JavaFX Best Practices

Sauberer Code

- Aussagekräftige Namen
    - summePreise anstatt x oder summerDerEinkaufspreise
- Kein doppelter Code
    - Lieber eine extra Methode anstatt copy-paste
- Methoden keine langen Parameterlisten übergeben
    - do(car) anstatt do(tire1, tire2, tire3, tire4, motor)
- Feature Neid an Daten anderer Klassen vermeiden
- Keine Datenklassen
    - Anstatt if(carTiresMotors = car.getTires() + car.getMotors() > 5) if(car.getTiresMotors() > 5) nutzen, also Funktionalitäten dort implementieren wo sich die benötigten Parameter befinden
- Polymorphismus statt Case-Anweisung

```
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCocon
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

```
abstract class Bird {
  // ...
  abstract double getSpeed();
}

class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}

// Somewhere in client code
speed = bird.getSpeed();
```

- Keine Variablen die nur in manchen Fällen verwendet werden
    - Int a = SumTires;
      If(hasTires){
        x = SumTires*2;
      } else {
        ...
      } :(
- Keine Unterklassen die geerbte Methoden oder Daten nicht benötigen

Public class Car(){ Tires = 4 };
    Public class Boat() extends Car{ Sails = 3;} :(

- Kommentare sind keine Entschuldigung für überkomplizierten/ schlechten Code

- Kein Exhibitionismus von internen Details einer Klasse

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- Einfaches Design anstatt Versteifung auf Entwurfsmuster
- Zu Komplexe Verzweigung (durch Prüfung von für den Code-Block unwichtigen Details)
    - if(if(if(if()else())else(if())))
- Zu tiefe Verschachtelung
    - While(){while(){if(){while()}}}
- Einheitliche Dokumentierung
- Eine Klasse = eine Aufgabe – also lieber viele kleine Klassen als wenige große
    - Car und Straße Klasse anstatt nur die Klasse Verkehr
- Klassen erweitern statt modifizieren
    - bestehenden Code nicht ändern sondern über Vererbung oder mithilfe von Interfaces neue Funktionalitäten einfügen
    - Anstatt in Car Daten für F1 Wagen zu schreiben die normale Autos nicht benötigen neue abgeleitete Klasse Rennauto einführen
- Abgeleitete Klassen verhalten sich wie die Basisklasse –
    - damit beispielsweise Parameter einheitlich von anderen Klassen genutzt werden können, also keine neuen Exceptions oder andere Modifizierungen
- Starke Abhängigkeiten zwischen Klassen vermeiden

Bild 4: Starke Kopplung

– stattdessen Abhängigkeiten über Interfaces implementieren

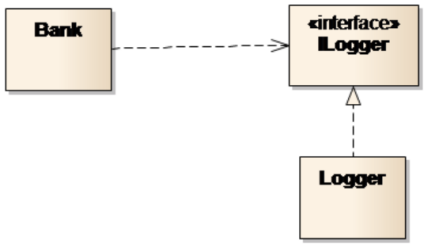

Bild 5: Schwache Kopplung durch Assoziation

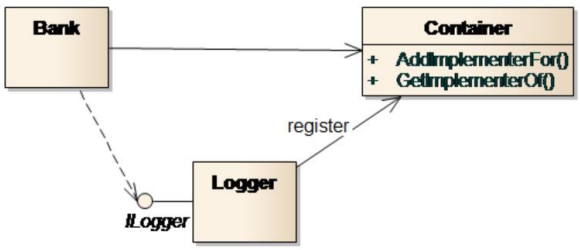

Bild 6: Nutzung eines IoC-Containers

Logger an Konstruktor Bank übergeben - Anfrage an Interface nach Implementierung

JavaFX

- Eigener Preloader um Ladezeiten zu verkürzen - https://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.htm
- Model-View-Controller
  - Model = Objekte
  - View durch FXML
  - Controller = Java Code der die Benutzerinteraktion mit dem GUI definiert
- CSS zum stylen von GUI Komponenten
- Anweisungen (die große Mengen an Daten verarbeiten) in seperatem Thread im Hintergrund laufen lassen