During the next iteration implement the following improvements in your development process and be ready to demonstrate and discuss them during project review (failing to implement them will result in penalty to the grade for the given number of points).

Amplify learning (1 point):

Increase the feedback about the product to the team in the following ways:

a. Write and run developer tests as you write the code (book projects, brainstorm on how you can fulfill this part).

b. Write and run customer tests and make them automated.

c. Observe usability tests of each feature as it nears completion, so you can see how user reacts to your implementation.

Decide as Late as Possible (1 point):

List decisions that are about to be made on your project. Group the list of decisions into two categories - tough to make and easy to make. Then discuss what information you would need to turn each tough decision into an easy decision. Pick three tough decisions and apply some of the delaying tactics under to delay those decisions as long as possible:

a. **Share partially complete design information.** The notion that a design must be complete before it is released is the biggest enemy of concurrent development. Requiring complete information before releasing a design increases the length of the feedback loop in the design process and causes irreversible decisions to be made far sooner than necessary. Good design is a discovery process, done through short, repeated exploratory cycles.
b. **Organize for direct, worker-to-worker collaboration.** Early release of incomplete information means that the design will be refined as development proceeds. This requires that people who understand the details of what the system must do to provide value must communicate directly with people who understand the details of how the code works.

c. **Develop a sense of how to absorb changes.** In "Delaying Commitment" Harold Thimbleby observes that the difference between amateurs and experts is that experts know how to delay commitments and how to conceal their errors for as long as possible. Experts repair their errors before they cause problems. Amateurs try to get everything right the first time and so overload their problem-solving capacity that they end up committing early to wrong decisions. Thimbleby recommends some tactics for delaying commitment in software development, which could be summarized as an endorsement of object-oriented design and component-based development:

   - **Use modules:** Information hiding, or more generally behavior hiding, is the foundation of object-oriented approaches. Delay commitment to the internal design of the module until the requirements of the clients on the interfaces stabilize.
   - **Use interfaces:** Separate interfaces from implementations. Clients should not depend on implementation decisions.
   - **Use parameters:** Make magic numbers—constants that have meaning—into parameters. Make magic capabilities like databases and third-party middleware into parameters. By passing capabilities into modules wrapped in simple interfaces, your dependence on specific implementations is eliminated and testing becomes much easier.
   - **Use abstractions:** Abstraction and commitment are inverse processes. Defer commitment to specific representations as long as the abstract will serve immediate design needs.
   - **Avoid sequential programming:** Use declarative programming rather than procedural programming, trading off performance for flexibility. Define algorithms in a way that does not depend on a particular order of execution.
   - **Beware of custom tool building:** Investment in frameworks and other tooling frequently requires committing too early to implementation details that end up adding needless complexity and seldom pay back. Frameworks should be extracted from a collection of successful implementations, not built on speculation.

- **Avoid repetition:** This is variously known as the Don't Repeat Yourself (DRY)[17]or Once And Only Once (OAOO)[18]principle. If every capability is expressed in only one place in the code, there will be only one place to change when that capability needs to evolve, and there will be no inconsistencies.
- **Separate concerns:** Each module should have a single, well-defined responsibility. This means that a class will have only one reason to change.[
- **Encapsulate variation:** What is likely to change should be inside; the interfaces should be stable. Changes should not cascade to other modules. This strategy, of course, depends on a deep understanding of the domain to know which aspects will be stable and which variable. By application of appropriate patterns, it should be possible to extend the encapsulated behavior without modifying the code itself.[
- **Defer implementation of future capabilities:** Implement only the simplest code that will satisfy immediate needs rather than putting in capabilities you "know" you will need in the future.[21]You will know better in the future what you really need then, and simple code will be easier to extend if necessary.
- **Avoid extra features:** If you defer adding features you "know" you will need, then you certainly want to avoid adding extra features "just-in-case" they are needed. Extra features add an extra burden of code to be tested, maintained, and understood. Extra features add complexity, not flexibility.

d. **Develop a sense of what is critically important in the domain.** Forgetting some critical feature of the system until too late is the fear that drives sequential development. If security, or response time, or failsafe operation are critically important in the domain, these issues need to be considered from the start; if they are ignored until too late, it will indeed be costly. However, the assumption that sequential development is the best way to discover these critical features is flawed. In practice, early commitments are more likely to overlook such critical elements than late commitments, because early commitments rapidly narrow the field of view.

e. **Develop a sense of when decisions must be made.** You do not want to make decisions by default, or you have not delayed them. Certain architectural concepts such as usability design, layering, and component packaging are best made early so as to facilitate emergence in the rest of the design. A bias toward late commitment must not degenerate into a bias toward no commitment. You need to develop a keen sense of timing and a mechanism to cause decisions to be made when their time has come.

f. **Develop a quick response capability.** The slower you respond, the earlier you have to make decisions. Dell, for instance, can assemble computers in less than a week, so it can decide what to make less than a week before shipping. Most other computer manufacturers take a lot longer to assemble computers, so they have to decide what to make much sooner. If you can change your software quickly, you can wait to make a change until customers know what they want.