

CS 683 Emerging Technologies
Fall Semester, 2004
Doc 26 Glorp
Contents

References.....	2
Some Problems with Objects & Relational Databases	3
Glorp	5
Unit of Work	5
Types of Queries.....	12
Block Query Language	14
One to Many	17
Classes.....	17
Database Tables	19
Mapping.....	20
Sample Usage	23
Many to Many	26
Tables.....	26
Classes.....	27
Mapping.....	29
Some Usage.....	32

Copyright ©, All rights reserved. 2004 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (<http://www.opencontent.org/opl.shtml>) license
defines the copyright on this document.

References

Glorp web site <http://glorp.org/>

Glorp Tutorial, Roger Whitney

Hibernate in Action, Bauer & King, Manning, 2005

Some Problems with Objects & Relational Databases

- Granularity
- Mapping of Base types
- Subtypes
- Identity
 - Database identity (primary key)
 - Object identity (java ==)
 - Object equality (java equals())
- Associations
- Object graph navigation
- Separating persistence & business logic

Issues with O/R layers

- What are the requirements for a persistent class
- How is the mapping defined
- How do we map inheritance hierarchies
- How does database identity relate to object identity & equality
- How does persistence logic interact with business objects
- What is the lifecycle of a persistent object
- What are the facilities for sorting, searching & aggregating?
- What is the query language
- How efficient are the queries
- How are transactions & concurrency handled

Glorp Unit of Work

Transaction that tracks all object read from database

Following are equivalent

```
session beginUnitOfWork.  
foundPerson :=  
  session  
    readOneOf: Person  
    where: [:each | each lastName = 'Chan'].  
foundPerson firstName: 'RamJet'.  
session commitUnitOfWork
```

```
session inUnitOfWorkDo:  
  [foundPerson :=  
    session  
      readOneOf: Person  
      where: [:each | each lastName = 'Chan'].  
    foundPerson firstName: 'Ramjet']
```

SQL executed

```
SELECT t1.first_name, t1.last_name  
FROM PEOPLE t1  
WHERE (t1.last_name = 'Chan') LIMIT 1  
Begin Transaction  
UPDATE PEOPLE SET first_name = 'Jose',last_name = 'Chan'  
WHERE last_name = 'Chan'  
(0.06 s)  
Commit Transaction
```

Adding new Objects

```
session beginUnitOfWork.  
person := Person first: 'Pete' last: 'Chan'.  
session register: person.  
session commitUnitOfWork.
```

```
person := Person first: 'Pete' last: 'Chan'.  
session beginUnitOfWork.  
session register: person.  
session commitUnitOfWork.
```

SQL executed

```
Begin Transaction  
INSERT INTO PEOPLE (first_name,last_name) VALUES  
( 'Pete','Chan')  
(0.019 s)  
Commit Transaction
```

Deleting Objects

```
session inUnitOfWorkDo:  
  [foundPerson :=  
    session  
      readOneOf: Person  
      where: [:each | each lastName = 'Chan'].  
    session delete: foundPerson]
```

SQL executed

```
Begin Transaction  
DELETE FROM PEOPLE WHERE last_name = 'Chan'  
(0.043 s)  
Commit Transaction
```

Updating Object from Database

```
foundPerson :=  
  session  
    readOneOf: Person  
    where: [:each | each lastName = 'Chan'].
```

Time passes and database may be modified by other process

```
session refresh: foundPerson
```

Objects spanning different Units of Work

Need to register the object in the second unit of work

```
foundPerson :=  
  session  
    readOneOf: Person  
    where: [:each | each lastName = 'Chan'].
```

Blah blah

```
session inUnitOfWorkDo:  
  [ session register: foundPerson. foundPerson firstName: 'Jose']
```

Objects spanning different Sessions

Avoid doing this in GLORP

```
foundPerson :=
```

```
  session
```

```
    readOneOf: Person
```

```
    where: [:each | each lastName = 'Chan']
```

```
session := GlorpSession new.
```

```
session system: (PersonDescriptor forPlatform: login database).
```

```
session accessor: accessor.
```

```
session inUnitOfWorkDo:
```

```
  [ x := session refresh: foundPerson.
```

```
  x firstName: 'Roger']
```

Rolling back a Unit of Work

Explicitly

```
session beginUnitOfWork.  
Blah  
Blah  
session rollbackUnitOfWork.
```

Implicitly

```
session inUnitOfWorkDo:  
[ blah.  
  Blah  
  Some exception is raised  
  blah]
```

Types of Queries

- SQL
- Query objects
- Block query language

SQL

The goal of O/R layers is to avoid using SQL directly

```
login := Login new database: PostgreSQLPlatform new;  
  username: 'usernameHere';  
  password: 'passwordHere';  
  connectionString: '127.0.0.1_test'.
```

```
accessor := DatabaseAccessor forLogin: login.
```

```
accessor login.
```

```
result := accessor basicExecuteSQLString: 'select 1 + 1'.
```

```
result next first “return 2”
```

```
accessor logout.
```

Query Objects

byLastName := Query readManyOf: Person.

byLastName

 orderBy: #lastName;

 orderBy: #firstName.

result := session execute: byLastName.

Block Query Language

Read methods in Session

readOneOf: readOneOf:where:
readManyOf: readManyOf:where:
readManyOf:limit: readManyOf:where:limit

readManyOf always returns a collection, which may be empty

```
session readManyOf: Person limit: 2
session
  readManyOf: Person
  where: [:each | each firstName = 'Sam']
  limit: 3
```

where Block Restrictions

Messages that can be sent to the block argument (each)

- Names of instance variables
- =
- <>
- ~=
- notNIL
- isNIL

sam := session readOneOf: Person where: [:each | each firstName = 'Sam'].
result := session readManyOf: Person where: [:each | each <> sam] limit: 4.

Messages that can be sent to instance variables

- Binary comparison operators: <, >, =, ~=, >=, <=, <>
- like:
- isNIL, notNIL
- Methods in the api protocol of Glorp.ObjectExpression

More operations exist, but require more detailed situations

```
result := session readManyOf: Person where: [:each | each id >= 1].
```

```
result := session readManyOf: Person where: [:each | each firstName like: 'S%'].
```

```
result :=
  session readManyOf: Person where: [:each | (each firstName like: 'S%')
not ].
```

```
result := session readManyOf: Person where: [:each | each firstName
notNIL].
```

Boolean Combinations

```
result := session
  readManyOf: Person
  where:
    [:each |
      (each firstName like: 'S%') & (each lastName = 'Olson')].
```

One to Many

Example

Person with multiple email addresses

Classes EmailAddress

```
Smalltalk defineClass: #EmailAddress
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'userName host id '
  classInstanceVariableNames: "
  imports: "
  category: 'GlorpExperiments'
```

EmailAddress class method

```
name: aNameString host: aHostString
^(super new)
  userName: aNameString;
  host: aHostString
```

Plus standard accessor methods

Person

```
Smalltalk defineClass: #Person
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'firstName lastName id emailAddresses '
  classInstanceVariableNames: "
  imports: "
  category: 'GlorpExperiments'
```

Person class method

```
first: firstNameString last: lastNameString
  ^self new setFirst: firstNameString last: lastNameString
```

Person instance methods

```
setFirst: firstNameString last: lastNameString
  firstName := firstNameString.
  lastName := lastNameString.
  emailAddresses := OrderedCollection new.
```

```
addEmailAddress: anEmailAddress
  emailAddresses add: anEmailAddress
```

```
emailAddresses
  ^emailAddresses
```

Database Tables

EMAIL_ADDRESSES Description

Column	Description
id	Primary Key for table
user_name	User name of the email address
host	The email host
person_id	Foreign key to person table

PEOPLE Description

Column	Description
id	Primary Key for table
first_name	First name of person
last_name	Last name of person

Example

PEOPLE

Id	First_name	Last_name
1	Roger	Whitney
2	Leland	Beck

EMAIL_ADDRESSES

Id	User_name	Host	Person_id
1	whitney	cs.sdsu.edu	1
2	whitney	rohan.sdsu.edu	1
3	whitney	math.sdsu.edu	1
4	beck	cs.sdsu.edu	2

Mapping

```
Smalltalk defineClass: #GlorpTutorialDescriptor
  superclass: #{Glorp.DescriptorSystem}
  indexedType: #none
  private: false
  instanceVariableNames: "
  classInstanceVariableNames: "
  imports: ' Glorp.* '
  category: 'GlorpExperiments'
```

Instance Methods

```
allTableNames
```

```
  ^#('PEOPLE' 'EMAIL_ADDRESSES')
```

```
constructAllClasses
```

```
  ^(super constructAllClasses)
```

```
    add: Person;
```

```
    add: EmailAddress;
```

```
    yourself
```

```
classModelForEmailAddress: aClassModel
```

```
  aClassModel newAttributeNamed: #id.
```

```
  aClassModel newAttributeNamed: #userName.
```

```
  aClassModel newAttributeNamed: #host.
```

```
classModelForPerson: aClassModel
```

```
  aClassModel newAttributeNamed: #id.
```

```
  aClassModel newAttributeNamed: #firstName.
```

```
  aClassModel newAttributeNamed: #lastName.
```

```
  aClassModel
```

```
    newAttributeNamed: #emailAddresses
```

```
    collectionOf: EmailAddress.
```

Instance Methods Continued

```
descriptorForEmailAddress: aDescriptor
| table |
table := self tableNamed: 'EMAIL_ADDRESSES'.
aDescriptor table: table.
(aDescriptor newMapping: DirectMapping)
  from: #userName to: (table fieldNamed: 'user_name').
(aDescriptor newMapping: DirectMapping)
  from: #host to: (table fieldNamed: 'host').
(aDescriptor newMapping: DirectMapping)
  from: #id to: (table fieldNamed: 'id').
```

```
descriptorForPerson: aDescriptor
| personTable |
personTable := self tableNamed: 'PEOPLE'.
aDescriptor table: personTable.
(aDescriptor newMapping: DirectMapping)
  from: #firstName
  to: (personTable fieldNamed: 'first_name').
(aDescriptor newMapping: DirectMapping)
  from: #lastName
  to: (personTable fieldNamed: 'last_name').
(aDescriptor newMapping: DirectMapping)
  from: #id
  to: (personTable fieldNamed: 'id').
(aDescriptor newMapping: OneToManyMapping)
  attributeName: #emailAddresses;
  orderBy: #userName
```

Instance Methods Continued

tableForEMAIL_ADDRESSES: aTable

| personId |

aTable createFieldNamed: 'user_name' type: (platform varChar: 50).

(aTable createFieldNamed: 'host' type: (platform varChar: 50)).

(aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.

personId := aTable createFieldNamed: 'person_id' type: platform int4.

aTable

addForeignKeyFrom: personId

to: ((self tableNamed: 'PEOPLE') fieldNamed: 'id').

tableForPEOPLE: aTable

(aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.

(aTable createFieldNamed: 'first_name' type: (platform varChar: 50)).

(aTable createFieldNamed: 'last_name' type: (platform varChar: 50)).

Sample Usage Saving a Person

```
person := Person first: 'Sam' last: 'Whitney'.
```

```
email := EmailAddress new.
```

```
email
```

```
  host: 'cs.sdsu.edu';
```

```
  userName: 'whitney'.
```

```
person addEmailAddress: email.
```

```
email := EmailAddress new.
```

```
email
```

```
  host: 'rohan.sdsu.edu';
```

```
  userName: 'whitney'.
```

```
person addEmailAddress: email.
```

```
session inUnitOfWorkDo: [session register: person].
```

The entire object graph is saved

Simple Read

```
foundPerson :=  
  session  
    readOneOf: Person where: [:each | each firstName = 'Sam'].
```

SQL executed

```
SELECT t1.id, t1.first_name, t1.last_name  
FROM PEOPLE t1  
WHERE (t1.first_name = 'Sam') LIMIT 1
```

Note the emailAddresses are not fetched. A proxy is used.
When you try to access an email address they are then fetched

```
foundPerson emailAddresses first
```

SQL executed

```
SELECT t1.user_name, t1.host, t1.id  
FROM EMAIL_ADDRESSES t1  
WHERE (t1.person_id = 1) ORDER BY t1.user_name
```

Complex Read

```
foundPerson :=  
  session  
    readOneOf: Person  
    where:  
      [:person |  
        person emailAddresses  
          anySatisfy: [:address | address host ='cs.sdsu.edu']]
```

Generates & runs the SQL

```
SELECT t1.id, t1.first_name, t1.last_name  
FROM PEOPLE t1  
WHERE EXISTS (SELECT t2.id  
FROM EMAIL_ADDRESSES t2  
WHERE ((t2.host = 'cs.sdsu.edu') AND (t1.id = t2.person_id)))  
LIMIT 1
```

Many to Many

Example

- Books on order
- A person may have many books on order
- A book may be ordered by many people

Tables BOOKS

id	title
1	Code Complete
2	Palm OS
3	Cat in the Hat

PEOPLE

Id	First_name	Last_name
1	Sam	Hinton
2	Martin	Fowler

BOOKS_ON_ORDER

customer_id	book_id
1	3
1	1
2	3

Classes

Book

```
Smalltalk defineClass: #Book
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'id title '
  classInstanceVariableNames: "
  imports: "
  category: 'GlorpExperiments'
```

Standard accessor methods not shown

Person

```
Smalltalk defineClass: #Person
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'firstName lastName id booksOnOrder '
  classInstanceVariableNames: "
  imports: "
  category: 'GlorpExperiments'
```

Class Method

```
first: firstNameString last: lastNameString
  ^self new setFirst: firstNameString last: lastNameString
```

Instance Methods

```
setFirst: firstNameString last: lastNameString
  firstName := firstNameString.
  lastName := lastNameString.
  booksOnOrder := OrderedCollection new.
```

```
addBook: aBook
  booksOnOrder add: aBook
```

```
books
  ^booksOnOrder
```

Mapping

```
Smalltalk defineClass: #GlorpTutorialDescriptor
  superclass: #{Glorp.DescriptorSystem}
  indexedType: #none
  private: false
  instanceVariableNames: "
  classInstanceVariableNames: "
  imports: ' Glorp.* '
  category: 'GlorpExperiments'
```

Instance Methods

```
allTableNames
  ^#( 'PEOPLE' 'BOOKS_ON_ORDER' 'BOOKS')

constructAllClasses
  ^(super constructAllClasses)
  add: Person;
  add: Book;
  yourself

classModelForBook: aClassModel
  aClassModel newAttributeNamed: #id.
  aClassModel newAttributeNamed: #title.

classModelForPerson: aClassModel
  aClassModel newAttributeNamed: #id.
  aClassModel newAttributeNamed: #firstName.
  aClassModel newAttributeNamed: #lastName.
```

Mapping Continued

```
descriptorForBook: aDescriptor
| table |
table := self tableNamed: 'BOOKS'.
aDescriptor table: table.
(aDescriptor newMapping: DirectMapping)
  from: #title
  to: (table fieldNamed: 'title').
(aDescriptor newMapping: DirectMapping)
  from: #id
  to: (table fieldNamed: 'id').

descriptorForPerson: aDescriptor
| personTable |
personTable := self tableNamed: 'PEOPLE'.
aDescriptor table: personTable.
(aDescriptor newMapping: DirectMapping)
  from: #firstName
  to: (personTable fieldNamed: 'first_name').
(aDescriptor newMapping: DirectMapping)
  from: #lastName
  to: (personTable fieldNamed: 'last_name').
(aDescriptor newMapping: DirectMapping)
  from: #id
  to: (personTable fieldNamed: 'id').
(aDescriptor newMapping: ManyToManyMapping)
  attributeName: #booksOnOrder;
  referenceClass: Book
```

Mapping Continued

tableForBOOKS: aTable

(aTable createFieldNamed: 'title' type: (platform varChar: 100)).

(aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey

tableForPEOPLE: aTable

(aTable createFieldNamed: 'id' type: platform sequence) bePrimaryKey.

(aTable createFieldNamed: 'first_name' type: (platform varChar: 50)).

(aTable createFieldNamed: 'last_name' type: (platform varChar: 50)).

tableForBOOKS_ON_ORDER: aTable

| custKey bookKey |

custKey := aTable createFieldNamed: 'customer_id' type: (platform int4).

aTable addForeignKeyFrom: custKey

to: ((self tableNamed: 'PEOPLE') fieldNamed: 'id').

bookKey := aTable createFieldNamed: 'BOOK_ID' type: (platform int4).

aTable addForeignKeyFrom: bookKey

to: ((self tableNamed: 'BOOKS') fieldNamed: 'id').

Some Usage

Add some Books

```
session inUnitOfWorkDo:  
  [books := #( 'Code Complete' 'Palm OS' 'Cat in the Hat' )  
    collect: [:each | Book title: each ].  
  session registerAll: books.  
  session register: (Person first: 'Sam' last: 'Hinton').  
  session register: (Person first: 'Martin' last: 'Fowler')].
```

Read some books

```
cat := session readOneOf: Book where: [:each | each title = 'Cat in the Hat'].  
code := session readOneOf: Book where: [:each | each title = 'Code  
Complete'].
```

Order some books

```
session beginUnitOfWork.  
sam := session readOneOf: Person where: [:each | each firstName = 'Sam'].  
sam  
  addBook: cat;  
  addBook: code.  
martin := session readOneOf: Person where: [:each | each firstName =  
'Martin'].  
martin addBook: cat.  
session commitUnitOfWork.
```

Reading

Books ordered by Sam

```
sam := session readOneOf: Person where: [:each | each firstName = 'Sam'].
sam books
```

People who ordered “Cat in the hat”

```
waitingForCat := session
  readManyOf: Person
  where:
    [:each |
     each booksOnOrder anySatisfy: [:book | book title like: 'Cat%']].
```