Implementing maintainable software

By Bytecraft_

Agenda

- Introduction
- Agile
- Characteristics of maintainable code
- Techniques

Who?

- Bytecraft_
  - Software craftsmanship
    - Raising the bar
    - https://manifesto.softwarecraftsmanship.org/
  - https://www.bytecraft.fi/ (sorry finnish only)
- Antti Ahonen

Some Agile Principles

- Satisfy the customer through **early and continuous delivery** of valuable software.
- **Working software** is the primary measure of progress.
- **Deliver** working software **frequently**, from a couple of weeks to a couple of months.
- Welcome **changing requirements**, even late in development.
- Business people and developers must **work together daily** throughout the project.
- Continuous **attention to technical excellence** and good design enhances agility.

http://www.agilemanifesto.org/

Characteristic of maintainable software

- Can be changed easily
  - Malleable
- High software internal quality
- Built-in quality, by the developer
  - Constant work, in every decision
  - Professional pride
- Present at every abstraction level
  - Architecture, line of code
- Documents itself

https://www.shutterstock.com/fi/image-photo/colorful-children-building-bricks-1142522201

Techniques

High-level guide

- Extreme programming
  - Small Releases
  - Simple Design
  - Testing
  - Refactoring
  - Pair Programming
  - Collective Ownership
  - Continuous Integration
  - Coding Standard

=> Actually normal programming nowadays

https://www.agilealliance.org/glossary/xp

Simple design

- **YAGNI** https://martinfowler.com/bliki/Yagni.html
  - Avoid generalization until you actually need it
  - Even if you know the whole product vision and the stories, only build the software around the current sprint stories
- **KISS**

→ Both are there to remind you to avoid over-engineering

https://effectivesoftwaredesign.com/2013/08/05/simplicity-in-software-design-kiss-yagni-and-occams-razor/

KISS

```
const myCondition = true;
const conditionAction = () => console.log("condition")
const myOtherCondition = false;
const otherConditionAction = () => console.log("otherCondition");
const alwaysAction = () => console.log("conditions evalutuated");

const complex = () => {
 //CLEVER CODE? maybe not, but complicated at least
 (((myCondition && conditionAction()) || (myOtherCondition && otherConditionAction())) || true) && alwaysAction();
}

const simple = () => {
 //SIMPLE
 if (myCondition) {
   conditionAction();
 }
 if (myOtherCondition) {
   otherConditionAction();
 }
 alwaysAction();
}
```

Naming

- Aim to **reveal intent**

```
const d = 5; //days elapsed since beginning
//VS
const daysElapsedSinceBeginning = 5;
```

- Name for concepts, not language types

```
const userList = []
//VS
const users = []
```

- Pick one word, use consistently for technical & domain concepts
  - Get, fetch, query, search
  - Client, customer, buyer, patron, consumer, shopper

https://dzone.com/articles/naming-conventions-from-uncle-bobs-clean-code-phil

Comments

- First and foremost, explain the **intent** with code
- Comments can help, but they are no substitute for good code
- Inherent problem of comments? → Going **stale** much easier than code
- Even though most of the time comments are a code smell, there do exist **good comments**
- **Best comments** are **structured comments** that generate **living documentation**:
  - Language mechanisms (annotations, decorators etc) used for commenting, that generate runnable documentation from code → *Swagger*
    - Or documentation that can be used to generate code (*OpenAPI*)
  - Testing frameworks : Injecting comments as part of method names, structure → reports
    - Or test files that are text documentation, but can be run → *Cucumber*
  - With shared/public libraries, use with thought involved: *JavaDoc, JSDoc, PyDoc* etc..

Examples of bad comments

- Redundant

```
/**
* Fetch customer by customer identifier
* @param customerIdentifier    identifier for customer
* @return                      customer for the idenfifier
* @see                         Customer
*/
@Override
Customer fetchCustomerBy(Customer.Identifier  customerIdentifier)
```

- Clearing intention, that could be done with code

```
//customer has wanted order template
if (!customer.orderTemplate.empty &&
  customer.orderTemplate.get().identifier.valueLong ==
templateId.valueLong)
//VS
if (customer.hasOrderTemplate(templateId))
```

- Intent pt2, magic numbers

```
//search for cat in animal category
search("cat", 1)
//search for bmw in vehicle category
search("bmw", 5)
//VS
const ANIMAL_CATEGORY = 1
const VEHICLE_CATEGORY = 5
search("cat", ANIMAL_CATEGORY)
search("bmw", VEHICLE_CATEGORY)
```

- ‘Read more about bad / good comments: https://blog.usejournal.com/stop-writing-code-comments-28fef5272752

Functions

- **Small**
- **Do only one thing**
- Intention revealing naming
- Aim for single level of abstraction
  - → SLAP https://hackernoon.com/object-oriented-tricks-6-slap-your-functions-a13d25a7d994
- DRY
- Try to avoid long list of arguments
- Aim for pure functions, avoid side-effects
- Design the (public) methods to be used with one way only
  - In general, prefer for example method overloading, multiple functions over boolean flags/other conditional arguments
  - Avoid primitives, use type system
- When calling, use named arguments if language supports

https://www.todaysoftmag.com/article/1071/clean-code-functions

Refactoring

- You can and should **first make code work**
- **Do not refactor without testing**, automated tests help a lot with refactoring
- Work in small, single changes at a time → verify with tests everything works
- Refactoring doesn't add any new functionality or remove existing ones
- Refactoring != Restructuring

https://dzone.com/articles/code-refactoring-techniques

- Make it readable
  - Naming, only valuable comments
- **DRY**
  - Extract method
  - Extract variable
- Reduce complexity
  - KISS
  - Correct abstractions that relate to your domain concepts
- More knowledge about the domain → refactor it as clear concepts into your software

Refactoring, functions example

```
void create(Order order, Customer customer, Boolean cust, Boolean ini) {
    //store persisted customer here
    def e1 = null
    //create customer conditionally
    if (cust) {
        //new customer with identifier, firstname and lastname
        e1 = new CustomerEntity(customer.identifier.valueLong, "Doe", "Doerson")
        //persist the customer
        e1 = entityManager.persist(e1)
    }
    //validate the order
    order.doStuff()
    //if no customer creation, but initialization should happen, initialize default products for order
    if (ini && !cust) {
        order.products.each { product ->
            product.amount = 1
        }
    }
    //create customer
    def o = new OrderEntity(order.identifier.valueLong, order.products, cust && !ini ? e1 : null)
    //persist the order
    repository.save(o)
}
```

Better methods?

```
Customer createCustomer(Customer customer) {
    def customerEntity = new CustomerEntity(
            id: customer.identifier.valueLong,
            firstName: customer.firstName,
            lastName: customer.lastName
    )
    return modelMapper.map(repository.save(customerEntity), Customer)
}
Order createOrder(Order order, Customer persistedCustomer) {
    order.validate()
    def orderEntity = new OrderEntity(
            id: order.identifier.valueLong,
            products: order.products,
            customer: modelMapper.map(persistedCustomer,
CustomerEntity)
    )
    return modelMapper.map(repository.save(orderEntity), Order)
}
```

```
static PRODUCT_DEFAULT_AMOUNT = 1

Order createDefaultInitializedOrder(Order order) {
    order.validate()
    order.products = initializeProductDefaultAmountsFor(order)
    //OR
    order.initializeProductDefaultAmounts()

    def orderEntity = modelMapper.map(order, OrderEntity)

    return modelMapper.map(repository.save(orderEntity), Order)
}

private static Set<OrderProduct>
initializeProductDefaultAmountsFor(Order order) {
    return order.products.collect { product ->
        new OrderProduct(
                code: product.code,
                amount: PRODUCT_DEFAULT_AMOUNT
        )
    }
}

void initializeProductDefaultAmounts() {
    products.each { product ->
        product.amount = PRODUCT_DEFAULT_AMOUNT
    }
}
```

More refactoring examples

```
@Override
Plan updatePlanWith(
        Customer.Identifier customerIdentifier,
        Plan.Identifier planIdentifier,
        Set<Receipt> receipts
) {
    Customer customer = persistence.fetchCustomerBy
customerIdentifier
    if (customer.plan && !customer.plan.empty && !customer.plan.id
== planIdentifier) {
        def errorMessage = "..."
        throw new NotFoundException(errorMessage)
    }
    def plan = customer.plan.get()
    plan.receipts = fetch(receipts)
    persistence.update(customer).plan.get()
}
```

```
@Override
Plan fetchPlanWith(
        Customer.Identifier customerIdentifier,
        Plan.Identifier planIdentifier
) {
    Customer customer = persistence.fetchCustomerBy
customerIdentifier
    if (customer.plan && !customer.plan.empty && !customer.plan.id
== planIdentifier) {
        def errorMessage = "..."
        throw new NotFoundException(errorMessage)
    }
    customer.consumptionPlan.get()
}
```

What are the reasons to refactor here?

Refactored example

```
@Override
Plan updatePlanWith(
        Customer.Identifier  customerIdentifier,
        Plan.Identifier  planIdentifier,
        Set<Receipt>  receipts
) {
    Customer  customer  = persistence.fetchCustomerBy
customerIdentifier
    def plan = getConsumptionPlan( customer,
consumptionPlanIdentifier )
    plan.products  = fetch(products)
    persistence.update(customer).consumptionPlan.get()
}
private Plan getPlan(Customer  customer, Plan.Identifier  planIdentifier) {
    boolean  planExistsFor  = { c -> c.consumptionPlan && !c.consumptionPlan.empty  }
    boolean  hasWantedPlan  = { c -> c.consumptionPlan.get().id ==  consumptionPlanIdentifier  }
    boolean  customerHasPlan  = planExistsFor(customer) && hasWantedPlan(customer)
    if (!customerHasPlan) {
        def errorMessage  = "..."
        throw new  NotFoundException( errorMessage )
    }
    customer.consumptionPlan.get()
}
```

```
@Override
Plan fetchConsumptionPlanWith(
        Customer.Identifier  customerIdentifier,
        Plan.Identifier  planIdentifier
) {
    Customer  customer  = persistence.fetchCustomerBy
customerIdentifier
    getConsumptionPlan( customer, consumptionPlanIdentifier )
}
```

SOLID

- **S**RP – Single Responsibility Principle
- **O**CP – Open/Closed Principle
- **L**SP – Liskov Substitution Principle
- **I**SP – Interface Segregation Principle
- **D**IP – Dependency Inversion Principle

https://deviq.com/solid/

Classes, interfaces

- Like functions, classes and interfaces should be small
- Size measured by responsibilities
  - Classes:
    - Aim for single responsibility → Single Responsibility Principle (SRP) → only one reason to change:
      https://medium.com/@severinperez/writing-flexible-code-with-the-single-responsibility-principle-b71c4f3f883f
      Aim for **cohesion:** instance variables should be operated by maximal amount of class methods
  - Interfaces (Interface Segregation Principle):
    - https://medium.com/@severinperez/avoiding-interface-pollution-with-the-interface-segregation-principle-5d3859c21013
- The name of the class/interface should reveal the responsibility
  - Try to avoid general names like Processor, Handler

SRP, Interface segregation with example

```
interface OrderHandler {
    void printOrder(Order order)
    long calculateOrderAmountWithTax(Orderorder, Tax taxRate)
    long calculateOrderAmount(Orderorder)
    Order persistOrder(Orderorder)
}
```

Too many responsibilities:

- Order presentation
- Order calculations
- Order persistence

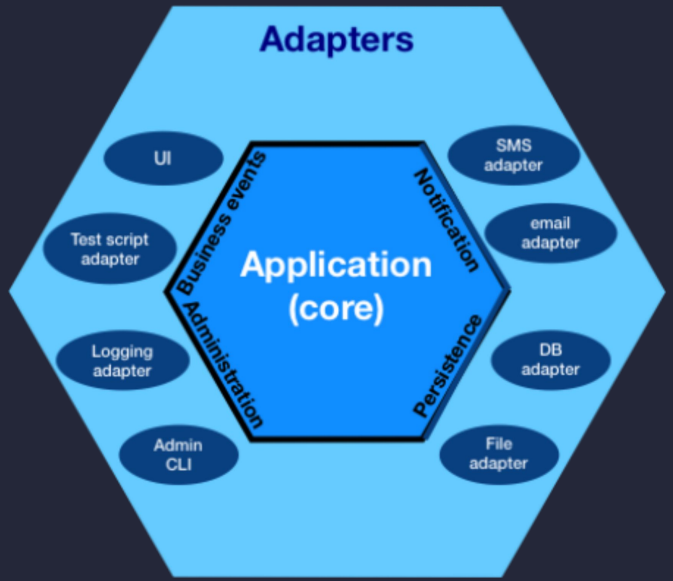
```
interface OrderPresenter {
    void presentOrder(Order order)
}

class OrderCalculator {
    long calculateOrderAmountWithTax(Order order, Tax
taxRate)
    long calculateOrderAmount(Order order)
}

interface OrderPersistence {
    Order persistOrder(Order order)
}
```

Better, each class / interface has single responsibility and single reason to change

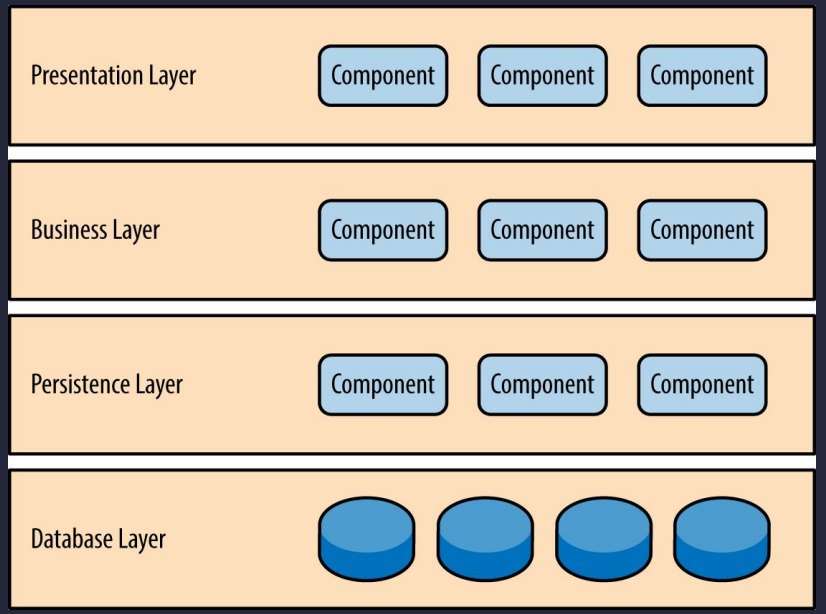
Few words about architecture

- Simple design
  - In this course, or even in a lot of work projects, you probably won't need multiple separate (micro)services
- Use the PO, or optimally the end-users, to get domain understanding
  - Keep that understanding (terms) consistent in every part of your applications
  - But do the domain processing only in one place: the backend of your application
    - And only in one abstraction layer inside the backend
- End user needs drive the architecture

Architectures



By Cth027 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=81276242



https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/assets/sapr_0101.png

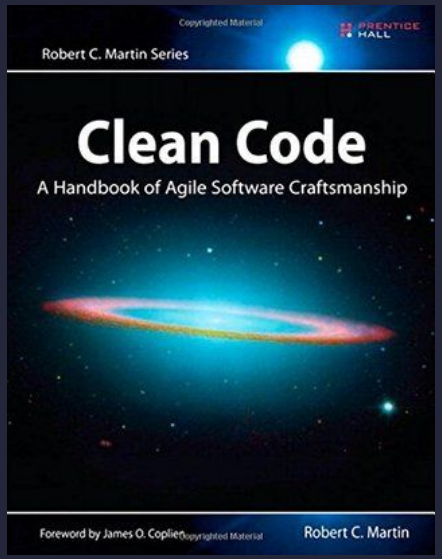
Clean Code

Book by Robert C. Martin

https://www.goodreads.com/book/show/3735293-clean-code

List of topics, a cheatsheet:

https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29



Next week

Probably the most important part for maintainable code: **automated testing**

- Basics of unit and integration testing
- Backend test examples
- React test example