

Advanced O/R Mapping with Glorp

Introduction

➔ Advanced

- ➔ Assumes basic familiarity with O/R and Glorp concepts
- ➔ Touches on a variety of topics
- ➔ Motivating example/theme

Metaphor/Motivating Example

- ➔ Ruby on Rails/ActiveRecord
 - ↳ Reading Database Schema
 - » Some interesting mapping issues
 - ↳ Creating Mappings Dynamically
 - » Conventions/Grammar
 - ↳ APIs
 - » Querying
 - » Globals, Singletons, Transactions
 - » OO
 - ↳ Web Integration

GLORP

- ➔ Open Source (LGPL(S)) mapping library
- ➔ Metadata-driven
- ➔ Persistence by reachability
- ➔ Transaction-based

Ruby on Rails

- “Opinionated software”
- Rails
 - ↳ Go really fast, but only in one direction
- Reaction against J2EE/.NET
- “Greenfield” projects
- Ruby-Based
 - ↳ Smalltalk-like, scripting language
 - ↳ Some very Smalltalkish “tricks” in rails
- ActiveRecord pattern for persistence

Architecture: Glorp and ActiveRecord

- Metadata vs. convention-driven
- Glorp: Explicit metadata
 - ↳ Tables
 - ↳ Classes
 - ↳ Descriptors/Mappings
- ActiveRecord
 - ↳ Strict naming conventions
 - ↳ Aware of language forms
 - ↳ Hints at the class level
 - ↳ Code Generation (mostly for web)

Brokers

➔ Glorp

- ↪ Single broker (session)
 - » Responsible for object identity
 - » Manages automatic writes
- ↪ Multiple clients use multiple sessions
- ↪ Independent of other frameworks

➔ ActiveRecord

- ↪ Classes as brokers
 - » No object identity
 - » Single global session
- ↪ Explicit writes
- ↪ Tightly integrated with web framework

Domain Model

➔ Glorp

- ➔ No metadata in domain classes
- ➔ Premium on flexibility, ability to optimize
- ➔ Expect existing classes, schema

➔ ActiveRecord

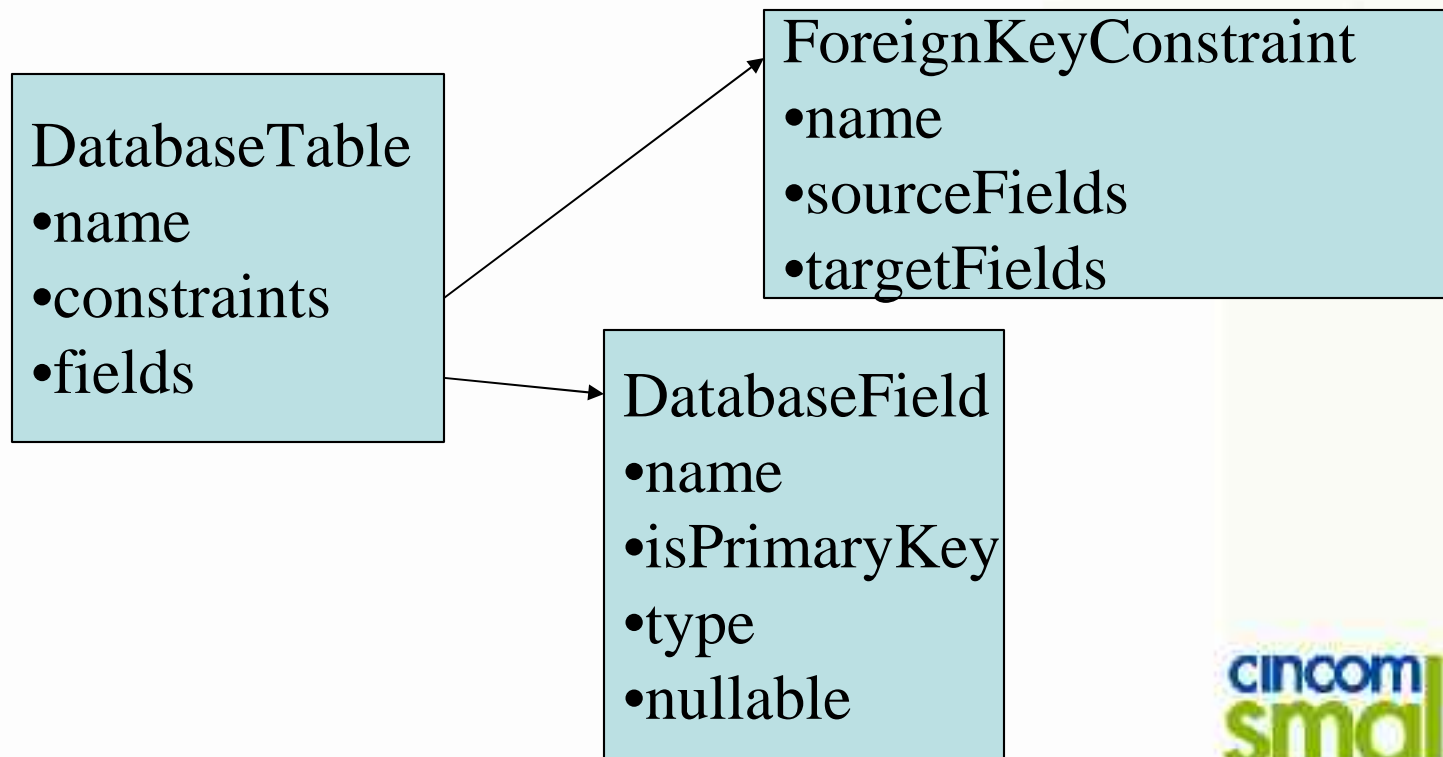
- ➔ Premium on simplicity
- ➔ Minimal metadata, but on domain classes
- ➔ May not even be a domain model
 - » Use ActiveRecord directly
 - » Instance variables as properties

Goal

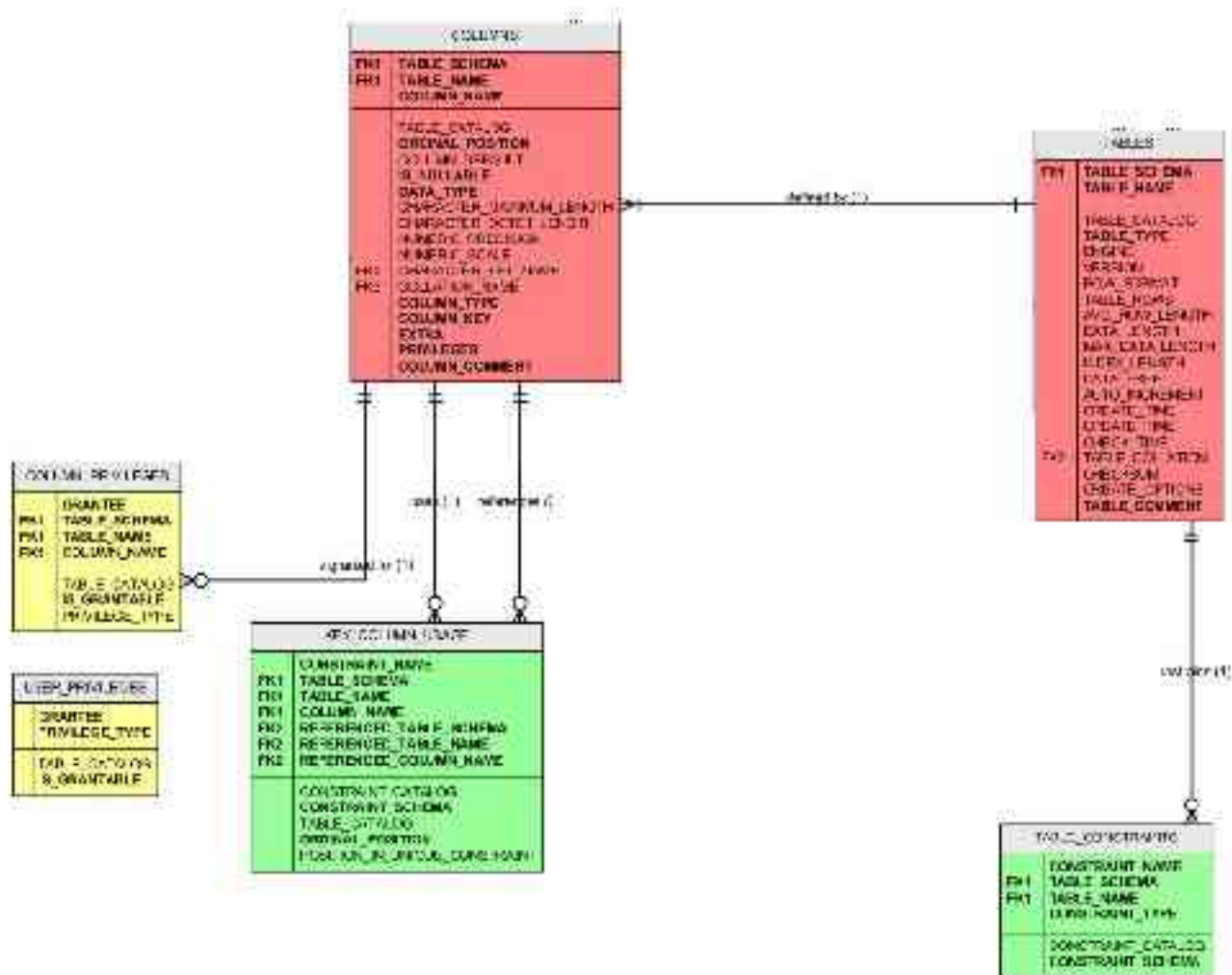
- ➔ Can we provide some of the benefits, but without losing our advantages
- ➔ “Hyperactive Records”
 - ↪ Automatic persistence
 - ↪ Convention-driven
 - ↪ But be less restrictive
 - ↪ Use a bit more information (constraints)
 - ↪ Allow a graceful transition

Issue: Reading Schema

- Before we can automate, we need to read the database schema.
- A nicely recursive problem



INFORMATION_SCHEMA



Mapping DatabaseField

DatabaseField

- name
- isPrimaryKey*
- type
- nullable

COLUMNS

- TABLE_NAME
- COLUMN_NAME
- DATA_TYPE
- IS_NULLABLE

```
table := self tableNamed: 'columns'.  
(aDescriptor newMapping: DirectMapping)  
  from: 'name'  
  to: (table fieldNamed: 'column_name').
```

Mapping #isPrimaryKey

- ST: a boolean value
- DB: primary key constraints are entities
- Columns used in a constraint are listed in `key_column_usage`
- For a field, do any primary key constraints exist that make use of it
- Mapping a two-level join to a boolean

Mapping #isPrimaryKey

```
(aDescriptor newMapping: DirectMapping)  
  from: #isPrimaryKey  
  to: [:each |  
       each primaryKeyConstraints notEmpty].
```

- ➔ Direct mapping, but to an expression
 - ➔ “each” is the field we’re referring to
 - ➔ primaryKeyConstraints is another relationship
 - ➔ notEmpty is a subselect operation

Mapping #primaryKeyConstraints

```
(aDescriptor newMapping: ToManyMapping)
  attributeName: #primaryKeyConstraints;
  referenceClass: PrimaryKeyConstraint;
  beForPseudoVariable;
  useLinkTable;
  join: (Join
    from: colNameSrc to: colNameLink
    from: schemaSrc to: schemaLink
    from: tableNameSrc to: tableNameLink;
  reverseJoin: (Join
    from: cnstrtNameLink to: cntstrtNameTarget
    from: schemaLink to: schemaTarget
    from: tableNameLink to: tableNameTarget
    from: 'PRIMARY KEY' to: targetType
```

➔ **Aside:** This is why I hate domain keys

Subselects

- In queries, several “collection” operations that turn into different kinds of subselects
- isEmpty/notEmpty
- select:
- anySatisfy:/noneSatisfy:
- sqlCount

```
read: Customer
  where: [:each |
    (each orders select: [:order |
      order amount > 1000])
      sqlCount > 5].
```

Reading Schema Summary

- sourceFields and targetFields worse
- Information_schema variations, limits
- Works for Oracle, Postgresql
- No changes at all to the domain model
 - ↳ But easier because read-only
 - ↳ Several pseudoVariables
- Good motivation for automatic mapping

Back to ActiveRecord

→ Glorp metadata

- ↳ defined in DescriptorSystem
- ↳ Methods for tables, classes, mapping
- ↳ E.g. #descriptorForCustomer:
- ↳ Lists allTables, allClasses

ActiveRecord DescriptorSystem

- ➔ All subclasses of ActiveRecord
- ➔ Read allTables from the database
 - ↪ For each class name, find table name
 - ↪ Find link tables from constraints or hints
- ➔ For each inst var/field name, figure out the mapping
 - ↪ Naming convention
 - ↪ Database constraints

Aside: Inflector

- ➔ Ruby on Rails class
- ➔ Knows basic grammar forms (English)
- ➔ Knows class/inst var/field/table naming and capitalization
 - ↳ Person class -> PEOPLE table
 - ↳ OVERDUE_ORDER_ID -> overdueOrder
- ➔ Big ball of regular expressions

Adding Mappings

- ➔ For each instance variable
 - ↪ Look for direct field match
 - ↪ name => NAME
- ➔ For each column
 - ↪ Look for an instance variable name
 - ↪ Look for foreign key constraints
 - ↪ Look for hints
 - ↪ Use naming conventions
 - ↪ ACCOUNT_ID => account

Aside: Hints

- Ruby on Rails uses class data to tell it how to create relationships that are ambiguous
- `hasMany`, `hasAndBelongsToMany`
- `tableName` (added)

Adding Mappings

- Created direct mappings
- Created relationships using fields
- Create relationships without fields
 - ↳ Link table
 - ↳ Reverse foreign key
- Find target class/table based on name

Demo

Incremental Setup

- We want to do as little work as necessary
- How to “tweak” an automatically generated mapping
- `#mappingNamed:do:`

```
self mappingNamed: #bankCode do:  
  [:mapping | mapping type: Integer].
```

References

➔ GLORP

↳ <http://www.glorp.org>

↳ <http://glorp.sourceforge.net>

➔ Ruby on Rails

↳ <http://www.rubyonrails.org/>

↳ Lots of other links

The End

Subselects

- ➔ In SQL terms, a nested query
- ➔ Many different uses
 - ↳ tend to make the brain hurt
- ➔ Glorp provides various shortcuts for specific Smalltalk semantics, plus a general mechanism
 - ↳ sometimes also make the brain hurt
 - ↳ still settling on semantics, naming

Subselect Example

➔ e.g. expand out the previous

```
... where: [:each | each members  
anySatisfy: [:eachMember | eachMember  
name like: 'Alan%']]].
```

```
SELECT <project fields>  
FROM PROJECT t1  
WHERE EXISTS (  
    SELECT <whatever> FROM MEMBER s1t1 WHERE  
    s1t1.proj_id = t1.id)
```

Aggregating

- Two forms of aggregating
- At the query level
 - ↳ aQuery retrieve: [:each | each value sum]
 - ↳ Puts an aggregate into the fields of the SQL
 - ↳ `SELECT ... SUM(t1.value)`
- Within a where clause
 - ↳ where: [:each | (each value sqlSum) > 10]
 - ↳ Creates a subselect of that aggregate
 - ↳ `SELECT ... WHERE (SELECT SUM (s1t1.value) FROM ... WHERE ...) > 10`
- min, max, average, count, etc.

Still More Aggregating

- ➔ Also within a where clause

```
expression count: [:x | x attribute]
```

- ➔ or more generally

```
expression
```

```
count: [:x | x attribute]
```

```
where: [:x | x something = 5].
```

- ➔ More awkward than

```
expression sqlCount
```

- ➔ Not really more powerful

General Aggregations

➤ General facility

```
read: GlorpCustomer
  where: [:each | each
    (each
      aggregate: each accounts
      as: #countStar
      where: [:acct | acct price > 1000])]
    = 1].
```

➤ Really awkward

➤ More general

- Only requires the underlying function to exist

Select:

- `count:where:` suggests a more Smalltalk-like form

```
where: [:each |  
    (each users select: [:eachUser |  
        eachUser name like: 'A%'])  
    sqlCount > 3].
```

- Or we could apply other operations e.g. `anySatisfy:` to the filtered collection.

Fully General Subselects

- A subselect is represented by a query.

```
aCustomer accounts
```

```
  anySatisfyExists: [:eachAccount |  
    eachAccount in:  
      (Query  
        read: GlorpBankAccount  
        where: [:acct |  
          acct balance < 100])]]].
```

- Very general, but awkward
- Often putting queries into block temps, setting retrieve: clauses, etc.

Correlated Subselects

- Are the internal selects effectively constants, or do they refer back to things in outer queries
- Slower in database, but more powerful

```
read: StorePackage where: [:each |
  | q |
  q := Query read: StorePackage
    where: [:eachPkg |
      eachPkg name = each name].
  q retrieve: [:x | x primaryKey max].
each username = 'aknight' & (each primaryKey
= q)].
```

OK, No More Subselects

- Yes, these are complicated
- Sometimes you need them
- The tests are a good resource for code fragments to copy from
- Or just experiment until you get SQL (and results) you want