Design Patterns

- A design pattern is a reusable solution to a commonly occurring problem. The pattern describes the process to solve a problem. A design pattern can be applied to a variety of software problems across domains and infrastructures. Patterns comply with and formalize best practices. Object-oriented design patterns typically include relationships among classes.
- Many design patterns exist, however only a few will be discussed and used in Programming 3.

The Model-View-Controller Design Pattern

- MVC was conceived in 1978 as a design solution to a problem. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user
- MVC has been widely adopted as an architecture for Internet applications using popular programming languages. Several web application frameworks have been created (such as ASP.NET MVC as well as Ruby on Rails) that enforce the pattern
- The MVC design pattern reduces complexity by removing unnecessary dependencies. This allows the code to be reusable without modification.

MVC Components

- The MVC design pattern is comprised of 3 components. The model, the view and the controller.

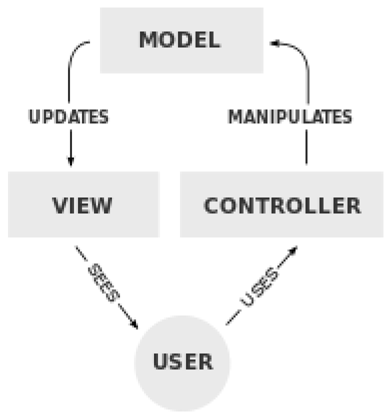


Figure 1- https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

The Model

- The model is a representation of data. The model could contain a single class, or a number of interrelated classes. The classes are used to store and manipulate state – typically in the form of a database. The model does not depend on the view or controller.

The View

- The view displays the model data (although it can display information not found in the model), and sends user actions (e.g. button clicks) to the controller.

The Controller

- The controller provides model data to the view, and interprets user actions such as button clicks or url requests.

Advantages of the MVC Pattern

- Separation of concerns. The separation the three components, allows the re-use of the business logic across applications. Lightweight user interfaces can be developed without concern of the business logic.
- Developer specialization and focus. The developers of Views can focus exclusively on the UI. The developer of Models can focus exclusively on the business logic. The developers of the Controllers can focus on responding to user actions.
- Change accommodation. User interfaces tend to change more frequently than business rules. As such the scope of the change can be limited to the Views.

Entity Framework

- Entity Framework (EF) is an object-relational model. It enables developers to work with relational data using domain-specific objects. The use of Entity Framework eliminates (or significantly reduces) code specifically written to access database data.
- Microsoft provides the following definition of Entity Framework:

  Entity Framework (EF) is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write. Using the Entity Framework, developers issue queries using LINQ, then retrieve and manipulate data as strongly typed objects. The Entity Framework's ORM implementation provides services like change tracking, identity resolution, lazy loading, and query translation so that developers can focus on their application-specific business logic rather than the data access fundamentals

  https://msdn.microsoft.com/en-us/data/aa937709.aspx

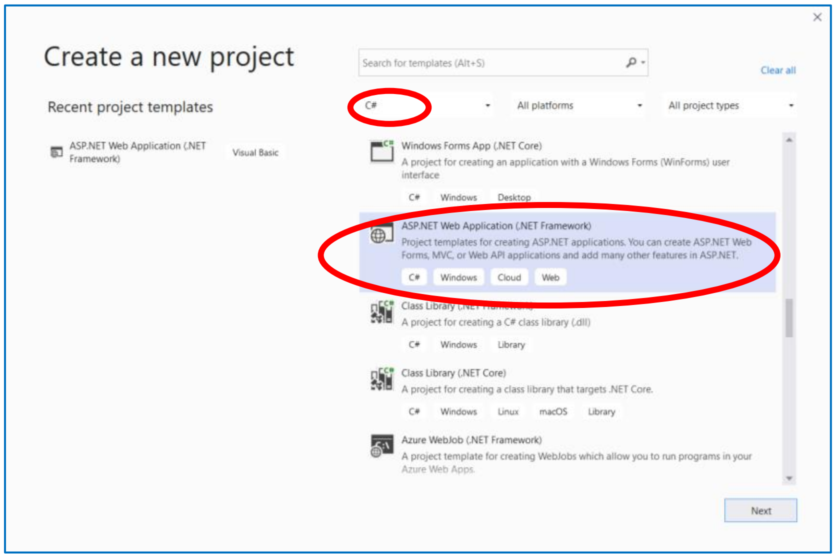
- Using Entity Framework, one can follow a:
  - Database-First Approach: Generate data access classes based on an existing database
  - Code-First Approach: Generate a database based on domain classes (used in this course).
  - Visual Design Approach: Design the database using a visual designer and create the database and classes based on that design

ASP.NET MVC

- ASP.NET MVC is included within the .NET Framework 2012 and later.
- The MVC framework is defined in the System.Web.MVC assembly
- ASP.NET MVC follows the Model-View-Controller design pattern, but does so within the context of .NET. As such, some of the behavior is not completely in line with the MVC pattern as will be noted throughout the course.
  - In addition, some domain-specific needs do not naturally fit within the MVC pattern. In those cases, deviations from the pattern will occur
- ASP.NET MVC framework is a lightweight, highly testable presentation framework that is integrated with existing ASP.NET features including
  - Master pages
  - Membership-based authentication

Create a new ASP.NET MVC Application

- Add a New Project to your solution. In the New Project dialog box:
- Choose the **C#** language
- Select the ASP.NET Web Application (.NET Framework) option and click **Next**



ASP.NET Web Application (.NET Framework) Visual Basic

Windows Forms App (.NET Core)
A project for creating an application with a Windows Forms (WinForms) user interface
C# Windows Desktop

ASP.NET Web Application (.NET Framework)
Project templates for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.
C# Windows Cloud Web

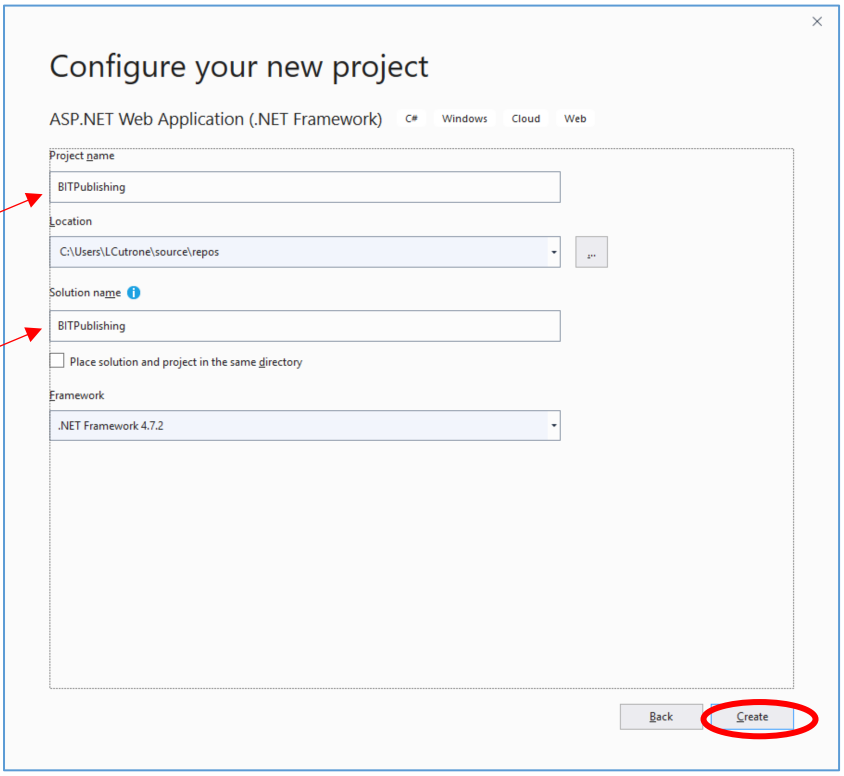
Class Library (.NET Framework)
A project for creating a C# class library (.dll)
C# Windows Library

Class Library (.NET Core)
A project for creating a class library that targets .NET Core.
C# Windows Linux macOS Library

Azure WebJob (.NET Framework)
A project template for creating WebJobs which allow you to run programs in your Azure Web Apps.

Figure 2

- Provide a **Project name**, **Location** and **Solution name**
- Select the default **Framework**
- Click **Create**



Configure your new project

ASP.NET Web Application (.NET Framework) C# Windows Cloud Web

Project name: BITPublishing

Location: C:\Users\LCutrone\source\repos

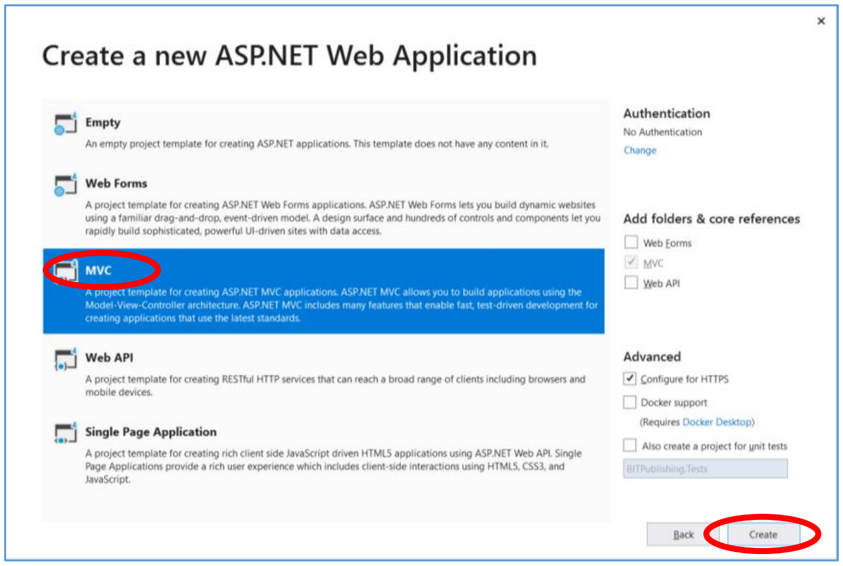
Solution name: BITPublishing

☐ Place solution and project in the same directory

Framework: .NET Framework 4.7.2

Figure 3

- Choose the **MVC** Project Type and click **Create**



Create a new ASP.NET Web Application

Empty
An empty project template for creating ASP.NET applications. This template does not have any content in it.

Web Forms
A project template for creating ASP.NET Web Forms applications. ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access.

MVC
A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Web API
A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.

Single Page Application
A project template for creating rich client side JavaScript driven HTML5 applications using ASP.NET Web API. Single Page Applications provide a rich user experience which includes client-side interactions using HTML5, CSS3, and JavaScript.

Authentication
No Authentication
Change

Add folders & core references
☐ Web Forms
☑ MVC
☐ Web API

Advanced
☑ Configure for HTTPS
☐ Docker support
(Requires Docker Desktop)
☐ Also create a project for unit tests
BITPublishing.Tests

Figure 4

- The directory structure within the project should be as shown including Models, Views and Controllers folders:

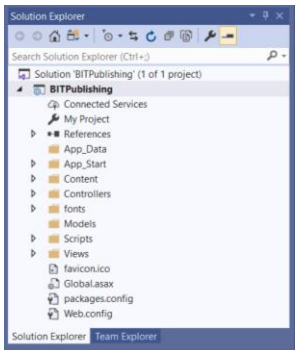


Figure 5

The ASP.NET MVC Model

- A model is a class with attributes and/or operations
  - The operations may be limited to Auto-Implemented properties (getters & setters), or may include more complex functionality
  - Some models will not contain attributes (e.g. inherited models containing only implementations for methods meant to be overridden).
    - These are known as Service Objects
- The model defines the data and relationships among data within the application. A model represents the state of a particular aspect of the application
- Models are objects used to send information to the database. They represent the domain of the application
- Models are objects one may wish to **CR**eate, **U**pdate **D**elete and **d**isplay (CRUDd)
- To create a new Model or set of Models, right click on the Models folder and choose **Add Class…**

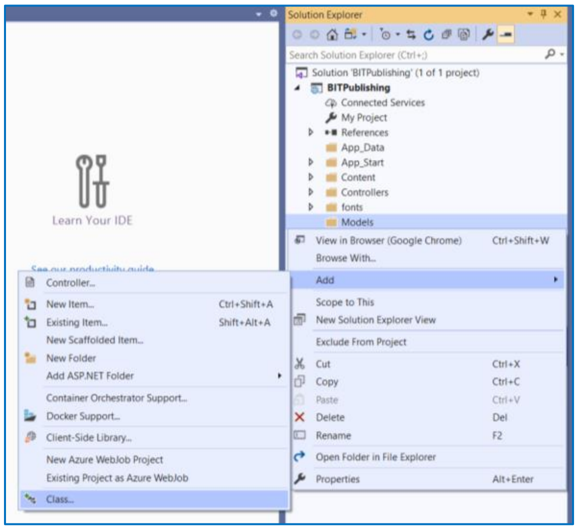
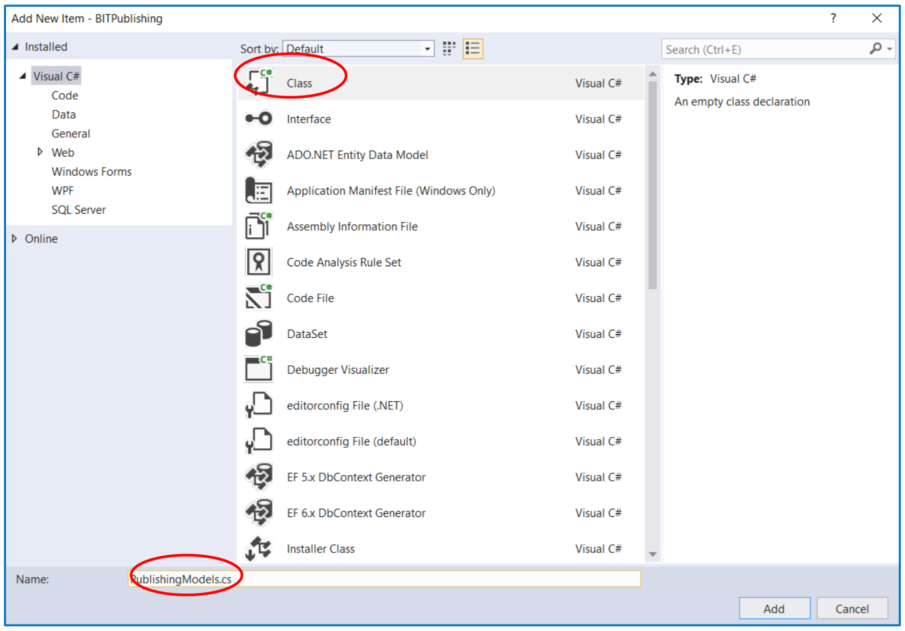
- 

Figure 6

- One or more models can be written in a single class file
- In this course, several classes will be written in the same file. As such, provide a **filename** that is indicative of the domain to which the classes belong


Add New Item - BITPublishing
Installed
Visual C#
Code
Data
General
Web
Windows Forms
WPF
SQL Server
Online
Sort by: Default
Search (Ctrl+E)
Class Visual C#
Interface Visual C#
ADO.NET Entity Data Model Visual C#
Application Manifest File (Windows Only) Visual C#
Assembly Information File Visual C#
Code Analysis Rule Set Visual C#
Code File Visual C#
DataSet Visual C#
Debugger Visualizer Visual C#
editorconfig File (.NET) Visual C#
editorconfig File (default) Visual C#
EF 5.x DbContext Generator Visual C#
EF 6.x DbContext Generator Visual C#
Installer Class Visual C#
Add
Cancel

Type: Visual C#

An empty class declaration

Name: PublishingModels.cs

Figure 7

Defining a Model

- To define a model, create a class within the Models folder of your MVC project
- Define the model (class) based on the class diagram provided
- Models can be created in individual C# files (that is, one file per class), or all models can be created within the same C# file (done in this course)
- The following class diagram will be used as demonstration:

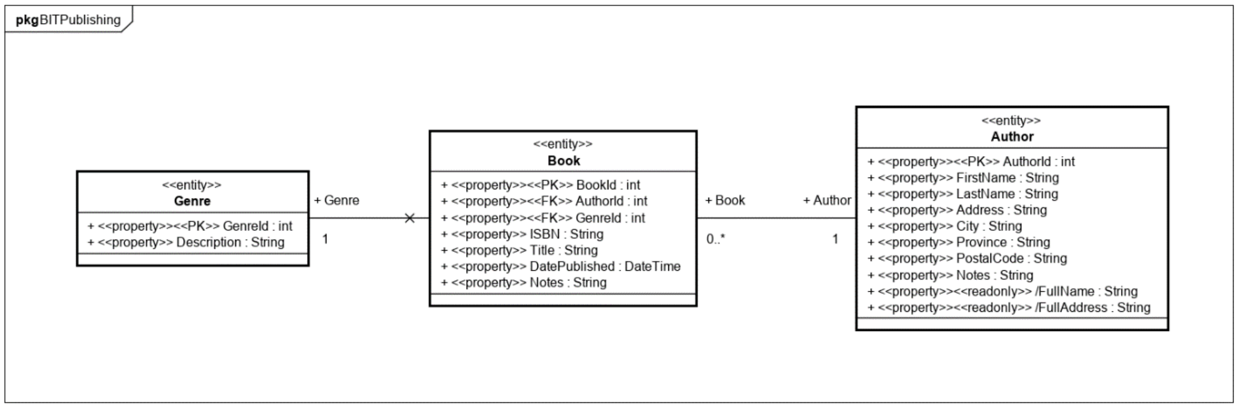

pkgBITPublishing
<<entity>>
Genre
+ <<property>><<PK>> GenreId : int
+ <<property>> Description : String
+ Genre
1
<<entity>>
Book
+ <<property>><<PK>> BookId : int
+ <<property>><<FK>> AuthorId : int
+ <<property>><<FK>> GenreId : int
+ <<property>> ISBN : String
+ <<property>> Title : String
+ <<property>> DatePublished : DateTime
+ <<property>> Notes : String
+ Book
0..*
+ Author
1
<<entity>>
Author
+ <<property>><<PK>> AuthorId : int
+ <<property>> FirstName : String
+ <<property>> LastName : String
+ <<property>> Address : String
+ <<property>> City : String
+ <<property>> Province : String
+ <<property>> PostalCode : String
+ <<property>> Notes : String
+ <<property>><<readonly>> /FullName : String
+ <<property>><<readonly>> /FullAddress : String

Figure 8

- The following code examples provide the class definition of the models as defined by the class diagram:

```
    public class Genre
    {
          public int GenreId {get;set;}
          public string Description{ get; set; }
    }
```

Code 1

```
public class Book
    {
          public int BookId{ get; set; }
          public int AuthorId{ get; set; }
          public int GenreId{ get; set; }
          public string ISBN{ get; set; }
          public string Title{ get; set; }
          public DateTime DatePublished{ get; set; }
          public string Notes{ get; set; }
    }
```

Code 2

```
public class Author
   {
         public int AuthorId{ get; set; }
         public string FirstName{ get; set; }
         public string LastName{ get; set; }
         public string Address{ get; set; }
         public string City{ get; set; }
         public string Province{ get; set; }
         public string PostalCode{ get; set; }
         public string Notes{ get; set; }
         public string FullName()
         {
               return String.Format("{0} {1}", FirstName, LastName);
         }
         public string FullAddress()
         {
               return String.Format("{0} {1}", Address, City);
         }
   }
```

Code 3

Data Annotations

- Data annotations are metadata that help to describe the Models or the attributes within the models.
- Data annotations require the use of the following namespaces:
  - System.ComponentModel.DataAnnotations – for data annotations
  - System.ComponentModel.DataAnnotations.Schema – for identity field annotation
- Data annotations appear above the property to which they apply. Multiple data annotations can be applied to the same property. They will apply to the next property that appears below the annotation.
- Some of the annotations are used to help generate the appropriate behavior in the views (e.g. data validation). For example:
  - Ensuring data is appropriately entered into a field (e.g. within a range) – Annotated with `[Range (0,10)]`
- Some of annotations are used to clarify database relationships (if it cannot be readily inferred) For example:
  - Primary keys annotated with `[Key]`
    - This will be used for super classes such that the super class will produce a table and the sub classes will produce rows in that table.
  - Foreign keys – annotated with `[ForeignKey("navigationPropertyName"]`
  - Not null – annotated with `[Required (ErrorMessage = "Optional Error Message")]`
- Annotations can also be used to control the display of data in the views
  - Headings whose name does not match the attribute name – annotated with `[DisplayName("Heading Format")`

        - **Note**: escape characters may be used to cause line feeds when the DisplayName is eventually shown in the corresponding View(s)

          ```
          [DisplayName("Heading\nFormat")
          ```
    - Formatting of Date data based on a predefined format can be decorated with the following annotation:

      ```
      [DisplayFormat(DataFormatString = "{0:MMM d, yyyy}")]
      ```
    - Preventing the display of certain attributes can be handled using the following annotation:

      ```
      [ScaffoldColumn(false)]
      ```
- A single annotation can be used to apply formatting for display as well as data entry in the View(s)
    - Formatting of Currency data based on a predefined format – and requiring that format for data entry in the views

      ```
      [DisplayFormat(ApplyFormatInEditMode = true,
                     DataFormatString = "{0:c}")]
      ```
- Regular expressions can be applied to an annotation to limit data entry to those values that match the regular expression
    - Formatting a string to be 2 uppercase characters

      ```
      [RegularExpression("[A-Z][A-Z]")]
      ```
- All annotations have an ErrorMessage= overload such that custom error messages can be defined for display when the user violates the rules of the annotation.
- Custom data annotations can also be created if the predefined annotations do not meet a particular need in an application.
- The Author class defined above, could be decorated with data annotations as follows (**Note:** these are just examples of a few annotations, more could be applied):

```
public class Author
   {
         [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]
         public int AuthorId{ get; set; }

         [Required]
         [StringLength(35, MinimumLength = 1)]
         [Display(Name = "First\nName")]
         public string FirstName{ get; set; }

         [Required]
         [StringLength(35, MinimumLength = 1)]
         [Display(Name = "First\nName")]
         public string LastName{ get; set; }

         [Required]
         [StringLength(300, MinimumLength = 1)]
         [Display(Name = "Street\nAddress")]
         public string Address{ get; set; }

         [Required]
         public string City{ get; set; }
```

```
        [Required]
        [StringLength(2)]
        [RegularExpression("[A-Z][A-Z]")]
        public string Province{ get; set; }

        [Required]
        public string PostalCode{ get; set; }

        public string Notes{ get; set; }

        public string FullName()
        {
              return String.format("{0} {1}", FirstName, LastName);
        }
        public string FullAddress()
        {
              return String.format("{0} {1}", Address, City);
        }
    }
```

Code 4

- Only a few annotations have been discussed here. For a full list of Data Annotations please visit the System.ComponentModel.DataAnnotations namespace documentation page.

Navigational Properties

- Defining relationships between classes, and eventually between relational database tables are achieved through the use of navigational properties. This is achieved using an instance of the class to which the navigation line points on the class diagram.
- Navigational Properties are represented by the line on the class diagram.
- Consider the following class diagram

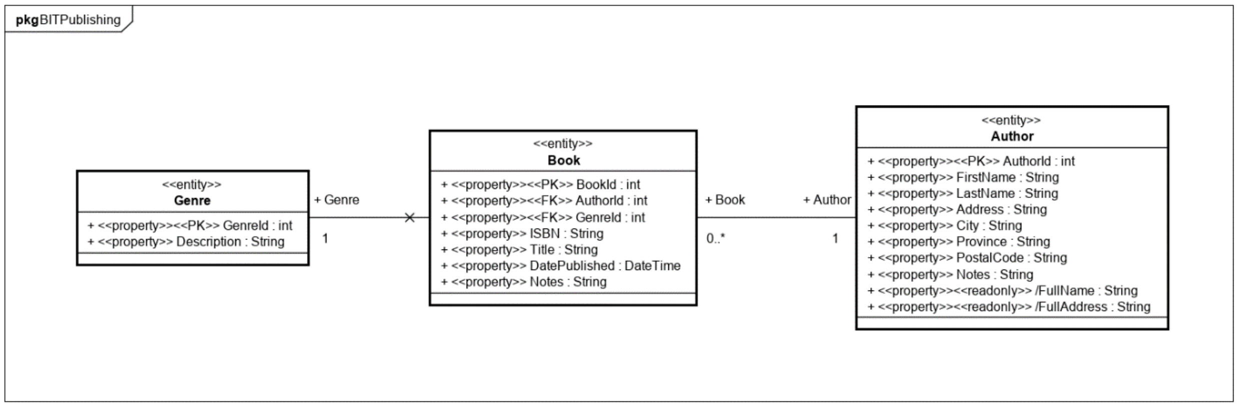

pkgBITPublishing
<<entity>>
Genre
+ <<property>><<PK>> GenreId : int
+ <<property>> Description : String
+ Genre
1
<<entity>>
Book
+ <<property>><<PK>> BookId : int
+ <<property>><<FK>> AuthorId : int
+ <<property>><<FK>> GenreId : int
+ <<property>> ISBN : String
+ <<property>> Title : String
+ <<property>> DatePublished : DateTime
+ <<property>> Notes : String
+ Book
0..*
+ Author
1
<<entity>>
Author
+ <<property>><<PK>> AuthorId : int
+ <<property>> FirstName : String
+ <<property>> LastName : String
+ <<property>> Address : String
+ <<property>> City : String
+ <<property>> Province : String
+ <<property>> PostalCode : String
+ <<property>> Notes : String
+ <<property>><<readonly>> /FullName : String
+ <<property>><<readonly>> /FullAddress : String

Figure 9

  - To simplify this example, assume that each Book can have only one Author, but an Author can write more than one Book.
  - In this example, the AuthorId and the GenreId properties in the Book class are foreign key properties.

- In this example the association lines from Book to Author and Genre are navigational properties.
- The navigational property is needed to satisfy the object-oriented relationship between the classes.
- The foreign key property is needed to satisfy the foreign key relationship in the eventual relational database.
- This is an example of the object-relational model as was discussed in an earlier topic. See Entity Framework.
- Navigational properties may be used to support the Entity Framework concept of Lazy Loading:
  - With lazy loading, Entity Framework automatically loads a related entity when the navigation property for that entity is dereferenced.
  - To enable lazy loading, make the navigation property ***virtual***. For example, in the Book class:

```
public class Book
{
    // (Other properties)


    //navigational properties
    public virtual Author Author{ get; set; }
    public virtual Genre Genre{ get; set; }
}
```

Code 5

- When evaluating the Navigation lines, pay particular attention to the cardinality. For example, an Author can write many books.
- To indicate this “many” cardinality, define the navigation property as an ICollection:

```
public class Author
{
    // (Other properties)


    //navigational properties
    public virtual ICollection<Book> Book{ get; set; }
}
```

Code 6

Reference Material

https://en.wikipedia.org/wiki/Software_design_pattern
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
http://www.tomdalling.com/blog/software-design/model-view-controller-explained/
http://www.codeproject.com/Articles/585873/Basic-Understanding-On-ASP-NET-MVC
https://msdn.microsoft.com/en-us/data/aa937709.aspx
http://www.entityframeworktutorial.net/what-is-entityframework.aspx
https://en.wikipedia.org/wiki/ASP.NET_MVC_Framework
http://www.codeproject.com/Articles/476967/WhatplusisplusViewData-2cplusViewBagplusandplusTem
http://www.asp.net/web-api/overview/data/using-web-api-with-entity-framework/part-4
https://sourcemaking.com/design-patterns-and-tips