AT-10402
LARGE
BOARD GAME SLEEVES
100 Board Game Card Sleeves
Size: Large
(for cards measuring up to 59x92mm)

BoardGameSleeves.com

ARCANE TINMEN

Brian Munksgaard & Jens Christian Rasch | EAAA Backend | December 2016

CONTENT

INTRODUCTION 1
ASP.NET MVC 2
Architecture 2
Asp.Net MVC Components 2
Conventions 4
IMPLEMENTATION 6
Visual ELEMENTS 6
Page layout 6
Sketches 7
List Sleeves View (public area) 8
Cart Information (public area) 8
Checkout View (public area) 9
Product details view (public area) 9
Product list view (admin area) 10
Create product view (admin area) 10
Grid 11
Uml Diagrams 11
Domain Model 11
Mapping between controllers and views 11
Layout and Views 12
ListViewModels and PartialViews 13
Layouts 13
Routes 14
Model validation 14
Entity Framework 16
Database migration 17
Seeding database data 17
REPOSITORIES 18
CONCLUSION 20

INTRODUCTION

Arcane Tinmen are in the process of updating their online presence and as part of this work, they want to split their current website up, so that they have different websites for each of their brands. To begin this work, they have asked for an updated website for the brand Board Game Sleeves, to which they have acquired a corresponding domain name (boardgamesleeves.com).

This paper will contain two parts where the first will go through the aspects of an ASP.NET MVC website, the components and conventions that ASP.NET MVC is built upon as well as some theory behind building a website using this framework. The second part of the paper will be a detailed description of how we have used these components and conventions in our implementation of a new website for boardgamesleeves.com as well as a walkthrough of the functionality that we have implemented.

Along with this paper, we have included the code for the website (in a Visual Studio solution) as well as the database with some default data (for testing purposes).

ASP.NET MVC

As mentioned in the introduction we will use the ASP.NET MVC framework to create the Boardgamesleeves.com website. In the following sections, we will describe the components that comprise the framework and some conventions that the framework builds upon.

ARCHITECTURE

ASP.NET MVC is built upon an architecture that is created to help structuring the code being developed as well as support the DRY principal. The architecture is using "convention above configuration" which means that it is using conventions for the folder structure and naming of the different components.

An MVC website consists of 3 fundamental components:

- Controllers
- Views
- Models

Models represents the data of the application along with any business logic there might be.

Views are templates for generating dynamic HTML for showing the UI of the application. They define how the look and feel of the website should be and how the different components should be placed in relation to each other. A view can either be loosely or strongly coupled. A strongly coupled view means that there is a tight coupling with the model, which the view can use in its layout and displaying of data.

Controllers are the coupling between the models and the views. Controllers handle the HTTP requests as well as fetching the data from the models and sending the relevant data to the views. Controllers can also contain logic to choose between different views or to map Model data to a ViewModel.

ASP.NET MVC COMPONENTS

Apart from the three basic components of an MVC application mentioned above, several other components are a part of the ASP.NET MVC framework. In the table below, we will briefly mention these and explain their usage as well as the reason for them being in the framework.

Component	**Description**
Controller	Controllers are the binding between Views and Models. The controllers handle the incoming URL requests, fetch the data through the models and send the relevant data through to the Views. The controllers can also contain logic for choosing which View to show or mappings between Models and ViewModels.
View	Views are templates for creating dynamic content. They define how the pages look and

	how the different components should be placed relative to each other.
Partial View	A partial view contains small parts that can be included inside a View. It is especially useful for different parts of a website that is used multiple places, like a shopping cart.
HTML Helpers methods	HTML Helpers are a way of combining parts of markup and code in such a way that they can be reused throughout the MVC application. There are 2 types of helpers: Inline, Custom and Built-in helpers. Inline helpers are all created inside the view where they are being used, and are specified by using the @helper tag. Custom helpers are helper methods that you write yourself and can reuse in multiple Views. These helpers are created either by creating an extension method for the HtmlHelper class or by creating a static method in a utility class. Built-in helpers are helper methods that are delivered as part of the MVC framework to make creating views easier for the developer. These helpers exist in 2 forms: Standard helpers and Strongly Typed helpers. The strongly typed helpers have a strong coupling between the model and the field in the View.
Child Actions	A child action is a controller method that you can call from inside a View and you can set it up so you can only call this type of method from within Views. Another big difference between child actions a "normal" controller action is that a child action only return a small part of some markup whereas a "normal" action can return an entire View. Child actions are a good way to create widgets or smaller parts of a markup, that can be reused through a website.
Routes	The routing system in ASP.NET MVC specifies how incoming HTTP requests will be handled and how they are directed to the different controllers. The different routes can be specified in the RouteConfig.cs file and can contain any number of different routes for the website. We will describe this in more detail when we describe the routes that is set up for BoardGameSleeves.com later in this paper.
Models & ViewModels	Models represent the data that the website is working on, along with any business logic there

	might be. The difference between Models and ViewModels is that ViewModels are centered on what is to be shown on a View where the Models are centered on the persisted data in the domain.
Layouts	It is possible in an ASP.NET MVC to specify layouts as part of a single page, or have pages use a common shared layout. No matter what is used, the layout specifies how the different components are positioned on the individual HTML pages that the controller send back to the browser.
Model Binding (IModelBinder)->State	Model binding is the mapping of data from the HTTP Request to parameters used in the action methods of a Controller. Binding can be done to both simple and complex types. It is possible to create your own model binder logic in classes that extends the IModelBinder interface.
Areas	Areas are a way to organize a large project or website into smaller and manageable units. An application can contain multiple areas each defining a small portion of the entire application. An example of areas could be the shopping system or the administration system of a website.

As mentioned earlier the table above is only a short mention of the different components that are part of the ASP.NET MVC framework. In the rest of this paper, we will describe in more detail, how we have used these components and why we have chosen to use the components that we have.

CONVENTIONS

As we mentioned in the section about architecture, the ASP.NET MVC framework is using a couple of naming and folder conventions that makes the building of ASP.NET MVC websites easier from a developer perspective. These conventions are not something that you have to follow as a developer, but by following them, some functionality will work out of the both without the developer needing to think about it or create specific handling of it in the application.

An MVC application has by default a folder structure like the one below. As with all other convetions it is not a structure that you as a developer has to follow.

Folder	Purpose
/Controllers	All controller classes that handles HTTP requests are placed here.
/Models	All classes that handles data and business logic are placed in this folder.
/Views	HTML templates for handling of the UI is placed in this folder.

/App_Data	If the project needs any data files they can be written to and read from this folder.
/App_Start	Any configuration for things like routing, bundling and web api can be placed in this folder.
/Scripts	This folder contains all the JavaScript files and scripts which the website uses.
/Content	Any files like CSS, image or other content used by the website can be placed in this folder.

By default if you follow the folder conventions the framework will automatically be able to figure out which Views corresponds to the Actions in the controllers. That means that the developer will not have to set up any system for making sure the correct View is returned, but can rely on the framework to figure this out.

Another convention is the naming of the controllers. A controller can be named by combining the name of the Model that the controller references with the word “Controller”. By using this convention, the framework and routing system can automatically connect a HTTP request for a URL like http://mysite.com/Product/Index to the Index action of a Controller called ProductController.

Tightly coupled with the controllers is the naming of the views. The convention specifies that you name the view like the corresponding action method in the controller, and you place it in the Views folder in a subfolder named after the relevant controller. This will make it possible for the framework to locate the correct view automatically, when an action on a controller is being invoked.

IMPLEMENTATION

In the rest of this paper, we will describe how we have used the conventions and components to implement the new website for BoardGameSleeves.com. We will give examples of how we have solved different aspects concerning the conventions and components, as well as describe the decisions we have made throughout the project.

For clarity, we have included the final product (in a Visual Studio solution) with this paper.

As part of our implementation of the BoardGameSleeves.com website we have implemented the following functionality:

- A page showing a list of the products that the website has for sale.
- A shopping functionality implemented on the product page.
- A shopping cart making it possible for customer to maintain their shopping cart.
- A checkout page where customer can enter their shipping information.
- An administration area where information about the products can be maintained.
- Administration functionality for maintaining Games and the sleeve size that fit them.

VISUAL ELEMENTS

When building a web site one should carefully consider the visual elements on the site. Depending on:

- The audience/users.
- The device (desktop/mobile/etc.) being used to access the site.
- The usage context.
- The actual site content.
- The message you want to send.

We can determine color, typography, visual hierarchy and which gestalt principles to use. Also, a style guide and a site CSS file should be created.

However, since this is beyond the scope of this project, we have decided to use the all the standards in the Bootstrap CSS library, including the Bootstrap grid system. This way we can quickly get a fair UI up and running. You might consider this project a prototype of the system where the visual branding will be done later in the project.

PAGE LAYOUT

To determine the general page layout, we used UI design sketches. From the sketches we can see that a page has a header, a body and a footer.

- The header contains the site logo, site title and, if applicable, shopping cart info. In the site administration area, the cart info is not displayed. Clicking on the cart will take us to the cart page.
- The footer currently only contains a simple copyright statement. The footer could also be used to hold contact information and references to Arcane Tinmen social media pages.

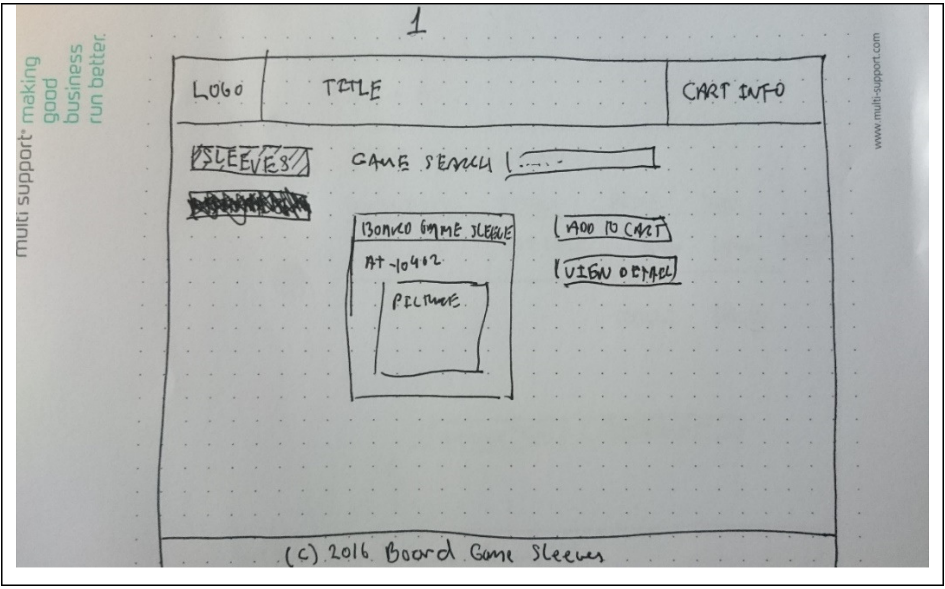
- The body has two parts, navigation to the left and content to the right. The navigation buttons are based on the different product categories. Currently, the only category is ‘Sleeves’.

SKETCHES

During the sketching phase of this project, we came up with the following layouts for the website. Mind you that these are only sketches and are work in progress. The final product could look very different from these. The sketches may be considered an early paper based prototype. They are used to get us started.

List Sleeves View (public area)

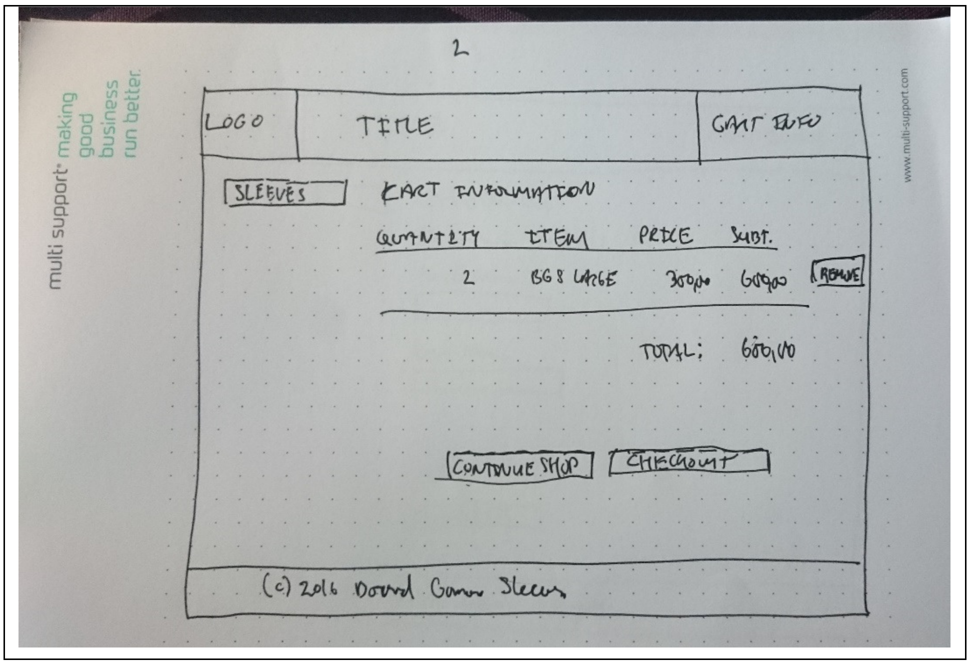
This view is used to list board game sleeves. Using the ‘Game Search’ textbox the view can be filtered to list only the sleeves that matches the search criteria. The list will show basic information and an image for each product. Next to each product there are two buttons, ‘Add to Cart’ and ‘View Details’.



When adding a product to the cart, the cart is updated, but the page is not changed.

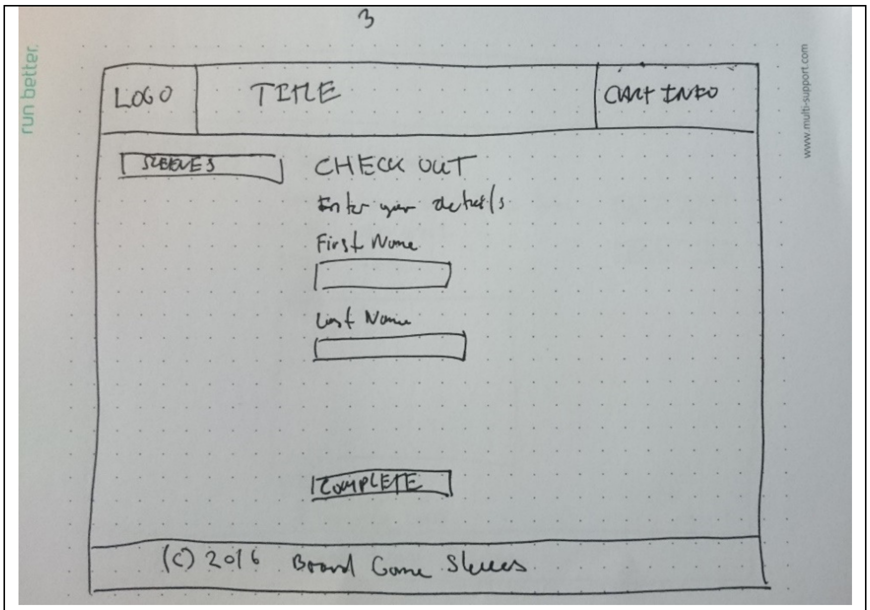
Cart Information (public area)

This view is used to list the items in the cart. From here we can remove an item from the cart. From this view we can also continue shopping or go to the checkout page.



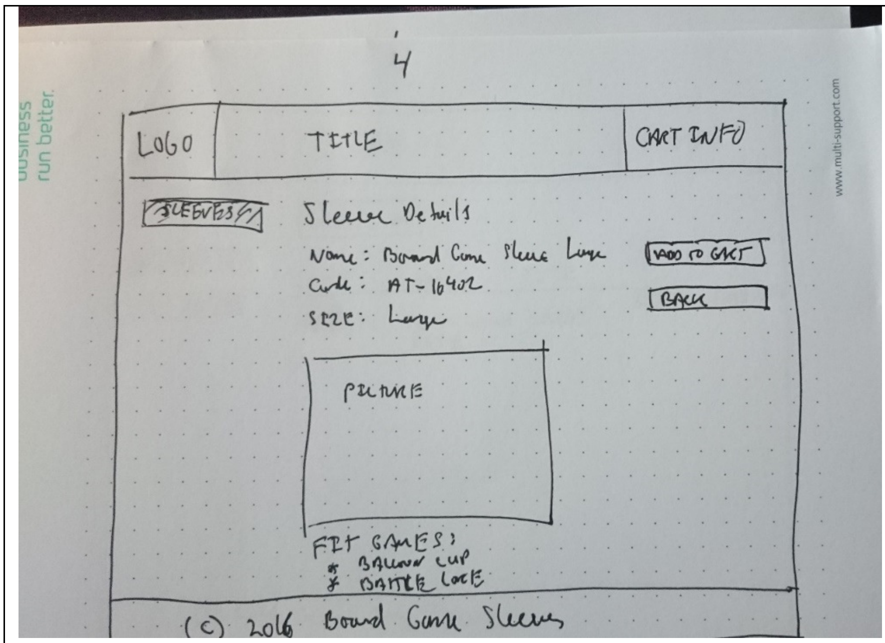
Checkout View (public area)

In the checkout view, the user enters the shipping details. When the ‘Complete’ button is clicked a receipt page is displayed. The checkout view should probably also have a ‘Continue Shopping’ button.



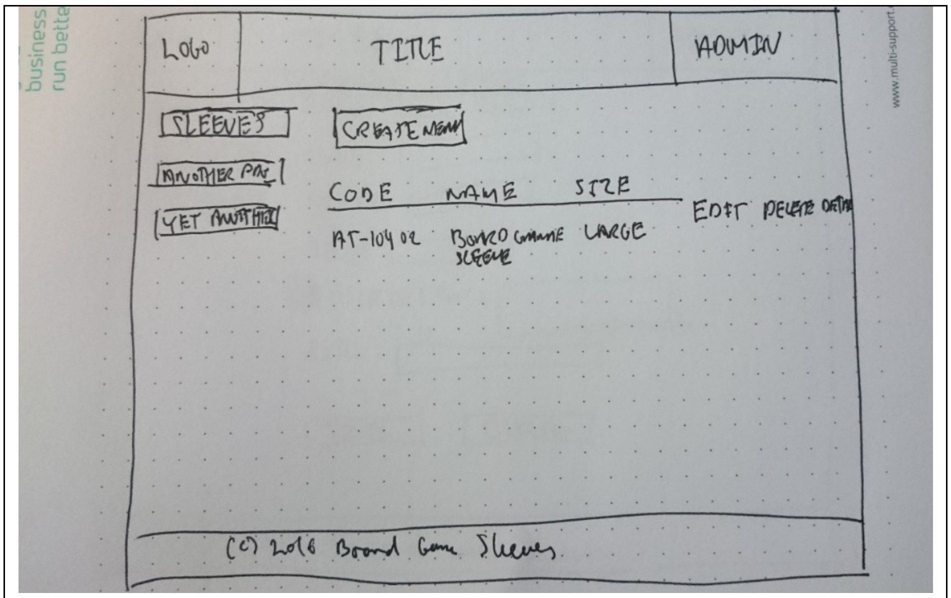
Product details view (public area)

This view is used to display detailed product information including a list of games that matches the current product (sleeve). In the view there are two buttons, ‘Add to Cart’ and ‘Back’. A click on either takes the user back to the list view. Of course, when clicking the ‘Add to Cart’, the product is first added to the cart before returning to the list view.



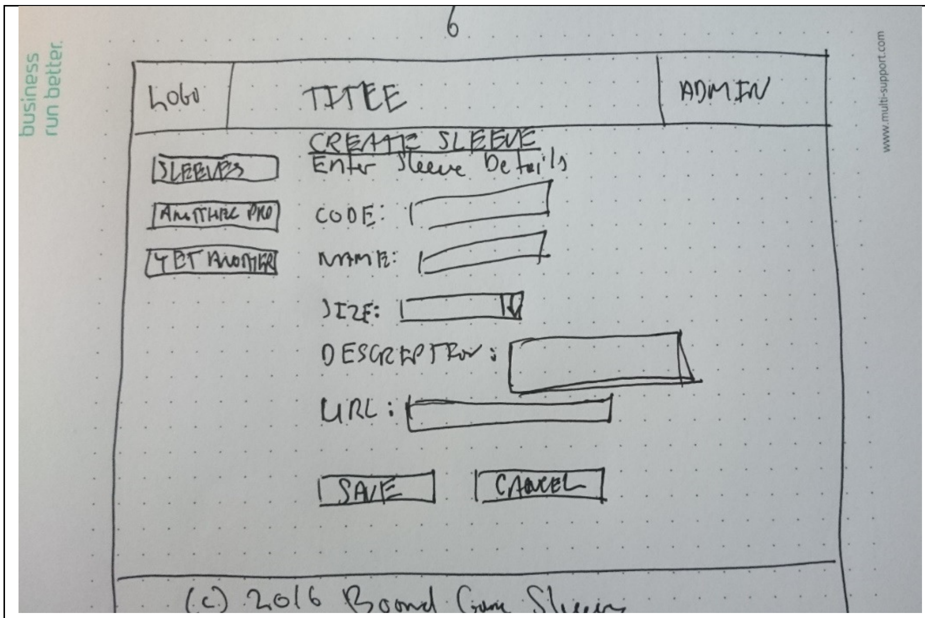
Product list view (admin area)

In the product list view in the administration area, an admin user can add, edit, display or delete an item within the different product categories. Please remember that we have currently only one category ‘Sleeves’.



Create product view (admin area)

As the view title implies, this view is used to create new products within the current product category.



Grid

The sketches lead us to the following grid:

col-md-12	
col-md-2	col-md-10
col-md-12	

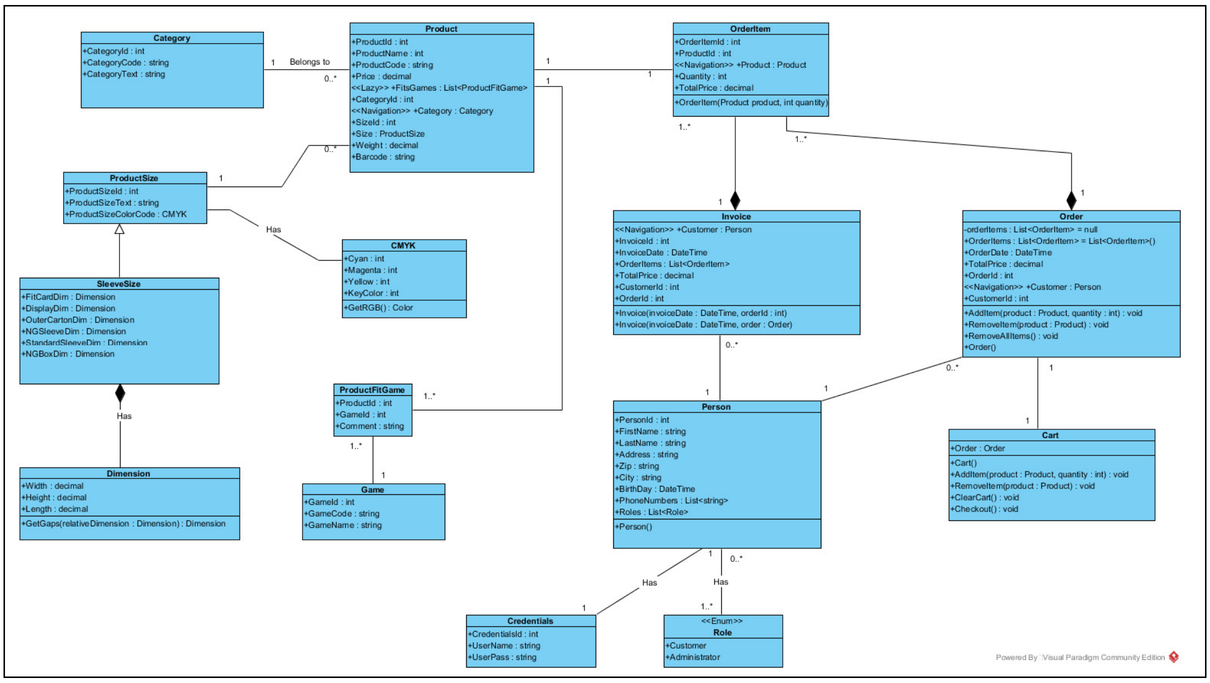
In the html code we use the Bootstrap CSS classes col-md-X to define the grid.

UML DIAGRAMS

From the site requirements, the sketches and the web stuff from Arcane Tinmen, we have built the following models and diagrams:

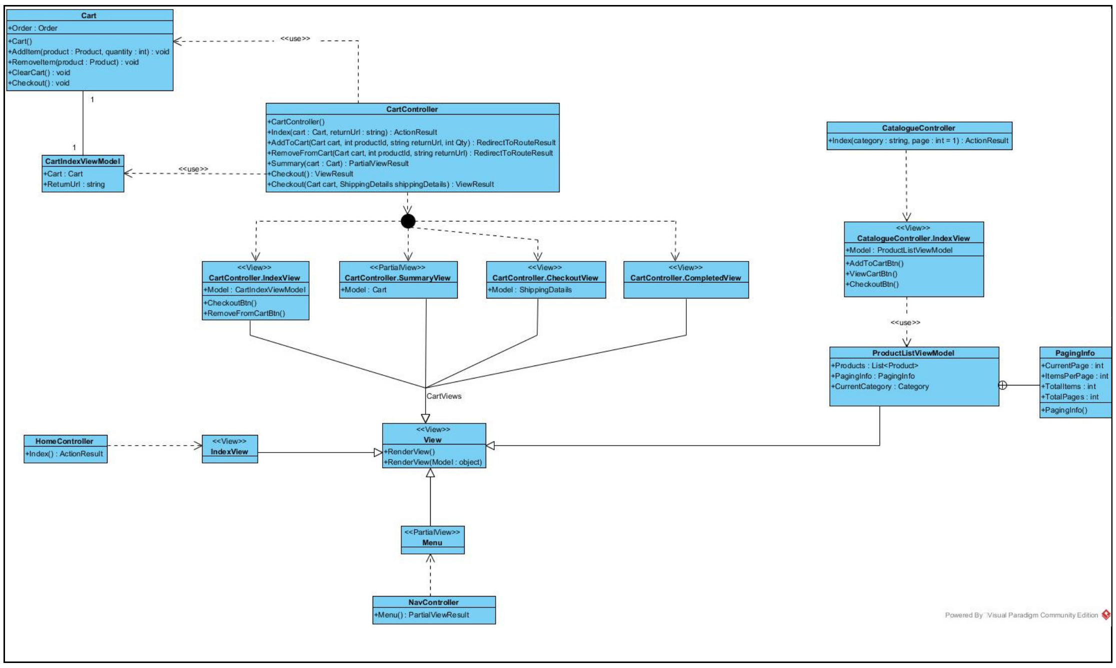
Domain Model

Below you can see the domain model for the new website.



Mapping between controllers and views

We have chosen to map the controllers and views like expressed in the image below.


Cart
+Order : Order
+Cart()
+AddItem(product : Product, quantity : int) : void
+RemoveItem(product : Product) : void
+ClearCart() : void
+Checkout() : void
<<use>>
1
1
CartIndexViewModel
+Cart : Cart
+ReturnUrl : string
<<use>>
CartController
+CartController()
+Index(cart : Cart, returnUrl : string) : ActionResult
+AddToCart(Cart cart, int productId, string returnUrl, int Qty) : RedirectToRouteResult
+RemoveFromCart(Cart cart, int productId, string returnUrl) : RedirectToRouteResult
+Summary(cart : Cart) : PartialViewResult
+Checkout() : ViewResult
+Checkout(Cart cart, ShippingDetails shippingDetails) : ViewResult
CatalogueController
+Index(category : string, page : int = 1) : ActionResult
<<View>>
CatalogueController.IndexView
+Model : ProductListViewModel
+AddToCartBtn()
+ViewCartBtn()
+CheckoutBtn()
<<use>>
<<View>>
CartController.IndexView
+Model : CartIndexViewModel
+CheckoutBtn()
+RemoveFromCartBtn()
<<PartialView>>
CartController.SummaryView
+Model : Cart
<<View>>
CartController.CheckoutView
+Model : ShippingDatails
<<View>>
CartController.CompletedView
ProductListViewModel
+Products : List<Product>
+PagingInfo : PagingInfo
+CurrentCategory : Category
PagingInfo
+CurrentPage : int
+ItemsPerPage : int
+TotalItems : int
+TotalPages : int
+PagingInfo()
CartViews
HomeController
+Index() : ActionResult
<<View>>
IndexView
<<View>>
View
+RenderView()
+RenderView(Model : object)
<<PartialView>>
Menu
NavController
+Menu() : PartialViewResult
Powered By Visual Paradigm Community Edition

The superclass View with the methods RenderView and RenderView(Model: object)

View classes where buttons/links etc. are shown as function/methods. View data is shown as properties.

A controller is an ordinary class.

This gives us the possibility to show which controllers are using the different views.

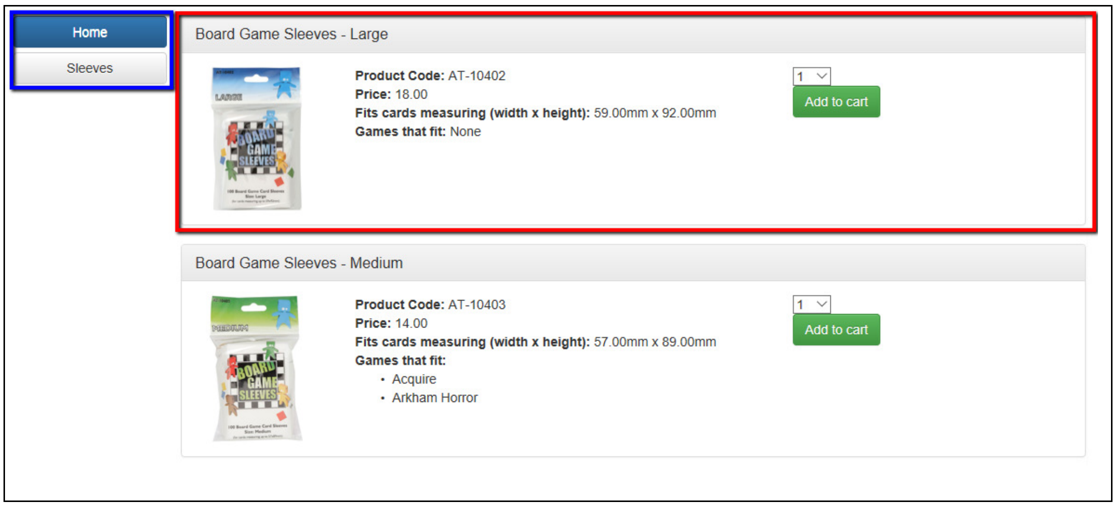
LAYOUT AND VIEWS

To maintain a better control of all the Views and Controllers in our solution we have decided to take advantage of something in ASP.NET MVC called Areas. This will in essence create a MVC file structure in each Area and therefore create a separation of the files so that we can make change to one area of the website without it interfering with the other.
The two areas we have decided to create are Admin and Public. The Public area will be the website itself including the shop functionality while the Admin area will include all the functionality for maintaining the data behind the website.

Each area will have its own URL layout. To access the Public area you would need to call http://server/Public/Catalogue. This will bring the customer to a list of products in the website.
To access the Admin area you would need to use an URL like http://server/Admin/Products. This will bring you to a page where all the products in the website can be maintained and new products can be created. The administration also contains functionality for maintaining games and which sleeves that fit the games.

ListViewModels and PartialViews

To show the list of products in Public area we have used a ListViewModel instead of using the products model directly. We have done this because we needed to make sure we had information about the selected category along with information needed for our paging of products. The ListViewModel is then sent to the Index view of the Catalogue controller but the actual displaying of information about the individual products are done through the use of an inline HTML helper which in turn uses a partial view called ProductSummary to create the actual HTML needed. The result shown to the customer is like the image below. The part surrounded by the red box is the HTML built by the partial view, and the menu in the blue box is from another partial view that only handles the displaying of the menu for the website.



Board Game Sleeves - Large

Product Code: AT-10402
Price: 18.00
Fits cards measuring (width x height): 59.00mm x 92.00mm
Games that fit: None

Board Game Sleeves - Medium

Product Code: AT-10403
Price: 14.00
Fits cards measuring (width x height): 57.00mm x 89.00mm
Games that fit:
- Acquire
- Arkham Horror

The location of all these views, partial views and ListViewModels are all created using the naming and location conventions that ASP.NET MVC specifies.

The reason we use a partial view to create the actual HTML needed to show the product information is because it is a small part of HTML that can be reused throughout the website and as such a partial view is more efficient than having to implement modifications multiple places every time something changes.

Layouts

Common for both areas are that they make use of a shared layout in all the views. This makes it possible for us to implement a common look and feel for each area and as with the use of partial views we only need to make changes in one file instead of in each view.

The reason for us to use separate layout for each of the areas instead of using one shared layout for them both is that this gives us the flexibility of changing layout for each area without the changes influencing the other area.

ROUTES

In the Admin area we have also implemented two extra routes beside the default route. All three can be seen below.

```
context.MapRoute(
    name: "Admin_Products",
    url: "Admin/Products/{action}/{category}/{id}",
    defaults: new { controller = "Products", action = "Index", category =
 "Sleeves", id = UrlParameter.Optional },
    constraints: new { category = @"\w+" }
);

context.MapRoute(
    name: "Admin_ProductGameRelation",
    url: "Admin/ProductFitGames/Delete/{gameId}/{productId}",
    defaults: new { controller = "ProductFitGames", action = "Delete" }
);

context.MapRoute(
    "Admin_default",
    "Admin/{controller}/{action}/{id}",
    new { action = "Index", id = UrlParameter.Optional }
);
```

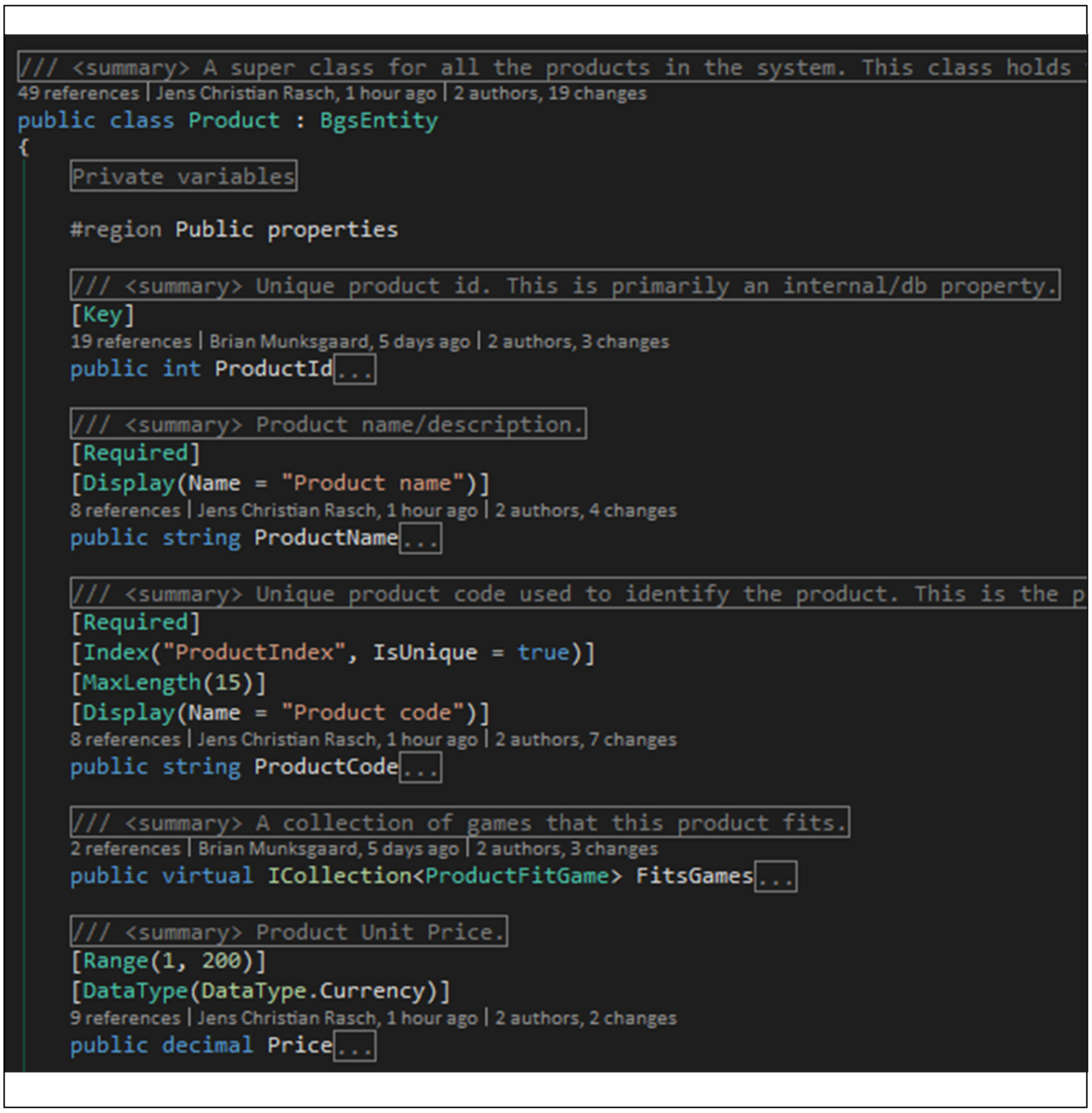
The route Admin_Products is created to make it possible to specify the product category directly in the URL. For the Public area we have chosen to implement this in a way where the category is given as an URL parameter instead.

The Admin_ProductGameRelation route is created to handle the deletion of the relationships between Games and Products. We have chosen to do it this way because the deletion needs two ID's to be able to delete the correct relationship and by creating this route both ID's can be specified as part of the URL directly.

MODEL VALIDATION

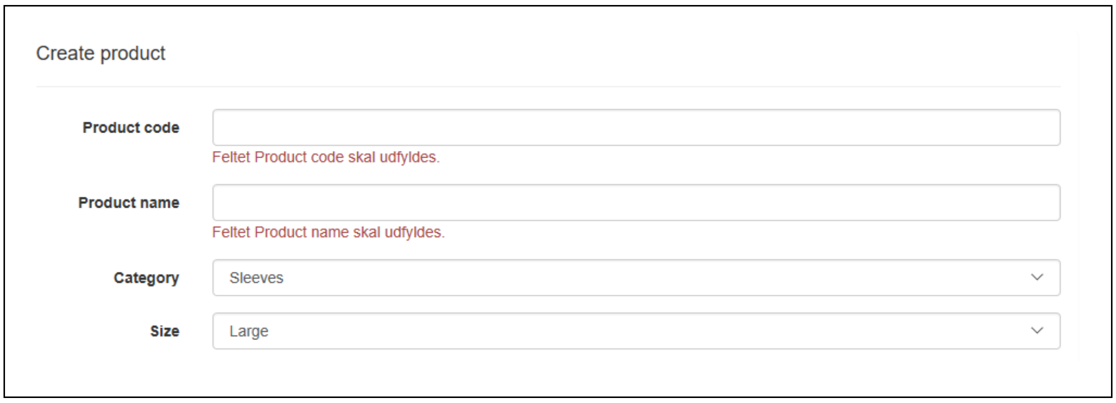
During the work of modelling our domain to model classes, we have used data annotation to help with model validation. Some classes have more annotations that other, but almost all of our models have some annotations.

The annotations that we have used has primarily been in the model classes, which are used in the Administration part of the website. Here the annotations has been a tool for validating the model during creation and editing of data. An example of how we have used these annotations for model validation can be seen in the code snippet below.



```
/// <summary> A super class for all the products in the system. This class holds
49 references | Jens Christian Rasch, 1 hour ago | 2 authors, 19 changes
public class Product : BgsEntity
{
    Private variables

    #region Public properties

    /// <summary> Unique product id. This is primarily an internal/db property.
    [Key]
    19 references | Brian Munksgaard, 5 days ago | 2 authors, 3 changes
    public int ProductId...

    /// <summary> Product name/description.
    [Required]
    [Display(Name = "Product name")]
    8 references | Jens Christian Rasch, 1 hour ago | 2 authors, 4 changes
    public string ProductName...

    /// <summary> Unique product code used to identify the product. This is the p
    [Required]
    [Index("ProductIndex", IsUnique = true)]
    [MaxLength(15)]
    [Display(Name = "Product code")]
    8 references | Jens Christian Rasch, 1 hour ago | 2 authors, 7 changes
    public string ProductCode...

    /// <summary> A collection of games that this product fits.
    2 references | Brian Munksgaard, 5 days ago | 2 authors, 3 changes
    public virtual ICollection<ProductFitGame> FitsGames...

    /// <summary> Product Unit Price.
    [Range(1, 200)]
    [DataType(DataType.Currency)]
    9 references | Jens Christian Rasch, 1 hour ago | 2 authors, 2 changes
    public decimal Price...
```

Here we have a section of the Product model where we have specified that the property ProductName and ProductCode are both required fields, which will make ASP.NET MVC automatically make the model invalid in any create or update request if any of these two fields are left empty (as can be seen from the screen dump below).
Another annotation we have used is the DataType for the property Price, to let Entity Framework know specifically that this property will contain currency information and not just a decimal number.


Create product

Product code	Feltet Product code skal udfyldes.
Product name	Feltet Product name skal udfyldes.
Category	Sleeves
Size	Large

One annotation that we have used in a lot of model is the Display(Name = “”) annotation. This gives us the possibility to specify what label should be shown on the View. If we did not use that annotation the label would be the property name exactly as it is in the model and this would not always be a user-friendly label to show.

Another annotation we have used in almost all of our models is the Key annotation. This denotes that the property should be the primary key for that model when Entity Framework creates the table. This is not strictly necessary since the convention above configuration will figure this out by itself as long as we name our properties as “ID” or “modelnameID”. We have decided to add the annotation anyway for clarity.

ENTITY FRAMEWORK

In the implementation of the database for our website, we have used the Entity Framework package to connect the model classes we have created to the tables in the database. Entity Framework is an ORM framework provided by Microsoft to automate database related activities for applications. It enables us as developers to focus on creating the models and annotate the properties and classes, and then Entity Framework will translate those models and annotations into tables and relations in the database.

Entity Framework has different ways you can use it to handle the database creation and access. We have used the code first approach when developing the website, which means we have created the model classes, and then let Entity Framework handle the work of creating the database and the tables needed for our application to work based on these classes. One of the benefits of this approach is that Entity Framework is using any annotations we might have put on model classes and their properties to help generate the tables in the database the correct way. Entity Framework will also handle any relationships between tables and the usage of foreign keys.

As with many other things in ASP.NET MVC, Entity Framework also works on a premise of conventions above configuration. One of the places where this is noticeable is when specifying a primary key for a model class. Entity Framework will automatically specify any property named “ID” or “modelnameID” as the primary key of the corresponding table. We have taken advantage of this convention above configuration in a number of ways throughout our usage of Entity Framework.

Database migration

To handle changes in the database during the development of the new website we have enabled and used code first migrations. The benefit of using migrations in our project is that we can make sure that any changes to the models is reflected in the database that is being used – and which is included in our delivered Visual Studio project. Another benefit is that we are able to seed the database with test/demo data, which we have used throughout the development process.

To enable migrations in an MVC project, you need to run a couple of commands in the NuGet Package Manager Console and then make small modifications to the migration configuration, to make sure you are seeding the demo data. The commands you need to enable migrations are:

Command	Explanation
enable-migrations	Will set up the migrations for the project and create the necessary files and folders needed for migration to work.
add-migration InitialCreate	Will create a migration file, which the next command will use. The “InitialCreate” is for naming the migration so different migrations can be told apart. A migration file contains all the necessary commands to update a database with latest changes to the models, or to remove these changes form an updated database.
update-database	This command will check the database to see what migrations has already been implemented, and then make sure to update the database to the latest migration. This command will also run the Seed method which will make sure the data in the database is up to date with the latest changes to tables and models.

Seeding database data

As mentioned above by using Entity Framework and its migrations possibilities, we have been able to make sure that the database is seeded with initial data. To seed some data as part of migrations means that you can make sure that data, which is needed for your application to work, will be inserted into the tables during the creation of the database.

The data that we have chosen to include, as part of our solution is data that will reflect a website that is already up and running. Normally you will only seed the data that is necessary, but since our application should be able to show a running system, we have chosen to add some extra data such as a couple of users of the system.

Another benefit of using migrations and seeding the database is that we during our development phase have had a possibility of “resetting” that data in the database by running the command “update-database” again. If no changes has been made to any model and no new migrations have been created, this command will just run the seeding of the database, and the way we have set up the seed method, all data will be added or updated which will result in any data that have been altered or deleted will be reset.

REPOSITORIES

To access the data that is stored in the database we have used the principle of separation of concerns. We have done that by using the Repository and Unit of Work design patterns.

Instead of creating a repository for each entity, a rather cumbersome process, we have created a generic repository interface and class. The generic repository can be used for the basic CRUD operations. If more special operations are needed the repository class can be inherited and extended. The generic repository class has the following methods:

- `public Repository(BgsContext db, bool saveChanges = true)`
- `public T GetItem(int id)`
- `public IList<T> GetItems()`
- `public int SaveItem(T t)`
- `public T DeleteItem(int id)`

The methods are fairly straightforward except for the SaveItem(…) method which is used both for create and update. The method is shown below:

```
public int SaveItem(T t)
{
    if (t.EntityId == 0)
    {
        dbSet.Add(t);
        if (saveChanges)
        {
            db.SaveChanges();
        }
    }
    else
    {
        db.Entry(t).State = EntityState.Modified;
        if (saveChanges)
        {
            db.SaveChanges();
        }
    }
    return t.EntityId;
}
```

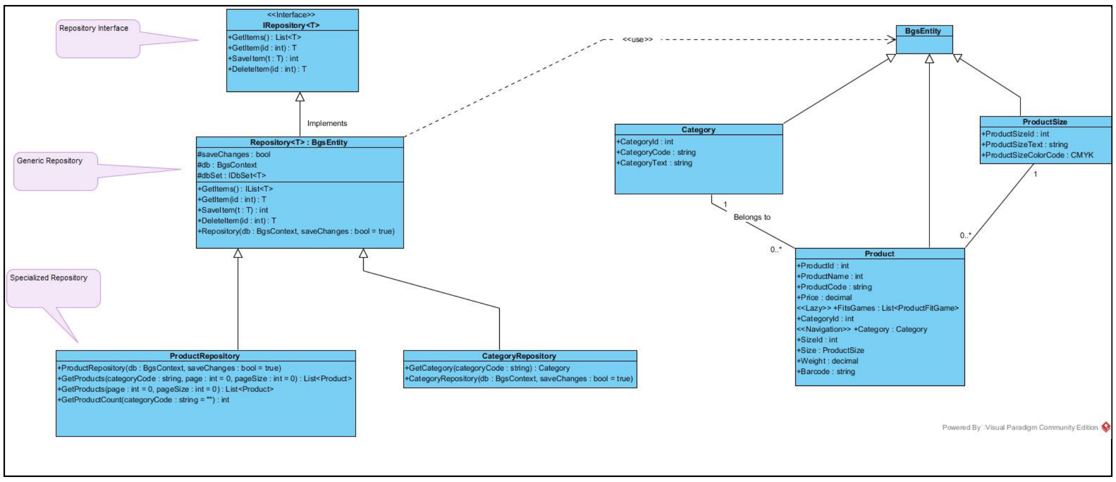
The problem is that on an update we need to know the id key of the item being updated. This is solved by letting the generic repository work on an entity super class named BgsEntity like this:

```
public class Repository<T> : IRepository<T> where T : BgsEntity
```

The BgsEntity class uses reflection to find the property that will return the correct id:

```
/// <summary>
/// Top level model class. All domain classes that are saved and
/// restored from the database should inherit from this class.
/// The generic repository can then be used to perform CRUD like
/// operations on the descendant class.
/// </summary>
public class BgsEntity
{
    /// <summary>
    /// Returns the current entity id.
    /// </summary>
    public virtual int EntityId
    {
        get
        {
            return GetEntityId(this.GetType());
        }
    }

    /// <summary>
    /// Retrieve entity from current class or base class.
    /// </summary>
    /// <param name="t"></param>
    /// <returns></returns>
    public virtual int GetEntityId(Type t)
    {
        try
        {
            string typeName = t.Name;
            string propertyName = typeName + "Id";
            PropertyInfo p = t.GetProperty(propertyName);
            if (p == null)
            {
                return GetEntityId(t.BaseType);
            }
            else
            {
                return (int)p.GetValue(this, null);
            }
        }
        catch (Exception)
        {
            return 0;
        }
    }
}
```

So, any model class that is a descendant of BgsEntity can use the generic repository. In an UML diagram it looks like this:


Repository Interface
<<Interface>>
IRepository<T>
+GetItems() : List<T>
+GetItem(id : int) : T
+SaveItem(t : T) : int
+DeleteItem(id : int) : T
Implements
<<use>>
Generic Repository
Repository<T> : BgsEntity
#saveChanges : bool
#db : BgsContext
#dbSet : IDbSet<T>
+GetItems() : IList<T>
+GetItem(id : int) : T
+SaveItem(t : T) : int
+DeleteItem(id : int) : T
+Repository(db : BgsContext, saveChanges : bool = true)
Specialized Repository
ProductRepository
+ProductRepository(db : BgsContext, saveChanges : bool = true)
+GetProducts(categoryCode : string, page : int = 0, pageSize : int = 0) : List<Product>
+GetProducts(page : int = 0, pageSize : int = 0) : List<Product>
+GetProductCount(categoryCode : string = "") : int
CategoryRepository
+GetCategory(categoryCode : string) : Category
+CategoryRepository(db : BgsContext, saveChanges : bool = true)
BgsEntity
Category
+CategoryId : int
+CategoryCode : string
+CategoryText : string
1
Belongs to
0..*
ProductSize
+ProductSizeId : int
+ProductSizeText : string
+ProductSizeColorCode : CMYK
1
0..*
Product
+ProductId : int
+ProductName : int
+ProductCode : string
+Price : decimal
<<Lazy>> +FitsGames : List<ProductFitGame>
+CategoryId : int
<<Navigation>> +Category : Category
+SizeId : int
+Size : ProductSize
+Weight : decimal
+Barcode : string
Powered By Visual Paradigm Community Edition

CONCLUSION

Our implementation of the BoardGameSleeves.com website is the initial and first version of a fully implemented dynamic website built using the ASP.NET MVC framework. Throughout the process, we have constantly been working back and forth designing the model and implementing the changes. A lot of the knowledge of how to build a website like this has come from trial and error. One of the things that we could see the side benefitting from is a details page for the products, to let the customers see all of the data that is available on the model.

Another functionality that we would like to have had time for is the implementation of a search functionality where customers could search for games and then get a list of Sleeves that would fit that particular game. Implementing a functionality like this could take advantage of the WebAPI and Ajax functionality that is a part of the ASP.NET MVC framework.

As mentioned in the section about model validation, we have annotations throughout the solution and in our models. There are probably models that could benefit from some extra annotations, but the places we have used them were the places that benefitted the most from the use of annotations and model validation.

The work on this project has shown us that a website like this is comprised of many parts, and that the model coming from initial brainstorming might not always be the model and layout that the final product ends up having. We started out with a model that looked vastly different from the one that we have described in the section about the domain model. In the same way the layouts that we did during the sketching phase of this project, are not 100% like the final Views. We used the sketches as inspiration during the implementation phase.

This project has overall been a good learning experience concerning the possibilities of the ASP.NET MVC framework, and how a website could be built using this framework.