Sitecore E-Commerce Services 2.2 on CMS 7.0 or Later

Sitecore E-Commerce API Reference Guide

A reference guide for the Sitecore E-Commerce API

Table of Contents

Chapter 1

Introduction

This guide describes the Sitecore E-Commerce Services (SES) API and some useful extensions to its functionality.

It is useful for developers who are looking for information about the SES API. It gives the reader a description for the contract/class functionality, parent classes, implementation, important methods/properties and some sample code.

This document contains the following chapters:

- **Chapter 1 — Introduction**
  This chapter is an introduction to the guide.
- **Chapter 2 — The Sitecore E-Commerce Services API**
  This chapter is an API reference.

1.1 Glossary

This section defines some of the terms used in this guide.

Component

A package or a module that encapsulates a set of contracts and implementations or related functionalities or data.

Contract

An interface or an abstract class.

Implementation

A class that implements a contract

Object

An instance of a class.

Unity

A lightweight, extensible dependency injection container.

It facilitates building loosely coupled applications and provides developers with the following advantages:

- Simplified creation of objects, especially for hierarchical object structures and dependencies.
- Abstraction of requirements; this allows developers to specify dependencies at run time or in configuration and simplify management of crosscutting concerns.
- Increased flexibility by deferring component configuration to the container.
- Service location capability, which allows clients to store or cache the container.
- Instance and type interception.

  For more information about the Unity Application Block, see http://unity.codeplex.com/, http://msdn.microsoft.com/en-us/library/ff663144.aspx and the *SES Developer's cookbook* where the Unity configuration is explained in more detail.

Chapter 2

The Sitecore E-Commerce Services API

This chapter describes the SES contracts that constitute the SES API.

SES uses Unity which has a component-based architecture to configure a number of contracts that exist in assemblies that match their namespaces. The `Sitecore.Ecommerce.DomainModel.dll` is the assembly that contains the contracts and The `Sitecore.Ecommerce.Kernel.dll` assembly that contains the default implementations.

Each section in this chapter represents a component in SES. In each section, there are class diagrams to show the contracts and corresponding default implementations of each of the components, tables to describe each contract's functionality, implementation and sometimes sample code snippets.

There is also a section that describes the webshop site settings.

This chapter contains the following sections:

- The SES Configuration Components
- The SES Customer Components
- The SES Product Components
- Product Information Management
- The SES Product Catalog Components
- The SES Order Components
- The SES Product Stock Components
- The SES Shipping Components
- The SES Shopping Cart Components
- The SES Pricing Components
- The SES Payment Providers Components
- The SES Content-to-Object Mapping Components

- The SES Search Provider Components
- The SES Analytics Component
- The SES Product Resolver Components
- Miscellaneous SES Components

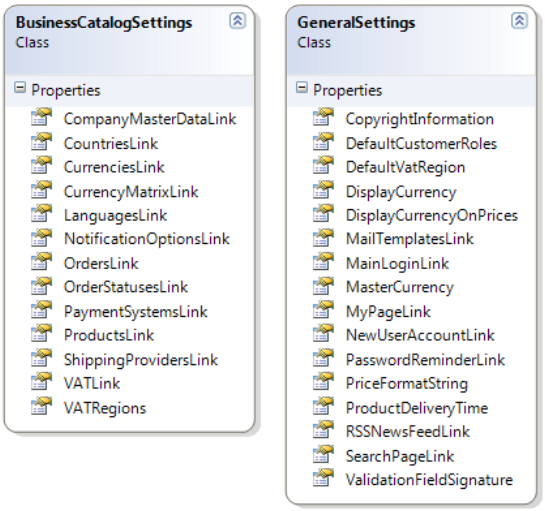
2.1 The SES Configuration Components

The SES configuration contracts and implementation classes describe the various configuration options that control how a variety of system components work. Some of these classes are about presentation logic.

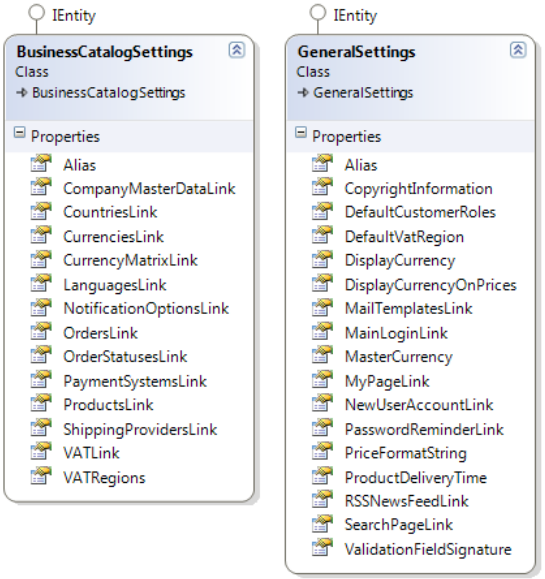
This set of components consists of two groups: non-presentation related and presentation related.

Non-Presentation Related Configuration Objects

The following class diagram gives you an overview of the non-presentation related configuration contracts:

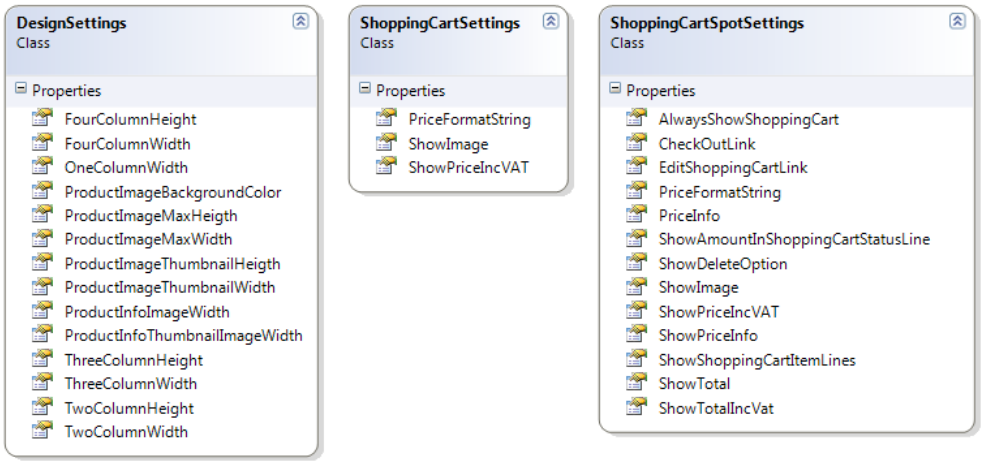


The following class diagram gives you an overview of the implementation classes of the non-presentation related configuration contracts:

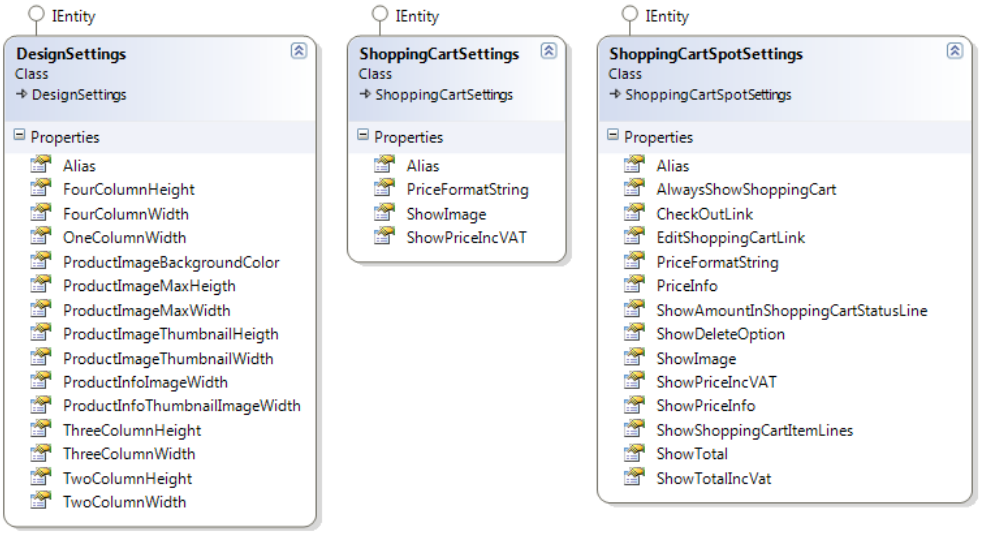


Presentation Related Configuration Objects

The following class diagram gives you an overview of the presentation related configuration contracts:



The following class diagram provides an overview of the implementation classes of the presentation related configuration contracts:



Note
We recommend that you do not modify the `DesignSettings`, `ShoppingCartSettings` and `ShoppingCartSpotSettings` objects because they are read by the presentation components. However, you can safely extend them by extending the contract and the implementation and configuring them in the `Unity.config` file.

2.1.1 Configuration Contracts

The following table describes each of the configuration related contracts. It presents the contract's functionality and default implementation.

It also presents the parent contract that this class implements.

Contract	Description
`BusinessCatalogSettings`	The default implementation of the Domain Model uses this contract — `Sitecore.Ecommerce.DomainModel.Configurati`

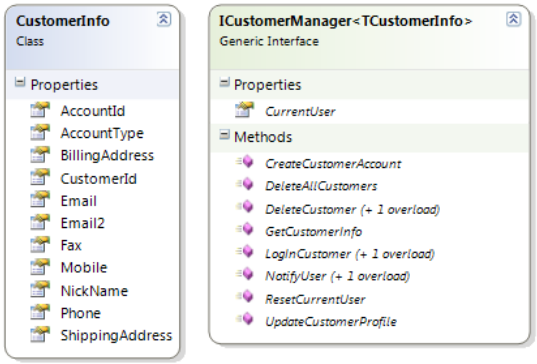
Contract	Description
	ons.BusinessCatalogSettings — to determine the root items for various SES business information stores, such as the product and order stores. The default implementation of this contract — Sitecore.Ecommerce.Configurations.BusinessCatalogSettings – retrieves field values from the *Site Settings/Business Catalog* item of the current site — (<home>/Site Settings/Business Catalog). **Note** You can change the Site settings location by changing the following attribute value in the site registration. EcommerceSiteSettings="/Site Settings" See the example site registration in the Sitecore.Ecommerce.Examples.config file.
DesignSettings	Sitecore.Ecommerce.DomainModel.Configurations.DesignSettings exposes the layout and presentation configuration settings for the presentation components on the managed websites. The default implementation of this contract — Sitecore.Ecommerce.Configurations.DesignSettings — retrieves field values from the *Site Settings/Design Settings* of the current site — (<home>/Site Settings/Design Settings).
GeneralSettings	Sitecore.Ecommerce.DomainModel.Configurations.GeneralSettings exposes the global configuration settings. The default implementation of this contract — Sitecore.Ecommerce.Configurations.GeneralSettings – retrieves field values from the *Site Settings/General* item of the current site — (<home>/Site Settings/General).
ShoppingCartSettings	Sitecore.Ecommerce.DomainModel.Configurations.ShoppingCartSettings exposes the configuration settings for individual shopping carts. The default implementation of this contract — Sitecore.Ecommerce.Configurations.ShoppingCartSettings — manages information in the *Site Settings/Shopping Cart* item of the current site — (<home>/Site Settings/Shopping Cart).

Contract	Description
`ShoppingCartSpotSettings`	`Sitecore.Ecommerce.DomainModel.Configurations.ShoppingCartSpotSettings` exposes the configuration settings for the presentation components that display an individual shopping cart. The default implementation of this contract — `Sitecore.Ecommerce.Configurations.ShoppingCartSpotSettings` — accesses the *Site Settings/Shopping Cart Spot* item of the current site — (`<home>/Site Settings/Shopping Cart Spot`).

2.2 The SES Customer Components

The SES Customer model consists of `CustomerInfo` and `ICustomerManager` contracts that provide and manage the customer's information.

The following class diagram gives you an overview of the Customer contracts:

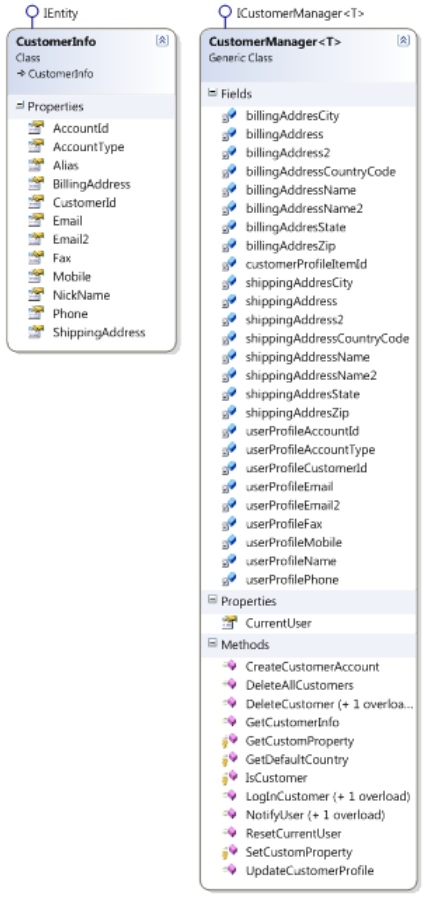


The default implementations of the Customer related contracts are using the Sitecore ASP.NET membership provider. The setting that indicates, which security roles the users should be members of, is configured by the `DefaultCustomerRoles` property of the `GeneralSettings` class, see the section *Configuration Contracts*.

SES creates users in the site context domain with the default implementation.

Note
The domain can be specified at the site definition in the `Web.config` file. If the roles specified in setting `DefaultCustomerRoles` are not in that domain, then the users will not be added to the roles and a log entry will be created.

The following class diagram gives you an overview of the customer implementation:



2.2.1 The Customer Contracts

The following table describes each of the customer related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

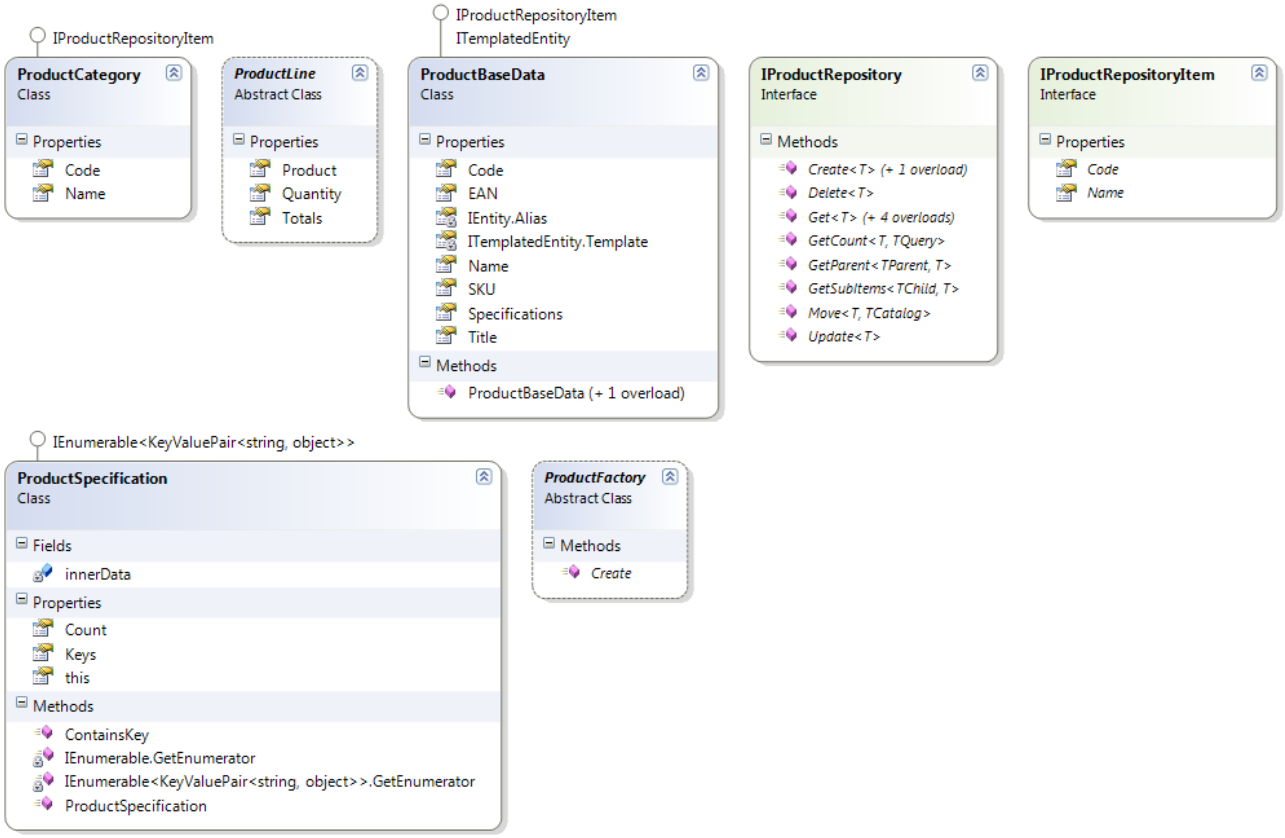
Contract	Description
`CustomerInfo`	`Sitecore.Ecommerce.DomainModel.Users.CustomerInfo` exposes information about a customer. The default implementation of this contract — `Sitecore.Ecommerce.Users.CustomerInfo` — provides basic customer information.
`ICustomerManager`	`Sitecore.Ecommerce.DomainModel.Users.CustomerManager` defines a programming interface for managing information about customers. The default implementation of this contract — `Sitecore.Ecommerce.Users.CustomerManager` — manages customer information in the Sitecore ASP.NET membership database. A pipeline called `CustomerCreated` can be modified or extended to add custom logic. This pipeline is in located in the

Contract	Description

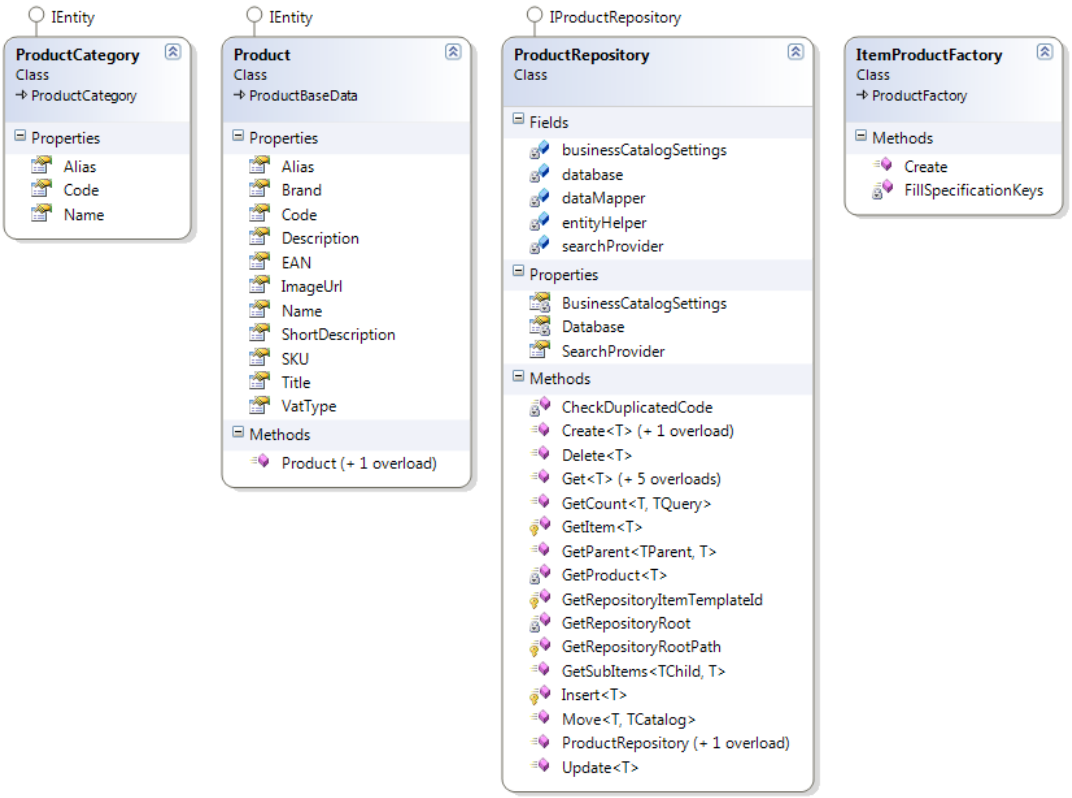
	`Sitecore.Ecommerce.config` file.
	`<customerCreated>`
	`<processor type="Sitecore.Ecommerce.Pipelines.CustomerCreated.ConfigureSecurity, Sitecore.Ecommerce.Kernel"/>`
	`<processor type="Sitecore.Ecommerce.Pipelines.CustomerCreated.LogIn, Sitecore.Ecommerce.Kernel"/>`
	`<processor type="Sitecore.Ecommerce.Pipelines.CustomerCreated.SendNotification, Sitecore.Ecommerce.Kernel"/>`
	`</customerCreated>`
	
	As a default setting, all the roles defined in the general setting "DefaultCustomerRoles" are added to the user's membership.

2.3 The SES Product Components

The following class diagram gives you an overview of the product contracts:



The following class diagram gives you an overview of the product implementation:



2.3.1 The Product Contracts

The following table describes each of the product related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description

`IProductRepository`	`Sitecore.Ecommerce.DomainModel.Products.IProductRepository` defines a programming interface for managing a product catalog.
	
	The default implementation of this contract — `Sitecore.Ecommerce.Products.ProductRepository` — manages the descendants of the item specified in the *Business Catalog* item in the **Products Link** field of the current site — (`<home>/Site Settings/Business Catalog`).
	
	Examples:
	`// Reading default product data`
	`publ1ic void ShouldReadDefaultProductData(IProductRepository repository)`
	`{`
	`ProductBaseData productBase = repository.Get<ProductBaseData>("1002");`
	
	`Product product = productBase as Product;`
	
	`Assert.IsNotNull(product);`
	`}`
	`// Reading custom product data`
	`public void`

Contract	Description

	`ShouldReadCustomProductData(IProductRepository repository)`
	`{`
	`SlrCamera product = repository.Get<SlrCamera>("1002");`
	`string Exposure = product.Specifications["Exposure"];`
	`}`

`IProductRepositoryItem`	`Sitecore.Ecommerce.DomainModel.Products.IProductRepositoryItem` represents any item in a product repository, such as a product or a product category. All the items in a product repository implement this contract. For more information about product repositories, see the SES `IProductRepository` contract. For more information about products, see the SES `ProductBaseData` contract. The default implementations of this contract include the `ProductBaseData` and the `ProductCategory` contracts.
`ProductBaseData`	`Sitecore.Ecommerce.DomainModel.Products.ProductBaseData` implements the `IProductRepositoryItem` and the `ITemplatedEntity` interfaces. This contract presents essential information about a product: • `Code` • `EAN` which stands for the European Article Number • `SKU` which stands for the Stock-Keeping Unit. • `Title` This contract has a corresponding CMS template. This template is registered in the `Sitecore.Ecommerce.Config` file `<setting name="Ecommerce.Product.BaseTemplateId" value="{02870C17-4273-4242-89A4-E973C3CF8EC0}" />` **Note** You should not replace or overwrite the `ProductBaseData` contract and template. Instead create a custom product class and inherit from it. The default implementation of this contract — `Sitecore.Ecommerce.Products.Product` — presents common information about a product, such as the product name and the product description.
`ProductCategory`	`Sitecore.Ecommerce.DomainModel.Products.ProductCategory` implements the `IProductRepositoryItem` interface and represents a category of products. The default implementation of this contract — `Sitecore.Ecommerce.Products.ProductCategory` — represents basic information about a product category, such as the product category name and the product category code.

Contract	Description
`ProductLine`	`Sitecore.Ecommerce.DomainModel.Products.ProductLine` represents information about a specific product in a business entity, such as the quantity of a product that is in a shopping cart or order. The default implementations of this contract include the `OrderLine` contract and the `ShoppingCartLine` contract.
`ProductSpecification`	`Sitecore.Ecommerce.DomainModel.Products.ProductSpecification` presents product specifications in a dictionary-like format. It contains a list of key-value pairs which describes each item in the specifications collection. For example: • The specification for the SLR camera has the fields: “Effective Pixels” and “Image Sensor”. • The specification for the Lenses has the fields: “Focal Length”, “Maximum Aperture” and “Minimum Aperture”. For more information, see the section *Using Product Specification to Extend Products*.
`ProductFactory`	`Sitecore.Ecommerce.DomainModel.Products.ProductFactory` is used for product instance creation. The `Create` method receives product template ID and returns a new product instance based on the `ProductBaseData` contract. The default implementation of this contract is `Sitecore.Ecommerce.Products.ItemProductFactory`. The default product factory does two things: 1. It returns `a` product instance. It resolves the product from `Unity.config` file using the *template* parameter of the `Create` method. In Unity the mapping between the template ID and the product class is configured like the following example, where the name attribute contains the template ID: `<register type="ProductBaseData" mapTo="SlrCameraProduct" name="{B072B7C7-6F3F-4316-B8D7-010629AEBEF1}"/>` 2. It populates the `ProductSpecification` collection. For more information, see the section *Using Product Specification to Extend Products.* **Note** Creating a product using a product factory will create the product instance and not the corresponding product item in the CMS.

2.4 Product Information Management

This section describes some product information management improvements.

There are two ways to add custom product information to SES:

- Use the *Product Specifications* collection for standard fields that contains simple product specification data. This is the recommended approach if you only need one product class that handles many specialized product templates in CMS (one – to – many relationship). For this to work and to be able read the data through the API, all the specification data (fields) must be located in a template section called Specification.

  For more information, see the section *Using Product Specification to Extend Products.*

- Creating some custom product classes for each specialized product template. This is the recommended approach if you need to add fields to product templates which are not located in a template section named Specification. In this case, you must create a custom product class to be able to read the data through the API.

  For more information, see the section *Creating a New Product Class*.

2.4.1 Using Product Factory to Create a Product Instance

The Product Factory component is used to construct instances of products classes.

Contract and Implementation

The Product Factory located in the `Sitecore.Ecommerce.DomainModel.Products` namespace and has one method `Create`, which takes a string parameter `template` and returns a product class instance based on the `ProductBaseData` contract.

Example:

```
public abstract ProductBaseData Create(string template);
```

The default implementation of the factory is the `Sitecore.Ecommerce.Products.ItemProductFactory` class that is located in `Sitecore.Ecommerce.Kernel.dll` assembly. In the default implementation, the parameter `template` is assumed to be a product template ID.

Creating a New Product Class

If you want to implement your own product class you can inherit from either: `Sitecore.Ecommerce.DomainModel.Products.ProductBaseData` class or `Sitecore.Ecommerce.Products.Product` class.

The base class for all the products is `ProductBaseData` from the `DomainModel` namespace. There is a default product implementation located in the *Kernel* project which has some additional properties such as *Description* and *Brand*. If you want to use your custom products along with the Example Pages you must inherit from the *Product* template and class. If not, you must create a custom template and inherit from `ProductBaseData`.

Registering New Product Classes in Unity

The Product Factory instantiates product classes using Unity `IoCContainer`. By default, `ProductBaseData` is mapped to the *Product* class:

Example:

```
        <alias alias="ProductBaseData"
type="Sitecore.Ecommerce.DomainModel.Products.ProductBaseData,
Sitecore.Ecommerce.DomainModel"/>
        <alias alias="SitecoreProduct" type="Sitecore.Ecommerce.Products.Product,
Sitecore.Ecommerce.Kernel"/>

        <container>
          <register type="ProductBaseData" mapTo="SitecoreProduct"/>
        </container>
```

You must register the new product classes in Unity giving it the template ID to map to.

The following snippet shows you how to register it.

Example:

```
        <alias alias="FlashProduct" type="Sitecore.Ecommerce.Examples.Products.Flash,
Sitecore.Ecommerce.Custom"/>
        <alias alias="LenseProduct" type="Sitecore.Ecommerce.Examples.Products.Lense,
Sitecore.Ecommerce.Custom"/>
        <alias alias="OtherAccessoryProduct"
type="Sitecore.Ecommerce.Examples.Products.OtherAccessory, Sitecore.Ecommerce.Custom"/>
        <alias alias="PsCameraProduct" type="Sitecore.Ecommerce.Examples.Products.PsCamera,
Sitecore.Ecommerce.Custom"/>
        <alias alias="SlrCameraProduct" type="Sitecore.Ecommerce.Examples.Products.SlrCamera,
Sitecore.Ecommerce.Custom"/>

        <container>
          <register type="ProductBaseData" mapTo="FlashProduct" name="{95681CF6-3635-49EC-
A09A-CC548FA62389}"/>
          <register type="ProductBaseData" mapTo="LenseProduct" name="{8FAC8E12-7459-43F8-
97E8-1BC6840B9226}"/>
          <register type="ProductBaseData" mapTo="OtherAccessoryProduct" name="{A93FA2C4-3AE4-
45C2-8C3F-EFA7E129537E}"/>
          <register type="ProductBaseData" mapTo="PsCameraProduct" name="{7BD2FBC6-061B-40DD-
B1F9-D8603A701624}"/>
          <register type="ProductBaseData" mapTo="SlrCameraProduct" name="{B072B7C7-6F3F-4316-
B8D7-010629AEBEF1}"/>
        </container>
```

Note
The classes are implemented in the Example Pages package.

Instantiating a Product

Example:

```
        ProductFactory factory = Context.Entity.Resolve<ProductFactory>();
        const string ShoeTemplate = "<Shoe Template ID>";

        ProductBaseData product = factory.Create(ShoeTemplate);
```

If the template ID is not found in the database, the `InvalidOperationException` exception is thrown.

If a product class with a specific template ID is not registered in Unity, using the `Name` attribute, the default mapping is used, which is the registration without the `Name` attribute.

2.4.2 Using Product Specification to Extend Products

`ProductSpecification` is a new business entity that is intended to simplify product information management. It is a dictionary-like entity which allows dynamic storing and reading of key-value pair data.

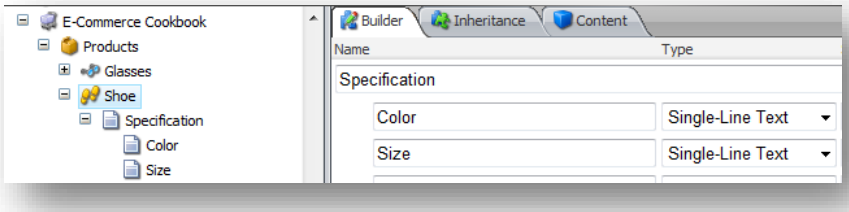
The default implementation assumes that the product specifications are stored in a template section called **Specification**. The fields located in the template section called **Specification** in all the inherited templates are included in the same product Specifications collection. For example, if the SLRCamera template inherits from the Cameras template and they both contain a template section called **Specification**, then key-value pairs based on the fields from both templates are read and mapped to the `ProductSpecification` collection on the product class, when the products are resolved through the API.

Creating Product Templates with Specifications

To extend the product information with new fields:

1. Create a new product template and inherit from a template `/sitecore/templates/Ecommerce/Product/Product`.
2. Create a template section called **Specification**.
3. Add fields to the section that will contain the additional specification data.

Now your product template is ready for use. The following is an example of a specialized Shoe template that adds two additional fields *Color* and *Size* to the template section called **Specification.** These fields are then mapped to the Specifications collection of the product instance.



Creating a Product

To create a new product using the SES API:

1. Create a product instance using the Product Factory.
2. Populate the product data along with the key value pairs in the Specifications collection.
3. Save the product using Product Repository — see the section *Saving a Product.*

Populating Product Data

The default Product Factory implementation uses the `Template` parameter to read all the specification fields located in the **Specification** sections of the product template and the inherited templates. When the factory creates the product instance, the Specifications collection is populated with the fields found in the new product instance. In the Shoe example, it will contain two keys: *Color* and *Size* that were read from the *Shoe* template.

The following snippet shows you how to set data to the product instance.

Example:

```
product.Code = "1001";
product.Title = "Sandals";

product.Specifications["Color"] = "Black";
```

```
        product.Specifications["Size"] = "36-38";
```

Any attempts to set collection values for keys that are not part of the templates sections called **Specification** and consequently not a part of the Keys in the collection, will result in the exception `KeyNotFoundException`.

Example:

```
        product.Specifications["Some invalid key"] = "any value"; // throws
KeyNotFoundException.
```

Saving a Product

You must use `ProductRepository` instance to store new products in the CMS.

If you use the Product Factory to create a product instance (see the section *Instantiating a Product*), it will just create the object instance and not the corresponding product item in the CMS. To create and save the product in the CMS, you must call an additional method.

The following snippet shows you how to call it:

```
        IProductRepository repository = Context.Entity.Resolve<IProductRepository>();

        repository.Create(product);
```

Reading Product Data

You must use the product repository to read the product data. The keys-value pairs which are located in the Specifications collection depend on the fields in the template sections named **Specification** of the corresponding product template, as described in the section *Populating the Product Data*. If the template has base templates that also contain **Specification** sections, the keys are aggregated into the same Specifications collection.

Example:

```
        IProductRepository repository = Context.Entity.Resolve<IProductRepository>();

        ProductBaseData camera = repository.Get<ProductBaseData>("1002");
        Assert.AreEqual("10.1 million", camera.Specifications["Effective Pixels"]);

        ProductBaseData lense = repository.Get<ProductBaseData>("4001");
        Assert.AreEqual("105mm (157.5mm when used with Nicam DX format)  ",
lense.Specifications["Focal length"]);

        ProductBaseData flash = repository.Get<ProductBaseData>("2002");
        Assert.AreEqual("25 to 1000  ", flash.Specifications["ISO range in TTL auto flash
mode"]);
```

2.4.3 Entity Mappers

SES contains a number Entity mappers designed to simplify data mapping between the SES entities and the CMS items. It is possible to convert primitive and custom types if custom converters are implemented.

Note
Only `Get` methods of the default Product Repository uses Entity Mappers.

An Entity Mapper has a simple `Map` method which gets the source and the target instances. It analyzes the source type members and creates a list of the Member Converters. It calls each specific member converter and saves the results to the target object.

Default Mappers Implementation

There are two default Entity Mappers available by default that convert entities to items and vice-versa.

Example:

```
        <alias alias="EntityToItemMapper"
type="Sitecore.Ecommerce.Data.Mapping.EntityToItemMapper, Sitecore.Ecommerce.Kernel"/>
        <alias alias="ItemToEntityMapper"
type="Sitecore.Ecommerce.Data.Mapping.ItemToEntityMapper, Sitecore.Ecommerce.Kernel"/>

        <container>
          <register type="EntityMapper[IEntity, Item]" mapTo="EntityToItemMapper"/>
          <register type="EntityMapper[Item, IEntity]" mapTo="ItemToEntityMapper"/>
        </container>
```

Entity Member Converters

The Entity member converters are designed to convert a specific entity member type to a storage object and vice-versa. The storage object is an item in the default implementation.

Creating a New Converter

All the entity member converters should implement either the `Sitecore.Ecommerce.DomainMode.Data.IEntityMemberConverter` interface or inherit from the `Sitecore.Ecommerce.DomainModel.Data.EntityMemberConverter<TEntityMember, TStorage>` class.

The implementation based on the abstract class is recommended. It allows using strongly typed parameters and avoids type casting.

Resolving a Converter

All the converters must be registered in Unity. The `Sitecore.Ecommerce.Data.Mapping.EntityMemberConverterLookupTable` class is responsible for resolving the converters. The converters are resolved according to the following algorithm:

1. If the entity member has been augmented with the SES specific Entity attribute, the `MemberConverter` property of Entity attribute explicitly specifies the exact Converter to use. The `MemberConverter` property must contain a name of a Converter specified in Unity and it will throw an exception, otherwise.
2. If no explicit Entity attribute with the `MemberConverter` property set is specified for an entity member, the `DataMapper` tries to combine the entity member type name with the suffix `EntityMemberConverter`. That is the default way that Converters such as `BooleanEntityMemberConverter` and `DateTimeEntityMemberConverter` are resolved.
3. If the first two steps have not resolved the default converter — `ConvertibleEntityMemberConverter` — is used.

Default Implementation

The Entity member Converters are located in the `Sitecore.Ecommerce.Data.Mapping.Converters` namespace. Four entity member Converters are available by default:

- `ConvertibleEntityMemberConverter`
- `BooleanEntityMemberConverter`
- `DateTimeEntityMemberConverter`
- `ProductSpecificationEntityMemberConverter`

The default entity member converters registered in the `Unity.config` file:

```
        <alias alias="ConvertibleEntityMemberConverter"
type="Sitecore.Ecommerce.Data.Mapping.Converters.ConvertibleEntityMemberConverter,
Sitecore.Ecommerce.Kernel"/>
        <alias alias="BooleanEntityMemberConverter"
type="Sitecore.Ecommerce.Data.Mapping.Converters.BooleanEntityMemberConverter,
Sitecore.Ecommerce.Kernel"/>
        <alias alias="DateTimeEntityMemberConverter"
type="Sitecore.Ecommerce.Data.Mapping.Converters.DateTimeEntityMemberConverter,
Sitecore.Ecommerce.Kernel"/>
        <alias alias="ProductSpecificationEntityMemberConverter"
type="Sitecore.Ecommerce.Data.Mapping.Converters.ProductSpecificationEntityMemberConverter,
Sitecore.Ecommerce.Kernel"/>
        <container>
          <register type="IEntityMemberConverter" mapTo="ConvertibleEntityMemberConverter" />
          <register type="IEntityMemberConverter" mapTo="BooleanEntityMemberConverter"
name="BooleanEntityMemberConverter"/>
          <register type="IEntityMemberConverter" mapTo="DateTimeEntityMemberConverter"
name="DateTimeEntityMemberConverter"/>
          <register type="IEntityMemberConverter"
mapTo="ProductSpecificationEntityMemberConverter"
name="ProductSpecificationEntityMemberConverter"/>
        </container>
```

Convertible Entity Member Converter

The `ConvertibleEntityMemberConverter` is the default entity member converter which is used to map all the primitive types which implement the `System.IConvertible` interface.

Note
The convertor does not map `Boolean` and `DateTime` values.

BooleanEntityMemberConverter and DateTimeEntityMemberConverter

Sitecore stores `Boolean` and `DateTime` values types in a specific format and that is why the types have their own specific converters.

ProductSpecificationEntityMemberConverter

The product Specifications collection is not a simple property type and therefore has its own converter. The `ProductSpecificationEntityMemberConverter` takes care of converting all the key-value pairs mapped between the collection and the product template. It uses the algorithm for converting the values that is described in the section *Resolving a Converter*

Field-Based Entity Member Converter

You must use the `Sitecore.Ecommerce.Data.Mapping.FieldBasedEntityMemberConverter` as a base class for converters that are designed to work with item fields. The class has the `StorageObject` property of type `Sitecore.Data.Fields.Field` and contains the storage field.

How to Create Custom Entity Class

There are some examples of custom products located in the `Sitecore.Ecommerce.Examples.Products` namespace of the `Sitecore.Ecommerce.Custom` assembly. Here is an example for the SLR Camera:

Example:

```
        namespace Sitecore.Ecommerce.Examples.Products
        {
          using Ecommerce.Products;

          // <summary>
```

```
        // Defines the SLR camera class.
        // </summary>
        public class SlrCamera : Product
        {
          // <summary>
          // Gets or sets the effective pixels.
          // </summary>
          // <value>
          // The effective pixels.
          // </value>
          public string EffectivePixels { get; set; }

          // <summary>
          // Gets or sets the image sensor.
          // </summary>
          // <value>
          // The image sensor.
          // </value>
          public string ImageSensor { get; set; }
        }
      }
```

This class extends the default Product class with two new properties `EffectivePixels` and `ImageSensors`. The properties are mapped to the template fields *Effective Pixels* and *Image Sensors*. Note that item field names contain spaces and can be mapped correctly. This logic for resolving the field name mapping is implemented in the `Sitecore.Ecommerce.Data.Mapping.FieldNamingPolicy` class.

2.4.4 Product Repository

How to Create Product in a Category

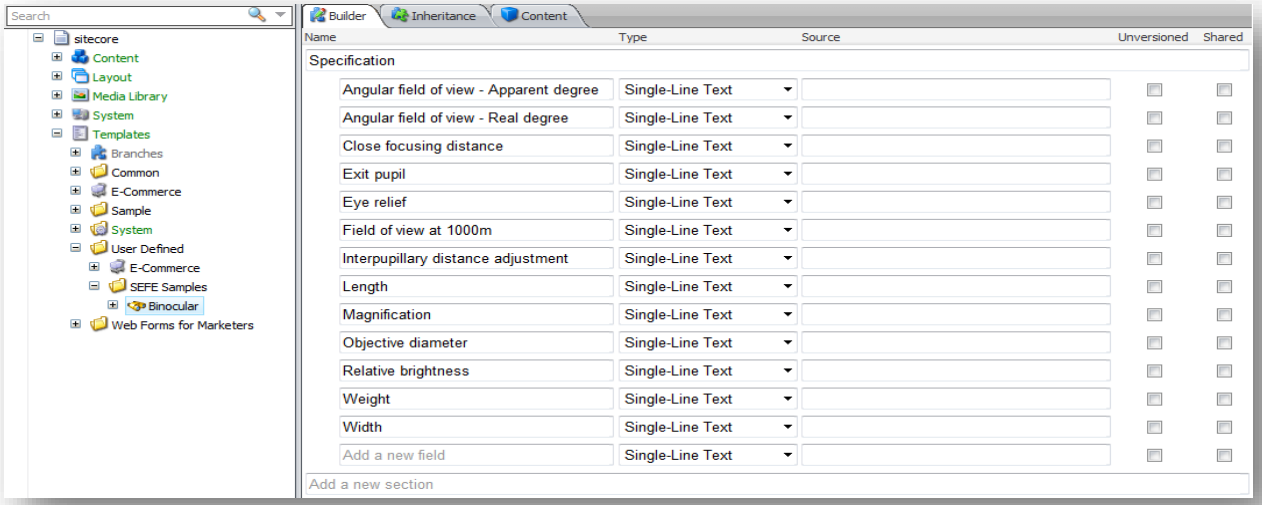
This code shows how to create a Binocular product in a given category. There is a test template Binocular that adds some new specification fields. The example creates a new instance of the default product class based on the given template ID, specifies some test values, and saves it in the Binoculars category of the repository.

Example:

```
      namespace SES.Samples
      {
        using Sitecore.Ecommerce;
        using Sitecore.Ecommerce.DomainModel.Products;

        public class ProductRepositorySample
        {
          public void HowToCreateProductInCategory()
          {
            // Instantiate Product Repository using Unity IoCContainer.
            IProductRepository repository = Context.Entity.Resolve<IProductRepository>();

            // Get the category from the repository to ensure that it exists.
            ProductCategory binocularsCategory =
repository.Get<ProductCategory>("Binoculars");

            // Create new category if nothing found.
            if (binocularsCategory == null)
            {
              binocularsCategory = Context.Entity.Resolve<ProductCategory>();

              // Specify required product category parameters such Code and Name.
              // Code is used to find a category in repository.
              // Name is a product item name in default implementation.
              binocularsCategory.Code = "Binoculars";
              binocularsCategory.Name = "Binoculars";

              repository.Create(binocularsCategory);
            }
```

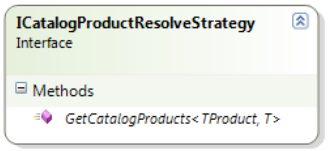
```
            // Instantiate Product Factory to create a product instance.
            ProductFactory factory = Context.Entity.Resolve<ProductFactory>();

            // Create instance of SLR camera product class.
            const string BinocularTemplateId = "{4703A513-F89A-4F45-96B9-D3CE94A3E43A}";
            ProductBaseData binocular = factory.Create(BinocularTemplateId);

            binocular.Code = "8x42HG L DCF";
            binocular.Name = "8x42HG L DCF";

            // Fill the product specifications.
            binocular.Specifications["Magnification"] = "8";
            binocular.Specifications["Objective diameter"] = "42";
            binocular.Specifications["Angular field of view - Real degree"] = "7.0";
            binocular.Specifications["Angular field of view - Apparent degree"] = "52.1";
            binocular.Specifications["Field of view at 1000m"] = "122";
            binocular.Specifications["Exit pupil"] = "5.3";
            binocular.Specifications["Relative brightness"] = "28.1";
            binocular.Specifications["Eye relief"] = "20.0";
            binocular.Specifications["Close focusing distance"] = "3.0";
            binocular.Specifications["Weight"] = "795";
            binocular.Specifications["Length"] = "157";
            binocular.Specifications["Width"] = "139";
            binocular.Specifications["Interpupillary distance adjustment"] = "56-72";

            // Create new product in the repository and put it into Binoculars category.
            repository.Create<ProductBaseData, ProductCategory>("Binoculars", binocular);
        }
    }
}
```

The Binocular template is inherited from the `/sitecore/templates/Ecommerce/Product/Product` template that is a base for all the product templates. The following image shows the fields used in the example:

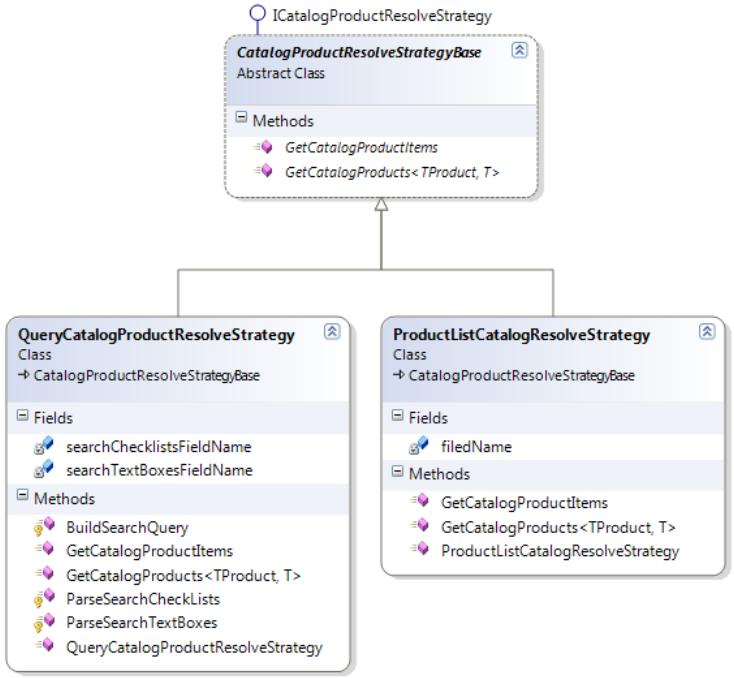


2.5 The SES Product Catalog Components

The following class diagram gives you an overview of the product catalog contracts:



The following class diagram gives you an overview of the product catalog implementation:



2.5.1 The Product Catalog Contract

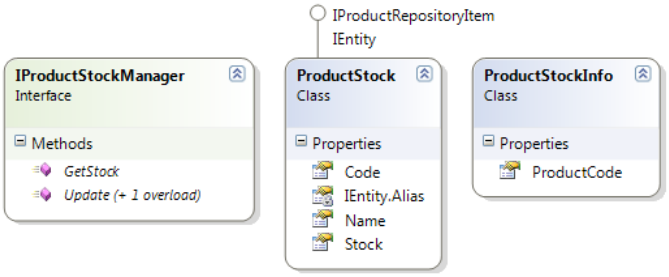
The following table describes each of the product catalog related contract. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`ICatalogProductResolveStrategy`	`Sitecore.Ecommerce.DomainModel.Catalogs.ICatalogProductResolveStrategy` defines the API that should be used to retrieve specified products from a product repository. Sitecore provides two default implementations of the `ICatalogProductResolveStrategy` contract: • The Product List product resolution strategy — `Sitecore.Ecommerce.Catalogs.ProductListCatalogResolveStrategy` retrieves one or more items based on their IDs. • The Query product resolution strategy — `Sitecore.Ecommerce.Catalogs.QueryCatalogProductResolveStrategy` returns products that match

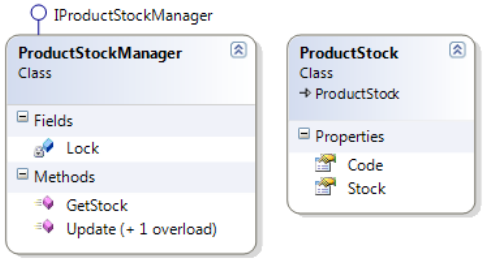
Contract	Description
	the search query. When you create an item that presents a number of products on a website, you must use one of the `ICatalogProductResolveStrategy` implementations to determine how to specify which products should be displayed. SES stores the user's selections as parameters in the fields of the item, and the presentation components use those fields to determine which products to display. The Product Page editor that appears for items based on the `Ecommerce/Product Categories/Product Search Group` data template uses these two `ICatalogProductResolveStrategy` implementations. SES manages the `ICatalogProductResolveStrategy` definition items beneath the `Sitecore/System/Modules/Ecommerce/System/Product Selection Method` item. The `Sitecore.Ecommerce.Xsl.XslExtensions.GetProductsForCatalog()` XSL extension method (should be used with items based on the `Ecommerce/Product Categories/Product Search Group` data template). It returns a list of the products that were retrieved using the strategy selected in the current item. To expose this method as `sc:GetProductsForCatalog()` in an XSL rendering, add the following attribute to the `/xsl:stylesheet` element in the `.xslt` file: `xmlns:ec="http://www.sitecore.net/ec"` To return the products on the webpage item, you can configure the implementations of the `ICatalogProductResolveStrategy` contract to search for specific fields on the webpage in the repository.

2.6 The SES Product Stock Components

The following class diagram gives you an overview of the product stock contracts:



The following class diagram gives you an overview of the product stock implementation:



2.6.1 The Product Stock Contracts

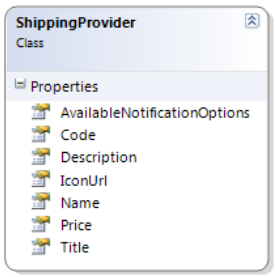
The following table describes each of the product stock related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`IProductStockManager`	`Sitecore.Ecommerce.DomainModel.Products.IProductStockManager` allows you to read and update the stock amount for specific products in the product repository. **Example:** `public void ShouldReadStockFromProductItem() { IProductStockManager stockManager = Context.Entity.Resolve<IProductStockManager>(); ProductStockInfo stockInfo = new ProductStockInfo { ProductCode = "1002" }; long stock = stockManager.GetStock(stockInfo).Stock; }` This contract has two implementations: • The `ProductPriceManager` class in the Kernel. • The `RemoteProductPriceManager` class in the Service model — This implementation is a service that is used in case of split content management and content delivery environment. For more information, see the *SES Scaling Guide*.

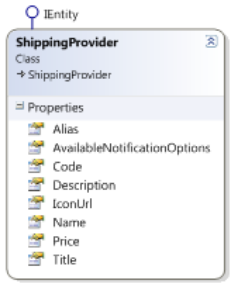
Contract	Description
`ProductStock`	`Sitecore.Ecommerce.DomainModel.Products.ProductStock` represents the stock amount of a given product returned from the `IProductStockManager`. The default implementation of this contract is the `Sitecore.Ecommerce.Products.ProductStock` class, which implements the `IProductRepositoryItem` interface.
`PoductStockInfo`	`Sitecore.Ecommerce.DomainModel.Products.ProductStockInfo` is both the contract and the implementation passed to `IProductStockManager` methods representing the arguments being used.

2.7 The SES Shipping Components

The following class diagram gives you an overview of the shipping contract:



The following class diagram gives you an overview of the shipping implementation:



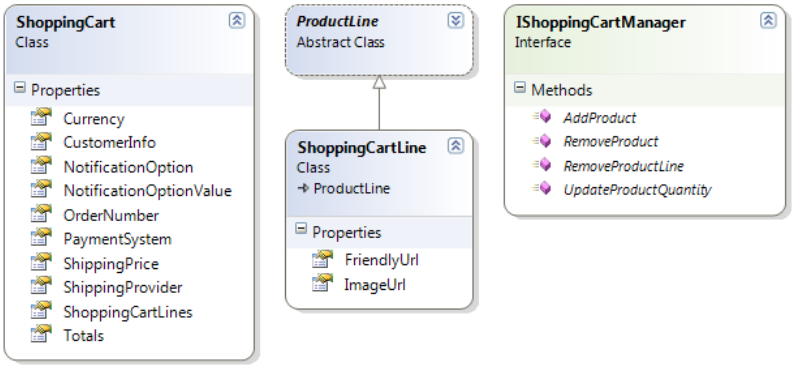
2.7.1 The Shipping Contract

The following table describes the shipping related contract. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

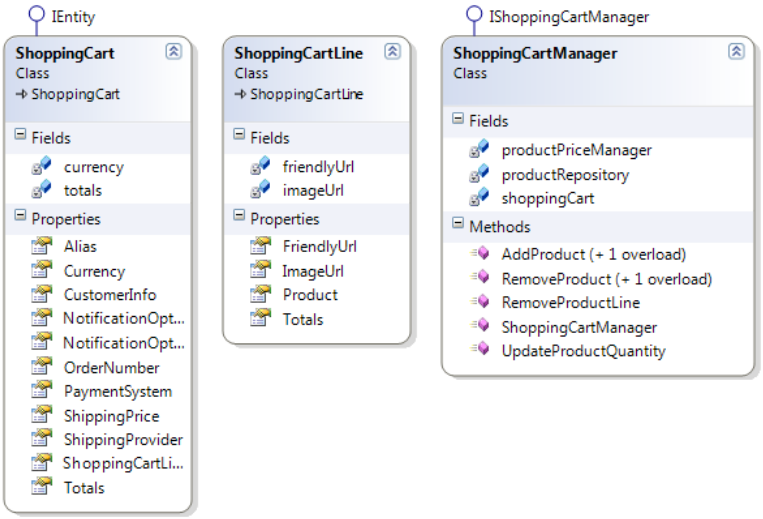
Contract	Description
`ShippingProvider`	`Sitecore.Ecommerce.DomainModel.Shippings.ShippingProvider` exposes information about a shipping system. The default implementation of this contract — `Sitecore.Ecommerce.Shippings.ShippingProvider` — represents the children of the item specified in the System Links section in the **Shipping Providers Link** field of the Business Catalog of the current site — (`<home>/Site Settings/Business Catalog`). **Note** The default implementation cannot communicate with the external shipping providers.

2.8 The SES Shopping Cart Components

The following class diagram gives you an overview of the shopping cart contracts:



The following class diagram gives you an overview of the shopping cart implementation:



2.8.1 The Shopping Cart Contracts

The following table describes each of the shopping cart related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`ProductLine`	See the section *The Product Contracts*.

`IShoppingCartManager`	`Sitecore.Ecommerce.DomainModel.Carts.IShoppingCartManager` defines a programming interface for managing the content of a shopping cart.
	
	The default implementation of this contract — `Sitecore.Ecommerce.Carts.ShoppingCartManager` — stores information in the ASP.NET session.
	
	Example:
	
	`public void ShouldAddProductToShoppingCart()`
	`{`

Contract	Description
	`IShoppingCartManager cartManager = Context.Entity.GetInstance<IShoppingCartManager>(); cartManager.AddProduct("1002", 2); ShoppingCart cart = Context.Entity.GetInstance<ShoppingCart>(); IList<ShoppingCartLine> lines = cart.ShoppingCartLines; }`
`ShoppingCart`	`Sitecore.Ecommerce.DomainModel.Carts.ShoppingCart` exposes information about the state of an individual shopping cart, such as the customer associated with the cart, and the contents of the cart. The default implementation of this contract — `Sitecore.Ecommerce.Carts.ShoppingCart` — implements typical shopping cart functionality.
`ShoppingCartLine`	`Sitecore.Ecommerce.DomainModel.Carts.ShoppingCartLine` implements the `ProductLine` class and exposes information about a line item in a shopping cart. The default implementation of this contract — `Sitecore.Ecommerce.Carts.ShoppingCartLine` — represents the descendants of an order item as described in the section *The SES Order*.

2.8.2 Extending the ShoppingCartLine

When a product is added to a shopping cart, a shopping cart line is created. The shopping cart line represents a product in the cart. An add-on is a product that is added to a cart, but some additional information must be recorded. You need to know if the product is an add-on for another product. This can be accomplished by saving the product code for the *parent* product.

This section describes how to extend the class that represents a `ShoppingCartLine` in order to accommodate this information.

1. In Visual Studio, add a new class named `Sitecore.MySES.Extensions.AddOn.ShoppingCartLine`.
2. Add the following code to the class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Sitecore.Marketing.SES.Extensions.AddOn
{
    public class ShoppingCartLine : Sitecore.Ecommerce.Carts.ShoppingCartLine
    {
        public string ParentProductCode { get; set; }
    }
}
```

2.8.3 Extending the ShoppingCartManager

You can use the `ShoppingCartManager` class to create the `ShoppingCartLine` and to add the `ShoppingCartLine` to the cart.

This section describes how to extend the class that represents the `ShoppingCartManager` to accommodate the *parent* product code.

1. In Visual Studio, add a new class named `Sitecore.MySES.Extensions.AddOn.ShoppingCartManager`.

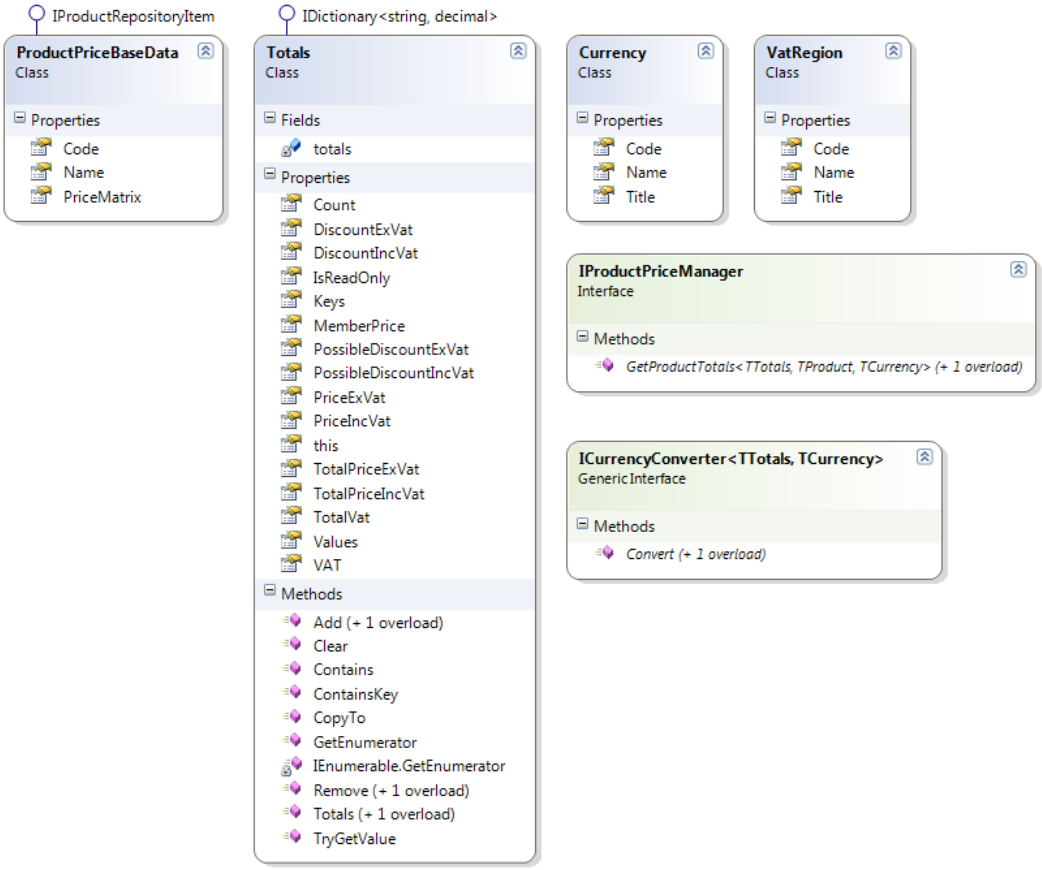
2. Add the following code to the class:

```
        using System;
        using System.Collections.Generic;
        using System.Linq;
        using System.Text;
        using Sitecore.Ecommerce;
        using Sitecore.Ecommerce.Carts;
        using Sitecore.Ecommerce.DomainModel.Prices;
        using Sitecore.Ecommerce.DomainModel.Products;
        using Sitecore.Ecommerce.DomainModel.Currencies;

        namespace Sitecore.Marketing.SES.Extensions.AddOn
        {
            public class ShoppingCartManager : Sitecore.Ecommerce.Carts.ShoppingCartManager
            {
                public ShoppingCartManager(IProductRepository productRepository,
                                           IProductPriceManager productPriceManager)
                    : base(productRepository, productPriceManager)
                {
                }
                protected virtual ShoppingCartLine GetShoppingCartLine(
                                           string parentProductCode,
                                           string addonProductCode)
                {
                    var product = GetProduct(parentProductCode);
                    var addon = GetProduct(addonProductCode);
                    var cart = Sitecore.Ecommerce.Context.Entity.GetInstance<ShoppingCart>();
                    foreach (var line in cart.ShoppingCartLines)
                    {
                        if (string.Equals(line.Product.Code, addonProductCode))
                        {
                            var line2 = line as ShoppingCartLine;
                            if (line2 == null)
                            {
                                continue;
                            }
                            if (string.Equals(line2.ParentProductCode, parentProductCode))
                            {
                                return line2;
                            }
                        }
                    }
                    return null;
                }

                protected virtual ProductBaseData GetProduct(string productCode)
                {
                    if (string.IsNullOrEmpty(productCode))
                    {
                        return null;
                    }
                    var repository =
Sitecore.Ecommerce.Context.Entity.Resolve<IProductRepository>();
                    return repository.Get<ProductBaseData>(productCode);
                }

                public virtual void AddAddOn(string productCode, string parentProductCode)
                {
                    var line = this.GetShoppingCartLine(productCode, parentProductCode);
                    if (line != null)
                    {
                        return;
                    }
                    line = Sitecore.Ecommerce.Context.Entity.Resolve<ShoppingCartLine>();
                    line.Product = GetProduct(productCode);
                    line.ParentProductCode = parentProductCode;
                    line.Quantity = 1;
                    var cart = Sitecore.Ecommerce.Context.Entity.GetInstance<ShoppingCart>();
                    var mgr =
Sitecore.Ecommerce.Context.Entity.GetInstance<IProductPriceManager>();
```

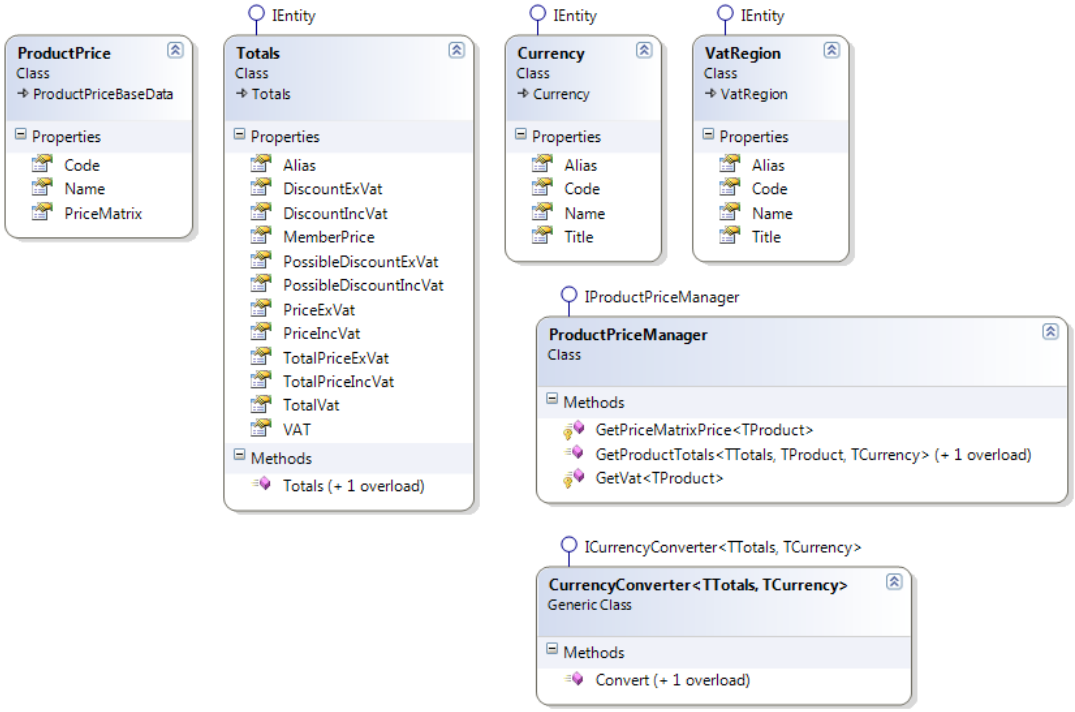
```
                    line.Totals = mgr.GetProductTotals<Totals, ProductBaseData, Currency>(
                                                            line.Product, cart.Currency,
line.Quantity);
                    cart.ShoppingCartLines.Add(line);
                }
            }
        }
```

2.9 The SES Pricing Components

The following class diagram gives you an overview of the pricing contracts:



The following class diagram gives you an overview of the pricing implementation.



2.9.1 The Pricing Contracts

The following table describes each of the pricing related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`Totals`	`Sitecore.Ecommerce.DomainModel.Prices.Totals` implements the `IDictionary` interface and exposes information about pricing totals for an order. The default implementation of this contract — `Sitecore.Ecommerce.Prices.Totals` — stores data in session during transactions and persists that data in order items as described in the section, *The SES Order*.
`VatRegion`	`Sitecore.Ecommerce.DomainModel.Addresses.VatRegion` exposes information about a tax region.
`IProductPriceManager`	`Sitecore.Ecommerce.DomainModel.Prices.IProductPriceManager` defines a programming interface for product pricing. This contract has two implementations: • The `ProductPriceManager` class in the Kernel. This implementation calculates the price for a product. As different prices apply to different customers, a *Totals* object is used to represent the price. The base price comes from the pricing information stored on the product definition item in the Product Meta Info section in the Price field. The VAT rate that is associated with the product is also included in this calculation.

Contract	Description

	Example:
	`public void GetProductTotalsTest()`
	`{`
	`IProductRepository productProvider = Context.Entity.Resolve<IProductRepository>();`
	`ProductBaseData product = productProvider.Get<ProductBaseData>(this.ProductItemId.ToString());`
	`IProductPriceManager productPriceManager = Context.Entity.Resolve<IProductPriceManager>();`
	`Totals totals = productPriceManager.GetProductTotals(product);`
	`}`
	• The `RemoteProductStockManager` class in the Service model — This implementation is a service that is used when the content management and content delivery systems have been separated.
	
	For more information, see the *SES Scaling Guide*.

`ICurrencyConverter`	There are two currencies in SES: Master and Display currency. You can set them in the General Settings item. The Master currency is defining the default currency used in the product repository and the Display currency is used in case you want to display a different currency at the frontend. If Master and Display currencies are different, the implementation of the contract `Sitecore.Ecommerce.DomainModel.Currencies.ICurrencyConverter is resolved and` is responsible for converting product price from Master currency to Display currency. The default implementation uses the conversion rates from the *Business Catalog*. `Sitecore.Ecommerce.Prices.ProductPriceManager` uses the `ICurrencyConverter` interface. The default implementation of this contract is `Sitecore.Ecommerce.Currencies.CurrencyConverter`.
`Currency`	`Sitecore.Ecommerce.DomainModel.Currencies.Currency` exposes information about a currency. The default implementation — `Sitecore.Ecommerce.Currencies.Currency` — of this contract represents the children of the item specified by the: • Business Catalog, • System Links section, • Currencies Link field (`<home>/Site Settings/Business Catalog`).

Contract	Description
`ProductPriceBaseData`	`Sitecore.Ecommerce.DomainModel.Products.ProductPriceBaseData` represents the product price information. Contains the Price Matrix (XML as simple string) and product code. Implements `IProductRepository`.

2.9.2 Adding a Price Type to the Default IProductPriceManager Implementation

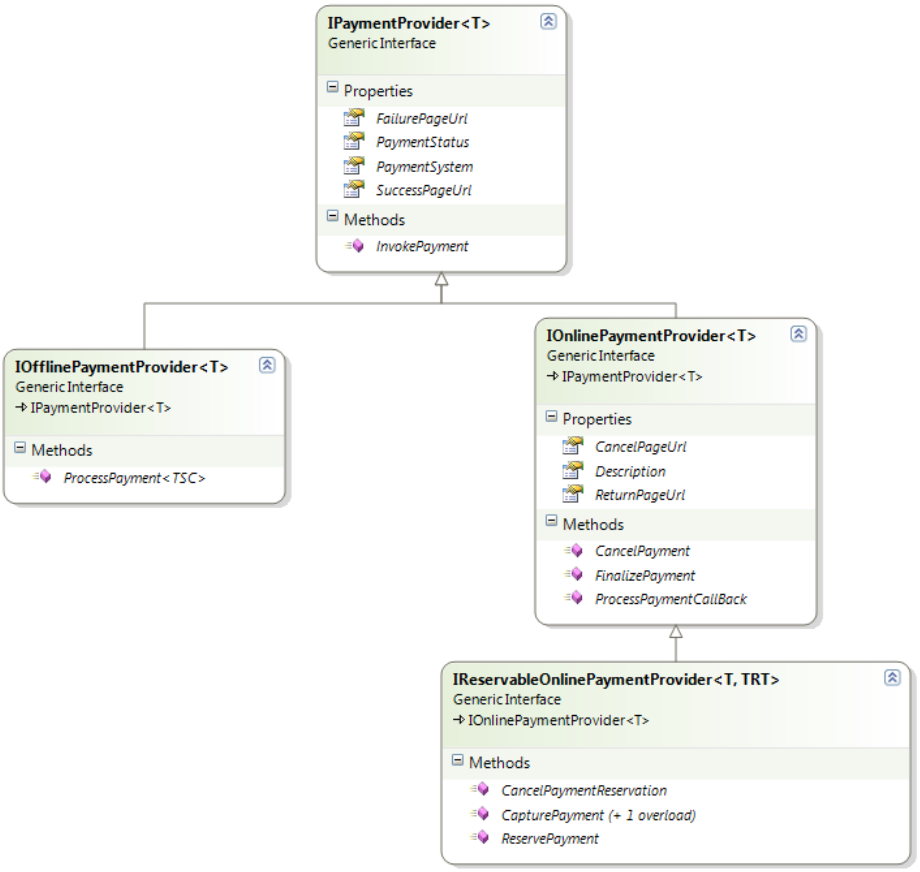
To add a price type to the default `IProductPriceManager` implementation:

1. In the **Content Editor**, select the /`Sitecore/System/Modules/Ecommerce/PriceMatrix/Shop item`.
2. In the **Content Editor**, insert a new price type definition item using the `Ecommerce/Price Field/PriceMatrixPrice` data template.
3. In the new price type definition item, in the **Data** section, in the Title field, enter the label for the new price type.
4. In the **Content Editor**, sort the price type definition items to control their order of appearance in the Price field of product definition items.
5. In the **Content Editor**, edit product definition items. In the Product Meta Info section, in the Price field, enter values for the new price type.
6. Update rendering components to apply the new price type as appropriate.

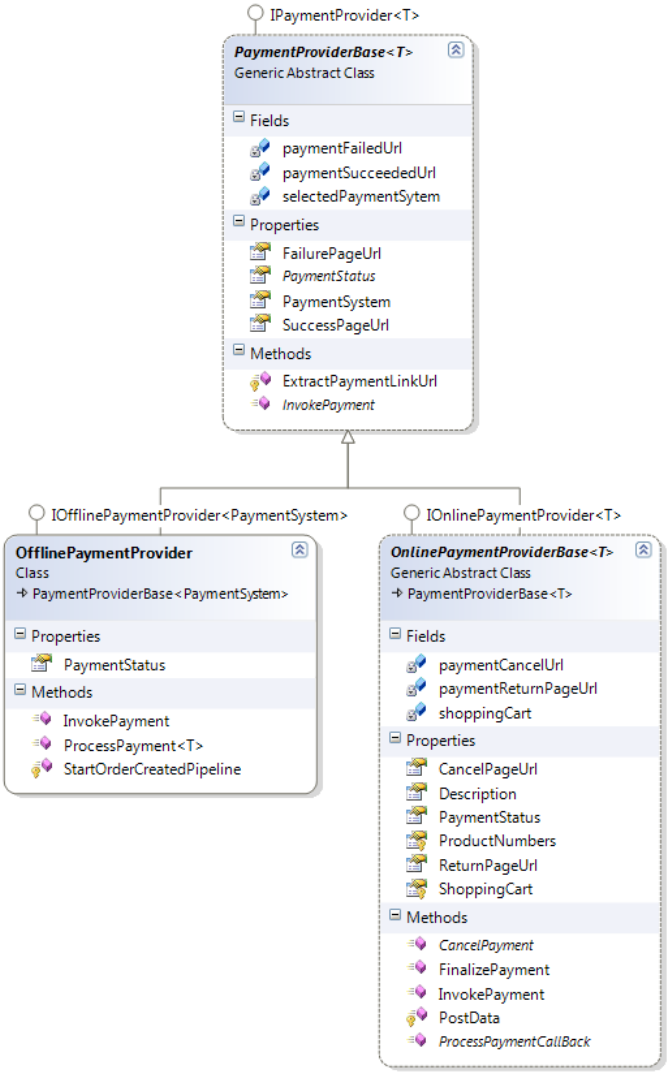
To access the new price type for a product, pass the value of the Title field in the product price type definition item as the second parameter to the `GetPriceMatrixPrice()` method of the `IProductPriceManager` contract.

2.10 The SES Payment Providers Components

The following class diagram gives you an overview of the payment provider contracts.



The following class diagram gives you an overview of the payment provider implementation.



2.10.1 The Payment Providers Contracts

The following table describes each of the payment providers related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

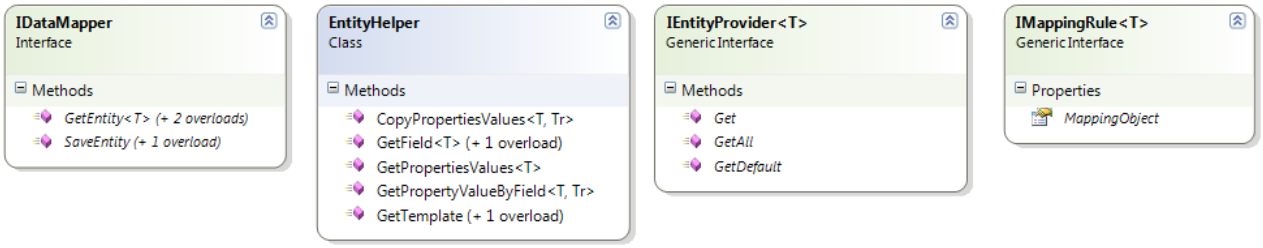
Contract	Description
`PaymentSystem`	`Sitecore.Ecommerce.DomainModel.Payments.PaymentSystem` exposes information about an online payment provider gateway. For more information about payment providers, see the manual *SES Payment Provider Guide*. The default implementation of this contract — `Sitecore.Ecommerce.Payments.PaymentSystem` — represents a child of the item specified by the: • Business Catalog • System Links section

Contract	Description
	• Payment Systems Link field (<home>/Site Settings/Business Catalog).
PaymentProvider	Sitecore.Ecommerce.DomainModel.Payments.PaymentProvider is the base contract for all of the SES payment providers. This contract has two methods: • Invoke • ProcessCallback
IReservable	Sitecore.Ecommerce.DomainModel.Payments.IReservable is an additional contract for payment providers that is used for payment reservation and deferred capturing. This contract has three methods: • Invoke to invoke a payment. • Capture to capture a payment and save the value of the payment as a persistent value in the HTTP context. • CancelReservation to cancel a reservation
ITransactionData	Sitecore.Ecommerce.DomainModel.Payments.ITransactionData defines a programming interface to persist payment transaction information between HTTP requests. The default implementation of this contract — Sitecore.Ecommerce.Payments.TransactionData — stores data in the ASP.NET session.

For more information, see the manual *SES Payment Provider Guide*.

2.11 The SES Content-to-Object Mapping Components

The following class diagram gives you an overview of the object content management data contracts.



2.11.1 The Content-to-Object Mapping Contracts

The following table describes each of the Content-to-Object Mapping (COM) related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description

`IDataMapper`	`Sitecore.Ecommerce.Data.IDataMapper` defines a programming interface to help various data manager objects abstract storage.
	
	The default implementation of this contract — `Sitecore.Ecommerce.Data.DataMapper` — represents data as Sitecore items.
	The default `IDataMapper` implementation uses the `Entity` attribute in .NET to determine the data templates and fields associated with various data elements.
	
	For example, the `Entity` attributes in square brackets ("[]") define the ID of a data template for products and the name of a field in that data template that contains the specified property:
	`[Entity(TemplateId = "{B87EFAE7-D3D5-4E07-A6FC-012AAA13A6CF}")]`
	`public class Product : DomainModel.Products.ProductBaseData, IEntity`
	`{`
	`[Entity(FieldName = "Name")]`
	`public override string Name { get; [NotNullValue] set; }`
	`...`
	}

`EntityHelper`	`Sitecore.Ecommerce.Data.EntityHelper` provides an API that the default implementation of the `IDataMapper` contract uses to access the value of the `Entity` attributes in .NET code. The class that defines the `EntityHelper` contract also serves as the default implementation of the contract.

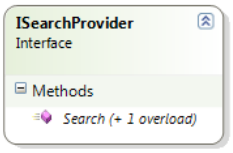
Contract	Description

`IEntityProvider`	`Sitecore.Ecommerce.DomainModel.Data.IEntityProvider` provides an API to access a variety of similar data types.
	
	The default implementation of this contract — `Sitecore.Ecommerce.Data.EntityProvider` — retrieves data of items based on the `Ecommerce/Business Catalog/Option Value` data template or any data template that inherits from that data template. You can use the `IEntityProvider` contract to access information about countries, country states, currencies, delivery alternatives, language option values, notification options, payments, and VAT option values.
	For example, to access information about every country:
	`Using Sitecore.Ecommerce.DomainModel.Data;`
	`Using Sitecore.Ecommerce.DomainModel.Addresses;`
	`...`
	`IEntityProvider<Country> countries = Sitecore.Ecommerce.Context.Entity.Resolve<IEntityProvider<Country>>();`
	`foreach (Country country in countries.GetAllEntities())`
	`{`
	`...`
	`}`
	
	To use country code to access a specific country:
	`Country unitedStates = countries.GetEntityByCode("US");`

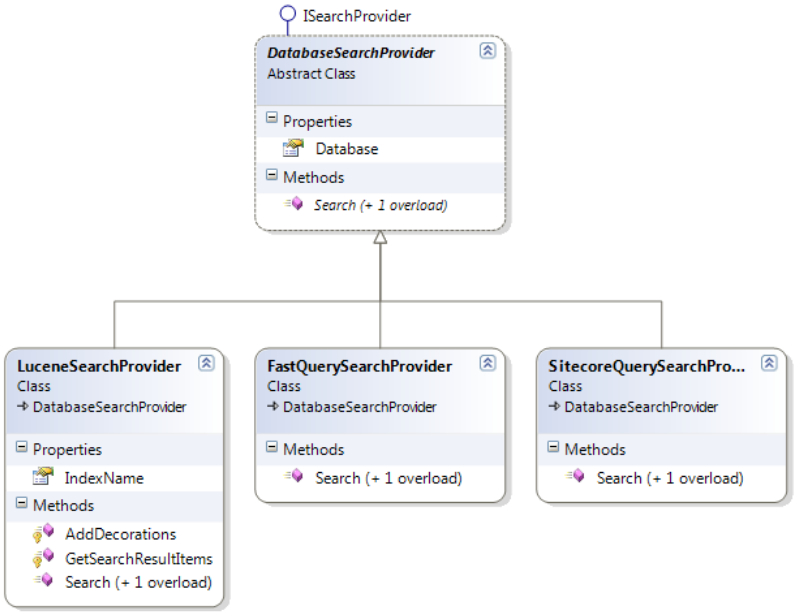
`IMappingRule`	`Sitecore.Ecommerce.Data.IMappingRule` defines a programming interface that represents adapters for mapping between physical and logical storage for complex types, including conversion between system and Sitecore internal data types such as dates in the ISO string format used by Sitecore. Sitecore provides two default implementations of this contract: • The `Order` mapping rule (`Sitecore.Ecommerce.Data.OrderMappingRule`) implementation of the `IMappingRule` contract adapts orders from items in the content tree. • The `OrderLine` mapping rule (`Sitecore.Ecommerce.Data.OrderLineMappingRule`) implementation of the `IMappingRule` contract adapts order lines from items in the content tree. The default implementation of this contract uses Unity to determine which `IMappingRule` to use. The default configuration uses `OrderMappingRule` and `OrderLineMappingRule`. However, you could change the `Unity.config` file to use different `IMappingRule` objects.

2.12 The SES Search Provider Components

The following class diagram gives you an overview of the search provider contracts:



The following class diagram gives you an overview of the search provider implementation:



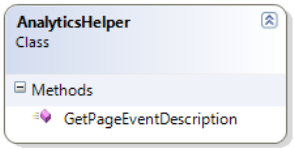
2.12.1 The Search Provider Contracts

The following table describes each of the contracts that are related to the payment providers. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`ISearchProvider`	`Sitecore.Ecommerce.Search.ISearchProvider` defines a programming interface for locating items that match specific criteria. SES provides three implementations of this contract: • The Lucene search provider (`Sitecore.Ecommerce.Search.LuceneSearchProvider`). • The Sitecore Query search provider (`Sitecore.Ecommerce.Search.SitecoreQuerySearchProvider`). • The Sitecore Fast Query search provider (`Sitecore.Ecommerce.Search.FastQuerySearchProvider`).

2.13 The SES Analytics Component

The following class diagram gives you an overview of the analytics contract.



2.13.1 The Analytics Contract

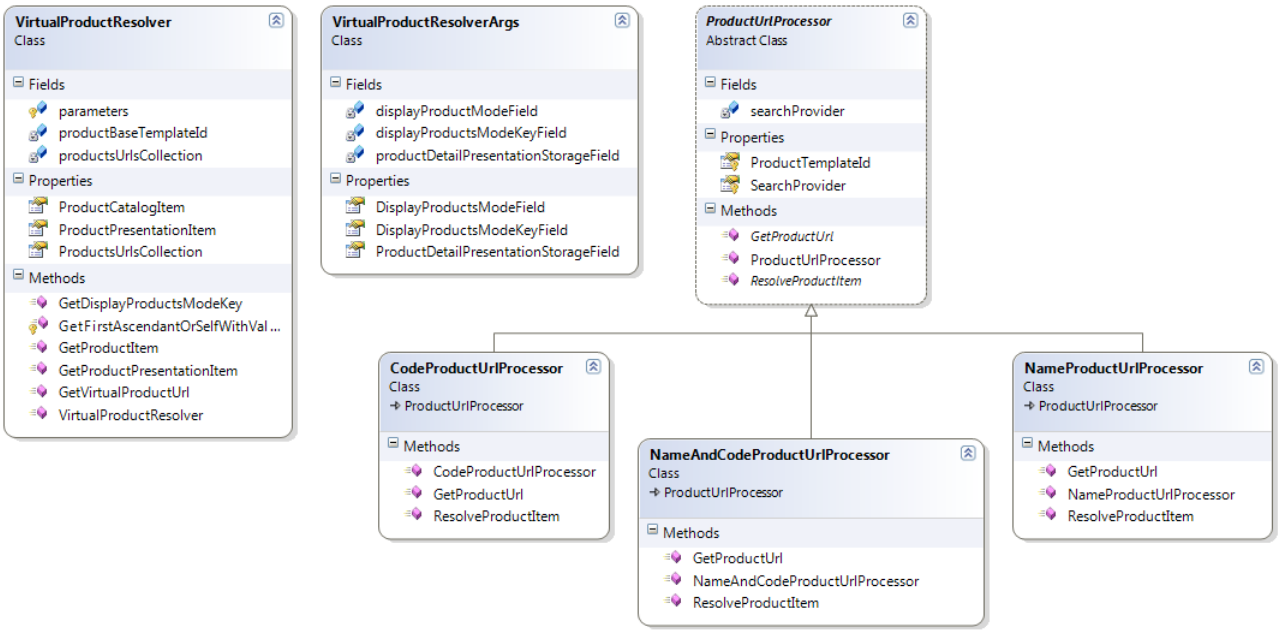
The following table describes the analytics contract. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`AnalyticsHelper`	This contract supports the integration between the Sitecore Digital Marketing System (DMS) and SES. For more information about the DMS, see http://www.sitecore.net/en/Products/Digital-Marketing-System.aspx For more information about using APIs to access SES events, see the classes in the `Sitecore.Ecommerce.Analytics.Components.PageEvents` namespace and the `Sitecore.Ecommerce.Analytics.AnalyticsHelper` class.

For more information about SES and DMS, see the manual *SES DMS Cookbook*.

2.14 The SES Product Resolver Components

The following class diagram gives you an overview the search product resolver contracts.



2.14.1 The Product Resolver Contracts

The following table describes each of the product resolver related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`ProductUrlProcessor`	`Sitecore.Ecommerce.Catalogs.ProductUrlProcessor` defines two programming interfaces — one that determines the URL of a product item and another that determines the product specified by a URL. Product resolvers control how SES constructs and parses the URLs of product pages. SES provides multiple implementations for the `ProductUrlProcessor` contract: • `Sitecore.Ecommerce.Catalogs.NameProductUrlProcessor` that uses product names. • `Sitecore.Ecommerce.Catalogs.CodeProductUrlProcessor` that uses product codes. • `Sitecore.Ecommerce.Catalogs.NameAndCodeProductUrlProcessor` that uses product names and codes. By default, the product URLs begin with the path to the page that links to the product. For example, if the *Products* item of a managed website contains a link to a product called *product_name* with a code called *product_id*, the default URL that is generated for that

Contract	Description
	product is `/products/product_name.aspx`, `/products/product_name_product_id.aspx`, or `/products/product_id.aspx`, depending on the `ProductUrlProcessor` implementation that SES applies. For more information about the `ProductUrlProcessor` implementation that SES applies, see the section *How to Specify the Product URL Format* in the *SES Developer's Cookbook*.
`VirtualProductResolver`	`Sitecore.Ecommerce.Catalogs.VirtualProductResolver` defines an API to determine the Sitecore item that represents a product. This item is specified by a URL generated by a `ProductUrlProcessor` implementation. The `VirtualProductResolver` contract applies the `ProductUrlProcessor` contract that is appropriate in the context to determine the item specified by the URL. The `ProductResolver` processor that SES adds to the `httpRequestBegin` pipeline defined in the `Web.config` file uses the `VirtualProductResolver` to determine the item associated with a requested URL. The class that defines the `VirtualProductResolver` contract also serves as the default implementation of the `VirtualProductResolver` contract. For more information about product URLs and product resolution, see the corresponding sections.
`VirtualProductResolverArgs`	`Sitecore.Ecommerce.Catalogs.VirtualProductResolverArgs` is an argument class that wraps parameters for passing it in the SES model.

2.14.2 Adding a ProductUrlProcessor Implementation

You can add a `ProductUrlProcessor` implementation to define a custom format for product URLs.

To add an implementation of the `ProductUrlProcessor` contract:

1. In the Visual Studio project, add a class that inherits from the `ProductUrlProcessor` base class — `Sitecore.Ecommerce.Catalogs.ProductUrlProcessor`.
2. In the new class, implement a constructor that accepts an object based on the `ISearchProvider` contract.

   For more information about the `ISearchProvider` contract, see the description of the `ISearchProvider` contract.
3. In the new class, implement the `GetProductUrl()` method to return the URL to use for a product.
4. In the new class, implement the `ResolveProductItem()` method to return the product item associated with a URL of a product.
5. In the Unity configuration, add a `/unity/alias` element. Set the name attribute of the new /unity/alias element to the name of the class. Set the type attribute of the new `/unity/alias` element to the .NET type of the class.

Example:

```
<alias name="MyProductUrlProcessor"
  type="MyNamespace.MyProductUrlProcessor, MyAssembly"/>
```

6. In the Unity configuration, add a `/unity/container/register` element. Set the type attribute of the new `/unity/container/register` element to `ProductUrlProcessor`. Set the `mapTo` attribute of the new `/unity/container/register` element to the name attribute of the new `/unity/alias` element. Set the name attribute of the new `/unity/container/register` element to a unique prefix based on the implementation, such as `My`. Copy the elements enclosed in one of the other `/unity/container/register` elements with a value of `ProductUrlProcessor` for the type attribute.

   Example:

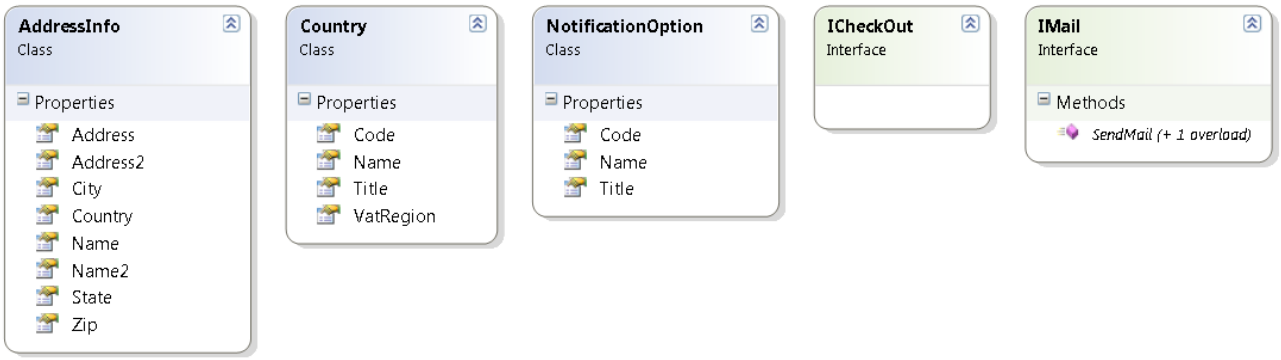
```
<register type="ProductUrlProcessor"
  mapTo="MyProductUrlProcessor" name="My">
  <lifetime type="perthread"/>
  <constructor>
    <param name="searchProvider">
      <dependency name="FastQuerySearchProvider"/>
    </param>
  </constructor>
</register>
```

7. In the **Content Editor**, beneath the `/Sitecore/System/Modules/Ecommerce/System/Display Product Modes` item, insert a `ProductUrlProcessor` definition item using the `Ecommerce/Settings/Settings` Item data template. Give the new `ProductUrlProcessor` definition item a meaningful name based on the implementation, such as `MyProductUrlProcessor`.

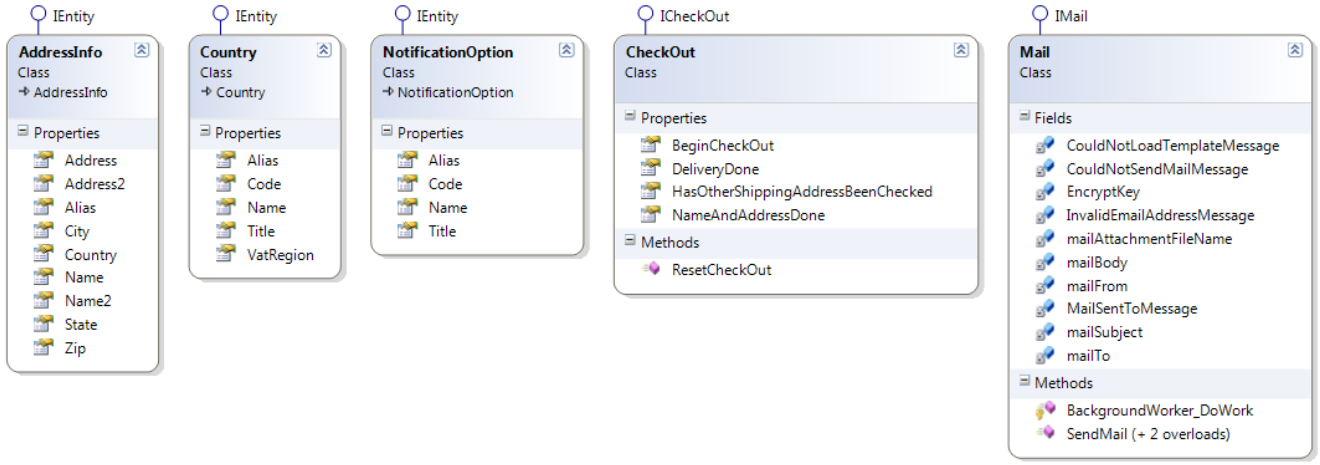
8. In the new `ProductUrlProcessor` definition item, in the **Data** section, in the **Key** field, enter the value of the name attribute of the new `/unity/container/register` element, for example `My`.

2.15 Miscellaneous SES Components

The following class diagram gives you an overview of the miscellaneous contracts.



The following class diagram gives you an overview of the miscellaneous implementation.



2.15.1 Miscellaneous Contracts

The following table describes each of the miscellaneous contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`AddressInfo`	`Sitecore.Ecommerce.DomainModel.Addresses.AddressInfo` exposes information about a physical address. The default implementation of this contract — `Sitecore.Ecommerce.Addresses.AddressInfo` — represents typical address information.
`Country`	`Sitecore.Ecommerce.DomainModel.Addresses.Country` exposes information about a country. The default implementation of this contract — `Sitecore.Ecommerce.Addresses.Country` — represents the children of the item specified by the: • Business Catalog item

Contract	Description
	• System Links Section • Countries Link field (<home>/Site Settings/Business Catalog).
NotificationOption	Sitecore.Ecommerce.DomainModel.Shippings.NotificationOption exposes information about how a customer prefers to receive notification about the status of an order. The default implementation of this contract — Sitecore.Ecommerce.Shippings.NotificationOption — specifies that Sitecore sends an e-mail to customers about each order that they place on the webshop.
ICheckOut	Sitecore.Ecommerce.DomainModel.CheckOuts.ICheckOut defines a programming interface to determine or alter the state of the shopping checkout process. Before Sitecore renders a checkout page, the checkout page accesses the properties and methods in the default implementation of the ICheckOut contract to ensure that the preceding process has been completed.
IMail	Sitecore.Ecommerce.DomainModel.Mails.IMail is used to send e-mails using a template-based or a custom method. It defines a programming interface for sending e-mail. The default implementation of this contract — Sitecore.Ecommerce.Mails.Mail — uses the MailServer, MailServerUserName, MailServerPassword, and MailServerPort settings in the Web.config file.

Chapter 3

Appendix — SES Order Manager Components

This chapter describes the SES Order Manager components that were used in earlier version of SES.

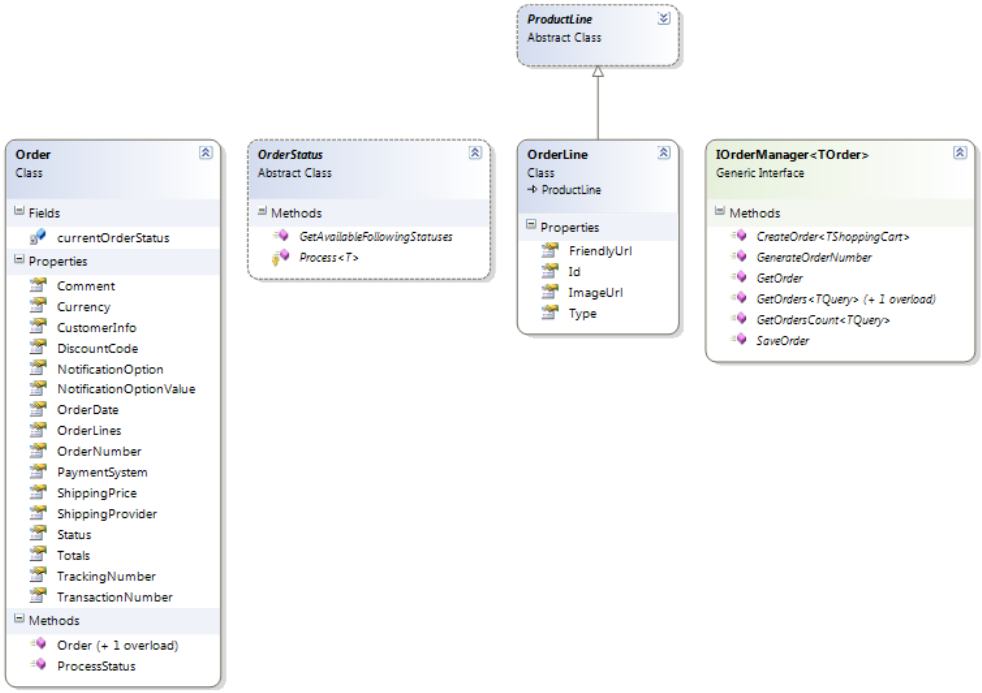
The Order Manager is introduced with Sitecore E-Commerce 2.2 and replaces these components.

This chapter contains the following sections:

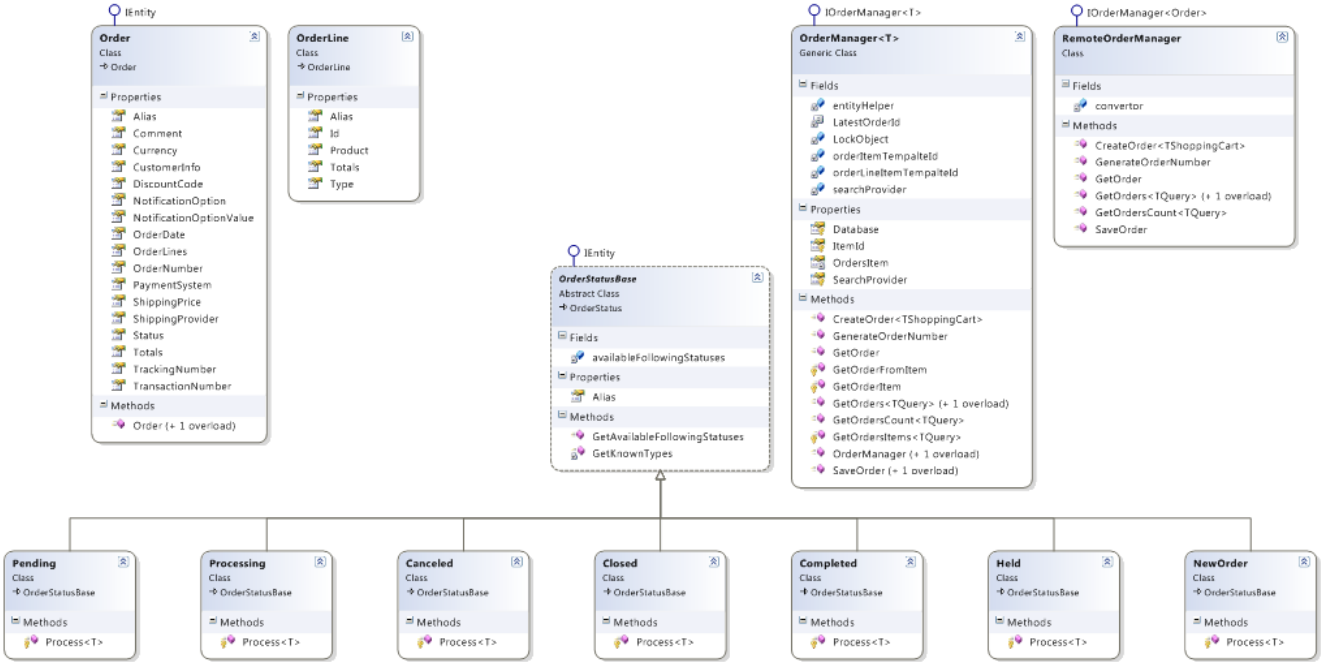
- The SES Order Components

3.1 The SES Order Components

The following class diagram gives you an overview of the order contracts:



The following class diagram gives you an overview of the order implementation:



3.1.1 The Order Contracts

The following table describes each of the order related contracts. It presents the contract's functionality and default implementation. It also presents the parent contract that this class implements.

Contract	Description
`Order`	`Sitecore.Ecommerce.DomainModel.Orders.Order` exposes information about individual orders. The default implementation of this contract — `Sitecore.Ecommerce.Orders.Order` — represents the descendants of the item specified in the Business Catalog in the **Orders Link** field of the current site — (`<home>/Site Settings/Business Catalog`). To integrate an external order management system, you do not need to implement the `Order` contract. Instead, implement the `IOrderManager` contract to manage orders.
`OrderStatus`	`Sitecore.Ecommerce.DomainModel.Orders.OrderStatus` presents the status of an order. **Note** There is a one-to-one mapping between statuses defined in the CMS content for a webshop and the status types registered in Unity. There cannot be a status defined in the content without also being registered in Unity. The `OrderStatus` contract exposes a method called `Process` that executes the business logic whenever the order enters the state. Each of the following contract implementations can contain logic to apply when the system updates the status of an order. The default `OrderStatus` implementations include: • Completed (`Sitecore.Ecommerce.Orders.Statuses.Completed`) • Closed (`Sitecore.Ecommerce.Orders.Statuses.Closed`) • Held (`Sitecore.Ecommerce.Orders.Statuses.Held`) • Pending (`Sitecore.Ecommerce.Orders.Statuses.Pending`) • Processing (`Sitecore.Ecommerce.Orders.Statuses.Processing`) • Canceled (`Sitecore.Ecommerce.Orders.Statuses.Canceled`) • New (`Sitecore.Ecommerce.Orders.Statuses.New`) • Captured (`Sitecore.Ecommerce.Orders.Statuses.Captured`)
`OrderLine`	`Sitecore.Ecommerce.DomainModel.Orders.OrderLine` implements the `ProductLine` class and exposes information about an order line item on an order. The default implementation of this contract — `Sitecore.Ecommerce.Orders.OrderLine` — represents the

Contract	Description
	descendants of an order item.
`IOrderManager`	`Sitecore.Ecommerce.DomainModel.Orders.IOrderManager` defines a programming interface for managing orders. This contract has two implementations: • `OrderManager` in the Kernel — This implementation accesses the descendants of the item specified in the Business Catalog in the **Orders Link** field of the context site — (`<home>/Site Settings/Business Catalog`). **Note** This implementation writes order information to the Sitecore Master database. • The `RemoteOrderManager` in the Service model — This implementation is a service that is used when the content management and content delivery systems have been separated. For more information, see the *SES Scaling Guide*.

3.1.2 Implementing the Order Contract

To implement the `Order` contact:

1. In the Visual Studio project, create a class that implements the `Order` contract— `Sitecore.Ecommerce.DomainModel.Orders.Order`— to store information about an order.
2. In the new class, implement a constructor that accepts an object that implements the `OrderStatus` contract.

   For more information, see the description of the `OrderStatus` contract.
3. You can also implement the `OrderLine` contract.

   For more information, see the description of the `OrderLine` contract.
4. Update the Unity configuration to use your implementation of the new `Order` implementation.

   For more information about updating the Unity configuration, see the section *How to Replace a SES Component* in the *SES Developer's Cookbook*.

Example:

```
<alias alias="MyOrder" type="MyNamespace.MyOrder, MyAssembly"/>
...
<register type="Order" mapTo="MyOrder">
...
```

3.1.3 Overriding an OrderStatus Implementation

The `OrderStatus` contract exposes a method called `Process` that executes the business logic whenever the order reaches the state. For example, you may need to replace the logic executed for the status `Pending`.

To override the logic that SES applies when an order reaches an existing order status:

1. In the Visual Studio project, create a class that inherits from the `Sitecore.Ecommerce.Orders.Statuses.OrderStatusBase` class or from the class that provides the default implementation of the order status.

2. In the new class, implement the `Process()` method, which may call the `Process()` method in the base class.
3. In the Unity configuration, create a new `/unity/alias` element to register the new implementation.

   For more information about adding an implementation to Unity configuration, see the section *How to Add an Implementation to the Unity Configuration* in the *SES Developer's Cookbook.*
4. In the Unity configuration, update the `/unity/container/register` element for the order status to use your implementation.

   For more information about updating Unity configuration, see the section *How to Replace a SES Component* in the *SES Developer's Cookbook*.

3.1.4 Implementing a New Order Status

To implement a new order status:

1. In the Visual Studio project, create a class that inherits from the `Sitecore.Ecommerce.Orders.Statuses.OrderStatusBase` class.
2. In the new class, implement the `Process()` method to contain logic for SES to be applied when placing the order into that status.
3. In the Unity configuration, add a `/unity/alias` element to register the new implementation.

   For more information about adding an implementation to Unity, see the section *How to Add an Implementation to the Unity Configuration* in the *SES Developer's Cookbook*.

   Example:

```
        <alias alias="ShippedOrderStatus" type="MyNamespace.ShippedOrderStatus,
MyAssembly"/>
```

4. In the Unity configuration, add a `/unity/container/register` element to define a mapping for the new implementation. Set the type attribute of the new `/unity/container/register` element to `OrderStatus`. Set the mapTo attribute of the new `/unity/container/register` element to the alias attribute of the new `/unity/alias` element. Set the name attribute of the `/unity/container/register` element to identify the status.

   Example:

```
        <register type="OrderStatus" mapTo="ShippedOrderStatus" name="Shipped">
          <interceptor type="VirtualMethodInterceptor"/>
          <policyInjection/>
        </register>
```

5. In the **Content Editor**, select the item specified in the field named **Order Statuses Link** in the **System Links** section of the child named **Business Catalog** of the **Site Settings** child of the home item of the managed website—`<home>/Site Settings/Business Catalog`
6. In the **Content Editor**, insert an order status definition item using the `Ecommerce/Business Catalog/Order` Status data template.
7. In the new order status definition item, in the **Data** section, in the **Code** field, enter the name attribute of the new `/unity/container/register` element in the Unity configuration.
8. In the new order status definition item, in the **Data** section, in the **Title** field, enter the label that should appear in the user interface to transition an order to this status. Enter the same value for the **Name** field in the **Data** section.
9. In the new order status definition item, in the **Data** section, in the **Available List** field, select the order statuses that the user can apply to an order currently associated with this order status.

3.1.5 Assigning an Order Status

In order to set an order status value, it needs to be created first. You can use the `Sitecore.Ecommerce.Entity.Resolve()` method to *resolve* an order status. *Resolving* is a Unity's method of creating a new instance of a specific type. The following code snippet shows you how to assign the Shipped order status to an order:

Example:

```
using Sitecore.Ecommerce.DomainModel.Orders;
...
IOrderManager<Order> orderManager = Sitecore.Ecommerce.Context.Entity.Resolve
  <IOrderManager<Order>>();
Order order = orderManager.GetOrder("order number");
order.Status = Sitecore.Ecommerce.Context.Entity.Resolve<OrderStatus>("Shipped");
orderManager.SaveOrder(order);
```

3.1.6 Integrating an Order Management System

To integrate an external order management system:

1. Optionally, implement the Order contract.

   For more information about the Order contract, see the section *The Order Contracts*.

2. In the Visual Studio project, create a class that implements the `IOrderManager` contract to abstract the order management system.

3. In the new class, implement the `GetOrder()` method to retrieve information about an order from the external order management system, and return an object that implements the Order contract to contain that information.

4. In the new class, implement the `GetOrders()` method to retrieve orders matching a given query from the external order management system.

5. In the new class, implement the `CreateOrder()` method to create an order in the external order management system.

6. In the new class, implement the `SaveOrder()` method to update an order in the external order management system.

7. In the new class, implement the `GenerateOrderNumber()` method to generate an order number appropriate for the external order management system.

8. In the Unity configuration, add an element in `/alias/alias` for your `IOrderManager` implementation.

   For more information about adding an implementation to the Unity configuration, see the section *How to Add an Implementation to the Unity Configuration* in the *Sitecore E-Commerce Developer's Cookbook*.

9. Configure SES to use the `IOrderManager` implementation. Update the `mapTo` attribute of the `/unity/container/register` element named `IOrderManager` to the value of the alias attribute of the new `/unity/alias` element that specifies your `IOrderManager` implementation.

   For more information about configuring SES to use your implementation, see the section *How to Replace a SES Component* in the *SES Developer's Cookbook*.

For more information about Unity configuration, including instructions to use different implementations under different conditions, see the section *Dependency Injection* in the *SES Developer's Cookbook*.

For an example about extending the `OrderManager`, see the section *Extending the OrderManager*.

Note
If you integrate SES with an external order management system, Sitecore recommends that you also write orders data to Sitecore, so that the website can continue to process orders even when the external order management system is unavailable.

3.1.7 Extending the OrderLine

In the same way as a `ShoppingCartLine` represents a product in a cart, an `OrderLine` represents a product in an order. When an add-on product is added to an order, the corresponding `OrderLine` needs to be able to store the *parent* product code.

This section describes how to extend the class that represents an `OrderLine` to accommodate the *parent* product code.

1. In Visual Studio, add a new class named

   `Sitecore.MySES.Extensions.AddOn.OrderLine`

2. Add the following code to the class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Sitecore.Marketing.SES.Extensions.AddOn
{
    public class OrderLine : Sitecore.Ecommerce.Orders.OrderLine
    {
        public string ParentProductCode { get; set; }
    }
}
```

3.1.8 Extending the OrderLine Data Template

In the previous section, you extended the `OrderLine` object to store a *parent* product code. By default, SES uses Sitecore items to store order lines. Since the `OrderLine` has been extended, the data template that represents an order line in Sitecore must also be extended.

This section explains how to extend the data template that represents an order line.

1. In the **Content Editor**, select the `/sitecore/templates/Ecommerce/Order/OrderLine` item.
2. Create a field named `ParentProductCode` in the **Data** section.

   Set the following properties:

   Type: *Single-Line Text*

   Unversioned: *checked*

   Shared: *checked*

3.1.9 Extending the OrderManager

The `ShoppingCartManager` class stores information in memory, so its logic is pretty basic. The `OrderManager` class does a lot more, but the basic idea is simple enough — take information from one place (a cart) and save it to another (an order).

This section describes how to extend the `OrderManager` class in order to accommodate for the *parent* product code.

1. In Visual Studio, add a new class called `Sitecore.MySES.Extensions.AddOn.OrderManager`.

2. Add the following code to the class:

```
        using System;
        using System.Collections.Generic;
        using System.Linq;
        using System.Text;
        using Microsoft.Practices.Unity;
        using Sitecore.Configuration;
        using Sitecore.Data;
        using Sitecore.Data.Items;
        using Sitecore.Diagnostics;
        using Sitecore.Ecommerce.Data;
        using Sitecore.Ecommerce.DomainModel.Carts;
        using Sitecore.Ecommerce.DomainModel.Data;
        using Sitecore.Ecommerce.DomainModel.Payments;
        using Sitecore.Ecommerce.Orders.Statuses;
        using Sitecore.Ecommerce.Payments;
        using Sitecore.Ecommerce.Search;
        using Sitecore.Ecommerce.Utils;
        using Sitecore.SecurityModel;

        namespace Sitecore.Marketing.SES.Extensions.AddOn
        {
            public class OrderManager<T> : Sitecore.Ecommerce.Orders.OrderManager<T>
                                           where T :
Sitecore.Ecommerce.DomainModel.Orders.Order
            {
                public OrderManager()
                    : base()
                {
                }

                public OrderManager(ISearchProvider searchProvider)
                    : base(searchProvider)
                {
                }

                private static TemplateItem _orderItemTemplate = null;
                protected virtual TemplateItem OrderItemTemplate
                {
                    get
                    {
                        if (_orderItemTemplate == null)
                        {
                            var id =
Settings.GetSetting("Ecommerce.Order.OrderItemTempalteId");
                            _orderItemTemplate = this.Database.GetTemplate(new ID(id));
                        }
                        return _orderItemTemplate;
                    }
                }

                private static TemplateItem _orderLineItemTemplate = null;
                protected virtual TemplateItem OrderLineItemTemplate
                {
                    get
                    {
                        if ( orderLineItemTemplate == null)
                        {
                            var id =
Settings.GetSetting("Ecommerce.Order.OrderLineItemTempalteId");
                            _orderLineItemTemplate = this.Database.GetTemplate(new ID(id));
                        }
                        return _orderLineItemTemplate;
                    }
                }

                protected virtual Item OrdersItem
                {
                    get
                    {
                        Assert.IsNotNull(this.Database, "Orders database not found.");
                        return this.Database.GetItem(this.ItemId);
                    }
```

```
        }

        protected virtual T CreateOrderEntity<TShoppingCart>(
                                TShoppingCart shoppingCart) where TShoppingCart :
ShoppingCart
        {
            var orderEntity = Sitecore.Ecommerce.Context.Entity.Resolve<T>();
            var entityHelper =
Sitecore.Ecommerce.Context.Entity.Resolve<EntityHelper>();
            entityHelper.CopyPropertiesValues<TShoppingCart, T>(shoppingCart, ref
orderEntity);
            return orderEntity;
        }

        protected virtual void AddOrderLines<TShoppingCart>(
               T orderEntity, TShoppingCart shoppingCart) where TShoppingCart :
ShoppingCart
        {
            foreach (ShoppingCartLine cartLine in shoppingCart.ShoppingCartLines)
            {
                var orderLine = ConvertToOrderLine(cartLine);
                orderEntity.OrderLines.Add(orderLine);
            }
        }

        protected virtual OrderLine ConvertToOrderLine<TShoppingCartLine>(
                        TShoppingCartLine cartLine) where TShoppingCartLine :
ShoppingCartLine
        {
            var orderLine = Sitecore.Ecommerce.Context.Entity.Resolve<OrderLine>();
            orderLine.Product = cartLine.Product;
            orderLine.Totals = cartLine.Totals;
            orderLine.Quantity = cartLine.Quantity;
            orderLine.FriendlyUrl = cartLine.FriendlyUrl;
            orderLine.ParentProductCode = cartLine.ParentProductCode;
            return orderLine;
        }

        protected virtual void SetOrderDetails<TShoppingCart>(
               T orderEntity, TShoppingCart shoppingCart) where TShoppingCart :
ShoppingCart
        {
            var tData = Sitecore.Ecommerce.Context.Entity.Resolve<ITransactionData>();
            var persistentValue = tData.GetPersistentValue(
                               shoppingCart.OrderNumber,
TransactionConstants.TransactionNumber);
            var transactionNumber = TypeUtil.TryParse<string>(persistentValue,
string.Empty);
            if (!string.IsNullOrEmpty(transactionNumber))
            {
                orderEntity.TransactionNumber = transactionNumber;
            }
            orderEntity.OrderDate = System.DateTime.Now;
        }

        protected virtual void SetOrderStatus<TShoppingCart>(
               T orderEntity, TShoppingCart shoppingCart) where TShoppingCart :
ShoppingCart
        {
            orderEntity.Status =
Sitecore.Ecommerce.Context.Entity.Resolve<NewOrder>();
            orderEntity.ProcessStatus();
        }

        protected virtual void SaveOrder<TShoppingCart>(
               T orderEntity, TShoppingCart shoppingCart) where TShoppingCart :
ShoppingCart
        {
            Item orderItem;
            using (new SecurityDisabler())
            {
                orderItem = this.OrdersItem.Add(shoppingCart.OrderNumber,
                                                this.OrderItemTemplate);
                Assert.IsNotNull(orderItem, "Failed to create to order item");
```

```
                    if (orderEntity is IEntity)
                    {
                        ((IEntity)orderEntity).Alias = orderItem.ID.ToString();
                    }
                }
                try
                {
                    this.SaveOrder(orderItem, orderEntity);
                }
                catch
                {
                    using (new SecurityDisabler())
                    {
                        orderItem.Delete();
                    }
                    throw;
                }
            }

            public override T CreateOrder<TShoppingCart>(TShoppingCart shoppingCart)
            {
                Assert.IsNotNull(shoppingCart, "Shopping Cart is null");
                //Creating a new order means creating a new Sitecore item,
                //so get data template must be specified
                Assert.IsNotNull(this.OrderItemTemplate, "Order item template is null");
                var orderEntity = CreateOrderEntity(shoppingCart);
                AddOrderLines(orderEntity, shoppingCart);
                SetOrderDetails(orderEntity, shoppingCart);
                SetOrderStatus(orderEntity, shoppingCart);
                SaveOrder(orderEntity, shoppingCart);
                return orderEntity;
            }
        }
    }
```

3.1.10 Extending the OrderLineMappingRule

SES handles the work of creating the Sitecore items needed to accommodate an order and its order lines, as long as the order information is provided to SES. The `OrderManager` handles the order information in SES. The `OrderManager` also specifies which data template should be used for the order and order lines.

One thing that is not specified in the `OrderManager`, however, is the mapping of entity values to Sitecore item fields. This *mapping rule* is the entity that handles this mapping. Since you added a new field on the `OrderLine` class and the `OrderLine` data template, you need to define the mapping between the two.

This section describes how to extend the class that represents the `OrderLineMappingRules` in order to map the new property in the `OrderLine` class to the corresponding field on the `OrderLine` data template.

1. In Visual Studio, add a new class named `Sitecore.MySES.Extensions.AddOn.OrderLineMappingRule`.
2. Add the following code to the class:

```
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using Sitecore.Ecommerce.Data;
    using Sitecore.Ecommerce.Validators.Interception;

    namespace Sitecore.Marketing.SES.Extensions.AddOn
    {
        public class OrderLineMappingRule : Sitecore.Ecommerce.Data.OrderLineMappingRule
        {
            [Entity(FieldName = "ParentProductCode")]
            public virtual string ParentProductCode
            {
```

```
                    get
                    {
                        if (this.MappingObject is OrderLine)
                        {
                            var line = this.MappingObject as OrderLine;
                            return line.ParentProductCode;
                        }
                        return string.Empty;
                    }
                    [NotNullValue]
                    set
                    {
                        if (this.MappingObject is OrderLine)
                        {
                            var line = this.MappingObject as OrderLine;
                            line.ParentProductCode = value;
                        }
                    }
                }
            }
        }
```