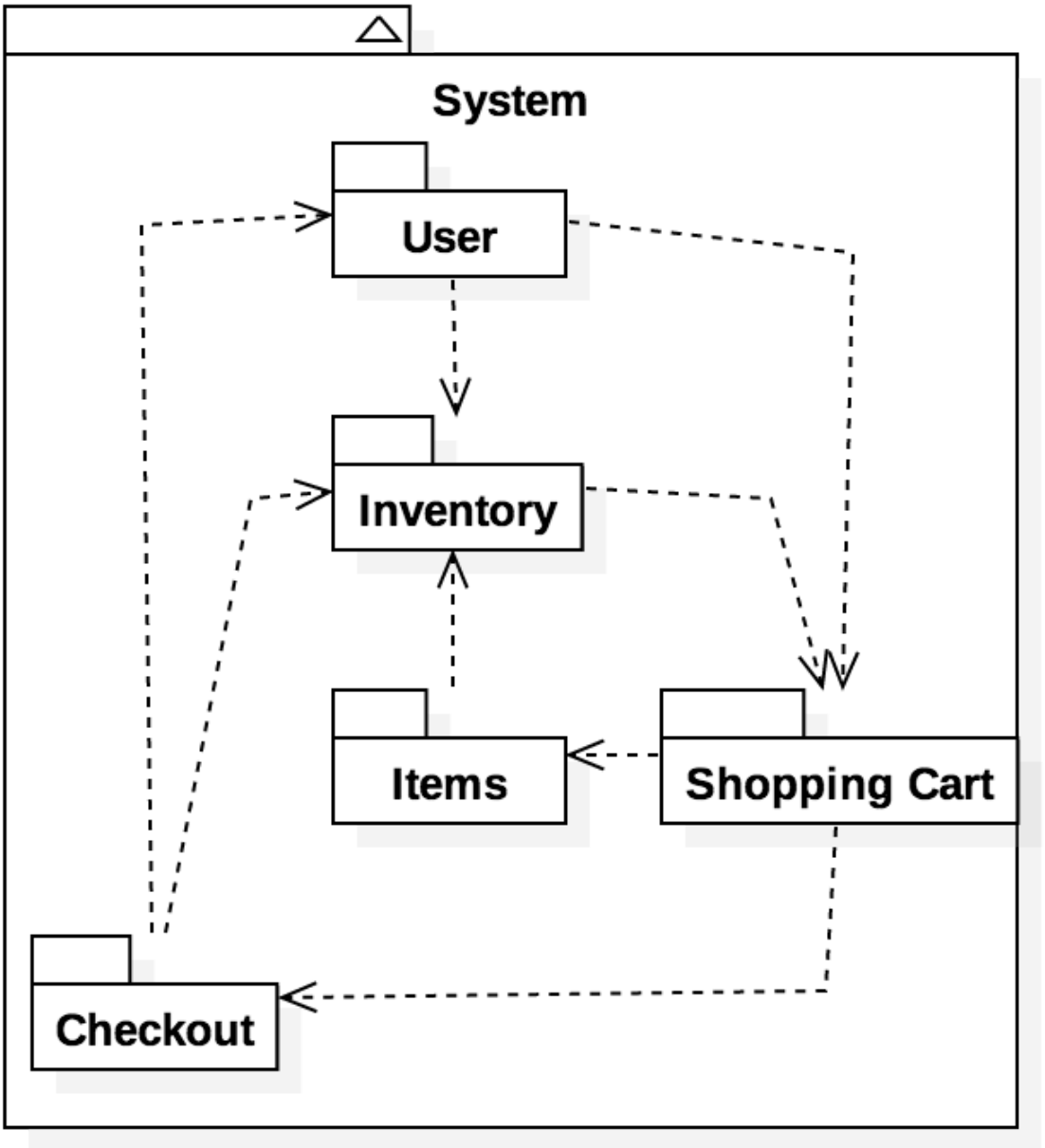
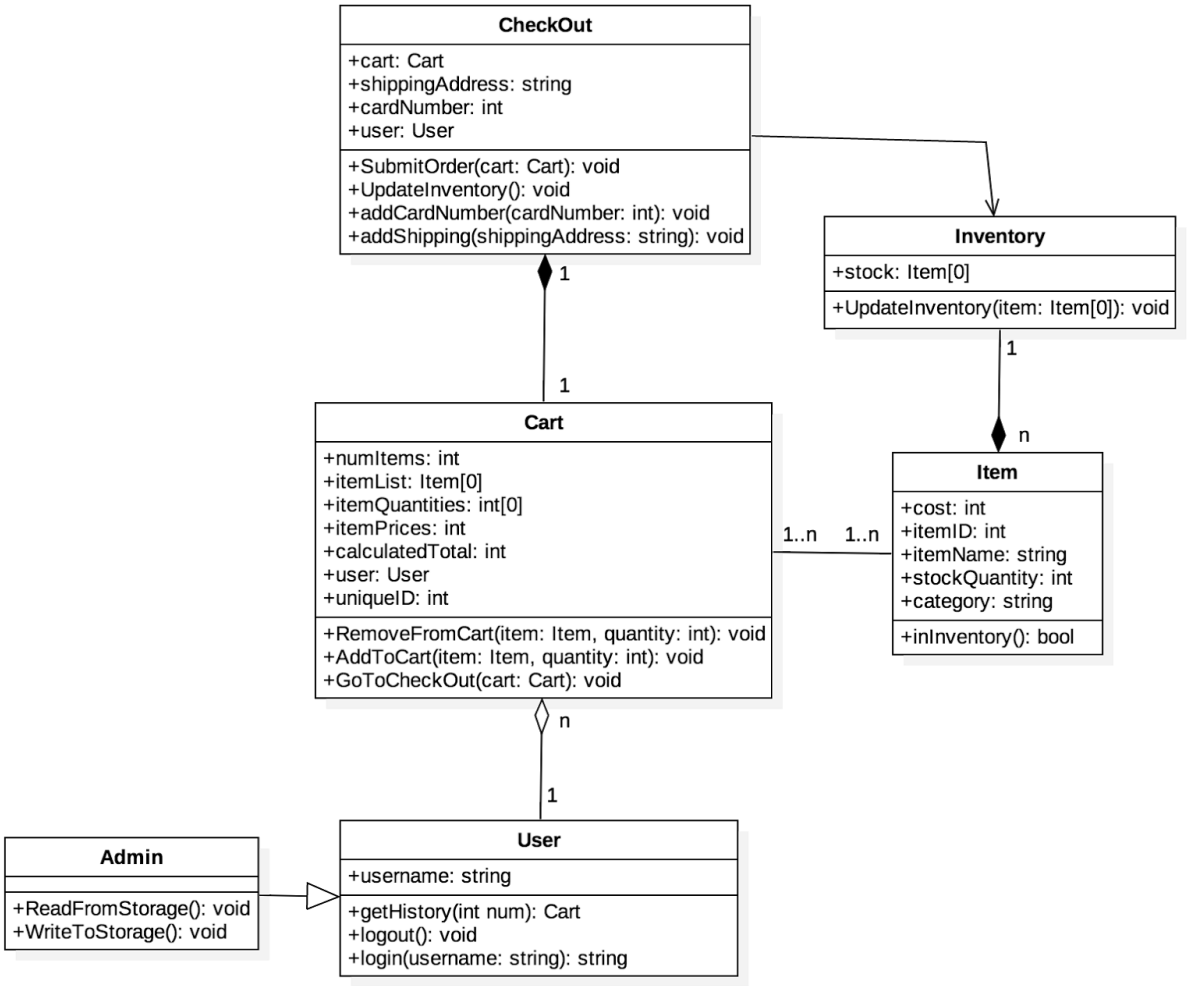
**Package Diagram**

The package diagram makes the prototype and its pieces easy to understand at a fundamental level. This is beneficial for a client, one of the stakeholders, who do not necessarily need to understand everything underneath that will make the whole system work. Another stakeholder that can benefit from this diagram is obviously the development team. Early on in the development process, having a diagram as simple as this one allows the team to discuss what is needed at each section of the model in order to implement all features the system will need. This model shows which classes, or packages, access and depend on other classes.



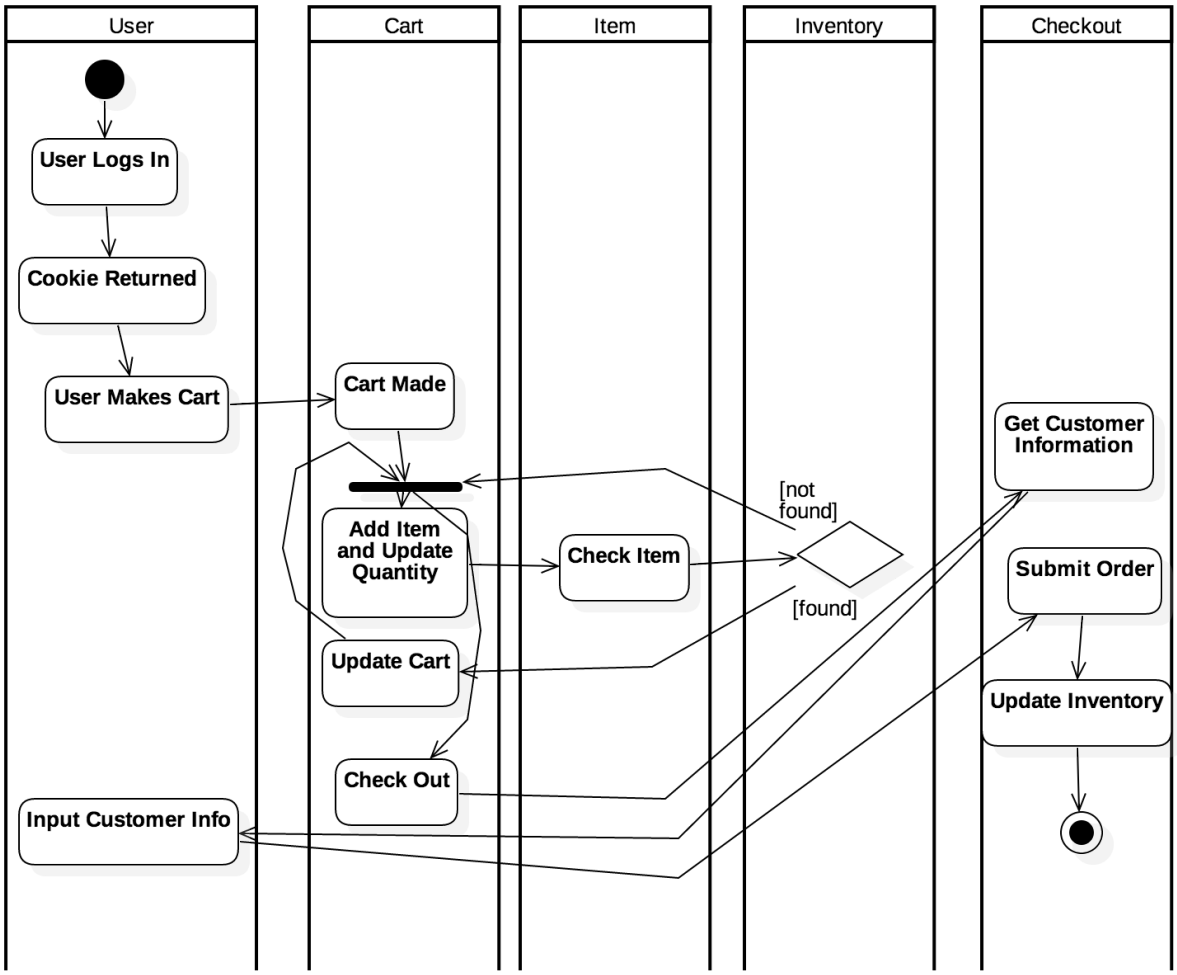
**Class Diagram**

Each User can have multiple Carts; a Cart cannot exist without a User having created it. Each Cart can check out only once, and as such a CheckOut class can't exist without a Cart object. One or more Items can exist in one or more Carts. The Admin class inherits from the User class and has the added functionality of writing/reading from storage mediums such as files or databases. Only one Inventory exists, and it contains one or more Items. Class diagrams are important for developers that need to know what they need to implement, and they allow testers to know what parts interact with other parts to ensure the software works according to specification.



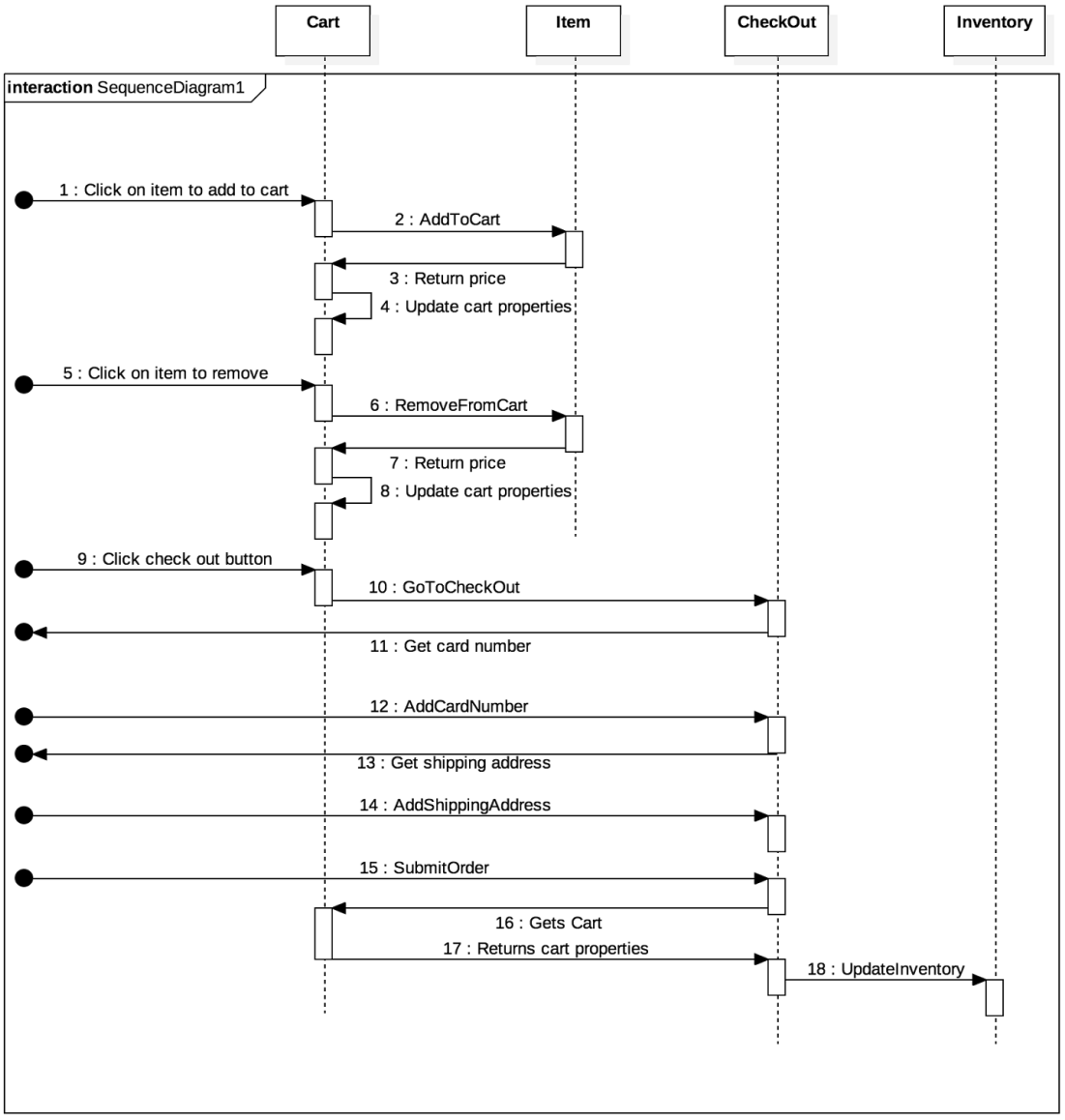
**Activity Diagram**

The user logs in, and the cookie string is returned for the browser; then the user makes a cart. The user can then add an item, update the quantity, or check out. When the user adds an item or updates the quantity, the inventory class checks to make sure the item is available. If it is found, it updates the cart. When the user checks out, the checkout class asks the user for information, such as credit card and shipping address. The user then confirms the purchase and it updates the inventory. This activity diagram useful to the user and the developer because it allows them both to see what choices can be made. The developer also has to be able to return an empty item object when the item is not found in the inventory.



**Sequence Diagram**

This shows how the user can add and remove Items from the Cart. The Cart object interacts with the Item objects and then updates the Cart object. When the user wants to check out, the user clicks the check out button, which passes the Cart object into the CheckOut object, which then requests the card number and shipping address from the user. Then the user clicks the submit order button and the CheckOut object checks the Cart object and then updates the Inventory object. The sequence diagram is useful to the user and developers. This shows the user what the interface is and what they have to do when adding/removing items and checking out. This shows the developer how classes interact with each other by using the method calls.



**Object-Oriented Terms**

Object - an instance of a class; great for code reusability

Class - a program-code template for creating an object; allow code organization and easily reusable pieces

Inheritance - the passing of one class' attributes to another (child) class closely related to its parent class; saves time and allows the main code to not have to be compiled again

Interface - description of the actions that an object can do; they provide signatures to functions

Polymorphism - the ability to redefine methods for derived classes; it supports the building of extensible systems

Abstraction - hiding all but the relevant data of an object to reduce complexity; allows the essential components to be viewed easily

Encapsulation - binding together data and functions that manipulate data to keep them safe from outside use; easily allows one class to interact with another class with only little information known about the first class

Modularity - extent to which an application may be divided into smaller modules; allows pieces to easily be debugged independently

Coupling - manner and degree of interdependence between software modules; shows a similarity relation between two classes

Cohesion - degree to which the elements of a module belong together; having only closely related methods within a class demonstrate high cohesion

Aggregation - indicates that one class is part of another class; child class instance can outlive parent class because they only have a relationship, and one object does not own the other object

Dependency - also called a using relationship, which means one class is dependent on another class; it makes it easy to manage objects with dependencies on other objects

Composition - another form of aggregation relationship but child class' instance lifecycle is dependent on the parent class' instance; focuses on having instances of classes within classes to allow code reusability

Association - how a class relates to another class; a linkage between two classes; used to show relationships where one classes forces another class to complete an action on its behalf