Software Design Patterns

Einführung II

Termine

22.04. - Verlegt (Christian Stein / Glenn Bernhardt: Sonstige)
29.04. - Jan Rausch: Anpassung
06.05. - Johannes Schmidt: Message Routing
13.05. - Ralf Rublack: Fabrik
20.05. - René Speck: Servicevariation
27.05. - Daniel Gerber: Zusammensetzung
03.06. - Michael Hütel: Message Transformation
10.06. - Mario Heidenreich: Zugriff
17.06. - Michael Lühr: MVC
24.06. - Daniel Müller: Kommando
01.07. - Martin Czygan: Ereignisbearbeitung
08.07. - Christian Kube: Kommunikation

Objektorientierung

- OO-Basics
  - Abstraktion
  - Kapselung
  - Polymorphismus
  - Vererbung

Unser Next-Gen-MMO!

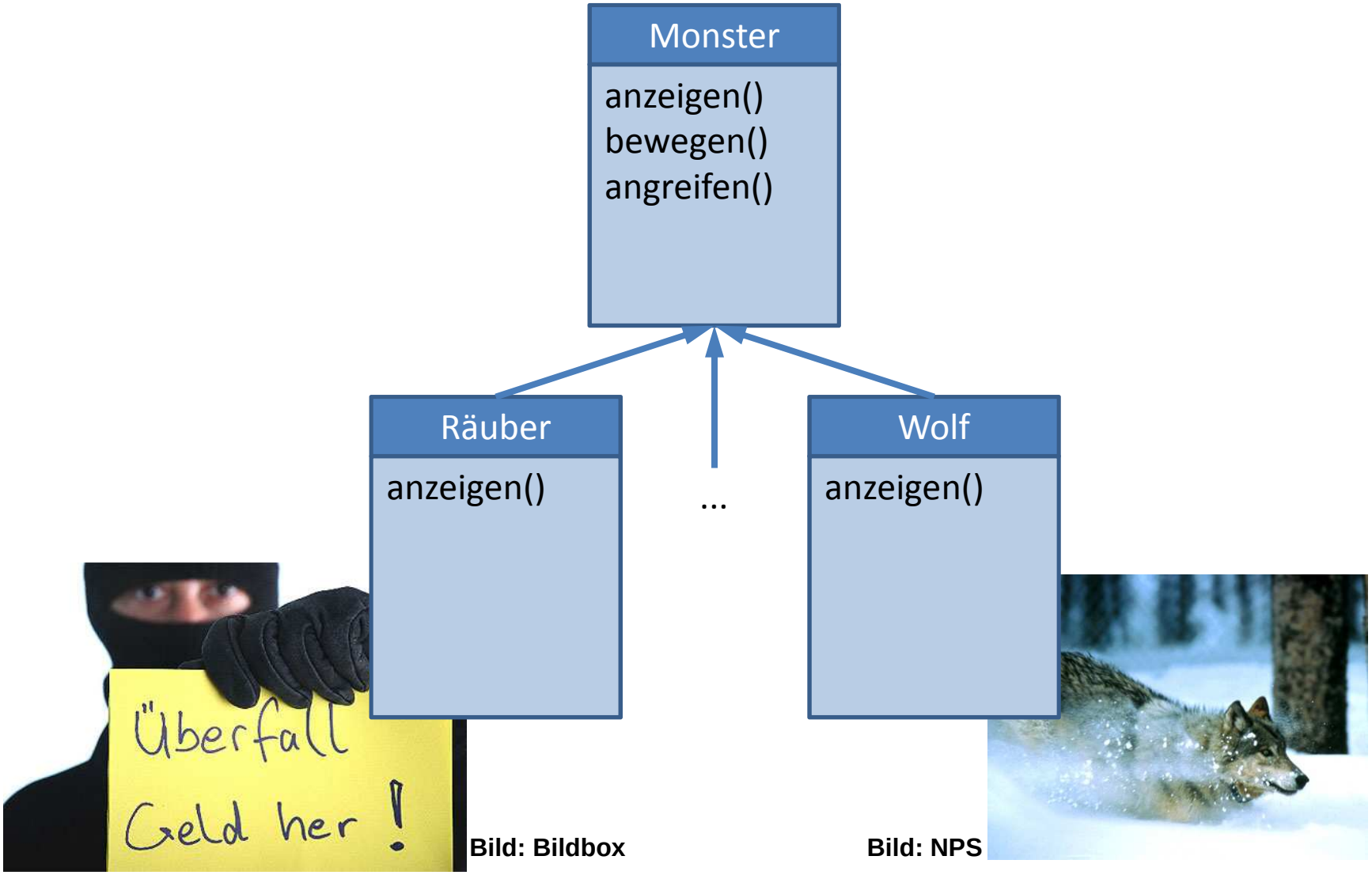


Bild: Bildbox

Bild: NPS

Neue Monster braucht das Land!

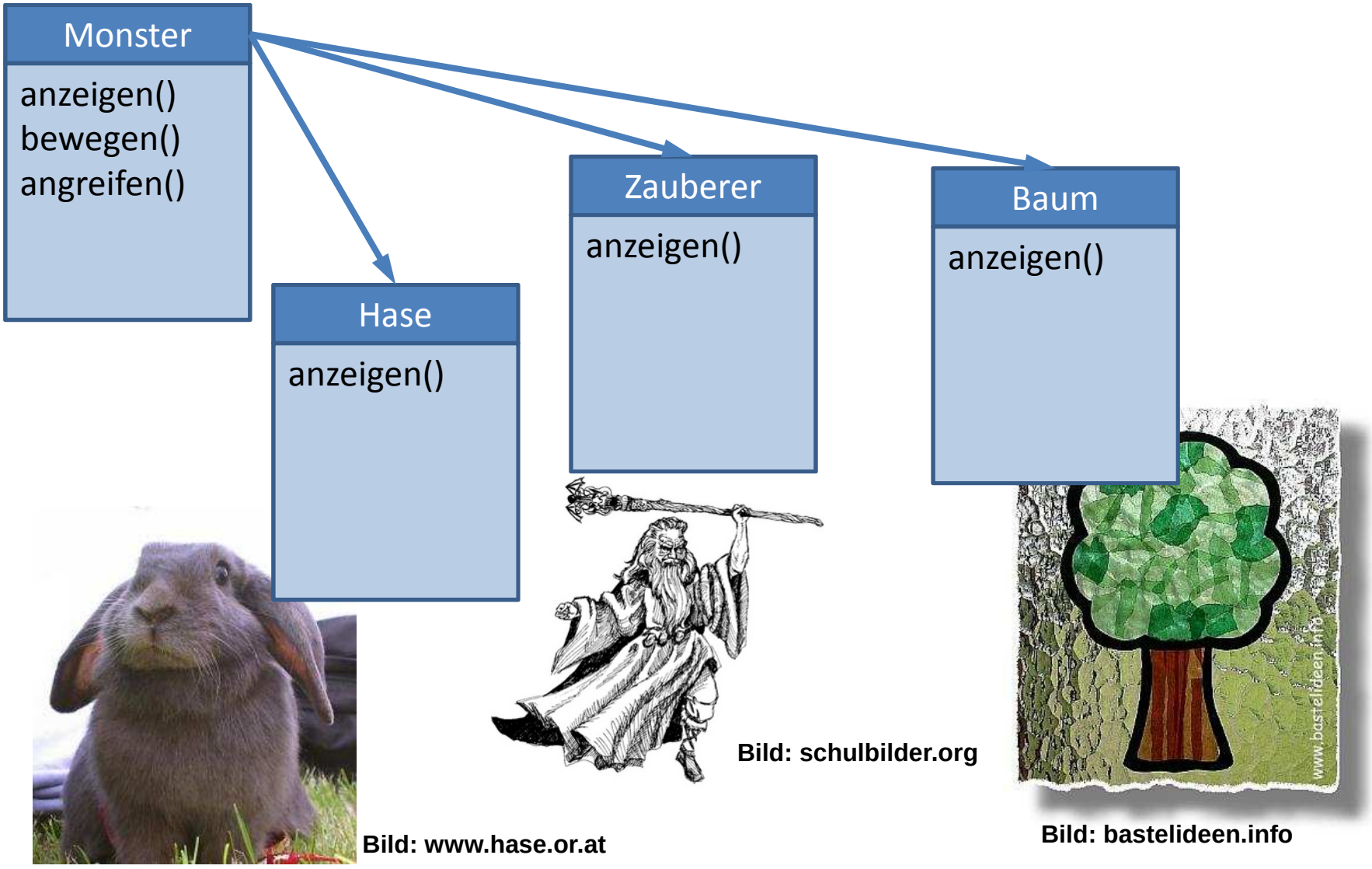
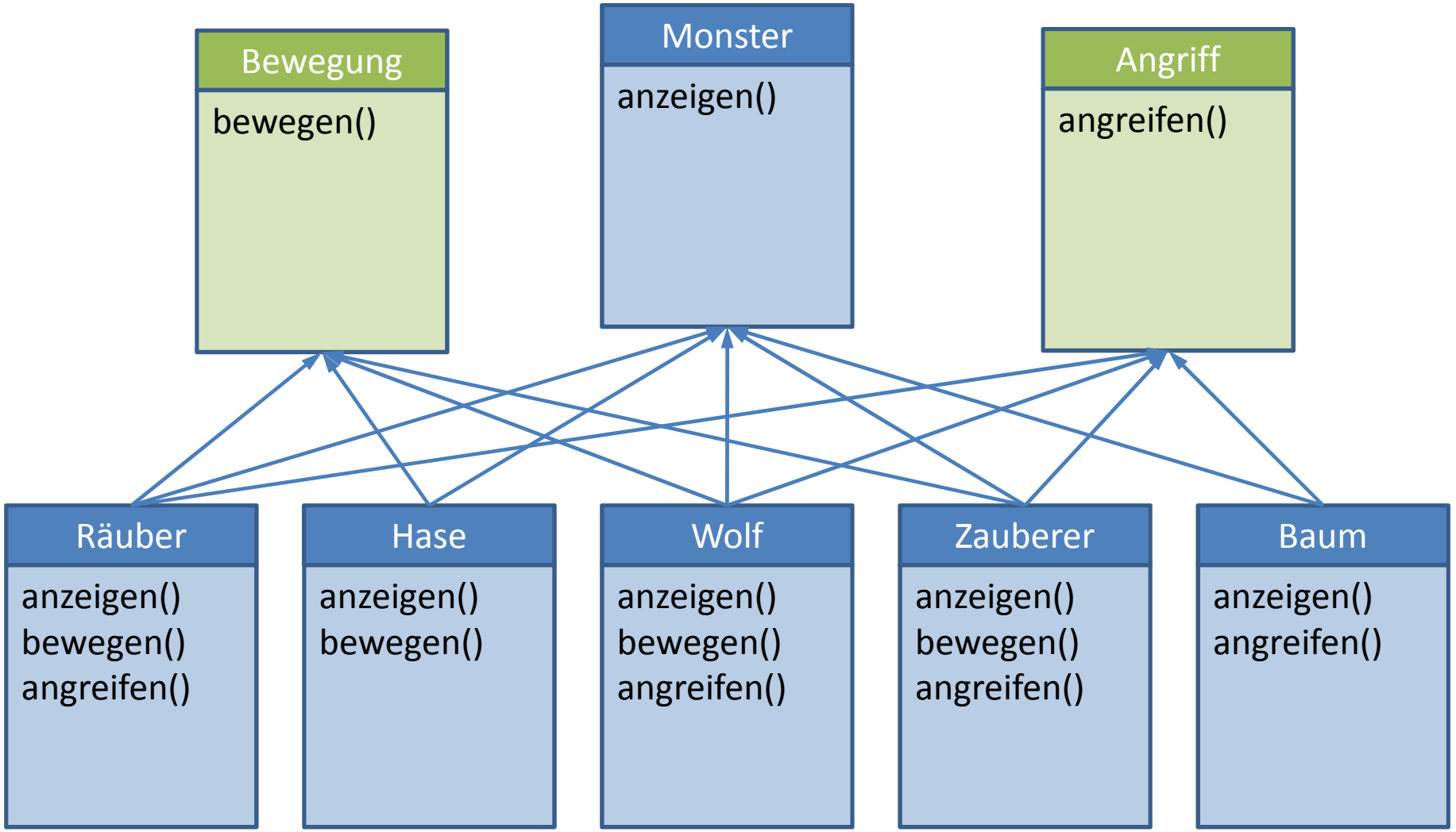


Bild: www.hase.or.at

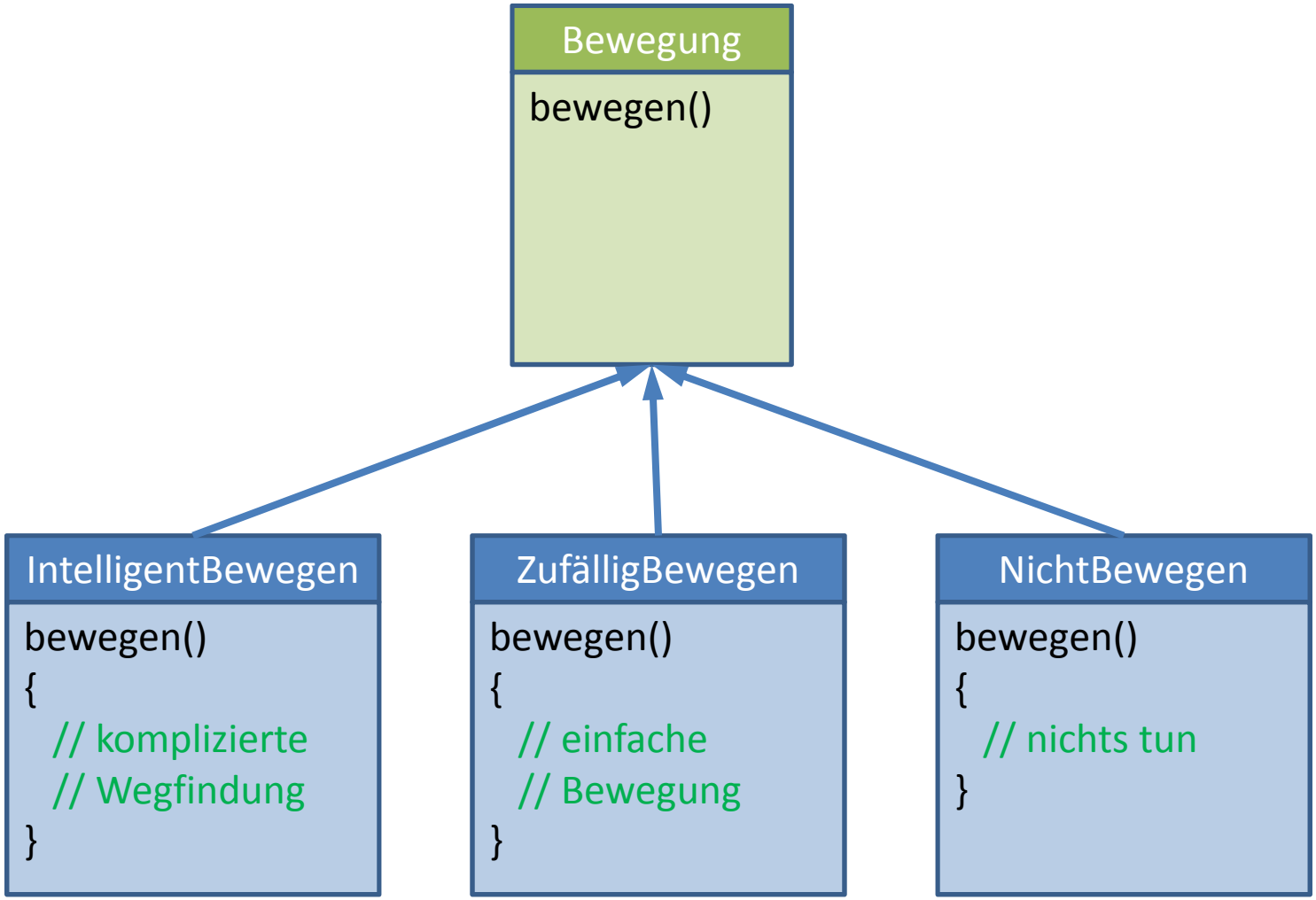
Bild: schulbilder.org

Bild: bastelideen.info

Monstervererbung



Bewegungskapseln



Das neue generische Monster

Monster
Bewegung myBewegung; Angriff myAngriff;

anzeigen()
doBewegen()
doAngriff()
{
  myAngriff.angreifen()
}

Bild: Audri Phillips

Der neue Monsterhase

```
public class Hase : Monster
{
    public Hase() // Konstruktor
    {
        myBewegung = new ZufaelligBewegen();
        myAngriff = new KeinAngriff();
    }

    public void anzeigen()
    {
        cout << "Mein Name ist Hase";
    }
}
```

Bild: Jeff West

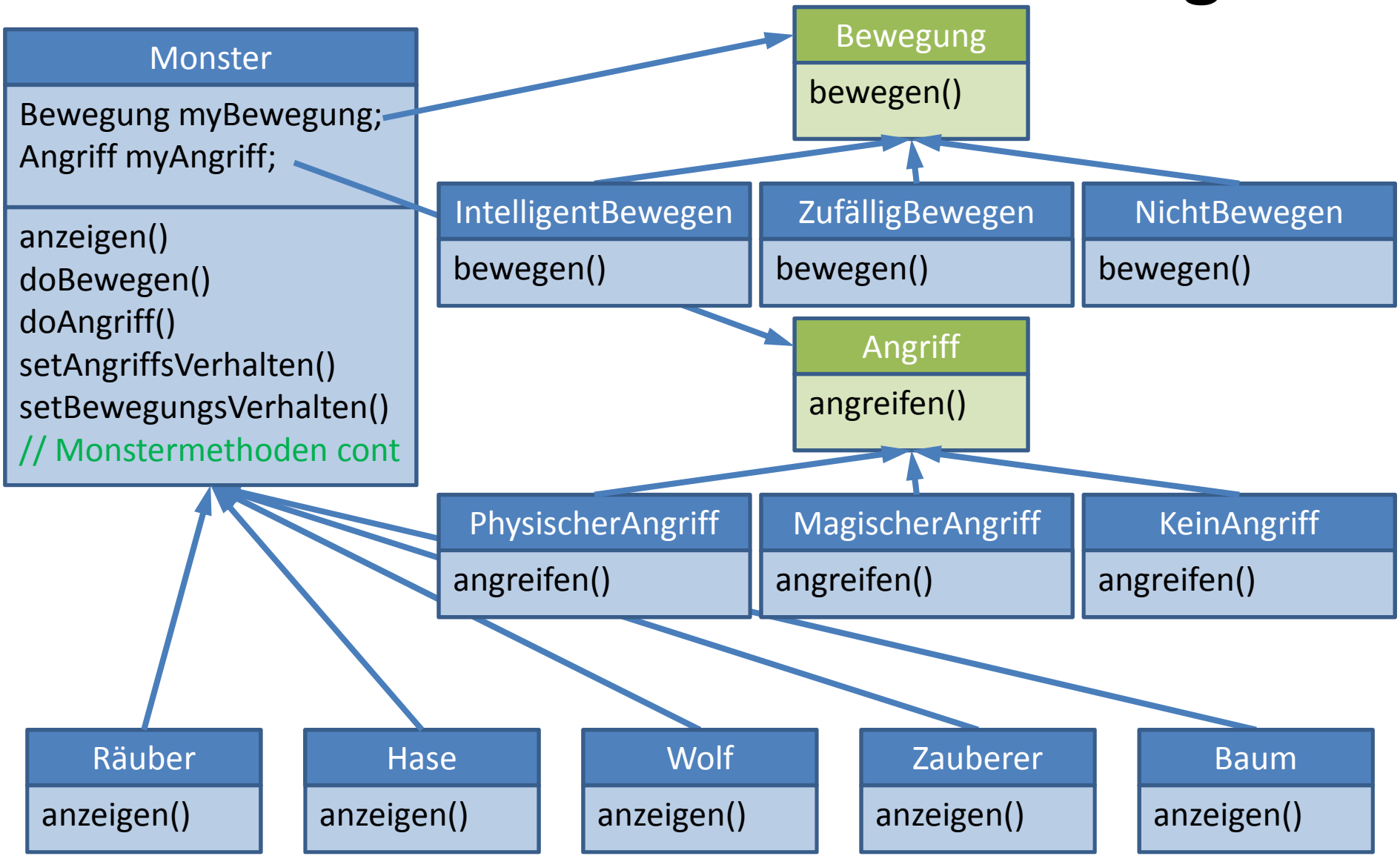
Das dynamische Monster

Verhalten dynamisch ändern

```
public void setAngriffsVerhalten
                (Angriff neuerAngriff)
{
   myAngriff = neuerAngriff;
}
```

Bild: blazemongr

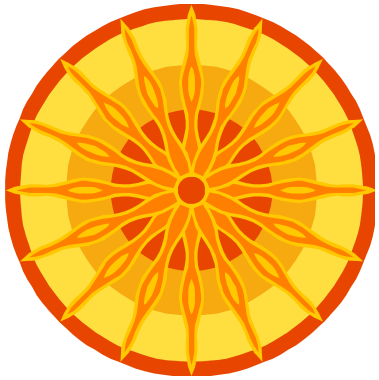
Monströse Zusammenfassung



Entwurfsprinzipien

- Identifizierung und Trennung von Aspekten die Änderungen unterliegen können und statischen Aspekten
- Programmieren auf eine Schnittstelle / Basisklasse, nicht auf konkrete Implementierung
- HAT-EIN (Komposition) ist flexibler als IST-EIN (Vererbung)
- lockere Bindung zwischen interagierenden Objekten
- Klassen sollten offen für Erweiterung aber geschlossen für Veränderung sein
- Auf Abstraktion stützen, nicht auf konkrete Klassen

Muster für Muster



- Muster
  - Kontext
    - Designsituation, die ein Problem hervorruft
  - Problem
    - Menge an Kräften, die wiederholt im gegebenen Kontext auftreten
  - Lösung
    - Balancieren der Kräfte
      - Struktur mit Komponenten und Beziehungen zwischen diesen
      - Laufzeitverhalten

Muster für Muster

- Synonym: Auch bekannt als …
- Zweck: Wozu dient das Pattern
- Kontext
- Problem
- **Lösung ((UML)-Diagramm + Flussdiagramm)**
- Motivation: Beispielszenario für Problem

Muster für Muster

- **Vorteile: Die Pros**
- **Nachteile: Die Contras**
- Verwendung: Spezielle Anwendungsgebiete
- Varianten: Abwandlungen + Bedingungen
- Quellen: Weiterführende Infos

Allgemeine Prinzipien der Softwarearchitektur

- Softwarearchitektur
  - Beschreibung der Subsysteme und Komponenten eines Softwaresystems und deren Beziehungen
  - Verschiedene Sichten zum Darstellen relevanter Eigenschaften

Allgemeine Prinzipien der Softwarearchitektur

- Komponente
  - gekapselter Teil eines Softwaresystems
  - Interface zur Kommunikation
  - Bausteine der Struktur eines Systems
  - Auf Programmiersprachenebene: Module, Klassen

Allgemeine Prinzipien der Softwarearchitektur

- Beziehung
  - Verbindung zwischen Komponenten
  - statisch: sichtbar im Quellcode
  - dynamisch: nicht direkt sichtbar

Techniken des Softwaredesigns

- Abstraktion
  - Detaillierungsgrad einer Komponente
  - Abstraktionsebene
  - Grobgranulare Komponenten setzen sich aus feingranularen Komponenten zusammen

Techniken des Softwaredesigns

- Kapselung
  - Gruppieren von Elementen einer Abstraktionsebene
  - Abgrenzung von Funktionalität und Struktur
  - Abgrenzung der Abstraktion

Techniken des Softwaredesigns

- Information Hiding
  - Verbergen der Implementation vor dem Client
  - erreicht durch Kapselung
  - welche Informationen verborgen werden ist abhängig von der Anwendung
  - Reflection weicht Prinzip auf, aber in einer definierten Art und Weise

Techniken des Softwaredesigns

- Modularisierung
  - Komposition eines Softwaresystems und Gruppierung in Subsysteme und Komponenten
  - Physische Pakete
  - Verringerung der Systemkomplexität durch wohldefinierte und dokumentierte Grenzen

Techniken des Softwaredesigns

- Aufteilung der Aufgaben (Separation of Concerns)
  - Separierung unähnlicher Verantwortlichkeiten
  - Eine Komponente mit mehreren Verantwortlichkeiten sollte von jeder anderen getrennt sein

Techniken des Softwaredesigns

- Coupling and Cohesion
  - Cohesion: Stärke der Vernetzung von Elementen und Funktionen innerhalb eines Moduls
  - Coupling: Stärke der Verbindung zwischen Modulen
  - niedrige Verbindung zu externen Komponenten
  - hoher Zusammenhalt interner Komponenten

Techniken des Softwaredesigns

- Sufficiency, Completeness and Primitiveness
  - Sufficient: Charakteristiken, die eine sinnvolle und effektive Arbeit mit der Komponente ermöglichen
  - Complete: Komponente umfasst alle relevanten Charakteristiken seines Abstraktionsgrades
  - Primitive: Einfache Implementierung der einzelnen Funktionen

Techniken des Softwaredesigns

- Separation of Interface and Implementation
  - Interface: Definiert die Funktionalität der Komponente und wie diese zu benutzen ist
  - Implementation: Der eigentliche Code zur Bereitstellung der vom Interface angebotenen Funktionalitäten
  - Veränderbarkeit
  - Information Hiding

Techniken des Softwaredesigns

- Single Point of Reference
  - Jedes Element einer Software sollte nur einmal deklariert und definiert werden
  - Vermeidung von Inkonsistenzen
  - Wartbarkeit

Techniken des Softwaredesigns

- Divide and Conquer
  - Teilen einer Aufgabe in kleinere Teile (Top-down Design)
  - Separation of Concerns

Nichtfunktionale Eigenschaften

- Changeability
  - häufig lange Lebensdauer von Softwaresystemen
  - Änderung der Anforderungen während der Lebenszeit
  - Verbesserungen und Fehlerbehebung
  - Aspekte: Wartbarkeit, Erweiterbarkeit, Umstrukturierbarkeit, Portierbarkeit

Nichtfunktionale Eigenschaften

- Interoperability
  - wohldefinierter Zugriff auf extern sichtbare Funktionalitäten und Datenstrukturen
  - Interaktion mit anderen Programmen (speziell anderer Programmiersprachen)

Nichtfunktionale Eigenschaften

- Efficency
  - Benutzung der zur Verfügung stehenden Ressourcen
    - Antwortzeiten
    - Datendurchsatz
    - Speicherbedarf
  - Kommunikation mit anderen Systemen

Nichtfunktionale Eigenschaften

- Reliability
  - Fähigkeit einer Software, ihre Funktionalität aufrechtzuerhalten
    - System- und Anwendungsfehler
  - Fehlertoleranz
    - Wiederaufbau einer Verbindung
  - Robustheit
    - Schutz der Software gegen inkorrekte Benutzung

Nichtfunktionale Eigenschaften

- Testability
  - Erleichterung der Tests durch die Architektur
    - bessere Fehlererkennung / -behebung
    - Debug-Code / Debug-Komponenten
  - Profitiert von Modularisierung

Nichtfunktionale Eigenschaften

- Reusability
  - Reduktion von Zeit und Kosten bei der Softwareentwicklung
  - verbesserte Qualität
  - Entwicklung "with reuse"
  - Entwicklung "for reuse"
  - Changability