



PM1/PT Ruby: Objektinteraktion und Debugger



Konzepte & Techniken

Konzepte

- Abstraktion
- Modularisierung
- Objekterzeugung
- Objektdiagramme
- Methodenaufrufe

Techniken

- Methoden mit Default-parametern
- Das Objekt *self*



Das Uhrenbeispiel

- Das Projekt, das die Interaktion von Objekten demonstriert, modelliert die Anzeige einer Digitaluhr.
- Die Anzeige zeigt Stunden und Minuten, die durch einen Doppelpunkt voneinander getrennt sind.
- Anfangs soll die Digitaluhr einer 24 Stundenanzeige ausgeben.
- Die Anzeige geht von 00:00 bis 23:59.
- Eine amerikanische Digitaluhr mit 12-Stundenanzeige verschieben wir auf eine nachfolgende Übung.



Abstraktion

- **Abstraktion** ist die Kunst, Details zu ignorieren und den Fokus auf eine höhere Ebene der Betrachtung zu lenken.
- Für die Lösung der Digitaluhr könnten wir das Verhalten in einer einzigen Klasse realisieren.
- Wir wollen jedoch untersuchen, ob wir Teilaufgaben identifizieren können, die wir in eigenen Klassen modellieren.
- **Grund:** Mit einer Zerlegung in Teilaufgaben lässt sich die **Komplexität** einer Gesamtaufgabe besser handhaben, da man nicht immer alle Details der Lösung überblicken muss.
- In späteren Projekten werden die Aufgaben komplizierter, so dass dieses Vorgehen notwendig wird, um die Lösung zu überblicken.
- Schauen wir uns das Vorgehen am Beispiel der *Autoproduktion* genauer an.



Beispiel Automobilproduktion

- IngenieurInnen eines Automobilherstellers sollen gemeinsam ein neues Modell entwickeln.
- Sie sind Experten für unterschiedliche Bereiche in der Produktion: äußere Form, Größe und Position des Motors, Innenraumausstattung etc.
- Eine andere Person ist verantwortlich für die Entwicklung des Motors, Zylinder, Einspritztechnik, Vergaser, Elektronik. Diese Person betrachtet den Motor nicht als Ganzes, sondern als komplexes Produkt aus Einzelteilen, z.B. auch der Zündkerze.
- Eine weitere Person ist verantwortlich für die Entwicklung der Zündkerzen (in der Realität bei einem Zulieferer). Sie betrachtet die Zündkerze als komplexes Gebilde aus Einzelteilen und macht sich Gedanken, welches Metall am Besten geeignet ist, bzw. welche Isolierung zum Einsatz kommen soll



Beispiel Automobilproduktion

- Ähnliches gilt für viele Teile im Auto. Ein Designer betrachtet den Reifen als Gesamtgebilde. Ein Ingenieur macht sich Gedanken über die genaue chemische Zusammensetzung für die Gummimischung und den Reifenaufbau.
- Dieser Ingenieur sieht den Reifen als komplexes Gebilde.
- Der Automobilhersteller kauft den Reifen ein und betrachtet den gesamten Reifen als Teil des Automobils.
- Zugrunde liegt hier immer das Prinzip der Abstraktion.
- Der Punkt: Wenn wir die Einzelteile eines Autos in allen Details betrachten, dann ist die Produktion eines Autos so komplex, dass eine einzelne Person den Prozess nicht mehr durchschauen kann.
- Automobile können heute nur gebaut werden, da durch **Modularisierung** und **Abstraktion** der Herstellungsprozess beherrschbar wird.



Beispiel Automobilproduktion

- Der **Punkt**: Wenn wir die Einzelteile eines Autos in allen Details betrachten, dann ist die Produktion eines Autos so komplex, dass eine einzelne Person den Prozess nicht mehr durchschauen kann.
- Automobile können heute nur gebaut werden, da durch **Modularisierung** und **Abstraktion** der Herstellungsprozess beherrschbar wird.
- **Modularisierung** ist der Prozess der Zerlegung eines komplexen Ganzen in beherrschbare Einzelteile, die getrennt voneinander hergestellt werden können und die in definierter Weise miteinander interagieren.
- **Abstraktion** und **Modularisierung** sind Gegenpaare. Während Modularisierung die Zerlegung in Teilaufgaben und deren Lösung ermöglicht, erlaubt Abstraktion den Überblick über das Zusammenspiel der Teilaufgaben zu behalten.



Modularisierung am Uhren-Beispiel

- **Erster Ansatz:** Die Anzeige besteht aus vier Ziffern.
- **Zweiter Ansatz:** Die Anzeige besteht aus je zwei Anzeigen à 2 Ziffern, je 2 für die Stunden- und 2 für die Minutenanzeige.
- Die Stundenanzeige zählt von 00 bis 23 Uhr hoch und setzt dann auf 0 zurück.
- Die Minutenanzeige zählt von 00 bis 59 Minuten hoch und setzt dann auf 0 zurück.
- Da sich beide Anzeigen ähnlich verhalten, können wir von den Unterschieden auf eine 2-ziffrige Nummernanzeige **abstrahieren**.
- Die Nummernanzeige beginnt bei 0 an zu zählen und wird bis zu einem Limit inkrementiert. Wenn das Limit erreicht ist, wird die Anzeige auf 0 zurückgesetzt.
- Der *einzigste Unterschied* zwischen Stunden und Minuten ist der konkrete Wert für das Limit.



Implementierung der Digitaluhr

Gerüst der Nummernanzeige

```
class NummernAnzeige

  def initialize(limit)
    @limit = limit
    @wert = 0
  end
end
```

Gerüst der Uhrenanzeige

- Wir können eine Uhrenanzeige implementieren, wenn wir diese aus zwei Nummernanzeigen für Stunden und Minuten zusammensetzen.

```
class UhrenAnzeige
  def initialize(stunde=0,minute=0)
    @stunden =
      NummernAnzeige.new(24)
    @minuten =
      NummernAnzeige.new(60)
    @zeitanzeige = ""
    ...
  end
end
```



Klassen- und Objektdiagramme

Klassendiagramm

- Ein **Klassendiagramm** zeigt die Klassen einer Anwendung.
- Ein Klassendiagramm liefert Informationen über den Quelltext und zeigt eine statische Sicht auf das Programm.

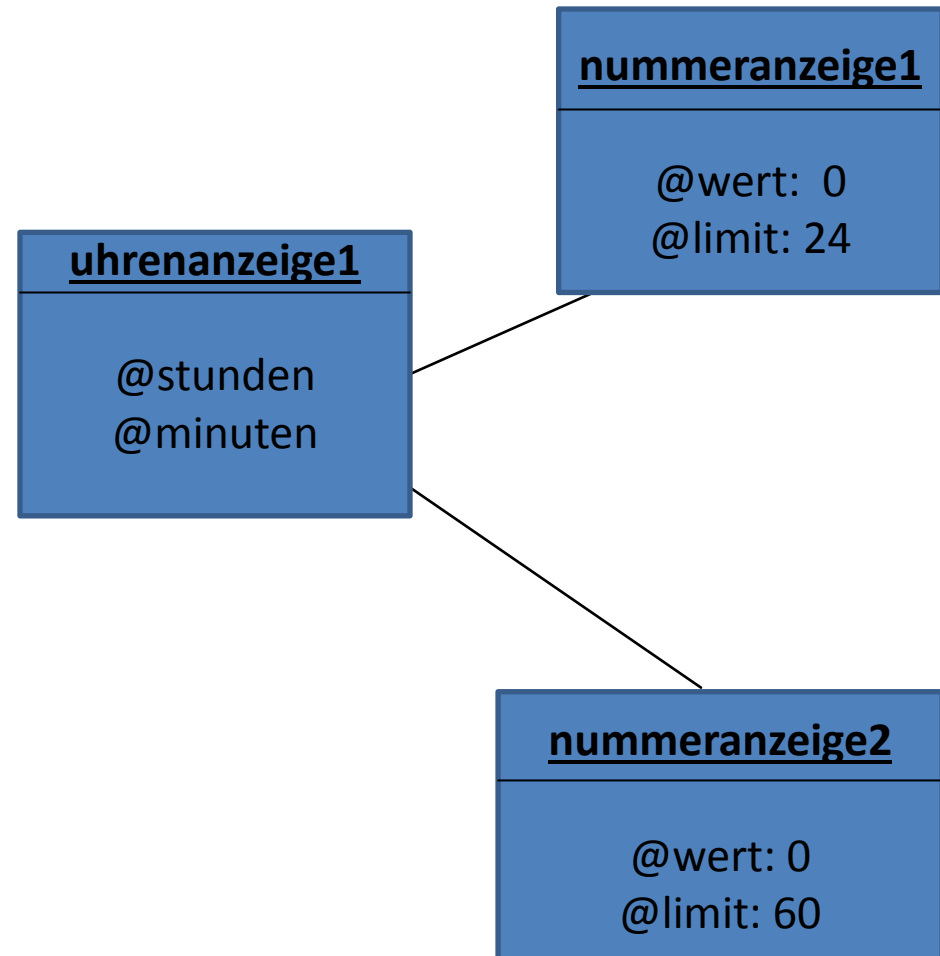
Objektdiagramm

- Ein **Objektdiagramm** zeigt Objekte und ihre *Beziehungen* zu anderen Objekten zur Laufzeit, während eines bestimmten Zeitpunktes der Ausführung eines Programms.
- Ein Objektdiagramm zeigt die dynamische Sicht auf ein Programm.



Objektdiagramm für eine Uhrenanzeige

- Wir erzeugen mit der Toolbox eine Uhrenanzeige.
- Das Objektdiagramm enthält dann drei Objekte.
- Eine Uhrenanzeige und zwei Nummernanzeigen.
- Die Uhrenanzeige enthält die beiden Nummernanzeigen nicht, sondern referenziert diese über die Instanzvariablen **@stunden** und **@minuten**.





Übungen

- **Ü4-1:** Wir erzeugen im Projekt Laborkurs einen Laborkurs und 3 Studenten. Anschließend tragen wir die 3 Studenten in den Laborkurs ein. Zeichnen Sie ein Objektdiagramm für den Laborkurs.
- **Ü4-2:** Zu welchem Zeitpunkt ändert sich ein Klassendiagramm?
- **Ü4-3:** Zu welchem Zeitpunkt ändert sich ein Objektdiagramm? Wie ändert sich ein Objektdiagramm?
- **Ü4-4:** Wir öffnen das Projekt v4-Uhrenanzeige und starten die Toolbox, erzeugen ein Objekt für die UhrenAnzeige und öffnen den Inspektor. Dann rufen wir Methoden auf der Uhrenanzeige auf und überprüfen die Veränderungen im Objektinspektor.



Implementierung

NummernAnzeige

```
class NummernAnzeige

  def initialize(limit) ... end
  def erhoehen() ... end
  def limit() ... end
  def wert() ... end
  def wert=(neuer_wert) ... end
  def to_s() ... end

end
```

UhrenAnzeige

```
class UhrenAnzeige
  def
    initialize(stunde=0,minute=0)
  end
  def
    setze_uhrzeit(stunde,minute)
  end
  def anzeige_aktualisieren()
  end
  def takt_signal_geben()
  end
  def gib_uhrzeit()
  end
  def to_s()
  end
end
```



Implementierung der Klasse *NummernAnzeige*

- **Bestandteile:**

- Initialisierung: Methode *initialize(limit)*
- inkrementieren: Methode *erhoehen()*
- Wert der Anzeige setzen: Methode *wert=(neuer_wert)*
- Limit bzw. Wert ausgeben: Methoden *wert(), limit()*
- Darstellung als lesbare Zeichenkette für die Anzeige: Methode *to_s()*

```
class NummernAnzeige

  def initialize(limit) ... end
  def erhoehen() ... end
  def limit() ... end
  def wert() ... end
  def wert=(neuer_wert) ... end
  def to_s() ... end

end
```



Methode *initialize(limit)*

- Wir erzeugen eine Nummernanzeige immer mit einem Limit, das die obere Grenze für das Inkrementieren vorgibt.
- Da die Nummernanzeige immer bei 0 zu zählen beginnt, setzen wir den Wert auf 0.
- Beide Größen beschreiben den Objektzustand einer Nummernanzeige, der während der Lebensdauer verändert wird. Daher speichern wir diese Größen in Instanzvariablen *@wert* und *@limit*.

```
class NummernAnzeige
```

```
# die Zahlen / Ziffern liegen in  
    einem Intervall von 0 bis  
    ausschließlich Limit  
# der Wert ist anfangs 0
```

```
def initialize(limit)
```

```
  @limit = limit
```

```
  @wert = 0
```

```
end
```

```
end
```



Methode *erhoehen()*

- Die Methode *erhoehen* muss sicherstellen, dass *@wert* bei jedem Aufruf um 1 erhöht wird und der Wert auf 0 zurückgesetzt wird, wenn *@wert == limit*.
- Das erreichen wir mit dem arithmetischen Ausdruck, in dem nach der Erhöhung um 1 das Ergebnis mit dem Modulo Operator *%* auf den ganzzahligen Rest der Division mit dem Limit abgeschnitten wird.

```
class NummernAnzeige
```

```
# Erhoehe den Wert um 1.  
Wenn der Wert das Limit  
erreicht hat,  
# setze Wert auf 0.
```

```
def erhoehen()
```

```
    @wert = (@wert+1)%limit
```

```
end
```

```
end
```




Der Modulo Operator

- Der Modulo-Operator berechnet den Rest einer ganzzahligen Division.
- Das Ergebnis der Division **27 / 4** kann durch zwei Zahlen ausgedrückt werden:
 - Ergebnis = 6, Rest = 3
- Der Modulo Operator liefert den Rest der Division. Das Ergebnis von **(27 % 4)** ist daher **3**.
- **Ü4-5: Was sind die möglichen Ergebnisse von $n\%5$, wenn n eine ganze Zahl > 0 ist.**
- **Ü4-6 Was sind die möglichen Ergebnisse von $n\%m$, wenn n und m positive ganze Zahlen sind.**
- **Ü4-7: Schreiben Sie die Methode `erhoehen()` um und verwenden Sie eine `if` Anweisung.**



Methode *wert=(neuer_wert)*

- Wird der Wert einer NummernAnzeige explizit gesetzt, dann muss sicher gestellt werden, dass der neue Wert in definierten Wertebereich *[0,limit)* liegt. Ansonsten bleibt der Wert unverändert.
- Die Intervallprüfung nehmen wir mit Hilfe eines logischen Ausdruck und zweier Vergleichsoperatoren vor, wie wir sie in der letzten Vorlesung kennengelernt haben.

```
class NummernAnzeige
```

```
  # Setze den Wert auf den neuen Wert, wenn der neue Wert  
  # größer gleich 0 und kleiner Limit ist
```

```
  def wert=(neuer_wert)
```

```
    if( neuer_wert >=0 && neuer_wert < limit)
```

```
      @wert = neuer_wert
```

```
    end
```

```
  end
```

```
end
```



Übungen

- **Ü4-6:** Wie verhält sich die Methode, wenn der Operator `&&` durch den Operator `||` in `wert=(neuer_wert)` ersetzt wird?
- **Ü4-7** Schreiben Sie bitte einen logischen Ausdruck mit zwei Variablen `a` und `b`, der `true` liefert, wenn entweder `a` und `b` beide `true` oder `a` und `b` beide `false` sind!
- **Ü4-8** Schreiben Sie bitte einen logischen Ausdruck mit zwei Variablen `a` und `b`, der `true` liefert, wenn nur genau einer von beiden `a` oder `b` `true` ist und der `false` liefert, wenn `a` und `b` beide `true` oder `a` und `b` beide `false` sind. (Exklusives Oder)!



Methoden *wert* und *limit*

- Die Methoden *wert()* und *limit()* liefern den Inhalt der Instanzvariablen *@wert* und *@limit* zurück.
- **Kennen Sie eine äquivalente Schreibweise in Ruby?**

```
class NummernAnzeige

  def limit()
    return @limit
  end

  def wert()
    return @wert
  end
end
```



Methode *to_s*

- Die Methode *to_s()* bereitet den Zustand der NummernAnzeige als lesbare Darstellung für die Anzeige aus und gibt eine Zeichenkette zurück.
- Dabei werden einziffrige Werte mit einer führenden 0 eingerückt.
- **Kennen Sie eine Alternative für die Erzeugung der Zeichenkette, die mit *return* zurückgegeben wird?**

```
class NummernAnzeige
```

```
  # Darstellung der  
  # Nummernanzeige als String  
  # Ist der Wert kleiner 10, wird  
  # die Zahl mit  
  # einer führenden 0 dargestellt
```

```
  def to_s()  
    s = ""  
    if (@wert < 10)  
      s = "0"  
    end  
    return s + @wert.to_s  
  end  
end
```



Implementierung der Klasse *UhrenAnzeige*

- **Bestandteile:**

- Initialisierung: Methode *initialize(stunde=0,minute=0)*
- Uhrzeit setzen: Methode *setze_uhrzeit(stunde, minute)*
- Wert für die Anzeige aktualisieren: Methode *anzeige_aktualisieren()*
- Uhrzeit für die Darstellung ausgeben: Methoden *gib_uhrzeit()*
- Darstellung als lesbare Zeichenkette für die Anzeige: Methode *to_s()*

```
class UhrenAnzeige
  def
    initialize(stunde=0,minute=0)
  end
  def
    setze_uhrzeit(stunde,minute)
  end
  def anzeige_aktualisieren()
  end
  def takt_signal_geben()
  end
  def gib_uhrzeit()
  end
  def to_s()
  end
end
```



Methode *initialize*

- Bei der Erzeugung einer Uhrenanzeige, werden die Nummernanzeigen für Stunden und Minuten sowie eine Instanzvariable *@zeitanzeige*, die die Darstellung der Uhrenanzeige als String enthält, erzeugt.
- Wird bei der Erzeugung die Methode *initialize* ohne Parameter aufgerufen, dann sind die aktuellen Parameter *stunde* und *minute* gleich 0. Das wird erreicht durch Default-Parameter in der Methodendefinition.
- Wird *initialize* mit Parametern aufgerufen, dann ist der Wert für *stunde* und *minute* gleich dem Wert der aktuellen Parameter beim Aufruf.
- Die Initialisierung endet mit dem Aufruf der internen Methode *setze_uhrzeit*.

```
class UhrenAnzeige
```

```
  def initialize(stunde=0,minute=0)
    @stunden = NummernAnzeige.new(24)
    @minuten = NummernAnzeige.new(60)
    @zeitanzeige = ""
    setze_uhrzeit(stunde,minute)
```

```
  end
```

```
end
```



Defaultparameter für Methoden

- Default-Parameter in Methodendefinitionen belegen beim Aufruf von Methoden die aktuellen Parameter mit Standardwerten, wenn für diese Parameter keine Werte übergeben werden.
- Beim Aufruf der Methode können aber müssen keine Werte für Default-Parameter übergeben werden. Dabei können die Default-Parameter von hinten gelesen ausgelassen werden.
- Sind **n** Default-Parameter und **m** normale Parameter in einer Methodendefinition enthalten, so kann die Methode mit **m, m+1,...,m+n** aktuellen Parametern aufgerufen werden.
- Default-Parameter müssen in der Definition immer als letzte Parameter stehen.

```
uaz = UhrenAnzeige.new()  
puts uaz.gib_uhrzeit()  
uaz = UhrenAnzeige.new(13,56)  
puts uaz.gib_uhrzeit()  
uaz = UhrenAnzeige.new(13)  
puts uaz.gib_uhrzeit()
```



```
00:00  
13:56  
13:00
```




Default-Parameter für Methoden

- In Ruby identifiziert der **Name** einer Methode diese Methode eindeutig. Die Anzahl der Parameter ist unerheblich.
- **Was zu Verwirrungen führen kann:** In Ruby ist es erlaubt verschiedene Definitionen für einen Methodennamen in einer Klasse aufzuschreiben. Nur die letzte Definition ist gültig.
- Wenn wir z.B. wie rechts zwei `initialize` Methoden für die Klasse `UhrenAnzeige` schreiben, dann „gewinnt“ die letzte Definition und der Aufruf von `new` ohne Parameter ist nicht mehr möglich.
- **Führen Sie bitte in einem RubyScript vor, dass die letzte Behauptung gilt.**

```
class UhrenAnzeige

  #
  # Initialisierung ohne Defaults.
  # Das letzte Initialize gewinnt.

  def initialize()
    @stunden = NummernAnzeige.new(24)
    @minuten = NummernAnzeige.new(60)
    @zeitanzeige = ""
    setze_uhrzeit(0,0)
    @number = instanzen_zaeher()
  end

  def initialize(stunde,minute)
    @stunden = NummernAnzeige.new(24)
    @minuten = NummernAnzeige.new(60)
    @zeitanzeige = ""
    setze_uhrzeit(stunde,minute)
    @number = instanzen_zaeher()
  end
end
```



Methode *setze_uhrzeit*

- Die Methode *setze_uhrzeit* passt den Wert für *@stunden* und *@minuten* an und ruft dazu eine **externe** Methode auf den Objekten auf, die durch *@stunden* und *@minuten* referenziert werden.
- Danach wird die Anzeige durch Aufruf der internen Methode *anzeige_aktualisieren* aktualisiert.

```
class UhrenAnzeige
  # setzt die Uhrzeit für die in 'stunde' und 'minute'
  # übergebenen Werte
  def setze_uhrzeit(stunde,minute)
    @stunden.wert=(stunde)
    @minuten.wert=minute
    anzeige_aktualisieren()
  end
end
```



Methode *takt_signal_geben*

- Die Methode *takt_signal_geben* erhöht die Minuten der Uhrenanzeige um eine Minute.
- Wenn das Minutenlimit erreicht ist, dann werden die Stunden um eins erhöht.
- Anschließend wird die Anzeige auf den aktuellen Zustand der Uhrenanzeige angepasst.
- In einer realen Digitaluhr wird das Taktsignal durch einen elektronischen Taktgeber alle 60 Sekunden vorgegeben.
- Hier simulieren wir den Takt durch manuelle Eingabe.

```
class UhrenAnzeige
  def takt_signal_geben()
    @minuten.erhoehe()
    if (@minuten.wert == 0)  # Minutenlimit wurde erreicht
      @stunden.erhoehe()
    end
    anzeige_aktualisieren()
  end
end
```



Methode *anzeige_aktualisieren*

- Die Methode *anzeige_aktualisieren* passt den Inhalt der Instanzvariablen auf den Zustand der Instanzvariablen *@stunden* und *@minuten* an.
- Dazu wird die externe Methode *to_s* auf den Instanzvariablen *@stunden* und *@minuten* aufgerufen.
- **Gibt es eine alternative und kürzere Schreibweis für das Erzeugen der Zeichenkette, die *@zeitanzeige* zugewiesen wird?**

```
class UhrenAnzeige
```

```
  # Aktualisiere die Zeichenkette für die UhrenAnzeige
```

```
  def anzeige_aktualisieren()
```

```
    @zeitanzeige = @stunden.to_s() + ":" + @minuten.to_s()
```

```
  end
```

```
end
```



Methode ***gib_uhrzeit***

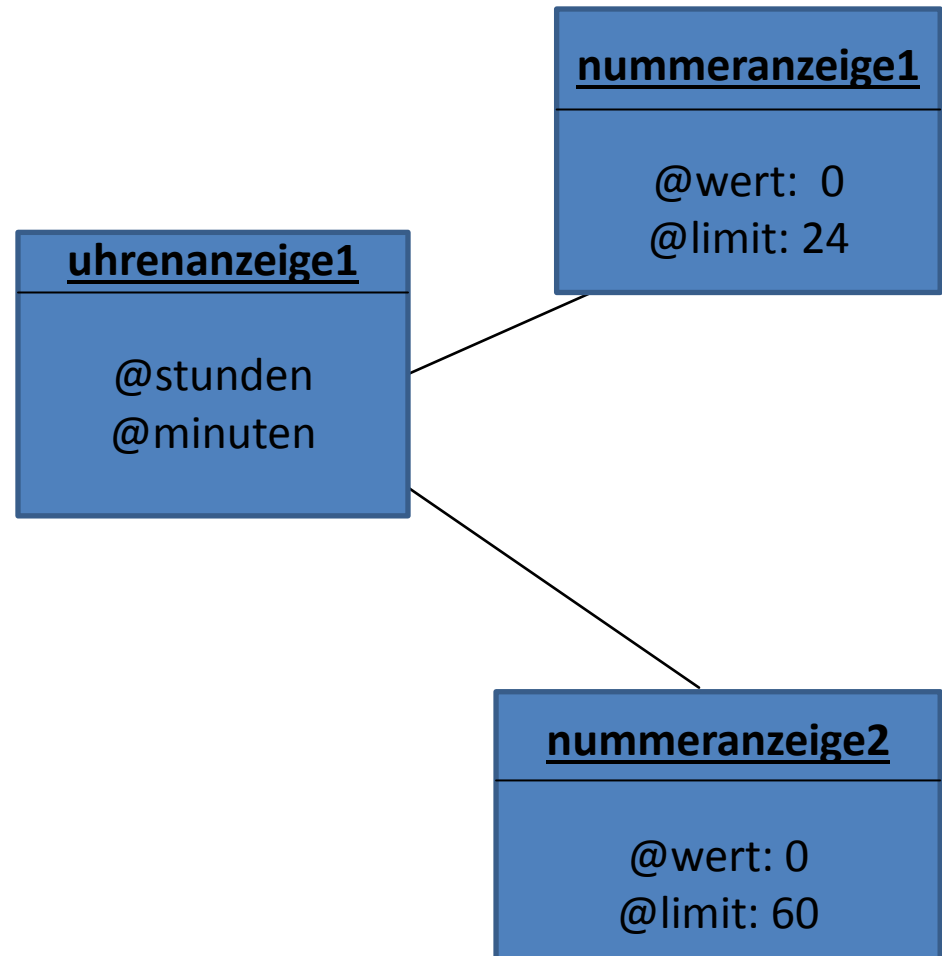
- Die Methode ***gib_uhrzeit*** liefert eine lesbare Darstellung für die Uhrenanzeige als Zeichenkette zurück.
- Sie verwendet dazu die interne Methode ***to_s***.

```
class UhrenAnzeige
  # Gibt die UhrenAnzeige als lesbare Zeichenkette aus
  def gib_uhrzeit()
    return @zeitanzeige
  end
end
```



Objekte erzeugen Objekte

- Das Objektdiagramm rechts zeigt, dass beim Erzeugen einer **UhrenAnzeige** auch zwei **NummernAnzeigen** erzeugt werden.
- Dies wird erreicht, wenn beim Aufruf der Methode **initialize** in **UhrenAnzeige** auf der Klasse **NummernAnzeige** **new** aufgerufen wird.
- Durch Zuweisung der Objekte, die durch **new** entstehen, an die Instanzvariablen **@stunden** und **@minuten**, hält das Objekt **uhrenanzeige1** Referenzen auf diese Objekte.





Objekte erzeugen Objekte

- Werden zur Laufzeit Referenzen auf Objekte benötigt, so müssen Sie an den entsprechenden Stellen in den Methodendefinitionen die Erzeugung dieser Objekte mit *new* vorsehen.
- Dann enthalten **Methodendefinitionen** **Methodenaufrufe** auf Objekten.
- In den Methodenaufrufen werden aktuelle Parameter übergeben, die entweder feste Werte oder Variablen aus dem Gültigkeitsbereich der definierenden Methode sind, die diese Aufrufe enthält.

```
class UhrenAnzeige
  def initialize(stunde=0,minute=0)
    @stunden = NummernAnzeige.new(24)
    @minuten = NummernAnzeige.new(60)
    @zeitanzeige = ""
    setze_uhrzeit(stunde,minute)
  end
end
```

*aktuelle Parameter
feste Werte*

*aktuelle Parameter
Variablen stunde, minute*



Interne Methodenaufrufe

- Die letzte Zeile im *initialize* der Uhrenanzeige ist die Anweisung *setze_uhrzeit(stunde,minute)*
- Diese Anweisung ist ein **Methodenaufruf**, genauer ein **interner Methodenaufruf**.
- Wir sprechen von internen Methodenaufrufen, wenn die Klasse die Definition der aufgerufenen Methode selbst enthält.
- Wenn ein Methodenaufruf erfolgt, dann wird die passende Methode ausgeführt und anschließend kehrt die Ausführung an die Aufrufstelle zurück.
- Der Adressat für einen internen Methodenaufruf ist immer das aktuelle Objekt.

```
class UhrenAnzeige
  def initialize(stunde=0,minute=0)
    @stunden = NummernAnzeige.new(24)
    @minuten = NummernAnzeige.new(60)
    @zeitanzeige = ""
    setze_uhrzeit(stunde,minute)
  end
end
```

*aktuelle Parameter
Variablen stunde, minute
für einen internen Methodenaufruf*



Externe Methodenaufrufe

- Die Methode `takt_signal_geben` in der Klasse `UhrenAnzeige` erhöht die Minuten in einem extern vorgegebenen Takt.
- Sie ruft dazu die Methode `erhoehen` auf dem über die Variable referenzierten Objekt `@minuten` auf.
- Es handelt sich um einen externen Methodenaufruf, da der Aufruf auf einem anderen Objekt erfolgt.
- Externe Methoden werden immer über die Punkt Notation aufgerufen. Vor dem Punkt steht immer das Objekt, auf dem die Methode aufgerufen wird, nach dem Punkt die Methode mit den aktuellen Parametern.

```
class UhrenAnzeige
  def takt_signal_geben()
    @minuten.erhoehen()
    if (@minuten.wert == 0)  # Minutenlimit wurde erreicht
      @stunden.erhoehen()
    end
    anzeige_aktualisieren()
  end
end
```



Zusammenfassung

- Wie haben wir in diesem Beispiel das Konzept der Abstraktion verwendet, indem wir Aufgabe in kleinere Aufgaben zerlegt haben?
- Der Blick in die Methoden der Uhrenanzeige zeigt, dass wir zwei Nummernanzeigen erzeugen, ohne dass wir daran interessiert sind, wie diese Objekte intern arbeiten.
- Wir rufen die Methoden **erhoehen**, **wert=()**, **to_s()** auf diesen Objekten auf ohne dass uns interessiert, wie diese Methoden arbeiten.
- Die Klasse **UhrenAnzeige** abstrahiert von der internen Arbeitsweise der Klasse Nummernanzeige.
- Wenn in einem realen, größeren Softwareprojekt zwei Personen an der Entwicklung von unterschiedlichen Klassen arbeiten, dann müssen diese sich über die Schnittstellen der Klassen abstimmen.
- Die Schnittstellen lassen sich unmittelbar aus den externen Methodenaufrufen bestimmen.



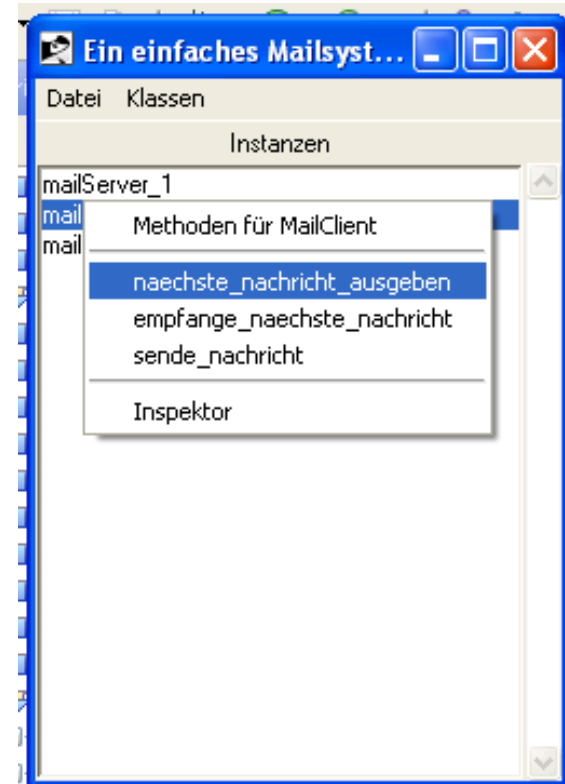
Ein einfaches Mailsystem

- Am Beispiel eines einfachen Mailsystems wollen wir die Kenntnisse über Objektinteraktion vertiefen.
- Dabei soll die Benutzung des Debuggers für die Inspektion der Objekte zur Laufzeit und als Hilfsmittel zur Fehlersuche eingeführt werden.
- Ein Debugger ist ein Softwaretool, mit dem die Ausführung eines Programms zur Laufzeit möglich ist.
- In der Computersprache werden Fehler häufig Bug (engl. Käfer) genannt. Das Wort Debugger ist davon abgeleitet und bedeutet soviel wie „Käfer“ bekämpfen.



Ein einfaches Mailsystem

- Im Projekt **v4-EinfachesMailSystem** wird simuliert, dass sich Benutzer gegenseitig Mails schicken. Benutzer verwenden dazu eine Mailclient, der Nachrichten an einen Server schickt, der wiederum die Mails an einen Mailclient eines anderen Benutzer weiterleitet.
- Wir erzeugen einen **MailServer** und zwei **MailClients**, denen wir bei der Erzeugung den gerade erzeugten **MailServer** mitgeben und experimentieren mit den Methoden der **MailClients**.





Ein einfaches Mailsystem

- Bei der Untersuchung des Projektes fällt auf:
- Es enthält die 3 Klassen *MailClient*, *MailServer* und *Nachricht*.
- Ein *MailServer* Objekt muss erzeugt werden, das von allen Mail-Clients genutzt wird.
- Jeder Mail-Client ist mit einem Benutzernamen verbunden.
- Nachrichten können von einem Client zu einem anderen über eine Methode verschickt werden.
- Nachrichten können von einem Mail-Client über eine Methode empfangen werden, die Nachrichten einzeln vom Server abholt.
- Der Benutzer erzeugt nicht explizit Nachrichten. Nachrichten werden intern in den beiden anderen Klassen verwendet, um Nachrichtentexte zu erzeugen, auszutauschen oder zu speichern.



Ein einfaches Mailsystem

- Die drei Klassen sind unterschiedlich komplex.
- Die Klasse **Nachricht** ist sehr einfach zu verstehen und wird nur kurz skizziert.
- Die Klasse **MailServer** ist sehr komplex und benutzt Konzepte, die wir erst später kennenlernen. Wir verlassen uns auf die Funktionsweise des Servers und besprechen diese im Detail nur auf Nachfrage bzw. als Überleitung zur nächsten Vorlesung
- Das Hauptaugenmerk liegt hier auf der Klasse **MailClient**.



Ein einfaches Mailsystem

- Die Hauptaufgabe der Klasse **Nachricht** besteht darin, die Beziehung zwischen Sender, Empfänger und Nachrichtentext zu speichern. Diese Beziehung wird im **initialize** hergestellt.
- Die anderen Methoden dienen der Ausgabe oder sind einfache sondierende Methoden.

```
class Nachricht
  attr_reader :empfaenger, :sender,
              :text

  def
    initialize(sender, empfaenger, text)
      @sender = sender
      @empfaenger = empfaenger
      @text = text
      @number = instanzen_zaeher()
    end

  def ausgeben()
    puts " Von: #{sender}\n An:
        #{empfaenger}\nText: #{text}"
    end

  def to_s()
    "Nachricht(von: #{sender}, an:
      #{@empfaenger, #{@text})"
    end
end
```



Ein einfaches Mailsystem

- Der *MailClient* enthält Methoden zum Versenden, Empfangen und Ausgeben von empfangenen Nachrichten.
- Die Implementierung und das Verhalten dieser Methoden zur Laufzeit werden wir uns nun im Debugger anschauen.

```
class MailClient
  def initialize(benutzer, server)
    @server = server
    @benutzer = benutzer
    @number = instanzen_zaeher()
  end
  attr_reader :benutzer
  def sende_nachricht(empfaenger, text)
    ...
  end
  def empfangen_naechste_nachricht() ...
  end
  def naechste_nachricht_ausgeben() ...
  end
  def to_s()
    end
end
```




Nutzung des Debuggers

- Wir erzeugen im Projekt **v4-EinfachesMailSystemDebug** ein **Szenario** für den Debugger.
- Das Szenario ist ein Ruby-Script
 - das zwei Mail-Clients mit Benutzer Siggie und Sara enthält.
 - in dem Siggie eine Nachricht an Sara schickt.
 - in dem anschließend Sara die Nachricht vom Server abrufen und ausgeben lässt.
- Wir führen das Script noch nicht aus, sondern bereiten zuerst die Implementierung für die Nutzung des Debuggers vor.

szenario.rb

```
require "MailServer"
require "MailClient"

server = MailServer.new()
siggi =
  MailClient.new("Siggie", server)
sara = MailClient.new("Sara", server)
siggi.sende_nachricht("Sara", "heute
  abend lange Kinonacht?")
sara.empfange_naechste_nachricht()
```



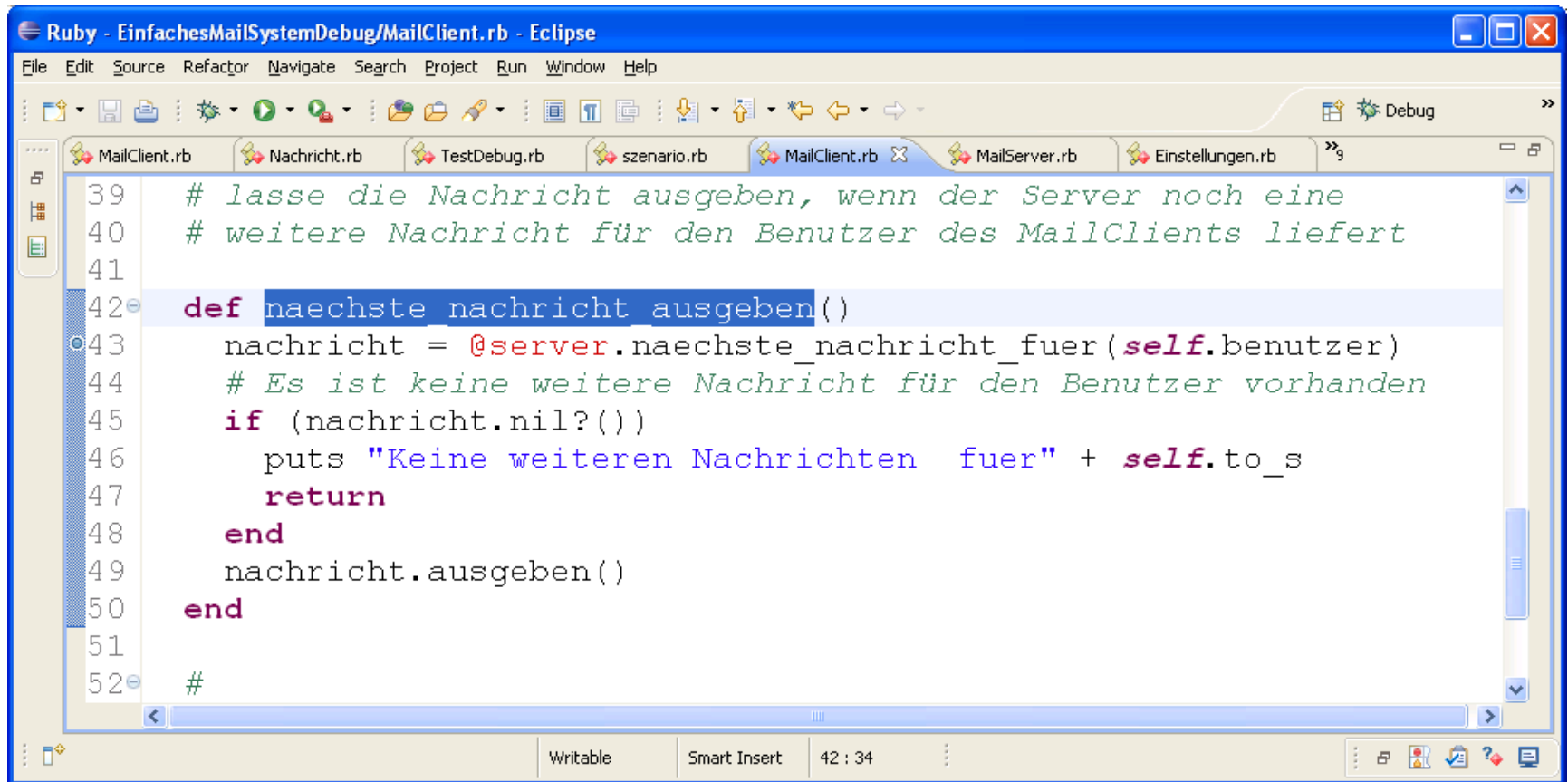
Nutzung des Debuggers

Vorbereitung

- Wir wollen uns nun genau den Ablauf der Methode `naechste_nachricht_ausgeben` anschauen.
- Dazu setzen wir im Texteditor in dieser Methode einen Haltepunkt (engl. Breakpoint) bei der ersten Zuweisung.
- Dies erreichen wir durch einen Doppelklick auf die graue Leiste links neben dem Editorfenster. Es erscheint ein kleiner blauer Kreis vor der markierten Stelle.
- Ein Breakpoint ist für den Debugger eine Haltemarkierung bei der Ausführung einer Methode.



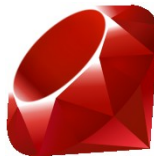
Breakpoint in *naechste_nachricht_ausgeben*



```
Ruby - EinfachesMailSystemDebug/MailClient.rb - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

MailClient.rb Nachricht.rb TestDebug.rb szenario.rb MailClient.rb MailServer.rb Einstellungen.rb

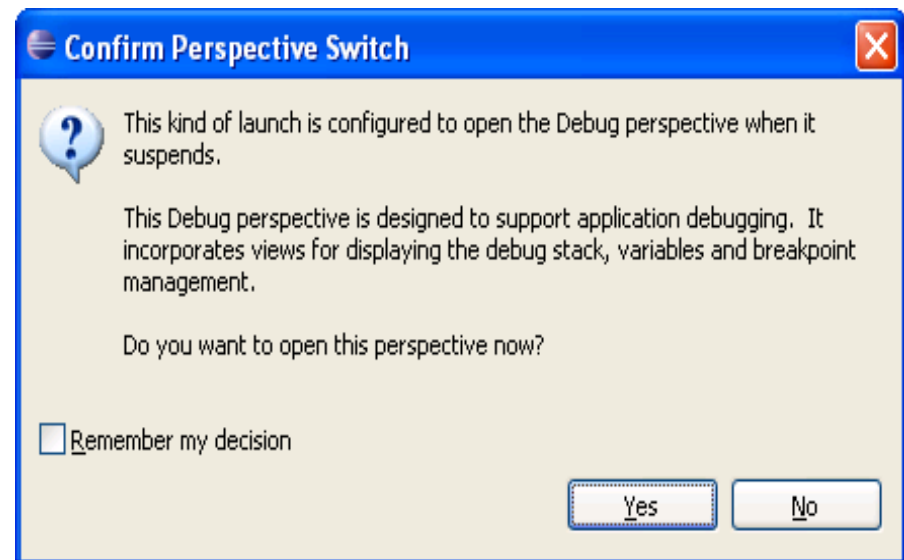
39 # lasse die Nachricht ausgeben, wenn der Server noch eine
40 # weitere Nachricht für den Benutzer des MailClients liefert
41
42 def naechste_nachricht_ausgeben()
43   nachricht = @server.naechste_nachricht_fuer(self.benutzer)
44   # Es ist keine weitere Nachricht für den Benutzer vorhanden
45   if (nachricht.nil?())
46     puts "Keine weiteren Nachrichten fuer" + self.to_s
47     return
48   end
49   nachricht.ausgeben()
50 end
51
52 #
```



Nutzung des Debuggers

Starten des Debuggers

- Im Kontextmenü des RubyScripts *szenario.rb* rufen Sie den Debugger durch *Debug As → RubyScript* auf.
- Es erscheint eine Message-Box, in der Sie gefragt werden, ob Sie in die **Debug Perspektive** wechseln wollen: Antworten Sie mit „yes“.
- Es erscheint die Debug-Perspektive in Eclipse.





Übersicht Debug-Perspektive

Debug - EinfachesMailSystemDebug/MailClient.rb - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Debug

Variables Breakpoints Expressions

Name	Value
Global Variables	
Class Variables	
nachricht	nil
self	MailClient (id=46779370)

nil

MailClient

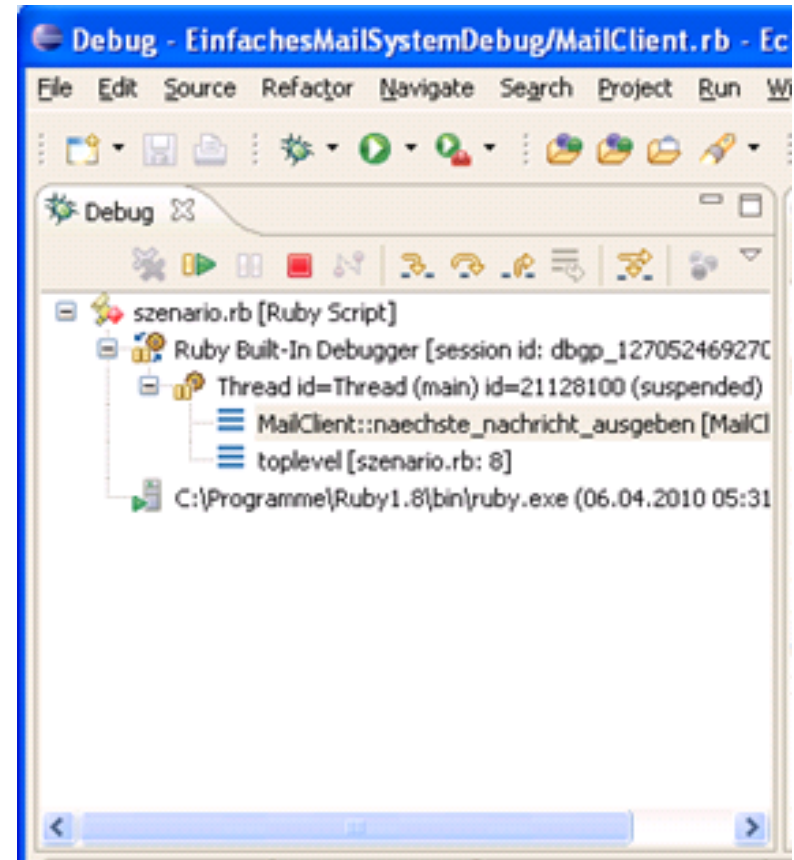
- initialize(benutzer, server)
- benutzer()
- sende_nachricht(empfaenger, nachrichten_text)
- empfangen_naechste_nachricht()
- naechste_nachricht_ausgeben()
- to_s()

```
39 # lasse die Nachricht ausgeben, wenn der
40 # weitere Nachricht für den Benutzer des
41
42 def naechste_nachricht_ausgeben()
```



Erläuterung zur Übersicht

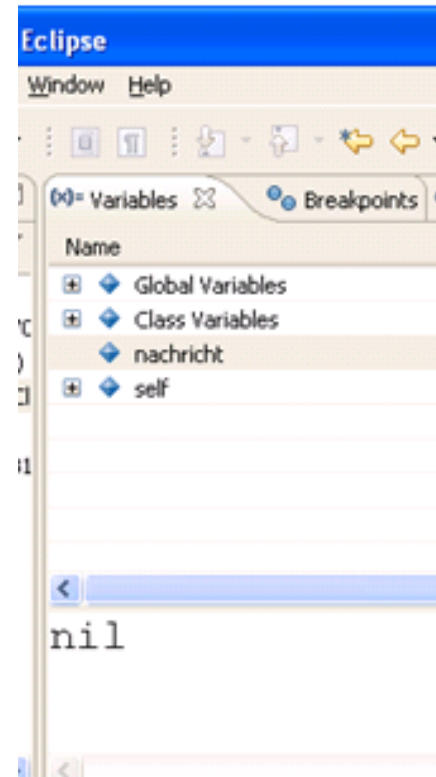
- **Oben links:** Das **Debug** Fenster. Hier sehen wir den aktuellen Ausführungstand der Methodenaufrufe. Obenauf liegt der Methodenaufruf, in dem der Debugger gerade anhält.
- **In der Mitte und unten:** Texteditoren, Outline und Konsole, wie wir Sie bereits aus der normalen Ruby-Perspektive kennen.





Erläuterung zur Übersicht

- **Oben rechts:**
 - Ein Reiter, der die Variablen und deren Werte zum aktuellen Ausführungszeitpunkt zeigt. Hier sind nur die Variablen angezeigt, die in und für die Methode zum sichtbar sind.
 - Ein Reiter, der alle im Projekt gesetzten Breakpoints zeigt.





Erläuterung zur Übersicht: In der Mitte vergrößert der Quelltext

```
Debug - EinfachesMailSystemDebug/MailClient.rb - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help

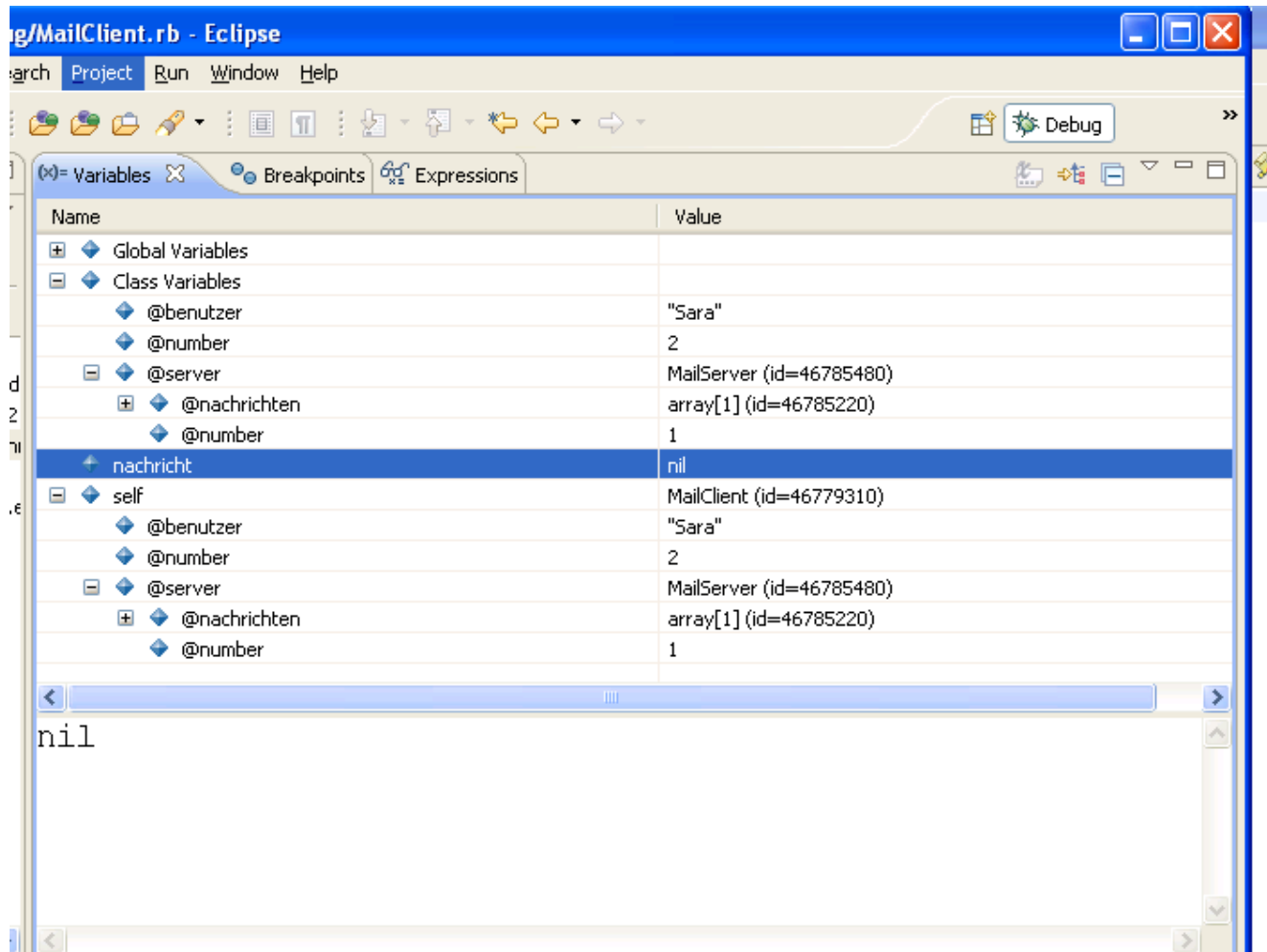
MailClient.rb Nachricht.rb TestDebug.rb szenario.rb MailClient.rb MailServer.rb Einstellungen.rb

39 # lasse die Nachricht ausgeben, wenn der Server noch eine
40 # weitere Nachricht für den Benutzer des MailClients liefert
41
42 def naechste_nachricht_ausgeben()
43   nachricht = @server.naechste_nachricht_fuer(self.benutzer)
44   # Es ist keine weitere Nachricht für den Benutzer vorhanden
45   if (nachricht.nil?())
46     puts "Keine weiteren Nachrichten fuer" + self.to_s
47     return
48   end
49   nachricht.ausgeben()
50 end
51
52 #
53 # Darstellung eines MailClients als lesbare Zeichenkette
54 #
55 def to_s()
56   return "MailClient von #{@benutzer}"
57 end
58
59 end

Writable Smart Insert 43 : 1
```




Die Variablensicht

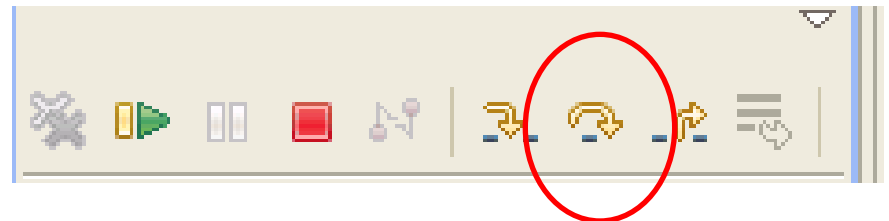


- Die Variable *nachricht* ist noch undefiniert, da die Zuweisung noch nicht ausgeführt wurde.



Zeilenweise Ausführen von Quelltext

- Wir können den Debugger nutzen, um eine Quelltext zeilenweise auszuführen.
- Dazu bedienen wir das Step Over-Symbol im Debug Fenster.
- Dann wird eine Zeile des Quelltextes ausgeführt und der Debugger hält in der nächsten Zeile.





Hineinschreiten in Methoden

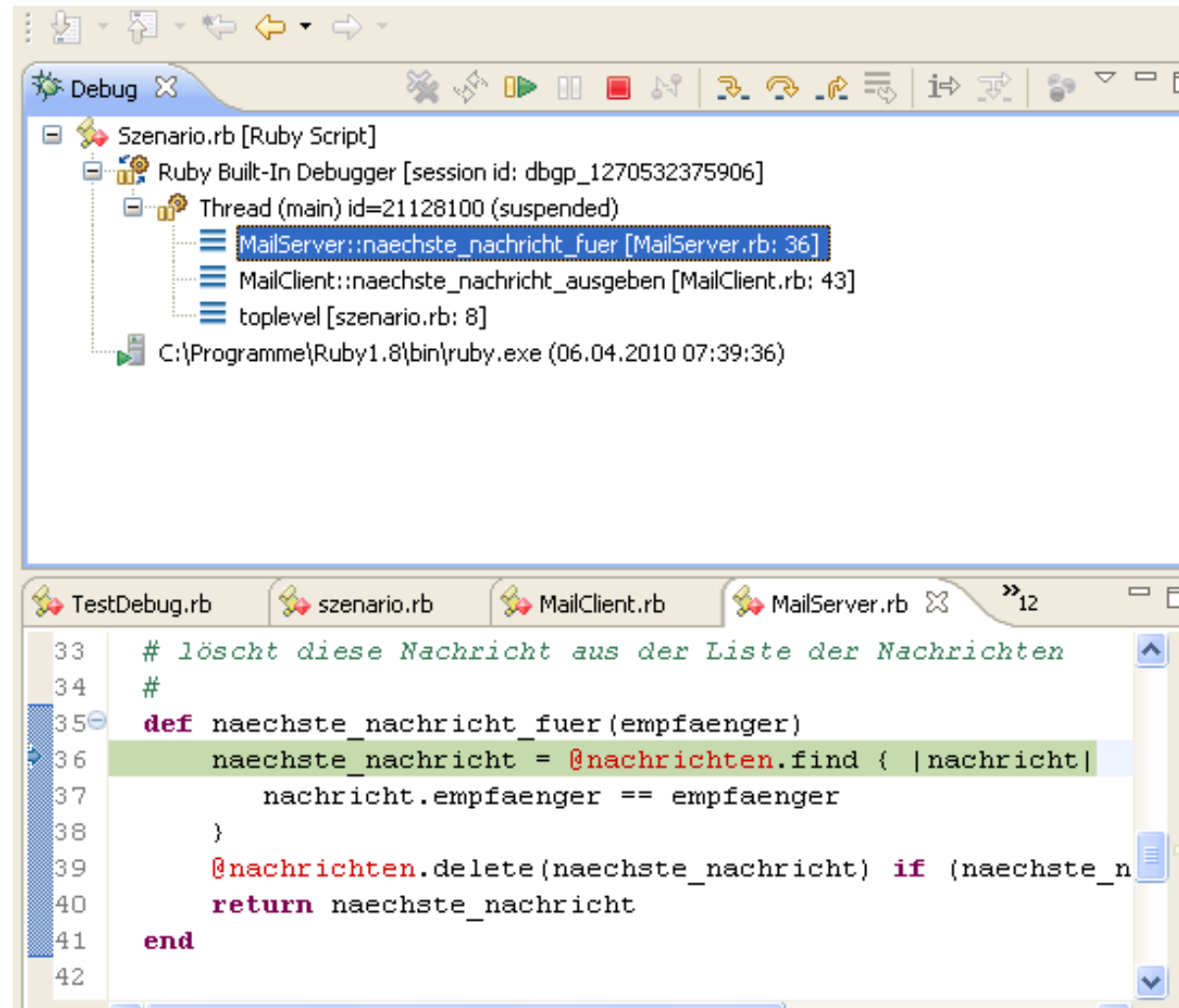
- Wir können den Debugger nutzen, um in eine Methode hineinzuschreiten, die in einer umgebenden Methode angerufen wird.
- Dazu bedienen wir das Step Into-Symbol im Debug Fenster.
- Wir starten das Szenario erneut und erzeugen die gleiche Ausgangsposition für den Debugger wie zuvor. Er hält an dem gesetzten Breakpoint.
- Die Zuweisung enthält auf der rechten Seite einen Methodenaufruf auf dem Server-Objekt.
- In diese Methode werden wir jetzt absteigen.





Debugger nach Step Into

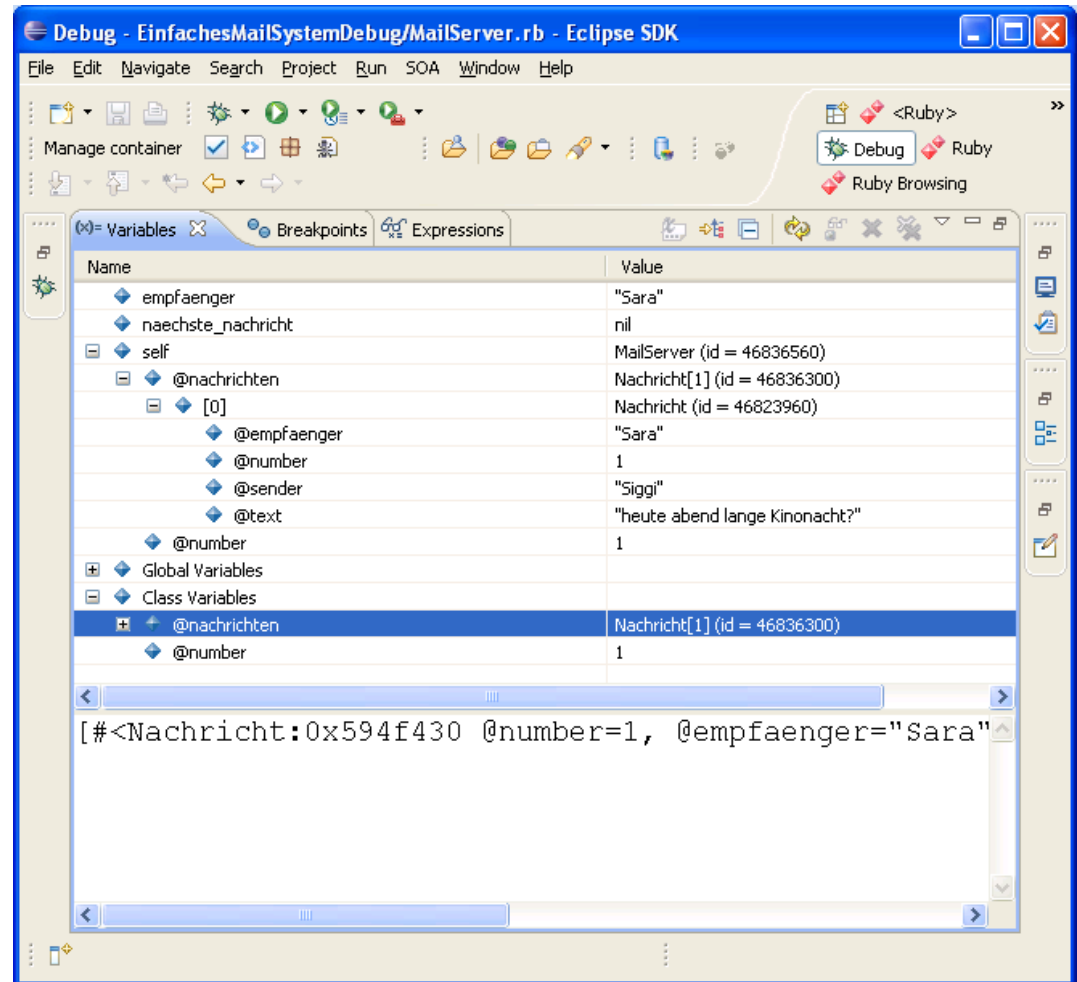
- Auf dem Methoden-Aufrufstack im Debug Fenster liegt an oberster Stelle jetzt die Methode des Servers.
- In der Scriptsicht ist in der Klasse **MailServer** die Zeile der Methode selektiert, an der der Debugger nach Step Into hält





Debugger nach Step Into

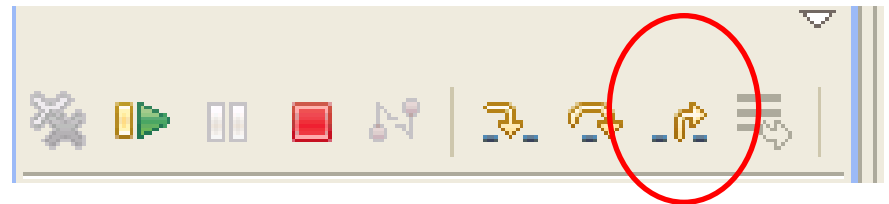
- Im Variablenreiter sind jetzt die Variablen der Klasse MailServer und die lokalen Variablen der Servermethode *naechste_nachricht_fuer* sichtbar.





Zurückschreiten aus Methoden

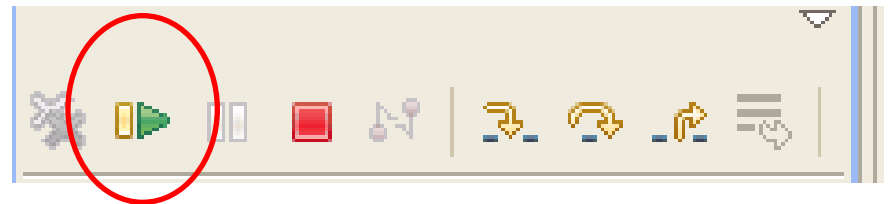
- Mit dem Zurückschreiten aus Methoden (Step Return) wird die oberste Methode im Debug Fenster ausgeführt und beendet. Der Debugger hält hinter der Zeile des Aufrufs dieser Methode.





Springen zum nächsten Haltepunkt

- Es lassen sich auch mehr als ein Breakpoint im Quelltext setzen.
- Dann wird mit Resume die Ausführung zwischen zwei Breakpoints abgespielt und der Debugger hält am nächsten Breakpoint in der Ausführungsreihenfolge.
- Die gesetzten Breakpoints in einem Projekt können wir uns jederzeit über den Reiter Breakpoints anschauen, diese deaktivieren oder löschen.





Das Objekt self

- In der Methode `sende_nachricht` wird der Sender der Nachricht mittels `self.benutzer` ermittelt.
- Dieser Aufruf ist äquivalent zum Aufruf der internen Methode `benutzer` ohne das Objekt `self` als Ziel des Methodenaufrufs.
- Alle internen Methodenaufrufe werden an das Objekt `self` geschickt.
- `self` ist das aktuelle Objekt, das eine Methode ausführt.
- Wie sich der Zustand des Objektes `self` während der Ausführung verändert, können wir sehr gut im Variablen Reiter des Debuggers nachvollziehen.

```
def
  sende_nachricht(empfaenger, nachrichten_text)
    nachricht =
      Nachricht.new(self.benutzer, empfaenger, nachrichten_text)
    @server.leite_weiter(nachricht)
end
```




Debugger Resume

- Da wir mit dem Debugger die Zustandsänderung von Objekten, der Inhalte von Variablen schrittweise nachvollziehen können, ist der Debugger sehr gut geeignet Fehlerursachen im Programm aufzuspüren, die durch falsche Werte oder falsche Objekttypen verursacht werden.
- Der Debugger ersetzt bei richtigem Gebrauch die puts Anweisungen, die nur der Fehlersuche dienen.
- **Ü4-9: Erzeugen Sie bitte ein Szenario für die Uhrenanzeige und rufen Sie die Methode setze_uhrzeit mit den Werten (27, 34) auf! Was ist das Ergebnis? An welcher Stelle verhält sich das Programm eigenartig? Suchen Sie diese Stelle mit dem Debugger!**



Zusammenfassung

- **Abstraktion** ist die Fähigkeit Details auszublenden um die Kerneigenschaften eines Problems zu erkennen
- **Modularisierung** ist die Zerlegung eines komplexen Problems in Teilprobleme, die separat gelöst werden können.
- **Objektdiagramm** Ein Objektdiagramm zeigt die Objekte und ihre Beziehungen zu einem bestimmten Zeitpunkt während der Laufzeit eines Programms
- **Objektreferenz** Variablen speichern Referenzen auf Objekte.
- **Objekterzeugung** Objekte können Objekte mit **new** erzeugen.
- **Interner Methodenaufruf** Methoden können Methoden der gleichen Klasse aufrufen. Der Adressat des Methoden-Aufrufs ist zur Laufzeit immer das aktuelle Objekt selbst.
- **Externer Methodenaufruf** Methoden können Methoden anderer Objekte aufrufen. Diese Objekte können Objekte anderer Klassen aber auch Objekte der eigenen Klasse sein.
- **self:** **self** ist die Referenz auf das aktuelle Objekt. Wenn auf einem Objekt eine Methode aufgerufen wird, dann kann man in der Methode mit **self** auf dieses Objekt Bezug nehmen.
- **self und interne Methodenaufrufe:** Interne Methodenaufrufe beziehen sich immer auf **self**.



Zusammenfassung

- **Eindeutigkeit von Methodennamen** In Ruby gibt es in einer Klasse **nur genau eine** Methodendefinition für einen Methodennamen.
- **Default-Parameter:** Default-Parameter sind formale Parameter in Methodendefinitionen, die bei der Definition mit Werten vorbelegt werden. Beim Aufruf einer solchen Methode können die Default-Parameter weggelassen werden. Werden für die Defaults beim Aufruf Werte übergeben, dann überschreiben diese Werte die Default-Werte.
- **Debugger** Ein Debugger ist ein Softwarewerkzeug mit dem die Ausführung des Programmes untersucht werden kann. Untersucht werden kann sowohl die Aufrufhierarchie der Methoden (**der Aufrufstack**) als auch der Zustand der beteiligten Objekte und die Werte der Variablen.