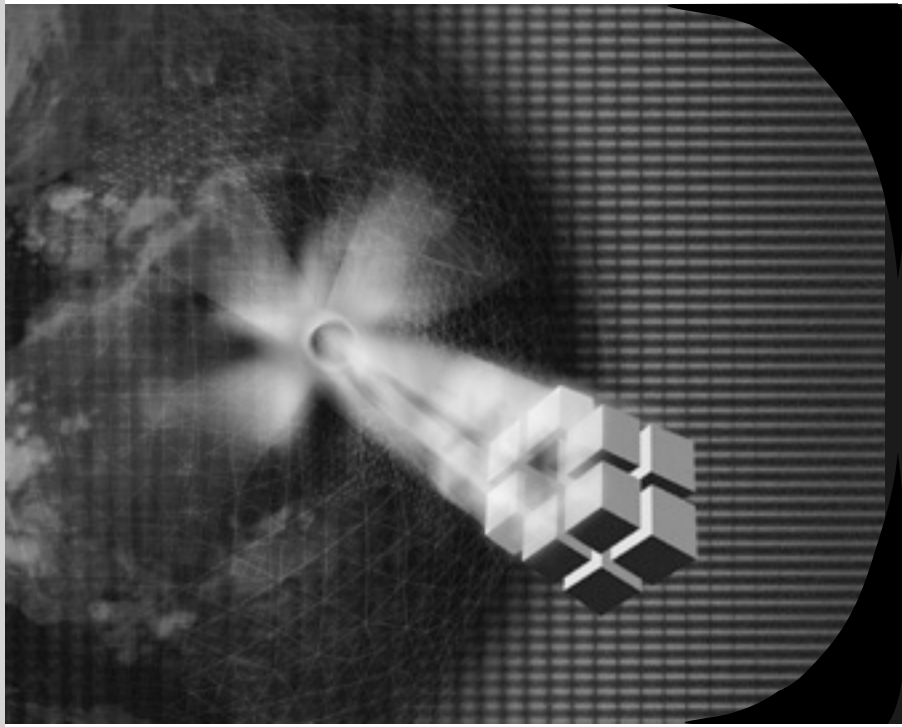
CINCOM Smalltalk™



Using GLORP

Alan Knight (knight@acm.org)

Cincom Systems of Canada

CINCOM

About Me

- With Cincom Systems since 2000
- Previously with The Object People
- Chief Architect for TOPLink family of O/R products
- On the EJB 2.0 and JDO expert groups
- Lead on the GLORP open source O/R mapping project
- Lead on next-generation database mapping frameworks for VisualWorks

About this Tutorial

- 3.5 hours, half hour break in the middle
- Two hands-on sessions + possible demo
- Using, so more focused on how than why
- Medium-Basic
  - Assumes little knowledge to start, but covers some fairly advanced topics
- Flexible

Outline

- Introduction
- Basic Concepts and Terms
- Hands-on 1 (examining a simple system)
- Relationships, Queries, Modifications
- Hands-on 2 (extending the simple system)
- More Stuff

What is GLORP?

- Open Source (LGPL(S)) mapping library
- “Generic Lightweight Object-Relational Persistence”
- Portable across dialects

Why Do We Need Mapping?

- Most programming is OO
- Most databases are relational
- “Impedance mismatch”
- Ignoring either world can cause big problems

Why is this hard?

- Object identity vs primary keys
- Pointers vs. foreign keys
- Networks of objects vs. rows
- Queries vs. traversing relationships
- Encapsulation vs. program independence
- The role of the application
- nil not NULL

Approaches

- Many different approaches to the problem
  - Embedded SQL – SQLJ
  - Relational-Centric – PowerBuilder, ADO
  - OODB and OODB-like – Gemstone, ODMG, JDO
  - Mapping – Lens, EJB, TOPLink

Variations on Mapping

- Metadata or code generation
- How to associate objects with transactions
- Expressing queries
  - SQL, OO query language, objects as queries, special syntax
- Explicit or automatic writes
- How are objects marked dirty?
- When do objects get removed from cache?
- Different framework architectures
  - Brokers (single or multiple)
  - Subclassing from PersistentObject

Glorp Terminology

- ClassDescription
  - Instance variables, cardinality, types
- DatabaseTable
  - Fields, Types, Primary Keys, Sequences, Foreign Key Constraints
- Descriptor
  - Describes relationship between class and tables
- Mapping
  - Information for one instance variable
- DescriptorSystem
  - Where we define the above

…Terminology

- Session
  - The broker - main interface to GLORP, “singleton”
- Unit of Work
  - How we write
  - Object Level Transaction
- Registration
  - How we tell GLORP an object might change
- Query
  - How we read
- Cache
- Proxies
- Joins
  - Describes relationship between tables

Unit of Work

- **Objects are registered within a unit of work**
  - **New/modified registered objects detected**

```
| thing |
session beginUnitOfWork.
Thing := SomeClass new.
session register: thing.
thing foos first name: 'newName'
thing addFoo: Foo new.
session commitUnitOfWork.
```

Hands-on Example

- Sourceforge type application
  - Users
  - Projects
  - Tasks
- Pre-built very simple application

Hands-on 1

CINCOM Smalltalk™

Hands-On Review

- Saw a simple, but functional example
  - Connect/disconnect
  - Domain objects and their descriptor system
  - Create tables based on metadata
  - Insert new objects
- Missing some obvious pieces
  - No reading
  - No relationships between objects
  - Only inserts, no update

Relationships

- **Recall from the hands-on descriptor system**

```
(aDescriptor newMapping: DirectMapping)
        from: #id
        to: (table fieldNamed: 'ID').
```

- **This defines a relationship to a simple type**
- **Different types of mappings define different kinds of relationships, and have different parameters**

Object Types

- "Simple" vs. "Complex" objects
- Not well-defined
- Simple
  - No descriptor
  - Represented by a single database column
  - Normally immutable
- Complex
  - Has a descriptor
  - Corresponds to one or more database rows
  - Mutable

Basic Mapping Types

- DirectMapping
  - Simple types
- OneToOneMapping
  - To a single complex object
- ToManyMapping
  - To a collection of complex objects

Adding Relationships

- Consider adding relationships to our model
- Project
  - Administrator
  - Members
- Need to define class model changes
  - Attribute name
  - Attribute type
  - Collection?
  - Collection type

Class Model changes


```
classModelForProject: aClassModel
  aClassModel newAttributeNamed: #id.
  aClassModel
          newAttributeNamed: #administrator
          type: User.
  aClassModel
          newAttributeNamed: #members
          collectionOf: User.
```

Table Relationships

- We must also define the database level relationships.
- Tables define
  - Field name
  - Field type
  - Foreign key constraints
- Note that field types are “platform” (i.e. database) specific
- For the “administrator” relationship the TUT_PROJECT table has a foreign key to the TUT_USER table

```
…
adminId := aTable createFieldNamed: 'ADMIN_ID' type: platform
   int4.
userId := (self tableNamed: 'TUT_USER') fieldNamed: 'ID'.
aTable addForeignKeyFrom: adminId to: userId.
…
```

… Table Relationships

- **The project->members relationship uses a link table.**

…

```
tableForPROJECT_MEMBERS_LINK: aTable


    | projectId userId |
    projectId := (aTable createFieldNamed: 'PROJECT_ID'
    type: platform int4).
    aTable addForeignKeyFrom: projectId to: ((self
    tableNamed: 'TUT_USER') fieldNamed: 'ID').
    userId := aTable createFieldNamed: 'USER_ID' type:
    platform int4.
    aTable addForeignKeyFrom: userId to: ((self
    tableNamed: 'TUT_USER') fieldNamed: 'ID').
```

…

Aside: Creating Tables

- Creating tables in code is quite repetitive and tedious
- Would be nice to be able to read schema from the database
- An interestingly recursive problem
  - Schema defined as tables in DB
  - Glorp metadata defined as objects

Defining Relationship Mappings

- **Mappings define**
  - **Attribute name**
  - **Join**
  - **…other optional properties**
- **Other required properties (e.g. type) come from from the classDescription or databaseTable**

```
Join
    from: (myTable fieldNamed: 'ADMIN_ID')
    to: (userTable fieldNamed: 'ID')).
```

Joins

- Three different things we need to know about the relationship between objects
  - How to read
  - How to write
  - How to join across it in a query
- We can get all 3 from the Join
- Note that "direction" of the foreign key doesn't matter
  - My foreign key field = other primary key field
  - My primary key field = other foreign key field
- Joins can have composite keys

Implied Joins

- Often, the join can be computed from the foreign key relationship between the tables
- We know
  - Source class (from our descriptor)
  - Source table (from our descriptor)
  - Target class (from the classDescription)
  - Table(s) for target class from its descriptor
  - Foreign key relationship between source and target tables (from databaseTables)

Link Tables

- **Some relationships, particularly many-to-many, may use a link table**
- **Specified as "useLinkTable"**

```
(aDescriptor newMapping: ToManyMapping)
    attributeName: #members;
    useLinkTable;
    join: (Join
          from: (myTable getField: 'ID')
          to: (linkTable getField: 'PROJ_ID')
```

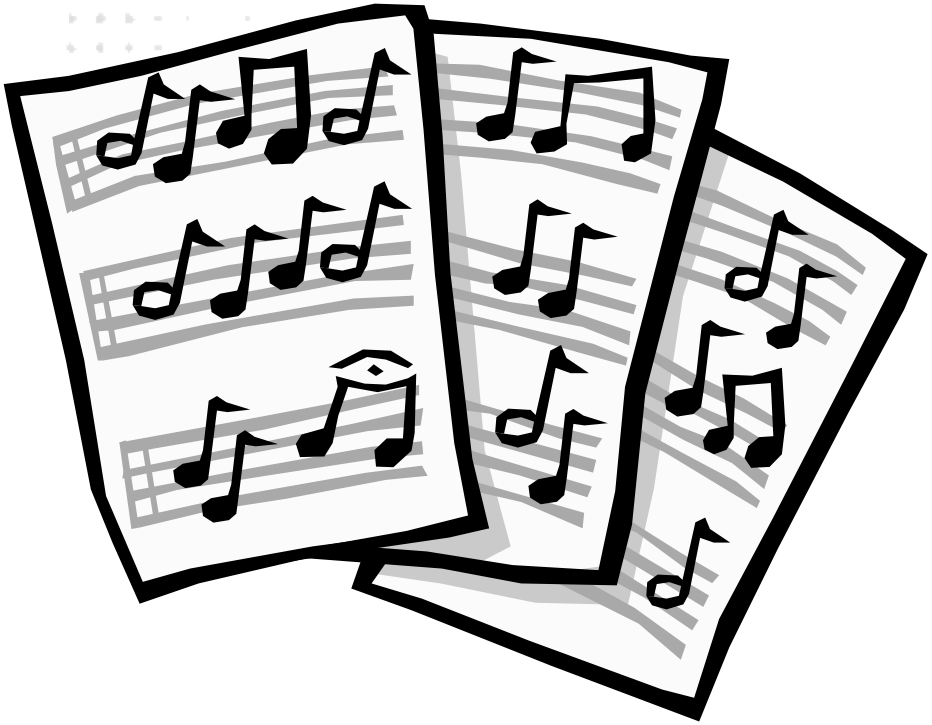
Writing Relationships

- **Related objects are automatically written**
- **Must be reachable from a registered object**
- **Note: the objects don't contain foreign keys**

```
project := Project new.
project name: self projectNameHolder value.
user := User new.
user name: self userNameHolder value.
project admin: user.
session transact: [session register: project].
```

Reading

CINCOM Smalltalk™



Queries

- **All reads go through the session**

```
allProjects := session readManyOf: Project.

admins := allProjects
   collect: [:each | each admin].
```

Where Clause

- **The where clause is specified as a Smalltalk block, in terms of the object attributes and relationships**

```
aMonthAgo := Date today subtractDays: 30.
newUsers := session
  readManyOf: User
  where: [:each | each joined > aMonthAgo].
```

- **SQL**

```
SELECT … FROM TUT_USER t1 WHERE t1.JOINED > ?
```

Reading with Relationships


```
me := session
   readOneOf: User
   where: [:each | each name = 'Alan Knight'].
myProjects := session
    readManyOf: Project
    where: [:each | each admin id = me id].
```

- **Note the syntax for reading a single object**
- **SQL**

```
SELECT t1.… FROM TUT_PROJ t1, TUT_USER t2
   WHERE t1.ADMIN_ID = t2.ID AND t2.id = ?
```

Comparing Objects

- **Comparing ids is unpleasant.**
- **Prefer**

```
myProjects := session
    readManyOf: Project
    where: [:each | each admin = me].
```

- **Resolves down to the same thing at the SQL level**

Querying with Collections

- **We can query across relationships that are collections**

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

- **The only operations allowed are anySatisfy: and noneSatisfy:**
  - **variations anySatisfyJoin:, anySatisfySubselect:**
- **SQL**

```
select DISTINCT…
select … WHERE EXISTS ….
```

Query Objects

- Many different options for querying
  - order by
  - extra things to retrieve
  - expected number of results
  - collection type of results
  - should we refresh if the object is already in memory
- Also want to reuse queries with different parameters
- Difficult with methods on session
- So, use query objects.
- Session methods are shortcuts

```
query := Query readManyOf: User.
session execute: query.
```

Ordering

- To read results in a particular order
- Ordering specified by block, similar to where clause block
- Symbol also allowable
- Multiple orderBy: allowed, orders by A, then B, etc.

```
userQuery := Query readManyOf: User.
userQuery orderBy: [:each |
  each name descending].
userQuery orderBy: #joined.
session execute: userQuery
```

Proxies

- Relationships from read objects
  - If we read a project, we must read its admin
  - If we read a user we must read their projects
  - Rapidly leads to reading everything…
- Solution… Proxies
- Replace relationships with a stub
  - contains query, session, and parameters
  - doesNotUnderstand: handler
  - triggers query execution

… Proxies

- **Consider the earlier code fragment**

```
allProjects := session readManyOf: Project.
admins := allProjects
   collect: [:each | each admin].
```

- **Results in the SQL**

```
SELECT … FROM TUT_PROJ
SELECT … FROM TUT_USER WHERE ID=1
SELECT … FROM TUT_USER WHERE ID=2
SELECT … FROM TUT_USER WHERE ID=3
…
```

Cache

- Important to maintain object identity
  - read user u
  - p := u projects
  - p members includes: u.
- Keep a cache of objects
- About correctness, not performance!
- Also used to determine insert/update
- Different policies for when to remove things from cache

Unit of Work

CINCOM Smalltalk™

Unit of Work

- Recall the basic unit of work
  - session transact: [… register: anObject].
- Now we'll look at
  - Modifying objects
  - Rollback
  - Write Order

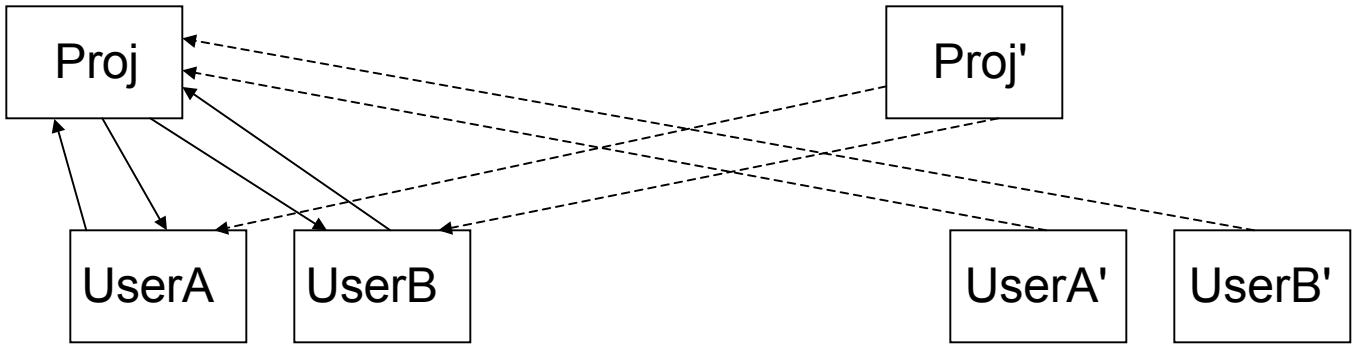
Modifying Objects

- **Modifications to registered objects are automatically detected**
- **Objects must be registered *before* changing**

```
allProjects := session readManyOf: Project.
newProj := allProjects detect: [:each |
  each name = 'Unnamed project'].
session beginUnitOfWork
session register: newProj.
me := User new name: 'Me'.
newProj admin: me.
newProj addMember: me.
newProj name: 'MegaThing!'.
session commitUnitOfWork.
```

Change Detection

- When you register an object, Glorp makes a shallow copy of it, and its transitive closure
- On commit, we generate rows and compare
- Only rows with differences are written



Rollback

- On unit of work rollback, we revert the state of the original objects to that of the copies
- Yes, this works
- Collections
  - Must register their internals
  - Have to reverse become: operations for size changes

Implications

- No write barrier
  - copy-on-register
  - objects must be registered before changes are made
- No back-references needed
  - e.g. Project members don't need to know their project(s)
- Changes applied to originals
  - One unit of work at a time (per session)
- Note: Objects read while a unit of work is active are automatically registered

Hands-on Example

- Same model
- Querying the database
- Adding relationships
- Writing related objects
- Reading based on relationships
- Proxies
- Modifying objects
- Rollback

Hands-on 2

CINCOM Smalltalk™

Hands-On Review

- Read objects, including using where clauses
- Added a to-many relationship
- Wrote related objects
- Read back using a join to related objects
- Read in and modified objects
- Read in objects, rolled back changes

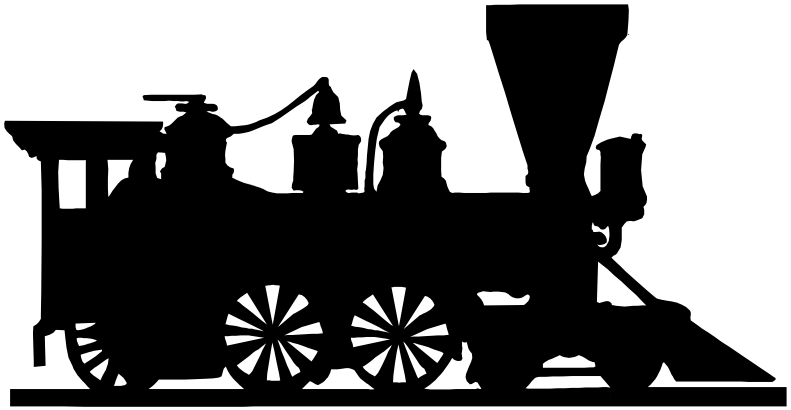
Complications

- We've covered the most basic operations
- Other Considerations
  - Performance
  - Performance
  - Performance
  - Complex Mappings
  - Complex Queries
  - Locking
  - Performance
  - Database Functions
  - Internal Mechanisms
  - Performance

Write Optimizations

CINCOM Smalltalk™

- Prepared Statements
- Sequence Generation
- Multiple Inserts



Prepared Statements

- Dynamic vs. Static SQL
  - Static is faster, but less flexible
  - Overhead of re-preparing statements
- Harder to use purely static from a mapping layer
- Cache prepared statements and re-use
  - Limited size cache
  - Can be turned on/off
- Parameterized statements
  - “Bind” the actual values at execution time
- Benefits vary a lot by database
  - Particularly important for Oracle

Sequencing: Generated Keys

- Primary keys can be generated or “natural”
- Two primary mechanisms for generating
  - Sequences
  - Identity Columns
- Syntax varies by database

Sequencing: DB Sequences

- The database can give us the “next” value
- Oracle, others
- Minimizes transaction conflicts
- Can have “holes” in the sequence
- Often increment can vary
- Simple usage

```
INSERT… VALUES (NEXTVAL(X)…
```

- But we can also pre-read many values

Sequencing: Identity Columns

- We're not allowed to set a value
- Database will automatically generate after insert
- Sybase, SQL Server
- Means we need to read back if we want to know the primary key given to the object

  Select @@IDENTITY
- Cannot pre-read
- Cannot write multiple objects at a time
- Seemed like a good idea at the time

Sequencing: Glorp Usage

- For identity columns we can't optimize
- For sequences we can read everything in advance
- Strategies vary by database: DatabaseSequence
- E.g. Oracle
  - select seq.nextval from a table with lots of elements where rownum <= number needed.
  - By default use the table being inserted into
  - Fall back to SYS.ALL_OBJECTS

Multiple Inserts

- Often, round trips to the DB are the bottleneck
- Minimize number of statements by grouping
- Database-specific techniques
  - Oracle Array-Binding
  - Multiple statements grouped together

Grouping: Array Binding

- Single statement
- Bind arrays of arguments, not one
- Works best with inserts
  - All values specified

Aside: The Write Process

- Glorp writes in two stages
  - 1) Build a RowMap
  - 2) Write the rows
- Benefits
  - A row can be easily built by more than one object
  - We can group like rows together, so we can use features like array binding
  - We can determine the required write order (we'll come back to that)

Grouping: Multiple Statements

- We can append statements together
- Supported by most databases
- Harder to use with binding
- Harder to detect the cause of errors in specific statements
  - e.g. optimistic locking

```
INSERT INTO … VALUES(1,2,3);INSERT INTO
… VALUES(4,5,6);INSERT INTO….
```

Write Optimizations

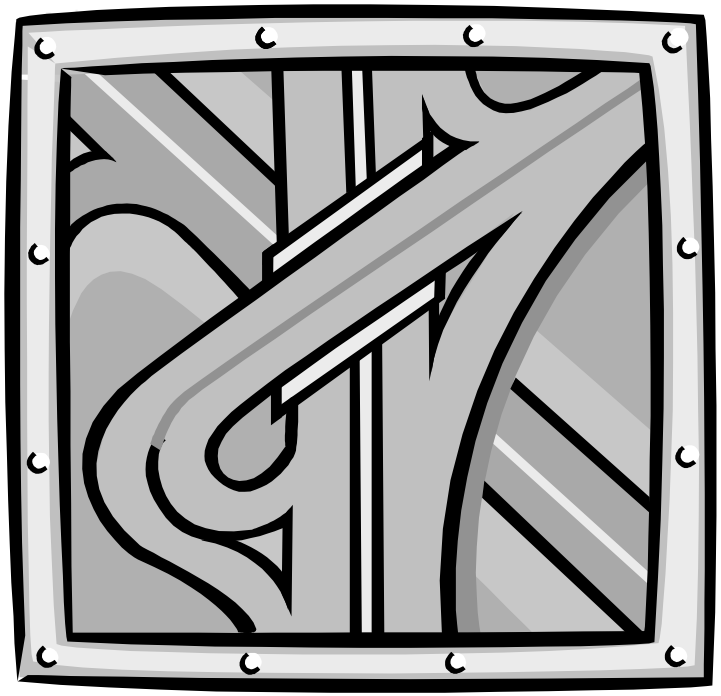
- Get all sequence numbers at the beginning of a transaction (except for identity column DBs)
- Prepared statements are cached, and arguments bound
- For inserts we use Oracle array binding, or grouping of statements

Aside: Write Order

- Databases often have integrity constraints
- Often checked at statement execute time rather than commit time
- So, cannot write rows with foreign keys until the referred-to rows have been written
- Also, some databasea are page-locking
  - Reduces deadlocks if tables are written in a consistent order

More Mappings

CINCOM Smalltalk™



Mapping Types

- Recall the basic mapping types
  - Direct
  - OneToOne
  - ToMany
- Relationship mappings may or may not use a link table

Descriptor Options

- Multiple Tables
  - One table is primary
  - Joins specified for additional tables
- Caching policy
- Inheritance
  - Many options
  - 3 strategies
- Imaginary Tables

Mapping Options

- readOnly
  - cut transitive closure
  - map foreign keys
  - Attributes mapped to functions
- writeOnly
  - log/audit information
- pseudoVariable
  - refer to unmapped columns
  - [:each | each ownerId ~= nil]
- debugRead/debugWrite
- type

Embedded Values

- One to one mapping into the same table
- EmbeddedValueOneToOneMapping
- e.g. Currency
  - no primary key
  - doesn't exist independently
  - can have field translations to allow embedding one class in multiple places
  - nestable

Dictionaries

- In memory just a specialization of collections
- Database can be much more complicated
  - Is the key a simple type?
  - Is the key part of the value?
  - If not, how are they related (e.g. part of link table?)
- Simple cases supported
  - key in link table, value as object
  - probably others, but no tests
- Queries can also return dictionaries

Special-Purpose Mappings

- ConstantMapping
  - Read or write a constant value
  - More useful than you might think
  - Special case for the session as a constant
- ConditionalMapping
  - Do something different depending on a field or attribute value
  - Constant mapping also useful as one case of a condition
- Ad Hoc Mapping
  - Plug in your own blocks. Do anything.

Relationship Mapping Options

- proxy
- orderBy
- shouldWriteTheOrderField
- collection type
- separate link table and target table joins
- row map key customization (don’t ask)
- hints for the link table
- filtered reads (optimization)

Read Optimizations

CINCOM Smalltalk™

Read Optimizations Overview

- Reads can be very time-consuming
  - Proxies fault one by one
  - Queries can be expensive
- Optimizations available
  - Complex where conditions
  - Reading subset of data/non-object data (retrieve:)
  - Reading additional data (alsoFetch:)
  - Database Functions
  - Cursors
  - union:, except:
  - write your own SQL

Optimizing with where clauses

- What's actually faster depends a *lot* deal on the database
- Optimizers don't
- For high performance, often have to start with an idea of the SQL you want and reverse engineer
- Joins nest indefinitely
  - where: [:each | each owner parent thing value > 2]
- anySatisfy:
  - each owners anySatisfy: [:eachOwner |
    - eachOwner parents anySatisfy: [:eachParent ...]
- Outer joins

Outer Joins

- In the database, joins require data on both sides
- Consider ordering projects by admin name.
  - Projects with no admin disappear from the list
- An outer join returns everything on the "left" side with nulls for missing "right" side entries
- Syntax varies
  - =+
  - (*)
  - LEFT OUTER JOIN ... ON

Reading non-Object Data

- Reading pure data, ordering

  query := Query readManyOf: Project.

- Aggregate functions

  query orderBy: [:each | each name].

  query retrieve: [:each | each name distinct].

  query retrieve: [:each | each dateJoined max].

- Retrieving pieces of objects

  query retrieve: [:each | each id].

  query retrieve: [:each | each name].

  query retrieve: [:each | each admin] (changing contexts)

- Note: All internal queries generated by user-accessible mechanisms.

alsoFetch:

- **Like retrieve:, but brings back the data in the background**

```
query readManyOf: Project.
query alsoFetch: [:each | each admin].
query alsoFetch: [:each | each members].
```

Filtered Reads

- Two main uses
- In general, get our results as a subset of a larger group
- On a mapping, slightly more complicated
  - Build our proxy based on our "parent" query
  - When it fires, read all related objects
  - Everything retrieved by the parent query gets its results by filtering ours

Filtered Read Example

- Use filtering on the admin->members relationship
  - read all Projects where the admin joined within 1 month, 100 total
  - each project has a proxy for members
  - when we touch members for the first project, all members for all those projects will be read
  - proxies filter their results
- Possibly the most generally useful

Functions

- A small set of database functions is available
  - Others are easy to add
  - Useful for things other than optimization (e.g. asc/desc)
- Used by name in an expression block
  - [:each | each name distinct count]
- Sample
  - DISTINCT
  - COUNT / COUNT(*)
  - MIN/MAX
  - ||
  - isNIL/notNIL

Mapping to Functions

- Mappings can use functions in place of fields
- Mappings start getting complex
- e.g. versions
  - StorePackage
  - StoreVersionlessPackage
- Versionless package maps to [:each | each name distinct].

Cursors

- Warning: Not useful on PostgreSQL
- Queries can return a stream of results rather than a collection
- Database won't compute results until they're asked for
- Can be very useful when only a small subset of a potentially large result is needed
- query collectionType: GlorpCursoredStream
- Also note GlorpVirtualCollection
  - a collection that wraps a stream internall
  - but size requires a separate query

UnionAll:/Except:

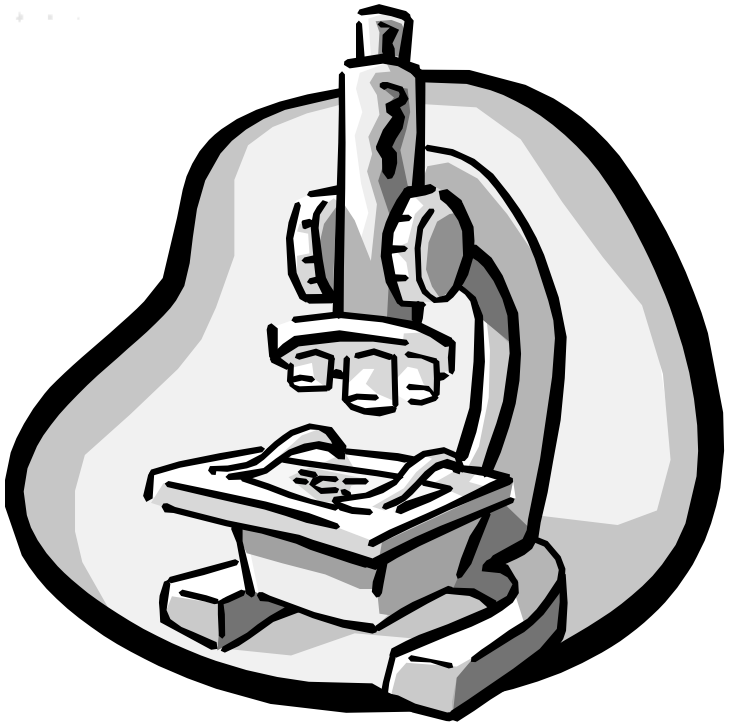
- Can combine multiple queries
- UnionAll: returns results of all subqueries combined
- Except: excludes the results of the argument subquery
- Other variations possible
- AND:/OR: also work, but much simpler to implement

Write Your Own SQL

- Limited support for plugging in your own SQL
- Queries generate Command objects
- SQLStringSelectCommand
- session accessor executeCommand:
- query command: aCommand
- Your responsibility that the result set matches what Glorp expects

Other Topics

CINCOM Smalltalk™



Query Blocks

- Used for where clause, ordering, etc.
- A subset of allowable Smalltalk syntax
- Used to create a GlorpExpression (parse tree)
- Not parsed
  - Pass in a doesNotUnderstand: proxy
  - evaluate the block
  - proxy accumulates message sends, returns a new proxy
  - at the end, build an expression from the tree

Query Block Limits

- ifTrue:/ifFalse:
- complex execution paths in general
  - [:each | each members do: [:eachUser | ...]
  - you can actually loop, you just have to be careful which objects are real and which ones aren't
- inlined messages

Expressions

- Expressions can also be built manually
- Instances of GlorpExpression
- Operations
  - get: #attributeSymbol
  - getField:

```
(BaseExpression new get: #admin) get: #id.
```

Locking

- Important in a multi-user application
- Pessimistic
  - Lock rows in database
  - Most appropriate for batch
  - Not always clear how to do it (cf Oracle)
- Optimistic
  - Never commit inconsistent data
  - Most appropriate for interactive
- Glorp supports only optimistic

Optimistic Locking

- Can specify a lock field on the table
- When we write, check that the value matches what we think it should be
- UPDATE... WHERE LOCK=2
- Check the row count coming back. If not equal to the number we think it should be, we failed
- Version number generation handled by field
  - Similar mechanism as sequence generation
  - Timestamps also supported
  - Automatic based on underlying type

Summarizing

CINCOM Smalltalk™

- What We've Seen
- Gaps
- Neat Implementation Tricks
- Gotchas
- Future Plans
- Wrap-up

What We've Seen

- Session
- Metadata: Descriptors, Mappings, ClassModels, DatabaseTables
- DescriptorSystem
- Unit of Work
- Registration
- Queries, query blocks
- Relationships, Joins
- Many optimization options

Gaps

- Stored procedures
- Meaningful exceptions
- Thread safety
- Connection pooling
- Nested units of work
- Performance tuning
- Tools
- Documentation
- Validation
- Error Messages
- Reading schema from database

Particularly Cool Tricks

- Rollback
- RowMaps
- Blocks -> Expressions
- Join Handling

Gotchas

- isNil/notNil inlined in some dialects
- and: inlined, use & or AND:
- Null is not nil

Change Hats: VisualWorks

- Next-generation database frameworks, inputs
  - VisualWorks Object Lens
    - Strong in many respects, but very dated
    - Client-server orientation
  - Object Studio POF
    - Very strong modelling
  - GLORP
    - Open-source
    - Extremely flexible mapping layer
  - SQLWorks
    - Good server orientation
    - *very* high-performance
- Goal: Synthesize the best of all these

Acknowledgements

- The Object People
- Cincom
- All the contributors and users of GLORP

References

- GLORP
  - http://www.glorp.org
  - http://glorp.sourceforge.net
- General
  - Ambler: Object Primer, http://www.agiledata.com (good emphasis on importance of both worlds)
  - Fowler: Patterns of Enterprise Application Architecture (good patterns, once you ignore the non-domain model stuff)
  - Fabian Pascal: Practical Issues in Database Management (pure relational extremist)

Transaction Issues

- One transaction at a time per session
  - Very simple usage model.
  - Work directly with original objects
  - No code modification
  - Works in a server, but with no sharing between users
- Parallel transactions may be desirable
  - Sharing read-only objects on a server
  - What-if scenarios
- Two possibilites
  - Explicit copies (TOPLink/Java)
  - Code-generation/modification (Object Extender/EJB/JDO)

Imaginary Tables

- Objects can map to more than one row
- Or less than one
- Embedded values a very simple case
- Recall mapping to a DISTINCT field
- Consider an object that combines several others, but has no row
- StoreClassExtension
  - ClassDefinition
  - Methods
  - Shared/Class Variables

Cache Policies

- Several policies available
  - Keep forever
  - Timed Expiry
  - Weak References
    - But with strong subset
  - Expiring proxies