Javelin

5

Where do I Start?

Working With Classes

Development Strategy

InSync Code Generation Technology

General

Editing Classes

Javelin provides quick access to <u>class details</u>, <u>member data and member functions</u> It also provides access to the <u>inheritance</u> properties of a class.

This information is broken up into three categories: Details, Members, Inheritance, each of which is explained in the relevant section of this manual.

To edit a particular category:

1. Click the right mouse button on the class you desire to edit.
2. A floating pop up menu will appear. Choose the category that you wish to edit for this class.
3. The appropriate dialog box will now appear. View/Modify or modify as necessary.

OR

To go straight to a class' members dialog box:

1. Double click on the class in the design slate.

Object Databases

Object Databases differ greatly from traditional relational databases. Object Databases have the unique benefit of being able to store and load objects as objects instead of having to break them down into their component parts.

If you are interested in using an object oriented database to store your persistent objects then contact Step Ahead Software Pty Ltd (+61 2 477 3998) for advise on the most appropriate choice of ODBMS for your use.

empty space

Empty Space is any space on the design slate that is not occupied by a class, relationship, text or other object.

Pricing and Ordering

If this is an evaluation version of the product then you can find pricing and purchasing information in the file order.txt that was installed with the application. This file contains a product order form.

VISA/MASTERCARD and Bankcard purchases may be made by phone or fax.

Call:

From USA 1800 210 9427 (toll free)
Outside USA +61 2 477 3398

to place your order

Extra contact points

For more information you may also contact us via:

Email: stepsoft@ozemail.com.au
Web Site: http://www.ozemail.com.au/~stepsoft

Javelin Technical Support

Tech Help

For technical support the following avenues are available:

Online Help.

Printed Tutorial.

Your Step Ahead Software dealer.

Step Ahead Software at stepsoft@ozemail.com.au or compuserve: 100241,3111.
Please quote the product name and version number, your name and your company name and a description of the problem plus any same data that may be required to reproduce the problem.*

Our Web Site: http://www.ozemail.com.au/~stepsoft

Please register your copy now if you haven't already done so. This will make it possible for us to give you advance notice of upgrades and you will be elligable to receive the registered version which doubles the class/member limits from 12 to 24.

For user feedback feel free to contact Step Ahead Software Pty Ltd Direct via email: stepsoft@ozemail.com.au OR Compuserve: 100241,3111

*Support arrangements are subject to change without notice.

Import

Javelin provides an import facility which you can use to import existing classes and their code

Once the import process is complete you can start working on a higher plane by making future changes from Javelin, saving time and effort and maintaining synchronization between your Javelin design database and the code using InSync code synchronization technology.

IMPORTANT:

1. Prior to any import session you must backup your entire application and all its support files (.java for Java and .h, .cpp, .vcw, .prj, .mak etc., for C++)
2. After importing you must verify that the imported classes and code perform equivalently to the previous version. Javelin makes certain assumptions and can rearrange certain code constructs differently to the original code. See below.

ADVICE: If you have a large project to import we advise that you to break it down into logical file groupings stored in subdirectories. Store all files for associated classes in the the same subdirectory. You can then approach the import process in a piecemeal way if you wish - importing, integrating and verifying. Even before the whole project is imported you will be able to utilize the benefits of Javelin for those classes that have been imported. You should read Software Quality for more details.

Import Process

The steps involved in the import process are simple:

1. Read Importing Constraints and Assumptions and inspect the code to be imported for compliance to these assumptions
2. Build an import specification file. This specifies which files will be implemented. See Building and Import Specification File
3. Once you have built the required import specification file you can perform an import process by selecting File|Import....
4. Select the .imp file you wish to use for your import session.

Javelin brings up a window showing the status of the import and adds the classes to the design slate as it progresses.

The classes will be arranged starting from the left and moving across to the right. When the right edge of the design slate is reached the next class will appear at the left edge and down on the next row. You can easily arrange them properly by dragging them to their appropriate positions. If you inadvertantly press on the top section of a class and the cursor changes to the "create relationship" cursor then simply click on another class and press cancel on the dialog box that appears.

See also

Importing Constraints and Assumptions

Building an Import Specification File

<u>Software Quality</u>

Creating Inheritance Relationships

You can specify that a particular class inherits from another easily using drag and drop:

1. Press the left mouse button on the top section of the class which is to inherit features of another class.
2. Drag the mouse to the class which is to be a super (base) class. You can release and press the mouse at various points to create "bends" in the line.
3. Release the mouse button and select extends (inheritance) from the pop up menu.

Javelin will draw a line from the top of the sub (derived) class to the bottom of the super (base) class. See Relationships for more information on working with relationships.

`C++`

By default inheritance relationships are created with public access to the base class and they do not inherit the base class virtually (as in virtual specifier for base classes in C++ - relevant to multiple inheritance)

Step Ahead Software - *Keeping you a step ahead!*

Access

Each member of a class module is assigned a level of access. In Javelin the access levels are Public, Protected, Private and Non Member.

The first three levels are identical to their C++ meaning. Non Member access is assigned to members which are to be declared outside of the class declaration but within the class module. Examples of non members are global variables, non member functions etc.,

Creator Database

The database which Javelin uses to store and retrieve design information entered by the user. Class names, member types., etc., are all stored in the creator database. The information in the database is used at various times by Javelin. At code generation time, for instance, Javelin extracts information from the creator database in order to generate code.

Function Body

The code written between the opening { and closing } of a function.

Function Stub

A function stub is a piece of code defining the type, name and parameters of a function but with an empty function body. For example:

```
int
ChangeStyle(int NewStyle)
{
}
```

is a function stub. Javelin generates function stubs in the .cpp files when a function is added to the design. Changes made to the function body in third party editors are preserved so long as the .cpp file is saved before Javelin does the next regeneration.

Member

A member is a data or function declared in a class module. This term covers both members declared within the class declaration itself and members declared outside of the class declaration.

Module

A module is any separately compiled set of code, usually stored in a file with extension .cpp. Javelin uses the convention that there is one module for each class.

Scope

Scope is the term used to describe the "visibility" of a piece of data or a function. For example if a variable is declared at the start of a function then that variable is said to have scope only within that function. Outside of that function it has no scope: it is not visible. Global variables can have scope throughout an entire application. Private members of a class only have scope within member functions of that class.

Introduction to Javelin/Visual Classworks

Getting Started

To get started with Javelin it is recommended that you first work through the Getting Started Tutorial after reading this introduction. The tutorial can be opened by double clicking on the Getting Started icon in the Javelin program manager group.

Javelin, What is it?

Javelin is a graphical tool for the development of object oriented JAVA code. Javelin makes full use of the Windows environment to take developers to a new productivity high with its powerful, intuitive and yet easy to learn graphical interface. It's sister product, Visual Classworks, provides similar features for C++ developers. Due to the similar nature of Java and C++ and the Javelin and Visual Classworks products they both share this common on line help file though the getting started tutorials are separate for each product. The word Javelin will be used to collectively refer to Javelin and Visual Classworks.

Throughout this on line help system the following icons help to indicate Javelin and Visual Classworks specific features:

java   Indicates JAVA or Javelin specific features
C++   Indicates C++ or Visual Classworks specific features

If you can use a mouse you can drive Javelin!

At one level Javelin appears more like a drawing package tailored for object oriented modeling but as you explore more deeply you will discover that its powers transcend far beyond that of a mere drawing package or object browser. Almost every object on the design slate can be manipulated by the left and/or right mouse button.

Feature Summary

You add new classes with a click of the right mouse button. You can drag and drop between classes to establish relationships between classes (Inheritance, References). You can view/add/modify/delete members of a class by selecting the Members option after clicking the right mouse button on a class.

Javelin can generate a significant amount of the code required to implement the design, saving the programmer much time and effort. It uses the powerful InSync code regeneration technology which preserves changes which programmers make to the code within function bodies and certain other sections of the source files.

Javelin is designed from the ground up to remain with projects throughout their life cycle and serves as a central depository of information related to the various components of the project. By using it throughout all phases of a project's life cycle Javelin, by nature, serves a dual purpose as a valuable integrated, dynamic project documentation tool in a way that no unintegrated documentation tool can. As the design changes so does the documentation. The integrated phase tracker lets you track what phase each component of your design in at so you'll have some sense of the progress that you've made.

The fact that the design and specification of the class hierarchy and its documentation are integrated means that by using Javelin throughout a project's life cycle it is easy to maintain up to date, relevant documentation. Accurate documentation is a key need especially if someone else has to understand or modify your code. Javelin can also create professional hard copy documentation on your Windows

graphics printer.

C++

Visual Classworks also supports the generation of class declarations compliant with the Object Oriented Database POET(TM) and others. When you're ready to start utilizing the power of Object Oriented Databases Visual Classworks can be told to generate code for POET. Step Ahead Software is planning to release a version with built in code generation for the OMG (Object Management Group) standard when these databases arrive. Until that time Visual Classworks can still be used to generate a variety of object base class declarations for databases other than POET using the customizable reference code generation strings.

Design Slate

The design slate and your mouse replace the paper and pencil that an analyst/programmer may have traditionally used in the analysis and design phase of an object oriented project. The slate is a graphical editor which allows the design and refinement of your application to take place on screen. The classes are shown as rectangles with the class name appearing inside. Inheritance relationships are shown as a line drawn from the top of the derived class to the bottom of the base class. Classes marked as being library classes are shown in grey. Normal classes are shown in green.

Viewing the Classes in the Slate

It is possible to view the classes in the slate in various ways. You can zoom in on sections of the slate to focus on a particular group of classes. You can fit all classes on the slate and it is also possible to see the entire slate. You can move classes and hide or show the inheritance and/or reference relationships.

The Slate Map window can be used to see an overall view of your classes. You can use it to change the zoom setting or change the area which is exposed in the design slate window.

Slate Right Click Menu

You can add a new class or a text frame at the current cursor position by pressing the right mouse button on empty space in the slate. Select the appropriate option from the menu that pops up. The width of a text frame can be adjusted by dragging the size bar on the right of it.

Zoom In

The term zoom in Javelin refers to the expansion of a particular section of the slate so that you can see it in greater detail. To zoom in on an area in the slate which is currently visible:

1. Press and hold the left mouse button on the top left corner of the area you wish to zoom in on. It must not be over any object.
2. Drag the mouse to the bottom right corner of the area to be zoomed.
3. Release the mouse button. The rectangle that you made with the drag operation will be displayed as large as Javelin can manage and still fit all the required area in the slate.

Fit All

A "fit all" operation will scale the slate such that all objects, no matter where they are located on the slate, will appear on the screen.

To perform a "fit all" operation:

1. Double click the left mouse button on any empty space in the slate or select View | Fit All.

NOTE: With large class hierarchies the rectangles shown may be quite small. To see a particular area in greater detail simply perform a Zoom In operation on the required area of interest. This can be done either directly in the design slate or via the Slate Map.

Without zooming you can find the name of a class by moving the mouse over it. Its name will appear in the tool bar.

Show Entire Slate

The slate is of a fixed size. To see where your classes are located with respect to the borders of the slate perform a Show Entire operation and the entire slate will be scaled to fit in the window. This is an option in the View menu.

View Inheritance, References

Use these menus to toggle the visibility state of Inheritance and Reference relationships.

View Progress

Class phases are shown as part of the class icon when View|Show Progress is checked.

Basic Class/Interface Operations

Java has a concept of both classes and interfaces. C++ supports only clases. In many places throughout this documentation the word class is used to include both classes and interfaces in the case of Java. The operations that can be performed on classes in the design slate are:

Adding a class

Selecting Classes

Moving Classes

Editing Classes

Deleting a Class

Showing Members in Class Icon

Class Details

Viewing/Modifying Class Details

The class details can be viewed or modified using the class details dialog box. To access to this dialog:

1. Press the right mouse button on the desired class.
2. A pop up menu should appear. Select the Details option on the pop up menu.
3. A dialog box will appear showing you the current values of the class details. You can modify these and then press Ok.

The fields of the class details dialog and their purpose are listed below:

Name

Type the name of the class in this field. This name is used by various other components in Javelin to build unique identifiers based on the name.

NOTE: You should always name your classes as singular and not plural. For example you should not call a class for Person objects Persons. It should be called Person.

C++ Filename

This specifies the filename Visual Classworks will use when generating the header and implementation files. The prefix given in Path will be added and the extensions .h and .cpp will be suffixed to this name to create the full filename for code generation of a particular module.

java JAVA filenames are dictacted by the name of their public class

Path

Specifies the prefix that will be added to the filename and extension when generating files for classes.

java Override

Override dictates the type of "extends" relationships that can be built with this class. An abstract class can never be instantiated. A final class can never be "extended". A normal class can be both instantiated and extended.

C++ Abstract Class

Certain streamability functions should not be generated for abstract classes. The build function, for example, normally creates an instance of a class but if the class is an abstract class (ie., it contains pure virtual functions or does not redefine a pure virtual function in a base class) then this class will not compile successfully. Check this box for any abstract classes you have added to Javelin.

Library Class, use as reference only.

Check this check box if you don't want any code to be generated for this class. This is used when you want to add classes to Javelin to represent classes provided in class libraries such as Borland C++ Object Windows library or Container Class library. You obviously don't want code generated for classes that already exist but you would like them to be represented in your Javelin project so that you can create classes that derive

from them.

C++ Object Identity

Check this check box if you wish Javelin to generate a set of five functions which are declared pure virtual in the Object class of the container class library. The functions are:

```
virtual classType isA() const { return MyClass; }
virtual char *nameOf() const;
virtual int virtual hashValueType hashValue() const;
virtual void printOn(Rostream outputStream) const;
```

This feature is very useful in many cases when creating classes derived from the "Object" class in Borland's container class library. It saves you the effort of adding these functions manually. Function stubs are created for most of these functions although isA and nameOf in most cases need not be changed.

Development Phase

This combo box allows you to specify the current development phase of the class. This is represented by a phase graphic.

C++ Extras Sections

Extra .h includes

Anything you enter into this field will be copied to the .h file prior to the declaration of the class. This can be used to create typedefs or enumerated types which need to be used in the class declaration. It can also be used to #include .h files that contain types, structs, enumerations etc., which are required by your class.

Class Extras

Anything you enter into this field will be generated immediately after the opening curly bracket of the class declaration. Examples of uses for this include declaring functions or classes as being friends of the given class.

After Class

Anything you enter into this field will be placed in the keep section after the class declaration. You can use this to specify indexes etc., if you are using POET or some other object database.

Extra .cpp includes

Anything you enter into this field will be copied to the .cpp file prior to the functions of the class. This can be used to for adding extra C++ code for a particular module.

java Extras Sections

Before Class

Anything you enter into this field will be copied to the .java file prior to the declaration of the class. This can be used to create import and package desclarations.

Within Class

Anything you enter into this field will be generated immediately after the opening curly bracket of the class declaration.

Description

This facility allows the documentation of your class. It should explain what the class represents and what it does. This should be a general overview of the class and explain its purpose in your application. You have successfully documented a class when someone who knows little detail of your software can tell by reading it what the responsibilities of the class are. This text will be generated in the form of a comment in the .h and .cpp files.

C++ Header and Source

These fields let you specify the extensions used for files generated by Javelin. If you change these Javelin will rename the files for you. You should never move or rename a file directly otherwise Javelin will lose track of it. To move a source file change its path to a valid path from in Javelin and it will be moved to the new path.

C++ Persistent

Check this box if you want your class to be persistent. The effect of this option depends on the Persistent (see Object Databases) option you have selected in Options | Persistent.

Notes for Borland C++ Users: To be streamable a class must ultimately be derived from Borland's TStreamable class.

If TStreamable is not an ancestor of your class then create a class and call it TStreamable. Check its Library Class check box. Change its file name to OBJSTRM.H.

With Javelin all you really need to do manually is change the read and write functions of your class to stream the information that you want stored and retrieved for the class.

When a class is streamable it makes reading it and writing it very easy. You just use the overloaded bit shift operators in a way similar to their use with C++ standard input and output streams. If you want to learn more about using Borland's support for Streamable Objects then read about them in your compiler's manual.

Deleting a Class

Only classes that do not have any <u>relationships</u> with other classes can be deleted. You need to remove all relationships from a class before it can be deleted.

Warning: Deleting a non library class will delete its source and header files also. You should make sure that really you wish to do this before proceeding.

To delete the class:

1. Select the class by clicking the left mouse button on it.
2. Select the class menu and select the Delete option.
3. A warning will appear. Press Ok to proceed or Cancel to avoid deleting the class and its associated source files.

Class Members

Any C++ class can be made up of many class members. Each of these members is either a data or function and hence they are called member data and member functions respectively. In traditional C++ addition of new data members involved locating the file that includes the class' declaration and adding the new class. Adding function members meant that as well as editing the declaration the function definition also had to be typed into the file in which the class was defined.

With Javelin members are added via the design slate. They are specified in special member specification dialog boxes. See Code Generation for information about what Javelin does at code generation time with the member specifications you have entered. The code in a function can be edited conveniently from within Javelin by selecting the member and pressing the Source button or by holding CTRL and double clicking on the function. See Editor for details of advanced editor features.

Access To Class Members

The class members can be viewed or modified using the class members dialog box. To get access to this dialog:

1. Press the right mouse button on the desired class.
2. A pop up menu should appear. Select the Members option on the pop up menu.
3. The members dialog box will appear which contains two sections, each a list box. The top list box lists the data and the bottom one lists the functions associated with the class' module.

Class Member Symbols

The first thing you will notice about the Members dialog is that it uses symbols, not just text, to represent many of the attributes of the members. The graphical representation provides a great deal of information at a glance. In C++ combinations of access specifiers and storage specifiers work together to affect the scope (see glossary) and storage of class members and non members. The number of combinations make for a complex C++ syntax. In preference to showing member declaration syntax in the dialog box we have chosen to represent the end result of a set of specifications applied to each member.

For instance global data could be shown by the keyword extern and no static keyword. The result of this keyword is a global variable. In Javelin this is shown by a picture of the globe in the left column.

The access of members is also shown graphically. Public, protected and private members are represented by a door. Public access means that access is given to all. This is shown by an open door.

A member with protected access is only available to classes that derive from the current class. This is shown by a guard standing at the door, restricting access to descendants of the class.

Private access members can only be used by the class itself and thus a locked door is shown.

For class members that are static the images are the same as those above except that a "1" is painted on the door. This is more appropriate to data members than function members but, even so, it is used for both. The 1 represents that, at least for data members, only one (1) instance of that member will ever exist regardless of how many objects of its class have been constructed. This feature can often be used to avoid the need to create a global variable.

Variables and functions which have only scope within a one source (.cpp) file (module scope) are shown as a picture of a source file.

Variables and functions can be marked as constant in C++ using the const keyword. In Javelin you simply check the Constant check box. Const data can not be changed except at initialization and const functions can not change any data. As a screwdriver is often used to adjust things non const data or functions are represented by a screwdriver.

Constant data or functions have a "no-screwdrivers" symbol.

Member functions in C++ can be virtual (can be redefined in a derived class), pure virtual (not implemented in this class but must be implemented in a derived class) and non virtual. This is represented by a series of rectangles representing classes. Filled rectangles represent that the function is implemented in that class (ie., not pure virtual). The symbols are as follows:

Virtual function

Pure Virtual function

Non Virtual function

The development phase of each member is shown as a phase graphic similar to the ones shown for classes.

Modifying Function Source

Javelin provides fast access to the code of your functions. This saves time by avoiding the need to search through an endless maze of text files to find what you're after. To edit the source of a particular function simply hold down the CTRL key and double click on its name in the list box. An alternative to this method is to select the class by clicking on it and then clicking on the source button.

Modifying Member Specifications

Modification of members is performed in their appropriate member specification dialog. To bring up a specification dialog box for a particular member simply double click on its name in the list box.

For details of class member specifications see the following sections:

Member Data Specification
Member Function Specification

It should be noted that data and functions can have any one of four access levels: Public, Protected, Private, non member. See your C++ manual for an explanation of the first three of these.

The non-member option specifies a data or function that is not a member of the class.

For the most part they are treated like real members and referred to generically as members Javelin. They are added, deleted or modified just like real members. They differ in that when code for the class is generated they will be generated in the module in which the class is generated but outside of the class' definition. These non member data or functions can be given static qualifiers to reduce their scope to only the one module in which they appear.

Adding New Members

To add a new member press the New button of the associated list box. eg., to add a new function member press the New button associated with the bottom section.

Deleting a Member

To delete a member:

1. Click the member to be deleted. 2. Hit the Cut button associated with its list box (ie., top Cut button to delete a data member, bottom Cut button to delete a member function.

To Cut/Copy/Paste a Member

The usual cut copy and paste operations can be performed across different classes. This makes it fast to redefine a class in a derived class. Simply copy the member in the base class, open up the members dialog for the derived class and press the appropriate Paste button.

Sometimes a class has many functions with similar parameters and return type but a different name. Rather that retype all the details you can copy and paste the function which the others are to look like. As Javelin will not allow two members with the same name: copy the original member, then rename the original and then Paste the copied version.

Member Data Specification

Javelin allows the user to enter the specifications of data members via a Member Data Specification dialog box. To get access to the specifications of a particular data member see Class Members

Name

Specifies the name of the data, eg. Counter It is also used specify array types (see C/C++ Arrays in this section).

Type

Specifies the type of the data variable.

Primitive data types

To specify that a data member is an integer specify "int" in the type field as you would in normal C/C++.

C/C++ Arrays:

For most primitive data types you declare the type as explained above. For arrays, however, you specify the type of the elements in the array in the type field and add the array suffix "[a constant]" at the end of the Name field. Where a constant can be blank for an unbounded array or an integer for a fixed length array. For example: to declare an array of MYSTRINGLEN chars,

Name: MyString[MYSTRINGLEN]

Type: char

You might like to define MYSTRINGLEN in the extra .h includes section of the class details.

Development Phase

This combo box allows you to specify the current development phase of the data member. This is represented by a phase graphic.

Initialisation

The categories of data members which can be have an initializer specified are static data members, global data, module data (static non members). In fact it is legal to enter an initial value for any category of data member but only the above mentioned categories will utilize the initialization. You may still want to enter an initialized value for documentation purposes only.

If the type is a simple type (eg., int) then the initial value can be specified as: = 3. For types which are classes the initialization is a matter of following the name with the parameters that you wish the constructor to be called with. For example: to initialize a circle object of name MyCircle place (30, 20, 20) in the initialization field to cause the circle to be initialized with its origin at 30, 20 and with radius 20.

Javelin will generate data declarations with only the text entered into the name field but definitions will be generated with the name text suffixed by any text in the initialization field.

Specifies the C++ access level: public, protected or private to specify access for member data; or; non-member which specifies that this data is not a member of the class itself but that it is to be declared and defined in the class files but outside the class declaration.

In some cases it is required to have a variable defined in the file (file scope) which can be used by the class member functions and non member functions even though it is not a member of the class. The non-member option facilitates this.

The effect of this field depends on whether or not the data is defined to be member data (private, protected, public) or non member data.

Static Member Data:

Static member data are generated with the static keyword before them in the class declaration if this field is checked.

Static Non Member Data:

If Static is not checked then the variable is declared with extern in the .h file and defined without extern nor static in the .cpp file. This in effect creates a global variable available to all modules.

If Static is checked then the variable is defined with the static keyword in the .cpp file. This creates a static variable with scope only within the module in which it is defined.

Persistent

Check this check box if you would like this data to be read and written to pstream whenever the class is read or written to a pstream. This only affects data in a class which has been marked as Streamable (ie., Make Streamable check box in its class details dialog has been checked. When the read/write functions are *first* created (ie., when the class is made streamable) for this class' .cpp file then the following lines will be added to its read and write functions.

```
MyClass::read:
is >> MyData;
MyClass::write
os << MyData;
```

For most primitive data types such as int's, float's etc., the streaming code generated will suffice but for compound data types such as strings etc., the code generated in the .cpp file would have to be changed. For example strings can be read from a stream using:

```
Str = is.readString();
```

Description

Allows the programmer to describe what the data is and briefly how it is used. Any special conditions regarding its use should also be mentioned in the description. For example:

Name: pSquare
Type: Square *

Description: This is a pointer to a square object used to draw a border around the outside. It is constructed when the window is first opened.

NOTE: Always check pSquare for NULL before use.

Slate Map

The Slate Map is an optional separate window which constantly shows the entire slate in a condensed view. It is tightly linked with the main <u>design slate</u> window. The area of the slate that is visible in the main design slate window is indicated by a grey rectangle in the Slate Map.

Opening/Closing the Slate Map

The Slate Map is toggles between open and closed with each press of the button in the tool bar.

If you wish to open the slate map and commence a magnify operation in one step you can press the button in the tool bar. The Slate Map will open and you can then use it to complete the magnify operation. See below.

Exposing Regions of the Design Slate (Advanced Scrolling).

To expose a different region of the design slate perform the following steps in the Slate Map:

1. Move the mouse over the grey rectangle indicating the currently exposed region and press and hold the left mouse button.
2. Move the rectangle to cover the area of the slate that you wish to be exposed in the design slate window.
3. Release the mouse button.

The main design slate window will now be redrawn to expose the area specified.

Magnifying using the Slate Map

The magnification available in the Slate Map enhances that which is available in the Design Slate directly. Even though you can magnify directly in the design slate this only permits zooming into an area which is smaller than the area currently shown in the window. As the slate map shows the entire slate it is possible to zoom to any region in the entire slate.

To zoom using the slate map:

1. Press the button in the tool bar and then move the mouse into the Slate Map window.
2.   Press the left mouse button and drag a rectangle around the area that you wish to expose in the design slate window and release the mouse.

The design slate will be redrawn to expose the selected area.

Member Function Specifications

Javelin allows the user to enter the specifications of function members via a Member Function Specification dialog box. To get access to the specifications of a particular function member see <u>Class Members</u>

Function Categories

In C++ most member function declarations and definitions conform to a common format. The general format is shown below: General Function Declaration Format:

```
funcType functionName(type1 param1, type2 param2);
General Function Definition Format: funcType functionName(type1 param1, type2
param2)
{
   // code in here
}
```

While most member functions in C++ conform to this format there are some, such as Constructors and Destructors that can differ from the standard format. In addition to this users of Turbo C++ and Borland C++ often take advantage of Message Response Functions which can make the job of responding to Windows messages a lot easier. Message Response Functions are declared differently to the standard C++ member function declaration format.

The different categories of functions supported by Javelin and their specifications are explained below:

General Functions

Functions in this category usually make up the majority of member functions in a C++ application. In Javelin General functions are any functions that are not Constructors, Destructors or message response functions. There are two types of General functions: member (private, protected or public) and non member.

For General type functions you can specify the following:

Name

The function's name.

Type

The type of the value returned by the function. For example int, float etc., Type the word void if no value is returned by the function.

Parameters

Enter the parameters and their types just as you normally would add parameters to a function. For example a function that takes an int and a float may have a Parameters field that looks like this:

```
int Count, float Value
```

NOTE: Do not add round brackets. These are taken care of by Javelin.

Development Phase

This combo box allows you to specify the current development phase of the function member. This is represented by a <u>phase graphic</u>.

Access

Specifies the C++ <u>access</u> level: public, protected or private to specify access for the

member function; or; non-member which specifies that this function is not a member of the class itself but that it is to be declared and defined in the class module but outside of the class declaration.

In some cases it is required to have a function defined in the file (file scope) which can be used by the class member functions and non member functions even though it is not a member of the class. The non-member option facilitates this.

Static

Check this to declare a member or non member function as static.

Inline

Check this to declare an inline function. This version of Javelin does not support code generation of inline functions and requires you to manually change the .h file for a class. If you require inline functions then you should change the path of the class to avoid Javelin overwriting your changed version of the .h file.

Virtual

Check this field to declare a function as virtual.

Pure Virtual

Check this field to declare a function as pure virtual. No function stub will be generated for this in the .cpp file.

Const

Check this field to declare a function as const.

Description

Allows the programmer to describe what the function does and briefly what it does it to. Any special conditions regarding its use should also be mentioned in the description. For example:

Name: Show
Type: void
Parameters: HDC hDC

Description Shows the square object in the display context given in hDC using the X,Y data from base class Shape as the left, top corner of the square and SideLength as the length of each side. NOTE: Must check that X,Y are within a specified rectangle.

Constructors/Destructors

Constructors and destructors are different in that they take on the name of the class to which they belong. Destructors also add a '~' prefix to the name. They also differ in the fact that they do not have a type. Javelin generates Constructors and Destructors without a type.

When generating function stubs in the .cpp file for a class Javelin generates the code to call base constructor of the class but without any parameters for these base constructor calls. It it up to the user to manually insert the parameters for the calls to the base class

constructors from within Javelin's function source edit facility or from a third party editor.

For example: ClassA is derived from ClassB and ClassC. The constructor for ClassA will be generated as follows:

```
ClassA::ClassA(int ID, float Value)
   : ClassB(), ClassC()
{
}
```

It is up to the user to enter parameters for the calls to the base classes in the .cpp files.

Message Response Functions (Borland C++ Version 3.x Only)

Borland C++ 3.1 introduced a concept called Message Response Functions (see chapter 1 page 9, Object Windows for C++). This increases the ease at which a programmer handles the messages being generated from the Windows system.

A Message Response member function can be added to a class by selecting Message Response as the function type when adding a new function. Message response functions should never be non member functions.

You specify which message that you wish the function to respond to by typing the identifier representing that message. For example if you want to create a function which is called whenever Windows tells you that you need to repaint a section of your window you would create a message response function for the WM_PAINT message. To do this specify WM_PAINT in the message field.

Borland encourage programmers to follow their convention and in most cases name message response functions after the message they respond to in the following way:

1. Remove the first underscore.
2. Make words lowercase except the first letter of each word and the letters signifying the type of message.

So the name of the function responding to the WM_PAINT message should be WMPaint. WM_DRAWITEM becomes WMDrawItem. Following this convention makes obvious good sense.

NOTE: When generating the function prototype Javelin automatically adds the appropriate WM_FIRST, ID_FIRST or CM_FIRST if the first two letters of the Message you type in match WM, ID or CM respectively.

When the code is generated for this will look like this:

.h file:

```
virtual void WMPaint(RTMessage Msg)
   = [WM_FIRST + WM_PAINT];
```

.cpp file:

```
// Comment
void
clsname::WMPaint(RTMessage Msg)
{
}
```

Inheritance

Inheritance is one of the most important features of true object oriented programming. It frees the developer from unnecessary duplication of effort by allowing classes to inherit "features" of other classes, rather than redundantly duplicate the effort as in non object oriented programming languages. You should always keep your eye out for good object oriented programming journals that cover topics such as "Inheritance: when to use it and when it should not be used".

Inheritance specifies that a class inherits features from another and can extend these features by adding extra processing or completely replace the processing of the super class.

In **C++** terminology we say:

> "A sub class *extends* one and only one super class. Inheritance relationships are drawn as a line with a solid arrow pointing to the super class from the sub class."

In **C++** terminology we say:

> "A derived class *inherits* from one or more base class. Inheritance relationships are drawn as a line with a solid arrow pointing to the base class from the derived class."

The inheritance related operations are:

Creating Inheritance Relationships

Modifying Inheritance Relationships

Deleting Inheritance Relationships - See Relationships

Modifying Inheritance Relationships

C++

To modify specific details of a class' inheritance:
1. Press the right mouse button on the class.
2. Select Inheritance from the pop up menu.

A dialog box will appear allowing you to specify the following inheritance details:

Order of base classes

To rearrange the order of base classes:
1. Press the left mouse button on the name of the base class that you wish to move.
2. While keeping the button presses, drag the cursor to the position in the list that you desire this base class to appear at.
3. Release the mouse button. Javelin will move the base class to the position you specified and automatically adjust the other classes as necessary.

Removing Inheritance Relationships

See Relationships for details of how to delete relationships.

Public, Protected Access Modifier

In C++ when specifying that a class is derived from one or more other classes you have the option of specifying the level of access. This can be Public or Protected. To change the access level in Javelin:

1. Select the appropriate base class in the list box.
2. Press the appropriate radio button: Public or Protected according to the base class access that you wish to grant your derived class.

Virtual Base Class

In C++ when specifying that a class is derived from one or more other classes you have the option of using the virtual specifier. The Javelin default is not virtual. (Refer to a C++ language reference manual for more information)

This can be specified in a class' inheritance dialog box by the following process:

1. Select the base class in the list box that you want to be a virtual base class.
2. Check the check box labeled Virtual.

Code Generation

InSync Code Synchronisation Technology

Javelin provides advanced code synchronisation facilities via its InSync technology. InSync allows the design and code to be kept in synchronisation with each other automatically. Any changes made to functions bodies and special "Keep" sections are preserved when a design change causes an automatic regeneration of the relevant code. This means that you can work within or outside of the tool when implementing your designs.

Class Header files (.h)

A class header is generated for each class in Javelin that does not have the Library option set in the class details dialog box.

Javelin takes the class specification information and generates a header file based on that information. It will regenerate this whenever it thinks this is necessary.

Both class header and source files are generated and regenerated to the directory specified by the Path field in class details. If this is a relative path then it is relative to the current working directory.

Descriptions associated with data in a class are generated as comments above the declaration of each data (or variable) in the header file.

If you look at generated code you will notice how it contains markers embedded in comments. For example: // -[Member_Data_Decs]-. These markers mark out sections of the code. A section is all of the code from the end of one marker to the beginning of the next. The Member_Data_Decs section contains the declarations for the member data of the class. Certain sections are regenerated each time Javelin regenerates the file. Other parts of the code are preserved through the regeneration process. You must not add or modify any section markers.

Any section whose section marker begins in the word "Keep" is preserved whenever a regeneration occurs. For example: if you add code to the section marked Keep_h_Extras then this will remain in the file the next time the file is regenerated. This enables you to add extra types, or extra #include lines. These lines can be added outside of Javelin in a separate text editor during the debug stage, for example, or some sections even have special access provided within Javelin itself (See Class Details). There are many keep sections throughout the code. It is important that you close down any edit session with a generated file before you cause Javelin to regenerate that file.

Source files (.cpp)

A class implementation file (commonly referred to as "source file") is generated for each class in Javelin that does not have the Library option set in the class details dialog box.

Javelin takes the class specification information and generates a source file based on that information. It will regenerate this file whenever it thinks this is necessary. You can make changes to the function bodies (code between opening { and closing }) in the .cpp file and the changes will be preserved the next time Javelin regenerates the .cpp file.

Certain other portions of code also enable changes to be made which will be preserved.

Descriptions associated with functions in a class are generated in the .cpp file. This is so that you can specify what a function must do in the design phase and then when the time comes to implement the function the specification is right there where you need it.

genclsid.h

This file is regenerated whenever a class name is changed or a new class is added. It contains an enumeration called GenClassId which contains the names of all the non library classes (See class details). This provides each class with a unique Id which can be used by the isA function in, for example, the container class libraries. The identifiers are created by adding the suffix "Class" to the class name. The first class id is assigned 7000 and the others follow from this.

If your project contained two classes YourClassA and YourClassB then the contents of the ClassId enumeration in the generated genclsid.h file would be similar to the following:

```
enum ClassId
{
   ShapeClass = 7000,
   YourClassAClass,
   YourClassBClass,
   __lastGeneratedClass
};
```

Limits

The current version of Javelin does not have explicit support for friends or templates. Non supported language constructs that you require can be added to the "h Extras", "class Extras" or "cpp extra" section of class details dialog box. Even though template classes can not be generated from Javelin they can be instantiated and used as valid types just like any other types.

Selecting Classes

Javelin has a powerful and flexible approach to single and multiple class selection.

A selected class is shown with white rectangles at its corners. These are may not be used to change the size of a class as the size of a class icon is automatically adjusted according to its name and its members names. Selection of classes is possible via the following mechanisms:

Selecting a single class

Left Click on a class

If the class is already selected it does nothing.
If not selected then it deselects all other classes and selects it.

Selecting a hierarchy branch

SHIFT + Left Click on a class

Selects it and all of its derived classes recursively, retaining all other selections.
HINT:
Selection of multiple classes is a convenient way to move groups of classes around on the design slate. When used in combination with Selecting a hierarchy branch you have a very easy way to move sub branches of the hierarchy.

Toggling the selection state of a class

CTRL + Left Click on a class toggles the selection state of the class.

Selecting all classes within a specified rectangle

SHIFT + Left Click on empty space and drag selects all classes enclosed by the dragged rectangle.

Editor

Javelin maintains a close relationship with your source code. The source can be edited using either the smart in built editor or a third party text editor. It is invoked from various points in the tool. The editor can be specified as described in <u>Menus</u> options.

Advanced In Built Editor Features

User Specified Help search

The in built editor provides access to a help file which you can specify in visclass.ini in the [Editor] section under the entry "HelpFile". Once a help file is specified Javelin will search in it for the word at the editor's caret whenever CTRL-F1 is pressed.

Tab Width

You can specify the width of tab spacing in the in built editor by adding an entry called TabWidth=x in visclass.ini in the [Editor] section. x is the width of the tab spacing measured in character widths.

Design Look Up and Paste

Another powerful feature takes advantage of Javelin's knowledge of your design. Click the right mouse button to get a list of all classes. Selecting a class brings up its members dialog. At this point you can examine a member's description or cause a member to be copied to the editor at the current caret position. This is particularly helpful when implementing a call to a member function for which you do not know the parameters or their type. You can also copy the name of a class to the editor by pressing the Name button.

Life Cycle

Phases of Development

Professional object oriented design has five basic phases: analysis, design, implementation, testing and complete. Not only does the project as a whole experience these phases but each component of the project (subsystem, class, function member etc.,) can also experience these phases. The object oriented methodology and Javelin's synchronization allows easy transition between each of the phases, enabling incremental development: The project can start with a modest specification and then be incrementally improved upon until the desired result is obtained.

Javelin takes a minimalistic approach to object oriented methodologies. Javelin provides a productivity tool which helps developers develop their designs with an uncomplicated object oriented software methodology that supports projects through the above phases of development. To facilitate incremental development Javelin includes facility which enables you to track the current phase of each component. See Development Progress Tracker

The initial task for any system involves determining what the system is to do. You then model the system in Javelin by adding classes that represent the entities ("things") and the relationships between them. From then on the development tasks are as specified in the following table:

Development Phases and Tasks

Phase	Class	Member Function	Member Data
Analysis	Think through the basic behaviour of the class and document this in the class description. Determine this class' relationships with other classes and express these by dragging and dropping between classes.	Specify what the function will do and from this choose an appropriate name for the function (usually containing a verb). Document the function's purpose in the description field.	Choose an accurate name for the data member and document its purpose in the its description field.
Design	Add the data and function members required to model the class' behaviour. The members need only be analysed at this phase.	Determine the appropriate arguments and return type for the function. Special conditions should be noted in the description field.	Choose the most appropriate type for the data member (ie., float, double, etc.,)
Implementation	A class is being implemented while ever any of its members are	Use the in-built or an external editor to write the code for the function	Javelin automatically generates member declarations.

	not yet implemented.	body.	
Testing	Create object's of this class and exercise them by testing their behaviour under a wide variety of conditions. Some aspects of testing may not be possible until other classes have been implemented.	Exercise the function under a wide variety of conditions to ensure that it performs as specified in the description. Modify the description field to accommodate any changes that were made.	Test that the value of this field remains sensible under a variety of conditions. This is done by testing its value at various critical points in the application's execution.
Complete	The class has been tested and proven to model the behaviour specified during the analysis phase.	The function can be relied upon to perform as specified.	The data member's type has proven to be appropriate and its value remains sensible throughout program execution.

The graphic used to represent each phase and the tasks which are to be performed during each phase for each of the components are shown above.

Development Progress Tracker

Javelin features a development tracking ability which can provide you with up to date information about your project's progress. You can easily see what has been completed already and what needs to be done next. You no longer have to rely on your compiler or severe crashes to let you know what else needs to be done.

The Javelin Phases of Development

Each class, data member and function member has a sense of its current phase of development. The phases of development are Analysis, Design, Implementation, Testing and Complete. For more information about these phases see Life Cycle. Each component starts out in life in the analysis phase except in special cases mentioned below.

Classes, Data members and Function Members all have a phase graphic field in their respective dialog boxes which allows the user to change their current phase. The classes' phases are shown on the design slate whenever View|Show Progress is checked. The member phases are shown in the members dialog box.

Printing Development Progress

The development phase is shown on printouts as a tower of four bars instead of the above images. Each bar represents an incomplete phase starting with analysis at the top and ending with testing at the bottom. A completed class is shown by a tick so that it is easy to see at a glance what work needs to be done. Generally more bars means more development. As development progresses your progress is visually apparent as more and more ticks appear and you will get a sense of satisfaction knowing that you are getting somewhere instead of not knowing how far you've gone and how far you've got to go.

To print out the progress of your classes in the design slate:

1. Ensure that View|Show Progress menu is checked
2. Set the required page tiling options using File|Tiling
3. Select File|Print|Slate

Initial Phase of Components

When loading a project which was built using a previous version of Javelin or importing existing code Javelin sets classes and their members to the Complete phase.

When adding a new library class if you leave the phase at Analysis it will be changed to Complete automatically.

Development Phase Consistency Checking

Without automated checking of phase consistency it is possible to set a class to Complete when in fact one or more of its members are, let's say, in the implementation phase.

Javelin provides a mechanism for automatically checking and correcting the phase consistency of a class. This mechanism can be invoked manually whenever required by checking Check Phase Consistency Now in the dialog from the Options|Project menu and pressing Ok. This mechanism can be invoked automatically whenever a member or class's phase is changed by checking the Auto Phase Checking check box in the same dialog.

When a phase check occurs a class phase may be changed if it is not either in Analysis or Design phase. The possible transitions are given below along with their cause.

Promotion to a more advanced phase

You will be prompted prior to a phase promotion occurring even if Silent Phase Check is selected.

Implementation -> Testing: All members are at least at the Testing phase.

Testing -> Complete: All members are at the complete phase.

Demotion to an earlier phase

Complete -> Testing: At least one member is still being tested
Testing -> Implementation: At least one member is at the implementation phase or earlier

Silencing the Phase Checker

You can prevent Javelin from notifying you on inconsistencies by checking the Silent Phase Check check box in the project options dialog box. Certain notifications, such as when a class' phase is demoted from a more advanced phase, say complete, back to an earlier stage, say testing are always notified regardless of the value of Silent Phase Check.

Moving a Class

Moving will move a single class or all selected classes. To move a class or set of classes:

1. Press and hold the left mouse button on the class that is to be moved.
2. Drag the mouse until the class(es) is at the desired location.
3. Release the mouse. If the class was selected then it and all other selected classes will move also.

HINT: When used with the Shift+Left Click option it becomes an easy way to move sub branches of the class hierarchy. See Selecting Classes.

Class Documentation

While Javelin serves as an on line documentation tool for your C++ projects it is often necessary to produce hard copies of your work. Javelin can print class documentation to your windows printer for all of your classes, including all member details. For information on printing out the design slate see Design Slate Printing Your Design.

To print documentation for a single class:

1. Select the required class by clicking the left mouse button on it.
2. Select the Class Menu and choose the Document option.

To print documentation for the all classes:

1. Repeat the above procedure but choose Document All from the Class Menu.

Documentation - Printing Your Design

Javelin provides facilities for producing professional hard copy documentation of your projects on Windows printers capable of graphical output (ie., most modern printers). The user can print out the design slate (a graphical printout) and the class documentation (a textual printout). These are described below:

Design Slate Printing

The design slate print facility allows professional hardcopy documentation of the graphical design as it appears in the design slate. For information concerning the detailed documentation of each Class in a text format, see Class Documentation below.

Tiling Option

As the design slate for many projects can contain many classes Javelin allows for design slate printouts that span more than one page. When the printout is complete the separate pages can be trimmed as required and pasted together to form a large printout (wallpaper for your office wall). The separate pages can be seen as tiles making up your final masterpiece - not unlike mosaic artwork (pieces of art made up from the arrangement of many coloured tiles). This process is often called mosaic printing.

The final size of the printout is determined by the parameters entered into the "Tile Options..." menu item in the File pop up pop up. Selecting this menu item will bring up a dialog box asking you to enter the dimensions of your printout. The units of the dimensions are pages of the currently selected paper in the current printer. So entering Width 2, Height 1 and pressing Ok specifies to Javelin that design slate printouts will be 2 pages wide by 1 page high. If A4/portrait orientation is the current paper type/orientation then this would make a printout, from two A4 sheets, slightly smaller (allowing for necessary overlap) than the size of an A3 sheet. If your design is very large you could specify a larger Width and Height.

Print Slate

When the desired tile options have been specified ensure that you have selected the printer driver appropriate for you printer. To commence printing of the design slate select the Print Slate option in the File pop up menu. Javelin will scale the graphical information in the design slate to best fit the total size of the printout (see above "Tiling Options"), however the aspect ratio of the design slate will be preserved. This may explain why Javelin does not seem to fully utilize all the pages in a particular printout.

See also **Class Documentation**

Project Options

Javelin options allow a programmer to tailor the system for their needs. Options are accessed via the Options menu choices Project, Persistence and editor.

Project

Project options allow the setup of each of your projects as described below.

Project Name

Name of the project.

For Company

Name of the company that this project is being produced for. This name appears on slate printouts.

Designer

Name of the individual in charge of the project.

Copyright Notice

The text in this field will appear in the Copyright notice section of each source file.

Auto Phase Check

Checking this option causes phase checks to occur after a class is modified or any of its members are modified. See Development Progress Tracking.

Silent Phase Check

Checking this option causes phase checking to be silent. In other words you will not be prompted by a message whenever a class' phase is changed back to an earlier phase. You will however be prompted if Javelin has found that a class' phase should be advanced.

Check Phase Now

Checking this option causes a phase check to take place immediately after Ok is pressed.

Persistence

Javelin can generate code for a number of different object persistent solutions. If you have purchased the POET object database select the POET option and specify a file name for the database you will produce. Prior to switching from OWL 1.0 streamability you should delete the streamability functions read, write and the streamable constructor from each of your persistent classes. You may like to back up these files before doing this if you want to keep a copy of these functions.

Once this option is selected Javelin will perform any following code generations with the POET "persistent" keyword class prefix. You should perform a Generate|All Classes after changing this option. For data members that are marked as "not persistent" the word "transient" will precede their declaration. The default filename extensions for new classes become .hcd and .cpp and generated source files will contain code that #includes "myheader.hxx" where myheader is the filename of the class module being generated. myheader.hxx is the file created by running the POET precompiler on the myheader.hcd file created by Javelin. See POET manuals for more detail.

If you are using OWL 1.0 you might like to use the streamability supported by that library. If so select OWL 1.0 Streamability.

If you are storing your classes in a relational database then select None. You will have to break your

objects apart to store them in tables. This indeterminate process is not support explicitly by Javelin. Step Ahead Software keeps abreast of the latest ADVANCES IN OBJECT ORIENTED technology so do not hesitate in contacting us directly regarding the use of Object Databases.

Editor

Editor options allow the user to specify an external editor to be used when editing source code (specifying "none" results in the default in built editor being used).

Adding a class

To add a new class to the Javelin design slate:

1. Select the Class menu and choose the New option or click the right mouse button on empty space in the design slate and choose "Add Class" from the pop up menu that appears. Choose "Add Interface" to add a new interface. The new class will be called UnnamedClass and will not be inherited from any other class.
2. Edit the class' details in the <u>class details</u> dialog box that appears. Click on Ok when complete.
   NOTE: You will not be able to access the source code via the buttons on the bottom of the dialog box until you have hit Ok.
3. After entering the class details you can then specify its <u>inheritance</u> and <u>reference relationships</u>.

NOTE: Whenever you add a new class you should add it to your compiler's make file or project file. See <u>code generation</u> for more instructions on how to best create .cpp files.

Javelin automatically #include's the header files of any class which the generated class inherits from. <u>Reference relationships</u> on the other hand, can be implemented in a variety of ways it is left to the user to #include a referenced class if necessary.

Menus

File

New Project
Saves current project and creates a new one
Open Project
Saves current project and opens an existing one
Save
Saves current project - you should do this regularly as this forces Javelin to save your work. Javelin saves a backup file (*.bak) prior to each save.
Save As
Saves current project under a user specified name.
Tile Options
Sets the tiling options for slate printouts.
Print Preview
Opens a print preview of the slate print out.
Print
Commences a particular print request.
Print Setup
Allows printer setup.
Import
Launch the cooperative import facility.
Exit
Exits Javelin and saves current design information.

Edit

Delete
Deletes the currently selected class.

Class

New
Adds a new class to the design slate.

Details or
Brings up a dialog box allowing the user to modify the details of the currently selected class
Members or

Brings up a dialog box allowing the user to modify the members and non members of the currently selected class.

View

Entire Slate
Zooms out as far as possible to view the entire slate which has a border drawn around it.
Fit All
Scales the slate to fit all classes.
Show Inheritance
Toggles visibility state of inheritance relationships.
Show References
Toggles visibility state of reference relationships.
Show Progress
Toggles visibility of the current phase image for the class icons.

Generate

Class Code or
Generate code for the currently selected class. (This is done automatically by Javelin whenever it thinks it necessary)
All Code or

Generate code for the all non library classes and generate the genclsid.h file.

Options

Project
Set project options such as project name and source code Copyright Notice etc.,
Persistence
Tells Javelin which method of object persistence you are using, if any.
Editor
Brings up a dialog box allowing the user to specify the editor that will be used (none == use in built editor)

Help

Index
Brings up the index for on line help
Technical Support
Instructions for obtaining technical support
About
Shows the Javelin Logo and information about Javelin

Associations

Association relationships are used to model the way that one object is often "composed of" or "refers to" one or more other objects.

NOTE: Association relationships are only supported in the professional edition.

See also Code Generation for Associations.

An example of a association is a Company which, among other things, *employs* one or more Employees. This is called a *one-to-many* association because one company employs many employees. The inverse of this is that each employee *is employed by* a company. The words "employs" and "is employed by" are called roles because they describe the role that each object plays in relation to the other. Javelin enables the naming of these role names on associations in the design slate and the code generated for associations using these names as the names of data members used to implement the associations.

An important term used in associations is cardinality. Each end of a association relationship has its own cardinality. The cardinality is the "one" or "many" of it's end of the relationship. In the above example the cardinality of Employee is "many" while that of Company is "one". Cardinality can be changed at will by the user.

Appearance

Association relationships are drawn as a line between two classes with Cardinality anchors at each end.

Creating Associations

To establish a association relationship between two classes:

1. Press and hold the left mouse button on the top section of one class.
2. Drag the mouse to the destination class and release the button. (You can drag to the same class that was dragged from. This is valid for associations but not for inheritance). You may lift the mouse button at various points on the path to create bends.
3. You will be asked to specify the type of relationship that you wish to establish between the classes. For a association select whichever is required: one-to-many, many-to-one, many-to-many, one-to-one.

A line will now be drawn with ends, called Cardinality Anchors, to represent the cardinality using OMT (James Rumbaugh) notation. ie., Cardinality "1" is shown as a single line. Cardinality "many" is shown as three "spokes".

Modifying Association Details

To change the details of an association:

1. Press the right mouse button on the cardinality anchor at either end of a relationship.
2. Select association options.
3. A dialog will appear containing fields that describe the association on the top half. The bottom half describes the inverse of the association. ie., the association in the reverse direction.
4. You can change the cardinality by selecting a choice from the combo box. You can also change the role name (eg., "is_employed_by") and the type of association (has, contains or none). The entire set of controls should read like a sentence if you have the association type and cardinality set up correctly. If cardinality is "many" Javelin

appends a (s) to the right hand class name to help the sentence read well.

NOTE: You should always name your classes as singular and not plural. For example you should not call a class for Person objects Persons. It should be called Person.

NOTE: You can achieve steps 1 and 2 by double clicking on the icon at the end of the relationship.

Code Generation for Associations.

Associations between classes can be implemented in C++ in a variety of ways. The implementation used is normally dictated by the underlying database used to store the objects. Within a class declaration we call data members that refer to other classes as "reference members" (not to be confused with references using the '&' operator in C++). The data contained within them is normally not used as pure data but it used to provide access to another class. A typical example of a reference is a data member in one class which is a pointer to another class. Another example is long integer used as a foreign key to refer to a class (represented by a record) in a table in an SQL database.

This section describes how Javelin can be tailored to code suiting a wide variety of compilers and databases.

NOTE: Association relationships is only supported in the professional edition.

Changing Cardinality

Javelin is capable of generating class declarations with a variety of reference options. The cardinality anchor at each end of an reference relationship is responsible for the reference code generated for the class at the other end of the relationship. To change the code generation for reference options:

1. Bring up the Reference Details dialog box by pressing the right mouse button on a cardinality anchor.
2. Press the Code Generation button.
3. A dialog box will appear. The type of dialog box depends on the Persistence option currently selected. (Choosing Options|Persistence menu allows you to change the current Persistence but it changes it for all reference code generation).
4. Choose the options specific to the current persistence. See any relevant third party documentation if necessary to understand the terms.

If the built in code generator options are not exactly what is required you can choose "Custom" and select from code generation reference strings which you can add to in ccreator.ini in the windows directory.

Specifying Custom reference strings

Custom reference strings are grouped into two categories. A category for when cardinality is zero or one and another when cardinality is many. The strings for these sections are stored in [CustomCodeGenRef_1] and [CustomCodeGenRef_m] sections in ccreator.ini. Each entry in these sections must begin with a number followed by '=' and then followed by name, a comma, and then the reference code generation string. Num=AnyName,CodeGenString

The reference code generation string can be made up of any characters but some have special meaning which results in them being translated. Translation characters are prefixed by a '%'. The current set of translation characters allow a single string to be used for many different reference relationships.

The translation characters are as follows where "this class" is the class being generated:
```
%c - name of the class that this class refers to.
%r - role that this class plays in relation to the other class.
%C - name of this class.
%R - role that other class plays in relation to this class.
%s - semi colon ';' character.
```
For example:
```
[CustomCodeGenRef_1]
1=Pointer,%c *%r%s
2=ForeignKey,LONG %cID
```
There are currently three in built code generators capable of generating code specific to a variety of needs. The code generators are:
POET - for the POET object database
OWL 1.0 - for Borland OWL 1.0
Custom - User configurable generation options

The choice of code generator affects how Javelin generates class declarations.

Within each code generator the user can make choices to further specify how code should be generated. In   all cases if a user is not satisfied with the set provided by a particular code generator they can use the custom code generator.

Relationships Between Classes

An important part of object oriented programming involves modeling not only the classes but the relationships between them. Inheritance, Implementation and Associations (associations are only supported in the professional edition) are the basic types of relationships.

Types of Relationships

> Inheritance
> Implementation
> Associations

Relationship Operations:

> Creating a Relationship
> Inserting, Moving and Deleting Bends
> Deleting Relationships

Creating a Relationship

You can specify that a class relates to another easily:

1. Press the left mouse button on the top section of the first class.
2. Drag the mouse to the other class. You can release and press the mouse at various points to insert "bends" in the line.
3. Release the mouse button and select the type of relationship required. There is one inheritance option and four reference options to choose from.

Whenever a class is moved its relationships move with it. All relationships must be removed before a class can be deleted.

Manipulating Bends

Bends allow relationships to be positioned so as they avoid clashing with other objects and thus avoid a confusing, messy display. The operations that are related to bends are as follows:

Inserting a bend

1. Click the right mouse button on a part of the line making up the relationship. A pop up menu will appear.
2. Select Add Bend.

Moving a bend

1. Press and hold the left mouse button on the appropriate bend.
2. Drag the mouse until the bend is at the desired location.
3. Release the mouse button.

Deleting a bend

1. Press and hold the right mouse button on the appropriate bend. A pop up menu will appear.
2. Select Delete Bend.

Deleting Relationships

To delete a relationship:

1. Press the right mouse button on a part of the line making up the relationship. A pop up menu will appear.
2. Select Delete Entire.

Cardinality

Each end of a reference relationship has a notion of the potential quantity of the thing that it points to. This potential quantity is known as 'Cardinality'.

For example:

Take two classes A and B with a reference relationship between them being A has one-to-many B's. The cardinality at the A end of the relationship is one. The cardinality at the B end is many.

Synchronizing Design and Code with InSync

As stated in the Introduction and Life Cycle sections Javelin is intended to be a tool assisting programmers during the full development cycle of a product, not just at the analysis or design stages. To do this requires a little discipline but this is insignificant compared to the rewards: a design which is synchronized with its implementation; quality and reusability; continual productivity gains; and; the ability to work at a higher level. InSync technology makes it all possible.

Philosophy

Javelin creates and manipulates two files for every class: *.h and *.cpp. Javelin generates the .h and .cpp for you whenever you add a new class. You should add any new class' .cpp files to your compiler's project or make file. The files extensions used for a class' files can be specified using the Class Details dialog.

Javelin manipulates the header and source files whenever changes to the design cause it to do so, thus keeping the code in sync with the design. It is designed so that in most cases you can use header files as generated although it may be necessary to add certain #includes or other code required for the header in the Keep section.

You are able to modify the code within function bodies in .cpp source files. The changes that you make will be preserved through successive regenerations.

Changing Files From Outside Javelin

The code files for your classes have been generated with special section headers of the form -[Section Name]- which are described in more detail in Code Generation These section headers are what Javelin uses when regenerating code. It is essential that they are not disturbed. To maintain an up to date set of information in Javelin it is mandatory that you never edit generated files directly outside of Javelin with a text editor (except of course Keep sections and function bodies in the Function_Defs section). Let Javelin generate and regenerate these files for you. If a change is needed make the change in Javelin and let it automatically regenerate the required class files. Making changes this way insures that Javelin and the class header and source file are both updated and remain in sync with each other.

NOTE 1:

Changing the class header file directly (outside of any keep sections) will result in Javelin and the header file being out of sync. In fact if you do make changes directly to a class file outside of any Keep section or function body these changes will be lost the next time Javelin regenerates the file. If you require a file that is different from that generated by Javelin and your changes can not be incorporated using one of the Keep sections then you must change the Path field in the class details dialog box to avoid overwriting your changes in the file. If this is the case then also make sure that the compiler finds your .h file and not the one generated by Javelin.

NOTE 2:

If you ever leave a header file open in an integrated environment such as Borland C++ for Windows or Microsoft Visual C++ and Javelin regenerates that file then the compiler will recompile the version that is in the integrated environment and not the newly generated one. So you may still have that error that you thought you had fixed. Again, if you close files after editing directly with a text editor you will not end up with this problem. Beware in cases where the integrated environment opens up a file to locate where an error has occurred. Ensure that you close this file down soon after you have examined the error. This should prevent the problem, just explained, from happening.

Changing Class Modules

Whenever a function name, type or parameter is changed then Javelin will regenerate the header and source file for that class. This keeps the Creator Database in sync with the code.

Methodology

Any strategy used by a group of software engineers in their approach to software development.

Software methodologies come in all shapes and sizes. Some are complex, require expensive tools to use effectively, and take a huge investment in time, training and money before they can be used. Others are smaller, simpler, easier to learn and provide excellent productivity gains with a minimal investment in time and money.

As object oriented programming is still a relatively new technology many of the larger complex methodologies are still evolving and changing. Many would agree that they have not yet become stable.

Phase Graphic

Each of the phases of development are represented by a different graphic image on the design slate and in dialog boxes as follows:

Phase	Analysis	Design	Implementation	Testing	Complete
Description	Light bulb representing the ideas phase.	Pencil and ruler - traditional tools of designers.	Building of a brick wall	Purifying a compound with heat from a Bunsen burner	A green t

See <u>Life Cycle</u> for more information about the phases of development.

Software Quality

This following describes some hints that will improve the quality of your software and improve the results of importing code into Javelin

NOTE:

Javelin will attempt to import information from your existing source code and add this information to the design database. In the process it will generate new source and header files to the directories that you specify in the import specification file (.imp). It is up to you to verify that the newly generated code, when compiled, performs in the same way as the non imported code did.

Software Quality

The following two sections describe strategies that can help you improve the quality, ease of use and productivity of your development. While Javelin can import files not conforming to these suggestions the result will be much better if you take a step back for a time and consider their benefits.

Classes and Modules

Each header file is assumed to have a least one class declaration. Javelin works with a strategy that assigns one module (a source file with a matching header file) to each class definition. This complies with certain software quality standards which command, as a minimum, a complete unit test whenever a change is made to any part of a module. When multiple classes appear in a single module a change to one class involves full testing of all of the other classes. Keeping only one class in each module keeps things tidy and helps maintain workable test schedules.

In the case of user classes (ie., non library classes) the import process will add a class to the database for only the first class declaration encountered in each header file. If your existing source code has many classes per module then you should consider restructuring the source prior to doing your import. Restructuring will give a better import result.

The Subsystem Approach

If you are working on a project with more than a few classes then you should consider structuring your project as an assembly of subsystems. This does not mean rewriting code so much as arranging source files in appropriately named subdirectories. Javelin supports class files that are located in separate directories. To break a large system into subsystems first try and find classes that work together to form a similar function. Often classes within the same branch of a class hierarchy are candidates for the same subsystem. Sometimes there may also be a set of classes that help manage the classes in a branch of the hierarchy. These managing classes should also be part of the same subsystem. Make a well named subdirectory for each subsystem and move the class files to their appropriate subdirectories.

See also

Importing Constraints and Assumptions

Constraints And Assumptions:

Javelin imposes some constraints and makes some assumptions about the import process. Many of these are simply good programming practice anyway.

Javelin makes certain assumptions about the code being imported.

Compilation Units

Compilation units are the basic unit that a compiler sees when compiling. It is also the basic unit that Javelin deals with when importing. Here is the definition of a compilation unit in C++ and Java.

C++
A C++ compilation unit is typically a class source file (.cpp, .cc, .cxx) and a corresponding header file that declares the class(es) (.h, .hh). A C++ source file can include many other header files as well but for the purpose of importing a compilation unit into Javelin only the source file and its matching header file are important.

C++
A Java compilation unit is a single .java file.

One User class per Compilation Unit

User classes in the context of importing in Javelin are classes for which code generation and synchronization will take place when changes are made via the Javelin tool. Library classes on the other hand are classes which can be displayed in the design slate and used in inheritance and association relationships but for which no code generation and synchronization will take place.

When Javelin imports a compilation unit the first class declaration it encounters will become a user class (if user class is specified in the import specification file) but all classes defined after that class in the same compilation unit will be imported as library classes.

C++ Static Initializers

Static initializers are pieces of code that can be used to initialize static variables in a class when the application is started. Java allows many static initializers per class and the order of static initializers and the order of the appearance of static data declarations with initializers is important. Javelin supports a single static initializer. It will allow you to import more than one but they will be added to the generated source file in reverse order to that they were imported in. All data declarations will appear in order of access and then in alphabetical order in generated source file. After all data declarations will appear the method declarations. Included in this groups is the static initializers.

When importing code with static initializers and static data declarations with initial values it is important to check that the code generated from the import procedure performs in the same way as the original code given the above information concerning how Javelin deals with initializers.

Function and Data Declarations

C++ is very flexible in the way in which data and functions can be declared. As in C you can declare one or more data and functions in the same statement like this:

int Counter, Dog, Cat;

Java allows this type of declaration for data but not methods.

While this is may help reduce a few keystrokes it does nothing for the readability of the code, expecially when you consider how you would add a comment to each individual data declaration

Javelin will import a statement with multiple data declarations like that above. Each declarator will appear separately in Javelin and when code is generated for the class seperate declarations, each on their own line will appear.

However for function declarations only a single declaration is permitted per statement (the Java language confines you to this anyway). Any statements with multiple function declarations will be added to an appropriate "extras" section and will not be added explicitly to the database.

The code in function bodies can be made up of any tokens so that exceptions, classes unknown to Javelin etc., can all be used within function bodies. The only contraint is that every '{' is equally matched by a '}'. What goes on in between a '{' and '}' is irrelevent to Javelin.

C++ The rest of this topic refers to importing C++ classes only.

Preventing Multiple Inclusion of Header files

Javelin assumes that each header file has a

```
#ifndef __FNAME_H or #if !defined(__FNAME_H)
#define __FNAME_H

Code

#endif
```

construct to prevent multiple inclusion. Javelin generates its own #ifndef construct to prevent multiple inclusion of header files and thus will not import the first #if that it encounters in a header file.

Matching Function Declarations to Function Definitions

Javelin assumes that the naming of parameters in a function declaration is the same as in its definition. It uses this assumption when matching function definitions and function declarations during importing. If this is not the case then the code should be modified as appropriate. Javelin will, however, allow a function definition's arguments to have initializers. It will match the proper function definition which will not contain initializers.

Definitions

Javelin assumes that definitions of declarations in a header file are either contained within the header file as inline (in the case of functions) or in the associated source file. It will not attempt to look for them in other files.

Discerning Macro Invocations and function declarations

Macro invocations are sometimes used within a class declaration. These can look like declarations of functions that do not have a return type. Javelin will assume that constructs of the following forms are macros and place them in a keep section and not create a new member for them:

identifier;

identifier(param1, param2, param3, ...)   from 1 to 7 single token parameters separated by commas

Templates

Though the creation of template classes is supported by Javelin it is not possible to import template classes at the current version.

See also

        <u>Importing</u>
        <u>Building an Import Specification File</u>
        <u>Software Quality</u>

Building an Import Specification File

An import specification file (*.imp) is used during an import session to tell Javelin which files to import and where to place them. This is a text file which can be created automatically or manually using a text editor. Building an import specification file is the first step in the process of importing. Read <u>Importing</u> if this is your first import session.

Building an Import Specification File Automatically

1. Open the Build Import Specification File Dialog Box by selecting the File|Build Import Spec File... menu option.
2. Enter the name of an existing .imp file or a new one that you wish to create.
3. The Import Specification Dialog box should now be open.
4. Edit the fields as appropriate (ensuring that destination directory is different to source directory)
5. Press Add Filenames to Imp. This adds all class filenames found in the source directory to the .imp file.
6. Repeat steps 4 and 5 for different source and destination directories if required.

The controls of the dialog box are as follows:

C++ Header Extension

The file extension of header files that you wish to import (typically h). Do not add the '.'

C++ Source Extension

The file extension of source files that you wish to import (typically cpp). Do not add the '.'

Library

When this box is checked it specifies that all class filename entries added to the .imp file will become library classes for which no code generation and synchronization will take place.

C++ Persistent

When this box is checked it specifies that the classes will be persistent. ie., Javelin will generate special code depending on the type of the persistence you have specified in Options|Persistence.

Source Directory

Directory which contains the files to be imported

Destination Directory

Directory to which files will be generated when the import process is started

Add Import Filenames

Pressing this button will invoke a search for files in the source directory which have the given header extension in the case of C++ or the .java extension in the case of Java. Each file found will be appended to the end of the current .imp file.

NOTE: This button does not perform the import itself. This is done using File|Import...

Format of Import Specification Files (*.imp)

An import specification file (*.imp) contains a line for each class to be imported. Each line must contain fields of the following format:

```
filename hext* cppext* srcdir destdir flags
```

*hext and cppext are only used for C++ not Java. Omit these in the case of Java importing.

where:

filename - name of file to import (no extension or directory),

hext - header extension (no .)

cppext - source extension (no .)

srcdir - source path (no trailing slash)

destdir - destination path (no trailing slash) - NOTE: This MUST NEVER be the same as srcdir!

flags -   3 or 4 char string made up of L|U P|T O|N [A] eg., LTO with no spaces

L|U Library or User class, P|T Persistent or Transient class,

C++ O|N Object Identity functions or No Object identity functions and

[A] (optional) specifies that the class is abstract. See User Manual for explanations of these terms.

An example .imp file might look like this for C++:

```
person h cpp src . UPNA
employee h cpp src . UPN
```

An example .imp file might look like this for Java:

```
person src . UA
employee src . U
```

WARNING: The source and destination paths should be different for each file used for a user class because Javelin will generate code to the destination path. If this is the same as the source path then the ORIGINAL FILE MAY BE OVERWRITTEN.

See also

Importing

Contraints and Assumptions

Software Quality

Javelin allows you to extract design information from the tool via DDE. You can use the word macro in OODESIGN.DOT from within Word for Windows to extract a design and built an impressive document. To read more about this feature please read the separate document supplied with the application called DDEMAN.DOC. This document is in Word for Windows format.

Differences between Javelin and JAVelin

Javelin is designed for C++ developers while JAVelin is designed for JAVA developers. Because of this the dialog boxes for classes, data members and function members have slightly varying options based on the different features of each language.

Differences between standard and professional versions

The professional version of each tool allows associations between classes to be modelled. This makes it possible to model one-to-many relationships etc., between classes. The standard editions permit only inheritance (and "Implements" - JAVA specific) to be modelled.

Code Generation

JAVA does not have a concept of header and source file as does C++ so JAVelin generates and manages only a single file for each class while Javelin manages both a header and source file. JAVelin does not generate a genclsid.h file as does Javelin whenever a new class is added or a class name is changed.

Implementation Relationships

C++

Implements is a JAVA specific relationship and is used when a class implements a particular Interface or set of methods gathered in a class like structure called an interface.

Showing Members In Class Icon

By default the names and type of all non private members are displayed in the class icon when the members are created. You can turn this feature on or off for each individual member using the check box in the data or method's dialog box.

Storing of Objects

Depending on your application you may need the ability to generate code to assist in the storage of your objects so that they can be brought back to life after shutting down your application. The ability of an object to "persist" beyond the life of the application is known as persistence.

The persistent options dialog box can be displayed by choosing Options|Persistence. This lets you choose from a number of object persistence libraries (such as Borland C++ persistence class library 3.x or 4.x-5.x), various object databases or customized.

C++   Borland 4.x-5.x object streaming

Borland 4.x-5.x object streaming

This section explains how to take advantage of the very automated support for this type of obejct streaming.

If you have selected Borland 4.x-5.x object streaming then do not derive your classes from the TStreamableBase as this is done automatically for any class that has persistence enabled and does not already have a base class.

Code generated in read/write functions

The code generated for data members in the className::read::Streamer() and className::write::Streamer() functions follows a simple format.

If the type of the data is char*, char *, or LPSTR then the code generated will be

```
GetObject()->name = in.readString();
out.writeString(GetObject()->name);
```

If the type of the data is char then an assumption is made that it is a fixed length array of char and the following is generated:

```
char *tmpname = in.readString();
strcpy(GetObject()->name, tmpName);
if ( tmpName )
    free(tmpName);

out.writeString(GetObject()->name);
```