Software Engineering Kitchen - Part 1

1. Clean Code

- Boy Scout Rule: Leave the compound cleaner than you found it.

Meaningful Names

- Use Intention Revealing Names
- Avoid abrevations
- Avoid disinformation
  - `Customer[] customerList; Table theTable;`
  - `Customer[] customers; Table customers;`
- Avoid variable names like lowercase L or uppercase O
- Make meaningful distinctions
  - public static void copyChars(char a1[], char a2[])
  - public static void copyChars(char source[], char destination[])
- Use Pronounceable Names
  - If you can't pronounce it, you can't discuss it without sounding like an idiot.
- Use searchable names
  - Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.

  - MAX_CLASSES_PER_STUDENT vs. 7
  - e is a poor choice for any variable
- Avoid Encodings
  - private String m_dsc;
  - private String description;
- Avoid mental mapping
  - Readers shouldn't have to mentally translate your names into other names they already know.

Class

- Classes and objects should have noun or noun phrase names
  - **Customer, WikiPage, Account, and AddressParser.**
- Use names from the problem domain
  - **Customer,Bookmark,Document,Measurement**
- A class should have a single responsibility
- The class name should convey its purpose
- Use compound names if necessary
  - **Example: TextStyle, LevelEditor**
- Use no more than two or three words
- Avoid words like **Manager,Processor,Data,Objector Info** in the name of a class.
- A classname should not be a verb.

Method

- The method name should describe what the method does
  - `place(); perform();`
  - `placeOrder(); addToShoppingCart();`
- Avoid ambiguous method name
  - `bucket.empty()`
  - `bucket.isEmpty(); bucket.makeEmpty();`

- Use positive concepts to name your functions
  - `isConnected()`
  - `isUnconnected()`

Method Names

- Method names should describe everything that the routine does.
  - sharpenAndSaveImage()
- If this makes your function names too long, then this may indicate that they are performing too many tasks.
- Avoid abbreviations
- Use correct terms for methods that form natural pairs
  - openWindow() and closeWindow()
- Similar things should look similar
  - `map.size(); list.size(); vector.length();`
  - `map.size();list.size(); vector.size();`
- Don't be cute
  - `throwHandGrenade();` `eatMyShorts();` `whack();`
- Don't use slang
- Methods should be small
  - Just a few lines (2 - 10)
  - They should tell a story
  - Signs it is too long
    - *Scrolling required*
    - *Multiple conditions*
    - *Naming issues*
    - *Whitespace and comments*
    - *Hard to digest*
  - Simple methods can be longer
    - Complex methods should be short

- Method should do one and do it well
- Method should not have side effects
- Command Query Seperation
  - *Methods should either do something or answer something, but not both.*
  - *Either your method should change the state of an object, or it should return some information about that object*
- Encapsulate boolean expression
- Avoid deep nesting: extract a method
- Avoid deep nesting: return early
- Avoid deep nesting: fail fast

Method Parameters

- Parameter names and types are not always visible when in the client
  - **`Document.remove(Tag tag);` → `Document.removeTag(Tag tag);`**
- The Boolean Parameter Trap
  - Boolean parameter often lead to unreadable code
    - It's almost invariably. a mistake to add a boolean parameter to an existing function.
    - `widget.repaint(false);` It is not clear what this method does
    - widget.repaint(); widget.repaintWithoutErasing();
- Avoid boolean parameter
- Use Interface Type

Comment

- Purpose of a comment is to explain code that does not explain itself
- Don't use a comment when you can use a method or a variable
- General Rules:
  - *Prefer expressive code over comments.*
  - *Use comments when code alone can't be sufficient.*
- Good Comments

- Legal Comments
- Informative Comments
- Clarification
- TODO Comments

- Bad Comments
  - Redundant comments
  - Intent comments
  - Divider comments
  - Mumbling
  - Redundant comments
  - Journal comments
  - Noise comments

2. Design

- Anemic domain model
  - Classes in the model have no business logic
  - Disadvantages:
- Rich domain model
- Domain Model Patterns
  - Entities
    - A class with identity
    - Mutable
    - State may change after instantiation
    - The entity has an lifecycle
    - Changing attributes doesn't change which one we're talking about
      - Identity remains constant throughout its lifetime
  - Value Objects
    - Has no identity
      - Identity is based on composition of its values
      - Value objects tell something about another object
      - Technically, value objects may have IDs using some database persistence strategies.
        - But they have no identity in the domain
    - Immutable
      - State cannot be changed after instantiation
      - Once created, a value object can never be changed
    - Other Characteristics
      - Attribute-based equality
        - 2 value objects are equal if they have the same attribute values
      - Cohesive
        - Encapsulate cohesive attributes

- Behavior rich
  - Value objects should expose expressive domain-oriented behavior
- Combinable
  - Can often be combined to create new values
- Self-validating

  Value objects should never be in an invalid state
- Testable

  Value objects are easy to test because of these qualities
  - **Immutable:** We don't need mocks to verify side effects
  - **Cohesion:** We can test the concept in isolation
  - **Combinability:** Allows to express the relationship between 2 value objects

- Reason to make class immutable
  - Less prone to errors
  - Easier to share
  - Thread safe
  - Combinable
  - Self-validating
  - Testable
- When to use value objects?
  - Representing a descriptive identity-less concept
  - Enhancing explicitness
  - Static factory methods
  - Collection avoidance
    - Be careful with collections of value objects
      - `Customer → Phone Number (X)`
      - `Customer → homeNumber, mobileNumber, workNumber`
- **Note: Always prefer value objects over entities in your domain model**

- Domain Services
  - Sometimes behavior does not belong to an entity or value object. But it is still an important domain concept. Use domain services.
  - Interface is defined in terms of other domain objects
  - Characteristics
    - Stateless
      - Have no attributes
    - Represent Behavior

      No identity
    - Often orchestrate multiple domain objects
- Domain Events
  - Classes that represent important events in the problem domain that have already happened
    - Immutable
  - Events are raised and event handlers handle them.
  - Some handlers live in the domain, and some live in the service layer.
- In Design, what does the principle of high cohesion, loose coupling mean?

- SOLID principles
  - **Single Responsibility Principle:** A class should have only one reason to change
  - **Open Closed Principle:** Software entities (classes, module, functions, etc) should be open for extension, but closed for modification.
  - **Liskov Substitution Principle:** Subtypes must be substitutable for their base (super) types.
  - **Interface Segregation Principle:** Client should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies.
  - **Dependency Inversion Principle:** Abstractions should not depend upon details. Details should depend upon abstractions.

- What does the Principle of High Cohesion, Loose Coupling mean?

3. Architecture

- Architecture Artifacts

  Component Diagram, Sequence Diagram, Deployment Diagram

- Design Principles
  - Keep it Simple
  - Keep it Flexible
    - More flexibility leads to more complexity
    - Everything changes
      - Business
      - Technical
  - Loose Coupling
  - Separation of Concern
    - Separate technology from business
    - Separate changing from non-changing
    - Separate business process from application logic
    - Separate implementation from application logic
    - Separate stable things from changing things
  - Information Hiding
  - Principle of modularity
- Domain Driven Design
  - An approach to software development where the focus is on the core Domain.
    - We create a domain model to communicate the domain
    - Everything we do (discussions, design, coding, testing, documenting, etc.) is based on the domain model.
  - Domain modelling
    - An abstraction of reality designed to manage complexity of specific business cases

Extracts domain essential elements

- Relevant to a specific use
- Layers of abstractions representing selected aspects of the domain
- Contains concepts of importance and their relationships
- Shows the important concepts of the domain. Example: Rich Domain Model
- More complexity -> More modeling

- Principles of DDD
  - Use one common language to describe the concepts of a domain
    - Ubiquitous Language
      - Language used by the team to capture the concepts and terms of a specific core business domain.
  - Create a domain model that shows the important concepts of the domain
  - Let the software be a reflection of the real world domain
  - Create small contexts in which a domain model is valid (Bounded Context)
- Advantages of Domain Model
  - Improves understanding
  - Validates understanding
  - Improves communication
  - Shared glossary
  - Improves discovery
- Big Ball of Mud
  - A change in one subdomain can effect all other subdomains
- Context
  - A specific domain term may have a different definition in different context.
  - Bounded Context: Each bounded context has its own model
- Why DDD?

- The hardest part of software development is to understand what the business wants and how the business works

- Architecture Styles
  - Layering (Client, Presentation, Service, Business Logic, Integration)
    - Presentation
    - Service
    - Business (Domain)
    - Data Access (Persistence)
    - Integration
  - Domain Driven Design
  - Object Oriented
    - Advantages:
      - High Cohesion, Loose Coupling
      - Encapsulation: Data Hiding
      - Flexibility: Polymorphism
      - Reuse? Inheritance
  - Component-based
    - Decompose the domain in functional components.
      - What is Component?
        - There's no definition
        - What we agree upon
          - A component has an interface.
            - Interface:
              - The interface tells what you can do (but not how)
          - A component is encapsulated
            - Encapsulation:
              - The implementation details are hidden
        - Plug-and-play
        - A component can be single-unit-of-deployment

- Advantages of Component
  - High Cohesion, Loose coupling
  - Flexibility
  - Reuse?
- Internals of component
  - Interface
  - Façade/ Service

- Service Oriented Architecture
  - Defines a way to make software components reusable and interoperable via service interfaces.
  - Multiple Services communicating through Enterprise Service Bus (ESB)
- Microservices
  - Small Independent services
    - Simple and lightweight
    - Runs in an independent process
    - Technology agnostic
    - Decoupled
  - Orchestration
    - Single Brain
    - Single Point of Failure
  - Choreography
    - No Central Brain
    - Independent services

- Artifacts produced from Domain Modelling
  - UML Class Diagram
  - Sequence Diagram
- What is the difference between Loose Coupling versus Tight Coupling? Give example
  -

Difference Table

Loose Coupling	Tight Coupling
Improves the test ability	Not good at the test-ability
Helps us follow the GOF principle of program to interfaces, not implementations	Does not provide the concept of interface
Easier to swap other pieces of code/modules/objects/components in loose coupling	In Tight coupling, it is not easy to swap the codes between two classes
Loose coupling is highly changeable	Tight coupling does not have the changing capability.
Less interdependency	**More interdependency**
Less Coordination	**More Coordination**
Less Information flow	**More Information flow**

- Source Code Example

4. Analysis

- Agile Requirements
  - Text or statement
  - Context Diagram
    - Understand the big picture
  - Personas
    - Identify the users
  - Use case Diagram
    - Understand the goal of users
  - User story map
    - Split user stories
    - Write acceptance criteria
    - Find and understand the user stories
  - Artifacts
    - Activity Diagram, State Machine Diagram, Class Diagram, Sequence Diagram, Business Rules, Non-functional Requirement, User Scenario's UI Screen
- Analysis Model
  - Structure of the domain model
    - Objects: Class Diagram
  - Behavior of the domain model
    - Sequence diagram
      - Toggle
    - Communication Diagram (not much used)
- OO concepts
  - Object
    - An object is a concept or a thing in the problem domain
    - An object has Data and Behavior
    -

Object: Instance of a class with its own identity, state and behavior

- **Class:** abstract definition of an object
- Inheritance
  - Is-a relationship between generic and specific classes
- Polymorphism
  - Polymorphism = many forms
  - Objects behave differently on the same command

- Noun/Verb Analysis
  - Noun becomes classes or attributes
  - Verb becomes method
- UML class diagram element
  - Describes the structure of a system by showing the classes, their attributes, and the relationships
- Domain model
  - The domain model shows the entities, attributes and relationships of the problem domain
- Interaction Diagram
  - Sequence Diagram
    - Shows the sequence of communication
    - Shows how the different objects interact with each other.
- **Check-in Book Scenario (for reference)**
  - Actor scans the library card of the library member with the scanner
  - The system shows library member information (name, address, phone, checked-out books, reserved books, payments due)
  - Actor gives the command to check-in books
  - System gives the message to scan the book
  - The actor scans the book with the scanner
  - The system sends an email to the first library member who reserved this book.
  -

The system shows the library member information with the updated checked-out books list and the updated payments due

- Repeat step 5-7 till all books are checked in

5. Requirements-2

User Story

- As a <**role**> I want <**some activity**> so that <**business value**>
- **A good user story (INVEST)**
  - Independent
    - A story can be developed and tested on its own
    - Dependent stories are difficult to implement in isolation
    - Dependent stories are hard to prioritize and to estimate
  - Negotiable
    - Story cards are reminders
      - Not contract
    - Details are found in the conversation
    - User stories are placeholders for requirements
      - To be discussed
      - To be changed if needed
      - To be developed
      - To be tested
      - To be accepted
  - Valuable
    - A user story gives some business value
    - A user story is a vertical slice (of the cake)
      - Not a horizontal slice
  - Estimable
    - If the team is unable to estimate a user story, it generally indicates that the story is too large or uncertain.
  - Small
    - User stories should be implemented, tested and accepted within a sprint (few days)

- Small stories are simpler to
  - Understand
  - Estimate
  - Implement
  - Test
- Small stories give faster feedback
- Testable
  - If a story does not appear to be testable, then the story is probably ill-formed, overly complex, or perhaps dependent on other stories in the backlog
  - It should be easy to write acceptance criteria for a user story.
- User stories are high level descriptions of the requirement to be built
- User stories are a promise for a conversations
  - Details come up during the conversation
- **Flat Backlog Trap:** Bag of context-free mulch

User Story Mapping

- Approach to visually organize and prioritize what you need to do (user stories)
- Visualize the workflow or value chain
- Show the relationships of larger stories to their child stories
- Help confirm the completeness of your backlog
- Provide a useful context for prioritization
- Plan releases in complete and valuable slices of functionality.
- How to deal with time pressure?
  - Minumum Viable Product
- Good user stories
  - Tell the stories that generates energy, interest, and vision in the listener's mind

Advantages of story mapping

- Improves collaboration
- Improves the backlog
- Allows for scenarios walkthroughs
- Easy to create
- Visualizes the release backlog
- Validation with customers
- Shows the big picture
- Helps with prioritization

- What do I do if a user story is too long?
  - User stories should be small
    - Should fit within an iteration (few days)
    - Small stories are simpler to
      - Understand
      - Estimate
      - Implement
      - Test
    - Small stories give faster feedback
  - Split large user stories into smaller user stories.
- Splitting User Story Pattern
  - Workflow steps
    - Create a story of the basic workflow and then create new stories for enhanced workflows.
  - Operations
    - Make a user story for every operation
  - CRUD

    Create, Read, Update and Delete
  - Business rule variation

- Create a new story for every variation in business rules
- Simple/complex

  Capture the simplest version that works in a story and make a new story for every variation and complexity
- Variation in data

  Create one story that uses one kind of data and create other stories that uses other kinds of data
- Effort
- Interface variation
- Defer quality

Acceptance Criteria

- Acceptance criteria define when the user story is done

- They contain the details of the behavior of the system
- They serve to record the decisions made in the conversation
  - Toggle
- They are the input for testing the user story
- **Good User Story Acceptance Criteria**
  - The product owner is the primary owner of the tests
  - The acceptance criteria are black box
    - **No design or implementation in acceptance criteria**
  - They test all possible scenarios
  - Write at least one acceptance criteria for every user story
  - They are high level
    - You don’t need to write down all details

Scenario’s

- Sequential interaction between the actor(s) and the system

- Only used for interaction with a system
- Main Scenario
  - The actor reached the goal
  - Happy day scenario
  - **Example (Main Scenario)**
    - The customer enters the ATM card
    - The ATM asks for the PIN code
    - The customer enters the PIN code
    - The ATM asks for making a choice
      - 1. Withdraw cash
      - 1. Request account balance
    - The customer chooses “Withdraw cash”
    - The ATM asks for the amount. It shows the options: 10 euro, 25 euro, 50 euro, 100 euro, 150 euro, 200 euro or ‘enter amount’
    - The customer chooses a fixed amount
    - The ATM shows that the transaction is in progress
    - The ATM sends the transaction to the bank system
    - The bank system notifies the ATM that the transaction is approved
    - The ATM asks if the customer wants a receipt
    - The customer selects “yes”
    - The ATM ejects the bank card
    - The customer takes the bank card
    - The ATM ejects the cash
    - The customer takes the cash
    - The ATM ejects the receipt
    - The customer takes the receipt

Alternative Scenario

- Alternative may rejoin the main scenario
- Alternative may end the use case in failure
- **Example (Alternative Scenario)**
  - Request account balance
    - The customer chooses “request account balance”
    - The use case “Request account balance” is executed
  - Enter amount
    - The customer chooses “enter amount”
    - The ATM asks for the desired amount
    - The customer enters the desired amount
    - Continue at point 8 of the main scenario
  - Inserted card is not a correct bank card
    - The ATM shows the message that the inserted card is not a correct bank card.
    - The ATM ejects the card
    - The customer takes the card
  - Pin code is not correct
    - The ATM shows the message that the PIN code is not correct and gives the customer the possibility to try again
    - The customer enters the PIN code again
    - Continu at point 4 of the main scenario
- **Note: Keep scenario descriptions short**
  - The customer adds the product to the shoppingcart
  - The customer selects the product that he/she wants to order.
  - The customer commands the system to add this product to the shoppingcart.
- Use cases and UI
  - Do not use UI details in use cases
    -

User interfaces is design
- With UI is the focus on the system
  - With use cases is the focus on the user and its goal
- Describe only interactions
  - The system is a black box
    - The system shows the user that the information is saved
    - The system saves the information
    - The system shows the requested information
    - The system retrieves the information from the database
    - The system shows a validation error message
    - The system validates the entered data
- Do not describe technical aspects
  - The system is a black box.
  - Also business people must be able to understand use cases
    - The system shows the message that the information is saved
    - The system sends a message to the order system
    - The system saves the information in the database
    - The system sends a JMS message to the order system
- Short scenario’s
  - Maximal 15 to 20 scenario steps

Non-functional Requirements

- Qualities of the system
- Performance, Scalability, Robustness, ...
- Qualities noticeable at runtime
  - Performance
  - Security
  - Availability
  - Functionality

- Usability
- Qualities not noticeable at runtime
  - Modifiability
  - Portability
  - Reusability
  - Integrability
  - Testability
- FURPS Model
  - Functionality
    - evaluate the feature set and capabilities of the program, the generality of the functions delivered and the security of the overall system
  - Usability
    - consider human factors, overall aesthetics, consistency, and documentation
  - Reliability
    - measure the frequency and severity of failure, the accuracy of outputs, the ability to recover from failure, and the predictability
  - Performance
    - measure the processing speed, response time, resource consumption, throughput and efficiency
  - Supportability
    - measure the maintainability, testability, configurability and ease of installation
- Business Rules
  - A Business Rule is a statement which defines or constrains some aspect of the business.
  - A business rule is valid even without a system
    - Customers can take maximal 8 books home
  - Business rules are an enterprise level asset

  - Write business rules in a separate document
  - **Example (Business rules):**
    - We don't charge tax on shipping charges
    - If payment is not received within 30 days, we send a payment reminder
    - All coming orders > € 500 must be reviewed by a manager.

Describe UI

- Get an early feedback from the users of the system about the proposed and chosen concepts.
- Informal, interactive
- An early review of the user interface
- Easy to make, cheap to change
- Avoid the "yes, but..." syndrome

Agile requirements best practices

- Focus on telling stories, not writing stories
- Take enough time for requirements
- Requirements are never done
- Get feedback fast and often
- Requirements may change
  - Discuss the impact with the stakeholders
- Prioritize requirements
  - Find the minimal viable solution
- Focus on the users needs instead of the solution
- Requirements are discovered, not gathered
- Do not specify all requirements upfront
  - Work iterative

- Know the goal of the diagram or text before you start
  - Not more diagrams than needed
  - Keep it simple
  - Not too much information on 1 diagram