Chapter 1: Well design app rook

1. How do we write a great software?

* **Make sure your software does what the customer wants it to do.** This step focuses on the customer. Make sure the app does what it is supposed to do first. This is where getting good requirements and doing some analysis comes in.
* **Apply basic OO principles to add flexibility.** Once your software works, you can look for any duplicate code that might have slipped in, and make sure you’re using good OO programming techniques. *That’s the power of encapsulation: by breaking up the different  
  parts of your app, you can change one part without having to change all the other parts. In general, you should encapsulate the parts of your app that might vary away from the parts that will stay the same.*
* **Strive for a maintainable, reusable design.** Got a good object-oriented app that does what it should? It’s time to apply patterns and principles to make sure your software is ready to use for years to come.
* *Flexibility:* Use me so that your software can change and grow without constant rework. I keep your application from being fragile.
* *Encapsulation:* You use me to keep the parts of your code that stay the same separate from the parts that change; then it’s really easy to make changes to your code without breaking everything.
* *Functionality:* Without me, you’ll never actually make the customer happy. No matter how well-designed your application is, I’m the thing that puts a smile on the customer’s face.
* *Design Pattern:* I’m all about reuse and making sure you’re not trying to solve a problem that someone else has already figured out.
* OOA&D is really just an approach to writing software that focuses on making sure your code does what it’s supposed to, and that it’s well designed. That means your code is flexible, it’s easy to make changes to it, and it’s maintainable and reusable.

Chapter 2: Gathering requirements

1. **A requirement** is a singular need detailing what a particular product or service should be or do. It is most commonly used in a formal sense in systems engineering or software engineering.

* The best way to get good requirement is to understand what a system is supposed to do.
* Your system must work in the real world so plan and test for when things go wrong.

1. **A use case** is a technique for capturing the potential requirements of a new system or software change. Each use case provides one or more scenarios that convey how the system should interact with the end user or another system to achieve a specific goal.

* *Clear Value:* Every use case must have a clearvalue to the system. If the use case doesn’t help the customer achieve their goal, then the use case isn’t of much use.
* *Start and Stop:* Every use case must have a definite starting and stopping point. Something  
  must begin the process, and then there must be a condition that indicates that the process is complete.
* *External Initiator:* Every use case is started off by an externalinitiator, outside of the system. Sometimes that initiator is a person, but it could be anything outside of the system.

Chapter 3: Requirement change

* Requirement always change. If you’ve got good use cases, though, you can usually change your software quickly to adjust to those new requirements.
* A complete path throught a use case, from the first step to the last, is called **a scenario**.
* Most use cases have several different scenarios, but they always share the same user goal.
* Sometimes a change in requirements reveals problems with your system that you didn’t even know were there.
* Change is constant, and your system should always improve every time you work on it.

Requirements are things your system must do to work correctly