Design Patterns

...

Riedmann

Jemand hat Dein Problem bereits gelöst

Patterns

- Strategy
- Observer
- Singleton
- Factory
- Command
- Decorator
- Anti-Patterns

OO Prinzipien

Wenn folgende Prinzipien eingehalten werden entstehen automatisch Patterns!

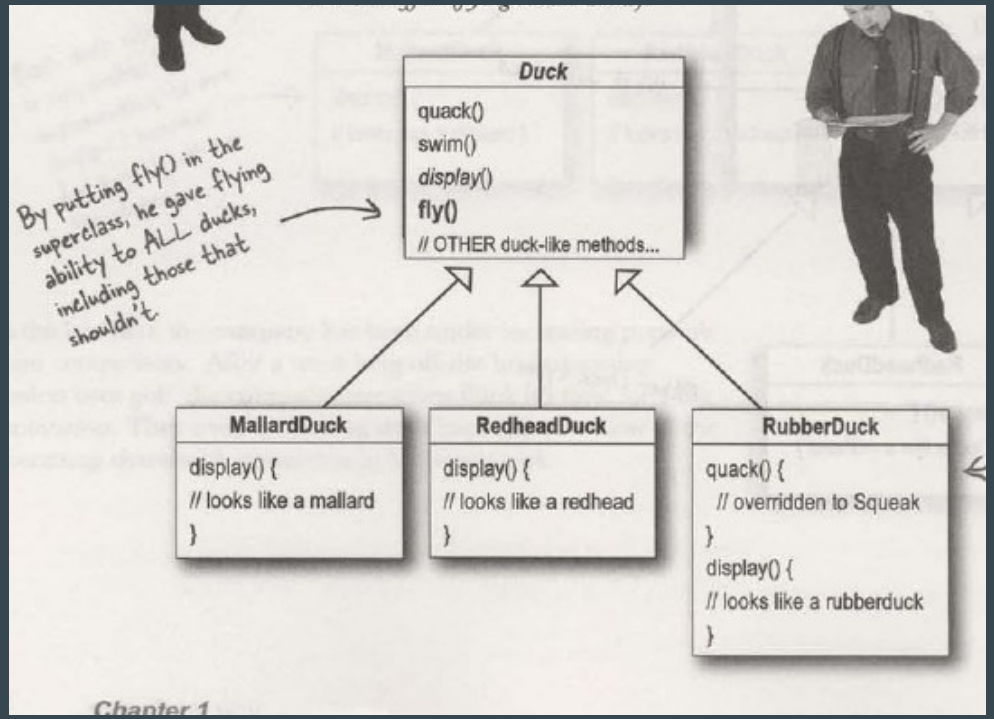
- Programmiere gegen ein Interface
- Kapsle die Dinge die variieren
- Bevorzuge Komposition gegenüber Vererbung
- DRY - Dont repeat yourself
- Klassen die miteinander agieren sollen nicht zu eng verwoben sein (loosly coupled)

Strategy Pattern

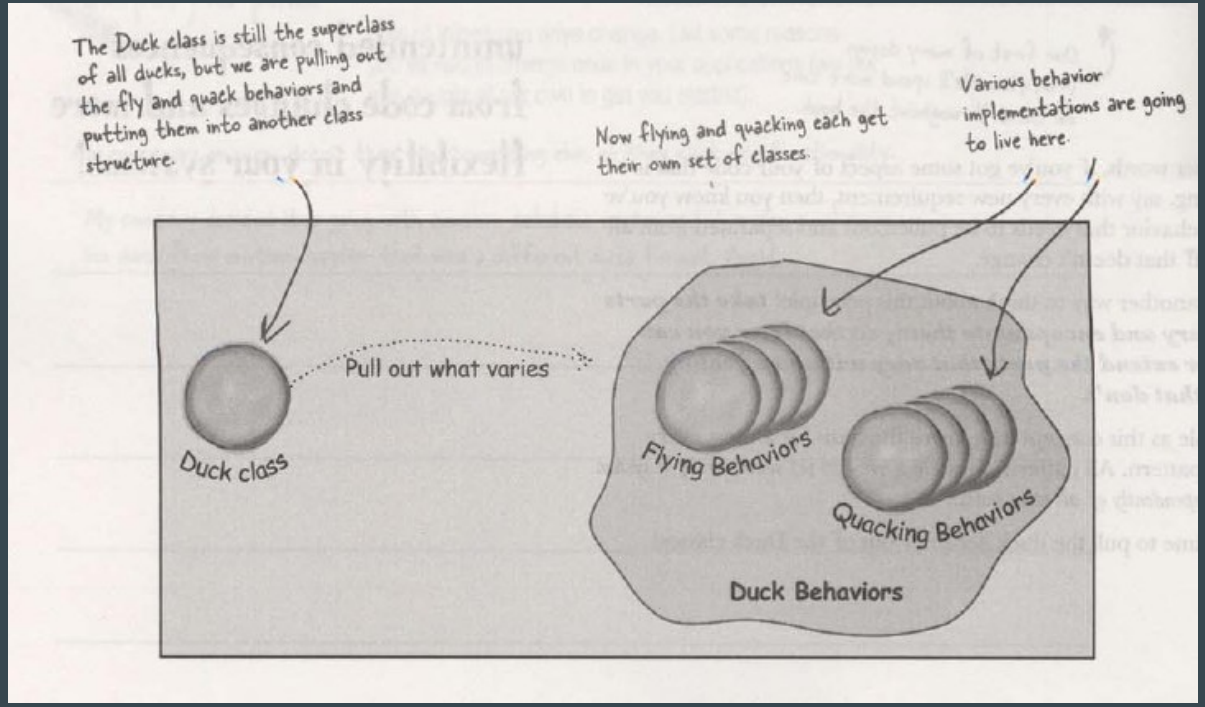
Beispiel: Flugverhalten von Objekten

- Wenn dies mit Vererbung implementiert wird bekommt man 100te Klassen
- Das wird absolut unübersichtlich
- Besser:
  - Komposition (anstatt Vererbung)
  - Die Flugstraegie wird in eigene Klassen gezogen

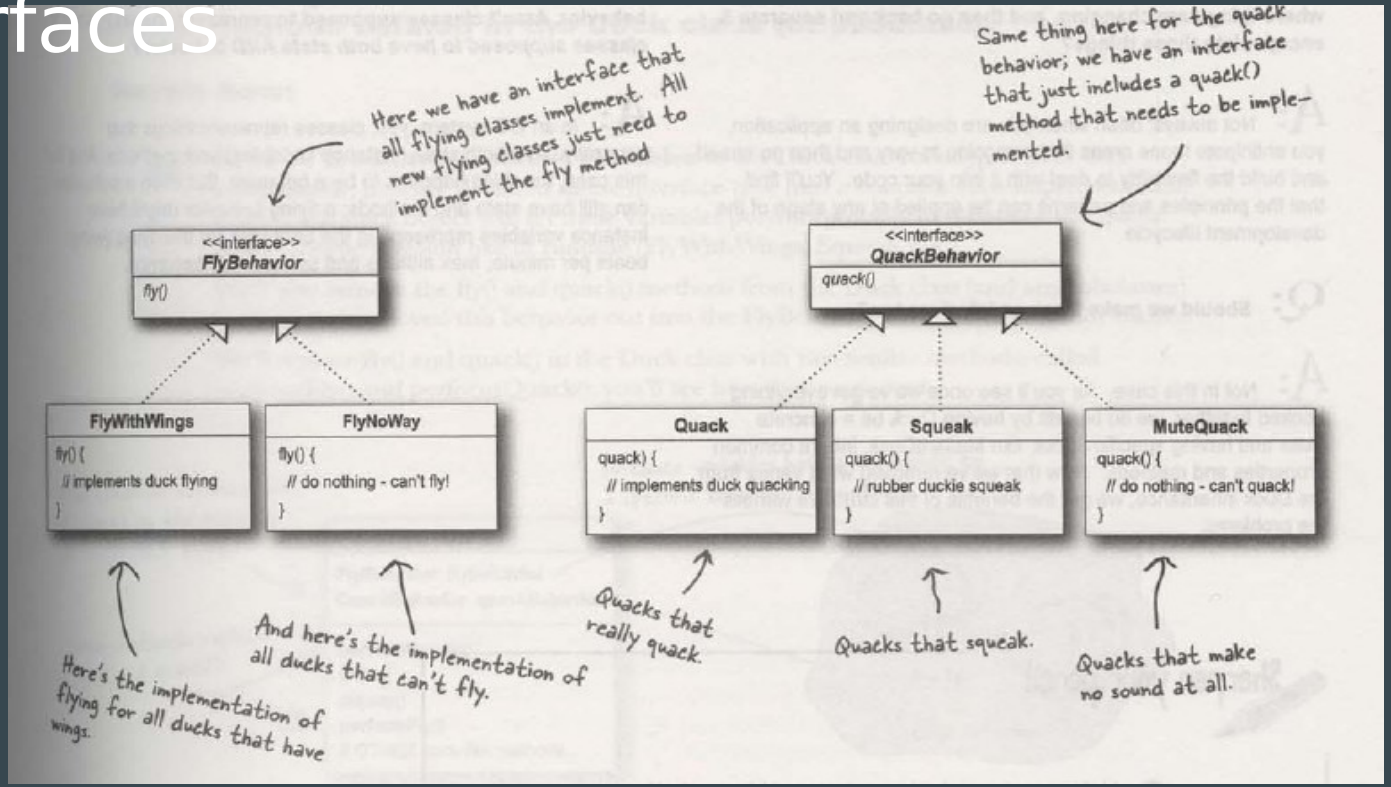
Strategy Pattern



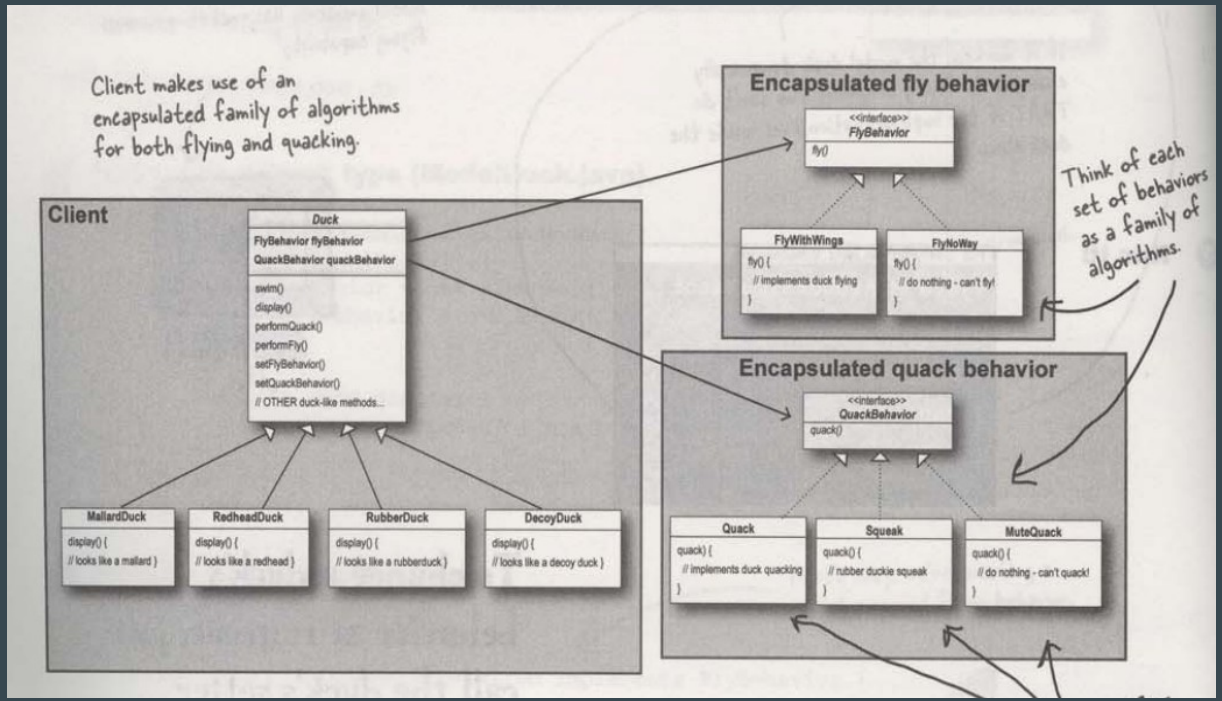
Strategy Pattern



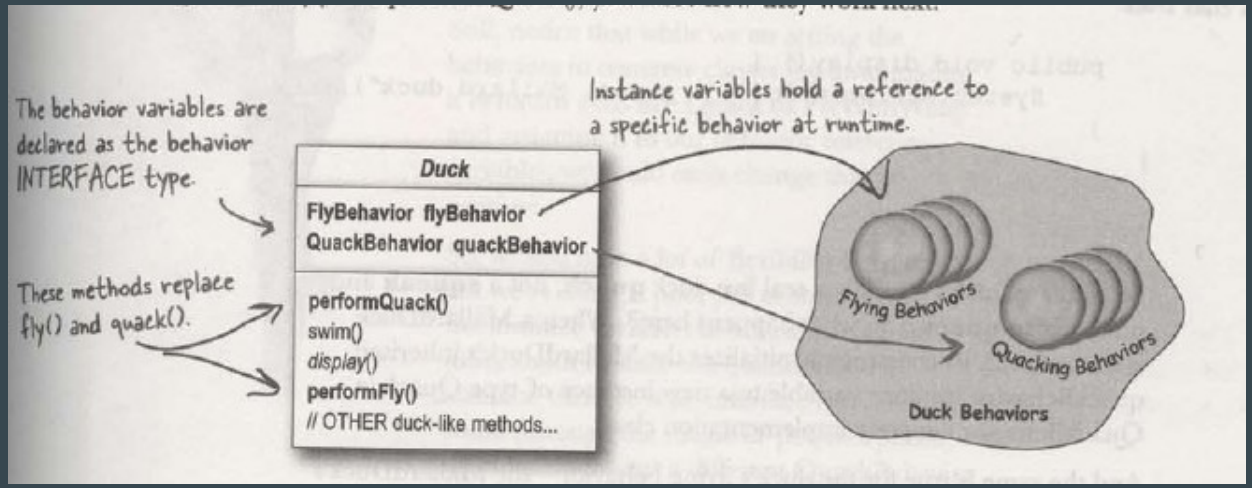
Strategy Pattern - Programmiere gegen Interfaces



Strategy Pattern



Strategy Pattern



The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Verpasse nichts wichtiges über das Du informiert werden solltest

Observer Pattern

Beispiel:

- Ein Temperatursensor überwacht die Raumtemperatur
- Wenn eine gewisse Temperatur überschritten wird müssen mehrere Systeme informiert werden
  - Rollos
  - Heizung
  - Info per SMS

Observer Patterns

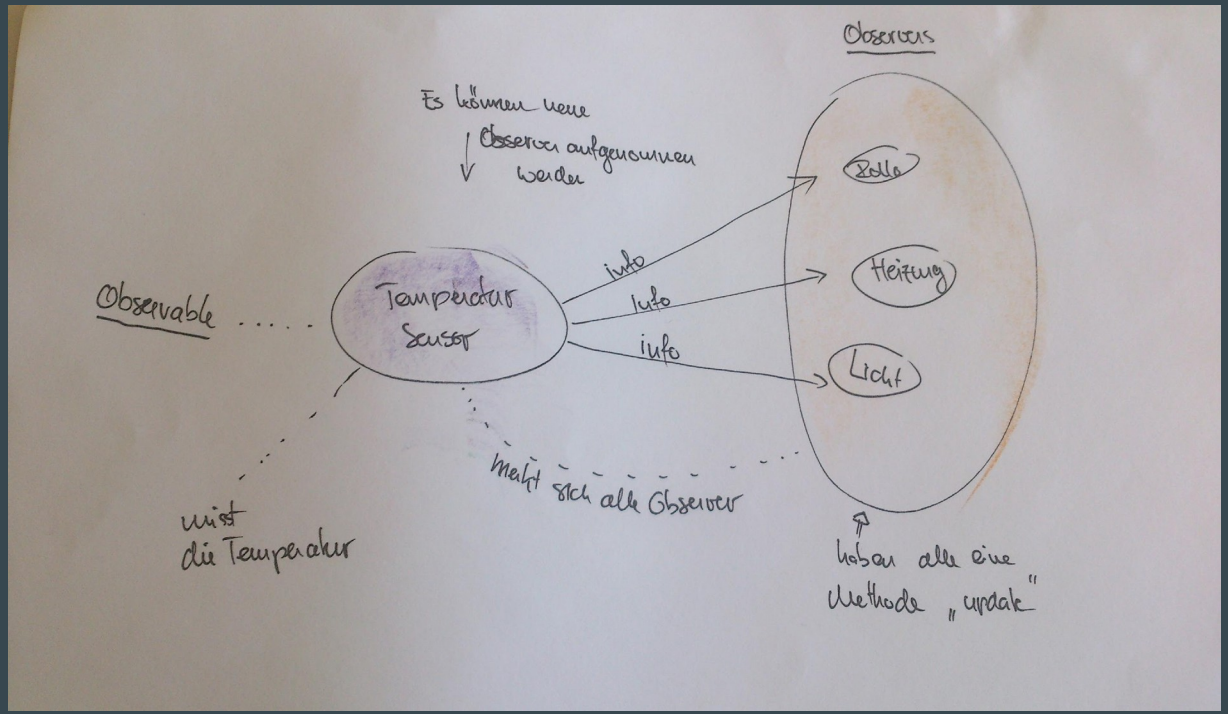
- Inversion of Control (IOC)
  - Nicht die Rollos fragen ständig nach wie die Temperatur ist sondern der Sensor informiert alle Anderen
  - Infopfad wird umgedreht (Inversion)
  - Auch „Hollywood Prinzip“ genannt → Don't call us, we call you

Don’t call us, we call you
“Hollywood Prinzip”

Observer Pattern

- Temperatursensor
  - Ist ein Observable → er wird überwacht
  - Kennt alle, die ihn überwachen
- Rollos, Heizung
  - Sind Observers → sie überwachen den Sensor
  - Werden vom Sensor automatisch informiert, wenn die Temperatur einen gewissen Wert übersteigt

Observer Pattern



Singleton

Manchmal will man sicherstellen, dass eine Klasse nur EIN mal im gesamten System existieren kann. Folgende Aspekte können erreicht werden

- Globaler Zugriff auf das Singleton
- Das Singleton-Objekt kann niemandem genau zugeordnet werden
- Lazy Initialization ist gewünscht (Objekte sollen erst initialisiert werden wenn sie benötigt werden)

Singleton

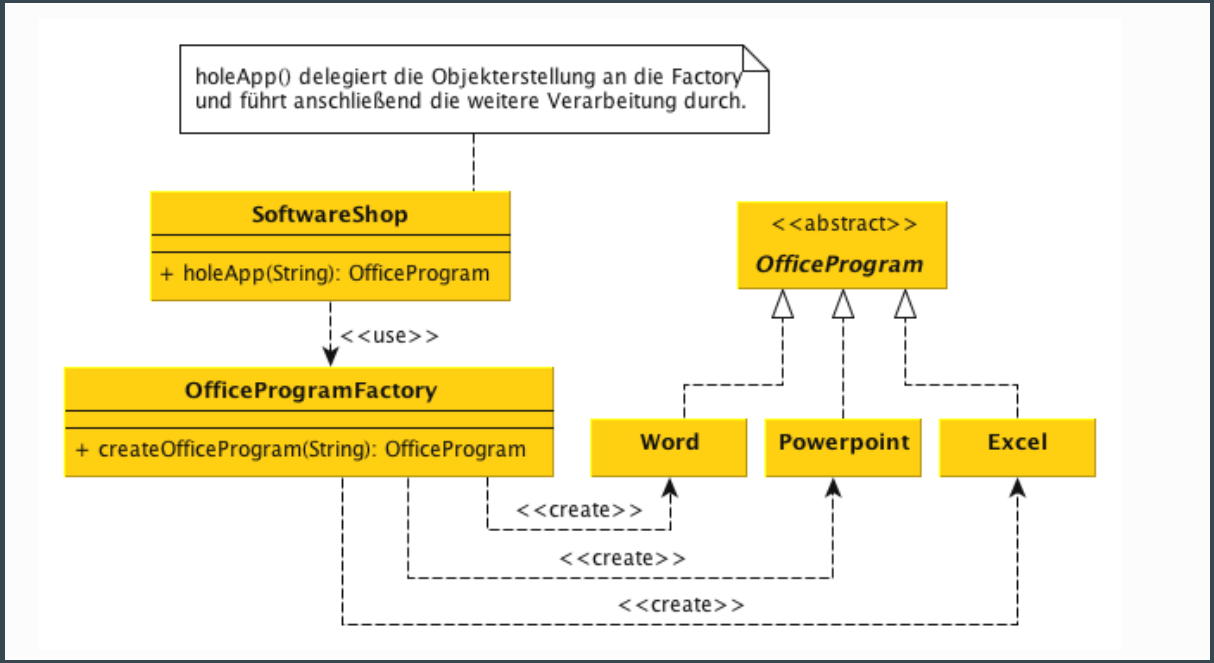
Beispiele

- Zählerobjekt in einem Spiel
- Druck-Spooler
- Anstatt globalen Variablen

Factory

- Die Factory liefert Objekte zurück
- Der Benutzer benötigt KEIN Wissen über die interne Erstellung
- Die Factory kann aufgrund interner Information (z.b. OS) unterschiedliche Objekte zurückliefern
- Die Factory liefert immer den Interface-Typen zurück

Factory


holeApp() delegiert die Objekterstellung an die Factory
und führt anschließend die weitere Verarbeitung durch.
SoftwareShop
+ holeApp(String): OfficeProgram
<<use>>
OfficeProgramFactory
+ createOfficeProgram(String): OfficeProgram
<<abstract>>
OfficeProgram
Word
Powerpoint
Excel
<<create>>
<<create>>
<<create>>

Factory

- Die Factory in obigem Beispiel liefert ein Objekt vom Typ OfficeProgram zurück
- Das kann eines der folgenden Objekte sein
  - Word
  - Powerpoint
  - Excel

Anti-Patterns

- Beschreiben typische Lösungen die NICHT gewünscht sind
- Anti-Patterns beschreiben Lösungen, die zu Schwierigkeiten führen
- Anti-Patterns beschreiben die typischen Schwachstellen in der Programmierung

BLOB

Do you remember the original black-and-white movie The Blob? Perhaps you saw only the recent remake. In either case, the story line was almost the same: A drip-sized, jellylike alien life form from outer space somehow makes it to Earth.

Whenever the jelly thing eats (usually unsuspecting earthlings), it grows. Meanwhile, incredulous earthlings panic and ignore the one crazy scientist who knows what's happening. Many more people are eaten before they come to their senses. Eventually, the Blob grows so large that it threatens to wipe out the entire planet.

Quelle: https://sourcemaking.com/antipatterns

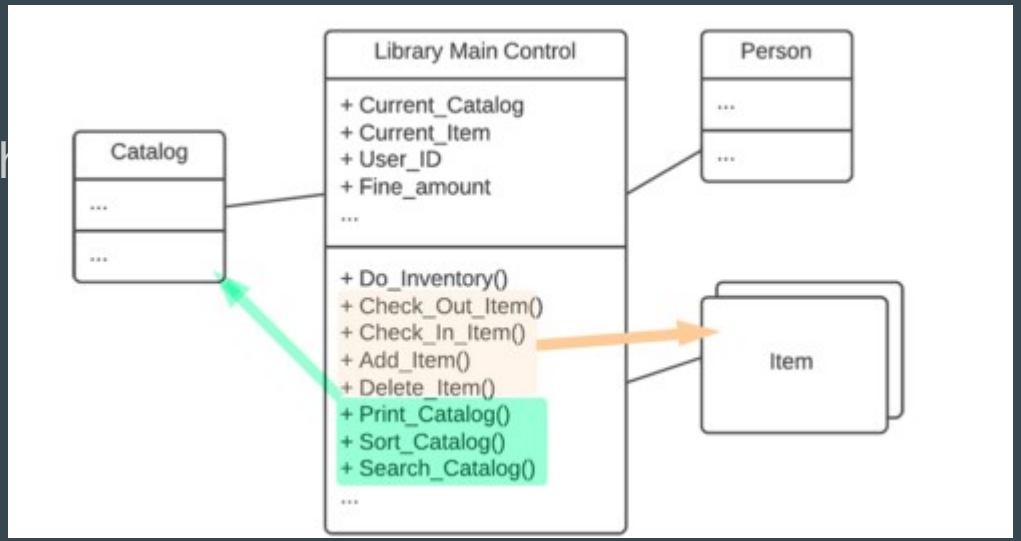
BLOB

- Eine Klasse kümmert sich um fast alles
- Die restlichen Klassen sind meist nur noch kleine Datencontainer
- Ist im Prinzip ein prozedurales Programm (also keine OO)
- Ursachen
  - Fehlendes Design
  - Ein Prototyp wird ständig erweitert
  - GUI basierte Progammierung z.B. mit visual Studio führt wg. Unwissenheit oft zu solchen Klassen
- Symptome
  - Einzelne Klassen mit unzähligen Instanzvariablen und Methoden
  - Kein OO Design, keine Design-Patterns

BLOB

Lösung:

- Eine Klasse macht eine Sach
- Refactoring



Lava Flow - Dead Code

- Typische Symptome
  - “Ich habe keine Ahnung was diese Klasse macht”
  - “Das hat man gemacht bevor ich gekommen bin”
  - Ein großteil des Codes wird nicht verstanden
  - Man weiss nicht einmal ob der Code benötigt wird oder nicht
- Konsequenzen
  - Sehr schwierig zu finden
  - Viel Code wird in den Speicher geladen