PM1/PT Ruby: Datenkapselung und Objektkopien

Datenkapselung

- Datenkapselung ist das Verbergen der internen Struktur vor der Außenwelt und das Anbieten einer schmalen Schnittstelle für die Manipulation eines Objektes, um ein maximales Maß an Entkopplung zwischen Klassen zu erreichen.

- Wichtigste Daumenregel: veränderliche Objekte sollten möglichst nicht über Reader an die Außenwelt gegeben werden. Änderungen sollten kontrolliert über Schnittstellenmethoden erfolgen und nicht über unkontrollierte Writer.

- Vorsicht bei Readern: Geben Reader veränderliche Objekte zurück, dann kann die Außenwelt unkontrollierbare Änderungen vornehmen.

Datenkapselung

- **Ü-14-b-1 und Antibeispiel 1:** Die Klasse ***Stack*** hat einen Reader für den Inhalt. Dann lässt sich sehr leicht das für den ***Stack*** definierte Verhalten zerstören. Wie?

```
class Stack
  attr_reader :inhalt
  def initialize()
    # leere stack
    @inhalt = []
  end
  def empty?() ... End
  def push(elem) ... End
  def pop() ...  End
  def peek()...  end
end
```

Datenkapselung

- Im ersten Fall zerstören wir extern die Reihenfolge des Stacks, indem wir das unterste Element auf den Stack legen ➔ ***pop*** holt dann das erste und nicht das letzte Element aus dem Stack.

- Im zweiten Fall leeren wir den Inhalt des Stacks extern. ***pop*** erzeugt dann einen Fehler ***"Empty Stack",*** der dem zugesicherten Verhalten eines Stacks wiederspricht.

```
s1 = Stack.new()
s1.push(3)
s1.push(5)
s1.push(7)
inhalt = s1.inhalt()
# unterstes Element auf den Stack legen
inhalt << inhalt.shift()
puts s1.pop() # => 3

s1.push(3)
s1.push(5)
s1.push(7)
inhalt = s1.inhalt()
# s1 extern leeren
inhalt.replace([])
puts s1.pop() #=> RuntimeError
```

Datenkapselung im Zuul

- **Zaubertrank stehlen**: Gibt die Klasse Spieler ihr Gepäck über einen **Reader** für die Außenwelt frei, dann können andere Spieler den Zaubertrank oder andere für sie nützliche Gegenstände aus dem Gepäck stehlen.

- **Unfreiwillige Teleportation**: Gibt ein Spieler den aktuellen Raum durch einen **Writer** für die Außenwelt frei, dann können andere Spieler diesen Spieler in beliebige Räume versetzen. Der Pfad des Spielers durch die Räume wird nicht aktualisiert.

- **Magie der verschwindenden Gegenstände**: Gibt ein Raum die in ihm enthaltenen Gegenstände über einen **Reader** für die Außenwelt frei, dann können böse Spieler Gegenstände eines Raumes löschen, insbesondere auch den nützlichen Zaubertrank, oder gefährliche Gegenstände für den Gegenspieler in dem Raum positionieren.

➔ Alle genannten Fälle **dürfen nicht auftreten** und deuten auf ein nicht gut durchdachtes Design hin.

➔ Reader / Writer für die genannten Fälle sind nicht erlaubt.

Aufbrechen der Datenkapselung

- In einigen Fällen benötigen wird jedoch zumindest einen Reader auf veränderliche Objekte.
- Wenn ein Briefträger Briefe austrägt, dann benötigt er Zugriff auf Straße und Hausnummer einer Adresse.
- Die Reader auf diese Eigenschaften einer Adresse geben Referenzen auf veränderliche Objekte (Strings) zurück.

- **Lösung**: Wir geben nur unveränderliche Objekte (**Immutables**) über die Schnitt-stelle eines Objektes nach außen.
  - Wir frieren die referenzierten Objekte ein oder
  - geben statt der Referenzen, Kopien auf die Objekte zurück.

Mutable und Immutable Objekte

Mutable Objekte

- Objekte, deren Zustand sich ändern kann.
- verfügen typischerweise über modifizierende Methoden.
- verfügen über Reader, die Referenzen auf ***Mutable*** Objekte zurückliefern.

Immutable Objekte

- Objekte, deren Zustand sich nicht ändert, weder direkt durch Writer oder modifizierende Methoden noch indirekt durch Modifikationen auf referenzierten Objekten.
- ***Immutables*** sind gute Kandidaten für Schlüssel eines ***Hash***.

Immutables

Immutables

- Objekte vom Typ ***Numeric*** sind immutable.
- ***Symbol*** ist immutable.
- Objekte, deren Instanzvariablen nur Immutables referenzieren,
  - werden durch ***freeze*** zu Immutables (Bsp. ***String***).
  - sind immutable, wenn sie weder öffentliche Reader noch Writer anbieten. (Bsp. ***Telefonnummer***)

Immutables

- ***Range*** ist immutable, wenn der Wertebereich des Intervalls nur immutable Objekte enthält. Beispiele: ***1..6.***
- Ein ***Array*** wird immutable, wenn es eingefroren wird und nur immutable Objekte enthält. Beispiel: ***[1,2,3,5].freeze***

Schutz vor Veränderung durch Einfrieren - Beispiel: Wochentage

- Wir wollen die Tage einer Woche als Objekte einer Klasse darstellen.
- Wir müssen sicherstellen, dass es zu jedem Wochentag nur genau ein Objekt gibt.
- Wir müssen sicherstellen, dass es nur genau 7 Wochentage gibt.

- Wir stellen die Tage einer Woche als Konstanten der Klasse Wochentag dar.
- Um sicher zu stellen, dass nur genau 7 Wochentage und nur genau eine Instanz pro Wochentag erzeugt werden, verhindern wir das externe Erzeugen neuer Wochentage: ***new*** wird zur privaten Klassenmethode. (***private_class_method***)
- Um die Konstanten vor Veränderungen ihrer Instanzvariablen zu schützen, dürfen wir keine Writer einführen.
- Um die Konstanten vor unzulässigen Änderungen zu schützen, frieren wir die Konstanten ein. (***freeze***) **Ist das die Lösung?!**

Immutables und *freeze*

- Im Wochentagbeispiel haben wir das Problem, dass den Konstanten ein neuer Wert zugewiesen werden kann.

  Wochentag::Mo = nil

- In Ruby gibt es die Möglichkeit, Modifikationen von Objekten zu unterbinden, indem man Objekte einfriert (***freeze***).

- ***Mo.freeze*** führt aber nicht zu dem gewünschten Ergebnis, wir dürfen der Konstante dennoch einen neuen Wert zuweisen.

```
class Wochentag
  private_class_method :new
  attr_reader :pos
  attr_writer :pos

  def initialize(tag, pos)
    @tag = tag
    @pos = pos
  end

  Mo = new(:Mo,1)
  Mo.freeze()
end
Wochentag::Mo = nil
```

```
wt/V5-6 wochentag.rb:26: warning: already
initialized constant Mo
```

Einfrieren von Objekten (*freeze*)

- Was bewirkt dann ***freeze***?
- Wir machen die Instanzvariable ***@pos*** schreibbar, frieren ***Mo*** ein, und versuchen dann ***@pos*** einen neuen Wert zuzuweisen.
- Jetzt erhalten wir einen ***TypeError***, der die Änderung eines eingefrorenen Objektes unterbindet.
- ***freeze*** unterbindet also das Zuweisen neuer Werte an die Instanzvariablen eines Objektes.

```
class Wochentag
  private_class_method :new
  attr_reader :pos,
  attr_writer :pos
  def initialize(tag, pos)
    @tag = tag
    @pos = pos
  end

  Mo = new(:Mo,1)
  Mo.freeze()
end
Wochentag::Mo.pos = 3
```

```
wt/V5-6 wochentag.rb:20:in `pos=': can't modify frozen object
(TypeError)
from wt/V5-6 wochentag.rb:20
```

Einfrieren von Klassenkonstanten (*freeze*)

- Die Lösung des Problems ist das Einfrieren der Klasse ***Wochentag***.

```
class Wochentag
  private_class_method :new
  attr_reader :pos
  def initialize(tag, pos)
    @tag = tag
    @pos = pos
  end

  Mo = new(:Mo,1)
  Mo.freeze()
end

Wochentag.freeze()
Wochentag::Mo = nil
```

```
Wochentag.rb:16:in `<main>': can't modify frozen class
(RuntimeError)
```

Einfrieren von *Array/String*

- Alle Arten von Objekten lassen sich einfrieren.

- ***Array (String)***:
  - Bei eingefrorenen Arrays und Strings kann die Größe nicht verändert werden.
  - Bei eingefrorenen Arrays und Strings ist Elementzuweisung nicht erlaubt.

```
a = [1,2,3]
a.freeze()
a << 4
```

```
freeze/array_freeze.rb:6:in `<<': can't
modify frozen array (TypeError)
from freeze/array_freeze.rb:6
```

```
a = [1,2,3]
a.freeze()
a[1] = 4
```

```
freeze/array_freeze.rb:9:in `[]=': can't
modify frozen array (TypeError)
from freeze/array_freeze.rb:9
```

Einfrieren von *Hash*

- Alle Arten von Objekten lassen sich einfrieren.

- ***Hash***:
  - die Größe darf nicht verändert werden.
  - den Schlüsseln darf kein neuer Wert zugewiesen werden.

```
b = {1=>'a', 2=>'b', 3=>'c'}
b.freeze()
b.delete(2)
```

```
freeze/array_freeze.rb:13:in `delete':
    can't modify frozen hash (TypeError)
from freeze/array_freeze.rb:13
```

```
b = {1=>'a', 2=>'b', 3=>'c'}
b.freeze()
b[2] = 'd'
```

```
freeze/array_freeze.rb:14:in `[]=': can't
    modify frozen hash (TypeError)
from freeze/array_freeze.rb:14
```

Einfrieren nur auf oberster Ebene

- Es werden nur die Instanzvariablen der obersten Ebene eingefroren, nicht die Objekte, die von den Instanzvariablen referenziert werden.

- **Im Beispiel**: Es wird nur das Arrayobjekt ***pa*** eingefroren, aber nicht das Personenobjekt im Array. Daher darf dem Namen des Objektes ***Person*** (***pa[0].name***) ein neuer Wert zugewiesen werden.

```
class Person
  attr_reader :name
  attr_writer :name
  def initialize(name)
   @name= name
 end
end

pa = [Person.new('Finn')]
pa.freeze
pa[0].name='Linus'
p pa
pa[0].name[0,1]='A'
p pa
```

```
[#<Person:0xa62200 @name="Linus">]
[#<Person:0xa62200 @name="Ainus">]
```

Einfrieren nur auf oberster Ebene

- Möchte man auch die Änderung der ***Person*** unterbinden, muss diese ebenfalls eingefroren werden.

- **Beachte**: Das Einfrieren von ***Person*** bewirkt natürlich nicht, dass auch der String, den die Instanzvariable ***name*** referenziert, eingefroren wird.

```
class Person
  attr_reader :name
  attr_writer :name
  def initialize(name)
   @name= name
 end
end

pa = [Person.new('Finn').freeze()]
pa.freeze
pa[0].name[0,1]='A'
p pa
pa[0].name='Linus'
p pa
```

```
[#<Person:0xa62200 @name="Ainn">]
```

Prüfung auf den Zustand „eingefroren“ (*frozen?*)

- ***frozen?*** prüft, ob ein Objekt eingefroren wurde.
- Eingefrorene Objekte können nicht wieder aufgetaut werden.

```
b = {1=>'a', 2=>'b', 3=>'c'}
b.freeze()
puts b.frozen?()
```

```
true
```

Mutables, Immutables und Kopien

Mutable Objekte

- Objekte, deren Zustand sich ändern kann.
- Das Kopieren von Mutables muss ***tiefe Kopien*** erzeugen. Nur so kann sicher gestellt werden, dass das Original an keiner Stelle verändert werden kann.

Immutable Objekte

- Objekte, deren Zustand sich nicht ändert, weder direkt durch Writer oder modifizierende Methoden noch indirekt durch Modifikationen auf referenzierten Objekten.
- ***Immutables*** müssen nicht kopiert werden, da keine Gefahr besteht, dass das Objekt selber oder eines der referenzierten Objekte geändert wird. Es können immer Referenzen zurückgegeben werden.

Flache und tiefe Kopien

Eine flache Kopie eines Objektes

- erzeugt eine neue Instanz vom selben Typ wie der des Objektes.
- kopiert die Instanzvariablen aber nicht die referenzierten Objekte.
- ***Object.clone()*** und ***Object.dup()*** erzeugen flache Kopien.

Eine tiefe Kopie eines Objektes

- erzeugt eine neue Instanz vom selben Typ wie der des Objektes.
- erzeugt von allen Objekten, die die Instanzvariablen referenzieren, eine ***tiefe Kopie***.
- für ***tiefe Kopien*** muss
  - entweder die ***clone*** oder ***dup*** Methode überschrieben werden.
  - oder das Kopierverhalten des Moduls ***Marshal*** genutzt werden.

Flache und tiefe Kopien von Objekten am Beispiel

- *Wurde ein Objekt* **eingefroren**, *kann es auch selbst keine Änderungen mehr an sich vornehmen.*

- Häufig benötigen wir nur den Schutz vor Veränderung des Objektzustandes von außen und wollen selber intern unseren Zustand kontrolliert ändern können.

➔ In diesem Fall unterbinden wir schreibende Zugriffe auf die Instanzvariablen für die Außenwelt **und**

➔ geben bei lesenden Zugriffen auf Instanzvariablen, deren Objekte **mutables** sind, **Kopien** dieser Objekte zurück.

- *Das* **Einfrieren** *von Objekten schützt das Objekt vor Modifikationen auf oberster Ebene.*

- Manchmal möchten wir das Objekt und alle im Objekte referenzierten Objekte vollständig schützen.

➔ in diesem Fall erzeugen wir **tiefe Kopien** von Objekten durch rekursives Kopieren der Instanzvariablen.

Flache und tiefe Kopien von Objekten am Beispiel

- Sie sind die IT-Verantwortliche für das Kundensystem einer noch solventen Bank.
- Ihre Richtlinien schreiben vor, dass Adressänderungen nur nach vorheriger Adressprüfung erfolgen dürfen.
- Sie sind der Revision gegenüber für die Einhaltung der Richtlinien nachweispflichtig.
- Ihre Abteilung ist Dienstleister für alle Kontensysteme und muss Auskunft über Kundendaten geben können.
- Wenn Sie ***mutable*** Kundendaten nach außen geben, müssen Sie diese vor unerlaubten Änderungen schützen.
- Für Namensänderungen existieren vergleichbar strenge Richtlinien und komplexe Prozesse.

Kopien von Objekten: Am Beispiel

```
class Kunde
  def initialize(name,adresse)
     @name = name
     @adresse = adresse
  end
  def name
    return @name.clone
  end
  def adresse
    return @adresse.clone
  end
  def adresse_aendern
    # komplexer geschaeftsprozess
  end
  def name_aendern
    # komplexer geschaeftsprozess
  end
  def to_s
    return "#@name, #@adresse"
  end
end
```

Da externe Nutzer Informationen über ***@name*** *und* ***@adresse*** *benötigen, brauchen wir die Reader* ***name***, ***adresse***. *Um die in* ***@name*** *und* ***@adresse*** *referenzierten Objekte zu schützen, kopieren wir diese zuvor mit* ***clone***.

Da wir ***@name*** *und* ***@adresse*** *intern ändern müssen, können wir die Objekte* ***nicht einfrieren***.

Object.clone erzeugt flache Kopien

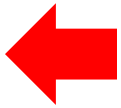
```ruby
# Flache Kopie bei clone
# Object.clone erzeugt eine
# flache Kopie
class Adresse
  attr_reader :str, :ort
  attr_writer :str,:ort
  def initialize(str,ort)
    @str = str
    @ort = ort
  end
  def to_s
    "#@str, #@ort"
  end
end
```

- ***Adresse*** *bietet für* ***@str*** *und* ***@ort*** *sowohl Reader als auch Writer an. Da wir in* ***Kunde k1*** *eine Kopie von* ***@adresse*** *erzeugen, können sich die Änderungen über Writer in* ***Adresse*** *nicht in* ***k1*** *auswirken.*

- *„Gefährlich" sind hier die Reader, da sie Referenzen auf* ***mutable*** *Objekte nach außen geben.*

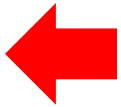
Object.clone erzeugt flache Kopien

```
k1 = Kunde.new('Liz C',
  Adresse.new('Bruecke 1',
          '23457 Niemandsland'))

k1.name[4] = 'Taylor'
ad = k1.adresse
ad.str = 'Noway'
puts k1

Liz C, Bruecke 1,
23457 Niemandsland
```



*Änderungen an den modifizierbaren Objekten, die von **@name** und **@adresse** referenziert werden, sind im Kunden **k1** nicht sichtbar, da sie auf den Kopien **name@clone** und **@adresse.clone** durchgeführt werden.*

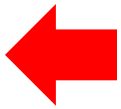
```
ad = k1.adresse
ad.str[0..1] = 'L'
ad.ort[0..ad.ort.size] = ''
puts k1

Liz C, Luecke 1,
```



*Änderungen auf **@str** und **@ort** der Adresse sind im Kunden **k1** sichtbar, da sie auf dem Originalen **ad @str** und **ad @ort** durchgeführt werden.*

Tiefe Kopien von *Adresse* durch Überschreiben von *clone*

```
class Adresse
  attr_reader :str, :ort
  attr_writer :str,:ort
  protected :str=,:ort=
  def initialize(str,ort)
    @str = str
    @ort = ort
  end
  def to_s
    "#@str, #@ort"
  end
  def clone
    cloned = super
    cloned.str = @str.clone
    cloned.ort = @ort.clone
    return cloned
  end
end
```



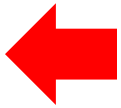
*Damit sich Änderungen auf den von **@str** und **@ort** referenzierten Objekten in **Adresse** nicht auf **Kunde** auswirken, benötigen wir tiefe Kopien dieser Objekte.*
*Dazu überschreiben wir die Methode **clone** aus **Object** in **Adresse.***

***clone** erzeugt eine tiefe Kopie, da sowohl **@str** als auch **@ort** rekursiv kopiert werden.*

Tiefe Kopien durch Überschreiben von *clone* in allen Klassen

```
k1 = Kunde.new('Liz C',
  Adresse.new('Bruecke 1',
          '23457 Niemandsland'))

ad = k1.adresse
ad.str[0..1] = 'L'
ad.ort[0..ad.ort.size] = ''
puts k1
```

```
Liz C, Bruecke 1,
23457 Niemandsland
```



*Änderungen auf **@str** und **@ort** der **Adresse** sind **jetzt** im Kunden **k1** **nicht sichtbar**, da sie auf den Kopien **@str.clone** und **@ort.clone** durchgeführt werden.*

Flache und tiefe Kopien mit *clone* und *dup*

- Methoden ***clone*** (und ***dup***) sind in ***Object*** für alle Klassen implementiert.
- ***clone*** und ***dup*** erzeugen ***flache Kopien*** von Objekten, d.h.:
- werden ***tiefe Kopien*** benötigt,
- müssen Klassen die Methode ***clone*** und ***dup*** überschreiben, und rekursiv alle referenzierten Objekte tief kopieren.

- ***Im Beispiel erzeugt Adresse eine vollständige tiefe Kopie***

Tiefe Kopien mit *Marshal load* und *dump*

- Eine weitere Möglichkeit tiefe Kopien zu erzeugen, sind die Methoden ***load*** und ***dump*** des Moduls ***Marshal***.
- ***dump*** schreibt Objekte als Byte-Strom und ***load*** liest Objekte von einem Byte-Strom.
- ***load*** erzeugt ein vollständig neues Objekt rekursiv für alle referenzierten Objekte. Dadurch entsteht eine tiefe Kopie des Originals.
- Es gibt Objekte, die mit dem ***Marshal dump*** nicht geschrieben werden können. Mehr dazu später.
- In unserem Beispiel lässt sich in der Klasse ***Kunde*** auf diese Weise im Reader ***adresse*** eine vollständige Kopie der ***Adresse*** erzwingen.
- In ***Adresse*** muss keine tiefe Kopie erzeugt werden.

Tiefe Kopien mit *Marshal load* und *dump*

```
class Kunde
  def initialize(name,adresse)
     @name = name
     @adresse = adresse
  end
  def name
    return @name.clone
  end
  def adresse
    puts "tiefe Kopie mit Marshal
    dump/load"
    return
    Marshal.load(Marshal.dump(@adresse))
  end
  def to_s
    return "#@name, #@adresse"
  end
end
```

hier wird die tiefe Kopie von ***@adresse*** erzeugt

Tiefe Kopien mit *Marshal load* und *dump*

```
class Adresse
  attr_reader :str, :ort
  def initialize(str,ort) ...
  end
  def to_s  ...
  end
end

k1 = Kunde.new('Liz C',
  Adresse.new('Bruecke 1','23457
   Niemandsland'))
ad = k1.adresse
ad.str[0..6] = 'X'
ad.ort[0..ad.ort.size] = ''
puts k1
```

in ***Adresse*** benötigen wir kein ***clone***

Liz C, Bruecke 1, 23457 Niemandsland

Kunde ***k1*** bleibt auch hier unverändert.

Mutables, Immutables und Kopieren

Mutables

- Wenn ***mutable*** Objekte von der Aussenwelt nicht verändert werden dürfen, dann muss immer eine ***tiefe Kopie*** der mutable Objekte erzeugt werden. (Beispiel: ***@adresse*** und ***@name***).

- tiefe Kopien können durch mehrere Techniken erzeugt werden:
  - Überschreiben von ***clone*** und ***dup***
  - Nutzen von ***Marshal.dump*** und ***Marshal.load***

Immutables

- Für immutable Objekte können immer die Referenzen auf diese Objekte verwendet werden.

Unterschied zwischen *clone* und *dup*

- ***clone*** und ***dup*** sind Methoden in ***Object,*** die flache Kopien des Empfängerobjektes erzeugen.
- ***clone*** und ***dup*** erzeugen eine neue Instanz der Klasse des zu kopierenden Objektes und kopieren dann die Instanzvariablen.
- ***clone*** kopiert den vollständigen Objektzustand.
- ***dup*** kopiert den Inhalt eines Objektes.
- Was ist nun der Unterschied zwischen Inhalt und vollständiger Zustand?

- Der ***Inhalt*** eines Objektes sind seine Instanzvariablen.
- Der ***vollständige Zustand*** eines Objektes ist sein ***Inhalt*** und ob das Objekt ***eingefroren*** ist.

Einfrieren (*freeze*) und Kopieren (*clone, dup*)

- Beim Kopieren von Objekten
  - kopiert ***clone*** den vollständigen Objekt-zustand inklusive des ***frozen*** Zustands.
  - kopiert ***dup*** nur den Inhalt des Objektes.
- Durch ***clone*** entstandene Kopien können daher ebenso wenig wie die Kopiervorlage modifiziert werden.
- Durch ***dup*** entstandene Kopien können modifiziert werden.

```
h = {1=>'a', 2=>'b', 3=>'c'}
h.freeze()

hclone = h.clone
hdup = h.dup

puts hclone.frozen?
puts hdup.frozen?
```

```
true
false
```

Übungen

- **Ü-14-b-2**: Schreiben Sie eine Klasse WG (Wohngemeinschaft), die Methoden für das einziehen und ausziehen von Personen hat. Eine WG hat eine Adresse. Die einzelnen Personen erhalten die WG Adresse, wenn sie einziehen. Schreiben Sie für Personen eine Methode ***adresse_aendern***, die Strasse, Hausnummer, Plz und / oder Ort ändern kann. Bei Umzügen innerhalb eines Ortes oder Postleitzahlenbereichs müssen nur Strasse und Hausnummer geändert werden.