
• AngularJS	JavaScript	85
• Django	Python	81
• Yii	PHP	81
• Symfony	PHP	81
• CakePHP	PHP	81
• Laravel	PHP	78

Frameworks

- .NET

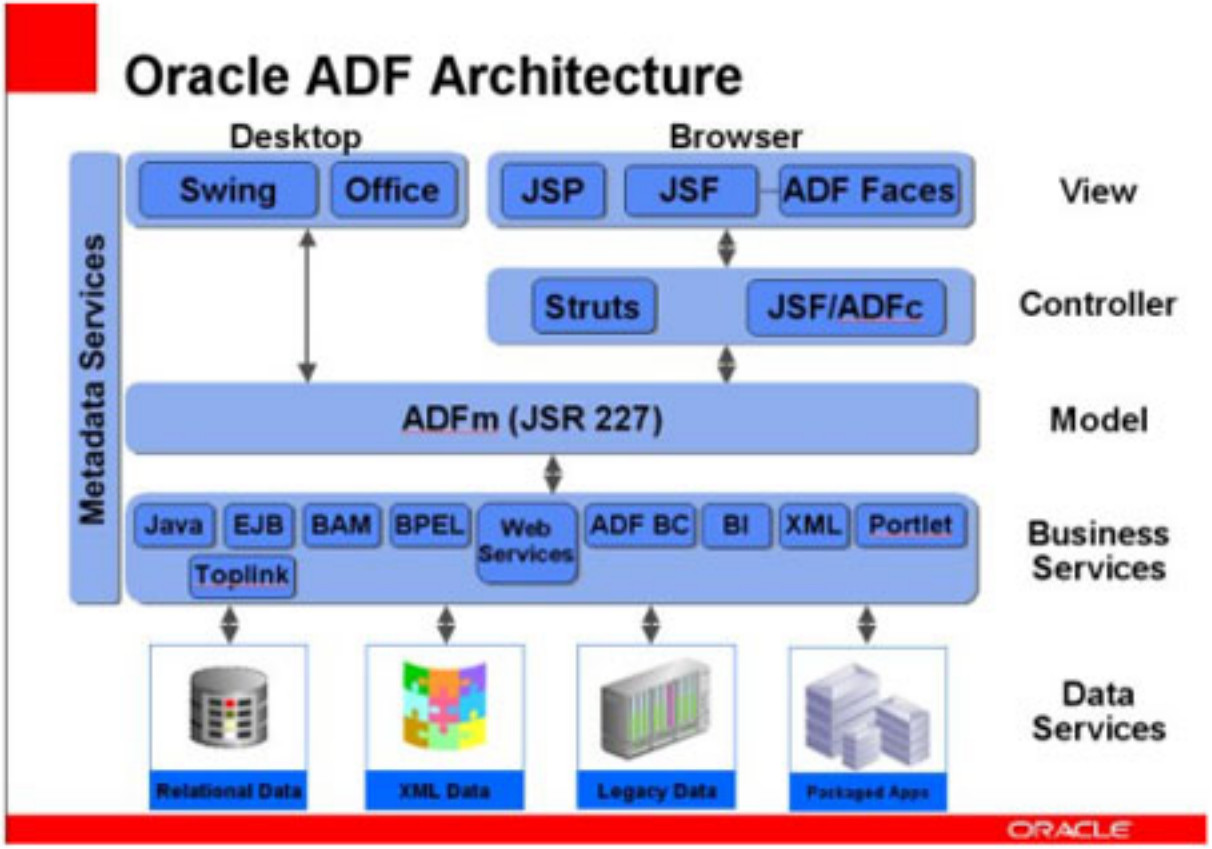


ASP.NET

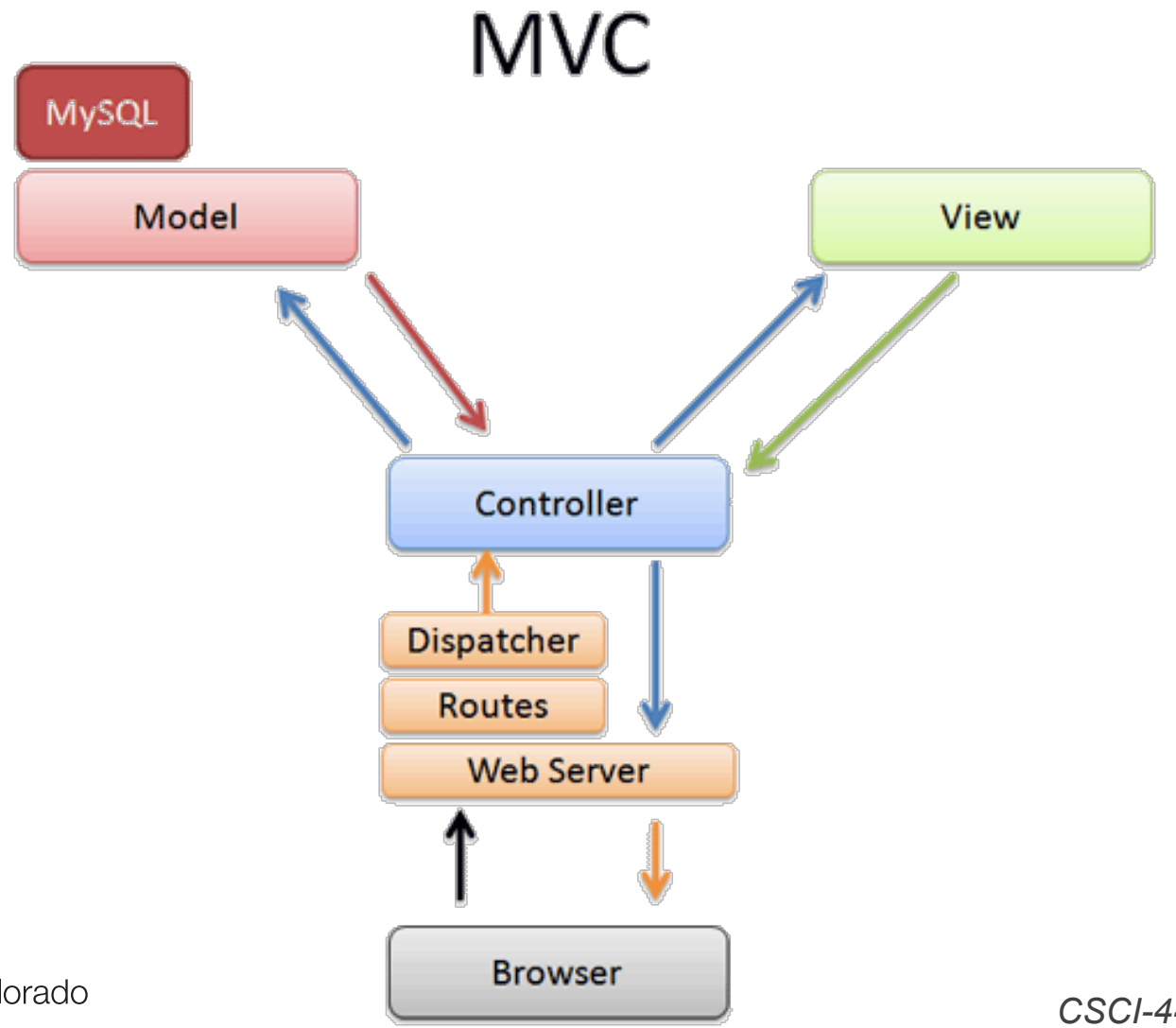

User Interaction
Adapter
ASP.NET Controls
View
ASPX page and controls
Code-behind file
Presenter
Service Agent
Network
Model
Data or Business Layer
Web Service
Provider
Provider

Oracle ADF

- Application Development Framework



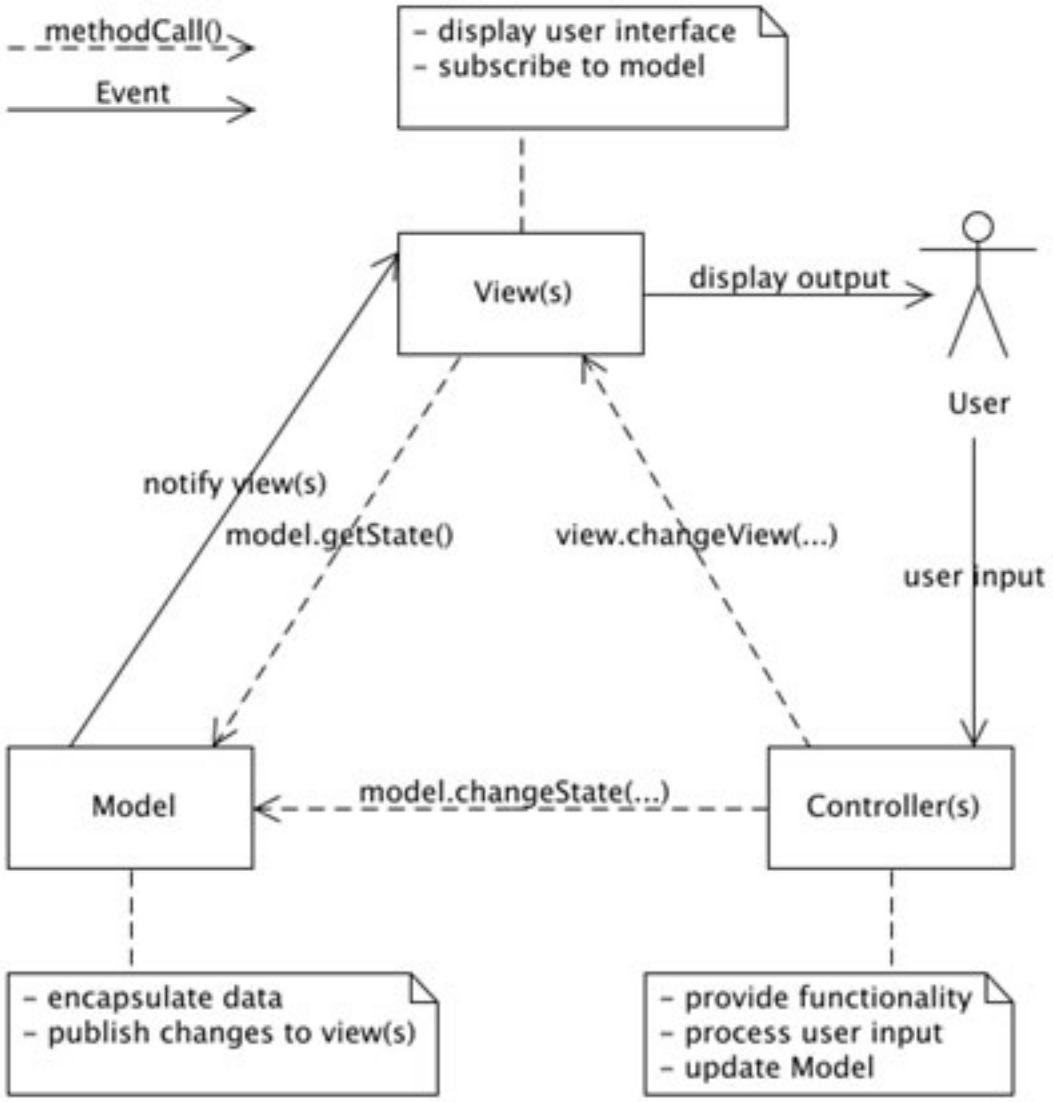
Cake PHP Framework



Java


Java

- J2EE


methodCall()
Event
- display user interface
- subscribe to model
View(s)
display output
User
notify view(s)
model.getState()
view.changeView(...)
user input
Model
model.changeState(...)
Controller(s)
- encapsulate data
- publish changes to view(s)
- provide functionality
- process user input
- update Model

Why?

- "*Designing object-oriented software is hard and designing reusable object-oriented software is even harder.*" - Erich Gamma

- Experienced designers re-use solutions that have worked in the past.

- Well-structured object-oriented systems have recurring patterns of classes and objects.

Customization: Build Custom Objects

Reuse

- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation. The new object with more functionality is an ***aggregation*** of existing objects.
  - Delegate tasks to the aggregate object.
- Inheritance (also called White-box Reuse)
  - New functionality is obtained by ***inheritance.***
  - Access to the development products (models, system design, object design, source code) must be available.

Why?

- Facilitate Communication

Heads First Design Patterns

Why?

Benefits of Design Patterns

- Capture expertise and make it accessible to non-experts in a standard form
- Facilitate communication among developers by providing a common language
- Make it easier to reuse successful designs and avoid alternatives that diminish reusability
- Facilitate design modifications
- Improve design documentation
- Improve design understandability

Types of Patterns

- There are generally 3 types of patterns:
  - Architectural Patterns
  - Design Patterns
  - Programming Patterns

Architectural Patterns

- Fundamental structural organization for software systems
  - Large, overall scope
  - Overall pattern followed by the whole system
- Provides
  - a set of predefined sub-systems,
  - specifies their responsibilities, and
  - includes rules for establishing relationships between them

Architectural Patterns Examples

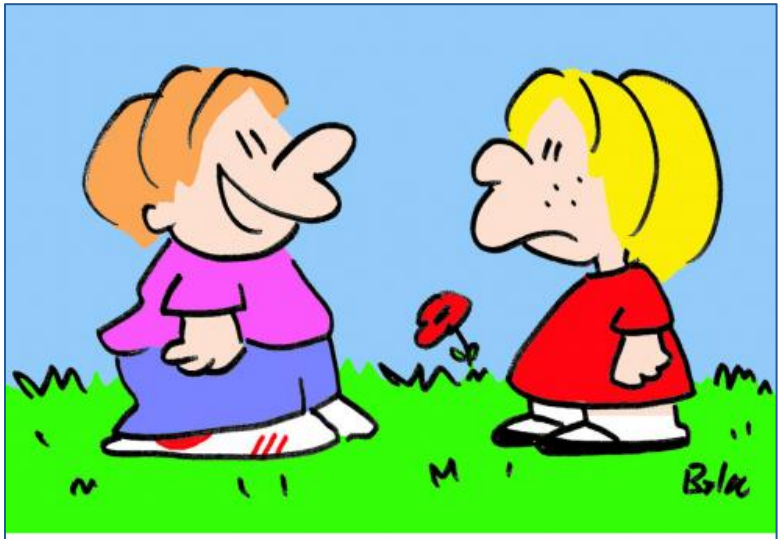
Examples

- 3-tier database systems
  - Database,
  - Intermediate DB Layer,
  - User-Application
- Client/Server
- Component (Module) -Based Software Systems
- Feedback Systems
- Event-Driven Systems

Design Pattern Types

Types

- **Creational Patterns**
  - Vary object creation
- **Structural Patterns**
  - Vary object structure
- **Behavioral Patterns**
  - Vary the interactions/behavior you want



Design Pattern Catalog

Types of Patterns

- There are generally 3 types of patterns:
  - Architectural Patterns
  - Design Patterns
  - Idioms

Creational Patterns

Abstract Factory
Builder
Factory Method
Prototype
Singleton

Structural Patterns

Adapter
Bridge
Composite
Container
Decorator
Façade
Flyweight
Proxy

Behavioral Patterns

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Multiple Dispatch
Observer
State
Strategy
Template Method
Visitor

Programming Patterns

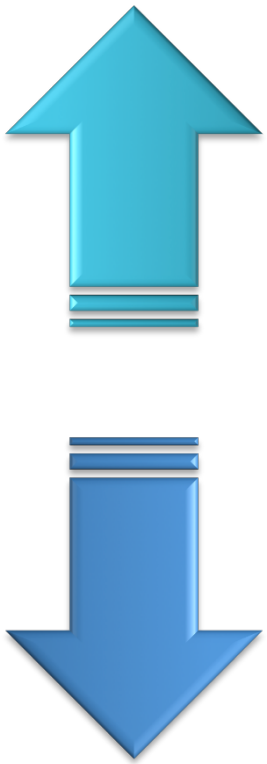
- Aka Programming Idioms
  - Reoccurring solutions to common programming problems
  - Low-level patterns specific to a programming language
  - During implementation phase, look for programming idioms

```
map(operator O, list L)
{
    if (L contains no elements) quit;
    h ← the first element of L.
    apply O to h;
    map(O, L minus h);
}
```

```
public Vector map(Vector l)
{
    Vector result = new Vector();
    Iterator iter = l.iterator();
    while(iter.hasNext())
    {
        result.add(f(iter.next));
    }
    return result;
}
```

Levels of Abstraction

- Complex design for entire application or system.
- Solution to a general design problem in a particular context.
- Simple re-usable design class such as a linked list, hash table, etc.



More Abstract

GoF In the middle

More Concrete

GoF

Gang of Four

GoF Design Patterns

- "*A design pattern*
  - *names,*
  - *abstracts, and*
  - *identifies key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.*"

GoF Classification

- **Purpose** - what a pattern does
  - **Creational Patterns**
    - *Concern the process of object creation*
  - **Structural Patterns**
    - *Deal with the composition of classes and objects*
  - **Behavioral Patterns**
    - *Deal with the interaction of classes and objects*

GoF Classification

- **Scope** - what the pattern applies to
  - **Class Patterns**
    - *Focus on the relationships between classes and their subclasses*
    - *Involve inheritance reuse*
  - **Object Patterns**
    - *Focus on the relationships between objects*
    - *Involve composition reuse*

GoF: Classifications

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

GoF Essential Elements

- **Pattern Name**
  - Concise, meaningful name for a pattern improves communication among developers
- **Problem**
  - What is the problem and context where we would use this pattern?
  - What are the conditions that must be met before this pattern should be used?
- **Solution**
  - A description of the elements that make up the design pattern
  - Emphasizes their relationships, responsibilities and collaborations
  - Not a concrete design or implementation; rather an abstract description
- **Consequences**
  - The pros and cons of using the pattern
  - Includes impacts on reusability, portability, extensibility

GoF Pattern Template (1/3)

- **Pattern Name and Classification**
  - A good , concise name for the pattern and the pattern's type
- **Intent**
  - Short statement about what the pattern does
- **Also Known As**
  - Other names for the pattern
- **Motivation**
  - A scenario that illustrates where the pattern would be useful
- **Applicability**
  - Situations where the pattern can be used

GoF Pattern Template (2/3)

- **Structure**
  - A graphical representation of the pattern
- **Participants**
  - The classes and objects participating in the pattern
- **Collaborations**
  - How to do the participants interact to carry out their responsibilities?
- **Consequences**
  - What are the pros and cons of using the pattern?
- **Implementation**
  - Hints and techniques for implementing the pattern

GoF Pattern Template (3/3)

- **Sample Code**
  - Code fragments for a sample implementation
- **Known Uses**
  - Examples of the pattern in real systems
- **Related Patterns**
  - Other patterns that are closely related to the pattern

Example Problem

The Problem

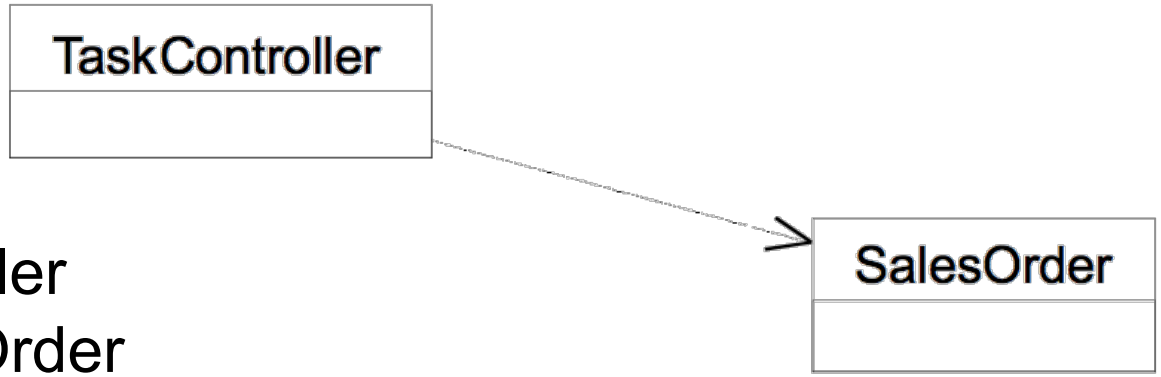
Problem:

- Web-enabled sales order system.
- A customer signs in and fills out the order.

The Problem

Problem:

- Web-enabled sales order system.
- A customer signs in and fills out the order.



Initial Solution:

- Have TaskController
  instantiate SalesOrder
  that handles filling out the sales order, etc.

The Challenge

The Challenge:

- As soon as we get new variations in tax we have to modify our SalesOrder object.

- Where should we add the new responsibility if taxation begins to vary?

New Requirements:

- Multiple Tax Domains
  - US Tax
  - Canadian Tax

Solution 1

New Requirements:

- Multiple Tax Domains
  - US Tax
  - Canadian Tax

Switch on each type

```
method calcTax
// use switch on type of tax rule to be used
//  TYPE US:
//        calc tax based on US rules
//        break
//  TYPE CANADIAN:
//        calc tax based on Canadian rules
//        break
```

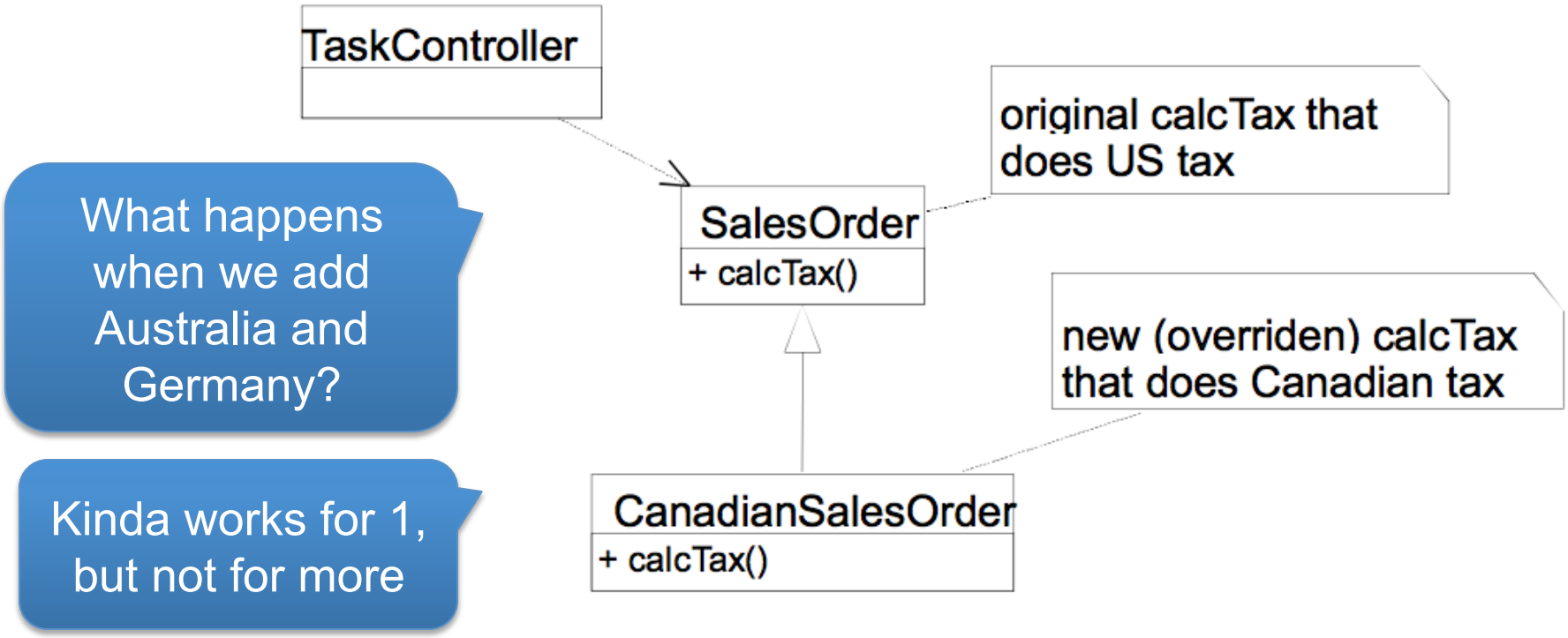
Could work if there is just US and Canada, but what if they expand and need more countries?

Solution 2

New Requirements:

- Multiple Tax Domains
  - US Tax
  - Canadian Tax

We can solve this problem
by specializing (**subclassing**) our first
SalesOrder object to handle the new tax rules

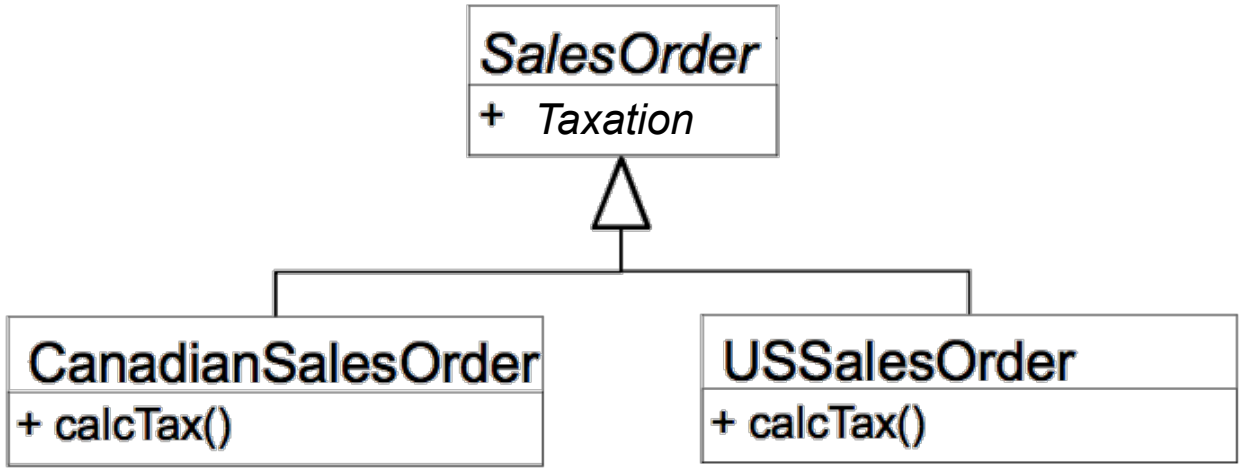


Solution 2b

New Requirements:

- Multiple Tax Domains
  - US Tax
  - Canadian Tax

Modify with
Abstract Inheritance
for specialization.



What happens when we add other rules that vary: date format, language, freight rules…

Leads to redundancy

Leads to hard to understand code

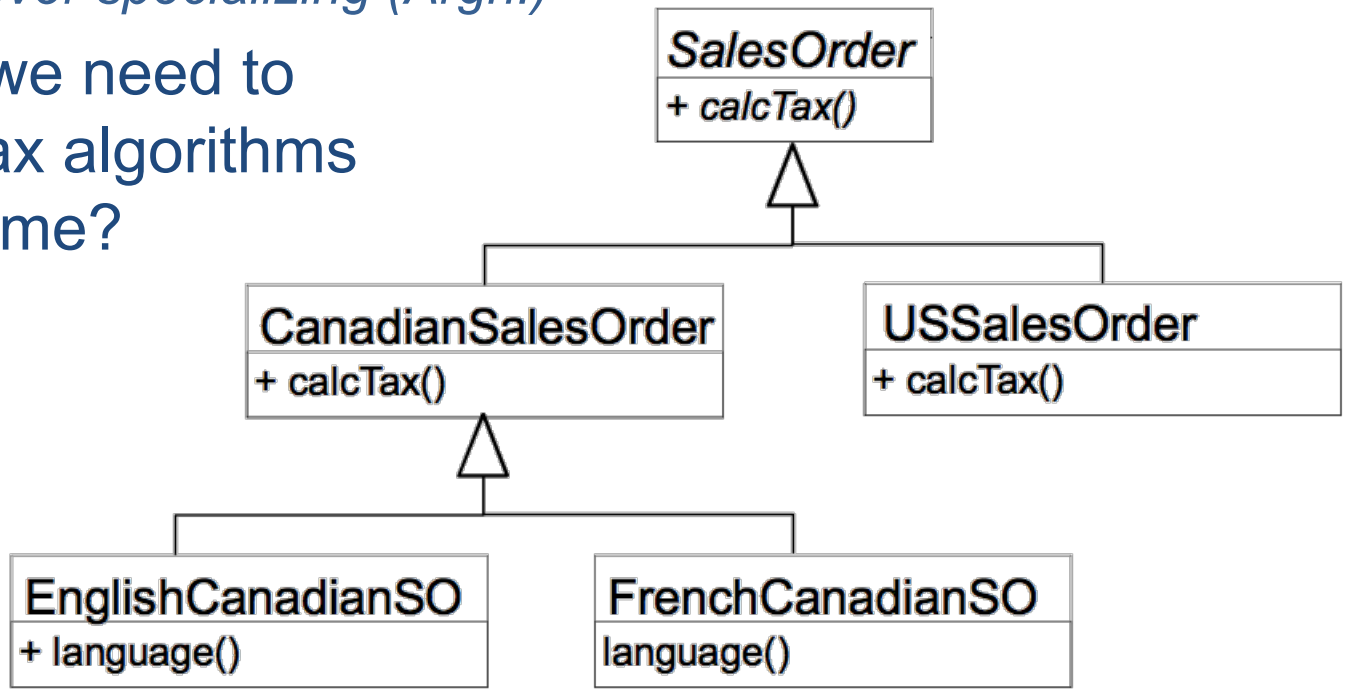
Solution 2b

New Requirements:

- Multiple Tax Domains
  - US Tax
  - Canadian Tax

Leads to maintenance issues.

- If we get new variations, where do we put them?
  - *Lots of switches… (code → hard to read)*
  - *Start over specializing (Argh!)*
- What if we need to switch tax algorithms at run-time?



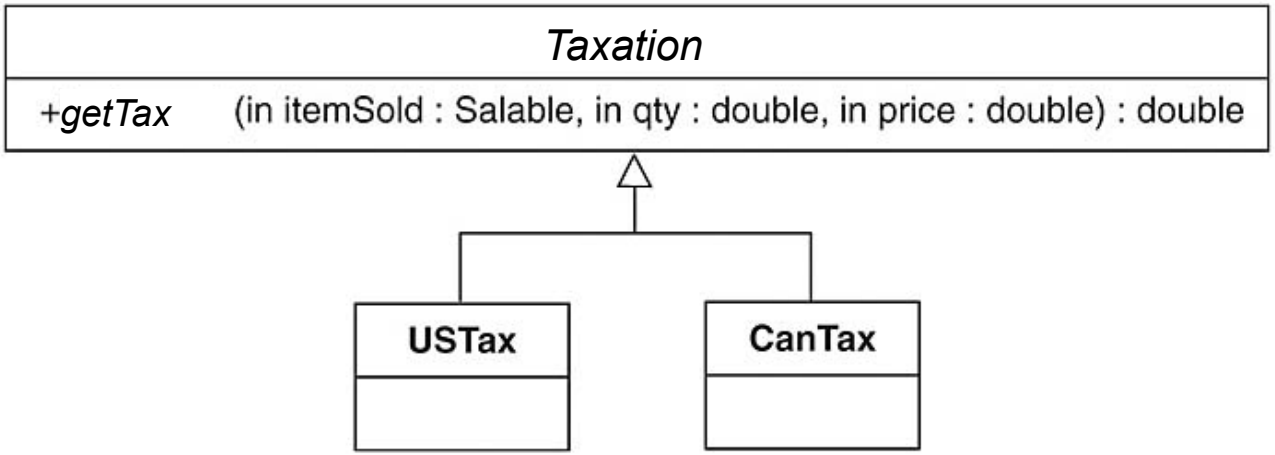
Solution 3

New Requirements:

- Multiple Tax Domains
  - US Tax
  - Canadian Tax

- Encapsulate varying tax rules
  - Create an **abstract class** that defines how to accomplish taxation conceptually
  - Derive concrete classes for each of the variations.

Create a **Taxation** object that defines the interface to accomplish this task. Derive the specific versions needed



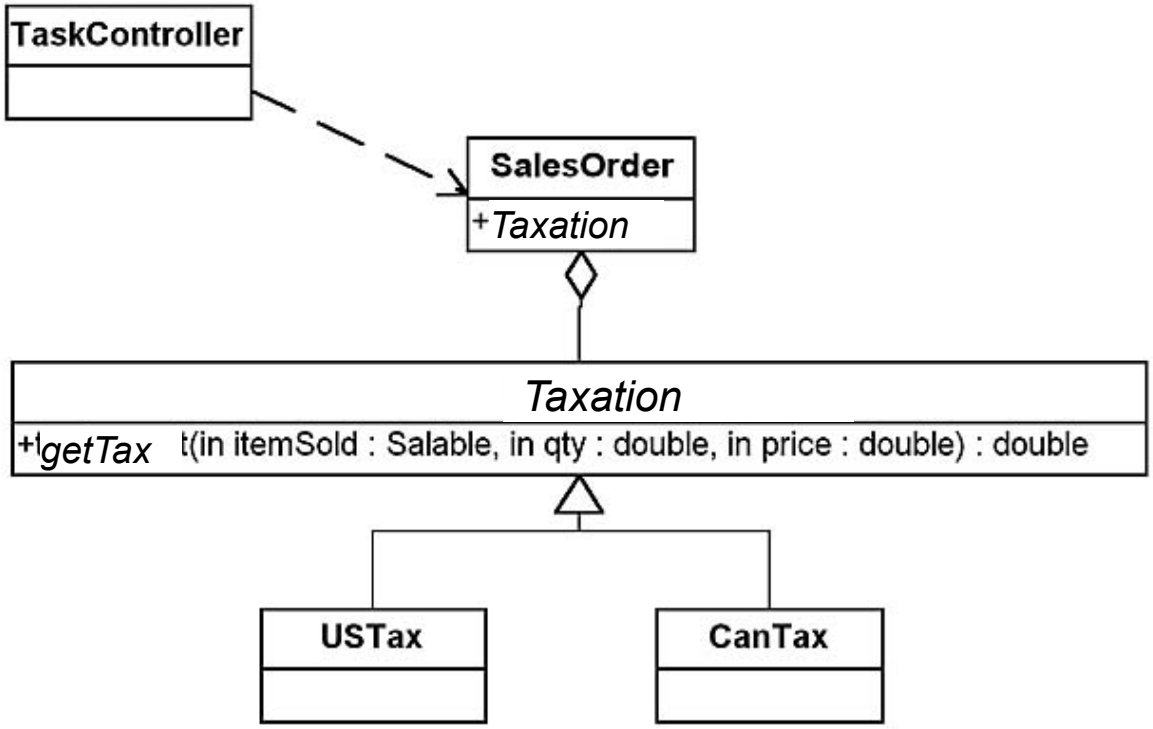
Solution 3b

New Requirements:

- Multiple Tax Domains
  - US Tax
  - Canadian Tax

- Use **aggregation** instead of inheritance

One SalesOrder class.
Have it contain
the Taxation class
to handle the variations



Solution 3b (for your reference)

New Requirements:
- Multiple Tax Domains
  - US Tax
  - Canadian Tax

```
public abstract class Taxation
{
   abstract public double getTax(
      long itemSold, double price);
}

public class CanTax extends Taxation
{
   public double getTax(
      long itemSold, double price) {
      …
   }
}
public class USTax extends Taxation
{
   public double getTax(
      long itemSold, double price) {
     …
   }
```

```
public class TaskController {
   public void process () {
      . . .
      // figure out which country you are in
      Taxation myTax;
      myTax= getTaxRulesForCountry();
      SalesOrder mySO= new SalesOrder();
      mySO.process(myTax);
   }
   private Taxation getTaxRulesForCountry() {
      …
   }
}
```

```
public class SalesOrder {
   public void process (Taxation taxToUse) {
      . . .
      // calculate tax
      double tax=
         taxToUse.getTax( itemNumber, price);
   }
}
```

Strategy Design Pattern

Strategy Design Pattern

Definition

"Used to create an interchangeable family of algorithms from which the required process is chosen at run-time."

Strategy Design Pattern

Intent

- Define a set of algorithms that can be used interchangeably
- Encapsulate each algorithm.
  - Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

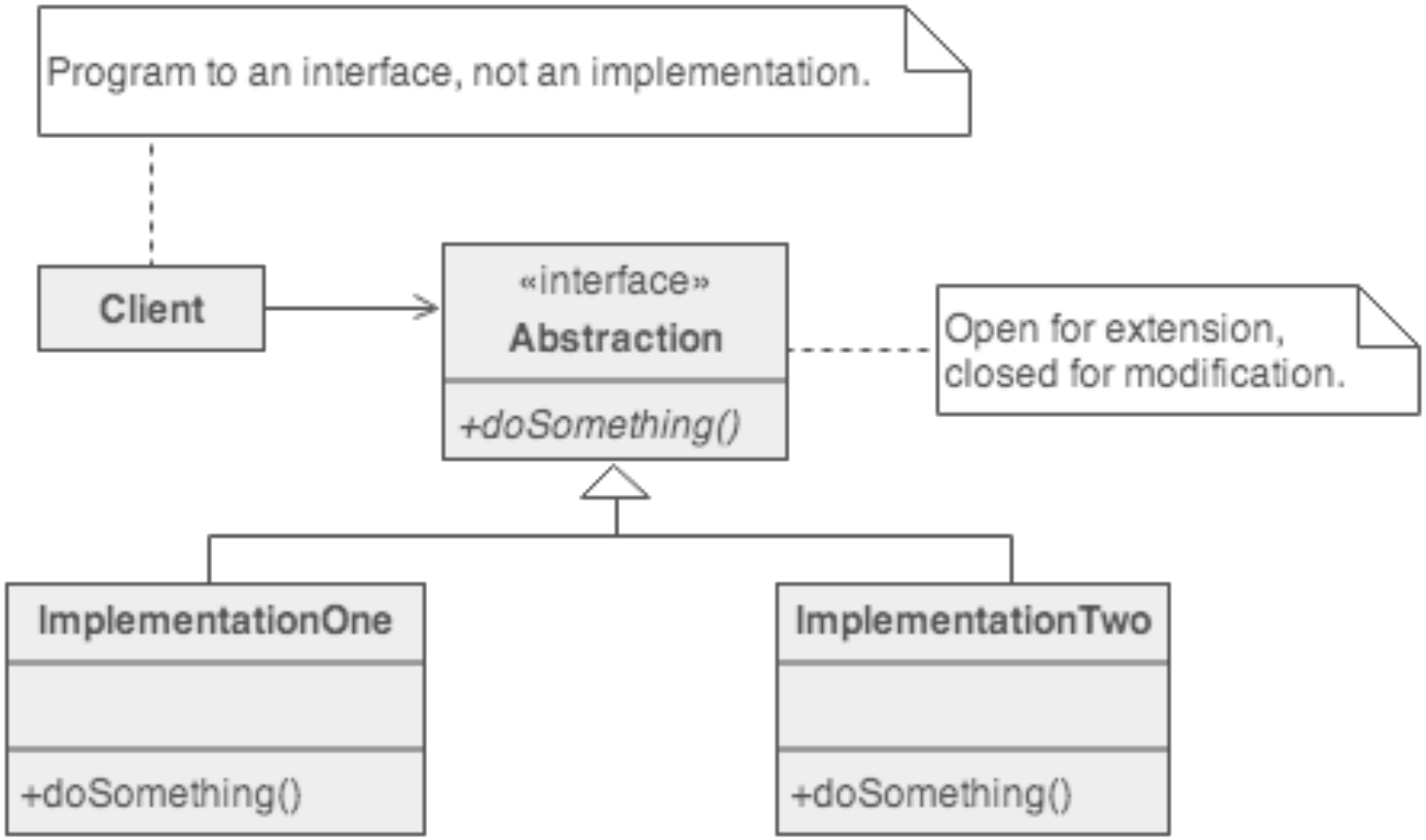
Strategy Design Pattern

Problem

- One of the dominant strategies of object-oriented design is the "open-closed principle".
  - Encapsulate interface details in a base class
  - Bury implementation details in derived classes
  - Clients can then couple themselves to an interface,
    - *Not have to experience the upheaval associated with change:*
    - *No impact when the number of derived classes changes,*
    - *No impact when the implementation of a derived class changes.*

Strategy Design Pattern

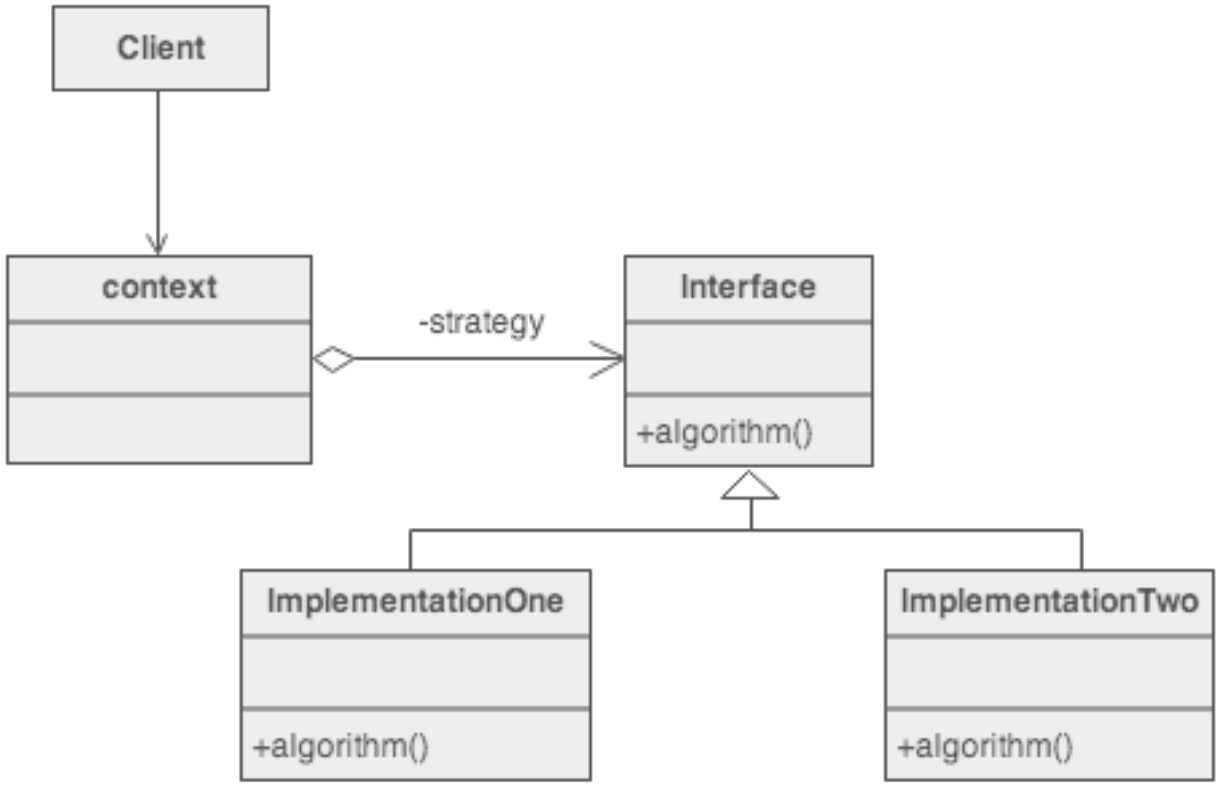
- Minimizes coupling: "Program to an interface"



Strategy Design Pattern

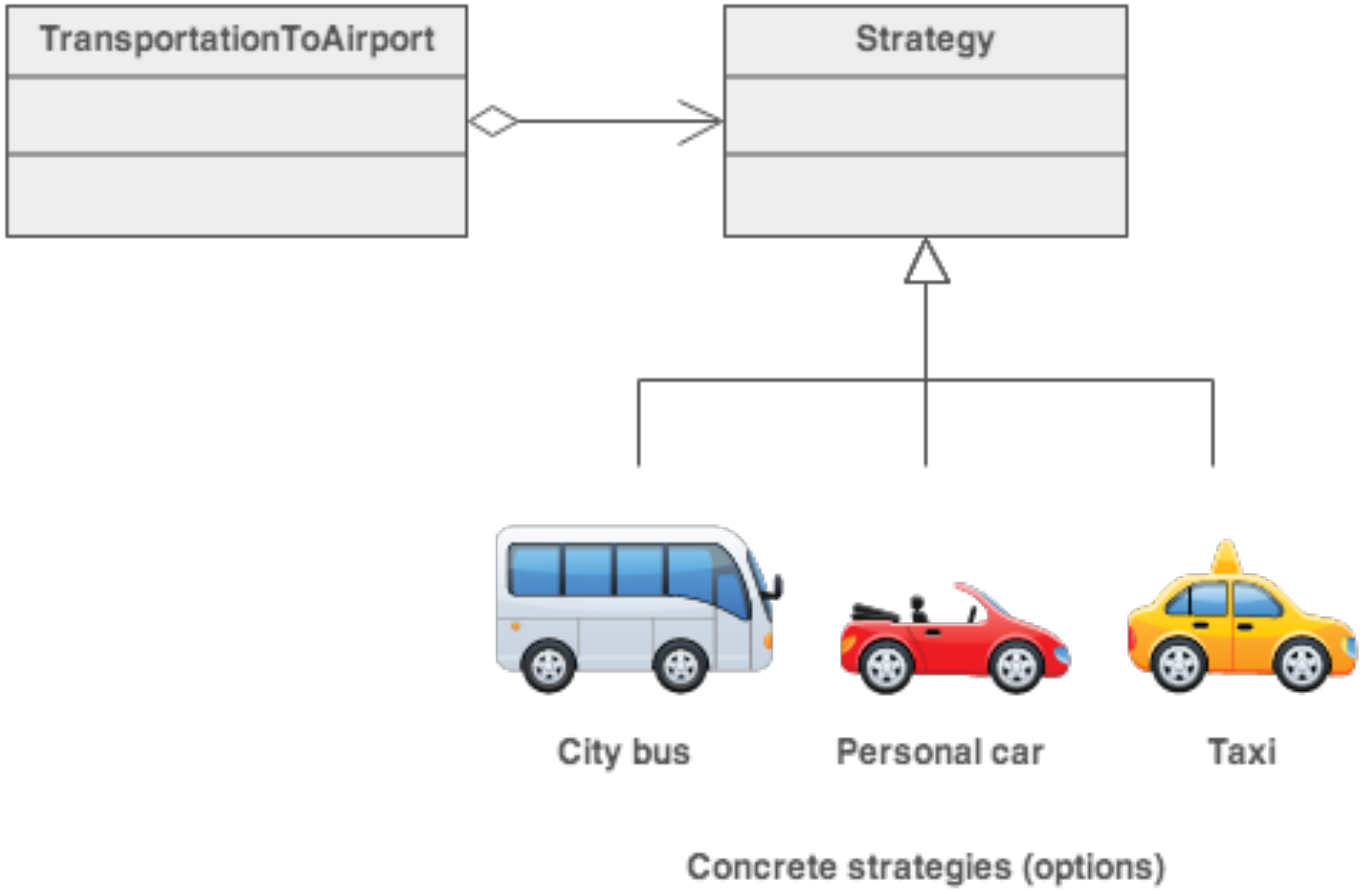
Structure

- Represent either an abstract base class, or
- The method signature expectations by the client



Strategy Design Pattern

Example



Strategy Design Pattern

Check list

- Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point".
- Specify the signature for that algorithm in an interface.
- Bury the alternative implementation details in derived classes.
- Clients of the algorithm couple themselves to the interface.

Strategy Design Pattern

Comparisons

- Strategy is like Template Method except in its granularity.
- State is like Strategy except in its intent.
- Strategy lets you change the guts of an object. Decorator lets you change the skin.
- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the 'handle/body' idiom. They differ in intent - that is, they solve different problems.
- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- Strategy objects often make good Flyweights.

GoF Solution…

GoF Solution

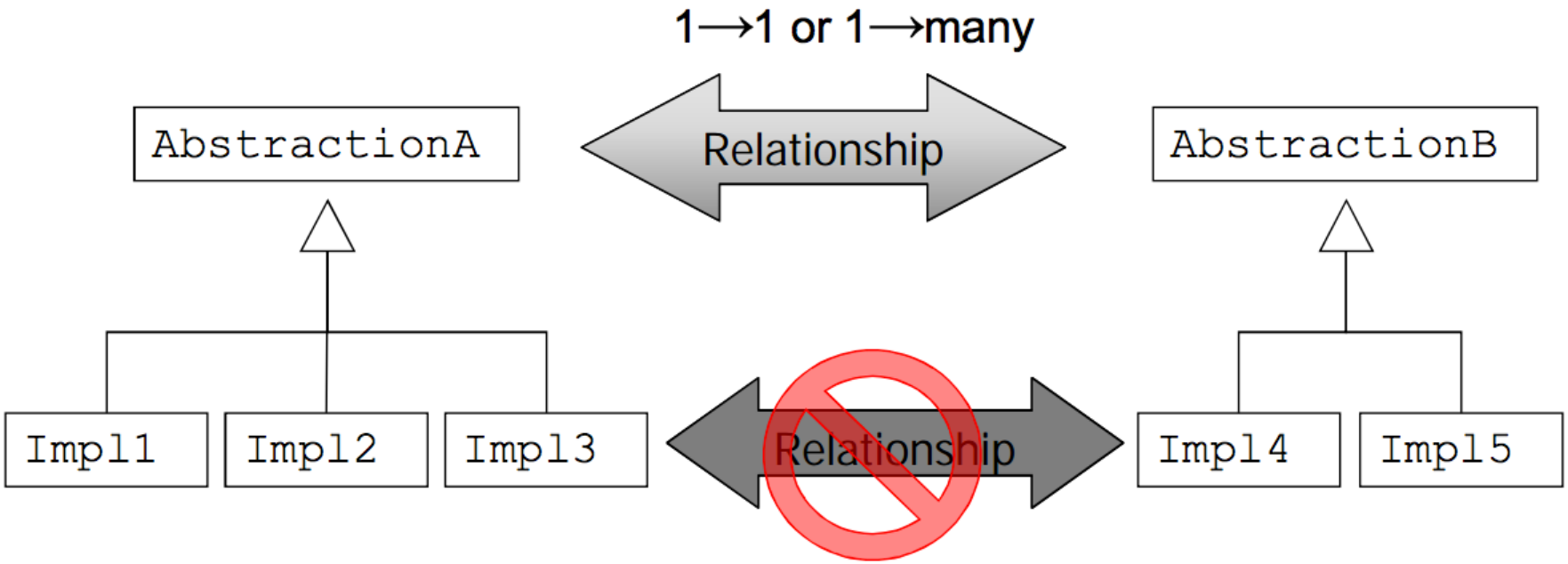
- **Design to interfaces**
- Favor **composition over** class **inheritance**
- Find what **varies** in your design and **encapsulate** it
  - "Encapsulate what varies"

Design to Interfaces

Design to Interfaces: Methods

- Craft method signatures from the perspective of the consuming entities
- Encapsulate: Hide the implementation of your methods
- If change underlying implementation, should not affect the consumer

Design to Interfaces



Inheritance vs. Composition

- 2 common techniques for facilitating object reuse:
  - **Class inheritance**
  - **Object composition**
    Object composition refers to assembling (*composing*) multiple objects together to get more complex functionality

- Each has their own advantages and disadvantages, but in general, one of the tenets of object-oriented programming is to

  Favor object composition over class inheritance

Why Composition is Better

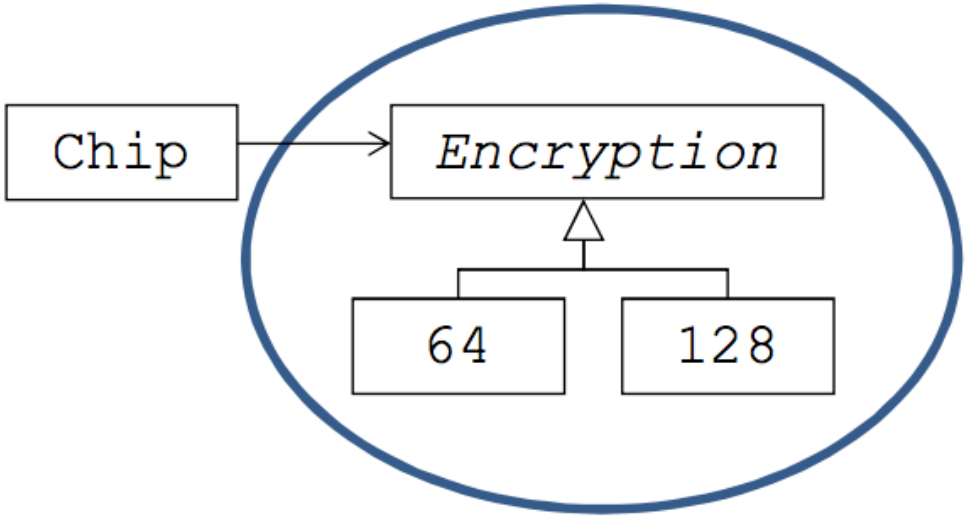
- Usually, inheritance is defined at compile-time, meaning you can't change implementations inherited from parent classes at run-time.

- Parents classes usually define at least part of their subclasses' physical representation.
  For this reason, it's often said that
  "*inheritance breaks encapsulation*".
  Normally, the subclass is implemented in terms of the parent's implementation, leading to a cascade of changes should a parent class be modified.

- Composition is defined at run-time by objects acquiring references to other objects. Composition requires objects to respect interfaces, which require objects to be designed so they don't stop you from using an object with many others

Encapsulate Variations

- Base classes encapsulate their implementing subclasses
- This encapsulates varying behavior

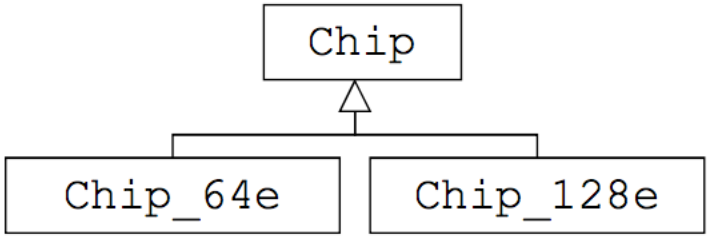


http://nwcpp.org/talks/2012/essential-skills-for-the-agile-developer-2012.pdf

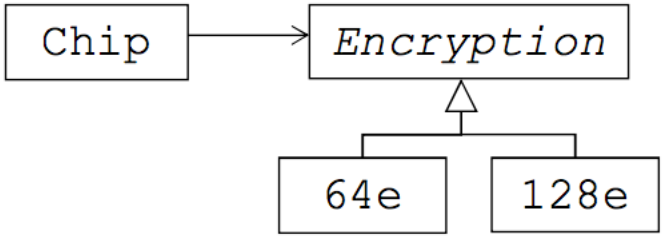
Favor Delegation Over Inheritance

Proper Use of Inheritance

- Define a class that encapsulates variation, contain (via delegation) an instance of a concrete class derived from the abstract class defined earlier



Class Inheritance to Specialize



Class Inheritance to Categorize

Who is the variation being hidden from?

1. Decoupling of concepts
2. Deferring decisions until runtime
3. Small performance hit

http://nwcpp.org/talks/2012/essential-skills-for-the-agile-developer-2012.pdf

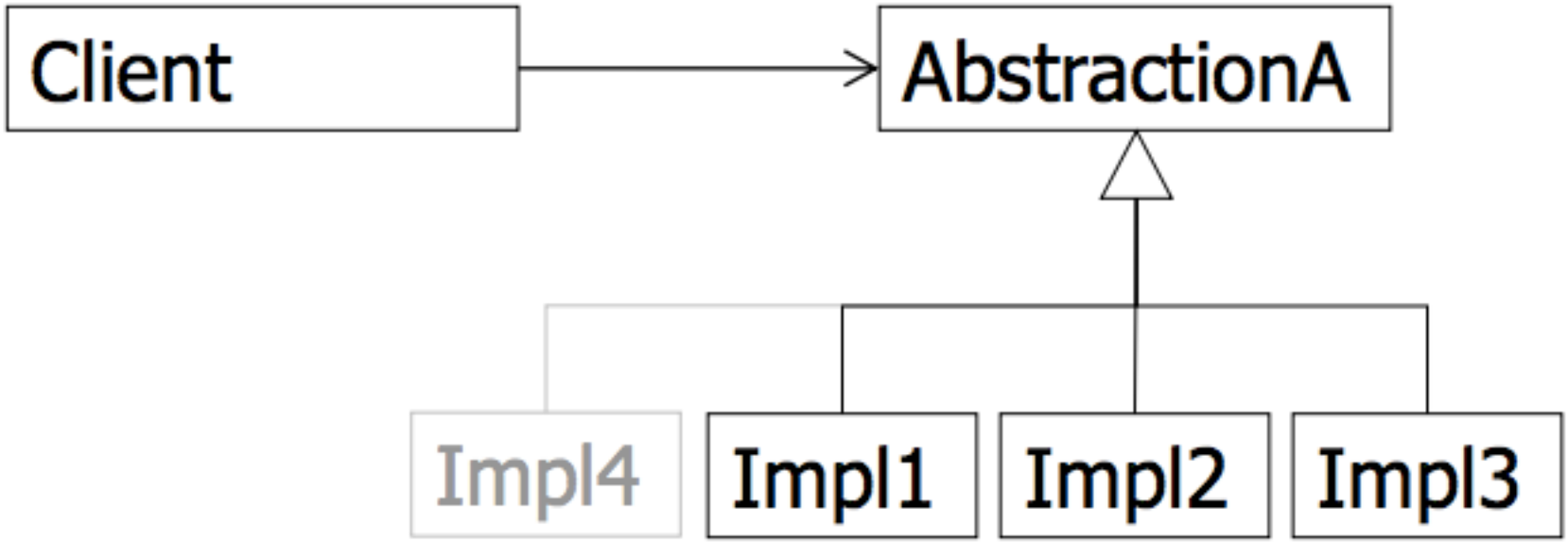
Encapsulate Variations

- Varying anything:
  - Varying design
  - Varying object creation
  - Varying relationships (1-1, 1-many)
  - Varying sequences and workflows
  - Etc…
- Encapsulating variation means to make it appear as if the varying issue is not varying
- Each pattern encapsulates a different varying thing

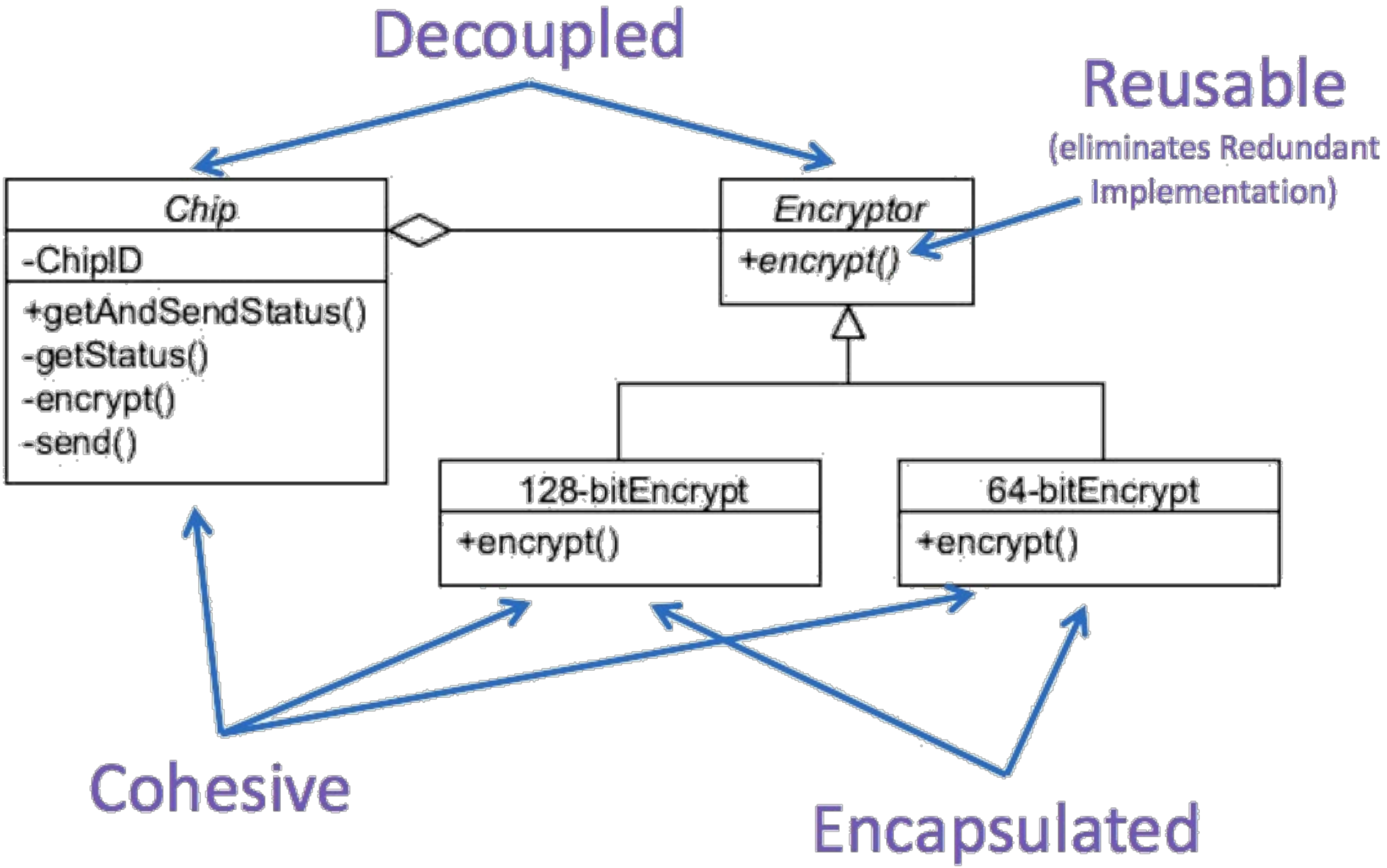
http://nwcpp.org/talks/2012/essential-skills-for-the-agile-developer-2012.pdf

Relates to the principles

- Deal with things at an abstract level
- Relates to the Open-Closed Principle



http://nwcpp.org/talks/2012/essential-skills-for-the-agile-developer-2012.pdf



http://nwcpp.org/talks/2012/essential-skills-for-the-agile-developer-2012.pdf

GoF Solution - Summary

- **Design to interfaces**
  - Helps eliminate redundant relationships
  - Avoids subclass coupling
- **Favor object delegation over class inheritance**
  - Promotes strong cohesion
  - Helps eliminate redundant implementation
- Consider what varies in your design and "**encapsulate the concept that varies**"
  - Promotes encapsulation
  - Decouples client objects from the services they use
  - Leads to component-based architectures

Design Patterns as Examples

- Each pattern is an example of a best practice for a given context.
- Design patterns are discovered, not invented: they are what successful designers have done to solve the same problem.
- Studying design patterns is a good way to study good design, and how it plays out under various circumstances