



# How and Where in GLORP: GLORP Tutorial Reference

Slides with **Orange Titles**: ESUG 2014 Tutorial.

Slides with **Blue Titles**: additional GLORP info.

Niall Ross, Cincom

GLORP in Cincom® VisualWorks®  
(assist Cincom® ObjectStudio®)



World Headquarters  
Cincinnati, Ohio

SIMPLIFICATION THROUGH INNOVATION™

Welcome  
August 18, 2014

# 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

# Glorp Documentation

- 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](http://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

And last, but not least ...

# ... my ESUG 2013 Talk

- 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:
  - Follow-up sessions during the conference

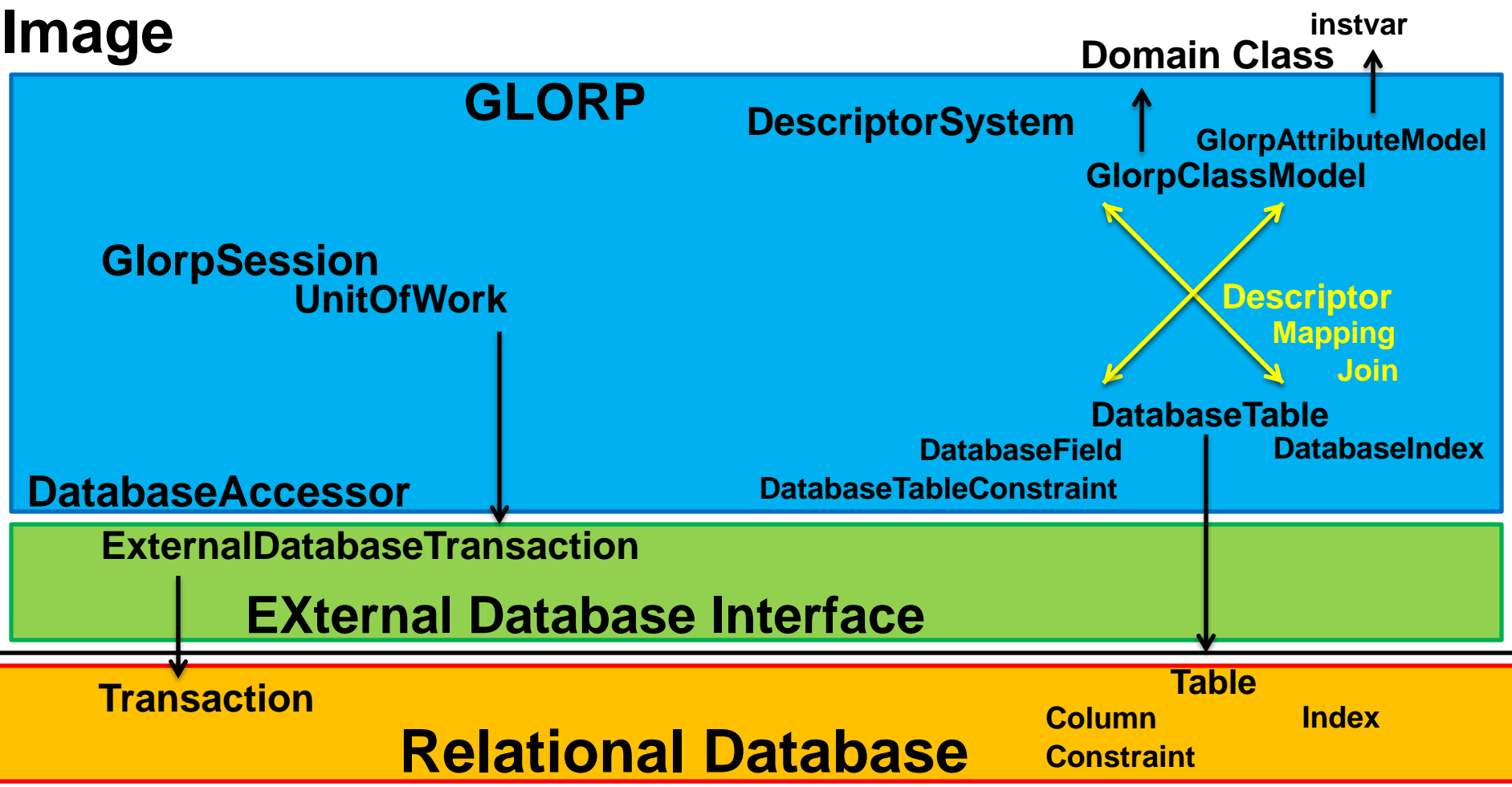
# Example of using Glorp in VisualWorks

- 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.
```

# GLORP Architecture

Image



# 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

# Subclass DescriptorSystem to model ...

- those (parts of) Smalltalk classes to persist
  - classModelFor<DomainClassName>:
- the database tables you will write to and read from
  - tableFor<TABLE\_NAME>:
- the mappings (“descriptors”) between the two
  - descriptorFor<DomainClassName>:

The GLORP f/w finds all the ...For<Name>: methods:  
creates parameter, names it, calls method with it



# Template: Class Model

```
classModelFor<DomainClassName>: aClassModel
```

```
  aClassModel
```

```
    newAttributeNamed: #<instvar>
```

```
    type: <InstvarClass>. “e.g. String, Integer”
```

```
  ...
```

# 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

# Template: Table

tableFor<TABLE\_NAME>: aTable

aTable

createFieldNamed: '<column\_name>'

type: platform <type>. "e.g. varchar: 255"

...

# 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:

## 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`)

# Template: Descriptor

descriptorFor<ClassName>: aDescriptor

table := self tableNamed: '<TABLE\_NAME>'.  
aDescriptor directMapping

from: #<instvar>

to: (table fieldNamed: '<column\_name>').

...

# 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

# 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



# 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”

# Template: Mapping with Join

aDescriptor oneToOneMapping

attributeName: #<name>;

join: (Join

from: (<table1> fieldNamed: '<columnA>')

to: (<table2> fieldNamed: '<columnB>')

...);

# 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

# 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: #fieldNameed:  
#parameter: #noneSatisfy: #getConstant: #count: #sum: #min: #max:  
#average: #sqlSelect: #includes: #aggregate:as:
  - Named attributes
  - Relationships

# 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)

# 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)

# Template: Session Querying

aClass := aGlorpSession

readOneOf: <Class> where: aOneArgBlock.

someClasses := aGlorpSession

read: <Class> where: aOneArgBlock ...

(see GlorpSession protocol 'api/queries')

# 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
    - use #read: not #readManyOf: or #returningManyOf:
- Like Seaside, utility protocol plus cascades
  - #read:limit: #read:where:limit: #read:orderBy: #read:where:orderBy:  
#count: #count:where:



# 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

# 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)

# 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

# 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.

# 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

```
packageldsOfInterest := #(25 36).
```

```
query := Query read: StoreMethodInPackage where:
```

```
[:each | (each package id in: packageldsOfInterest) AND: [each notExists:
```

```
  (Query read: StoreMethodInPackage where:
```

```
    [:mp | mp definition = each definition
```

```
    AND: [mp package id notIn: packageldsOfInterest]]]]].
```

```
query retrieve: [:each | each definition id distinct].
```

# Template: Session Writing

session

beginUnitOfWork;

register: obj;      “obj is a domain instance”

commitUnitOfWork.

session inUnitOfWorkDo: [session register: obj].

(see GlorpSession protocol ‘api/transactions’)

# 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)

# 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.



# 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



**© 2013 Cincom Systems, Inc.  
All Rights Reserved  
Developed in the U.S.A.**

CINCOM and the Quadrant Logo are registered trademarks of Cincom Systems, Inc.

All other trademarks belong to their respective companies.

# Contact info

- Glorp
  - [nross@cincom.com](mailto:nross@cincom.com) Glorp team
  - [dwallen@cincom.com](mailto:dwallen@cincom.com) Glorp team
  - [trobenson@cincom.com](mailto:trobenson@cincom.com) Major internal customer
- Star Team (Smalltalk Strategic Resources)
  - [sfortman@cincom.com](mailto:sfortman@cincom.com) Smalltalk Director
  - [athomas@cincom.com](mailto:athomas@cincom.com) Smalltalk Product Manager
  - [jjordan@cincom.com](mailto:jjordan@cincom.com) Smalltalk Marketing Manager
- <http://www.cincomsmalltalk.com>

The GLORP doctor is IN