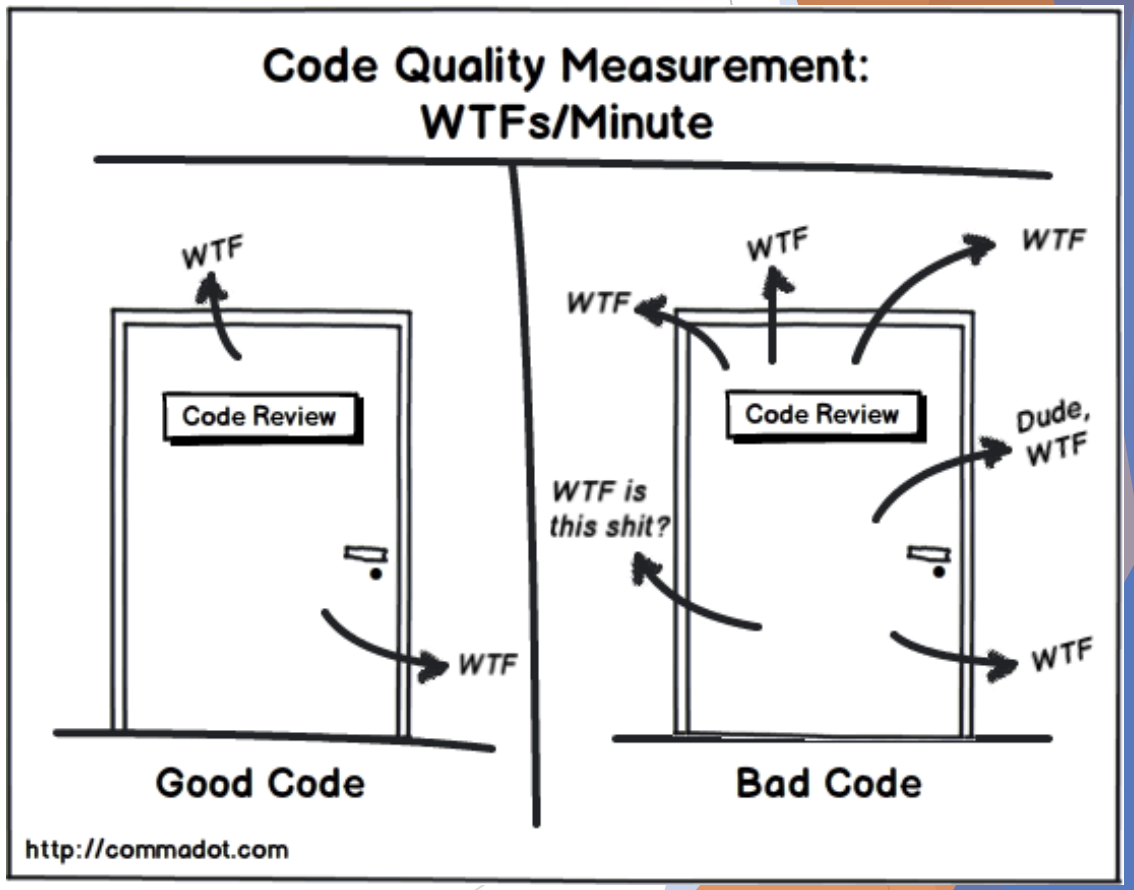
Hvordan skrive “god kode”

Hva er egentlig god kode?

- Hva som utgjør god kode kan deles opp i flere aspekter:

- Lesbarhet (ALLTID relevant)
  - Klart språk
  - Dokumentasjon av koden
- Opprettholdbarhet (Typisk relevant)
  - Modulær
  - Utvidbar
  - Testbar
- Effektivitet og stødighet (Kan være relevant)
  - Minimerer bruk av ressurser
  - Unngå unødvendig kode
  - Håndterer feil og uventede situasjoner

- Egentlig ingen fast definisjon ...



Lesbarhet

Jobb med navngivning

- Benytt navngivning som en måte å kommunisere ansvar / hensikt med klasser og metoder
  - Navnene bør være selvforklarende
  - Bør underforstått eller direkte beskrive hva klassen/metoden gjør
- Vær konsekvent med navngivning
  - De samme begrepene bør benyttes på tvers av applikasjonen
  - Etabler disse tidlig i prosessen (prosjektet «domene»)
- Unngå forkortelser (med unntak av slikt som DTO, ID osv.)
- Hvis du gjør det bra vil koden kunne leses som det var vanlig språk

Eksempel – Dårlig vs. god navngivning

Dårlig

```
public class Collection {
    // ...
    public void give(String string, Person p) {
        if (get(string) != null) {
            Book b = get(string);
            p.add(b);
            remove(b);
        }
    }
    //...
}
```

God

```
public class Library {
    // ...
    public void lendBook(String title, Member member) {
        if (bookIsAvailable(title)) {
            Book bookToLend = findBookByTitle(title);
            member.borrowBook(bookToLend);
            removeBookFromCollection(bookToLend);
        }
    }
    //...
}
```

Tips til navngivning - Klasser

- Bruk substantiv som reflekterer én type data eller konsept
  - F.eks. Customer, Library, DatabaseConnection ...
  - Igjen: Bruk standardiserte begreper i prosjektet
- Vær beskrivende og spesifikk
  - Manager vs. FileManager
  - Info vs. ProductInfo
- Knytt navnet til et klart og fokusert ansvar
  - F.eks. AuthenticationService, PaymentGateway, ServerRequestLoadBalancer

Tips til navngivning - Metoder

- Bruk verb som indikerer en handling
  - createProduct(), calculateTotalPrice(), sendEmailReceipt()
- Vær beskrivende og spesifikk
  - getInfo() vs. getUserContactInfo()
- Bruk ordet «is» ved boolean variabler
  - Kan ofte utvide objekt-variabler:
    - alarm.isActive(), product.isAvailable(), bookshelf.isEmpty()...
  - Kan noen ganger være tydeligere å legge ett ord før (Hvis hva det omhandler ikke er åpenbart ved bruk):
    - alarmIsActive(), productIsAvailable(), bookshelfIsEmpty()

Variabler

- Det er like viktig å godt navngi variabler for å gjøre individuelle kodelinjer forståelige
- Unngå variabler bestående av enkle bokstaver eller forkortelser
  - p vs. patient
  - ctx vs. context
  - Kan benytte slikt som x eller y hvis det bare gjelder løkke-iterering fra ett tall til et annet (alternativt: iteration eller index, som er litt mer beskrivende)
- Bruk variabelnavn til å gi innsikt i kontekst
  - book vs. bookToLend
  - picture vs. profilePicture
- Unngå for lange variabelnavn
  - Lange variabelnavn er et tegn på at du beskriver handlinger som skal skje i egne kodelinjer
  - F.eks. profilePictureToSendToUserPageInGUI

Dokumenter klasser og metoder

- Alle klasser og metoder bør dokumenteres med kommentarer
  - Med unntak av banale ting, slik som standard gettere/settere og konstruktører

- Dokumentasjonen bør beskrive overordnet formål, bruk og oppførsel
  - Beskriv overordnet hva klassen/metoden er ment til å gjøre (Typisk en setning eller to)
  - Gi kontekst i hvorfor klassen/metoden er nyttig og i hvilke tilfeller
  - Gi innsikt i hvordan klassen/metoden skal benyttes (typisk slikt som parametere og retur-verdier i metoder)
  - Skal IKKE beskrive individuelle kodelinjer (De burde snakke for seg selv...)

- Dokumentasjonen kan også gi
  - Informasjon om begrensninger og/avhengigheter
  - Eksempler på kode-bruk

Dokumentasjon - Klasse

- Beskriv formålet med klassen
  - Hva den representerer (Datatype vs. Samling med funksjonalitet)
  - Hva den overordnet gjør (men ikke hvordan...)
  - Spesifiser at klassen skal arves fra, hvis dette er tilfellet
- Beskriv avhengigheter klassen har, hvis relevant
- Eventuelt inkluder eksempel på bruk

```
/**
 * Represents a book in a library system.
 *
 * A Book object contains details about a book such as its title, author,
 * publication year, and ISBN number. This class provides basic methods for
 * retrieving and modifying these details.
 */
public class Book {
   //...
}
```

```
/**
 * Handles payment processing for orders in an e-commerce system.
 *
 * The PaymentProcessor class is responsible for processing payments via different
 * payment gateways such as credit cards and PayPal. It also handles refund processing.
 *
 * This class uses external payment APIs to complete transactions and assumes that
 * network communication and API credentials are properly configured.
 *
 * Example usage:
 * <pre>
 *   PaymentProcessor processor = new PaymentProcessor();
 *   processor.processPayment("creditCard", 150.00, "USD");
 * </pre>
 */
public class PaymentProcessor {
   //...
}
```

Dokumentasjon - Metode

- Beskriv overordnet hva metoden gjør og eventuelle forutsetninger
  - Beskriv eventuelle sideeffekter
  - Gi eventuelle eksempler på bruk
- Beskriv metodens parametere, hvis noen
- Beskriv hva som konseptuelt returneres, hvis retur-verdi
- Beskriv exceptions som kan forekomme, hvis noen

```
/**
 * Processes a payment through the specified method.
 *
 * @param paymentMethod The method of payment (e.g., "creditCard", "paypal").
 * @param amount The amount to be charged.
 * @param currency The currency in which the payment is made (e.g., "USD").
 * @return A boolean indicating whether the payment was successful.
 * @throws PaymentException if the payment fails due to an error in processing.
 */
public boolean processPayment(String paymentMethod, double amount, String currency) throws PaymentException {
    // Logic to process payment using the gateway
    return gateway.executeTransaction(paymentMethod, amount, currency);
}
```

Opprettholdbarhet

Lag små enheter

- Klasser og metoder burde være så små at de bare har ett overordnet ansvar
  - Går egentlig hand i hand med navngivning
- Tommelfingerregler
  - Hvis det er vanskelig å navngi en klasse/metode er det et tegn på at den gjør for mye (Bruk av ordet «og» er et meget tydelig tegn)
  - Hvis en metode er mer enn 10-20 linjer burde du vurdere å skille ut deler av den i egne metoder
  - En metode burde ha MAX 4 parametere, hvis fler, vurder å splitte metoden eller definere en klasse (DTO) som samler parameterene
  - Hvis en kodesekvens blir nestet i mer en 2-3 nivåer bør deler av koden typisk skilles i egne metoder
  - Hvis en klasse/metode er vanskelig å teste er dette ofte et tegn på at den har flere enn ett ansvarsområde og bør brytes ned
- Krever likevel erfaring ...

Klasse med for stort ansvar

- Hvor mange forskjellige ansvar har UserManager?
  - Registrere bruker
  - Autentisere bruker
  - Sende email til bruker
  - Et mer beskrivende navn kunne vært UserManagerAuthenticatorAndNotifier
- Burde brytes ned til tre klasser:
  - UserManager – Håndterer bare bruker-info
    - Registrere, hente, endre, osv.
  - AuthenticationService – Håndterer typer autentisering
    - Innlogging, sessions, handlingsrettingheter, osv.
  - NotificationService – Håndterer typer notifikasjon
    - Email, interne meldingssystemer,

```
public class UserManager {
    private Map<String, String> users; // Stores username and password
    private EmailService emailService; // Sends email notifications

    public UserManager() {
        // ...
    }

    /**
     * Registers a new user.
     */
    public boolean registerUser(String username, String password) {
        // ...
    }

    /**
     * Authenticates a user.
     */
    public boolean authenticateUser(String username, String password) {
        // ...
    }

    /**
     * Sends a welcome email to the user.
     */
    private void sendWelcomeEmail(String username) {
        // ...
    }
}
```

Metode med for stort ansvar

- Vi kan set at process order gjør ganske mange handlinger
  - Validering
  - Beregning
  - Discount
  - Betaling
  - Oppdatere lager
- Hver av disse handlingene burde defineres som egne metoder
  - Muligens også i egne klasser ...

```
public boolean processOrder(Order order, Customer customer) {
    // 1. Validate the order
    if (order == null || order.getItems().isEmpty()) {
        return false;
    }
    if (customer == null || !customer.isActive()) {
        return false;
    }

    // 2. Calculate total price
    double totalPrice = 0;
    for (Item item : order.getItems()) {
        totalPrice += item.getPrice();
    }

    // 3. Apply discounts
    if (customer.isLoyal()) {
        totalPrice *= 0.9; // 10% discount for loyal customers
    }

    // 4. Charge the customer
    boolean paymentSuccessful = chargeCustomer(customer, totalPrice);
    if (!paymentSuccessful) {
        return false;
    }

    // 5. Update inventory
    for (Item item : order.getItems()) {
        updateInventory(item);
    }

    return true;
}
```

Metode med delegert ansvar

- Vi deler opp i metoder (utelatt i eksemplet under) og kaller disse i den originale metoden:

```
public boolean processOrder(Order order, Customer customer) {
    if (!orderIsValid(order, customer)) {
        return false;
    }

    double totalPrice = calculateTotalPrice(order);
    double totalPriceWithDiscount = applyDiscounts(totalPrice, customer);

    boolean paymentSuccessful = chargeCustomer(customer, totalPriceWithDiscount);
    if (!paymentSuccessful) {
        return false;
    }

    updateInventory(order);

    return true;
}
```

Skriv kode som er enkel og konsis

- Enkel og konsis kode er enklere å forstå, modifisere og opprettholde
- Skriv koden på den enklest mulige måten
  - Test Driven Development og generell refaktorering kan hjelpe med dette
- DRY (Don’t repeat yourself)
  - Lag egne metoder for hyppig brukte kodesekvenser
  - Lag gjenbrukbare konstanter i stedet for å hardkode verdier
  - Bruk abstraksjonsteknikker slik som arv og interfaces ved felles oppførsel/egenskaper i forskjellige klasser
- YAGNI (You Ain’t Gonna Need It)
  - Bare skriv kode når du vet du trenger den

Vær litt forsiktig med arv ...

- Arv er (ironisk nok) tett-koblet kode...
  - En barneklasse er direkte knyttet til sin foreldreklasse
- Potensielle problemer:
  - En endring i en foreldreklasse kan medføre endringer i alle dens barneklasser
  - Det kan være en stor jobb å refaktorere endringer i arv-hierarkiet
  - En barne-klasse er påtvunget foreldreklassens funksjonalitet selv i tilfeller hvor den ikke har et behov
- Benytt bare arv hvis det finnes et tydelig «X ER en Y»-forhold!
- ... eller hvis du skriver et rammeverk hvor klassen er tiltenkt å arves for å utvide dens funksjonalitet

Alternativer til arv

- Composition
  - Når en klasse benytter funksjonalitet gjennom et tilhørende objekt
  - Brukes når det er et «X HAR en Y»-forhold
  - Klassen instansierer objektet selv
- Interface
  - Benyttes til å definere abstrakte metoder som kan implementeres unikt i mange forskjellige klasser
  - Mer fleksibelt enn å definere abstrakte metoder gjennom arv
  - En klasse kan implementere så mange interfaces som den ønsker
- Dependency injection
  - Egentlig det samme som en Composition, men objektet instansieres utenfor klassen og sendes i stedet med som en parameter i konstruktøren
  - Parameteren defineres ofte med et Interface (eller foreldre-klasse...)
  - Vi setter krav til objektets funksjonalitet, men er fleksible iht. implementasjon

Spør en AI-chatbot om tilbakemelding

- Skriv koden din selv som utgangspunkt, men spør gjerne en AI-chatbot om tilbakemelding på koden din og forslag til forbedring
  - Selve kodelogikken din
  - Dokumentasjonen din
- AI-chatbotter er ganske gode på å sjekke koden din opp mot «best practise»
  - Mye å lære av dette!
  - Kan påpeke ting du ikke var klar over
- … Men ikke ta endringene for god fisk!
  - Sørg for at du forstår forslagene og evaluer om disse er fornuftige.

Tips til dere og prosjektet

- Se på koden dere har skrevet til nå ...
  - Hvor lett er koden å lese og forstå (også for noen som ikke er kjent med den)?
    - Navngivning av Klasser, Metoder og Variabler
    - Dokumentasjon for ytterligere kontekst
  - Har klasser og metoder ett konkret ansvar?
  - Gjenbruker vi egentlig kode på en fornuftig måte
    - Bare arv hvis X er en Y
    - Bruk gjerne interfaces som et alternativ
    - Les dere opp på Composition og Dependency Injection

Mer om god kode

- https://www.youtube.com/@CodeAesthetic