**Chapter 4: Analysis**

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.

Textual analysis tells you what to focus on, not just what classes you should create.

Pay attention to the nouns in your use case, even when they aren’t classes in your system.

Think about how the classes you do have can support the behavior your use case describes.

The verbs in your use case are (usually) the methods of the objects in your system.

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.

* Analysis helps you ensure that your software works in the real world context, and not just in a perfect environment.
* Use cases are meant to be understood by you, your managers, your customers, and other programmers.
* You should write your use cases in whatever format makes them most usable to you and the other people who are looking at them.
* A good use case precisely lays out what a system does, but does not indicate how the system accomplishes that task.
* Each use case should focus on only one customer goal. If you have multiple goals, you will need to write mutiple use cases.
* Class diagrams give you an easy way to show your system and its code constructs at a 10,000-foot view.
* The attributes in a class diagram usually map to the member variables of your classes.
* The operations in a class diagram usually represent the methods of your classes.
* Class diagrams leave lots of detail out, such as class constructors, some type information, and the purpose of operations on your classes.
* Textual analysis helps you translate a use case into code-level classes, attributes, and operations.
* The nouns of a use case are candidates for classes in your system, and the verbs are candidates for methods on your system’s classes.

**Chapter 5 (part 1) good design = flexible software**

Abstract classes are placeholders for actual implementation classes.

The abstract class defines behavior, and the subclasses implement that behavior.

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.

“What is an INTERFACE?”

Coding to an interface, rather than to an implementation, makes your 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 created yet.

“What is ENCAPSULATION?”

We’ve talked a fair bit about encapsulation already, in terms of preventing duplicate code. But 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***.

Most of the time, abstracting out common properties leads you to encapsulation.

“What is CHANGE?”

You already know that the one constant in software is CHANGE. Software that isn’t well-designed falls apart at the first sign of change, but great software can change easily.

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.

**5 (part 2) good design = flexible software**

Code once, look twice (or more!)

Keep looking over your designs when you run into problems. A decision

you made earlier may be what’s causing you headaches now.

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

By encapsulating what varies, you make your application more flexible, and easier to change.

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

You’ll remove lots of methods from your classes, and avoid having to change your code when new properties are added to your app.

Most good designs come from analysis of bad designs.

Never be afraid to make mistakes and then change things around.

A cohesive class does one thing really well and does not try to do or be

something else.

cohesion. Cohesion measures 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.

Analysis and Design

Well-designed software is easy to change and extend.

Use basic OO principles like encapsulation and inheritance to make your software more flexible.

If a design isn’t flexible, then CHANGE IT! Never settle on bad design, even if

it’s your bad design that has to change.

Make sure each of your classes is cohesive: each of your classes should

focus on doing ONE THING really well.

Always strive for higher cohesion as you move through your software’s design

life cycle.

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.

Classes are about behavior and functionality.

**Chapter 6: solving really big problems**

You solve big problems the same way you solve small problems.

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.

You can solve a big problem by breaking it into lots of functional pieces, and then working on each of those pieces individually.

By encapsulating what varies, you make your application more flexible, and easier to change.

Coding to an interface, rather than to an implementation, makes your software easier to extend.

The best way to get good requirements is to understand what a system is supposed to do.

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

Great software is easy to change and extend, and does what the customer wants it to do.

What is the system like?

One way you can find out more about a system is to figure out what the system is like. In other words, are there some things that you do know about that the system functions or behaves like? This is called **commonality**... what things are similar?

What is the system not like?

Another great way to find out what a system should do is to figure out what it’s not like. This helps you determine what you don’t need to worry about in your system. This is called **variability**...

what things are different?

Get features from the customer, and then figure out the requirements you need to implement those features.

Always defer details as long as you can.

You still need to know what your system is supposed to do... but you need a BIG-PICTURE view.

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. 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.

Don’t forget who your customer really is.

Domain analysis helps you avoid building parts of a system that aren’t your job to build.

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.

Chapter 7: architecture

Bringing Order to Chaos

Architecture is your design structure, and highlights the most important parts of your app, and the relationships between those parts.

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.

The three Qs of architecture

1. Is it part of the essence of the system?
2. What the heck does it mean?
3. How the “heck” do I do it?

The essence of a system is what that

system is at its most basic level.

Focus on one feature at a time to reduce risk in your project.

Don’t get distracted with features that won’t help reduce risk.

Good design will always reduce risk.

Sometimes the best way to write great code is to hold off on writing code as long as you can.

You can use these three basic steps anytime you’re unsure about what a feature means, and how you need to implement that feature in your system.

1. Ask the customer
2. Commonality analysis
3. Implementation plan

When you find more things that are different about a feature than things that are the same, there may not be a good, generic solution.

Customers don’t pay you for great code, they pay you for great software.

* 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

Originality is Overrated

*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.*

Using proven OO design principles results in more maintainable, flexible, and extensible software.

Principle #1:

The Open-Closed Principle (OCP)

*Classes should be open for extension, and closed for modification.*

Principle #2:

The Don’t Repeat Yourself Principle (DRY)

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

DRY is about having each piece of information and behavior in your

system in a single, sensible place.

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 Single Responsibility

Principle correctly when each of your objects has only one reason to change

Spotting multiple responsibilities

Most of the time, you can spot classes that aren’t using the

SRP with a simple test:

1. On a sheet of paper, write down a bunch of lines like this: The [blank]

[blanks] itself. You should have a line like this for every method in the class you’re testing for the SRP.

2. In the first blank of each line, write down the class name; in the second

blank, write down one of the methods in the class. Do this for each

method in the class.

3. Read each line out loud (you may have to add a letter or word to get it to

read normally). Does what you just said make any sense? Does your class

really have the responsibility that the method indicates it does?

**1**

**2**

If what you’ve just said doesn’t make sense, then you’re probably violating the SRP with

that method. The method might belong on a different class... think about moving it.

Contestant #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!

“Subtypes must be substitutable for their base types”

It’s hard to understand code that misuses inheritance

Delegate functionality to another class

***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

When to use composition

When we reference a whole family of behaviors like in the Unit class, we’re using **composition**. The **Unit**’s weapons property is ***composed*** of a particular **Weapon** implementation’s behavior.

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.*

Pizza is actually a great example of composition: it’s composed of different

ingredients, but you can swap out different ingredients without affecting

the overall pizza slice.

When the pizza is gone, so are the ingredients...

There’s one important point we haven’t mentioned so far about composition. 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*.

In composition, the object composed of other behaviors owns those behaviors. When the object is destroyed, so are all of its behaviors.

The behaviors in a composition do not exist outside of the composition itself.

***Aggregation: composition, without the abrupt ending***

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

It’s easy to get confused about when you should use composition, and when you should use aggregation. The easiest way to figure this out is to ask yourself, *Does the object whose behavior I want to use exist outside of the object that uses its behavior?*

If the object does make sense existing on its own, then you should use aggregation; if not, then go with composition. But be careful! Sometimes the slightest change in the usage of your objects can make all the difference.

Inheritance is just one option

Delegation

***Delegate*** behavior to another class when you don’t want to change the behavior, but it’s not your object’s responsibility to implement that behavior on its own.

Composition

You can reuse behavior from one or more classes, and in particular from a family of classes, with ***composition***. Your object completely owns the composed objects, and they do not exist outside of their usage in your object.

Aggregation

When you want the benefits of composition, but you’re using behavior from an object that does exist outside of your object, use ***aggregation***.

All three of these OO techniques allow you to reuse behavior without violating the LSP

If you favor delegation, composition, and aggregation over inheritance, your software will

usually be more flexible, and easier to maintain, extend, and reuse.

Chapter 9 iterating and testing

The Software is Still for the Customer

You write great software iteratively.

Work on the big picture, and then iterate over pieces of the app until it’s complete.

Iterating deeper: two basic choices

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.

***Feature driven development***

*...is when you pick a specific feature in your app, and plan, analyze, and develop that feature to completion.*

You can also 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.

***Use case driven development***

*...is when you pick a scenario through a use case, and write code to support that complete scenario through the use case.*

Both approaches to iterating are driven by good requirements.

Because requirements come from the customer, both approaches focus on delivering what the customer wants.

You should test your software for every possible usage you can think of. Be creative!

Don’t forget to test for incorrect usage of the software, too. You’ll catch errors early, and make your customers very happy.

Test driven development focuses on getting the behavior of your classes right.

Good software is built iteratively. Analyze, design, and then iterate again, working on smaller and smaller parts of your app.

Each time you iterate, reevaluate your design

decisions, and don’t be afraid to CHANGE

something if it makes sense for your design.

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

When you are programming by contract, you’re working with client

code to agree on how you’ll handle problem situations.

When you’re programming defensively, you’re making sure the client

gets a “safe” response, no matter what the client wants to have happen.

Development Approaches

***Use case driven development*** takes a single use case in your system, and focuses on completing the code to implement that entire use case, including all of its scenarios, before moving on to anything else in the application.

***Feature driven development*** focuses on a single feature, and codes all the behavior of that feature, before moving on to anything else in the application.

***Test driven development*** writes test scenarios for a piece of functionality before writing the code for that functionality. Then you write software to pass all the tests.

Programming Practices

***Programming by contract*** sets up an agreement about how your software

behaves that you and users of your software agree to abide by.

***Defensive programming*** doesn’t trust other software, and does extensive error and data checking to ensure the other software doesn’t give you bad or unsafe information.

Chapter 10 the ooa&d lifecycle

Putting It All Together

* Feature List
* Use Case Diagrams
* Break Up the Problem
* Requirements
* Domain Analysis
* Preliminary Design
* Implementation
* Delivery

Your feature lists are all about understanding what your software is supposed to do.

Your use case diagrams let you start thinking about how your software will be used, without getting into a bunch of unnecessary details.

Use cases reflect usage, features reflect functionality.

The features in your system reflect your system’s functionality. Your system must do those things in order for the use cases to actually work, even though the functionality isn’t always an explicit part of any particular use case.

The features in your system are what the system does, and are not always reflected in your use cases, which show how the system is used.

Features and use cases work together, but they are not the same thing.

Your design decisions should be based on how

your system will be used, as well as good OO

principles.

You should only expose clients of your code to the classes that they NEED to interact with.

Classes that the clients don’t interact with can be changed with minimal client code being affected.

It’s your job to balance making sure the customer gets the functionality they want with making sure your code stays flexible and well-designed.

Sometimes the best code for a particular problem has

already been written. Don’t get hung up on writing code yourself if someone already has a working solution.

OOA&D is about having lots of options. There is never one right way to solve a problem, so the more options you have, the better chance you’ll find a good solution to every problem.

Leftovers

#1. IS-A and HAS-A

IS-A refers to inheritance

HAS-A refers to composition or aggregation.

Use inheritance when one object behaves like another, rather than just when the IS-A relationship applies.

#2. Use case formats

#3. Anti patterns

***Design Patterns***

*Design patterns are proven solutions to particular types of problems, and help us structure our own applications in ways that are easier to understand, more maintainable, and more flexible.*

Design patterns help you recognize and implement GOOD solutions to common problems.

***Anti Patterns***

*Anti-patterns are the reverse of design patterns: they are common BAD solutions to problems. These dangerous pitfalls should be recognized and avoided*

Anti patterns are about recognizing and avoiding BAD solutions to common problems.

#4. CRC cards

CRC stands for Class, Responsibility, Collaborator. These cards are used to take a class and figure out what its responsibility should be, and what other classes it collaborates with.

#5. Metrics

errors found in code

defect density = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

total lines of code / 1000

This number gives you some idea of how well you’re writing your code. If it’s high, look for design problems or inefficiencies

**abstractness metric**

A = N *a* / N*c*

Na is the number of abstract classes in a particular package or module of your software (this includes interfaces).

Nc is the total number of classes in the same package or module.

This number will always be between 0 and 1. Higher numbers mean more abstraction is being used, lower numbers represent less abstraction.

Robert Martin’s book called “Agile Software Development” has a lot more of these OOrelated metrics.

#6. Sequence diagrams

A sequence diagram is just what it sounds like: a visual way to show the things that happen in a particular interaction between an actor and your system.

#7. State diagrams

UML also contains a diagram called a **state machine diagram** or **statechart diagram**, which is usually just referred to as a **state diagram**. This diagram describes a part of your system by showing its various states, and the actions that cause that state to change. These diagrams are great for describing complex behaviors visually.

#8. Unit testing

#9. Coding standards and readable code

Reading source code should be a lot like reading a book. You should be able to tell what’s going on, and even if you have a few questions, it shouldn’t be too hard to figure out the answers to those questions if you just keep reading. Good developers and designers should be willing to spend a little extra time writing readable code, because it improves the ability to maintain and reuse that code.

Writing readable code makes that code easier to maintain and reuse, for you and other developers.

#10. Refactoring

**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.

Refactoring changes the internal structure of your code WITHOUT affecting your code’s behavior.

Speaking the Language of OO

Inheritence lets you build classes based on other classes, and avoid duplicating and repeating code.

**Polymorphism** is closely related to inheritance. When one class inherits from another, then polymorphism allows a **subclass** to **stand in** for the **superclass**.

Encapsulation is when you protect information in your code from being used incorrectly.

encapsulation. The process of enclosing programming

elements inside larger, more abstract entities. Also known

as information hiding, or separation of concerns.

Encapsulation separates your data from your

app’s behavior.

Then you can control how each part is used by the rest of your application.