PM1/PT Ruby: Klassenentwurf

Konzepte

Konzepte

- Entwurf nach Zuständigkeiten
- Kopplung
- Kohäsion
- Refactoring

Techniken

- öffentliche und private Objektmethoden

Einführung

- Zuul ist ein sehr einfaches Adventure Game, das sich an das von Will Crowther entwickelte Spiel „Colossal Cave Adventure“ anlehnt. (Mehr Infos: http://jerz.setonhill.edu/if/canon/Adventure.htm oder http:// www.rickadams.org/adventure)

- Zuul enthält keine grafische Oberfläche und ein sehr einfaches Modell einer virtuellen Welt.

- Das hier vorgestellte Grundgerüst ließe sich aber auf ein graphisch anspruchsvolles Spiel erweitern.

- Trotz seiner Einfachheit enthält Zuul bereits eine größere Menge von Klassen, die für die Realisierung des Spiels miteinander interagieren.

- Ziel dieser Vorlesung soll sein, ausgehend von einem ursprünglich schlechten Klassenentwurf, die Prinzipien eines guten Klassenentwurfs kennen- und anwenden zu lernen.

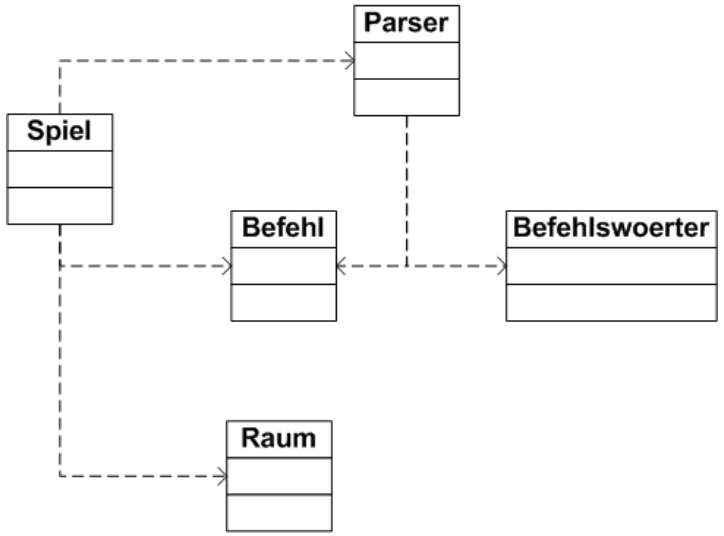
Übung

- Ü-9.1: Öffnen Sie das Projekt *V9-Zuul_schlecht* und untersuchen Sie die Anwendung. **Beantworten** Sie die Fragen:
  - Was tut die Anwendung?
  - Welche Befehle akzeptiert das Spiel?
  - Was bewirken die einzelnen Befehle?
  - Wie viele Räume gibt es in der virtuellen Umgebung?
  - Zeichnen Sie eine Karte der Räume.

- *Das Projekt heißt Zuul_schlecht, da es einige falsche Entwurfsentscheidungen enthält, die wir im Verlaufe dieser Vorlesung systematisch korrigieren wollen.*

Klassendiagramm für Zuul

- Das Projekt enthält 5 Klassen:
  - *Befehlswoerter*: definiert die gültigen Befehle für das Spiel, die sie in einem Array von Strings verwaltet.
  - *Parser*: liest Eingaben und versucht daraus Befehls-Objekte zu erzeugen.
  - *Befehl*: repräsentiert einen Befehl, den der Nutzer eingegeben hat. Die Klasse hat Methoden mit denen Befehle auf Gültigkeit geprüft werden können.
  - *Raum*: Räume modellieren die Orte in Zuul. Sie haben Ausgänge, die sie mit anderen Räumen verbinden
  - *Spiel*: Die Hauptklasse. Sie initialisiert das Spiel und liest in einer Schleife Befehle ein und führt diese aus. Die Klasse enthält die Implementierung der Befehle.



Übung

- **Ü-9.2:** Entwerfen Sie bitte ein eigenes Szenario für das Spiel! Zugelassen ist alles, bei dem sich Spieler durch unterschiedliche Räume bewegen. z.B.
  - Ein Maulwurf in seiner Höhle hat den Raum mit den Wintervorräten vergessen.
  - Ein Abenteuer kämpft in einem Verlies gegen Monster und andere Kreaturen.
  - Ein Sprengstoffkommando muss in einem Gebäude eine Bombe finden und entschärfen.
  - Eine Schatzsuche oder Schnitzeljagd in einem Labyrinth.
- Sie können die Gegenstände, die zu dem Spiel gehören beliebig erweitern (Falltüren, magische Gegenstände, Charaktere, die kooperieren, wenn Sie bestochen werden).
- Fabulieren Sie und **denken Sie dabei nicht an die Implementierung**!

Entwurfsprinzipien: Kopplung und Kohäsion

Kopplung

- beschreibt den Grad der Abhängigkeiten zwischen Klassen.
- Für gutes Softwaresystem sollte möglichst lose Koppelung zwischen Klassen angestrebt werden.
- Das heißt: Klassen sind möglichst unabhängig und kommunizieren mit anderen Klassen über möglichst schmale, wohl definierte Schnittstellen.

Kohäsion (Zusammenhalt)

- beschreibt, wie gut eine Programmeinheit eine logische Aufgabe abbildet.
- In einem System mit hoher Kohäsion ist jede Programmeinheit (Methode, Klasse, Modul, Teilsystem) für eine wohl definierte Aufgabe verantwortlich .
- Ein guter Klassenentwurf weist einen hohen Grad von Kohäsion auf.

Ihr eigenes Zuul Projekt

- Ü-9.4: Kopieren Sie bitte *V9-Zuul_schlecht* in eine eigenes neues Projekt und zeichnen Sie auf Papier eine Karte der Welt, die Sie in Ü-9.3 entworfen haben!

- Nennen Sie es nicht „schlecht“, da wir die erste Lösung soweit korrigieren, dass sie eine gute oder zumindest bessere Lösung wird.

- Passen Sie die Methode *raeume_anlegen* in ihrem neuen Projekt so an, dass sie die Räume und Ausgänge in Ihrer Welt anlegt.

Code-Duplizierung

- (ein und derselbe Quelltext erscheint mehr als einmal in einer Anwendung) ist ein Indiz für einen schlechten Entwurf und sollte auf jeden Fall vermieden werden.

- Bei der Wartung eines Programmes kommt es häufig vor, dass doppelter Code nur an einer Stelle geändert wird. Das führt zu Fehlern, die sich leicht vermeiden lassen, wenn in der Entwicklung doppelter Code beseitigt wird.

- In *V9-Zuul_schlecht* treffen wir die Code-Duplizierung in der Klasse *Spiel* in den Methoden *willkommenstext* und *wechsle_ raum* an.

- In beiden Methoden werden für den *@aktuellen_raum* die Ausgänge in exakt der gleichen Art und Weise ausgegeben.

- Hinweis: An dieser Stelle **müssen** Sie den Ausdruck der Quelltextes für *Spiel* vor sich liegen haben, oder in der Vorlesung auf Ihrem Rechner das Projekt *Zuul_schlecht* geöffnet haben, damit Sie die entsprechenden Stellen markieren und kommentieren können.

Code-Duplizierung

```
puts("Wir sind
    #{@aktueller_raum.beschreibung()}")
print("Ausgaenge nach: ")
print("#{'north ' if
    @aktueller_raum.nordausgang}")
print("#{'east ' if
    @aktueller_raum.ostausgang}")
print("#{'south ' if
    @aktueller_raum.suedausgang}")
print("#{'west' if
    @aktueller_raum.westausgang}" )
puts()
```

- Beide Methoden enthalten den neben-stehenden Quelltext zur Ausgabe der Rauminfo.

- Ü-9.5: Beseitigen Sie bitte die Code-Duplizierung indem Sie eine Methode *raum info_ausgeben* implementieren und diese anstelle der Ausgaben in den Methode *willkommenstext* und *wechsle_raum* verwenden!

Erweiterungen von Zuul

- *Zuul_schlecht* funktioniert, ist aber an einigen Stellen schlecht entworfen.

- Sobald wir Änderungen an einem Projekt vornehmen, werden wir feststellen, dass Änderungen an einem schlecht entworfenen Projekt aufwendiger sind als solche an einem gut entworfenen Projekt.

- **Erste Änderung**: Wir führen neue Richtungen (**up, down**) ein, um uns in einem mehrstöckigen Gebäude bewegen zu können.

- Mindestens zwei Klassen sind von der Änderung betroffen: *Spiel* und *Raum*

Neue Richtungen für Ausgänge in Zuul

Auswirkung auf Klasse *Raum*

- In der Klasse *Raum* werden die Ausgänge an zwei Stellen verwendet.
- In der Methode *setze_ausgaenge*, in der die Ausgänge eines Raumes initialisiert werden.
- In den Attribut Readern zu Beginn der Klasse.
- An **beiden Stellen** müssten wir die neuen Ausgänge aufnehmen.

```
class Raum

  attr_reader :beschreibung,
    :nordausgang,:ostausgang,:suedausgang,
    :westausgang

  # Erzeuge eine Raum mit einer
    Beschreibung
  # Anfangs hat ein Raum keine Ausgänge
  def initialize(beschreibung)
    @beschreibung=beschreibung
  end

  def setze_ausgaenge(norden, osten,
    sueden, westen)
    @nordausgang = norden
    @ostausgang = osten
    @suedausgang = sueden
    @westausgang = westen
  end
end
```

Neue Richtungen für Ausgänge in Zuul

Auswirkung auf Klasse *Spiel*

- In der Klasse *Spiel* wird in den Methoden
  - *raeume_anlegen,*
  - *willkommenstext*
  - *wechsle_raum*

  Bezug auf die Ausgänge von Räumen genommen.

- *raeume_anlegen* definiert die Ausgänge der Räume

- *willkommenstext* gibt die Ausgänge von *@aktueller_raum* aus.

- *wechsle_raum* benutzt den aktuellen Raum, um den nächsten Raum zu finden und die Ausgänge eines Raumes auszugeben.

Neue Richtungen für Ausgänge in Zuul

Auswirkung auf Klasse *Spiel*

```
def raeume_anlegen()
    # Räume erzeugen
    draussen = Raum.new("auf dem Campus")
    hoersaal = Raum.new("in BT-5 037")
    cafete = Raum.new("in Oh it's ...")
    labor = Raum.new("im Praktikumsraum")
    buero = Raum.new("im Praktikumsbuero")
    # Räume verbinden
    draussen.setze_ausgaenge(nil,hoersaal,labor,cafete)
    hoersaal.setze_ausgaenge(nil,nil,nil, draussen)
    cafete.setze_ausgaenge(nil,draussen,nil,nil)
    labor.setze_ausgaenge(draussen,buero,nil,nil)
    buero.setze_ausgaenge(nil,nil,nil,labor)
    @aktueller_raum = draussen
 end
```

Neue Richtungen für Ausgänge in Zuul

Auswirkung auf Klasse *Spiel*

```
def wechsle_raum(befehl)
    if (!befehl.hat_argument?())
      puts "Keine Richtung angegeben"
      return
    end
    richtung = befehl.argument()
    case richtung
    when 'north'
      naechster_raum = @aktueller_raum.nordausgang
    when 'east'
      naechster_raum = @aktueller_raum.ostausgang
    #... analog für die anderen Ausgänge
    end
    if(naechster_raum.nil?())
      puts("Kein Ausgang nach #{richtung}")
    else
      @aktueller_raum = naechster_raum
    #.. hier steht die bereits bekannte
    # Ausgabe
  end
```

Zu enge Koppelung von Klassen

- Dass alle Ausgänge an so vielen Stellen im Quelltext aufgezählt werden ist ein eindeutiges Zeichen für einen **schlechten** Entwurf.
- Stellen Sie sich vor, Sie wollen neben den Richtungen *up* und *down* auch noch die Richtungen *northwest, northeast*, etc. aufnehmen.
- Wir entscheiden uns für eine Lösung, die die Ausgänge in einem *Hash* speichert.
- Theoretisch sollte diese Änderung nur die Klasse *Raum* betreffen. Dies trifft in unserem Fall aber nicht zu, da die Klasse *Spiel* an vielen Stellen über die Attributreader Bezug auf die Instanzvariablen von Raum nimmt.
- Wir haben also hier einen klaren Fall von **zu enger Koppelung** zwischen zwei Klassen.

Kapselung

- Eines der Hauptursachen für die zu enge Koppelung ist, dass die Klasse ***Raum*** anderen Klassen Zugriff auf ihre Instanzvariablen und damit Zugriff auf die interne Struktur und die Implementierung der Ausgänge gibt.
- Dies verletzt ein weiteres fundamentales Prinzip für Klassenentwurf: die **Kapselung.**
- **Gute Kapselung** reduziert die Koppelung und führt zu besseren Entwürfen.
- Kapselung besagt, dass nur das **Was** einer Klasse (was sie anbietet) nach außen sichtbar wird, nicht das **Wie** (also die Implementierung).
- **Vorteil**: Wenn wir nur das **Was** anbieten, können wir die Implementierung nachträglich ändern, ohne dass andere Klassen Änderungen vornehmen müssen.
- Ein **elementarer** Schritt, um das Was vom Wie zu trennen, ist der **Verzicht** auf extern sichtbare **Attribut-Reader** und die Abstraktion auf allgemeinere sondierende Methoden.

Kapselung

- Ein **elementarer** Schritt, um das Was vom Wie zu trennen, ist der **Verzicht** auf extern sichtbare **Attribut-Reader** und die Abstraktion auf allgemeinere sondierende Methoden, wie im Beispiel rechts die Methode *ausgang(richtung)*.

```
class Raum

  def initialize(beschreibung)
    @beschreibung=beschreibung
  end

 # ausgelassen...

  def ausgang(richtung)
    case richtung
    when 'north'
      return @nordausgang
    when 'east'
      return @ostausgang
    when 'south'
      return @suedausgang
    when 'west'
      return @westausgang
    end
  end
end
```

Kapselung durch Einschränkung der Methodensichtbarkeit

- Eine bekannte Technik für die Kapselung der Interna einer Klasse sind **private Objektmethoden**.
- private Objektmethoden können nur vom Objekt *self* ausgeführt werden. Für alle anderen Objekte, auch nicht die Objekte derselben Klasse, sind diese Methoden **nicht sichtbar**.
- In Zuul ist die Methode *willkommenstext* eine private Objektmethode in der Klasse *Spiel*.
- Es ist wenig sinnvoll, dass externe Klassen diese Methode mitten im Spiel aufrufen.
- Private Objektmethoden werden in Ruby durch das Schlüsselwort *private*, dem die Symbole der Methodennamen folgen, ausgezeichnet.

```
private :willkommenstext
```

- Ü-9.6: Welche Methoden der Klasse *Spiel* sind ebenfalls Kandidaten für private Objektmethoden?

Kapselung durch Einschränkung der Methodensichtbarkeit

- Alternativ können Objektmethoden als private Methoden ausgezeichnet werden, in dem das Schlüsselwort *private* der Methode in einer Zeile vorangestellt wird.

- **Vorsicht**: Alle Methoden, die nach der Nennung von *private* definiert werden sind private.

- Die einzige Möglichkeit, die private Sichtbarkeit für nachfolgende Methoden aufzuheben, ist das explizite Kennzeichnen von Methodendefinitionen mit dem Schlüsselwort *public*. (oder *protected* später).

- Im Beispiel sind die Methoden *m1-m3 private*, die Methoden *m4-m5 public*

```
class Sichtbarkeiten

  private
  def m1()     ...   end

  def m2()     ...   end

  def m3()     ...    end

  public
  def m4()     end

  def m5()   end
end
```

Regeln zur Methodensichtbarkeit

- **alle Methoden** in eigenen Klassen sind per default ***public***
- **außer** der Methode ***initialize***, diese ist immer ***private.***

```
sicht =  Sichtbarkeiten.new()
#sicht.initialize()    Fehler
#sicht.m1()
#sicht.m2()
#sicht.m3()
sicht.m4()
sicht.m5()
```

Regeln zur Methodensichtbarkeit

- Private Methoden dürfen kein Empfängerobjekt haben.

```
class Sichtbarkeiten
  private
  def m1()  end
  def m2()  end
  def m3()  end
  public
  def m4()
    m1() # Ok
  end
  def m5()
    self.m1() # Fehler
  end
end


sicht.m4()
sicht.m5()  #Fehler
```

Regeln zur Methodensichtbarkeit

- *puts, p* sind private Methoden von *Object*.
- Wenn ein Rubyscript ausgeführt wird, dann wird eine Instanz von *Object* mit Namen *main* erzeugt. Alle Methodenaufrufe ohne expliziten Empfänger werden an *main* geschickt.
- Daher ist *puts/p* ohne Empfänger ok, *self.puts / self.p* liefert einen Fehler.

```
puts(self.class)
puts(self)
puts("Hello")
p("Hello")
self.puts("Hello")    # Fehler
self.p(" Hello")      # Fehler
```

```
Object
main
Hello
"Hello"
D:/haw/vorlesungen/rubypr1/sose10/workspace/V9-Zuul_schlecht/sichtbarkeiten/Sichtbarkeiten.rb:29: private method `puts' called for main:Object (NoMethodError)
```

Erweiterungen von Zuul

- An dieser Stelle passen wir auch die Methode *wechsle_raum* der Klasse *Spiel* an.
- Diese wird durch Verwendung der Methode *ausgang(richtung)* gleichzeitig um einige Zeilen kürzer.

```
def wechsle_raum(befehl)
    if (!befehl.hat_argument?())
      puts "Keine Richtung angegeben"
      return
    end
    richtung = befehl.argument()
    # Hier stand vorher ein längliches case
    naechster_raum =
         @aktueller_raum.ausgang(richtung)
    if(naechster_raum.nil?())
      puts("Kein Ausgang nach #{richtung}")
    else
      @aktueller_raum = naechster_raum
     # Hier steht noch die längliche
     # Ausgabe. Aber das ändern wir
     # auch noch
   end
end
```

Erweiterungen von Zuul

- Ü-9.7: Ändern Sie bitte die Implementierung der Klasse *Raum* wie gezeigt und passen Sie in der Klasse *Spiel* die Methode *wechsle_raum* an!

  Passen Sie bitte ebenfalls die Methode *rauminfo_ausgeben* so an, dass sie für die Ausgabe von Räumen eine Methode von *Raum* nutzt, die diesen als lesbare Zeichenkette zurückgibt!

  Implementieren Sie dazu die geeignete Methode in *Raum*!

Erweiterungen von Zuul

- Nach den Änderungen haben wir in der Klasse Raum keine Attribut-Reader mehr.
- Anstelle dessen eine sondierende Methode *ausgang(richtung)*

```
def ausgang(richtung)
    case richtung
    when 'north'
      return @nordausgang
    when 'east'
      return @ostausgang
    when 'south'
      return @suedausgang
    when 'west'
      return @westausgang
    end
  end
```

Erweiterungen von Zuul

- Nach den Änderungen haben wir in der Klasse ***Raum*** eine Methode, die für einen Raum eine lesbare Zeichenkette erzeugt:

```
def to_s()
   "Wir sind  #{@beschreibung}\n"+
   "Ausgaenge nach: " +
    "#{'north ' if @nordausgang}" +
   "#{'east ' if @ostausgang}" +
   "#{'south ' if @suedausgang}" +
   "#{'west' if @westausgang}\n"
 end
```

Erweiterungen von Zuul

- Nach den Änderungen haben wir die Methode *willkommenstext* in der Klasse *Spiel* reduziert auf:

```
def willkommenstext()
   puts("Willkommen bei Zuul\n" +
        "-- einem noch sehr langweiliges
   Spiel\n" +
        "Geben Sie 'help' ein, wenn Sie
   Hilfe brauchen\n" +
        "\n")
   rauminfo_ausgeben()
end
```

Erweiterungen von Zuul

- Nach den Änderungen haben wir die Methode *rauminfo_ausgeben* in der Klasse *Spiel* reduziert auf:

```
def rauminfo_ausgeben()
  puts("#{@aktueller_raum}")
end
```

Erweiterungen von Zuul

- Nach den Änderungen haben wir die Methode *wechsle_raum* in der Klasse *Spiel* reduziert auf:

```
def wechsle_raum(befehl)
   if (!befehl.hat_argument?())
     puts "Keine Richtung angegeben"
     return
   end
   richtung = befehl.argument()
   naechster_raum =
   @aktueller_raum.ausgang(richtung)
   if(naechster_raum.nil?())
     puts("Kein Ausgang nach #{richtung}")
   else
     @aktueller_raum = naechster_raum
     rauminfo_ausgeben
   end
 end
```

Erweiterung von Zuul

- Nach den Änderungen in den Klassen *Raum* und *Spiel* können wir nun die Klasse *Raum* um weitere Richtungen für Ausgänge erweitern, indem wir als Erstes einen *Hash* für die Ausgänge einführen.

- Wir müssen **keine Änderungen** in der Klasse *Spiel* vornehmen, da wir Spiel durch **lose Koppelung** unabhängig von der Implementierung in Raum gemacht haben.

- Die Umstellung auf den *Hash @ausgaenge* hat den positiven Effekt, dass sich die Implementierung der Methode *ausgang* auf einen Einzeiler verkürzt.

```
class Raum
  def initialize(beschreibung)
    @beschreibung=beschreibung
    @ausgaenge = {}
  end
  def setze_ausgaenge(norden, osten,
   sueden, westen)
    @ausgaenge['north'] = norden if norden
    @ausgaenge['east'] = osten if osten
    @ausgaenge['south'] = sueden if sueden
    @ausgaenge['west'] = westen if westen
  end
  def ausgang(richtung)
    return @ausgaenge[richtung]
  end
  def to_s()
    "Wir sind  #{@beschreibung}\n"+
    "Ausgaenge nach:
    "#{@ausgaenge.keys().join(' ')}"
  end
end
```

Erweiterung von Zuul

- Der Grund, warum wir alle diese Änderungen vorgenommen haben, war der Wunsch zusätzliche Richtungen für Ausgänge definieren zu können.

- Jetzt haben wir noch ein kleines Problem: In der Methode *setze_ausgaenge* sind die Richtungen noch „hart codiert“.

- Um diese letzte Hürde für die Einführung beliebiger weiterer Richtungen für Aus-gänge zu beseitigen, ersetzen wir diese Methode durch eine, in der wir zu einer Richtung einen Nachbarraum eintragen: *setze_ausgang(richtung, nachbar_raum)*

```
class Raum
  def initialize(beschreibung)
    @beschreibung=beschreibung
    @ausgaenge = {}
  end

  def setze_ausgang(richtung, nachbar_raum)
    @ausgaenge[richtung] = nachbar_raum
  end

  def ausgang(richtung)
    return @ausgaenge[richtung]
  end

  def to_s()
    "Wir sind  #{@beschreibung}\n"+
    "Ausgaenge nach:
    "#{@ausgaenge.keys().join(' ')}"
  end
end
```

Erweiterung von Zuul

- Jetzt müssen wir in der Klasse Spiel in der Methode *raeume_anlegen* noch auf die Methode *setze_ausgang* umstellen.

```
def raeume_anlegen()
    # Räume erzeugen
    draussen = Raum.new("auf dem Campus")
    hoersaal = Raum.new("in BT-5 037")
    cafete = Raum.new("in Oh it's ...")
    labor = Raum.new("im Praktikumsraum")
    buero = Raum.new("im Praktikumsbuero")

    # Räume verbinden
    draussen.setze_ausgang('east',hoersaal)
    draussen.setze_ausgang('south',labor)
    draussen.setze_ausgang('west',cafete)
    hoersaal.setze_ausgang('west',draussen)
    cafete.setze_ausgang('east',draussen)
    labor.setze_ausgang('north',draussen)
    labor.setze_ausgang('east',buero)
    buero.setze_ausgang('west',labor)
    @aktueller_raum = draussen
end
```

Erweiterung von Zuul

- Wir sind jetzt soweit, um vom Hörsaal in den Keller ab- und die Bibliothek aufzusteigen.

```
def raeume_anlegen()
  # Räume erzeugen
  # ... wie vorher
  hoersaal = Raum.new("in BT-5 037")
  bibliothek = Raum.new('in der Bibliothek')
  zentralwerkstatt = Raum.new(
                  'in der Zentralwerkstatt')
  # ... wie vorher

  # Räume verbinden
  # ...  wie vorher
  hoersaal.setze_ausgang('west',draussen)
  hoersaal.setze_ausgang('up', bibliothek)
  hoersaal.setze_ausgang(
                  'down', zentralwerkstatt)
  zentralwerkstatt.setze_ausgang(
                  'up',hoersaal)
  bibliothek.setze_ausgang(
                  'down', hoersaal)
  # ...  wie vorher
  @aktueller_raum = draussen
end
```

Entwurf nach Zuständigkeiten

- Konsequente Kapselung ist ein Entwurfsprinzip, um die Koppelung zwischen Klassen zu reduzieren und den Aufwand bei Änderungen in einer Anwendung erheblich zu reduzieren.

- Ein anderer wichtiger Aspekt, der den Grad der Koppelung beeinflusst, wird unter dem Begriff **Entwurf nach Zuständigkeiten** (*responsibility-driven design*) zusammengefasst.

- **Entwurf nach Zuständigkeiten** ist ein Entwurfsprozess, bei dem jeder Klasse eine klare Verantwortlichkeit zugewiesen wird.

- Dieser Prozess kann benutzt werden, um festzulegen, welche Klasse für welche Funktionen in einer Anwendung zuständig ist.

- Ein Beispiel für **falsche Zuständigkeit**, war die Ausgabe der Rauminfo-Details in der Klasse Spiel.

- Es liegt in der **Zuständigkeit einer jeden Klasse**, darüber zu entscheiden, wie Objekte als lesbare Zeichenkette dargestellt werden.

Änderungen lokal halten

- Eines der Hauptziele eines guten Klassenentwurfs lautet: **Änderungen lokal halten**.

- Eine Änderung an einer Klasse sollte möglichst geringe Auswirkungen auf andere Klassen haben.

- Dieser Anspruch lässt sich weitestgehend erfüllen, indem wir die Entwurfsprinzipien

  - Kapselung
  - lose Koppelung
  - und hohe Kohäsion

  anwenden.

Implizite Koppelung

- Die Verwendung von öffentlichen Attribut-Readern führt zu einer unnötig engen Koppelung zwischen Klassen.
- Es gibt allerdings noch eine weitere noch unangenehmere Form der Koppelung, die **implizite Koppelung**.
- Implizite Koppelung liegt vor, wenn sich eine Klasse auf interne Information einer anderen Klasse verlässt, diese aber nicht offensichtlich erkennbar ist.
- Wir werden diese implizite Koppelung entdecken, wenn wir versuchen, dem Spiel weitere Befehle hinzuzufügen.

- Wir wollen den Befehl *look* in die Befehlsliste aufnehmen.
- *look* ermöglicht die erneute Ausgabe eines Raumes.
- Dazu erweitern wir die Klassenkonstante *GUELTIGE_BEFEHLE* in der Klasse *Befehlswoerter* um die Zeichenkette „*look*“.

```
class Befehlswoerter
   GUELTIGE_BEFEHLE = [ "go",
   "quit", "help", "look"]
end
```

Implizite Koppelung

- Wenn wir das Spiel mit der erweiterten Liste der Befehlswörter starten und *look* eingeben, dann wird zunächst `"Was soll ich tun?"` ausgegeben.
- Der Grund: in der Methode *verarbeite_befehl* in *Spiel*, wird er Befehl *look* noch nicht behandelt.
- Wir erweitern *verarbeite_befehl* in *Spiel*, und implementieren die Methode *umsehen.*
- Dadurch haben wir mit kleinen Änderungen eine neue Methode in unser Spiel integriert.

```
def verarbeite_befehl(befehl)
    moechte_beenden = false;
    if befehl.unbekannt?()
      puts("Was soll ich tun?")
      return moechte_beenden
    end
    case befehl.befehlswort()
    # ... bekannte Befehle
    when 'look'
      moechte_beenden = umsehen()
    end
    return moechte_beenden
end

# Info des Raums ausgeben
def umsehen()
  puts("#{@aktueller_raum}")
end
```

Implizite Koppelung

- Auf den ersten Blick scheinen wir fertig zu sein, aber:
- Wenn wir jetzt *'help'* eintippen, offenbart sich die Misere. Der neue Befehl taucht in der Liste der möglichen Befehle nicht auf.
- Der **Grund** ist **implizite Koppelung**: Wenn sich die Befehle ändern, muss sich auch immer der Hilfetext ändern. Die Koppelung ist implizit, da es im Quelltext keinen Hinweis gibt, dass diese Abhängigkeit besteht.

```
Willkommen bei Zuul
-- einem noch sehr langweiligen Spiel
Geben Sie 'help' ein, wenn Sie Hilfe
    brauchen

Wir sind  auf dem Campus
Ausgaenge nach: east south west
>
look
Wir sind  auf dem Campus
Ausgaenge nach: east south west
>
help
Folgende Befehle zur Verfügung:
  go quit help
>
```

Beseitigung der impliziten Koppelung mit dem Entwurf nach Zuständigkeiten

- Wenn wir kurz darüber nachdenken, dann ist es die Klasse *Befehlswoerter*, die die Befehle kennt.
- Es liegt in der Zuständigkeit dieser Klasse die Liste der Befehle auszugeben, oder als Zeichenkette zu liefern.

```
class Befehlswoerter
  # Eine Klassenkonstante für
   gültige Befehle
  GUELTIGE_BEFEHLE = [ "go", "quit",
   "help", "look"]

  # Prüfe ob ein Befehlswort eine
  # gültiges Wort ist
  def ist_befehl?(wort)
    return
   GUELTIGE_BEFEHLE.include?(wort)
  end

  def alle_ausgeben()
    GUELTIGE_BEFEHLE.each {|befehl|
   print befehl + ' '}
  end
end
```

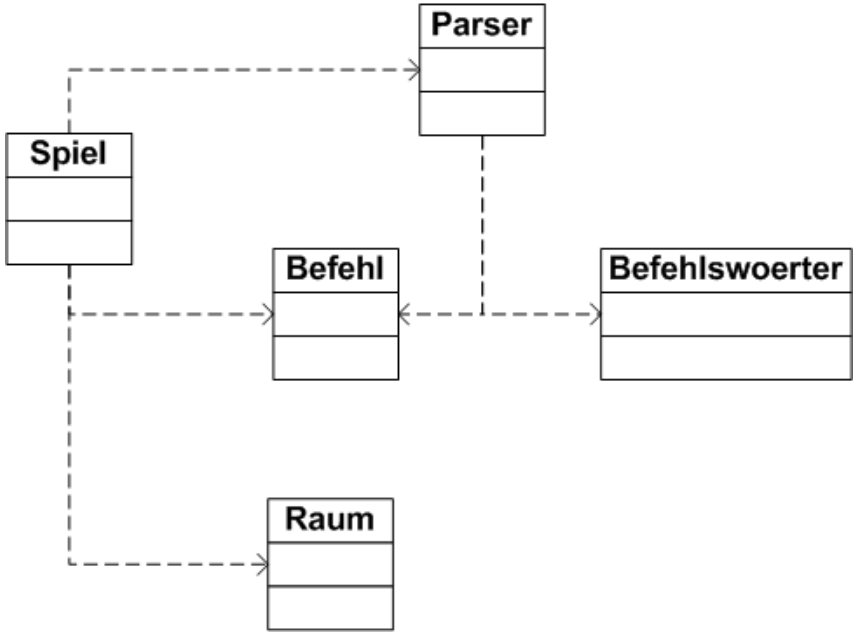
Beseitigung der impliziten Koppelung mit dem Entwurf nach Zuständigkeiten

- Jetzt haben wir ein weiteres Problem. Die Klasse Spiel weiß nichts über die Klasse *Befehlswoerter*. Nur die Klasse *Parser* kennt *Befehlswoerter.*

- Wir könnten nun eine Abhängigkeit von *Spiel* zu *Befehlswoerter* herstellen und in *Spiel* die Methode *alle_ausgeben* nutzen.

- Dadurch erhöhen wir die Koppelung: *Spiel* erhält eine zusätzliche Abhängigkeit zur Klasse *Befehlswoerter* .

- Die **Anzahl der Abhängigkeiten** zwischen Klassen ist ein **Maß für die Koppelung**. Daher sollte die Zahl möglichst gering gehalten werden.



Beseitigung der impliziten Koppelung mit dem Entwurf nach Zuständigkeiten

- Nur die Klasse *Parser* weiß etwas über *Befehlswoerter*. *Spiel* kennt nur den *Parser* aber nicht die Details der Imple-mentierung der Klasse im Umgang mit Befehlen.
- Daher werden wir in der Klasse Parser eine Methode *zeige_befehle* aufnehmen, die dem Spiel ermöglicht, vermittelt über den Parser die Liste der Befehle auszugeben.
- Wir erhalten die ursprünglichen Assoziationen und eine möglichst hohe Entkoppelung.

```
require "zuul/Befehl"
require "zuul/Befehlswoerter"

class Parser

  def initialize()
    @befehle = Befehlswoerter.new()
  end

  # Liefert den nächsten Befehl eines
  # Benutzers
  def liefere_befehl()
    ...
  end

  def befehle_ausgeben
    @befehle.alle_ausgeben()
  end
end
```

Vorausdenken

- Eine bessere Lösung wäre es, die Ausgabe der Befehle der nutzenden Klassen zu überlassen und die Befehle in *Befehlswoerter* nur als Zeichenkette zu liefern.

- Zeichenketten können in beliebige Ausgabe integriert werden und erlauben das Kombinieren der Ausgaben verschiedener Klassen.

- Ü-9.8: Implementieren Sie bitte die Änderungen in Ihrem Zuul-Projekt und passen die Methode *befehle_ausgeben* in der Klasse *Parser* an diese Änderung an!

```
class Befehlswoerter
  # Eine Klassenkonstante für
   gültige Befehle
  GUELTIGE_BEFEHLE = [ "go", "quit",
   "help", "look"]

  # Prüfe ob eine Befehlswort eine
   gültiges Wort ist
  def ist_befehl?(wort)
    return
   GUELTIGE_BEFEHLE.include?(wort)
  end

  def to_s()
   return GUELTIGE_BEFEHLE.join(' ')
  end
end
```

Kohäsion bei Methoden

- Eine **Methode mit hoher Kohäsion** ist verantwortlich für genau eine wohldefinierte Aufgabe.

- Beispiele für Methoden mit hoher Kohäsion:
  - *willkommenstext*
  - *verarbeite_befehl*

  in der Methode *spielen*.

- Wenn der Quelltext beider Methoden in der Methode *spielen* stehen würde, wäre der Quelltext schwerer zu verstehen und schlechter zu warten, als wenn kurze Methoden mit hoher Kohäsion verwendet werden.

Kohäsion bei Klassen

- Eine **Klasse mit hoher Kohäsion** definiert genau eine wohldefinierte Einheit.
- Als Beispiel für die Kohäsion von Klassen sollen jetzt Gegenstände in Zuul eingeführt werden
- Gegenstände haben eine Beschreibung und ein Gewicht haben. Das Gewicht entscheidet darüber, ob ein Gegenstand aufgehoben werden kann.
- **Naiver Ansatz**: Wir führen zwei weitere Instanzvariablen *@gegenstand_beschreibung* und *@gegenstand_gewicht* in der Klasse Raum ein.
- Das ist eine Verletzung der Kohäsion: Räume modellieren Räume und Gegenstände. Ein Gegenstand ist fest mit einem Raum verbunden.
- **Besser:** Eine eigene Klasse für Gegenstände einführen.

Kohäsion bei Klassen

- **Ü-9.9:** Erweitern Sie bitte Ihr Projekt Zuul, so dass jeder Raum einen Gegenstand enthält! Gegenstände haben eine Beschreibung und ein Gewicht. Beim Setzen der Ausgänge sollen den Räumen auch Gegenstände übergeben werden.

- **Ü-9.10:** Wie sollten die lesbarer Zeichenketten über die Gegenstände im Raum erzeugt werden? Welche Klasse erzeugt die Zeichenkette für den Gegenstand? Welche Klasse sollte diese ausgeben?

- Der Vorteil der Trennung von Raum und Gegenstand wird klarer, wenn ein Raum mehr als einen Gegenstand enthalten kann.

- **Ü-9.11:** Ändern Sie bitte das Projekt Zuul so, dass ein Raum mehrere Gegenstände enthalten kann! Benutzen Sie dafür eine Sammlung! Schreiben Sie bitte eine Methode *gegenstand_ablegen*, mit dem eine Gegenstand im Raum abgelegt werden kann!

Kohäsion und Wiederverwendbarkeit

- Ein großer Vorteil von Kohäsion ist das Potential für Wiederverwendbarkeit.
- Wenn wir eine eigene Klasse für Gegenstände definieren können wir Gegenstände überall wiederverwenden.
- Wiederverwendung ist auch ein wichtiger Aspekt bei Methoden.
- Wir wollen eine Methode *verlasse_raum (richtung)* schreiben, die den Raum zurückgibt, den wir betreten, und die Beschreibung dieses Raumes ausgibt.
- Wir verwenden die Methoden *ausgang(richtung)* und *to_s* von *Raum*, um die Methode *verlasse_raum* zu implementieren.
- Die Methode *to_s* wird nicht nur in *verlasse_raum,* sondern auch in den Methoden *wechsle_raum, willkommenstext* und *look* verwendet.
- Wir erreichen also durch **hohe Kohäsion** von Methoden, einen **hohen Grad an Wiederverwendung**.

Kohäsion für Wiederverwendbarkeit

- **Ü-9.12**: Implementieren Sie bitte den Befehl **back**, der den Spieler in den Raum zurückbringt, in dem er zuletzt gewesen ist! Der Befehl **back** hat kein Argument.

- **Ü-9.13:** Testen Sie bitte den neuen Befehl mit Runit! Vergessen Sie nicht die negativen Tests. Wie verhält sich Ihr Programm, wenn ihr Spieler nach **back** noch ein weiteres Argument angibt? Gibt es weitere negative Tests?

- **Ü-9.14**: Implementieren Sie bitte den Befehl **back** so, dass eine wiederholte Anwendung den Spieler mehrere Räume zurückversetzt! Benutzen Sie dazu bitte die Klasse Stack aus **Ü-7-c-4.**

Refactoring

- **Refactoring** ist das Restrukturieren eines bestehenden Entwurfs, um die Qualität eines Klassenentwurfs mit Blick auf zukünftige Änderungen und Erweiterungen zu erhalten.

- Refactoring wird dann vorgenommen, wenn bei Erweiterungen z.B. festgestellt wird, dass die Kohäsion der Methoden leidet, oder Implementierungsentscheidungen im Bezug auf gewählte Datenstrukturen für Erweiterungen nicht ausreichen.

- **Vor** jedem Refactoring sollte ein Satz von Regressionstests definiert werden, um Fehler, die beim Refactoring entstanden sind, erkennen zu können.

- **Beim** Refactoring werden strukturierte Änderungen am Quelltext vorgenommen, die die **Funktionalität erhalten** und die **Qualität verbessern**.

- **Nach** dem Refactoring können dann Erweiterungen in den Klassenentwurf aufgenommen werden.

Refactoring

- **Ü-9.15**: Welche grundlegenden Tests für die jetzige Version von Zuul sind notwendig?

- **Ü-9.16:** Können Tests in einem Programm mit Benutzereingaben automatisiert werden? Ist es möglich eine Form der Protokollierung zu nutzen? Können die Benutzereingaben beispielsweise von einer Datei gelesen werden? Welche Klassen müssen angepasst werden, um dieses zu ermöglichen?

Beispiel für Refactoring

Einführen von Spielern

- Wir **erweitern** Zuul so, dass **Spieler Gegenstände aufheben können und mit sich herumtragen können**. Das Szenario lässt sich wie folgt beschreiben:
  - Ein Spieler kann einen Gegenstand in einem Raum aufheben.
  - Ein Spieler kann beliebig viele Gegenstände bis zu einem Maximalgewicht mit sich tragen.
  - Einige Gegenstände können nicht aufgehoben werden.
  - Ein Spieler kann Gegenstände im aktuellen Raum ablegen.

Beispiel für Refactoring

Einführen von Spielern

- Erweiterungen:
  - Klasse ***Gegenstand*** um das Attribut ***@name.*** Der Name des Gegenstands kann benutzt werden, um einen Gegenstand über einen Befehl aufzuheben.
  - Sicherstellen, dass ein Gegenstand nicht aufgehoben werden kann, indem wir ihn schwerer machen.
  - Befehle **take** und **drop**, zum Aufnehmen und Ablegen von Gegenständen in Räumen, jeweils mit einem Argument, dem Namen eines Gegenstandes.
  - Eine Instanzvariable für die Gegenstände, das aktuelle Gepäck eines Spielers.

Beispiel für Refactoring

Einführen von Spielern

- **Frage**: An welcher Stelle im aktuellen Klassenentwurf merken wir uns das aktuelle Gepäck und die Tragkraft eines Spielers?

- **Naive Antwort**: Im Spiel, hier erzeugen wir die Räume und wissen wo ein Spieler startet.

- Diese Entscheidung führt zu einem **schlechten Entwurf**.

- **Probleme**:
  - Die Klasse Spiel ist bereits jetzt sehr umfangreich.
  - Wir würden einem Spiel nur **einen** Spieler zuordnen können.

- **Bessere Antwort**: das aktuelle Gepäck und die Tragkraft sind Eigenschaften eines *Spielers*. Wir benötigen eine neue Klasse ***Spieler***, die diese Eigenschaften aufnimmt.

Beispiel für Refactoring

Einführen von Spielern – Ändern bestehender Implementierung

- Die Instanzvariable *@aktueller_raum* in *Spiel* enthält Informationen über den Aufenthaltsort eines Spielers.
- Diese Eigenschaft gehört in die Klasse *Spieler*.
- Wenn der Spieler den aktuellen Raum kennt, dann kann auch er die Methoden *gegenstand_aufheben(), gegenstand_ ablegen()* anbieten.
- Das Verschieben der Instanzvariable *@aktueller_raum* in die Klasse *Spieler* ist eine Form des Refactorings.

Beispiel für Refactoring

Auswirkungen ohne Refactoring

- Nehmen wir an, Sie stehen in einem Projekt unter Zeitdruck und belassen *@aktueller_raum* als Instanzvariable in der Klasse *Spiel*.

- Eine Woche nach Auslieferung von Zuul kommt Ihre Chefin mit der Anforderung eines Multiplayer-Games, d.h., ihr Spiel soll von mehreren Spielern gleichzeitig gespielt werden können. Die Spieler sollen unterschiedliche Charaktere und Eigenschaften haben.

- Jetzt rächt sich die „Schlamperei“, das Datenfeld *@aktueller_raum* nicht in die Klasse *Spieler* verschoben zu haben.

- In ihrem Spiel können sich alle Spieler immer nur in dem selben Raum aufhalten.

Übungen

- **Ü-9.17:** Restrukturieren Sie bitte Ihr Projekt für eine neue Klasse ***Spieler***! Ein Spieler-Objekt sollte zu Beginn den aktuellen Raum halten, und einen Namen haben.

- **Ü-9.18:** Implementieren Sie bitte die Erweiterung, bei der der Spieler beliebig viele Gegenstände bis zu seiner maximalen Tragkraft aufnehmen kann! Implementieren Sie die Befehle ***take*** und ***drop*** unter Verwendung der Methoden eines Spielers.

- **Ü-9.19:** Legen Sie in einem Raum den Gegenstand **zaubertrank** ab. Erweitern Sie bitte das Spiel um den Befehl **drink**, der die maximale Tragkraft des Spielers erhöht!

Übungen

- Ü-9.20: Erweitern Sie bitte Zuul, so dass mehrere Spieler gleichzeitig nach dem Zaubertrank suchen dürfen! Wir gehen davon aus, dass alle Spieler kooperativ sind und nacheinander einen Spielzug machen. Ein Spielzug ist das Wechseln eines Raumes oder das Aufnehmen oder Ablegen von Gegenständen. Verwalten Sie bitte die Spieler in der Klasse Spiel als Array! Führen Sie einen neuen Befehl **spieler name** ein, mit dem ein Spieler anhand seines Namens ausgewählt wird! Testen Sie Ihre Erweiterung mit 2 Spielern!

- Ü-9.21: Gehen Sie davon aus, dass Spieler nicht kooperativ sind und ihr Spiel die Reihenfolge der Spieler vorgeben muss! Informieren Sie sich über die Datenstruktur **Queue** und skizzieren Sie, wie Sie diese Datenstruktur nutzen wollen, um die Reihenfolge der Spieler zu steuern!

Zusammenfassung

- **Koppelung** beschreibt den Grad der Abhängigkeit zwischen Klassen. Erstrebenswert ist eine lose Koppelung von Klassen, die über eine schmale Schnittstelle kommunizieren.

- **Kohäsion** beschreibt wie gut und vollständig eine Programmiereinheit eine Aufgabe abbildet. Ein guter Klassenentwurf weist einen hohen Grad von Kohäsion auf.

- **Code-Duplizierung** ist ein eindeutiges Zeichen für einen schlechten Entwurf und muss vermieden werden.

- **Kapselung** reduziert die Koppelung und führt zu besseren Entwürfen. Eine Faustregel besagt, dass der direkte Zugriff auf Instanzvariablen möglichst unterbunden werden sollte.

Zusammenfassung

- Ein technisches Hilfsmittel, um Kapselung zu erreichen, ist der Einsatz von **privaten Methoden,** also Methoden mit eingeschränkter Sichtbarkeit.

- **Entwurf nach Zuständigkeiten** ist ein Prozess, bei der jeder Klasse eine klare Verantwortlichkeit zugewiesen wird.

- **Änderungen** am Klassen sollten möglichst **lokal** gehalten werden, so dass diese keine Auswirkungen auf externe Klassen haben.

- **Refactoring** ist eine Aktivität, bei der ein bestehender Entwurf restrukturiert wird, um die Qualität des Entwurfs für Erweiterungen und Änderungen zu erhalten.