

## Building a Jasmine Database – an Introduction

**Jasmine Conference, CA-World99**  
**Session: JG105SA and JG105SB**  
**V1.00, July 1999.**

This presentation is designed to be a practical demonstration of building a Jasmine Database. Note that I say “database”, here and not “application”. While I will demonstrate a running set of Jasmine Studio scenes that use my classes, my primary focus will be the construction of the back end object database.

For those attending CA-World 99, I hope you will find this double presentation interesting. I intend to cover a broad range of topics as I construct all the elements of an object database, but note that time will prevent me going into real depth on any one issue (actually, we ran out of time, didn't we!). The real purpose here is to provide as complete a demonstration as I can. I am happy to answer questions both during and after, but point out that if you want in-depth answers, then that's what many of the other presentations at CA-World 99 are there for!

For those of you reading this paper on the Castle Software Australia web site, I hope you find it of use. I cannot publish all the source code for my classes here simply because it would quadruple the length of the paper. So I will print pertinent selections of code, and try to include batch files and other constructs where important.

Those of you who have talk with me about this topic will know that I am working on a full set of training and tutorial notes that will cover this and many other Jasmine topics. One of these days I might actually finish it too... sigh.

I'm going to assume you've installed your copy of Jasmine, and selected all the options. You will be aware that the developer's CD contains a fully operational product, with the exception of a connection limit, and some of the esoteric backup and restore utilities. There is nothing to stop you building and testing a fully operational pilot system and proving to your PHB that it might be worth a try.

### Design Goals

The database we are building here is a simple Timewriting database. Staff in a company record time (in hours) against project phases they are working on. Projects are divided into multiple Phases, and each Project belongs to a single Client. The Class design can be summarised as follows:

**Company.** Only one instance of company will exist. It holds collection references to the current clients, projects and staff

**Client.** Each client has a number of projects underway on their behalf.

**Project.** A project consists of one or more Phases. Staff are assigned to each project.

**Phase.** Each phase is a single unit of work, with a defined beginning and end. Time is recorded by Phase.

**Staff.** Staff work on projects and record time to each by phase.

**Time.** This class holds all the time recorded by staff on project phases each day. Time is recorded as hours and costs

Before anyone says anything, yes this is highly simplistic, and it does have definite flaws were I to use it in a real environment. As a first cut design it's more than enough to proceed with as a demonstration.

### Physical Storage

Jasmine commands and utilities are divided into two broad groups:

1. Operating system commands (programs) that act on the database's physical file storage: backup and restore, allocating file storage, defining class families, reading and defining configuration settings, starting and stopping the database engine, connecting to the database (from a client).
2. Database commands and methods that act at the class and instance level, on items stored inside the database. The include the Object Definition Language (ODL), Object Manipulation (OML) and Query Language (OQL). These commands can only be executed within Jasmine methods, the ODQL interpreter or Jasmine client programs.

Note that both of these tend to be lumped into the one heading **Object Data Query Language**, or **ODQL**. Most times you will think of ODQL as the programming language used to write, store and compile methods (item 2 above), and that's how I'll refer to it here.

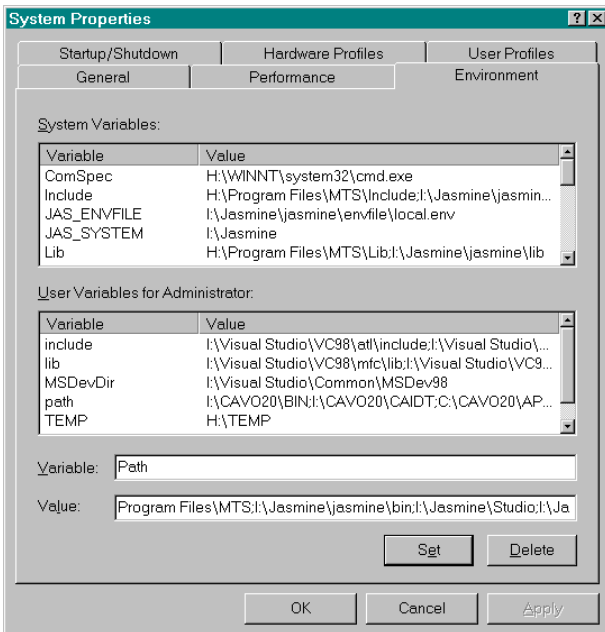
## Where is Everything?

At this point I have to apologise to the UNIX people out there. All my examples are from an NT system (NT Workstation 4.0 SR3). Its what I have in front of me now, and I will not enter into a debate on which environment is good for you. Just be aware that you should check the documentation before trying any of the scripts I demonstrate here. In some situations there are differences in syntax.

Ok, Jasmine is installed. So where is everything?

Run the **System** option on the **Control Panel**. Select the Environment tab and you will see something like Figure 1. Jasmine will have installed itself onto the Path, as well as various other environment variables. On important one to remember is **JAS\_SYSTEM**, which is the root drive/path Jasmine was installed on, and **JAS\_ENVFILE**, which is the default connection environment file (useful for batch files later!).

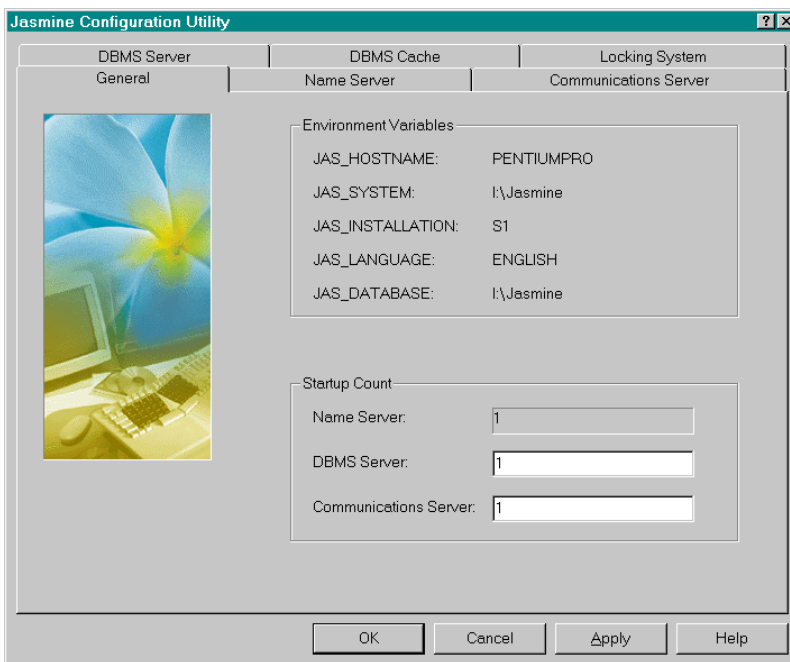
Other environment variables are used by the C++ compiler (Visual C++ here), and Jasmine also hooks into those automatically.



**Figure 1: Environment Variables on an NT server with Jasmine**

The above are the configuration setting known to the operating system. There are a host of others which are monitored and controlled using the Jasmine Configuration Utility **Jasconfig.exe** (Figure 2).

Most of the settings here don't need to be changed unless you are trying to optimise specific parts of the Jasmine server. Worth reading up on if want to be a Jasmine Database Administrator.



**Figure 2: The Jasmine Configuration Utility**

Other environment variables which concern us specifically can be found using the **Jaspreenv.exe** utility (and its sister functions **JasSetenv.exe** and **JasUnset.exe**). On my PC, jaspreenv produces the following:

#### **Using Jaspreenv:**

---

```
JAS_JOURNAL_MODE=OFF
JAS_LANGUAGE=ENGLISH
JAS_TIMEZONE_NAME=AUSTRALIA-VICTORIA
TERM_JASMINE=IBMPC
JAS_INSTALLATION=S1
JAS_CHARSETS1=IBMPC850
JAS_TEMPORARY=I:\Jasmine\jasmine\temp
JAS_CONNECT_RETRIES=5D
JAS_CONFIG=I:\Jasmine\jasmine\files
JAS_MSGDIR=I:\Jasmine\jasmine\files\ENGLISH
JAS_DATABASE=I:\Jasmine
JAS_HOSTNAME=PENTIUMPRO
JAS_METHOD_DIRECTORY=I:\Jasmine\jasmine\data\default\methods
JAS_STORE_EXTENTS=1
JAS_SIZE_STRIPE_1=16
JAS_EXTENT_1=I:\Jasmine\Jasmine\data\jas_extent_1
JAS_SIZE_EXTENT_1=2000
```

---

As a general rule you should not change these. They are quite useful however in pointing out where various core Jasmine files have been placed on our server. We can use this information to our advantage.

Note that the above utilities do not require Jasmine to be running. In fact many changes will insist on the server being shut down before you can make them.

## **Stores and Extents**

Like most server databases, we have to explicitly allocate filespace for the server to work with. While Jasmine does not offer too many features for optimisation here, it's got all the basics, so I expect more to come with each new release.

The basic logical unit of file allocation is the **Store**. Each Store is made up of one or more **Extents**. Each Extent can be one or more fixed length files. These have to be created by the database administrator as required. If a Store is filled before new Extents are created then an error will occur.

Thus by using multiple Extents, a Store can span drives and be as large as your operating system permits. Optimisation is achieved by placing Stores for different applications on different physical drives. Another optimisation method places Stores on different drives to the default System Store, Work Store and Transaction Store (the last two are specific to each session/connection between client and server)

Creating a Store is as simple as:

**SetupStores.bat: Batch file for creating our Store and Class Family**

```
@echo Create a store called CastleDemo01. This will consist of a single Extent 16M in size.
createStore -numberOfPages 2000 -pageSize 8 CastleDemo01 %JAS_SYSTEM%\jasmine\data\castle_1
pause

@echo Create a single class Family in this store, called castleCF01. Remember to
@echo mention that we'll always refer to it using an alias!
createcf -CFalias castleCF castleCF01 CastleDemo01
pause

@echo Now list out all the stores and class families to show the new addition...
liststore > StoreList.txt
```

This creates a Store by the name of **castle\_1** in the same directory as the existing Jasmine stores, that's 2000 \* 8K in size. The **pageSize** is important as this is the minimum amount that is read/written to the Store with each access.

At this point I need to make one observation. ALL Jasmine commands (and later the entire ODQL) is case sensitive. If I had specified "**-pagesize 8**" in the above batch file an error would have resulted. Those of you who come from a PC background like me will find this hard to get used to. If you ever get an odd error while using any Jasmine or ODQL command, my first suggestion is check your syntax and CASE --you probably made a #S%#\$^%@\$# typo!

I'll go into the Class Family stuff later on, but the last command in the above batch file dumps a description of all Stores in my database to a file, while is listed here:

**StoreList.txt: Our Jasmine Database now:**

```
===== S T O R E   C O N T E N T S =====
Store Name: system
Class Families:
    systemCF
Locations:
    I:\Jasmine\Jasmine\data\jas_extent_1
Page size: 8192
Total pages: 2000
Used pages: 508

===== S T O R E   C O N T E N T S =====
Store Name: dataStore
Class Families:
    jadelibCF1
    mediaCF1
    CAStore
    sqlCF
    WebLink
Locations:
    I:\Jasmine\Jasmine\data\jas_dataStore
Page size: 8192
Total pages: 8000
Used pages: 649

===== S T O R E   C O N T E N T S =====
Store Name: CastleDemo01
Class Families:
    castleCF01
Locations:
    i:\jasmine\jasmine\data\castle_1
Page size: 8192
Total pages: 2000
Used pages: 33
```

The information in this dump is quite interesting, especially when you compare it with the results of **Jaspreenv** earlier.

If we want to extend our Store at a later date, we can use the **extendStore** command as follows:

**SetupStores2.bat: Adding some more space**

```
extendstore -numberOfPages 1200 CastleDemo01 i:\Jasmine\jasmine\data\castle_2a
                                                i:\jasmine\jasmine\data\castle_2b
```

Here I decided to specify two files for this new Extent. If I rerun **listStore** then **CastleDemo01** now contains:

```
Locations:
    i:\jasmine\jasmine\data\castle_1
    i:\Jasmine\jasmine\data\castle_2a
    i:\jasmine\jasmine\data\castle_2b
```

It's important to note that **listStore** is currently the only way to see how much space is in use in your database, and whether you need to allocate more Extents. As such it's a command that must be used regularly by the database administrator.

In the current version (v1.21) you have no control over how space is allocated within a Store, so disk striping and other advanced techniques are not possible.

## Class Families

The Class Family is the basic logical unit within a Store. The name assigned to each Class Family is unique, although there are some interesting ways to work with that.

A Class Family is contained within a single Store. That is, all Class definitions within that Class Family, and instance data for those classes is kept together. As a general rule, you will keep all classes for a single application in the one Class Family. The reason for this has to do with **Aliasing**.

**Aliasing** is a technique whereby one Class family can be known by a different name while the database is running (more specifically, the name can be unique for any given session/connection). Thus, after using `createCF` in the batch file earlier, I run the following command:

---

```
copy local.env %JAS_ENVFILE%
```

---

which overwrites my default Jasmine Environment file `Local.env` with a new file containing:

---

**Environment File for Aliasing** `castleCF01`

---

```
bufferSize      1024
workSize        20480 10240
CF mediaCF      mediaCF1
CF jadelibCF    jadelibCF1
CF castleCF     castleCF01
```

---

This provides for two things:

- A more colloquial name for the Class Family for normal use.
- A way of creating and testing a completely separate class family that contains a newer version of that application (classes, data... etc) without affecting the production system. When I want to switch over, I simply change the ENV files on the client PCs to reflect the new Alias arrangement.

This way of separating Dev/Test/Prod systems is common on high end computing platforms.

Later on we'll see how you can use this to create multiple copies of `jadelibCF` and `mediaCF` to better isolate multimedia data and Jasmine Studio for different applications, providing for much better backup and restore management.

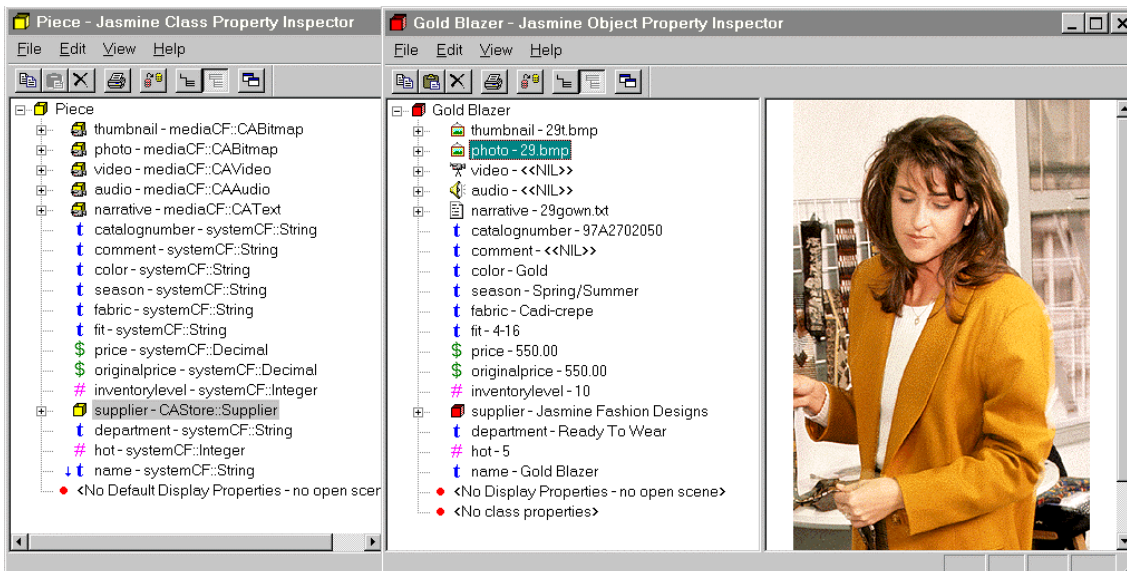
You can delete Class Families with one command: `deleteCF`. Be aware that this immediately erases all the contents of the Class Family (Classes, instance... etc) without pause. Useful for cleaning up during development, but something to be careful of otherwise. Unlike the `deleteStore` command, it won't ask if you are sure first.

## Class Definitions

### *The Database Administrator and ODQL Interpreter*

Now we want to build the internals of our database, we have two tools to use: The Database Administrator, and the ODQL Interpreter. Choosing the correct tool for each situation is very important, and impacts on your ability to administer, migrate, upgrade and support your database.

The **Database Administrator** is part of Jasmine Studio. It's a point and click tool for browsing the database schema and its contents (instances of objects in the database) as well as changing the database structure (Figure 3).



**Figure 3: The Database Administrator – Viewing Class schemas and instance data.**

The advantages of the Database Administrator are:

- It's very easy to learn and use.
- Only practical way to view multimedia data instances.
- Drag and Drop is convenient way of populating data, especially multimedia types.
- Building classes and compiling methods is a one-touch operation.
- (Is an essential party to building applications with Jasmine Studio)

The **ODQL Interpreter** is the antithesis of the Administrator. It is a command line environment which only accepts ODQL language input (Figure 4). While this makes the Interpreter hostile to the casual user, it's a very powerful tool. Consider it for the following:

- It's the only way to run ODQL scripts. These could be testing code for methods, creating instances of data, or generally automating some task.
- It's the only medium which supports the bulk creation of test data (apart from using the LOAD utility, which assume you already have the database created on another server).
- Supports detailed inspection of all class attributes, instance data... etc.
- Using ODQL is the only way you can specifically control how multimedia data is stored (Internal, Controlled or External storage)
- Classes defined here can select the type of collection properties they use (Array, List, set or Bag). The Database Administrator does not allow this for some reason.
- You can run ANY type of test **Query**, including the **Group By** construct which the administrator does not support at all.
- Use the ODQL Class to help you build queries, and the Tuple structures required to hold the results. This can get quite messy, and the ODQL class is a great help when writing methods. Note that this is distinct from ODQL-the-language..
- You can run test scripts and capture the output to text files, thus offering a way of documenting/validating work done.
- If your methods require code from C libraries to be linked in, you have to do the compilation step using the Interpreter.

As you can see, the Interpreter is used for two things: handling complex operations, and writing/running ODQL scripts

```

Command Prompt - codqlie
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>codqlie
Client ODQL Interpreter
Jasmine Version 1.1
Portions of this product Copyright 1996, 1997 Computer Associates International
, Inc.
Portions of this product Copyright 1996, 1997 FUJITSU LIMITED
Portions of this product Copyright 1996, 1997 Computer Associates International
, Inc. & FUJITSU LIMITED

Connecting to host PENTIUMPRO.
PENTIUMPRO(systemCF) >

```

**Figure 4: The ODQL Interpreter.**

I should note that the Interpreter is unforgiving when it comes to errors. Minor errors in case can cause long chains of messages to appear and errors in methods can be awkward to track down.

Complexity aside, those of you who are already database administrators for relational system will see the benefits of the ODQL Interpreter. Being able to record all database schema information, schema changes, compilation instructions and even data unloading and loading into scripts which can be executed, and re-executed at will is a vital part of database administration. If you lose a server and have to reconstruct the database from a particular point in time, you need full logs of all changes, and scripts to run to bring it up to par quickly. This is not something you use a point-and-click tool for.

In relational systems these scripts would be SQL and various stored procedures. In a Jasmine OODB it's ODQL. Most of the examples I will present to you here will be constructed using ODQL scripts. Most of them will be executed in batch format to save time, and to illustrate the point about automation.

Those of you familiar with C will recognise the syntax. I suggest the rest of you have a copy of the [JasRef.hlp](#) file handy as you read this document, and/or attend some of the ODQL specific sessions at CA-World 99.

During this presentation you will see me jump from the Interpreter to the Administrator and back again many times. As I said before, the Administrator is a much better tool for ad-hoc browsing, and showing you what I've done – but the Interpreter is a far better and quicker way of actually doing it!

## **Class Definitions**

Listing 1 below shows the complete class definition script for my database. While my class structure is rather flat (I'm not using inheritance here at all), I have deliberately constructed a **CastleComposite** Class to act as the abstract (base) class for my Class Family.

The reasons for this are:

- Having an abstract class means I can add common behaviour to all classes later (error logging, audit trails) by modifying this central class.
- It provides a logical link between unrelated classes in my database. Thus anyone looking at the system later will have no doubts about which classes are part of this application.
- It provides me with a single class hook to use if I need to generate queries returning my data (instead of other classes in the database).
- During Development and Testing a common requirement is to dump all data and reload a standard test set. having a common abstract class makes that delete process a one-step operation. Don't underestimate the value of this. Who here hasn't learnt a LOT about test strategies during Y2L projects recently!?

That said, I should mention that the most important class design decision is to ensure all classes for a single application are kept in the one Class Family. This is because the Unload and Load commands (or JasLoad and JasUnload) are easiest to use when whole Class Families are being shifted around.

There are a number of things I should explain about this listing:

- I am only defining the basic classes here (Properties), not their methods. Its sensible to keep method definitions separate, as during development you often need to debug, change and recompile methods, but you rarely need to change the core class structure. If you specify the Method definition here, you have to erase the class (and all instance data) each time you re-run this script. If you keep the method separate, you can erase and redefine them without affecting your test data.
- Notice that I specify the default Class Family as systemCF. When running any script, its wise not to make any assumptions about the Interpreter environment (that true of any programming frankly). I am similarly careful when using references to object properties that are my own classes, eg: `List <castleCF::Project> projectList`
- I am using Lists here because it will make certain methods easier to code later. In ODQL you can specifically select the collection type to use (Array, List, Set or Bag). Read up on each of these and pick the one most suited to your needs.
- I've commented my script file so I or any maintenance programmer can go back over it later, and still understand what was being done. Again, this is simply good coding techniques and takes no extra time at all if done while you write the script.
- I am making full use of the Description options in my Class definitions. Thus anyone viewing the class schema in the database administrator (or in the interpreter) will always have some help working out what each class and property is.
- The last line is the command to build the classes. In reality you would run the entire script first to check for syntactic errors, and maybe make some corrections. After a successful run, you would issue the **buildClass** call manually.

Of course, I've done that, so I know my script works. Thus my final addition is this buildClass command so I can reconstruct and rebuild my classes at a moment's notice. It also makes a complete record of what I did to create and build my classes.

***Listing 1 (Classes.odql): ODQL for defining and building classes in this database.***

---

```

/*****
File: Classes.odql

Class definitions for Demonstration Timewriting Database

v1.00  March 1999
Peter Fallon
Castle Software Australia
*****/

defaultCF systemCF;

/* Class: CastleComposite

Provided in case I need to define application
wide behaviour in the future such as error logging, audit trails...etc */

defineClass castleCF::CastleComposite
  super: systemCF::Composite
  description: "Base (Abstract) class for Demonstration database."
{
};

/* Class: Company

This class is at the top of the Object Hierarchy (containing tree) for
this system. It contains the master lists of active staff, projects and clients.
*/

defineClass castleCF::Company

  super: castleCF::CastleComposite
  description: "Principle container class for timewriting system. All work is done by the
Company."
{
  maxInstanceSize: 8;
  instance:
    String          companyTitle;
    String          address;

    List <castleCF::Staff>  staffList
      description: "List of Staff currently employed by the
company."
      default: List{};
    List <castleCF::Project>  projectList
      description: "List of active Projects being worked on by the
company."
      default: List{};

```

```

    List <castleCF::Client>    clientList
                               description: "List of the company's current clients."
                               default: List{};
};

/* Class: Client

Clients of the Company. Each may have one or more projects in progress. Note that past clients
remain instances of this class, but they won't be in the Company.ClientList{}
*/

defineClass castleCF::Client

    super: castleCF::CastleComposite
    description: "Client for whom project work is done."
{
    maxInstanceSize: 8;
    instance:
        String                name;
        String                address;

        List <castleCF::Project>    projectList
                                     description: "List of active Projects being worked on by the
company for this client."
                                     default: List{};
};

/*
Class: Project

A Project is the complete unit of work done for a client by the company. It consists of
one or more Phases. Time is written to Phases of the Project by staff.

Estimates are recorded as properties while Actuals are calculated live from methods.
*/

defineClass castleCF::Project

    super: castleCF::CastleComposite
    description: "Project class, containing all information about a specific project"
{
    maxInstanceSize: 4;
    instance:
        String                name;
        String                longDescription;
        Date                 projectStart;
        Date                 estimatedProjectFinish;
        Date                 actualProjectFinish;
        Decimal [10,2]       estimatedHours;
        Decimal [12,2]       estimatedCost;
        Boolean              completed;

        List <castleCF::Staff>    staffList
                                     description: "List of Staff assigned full time to this project."
                                     default: List{};

        List <castleCF::Phase>    phaseList
                                     description: "List of Phases that make up this Project."
                                     default: List{};

        castleCF::Client        client
                                     description: "Client paying the bills for this project.";
};

/*
Class: Phase

A Phase is a single unit of work in a project, such as Analysis or Construction.
Estimates are recorded as properties while Actuals are calculated live from methods.
*/

defineClass castleCF::Phase

    super: castleCF::CastleComposite
    description: "Project Phase class, containing information about a specific phase of a project"
{
    maxInstanceSize: 4;
    instance:
        String                name;
        String                longDescription;
        Date                 phaseStart;
        Date                 estimatedPhaseFinish;

```

```

Date                actualPhaseFinish;
Decimal [10,2]      estimatedHours;
Decimal [12,2]      estimatedCost;
Boolean             completed;

List <castleCF::Project> project
    description: "Parent project for this phase.";
};

/*
Class: Staff

Staff are the resources that do the work on all projects. All time
recorded is allocated to specific staff members.
*/

defineClass castleCF::Staff

    super: castleCF::CastleComposite
    description: "Staff class. The people that do all the work around here."
{
    maxInstanceSize: 4;
    instance:
        String                surname;
        String                givenName;
        String [10]           employeeNumber
            mandatory:
                unique;;
        Decimal [5,2]         hourlyRate
            default: 0;

        List <castleCF::Project> projectList
            description: "List of Projects this person is assigned to."
            default: List{};
};

/*
Class: Time

These are the basic units of time recording. instances of time are specific to
Staff, Project phases, and date.
*/

defineClass castleCF::Time

    super: castleCF::CastleComposite
    description: "Time class. The basic units of time recording used here."
{
    maxInstanceSize: 4;
    instance:
        Date                WhenDone;
        Decimal [10,2]       timeSpent
            default: 0;
        Decimal [12,2]       cost
            default: 0
            description: "Calculated from staff's hourly rate";

        castleCF::Phase     phase
            description: "Project Phase time is recorded for";

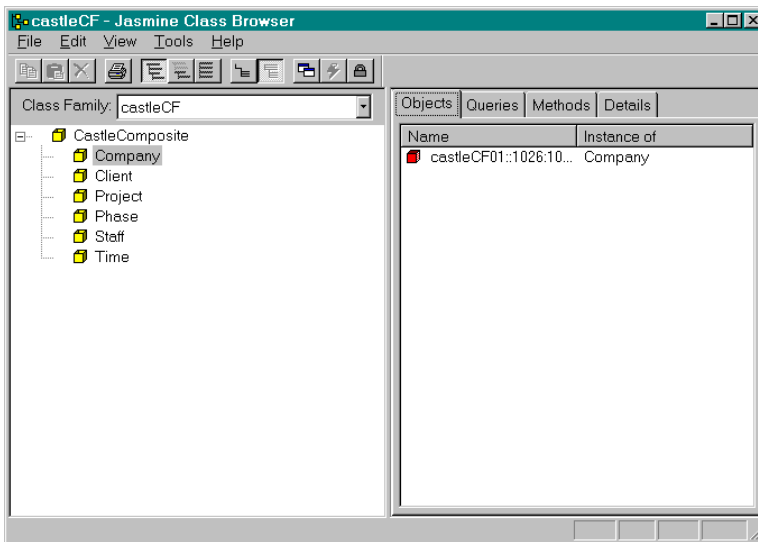
        castleCF::Staff     staff
            description: "Who did the work";
};
/*
Build all classes. Until this command is executed, you cannot create instances of these classes.
Note that compiling methods is a separate step, and doesn't have to be done if we just want
to create instances of these classes!
*/
buildClass castleCF::CastleComposite castleCF::Company
    castleCF::Client castleCF::Project castleCF::Phase castleCF::Staff castleCF::Time;

```

Listing 1 can be run at the NT Command line by:

```
codqlie -execFile Classes.odql
```

If I open the Database Administrator I can see and browse these new classes as Figure 5 demonstrates (note that Figure 5 also shows an instance of the Company Class that I created while experimenting with test data, but I'm getting ahead of myself... )



**Figure 5: The castleCF Schema after our classes are loaded**

I should explain something about **Defining** and **Building** Classes at this point.

There are four stages in constructing Classes:

1. **Defining the class.** This is what the script in Listing 1 was doing. At this point the database has stored the class definitions, but you cannot create or store instances of those classes yet. Note that you can define methods here, although its preferable not to (again, the source is recorded, but not usable).

If you make a mistake in your definition, you can correct it and re-run your define script. You don't need to explicitly delete a class definition until you Build it.

2. **Building the Class.** Think of this as a "compile" step for the class definitions. After a successful build, you can create instances of your classes. A class cannot be built until all its superclasses have been defined (they will be built automatically if you haven't done it explicitly yet), and neither can it be built unless all classes used as properties have been defined. for instance: I can't build the Staff Class above until I define the Project Class, as Staff has a property:

```
List<Project> projectList.
```

If you need to change the class definition after its built, you must either **delete()** it first (this deletes all instances of the class as well), or use discrete methods such as **addProperty()** which leave existing test data in place. there are class methods in Jasmine to edit and adjust most parts of a class definition after its built. However you do this, I suggest you ensure commented and dated scripts exist to track such changes.

3. **Defining Methods.** If you follow the process I'm using here, this will be done using the **addProcedure** command. This is where you define the source code for each method of each class. You can do this in any order you want, but you cannot compile a method until all dependant classes, types and methods are also defined. Note that jasmine performs some basic type and syntax checking of your ODQL at this point.

If you need to change the source code you must use the **removeMethod()** method to delete it first. This does not affect the class definitions or other method definitions.

4. **Compiling Methods.** This invokes the Microsoft VC++ compiler in background to compile your methods into DLLs (at least on NT systems anyway... ). You can specify a number of options here, including ones to retain source and debugging code, linking in extra object files, compile options... etc.

The DLLs created are placed in a method directory with the same name as the host Class Family. If you want to change a method, then use **removeMethod()**, redefine it and compile it. Note that every time you use **compileProcedure()** you are recompiling all methods for a given class.

## Loading Data

Normally the next step in construction would be to define the Methods for my Classes. However, I'd like to experiment with ODQL first as I'm unsure exactly what I want some methods to do, and I need to practice writing queries anyway.

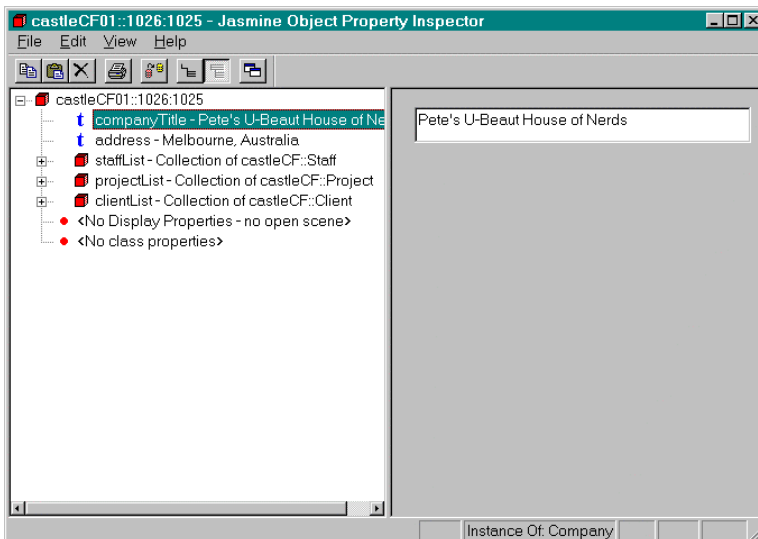
The obvious solution is to create some test data!

Test data can be created in several ways:

1. Use the Database Administrator. It only supports one object at a time however, so entering time data will be rather slow. As mentioned before, I won't have any way to log or record what I've done for later. If the amount of data you required was small (and includes multimedia types), this is still a good choice however. See Figure 6 below.
2. Use ODQL and create a script. Its more complex initially, but it IS a script – so it can be re-run against a fresh DB anytime... More importantly, I can write a program to generate any amount of data I wish, which is useful if I want to create and test queries.
3. LOAD data from a previous UNLOAD file. This is excellent for loading large amounts of data, but the assumption is that is already exists in some form elsewhere. its important to remember this however, as the LOAD file format is documented, so if you need to transfer data from another database type (say relational), you could write a program to output it in the Jasmine format.
4. Write a small client application and use it as a keypunching screen (you or a typist can then enter data at will). The VB/ActiveX interface would be ideal for this. The program will be a throwaway, but you may want to consider it if you need very particular test data that is not already in electronic format somewhere.

My situation here means that option 2) is the best choice. I need to create a small number of instances for staff, projects and clients, but a LOT of time entries. So a script is the best way to go.

Note that whatever method of data entry you chose, you should use Jasunload and create a backup of your test data as soon as you create it. This gives you a fast and easy method of restoring original test cases... etc when you really get down to business later on. **One of the key points to remember here is that nearly every good technique you've used for database system development in the past is either directly applicable with Jasmine, or has an object oriented equivalent.** At every point ask yourself, what have I done with (relational) databases before, and have I thought about applying it here?



**Figure 6. Simple data entry using the Database Administrator**

Actually I had another reason for creating data this way. At the time of writing this paper my ODQL was a bit rusty. This gave me a good chance to get back into practice!

Ok Listing 2 contains the script I used to create a few instances of the Company, Client Staff, Project and Phase classes. It's a time writing system simulating a small company operation, so this is sufficient for now (After all, I can add more cases to this script later if required).

This script and the next one highlight one interesting problem with ODQL here. Because this script must run inside the interpreter (in interpreted mode effectively) I am limited to strict ODQL code. Unlike methods which can utilise nearly anything you can code in C (assuming you link in the required object code later). One way around this is to write your own classes which simply provide the functionality you like to have here (file access, certain system functions, logging operations... etc). I haven't had the chance to do this here, but I hope the C gurus among you are thinking about this!

**Listing 2 (DataEntry01.odql): Code for creating a few instances of each class**

```

/* File: DataEntry01.odql

Data entry script when there are no methods defined yet.
Load a couple of objects into each class.

v1.00  March 1999
Peter Fallon
Castle Software Australia
*/

defaultCF castleCF;

/* Define a Company Record */
Company oCompany;
oCompany = Company.new();
oCompany.companyTitle = "Pete's Software Emporium";
oCompany.address = "Melbourne, Australia";

/* Hold that thought, and create a couple of Project instances
and add that to the Company object */

Project oProject1, oProject2;
oProject1 = Project.new();

oProject1.name = "Patient Needs Trial Database";
oProject1.longDescription = "Touch screen based questionnaire system for PeterMacCallum Cancer
Institute";
oProject1.completed = FALSE;
oProject1.projectStart = Date.construct(1999,3,1);
oProject1.estimatedProjectFinish = Date.construct(1999,8,30);
oProject1.estimatedHours = 250.0;
oProject1.estimatedCost = 20000.00;

oProject2 = Project.new();
oProject2.name = "Y2K Project";
oProject2.longDescription = "Financial System requires Y2K checking and certification";
oProject2.completed = FALSE;
oProject2.projectStart = Date.construct(1998,6,1);
oProject2.estimatedProjectFinish = Date.construct(1999,10,31);
oProject2.estimatedHours = 1500.0;
oProject2.estimatedCost = 100000.00;

oCompany.directAdd("projectList", oProject1);
oCompany.directAdd("projectList", oProject2);

/* Create some Client Objects
Add Clients to Company object, respective Project objects, and Projects to Clients
*/

Client oClient1, oClient2;

oClient1 = Client.new();
oClient1.name = "Peter MacCallum Cancer Institute";
oClient1.address = "St Andrews Place, Melbourne";
oClient1.directAdd("projectList", oProject1);
oCompany.directAdd("clientList", oClient1);
oProject1.client = oClient1;

oClient2 = Client.new();
oClient2.name = "Contracting Agency";
oClient2.address = "May as well be Woop Woop";
oClient2.directAdd("projectList", oProject2);
oCompany.directAdd("clientList", oClient2);
oProject2.client = oClient2;

/* Create Staff

Add to Company, and Project instances
*/
Staff oStaff1, oStaff2, oStaff3;

oStaff1 = Staff.new( employeeNumber:="XX1234567" );
oStaff1.surname = "Bloggs";
oStaff1.givenName = "Joseph";
oStaff1.hourlyRate = 55.00;
oStaff1.directAdd("projectList", oProject1);
oStaff1.directAdd("projectList", oProject2);

```

```

oCompany.directAdd("staffList", oStaff1);
oProject1.directAdd("staffList", oStaff1);
oProject2.directAdd("staffList", oStaff1);

oStaff2 = Staff.new( employeeNumber:="YY1111111" );
oStaff2.surname = "Doe";
oStaff2.givenName = "Jaqueline";
oStaff2.hourlyRate = 60.00;
oStaff2.directAdd("projectList", oProject2);

oCompany.directAdd("staffList", oStaff2);
oProject2.directAdd("staffList", oStaff2);

oStaff3 = Staff.new( employeeNumber:="XY3333333" );
oStaff3.surname = "Smith";
oStaff3.givenName = "Anonymous";
oStaff3.hourlyRate = 65.00;
oStaff3.directAdd("projectList", oProject2);

oCompany.directAdd("staffList", oStaff3);
oProject2.directAdd("staffList", oStaff3);

/* Create Project Phase instances
Add these to the Project Instances (each phase object can only belong to a single project)
*/
Phase oPhase1, oPhase2;

oPhase1 = Phase.new();
oPhase1.name = "Analysis and Specification";
oPhase1.longDescription = "Get user's business requirements down";
oPhase1.completed = TRUE;
oPhase1.phaseStart = Date.construct(1999,3,1);
oPhase1.estimatedPhaseFinish = Date.construct(1999,3,14);
oPhase1.actualPhaseFinish = Date.construct(1999,3,31);
oPhase1.estimatedHours = 50.0;
oPhase1.estimatedCost = 5000.00;
oPhase1.project = oProject1;

oPhase2 = Phase.new();
oPhase2.name = "Construction";
oPhase2.longDescription = "Build it - all the programming";
oPhase2.completed = FALSE;
oPhase2.phaseStart = Date.construct(1999,3,10);
oPhase2.estimatedPhaseFinish = Date.construct(1999,8,30);
oPhase2.estimatedHours = 200.0;
oPhase2.estimatedCost = 15000.00;
oPhase2.project = oProject1;

oProject1.directAdd("phaseList", oPhase1);
oProject1.directAdd("phaseList", oPhase2);

Phase oPhase21, oPhase22;

oPhase21 = Phase.new();
oPhase21.name = "Client Modelling System";
oPhase21.longDescription = "Y2K checking and certification";
oPhase21.completed = TRUE;
oPhase21.phaseStart = Date.construct(1998,6,1);
oPhase21.estimatedPhaseFinish = Date.construct(1998,12,31);
oPhase21.actualPhaseFinish = Date.construct(1999,1,31);
oPhase21.estimatedHours = 500.0;
oPhase21.estimatedCost = 35000.00;
oPhase21.project = oProject2;

oPhase22 = Phase.new();
oPhase22.name = "Internal Systems";
oPhase22.longDescription = "Y2K checking and certification";
oPhase22.completed = FALSE;
oPhase22.phaseStart = Date.construct(1998,12,1);
oPhase22.estimatedPhaseFinish = Date.construct(1999,10,31);
oPhase22.estimatedHours = 1500.0;
oPhase22.estimatedCost = 65000.00;
oPhase22.project = oProject2;

oProject2.directAdd("phaseList", oPhase21);
oProject2.directAdd("phaseList", oPhase22);

```

Running this script is a simple matter:

```
codqlie -execFile DataEntry01.odql
```

Or if the Interpreter was already running, I could use the line:

```
execFile DataEntry01.odql;
```

Listing 3 is the most interesting one, as it loops around, creating time entries for each phase of each project, for each person working on them according to the start and finish time of each phase. Phases without a finish date simply have time record to them until the current date. Ideally I wanted a random number generator to determine the hours values, but there isn't one in ODQL so I just fudged it using the modulus operand. This routine takes a while to run, but it crates several hundred entries – great fodder for writing test queries.

### ***Listing 3 (DataEntry02.odql): Creating a lot of Time instances***

```
/* File: DataEntry02.odql
Data entry script when there are no methods defined yet.
Create a series of instances of the Time Class.

v1.00 March 1999
Peter Fallon
Castle Software Australia
*/

defaultCF castleCF;

/* Get a list of all Projects: */
Bag<Project> oProjects;
Project oProject;
oProjects = Project from Project;

/* For Each Project, Get the staff and Phases */
scan(oProjects, oProject){

    /* for each person, create time entries for phases of the project */
    Staff oStaff;
    List<Staff> oStaffs;
    oStaffs = oProject.staffList;
    scan( oStaffs, oStaff ) {

        /* For each phase, create entries daily between Start and Completion Dates
        (or the current date if not completed) */
        Phase oPhase;
        List<Phase> oPhases;
        oPhases = oProject.phaseList;
        scan( oPhases, oPhase) {

            Integer iNumberOfDays;
            Integer z;
            Time oTime;

            if(oPhase.completed) {
                /* Phase is Completed - record time between the Start and Finish Dates*/
                iNumberOfDays = oPhase.actualPhaseFinish.difference(oPhase.phaseStart, DAY );
            }
            else {
                /* Phase is Not Completed - record time between the Start Date andToday*/
                iNumberOfDays = Date.getCurrent().difference(oPhase.phaseStart, DAY );
            }
            ;

            /* just to prove we were here... */
            oProject.name.print();
            oStaff.surname.print();
            oPhase.name.print();
            iNumberOfDays.print();

            /* For each day in this period, record hours for each personphase.Make
            the figure arbitrary, based on the day. Don't bother accounting for weekends */
            z=1;
            while ( z < iNumberOfDays ) {

                oTime = Time.new();
                oTime.whenDone = oPhase.phaseStart.add( z, DAY );
                oTime.timeSpent = (z % 7);
                oTime.cost = oTime.timeSpent * oStaff.hourlyRate;
                oTime.phase = oPhase;
                oTime.staff = oStaff;
                z = z + 1;
            }
        }
    }
};
```

## Experimenting with ODQL Code for Methods

The ODQL Interpreter is a rich environment for practicing your ODQL skills. You can sit at the command prompt and enter single commands a line at a time, or write scripts like the ones we've already used. Before you write the methods for your project, there will always be a few things you aren't sure about (Tuple syntax for certain queries will almost always be on my list you can be sure!). So let's run through a few things I did:

### Query Syntax

Queries are the heart of most Jasmine methods. Unless your classes maintain a lot of collection properties, then they are the only way to retrieve data for reporting, editing and maintenance.

What many of you may not know is that queries are far more powerful than the Database Administrator and Query Builder let you think. If you use the Query builder to construct a query, and then examine the ODQL generated, you will have something like:

---

```
Garment from Garment where Garment.season = "Summer";
```

---

This returns all instances of the **Garment** class (and its subclasses) where the **season** property contains the string "Summer". If you want to use the results of this query directly in Jasmine Studio, or the VB ActiveX interface, your query must always return a collection of objects as per the above. But the query in ODQL can do far more, just like an SQL query doesn't have to return all the columns of a table each time!

To fully discuss the Query construct, I'd need several pages, so I suggest you read the entry in the Jasref.hlp file for yourself. But keep in mind:

- You can return a collection of discrete value, or a collection of Tuples as well as objects
- Complex expressions can be used almost anywhere (including method calls)
- Queries can be based on Classes or other collections. Thus you can construct a query on the result of another. This is often done for optimisation reasons.
- In ODQL, you can then use various collection methods to manipulate the results, calculate sums, join two collections into one... etc

For example:

---

```
List<TT [String name, Bag<Person> manages, Integer seniority]> t1;
t1 = [ m.name(), m.manages(), m.seniority() ] from Manager m where m.sex == "F";
```

---

The above query returns a collection of Tuples. Each element contains a name, a collection of instances of the Person Class, and a numeric value. The collection only contains values for female managers in the Class Manager. Note that if we do use Tuples, we have to explicitly declare them (and this is where it gets tricky, but I'll show you how to solve that shortly).

### Group Syntax

If you think of the **Query** construct as the OO version of the SQL Select command, think of the **Group** construct as the OO version of using Select... .group by... in SQL. That is, it's a way of performing aggregate operations on groups of data quickly.

Go and re-read about queries. Nowhere can we SORT our data! Ok, we can use the **sort()** collection method on the results, as well as the **sum()** and other collection methods, but it's all after the fact. What if I want to sum the results from a method call, or do something more complicated? This is what the **Group** construct is about.

Group expressions can be used on classes, or on collections, like Queries. Unlike Queries however, Group does not support a where clause. This means that you almost always use Query and Group together, with the Group expression using the results of your Query.

The best way to elaborate is to show you...

### Experiments:

I wanted to look at my test data to check just what had been created, in broad terms. I wanted to see my Time data broken down by month/year with a total hours for each month. But the Interpreter kept rejecting my Group and Tuple definition.

Time to call up some assistance, in the form of the ODQL Class. This is a really nifty feature that allows you to create queries and Groups at runtime, have Jasmine tell you how the output will be structured (The Tuple definition), and will even run it for you if necessary.

Don't confuse this special Class with the ODQL language in general.

I entered the following at the command prompt in the Interpreter:

```
defaultCF castleCF;

ODQL.prepare( "test", "group t in Time by (t.whenDone.part( YEAR ), t.whenDone.part( MONTH ) )
with ( partition^timeSpent.sum() );");

ODQL.getOutputInfo( "test", "ODQL_STATEMENT").print();
ODQL.getOutputInfo( "test", "VARIABLE_DECLARATION", "oGroup").print();
```

And received the following back:

```
group t in Time by (t.whenDone.part( YEAR ), t.whenDone.part( MONTH ) ) with (
partition^timeSpent.sum() );

systemCF::Bag< 'oGroup'[ 'systemCF'::'Integer' 'C1', 'systemCF'::'Integer' 'C2',
'systemCF'::'Decimal'[18, 2] 'C3' ] > 'oGroup';
```

The first bit, in response to "ODQL\_STATEMENT" is just my query spat back at me. This was more for my sake, to ensure that I hadn't made a typing error. In fact, the first line, `ODQL.prepare()` would have generated an error of my query/group syntax was incorrect in any way.

The second line of output, in response to "VARIABLE\_DECLARATION" tells me how I should declare a Tuple variable to hold the results of this Group construct.

Let's put this to some use:

```
defaultCF castleCF;
systemCF::Bag< [ systemCF::Integer C1, systemCF::Integer C2, systemCF::Decimal[18, 2] C3 ] >
oGroup;

String.putString("Grouping time entries by Year/Month:");

oGroup = group t in Time by (t.whenDone.part( YEAR ), t.whenDone.part( MONTH ) ) with (
partition^timeSpent.sum() );

oGroup.print();
```

Printing the result of our Group produces the following:

```
Grouping time entries by Year/Month:
Bag{
  [C1: 1998, C2: 6, C3: 255.00],
  [C1: 1998, C2: 7, C3: 279.00],
  [C1: 1998, C2: 8, C3: 285.00],
  [C1: 1998, C2: 9, C3: 261.00],
  [C1: 1998, C2: 10, C3: 288.00],
  [C1: 1998, C2: 11, C3: 270.00],
  [C1: 1998, C2: 12, C3: 531.00],
  [C1: 1999, C2: 1, C3: 567.00],
  [C1: 1999, C2: 2, C3: 252.00],
  [C1: 1999, C2: 3, C3: 421.00],
  [C1: 1999, C2: 4, C3: 75.00]
}
```

As you can see, I built my data here in April 1999.

Now I have my confidence back, lets try something more ambitious and break these figure down by Project and Phase:

```
defaultCF castleCF;
systemCF::Bag< [ systemCF::Integer C1, systemCF::Integer C2, systemCF::String project,
systemCF::String phase, systemCF::Decimal[18, 2] C3 ] > oGroup2;

String.putString ("Grouping time entries by Year/Month/Project/Phase:");

oGroup2 = group t in Time by (t.whenDone.part( YEAR ), t.whenDone.part( MONTH ),
t.phase.project.name, t.phase.name ) with ( partition^timeSpent.sum() );

oGroup2.print();
```

Running this produces:

```
Grouping time entries by Year/Month/Project/Phase:
Bag{
  [C1: 1998, C2: 6, project: "Y2K Project", phase: "Client Modelling System", C3: 255.00],
  [C1: 1998, C2: 7, project: "Y2K Project", phase: "Client Modelling System", C3: 279.00],
  [C1: 1998, C2: 8, project: "Y2K Project", phase: "Client Modelling System", C3: 285.00],
  [C1: 1998, C2: 9, project: "Y2K Project", phase: "Client Modelling System", C3: 261.00],
  [C1: 1998, C2: 10, project: "Y2K Project", phase: "Client Modelling System", C3: 288.00],
  [C1: 1998, C2: 11, project: "Y2K Project", phase: "Client Modelling System", C3: 270.00],
```

```

[C1: 1998, C2: 12, project: "Y2K Project", phase: "Client Modelling System", C3: 270.00],
[C1: 1998, C2: 12, project: "Y2K Project", phase: "Internal Systems", C3: 261.00],
[C1: 1999, C2: 1, project: "Y2K Project", phase: "Client Modelling System", C3: 279.00],
[C1: 1999, C2: 1, project: "Y2K Project", phase: "Internal Systems", C3: 288.00],
[C1: 1999, C2: 2, project: "Y2K Project", phase: "Internal Systems", C3: 252.00],
[C1: 1999, C2: 3, project: "Patient Needs Trial Database", phase: "Analysis and
Specification", C3: 85.00],
[C1: 1999, C2: 3, project: "Patient Needs Trial Database", phase: "Construction", C3: 63.00],
[C1: 1999, C2: 3, project: "Y2K Project", phase: "Internal Systems", C3: 273.00],
[C1: 1999, C2: 4, project: "Patient Needs Trial Database", phase: "Construction", C3: 15.00],
[C1: 1999, C2: 4, project: "Y2K Project", phase: "Internal Systems", C3: 60.00]
}

```

Time spent learning just what you can do with Queries and Groups is time well spent, always. I agree this is a skill reserved for the database programmer who will write your methods. C Jasmine-API programmers also have access to this functionality for client applications, as do Weblink developers. The new client interfaces in Jasmine TND may also impact here (?).

### Method Definitions:

I'm a firm believer in the *suck-it-and-see* approach to programming. Planning and design is absolutely essential, but if you are still learning to cut code, the more you do the better, whether you are writing garbage or works of art.

Remember that to compile a method, any classes or methods it relies on must already be defined. Thus the simplest process to take when writing your methods is to start at the bottom of the dependency tree and work your way up. In my case, nearly everything relies on Time, so that's the first class I tackled.

Listing 4 shows my first attempts...

#### **Listing 4 (TimeMethods01.odql): Initial version of methods for Time Class:**

```

/* File: TimeMethods.odql

Method Definitions for Time Class

v1.00 May 1999
Peter Fallon
Castle Software Australia
*/
defaultCF castleCF;
/*
Method: AddTime

Adds a new time entry. If one is there matching those keys,
the hours are added to what is there. Returns success.

Only one instance of Time exists for key combinations...
*/

addProcedure Boolean castleCF::Time::class::addTime(Staff oStaff, Phase oPhase, Date dWhenDone,
Decimal [10,2] nHours )
    description: "Adds a new time entry. If one is there matching those keys, the hours are
added to what is there"
{
    $defaultCF castleCF;
    $Time oTime;
    $Bag<Time> oEntries;
    $Iterator<Time> itEntries;

    /* is there an existing entry? */
    $oEntries = Time from Time where Time.whenDone == dWhenDone and Time.staff == oStaff and
Time.phase == oPhase;

    $if (oEntries.count() == 0)
    {
        /* no existing entry - make one */
        $oTime = Time.new();
        $oTime.whenDone = dWhenDone;
        $oTime.timeSpent = nHours;
        $oTime.cost = oTime.timeSpent * oStaff.hourlyRate;
        $oTime.phase = oPhase;
        $oTime.staff = oStaff;
    }
    else
    {
        /* retrieve the existing entry - there will only be one */
        $itEntries = oEntries.createIterator();
        $itEntries.advance();
    }
}

```

```

        $oTime = itEntries.get();

        /* entry existings - Add to it... */
        $oTime.timeSpent = oTime.timeSpent + nHours;
        $oTime.cost = oTime.timeSpent * oStaff.hourlyRate;

        /* Tidy up... */
        $itEntries.delete();
    };

    $oEntries.delete();
    $return( TRUE );
};

/*
Method: UpdateTime

Creates a New time entry, replacing any existing one matching those keys.
use this when the external routines want to make a single update

Only one instance of Time exists for key combinations...
*/

addProcedure Boolean castleCF::Time::class:updateTime(Staff oStaff, Phase oPhase, Date
dWhenDone, Decimal [10,2] nHours )
    description: "Update time entry to specific value, overriding any existing one"
{
    $defaultCF castleCF;
    $Time oTime;
    $Bag<Time> oEntries;
    $Iterator<Time> itEntries;

    /* is there an existing entry? */
    $oEntries = Time from Time where Time.whenDone == dWhenDone and Time.staff == oStaff and
Time.phase == oPhase;

    $if (oEntries.count() == 0)
    {
        /* no existing entry - make one */
        $oTime = Time.new();
        $oTime.whenDone = dWhenDone;
        $oTime.timeSpent = nHours;
        $oTime.cost = oTime.timeSpent * oStaff.hourlyRate;
        $oTime.phase = oPhase;
        $oTime.staff = oStaff;
    }
    else
    {
        /* retrieve the existing entry - there will only be one */
        $itEntries = oEntries.createIterator();
        $itEntries.advance();
        $oTime = itEntries.get();

        /* entry existings - Change it... */
        $oTime.timeSpent = nHours;
        $oTime.cost = oTime.timeSpent * oStaff.hourlyRate;

        /* Tidy up... */
        $itEntries.delete();
    };

    $oEntries.delete();
    $return( TRUE );
};

/*
Method: DeleteTime

Removes the time entry matching those keys.

Only one instance of Time exists for key combinations...
*/

addProcedure Boolean castleCF::Time::class:deleteTime(Staff oStaff, Phase oPhase, Date dWhenDone
)
    description: "Delete a specific time entry"
{
    $defaultCF castleCF;
    $Time oTime;

```

```

$Bag<Time> oEntries;
$Iterator<Time> itEntries;

/* is there an existing entry? (Ignore if there isn't) */
$oEntries = Time from Time where Time.whenDone == dWhenDone and Time.staff == oStaff and
Time.phase == oPhase;

$if (oEntries.count() != 0)
{
    $scan( oEntries, oTime )
    {
        $oTime.delete();
    };
};

$oEntries.delete();
$return( TRUE );
};

/*
Method: getTimeHours

Retrieves the hours recorded matching those keys.
*/

addProcedure Decimal [10,2] castleCF::Time::class:getTimeHours(Staff oStaff, Phase oPhase, Date
dWhenDone )
description: "Retrieve hours for a specific time entry"
{
    $defaultCF castleCF;
    $Bag<Decimal [10,2]> oEntries;

    $oEntries = Time.timeSpent from Time where Time.whenDone == dWhenDone and
Time.staff == oStaff and
Time.phase == oPhase;

    /* is there an existing entry? */
    $if (oEntries.count() == 0)
    {
        $return(0);
    }
    else
    {
        $return(oEntries.sum());
    };
    $oEntries.delete();
};

/*
Method: getTimeCost

Retrieves the hours recorded matching those keys.
*/

addProcedure Decimal [12,2] castleCF::Time::class:getTimeCost(Staff oStaff, Phase oPhase, Date
dWhenDone )
description: "Retrieve costs for a specific time entry"
{
    $defaultCF castleCF;
    $Bag<Decimal [12,2]> oEntries;

    $oEntries = Time.cost from Time where Time.whenDone == dWhenDone and
Time.staff == oStaff and
Time.phase == oPhase;

    /* is there an existing entry? */
    $if (oEntries.count() == 0)
    {
        $return(0);
    }
    else
    {
        $return(oEntries.sum());
    };
    $oEntries.delete();
};

/*
Method: TotalProjectHours

```

Retrieves the hours recorded for a project in a specific month.

Note: yes, my query technique could use some optimising...

\*/

```
addProcedure Decimal [18,2] castleCF::Time::class:totalProjectHours( Project oProject, Integer
iYear, Integer iMonth )
```

```
  description: "Retrieve hours for a project/iMonth"
```

```
{
```

```
  $defaultCF castleCF;
```

```
  $Bag<Decimal [10,2]> oEntries;
```

```
  $oEntries = Time.timeSpent from Time where Time.whenDone.part(iYear) == iYear and
                                         Time.whenDone.part(MONTH) == iMonth and
                                         Time.phase.project == oProject;
```

```
  /* is there an existing entry? */
```

```
  $if (oEntries.count() == 0)
```

```
  {
```

```
    $return(0);
```

```
  }
```

```
  else
```

```
  {
```

```
    $return(oEntries.sum());
```

```
  };
```

```
  $oEntries.delete();
```

```
};
```

```
/*
```

```
Method: TotalProjectCosts
```

Retrieves the hours recorded for a project in a specific iMonth.

\*/

```
addProcedure Decimal [18,2] castleCF::Time::class:totalProjectCosts( Project oProject, Integer
iYear, Integer iMonth )
```

```
  description: "Retrieve hours for a project/iMonth"
```

```
{
```

```
  $defaultCF castleCF;
```

```
  $Bag<Decimal [12,2]> oEntries;
```

```
  $oEntries = Time.cost from Time where Time.whenDone.part(YEAR) == iYear and
                                         Time.whenDone.part(MONTH) == iMonth and
                                         Time.phase.project == oProject;
```

```
  /* is there an existing entry? */
```

```
  $if (oEntries.count() == 0)
```

```
  {
```

```
    $return(0);
```

```
  }
```

```
  else
```

```
  {
```

```
    $return(oEntries.sum());
```

```
  };
```

```
  $oEntries.delete();
```

```
};
```

```
/*
```

```
Method: TotalPhaseHours
```

Retrieves the hours recorded for a phase in a specific iMonth.

\*/

```
addProcedure Decimal [18,2] castleCF::Time::class:totalPhaseHours( Phase oPhase, Integer iYear,
Integer iMonth )
```

```
  description: "Retrieve hours for a project phase/iMonth"
```

```
{
```

```
  $defaultCF castleCF;
```

```
  $Bag<Decimal [10,2]> oEntries;
```

```
  $oEntries = Time.timeSpent from Time where Time.whenDone.part(YEAR) == iYear and
                                         Time.whenDone.part(MONTH) == iMonth and
                                         Time.phase == oPhase;
```

```
  /* is there an existing entry? */
```

```
  $if (oEntries.count() == 0)
```

```
  {
```

```
    $return(0);
```

```
  }
```

```
  else
```

```
  {
```

```

        $return(oEntries.sum());
    };
    $oEntries.delete();
};

/*
Method: TotalPhaseCosts

Retrieves the hours recorded for a phase in a specific iMonth.
*/

addProcedure Decimal [18,2] castleCF::Time::class:totalPhaseCosts( Phase oPhase, Integer iYear,
Integer iMonth )
    description: "Retrieve hours for a project phase/iMonth"
{
    $defaultCF castleCF;
    $Bag<Decimal [12,2]> oEntries;

    $oEntries = Time.cost from Time where Time.whenDone.part(YEAR) == iYear and
                                                Time.whenDone.part(MONTH) == iMonth and
                                                Time.phase == oPhase;

    /* is there an existing entry? */
    $if (oEntries.count() == 0)
    {
        $return(0);
    }
    else
    {
        $return(oEntries.sum());
    };
    $oEntries.delete();
};

/*
Method: TotalStaffHours

Retrieves the hours recorded for a person in a specific iMonth.
Note: yes, my query technique could use some optimising...

Note: There is no matching totalStaffCosts() method, as this is simply (totalStaffCosts() *
Staff.hourlyRate )
*/

addProcedure Decimal [18,2] castleCF::Time::class:totalStaffHours( Staff oStaff, Integer iYear,
Integer iMonth )
    description: "Retrieve hours for a person/iMonth"
{
    $defaultCF castleCF;
    $Bag<Decimal [10,2]> oEntries;

    $oEntries = Time.timeSpent from Time where Time.whenDone.part(YEAR) == iYear and
                                                Time.whenDone.part(MONTH) == iMonth and
                                                Time.staff == oStaff;

    /* is there an existing entry? */
    $if (oEntries.count() == 0)
    {
        $return(0);
    }
    else
    {
        $return(oEntries.sum());
    };
    $oEntries.delete();
};

/*
Method: TotalClientHours

Retrieves the hours recorded for a Client in a specific iMonth.
Note: yes, my query technique could really use some optimising...
*/

addProcedure Decimal [18,2] castleCF::Time::class:totalClientHours( Client oClient, Integer
iYear, Integer iMonth )
    description: "Retrieve hours for a client/iMonth"
{
    $defaultCF castleCF;
    $Bag<Decimal [10,2]> oEntries;

```

```

$oEntries = Time.timeSpent from Time where Time.whenDone.part(YEAR) == iYear and
                                           Time.whenDone.part(MONTH) == iMonth and
                                           Time.phase.project.client == oClient;
                                           /* very poor coding here I know... */

/* is there an existing entry? */
$if (oEntries.count() == 0)
{
    $return(0);
}
else
{
    $return(oEntries.sum());
};
$oEntries.delete();
};

/*
Method: TotalClientCosts

Retrieves the hours recorded for a client in a specific iMonth.
Note: yes, my query technique could **really** use some optimising...
*/

addProcedure Decimal [18,2] castleCF::Time::class::totalClientCosts( Client oClient, Integer
iYear, Integer iMonth )
description: "Retrieve hours for a client/iiMonth"
{
    $defaultCF castleCF;
    $Bag<Decimal [12,2]> oEntries;

    $oEntries = Time.cost from Time where Time.whenDone.part(YEAR) == iYear and
                                           Time.whenDone.part(MONTH) == iMonth and
                                           Time.phase.project.client == oClient;

    /* is there an existing entry? */
    $if (oEntries.count() == 0)
    {
        $return(0);
    }
    else
    {
        $return(oEntries.sum());
    };
    $oEntries.delete();
};

compileProcedure castleCF::Time;

```

This built and compiled beautifully!

Let's test it briefly in the Interpreter:

```

defaultCF castleCF;

Staff oStaff;
Bag<Staff> bagStaff;
bagStaff = Staff from Staff;
bagStaff = Staff from Staff;
scan (bagStaff, oStaff) { Time.totalStaffHours(oStaff, 1998, 11); };

```

And a correct result spat out. Great!

I've not made any attempt to optimise my code here, and I've no doubt some attention spent on indexes and a few benchmarks will work wonders and prevent deadlocks and other scaling issues, but that's beyond the scope of this presentation.

## Compiling and Testing

The following are pretty obvious comments, but I find I have to be reminded of them from time to time, you I'm doing the same for you.

- Build your methods in small lots. Compiler error messages are a pain to debug at present because the line number returned is the **real** line of C code that Jasmine sends to the compiler for a method, not the line of ODQL that you entered. As such, it doesn't hurt to do them a few at a time, and it certainly makes it easier to deduce where that stupid variable declaration you got wrong is!

- If/When you get runtime errors, don't despair over the message Jasmine gives you. Check out the Message.txt file that's in the %JAS\_SYSTEM%\Jasmine\Files\english directory. If your installation is not an English one, you'll find the file in one of the other parallel subdirectories. This file contains more helpful comments on the causes of each error.
- Test your queries to ensure they are returning the data you need, and check out their performance. If you think its slow, then there are many techniques you can employ. Check out the sessions on Optimisation at CA-World.
- Perform load testing (make sure you have enough test data in key areas – this is a common failing even for relational systems). Don't be afraid to try different things.
- Transactions. **ReadOnly** transactions will improve queries speed, but as the current release doesn't allow nested transactions you have to be especially careful how you apply these. Be as black box as possible.
- Test methods in the interpreter as they are compiled. If nothing else, this will improve your ODQL skills no end!

## More Methods

At this point, I'll mention for the people reading this paper that my presentation went on to run the scripts for methods for each of the classes in my database. To include those here would not be practical.

I will discuss one other set of methods however. I wanted a method that would output a monthly report of sorts. ODQL has some string handling methods, but by and large its abilities in this area are rudimentary. Obviously one of the reporting tools now available for Jasmine will be far better suited to this, but I wanted to restrict my code to pure ODQL for demonstration purposes. The result is in Listing 5

### **Listing 5 (testMethods03.odql): Trying my hand at an ASCII report.**

```

/* File: TestMethods03.odql
   Sample Report of Phase activity for a Month/Year

v1.00  June 1999
Peter Fallon
Castle Software Australia
*/

defaultCF castleCF;
Integer iYear, iMonth;
Phase oPhase;
List<Phase> oPhases;
Bag<Time> bagTime;
Bag< testquery[ Staff oStaff, Decimal[18,2] nHours, Decimal[18, 2] nCosts ] > oPhaseActivity;
testquery[ Staff oStaff, Decimal[18,2] nHours, Decimal[18, 2] nCosts ] oMonth;

iYear = 1999;
iMonth = 3;

/* Construct phase List */
oPhases = Phase from Phase;

/* For each Phase - get time entries for specified Month/Year
   and then produce a report of time entries by staff */

String.putString( "Testing code for Report\n-----\n" );
Date.getCurrent().print();
String.format( " Reporting Time Entries for Month %2s, Year %4s", iMonth, iYear).print();

scan( oPhases, oPhase ) {

    bagTime = Time from Time where Time.phase == oPhase and
                Time.whenDone.part(YEAR) == iYear and
                Time.whenDone.part(MONTH) == iMonth;

    oPhaseActivity = group t in bagTime by (t.staff) with ( partition^timeSpent.sum(),
partition^cost.sum() );

    String.format( "\nProject: %s, Phase: %s", oPhase.project.name, oPhase.name).print();

    if (oPhaseActivity.count()==0) {
        String.putString( "No time recorded for this phase in this month.\n");
    }
    else {
        String.putString( "Name                Hours    Costs\n-----\n" );
        scan( oPhaseActivity, oMonth ) {
            String.format( "%20s %7s %8s", oMonth.oStaff.surname, oMonth.nHours, oMonth.nCosts
).print();

```

```

    };
};
};

```

ODQL as a scripting language has no real ability to get user input for parameters or other runtime variables, so I just had to hardwire March 1999 as my report date, and also get it to report for all Phases. This is less important than the attempt to present and format data in some fashion from Queries and Group functions. The result was:

### ***The results of our TestMethods03.odql script:***

Testing code for Report

-----  
07/02/1999

Reporting Time Entries for Month 3, Year 1999

Project: Patient Needs Trial Database, Phase: Analysis and Specification

Name	Hours	Costs
Bloggs	85.00	4675.00

-----

Project: Patient Needs Trial Database, Phase: Construction

Name	Hours	Costs
Bloggs	63.00	3465.00

-----

Project: Y2K Project, Phase: Client Modelling System

No time recorded for this phase in this month.

Project: Y2K Project, Phase: Internal Systems

Name	Hours	Costs
Bloggs	91.00	5005.00
Doe	91.00	5460.00
Smith	91.00	5915.00

-----

Well, as I said before, ODQL can't do too much, but this is sufficient information for a multi-line text box in Jasmine Studio if quick feedback is required.

Converted to an instance method of the Phase class, with variables replaced by parameters, and print() calls replaced by adding results to a string variable (the return value of our method), we end up with:

### ***Listing 6: Method Phase::monthReport()***

```

addProcedure String castleCF::Phase::instance::monthReport( Integer iYear, Integer iMonth )
    description: "Retrieve costs for this phase in a given month"
{
    /*
    Method: monthlyReport
    Report of all activity on this phase for the given Month/year.
    Return a formatted multi-line string
    */
    $defaultCF castleCF;
    $Bag<Time> bagTime;
    $Bag< testquery[ Staff oStaff, Decimal[18,2] nHours, Decimal[18, 2] nCosts ] >
oPhaseActivity;
    $testquery[ Staff oStaff, Decimal[18,2] nHours, Decimal[18, 2] nCosts ] oMonth;
    $String sReturnValue;

    $sReturnValue = String.format("Time recorded for Month:%2s, Year:%4s\n", iMonth, iYear);

    $bagTime = Time from Time where Time.phase == self and
                Time.whenDone.part(YEAR) == iYear and
                Time.whenDone.part(MONTH) == iMonth;

    $oPhaseActivity = group t in bagTime by (t.staff) with ( partition^timeSpent.sum(),
partition^cost.sum() );

    $sReturnValue = sReturnValue.stringCat(
        String.format( "\nProject: %s, Phase: %s\n", self.project.name, self.name ) );

    $if ( oPhaseActivity.count()==0 ) {
        $sReturnValue = sReturnValue.stringCat( "No time recorded for this phase in this
month.\n" );
    }
    else {
        $sReturnValue = sReturnValue.stringCat( "Name                Hours    Costs\n-----
-----\n" );
        $scan (oPhaseActivity, oMonth ) {
            $sReturnValue = sReturnValue.stringCat(
                String.format( "%20s %7s %8s", oMonth.oStaff.surname,
oMonth.nHours, oMonth.nCosts ) );

```

```

    };
};

$return( sReturnValue );

/* housekeeping... */
$oPhaseActivity.delete();
$bagTime.delete();
};

```

## The Importance of Being Scripted

I've mentioned before that the reason to use the ODQL Interpreter for class and method development is that you have a written record of development. This means:

- You can use source code management and version control tools to record your database schema in full
- You have a chronological record of all changes.
- In case of failure you can restore from backups, and then make any required schema changes simply by running the last few scripts used again.
- Migrating to another (database) server is easy, so is reconstructing a database from scratch.
- The Dev/Test cycle almost always involves stripping and rebuilding a database several times. This gets real tedious if you don't have something like Listing7:

### Listing 7 (CreateEverything.bat): My Master Database Script.

```

@echo off

rem      Master batch file for creating the entire database, executing all scripts
rem      and batch files in the correct sequence. The only thing that needs editing
rem      is specific Class Family stuff related to Aliasing.

rem      (Assume LOCAL connection is the default one)

rem      Create stores and Class families:
call SetupStores
copy local.env %JAS_ENVFILE%

rem      Define and build all Classes
codqlie -execFile Classes.odql

rem      Construct some test data!
codqlie -execFile DataEntry01.odql
codqlie -execFile DataEntry02.odql

rem      Define and compile Methods for classes
codqlie -execFile TimeMethods2.odql
codqlie -execFile StaffMethods.odql
codqlie -execFile PhaseMethods.odql
codqlie -execFile ProjectMethods.odql
codqlie -execFile ClientMethods.odql
codqlie -execFile CompanyMethods.odql

```

Remember, once you create the database, use JasUnload (or Unload) to copy it in its entirety (with and without instance data). You can then replace all the execFile commands above with a single JasLoad, assuming you don't want to change anything.

Correctly dated and commented, you leave a complete trail for maintenance and support to follow.

There is one further point that must be made here, that related to what I've covered so far as well as application development. While you are constructing your classes, start thinking about your **backup and upgrade strategy** for your database. That is, read the migration notes that come on the Jasmine CD about what to do when a new version of Jasmine is released. It's essential you know how to cope with:

- Changes to built in Jasmine Classes. The multimedia classes are the main ones affected. If you store Multimedia data, then two things are vital: Store all data in your own classes, don't use the Bitmap and Movie classes, create your own subclasses such as CABitmap and CAMovie. This allows you to discretely back them up using JasUnload

The second option recommended is to create separate copies of the Multimedia Class Family for each application and use Aliasing to point to the one required for each app'. This means problems or issues with one app will not affect others. You will have to recopy the core mediaCF class family again for any upgrade, but it means you retain complete control over the situation. I show some example batch files for this later.

The other Classes most affect are Jasmine Studio (jadelibCF). Again, it's best to separate each Studio application into its own CF, and use aliasing to keep them separate.

- Start working on your batch files and scripts to do all of this as soon as your basic classes are built. Have your procedures automated and ready to go at a moments notice. Again, source and version management tools will integrate extremely well here.
- Plan your backup strategy. Understand how you will journal and backup stores. Understand the dependencies between stores according to instance data and make sure the right files are backed up in sync.
- Upgrades to the database engine are something you put a great deal of planning into when a production system is involved. Treat this just as seriously as you would upgrading a relational engine.
- Test all of the above and retest with data, and retest as soon as you start acceptance testing (ie: with real data). Having a strategy in place is nice, but having one that works is sorta important!

## Building an Application

Just how far I get with this during the presentation will depend on time and questions. I intend to show a few skeletons of client application using this database, but nothing polished as my focus was the database itself.

I do want to cover a few specific topics though:

### Creating Duplicate of jadelibCF and mediaCF

So how do we copy jadelibCF and mediaCF?

You need to read the help listings for the Jasunload and Jasload commands carefully and understand what they are doing for you. But this is the batch file I used (note that these commands are rather long, I've indented the second and successive lines in print to make it easier to read. In the original batch file each Jasunload or Jasload command forms a single line):

---

```
rem Jasmine must be running here!
jasunload %JAS_SYSTEM%\unload\jStudioBackup.uld -L %JAS_SYSTEM%\unload\jStudiomethods.lib -m
          %JAS_SYSTEM%\unload\jStudioMedia.uld -M mediaCF1 jadelibCF1 mediaCF1
```

---

The above creates three files, The class and instance data is about 675K, the methods about 1.5M, and the multimedia instances about 25M. Note that when making the initial copy, I need the multimedia data as instances are referenced in jadelibCF –and we can't separate those easily. So its better to copy the low, and cull data after we have deposited it elsewhere.

Here how I created new Class Families (you have to do this manually), and then loaded them with Studio data:

---

```
rem Jasmine must be running here!
createCF jadelibCF11 CastleDemo01
createCF mediaCF11 CastleDemo01
jasload %JAS_SYSTEM%\unload\jStudioBackup.uld -l %JAS_SYSTEM%\unload\jStudiomethods.lib -m
          %JAS_SYSTEM%\unload\jStudioMedia.uld -M mediaCF1 -a %JAS_SYSTEM%\jasmine\data\mmarea
          jadelibCF1=jadelibCF11 mediaCF1=mediaCF11
```

---

Note the `jadelibCF1=jadelibCF11 mediaCF1=mediaCF11` part. This is the most important option. It tells the load routines to change all references of `jadelibCF1` and `mediaCF1` to `jadelibCF11` and `mediaCF11` respectively. Its essential to backup and restore the two class families together to maintain object integrity throughout.

If you use multimedia data in your own database, you have to be similarly careful when unloading and loading your application. Starting with an independent `mediaCF11` is a great help!

If I do a Liststore now, I'll see the following for my CastleDemo01 Store:

---

```
==== S T O R E   C O N T E N T S =====
Store Name: CastleDemo01
Class Families:
    mediaCF11
    jadelibCF11
    castleCF01
Locations:
    i:\jasmine\jasmine\data\castle_1
    i:\jasmine\jasmine\data\castle_2a
    i:\jasmine\jasmine\data\castle_2b
Page size: 8192
Total pages: 3216
Used pages: 512
```

---

Ok, the final step is to adjust my environment file. Currently its:

---

```
bufferSize    1024
workSize      20480 10240
CF mediaCF    mediaCF1
```

---

---

```
CF jadelibCF  jadelibCF1
CF castleCF  castleCF01
```

---

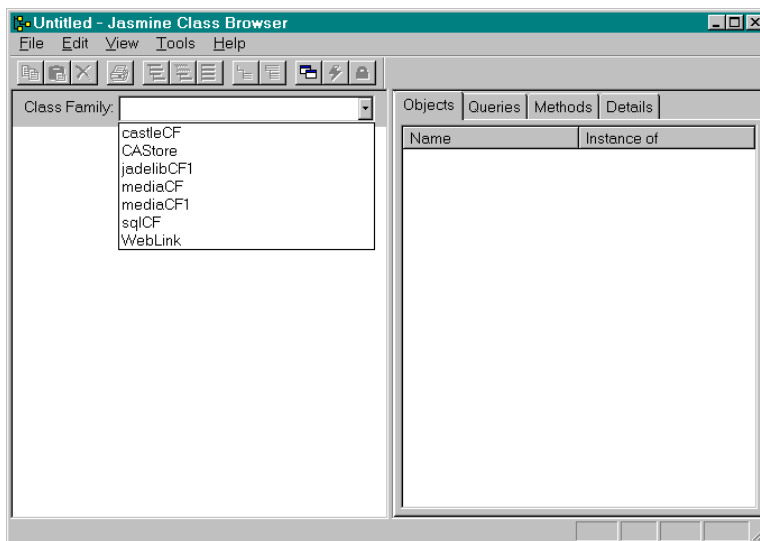
But I now want it to be:

---

```
bufferSize  1024
workSize    20480 10240
CF mediaCF  mediaCF11
CF jadelibCF jadelibCF11
CF castleCF castleCF01
```

If my connection uses this ENV file, any references to Jasmine Studio or the Multimedia classes will be directed to my new copies of them. If I need to go back and use the original class families at any time, I can restore the old file, or just set up a different connection to use it. The two, and their respective instance data and multimedia files are completely independent of each other.

Figure 7 shows something interesting. After the above has occurred, we can see the original class families **jadelibCF1** and **mediaCF1**. If we browse them we find their references are explicitly to each other. Meanwhile our new **mediaCF11** is of course aliased to **medaCF** and we cannot see a reference to the new **jadelibCF** as it's considered an internal Class Family to Studio like **system** (not subject to user fiddling).



**Figure 7: The Database Administrator after duplication.**

Let's try for the whole hog. You may have done the Studio Tutorial where you place multimedia items in certain resource classes inside the CAStore Class Family. From there you can drag and drop items onto scenes and Studio creates the appropriate widget for each item. Well since we want to be independent of the CA samples, let's copy those classes as well and put them into our own database castleCF.

First we unload them, just the class definitions this time. There aren't any methods here, but we do have to specifically name each class, as we want a portion of the CAStore Class Family, not the whole lot:

---

```
jasunload %JAS_SYSTEM%\unload\CAStoreStuff1.uld -s -c CAStore::Resource CAStore::Control
CAStore::MenuItem CAStore::ActivationObject CAStore::Language CAStore::Button CAStore::Label
jasunload %JAS_SYSTEM%\unload\CAStoreStuff2.uld -s -c CAStore::Decoration CAStore::Background
CAStore::Pictures CAStore::Sounds CAStore::Movies
```

---

I added to `-c` option above so the resulting unload files would have comments. The Top class in this hierarchy is **Resource**, but it is a subclass of **CAComposite**. I want to move it into **castleCF**, which doesn't have a **CAComposite**, so I need to change the **CAComposite** reference:

---

```
# Super Class Information
# cfname,clsname
"CAStore", "CAComposite"
```

---

to the composite class I'm using: **CastleComposite**:

---

```
# Super Class Information
# cfname,clsname
"CAStore", "CastleComposite"
```

---

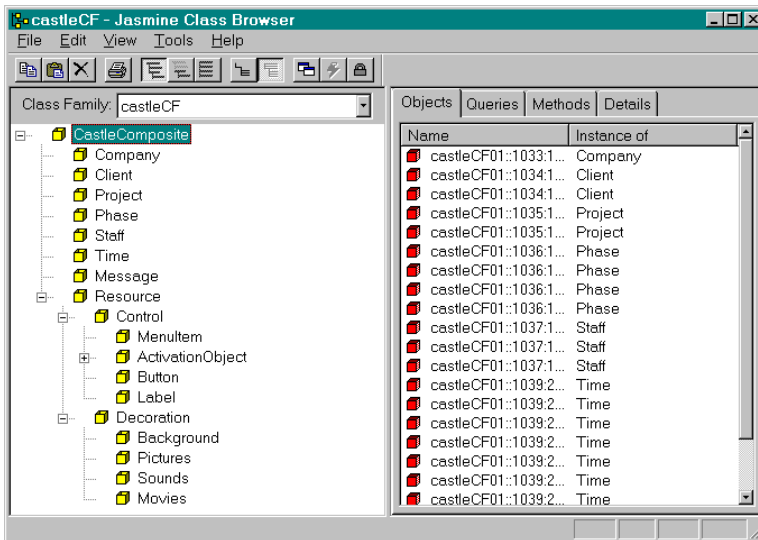
Notice that I'm leaving the Class Family reference alone. That's because I'm letting the Jasload command take care of those:

---

```
rem Jasmine must be running here!
jasload %JAS_SYSTEM%\unload\CAStoreStuff1.uld -s CAStore=castleCF01
jasload %JAS_SYSTEM%\unload\CAStoreStuff2.uld -s CAStore=castleCF01
```

---

And done! Figure 8 shows the class hierarchy in my Class family now:



**Figure 8:** I'm now set for Jasmine Studio development independent of all the sample code.

Using the above commands and batch files is not difficult, but I do suggest you get very familiar with the Jasmine documentation. Secondly, follow my recommendations in the "Importance of Being Scripted" section earlier and make sure you have backups before, during and after you try all of this, and plan for it as soon as possible in development (certainly BEFORE any Jasmine Studio development begins!!!).

## Conclusion

Before you say "What, isn't there more... ?", let me agree with you. However to treat any one of the subjects covered above properly would require an entire book! My purpose here was to provide a quick overview with solid examples.

Please don't be disillusioned by the detail I've shown you. Jasmine IS an easy product to use, especially compared to some relational products about. Your biggest hurdle may be understanding object oriented techniques, in analysis, design and programming. Much of the Jasmine advertising you will have seen focusses on client development gains provided by Jasmine Studio... etc. These gains are still real compared to client development with other products. They don't describe the back end server work simply because this cannot be scoped without first finding out exactly what you are doing. It's a "how long is a piece of string" argument. An as I've said already, **this work is no different in scope to that required by any other server product on the market.**

But I do believe that you will end up with a better solution using Jasmine. Read my paper titled "Why Objects, and Why an Object Database?" if you want some ideas on when object databases are a good idea.

## Author:

Peter Fallon is the director of Castle Software Australia Pty Ltd, a company providing custom software development and consulting. He has been developing and supporting PC database applications since 1986, ranging from small business systems to large corporate installations. [www.castlesoftware.com.au](http://www.castlesoftware.com.au)

Peter has spoken on topics covering both programming and project management at CA-Technicons 1992, 1993, and 1994 in the USA, and OO and Jasmine topics at IngresWorld 97 and Harmony 98 in Melbourne Australia. He was the editor of "Clippings", the magazine for Clipper User Groups in Australia and New Zealand for 3 years and Vice-President of the Clipper User Group (Melbourne) in 1992. Currently Peter is working on a number of Jasmine products and training services as well as various Visual Basic and Access development projects. He can be reached at [peterfallon@castlesoftware.com.au](mailto:peterfallon@castlesoftware.com.au)

## Copyright Notice

No part of this paper may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the authors. For information please contact: Castle Software Australia Pty Ltd.

**Trademark Acknowledgements**

Jasmine™ is a trademark of Computer Associates, CA-Visual Objects 2.0® is registered trademark of Computer Associates, Microsoft® Visual Basic® is a registered trademark of Microsoft and Microsoft® Office 97™ is a trademark of Microsoft.

All other product names and services identified throughout these notes are trademarks or registered trademarks of their respective companies. They are used throughout these notes in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with the notes.