Object Oriented Analysis and Design Principles

- **Encapuslation**: hide informations or behaviour from user or from other part of the application. Informations can be single property, group of property or methods. Keeps part of the data which stay same separate from the parts that change. So changing code in one place does not effect in other place. Whenver we see a duplicate codes look for a place to encapuslate it. Encapsulation is breaking an application up into logical parts.
- **Enum**: use enums to force tyoe safety and value safety in an application
- **Objects**: Objects are the black boxes which hide all data from user or from other parts of the

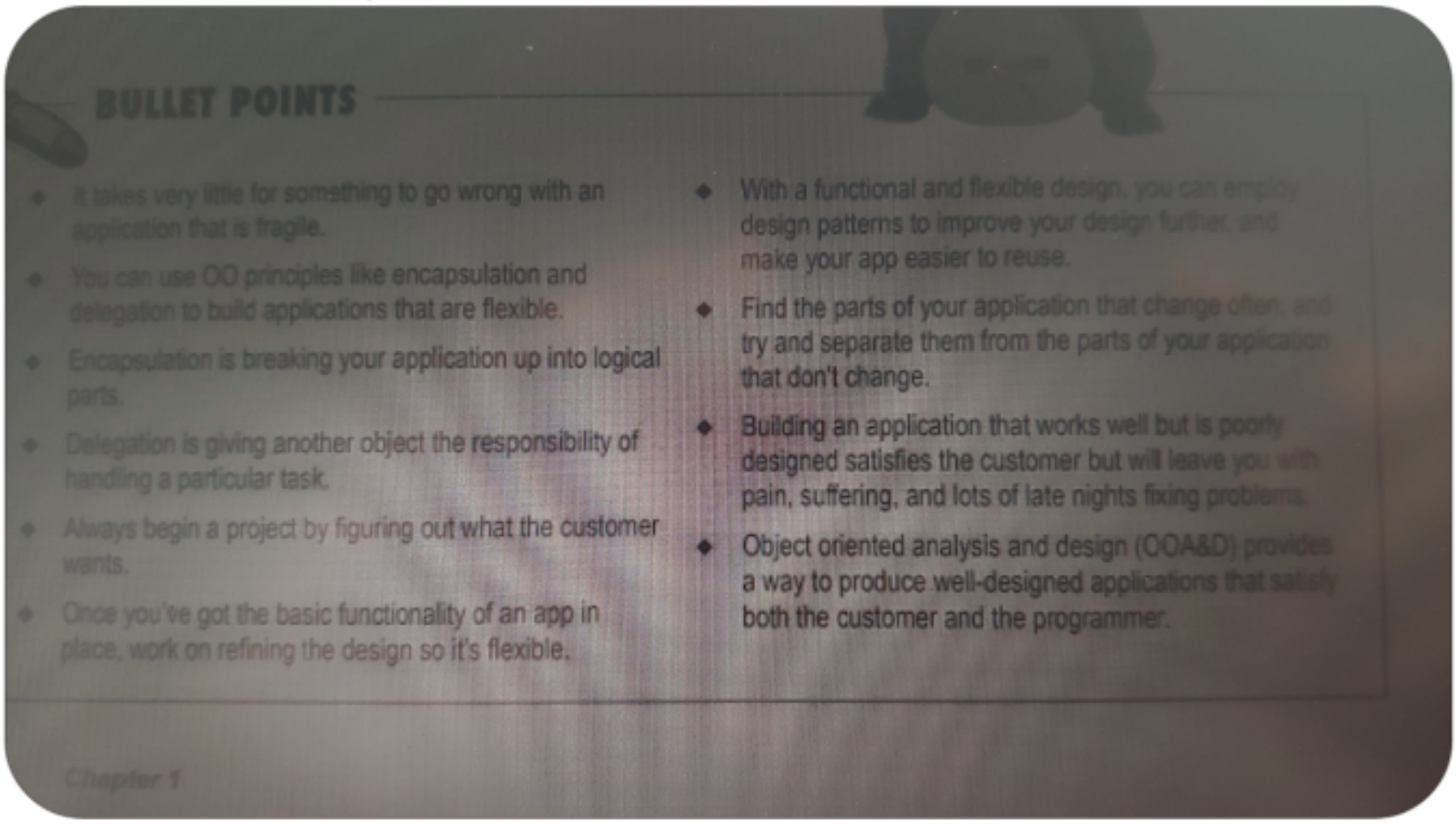
applications and explore only well designed interfaces to outside, through which can gets the required data and do the calculations. It's a good practice always push data to object and get as less as possible data from objects.

- **Delegation**: Delegate certain functionality to other objects which has the required data instead of fetching those data and do calculation upon them.
- **Composition**: An object has a reference of another object to delegate certain task to it. Composition is always better than inheritence as this reduces the coupling between objects
- **High Cohesion**: An object or a class will be highly cohesive when only related things are grouped together instead of

keeping all unnessaey things in a same class or object.

- **Loose Coupling**: when two objects can change independently without effecting each other then we can say both objects are loosely coupled to each other. This ll eliminates changing an object when dependent objects are changed.
- **Less Fragile and Highly Robust**: when an application is designed in such way that, changing in one part of the application will not break ornintriduce bugs in other parts of the application. If it dows so then we can say an applucation is highly robust and less fragile
- YAGNI: You are not gonna need it
- KISS: Keep it short and simple
- 3 steps to write great software: make sure your software does

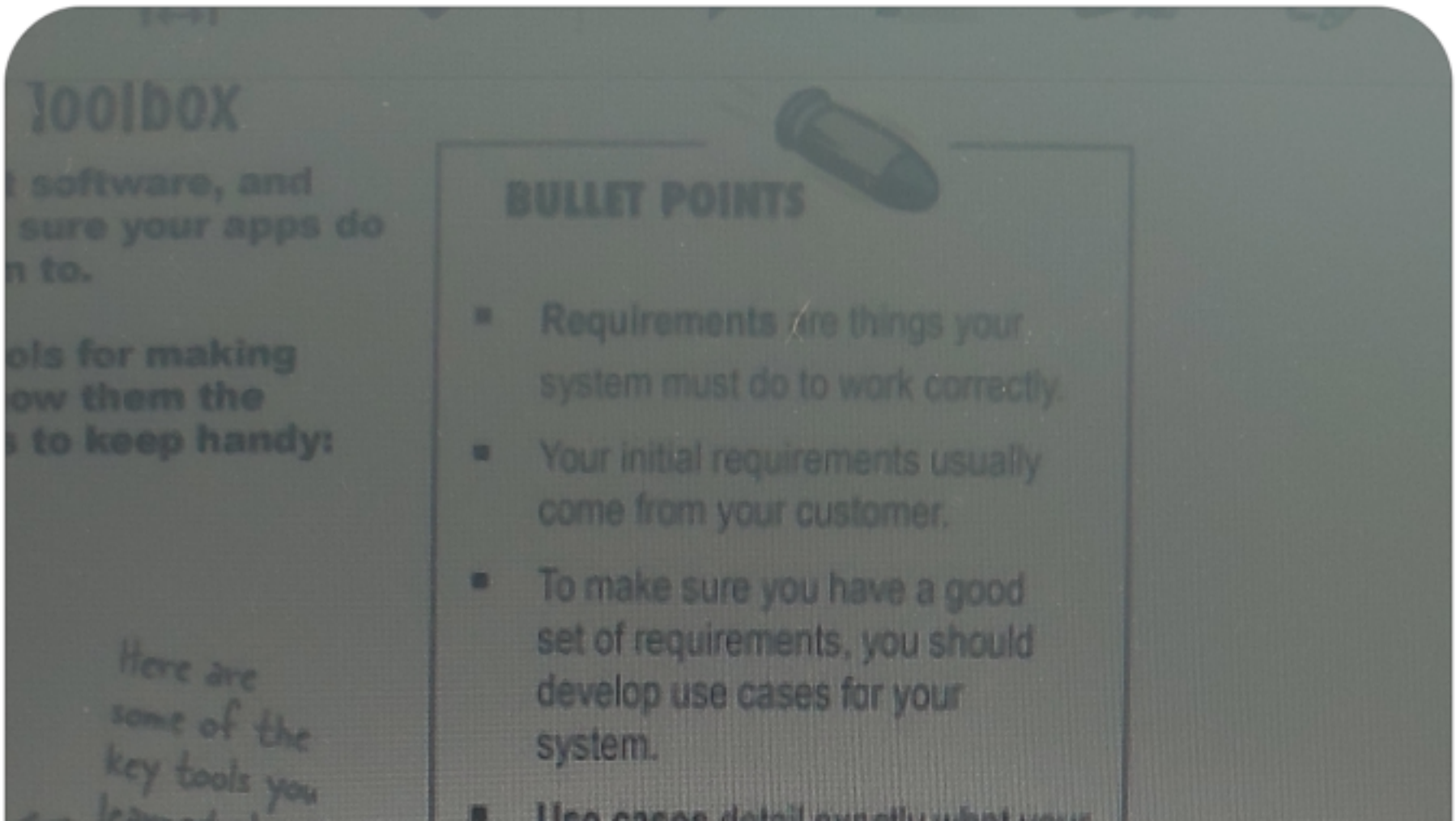
what the customer wants it to do. Apply basic OO principles to add flexibility. Strive for a maintainable reusable designs by applying design patterns.



BULLET POINTS

- It takes very little for something to go wrong with an application that is fragile.
- You can use OO principles like encapsulation and delegation to build applications that are flexible.
- Encapsulation is breaking your application up into logical parts.
- Delegation is giving another object the responsibility of handling a particular task.
- Always begin a project by figuring out what the customer wants.
- Once you've got the basic functionality of an app in place, work on refining the design so it's flexible.
- With a functional and flexible design, you can employ design patterns to improve your design further, and make your app easier to reuse.
- Find the parts of your application that change often, and try and separate them from the parts of your application that don't change.
- Building an application that works well but is poorly designed satisfies the customer but will leave you with pain, suffering, and lots of late nights fixing problems.
- Object oriented analysis and design (OOA&D) provides a way to produce well-designed applications that satisfy both the customer and the programmer.

Chapter 1



Toolbox

software, and
sure your apps do
n to.

ols for making
ow them the
s to keep handy:

Here are some of the key tools you

BULLET POINTS

- Requirements are things your system must do to work correctly.
- Your initial requirements usually come from your customer.
- To make sure you have a good set of requirements, you should develop use cases for your system.

...learned about in this chapter.

We'll be adding lots more tools to these other categories* in the coming chapters.

...es

- **Use cases** detail exactly what your system should do.
- A use case has a **single goal**, but can have multiple paths to reach that goal.
- A good use case has a **starting** and **stopping condition**, an **external initiator**, and **clear value** to the user.
- A use case is simply a story about how your system works.
- You will have at least one use case for each goal that your system must accomplish.
- After your use cases are complete, you can refine and add to your requirements.
- A requirements list that makes all your use cases possible is a good set of requirements.
- Your system must work in the real world, not just when everything goes as you expect it to.
- When things go wrong, your system must have **alternate paths** to reach the system's goals.

...Patterns will find these categories ...UID and design patterns go hand in hand.

...s page, ...OOA&D

BULLET POINTS

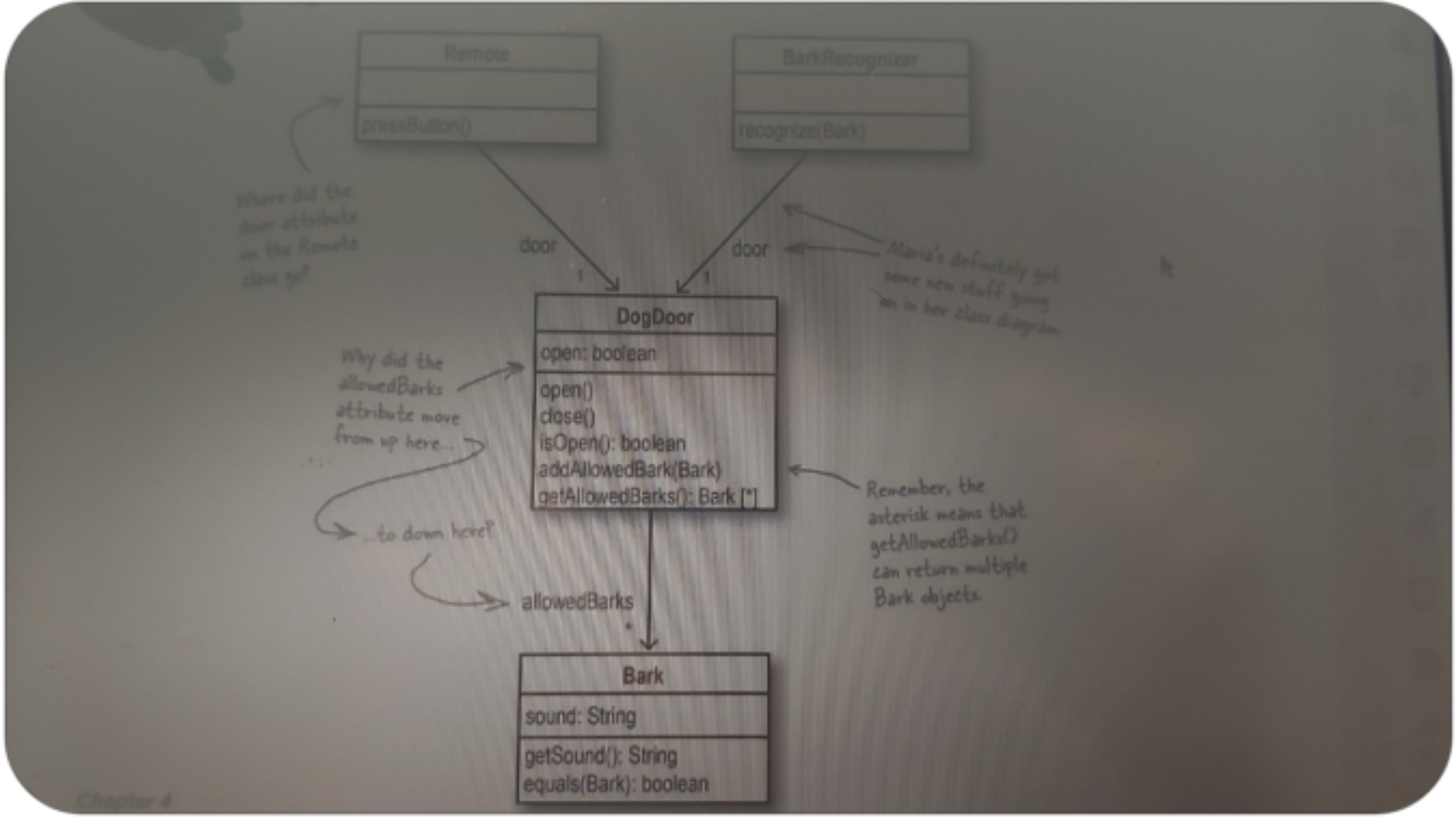
- Requirements will always change

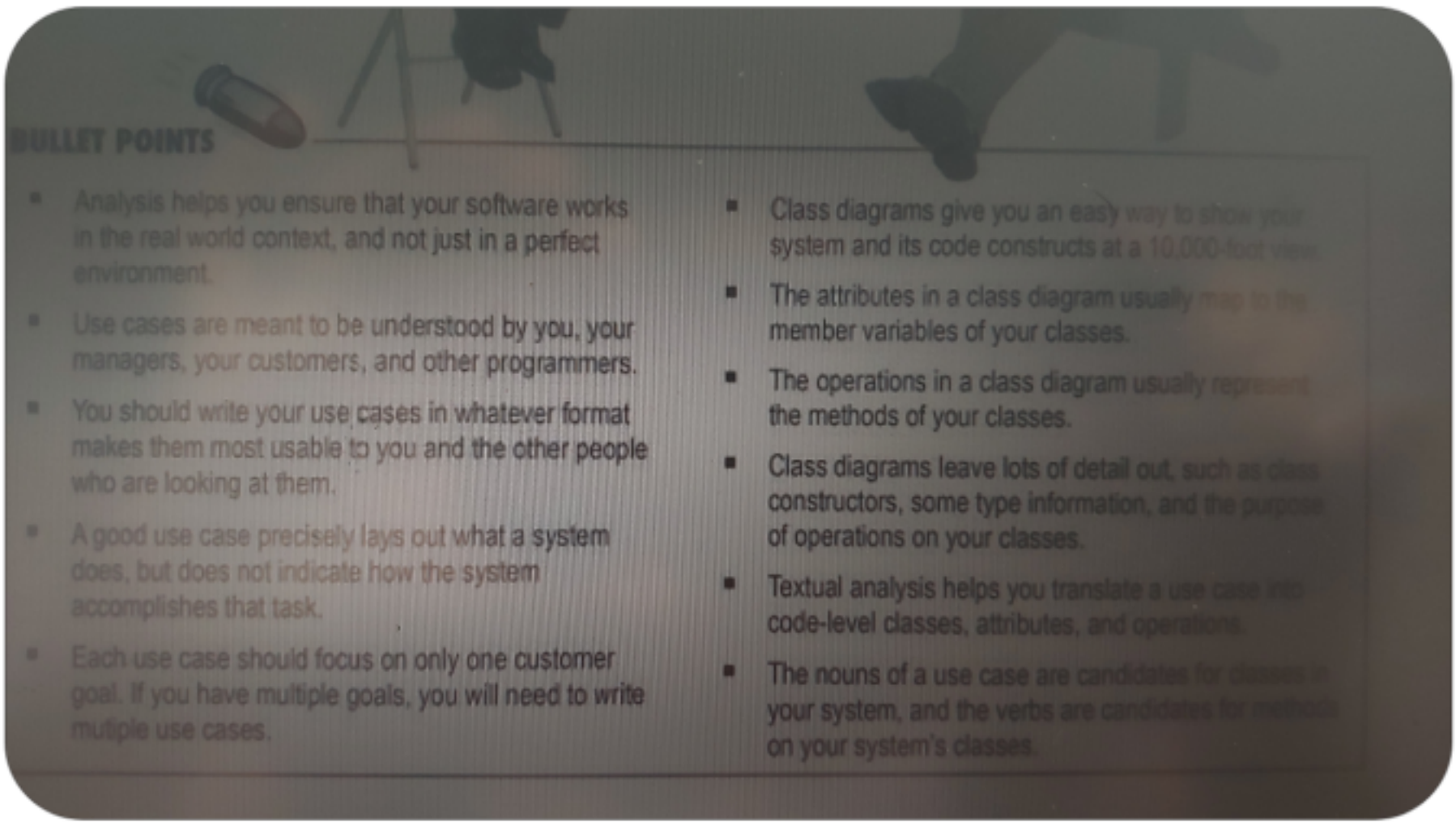
as a project progresses.

- When requirements change, your system has to evolve to handle the new requirements.
- When your system needs to work in a new or different way, begin by updating your use case.
- A **scenario** is a single path through a use case, from start to finish.
- A single use case can have multiple scenarios, as long as each scenario has the same customer goal.
- **Alternate paths** can be steps that occur only some of the time, or provide completely different paths through parts of a use case.
- If a step is optional in how a system works, or a step provides an alternate path through a system, use numbered sub-steps, like 3.1, 4.1, and 5.1, or 2.1.1, 2.2.1, and 2.3.1.
- You should almost always try to **avoid duplicate code**. It's a maintenance nightmare, and usually points to problems in how you've designed your system.

Gathering Requirements:

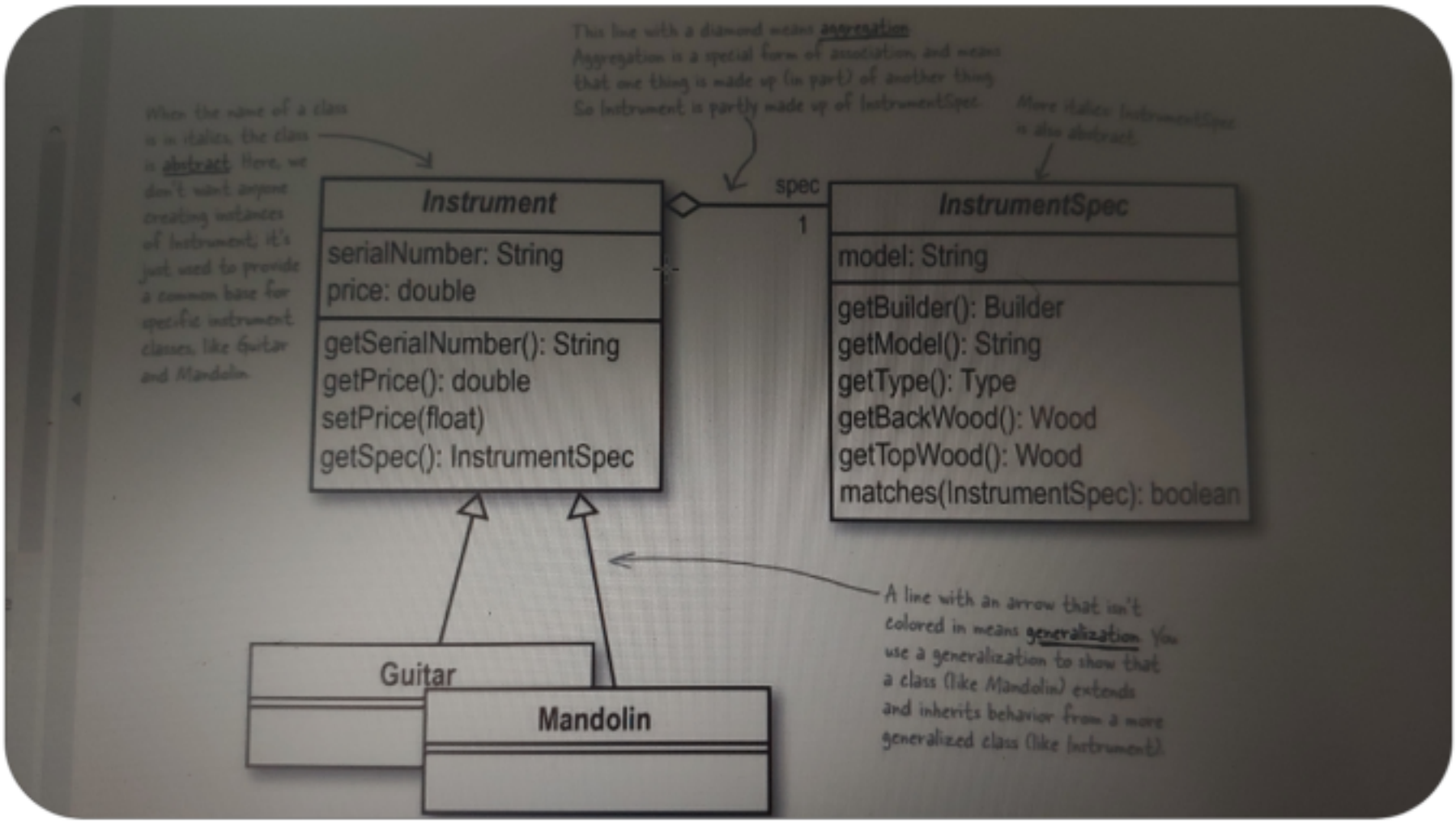
- **User Requirements**: Its a **specific thing** your **system** has to **do** to **work correctly**.
- **Use Cases**: It describes **what** your system **does** to accomplish a **particular customer goal**. Use cases contains starting and ending points, must have a clear value to the system and every use cases are started off by an external initiator. The nouns in a use cases are usually the classes we need to write. Each use cases will contain a main path and one or more alternate paths.
- **Textual Analysis**: looking at the nouns and verbs in use case to figure out classes and methods is called textual analysis
- **UML Diagram:**

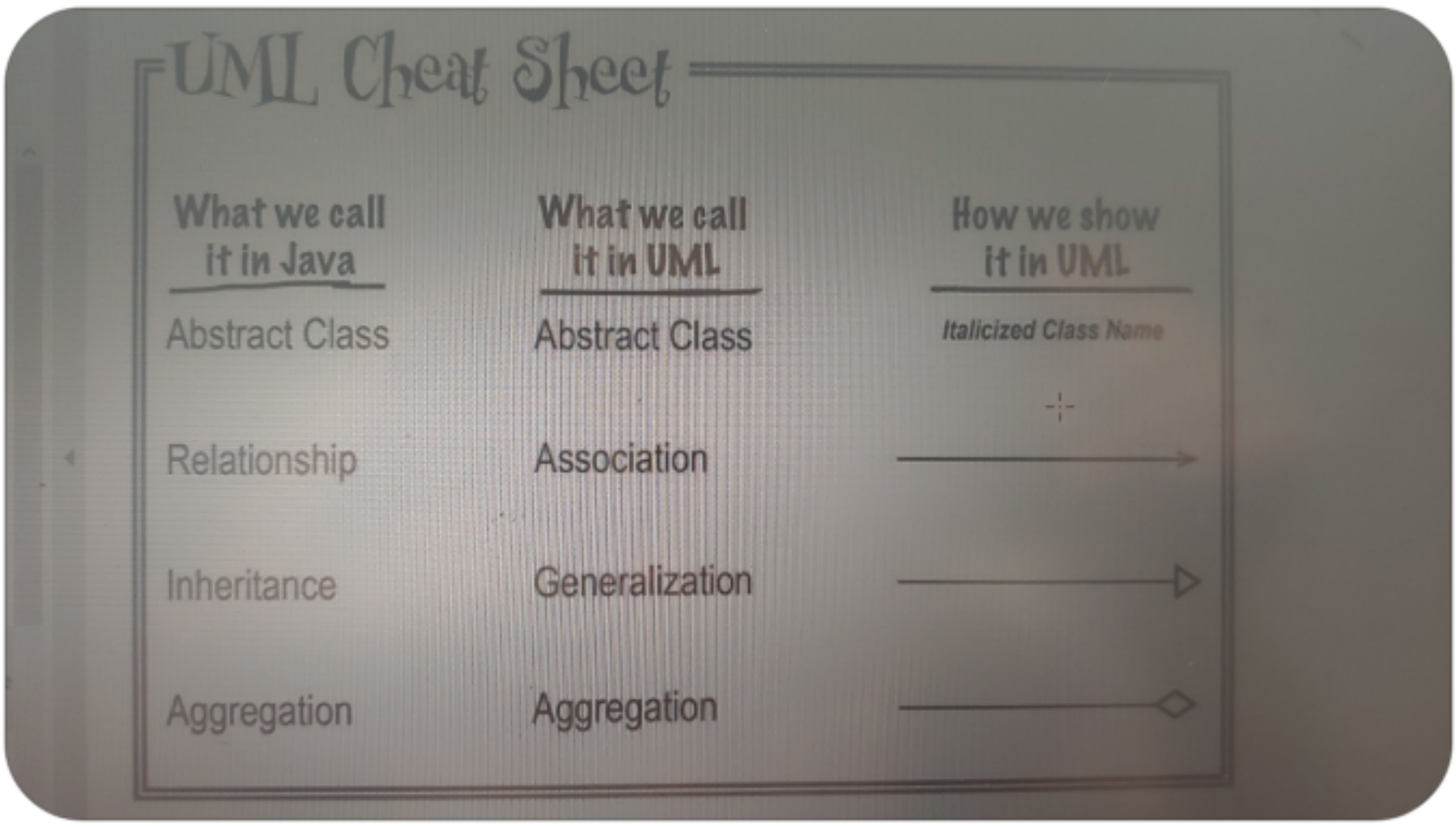




- **Abstract classes**: Abstract classes are placeholders for actual implementation classes.

The abstract class defines behaviour and the subclasses implement that behaviour.





OO Catastrophe or OO Principles

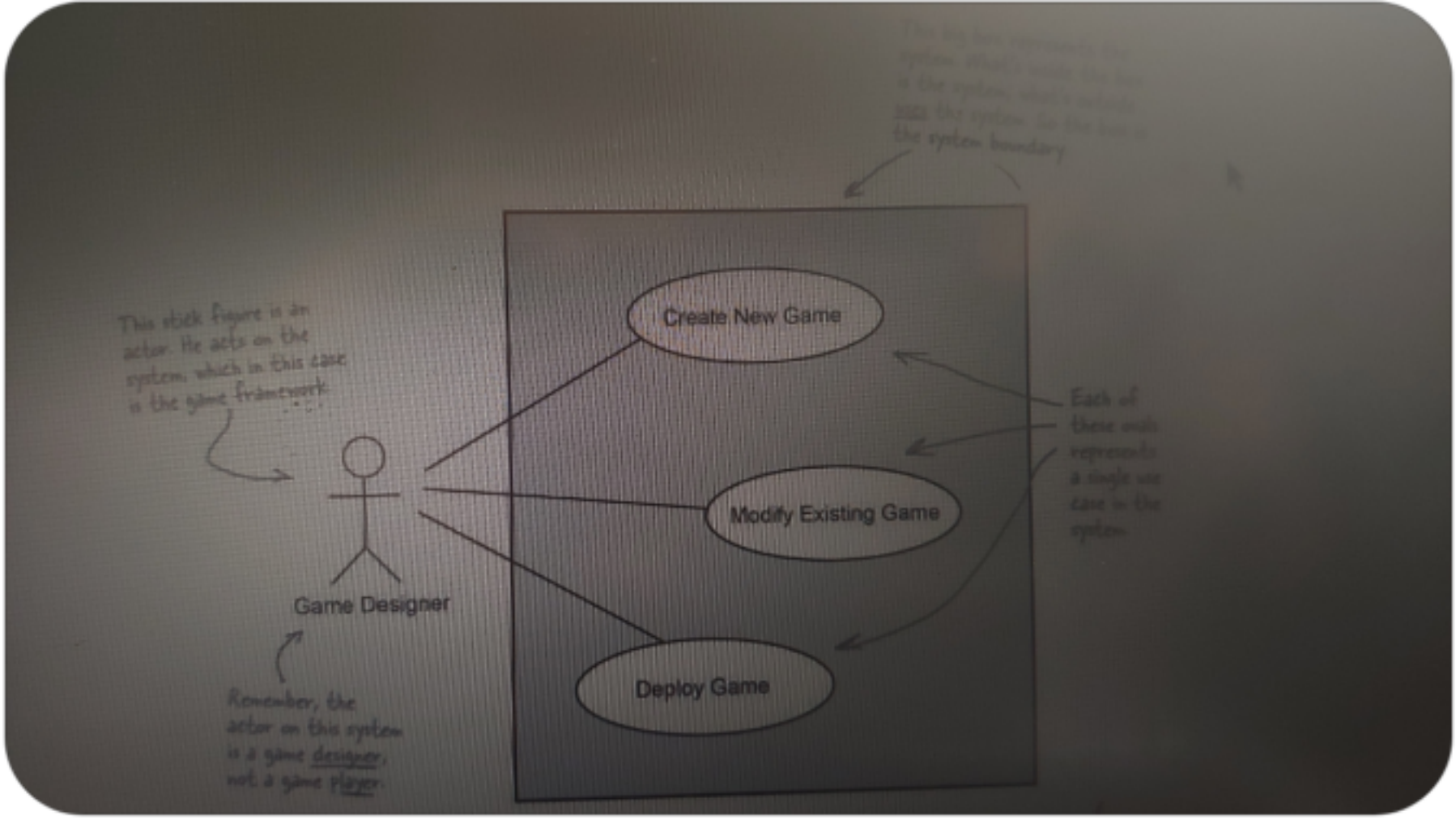
- ○ Coding to an interface rather than to an implementation makes software easier to extend. By coding to an interface, your code will work with all of the interfaces subclasses even ones that haven't been crrated yet.
- ○ **Encapsulation**: Along with preventing duplicate codes it also protect your classes from unnecessary changes. Move the behaviour which likely to change from the one which wont change frequently. **Encapsulates what varies**.
- ○ Each class has only one reason to change. Break up the functionality into multiple classes where each class does only one thing.
- ○ When we have a set of properties that vary across objects, use a collection like a Map to store

those properties dynamically. We can remove lots of methods from your classes and avoid having to change code when new properties are added to application.

- ○ A cohesive class does one thing really well and does not try to do or be something else. Each clasd has a very specific set of closely realted actions it performs.

Solving big problems

- ○ Get features from the customers and then figure out the requirements you need to implement those features.
- ○ Use case diagrams tells you everything the system needs to do in a simple easy to read format. It's a blueprints for your system

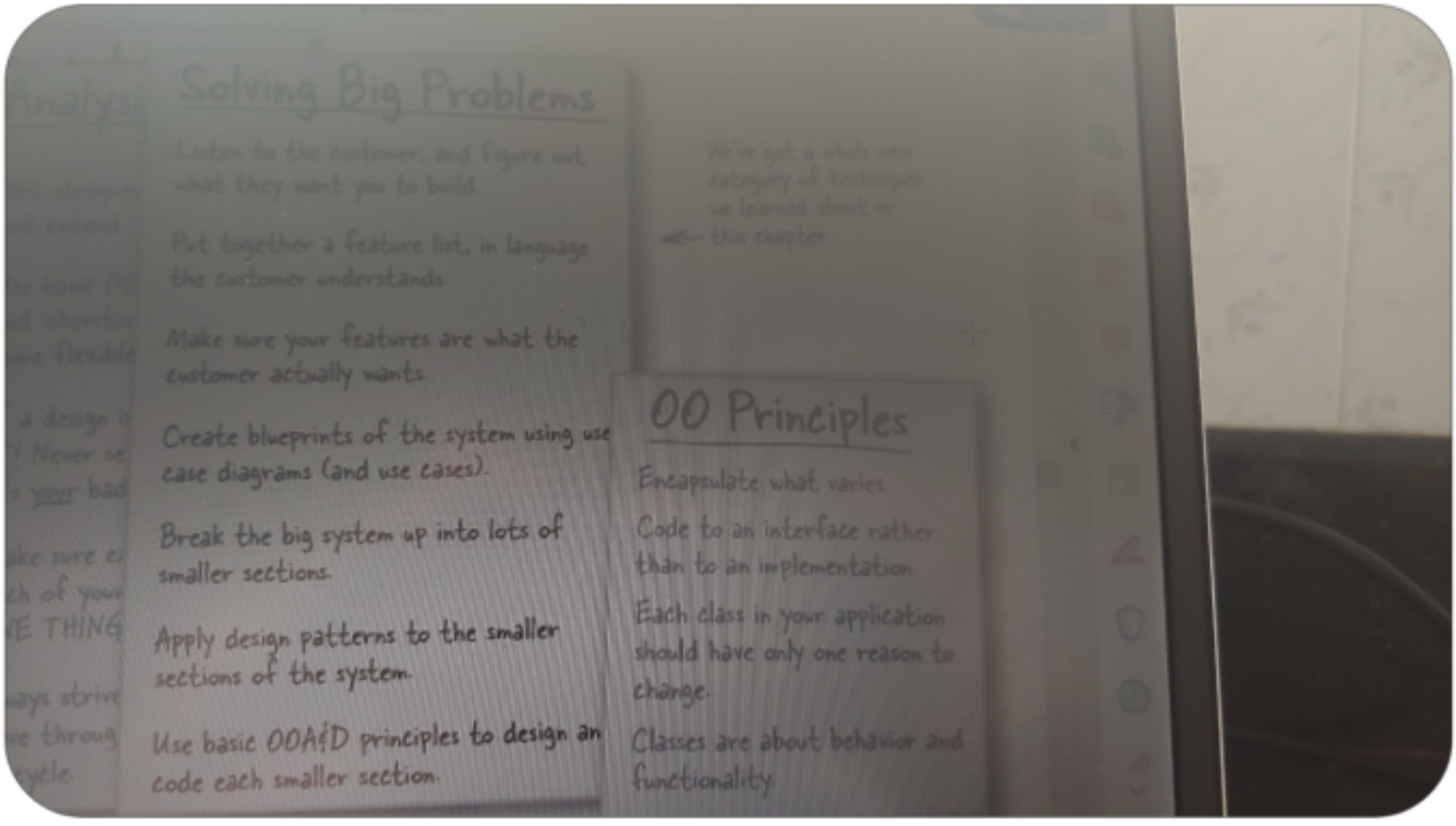


- Take the use case diagrams and make sure that all the use cases losted will cover all the featyres got from customer
- Domain analysis: process of identifying collecting organizing and representing the relevant information of a domain based upon the study of an existing systme and their development histories, knoweldge captured from domain experts underlying theory and emerging technology within a domain. Domain analysis

just means that we are describing a problem using terms the customer will understand

- ≡ ○ Breaking system into different modules
- ≡ ○ Apply some design pattern to architect the system like MVC

Domain analysis -> Features -> Use case diagram -> Requirements -> Use Cases -> Modules -> apply design pattern like MVC



==========

Java Multithreading and Concurrency