differentiation

Very robust and **highly configurable** message sending & receiving mechanism with **low cpu consumption & high speed**

- Implemented with **dynamic thread pool** and thread **wait & notify** mechanism.
- A message will be sent out again, if no ACK is received in a specified time period.

  This will continue to loop if the expected ACK never comes. The time period that a thread will quietly wait for the ACK can be configured by parameter, thus you can have different agility under different circumstances.

  - A very primitive way to implement this would be to keep the thread "busy waiting", i.e. keep the thread continuously checking whether the desired ACK have arrived or not. This would most certainly result in huge cpu consumption, especially when there are large amount of spawned threads concurrently doing this meaningless waiting. **Our implementation avoids thread busy waiting by utilizing the thread wait mechanism**: a message-sending thread would get a notification object when it is first created, after sending its message, it will enter a `while` loop and continuously `wait` for a specified time. During the wait period, the cpu consumption of this thread will be near-zero, resulting in great scalability and concurrency performance of our project.
  - Next thing, agility. Say a thread is now set to wait a certain period for its desired ACK to arrive. The intuitive thing to do would to be something like `Thread.sleep(sleep_time)`. This also has near-zero cpu consumption, however, we consider this way not agile enough: a thread would only be able to do the check in most of its life time, which means even if the ACK arrives a millisecond after its last check, the thread will have to wait until its next full sleep period to find this out. The drop-back of this aforementioned implementation would hence be that during a high volume access to the server, there could potentially be a huge amount of sleeping thread, more than which the memory could afford of. **Our implementation takes this burden off the server by taking advantage of the thread notification mechanism**: when a thread is spawned and submitted to our thread pool, it will get hold of a notification object. The receiver daemon will also keep this exact same object. **As soon as the receiver daemon has its parsers discovered the ACK** desired by this thread, it will **notify this waiting thread** to stop waiting and break out of its `while` loop. This guarantees that the sender thread will exist for as few time as possible, hence improves the performance and lowers the resource consumption of our project.
- A highly effective implementation to **guarantee the "At most (receive) once; At least (send) once" semantic**:
  - Our receiver daemon will save the sequence number of any already processed messages. If some time in the future it receives a message with this sequence number again, it will **skip processing** it and **also send back ACK to the message source once again**. This implementation guarantees that our amazon server can avoid confusion and chaos caused by the Two Generals Problem to the maximum.
- Our message receiver daemon only focuses on receiving messages from the socket channel, **any more complicated operation** other than reading would be **submitted to a thread pool** to guarantee the efficiency and effectiveness of the main receiver. This differentiates our implementation from those who use the main receiver thread to process every single message with the fact that **our implementation has better receiver effectiveness and can scale much better under high-concurrency circumstances**.

Our project has good **OO design** and obeys the **Software Design Principles**

- By utilizing the **Dependency Injection** and **Control Inversion** Design principles for **every controller, service and model** in our project, our project has achieved **great modularity and adaptivity** to potential changes in the future.
- Classes and functions are well separated according to their purpose and functionality: we have carefully separated our project into more **than 30 classes**. This respects the **Single Responsibility Principle** and lets our project have the **minimum level of code duplication** and the **maximum level of code reuse**, as well as **high coherence and low coupling** between modules.

Persistent Shopping Cart

- We have shopping cart for each user and it is persisted in the database, so that every user can login, add some products to cart, logout, and by next time they return to our website, the products they chose are still there.

Nearest-First warehouse finding algorithm

- After the customer buying the products and specifies their destination, we will automatically calculate the nearest warehouse for them based on our warehouse location data, to guarantee that they will get their products fast.

Password Hashing for **Security**:

- Every user's password is hashed with md5 algorithm for advanced security

Session-based login & logout:

- A user will have there user id stored in their session attribute, so that during the entire session of accessing our website, we will continuously have their user id to identify them.
- Logout is as simple as wipe out their id from their session.

Real-time package status **update** system, with package **arrival notification** :

- Our amazon server will **track the status** of a package from the point that the order is made, all the way until the package is delivered to our customer. During the whole process, we provide real-time status update for every package of every customer. It is as simple as entering their tracking number to view your packages' **latest transport information**.
- As soon as the package is **delivered**, our amazon server will receive notification **from the ups server**, and we are hence able to finalize the status of this package.

Merchandise viewing **by catalog**:

- Users can easily choose to view **a specific range** of our products by choosing from the catalog on the left panel on our website.