MODULARITÄT

Levi Bautz

MODULARITÄT

- Grundprinzip der Modularität
- Eigenschaften modularen Codes
- Module und Beispiele
- Zusammenfassung

GRUNDPRINZIP

- Logische, funktionsweise Aufteilung
- Module interagieren über Schnittstelle
- Kontrast zu monolitisch
- Moderner Standard
- Schließt OO nicht aus

Beispiele für Module

- Top-down Aufteilung:
- Programm
  - Libraries
    - Klassen
      - Methoden

EIGENSCHAFTEN

___

Eigenschaften

- Wiederverwendbarkeit
- Kapselung
- Aufteilung nach Funktion
- Daraus resultierend
  - Ersetzbarkeit
  - Verständlich
  - Leichteres Fehler beheben

Wiederverwenbarkeit

- Mehrfachverwendung zusammenfassen
- Libraries nutzen
- Spart eine Menge Zeit

```
public class
BeispielWiederverwendbarkeit
{
private int Treffer(objekt)
{
    if objekt.getroffen()
        {return -20;}
    else
        {return 0;}
}
}
```

Verkapselung

- Module klar trennen
- Interaktion über Interface
- Macht Änderungen leichter
- Übersichtlicher

```
public class VerkapselungBeispiel

{
public int getWert()
//Schnittstelle
    {return Wert;}
private void berechneWert()
//innere prozesse
    {...}
```

Aufteilung nach Funktion

- Kleine vs. große Teile
- Lesbarkeit/Verständnis vs. Funktionalität
- Eher Stilfrage als objektive Korrektheit
- Trozdem: Kleiner = modularer

```
public class Spieler
{
public void Detection+Schaden
                            (projektil)
{
    if (projektil.pos == spieler.pos)
        {int schaden = 20;
         Spieler.schaden(schaden);}
}
}
```

Alternativen:
-Alles im Spielermodul
-Hitdetection modularisieren
...

MODULE

___

Methoden

- Unterster Baustein
- Java: <sichtbarkeit><rückgabetyp><name><(parameter)>{code}
- C++

  <rückgabetyp><name><(parameter)> {code}

  Sichtbarkeit in Blöcken
- Python

  def <name><(parameter)>

  Code

Klassen

- Umfassen mehrere Methoden nach übergeordneter Funktion
- Class <name>
- Konstruktor
- Bietet Schnittstelle mit public Methoden
- In Python ist alles public

Libraries

	Static Library	Dynamic Library
Datentyp	-.a, .lib	-.so, .dll
Einbindung	-Zu Kompilierzeit	-Bei Laufzeit
Vorgehens-weise	-Fügt genutzten code aus Library ein	-Referenzen auf Library
Pro	-Sehr unabhängig -Theoretisch etwas schneller	-Kleinere Dateien -Leichtere Bearbeitung -Weniger kompilieren
Contra	-Größere Dateien -Schwere Bearbeitung	-Abhängiger -Eventuell langsamer

Datentypen

- .a, .lib
  - Konzeptgleich (außer Importlibrary)
  - Codearchiv
- .so, .dll
  - Konzeptgleich
  - Executable ähnliche Dateien
  - Bei .dll extra static lib oder im Code manuell loaden
  - Andere feine Unterschiede bei Ausführung

Einbindung

- Static:
  - Linker schreibt Code in executable
  - Header und Library benötigt
- Dynamic:
  - Linker schreibt Referenzen
  - Dynamic muss zur Laufzeit an richtigem Ort sein (Standardverzeichnis oder eigenes)
  - Loader lädt Funktionen

Java

- Packages sind keine Libraries
- JARs aber schon
- JAR erstellen: Export → JAR
- JAR einbinden:
  - Library dem Buildpath hinzufügen
  - Import <jarfilename>.Klassenname

Python

- Module über .py Dateien
- Packages sammeln diese
- Import <dateiname>
- Bei packages mit Punktnotation

Zusammenfassung

- Erleichtert große Probleme, Teamarbeit
- Erleichtert Weiterentwicklung
- Leichteres Verständnis
- Schneller Entwicklung
- Dynamic vs. Static
  - Dynamic ist ressourcensparender
  - Static schneller, simpler und sicherer

Literaturverzeichnis

- https://www.jwhitham.org/2017/10/dll.html
- https://en.wikipedia.org/wiki/Modular_programming
- https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html