Marshaling & DRb

oder: Es muss nicht immer eine Datenbank sein

Marshaling:

- Grundlagen: Marshal, PStore
- SDBM, xDBM
- YAML

Distributed Ruby:

- Verteilte Anwendungen - ganz einfach

Marshaling

- Die Aufgabe
  - Wir haben (komplizierte) Objekte aufgebaut.
  - Diese sollen nun auch außerhalb des Kontexts des laufenden Programms zur Verfügung stehen - ohne komplette Neuberechnung.

- Beispiele
  - Speichern von Objekten, zwecks späterer Weiterverarbeitung
  - Übertragung von Daten auf ein anderes DV-System in plattformunabhängiger Weise
  - Client/Server-Anwendungen mit verteilten Objekten

- Der Lösungsansatz:
  - "***Marshaling***" der Objekte
  - Auch genannt: "Objekt-**Persistenz"**. Java-Begriff: "**Serialisierung**"

Marshaling

- „marshaling“ oder „marshalling“ ?
  - PONS:
    marshalling yard = Rangierbahnhof
  - LEO Online-Wörterbuch (dict.leo.org):
    marshalling yard — Britisches Englisch
    marshaling yard — Amerikanisches Englisch
  - en.wikipedia.org
    Marshalling: Eintrag existiert
    Marshaling: Link zu „Serialization“
  - Google:
    marshalling — ca. 2,5 Mio Treffer
    marshaling — ca. 1 Mio Treffer
  - **Pickaxe-Buch:**
    marshaling (daher hier gewählt als Schreibweise)

Marshaling

- Einfaches *Marshaling*: Das Modul "Marshal"

```
# Ein nicht-triviales Objekt aus weiteren Objekten:
geom_objects = [My::Rect.new(4, 5), My::Square.new(5),
                My::Circle.new(3), ... ]
# Das wollen wir speichern in Datei "my_obj_store.dat":
File.open("my_obj_store.dat","w") do |file|
    Marshal.dump(geom_objects, file)
end
```

```
# Später / u.U. anderes Programm, andere Plattform:
# Objekt rekonstruieren:
file = File.open("my_obj_store.dat")
my_obj_array = Marshal.load(file)
file.close
# Weitere Verwendung ...
```

Marshaling

- Die Methoden **Marshal.dump** und **Marshal.load**
  - Anstelle der IO-Objekte nehmen diese Methoden auch Objekte an, die auf "**to_str**" ansprechen. Dazu zählen insbesondere String-Objekte.
  - Synonym zu Marshal.load: **Marshal.restore**

- Nicht speicherbare Objekte
  - Es gibt einige Objekte, deren Natur sich nicht zum *Marshaling* eignen. In Ruby sind das Objekte der Klassen:

    ```
    IO, Proc, Binding
    ```

  - Ebenso nicht serialisierbar: *Singleton*-Objekte

Marshaling

- Eigene Eingriffe:
  - **Situation**:
    Nicht alle Teile eines Objekts sollten gespeichert werden, z.B. solche, die sich leicht rekonstruieren lassen.

  - **Lösung**:
    Callback-Methoden **`marshal_dump`** und **`marshal_load`** in betroffenen Klassen implementieren.

```
marshal_dump
# Erzeugt Objekt mit den benötigten Informationen
# Klasse dazu ist flexibel wählbar

marshal_load( anObject )
# Erhält das von marshal_dump erzeugte Objekt,
# rekonstruiert damit das Erforderliche
```

Marshaling

- Beispiel:
  - Die Klasse **My::Rect** speichere nicht nur die **Kantenlängen**, sondern auch noch **Fläche** und **Umfang** des Rechtecks in Attributen.
  - Letztere lassen sich aber einfach rekonstruieren und müssen daher nicht mitgespeichert werden.

Marshaling

- marshal_dump und marshal_load am Beispiel My::Rect

```
class Rect              # In Modul "My"...
  attr_reader :a, :b, :area, :circumference
  def initialize( a, b )   # a, b: Hier nur "Integer"
    @a, @b = a, b
    @area = get_area; @circumference = get_circ
  end
  def get_area; @a * @b;     end
  def get_circ;  2 * (a+b); end
  def marshal_dump
    [@a, @b]  # Nur das Wesentliche speichern!
  end
  def marshal_load( params )
    @a, @b = params
    # Rekonstruktion:
    @area = get_area; @circumference = get_circ
  end
end
```

Marshaling

- *Marshaling* mit der Klasse **PStore**
  - Mehrere Objektsammlungen simultan in einer Datei
  - Transaktionsschutz, mit Methoden **abort** und **commit**.
  - Zugriff auf die Objektsammlungen erfolgt Hash-artig.
    Da die meisten Sammlungen Objekthierarchien sind, spricht man aber von "root" anstelle von "key".
- Beispiel (aus dem Pickaxe-Buch):
  - Serialisierung eines String-Arrays und eines Binärbaums

```
require "pstore"

class T       # Grundlage des Binärbaums im Beispiel
  def initialize( val, left=nil, right=nil )
      @val, @left, @right = val, left, right
  end
  def to_a; [ @val, @left.to_a, @right.to_a ]; end
end
```

Marshaling

```
store = PStore.new("/tmp/store") # R/W-Zugriff
store.transaction do
  store['cities']=['London', 'New York', 'Tokyo']
  store['tree'] =
     T.new( 'top',
      T.new('A', T.new('B'),
      T.new('C', T.new('D', nil, T.new('E'))) ) )
end   # 'commit' implizit bei normalem Ende!
```

```
# Einlesen:
store.transaction do
  puts "Roots: #{store.roots.join(', ')}"
  puts store['cities'].join(', ')
  puts store['tree'].to_a.inspect
end
```

Marshaling

```
# Ergebnis:
Roots: cities, tree
London, New York, Tokyo
["top", ["A", ["B", [], []], []], ["C", ["D", [],
   ["E", [], []]], []]
```

- Übersicht zu den PStore-Methoden
  - Klassenmethoden:

    `new`
  - Normale Methoden:

    `[], []=, roots, root?,`
    `path,`
    `abort, commit,`
    `transaction`

Marshaling

- *Marshaling* mit DBM (und "Verwandten")
  - Auf Unix-Systemen gibt es seit langem eine einfache Datenbank-Vorstufe unter dem Namen "dbm".
  - Mittels "dbm" lassen sich prinzipiell beliebige Datenstrukturen einem Suchschlüssel zuordnen und über diesen Schlüssel persistent speichern sowie effizient wiederherstellen.
  - Dies entspricht dem Verhalten einer persistenten Hash-Tabelle! Perl hatte daher "dbm" mit einem einfachen Hash-artigen, transparenten Zugriffsmechanismus versehen.
  - Ruby folgte mit Klasse "DBM" dieser Tradition!

- Alternativen
  - Unix: Implementierungsvarianten! dbm, gdbm, ndbm, sdbm, …
  - Windows: I.d.R. unbekannt.
  - Ruby (1.8): SDBM in Ruby implementiert → Immer vorhanden!

Marshaling

- *Marshaling* mit DBM (und "Verwandten")
  - Objekte der Klasse SDBM verhalten sich ähnlich wie Hashes, sind aber keine!
  - Bei Bedarf ist eine Umwandlung möglich mit **to_hash.**
  - Einschränkung: *key*- wie *value*-Objekte müssen Strings sein!
  - Konsequenz: ggf. kombinieren mit "Marshal" !
- Beispiel:

```
require "sdbm"

SDBM.open("my_data.dbm") do |d|
  d["cities"] = Marshal.dump( my_array_of_city_names )
  d["tree"] = Marshal.dump( my_T_obj )
end
```

```
# Später:
d = SDBM.open("my_data.dbm")
puts Marshal.load( d["cities"] ).join(', ')
d.close       # puts ergibt: "London, New York, Tokyo"
```

Marshaling

YAML: Vorstellung auf www.yaml.org:

- *YAML™ (rhymes with "camel") is a straightforward machine parsable data serialization format designed for human readability and interaction with scripting languages such as Perl and Python. YAML is optimized for **data serialization**, configuration settings, log files, Internet messaging and filtering. YAML™ is a balance of the following design goals:*
  - *YAML documents are very readable by humans.*
  - *YAML interacts well with scripting languages.*
  - *YAML uses host languages' native data structures.*
  - *YAML has a consistent information model.*
  - *YAML enables stream-based processing.*
  - *YAML is expressive and extensible.*
  - *YAML is easy to implement.*

- Bemerkungen
  - YAML ist einfacher, aber effizienter (in der Verarbeitung) als XML
  - Keine Rekonstruktion von Attributen wie etwa bei marshal_load() !

Marshaling

- *Marshaling* mit YAML
- Beispiel:

```
require "yaml"

class Rect
 … # hier die bisherigen Methoden…
  def to_yaml_properties
    %w{ @a @b }
  end
end
# Analog: zu sichernde Attribute auch für Square, Circle
```

```
# Später: lesbaren, speicherbaren YAML-String erzeugen
data = YAML.dump(geom_objects)
```

```
# Viel später: Rekonstruktion
some_objects = YAML.load(data)
```

- Demo: Aussehen der YAML-Strings

Marshaling

JSON: Vorstellung auf www.json.org:

- ***JSON*** *(JavaScript Object Notation) is a lightweight data-interchange format.*
  - *It is easy for humans to read and write.*
  - *It is easy for machines to parse and generate.*
  - *It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999.*
  - *JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.*
  - *These properties make JSON an ideal data-interchange language.*

- Bemerkungen
  - Ganz offensichtlich ein Mitbewerber von YAML
  - Inzwischen weit über JavaScript verbreitet

Marshaling

- *Marshaling* mit JSON (JavaScript Object Notation: json.org, RFC4627)
- Beispiel:

```
require "json"

class Rect
 … # hier die bisherigen Methoden…
   def to_json(*a)
      {
        'json_class'   => self.class.name,
        'data'         => [ self.a, self.b ]
      }.to_json(*a)
    end

    def self.json_create(o)
      new(*o['data'])
    end
end # Analog für Square, Circle
```

```
# Später: lesbaren, speicherbaren YAML-String erzeugen
data = geom_objects.to_json
```

```
# Viel später: Rekonstruktion
some_objects = JSON.parse(data)
```

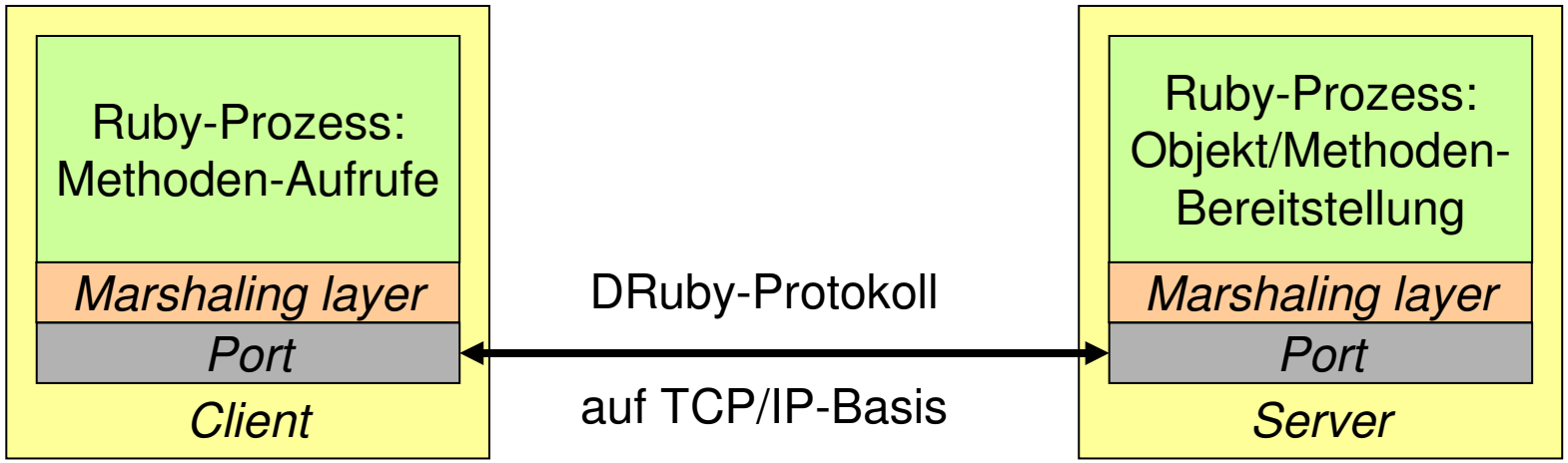
Distributed Ruby

Verteilte Anwendungen - ganz einfach
(RMI mit Ruby)

Distributed Ruby

- Objekte auf verteilten Systemen sind realisierbar mittels
  - Mechanismen zur Objekt-Persistenz (*Marshaling*)
  - Netzwerk-Protokollen, insb. TCP/IP

- Ruby beherrscht beides, daher war der Weg zu DRuby nicht mehr fern:



- Ergebnis:
  - Leistungen ähnlich wie elementare CORBA- oder Web Services-Funktionen sowie Java RMI, aber mit sehr geringem Aufwand!

Distributed Ruby

- Beispiel für ein Server-Programm

```
require "drb"

class ComputeServer
  def greetings
    "Hello from " + `hostname`.strip
  end
  def add( *args )
    args.inject{|s, x| s+x}
  end
end

aServerObj = ComputeServer.new
# Dieses Objekt soll nun den Clients dienen:
DRb.start_service('druby://localhost:12349',aServerObj)
# Normales Prozess-Ende verhindern:
DRb.thread.join
```

Distributed Ruby

- Beispiel für ein dazu passendes Client-Programm

```
require "drb"

DRb.start_service
obj = DRbObject.new( nil,
    'druby://servername.domain.tld:12349' )

# Methoden des entfernten Objekts nun lokal verwendbar:
puts obj.greetings  # "Hello from lx3-beam"
puts "2+3+4+5 = #{obj.add(2,3,4,5)}"
# etc., siehe Demo.
```

- Kriterien zur Verwendbarkeit
  - Ports verfügbar? Keine Kollisionen? Router/Firewall-Aspekte??
  - Nur Methoden eines Objekts pro Port: Ausreichend?
  - Performance: 50 remote calls / sec @ 233 MHz-CPU ok?

Datenbank-Anbindung

Einfache Tabellen mit CSV
RDBMS und SQL: Grundlagen
DB-spezifische Module, Beispiel MySQL
Ruby/DBI
ORM: *Object-relational mapping*

RDBMS und SQL: Grundlagen

- 1974: Grundlagenartikel „*A Relational Model of Data for Large Shared Data Banks“ von* Edgar F. Codd
- 1976: IBM definiert SEQUEL/2
  - Umbenennung in SQL aus rechtlichen Gründen
- 1980er Jahre: Oracle!
- 1986: Erster SQL-Standard, ANSI
- 1987: Erster SQL-Standard, ISO
- 1989: SQL89 (ANSI)
- 1992: SQL-92 bzw. **SQL2**
  - Noch heute die wichtigste Grundlage!
- 1999: SQL:1999 bzw. SQL3, ISO/IEC 9075
  - Noch nicht in allen Datenbanken implementiert
- 2003: SQL:2003 (noch selten anzutreffen!)

RDBMS und SQL: Grundlagen

- SQL: Structured Query Language
  - Einfach zu lernen, an natürliche Sprache angelehnt (Englisch)
  - Zielgruppe: Anwender
  - Deklarativ: Das Ziel wird beschrieben, nicht der Weg dorthin

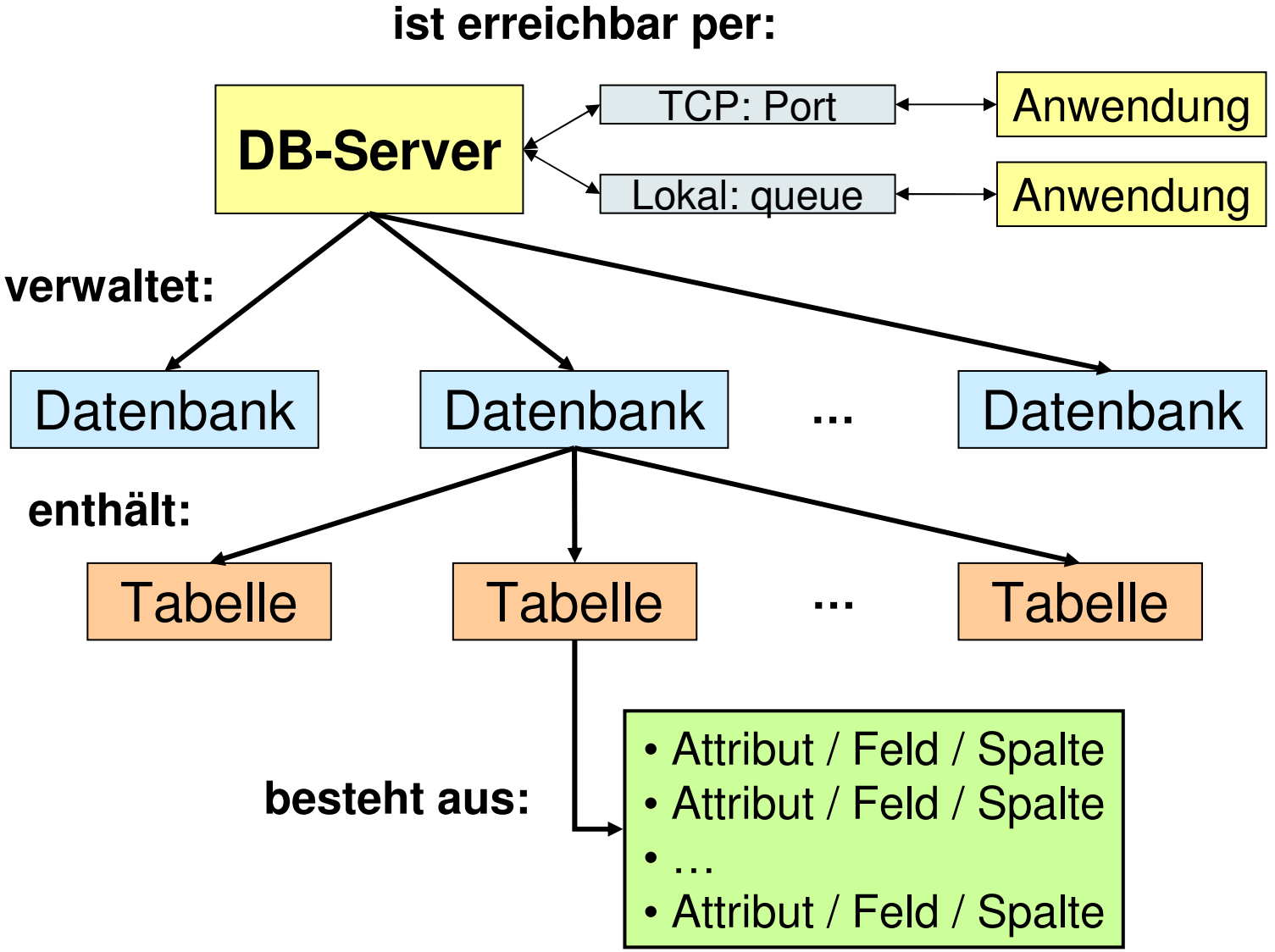
- Beachte: Reale RDBMS (relationale Datenbankmanagement-Systeme)
  - implementieren nicht alle Eigenschaften eines SQL-Standards
  - ergänzen diesen andererseits durch proprietäre Erweiterungen
- Die Folgen:
  - Die Portabilität von SQL-Statements leidet
  - Es gibt DB-spezifische APIs für diverse Programmiersprachen, z.B. für Oracle, PostgresQL, SAP-DB, mySQL, …
  - Ansätze zur Vereinheitlichung: ODBC, DBI

RDBMS und SQL: Grundlagen

- SQL: Structured Query Language, 4 Sprachebenen
- DQL: Database Query Language
  - SELECT
- DML: Database Manipulation Language
  - INSERT, UPDATE, DELETE
- DDL: Database Definition Language
  - CREATE, ALTER, DROP
- DCL: Database Control Language
  - GRANT, REVOKE

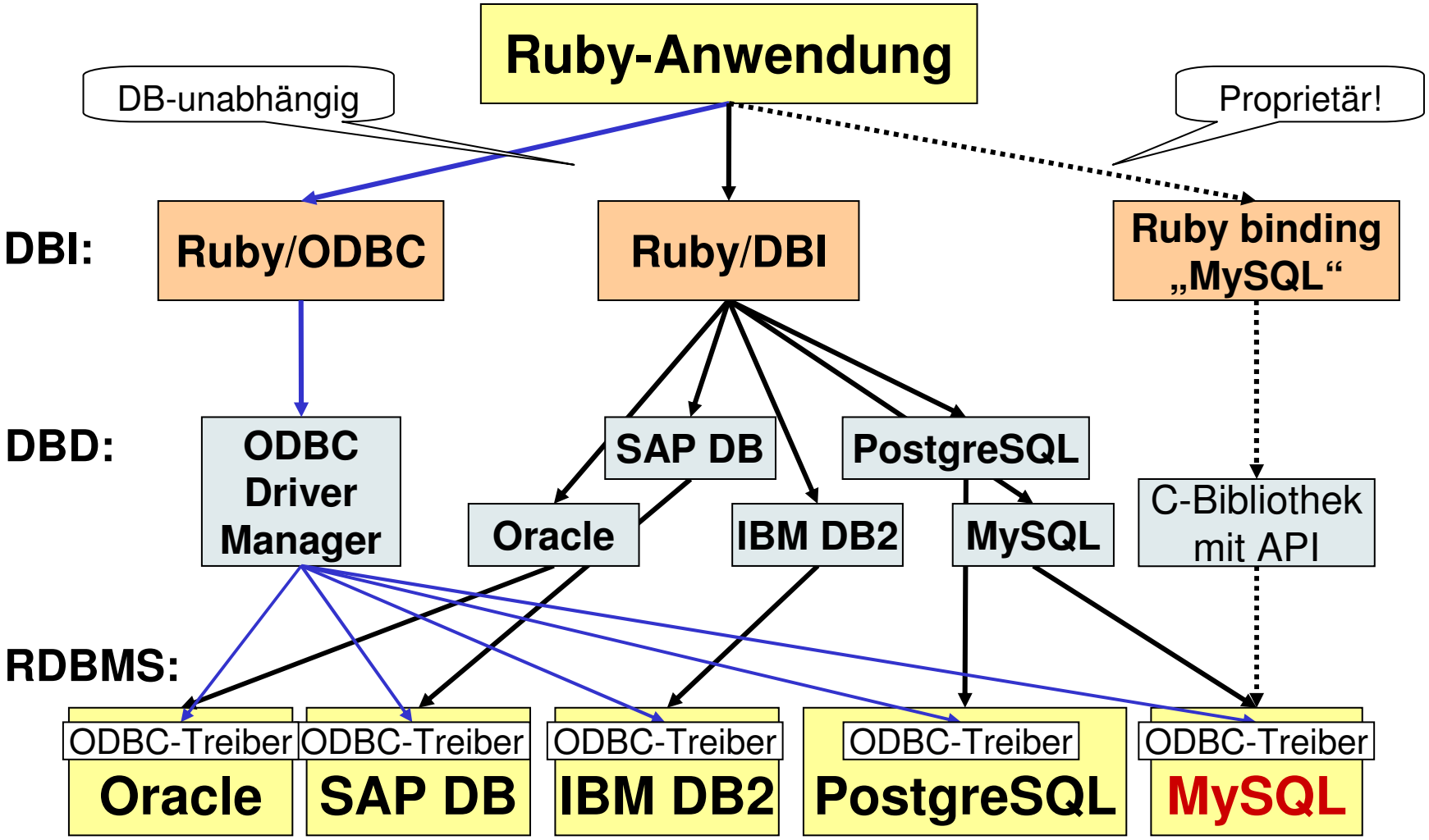
RDBMS und SQL: Grundlagen

- Ein DBMS



Umgang mit mehreren RDBMS

- Möglichkeiten der DB-Anbindung einer Ruby-Anwendung



DB-Module für Ruby

- **MySQL/Ruby**
  - Quelle: http://www.tmtm.org/en/mysql/ruby
  - Autor: Tomita Masahiro
  - Version: 2.8 (2008-09-29)
  - Konzept: Binding zum C-API „mysql“
  - Einführung: http://www.kitebird.com/articles/ruby-mysql.html

- **Ruby/DBI**
  - Quelle: http://rubyforge.org/projects/ruby-dbi/
  - Autor: Michael Neumann et. al.
  - Version: 0.4.1 (2008-11-28)
  - Konzept: 2-Schicht-Ansatz, DB-neutral aus Entwicklersicht
  - Einführung: http://www.kitebird.com/articles/ruby-dbi.html
  - Bemerkungen: Analog zum verbreiteten Perl DBI

ORM: *Object Relational Mapping*

- **Konzept**
  - Klasse ←→ Tabelle
  - Exemplar ←→ Zeile
  - Methode (Getter/Setter) ←→ Spalte
  - Zuordnungen
    - Automatisch dank sinnvoller Defaults
    - Und/oder: explizit zu konfigurieren

- **ActiveRecord** – eine ORM-Implementierung für Ruby
  - Quelle: http://rubyforge.org/projects/activerecord/
  - Autor: David Heinemeier Hansson
  - Version: 2.2.2 (2008-11-21)
  - Konzept: ORM, 2-Schicht-Ansatz, DB-neutral
  - Bemerkungen: Ein Ruby Gem als „**Rails**“-Spinoff
  - Einführung: „Active Web Development with Rails“, Kap. 14

DB-Anbindung: Demo-Code

- **SQL**
  - Neue DB anlegen, Rechte vergeben
  - Tabelle löschen, anlegen
- **Ruby/DBI**
  - Tabelle „books“: löschen, anlegen
  - Neue Tabellenzeilen erzeugen
  - DB-Abfragen, iterieren durch die Ergebnis-Zeilen
    - Basis: SQL
- **ActiveRecord**
  - Tabelle programmatisch anlegen & abbauen („*migration*“)
  - CRUD: Create, Read, Update, Delete
  - Verschiedene Beispiele zur objekt-orientierten Kapselung von DB-Zugriffen und SQL.
  - Gleiche Tabelle „books“ bzw. „morebooks“

- Beispiel-Code: Wird in Verzeichnis „11“ bereitgestellt!