.NET Design Patterns

Slides

https://www.accelebrate.com
(877) 849-1850 ❖ info@accelebrate.com

Customized Technical Training

Accelebrate
ACCELERATED LEARNING, CELEBRATED RESULTS®

On-Site, Customized Private Training

Don't settle for a one-size-fits-all class! Let Accelebrate tailor a private class to your group's goals and experience. Classes can be delivered at your site or online (or a combination of both) worldwide. Visit our web site at **https://www.accelebrate.com** and contact us at **sales@accelebrate.com** for details.

Public Online Training

Need to train just 1-3 people? Attend one of our regularly scheduled, live, instructor-led public online classes. Class sizes are small (typically 3-6 attendees) and you receive just as much hands-on time and individual attention from your instructor as our private classes. For course dates, times, outlines, pricing, and registration, visit **https://www.accelebrate.com/public-training-schedule.**

Newsletter

Want to find out about our latest class offerings? Subscribe to our newsletter **https://www.accelebrate.com/newsletter**.

Blog

Get insights and tutorials from our instructors and staff! Visit our blog, **https://www.accelebrate.com/blog** and join the discussion threads and get feedback from our instructors!

Learning Resources

Get access to learning guides, tutorials, and past issues of our newsletter at the Accelebrate library, **https://www.accelebrate.com/library**.

Call us for a training quote!

877 849 1850

Accelebrate, Inc. was founded in 2002 with the goal of delivering private training that rapidly achieves participants' goals. Each year, our experienced instructors deliver hundreds of classes online and at client sites all over the US, Canada, and abroad. We pride ourselves on our instructors' real-world experience and ability to adapt the training to your team and their objectives. We offer a wide range of topics, including:

- Angular, React, and Vue
- JavaScript
- Data Science using R, Python, & Julia
- Tableau & Power BI
- .NET & VBA programming
- SharePoint & Office 365
- DevOps
- iOS & Android Development
- PostgreSQL, Oracle, and SQL Server
- Java, Spring, and Groovy
- Agile
- Web/Application Server Admin
- HTML5 & Mobile Web Development
- Ansible & Chef
- AWS & Azure
- Adobe & Articulate software
- Docker, Kubernetes, Ansible, & Git
- IT leadership & Project Management
- AND MORE *(see back)*

"I have participated in several Accelebrate training courses and I can say that the instructors have always proven very knowledgeable and the materials are always very thorough and usable well after the class is over."

— Rick, AT&T

Like us on Facebook + Follow us on Twitter + Watch us on YouTube

Visit our website for a complete list of courses!

Adobe & Articulate
Adobe Captivate
Adobe Creative Cloud
Adobe Presenter
Articulate Storyline / Studio
Camtasia
RoboHelp

AWS, Azure, & Cloud
AWS
Azure
Cloud Computing
Google Cloud
OpenStack

Big Data
Actian Matrix Architecture
Alteryx
Apache Spark
Greenplum Architecture
Kognitio Architecture
Teradata
Snowflake SQL

Data Science and RPA
Blue Prisim
Django
Julia
Machine Learning
Python
R Programming
Tableau
UiPath

Database & Reporting
BusinessObjects
Crystal Reports
MongoDB
MySQL
Oracle
Oracle APEX
Power BI
PivotTable and PowerPivot
PostgreSQL
SQL Server
Vertica Architecture & SQL

DevOps, CI/CD & Agile
Agile
Ansible
Chef
Docker
Git
Gradle Build System
Jenkins
Jira & Confluence
Kubernetes
Linux
Microservices
Red Hat
Software Design

Java
Apache Maven
Apache Tomcat
Groovy and Grails
Hibernate
Java & Web App Security
JBoss
Oracle WebLogic
Scala
Selenium & Cucumber
Spring Boot
Spring Framework

JS, HTML5, & Mobile
Angular
Apache Cordova
CSS
D3.js
HTML5
iOS/Swift Development
JavaScript
MEAN Stack
Mobile Web Development
Node.js & Express
React & Redux
Vue

Microsoft & .NET
.NET Core
ASP.NET
Azure DevOps
C#
Design Patterns
Entity Framework Core
F#
IIS
Microsoft Dynamics CRM
Microsoft Exchange Server
Microsoft Office 365
Microsoft Project
Microsoft SQL Server
Microsoft System Center
Microsoft Windows Server
PowerPivot
PowerShell
Team Foundation Server
VBA
Visual C++/CLI
Web API

Other
Blockchain
C++
Go Programming
IT Leadership
ITIL
Project Management
Regular Expressions
Ruby on Rails
Rust
Salesforce
XML

Security
.NET Web App Security
C and C++ Secure Coding
C# & Web App Security
Linux Security Admin
Python Security
Secure Coding for Web Dev
Spring Security

SharePoint
Power Automate & Flow
SharePoint Administrator
SharePoint Developer
SharePoint End User
SharePoint Online
SharePoint Site Owner

SQL Server
Azure SQL Data Warehouse
Business Intelligence
Performance Tuning
SQL Server Administration
SQL Server Development
SSAS, SSIS, SSRS
Transact-SQL

Teleconferencing Tools
Adobe Connect
GoToMeeting
Microsoft Teams
WebEx
Zoom

Web/Application Server
Apache Tomcat
Apache httpd
IIS
JBoss
Nginx
Oracle WebLogic

*Visit **www.accelebrate.com/newsletter** to sign up and receive our **newsletters** with information about **new courses**, **free webinars**, **tutorials**, and **blog articles**.*

Call us for a training quote! 877 849 1850 (US/Canada) or +1 678 648 3133

.NET Design Patterns

Agenda

- Dealing with Complexity
- The Object-Oriented Paradigm
- Overview of UML
- Introduction to Design Patterns
- Structural Patterns (Part I)
  - Facade
  - Adapter
- Testability

.NET Design Patterns

Agenda

- Behavioral Patterns (Part I)
  - Strategy Pattern
  - Template Method Pattern
- Structural Patterns (Part II)
  - Decorator Pattern
  - Proxy Pattern
- Behavioral Patterns (Part II)
  - Observer Pattern
  - Command Pattern

.NET Design Patterns

Agenda

- Creational Patterns
  - Singleton Pattern
  - Object Pool Pattern
  - Factory Method Pattern
  - Abstract Factory Pattern
- Model-View-Controller (MVC)
- Architectural Patterns and Styles
- Designing with Patterns

.NET Design Patterns

Dealing with Complexity

- Functional Decomposition
- Requirements and Inevitable Change
- Coupling and Cohesion
- Unwanted Side Effects
- Perspectives
- Responsibilities

Dealing with Complexity

Introduction

- Many developers are simply told…
  - Find the nouns in the requirements and make them objects
  - Encapsulation means "data hiding"
  - Objects are things with data and behavior used to access and manipulate that data
- This is a limited view, constrained by a focus on how to implement objects
- It is incomplete

Dealing with Complexity

Functional Decomposition

- Functional decomposition is a natural way to deal with complexity
- Allows you to think in terms of the sequence of steps required
- Individual steps can often be broken down further into another sequence of steps

Dealing with Complexity

Functional Decomposition

- One problem with functional decomposition is that it often leads to one "main" program that is responsible for controlling its subprograms
- Saddles the main program with too much responsibility
  - Ensuring everything is working correctly, coordinating and sequencing functions

Dealing with Complexity

Functional Decomposition

- A better approach might be to have the subprograms responsible for their own behavior
- Tell the function to do something and trust that it will know how to do it
- This is a form of delegation

Dealing with Complexity

Functional Decomposition

- Another problem with functional decomposition is that it does not help us prepare the code for possible changes in the future
- Virtually any change will require a change to the main controlling function

Dealing with Complexity

Requirements and Inevitable Change

- What is true about the requirements you get from users?
  - Incomplete
  - Usually wrong
  - Misleading
  - Do not tell the whole story

Dealing with Complexity

Requirements and Inevitable Change

- Requirements always change
- Users' view of their needs change and they become aware of new possibilities for the software
- Developers' view of the users' problem domain changes as they become more familiar with it
- The environment in which the software is being developed changes

Dealing with Complexity

Requirements and Inevitable Change

- It is not easy or often even possible to predict what changes might occur
- You can usually anticipate where changes might occur
- Object orientation can be used to contain those areas of change and isolate the effects of change more easily

Dealing with Complexity

Requirements and Inevitable Change

- Stop complaining about changing requirements
- Change the development process so change can be addressed more effectively

Dealing with Complexity

Coupling and Cohesion

- Cohesion refers to how closely related the operations in a routine are related to each other
- Weakly cohesive classes are those that do many, unrelated tasks
  - Become entangled with most everything in a system

Dealing with Complexity

Coupling and Cohesion

- Coupling refers to the strength of a connection between two routines
- The goal is to create routines with internal integrity (strong cohesion) and small, direct, visible, and flexible relations to other routines (loose coupling)

Dealing with Complexity

Unwanted Side Effects

- Most developers have had the experience of making a change to code in one area that has unexpected impact on code in another area
- This type of bug is an unwanted side effect of making the change
- Often difficult to find because we usually don't notice the relationship that caused the side effects

Dealing with Complexity

Unwanted Side Effects

- Many times, actually fixing bugs takes a short amount of time
- A much longer amount of time is spent trying to discover how the code works, on finding the cause of bugs, and avoiding unwanted side effects

Dealing with Complexity

Perspectives

- Martin Fowler describes three different perspectives in the software development process
  - Conceptual
  - Specification
  - Implementation

Perspectives

Conceptual

- Represents the concepts in the domain under study
- Little or no regard to the software that might implement it
- What am I responsible for?

Perspectives

Specification

- The interfaces of the software
- Not the implementation
- How am I used?

Perspectives

Implementation

- Code that is created to perform the necessary actions
- How do I fulfill my responsibilities?

Dealing with Complexity

Perspectives

- Conceptual
  - What am I responsible for?
- Specification
  - How am I used?
- Implementation
  - How do I fulfill my responsibilities?

Dealing with Complexity

Responsibilities

- If managing a group of people to accomplish a task, you could…
  - Give explicit directions to everyone
  - Assign a responsibility to each person who performs the tasks needed to fulfill their responsibility

Dealing with Complexity

Responsibilities

- In the second case, you are communicating with the people on a conceptual level
  - You are telling people what you want and not how to do it
- Individual people perform specific instructions and are working at the implementation level
- You are insulated from changes in the implementation details

.NET Design Patterns

Object-Oriented Paradigm

- Objects and Responsibilities
- Single Responsibility Principle (SRP)
- Interfaces and Abstract Classes
- Encapsulation and Polymorphism
- Liskov Substitution Principle (LSP)
- Object Construction and Destruction
- Classes vs. Structs in .NET

Object-Oriented Paradigm

Objects and Responsibilities

- Objects have traditionally been defined as data with methods
  - This is a very limited way of looking at objects
- An advantage of objects is that you can define things that are responsible for themselves
  - The data in an object allows it to know what state it is in

Objects and Responsibilities

Perspectives

- At the conceptual level…
  - An object is a set of responsibilities
- At the specification level…
  - An object is a set of methods that can be invoked by other objects or by itself
- At the implementation level…
  - An object is code and data and computational interactions between them

Object-Oriented Paradigm

Single Responsibility Principle (SRP)

- A class should have only one reason to change
- A class should not have multiple responsibilities that can vary independently from each other
  - Results in unnecessary coupling between responsibilities
  - Affects reusability and testability

Object-Oriented Paradigm

Interfaces and Abstract Classes

- Because objects have responsibilities and objects are responsible for themselves, there has to be a way to tell objects what to do
- Many methods of an object will be identified as callable by other objects
- These methods become the object's public interface

Interfaces and Abstract Classes

Defining an Interface

- In C#, an interface and an abstract class can both be used to define a general public interface
- A class (concrete or abstract) can implement multiple interfaces
- A class can inherit from only one class
- An interface can inherit from multiple other interfaces
- An abstract class can contain common implementation code as well as non-public methods
- All methods of an interface are public and cannot contain any implementation code

Interfaces and Abstract Classes

Features

- Variables declared as an interface or abstract class type should be thought of as a conceptual placeholder for another class that implements the specifics of the concept that the abstract type represents
- Supports polymorphism
- Enforces appropriate type checking

Object-Oriented Paradigm

Encapsulation and Polymorphism

- Encapsulation often is described simply as hiding data
  - Not exposing internal data members to the outside world
- Encapsulation really refers to any kind of hiding
- When a module uses a variable of an abstract type to trigger functionality, the abstract type hides the types of classes derived from it

Object-Oriented Paradigm

Encapsulation and Polymorphism

- If you use encapsulation and follow the strategy that objects are responsible for themselves, the only way to affect an object will be to call a method on that object
- Encapsulation promotes loose coupling

Object-Oriented Paradigm

Liskov Substitution Principle (LSP)

- It is important that a class deriving from a base class support all of the behavior of the base class
- The user of the object (via a base class variable) should not be able to tell a derivation is present
- Makes all derivations interchangeable with each other (type encapsulation)
- Critical for ensuring polymorphic behavior can be fully leveraged

Liskov Substitution Principle (LSP)

Violations

- Narrow pre-conditions
  - Overridden methods of a subclass accept a smaller set of input values than its base class
- Widen post-conditions
  - Overridden methods of a subclass return a wider set of output values than its base class
- Violate reasonable client assumptions
  - The public interface of a type may lead the client to make some reasonable assumptions
  - A subclass should not violate those assumptions

Object-Oriented Paradigm

Object Construction and Destruction

- If objects are treated as self-contained units, it would be a good idea to have methods to handle construction and destruction
- An object's constructor should be used to perform any tasks needs to create a well-defined object
  - Perform initializations
  - Set default information
  - Set up relationships with other objects

Object Construction and Destruction

Constructors in .NET

- Whenever a class or a struct is created, its constructor is called
- If you do not provide a constructor for a class, one is created for you that sets member variables to their default value
  - 0 for value types
  - Null for reference types
- You cannot define a custom default constructor for a struct
- Default constructor for a struct is only invoked if the instance is created using the new keyword

Object Construction and Destruction

Constructors in .NET

- Both classes and structs can define constructors that take parameters
- If a base class constructor is not called explicitly, the default base class constructor is called implicitly
  - Member variables given an initial value as part of their declaration are initialized prior to the base class constructor call

Object Construction and Destruction

Static Constructors

- A static constructor is used to initialize static data or to perform a particular action that need to be performed exactly once
- Called automatically before the first instance is created or any static members are referenced
- Cannot have parameters
- If an exception is thrown, the runtime will not invoke it a second time
  - Class remains uninitialized for the lifetime of the application domain

Object Construction and Destruction

Destructors and Finalize

- The Finalize method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed and its memory reclaimed by the garbage collector
- If a type overrides Finalize, the garbage collector adds an entry for each instance to the finalization queue
- The runtime ignores finalizers on value types (e.g. structs)

Object Construction and Destruction

Destructors and Finalize

- Defining a destructor is equivalent to overriding the Finalize method with a call to the Finalize method of the base class

```
class Car
{
  ~Car()
  {
    // cleanup statements...
  }
}
```

```
class Car
{
  protected override void Finalize()
  {
    try
    {
      // cleanup statements...
    }
    finally
    {
      base.Finalize();
    }
  }
}
```


Object Construction and Destruction

Destructors and Finalize

- Carefully consider any case in which you think a finalizer is needed
- Favor using resource wrappers such as SafeHandle to encapsulate unmanaged resources when possible

Object Construction and Destruction

IDisposable

- Finalizers have some drawbacks
  - Called when the GC detects that an object is eligible for collection
  - When the CLR needs to call a finalizer, it must postpone collection of the objects memory until the next round of garbage collection
- The IDisposable interface should be implemented to provide a manual way to release unmanaged resources

Object Construction and Destruction

Dispose Pattern

- The Dispose Pattern is intended to standardize the usage and implementation of finalizers and the IDisposable interface

```
public class DisposableClass : IDisposable
{
  public void Dispose() {
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing = false) {
    if (disposing) { // Cleanup managed resources }
    // Cleanup native resources
  }

  ~DisposableClass() {
    Dispose(false);
  }
}
```

Object-Oriented Paradigm

Classes vs. Structs in .NET

- Structs are value types
- Structs are copied on assignment
- Structs can declare constructors that have parameters
- Structs can be instantiated without using the new operator
- Structs cannot inherit from another struct or class
- Structs cannot be the base of a class
- Structs can implement interfaces

.NET Design Patterns

Overview of UML

- Introduction
- Diagram Types
- Class Diagrams
- Sequence Diagrams

Overview of UML

Introduction

- The Unified Modeling Language (UML) is a visual language
- Diagrammatic representation of a software system and the relationships among entities

Overview of UML

Diagram Types

- Analysis
  - Use Case and Activity diagrams
- Design
  - Class and State diagrams
- Implementation
  - Interaction and Sequence diagrams
- Deployment
  - Deployment diagrams

Overview of UML

Class Diagrams

- The most basic of UML diagrams is the Class diagram
- Describes classes and shows the relationships among them
- Is-a relationship
  - When one class is a kind of another class
- Has-a relationship
  - One class contains another class
- Uses-a relationship
  - One class uses another class

Class Diagrams

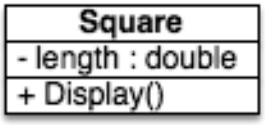
Composition vs. Aggregation

- Composition
  - The contained item is part of the containing item
  - Coincident lifetime
- Aggregation
  - The collection of sub-items can exist on their own

Class Diagrams

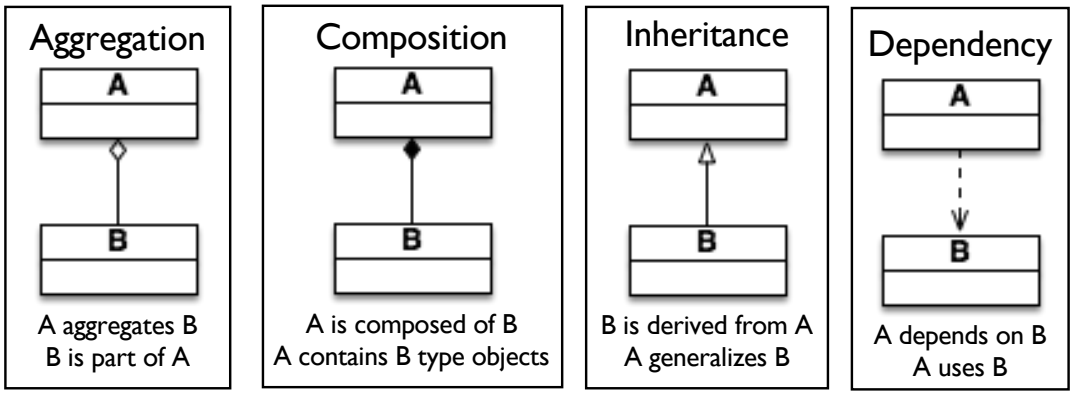
Classes

- UML can show the name of a class, the data members of the class, and the methods of the class
- The plus sign (+) is used to identify public members
- The minus sign (-) is used to identify private members
- The pound sign (#) is used to identify protected members
- An italic class name represents an abstract entity



Class Diagrams

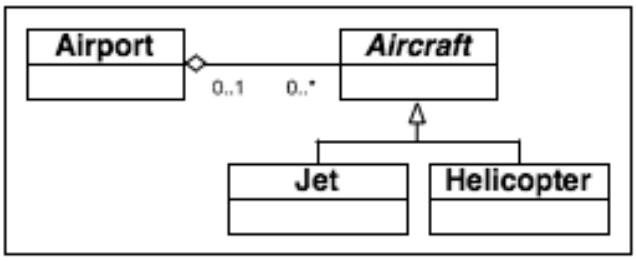
Relationships



Class Diagrams

Cardinality

- UML diagrams can include an indication of cardinality



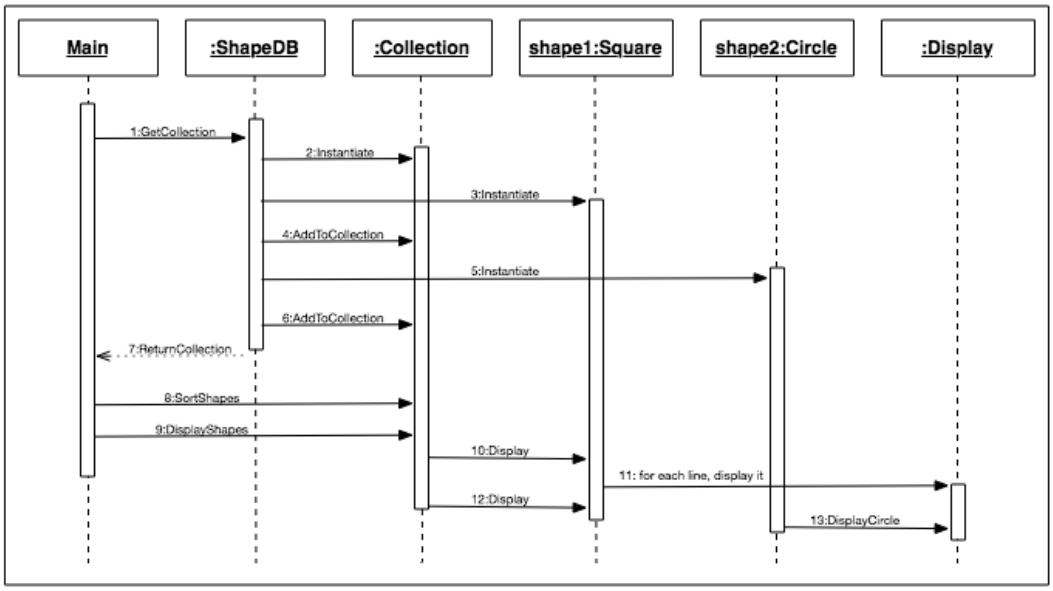
Overview of UML

Sequence Diagrams

- The UML diagrams that show how objects interact with each other are called Interaction diagrams
- The most common type of Interaction diagram is the Sequence diagram

Overview of UML

Sequence Diagrams



Overview of UML

Sequence Diagrams

- Rectangles at the top represent objects
- Vertical bars represent when an object is active
- Horizontal lines represent messages being sent between objects
  - Optionally, return values can be shown
- Add notes when necessary to communicate your design

.NET Design Patterns

Introduction to Design Patterns

- Origin of Design Patterns
- "Gang of Four" Patterns
- Key Features of Patterns
- Why Study Design Patterns?
- Design Strategies

Introduction to Design Patterns

Origin of Design Patterns

- The idea of design patterns originated with an architect named Christopher Alexander
- Alexander discovered that, for a particular architectural creation, good constructs had things in common with each other
- He called these similarities patterns

Introduction to Design Patterns

Origin of Design Patterns

"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

- Christopher Alexander

Introduction to Design Patterns

Origin of Design Patterns

- Alexander said that a description of a pattern involves four items
  - The name of the pattern
  - The purpose of the pattern, the problem it solves
  - How we could accomplish this
  - The constraints and forces we have to consider in order to accomplish it

Introduction to Design Patterns

"Gang of Four" Patterns

- The book that had the greatest influence on the use of patterns for software design was Design Patterns: Elements of Reusable Object-Oriented Software by Gamma, Helm, Johnson, and Vlissides
- Authors affectionately knows as the Gang of Four

Introduction to Design Patterns

"Gang of Four" Patterns

- Gang of Four book applied the idea of patterns to software design - calling them design patterns
- Described a structure within which to catalog and describe design patterns
- Cataloged 23 such patterns
  - Did not create the patterns
  - Formally identified patterns that already existed within the software community
- Postulated object-oriented strategies and approaches based on these design patterns

Introduction to Design Patterns

Key Features of Design Patterns

- Name
- Intent (purpose)
- Problem
- Solution
  - How the pattern provides a solution to the problem in the context in which it shows up
- Participants and collaborators
- Consequences (good and bad)
- Implementation
- Generic structure (typically UML)

Introduction to Design Patterns

Why Study Design Patterns?

- Reuse existing, high-quality solutions to commonly recurring problems
- Establish a common terminology to improve communication within teams
- Shift the level of thinking to a higher level perspective
  - Avoid thinking about details too early
- Improve the modifiability of code
  - Design patterns are typically time-tested solutions that have evolved into structures that can handle change more readily

Introduction to Design Patterns

Design Strategies

- The Gang of Four suggests a few strategies for creating good object-oriented designs
  - Design to an interface
  - Favor aggregation over inheritance
  - Find what varies and encapsulate it

.NET Design Patterns

Structural Patterns (Part I)

- Facade Pattern
- Adapter Pattern
- Facade vs. Adapter

Structural Patterns (Part I)

The Facade Pattern

- Intent
- Implementation
- Variations
- Hands-On Lab Exercise

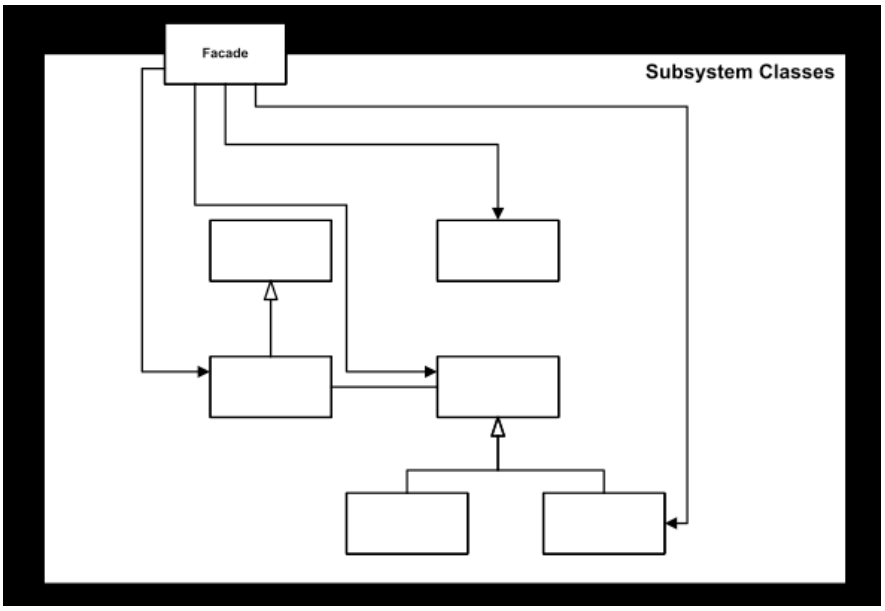
The Facade Pattern

Intent

- Provide a unified interface to a set of interfaces in a subsystem
- Define a higher-level interface that makes the subsystem easier to use
- May expose only a subset of the functionality of the underlying subsystem
- May choose reasonable default values to be used with the underlying subsystem

The Facade Pattern

Implementation



The Facade Pattern

Variations

- If a Facade can be made to be stateless, one Facade instance can be used by several other objects
  - Perhaps combined with Singleton
- The Facade can also be used to hide, or encapsulate, the system
  - Contain the system as private members of the Facade
  - Forcing all access to the system to go through the Facade can allow for easier monitoring of the system
  - Reduces the amount of effort required to switch to a different subsystem

Lab 1

The Facade Pattern

- Build a Facade over the .NET cryptographic subsystem
- Provide simple methods for encrypting and decrypting strings
  - Use the AES algorithm
  - Return the encrypted value as a Base64 encoded string

Structural Patterns (Part I)

The Adapter Pattern

- Intent
- Benefits
- Object Adapter vs. Class Adapter
- Variations

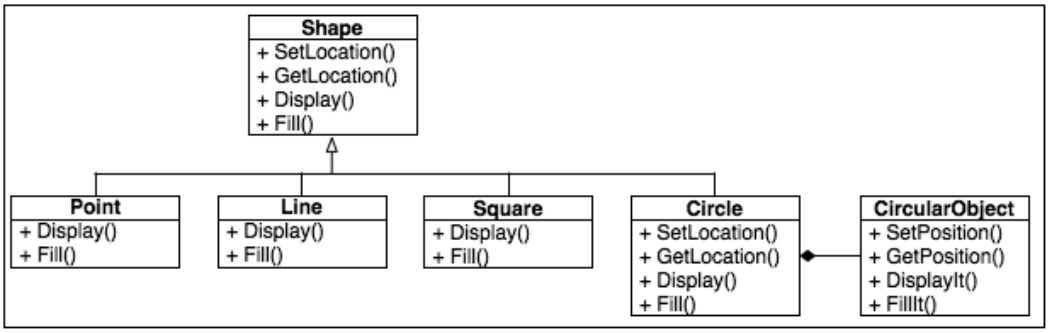
The Adapter Pattern

Intent

- Convert the interface of a class into another interface that the client expects
- Adapter lets classes work together that could not otherwise because of incompatible interfaces
- Especially useful for integrating an incompatible object into an existing hierarchy while preserving polymorphic behavior

The Adapter Pattern

Implementation



The Adapter Pattern

Object Adapter vs. Class Adapter

- The Object Adapter pattern relies on one object (the adapting object) containing another (the adapted object)
- The Class Adapter pattern uses multiple inheritance
  - Public and private inheritance can be used in languages that support it
  - In .NET, the adapter can inherit from the existing class and also implement the desired interface

The Adapter Pattern

Variations

- The adapter class may contain additional state or methods to satisfy the requirements of the interface
- Frees you from having to worry about the interfaces of existing classes when doing design
- If a class does what you need conceptually - you can always use the Adapter pattern to give it the correct interface

Structural Patterns (Part I)

Facade vs. Adapter

- Facade and Adapter are similar since they both wrappers around an existing interface
- Adapter involves a specific interface that you must design to - Facade does not
- With Facade, the motivation is to simplify the interface

.NET Design Patterns

Testability

- Introduction
- Unit Tests
- Integration Tests
- Test-Driven Development (TDD)
- Dependencies

Testability

Introduction

- Unit testing proves itself time an time again as one of the best ways a developer can improve code quality while gaining a deeper understanding of the functional requirements
- Keeping testability in mind when writing code tends to result in code that is more loosely coupled and maintainable
  - Even if you never end up writing a single unit test

Testability

Unit Tests

- A unit test is an automated piece of code that involves the unit of work being tested, and then checks some assumptions about a noticeable end result of that unit

Unit Tests

Definitions

- A unit of work is the sum of actions that take place between the invocation of a public method in the system and a single noticeable end result by a test of that system
- A noticeable end result can be observed without looking at the internal state of the system and only through its public API
  - Public method returns a value
  - Noticeable change to the behavior of the system without interrogating private state
  - Callout to a third-party system over which the test has no control

Unit Tests

Bad Unit Tests

- All developers have written unit tests before
- An ad-hoc console or WinForms application that tests a specific piece of functionality is a unit test

Unit Tests

Good Unit Tests

- Automated and repeatable
- Easy to implement
- Relevant tomorrow
- Easy to run
- Run quickly
- Consistent in its results
- Fully isolated (runs independently of other test)

Testability

Integration Tests

- Integration tests can be thought of as tests that aren't fast and consistent and that use one or more real dependencies of the units under test
  - A test that uses the real system time, the real filesystem, or a real database has stepped into the realm of integration testing
- Integrations tests are important counterparts to unit tests but they should be kept separate

Testability

Test-Driven Development (TDD)

- Once you know how to write structured, maintainable, and solid tests with a unit testing framework, the next question is when to write them
- A growing number of developers prefer writing unit tests before the production code is written

Test-Driven Development (TDD)

Steps

- Write a failing test to prove code of functionality is missing from the end product
  - Helps to test the test
- Make the test pass by writing production code that meets the expectations of the test
- Refactor your code

Test-Driven Development (TDD)

Benefits

- TDD done correctly can…
  - Improve code quality
  - Decrease the number of bugs
  - Raise your confidence in the code
  - Shorten the time it takes to find bugs
- TDD done incorrectly can…
  - Cause your project schedule to slip
  - Waste your time
  - Lower your motivation

Testability

Unit Testing Frameworks

- Unit testing frameworks help developers unit test their code and optionally run tests as part of an automated build
- The two most popular unit testing frameworks for .NET are MSTest and NUnit
- The built-in test runner in VS 2012 and later allows running tests written in other frameworks
  - Including NUnit via the NUnit test adapter for Visual Studio

Testability

Writing Unit Tests

- A unit test usually composed of three main actions
  - Arrange objects, creating and setting them up as necessary
  - Act on the object
  - Assert that something is as expected
- The Assert class has several helpful static methods
  - A method call that returns true is considered a pass while a method that returns false is considered a failure

Lab 2

Writing Unit Tests

- Write a unit test for the Facade created in the previous lab

Testability

Writing Unit Tests

- MSTest and NUnit both support attributes that can be used for unit test initialization and test cleanup
- TestInitialize and TestCleanup in MSTest
- SetUp and TearDown in NUnit

Testability

Expected Exceptions

- Sometimes, you want to make sure a specific unit of work will throw a particular exception under certain conditions
- The ExpectedException attribute is used in both MSTest and NUnit

Testability

Dependencies

- Stubs
- Interaction Testing
- Mock Objects
- Stubs vs. Mock Objects
- Isolation Frameworks

Dependencies

Stubs

- Often, the object under test relies on another object over which you have no control (or doesn't work yet)
- A stub is a controllable replacement for an existing dependency in the system
- By using a stub, you can test your code without dealing with the dependency directly

Stubs

Indirection

- "There is no object-oriented problem that can't be solved by adding a layer of indirection, except, of course, too many layers of indirection"
- Find the interface or API that the object under test works against
- Replace the underlying implementation of that interface with something that you have control over

Stubs

Extract Interface

- Often, it may be necessary to create a new interface that defines the dependency
- A fake object can then implement the same interface and stand in for the dependency

Stubs

Dependency Injection

- Object under test must provide a way for the dependency to be provided at runtime or injected
  - Constructor injection
  - Property injection
  - Parameter injection

Stubs

Visibility

- If adding methods specifically for testability (e.g. additional constructor), you can mark the method internal and use the InternalsVisibleTo attribute in AssemblyInfo.cs

```
[assembly:InternalsVisibleTo("CryptoLibTests")]
```

Testability

Interaction Testing

- Value-based testing checks the value returned from a method
- State-based testing is about checking for noticeable behavior changes in the system under test
- Interaction testing is testing how an object sends messages to other objects
  - Useful when calling another object is the end result of a specific unit of work

Testability

Mock Objects

- A stub is a fake object being used to stand in for a dependency
  - Allows for the rest of the object to be tested independently
- A mock object is used to test that your object interacts with other objects correctly
  - Mock object is a fake object that decides whether the unit test has passed or failed based on how it is used
  - Useful for objects that call third-party objects

Testability

Stubs vs. Mock Objects

- The key difference between a stub and a mock object is that you perform asserts against a mock object to determine test success

Testability

Isolation Frameworks

- Several issues are present when using manual mocks and stubs
- It takes time to write the mocks and stubs
- It's difficult to write stubs and mocks for complex objects
- To save state for multiple calls of a mock method requires a lot of boilerplate code
- It's hard to reuse mock and stub code for other tests

Isolation Frameworks

Introduction

- Isolation frameworks (i.e. mocking frameworks) can make the job of creating fake objects much simpler, faster, and shorter
- Lessens the need to write repetitive code to assert or simulate object interactions
- Can create and configure fake objects at runtime
  - Dynamic stubs and dynamic mocks

Isolation Frameworks

Examples

- NSub
- Moq
- FakeItEasy
- NSubstitute
- Typemock Isolator
- JustMock
- Rhino Mocks
- NMock
- Moles (named Microsoft Fakes in VS 2012+)

.NET Design Patterns

Behavioral Patterns (Part I)

- Handling New Requirements
- Open-Closed Principle (OCP)
- Strategy Pattern
- Template Method Pattern

Behavioral Patterns (Part I)

Handling New Requirements

- Disaster often comes in the long run from suboptimal decisions made in the short run
- There are several reasons projects tend to ignore long-term issues such as ease of maintenance or ability to change
  - We can't figure out how the requirements are going to change
  - If we try to see how things will change, we'll stay in analysis forever
  - If we try to write out software so we can add new functionality to it, we'll stay in design forever
  - We don't have the budget or the time - I'll do it later

Handling New Requirements

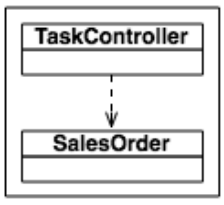
Anticipating Change

- It is sometimes impossible to anticipate the exact nature of the change
- It is easier to anticipate where the changes will occur and design for that type of change

Handling New Requirements

Case-Study Application

- Imagine an e-commerce system that needs to calculate the price for sales orders
- Part of the price calculation involves calculating sales tax
- Initial design includes a SalesOrder class that calculates the sales tax for an order using rules for US customers



Handling New Requirements

Case-Study Application

- Later, it becomes necessary to handle taxes on orders from customers outside the United States
- Possible solutions include…
  - Copy and paste
  - Switches or ifs on a variable specifying the case we have
  - Use delegates with a different one representing each case
  - Inheritance
  - Delegate the entire functionality to a new object

Handling New Requirements

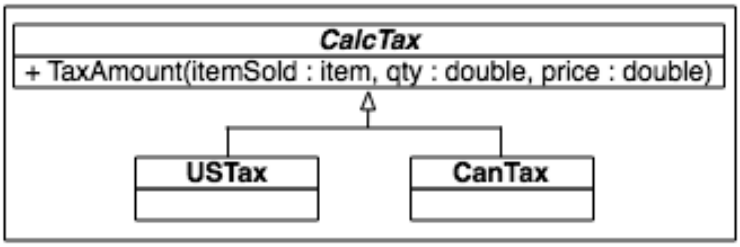
Design Pattern Approach

- Consider what should be variable in your design
- Encapsulate the concept that varies
- Favor object-aggregation over class inheritance

Handling New Requirements

Case-Study Application

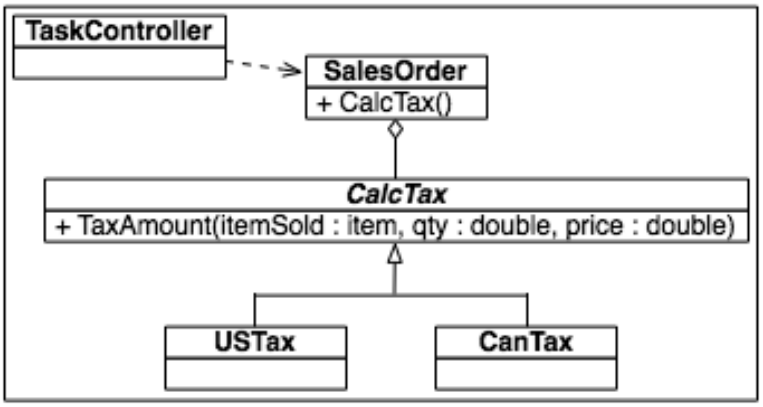
- In the example, it is the tax rules that are varying
- To encapsulate them would mean creating an abstraction that defines how to accomplish taxation conceptually
- Create a concrete class for each of the variations



Handling New Requirements

Case-Study Application

- SalesOrder can now contain the variation with aggregation



Handling New Requirements

Case-Study Application

- Improves cohesion
  - Tax calculation is now handled in its own class
- Improves flexibility
  - New tax rules just require us to derive a new class from CalcTax that implements them
  - SalesOrder can ask another object which then dynamically chooses which tax object to use
  - Allows business rule to vary independently from the SalesOrder object that uses it

Behavioral Patterns (Part I)

Open-Closed Principle (OCP)

- "Software entities should be open to extension but closed for modification"
- When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity
- If OCP is applied well, further changes of that kind are achieved by adding new code, not changing old code that already works

Behavioral Patterns (Part I)

The Strategy Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable
- Strategy lets the algorithm vary independently from clients that use it

The Strategy Pattern

Principles

- Objects have responsibilities
- Different, specific implementations of these responsibilities are manifested through the use of polymorphism

Lab 3

The Strategy Pattern

- Refactor an existing application to employ the Strategy pattern
- Use a TDD-style approach

The Strategy Pattern

Examples in the .NET Framework

- LINQ is a great example of the Strategy Pattern
  - Collection of extension methods (Where, OrderBy, etc.) that take delegate parameters
  - Delegate provides the strategy to use to perform the operation (filter, sort, etc.)

Behavioral Patterns (Part I)

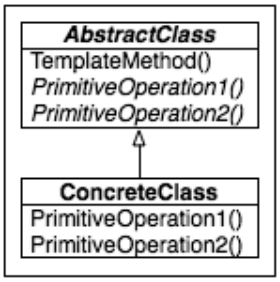
The Template Method Pattern

- Define the skeleton of an algorithm in an operation, deferring some of the steps to the subclasses
- Redefine the steps in an algorithm without changing the algorithm's structure

The Template Method Pattern

Intent

- Helps to generalize a common process at an abstract level
- Often times, there is a set of procedures that must be followed
  - All performed and in a specific sequence
- However, implementing some of the steps can vary
- Points of customization can be controlled through the use of virtual and abstract



.NET Design Patterns

Structural Patterns (Part II)

- Decorator Pattern
- Proxy Pattern

.NET Design Patterns

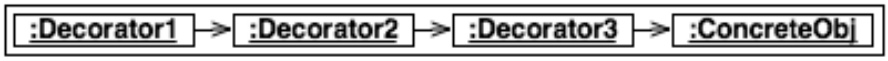
The Decorator Pattern

- Intent
- Implementation
- Combined with Factory

The Decorator Pattern

Intent

- Attach additional responsibilities to an object dynamically
- Decorators provide a flexible alternative to subclassing for extending functionality
- Provides the ability to compose functionality as a chain of objects



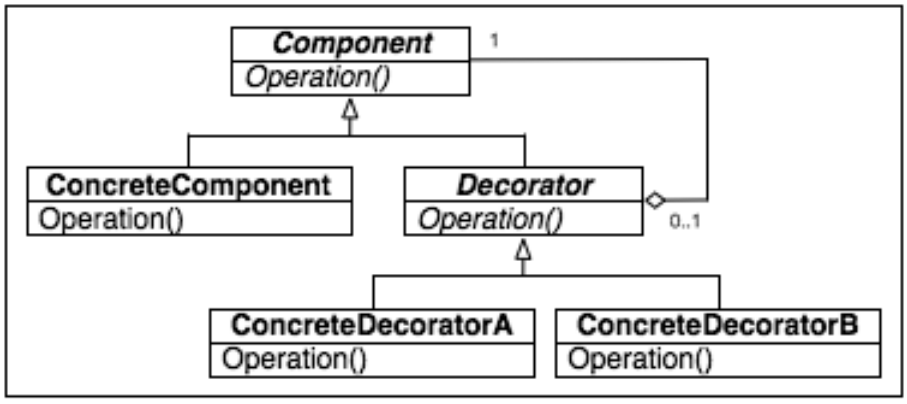
The Decorator Pattern

Implementation

- Each chain of objects starts with a Component (a ConcreteComponent or a Decorator)
- Each Decorator is followed either by another Decorator or by the original ConcreteComponent
- A ConcreteComponent always ends the chain

The Decorator Pattern

Implementation



The Decorator Pattern

Implementation

- Each Decorator object wraps its new function around its trailing object
- Each Decorator performs its added function either before its decorated function or after it

The Decorator Pattern

Combined with Factory

- The power of the Decorator pattern requires that the instantiation of the chains of objects be completely decoupled from the Client objects that use it
- Most typically accomplished through the use of factory objects that instantiate the chains based upon some configuration information

Lab 4

The Decorator Pattern

- Add new features to the project from the previous lab using the Decorator pattern

The Decorator Pattern

Examples in the .NET Framework

- Streams in the .NET Framework are designed around the Decorator Pattern
  - There are fundamentally different types of streams that are modeled using inheritance (FileStream, MemoryStream, NetworkStream)
  - Additional functionality (buffering, encryption, etc.) can be added to a stream dynamically

.NET Design Patterns

The Proxy Pattern

- Provide a surrogate or place-holder for another object to control access to it
- The Proxy pattern involves creating a class that implements the same interface that we are standing-in for
- The Proxy forwards requests to the real object which it stores internally
- Useful for methods that are time-consuming such as high latency network operations
  - Proxy can handle performing operations in a concurrent, non-blocking manner

The Proxy Pattern

Examples in the .NET Framework

- COM interoperability in .NET in a good example of the Proxy Pattern
  - When you add a reference to a COM object in Visual Studio, a proxy object is created
  - The proxy object exposes the same interface as the underlying COM object but handles all of the details necessary for properly handling the COM object

.NET Design Patterns

Behavioral Patterns (Part II)

- Observer Pattern
- .NET Events
- Command Pattern
- WPF and ICommand

Behavioral Patterns (Part II)

The Observer Pattern

- Intent
- Subject and Observers
- Responsibilities

The Observer Pattern

Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Also goes by the names Dependents and Publish-Subscribe

The Observer Pattern

Observers

- The objects that need to be notified of a change in state are called observers
- Frequently, the objects in your application that you would like to act as observers have different interfaces
- Is is desirable for all observers to have the same interface
  - Otherwise, the subject would have to be modified to handle each different type of observer

The Observer Pattern

Responsibilities

- The observers should be responsible for knowing what they are to watch for
- The subject should be free from knowing which observers depend on it
- There needs to be a way for the observers to register themselves with the subject
  - Attach, AddObserver, etc.
  - Detach, RemoveObserver, etc.
- It then becomes easy for the subject to call a defined method for each observer

The Observer Pattern

.NET Events

- The built-in features of .NET help you implement the Observer pattern with much less code
- An event represents an abstraction for the typical subject registration and notification functionality

The Observer Pattern

.NET Events

- Rather than implementing an observer interface and registering itself with the subject, an observer creates a specific delegate instance and registers that with the subject's event
  - Delegate type specified by the event declaration
- EventHandler and EventHandler<TEventArgs> are pre-defined delegate types that can be used

The Observer Pattern

.NET Events

- Using the event keyword in C# creates something like a property
- Underlying delegate stored in a private backing field
- Add and remove methods exposed publicly
- Allows any observer to subscribe but only the subject can invoke the delegate (fire the event)

.NET Events

Subject

```
public class Speedometer
{
  public event EventHandler ValueChanged;
  private int currentSpeed;

  public virtual int CurrentSpeed
  {
    get { return currentSpeed; }
    set
    {
      currentSpeed = value;
      OnValueChanged();
    }
  }

  protected void OnValueChanged()
  {
    if (ValueChanged != null)
      ValueChanged(this, EventArgs.Empty);
  }
}
```

.NET Events

Observer

```
public class SpeedMonitor
{
  public SpeedMonitor(Speedometer speedo)
  {
    speedo.ValueChanged += SpeedHasChanged;
  }

  private void SpeedHasChanged(object sender, EventArgs e)
  {
    // do something
  }
}
```


Lab 5

The Observer Pattern

- Examine an application that uses a manual implementation of the Observer pattern
- Refactor the application to use the .NET event system

Behavioral Patterns (Part II)

The Command Pattern

- Intent
- Typical Uses
- Hands-On Lab Exercise

The Command Pattern

Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- Allows you to decouple an object making a request from the object that receives the request and performs the action

The Command Pattern

Typical Uses

- The Command pattern can be used to handle UI events in a more loosely coupled fashion
  - Code that registers an event handler needs to be modified if the element is removed
  - WPF uses the Command pattern combined with data binding to maximize separation
- A command can be used to encapsulate both what it does and how to undo it

Lab 6

The Command Pattern

- Extend an existing application using the Command pattern

The Command Pattern

WPF and ICommand

- In an GUI application, functionality can be divided into higher-level tasks
- These tasks may be triggered by a variety of different actions and user-interface elements
  - Menus, keyboard shortcuts, toolbars, etc.
- WPF allows you to define these tasks as commands and connect controls to them
  - Avoids repetitive event handling code
  - Automatically manages the state of the user interface

WPF and ICommand

The WPF Command Model

- The WPF command model consists of four ingredients
  - Commands - represents the application task and keeps track of whether it can be executed
  - Command bindings - links a command to the related application logic
  - Command sources - triggers a command (MenuItem)
  - Command targets - element on which the command is being performed (TextBox for a Paste command)

WPF and ICommand

ICommand Interface

- The ICommand interface defines how commands work
- All WPF commands are instances of RoutedCommand which implements ICommand

```
public interface ICommand
{
  void Execute(object parameter);
  bool CanExecute(object parameter);

  event EventHandler CanExecuteChanged;
}
```


.NET Design Patterns

Creational Patterns

- Motivation
- Factories
- Singleton Pattern
- Object Pool Pattern
- Factory Method Pattern
- Abstract Factory Pattern

Creational Patterns

Motivation

- When designing objects, it is best to concerned first with their behavior (what the objects do and how other objects tell them to do it)
- Later, you need to be concerned about figuring out when a particular object is needed and making sure is it ready
- If the code that uses the objects also is responsible for instantiating the objects, the code can become complicated
  - Which objects to create
  - Which construction parameters are needed
  - How to use the object after construction
  - Maybe manage a pool of objects

Creational Patterns

Factories

- Factories help keep objects cohesive, decoupled, and testable
- A factory is a method, an object, or anything that else that is used to instantiate other objects

The Singleton Pattern

Intent

- The Singleton pattern is used to ensure a class only has one instance, and provide a global point of access to it

The Singleton Pattern

Structure

```
public class SingularObject
{
  private static SingularObject _instance;

  private SingularObject() { }

  public static SingularObject Instance
  {
    get
    {
      if (_instance == null)
      {
        _instance = new SingularObject();
      }
      return _instance;
    }
  }
}
```

The Singleton Pattern

Structure (Thread Safe .NET Implementation)

```
public class SingularObject
{
  private SingularObject() { }

  public static SingularObject Instance
  {
    get { return Nested.instance }
  }

  private class Nested
  {
    static Nested() { }

    internal static readonly SingularObject instance = new SingularObject();
  }
}
```


The Singleton Pattern

Testability

- Objects that use a singleton should typically obtain the instance and store it for future use
- Code that repeatedly obtains a singleton via a static method represents a dependency that can hurt testability

The Object Pool Pattern

Intent

- Manage the reuse of objects when it is either expensive to create an object or there is a limit on the number of a particular type that can be created
  - Network or database connections
  - In-memory external processes
  - Threads of execution

The Object Pool Pattern

Implementation

- A manager object implements methods to acquire a reusable object and return the object to the pool
- Sometimes, the reusable object provides a way to change its state so the manager knows if the object is available for reuse
- The manager may provide methods for controlling aspects of the pool (e.g. maximum pool size)

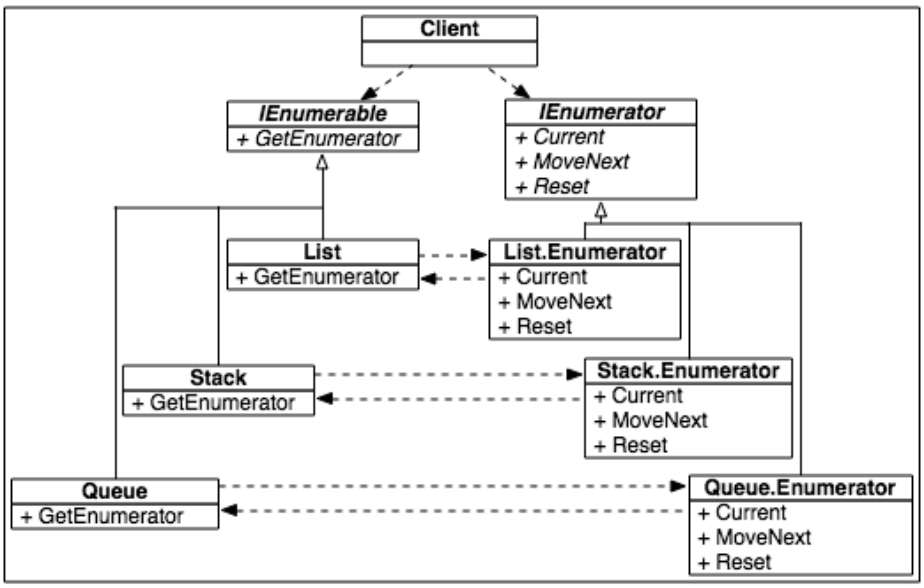
The Factory Method Pattern

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate
- Factory Method lets a class defer instantiation to subclasses
- One example in .NET is the GetEnumerator method defined by the IEnumerable interface

The Factory Method Pattern

Implementation



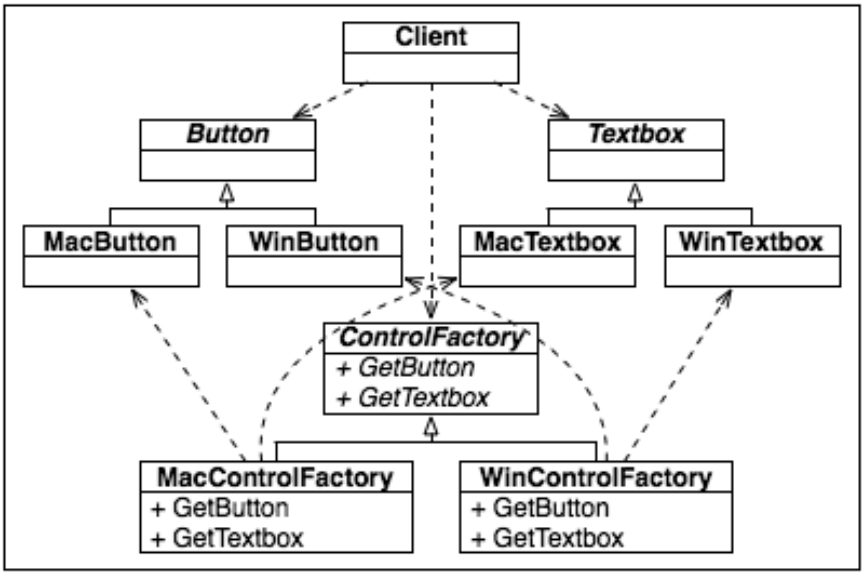
The Abstract Factory Pattern

Intent

- Provide an interface for creating families of related or dependent objects without specifying the concrete classes
- For example, a cross-platform UI framework may consist of several objects that need to be instantiated in a coordinated fashion
  - One set of objects is designed to work on one operating system while another set works on another operation system

The Abstract Factory Pattern

Implementation



.NET Design Patterns

Model-View-Controller (MVC)

- Introduction
- Model-View-Controller (MVC) Pattern
- Model-View-Presenter (MVP) Pattern
- Model-View-ViewModel (MVVM) Pattern
- WPF and ASP.NET MVC

Model-View-Controller (MVC)

Introduction

- The MVC pattern is a way of achieving a looser coupling between the constituent parts of a GUI application
- Model - The data of the application and associated business logic
- View - Visual representation of the model data
- Controller - Responsible for delivering model data to the view and acts as the destination of user actions

Model-View-Controller (MVC)

Models

- The model should not be aware of the existence of controllers or views
- Should be very reusable with minimal dependencies
- Can fire events that other components can subscribe to

Model-View-Controller (MVC)

Views

- Should only contain logic that is specific to presentation
  - Views are notoriously difficult to unit test
- Data should already be shaped for consumption by the view before it arrives
- Views can listen for model events and update themselves
  - Data binding mechanisms can help with this
- The Command pattern can be used to decouple UI events from controller code

Model-View-Controller (MVC)

Controller

- Responsible for determining the appropriate response for a given user interaction
- Responsible for obtaining model data and passing it to a view
  - May need to compose an entity based on multiple model objects or restructure data based on the needs of the view
- Should know about the existence of views and their individual purpose but not be aware of their internal implementation

Model-View-Controller (MVC)

Model-View-Presenter (MVP)

- MVP is a slight variation to MVC
- In MVC, actions arrive initially at the controller
  - Every action in the View correlates with a call to a Controller
- In MVP, actions route through the view to the presenter
- The MVP pattern is more prevalent in WinForms and WebForms application because of how they are architected
  - Code-behind files and postbacks

Model-View-Controller (MVC)

Model-View-ViewModel (MVVM)

- A ViewModel is an object specifically designed to be the object consumed by a view
- Useful when the View needs multiple model objects or to avoid adding view-specific artifacts to a model class
- WPF contains many features that support an MVVM approach
  - Data binding, data templates, commands, behaviors
- ViewModels are also useful in ASP.NET MVC applications

.NET Design Patterns

Architectural Patterns and Styles

- Component-Based Architecture
- Layered Architecture
- .NET Assemblies and Versioning
- N-Tier Architecture
- Service-Oriented Architecture (SOA)

Architectural Patterns and Styles

Component-Based Architecture

- Component-based architecture describes a software engineering approach to system design and development
- Focuses on the decomposition of the design into individual functional or logical components that expose well-defined communication interfaces
- Provides a higher level of abstraction than object-oriented design principles

Component-Based Architecture

Mechanism

- Components depend upon a mechanism within the platform that provides an environment in which they can execute, often referred to as component architecture
  - COM, CORBA, EJB
- Component architectures manages the mechanics of locating components and their interfaces, and passing messages or commands between components
- Useful for developing a pluggable or composite architecture
  - Microsoft Composite Application Library for WPF

Architectural Patterns and Styles

Layered Architecture

- A layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other
- Functionality within each layer is related by a common role or responsibility
- Communication between layers is explicit and loosely coupled
- With strict layering, components in one layer can interact only with components in the same layer or the one directly below it
- Can be accomplished using .NET assemblies

Layered Architecture

Uses

- Consider the layered architectural style if…
  - You have existing layers that are suitable for reuse in other applications
  - Cleaner high-level separation so that teams can focus on different areas of functionality
  - You must support different client types

Architectural Patterns and Styles

.NET Assemblies and Versioning

- The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest
- If an assembly has a strong name, versioning will be enforced
  - The application will only run with the versions they were built with unless overridden by explicit version policy in configuration files
- Assemblies in the global assembly cache take precedence over assemblies in the application directory

.NET Assemblies and Versioning

Version Number Convention

- Assembly version numbers are in the form <major version>.<minor version>.<build number>.<revision>
- Major: Backward compatibility can not be assumed
- Minor: Significant enhancement with the intention of backward compatibility
- Build: Recompilation of the same source
- Revision: Intended to be fully interchangeable

Architectural Patterns and Styles

N-Tier Architecture

- An N-tier architectural style is a layered style with each segment being a tier that can be located on a physically separate computer
- Benefits include improved scalability, availability, manageability, and resource utilization
- Communication between tiers is typically asynchronous in order to support better scalability
- Can be accomplished using .NET Remoting, WCF, or Web API

N-Tier Architecture

Uses

- Consider an N-tier architecture if…
  - The processing requirements of the layers in the application differ such that processing in one layer could absorb sufficient resources to slow the processing of the other layers
  - If security requirements of the layers in the application differ

Architectural Patterns and Styles

Service-Oriented Architecture (SOA)

- SOA enables application functionality to be provided as a set of services and application that make use of those services
- Services use standards-based interfaces
- Can package business processes into interoperable services, using a range of protocols and data formats
- Can be accomplished using WCF or Web API

.NET Design Patterns

Designing with Patterns

- Objects are things with well-defined responsibilities
- Objects are responsible for themselves
- Encapsulation means any kind of hiding
- Abstract our variations in behavior and data
- Design to interfaces
- Think of inheritance as a method of conceptualizing variation, not for making special cases of existing objects
- Separate the code that uses an object from the code that creates the object
- Consider the testability of your code before coding it