Week 5 Final Assignment: Architecture Design Document: ABC Online Store

Alicia Piavis

CST 307: Software Architecture and Design

Amr Elchouemi

01/20/2020

Table of Contents

Architecture Design Document: ABC Online Store

Overview

This design document outlines the architecture and design for the new ABC online store (e-commerce system). The team for the project will require one applications architect, one database administrator, three software engineers, and one product manager. The system will involve two architectural patterns—layered and MVC—and a number of design patterns including prototype, singleton, observer, iterator, state, session, transaction, and memento. Business and application views have been included in the form of a use case diagram, activity diagram, sequence diagram, deployment diagram, and class diagram. The object-oriented design for the system is outlined in the document as well.

Context

This project aims to fulfill a request from the CEO of the ABC store Corporation to build an e-commerce site that will draw business back to ABC stores from other e-commerce competitors, such as eBay and Amazon. ABC store sales and growth have dropped significantly since the early 2000's when other e-commerce sites emerged and allowed customers to purchase items 24/7 from the comfort of their homes. By allowing ABC customers the opportunity to purchase ABC store items online, and then choose if they would like instore pickup or delivery, the CEO hopes that business will be restored to the growth rates ABC stores were experiencing in the 1980's. The e-commerce site will also allow ABC stores to expand business by allowing third party vendors to sell similar items, for which ABC stores will collect a percentage of the profits. This project will require a team of architects, database administrators, product managers and engineers.

Team Roles

Applications Architect

Key Responsibilities

- Keep up with changing technical landscape
- Work closely with product manager and stakeholders
- Contribute to product architecture, design patterns, tools/framework selection and non-functional requirements
- Create innovative solutions to solve business challenges and enable rapid product development
- Work closely with onsite and offshore teams
- Design and develop large scale customer facing enterprise e-commerce application
- Be a mentor to the development team
- Develop and deploy microservices in cloud environment
- Communicate with customers and C-level executives

  * Responsibilities are modified from various job postings on Indeed.com (n.d.)

Database Administrator

Key Responsibilities

- Design, implement and administer database and instance access and security

- Perform day-to-day database administration activities to ensure integrity, security, performance, and availability of production and development environments
- Make recommendations to improve database and application design standards
- Work closely with other members of the development team to support project design, implementation and deliverables
- Create scheduled processes (SQL Agent Jobs, SSIS packages, PowerShell scripts, etc.) to automate routine functions
- Build SQL Server Integration Services packages and SQL Server Reporting Services reports
- Resolve database issues with support from the software engineers and applications architect

  * Responsibilities are modified from various job postings on Indeed.com (n.d.)

Software Engineer

Key Responsibilities

- Develop and maintain a new e-commerce application for ABC Stores
- Perform unit testing throughout the software development lifecycle
- Communicate with other members of the project team to ensure that software requirements are met
- Participate in daily scrum meetings and other scrum ceremonies
- Work with the applications architect and product manager to plan new features
- Develop a broad understanding the code base
- Maintain version control with GitHub

  * Responsibilities are modified from various job postings on Indeed.com (n.d.)

Product Manager

Key Responsibilities

- Create a product roadmap that addresses the project deadline, and accounts for unexpected slowdowns in the development process
- Support and review architecture, design, and development to ensure that business objectives and project goals are being met
- Assist with requirements gathering and refinement
- Represent the voice of the customer
- Identify new opportunities and possible future features
- Assure business commitments can be made and kept by the product team
- Communicate with stakeholders on status, trade-offs, issues and dependencies
- Rapid and appropriate escalation of critical issues
- Facilitate scrum ceremonies

  * Responsibilities are modified from various job postings on Indeed.com (n.d.)

How the These Roles Will Work Together to Complete the Project

This project team will involve:

- one applications architect
- one database administrator
- three software engineers
- one product manager

These roles will support one another throughout the lifecycle of the project. The process will require close communication in order to refine project requirements, architect the system for the ABC store e-commerce and inventory application, design the system, develop and test the software, and fulfill functional and non-functional requirements. By working collaboratively, involving stakeholders throughout the process, employing agile methods, and utilizing the scrum framework, the team will be able to achieve the ultimate project goal, customer satisfaction.

Requirements

Goals

- Create an online presence for the ABC stores
- Maintain the physical stores for ABC and leverage these stores as an advantage versus pure online stores like Amazon
- Link ABC stores and the online business to the key vendors in a way to reduce the inventory cost and pass that savings to ABC customers
- Link the online business to major carriers like Fedex, UPS, and DHL to minimize the cost of shipping to customers
- Create a portal within the ABC online stores where customers can also sell similar products with a small percentage that will go to ABC business
- Ensure that the online ABC business is available 24/7 without any interruption.

Functional Requirements

- Customers should be able to search for ABC store items through an online e-commerce website
- Customers should be able to purchase items from ABC stores online
- Customers should be able to choose whether they want to pick up the item in the store, or have it shipped to them
- Customers and store managers should be able to view the inventory for a specific item across different ABC stores
- Customers should be able to select Fedex, UPS, or DHL as a shipping option when they check out
- Vendors should be able to receive an automated notification when inventory for an item they provide is low at an ABC store
- Vendors/customers should be able to sell similar products on the ABC website, with a percentage going to the ABC stores

Non-Functional Requirements

- The ABC online store should be available 24/7 without any interruption
- A customer should be able to create an account within 1 minute
- A customer should be able to find and purchase an item online within 2 minutes
- Customer information should be kept secure (passwords for accounts should be encrypted)
- The website should be available on all major web browsers (Chrome, Firefox, Safari, Internet Explorer & Edge)
- The website should be accessible and adhere to ADA best practices

- The website should be culturally sensitive, so that the user base can grow outside of the US

Software Qualities

External Qualities

Availability

- The ABC online store should be available 24/7 without any interruption

Impact

This is one of the project goals that was presented by the CEO. Any downtime in the system will directly affect revenue generated from the online stores. Downtime will also affect the user experience and possibly deter users from returning to the site in the future. Any scheduled maintenance should avoid peak hours.

Usability

- A customer should be able to create an account within 1 minute
- A customer should be able to find and purchase an item online within 2 minutes
- The website should be accessible and adhere to ADA best practices

Impact

Usability affects whether or not users will return to the site. If the online store usability is not comparable to that of e-commerce competitors (eBay, Amazon, etc.), users may opt to use competitor e-commerce sites instead.

Portability

- The website should be culturally sensitive, so that the user base can grow outside of the US
- The website should be available on all major web browsers (Chrome, Firefox, Safari, Internet Explorer & Edge)

Impact

While the current state of ABC stores is that they have only a domestic presence, it is possible that the e-commerce site will allow for growth into international markets. In this case, it is important that the site is culturally sensitive. Furthermore, if the site is not accessible from all major web browsers, this will impact accessibility and the size of the user base.

Internal Qualities

Maintainability

- The system should have low coupling, high cohesion, and low lines of code (LOC)
- The system should be flexible, extensible, and modifiable
- The system should support preventative, adaptive, perfective, and corrective maintenance

Impact

One of the goals of the new system is growth of the ABC business. With growth comes the need to modify and extend a system. Designing the ABC online store for maintainability will make the growth of the business and the e-commerce site more efficient and achievable. On the other hand, a lack of maintainability may impede growth and scalability.

Interoperability

- The system should employ common standards (ex. JSON and HTTPS)
- The system should implement interfaces

Impact

The implementation of common standards and interfaces increases the interoperability of a system. If interoperability is not considered during implementation, different parts of the system may interpret specifications differently, it could be difficult to integrate a new system in the future, integration tests may prove difficult, integration issues may exist during a new release of the software, and integrations may be difficult to maintain as new versions of different parts of the system are released (Ingeo, 2018).

Testability

- The system should support unit testing
- Tests should be self-documenting
- Tests should have high controllability, high observability, high isolatability, high automatability, and low complexity

Impact

Well-architected systems are testable. The ability to test the system throughout the implementation phase will allow developers to catch and correct errors in the code more quickly. This will reduce development time and costs, and lead to more reliable code.

Architecture Descriptions

Stakeholders

- Users & Business Managers
- Database Designers & Administrators
- System & Software Engineers
- Acquirers, Operators, System Administrators, & System Managers

Business Architectural Views

Use Case Diagram

Figure 1.1 ABC Online Store Use Case Diagram

Click here to view the interactive UML Use Case Diagram.

Activity Diagram

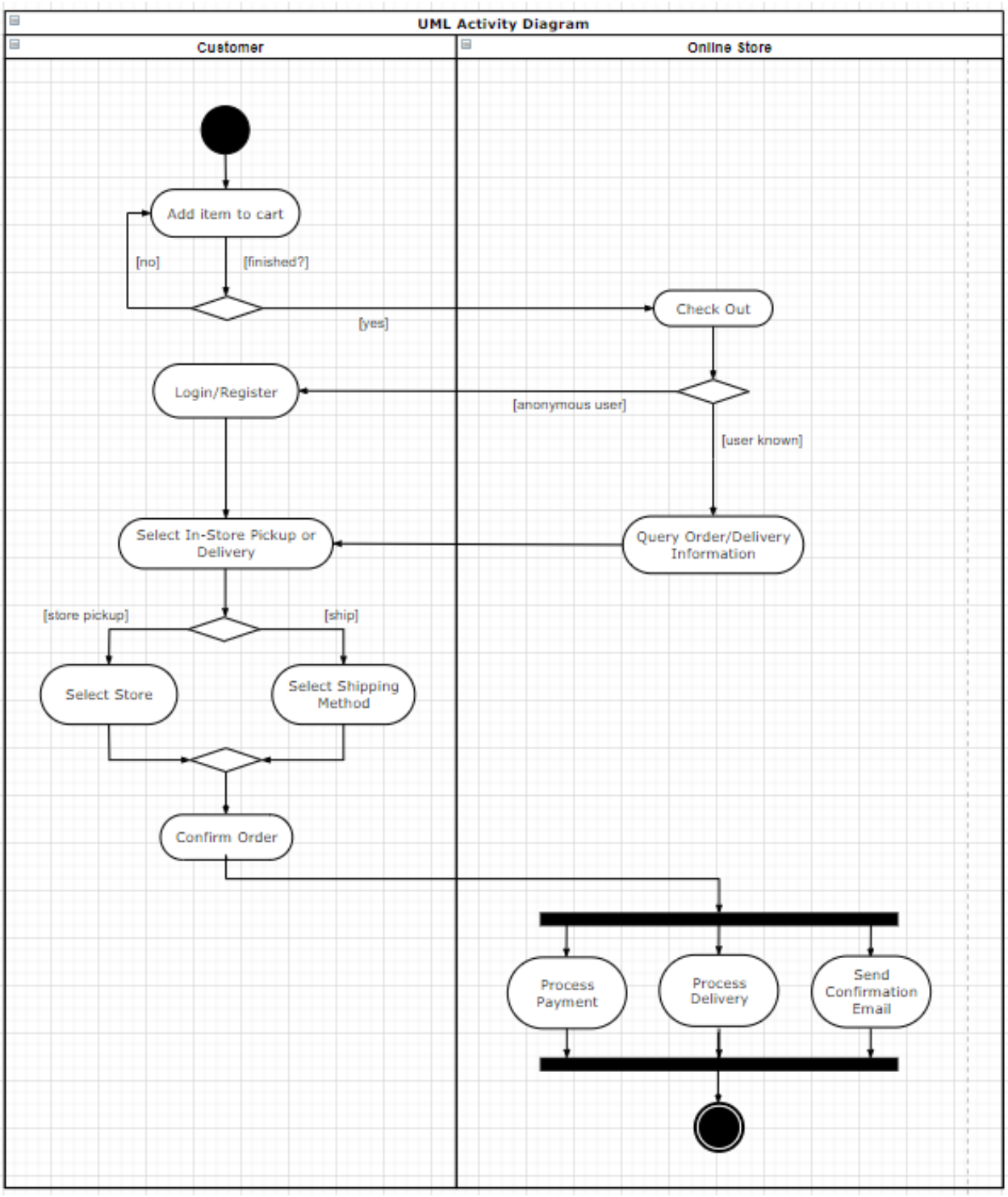

Figure 1.2 ABC Online Store Activity Diagram

Click here to view the interactive UML Activity Diagram.

Application Architectural Views

Sequence Diagram

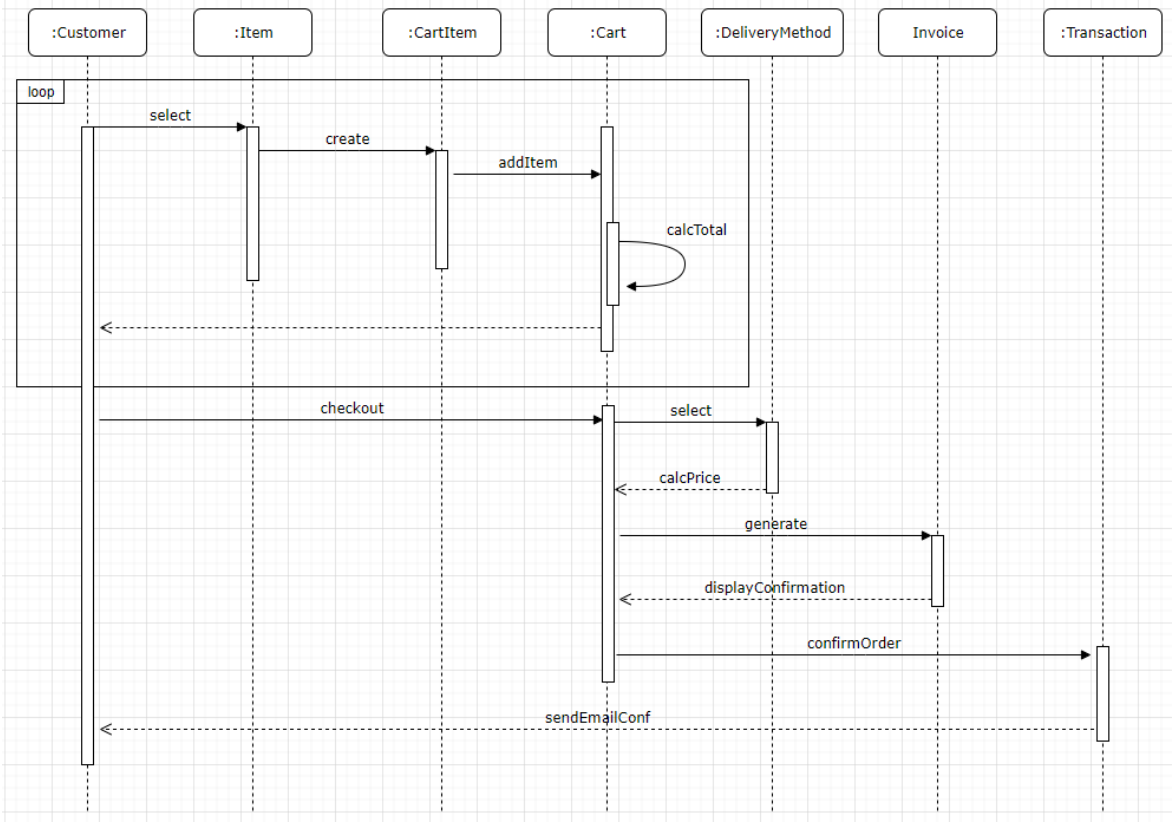

Figure 1.3 ABC Online Store Sequence Diagram

Click here to view the interactive UML Sequence Diagram.

Deployment Diagram

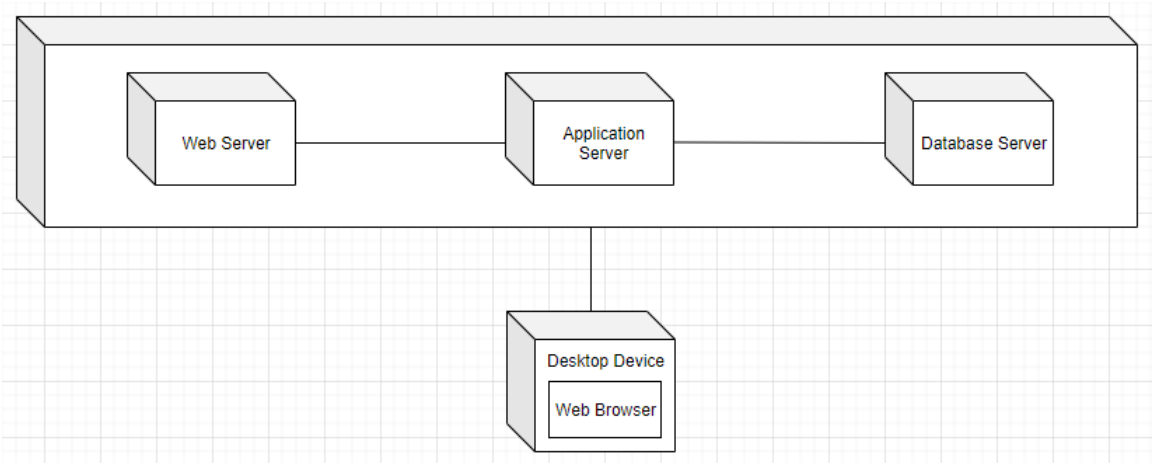

Figure 1.4 ABC Online Store Deployment Diagram

Click here to view the interactive UML Deployment Diagram.

Class Diagram

Refer to Object-Oriented Design section below.

Object-Oriented Design

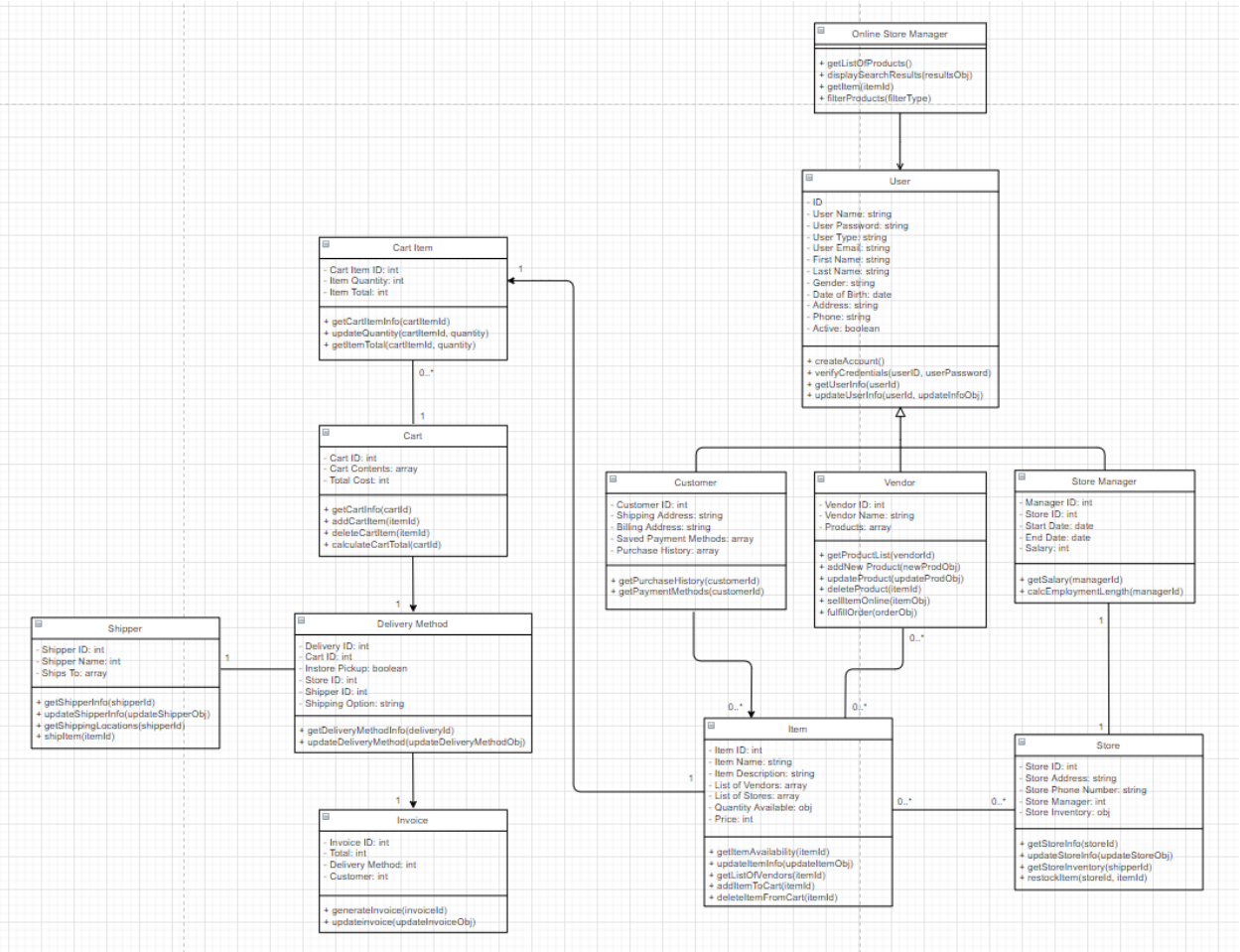

Figure 1.5 ABC Online Store Class Diagram

Click here to view the interactive UML Class Diagram.

Use of Design Best Practices

Encapsulation

Encapsulation involves separating features and methods in a class that are likely to change from those that will not (Dooley, 2017). This design principle prevents stable parts of the design from being affected by change. In Figure 1.5, the attributes related to an instance of a User will not change. For example, users of the ABC Online Store will always have an ID, user name, password, type, email, first name, last name, etc. However, over time the ABC Store Corporation may add new types of users that encompass new attributes. For example, Customer is a subclass of User that has unique attributes (i.e. billing address). Vendor is another subclass of User with unique attributes (i.e. products). ABC Stores may want to eventually add, or extend, new subclasses from User such as Administrator, Regional

Manager, etc. Because future subclasses may change, they are separated from the User class, which will not change over time. This is an example of encapsulation.

Open-Closed Principle (OCP)

The open-closed principle (OCP) can also be observed in Figure 1.5. This principle is very closely related to encapsulation. *Extending the "Open-Closed Principle" to Automated Algorithm Configuration* (2019) states that in the OCP, "frameworks should be 'open' in the sense of being extensible via the addition of new components, but 'closed' in that they do not require framework modification to achieve this." This means that attributes and methods that will not vary should be abstracted up into a super class. Therefore, when new features are requested, they can simply be added to the code, rather than requiring the modification of existing code. Again, the abstraction of the commonalities that the Customer, Vendor, and Store Manager classes share into the User class is an example of OCP. The User class is closed to modification, but open to extension. If new subclasses of User need to be added (Administrator, Regional Manager, etc.), this can easily be achieved without modifying existing code.

Don't Repeat Yourself (DRY)

Don't Repeat Yourself (DRY) attempts to avoid duplicate code. Whenever the same behavior can be observed in two or more places across a system, the behavior should be abstracted into a class (Dooley, 2017). This way, the behavior can be inherited and reused by instances of that class. In Figure 1.5, the User class is an abstraction of the Customer, Vendor, and Store Manager classes. Because the Customer, Vendor, and Store Manager classes share similar behaviors within the ABC Online Store (i.e. createAccount() and verifyCredentials()), these behaviors have been abstracted out into a separate class called User. This prevents the createAccount() and verifyCredentials() methods from being repeated in all three of the Customer, Vendor, and Store Manager classes.

Single-Responsibility Principle (SRP)

The single responsibility principle (SRP) is also utilized in the design of the ABC online store. As Dooley (2017) states, "A cohesive class does one thing well, and doesn't try to do anything else." In other words, a class should have a single functionality that it executes well, and each class should only have one reason to change. Classes that follow the SRP are marked by high levels of cohesion. According to *Refactoring - Improving Coupling and Cohesion of Existing Code* (2004), cohesion is "the degree to which elements of a class belong together" (Du Bois, Demeyer, & Verelst, 2004). An example of cohesion and the SRP can be seen in the CartItem class in Figure 1.5. This class has a single responsibility, to track a particular type of item in a customer's cart and the desired quantity of that item. This class allows a user to update the quantity of an individual item in the cart before they checkout. The CartItem class does not need all of the data pertaining to a particular item (such as item name, description, vendors, etc.), since that information is abstracted within the Item class. Therefore, the CartItem class encompasses a single responsibility, and everything within the class is cohesive, meaning it belongs together.

Liskov Substitution Principle (LSP)

The Liskov substitution principle (LSP) means that, "When you derive one object from another, the base-level semantics should not change. If you find yourself writing branching logic so that your function does one thing if provided with a base class, but something else for a derived class, you have violated this principle." (Spencer and Richards, 2015). To break this down, inheritance in object-oriented design should be clear and purposeful. There should never be an instance of a class that does not use an attribute or behavior of the superclass that it inherits from. Let's suppose that in Figure 1.5, billing address, vendor name, and store ID were listed as attributes under the User class instead of in the Customer, Vendor, and Store Manager classes respectively. This would mean that the vendor name attribute in User would be inherited but unused when instantiating a Customer object. Similarly, the

billing address would be inherited but unused when instantiating a Store Manager object. Likewise, the store ID would be inherited but unused when instantiating a Vendor object. This would violate the LSP. Instead, billing address is placed in the Customer class because it is unique to Customer, vendor name is placed in the Vendor class because it is unique to Vendor, and store ID is placed in the Store Manager class because it is unique to Store Manager. By doing this, we make all of the attributes inherited from the User class applicable to the subclasses.

Dependency Inversion Principle (DIP)

The basis for the dependency inversion principle (DIP) is that, "dependency modules are inverted for the purpose of rendering high-level modules independent of the low-level module implementation details" (Haoyu & Haili, 2012). Typically, in a top-down design approach, high level modules depend on the implementation of code at lower levels. The DIP is the inverse of that concept. Dooley (2017) goes on to explain the concept further, "The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details." In Figure 1.5, the User class exhibits the DIP, because it can still function without the lower level modules such as Customer, Vendor, and Store Manager. In other words, if the Customer, Vendor, and Store Manager classes did not exist, a generic User could still be instantiated using the attributes and behaviors within the User class.

Principle of Least Knowledge (PLK)

The principle of least knowledge (PLK), also known as the Law of Demeter, means that objects should be loosely coupled. *An Empirical Study on the Developers' Perception of Software Coupling* (2013) describes coupling as, "the measure of the strength of association established by a connection from one module to another." This means that modules, or classes, should function independently of one another. This is true of the design for the ABC online store, as there are very few instances where a class possesses a method that relies on information from another class (the addCartItem(itemId) method in the Cart class, which takes the argument itemId). By implementing the single-responsibility principle and making classes more independent from one another, the PLK can be enforced.

Architecture Design Process

Attribute-Driven Design (ADD)

The ADD (attribute-driven design) method was selected for the design of the ABC online store because the success of the store depends on customer and client satisfaction. Often, functional requirements are more heavily prioritized in software development projects. However, non-functional requirements, such as quality attributes, are just as important to consider. This is because quality attributes can both directly and indirectly affect the user experience. *A Prescriptive Approach to Quality-Focused System Architecture* (2017) states, "A system is considered successful if it meets stakeholder needs." Because the success of the ABC online store is heavily dependent upon user satisfaction, the ADD method was deemed the most practical.

ADD is a decomposition process which focuses on quality attributes in order to design the software architecture (Gielen, n.d.). In comparing ADD to the other architectural design processes, Zimeo, Oliva, Baldi, & Caracciolo (2013) state, "The ADD method, to design complex software systems, suggests to identify the architectural drivers from the software requirements with the aim of selecting the proper architectural patterns to adopt as a reference guide for the whole design phase. This allows software architects to reuse the consolidated experience as an existing knowledge base to reduce the design effort." ADD is beneficial because it is a step-by-step, iterative process that allows trade-offs between attributes to be revealed early on in the development process. This is beneficial to users of the ABC

online store, as they will likely see increased usability, security, performance, and availability. Figure 1.6 shows the ADD process.

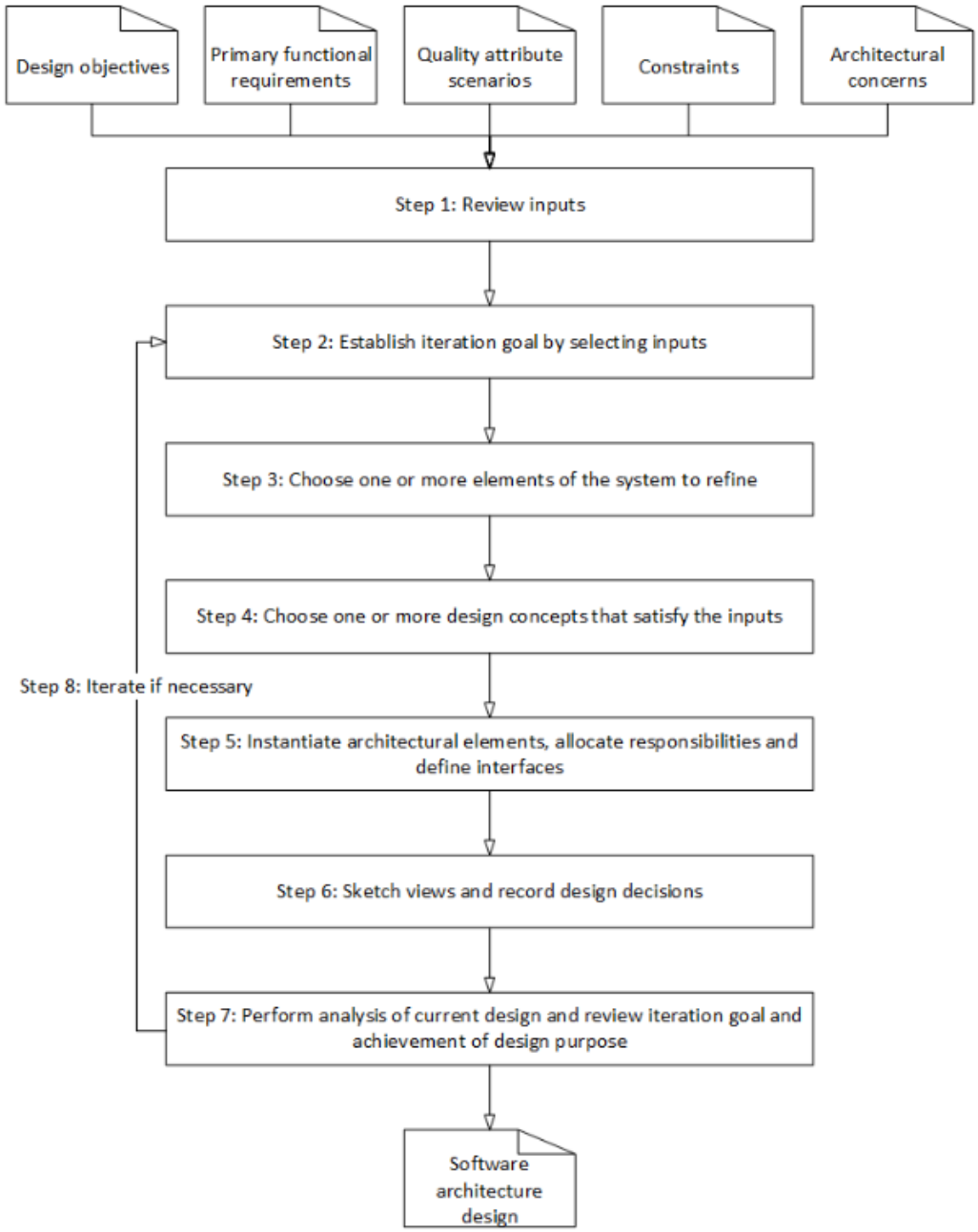

Figure 1.6 ADD Process (Ingeno, 2018)

Click here to view the interactive UML Sequence Diagram.

Architecture Patterns

The architecture patterns selected to implement the ABC online store are the layered pattern and the model-view-controller pattern. This decision was made after studying effective architecture patterns for e-commerce systems. The research resulted in the following:

> "A widespread architectural pattern that satisfies self-service interaction model is Model View Controller. This pattern, with its main variants (Model View Presenter and Presentation Abstraction Controller are the most known), suggests the organization of the presentation layer and its decoupling from the model of enterprise systems. However, additional patterns are needed to help the design of the other layers proposed by multi-layer and multi-tier decompositions. The layered organization allows for a clear separation of presentation, application logic and data management that multi-tiered architectures exploit in order to partition the different functional components onto dedicated resources, so ensuring important nonfunctional attributes, such as security, scalability, efficiency and data persistence" (Zimeo, Oliva, Baldi, & Caracciolo, 2013).

These findings show that MVC is commonly applied in the e-commerce space due to it's support of self-service interaction models (like the UI of the ABC online store). However, the findings show that use of the MVC pattern alone is not enough, since it only addresses the UI. A layered architecture pattern can be applied in this case to address the full needs of the system and compliment the MVC model selected for user interactions. Together these patterns will support security, scalability, efficiency, and data persistence, all of which are desired for the ABC online store.

Layered Architecture

Rationale

One of the architecture patterns selected for the ABC online ordering system is a layered architecture. According to Mark Richards (2015), "The most common architecture pattern is the layered architecture pattern, otherwise known as the n-tier architecture pattern." Layered architecture describes software that is organized into horizontal layers that are stacked on top of one another, which information can pass through. The three most common layers are presentation (which handles the user interface), business (which handles the business logic of the application), and data (which handles interactions with data sources, as well operations within tables). This pattern was chosen for the ABC online store because layered architectures are often implemented when it is beneficial to split a system into smaller components or layers that can be spread across servers, and when increased reliability is a priority (Packt Editorial Staff, 2018). In the case of the ABC online store, reliability is an important quality attribute, so a layered architecture is the most desirable.

Model-View-Controller

Rationale

Model-view-controller (MVC) was also selected as an architectural pattern for the ABC online store because this pattern is common in World Wide Web applications. The MVC pattern divides an application into three parts in an attempt to present information to the user differently than the way it gets represented internally. This make for a better user interface (UI) and user experience (UX): "MVC mostly relates to the UI / interaction layer of an application" (GeeksforGeeks, n.d.). The first part of MVC is Model. The Model holds the application data, but no logic as to how the data gets presented. The second part of MVC is View. The View presents data to the user. The Controller is the third part of

MVC, and serves as an intermediary between the Model and the View. The Controller listens for event triggers from the View in order to call the correct associated method on the Model. An event may be something as simple as a mouse click from the user. This pattern is beneficial in e-commerce applications, as is organizes the presentation layer and makes for an engaging UI.

Design Patterns

Prototype

When a user asks for more specific information about an item in the ABC online store, the application can "gather that information by instantiating the objects at predefined intervals and keep them in a cache, when an object is requested, it is retrieved from cache and cloned. When the legacy database is updated, discard the content of the cache and re-load with new object" (Kulkarni, Patil, Rane, & Meshram, 2012). This pattern is based on the ability to copy an object, rather than creating a new one from scratch, which saves time and leads to a better user experience.

Singleton

An example of a singleton in the ABC online store system is a database connection. A singleton pattern is used when a class will only be instantiated once. It is a simple pattern that allows a class to be defined. According to GeeksforGeeks (n.d.), "A singleton class shouldn't have multiple instances in any case and at any cost. Singleton classes are used for logging, driver objects, caching and thread pool, database connections."

Observer

In the ABC online store system, the observer design pattern can be used to notify a user as their items move throughout different stages of the shipping process (Kulkarni, Patil, Rane, & Meshram, 2012). The pattern can also be used to notify users of item availability. The observer pattern defines a dependency between objects so that dependents are notified when an object changes state (GeeksforGeeks, n.d.).

Iterator

The iterator pattern is used to iterate through objects. In the case of the ABC online store the iterator pattern can be used to traverse through the item list while a customer is searching for an item (Kulkarni, Patil, Rane, & Meshram, 2012).

State

This pattern is used when the behavior of an object changes based on its state. In the ABC online store, the state of an individual item may change from "available" (before a customer has added the item to their cart), to "in cart" (when the item has been added to the cart), to "purchased" (when the customer has confirmed the purchase and the transaction is complete. The behavior of the item may change throughout these state changes, because it wouldn't make sense for an item in the "available" state to have a behavior called returnItem(). However, returnItem() would make sense for an item in the "purchased" state.

Session

In the ABC online store system, this design pattern can be used to "maintain the state of the client as a session so that the validations or purchase can be associated with the session pattern" (Kulkarni, Patil, Rane, & Meshram, 2012). This will allow a customer to leave their virtual shopping cart during a session

to search for more items, but information in their cart will persist when they return to it. This way a customer can continuously leave and return to their cart in the same session until they are ready to check out.

Transaction

This pattern is a template for payment transactions and can be used in the ABC online store when a customer is ready to checkout with their shopping cart (Kulkarni, Patil, Rane, & Meshram, 2012).

Memento

This pattern uses checkpoints to restore the state of an object to a previous state. This is beneficial in the ABC online store when a user hits the Undo or Back button. In this case, the previous state of the object would be restored (Kulkarni, Patil, Rane, & Meshram, 2012).

Resources

Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., & De Lucia, A. (2013). An empirical study on the developers' perception of software coupling. 5th International Conference on Software Engineering (ICSE), Software Engineering (ICSE), 2013 35th International Conference On, 692–701. https://doi-org.proxy-library.ashford.edu/10.1109/ICSE.2013.6606615

Dooley, J. F. (2017). Software development, design, and coding: With patterns, debugging, unit testing, and refactoring (2nd ed.). Retrieved from https://www.vitalsource.com/

Du Bois, B., Demeyer, S., & Verelst, J. (2004). Refactoring - improving coupling and cohesion of existing code. 11th Working Conference on Reverse Engineering, Reverse Engineering, 2004. Proceedings. 11th Working Conference on, Reverse Engineering, 144–151. https://doi-org.proxy-library.ashford.edu/10.1109/WCRE.2004.33

GeeksforGeeks. (n.d.). Design Patterns. Retrieved January 20, 2020 from https://www.geeksforgeeks.org/mvc-design-pattern/

Gielen, F. (n.d.). What is the ADD Process? Coursera. Retrieved December 10, 2019 from https://www.coursera.org/lecture/iot-software-architecture/what-is-the-add-process-kkO5T

Haoyu, W., & Haili, Z. (2012). Basic Design Principles in Software Engineering. 2012 Fourth International Conference on Computational & Information Sciences, 1251. Retrieved from http://search.ebscohost.com.proxylibrary.ashford.edu/login.aspx?direct=true&db=edb&AN=86532662&site=eds-live&scope=site

Indeed.com. (n.d.) Find Jobs. Retrieved January 18, 2020 from indeed.com.

Ingeno, J. (2018). Software architect's handbook: Become a successful software architect by implementing effective architecture concepts. Retrieved from https://www.vitalsource.com/

Kulkarni, P. S., Patil, S. P., Rane, P. B., & Meshram, B. B. (2012). Use of Design Patterns in E-commerce Application: Survey. International Journal of Advanced Research in Computer Science, 3(1), 447. Retrieved from http://search.ebscohost.com.proxy-library.ashford.edu/login.aspx?direct=true&db=edb&AN=91876394&site=eds-live&scope=site

Neill, C. J., Sangwan, R. S., & Kilicay-Ergin, N. H. (2017). A Prescriptive Approach to Quality-Focused System Architecture. IEEE Systems Journal, Systems Journal, IEEE, 11(4), 1994–2005. https://doi-org.proxy-library.ashford.edu/10.1109/JSYST.2015.2423259

Packt Editorial Staff. (2018). What is a multi-layered software architecture? Retrieved December 21, 2019 from https://hub.packtpub.com/what-is-multi-layered-software-architecture/

Richards, M. (2015). Software Architecture Patterns. O'Reilly Media, Inc. Retrieved December 21, 2019 from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

Spencer, L. D., and Richards, S. H. (2015). Reliable JavaScript: How to Code Safely in the World's Most Dangerous Language, John Wiley & Sons, Incorporated. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/ashford- ebooks/detail.action?docID=4040657.

Swan, J., Adriænsen, S., Barwell, A. D., Hammond, K., & White, D. R. (2019). Extending the "Open-Closed Principle" to Automated Algorithm Configuration. Evolutionary Computation, 27(1), 173–193. https://doi-org.proxy- library.ashford.edu/10.1162/evco_a_00245

Zhang, A. (2018). How to write a good software design doc. Retrieved from https://medium.freecodecamp.org/how-to-write-a-good-software-design-document-66fcf019569c

Zimeo, E., Oliva, G., Baldi, F., & Caracciolo, A. (2013). Designing a Scalable Social E-Commerce Application. Scalable Computing: Practice & Experience, 14(2), 131. Retrieved from http://search.ebscohost.com.proxy-library.ashford.edu/login.aspx?direct=true&db=edb&AN=93425282&site=eds-live&scope=site