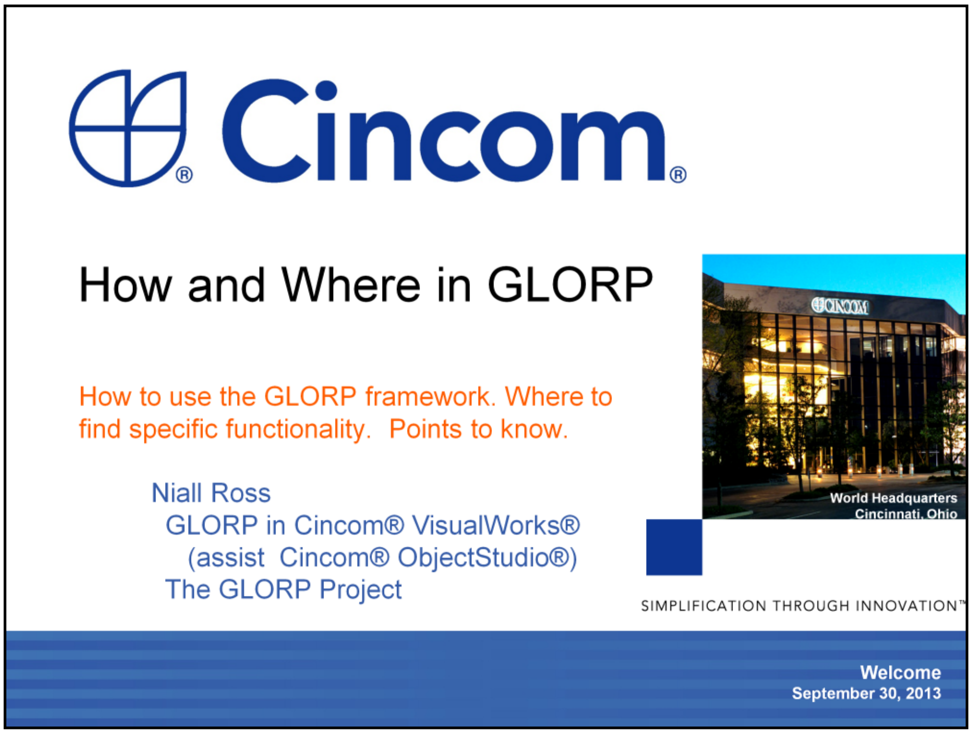


Cincom®

How and Where in GLORP

How to use the GLORP framework. Where to find specific functionality. Points to know.

Niall Ross
GLORP in Cincom® VisualWorks®
(assist Cincom® ObjectStudio®)
The GLORP Project

World Headquarters
Cincinnati, Ohio

SIMPLIFICATION THROUGH INNOVATION™

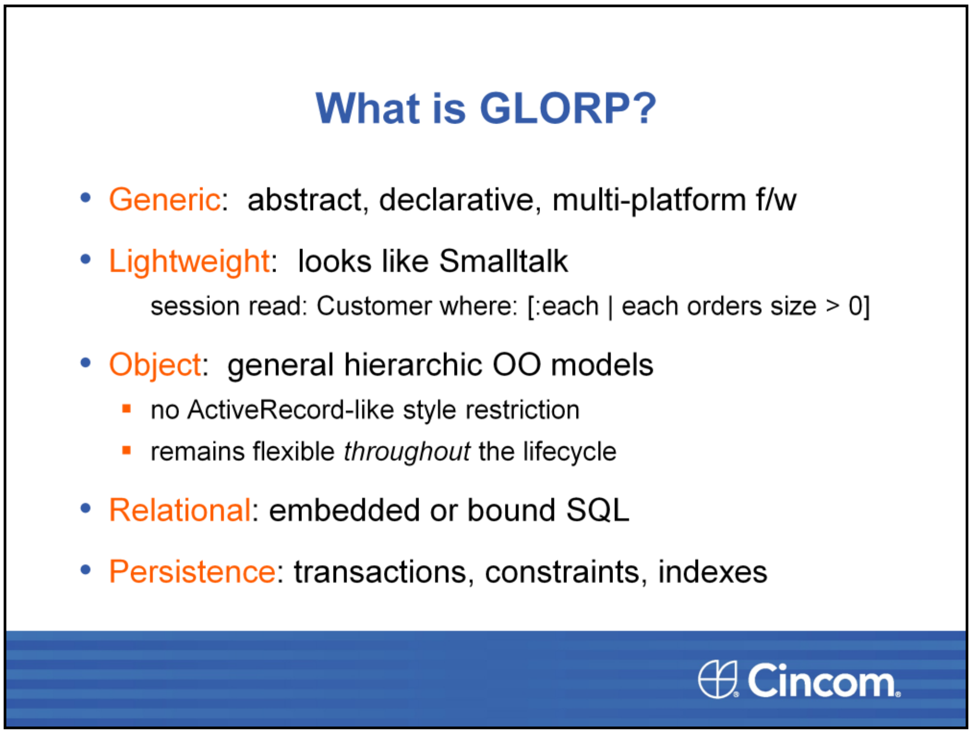
Welcome
September 30, 2013

My name is Niall Ross. My name is not Alan Knight. I started working with Glorp in 2004. I started doing weird stuff with Glorp in 2005. About a year ago, Alan gave me the Glorp OS project passwords, and Cincom put me in change of Glorp in VisualWorks.

In 90 minutes, I hope to show you some of what I know about Glorp, cover all my material, and even, with luck, get far enough to expose brutally the limits of my own knowledge and the fact that I am not Alan Knight.

DEMOS: to keep it simple and accessible, the demo code reuses existing GlorpTests for the most part. To join in those demos you can share, or just see the code on your machine, load GlorpTests. We'll be mainly using GlorpDemoDescriptorSystem and GlorpVideoDescriptorSystem. – and where we don't, you'll just have to watch my machine.

SHOW DefaultLogin.



What is GLORP?

- Generic: abstract, declarative, multi-platform f/w
- Lightweight: looks like Smalltalk

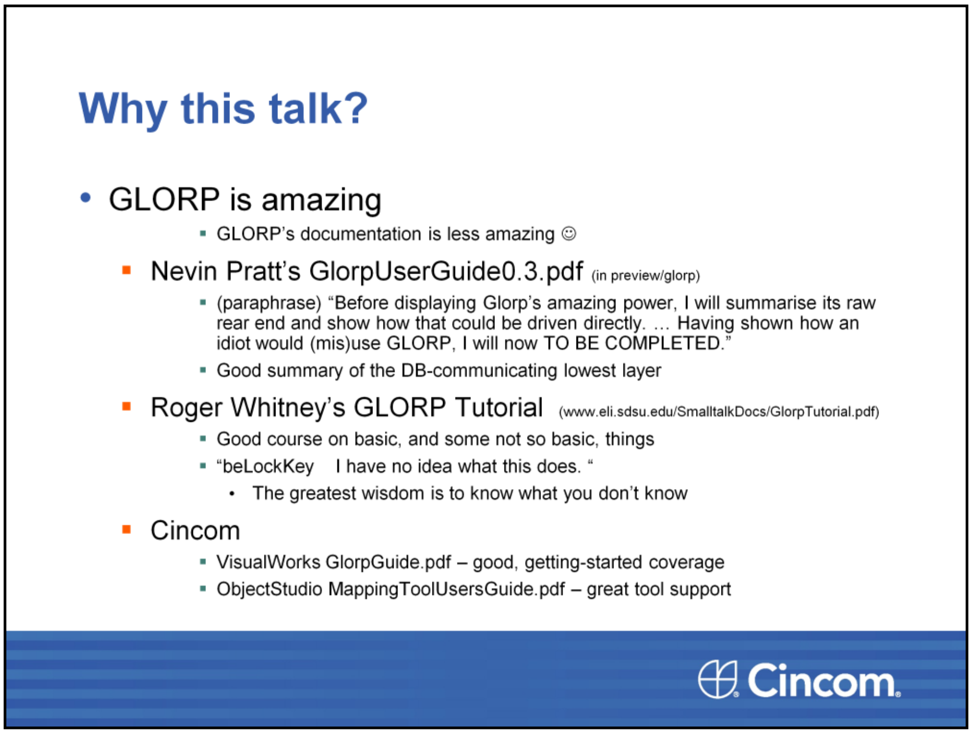
  session read: Customer where: [:each | each orders size > 0]
- Object: general hierarchic OO models
  - no ActiveRecord-like style restriction
  - remains flexible *throughout* the lifecycle
- Relational: embedded or bound SQL
- Persistence: transactions, constraints, indexes

Cincom

GLORP is the brainchild of Alan Knight, longtime head of our engineering team (now trying to make DART benefit from Smalltalk's insights).

Before he came to Cincom, Alan worked on TopLink. That taught him a great deal about object-relational mapping. GLORP was Alan getting TopLink right. As the saying goes, first make it run, then make it right. (And last make it fast , which I will talk about, but see also my colleague Tom Robinson's talk later today.)

GLORP has no ActiveRecord-like style restrictions, but more importantly, Glorp remains flexible all through the lifecycle. Too many frameworks are like Ruby on Rails; high speed in a greenfields situation and until you reach the end of the line, then suddenly abandoning you in untamed jungle. GLORP, as the product of a'make it right' phase, deals better with the real world.



Why this talk?

- GLORP is amazing
  - GLORP's documentation is less amazing ☺
  - Nevin Pratt's GlorpUserGuide0.3.pdf (in preview/glorp)
    - (paraphrase) "Before displaying Glorp's amazing power, I will summarise its raw rear end and show how that could be driven directly. … Having shown how an idiot would (mis)use GLORP, I will now TO BE COMPLETED."
    - Good summary of the DB-communicating lowest layer
  - Roger Whitney's GLORP Tutorial (www.eli.sdsu.edu/SmalltalkDocs/GlorpTutorial.pdf)
    - Good course on basic, and some not so basic, things
    - "beLockKey I have no idea what this does. "
      - The greatest wisdom is to know what you don't know
  - Cincom
    - VisualWorks GlorpGuide.pdf – good, getting-started coverage
    - ObjectStudio MappingToolUsersGuide.pdf – great tool support

Cincom

Why did I think there was need for a talk on Glorp. Well, Glorp is amazing. Glorp's documentation is less amazing, but we're making it better. Let's look at some existing resources.

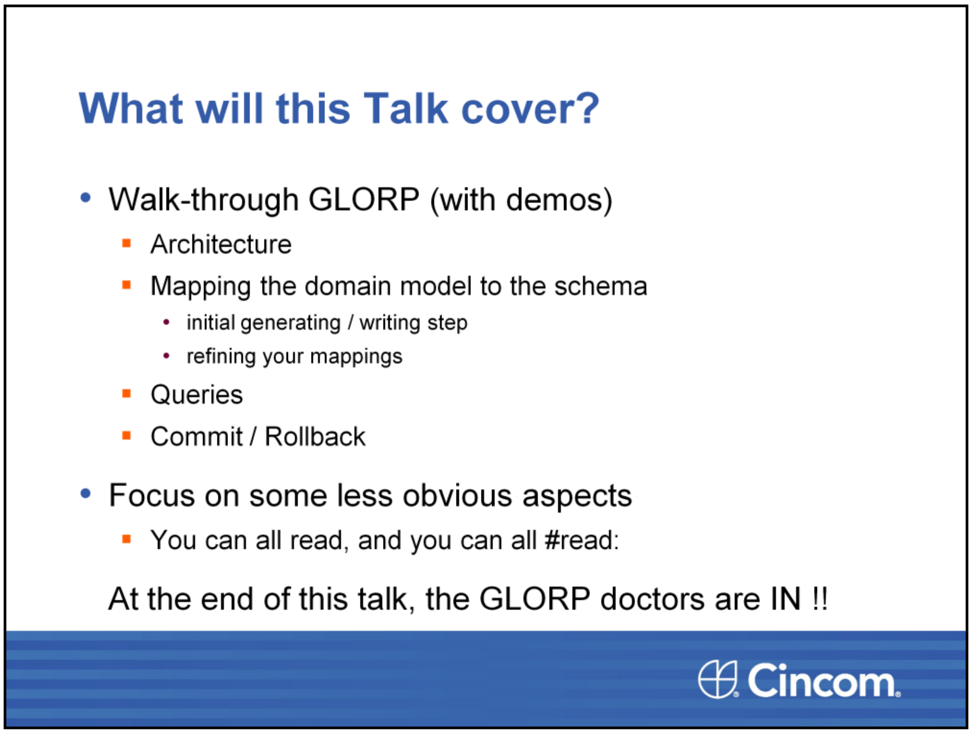
- Nevin Pratt's guide, though old, shows well how the back-end works, but is unfinished.

- Roger Witney's tutorial is a good get-started guide. Roger, like me, doesn't know everything about Glorp and has the honesty to say so.

- Our GlorpGuide.pdf will give you a clear start on the basics.

- Our ObjectStudio Mapping Tool guide will show you how to use that UI as a front-end to Glorp.

I decided to give this talk precisely to kick myself into documenting the vast amount of stuff that was not documented and not obvious.



What will this Talk cover?

- Walk-through GLORP (with demos)
  - Architecture
  - Mapping the domain model to the schema
    - initial generating / writing step
    - refining your mappings
  - Queries
  - Commit / Rollback
- Focus on some less obvious aspects
  - You can all read, and you can all #read:

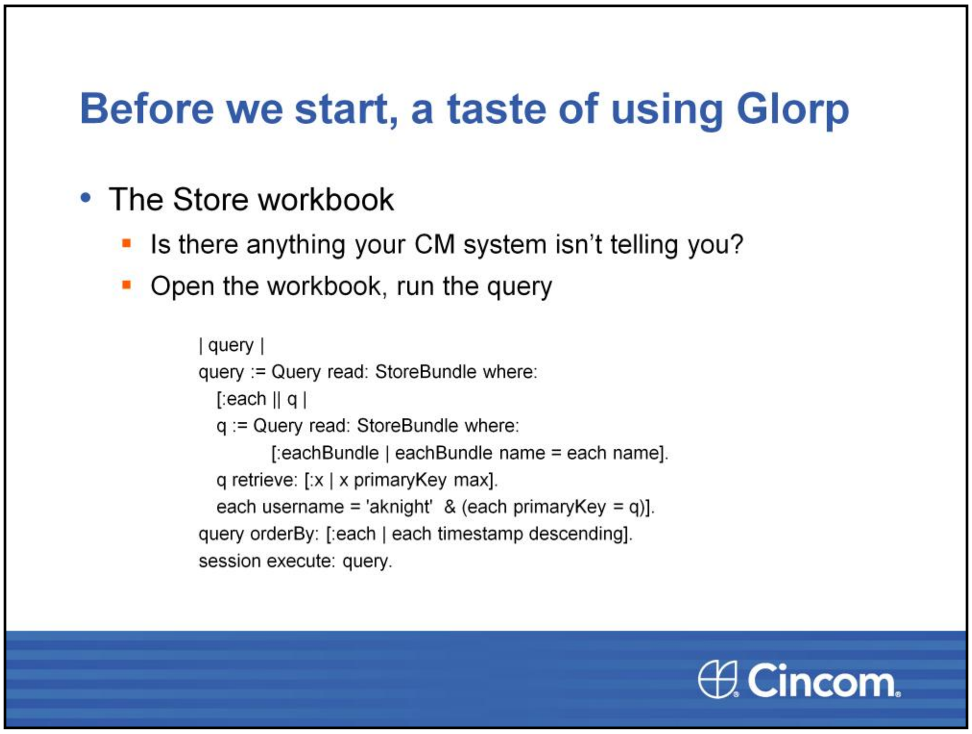
At the end of this talk, the GLORP doctors are IN !!

Cincom

We'll look at the architecture of of Glorp to try and give a context for what follows. Then we'll walk through the process. You need to represent your application's persistence schema in Glorp. An initial step of generating or writing will give me a brief opportunity for some boasting about recent developments and cool tools, but then we'll revert to the by-hand way and step through refining schemas. Next, how to read and write – that's what it's all about. Lastly we'll look at concurrency: how does Glorp address transactions.

Part of the purpose of this talk was to make me set down what I've had to work out over the years, things Alan has told me, etc. - and to discover what I did not yet know, or not well enough to explain it clearly. You can all read, and you can all master the basics of Glorp's #read: command in trivial 'one class=one table' scenarios. (Use Roger's course to help you on that.) I'll try to look at stuff that to Alan is doubtless also basic, but was not immediately obvious, at least to me.

Of course we're going to cover a lot of basics in order to highlight those points.



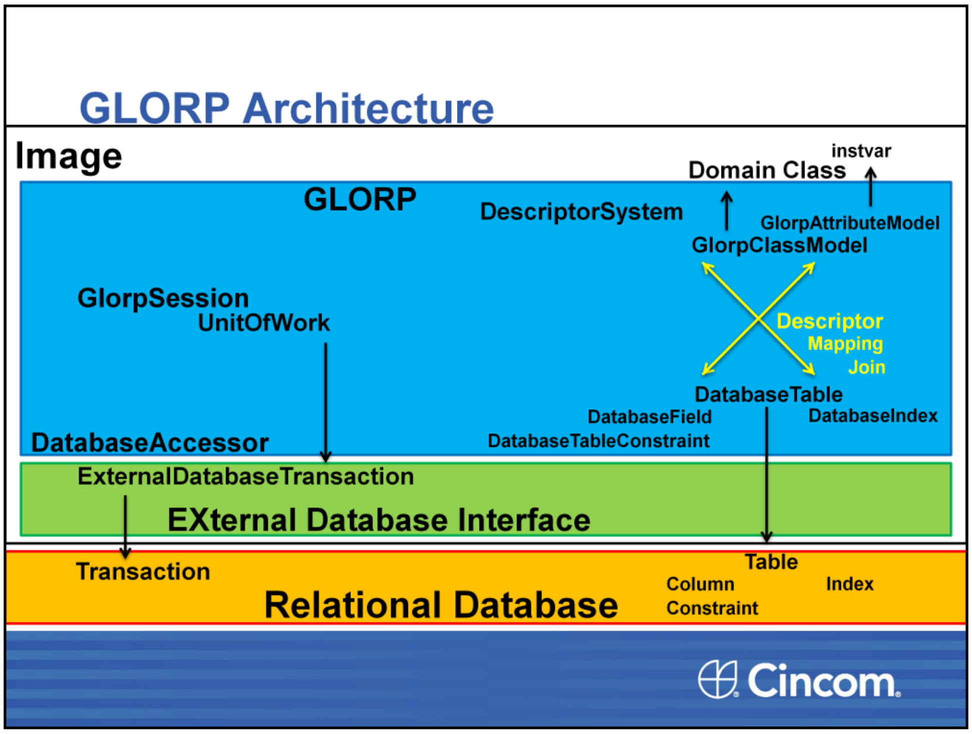
Before we start, a taste of using Glorp

- The Store workbook
  - Is there anything your CM system isn't telling you?
  - Open the workbook, run the query

```
| query |
query := Query read: StoreBundle where:
   [:each || q |
   q := Query read: StoreBundle where:
          [:eachBundle | eachBundle name = each name].
   q retrieve: [:x | x primaryKey max].
   each username = 'aknight'  & (each primaryKey = q)].
query orderBy: [:each | each timestamp descending].
session execute: query.
```

Cincom

In VisualWorks, the Store menu lets you open the store workbook on any Store login.



DATABASE LAYER: outside image (all other layers are within the image)

Supported databases include: Oracle, Oracle ODBC, PostgreSQL, Microsoft SQL Server, DB2, SQLite, Firebird/InterBase, and MySQL, also Microsoft Access, Ocelot, Adabas.

EXDI LAYER: any Glorp-using image will already have an 'external database interface' layer which knows how to communicate with relational databases. This EXDI layer must reify the database' commit/rollback state, at least in the form of commands it can pass on but more often as an actual object caching the state (the ExternalDatabaseTransaction object of the diagram).

Subclasses of Glorp's DatabaseAccessor class provide the dialect indirection for communicating with the dialect-specific EXDI layers. (Glorp also provides the class Dialect for some specific method calls, the number of which are being reduced as ANSI and other dialect compatibilities spread).

DOMAIN LAYER: the domain classes you wish to persist.

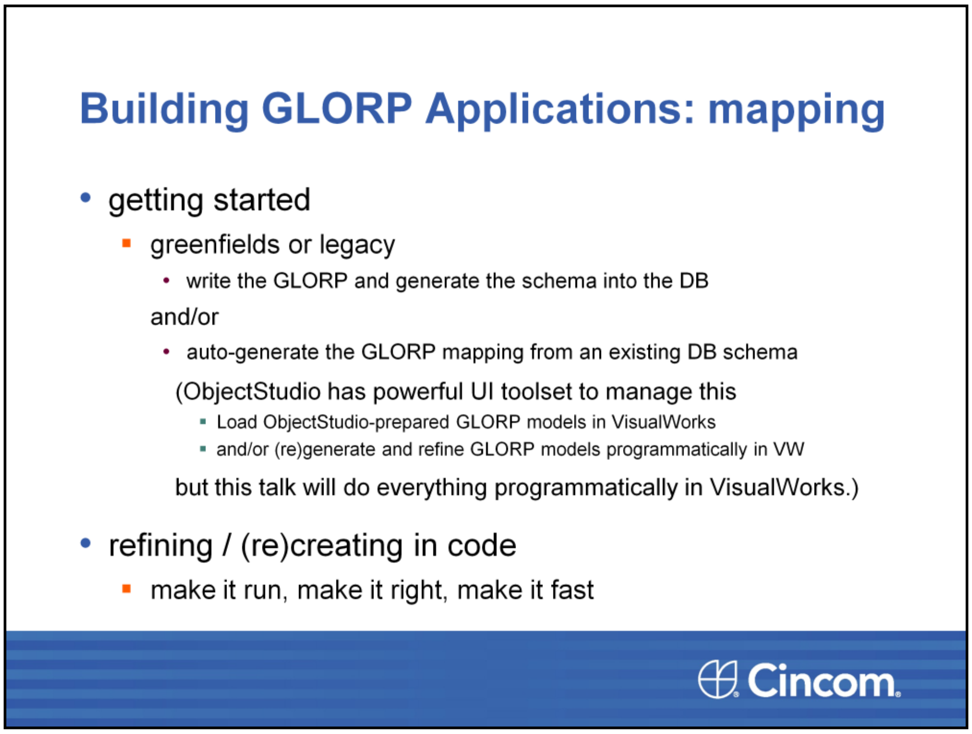
GLORP LAYER: Glorp models two aspects of the database:

• the transactional state of the RDB, which is usually already maintained by the external database interface layer, so Glorp can model that.

• the relational database' schema

Glorp also models the domain layer: which Classes, which instVars persist.

Above all, Glorp models the relationship between its model of (the relevant part of) the schema and its model of (the persistant part of) the domain.



Building GLORP Applications: mapping

- getting started
  - greenfields or legacy
    - write the GLORP and generate the schema into the DB

    and/or
    - auto-generate the GLORP mapping from an existing DB schema

      (ObjectStudio has powerful UI toolset to manage this
      - Load ObjectStudio-prepared GLORP models in VisualWorks
      - and/or (re)generate and refine GLORP models programmatically in VW

      but this talk will do everything programmatically in VisualWorks.)
- refining / (re)creating in code
  - make it run, make it right, make it fast

Cincom

This slide is the opportunity for a brief bit of boasting as regards the getting started step.

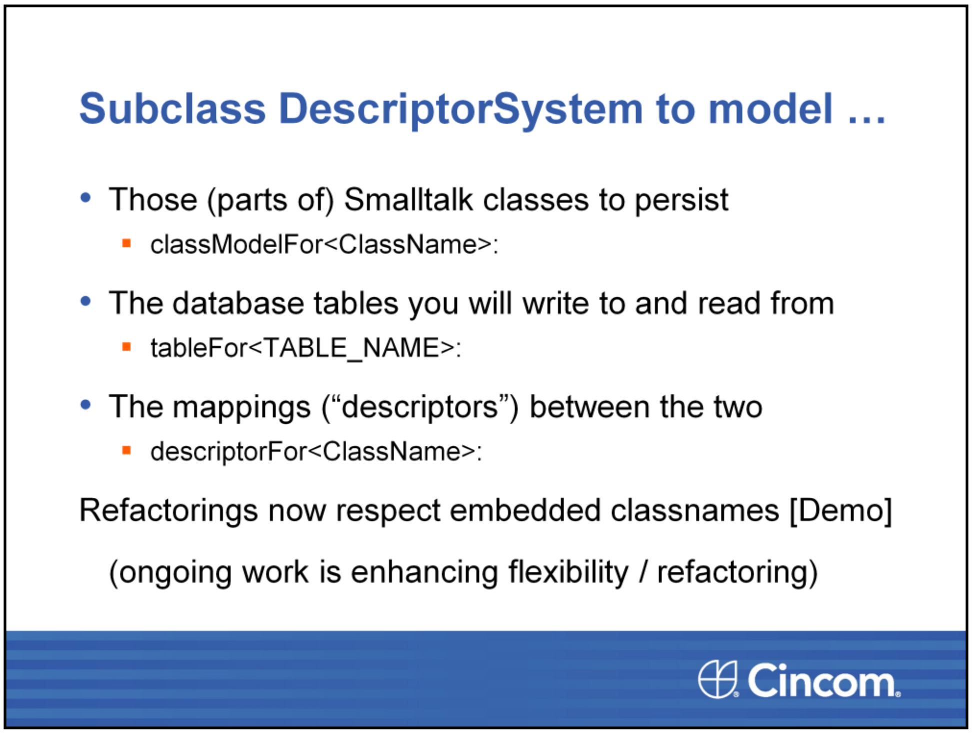
•If you're modelling your domain in a greenfields app, and then making a schema for it, you'd like some tools that would seamlessly move you from that modelling to that database mapping. Those of you who have seen Dirk's talks at prior ESUGs know there are cool tools in ObjectStudio that give you a UI and front-end to Glorp to let you do exactly that.

•If you have an existing schema for part of all of your application, and you saw my ESUG talk, you know that ObjectStudio's mapping tool can also do the reverse – turn the database' schema into Glorp.

However you get started, you will sooner or later be crafting / refining Glorp models by writing code in Glorp methods. That's what we'll look at in the rest of this talk. But first, the brief boastful demo.

•The latest VisualWorks can load ObjectStudio-developed models and refine them in VisualWorks while preserving their tool annotations, so now enterprise modelling in ObjectStudio while using, refining and deploying in VisualWorks is now a viable whole-lifecycle model.

•A minor addition is that some of the OS code to generate from legacy automatically is in VW (in preview - for good reason) but we expect it to provide service in the next release: for example, a VW image should be able automatically to verify that an installed Store database has run its updates successfully to have the latest configuration.



Subclass DescriptorSystem to model …

- Those (parts of) Smalltalk classes to persist
  - classModelFor<ClassName>:
- The database tables you will write to and read from
  - tableFor<TABLE_NAME>:
- The mappings (“descriptors”) between the two
  - descriptorFor<ClassName>:

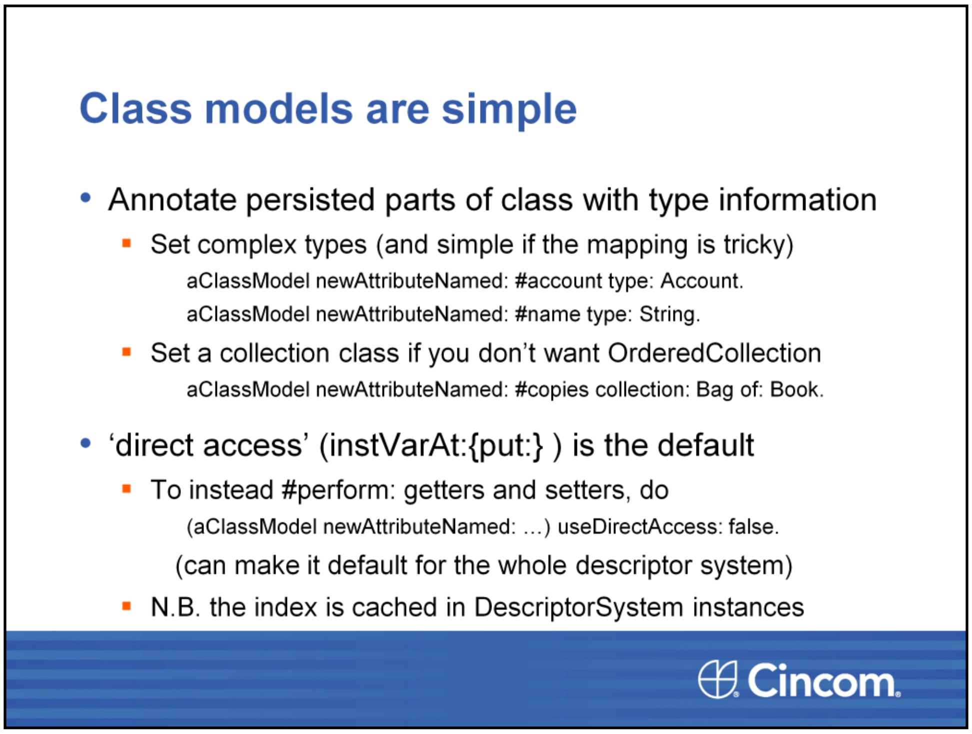
Refactorings now respect embedded classnames [Demo]

(ongoing work is enhancing flexibility / refactoring)

Cincom

The standard process, which need not be the only process, is to subclass DescriptorSystem, or one of our specialist abstract subclasses AtlasDescriptorSystem or ActiveRecordDescriptorSystem or MetadataDescriptorSystem, and then give it methods that embed data in their titles. This was not great for refactoring but (another boast alert – last one) I have done something about that.

DEMO renaming a domain class.


Class models are simple

- Annotate persisted parts of class with type information
  - Set complex types (and simple if the mapping is tricky)
    aClassModel newAttributeNamed: #account type: Account.
    aClassModel newAttributeNamed: #name type: String.
  - Set a collection class if you don't want OrderedCollection
    aClassModel newAttributeNamed: #copies collection: Bag of: Book.
- 'direct access' (instVarAt:{put:}) is the default
  - To instead #perform: getters and setters, do
    (aClassModel newAttributeNamed: …) useDirectAccess: false.
    (can make it default for the whole descriptor system)
  - N.B. the index is cached in DescriptorSystem instances

[DEMO] GlorpDemoDescriptorSystem>>classModelForGlorpBankAccount:

The class model's attributes must be annotated with the type information, since part of the impedance mismatch problem is that relational database schemas are typed. When an attribute is mapped directly to a single column, GLORP can often deduce the simple types - you will see models where Alan has not bothered to add them but just used newAttributeNamed:. Complex types must be stated.

Access is either direct (default) or by performing the getters and setters. Because the accessors are performed, you'd rather be direct unless you need the accessors called for some reason, but if you want them always called for your whole application, you can set that.

If a complex DescriptorSystem is instantiated a lot and changed little, or only changed in a distinct epoch, consider caching it. Every session will call all those ...For<Name>: methods as it creates . Be aware, this will cache the instVar indexes; if you shape-change any of the <Name> classes (e.g. add initial instVar that is not persisted), you'll need to recompute the descriptor system.

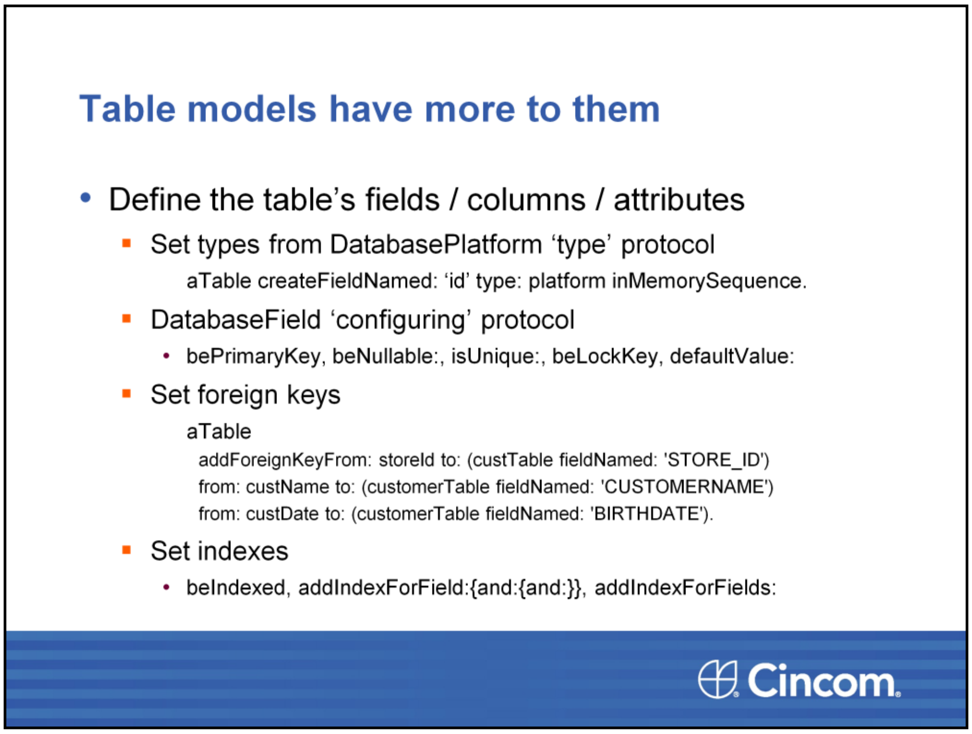


Table models have more to them

- Define the table's fields / columns / attributes
  - Set types from DatabasePlatform 'type' protocol

    aTable createFieldNamed: 'id' type: platform inMemorySequence.
  - DatabaseField 'configuring' protocol
    - bePrimaryKey, beNullable:, isUnique:, beLockKey, defaultValue:
  - Set foreign keys

```
aTable
  addForeignKeyFrom: storeId to: (custTable fieldNamed: 'STORE_ID')
  from: custName to: (customerTable fieldNamed: 'CUSTOMERNAME')
  from: custDate to: (customerTable fieldNamed: 'BIRTHDATE').
```

  - Set indexes
    - beIndexed, addIndexForField:{and:{and:}}, addIndexForFields:

Cincom

[DEMO] GlorpDemoDescriptorSystem>>tableForBANK_ACCT:

A field of type serial will automatically be made a primary key.

PERFORMANCE – Indexes

Writing these methods, use Extract-to-temp, Extract-to-method.

DatabasePlatform>>type protocol contains some fairly specific types

versionFieldFor:

generateOverExistingValues – if there already is a 1.0, do we allow publish of another 1.0.

You can create types too.

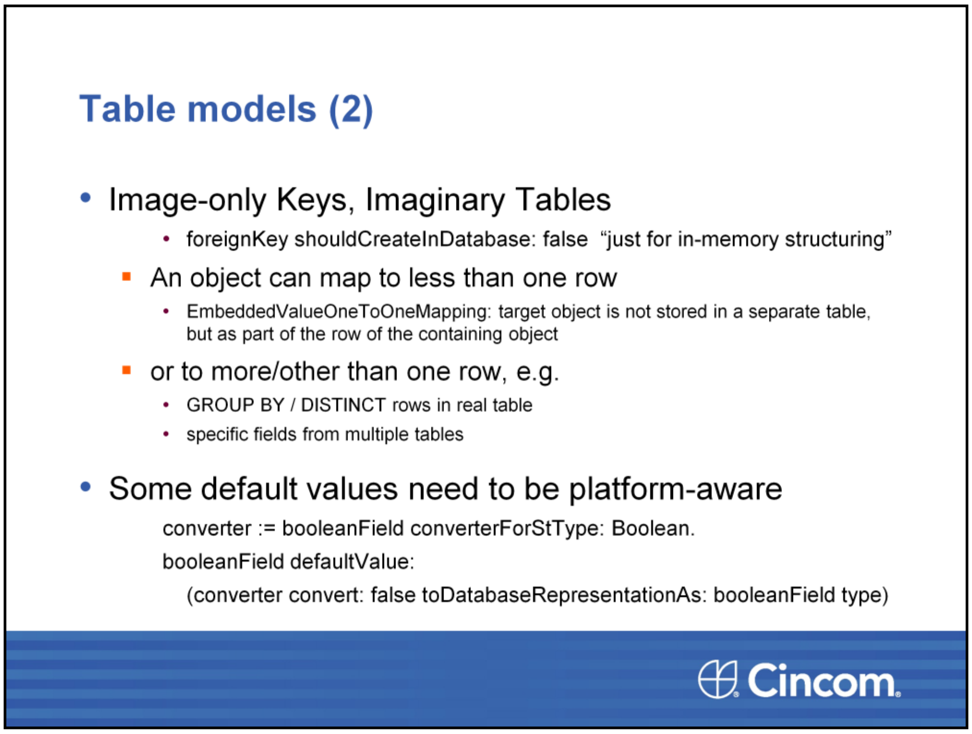


Table models (2)

- Image-only Keys, Imaginary Tables
    - foreignKey shouldCreateInDatabase: false “just for in-memory structuring”
  - An object can map to less than one row
    - EmbeddedValueOneToOneMapping: target object is not stored in a separate table, but as part of the row of the containing object
  - or to more/other than one row, e.g.
    - GROUP BY / DISTINCT rows in real table
    - specific fields from multiple tables
- Some default values need to be platform-aware

```
converter := booleanField converterForStType: Boolean.
booleanField defaultValue:
   (converter convert: false toDatabaseRepresentationAs: booleanField type)
```

Cincom

Not everything GLORP models is in the database, or vice versa. A key can be held in memory to help data integrity or ordering even though you know it’s not in the legacy database you are working with, or you even choose not to create it in the new database you are generating (maybe it applies in this descriptor system, but not in another that can also address the database).

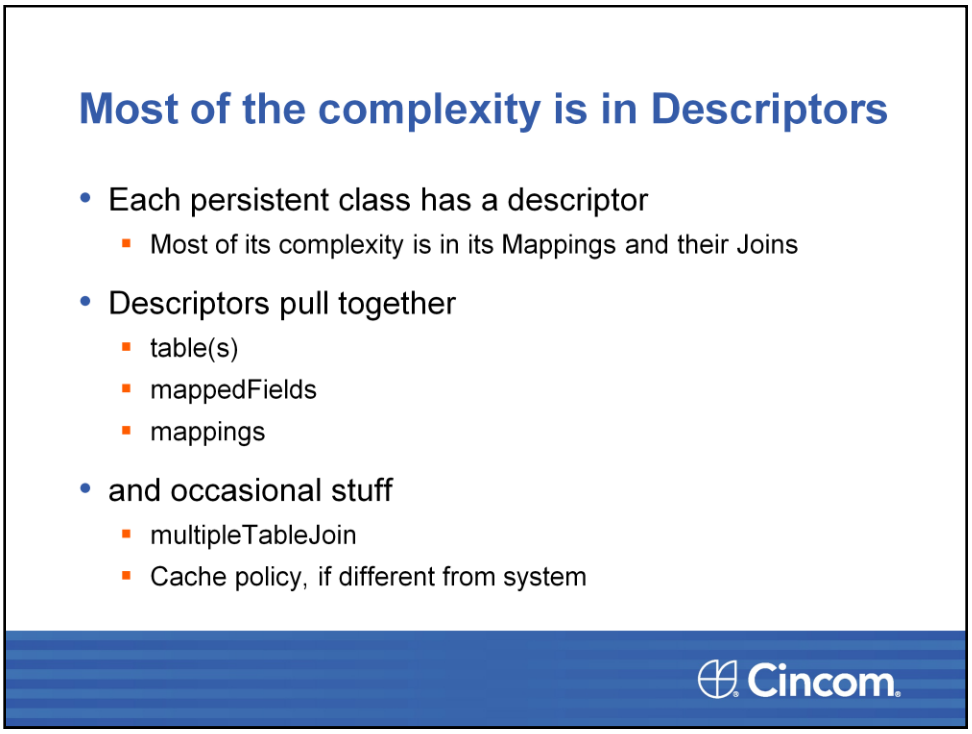
So why would you want to grab fields from multiple tables.

Example of less than one row: GlorpBankAccountNumber reifies some columns in BANK_ACCT table. GlorpBankAccount embeds this object and relates it to account holders, using the remaining columns.

Example of more than one row: GlorpVideoDescriptorSystem has tables for stores which rent and also sell videos, doing the latter both physically in stores and online. It has tables for these cases and for the credit histories of its customers, but no table just for customers. Glorp can synthesize a customer table by identifying same name, delivery address, etc.

The real tables for video purchase, rental and credit history all have a store id field, FK-linked to the store table's id field, and fields for the customer name and birthdate, which, with the store id field, are FK-linked to the imaginary table.

DEFAULT VALUES: At this point, we’re below where all the clever mapping is occurring – we’re setting up the machinery for clever mapping. So occasionally, more cluncky mapping occurs. In some database we represent a boolean directly, but in others it will need to be converted to an integer.


Most of the complexity is in Descriptors

- Each persistent class has a descriptor
  - Most of its complexity is in its Mappings and their Joins
- Descriptors pull together
  - table(s)
  - mappedFields
  - mappings
- and occasional stuff
  - multipleTableJoin
  - Cache policy, if different from system

DEMO/SHOW descriptorForGlorpBookstoreCustomer:

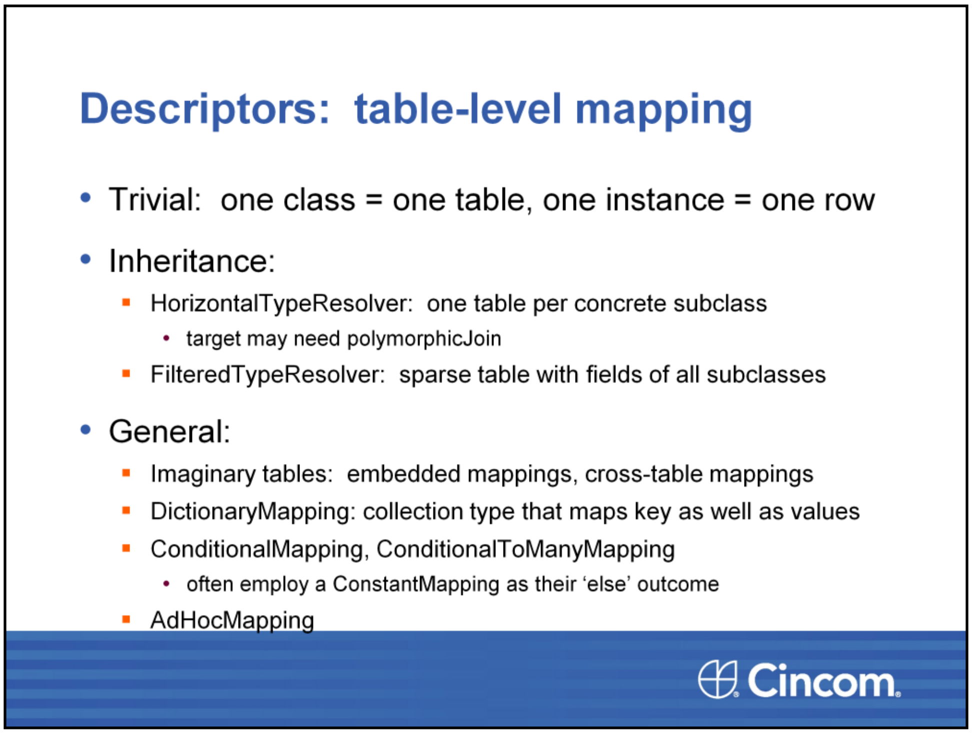
Enough descriptors have a single or main table that the first table is somewhat distinguished, but there can be many.

multipleTableJoin: suppose you map your object to multiple tables and some of your joins are outer joins. The GlorpDemoDescriptorSystem airline example includes an airline passenger where some are frequent flyers and others have never even joined a programme. You don’t want an inner join - that would mean anyone not in a frequent flyer programme was a non-person. So you use an outer join and then (only) you need to tell the descriptor. This is occasionally needed functionally when deleting if you’ve not mapped the primaryKeys of the secondary table but mainly for heuristics. Glorp can work out that you can ignore the outer join in many writing cases; it's not there so don't compute it. If the multiple table join is an outer join, and the row doesn't exist yet, then assume we won't need it and don't do the join. It means none of our mappings wrote to it.

Mostly you don't need this; the tables emerge obviously from the joins.

[The outer joins emerge from the joins - a join knows if it is outer; could we in fact make this a computed or cached value and spare the user thinking about it?]

Cache policy for a descriptor, i.e. for storing instances of its class in a UnitOfWork’s cache, can differ from the system’s default policy (though that is rare). (We may not have time to discuss cache policies.)



Descriptors: table-level mapping

- Trivial: one class = one table, one instance = one row
- Inheritance:
  - HorizontalTypeResolver: one table per concrete subclass
    - target may need polymorphicJoin
  - FilteredTypeResolver: sparse table with fields of all subclasses
- General:
  - Imaginary tables: embedded mappings, cross-table mappings
  - DictionaryMapping: collection type that maps key as well as values
  - ConditionalMapping, ConditionalToManyMapping
    - often employ a ConstantMapping as their 'else' outcome
  - AdHocMapping

Cincom

The trivial case of one class to one table we all easily understand. For examples of both kinds of inheritance, see the descriptor system presented STIC 2012 legacy talk, which is a development of the ESUG 2011 talk.)

DEMO LabDescriptorSystem and its subclasses.

typeResolverForConsultants

descriptorForDiagnosing/ReferringConsultants

register abstract and concrete

Then demo GeneticsDescriptorSystem>>descriptorForRassites:

polymorphicJoinTo:onFieldNamed:fromField:

This joins Rassites to Consultants conceptually, in fact to the two concrete subtables as Consultants is an abstract class and there is no Consultants table.

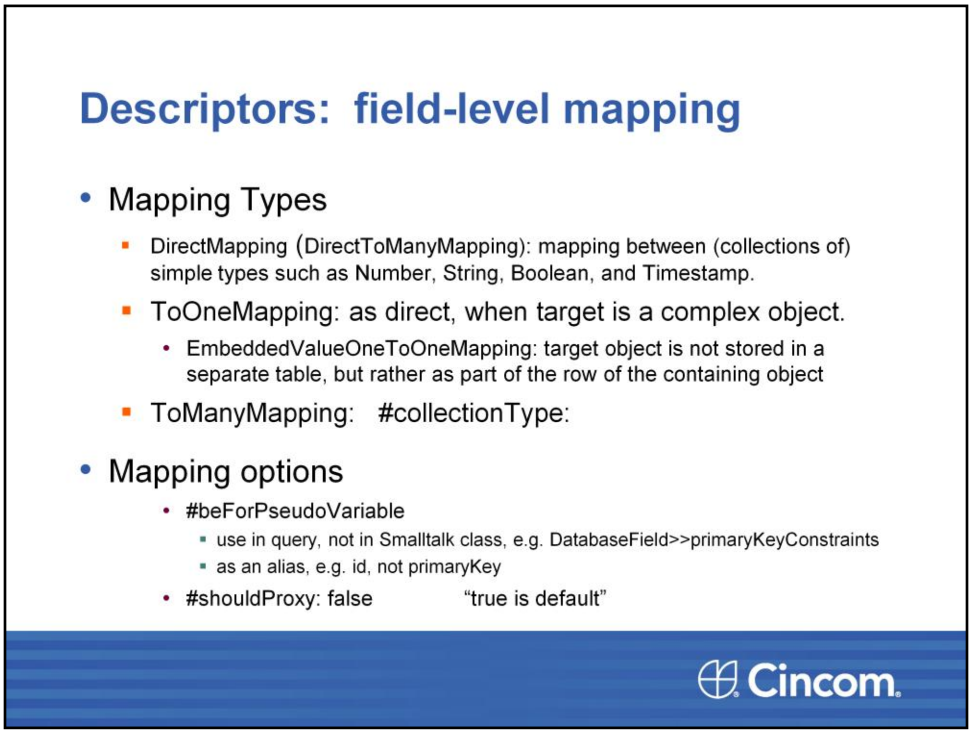
General

DEMO imaginary tables example with breakpoint to show RowMap.

GlorpObjectMappedToImaginaryTableTest>>testWrite

break UnitOfWork>>preCommit at end

AdHocMapping: when I said Glorp never landed you in the jungle …



Descriptors: field-level mapping

- Mapping Types
  - DirectMapping (DirectToManyMapping): mapping between (collections of) simple types such as Number, String, Boolean, and Timestamp.
  - ToOneMapping: as direct, when target is a complex object.
    - EmbeddedValueOneToOneMapping: target object is not stored in a separate table, but rather as part of the row of the containing object
  - ToManyMapping: #collectionType:
- Mapping options
  - #beForPseudoVariable
    - use in query, not in Smalltalk class, e.g. DatabaseField>>primaryKeyConstraints
    - as an alias, e.g. id, not primaryKey
  - #shouldProxy: false "true is default"

Cincom

Mapping Options include whether to proxy, the collection type, whether to provide separate link table and target table joins, and whether to give hints for the link table fields.

While SQL types such as String, Number, etc. can be mapped to corresponding Smalltalk classes, care must be exercised when mapping to types that do not possess identical dimensions.

ToManyMapping collectionType: Smalltalk provides a rich set of collection classes such as Dictionary, Bag, and SortedCollection, RDBMS provide a more limited repertoire of tables and blobs.

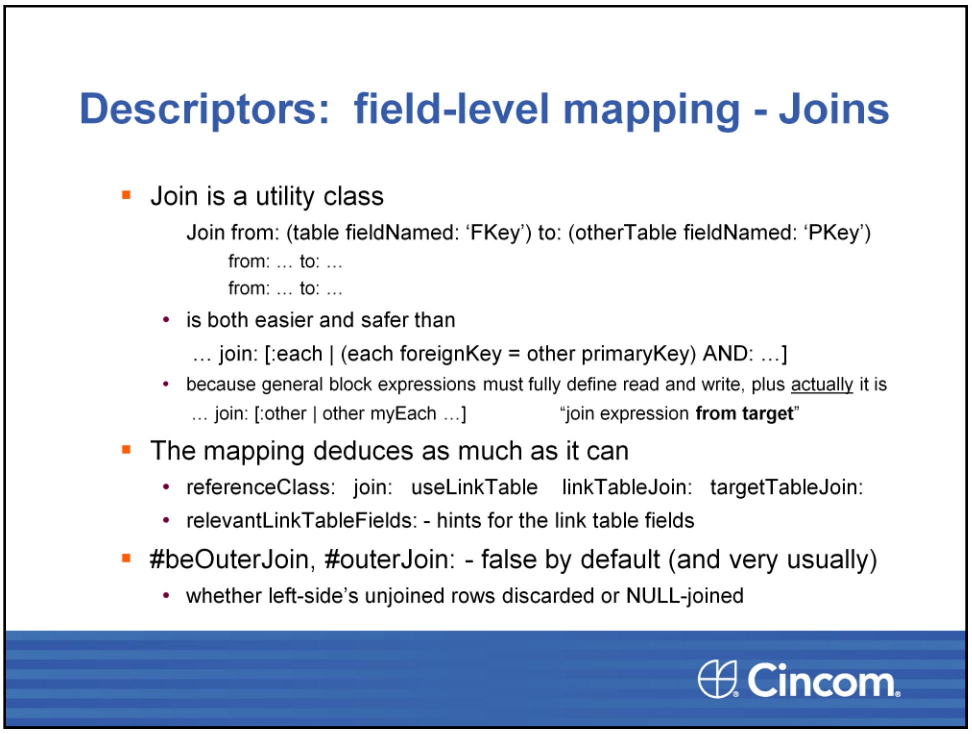
PseudoVariables: why send beForPseudoVariable? (1) When writing a where clause or other query aspect, you want to use this data because you know it is trivially there on the DB side, but you don’tt want to store it in that class on the Smalltalk side, e.g.

DatabaseField>>primaryKeyConstraints

DatabaseField does not have that instvar (it may participate, with others, in the primaryKeyConstraints – which should be singular primaryKeyConstraint and will be next cycle), but of course on the DB side, that is known and the where clause can use it. (So can use method to get DB column value inside query, but you do not want to store said value when retrieving whole object.)

(2) As an alias. (3) For ordering, using the same table or a value in a related table.

shouldProxy: controls whether you retrieve proxies or actual objects in your instvars. This affects performance.



Descriptors: field-level mapping - Joins

- Join is a utility class

  Join from: (table fieldNamed: ‘FKey’) to: (otherTable fieldNamed: ‘PKey’)
  from: … to: …
  from: … to: …

  - is both easier and safer than

    … join: [:each | (each foreignKey = other primaryKey) AND: …]
  - because general block expressions must fully define read and write, plus actually it is

    … join: [:other | other myEach …] “join expression **from target**”
- The mapping deduces as much as it can
  - referenceClass: join: useLinkTable linkTableJoin: targetTableJoin:
  - relevantLinkTableFields: - hints for the link table fields
- #beOuterJoin, #outerJoin: - false by default (and very usually)
  - whether left-side’s unjoined rows discarded or NULL-joined

Cincom

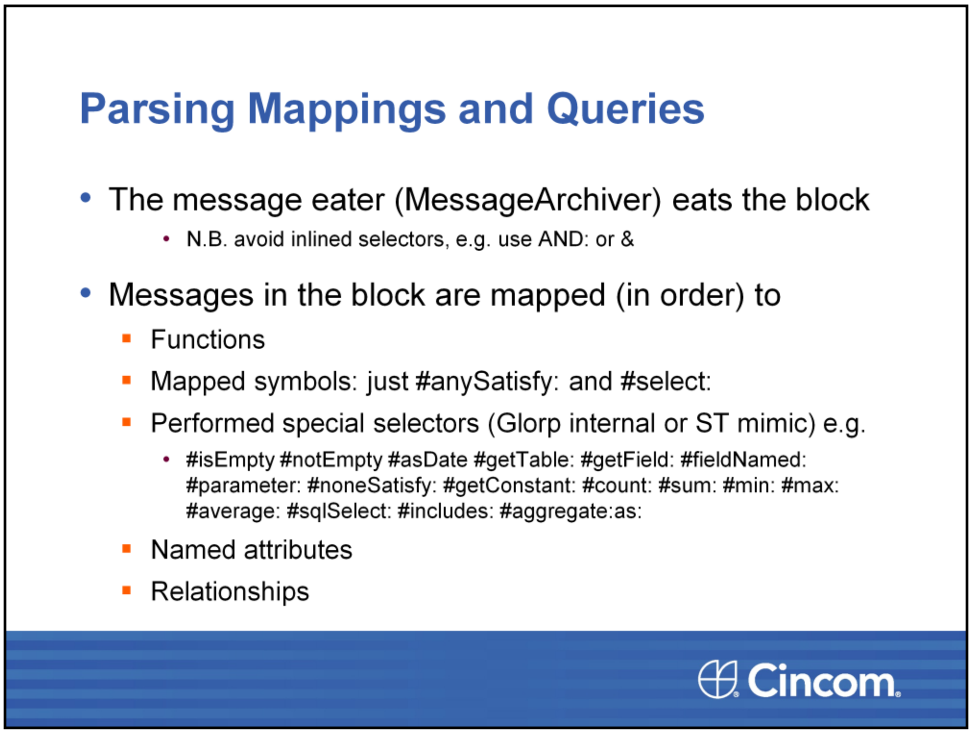
Implied Joins: often, the join can be computed from the foreign key relationship between the tables. We know Source class (from our descriptor), the Source table (from our descriptor), the Target class (from the classDescription) and the Table(s) for the target class from its descriptor. We also have the Foreign key relationship between source and target tables (from databaseTables). Some relationships, particularly many-to-many, may use a link table, specified as "useLinkTable“. GLORP will deduce the name of the link table if it follows standard patterns.

At other times, joins must be explicit and botb directions may need to be given. DEMO descriptorForGlorpVideoCustomer:

The method initializeJoin is a place to breakpoint if you suspect your descriptor system is not getting the join you expect.

The no-longer-used method mappingCriteria: (still visible in commented-out lines in some GlorpTest tests - see e.g. descriptorForGlorpMessage:), shows what early GLORP could not figure out.

Read-only mappings: in order to commit all of the changes in a unit of work, the entire graph of reachable objects must be searched. Read-only mappings truncate walking this graph.



Parsing Mappings and Queries

- The message eater (MessageArchiver) eats the block
  - N.B. avoid inlined selectors, e.g. use AND: or &
- Messages in the block are mapped (in order) to
  - Functions
  - Mapped symbols: just #anySatisfy: and #select:
  - Performed special selectors (Glorp internal or ST mimic) e.g.
    - #isEmpty #notEmpty #asDate #getTable: #getField: #fieldNamed: #parameter: #noneSatisfy: #getConstant: #count: #sum: #min: #max: #average: #sqlSelect: #includes: #aggregate:as:
  - Named attributes
  - Relationships

DEMO MessageArchiver with displayString (a very recently added method that can show a MessageArchiver instance, which is not otherwise easy to inspect).

If you put a breakpoint in a where-clause block, the debugger halts in the MessageArchiver's eating of the block, not, of course, in the resulting running of the SQL.

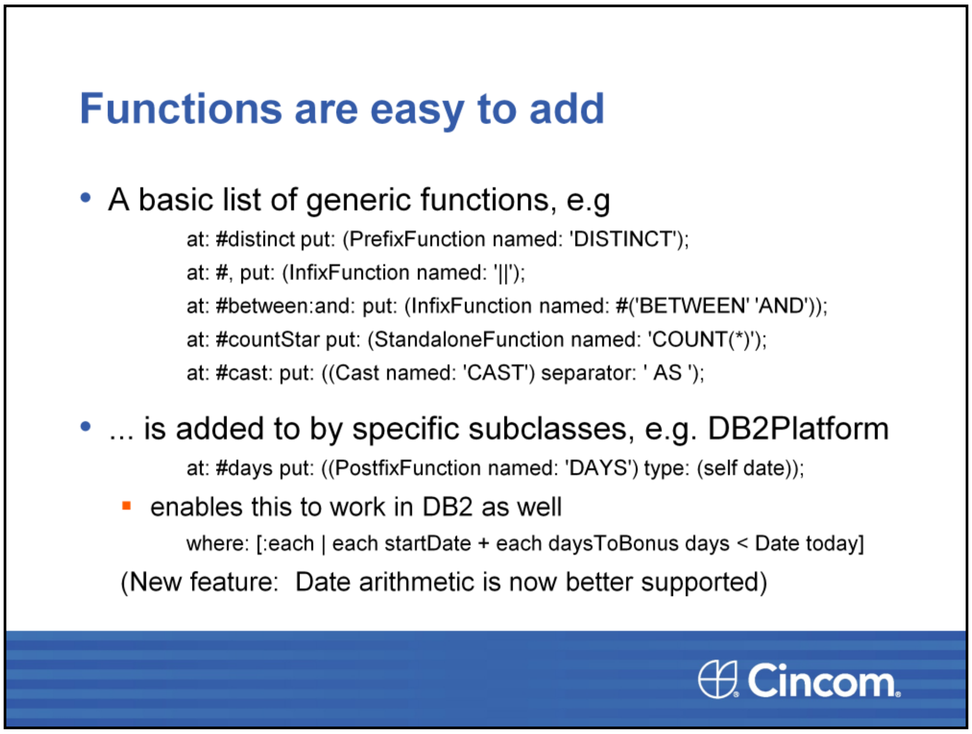
Use displayString (newly added, not yet in public domain Glorp version) to understand eaten blocks.

Special selectors assist in writing iterators in blocks in standard Smalltalk style. They also provide special handling for Glorp-used methods (e.g. getTable:, fieldNamed:, etc.) that must not be eaten.

zeroArgumentSpecialSelectors - #isEmpty #notEmpty #asDate

oneArgumentSpecialSelectors - #getTable: #getField: #fieldNamed: #parameter: #noneSatisfy: #getConstant: #count: #sum: #min: #max: #average: #sqlSelect: #includes:

twoArgumentSpecialSelectors - #aggregate:as:



Functions are easy to add

- A basic list of generic functions, e.g

```
at: #distinct put: (PrefixFunction named: 'DISTINCT');
at: #, put: (InfixFunction named: '||');
at: #between:and: put: (InfixFunction named: #('BETWEEN' 'AND'));
at: #countStar put: (StandaloneFunction named: 'COUNT(*)');
at: #cast: put: ((Cast named: 'CAST') separator: ' AS ');
```

- ... is added to by specific subclasses, e.g. DB2Platform

```
at: #days put: ((PostfixFunction named: 'DAYS') type: (self date));
```

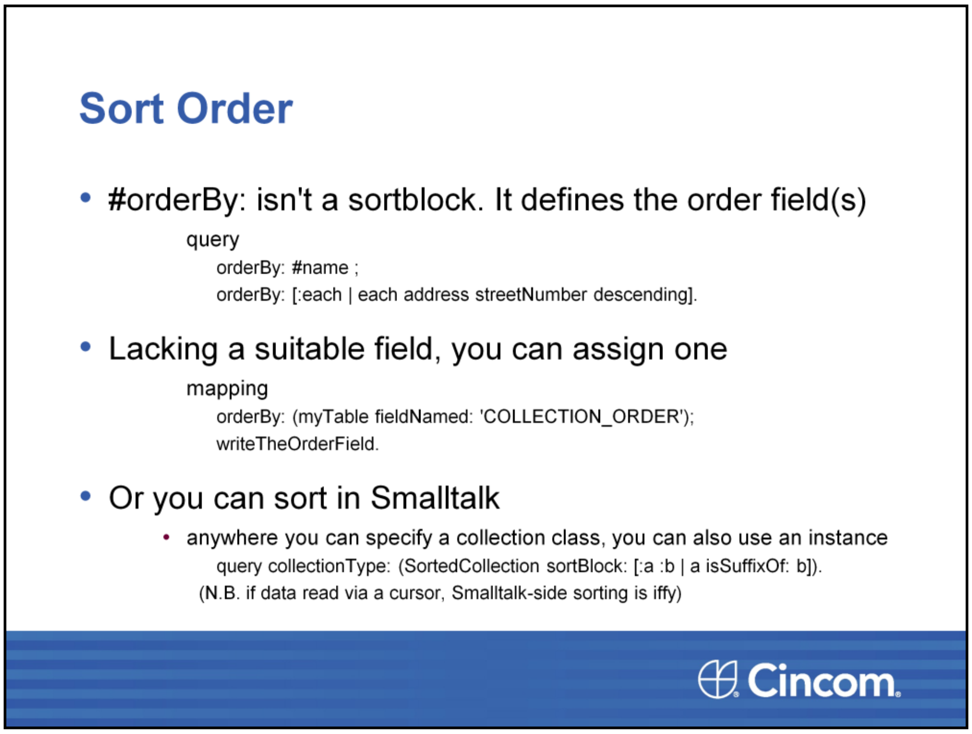
  - enables this to work in DB2 as well

```
where: [:each | each startDate + each daysToBonus days < Date today]
```

  (New feature: Date arithmetic is now better supported)

A DualRoleFunction is used when a database platform demands differently-phrased SQL in a where clause and in a select list.

DEMO functions in DB2Platform>>initializeFunctions.



Sort Order

- #orderBy: isn't a sortblock. It defines the order field(s)

```
query
    orderBy: #name ;
    orderBy: [:each | each address streetNumber descending].
```

- Lacking a suitable field, you can assign one

```
mapping
    orderBy: (myTable fieldNamed: 'COLLECTION_ORDER');
    writeTheOrderField.
```

- Or you can sort in Smalltalk
  - anywhere you can specify a collection class, you can also use an instance

```
    query collectionType: (SortedCollection sortBlock: [:a :b | a isSuffixOf: b]).
  (N.B. if data read via a cursor, Smalltalk-side sorting is iffy)
```

Cincom

If the information is there, then you can specify #orderBy: on your mapping. This isn't a sortblock, but rather defines the field(s) you want to order by.

```
 e.g.  orderBy: #name
       orderBy: [:each | each address streetNumber descending]
```

specify the order in which objects are read from the database. This can be done either by sending orderBy: messages to a Query object before executing it, or by using the orderBy: message when defining the domain mapping (the latter tells the database server to sort the results, which may be faster).

Sending orderBy: repeatedly adds criteria.

If the information to determine the sort order isn't there, but you really want to preserve the order, you can add a field and make Glorp store the information:

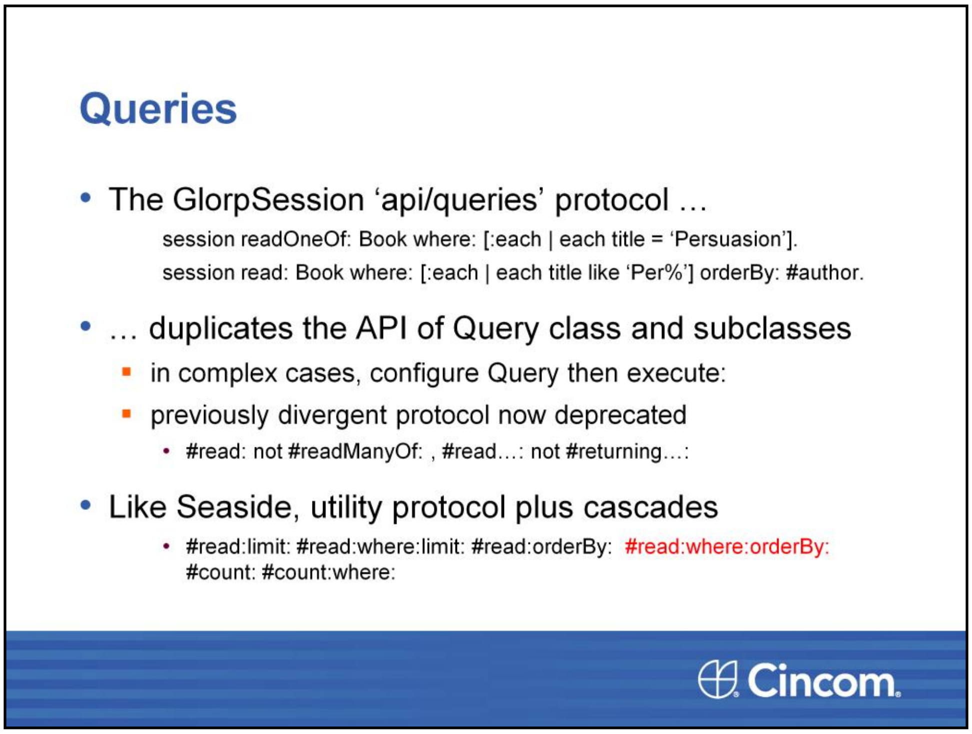
```
    orderBy: (myTable fieldNamed: 'COLLECTIONORDER');
    writeTheOrderField.
```

Finally, if you really want to do the sorting in Smalltalk (and be aware that this will defeat features like maintaining open cursors on large results sets) then it's slightly non-obvious, but in the places where you can specify collection types as classes, you can also use an instance. So

```
  query collectionType: (SortedCollection sortBlock: [:a :b | a < b]).
```

works.

Similarly you can set the collectionType on a mapping.


Queries

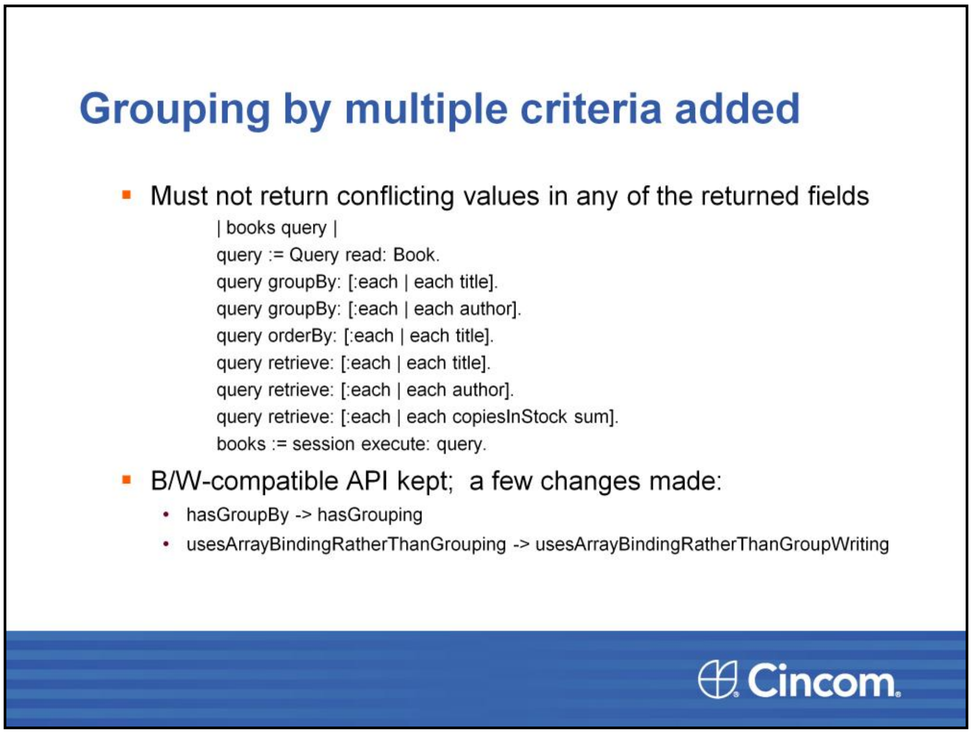
- The GlorpSession ‘api/queries’ protocol …

  session readOneOf: Book where: [:each | each title = ‘Persuasion’].

  session read: Book where: [:each | each title like ‘Per%’] orderBy: #author.
- … duplicates the API of Query class and subclasses
  - in complex cases, configure Query then execute:
  - previously divergent protocol now deprecated
    - #read: not #readManyOf: , #read…: not #returning…:
- Like Seaside, utility protocol plus cascades
    - #read:limit: #read:where:limit: #read:orderBy: #read:where:orderBy:
      #count: #count:where:

DEMO run query with breakpoint.

Like Seaside, GLORP uses cascades rather than huge repetitive API to create its model-defining methods. There is a good deal of utility API but where it runs out just use cascades. The point where the utility API runs out is slightly idiosyncratic – for example #read:where:orderBy: is not there (maybe I’ll add it).



Grouping by multiple criteria added

- Must not return conflicting values in any of the returned fields

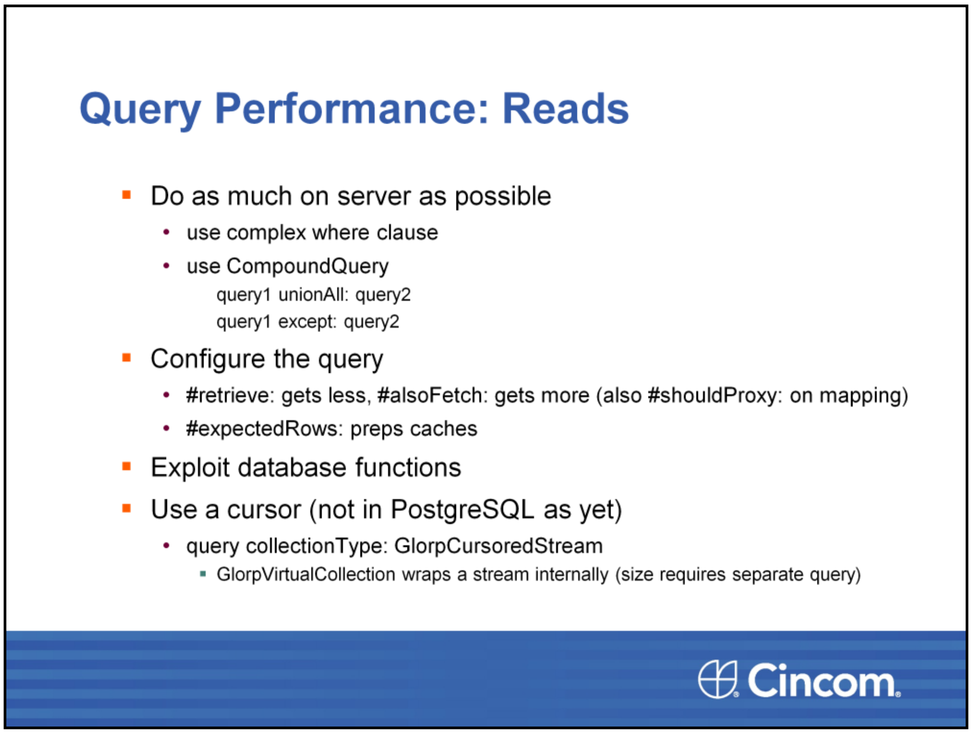
```
| books query |
query := Query read: Book.
query groupBy: [:each | each title].
query groupBy: [:each | each author].
query orderBy: [:each | each title].
query retrieve: [:each | each title].
query retrieve: [:each | each author].
query retrieve: [:each | each copiesInStock sum].
books := session execute: query.
```

- B/W-compatible API kept; a few changes made:
  - hasGroupBy -> hasGrouping
  - usesArrayBindingRatherThanGrouping -> usesArrayBindingRatherThanGroupWriting

groupBy: is now like orderBy: in that move than one can be sent to a query (recent change: previously, a Query could only have only one groupBy: criterion).

If you use groupBy: to aggregate rows based on certain columns, you must either aggregate any other columns (i.e. sum: them or min: or max: or whatever) or be confident that any such columns do not have divergent values in the rows where the grouped columns have the same values. (If there are and such divergences, the query returns an error.) So be aware multiple groupBy's will error if your data is not groupable.

Trivial API changes make this more similar to ordering (hasOrdering and hasGrouping, no longer hasGroupBy), and better distinguish between grouping and group-writing (a quite different concept relating to sending many SQL commands in a single trip to the database if the DB system allows it).



Query Performance: Reads

- Do as much on server as possible
  - use complex where clause
  - use CompoundQuery

    query1 unionAll: query2

    query1 except: query2
- Configure the query
  - #retrieve: gets less, #alsoFetch: gets more (also #shouldProxy: on mapping)
  - #expectedRows: preps caches
- Exploit database functions
- Use a cursor (not in PostgreSQL as yet)
  - query collectionType: GlorpCursoredStream
    - GlorpVirtualCollection wraps a stream internally (size requires separate query)

Reads can be very time-consuming. Proxies fault one by one. Queries can be expensive. Various optimizations are available:

•Complex where conditions

•Reading subset of data/non-object data (retrieve:)

•Reading additional data (alsoFetch:)

•Database Functions

•Cursors (not PostgreSQL)

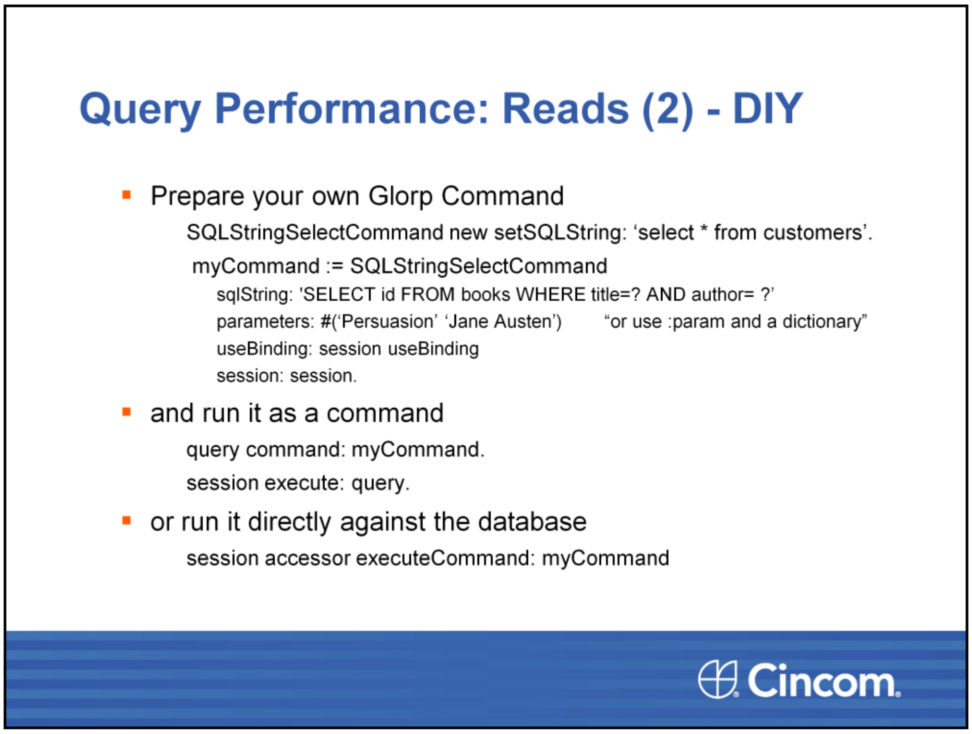
•Combine queries with union: unionAll: and except:

•Write your own SQL

•Queries can return a stream of results rather than a collection

query collectionType: GlorpCursoredStream

Also GlorpVirtualCollection (wraps a stream internally, size requires separate query)



Query Performance: Reads (2) - DIY

- Prepare your own Glorp Command

```
SQLStringSelectCommand new setSQLString: ‘select * from customers’.
 myCommand := SQLStringSelectCommand
    sqlString: 'SELECT id FROM books WHERE title=? AND author= ?’
    parameters: #(‘Persuasion’ ‘Jane Austen’)       “or use :param and a dictionary”
    useBinding: session useBinding
    session: session.
```

- and run it as a command

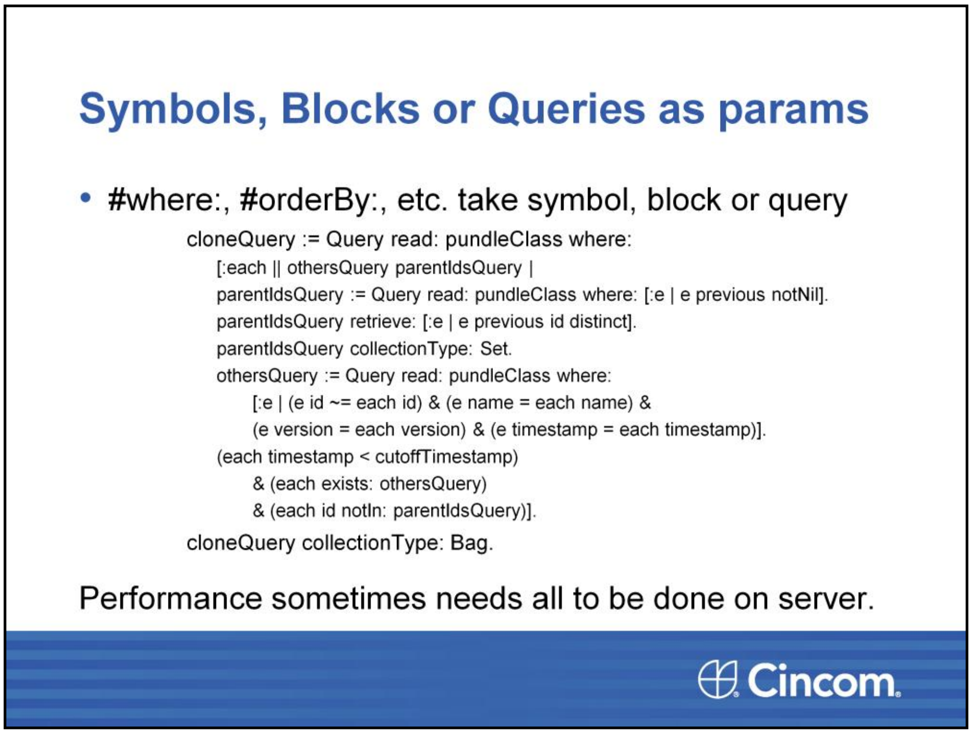
```
query command: myCommand.
session execute: query.
```

- or run it directly against the database

```
session accessor executeCommand: myCommand
```

Cincom.

By-hand SQL is sometimes useful.



Symbols, Blocks or Queries as params

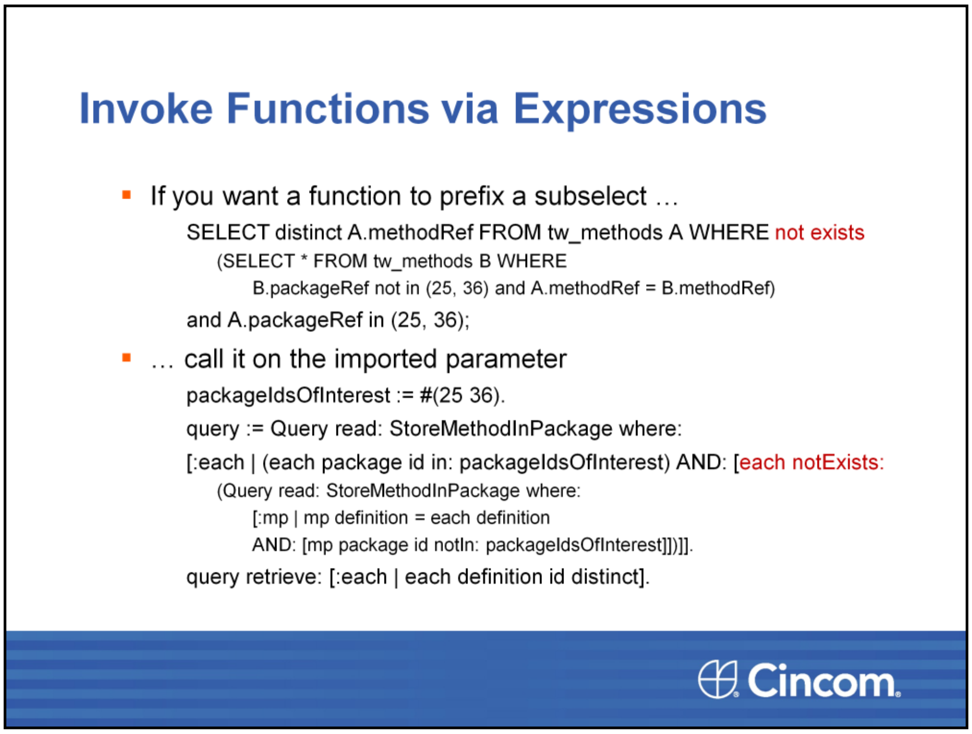
- #where:, #orderBy:, etc. take symbol, block or query

```
cloneQuery := Query read: pundleClass where:
    [:each || othersQuery parentIdsQuery |
    parentIdsQuery := Query read: pundleClass where: [:e | e previous notNil].
    parentIdsQuery retrieve: [:e | e previous id distinct].
    parentIdsQuery collectionType: Set.
    othersQuery := Query read: pundleClass where:
        [:e | (e id ~= each id) & (e name = each name) &
        (e version = each version) & (e timestamp = each timestamp)].
    (each timestamp < cutoffTimestamp)
        & (each exists: othersQuery)
        & (each id notIn: parentIdsQuery)].
cloneQuery collectionType: Bag.
```

Performance sometimes needs all to be done on server.

Where clauses and similar can be just symbols (i.e. the selectors of methods) or blocks or entire subqueries.

The example looks for pundleClass instances (earlier than cutOffTimestamp) that have no parents (and so trivial to delete) that are clones of others.



Invoke Functions via Expressions

- If you want a function to prefix a subselect …

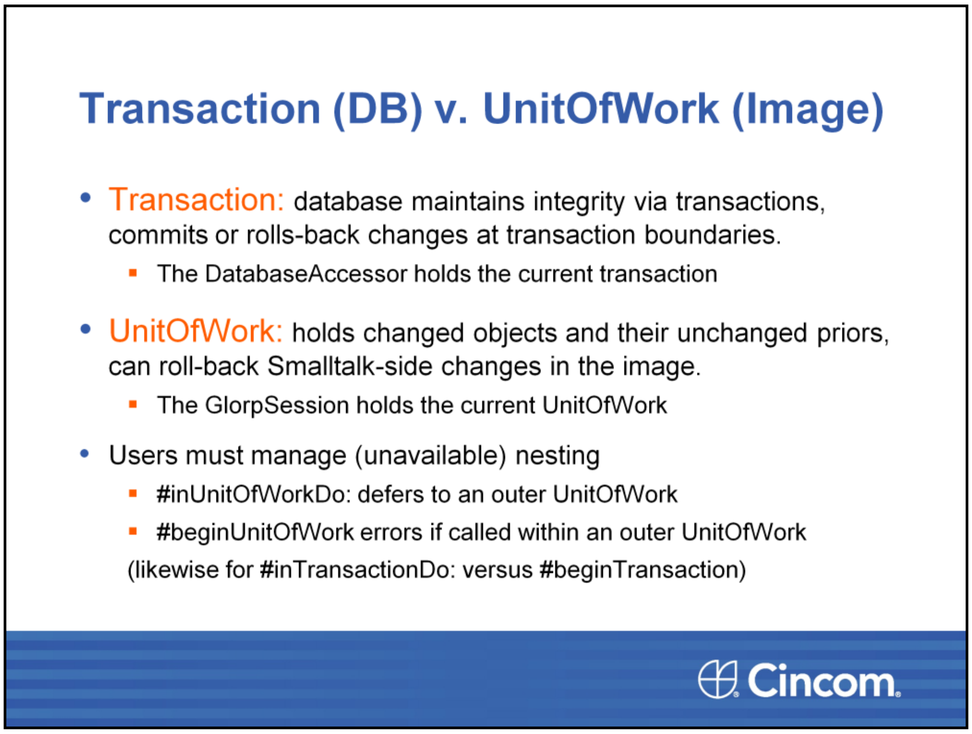
```
SELECT distinct A.methodRef FROM tw_methods A WHERE not exists
    (SELECT * FROM tw_methods B WHERE
        B.packageRef not in (25, 36) and A.methodRef = B.methodRef)
and A.packageRef in (25, 36);
```

- … call it on the imported parameter

```
packageIdsOfInterest := #(25 36).
query := Query read: StoreMethodInPackage where:
[:each | (each package id in: packageIdsOfInterest) AND: [each notExists:
    (Query read: StoreMethodInPackage where:
        [:mp | mp definition = each definition
        AND: [mp package id notIn: packageIdsOfInterest]])]].
query retrieve: [:each | each definition id distinct].
```

Functions are invoked by sending selectors (that map to the functions either generally or for a specific database platform: see methods createBasicFunctionsFor: and initializeFunctions). Smalltalk-style, the function selectors must be sent to expressions, although the resulting SQL keywords may appear prefixing subselects.

MappingExpression>>isEmpty calls notExist: and could be used in this example. I used notExists: to show more clearly how the SQL and the Smalltalk are related.



Transaction (DB) v. UnitOfWork (Image)

- Transaction: database maintains integrity via transactions, commits or rolls-back changes at transaction boundaries.
  - The DatabaseAccessor holds the current transaction
- UnitOfWork: holds changed objects and their unchanged priors, can roll-back Smalltalk-side changes in the image.
  - The GlorpSession holds the current UnitOfWork
- Users must manage (unavailable) nesting
  - #inUnitOfWorkDo: defers to an outer UnitOfWork
  - #beginUnitOfWork errors if called within an outer UnitOfWork

  (likewise for #inTransactionDo: versus #beginTransaction)

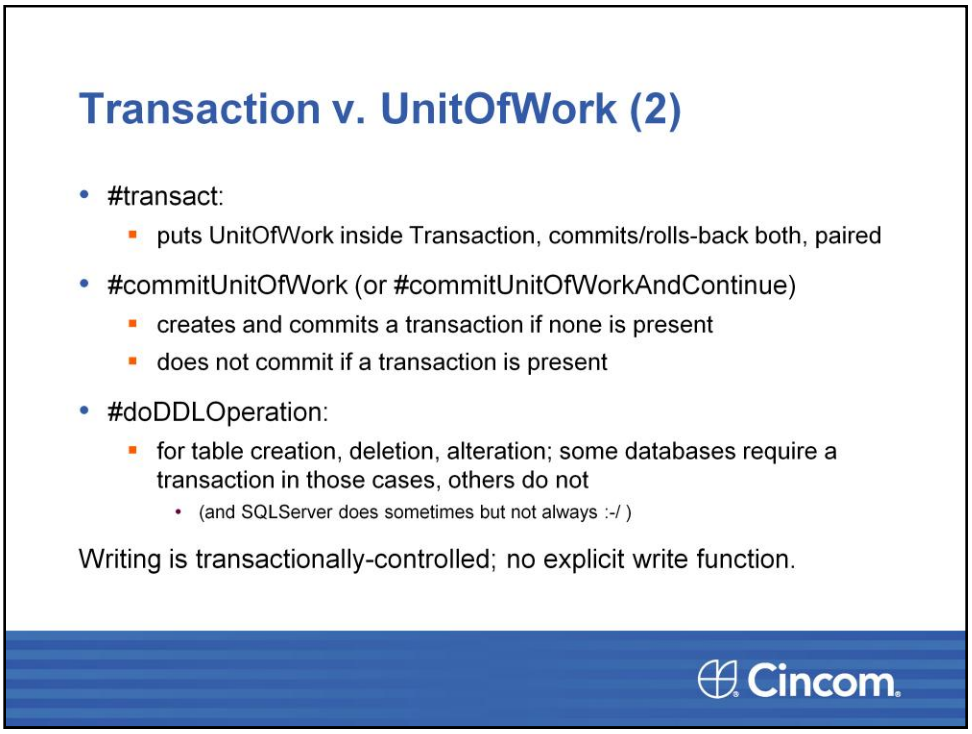
Cincom

There are methods inUnitOfWorkDo: and begin/rollback/commitUnitOfWork. Similarly there are inTransactionDo: and #begin/rollback/commitTransaction

There are two basic scenarios. In the first scenario, you wrap your updates in a UnitOfWork using #beginUnitOfWork, doing the updates, and then #commitUnitOfWork. Method #commitUnitOfWork privately sends #beginTransaction, writes all the data to disk, and then sends #commitTransaction. So, the UnitOfWork itself keeps track of all the updates, and if commited, it sends them on to the database within a private Transaction.

In the second scenario, you control the Transaction, starting with #beginTransaction. If you then wrap your updates inside a UnitOfWork, you get the same behavior as before--the UnitOfWork won't send to disk until you say #commitUnitOfWork. But this time, your UnitOfWork will notice that you have an ongoing Transaction so it sends neither #beginTransaction nor #commitTransaction. Instead, it expects you to send #commitTransaction later. In short, it won't touch the Transaction control since you own it. This scenario can be used to execute a series of operations, each in a separate UnitOfWork, where all can be committed or rolled back downstream.

inUnitOfWorkDo: - This method will only commit or rollback a unit of work that it creates. If there is an ongoing unit of work, the sender must commit or rollback. Likewise, for inTransactionDo:, if there is an ongoing transaction, the sender must commit or rollback the transaction. By contrast, calling beginUnitOfWork/Transaction will raise an error if one already exists.

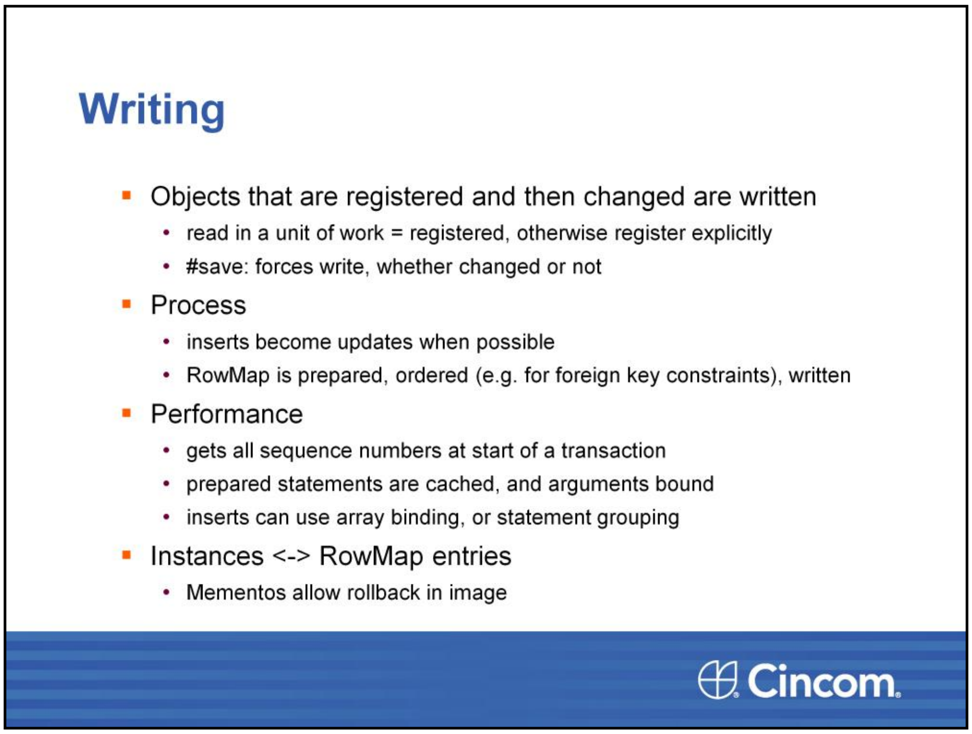


Transaction v. UnitOfWork (2)

- #transact:
  - puts UnitOfWork inside Transaction, commits/rolls-back both, paired
- #commitUnitOfWork (or #commitUnitOfWorkAndContinue)
  - creates and commits a transaction if none is present
  - does not commit if a transaction is present
- #doDDLOperation:
  - for table creation, deletion, alteration; some databases require a transaction in those cases, others do not
    - (and SQLServer does sometimes but not always :-/)

Writing is transactionally-controlled; no explicit write function.

The #modify:in: method is a shortcut for both doing a #transact: and registering an object. You can use #registerAsOld: method that can skip the #isNew: check.



Writing

- Objects that are registered and then changed are written
  - read in a unit of work = registered, otherwise register explicitly
  - #save: forces write, whether changed or not
- Process
  - inserts become updates when possible
  - RowMap is prepared, ordered (e.g. for foreign key constraints), written
- Performance
  - gets all sequence numbers at start of a transaction
  - prepared statements are cached, and arguments bound
  - inserts can use array binding, or statement grouping
- Instances <-> RowMap entries
  - Mementos allow rollback in image

Cincom

DEMO UnitOfWork using GlorpObjectMappedToImaginaryTableTest>>testWrite with break in preCommit, show RowMap.

We have done work to improve the range of making inserts be updates (8.0).

Performance:

=========

Get all sequence numbers at the beginning of a transaction (except for identity column DBs)

Prepared statements are cached, and arguments bound

For inserts, can use array binding, or grouping of statements depending what the database supports.

Contact info

- Glorp

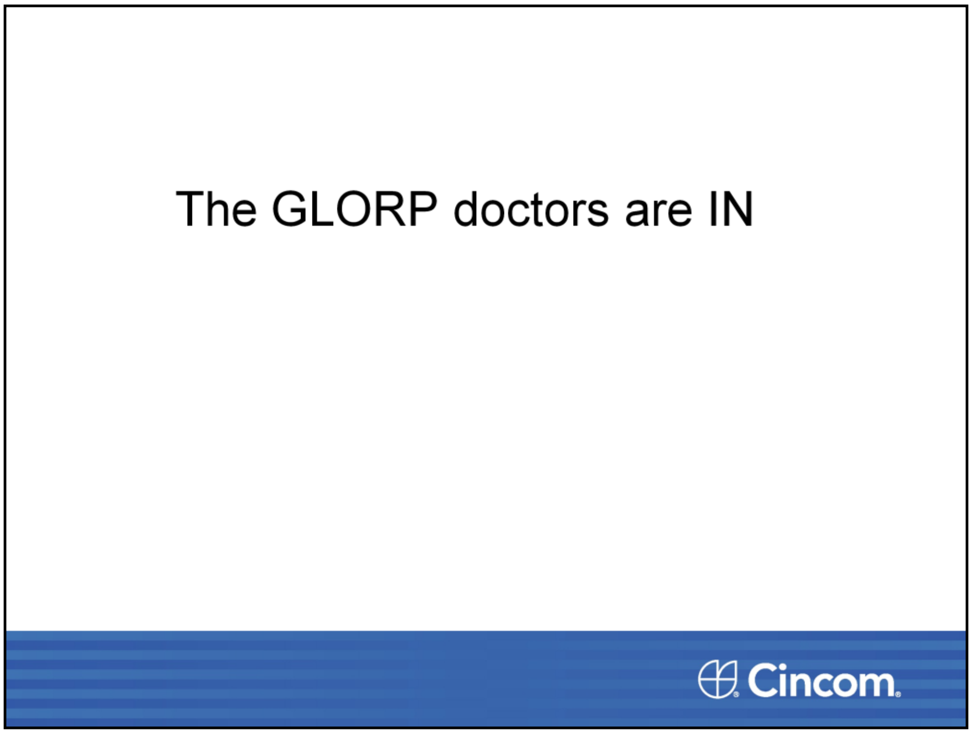
nross@cincom.com	Glorp team
dwallen@cincom.com	Glorp team
trobinson@cincom.com	Major internal customer

- Star Team (Smalltalk Strategic Resources)

sfortman@cincom.com	Smalltalk Director
athomas@cincom.com	Smalltalk Product Manager
jjordan@cincom.com	Smalltalk Marketing Manager

- http://www.cincomsmalltalk.com

Cincom



The GLORP doctors are IN

Cincom

Things We did not have Time for.

Sequences (also VWPoolingDatabaseAccessor).

Cache Policies: several policies available - Keep forever, Timed Expiry, Weak References (but with strong subset), and Expiring proxies. Or you can force cache expiration, either for individual objects, or an entire cache.

Filtered Reads (also mention PhantomMappings).

It is also possible to use anySatisfyJoin: in place of anySatisfy:. The resulting SQL will be slightly different, including an INNER JOIN.

Gaps in Glorp: Stored procedures, Meaningful exceptions, Thread safety, Connection pooling, Nested units of work, Performance tuning, Tools, Documentation, Validation, Error Messages

SelectCommand blockFactor