Head First OO Analysis & Design

**Chapter 1: Well-designed APPs**

**Well-designed 3 steps:**

1. Make sure your software does what the customer wants it to do.
2. Apply basic OO principles to add flexibility
3. Strive for a maintainable, reusable design.

**UML:** Unified Modeling Language.

Tips to search for mismatched object type:

1. Objects should do what their names indicate.
2. Each object should represent a single concept.
3. Unused properties are a dead giveaway.

**Encapsulation** allows you to group your application into logical parts.

* 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.

**Delegation:** Delegation is when an object needs to perform a certain task, and instead of doing that task directly, it asks another object to handle the task (or sometimes just a part of the task). Delegation makes your code more reusable. It also lets each object worry about its own functionality. This means your objects are more independent of each other, or more loosely coupled.

**Loosely coupled:** Loosely coupled is when the objects in your application each have a specific job to do, and they do only that job. So the functionality of your app is spread out over lots of well-defined objects, which each do a single task really well.

public List search (GuitaSpec searchspec) {

var matchingGuitars = new LinkedList();

foreach (guitar in Guitars) {

if (guitar.getSpec().matches(searchspec))

matchingGuitars.Add(guitar);

}

return matchingGuitars;

}

**Chapter 2: Gathering Requirements**

**Requirement:** **It’s a specific thing your system has to do to work correctly.** A requirement is usually a single thing, and you can test that thing to make sure you’ve actually fulfilled the requirement. “Scholar’s Corner”: A requirement is a singular need detailing what a particular product or service should be or do.

**Use case: A use case describes what your system does to accomplish a particular customer goal.** Use cases are all about the “what”. “Scholar’s Corner”: 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.

**One use case, three parts:** There are three basic parts to a good use case, and you need all three if your use case is going to get the job done.

1. Clear Value.  
   Every use case must have a clear value to the system. If the use case doesn’t help the customer achieve their goal, then the use case isn’t of much use.
2. 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.
3. External Initiator.

Every use case is started off by an external initiator, outside of the system. Sometimes that initiator is a person, but it could be anything outside of the system.

One of the key points about a use case is that it is focused on accomplishing one particular goal. If your system does more than one thing, then you’ll need more than one use case.

**Requirements:**

Good requirements ensure your system works like your customers expect.  
 Make sure your requirements cover all the steps in the use cases for your system.  
 Use your use cases to find out about things your customers forgot to tell you.  
 Your use cases will reveal any incomplete or missing requirements that you might have to add to your  
 system.

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.
* **Use cases** detail exactly what your system should do.
* A use case has a **single goal**, but can have multiple paths to reach this 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.

**Chapter 3: Requirements Change**

**Alternate path:** An alternate path is one or more steps that a use case has that are optional, or provide alternate ways to work through the use case. Alternate paths can be additional steps added to the main path, or provide steps that allow you to get to the goal in a totally different way than parts of the main path.

**Scenario:** A complete path through 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.

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.

**Chapter 4: Analysis**

Analysis helps you ensure your system works in a real-world context.

Write your use cases in a way that makes sense to you, your boss, and your customers. Analysis and your use cases let you show customers, managers, and other developers how your system works in a real world context.

using System;

public class Bark {

private string sound;

public Bark (string sound) {

this.sound = sound;

}

public string getSound() {

return sound;

}

public Boolean equals(Bark otherBark) {

if (this.sound.Equals(otherBark.getSound(),  
StringComparison.OrdinalIgnoreCase))

return true;

return false;

}

}

Delegation shields your objects from implementation changes to other objects in your software.

Looking at the nouns (and verbs) in your use case to figure out classes and methods is called textual analysis.

A good use case clearly and accurately explains what a system does, in language that’s easily understood. With a good use case complete, textual analysis is a quick and easy way to figure out the classes in your system.

**UML Class Diagrams:**

* A solid line from one class to another is called an association. It means that one class is associated with another class, by reference, extension, inheritance, etc. The line goes from the class with the reference to the class that is the type being referenced.
* The name of the attribute in the source class is written at the target end of the line.
* The number is the multiplicity of this association. It’s how many of the target type is stored in the attribute of the source class. The asterisk means “an unlimited number”.
* When you are using associations to represent attributes, you usually do not write the attribute that the association represents in the class’s attribute section.

**Chapter 5a: Good Design = Flexible Software**

Whenever you find common behavior in two or more places, look to abstract that behavior into a class, and then reuse that behavior in the common classes.

**UML Class Diagram:**

* When the name of a class is in italics, the class is abstract.
* A line with a diamond means aggregation. Aggregation is a special form of association, and means that one thing is made up (in part) of another thing.
* A line with an arrow that isn’t colored in means generalization. You use a generalization to show that a class extends and inherits behavior from a more generalized class.

One of the best ways to see if software is well-designed is to try and CHANGE it. If your software is hard to change, there’s probably something you can improve about the design.

**OO Catastrophe**

What is an INTERFACE?

This code construct has the dual role of defining behavior that applies to multiple types, and also being the preferred focus of classes that use those types. Suppose you’ve got an application that has an interface, and then lots of subclasses that inherit common behavior from that interface.

Represent an interface in UML: the <<interface>> word and italicized class name.

Anytime you’re writing code that interacts with these classes, you have two choices. You can write code that interacts directly with a subclass, or you can write code that interacts with the interface. When you run into a choice like this, you should always favor **coding to the interface, not the implementation**. Because it adds flexibility to your app. Instead of your code being able to work with only one specific subclass, you’re able to work with the more generic class.

By coding to an interface, your code will work with all of the interface’s subclasses-even ones that haven’t been created yet.

What is ENCAPSULATION?

It’s been responsible for preventing more maintenance problems than any other OO principle in history, by localizing the changes required for the behavior of an object to vary.

There’s more to encapsulation than just avoiding lots of copy-and-paste. Encapsulation also helps you protect your classes from unnecessary changes.

Anytime you have behavior in an application that you think is likely to change, you want to move that behavior away from parts of your application that probably won’t change very frequently. In other words, you should always try to encapsulate what varies.

What is CHANGE?

Every class should attempt to make sure that it has only one reason to this, the death of many a badly designed piece of software.

The easiest way to make your software resilient to change is to make sure ***each class has only one reason to change***. In other words, you’re minimizing the chances that a class is going to have to change by reducing the number of things in that class that can cause it to change.

When you see a class that has more than one reason to change, it is probably trying to do too many things. See if you can break up the functionality into multiple classes, where each individual class does only one thing—and therefore has only one reason to change.

OO Principles

Encapsulate what varies.  
 Code to an interface rather than to an implementation.  
 Each class in your application should have only one reason to change.

**Chapter 5b: Good Design = Flexible Software**

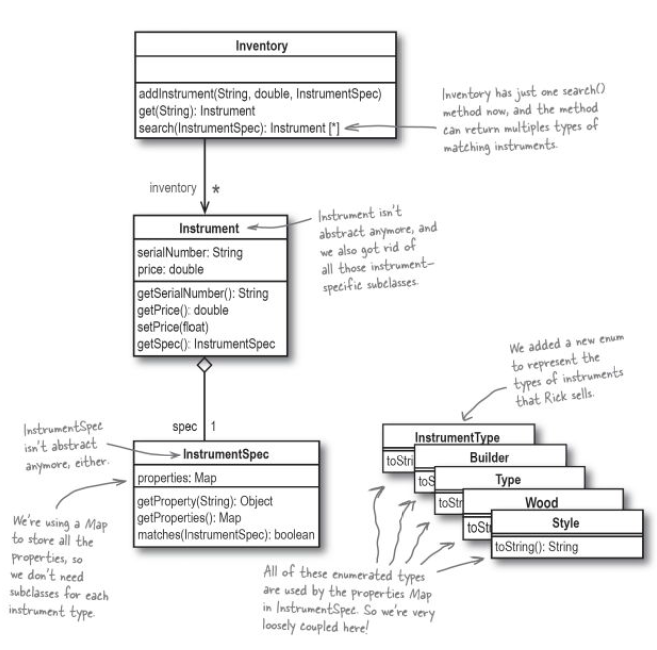
Classes are really about behavior!

Design is iterative… and you have to be willing to change your own designs, as well as those that you inherit from other programmers.

When you have a set of properties that vary across your objects, use a collection, like a Map, to store those properties dynamically.

A cohesive class does one thing really well and doesn’t try to do or be something else. The more cohesive your software is, the looser the coupling between classes.

**Cohesion:** Cohesion means the degree of connectivity among the elements of a single module, class, or object. The higher the cohesion of your software is, the more well-defined and related the responsibilities of each individual class in your application. Each class has a very specific set of closely related actions it performs.



**Chapter 6: Solving Really Big Problems**

What is a FEATURE?

A feature is just a high-level description of something a system needs to do. You usually get features from talking to your customers (or listening in on their conversations). A lot of times, you can take one feature, and come up with several different requirements that you can use to satisfy that feature. So figuring out a system’s features is a great way to start to get a handle on your requirements.

Use a feature or requirement list to capture the BIG THINGS that your system needs to do. Draw a use case diagram to show what your system IS without getting into unnecessary detail.

**Domain analysis** lets you check your designs, and still speak the customer’s language. Domain analysis is a process that we’re describing a problem using terms the customer will understand.   
“Scholar’s Corner”: **Domain Analysis:** the process of identifying, collecting, organizing, and representing the relevant information of a domain, based upon the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain.

**Design Patter:** A design pattern is just a way to design the solution for a particular type of problem. Once you’ve loaded your brain with a good working knowledge of patterns, you can then start to apply them to your new designs, and rework your old code when you find it’s degrading into an inflexible mess of jungle spaghetti code.

**Solving Big Problems:**

* Listen to the customer, and figure out what they want you to build.
* Put together a feature list in language the customer understands.
* Make sure your features are what the customer actually wants.
* Create blueprints of the system using use case diagrams (and use cases).
* Break the big system up into lots of smaller sections.
* Apply design patterns to the smaller sections of the system.
* Use basic OOA&D principles to design and code each smaller section.

**Bullet Points:**

* The best way to look at a big problem is to view it as a collection of smaller problems.
* Just like in small projects, start working on big projects by gathering features and requirements.
* Features are usually “big” things that a system does, but also can be used interchangeably with the term “requirements”.
* Commonality and variability give you points of comparison between a new system and things you already know about.
* Use cases are detail-oriented; use case diagrams are focused more on the big picture.
* Your use case diagram should account for all the features in your system.
* Domain analysis is representing a system in language that the customer will understand.
* An actor is anything that interacts with your system, but isn’t part of the system.

**Chapter 7: Architecture**

**Architecture** is your design structure, and highlights the most important parts of your app, and the relationships between those parts.   
**“Scholar’s Corner” architecture.** Architecture is the organizational structure of a system, including its decomposition into parts, their connectivity, interaction mechanisms, and the guiding principles and decisions that you use in the design of a system.

The things in your application that are really important are architecturally significant, and you should focus on them first.

Architecture is not just about the relationships between parts of your app; it’s also about figuring out which parts are the most important, so you can start building those parts first.

**The three Qs of architecture:**  
When you’re trying to figure out if something is architecturally significant, there are three questions you can ask:

1. Is this part of the essence of the system?  
   Is the feature really core to what a system actually is? Think about it this way: can you imagine the system without that feature? If not, then you’ve probably found a feature that is part of the essence of a system.
2. What the “heck” does it mean?  
   If you’re not sure what the description of a particular feature really means, it’s probably pretty important that you pay attention to that feature. Anytime you’re unsure about what something is, it could take lots of time, or create problems with the rest of the system. Spend time on these features early, rather than late.
3. How the “heck” do I do it?

Another place to focus your attention early on is on features that seem really hard to implement, or are totally new programming tasks for you. If you have no idea how you’re going to tackle a particular problem, you better spend some time up front looking at that feature, so it doesn’t create lots of problems down the road.

The essence of a system is what that system is at its most basic level.

The reason that these features are architecturally significant is that they all introduce RISK to your project. It doesn’t matter which one you start with—as long as you are working towards reducing the RISKs in succeeding. The point here is to REDUCE RISK, not to argue over which key feature you should start with first. You can start with ANY of these, as long as you’re focused on building what you’re supposed to be building.

**Bullet Points:**

* Architecture helps you turn all your diagrams, plans, and feature lists into a well-ordered application.
* The features in your system that are most important to the project are architecturally significant.
* Focus on features that are the essence of your system that you’re unsure about the meaning of, or unclear about how to implement first.
* Everything you do in the architectural stages of a project should reduce the risks of your project failing.
* If you don’t need all the detail of a use case, writing a scenario detailing how your software could be used can help you gather requirements quickly.
* When you ‘re not sure what a feature is, you should ask the customer, and then try and generalize the answers you get into a good understanding of the feature.
* Use commonality analysis to build software solutions that are flexible.
* Customers are a lot more interested in software that does what they want, and comes in on time, than they are in code that you think is really cool.

**Chapter 8: Design Principles**

Imitation is the sincerest form of not being stupid.

A **design principle** is a basic tool or technique that can be applied to designing or writing code to make that code more maintainable, flexible, or extensible.

**Principle #1: The Open-Closed Principle (OCP)**

Open-Closed Principle: Classes should be open for extension, and closed for modification.

The OCP is all about allowing change, but doing it without requiring you to modify existing code.   
Suppose you have a class with a particular behavior, and you’ve got that behavior coded up just the way you want it. Make sure that nobody can change your class’s code, and you’ve made that particular piece of behavior closed for modification. In other words, nobody can change the behavior, because you’ve locked it up in a class that you’re sure won’t change.   
But then suppose someone else comes along, and they just have to change that behavior. You really don’t want them messing with your perfect code, which works well in almost every situation… but you also want to make it possible for them to use your code, and extend it. So you let them subclass your class, and then they can override your method to work like they want it to. So even though they didn’t mess with your working code, you still left your class open for extension.

**Principle #2: The Don’t Repeat Yourself Principle (DRY)**

Avoid duplicate code by abstracting out things are common and placing those things in a single location.

**Principle #3: The Single Responsibility Principle (SRP)**

Every object in your system should have a single responsibility, and all the object’s services should be focused on carrying out that single responsibility. You’ve implemented the SRP correctly when each of your objects has only one reason to change. Cohesion is actually just another name for the SRP.

**Principle #4: The Liskov Substitution Principle (LSP)**

Subtypes must be substitutable for their base types. The LSP is all about well-designed inheritance. When you inherit from a base class, you must be able to substitute your subclass for that base class without things going terribly wrong. Otherwise, you’ve used inheritance incorrectly!

**Delegation** is when you hand over the responsibility for a particular task to another class or method. If you need to use functionality in another class, but you don’t want to change that functionality, consider using delegation instead of inheritance.

Use **composition** to assemble behaviors from other classes. Sometimes, delegation isn’t quite what you need; in delegation, the behavior of the object you’re delegating behavior to never change. But in some cases, you need to have more than one single behavior to choose from. In UML, we use a closed-in diamond at the end of a line to represent composition. **Composition** is most powerful when you want to use behavior defined in an interface, and then choose from a variety of implementations of that interface, at both compile time and run time.

**Composition** allows you to use behavior from a family of other classes, and to change that behavior at runtime.

When an object is composed of other objects, and the owning object is destroyed, the objects that are part of the composition go away, too. The behaviors in a composition do not exist outside of the composition itself.

What happens when you want all the benefits of composition—flexibility in choosing a behavior, and adhering to the LSP—but your composed objects need to exist outside of your main object? That’s where aggregation comes in. **Aggregation** is when one class is used as part of another class, but still exists outside of that other class.

**Aggregation versus Composition:** If the object does make sense existing on its own, then you should use aggregation; if not, then go with composition.

We started out this section talking about the LSP, and the basic idea that subclasses must be substitutable for their base classes. More importantly, though, now you have several ways to reuse behavior from other classes, beyond inheritance. Let’s take a quick look back at our options for reusing behavior from other classes, without resorting to subclassing: Delegation, Composition, Aggregation.

If you favor delegation, composition and aggregation over inheritance, your software will usually be more flexible and easier to maintain, extend, and reuse.

**Bullet Points:**

* The Open-Closed Principle keeps your software reusable, but still flexible, by keeping classes open for extension, but closed for modification.
* With classes doing one single thing through the Single Responsibility Principle, it’s even easier to apply the OCP to your code.
* When you’re trying to determine if a method is the responsibility of a class, ask yourself, *Is it this class’ job to do this particular thing?* If not, move the method to another class.
* Once you have your OO code nearly complete, be sure that you Don’t Repeat Yourself. You’ll avoid duplicate code, and ensure that each behavior in your code is in a single place.
* DRY applies to requirements as well as your code: you should have each feature and requirement in your software implemented in a single place.
* The Liskov Substitution Principle ensures that you use inheritance correctly, by requiring that subtypes be substitutable for their base types.
* When you find code that violates the LSP, consider using delegation, composition, or aggregation to use behavior from other classes without resorting to inheritance.
* If you need behavior from another class but don’t need to change or modify that behavior, you can simply delegate to that class to use the desired behavior.
* Composition lets you choose a behavior from a family of behaviors, often via several implementations of an interface.
* When you use composition, the composing object owns the behaviors it uses, and they stop existing as soon as the compositing object does.
* Aggregation allows you to use behaviors from another class without limiting the lifetime to those behaviors.
* Aggregated behaviors continue to exist even after the aggregating object is destroyed.

**Chapter 9: Iterating and Testing**

You write great software **iteratively**. Work on the big picture, and then iterate over pieces of the app until it’s complete.

When it comes to developing software, there is more than one way to iterate into specific parts of your application. You’ve got to take on smaller pieces of functionality, but there are two basic approaches to figuring out which small pieces to work on—and even what a “small piece” means in terms of your application.  
 (1) You can choose to focus on specific features of the application. This approach is all about taking one piece   
 of functionality that the customer wants, and working on that functionality until it’s complete.  
 (2) You can choose to focus on specific flows through the application. This approach takes a complete path   
 through the application, with a clear start and end, and implements that path in your code.

**Feature driven development:** When you are using feature driven development, you work on a single feature at a time, and then iterate, knocking off features one at a time until you’ve finished up the functionality of an application.

**Use case driven development:** With use case development, you work on completing a single scenario through a use case. Then you take another scenario and work through it, until all of the use case’s scenarios are complete. Then you iterate to the next use case, until all your use cases are working.

**Feature driven development is more granular; Use case driven development is more “big picture”.**

**FDD:   
\*** Works well when you have a lot of different features that don’t interconnect a whole lot.  
**\*** Allows you to show the customer working code faster.  
**\*** Is very functionality-driven. You’re not going to forget about any features.  
\* Works particularly well on systems with lots of disconnected pieces of functionality.

**UCDD:**  
 \* Works well when your app has lots of processes and scenarios rather than individual pieces of functionality.  
 \* Allows you to show the customer bigger pieces of functionality at each stage of development.  
 \* Is very user-centric.   
 \* Works particularly well on transactional systems, where the system is largely defined by lengthy processes.

Two ways to access properties is almost certainly going to mean duplicate code somewhere.

When you **program by contract**, you and your software’s users are agreeing that your software will behave in a certain way.

When you **program defensively**, you’re making sure the client gets a “safe” response, no matter what the client wants to happen.

**Bullet Points:**

* The first step in writing a good software is to make sure your application works like the customer expects and wants it to.
* Customers don’t usually care about diagrams and lists; they want to see your software actually do something.
* Use case driven development focuses on one scenario in a use case in your application at a time.
* In use case driven development, you focus on a single scenario at a time, but you also usually code all the scenarios in a single use case before moving on to any other scenarios, in other use cases.
* Feature driven development allows you to code a complete feature before moving on to anything else.
* You can choose to work on either big or small features in feature-driven development, as long as you take each feature one at a time.
* Software development is always iterative. You look at the big picture, and then iterate down to smaller pieces of functionality.
* You have to do analysis and design at each step of your development cycle, including when you start working on a new feature or use case.
* Tests allow you to make sure your software doesn’t have any bugs, and let you prove to your customer that your software works.
* A good test case only tests one specific piece of functionality.
* Test cases may involve only one, or several, methods in a single class, or may involve multiple classes.
* Test driven development is based on the idea that you write your tests first, and then develop software that passes those tests, the results is fully functional, working software.
* Programming by contract assumes both sides in a transaction understand what actions generate what behavior, and will abide by that contract.
* Methods usually return null or unchecked exceptions when errors occur in programming by contract environments.
* Defensive programming looks for things to go wrong, and tests extensively to avoid problem situations.
* Methods usually return “empty” objects or throw checked exceptions in defensive programming environments.

**Chapter 10: The OOA&D Lifecycle**

**Refactoring** is the process of modifying the structure of your code without modifying its behavior. Refactoring is done to increase the cleanness, flexibility and extensibility of your code, and usually is related to a specific improvement in your design.

