To determine what approach will be required to design and develop the solution, we needed to comprehend what the problem is. To facilitate an improved understanding, the problem was divide into smaller segments and grouped into logical modules. A high level strategic design is compiled to determine how these components would interact with one another. The initial components of the high level design are as follows, Billing Companies, Customer, Web Scraper, Scrape Scheduler and the Publish-Subscribe Messaging Pattern. An example listed below describes the thought process in order to determine how segments materialised into a logical module.

For example, a sample of questions listed below are used in order to define what forms part the logical module as well as what are the responsibilities. The Customer is used to demonstrate the example.

* What are the responsibilities of the Customer? Add a billing account, remove a billing account, update their details, etc.?
* Which area of the system will be impacted if the customer changes their information?
* How a customer would be notified if a statement has arrived or the system failed to retrieve it?

By constantly asking the type of question listed above the relationships between the modules began to reveal themselves. These disclosures provided insight into the vary complexities that existed in each component.

*A Messaging Pattern is a network-oriented*[*architectural pattern*](http://en.wikipedia.org/wiki/Architectural_pattern)*which describes how two different parts of a*[*message passing*](http://en.wikipedia.org/wiki/Message_passing) *system connect and communicate with each other* [4].

By leveraging of the Messaging Pattern, it endorsed the design to be loosely coupled and prompts asynchronous communication within the system. It also allows for each part of the application to run independently. For example, if the scrape scheduler stopped it should not bring the system to a halt.

Based on the strategic design, it was concluded that Domain-Drive Design (DDD) would be an appropriate technique to model the design of the system, based on the varying complexities of each component, as well as it emphasises will be on the core domain and its logic. Another technique known as Test-Driven Development (TDD) with the aid of Specific by Example, will be used in conjunction with DDD to develop of the software. DDD is used to guide design of the system for the top down, and TDD would directed the design through the development life cycle (bottom up). These techniques work well together because DDD systems make it easier to use TDD as it allows for easily navigating the system through unit tests.

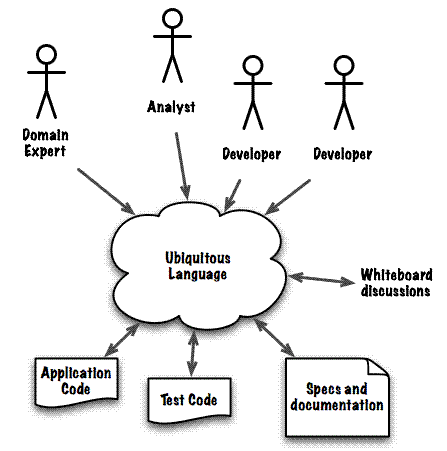
Using DDD allow the design of the system not to depend on the infrastructure (for example, a database). This is possible because the model code is separated from the infrastructure. This makes DDD persistence ignorant, therefore allowing the design to focus on solving the business problem rather than how the data would be structured. These logical modules are as known as domain models.

In order to successful execute DDD there needs to be a common language between the domain experts and the team that will be development the product.

*Since the model is the common ground, the place where the software meets the domain, it is appropriate to use it as the building ground for this language* [3].

This language is called the Ubiquitous Language. The requirement for the Ubiquitous Language is so that all team member are able to communicate clearly and accurately (Appendix B outline the Ubiquitous Language used for the project). It helps to create a shared understanding so the system. This help when a work breakdown structure needs to be created, as the language will allow developer work on different areas of project without much effort.

Figure 1 demonstrates the key role of Ubiquitous Language in DDD [1].



*Figure 1The Ubiquitous Language should be the only language used to express a model [1]*

Discuss how using the responsibility breakdown simplified the process of work allocation and implementation, which allows for developer swap out if it was needed.

Reference

[1] Alberto Brandolini,” Strategic Domain Driven Design with Context Mapping,” <http://www.infoq.com/articles/ddd-contextmapping>, November 2009 [2014-06-10]

[3] Abel Avram and Floyd Marinescu, “Domain-Driven Design Quickly”, InfoQ

[4] Messaging Pattern <http://en.wikipedia.org/wiki/Messaging_pattern> [2014-06-11]

[5] Soumen Chatterjee, Messaging Patterns in Service-Oriented Architecture, Part 1 <http://msdn.microsoft.com/en-us/library/aa480027.aspx>, April 2004,[2014-06-11]

[6] Event-Driven Messaging <http://soapatterns.org/design_patterns/event_driven_messaging> [2014-06-11]