Software design's practical use is to facilitate easy modification of the implementation by dictating how source code is organised, specifically for human consumption.

Therefore, developers unfamiliar with an application should be able to quickly appreciate the design through examination of the source code, which is the only definitive source of understanding of the behaviour of the system. To fulfil this goal in the domain of web development, an application should ideally be organised according to well established standards, and use libraries that favour common design conventions.

The Ruby on Rails library was used to implement this application because it clearly articulates the model-view-controller pattern. This has the advantage that the basic layout is common among Ruby on Rails projects, and so Ruby on Rails developers have a head start when approaching a project for the first time.

Another area in which design is useful is the modelling and appreciation of properties of the data being handled. The data model was kept simple, to allow addition of a JSON REST API at a later date, using the Ruby gem Rabl.

There are two main models: Order and Stock. The trickiest modelling challenge came when trying to model Stock. Stock is difficult because there may be multiple instances of it added at different times to create a total. The original spec called for multiple locations to be handled, but this was left out of the implementation to simplify the task at hand. Locations could easily be added, since multiple instances of stock are tied to a single product. Products are shared between Order Lines and Stock, reducing data duplication and reducing the risk of data inconsistency. This makes the model easy to change. For example, say a product description need changing. Rather than changing this on a field on every instance of stock, the product model is updated, and when a request for a stock's description comes in, this is delegated to the product.

In the case of an e-commerce web application, this data design is tightly coupled to the business model being articulated. In the case of an inventory system for a business, the data model is focused around maintaining a log of the orders made. Stock can be similarly modeled: as a log of incoming stock additions that combine to give the total stock level. The only other business concerns, such as making sure an item is only available when in stock, can be enforced as validations made on data about to enter this model. For example, whenever a new order line is about to be created, the stock levels are checked to make sure the item in question can be added to the order.

The business model is based around the user journey of creating and then fulfilling the order. It is articulated in the validation on the various models involved. Validation is important since this is the process by which mistakes and malicious vandalism are prevented, and the means by which a business model is kept useful and consistent. For example, it makes no sense for an order's status to change from 'confirmed' to 'pending', e.g to go backwards through the order process. This is prevented at both the presentation (view) and the model layer. Even if an attacker were to submit their own custom post requests to the HTML API, the model will reject the request as invalid.

In Mitch Kapor's terms of commodity, delight and firmness, the main priority of this software design is commodity. Does the design produce an application that is useful as an inventory system?

The basic operation of an inventory is present. Functionality is focused around the user journey of taking an order from the customer, adding products to that order, and then presenting those orders to warehouse staff to dispatch the instrument.

The design of this application went through two iterations. At first, it was to be based around providing a JSON REST API, along with a statically served single-page web application that consumed the JSON REST API.

While implementing the backend for this REST API proved trivial, implementing the frontend was not possible in the time available. As a result, it was decided that a simpler design would be followed, involving a backend which served dynamic HTML, combining the data with static HTML templates according to the underlying business and data model.

The implementation produced is perhaps best described as the first or second iteration in the agile development process. During this process, the minimum viable product is described, and subsequently implemented as quickly and as thoroughly as possible. In these terms, the implementation is complete: orders can be created and archived, new product lines can be added, and stock levels checked. However, subsequent iterations were not possible due to time constraints.