Written Report

COMP62542 Pattern-Based Software Development Coursework Assignment

**Student Name:** Michael Richards

**Student ID:** f78749mr (10362882)

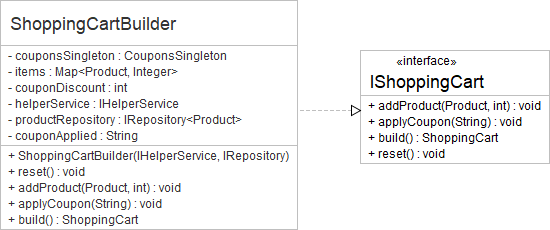
**Email Address:** michael.richards-2@postgrad.manchester.ac.uk

Word Count: 2614

# Description of Patterns Used

## Builder Pattern

The shopping cart builder class has been implemented using the builder pattern. It is used to construct a shopping cart object returned by the public *build* method and helps to separate the complex creation logic from other classes that rely on the shopping cart. It adds products to an internal list and then applies the discounts found on each product, as well as any additional coupon provided by the user. The returned shopping cart instance returned contains important properties that are later mapped to view models to display those results on the checkout page.



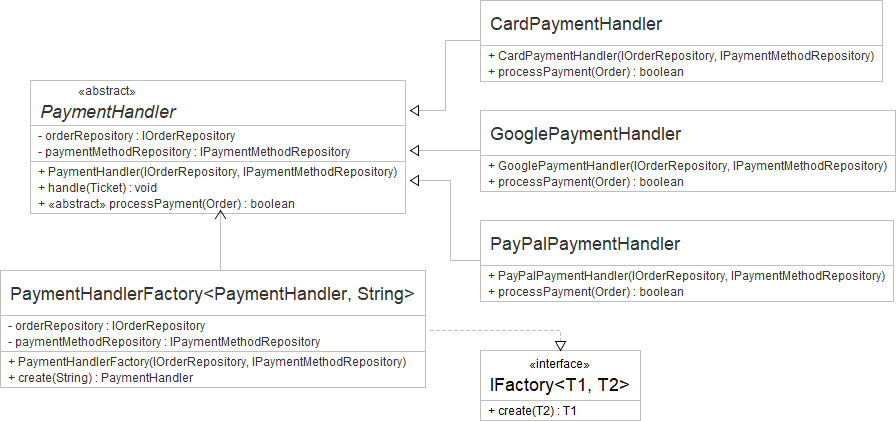
The concrete implementation of the builder includes a coupons singleton to check if the coupon received from the public *applyCoupon* method exists. If so, the coupon is assigned to the private *couponApplied* property. Similarly, products added using the public *appProduct* method are assigned to a private map object where the key is the product and value is the quantity of products. These private properties are then used during the build operation to construct the shopping cart complex object.

This build operation is used for multiple scenarios, however the logic to execute the builder and to update the database has been encapsulated in a unique command as discussed later. If this logic was duplicated, or the construction of the object varied across the application, then it would be hard to enforce the business logic involved with applying discounts. It would also make development tedious and harder to test.

The command class that uses the builder directly does not need to concern itself with the internal properties of the shopping cart object. Therefore, coupling is reduced because the command class does not need to depend on any encapsulated property within the builder and can interact with the public properties implemented by the interface as its only method of communication. This allows the builder to evolve or even be replaced for another builder implementing the same interface and the command will not be affected. This supports the programming to interfaces and designing for change principles, and favours object composition by relying on dependency injected repositories rather than relying on inheritance.

## Factory and Factory Method Patterns

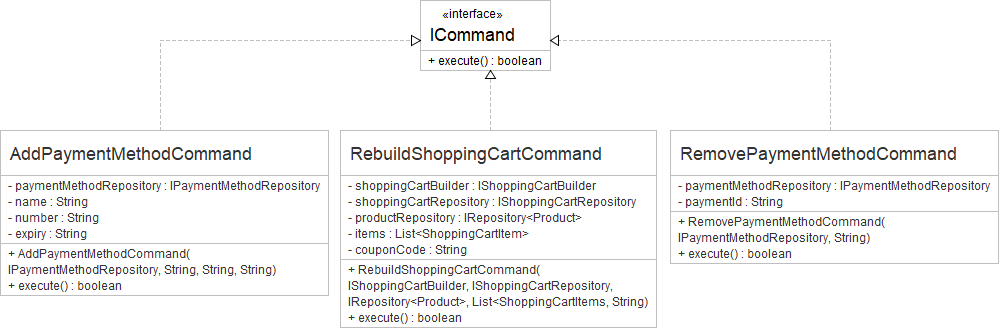
The payment handler factory implements a generic factory interface to allow other factories to implement the same pattern, if required, allowing the factory and client code to change independently. It uses a public *create* method to return a class that extends from an abstract payment handler class. The abstract class has an abstract *processPayment* factory/template method that is implemented differently for each of the three concrete payment handlers depending on which payment method the user has selected from the payment page.



The factory passes dependencies to the concrete subclasses when constructing them inside the *create* method. These dependencies are injected into the constructor of the factory, which improves cohesion and reduced coupling from client code that uses the factory because they do not know of the dependencies required. Any client using the factory also benefits from not needing to know the payment method involved, as the template method’s signature is identical for each payment handler and can be executed using the same pattern. The *create* method returns an instance of the abstract payment handler type so the payment process being used is unknown. This separates the responsibilities of both the client and the factory and thus increases cohesion and reduces coupling.

## Command Pattern

Common functionality that needed to be executed in isolation was grouped together into three different command classes; one for rebuilding the shopping cart (or creating one if no shopping cart previously existed), and two others for removing and adding payment methods. Each of these implements the *ICommand* interface, which consists of a single *execute* method that returns whether the command was successful or not. Although the application currently does not use this returned value, this has been added with the idea of handling failure in a user-friendly manner such as showing a failure message on the view. Then, the real error can be logged internally by the command.

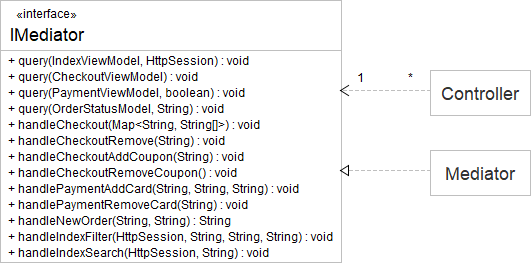


Using this pattern allows for commands to be easily swapped out for other commands because their interfaces are the same, and the client code only calls the same *execute* method and does not rely on more complex communication styles. This reduced coupling supports designing for change, but also allows for easily reusing code at any point in the system. You could also execute the command using asynchronously and handle the result later if it does not affect other client code.

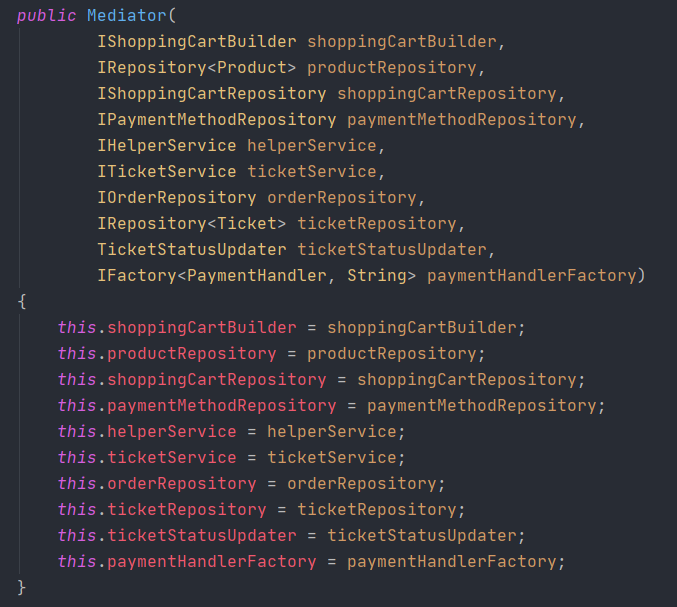
## Mediator Pattern

The mediator is used by many controllers to support the separation of concerns principle. The controllers are only concerned with handling requests, such as returning the user to the correct view, or redirecting them to another controller action. They do not communicate with any business/service layer component except for the mediator. The mediator handles two forms of communication from the controllers: queries and handlers (sometimes referred to as commands but to avoid confusion with the command pattern we will refer to them as handlers).

A controller calling a mediator query method will pass it the view model required to render the view. A query is only used to get data, such as handling a typical GET request, and does not change the state of the system. The mediator fills the view model after interacting with the service layer on behalf of the controller. A handler, on the other hand, handles requests aimed at changing the state of the system such as what is expected from a typical POST request. Handlers do not populate the view model and are not required to fetch anything so they usually return nothing except for the unique case of the *handleNewOrder* handler, which returns a ticket ID used as a redirect query string parameter by the controller.



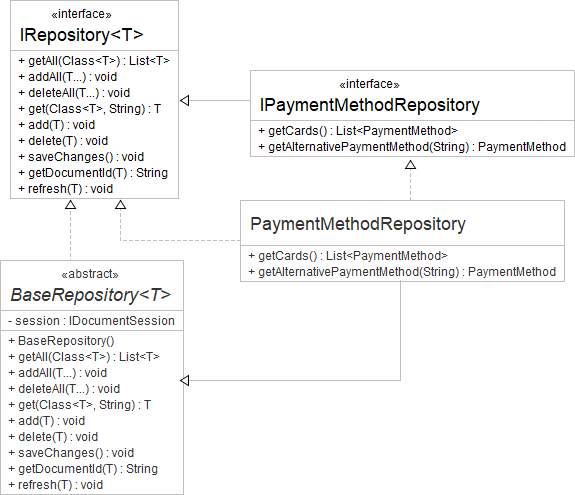
The mediator handles all dependencies between services, commands, repositories and other components such as factories and builders. Its responsibility is to orchestrate the complex communication between dependencies and returning the outcome of their execution. It does not contain any core business logic itself. This provides a clear separation between request/response logic and business logic. The mediator that implements the *IMediator* interface uses dependency injection where each dependency is an interface. This allows the concrete implementation of these dependencies to be changed without affecting the mediator’s communication flow of execution. The only negative with this design is that the mediator needs to only focus on communication logic, else is risks growing in complexity as the list of dependencies grow.



## Façade and Adapter Patterns

The façade pattern was used twice to implement the services and repositories in the system. Repositories had the single responsibility of persisting and fetching domain data from the database. Often, they would extend the behaviour beyond typical database operations, but the behaviour would be encapsulated within a single method which the client could call using an interface. Services worked similarly to repositories as they could, for example, create a domain object but then they might call a repository to store it. Services were used to provide helpful, reusable functionality in a less structured manner as opposed to commands which typically represented a single business use-case. Services helped to encapsulate this reusable logic inside a single method. These methods are more flexible then commands as they can accept varying arguments and return any value as described by their interface.

The repository pattern is a sub-pattern of the façade pattern and results the same outcome except they focus on database commands relating to individual domain object types in a generic fashion.

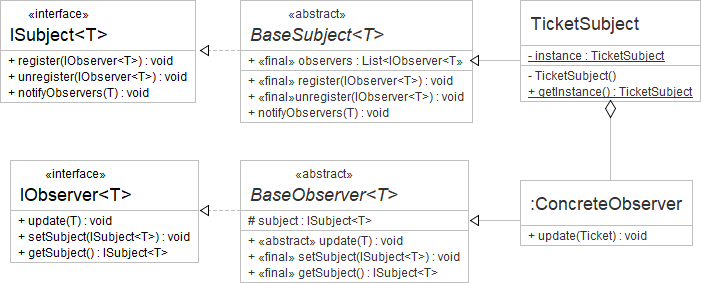


A repository concrete class implements the *IRepository* interface and extends the abstract *BaseRepository* class. It also provides a domain object as its generic type, such as a Product. This allows the concrete repository to make use of the default implementation of the *IRepository* interface provided by its abstract base/parent class, whilst also allowing it to extend this functionality by providing more specific repository functions relating to its generic domain type through the use of a more specialised interface. If a client class requires the use of a repository, it can choose either the *IRepository* interface or its more specific interface type, such as the *IPaymentMethodRepository*, if additional functionality is required. Because the more specific interface extends *IRepository*, the client can use methods contained in both interfaces regardless of whether the concrete repository or base repository implemented them.

This strategy helps provide flexible, reusable functionality without the client needing to know the complex, internal implementation details. The BaseRepository abstract class also works as an adapter by separating the dependency of the underlining datastore, such as RavenDB, from the rest of the application by transferring application operations into datastore specific operations.

## Observer Pattern

In the checkout system, there exists four different concrete observers representing different Order object states: created, accepted, processing, and complete. These states represent how far into the journal the user’s order is. The TicketSubject extends the abstract BaseSubject by passing the Ticket domain object as its generic type. Similarly, any concrete observer can implement the abstract base observer class by providing its own generic type, and this dictates what subject it can observe. For example, only observers whose generic type is a Ticket can observe a Ticket observer.



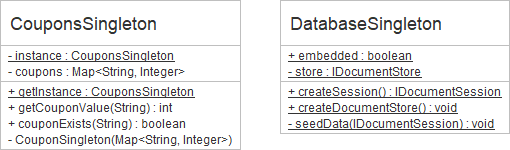
The abstract update function must be implemented by the concrete observers to control what happens when they are notified with a state. The type of the state object must match the generic type of the observer and subject respectfully. The subject is simplistic in design because it does not care about the state; it only forwards the state to the observers who can either respond or ignore the change. The subject has no specialised control over what observer to call and will call all observers when a client passes the new state to the *notifyObservers* method. This implementation allows for the observers to have full control over the response and remains completely isolated away from the client class that passed the state to the subject.

Observers can register or unregister independently from receiving updates from the subject, allowing for high flexibility and adaptability in the face of change. Additionally, with the added support of generic interfaces, additional subjects and observers can be formed in future with minimal effort.

## Singleton Pattern

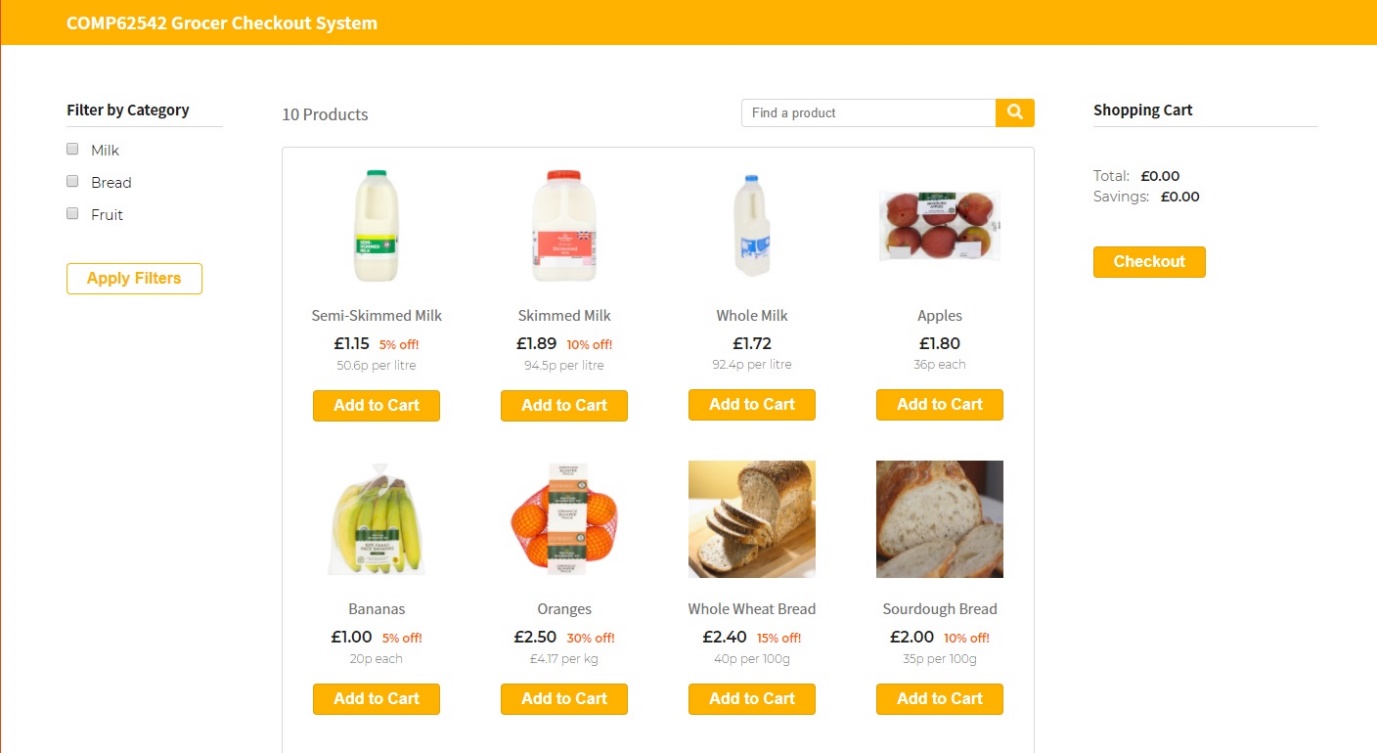
Two singletons are used in the checkout system: the coupons and database singleton. The coupons singleton returns an instance of itself, whereas the database singleton returns document sessions to begin a session/connection with RavenDB. This is used primarily by the BaseRepository abstract class within its constructor. It also handles loading the database only once, using the *createDocumentStore* method, when a repository requests its first session. This method also handles seeding data if the database has no data. This is useful for testing an application and demonstration purposes, such as loading many products so the product browsing page does not look empty.

The coupons singleton returns a fixed map of available coupons. These coupons should ideally be stored in the database, but the purpose of this singleton is to reduce the amount of database calls as much as possible. Singletons are often a good choice when an application requires access to the same data very frequently with infrequent updates to the data. For example, if an application’s pricing details for products rarely need updating but almost every web request requires access to that data, then instead of calling the database unnecessarily you can use a singleton to hold that data in memory. Then, if required, you could refresh the data with some other mechanism such as an API call that triggers a refresh method on the singleton.



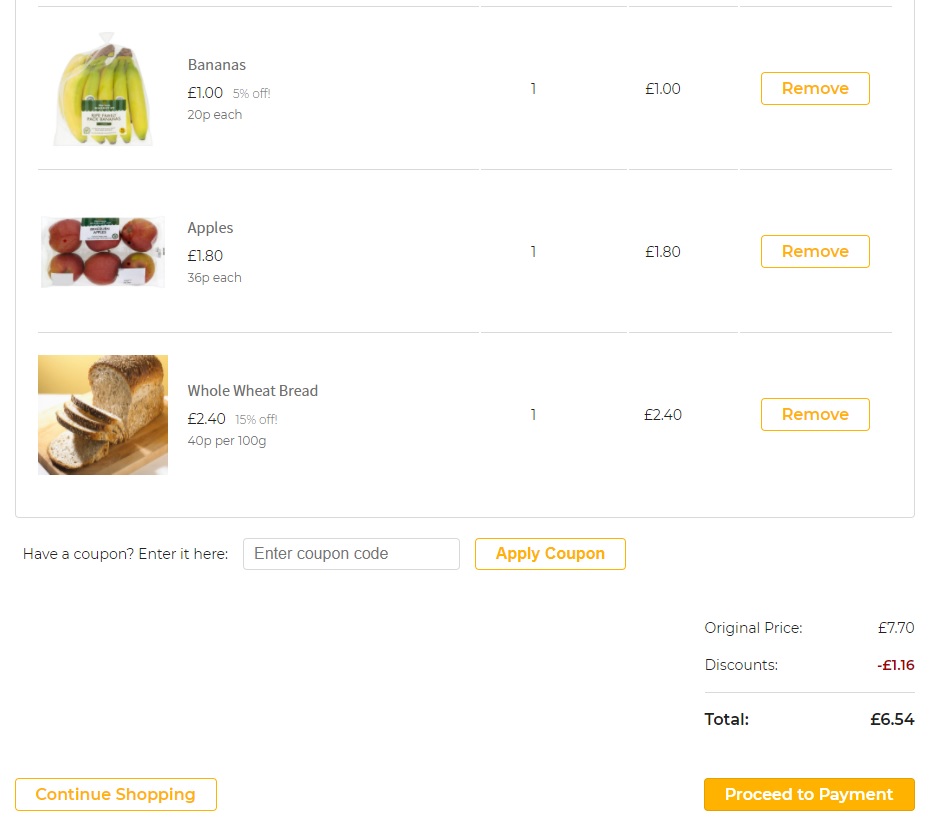
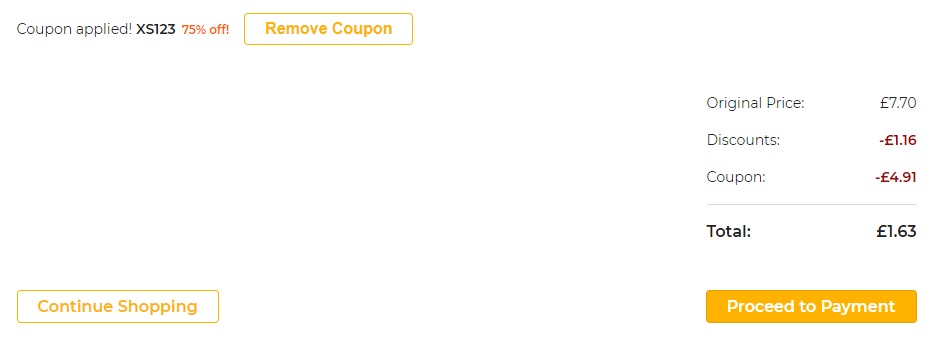
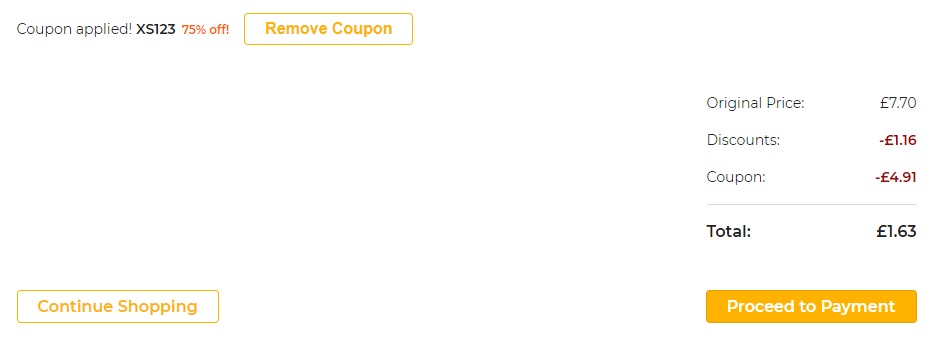
The Singleton pattern helps to simplify the creation of complex objects such as in the case of the database singleton which must start and configure RavenDB. This singleton also provides the benefit of hiding references to RavenDB and hence avoids unnecessary coupling of dependencies. One disadvantage is that static classes typically do not support interfaces, making them difficult to work with for dependency injection. However, they could perhaps provide a generic method for getting the singleton where that generic type is an interface. They you would have a mixture of the singleton and factory pattern which would provide further reduced coupling.

# Implementation Techniques

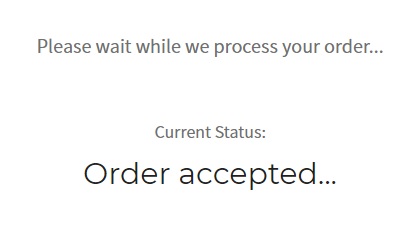
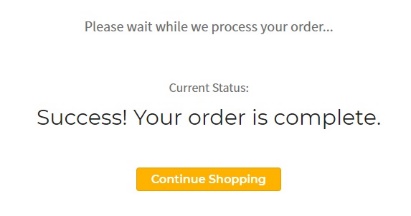
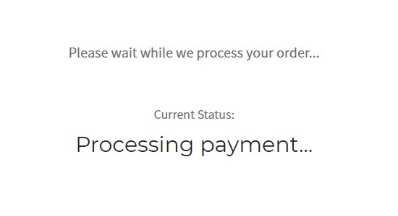
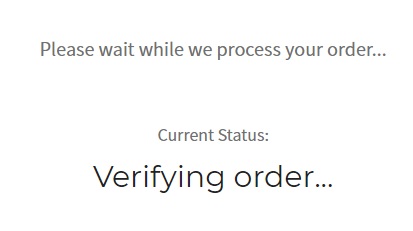
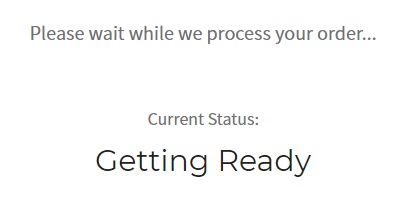
The homepage relies on the mediator to handle rendering different states of the view depending on what view model was passed to it. Using the query and handling naming convention for mediator method names, it was easy to hook a controller action with the correct mediator method and trust it to deal with the orchestration of business components. These components would then handle the business logic and would leave the mediator to focus only on orchestration concerns.

Each pattern and type of object has its own unique responsibility, allowing for clear separation of concerns through fixed responsibility boundaries. The command pattern helped to enforce this further by using an explicit interface which restricted the command pattern from returning anything more than a Boolean value.

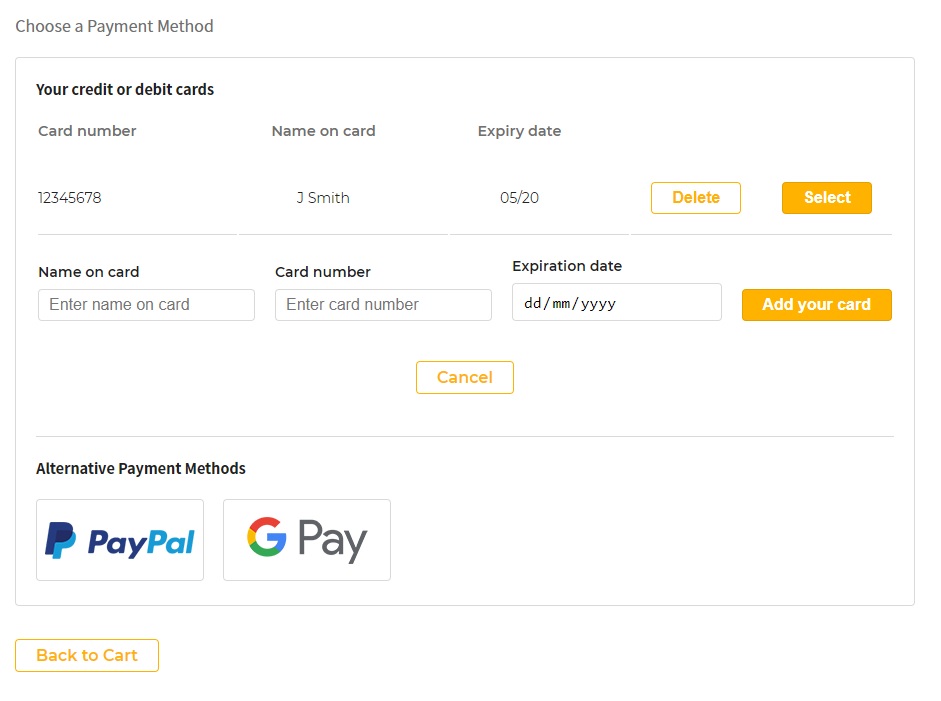
The shopping cart builder is a core pattern used in this application. It handles the core business logic of how discounts are calculated. The results of these calculations are then bundled together in a shopping cart object returned to the command that initiated the build operation, allowing client code to ignore all these details. This builder logic could then be easily tested and reused multiple times.



The final core feature is the use of the Observer pattern to progress the user through the payment and ordering journey. A simple JavaScript periodically calls a GET endpoint to get the current state of their order’s ticket. Each order is assigned a ticket, which contains the ticket’s state used by the observers. Once the payment and ordering process is complete, the ticket is assigned a completed status and the order is marked as paid for. The user is then greeted with a finished message and a button appears to allow them to return to the homepage.



The observers worked very well for monitoring state, but if the application was going to be used for real business purposes, you would benefit from allowing them to execute on a separate thread or process, because the observers currently only check the status when the user makes a request, which is not desirable. However, the application only caters for one user and thus is fit for purpose.



A final note is the use of the payment handler factory used to implement isolated payment handling logic independent of what type of payment method the user selected. This helped keep the logical flow very systematised, which also makes the entire system adaptable to change. For example, if another payment method was needed in the future, you only need to create the payment handler class that implements the required interface enforced by the factory, add another switch-case branch and then nothing else will need to change as the system can execute its handler without needing to know what it does. Besides using interfaces, allowing specific components to own their own responsibility within the system helped to produce an intuitive flow of logic that is highly testable and easier to debug. If logic failed at any point during the journey, it was easy to know where to look. For example, the mediator should never persist any object and a controller should never have access to any components besides the mediator and should not have any domain objects. The patterns used help enforce that level of isolation and facilitated the development experience.

# Execution Instructions