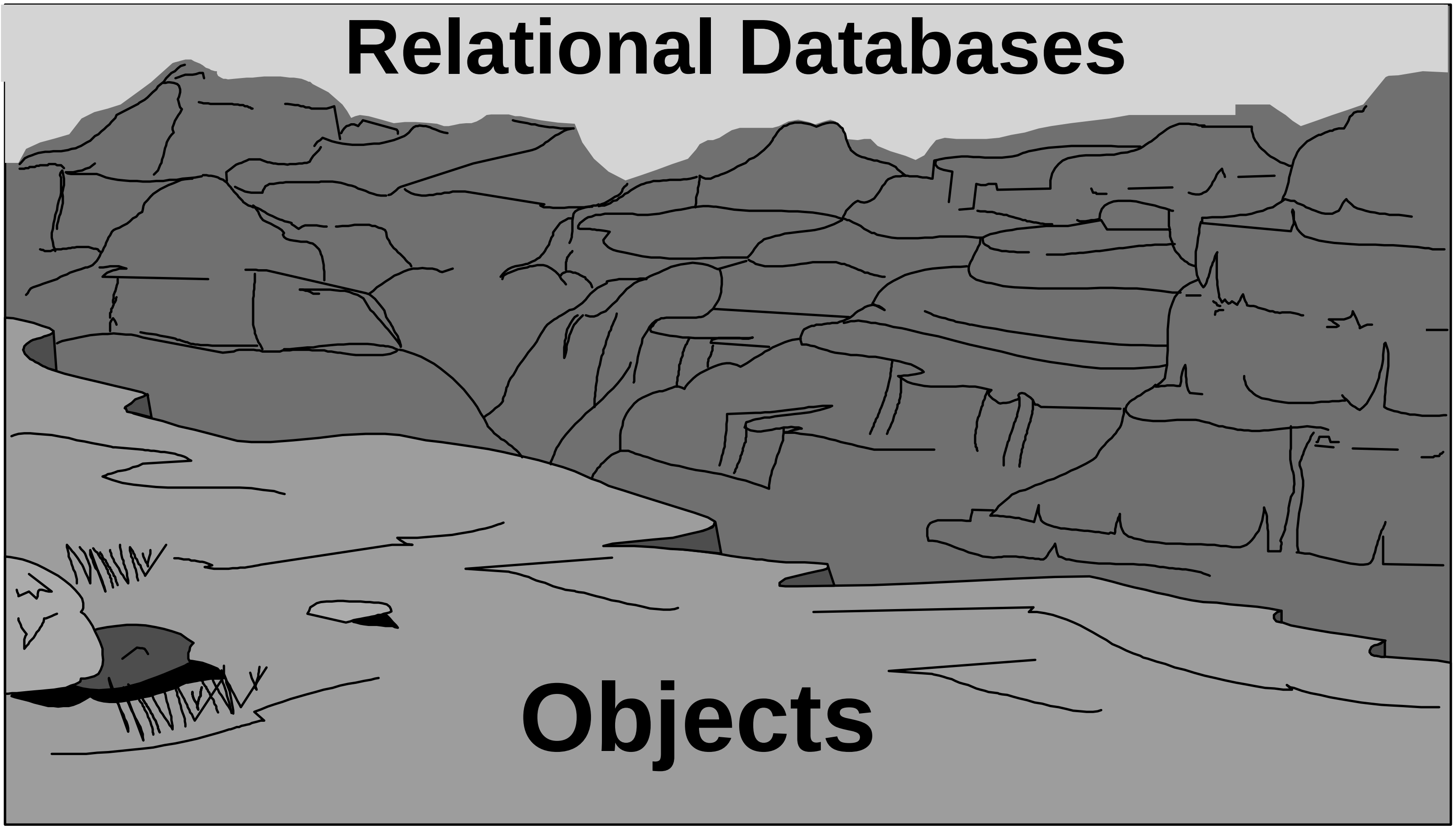
# Crossing Chasms



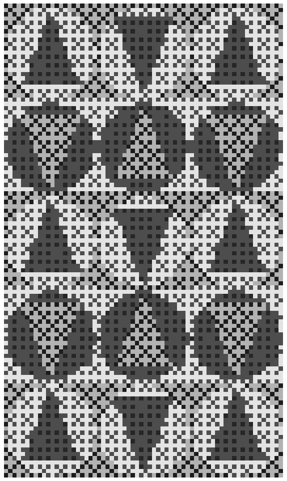
# Crossing Chasms

- Crossing the Chasm from Objects to Relational Databases
- A Presentation in Four Acts

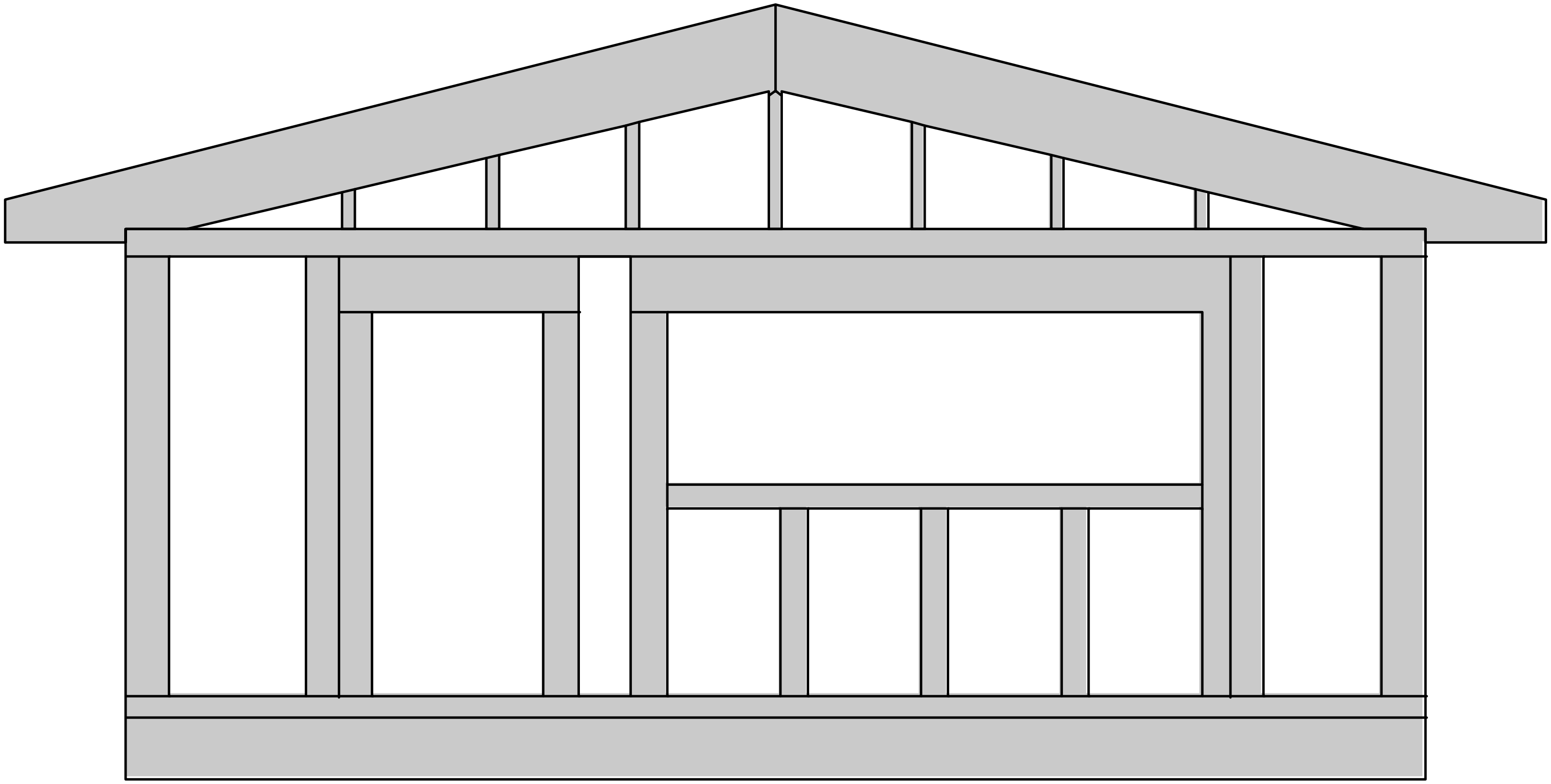
(C) 1996, 1997 Kyle Brown & Bruce Whitenack

# Patterns

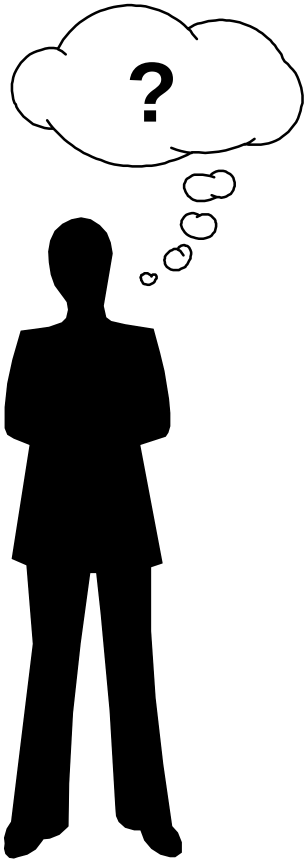
- Crossing Chasms is a pattern language
- A pattern has four parts
  - Context
  - Problem
  - Forces
  - Solution
- We will show you how our patterns are applied to a real problem.



# Act 1: Architectural Aims



# Pattern: Choosing a Database



- Context: You are about to embark upon a new project.
- Problem: You must choose to use either a relational or an object database to provide object persistence.

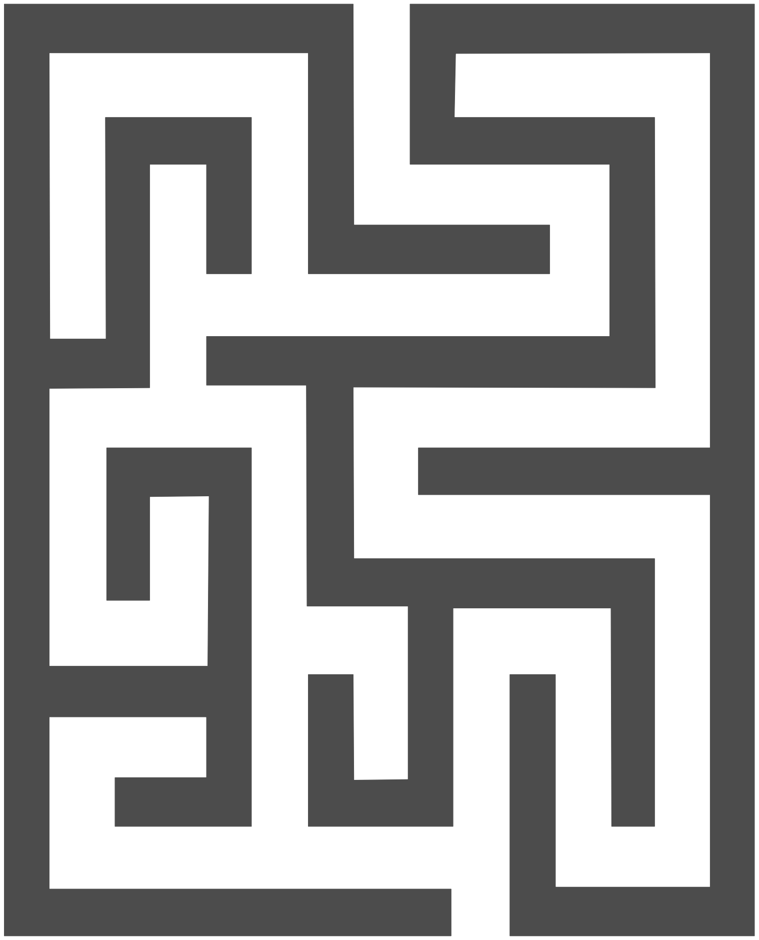
# Choosing a Database (2)

- Forces
  - Cost of Technology
  - Existence of legacy systems and software
  - Cost of Training
  - Fundamental structure of the application to be built

# Choosing a Database (3)

- Solution
  - If you are heavily constrained by legacy code or data, or have a significant investment in relational technology, then choose an RDBMS for object persistence.
  - On the other hand, if these do not apply, or if the structure of your application demands it, an ODBMS will be better.

# Pattern: Four Layer Architecture



- Context: You must have a coherent software architecture.
- Problem: What is the appropriate architecture for an OO client-server system?

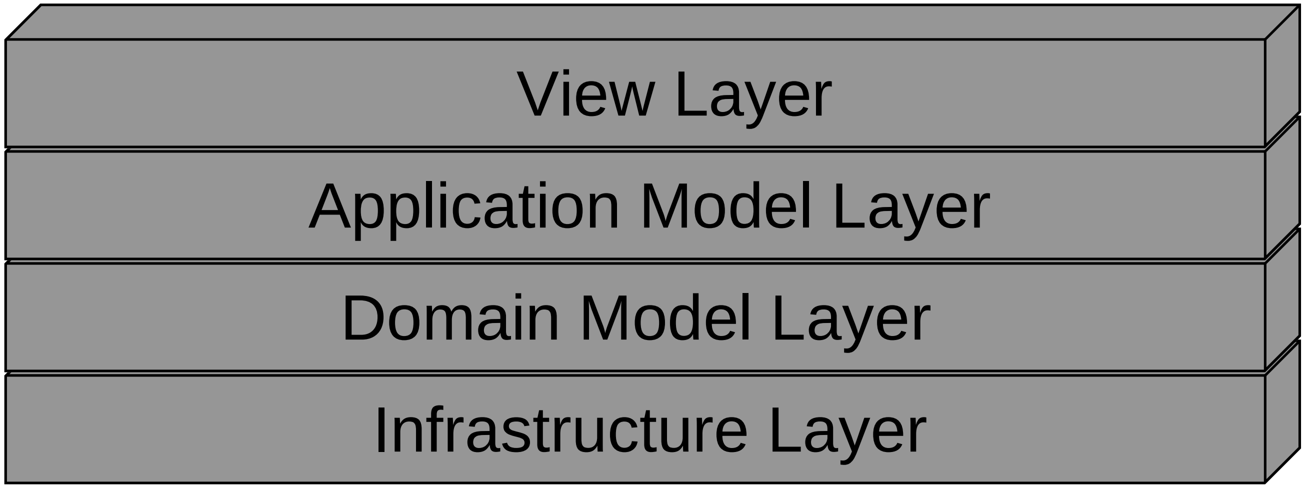
# Four Layer Architecture (2)

- Forces:
  - – portability to new environments, libraries, and tool vendors
  - – rational distribution of work among team members
  - – structure of existing tools and frameworks

# Four Layer Architecture (3)

- Solution
  - Employ a layered architecture with:
    - a View Layer
    - an Application Model layer
    - a Domain layer
    - an Infrastructure layer

# The Four Layers



View Layer

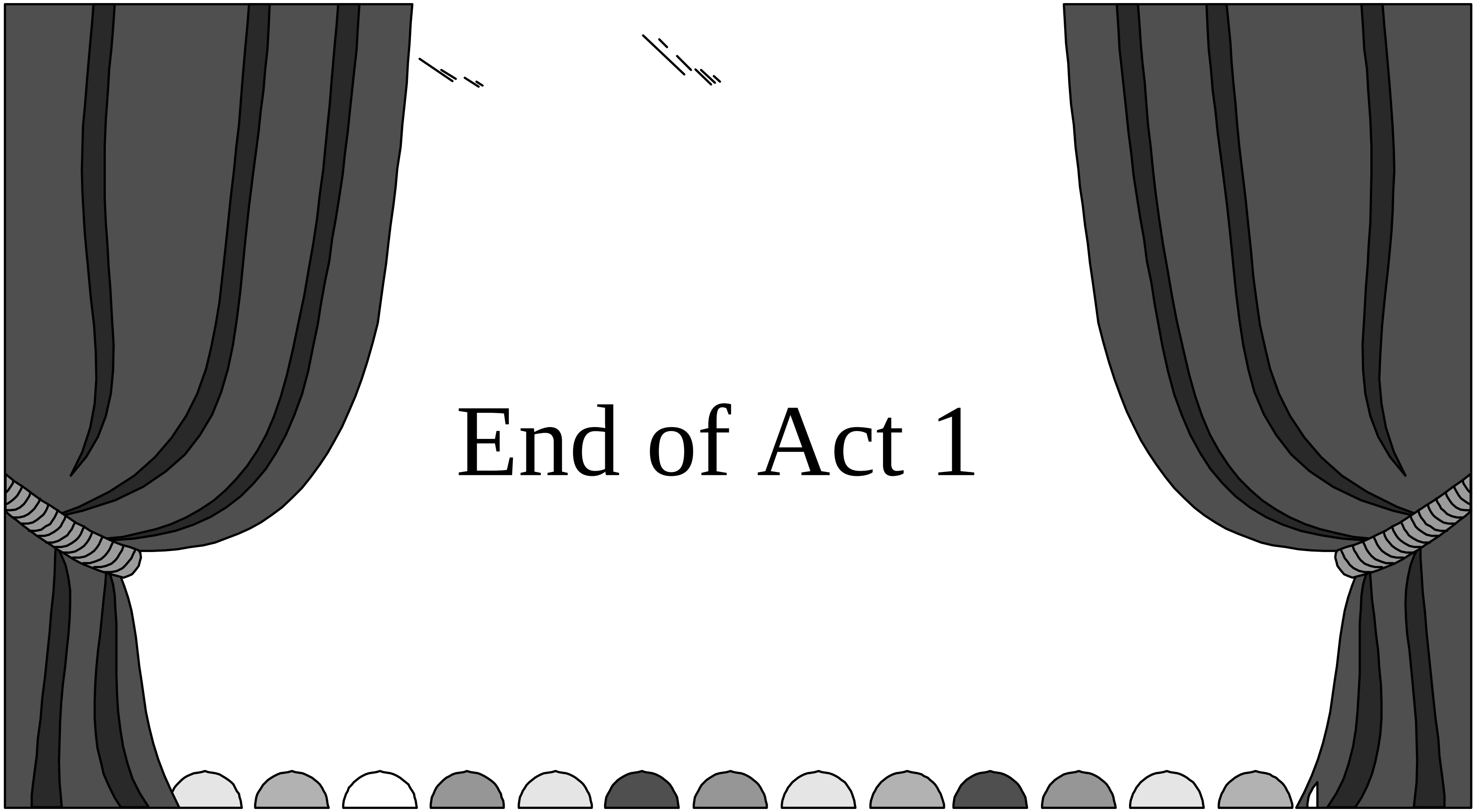
Application Model Layer

Domain Model Layer

Infrastructure Layer

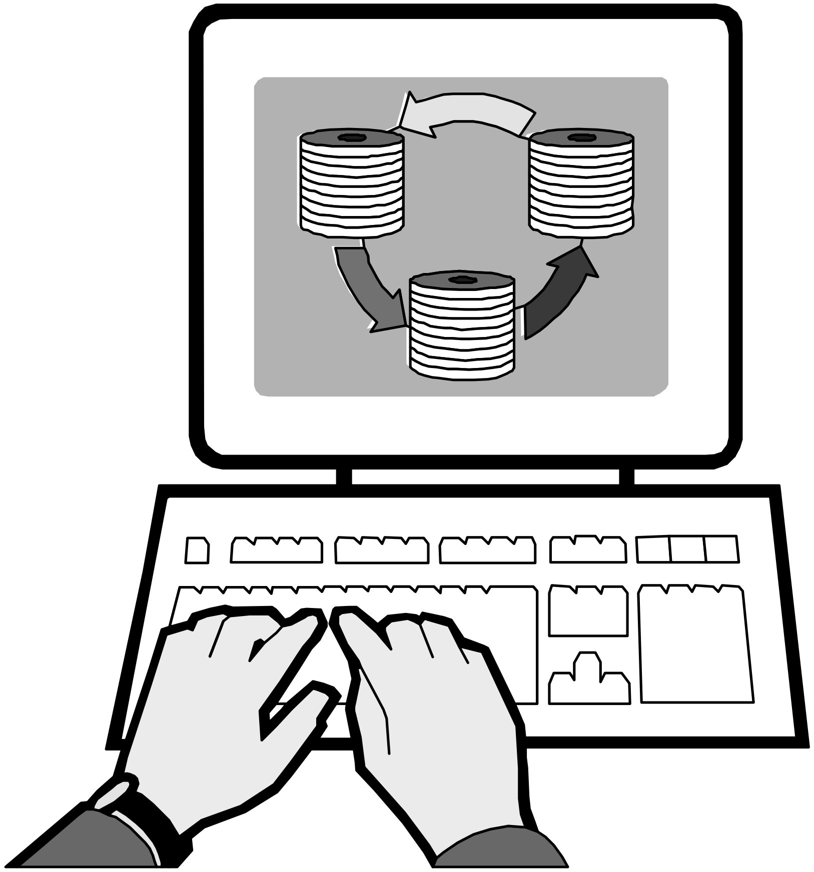
# Layers and Tiers

- In many cases the four layers can map onto “tiers”
- A two-tier system (fat client) places all four layers on a single client machine
- A three-tier system maps the domain and infrastructure layer (and possibly part of the App model layer) onto its own machine, leaving the view layer on the client machines

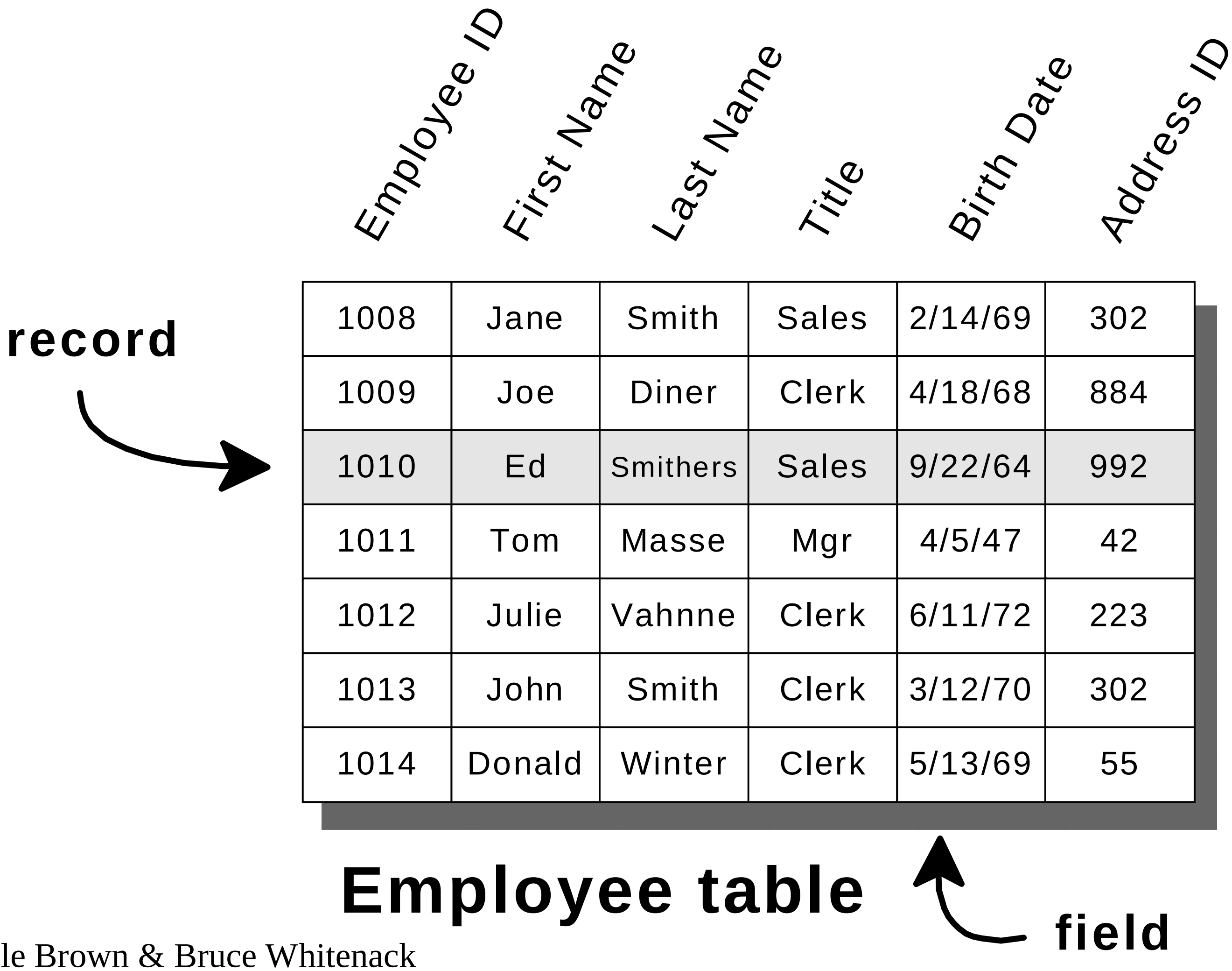


# End of Act 1

# Act 2: Database Schemes and Dreams



# Review of Relational Databases



record

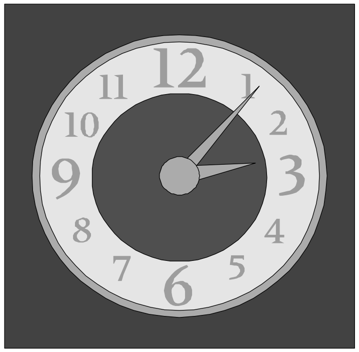
| Employee ID | First Name | Last Name | Title | Birth Date | Address ID |
|---|---|---|---|---|---|
| 1008 | Jane | Smith | Sales | 2/14/69 | 302 |
| 1009 | Joe | Diner | Clerk | 4/18/68 | 884 |
| 1010 | Ed | Smithers | Sales | 9/22/64 | 992 |
| 1011 | Tom | Masse | Mgr | 4/5/47 | 42 |
| 1012 | Julie | Vahnne | Clerk | 6/11/72 | 223 |
| 1013 | John | Smith | Clerk | 3/12/70 | 302 |
| 1014 | Donald | Winter | Clerk | 5/13/69 | 55 |

**Employee table**

field

# Pattern: Table Design Time

- Problem:
  - When is the best time to design your relational database schema during OO development?



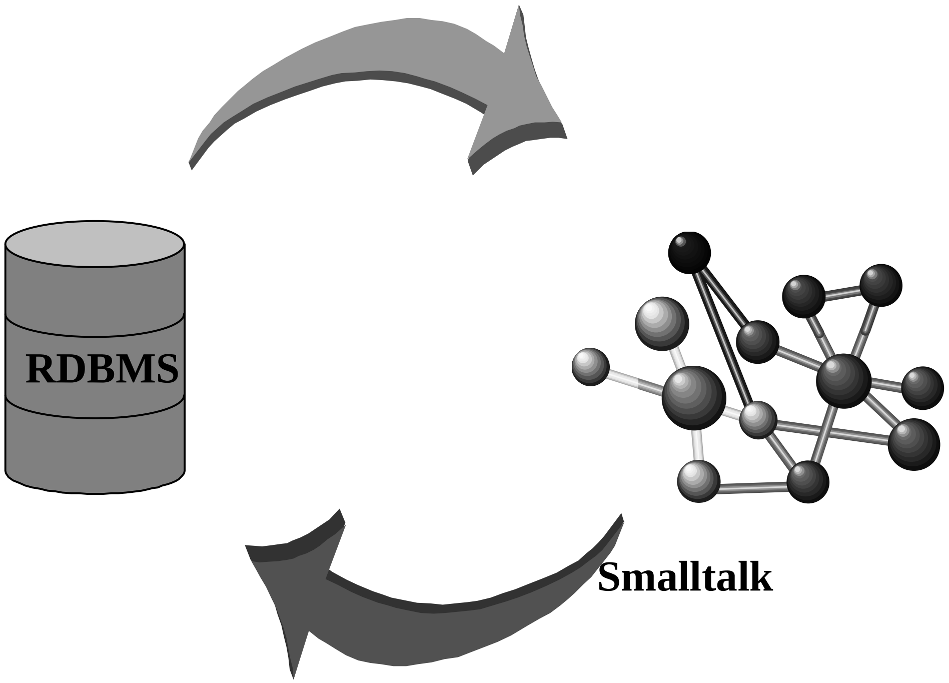
# Table Design Time Considerations

- Avoid data-centric design because it is usually more difficult to maintain
- Performance is dependent on the design of the database.
- Each OO design decision affects the database design. The models must be considered together.

# Table Design Time Solution

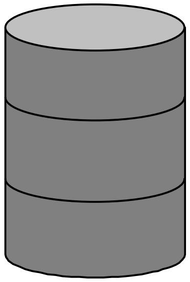
- Design the database schema based on your object model.
- For best results the database should be considered *during* the design of the object model, after an initial object design iteration.
- Incorporate a relational - object database framework soon as possible into your architectural prototype

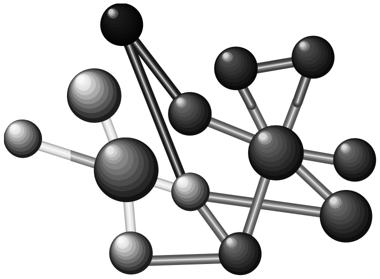
# Pattern: Representing objects as tables



- Problem: How do you map a set of objects into a relational database schema?

# Relational to Object Considerations





Tables define records ↔ Classes define objects

Records consist of fields ↔ Objects consist of attributes

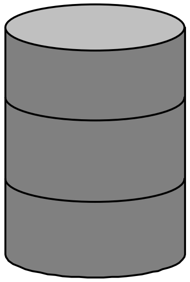
Records reference other records using a foreign key ↔ Objects reference other objects using a pointer

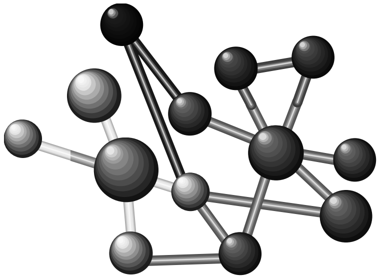
Fields in a table are statically typed ↔ Dynamic binding

Hard to represent complex relationships ↔ Easy to represent complex relationships

Inheritance

# Relational to Object Considerations





- Rows in Tables have keys
- Every row in a table has the same attributes

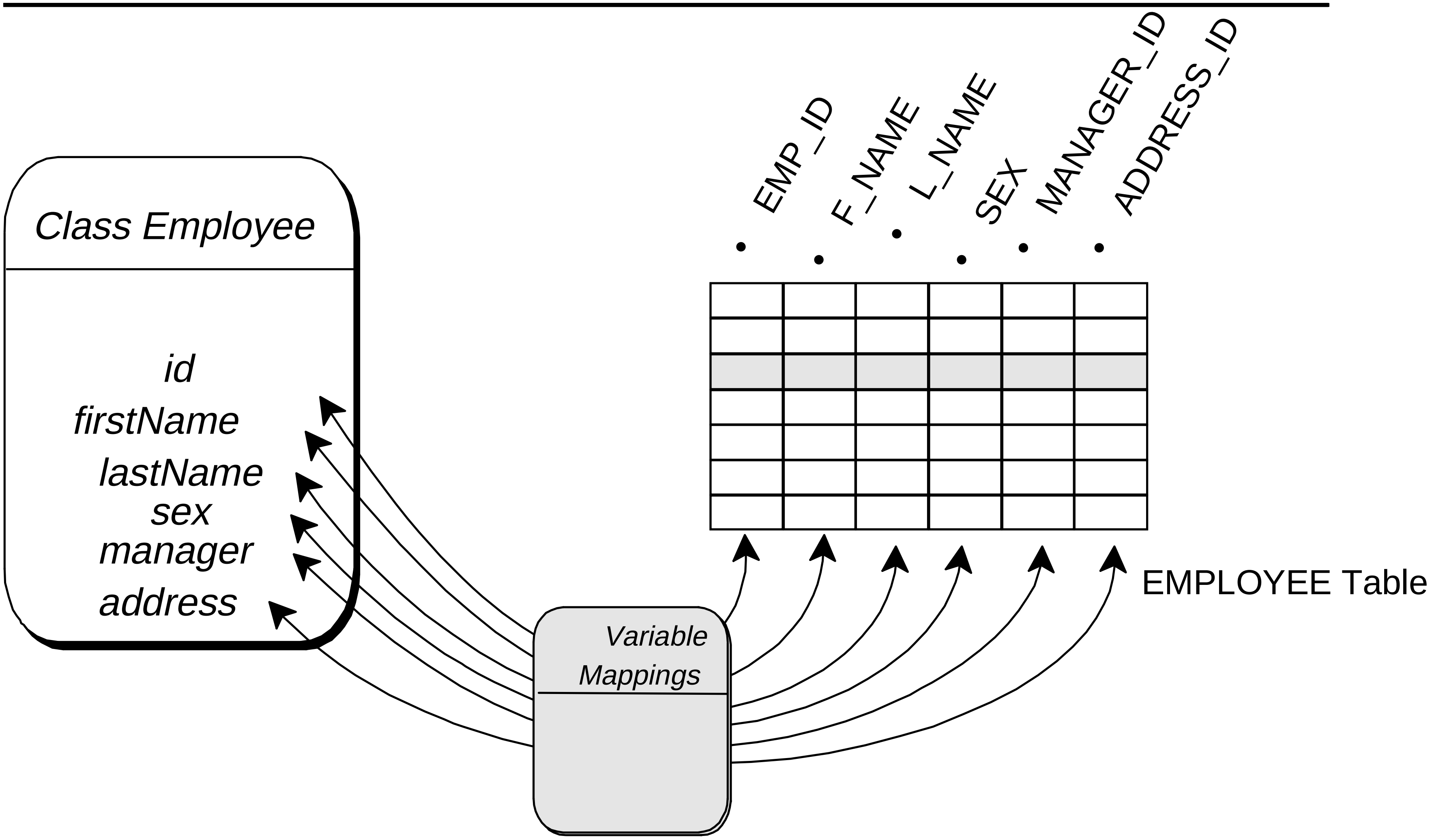
- Objects do not have keys but have object ids
- Objects can be in heterogeneous collections

- Data types do not match between a relational database and Smalltalk

# Solution - Representing Objects in Tables

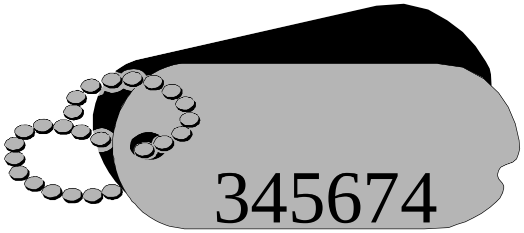
1. Define a table for each persistent class
2. Define columns whose values map directly to the values in the class' instance variables -- i.e. base data types. (String, Number, Date)
3. If the class has object relationships, define columns whose values are foreign key references to the tables that store the referenced objects.

# Initial Table Design



(C) 1996, 1997 Kyle Brown & Bruce Whitenack

# Pattern: Object Identity

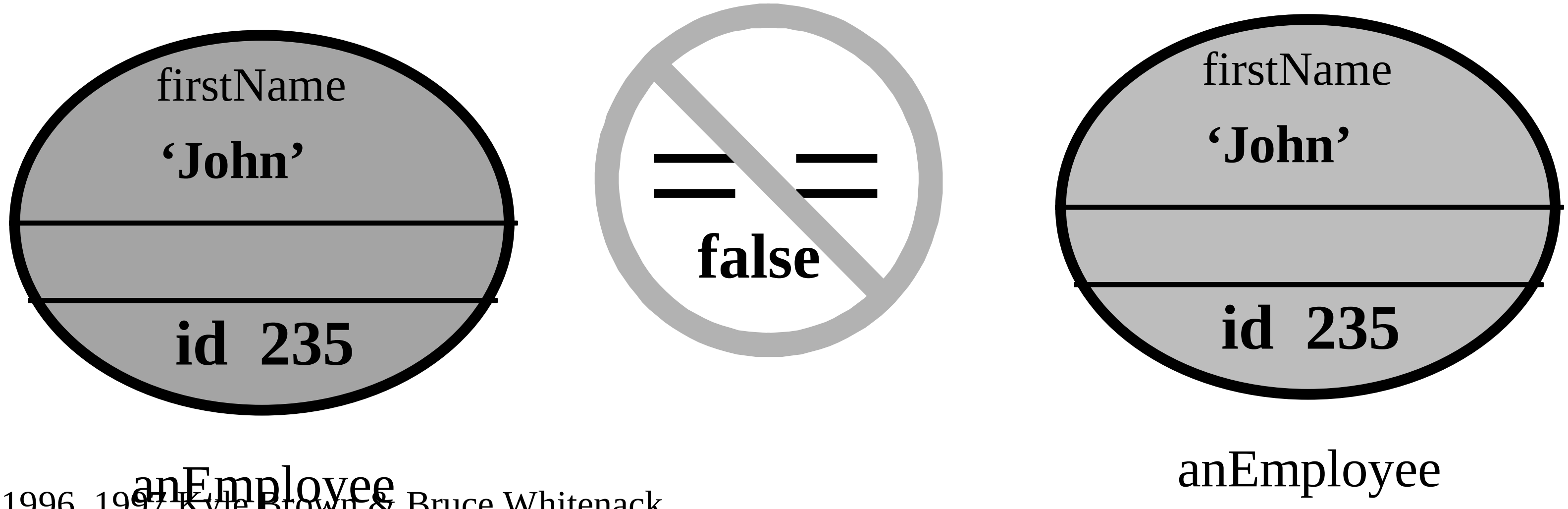


Problem:

- How do you preserve an object’s identity in a relational database?
- If two objects have the same set of attributes but are really different objects how do you keep their uniqueness in the relational database?
- How do you keep from creating multiple copies of the same object each time you read it in?

# Object Identity Considerations

- Each object's uniqueness must be preserved in the database
- There should be no unintended duplicates in the application as a result of reading in the same object twice.



# Solution: Create an Object Identifier

- Ensure each class stores a unique identifier -- i.e. define an *id* instance variable to store unique id for *Employee*
- Define the primary key field in the object’s table to store the unique identifier.
- Sequence number generation can  create unique ids for each object…create a sequence table if necessary
- Use an identity map (cache) keyed on the identifier which points to the instance of the object in the image. Prevents unwarranted duplicates during reads.

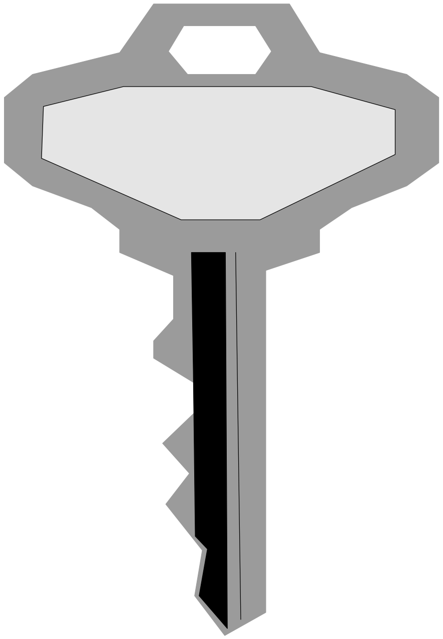
# Being unique

Sequence Numbers maintain uniqueness among Employee objects

| EMP_ID | F_NAME | L_NAME | BIRTH_DATE |
|---|---|---|---|
| 7 | Jeff | Jones | 9/11/70 |
| 8 | Liz | Taylor | 10/06/59 |
| 9 | Mary | Peters | 08/05/49 |

EMPLOYEE TABLE (partial)

# Pattern: Foreign Key Reference



- Problem: How do you represent objects that reference other objects that are not “base data types”?

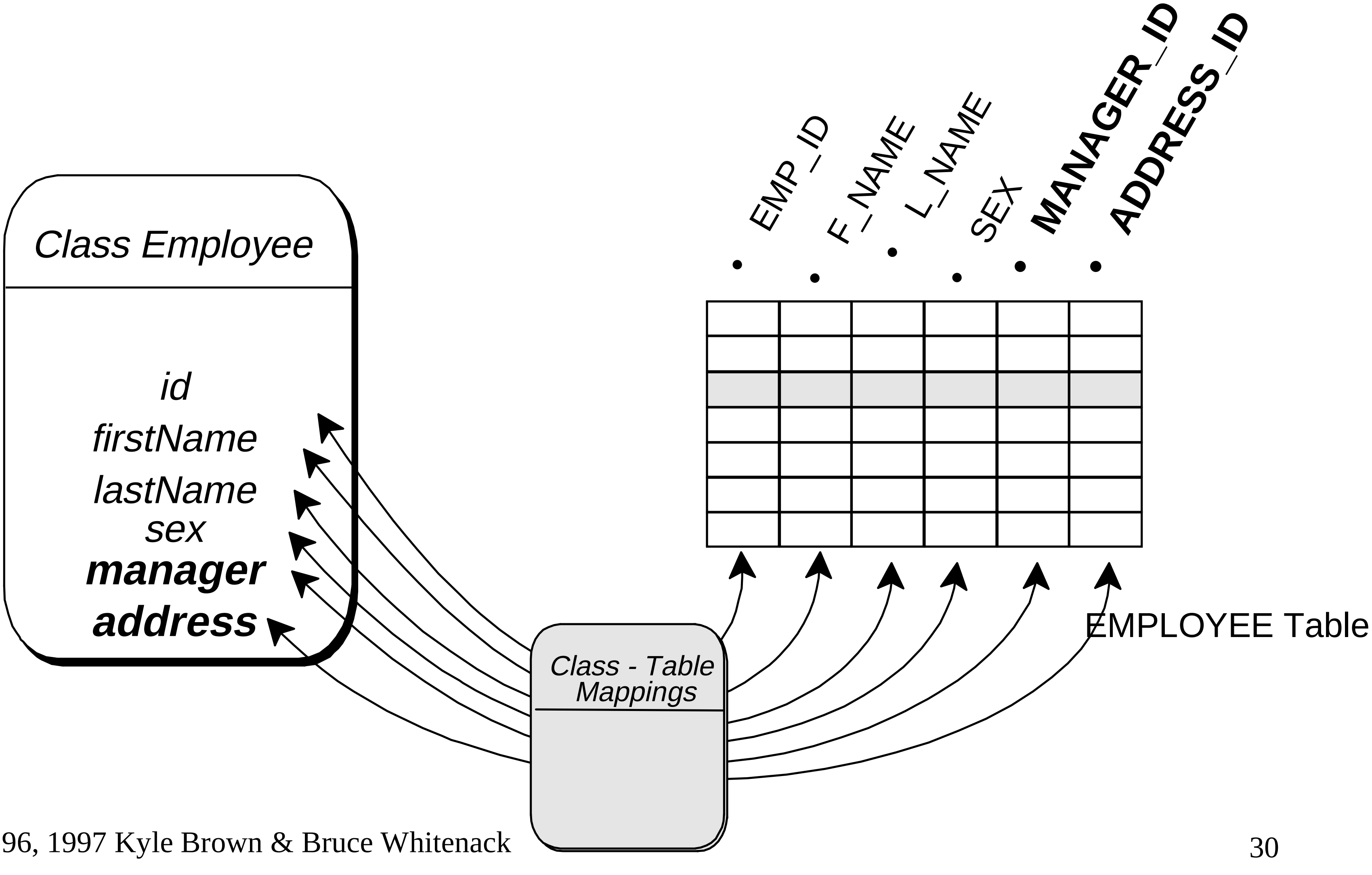
For example,  an employee can have a reference to an address object...

  – Note: A base data type refers to a database data type like CHAR This maps to a standard class like String .

# Solution: Foreign Key Reference

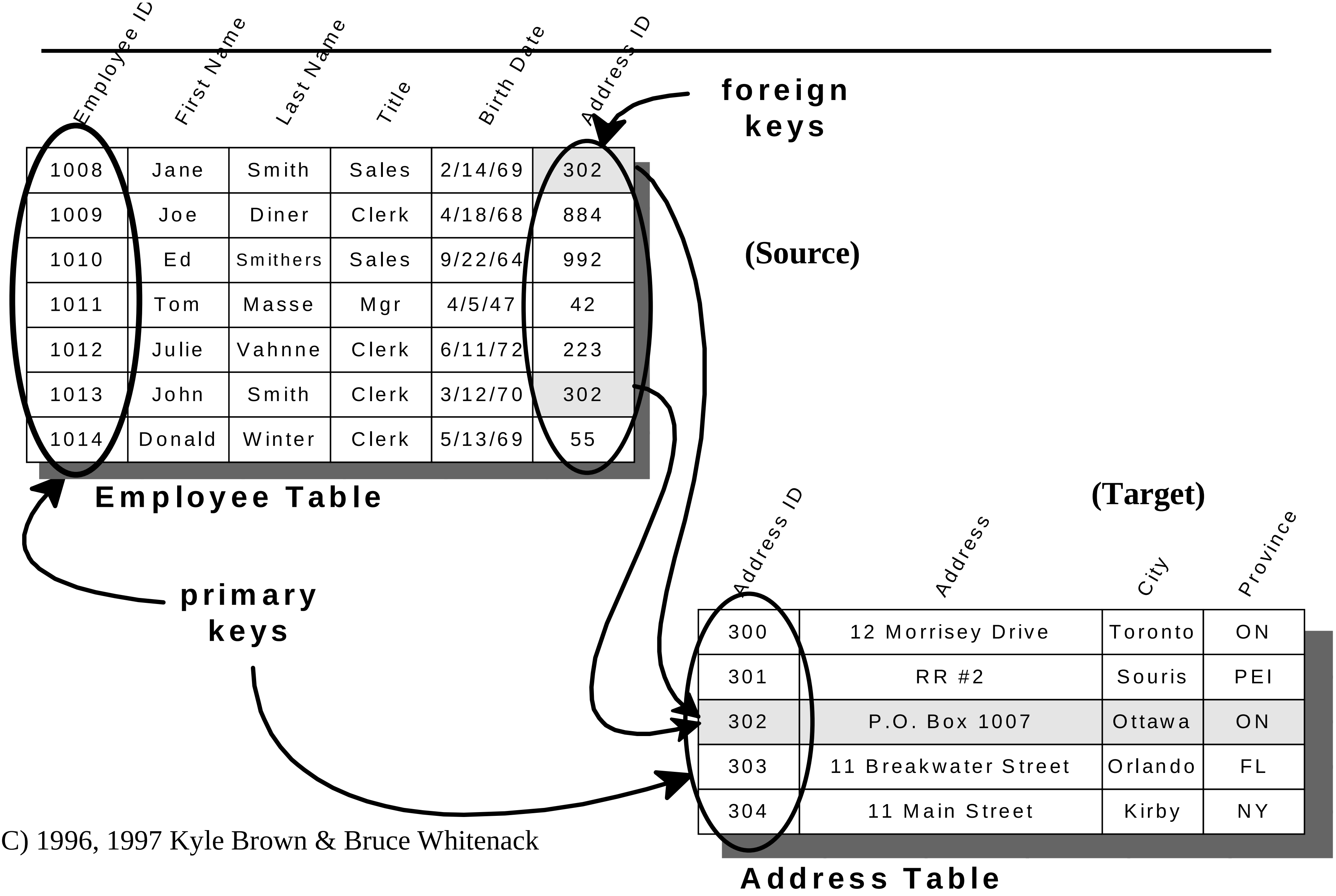
- Assign each object a unique object identifier (OID)
- Add a column for each instance variable that is not a base datatype or a collection.
- In that column store the OID of the referenced object
- Declare the column as a foreign key

# Foreign Key Column



(C) 1996, 1997 Kyle Brown & Bruce Whitenack

# The Keys to Consider



foreign keys

(Source)

| Employee ID | First Name | Last Name | Title | Birth Date | Address ID |
|---|---|---|---|---|---|
| 1008 | Jane | Smith | Sales | 2/14/69 | 302 |
| 1009 | Joe | Diner | Clerk | 4/18/68 | 884 |
| 1010 | Ed | Smithers | Sales | 9/22/64 | 992 |
| 1011 | Tom | Masse | Mgr | 4/5/47 | 42 |
| 1012 | Julie | Vahnne | Clerk | 6/11/72 | 223 |
| 1013 | John | Smith | Clerk | 3/12/70 | 302 |
| 1014 | Donald | Winter | Clerk | 5/13/69 | 55 |

**Employee Table**

(Target)

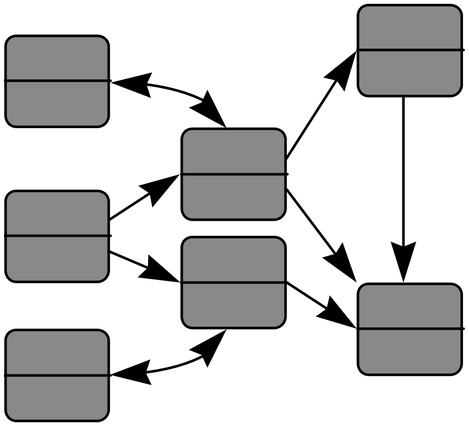
primary keys

| Address ID | Address | City | Province |
|---|---|---|---|
| 300 | 12 Morrisey Drive | Toronto | ON |
| 301 | RR #2 | Souris | PEI |
| 302 | P.O. Box 1007 | Ottawa | ON |
| 303 | 11 Breakwater Street | Orlando | FL |
| 304 | 11 Main Street | Kirby | NY |

**Address Table**

# Pattern: Representing object relationships

- Problem: How do you represent object relationships in a relational database?
- Context
  - –An object model is built with a number basic relationships:
    - 1 to 1
    - 1 to many
    - many to many



(C) 1996, 1997 Kyle Brown & Bruce Whitenack

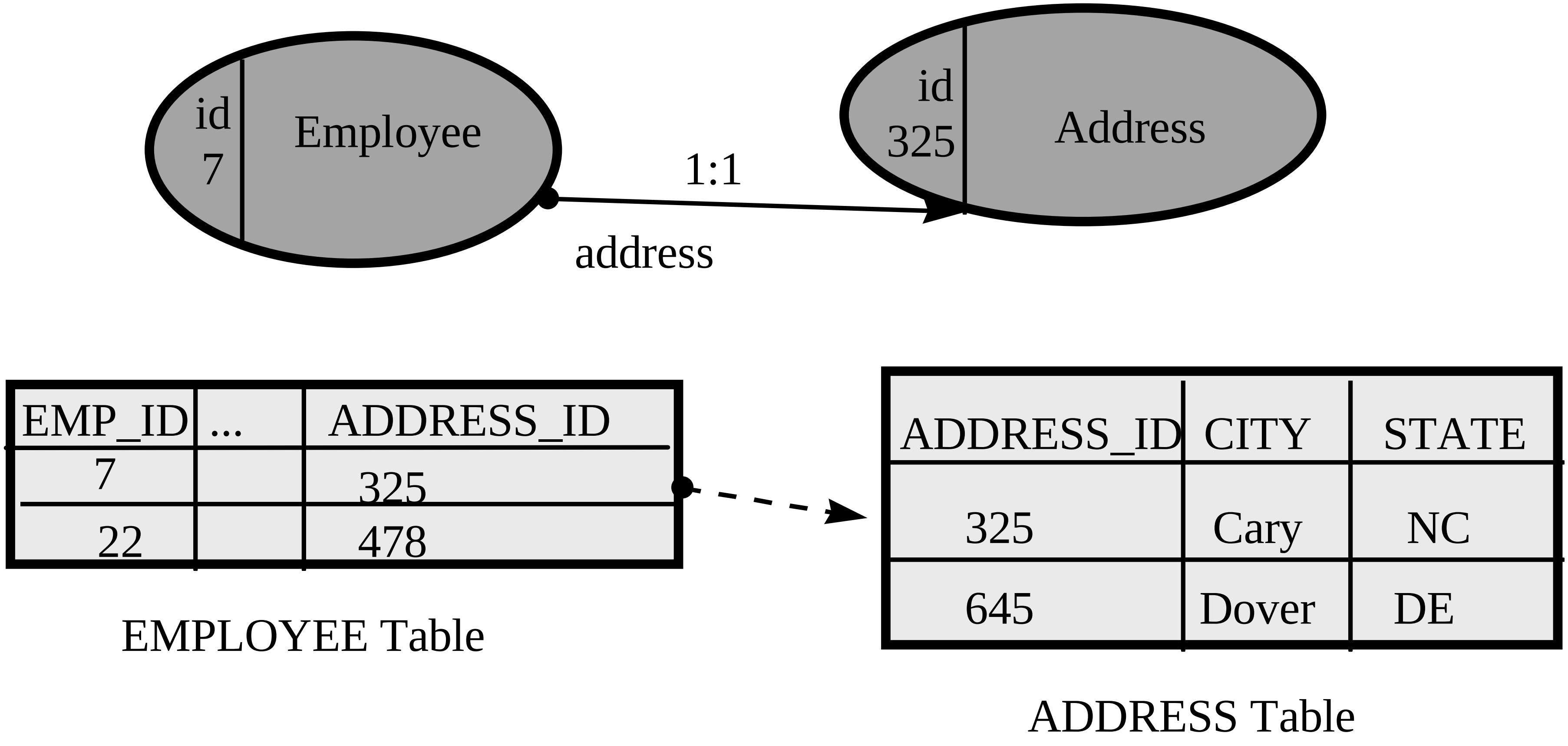
# Representing object relationships - Considerations

- In the object domain, the object always references it parts,
  - …whereas in the relational domain, in cases such as 1 to many relationship, each part references its owner
    - each employee has a key back to the manager (owner)
- In source object point of view, there is no many to many mapping, just 1 to many
    - Example: each messnger has pick-ups (1 to many), and a pick-up may have more than 1 messenger (1 to many) -- many to many.

# Solution: Representing object relationships

- Determine the types of relationships between objects
- Design table(s) corresponding to each domain object using the following guidelines:
  - **1 to 1** relationship uses ***foreign keys in the source table***
  - **1 to many** relationship uses ***foreign keys*** *from* ***each*** *of the* ***many*** *in the target table* to the one ‘parent’ record in the source table.
  - **many to many** relationship use a ***relationship table*** *with* ***foreign keys*** *that reference each of the related records in both tables*

# One-to-One Mappings



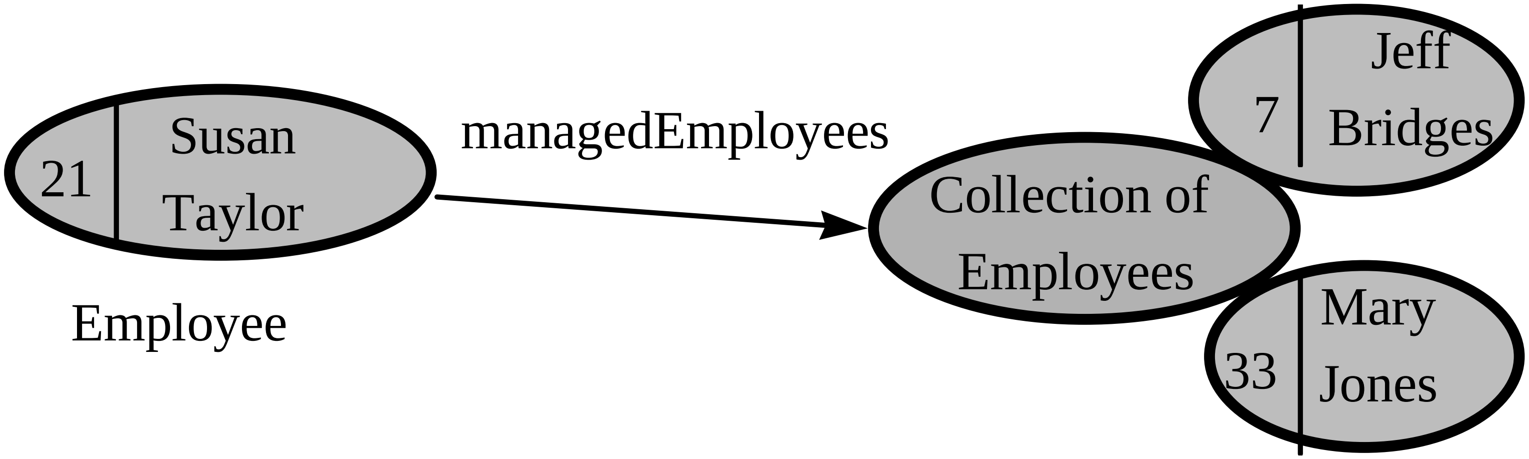
| EMP_ID | ... | ADDRESS_ID |
|---|---|---|
| 7 |  | 325 |
| 22 |  | 478 |

EMPLOYEE Table

| ADDRESS_ID | CITY | STATE |
|---|---|---|
| 325 | Cary | NC |
| 645 | Dover | DE |

ADDRESS Table

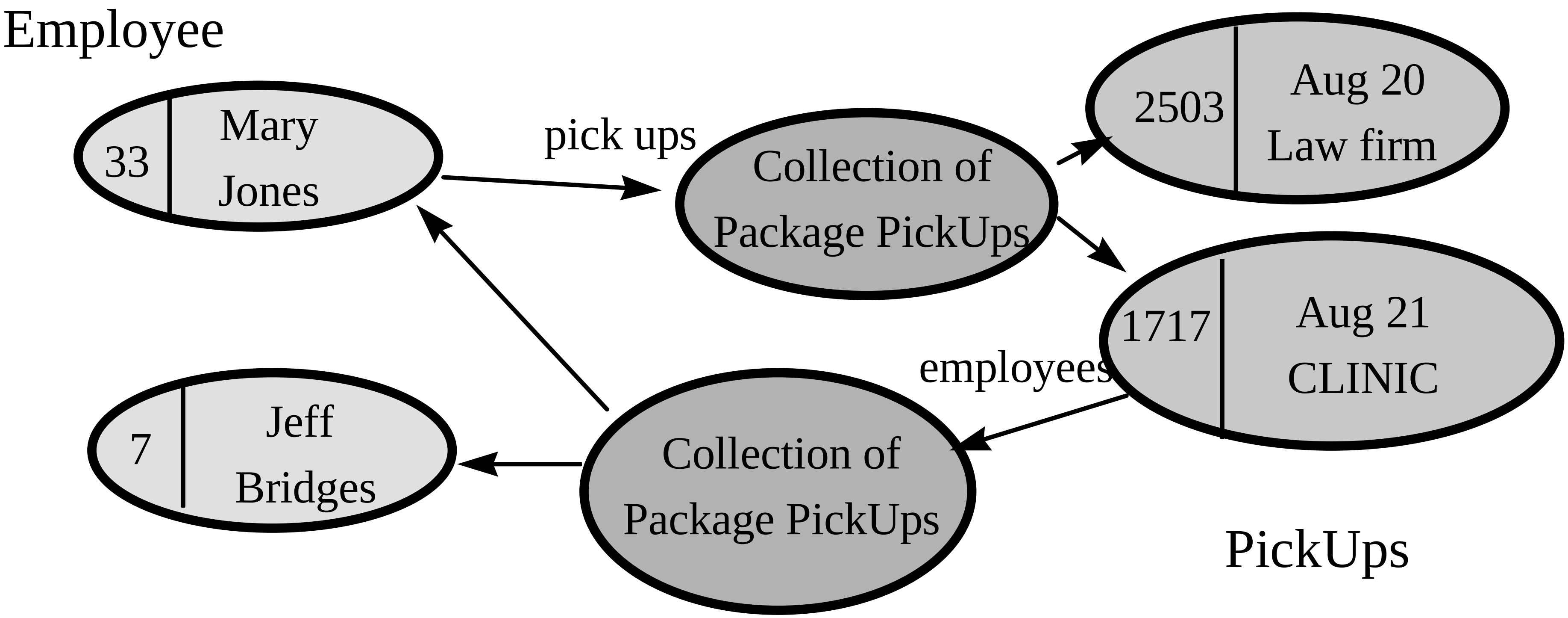
# One-to-Many Mapping



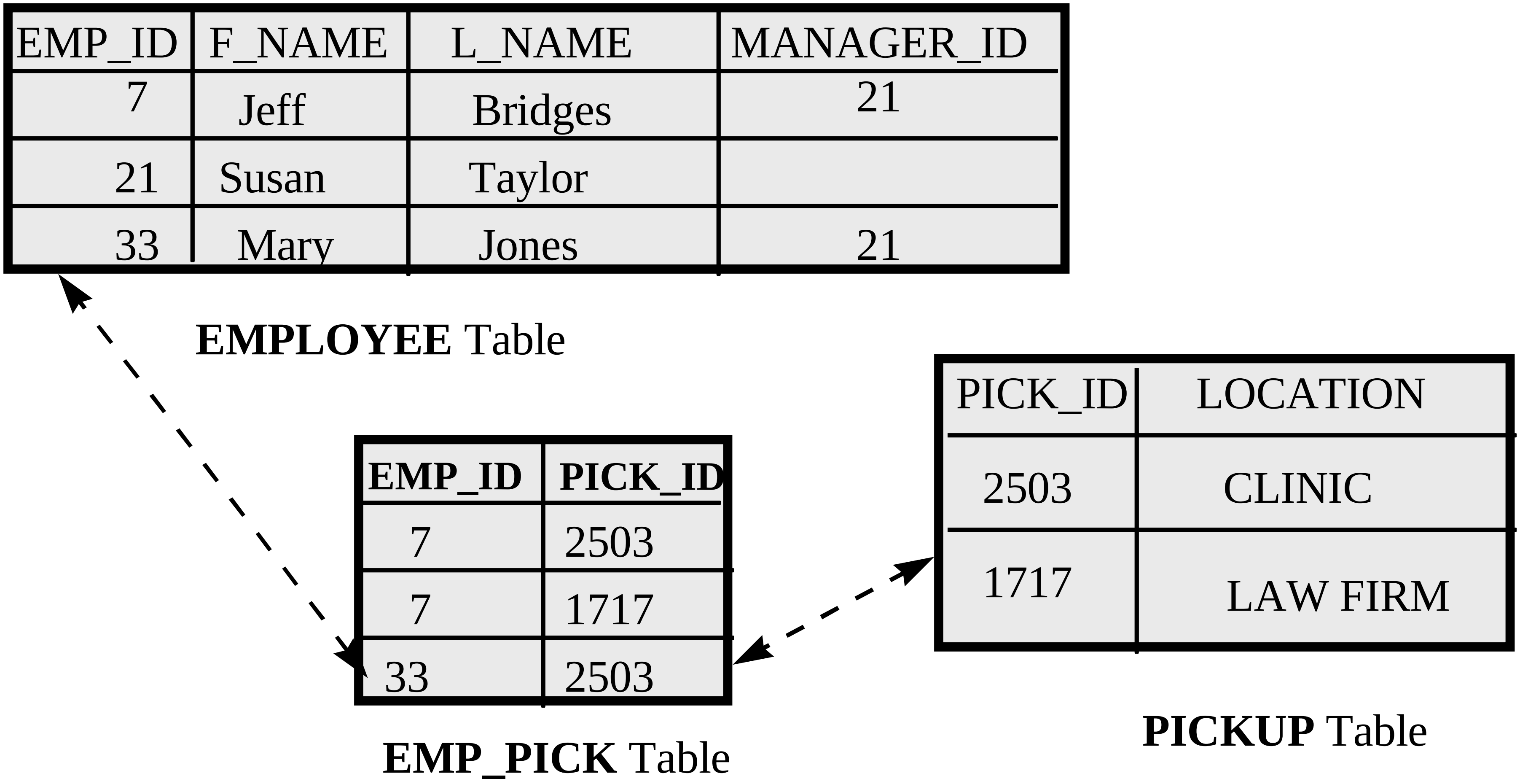
| EMP_ID | F_NAME | L_NAME | MANAGER_ID |
|---|---|---|---|
| 7 | Jeff | Bridges | 21 |
| 21 | Susan | Taylor | |
| 33 | Mary | Jones | 21 |

EMPLOYEE Table

# Many-to-Many Mappings



# Many to Many with Relationship Table



| EMP_ID | F_NAME | L_NAME | MANAGER_ID |
|---|---|---|---|
| 7 | Jeff | Bridges | 21 |
| 21 | Susan | Taylor | |
| 33 | Mary | Jones | 21 |

**EMPLOYEE** Table

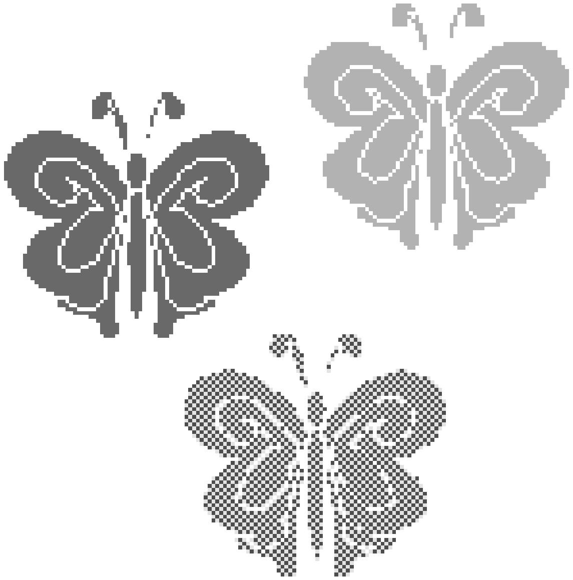
| **EMP_ID** | **PICK_ID** |
|---|---|
| 7 | 2503 |
| 7 | 1717 |
| 33 | 2503 |

**EMP_PICK** Table

| PICK_ID | LOCATION |
|---|---|
| 2503 | CLINIC |
| 1717 | LAW FIRM |

**PICKUP** Table

# Pattern: Representing Special Collections



- Problem: How do you represent special, (i.e.heterogeneous, ordered) collections in a relational database?

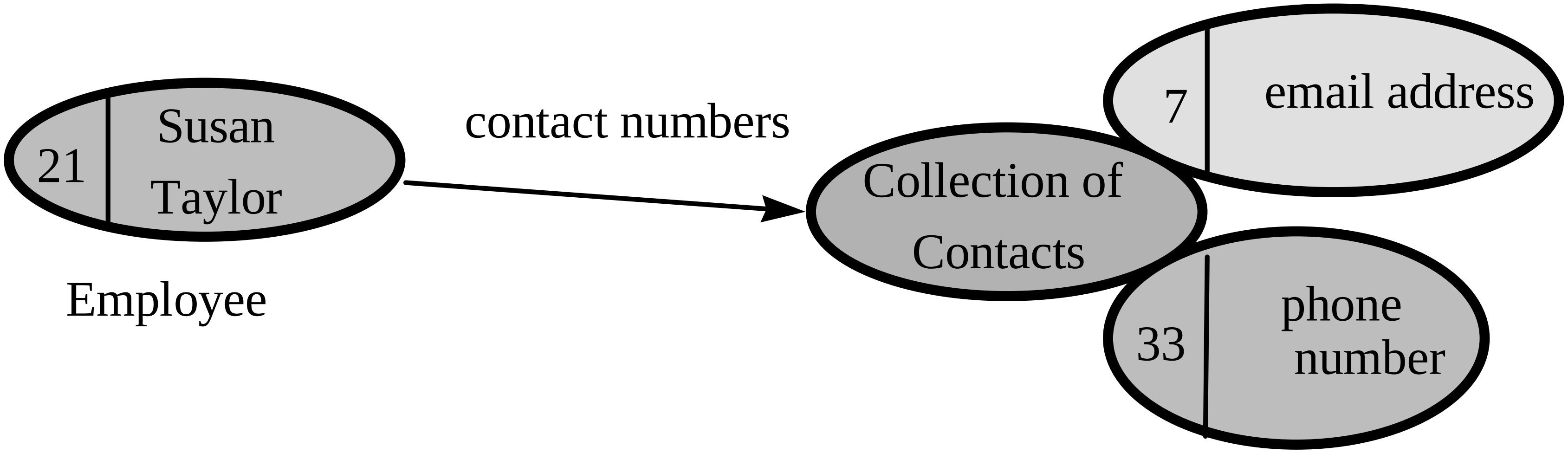
# Representing Special Collections (2)

- Forces
  - 1NF rule
  - Objects may be contained in many collections (M-N relationships)
  - Collections can be heterogenous - members can be of different classes
  - Collections can be ordered

# Representing Special Collections (3)

- Solution
  - Create a relationship table for each collection.  A relationship table maps the primary keys of the containing objects to the primary keys of the contained objects
  - The relationship stores other information
    - class of contained object
    - ordering information

# Heterogeneous Collection



# Heterogeneous Collection

| EMP_ID | F_NAME | L_NAME | .. |
|---|---|---|---|
| 7 | Jeff | Bridges | .. |
| 21 | Susan | Taylor | |
| 33 | Mary | Jones | .. |

**EMPLOYEE** Table

EMAIL

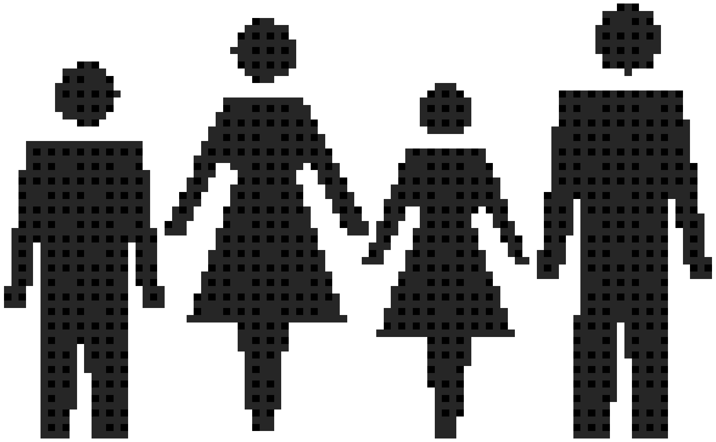
| ID | USER | DOMAIN |
|---|---|---|
| 17 | staylor | jewels.com |
| 29 | bwhiten | op.com |

| EMP_ID | CONT_ID | CON_TYPE |
|---|---|---|
| 21 | 17 | EMAIL |
| 21 | 69 | PHONE |
| 7 | 21 | PHONE |

PHONE

| ID | AREA_CODE | NUMBER |
|---|---|---|
| 21 | 919 | 345-5678 |
| 69 | 212 | 567-7896 |

# Pattern: Representing Inheritance



- Problem: How do you represent a set of classes in an inheritance hierarchy in a relational database?

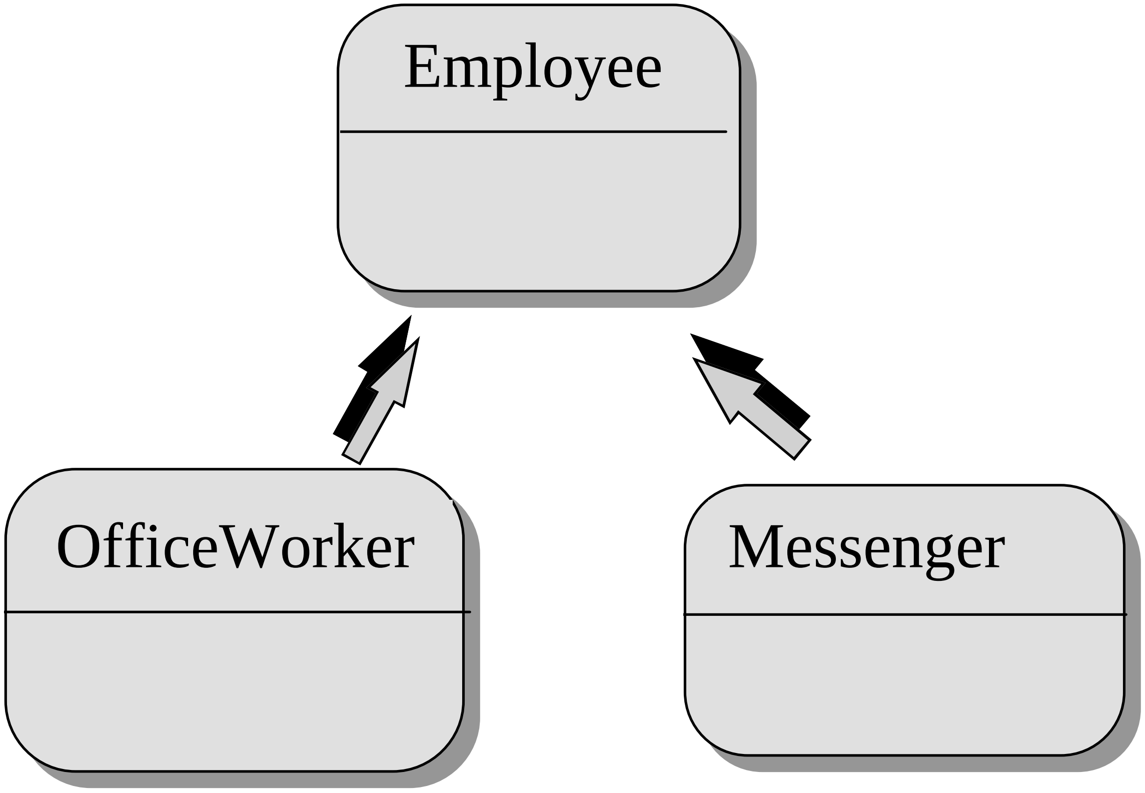
# Representing Inheritance

- Forces
  - Relational databases don’t provide support for inherited attributes
  - Object designs are rife with inheritance...

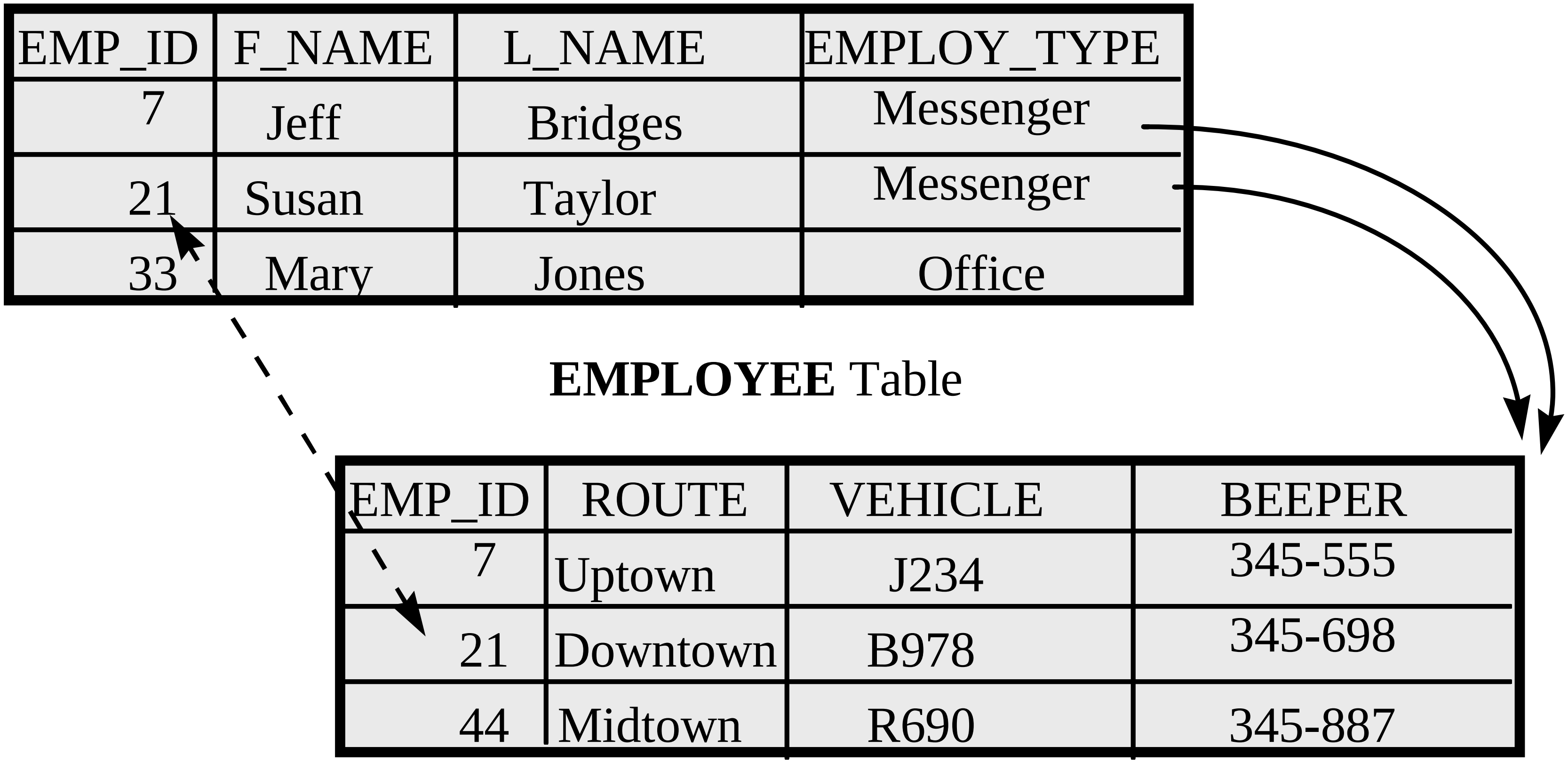
# Representing Inheritance

- Solution
  - Create a table for each class in a hierarchy
  - Add a column in the subclass tables for the (common) key
  - Create concrete subclass instances by JOINing the tables
  - If performance becomes an issue, create a table for each subclass that contains all the inherited attributes.

# Inheritance



# Inheritance

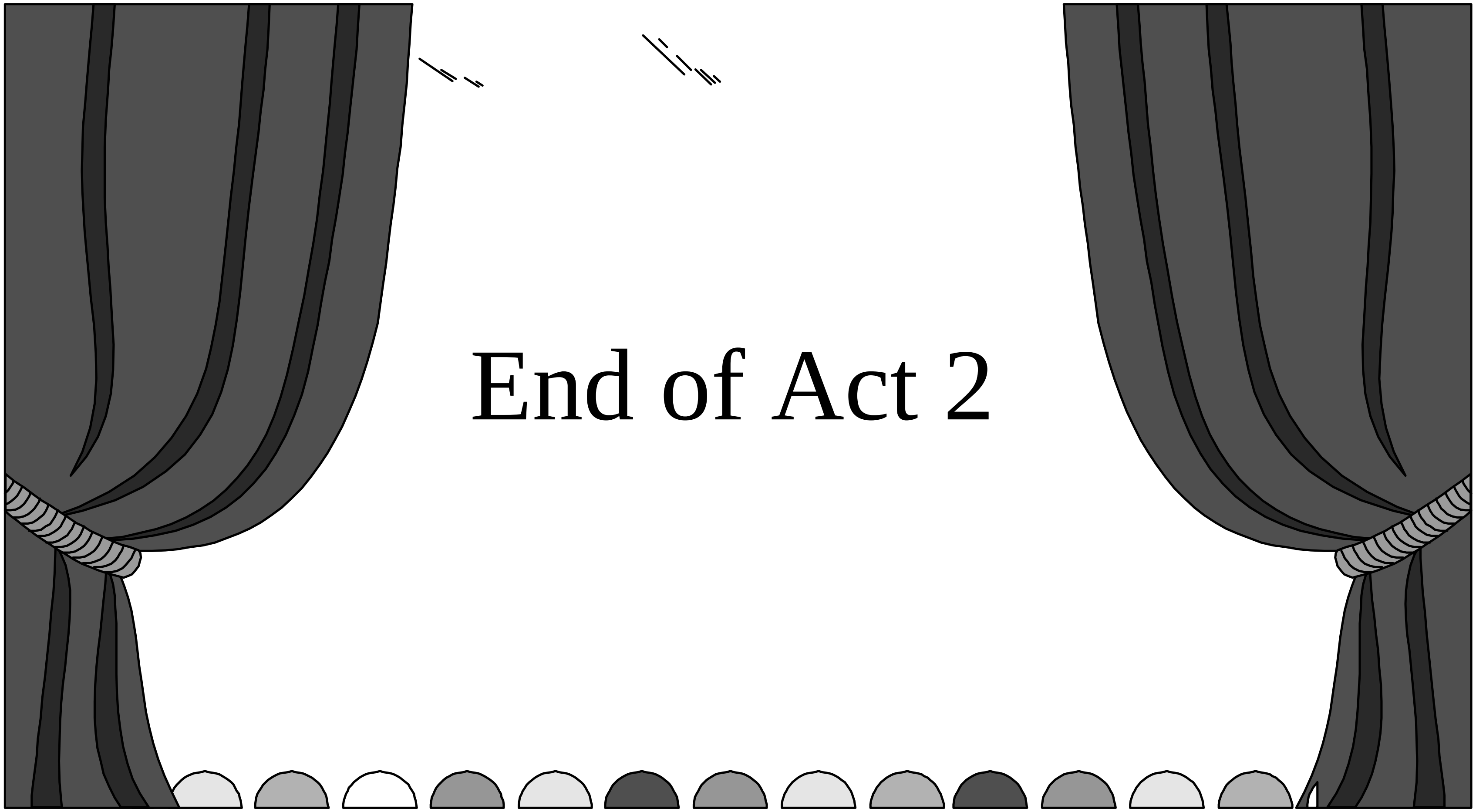


| EMP_ID | F_NAME | L_NAME | EMPLOY_TYPE |
|---|---|---|---|
| 7 | Jeff | Bridges | Messenger |
| 21 | Susan | Taylor | Messenger |
| 33 | Mary | Jones | Office |

**EMPLOYEE** Table

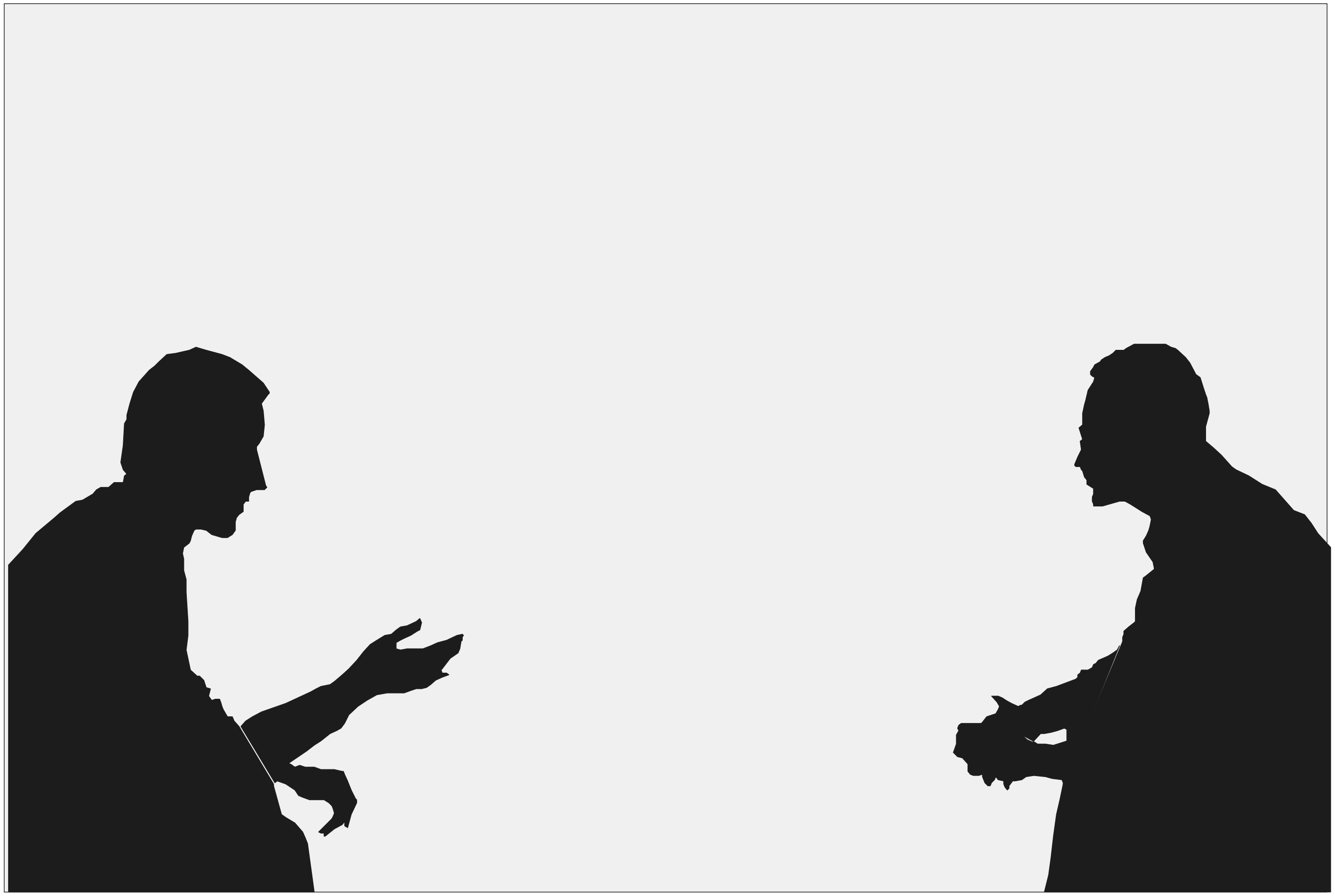
| EMP_ID | ROUTE | VEHICLE | BEEPER |
|---|---|---|---|
| 7 | Uptown | J234 | 345-555 |
| 21 | Downtown | B978 | 345-698 |
| 44 | Midtown | R690 | 345-887 |

**MESSENGER Table**



# End of Act 2

# Act 3: Brokering objects



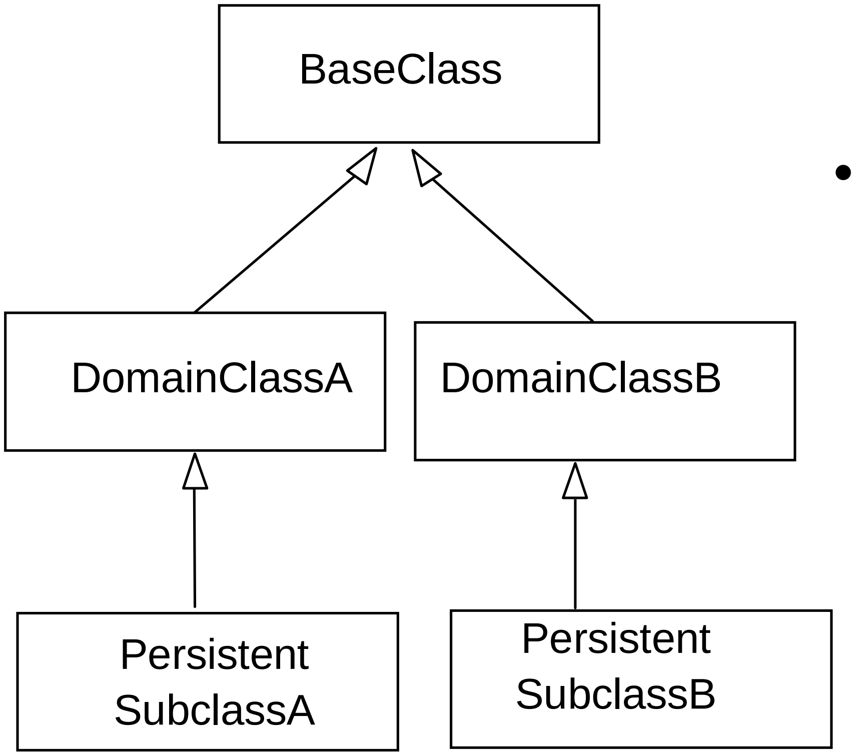
# Possible persistence architectures

- All-In-One
- Persistent Subclasses
- ?

# All-in-one architecture

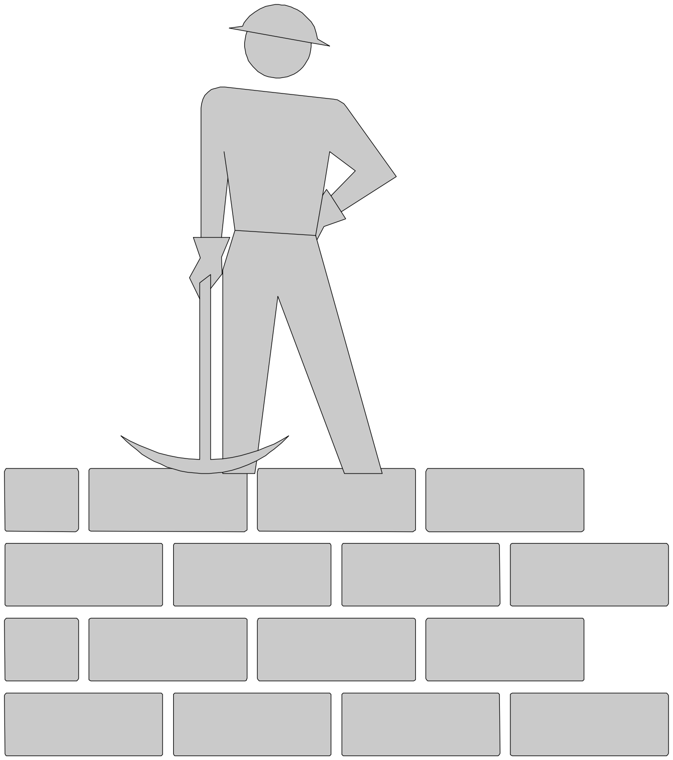
- One potential solution is to include database code directly in domain classes.
- This does not separate concerns and becomes unmaintainable.
- This is unfortunately what many vendors unwittingly promote.

# Persistent Subclasses



- Another solution is to make persistent subclasses of domain classes.
- This results in an explosion of subclasses.

# Pattern: Broker



- Problem: How do you separate the domain-specific parts of an application from the database-specific parts?

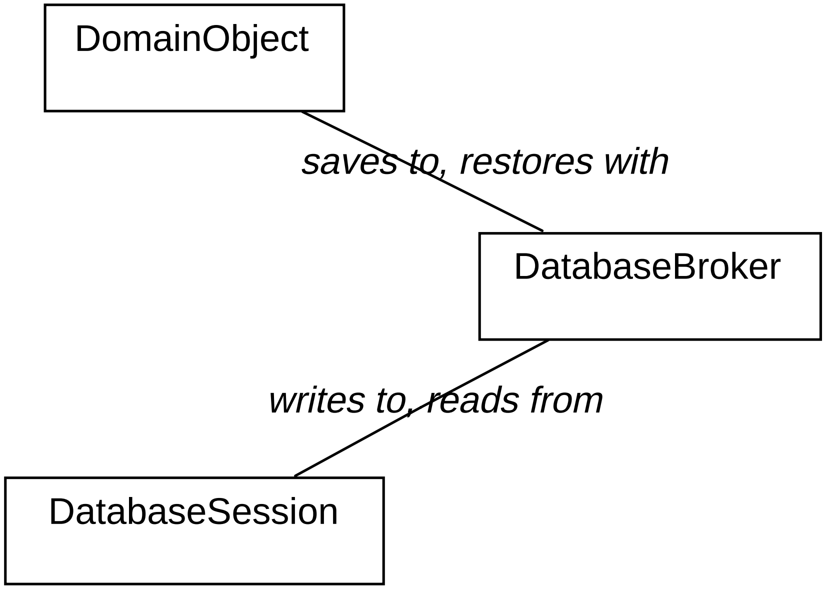
# Broker (2)

- Forces:
  - Architecture should be simple, but powerful and extensible
  - Persistence should be orthogonal to class
  - Must preserve encapsulation and separation of concerns
  - Solution should avoid class explosion

# Broker (3)

- Solution
  - Connect the database-specific classes and the domain-specific classes with an intermediate layer of Broker objects.
  - Brokers mediate between database objects and domain objects and are ultimately responsible for reading from and writing to the database.

# Generic Broker architecture



- Brokers collaborate with both domain objects and database objects.
- This allows domain objects to be ignorant of database issues.

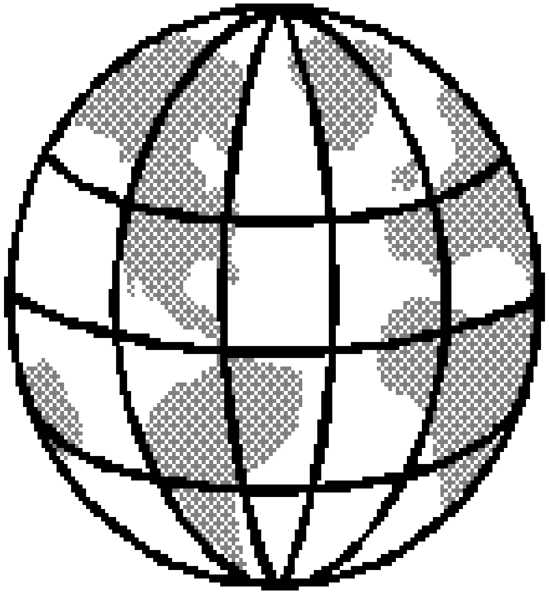
# Object Passivation

- A broker is responsible for writing out (passivating) objects
- View the object as a directed graph.
  - Do a post-order traversal that writes out the leaves before it writes out the intermediate nodes.
  - Use the OID’s generated at the leaves to form foreign-key columns.

# Object Activation

- A broker is also responsible for reading in objects from tables
  - At a minimum, load in the basic attributes of an object and its foreign-key OID’s.
  - As performance needs dictate, instantiate subordinate objects either immediately, or later using Proxies or deferred instantiation.

# Pattern: Mapping Metadata



- Problem
  - How do you define the mapping between an object class and the corresponding parts of a relational schema?

# Mapping tuples (2)

- Forces:
  - You want to avoid duplicated code
  - You would like to handle common situations in the same way

# Mapping tuples (3)

- Solution:
  - Reify the mapping into a set of Map classes that (at the least) map column names in a table to instance variable selectors.
  - More complex maps can map common relationships (1-1, 1-N, M-N) between objects into relational equivalents.

# Maps example (Smalltalk)

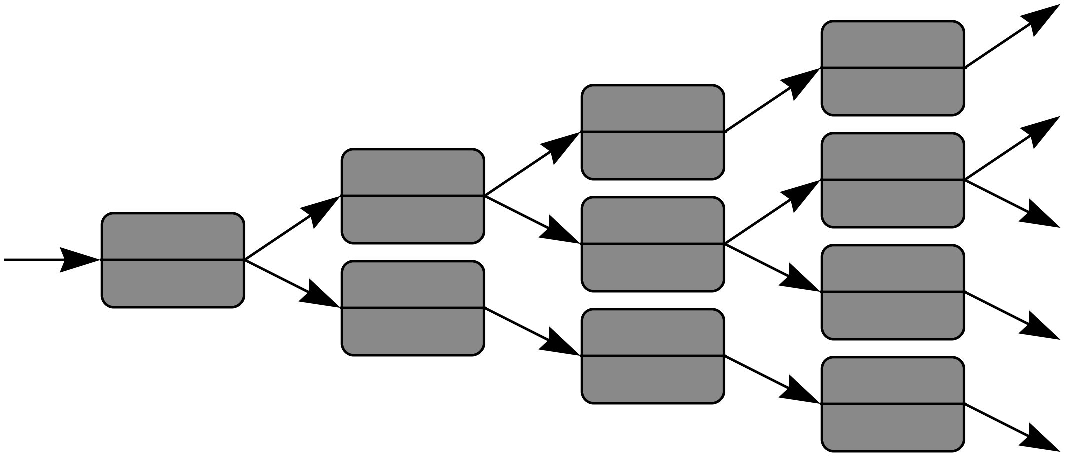
```
SomeDomainObject>>maps

^ RowMap new
        add: ( ColumnMap keyName: ‘user_id’
                        forAspect: #userId) ;
        add: (ColumnMap columnName: ‘full_name’
                        forAspect:  #fullName);
        add: (ColumnMap foreignKey: ‘address_id’
                        forAspect: #address);
        add: ( DateColumnMap columnName: ‘renewal-date’
                        forAspect: #renewalDate);
        yourself).
```

# Maps example (Java)

```
public RowMap toMapping()
  {
     RowMap newMapping = new RowMap();
     newMapping.baseObject = this;
     newMapping.tableName = "CustomerTable";
     newMapping.addStringOID("TelephoneNumber", telephone);
     newMapping.addMapForString("Name", name);
     newMapping.addMapForObject("Address", address);
     return newMapping;
  }
```

# Pattern: Proxy

- Problem: How do you instantiate large, complex objects without severe performance hits and memory problems?



(C) 1996, 1997 Kyle Brown & Bruce Whitenack

# Proxy (2)

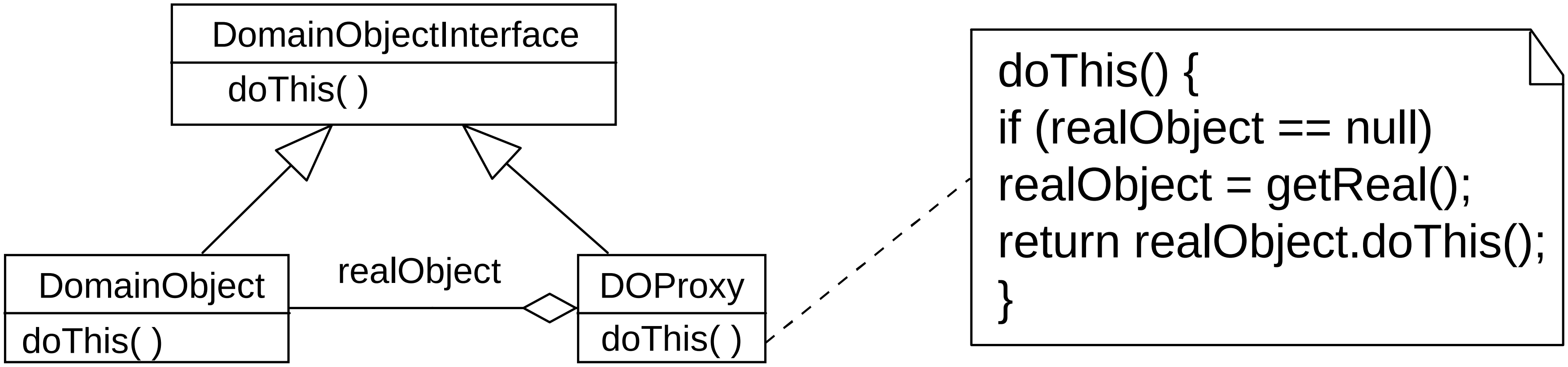
- Forces
  - Many objects are too big to instantiate in their entirety.
  - Applications still need to navigate over part of an object.

# Proxy (3)

- Solution
  - Use a proxy object in place of a full component for newly instantiated objects.
  - The Proxy provides sufficient identification information to instantiate itself when it receives a message meant for the actual object.
  - See [Gamma] for Smalltalk, C++ implementations

# Proxy (Java example)

- Create interfaces that your domain object and its proxy will implement



```
doThis() {
if (realObject == null)
realObject = getReal();
return realObject.doThis();
}
```

# Pattern: Query Objects

- Problem:
  - How do you handle the generation and execution of SQL statements in an OO way?

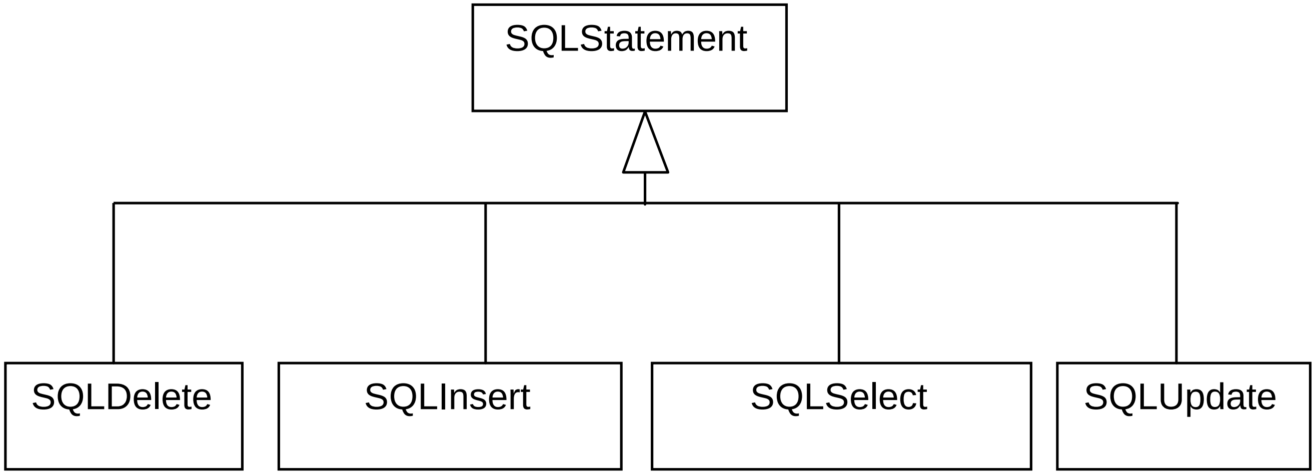
# Query Objects (2)

- Forces:
  - Want to minimize the exposure of the system to SQL
  - Want to maximize shared code

# Query Objects (3)

- Solution:
  - Write a set of classes that generate SQL code from other objects.
  - Query objects collaborate with Map Objects to generate SQL.

# Query object hierarchy



# SQL Statements (Smalltalk)

```
updateStatement := SQLUpdate new.
updateStatement columnMaps: aDO maps;
                tableName: aDO table;
                forObject: aDO.
updateStatement execute.
```

# SQL Statements (Java)

```
SQLStatement statement;
Hashtable allKeyValuePairs;
allKeyValuePairs = map.baseTypesAndForeignKeys();
statement = new SQLUpdateStatement();
statement.generateSQLFrom(allKeyValuePairs,
                    map.oidUpdateClause(), map.tableName);
```

# Pattern: Transaction objects

- Problem
  - How do you represent the concept of a database transaction in an OO language?

# Transactions (2)

- Forces
  - – SQL depends upon transactions to maintain database consistency
  - – OO languages do not (directly) support this concept.

# Transactions (3)

- Solution
  - – Build a Transaction class that represents a Logical Units of Work
  - – Use exception handlers around a block of code that executes SQL code that may fail.
  - – The exception handler will execute a ROLLBACK if an exception is raised, or a COMMIT if none occur.

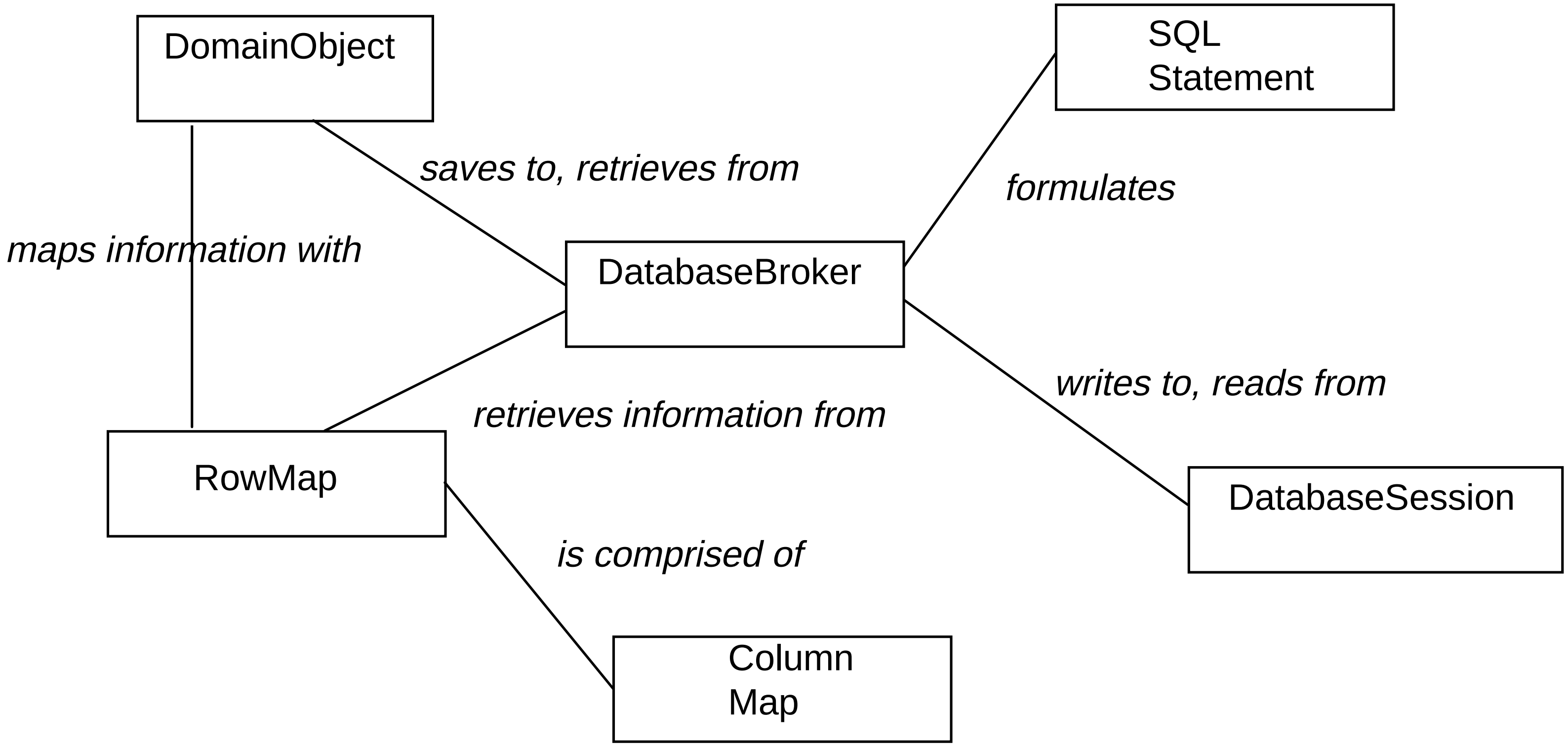
# Transactions (Smalltalk)

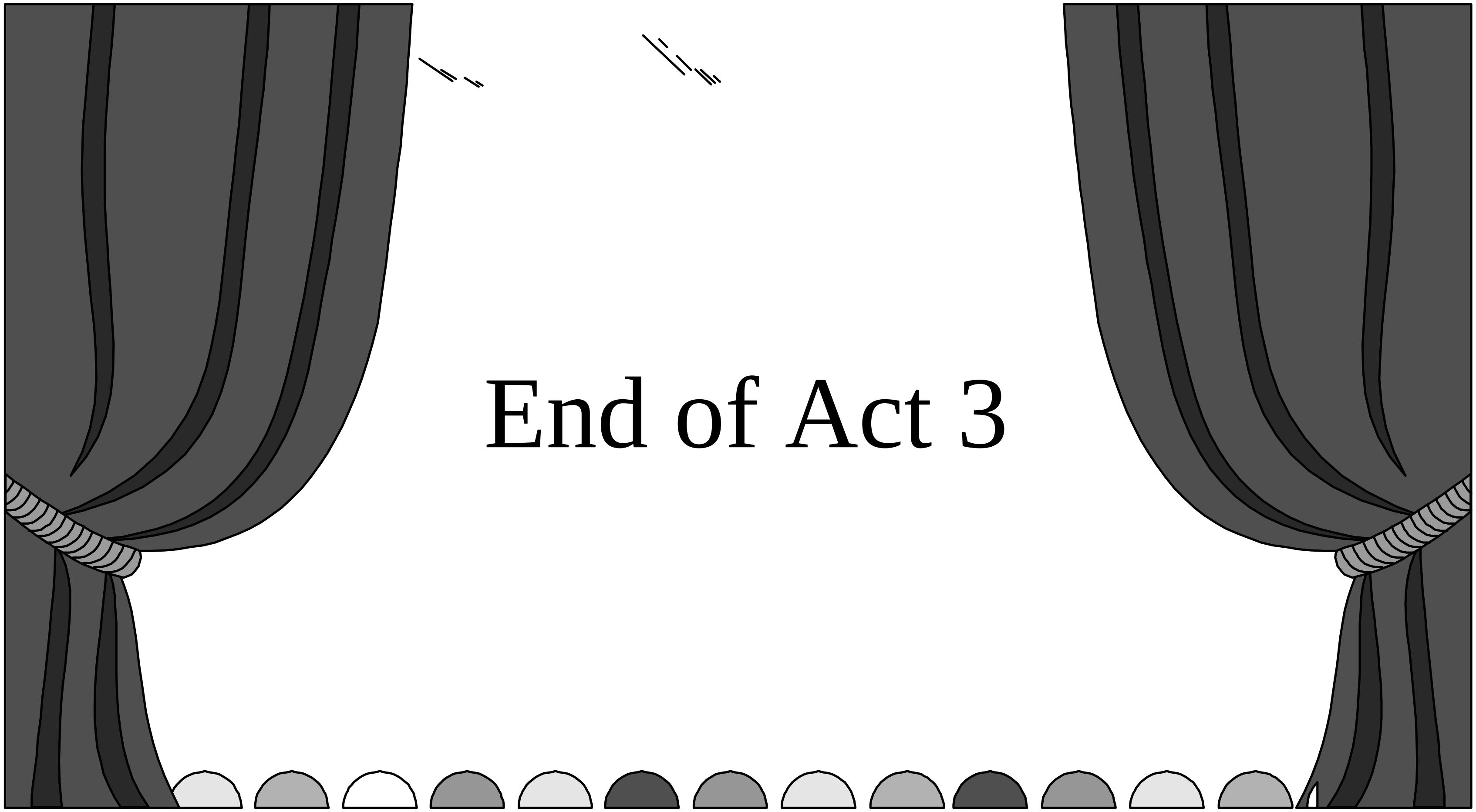
```
SQLTransaction>>doTransaction:

doTransaction: aBlock

        "execute aBlock within the context of a transaction"

        self class errorSignal
                handle: [:ex |
                        self execute: ’ROLLBACK’]
                do: [ aBlock value.
                        self execute: ’COMMIT’].
```

# Transactions (Java)

```
 try {
          ...try executing SQL Statements here…
          currentConnection.commit();
} catch (SQLException se) {
           try {
               currentConnection.rollback();
           } catch (SQLException nse) {
               ...handle truly fatal errors here...
           }
       }
```

# Object Relationships

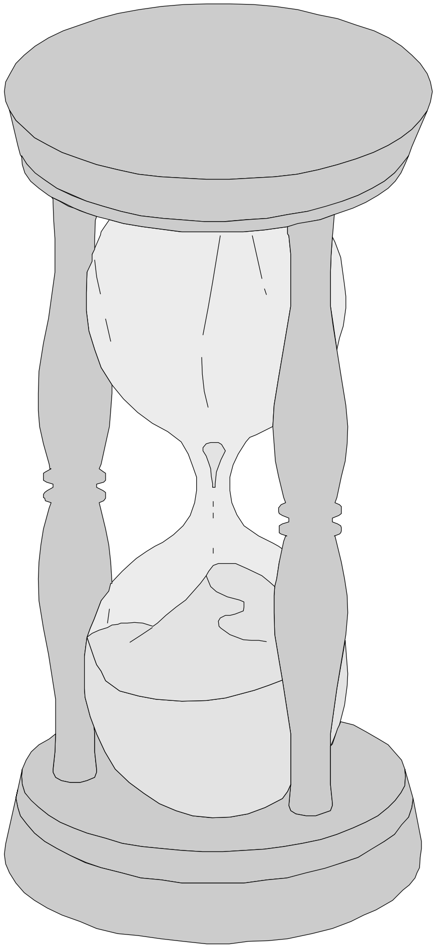




# End of Act 3

# Act 4: Client-Server Concerns

# Pattern: Cache Management



- Problem
  - How do you best manage the lifetime of persistent objects stored in an RDBMS and used on the client?

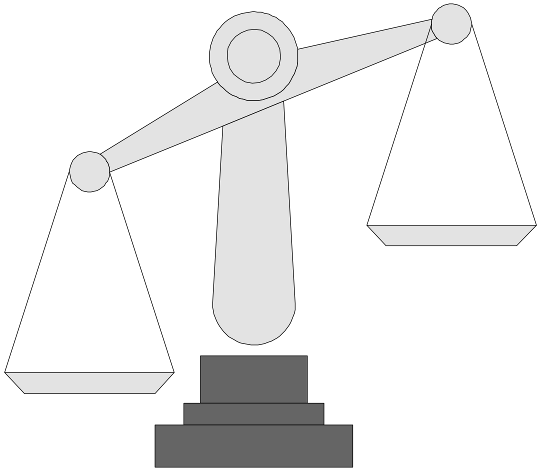
# Cache Management (2)

- Forces
  - Caches increase client performance, but increase client memory size
  - Caches can become out of date
  - Caching increases application complexity

# Cache Management (3)

- Solution
  - Use a Session object that has a bounded lifetime and is responsible for identity cache management of a limited set of objects.
  - Balance speed vs. space by flushing the cache as appropriate
  - Use a query before write (timestamp) technique to keep cache accurate

# Pattern: Distribution of Behavior



- Problem
  - How do you distribute behavior meaningfully between an OO client and a Relational server?

# Distribution of Behavior (2)

- Forces
  - RDBMS's will perform some functions (like sorting) much faster than a Smalltalk client.
  - Triggers in the RDBMS can provide behavior when changes occur
  - When business rules are implemented in a database it hurts portability and reuse. It aslo requires additional code management

# Solution: Distribution of Behavior

- Take a minimalist approach of “guilty until proven innocent”.
- Sorts, major queries (stored procedures), and aggregate functions are best done in the database.
- Triggers and other behavior are more worrisome. Be careful.

# Crossing Chasms

- To obtain a copy of the Crossing Chasms pattern language
  - try our web site host96.ksccary.com
  - or, send email to either
    - bruce@objectpeople.com
    - kbrown@ksccary.com
  - We have RTF, PDF and Postscript -- let us know which you prefer