Noter til "Refactoring"

2. Definition af Refactoring

- **Refactoring (navneord):** Ændring af softwarestruktur for at gøre det lettere at forstå og billigere at ændre uden at ændre dets observerbare adfærd.
- **Refactor (udsagnsord):** Omstrukturering af software ved at anvende en række refaktoreringer uden at ændre dets observerbare adfærd.

3. Formål med Refactoring

- Forbedre kodekvalitet
- Vedligeholdelsesevne
- Forståelighed
- Enkelhed
- Udvidelsesmuligheder
- Testbarhed
- Adhærere til SOLID-principper uden at ændre eksternt synlig adfærd

4. Hvornår skal man Refactor?

- Før du starter en ny funktion for at gøre det lettere at implementere funktionen.
- Efter alle tests er bestået for at sikre, at du ikke ændrer den synlige adfærd.

5. Code Smells

- **Bloaters:** Lange metoder, store klasser, primitive obsessioner, lange parameterlister, dataklumper.
- **Object-Orientation Abusers:** Switch-sætninger, midlertidige felter, nægtet arv, alternative klasser med forskellige interfaces.
- **Change Preventers:** Divergerende ændringer, shotgun surgery, parallel arv hierarkier.
- **Dispensables:** Dovne klasser, dataklasser, duplikeret kode, dødkode, spekulativ generalitet.
- **Couplers:** Feature envy, upassende intimitet, beskedkæder, mellemmand.

6. Forberedelse til Refactoring

- Sørg for at have et omfattende testsuite af koden, du skal ændre.
- Tag små skridt.
- Find en "code smell".
- Anvend en refaktorering og verificer, at alle tests stadig passerer.
- Brug kildekontrol.

7. Eksempler på Refactoring

- **Extract Method:**

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}
```

```
void printDetails(double amount) {
    std::cout << "name: " << _name << std::endl;
    std::cout << "amount: " << amount << std::endl;
}
```

- **Extract Class:**
  - Flyt relevante felter og metoder fra en klasse til en ny klasse.
- **Replace Conditional with Polymorphism:**
  - Brug polymorfi i stedet for betingede sætninger.

```
class Bird {
public:
    virtual double getSpeed() = 0;
};

class European : public Bird {
public:
    double getSpeed() override { return getBaseSpeed(); }
};

class African : public Bird {
public:
    double getSpeed() override { return getBaseSpeed() - getLoadFactor() *
numberOfCoconuts; }
};

class NorwegianBlue : public Bird {
public:
    double getSpeed() override { return isNailed ? 0 :
getBaseSpeed(voltage); }
};
```

8. Vigtige Refactorings

- **Move Field:**
  - Motivation: Hvorfor flytte et felt.
  - Mekanik: Sørg for, at feltet er indkapslet, opret felt og accessor i målklasse, test.

9. Praktiske Øvelser

- Start øvelser for at øve refaktorering.

10. Diskussion Spørgsmål

- Hvordan føltes det at arbejde med hurtige, omfattende tests?
- Lavede du fejl under refaktoreringen, som blev fanget af tests?
- Hvad ville du sige til en kollega eller chef om værdien af denne refaktorering?

11. Fremtidig Arbejde

- Øv ved at identificere "code smells" og bruge cheatsheets til at finde løsninger.
- Prøv refaktorering katas.

C++ Kodeeksempler for Refactoring

Extract Method Example:

```
#include <iostream>
#include <string>

class Customer {
private:
    std::string _name;

    void printBanner() {
        std::cout << "**************************" << std::endl;
        std::cout << "**** Customer Owes ******" << std::endl;
        std::cout << "**************************" << std::endl;
    }

    void printDetails(double amount) {
        std::cout << "name: " << _name << std::endl;
        std::cout << "amount: " << amount << std::endl;
    }

public:
    Customer(const std::string& name) : _name(name) {}

    void printOwing(double amount) {
        printBanner();
        printDetails(amount);
    }
};

int main() {
    Customer customer("John Doe");
    customer.printOwing(150.0);
    return 0;
}
```

Replace Conditional with Polymorphism Example:

```
#include <iostream>

class Bird {
public:
    virtual double getSpeed() = 0;
    virtual ~Bird() = default;
};

class European : public Bird {
public:
    double getSpeed() override {
        return 10.0; // Example base speed
    }
};

class African : public Bird {
    int numberOfCoconuts;
public:
    African(int coconuts) : numberOfCoconuts(coconuts) {}

    double getSpeed() override {
        return 10.0 - 1.0 * numberOfCoconuts; // Example calculation
    }
};
```

```
class NorwegianBlue : public Bird {
    bool isNailed;
    int voltage;
public:
    NorwegianBlue(bool nailed, int volt) : isNailed(nailed), voltage(volt) {}

    double getSpeed() override {
        return isNailed ? 0 : 10.0 * voltage; // Example calculation
    }
};

int main() {
    Bird* bird1 = new European();
    Bird* bird2 = new African(3);
    Bird* bird3 = new NorwegianBlue(true, 5);

    std::cout << "European bird speed: " << bird1->getSpeed() << std::endl;
    std::cout << "African bird speed: " << bird2->getSpeed() << std::endl;
    std::cout << "Norwegian Blue bird speed: " << bird3->getSpeed() <<
std::endl;

    delete bird1;
    delete bird2;
    delete bird3;

    return 0;
}
```

Disse eksempler viser, hvordan refaktorering kan forbedre kodekvaliteten ved at gøre den mere forståelig og vedligeholdelsesvenlig.