
Zim Class Specification

Zim

Version 1.3

Document Information:			
File Name:	zcs.doc	Printed on:	8/31/2001 10:26 AM
Date of Last Edit:	03/19/99 2:16 PM	Number of Pages:	30
Last Saved By:	Nicholas Henry	Saved Version:	9

Revision History

Version	Date	Reason
1.2	03/19/99 2:16 PM	Renaming the ZOF specification as the Zim Class Specification. Converted from HTML to a Word document to support ZIM TECHNOLOGIES 'white paper' publishing standards.
1.3	07/08/99 10:49 AM	<p>In Section 4.1.1 the class interface description suggested that the class name be passed for delegated calls. In fact this is true only for inherited calls. This has been correct.</p> <p>Rewrote notes 5 and 6 on page 19 to read more clearly.</p> <p>Section 4.1.3 defines exception handling.</p> <p>Section 4.4 defines abbreviation rules to create readable short names.</p> <p>Section 4.5 defines documentation rules for Zim classes.</p>

Table of Contents

REVISION HISTORY.....	I
TABLE OF CONTENTS.....	II
1 ZIM CLASS SPECIFICATION (ZCS).....	1
1.1 WHAT IS THE ZCS?	1
1.2 BACKGROUND	1
1.3 BENEFITS	1
2 OBJECT DEVELOPMENT IN ZIM	2
2.1 OBJECT AND CLASS.....	2
2.1.1 <i>User Interface Classes</i>	2
2.1.2 <i>Business Classes</i>	3
2.2 ATTRIBUTES AND BEHAVIOUR	3
3 DEFINING A ZIM CLASS	3
3.1 CREATING A CLASS	3
3.2 DEFINING A METHOD	4
3.3 METHOD DISPATCHER.....	5
3.4 CONSTRUCTOR AND DESTRUCTOR METHODS	6
3.5 PROTOCOL	6
3.6 INHERITANCE.....	7
3.6.1 <i>An Example of Run-time Inheritance</i>	9
3.7 PACKAGES.....	12
4 SPECIFICATION STANDARDS	12
4.1 PROTOCOLS	12
4.1.1 <i>Class Interface</i>	12
4.1.2 <i>Method Interface</i>	13
4.1.2.1 Public Methods	13
4.1.2.2 Private Methods	13
4.2 NAMING CONVENTIONS	13
4.2.1 <i>Notes</i>	17
4.3 LAYOUT AND STYLE	17
4.4 ABBREVIATION RULES	20
4.4.1 <i>Abbreviation Guide</i>	20
4.4.2 <i>Translation Descriptions</i>	21
4.5 DOCUMENTATION	21
4.5.1 <i>General Comments</i>	21
4.5.2 <i>Tag Summary</i>	23
4.5.3 <i>Class Comments</i>	22
4.5.4 <i>Method Comments</i>	22
4.5.5 <i>Example</i>	23
APPENDIX A – DEFINING ZIM CLASSES USING THE DEVELOPMENT CENTRE	26
DEFINING A ZIM CLASS IN THE OBJECT DICTIONARY	26
INHERITING ATTRIBUTES	27

1 Zim Class Specification (ZCS)

The purpose of this document is to specify how to write a Zim class.

1.1 What is the ZCS?

The Zim Class Specification provides a standard for the defining and implementing classes in Zim. Standards have been defined for:

- Class Interface
- Naming Conventions
- Layout and Style

1.2 Background

The Zim Class Specification is a refinement of the Zim Object Framework (ZOF). Originally ZOF not only described a method for defining and implementing classes in Zim, but also proposed a framework of base and system classes.

The specification component of ZOF has been extracted and converted to a 'white paper' called the Zim Class Specification. Any frameworks will be delivered as separate components, written to the ZCS specification. The ZCS specification is evolving and will be enhanced to incorporate other features over time.

1.3 Benefits

By using the Zim Class Specification, Zim developers will benefit from employing object based techniques. These benefits include:

[Harmon 93]:

- Faster development
- Increased Quality
- Easier maintenance
- Enhanced modifiability

[Booch 94]:

- Reuse of software and designs, frameworks

- Systems more change resilient, evolvable
- Reduced development risks for complex systems, integration spread out
- Appeals to human cognition, naturalness

2 Object Development in Zim

2.1 Object and Class

Although an object is an instance of a class it is easier to describe a Zim object first. An object is the encapsulation of attributes and behaviour.

Attributes values are stored on an Attribute Structure implemented using a form entity. State values are also stored on this structure (e.g. object modified).

Behaviour is described by methods, which are executed from a text or binary file.

A class is a template for multiple objects. In Zim, the instantiation of an object occurs when a class is assigned attributes. Only one object at a time can be instantiated. Zim does not currently support multiple instantiation.

Zim Classes can be divided into two main categories, User Interface and Business Classes.

2.1.1 User Interface Classes

User Interfaces (UI) Objects consists of windows, displays, forms, menus, toolbars and the behavioural code to operate these objects. UI Classes provides views of sets produced by Business Classes.

For example, the UI class *zCustomerUI* provides a view of the set *sCustomer* produced by *zCustomer*, a business class. There can be multiple views of the same set, for example *zCustomerFormUI*, which provides a view of a single record and *zCustomerTableUI*, which provides a view of multiple records.

There may also be other UI classes that do not provide an interface to a set, for example, a printer set-up dialog.

2.1.2 Business Classes

A business class derives its name from the set. For example, the set *sCustomer* is encapsulated by the business class *zCustomer*. Of course a set could be a complex object, for example, Order Header and Order Items which would be translate to the class *zOrder*.

2.2 Attributes and behaviour

To create a class the attributes and behaviour must be described. An attribute structure is defined in the object dictionary for the purpose of storing attribute values - state information and parameters. The attribute structure name is derived from the root class name prefixed with a lowercase 'a'. For example, the class *zCustomer* has an attribute structure *aCustomer*.

The behaviour is described using the Zim language as before. Previously, behaviour was described in procedures and local procedures, behaviour is now described in methods.

3 Defining a Zim Class

3.1 Creating a Class

A class in Zim consists of a document and an attribute structure defined in the Object Dictionary. The class declaration is defined in the document text file.

A class declaration:

```
class zCustomer(viMethod, viSelf)
endClass
```

As you can see PROCEDURE and ENDPROCEDURE are no longer used. These have been replaced with two new keywords CLASS and ENDCLASS. *viMethod* and *viSelf* are explained later in this document.

An attribute structure is defined once a class has been created. Attributes store an instance of the data object.

The attribute structure for $\mathcal{z}Customer$, $aCustomer$, might have the following fields:

FieldName	Type	Length	Description
ID	Int	10	Unique Identifier.
FirstName	Alpha	40	Class Attribute.
LastName	Alpha	50	Class Attribute.
Salutation	Alpha	10	Class Attribute.
StreetAddress	Alpha	80	Class Attribute.
City	Alpha	50	Class Attribute.
Country	Alpha	50	Class Attribute.
pMethod	Alpha	256	The method passed to this class. This parameter is a mandatory attribute for all structures.

3.2 Defining a method

```
method mAdd(viSelf)
endMethod
```

The keywords METHOD and ENDMETHOD have replaced LOCALPROCEDURE and ENDPROCEDURE.

The above method declaration does not contain any behaviour. The class $\mathcal{z}Customer$ below shows the class implementation, including behaviour for the add method.

```
%-----
method mAdd(viSelf)
    add Customer from aCustomer
endMethod
```

```

%-----
method mDelete (viSelf)

    delete sCustomer \
        where ID = aCustomer.ID

endMethod

%-----
method mpCustomer (viMethod, inout vtSelf)

    change aCustomer \
        let pMethod = viMethod

endMethod

%-----
method mpFinalize (viSelf)

endMethod

%=====
class zCustomer (viMethod, viSelf)

    mpCustomer (viMethod, viSelf)

    case
    when aCustomer.pMethod = 'add'
        mAdd (viSelf)

    when aCustomer.pMethod = ' delete'
        mDelete (viSelf)

    endCase

    mpFinalize (viSelf)

endClass

```

3.3 Method Dispatcher

Method dispatcher calls the appropriate method defined in messages that are sent to the class.

The method dispatcher is placed in the class body as a CASE statement. No behaviour is coded in the method dispatcher, only in the methods of the class.

Public methods, methods that can be called by another class, have a fixed protocol of one parameter, *viSelf*.

Private methods, methods that can be only called by the containing class, have no fixed number of parameters. These methods are never called in the method dispatcher or in the class body, except for constructor and destructor methods.

Section 4.3 Layout and Style describes a template for the dispatcher.

3.4 Constructor and Destructor Methods

Each class has two standard private methods - constructor and destructor.

The constructor method, which derives its name from the base name (e.g. *zCustomer* - *mpCustomer*), is invoked when the class is instantiated or called. The destructor method, *mpFinalize*, executes any clean up operations.

3.5 Protocol

A class has two parameters:

viMethod is used to pass the method to be executed by the class. The method is passed to the class as a parameter and is directly assigned to the attribute structure to improve readability and ensure upward compatibility. This notation is similar to other object-oriented languages that reference methods using the dot notation; e.g. *customer.add()*.

viSelf is used to pass the calling class base name. Passing this name allows a super-class that inherits a method to call or access the attributes of the subclass for run-time inheritance.

The base name is the name of the logical class. For example, the logical class *CustomerTableUI* is implemented as *zCustomerTableUI* (the class) and *aCustomerTableUI* (the attribute structure) in Zim.

Please review Section 3.6 for an example of run-time inheritance.

The message to an object consists of:

- data values assigned to the attribute structure
- any named sets

- the class call
- method
- self parameter

3.6 Inheritance

Inheritance provides a classification of objects that exploits their commonality. A hierarchy is formed where a child object inherits attributes and behaviour from a parent object. A parent may have many children, and that parent's parent may have many children.

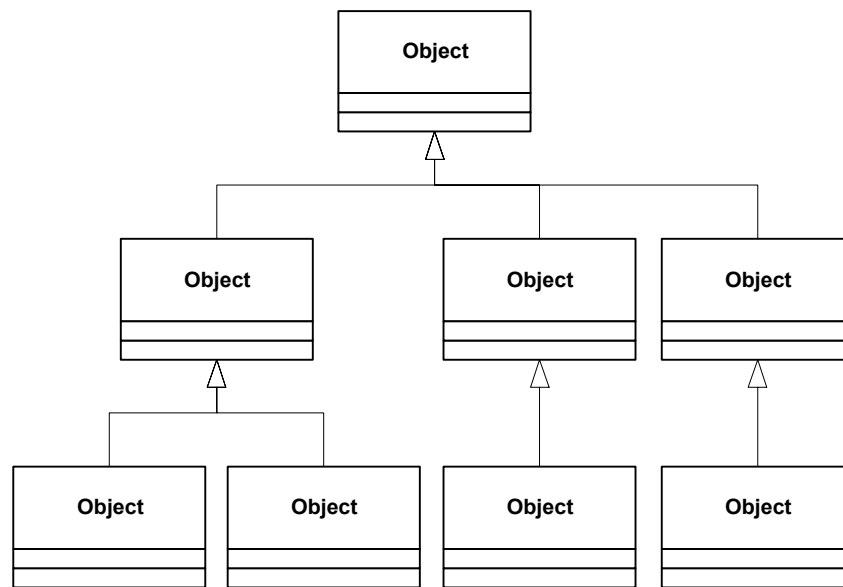


Figure 1: A class hierarchy

An example of a class hierarchy that is derived from an organisation domain may look like this:

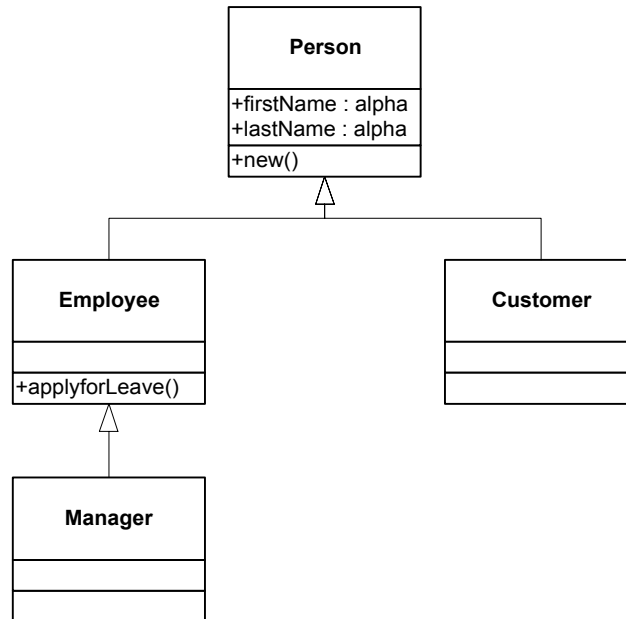


Figure 2: Class hierarchy derived from an organisation domain

In this example, the Manager object inherits the attributes *firstName* and *lastName* from the object *Person*. It also inherits the methods (behaviour) *new* (specifies how to create a new person) and *applyforLeave*.

By creating a hierarchy, the Zim developer can programme by extension rather than by re-invention [LaLonde 90].

ZCS does not define any standard inheritance mechanism. Zim developers can choose between generated (early binding) or run-time (late-binding) inheritance.

Generated inheritance uses the class hierarchy to generate a single executable class from parent source files. For example, *zManager* maybe generated by three templates, *tPerson*, *tEmployee*, and *tManager* to produce one class file.

Runtime inheritance uses the class hierarchy to execute parent classes on the fly. For example, to execute the method *applyForLeave*, *zManager* calls *zEmployee* passing its parameters to the parent class.

The *viSelf* component of the protocol provides the mechanism for run-time inheritance. In generated inheritance this would be ignored, but not omitted. This allows a generated class library to call a class library based on run-time inheritance.

3.6.1 An Example of Run-time Inheritance

Please note the following is only an example and that the ZCS does not standardise a method for inheritance.

Continuing the example from the previous section, the class $zManager$ inherits the method *'applyForLeave'* from $zEmployee$.

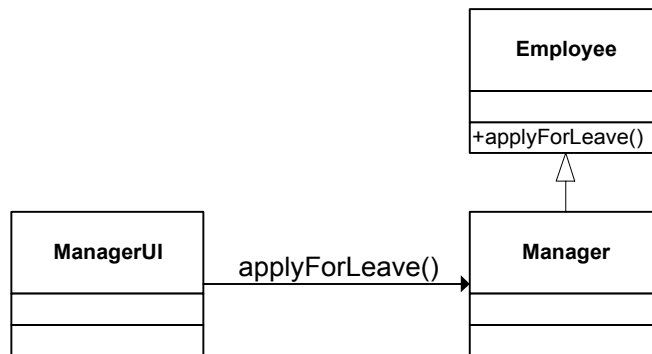


Figure 3: Manager inherits behaviour from Employee

In this example, the UI component $zManagerUI$ calls $zManager$ to *'applyForLeave'*. The call (known as a delegation) within $zManagerUI$ is implemented in Zim like this:

```
zManager('applyForLeave', '')
```

Note that *viSelf* is blank because the call is made across the hierarchy (a delegated call) not up the tree (an inherited call). The code for $zManager$ appears below. Notice that the method *'applyForLeave'* does not exist in $zManager$ and the method dispatcher calls $zEmployee$ using the otherwise component of the case statement.

Note that $zEmployee$ is an abstract class. This means that the class will only be executed using an inheritance call, not a delegated call. $zEmployee$ is an abstraction of $zManager$.

The code for $zManager$ is shown below followed by $zEmployee$.

```

%-----
method mAssignEmployee(viSelf)

    add ManagerEmployee from aManager

endMethod
  
```

```
%-----  
method mpManager(viMethod, inout vtSelf)  
  
    change aManager \  
        let pMethod = viMethod  
  
    let vtSelf = \  
        {vtSelf where vtSelf < '', 'zManager'}  
  
endMethod  
  
%-----  
method mpFinalize(viSelf)  
  
endMethod  
  
%=====  
class zManager(viMethod, viSelf)  
  
    mpManager(viMethod, viSelf)  
  
%Methods  
    case  
        when aManager.pMethod = 'assignEmployee'  
            mAssignEmployee(viSelf)  
  
        otherwise  
            zEmployee(aManager.pMethod, viSelf)  
  
    endCase  
  
    mpFinalize(viSelf)  
  
endClass
```

```

%-----
method mApplyForLeave (viSelf)

    change employee \
        where ID = aEmployee.ID \
        let ApplyForLeave=$true

endmethod

%-----
method mpEmployee (viMethod,viSelf)

    let <Self> = $replace(viSelf, 1, 1 a)
    change aEmployee from a#<Self>

    let aEmployee.pMethod = viMethod

endMethod

%-----
method mpFinalize (viSelf)

    change a#<self> from aEmployee

endMethod

%=====
class zEmployee (viMethod, viSelf)

    mpEmployee (viMethod,viSelf)

    case
    when aManager.pMethod = 'applyForLeave'
        mApplyForLeave (viSelf)

    endCase

    mpFinalize (viSelf)

endClass

```

There are five points to note in the code of *zEmployee*.

1. *mpEmployee* moves the attributes across from *aManager* to *aEmployee*.
2. *zEmployee* manipulates the structure of *aEmployee* only.
3. *mFinalize* moves the modified attributes back to *aManager*.

An example of a class library using a complete run-time inheritance mechanism is the [ZCL](#).

Please remember there are other variations of run-time inheritance that can be implemented. For example inheritance may be table driven. A child looks up a parent in a table and then the class is made to the parent. The parent still needs to know the child (for transferring of attribute values) and this is passed via *viSelf*.

3.7 Packages

Packages are a means to group classes into logical groups. The equivalent in Zim is the object directories. The directory allows you to manage a large group of classes and avoid naming conflicts. Classes are created in a directory exactly the same as other Zim objects.

4 Specification Standards

The information contained in this section is subject to change without notice.

4.1 Protocols

4.1.1 Class Interface

Class interface consists of the following:

Name	Description	Example
Class Name	The name of the class.	zCustomer
Method	The method to be executed.	add
Self	The child class.	Refer Section 3.6.
Attribute Structure	The attribute structure associated with the class.	aCustomer
Named Set	The named set associated with the class.	sCustomer

The interface of a Zim class is:

```
classname(method, self)
```

An example call to a Zim class:

```
zCustomer('add', '')
```

Note: Self is only used for inherited calls, not delegated requests. Please refer to the Inheritance section.

4.1.2 Method Interface

4.1.2.1 Public Methods

Other classes and the owning class can call a public method. Methods are defined in the method dispatcher.

The interface of a public method:

```
method(self)
```

An example call to a Zim class:

```
mAdd(viSelf)
```

4.1.2.2 Private Methods

The owning class can only call a private method. Private methods can have any number of parameters.

The interface of a private method:

```
method(<<param>>)
```

An example of call to private method:

```
mpGetStatus(viStatus, voMessage)
```

4.1.3 Exception Handling

Exceptions are passed via the standard exception attribute *oException* (e.g. *aCustomer.oException*).

To throw an exception, a string is assigned to the *oException* attribute and the 'set exception error' is executed.

```

method mAddItem (viSelf)

  % Some processing here
  %

  if $lastmember('sShoppingCart') = 0

    change aMakeAnOrder \
      let oException = 'noShoppingCartToUpdate'
      set exception error
    endif

  % Some more processing here
  %

endMethod % mAddItem

```

The client class catches the exception (i.e. the class that called the class that threw the exception). The code that catches all exceptions is as follows:

```

method mCallMakeAnOrder (viSelf)

  % Assign attributes here
  %

  zMakeAnOrder('addItem','')

  if (let aMakeAnOrder.oException = aShoppingCart.oException) > ''
    set exception error
  endif

endMethod % mCallMakeAnOrder

```

The code that catches specific exceptions is as follows:

```

method mCallMakeAnOrder (viSelf)

  % Assign attributes here
  %

  zMakeAnOrder('addItem','')

  if (let aMakeAnOrder.oException = aShoppingCart.oException) = \
    'noShoppingCartToUpdate'
    % do something specific
    %
  endif

endMethod % mCallMakeAnOrder

```

4.2 Naming Conventions

Item	Prefix	Suffix	Example	Notes
Class Items				
Class Business	z		zCustomer	1, 3
Class User Interface	z	UI	zCustomerUI	2, 3
Method	m		mOpen	
Constructor Method	mp		mpCustomer	
Finalize Method			mpFinalize	
Method Private	mp		mpAutoIncrement	4
Method External	z	<method>	zCustomerAdd	3
Attribute Structure	a		aCustomer	
Attribute Field Name Input	i		iAutoIncrement	
Attribute Field Name Output	o		oStatus	
Attribute Field Name Input/Output	<none>		FirstName	
Attribute FieldName Private	p		pMethod	
Parameter Input	vi		viMethod	
Parameter Output	vo		voKey	
Parameter Input/Output	vt		vtFirstName	
Parameter Local	vl		vlIsConnected	

Zim Objects			
Constant	c	cBlue	
Directory	<none>	Framework	
Display	d	dCustomer	
Document Structured	ds	dsCustomer	5
Document Unstructured	du	duCustomer	5
Entity	<none>	Customer	
Field	<none>	FirstName	
Form	f	fCustomer	
Menu	mn	mnCustomer	
Menu Item	mi	miSave	
NamedSet	s	sCustomer	
Relationship	<none>	Require	
Role	<none>	Cust	
Variable	v	vSetting	
Window	w	wCustomer	
Non-data Form Fields			6
Push Button	pb	pbCancel	
Scroll Bar	sb	sbCustomer	
Frame	fr	frGroup	
Image	im	imPhoto	
Arrays implemented with forms and displays			

Form	i	iTabControl	Item or instance
Display	r	rTabContorl	Array structure

4.2.1 Notes

1. Tables are named with a singular noun. The business class name that encapsulates a table(s) is a concatenation of the letter "z" and the table or named set name. The filename would have a ".z" file type e.g. *customer.z*
2. The user interface class filename would have a ".zui" file type e.g. *customer.zui*
3. Class names should be as short as possible and not more than 15 characters (including the "z"). This will leave a space for a type suffix e.g. *zCustomerUI* or a method name on an externalised method e.g. *zCustomerPrint*.
4. Private methods can have variable number of parameters.
5. It is recommended to use these prefixes for data storage documents. For example, an entity set Customer requires a temporary data storage document that is named *duCustomer*. Filenames for structured and unstructured documents are ".ds" and ".du" respectively. There may be times where a developer needs to use a document for persistent storage rather than an entity set. It would be permissible to omit the prefix in this case.
6. Prefixes are used for form fields that do not represent data attributes. For example the save button is labelled *pbSave*, not the prefix "*pb*". The form field *firstName*, which represents an attribute of Customer, has no prefixed.

4.3 Layout and Style

Feature	Rule
method	The structure of a method dispatcher follows this template:

dispatcher	<pre> Class z<ClassName>(viMethod, viSelf) mp<ClassName>(viMethod, viSelf) case when a<ClassName>.pMethod = 'method1' m<Method1> when a<ClassName>.pMethod = 'method2' m<Method2> endCase mpFinalise(viSelf) endClass </pre>
	the method name in the case statement matches the
case	<p>Zim commands, functions and system variables are in mixed case with the first alphabetic character in lower case</p> <pre> quitTransaction \$currentMember('sCustomer') \$setCount </pre> <p>object names are in mixed case with the first alphanumeric character upper case; name prefixes are in lower case</p> <pre> FirstName mpAutoIncrement aCustomer </pre> <p>method names are in mixed case with the first alphabetic character in lower case</p> <pre> where aCustomer.Method = 'setCreditLimit' </pre>
breaking and indenting lines	<p>use spaces, not tab characters</p> <p>indent 2 spaces (or logical multiples of) e.g. lining up components of a let statement requires 4 spaces</p> <pre> let aCustomer.FirstName = 'Gordon' \ aCustomer.LastName = 'Downie' </pre> <p>subcommands start on a new line</p> <pre> find all Customer Buy Product \ where LastName = 'Cohen' \ </pre>

```

    sorted by ProductName \
    keep Product \
    evaluate (let vlTotal = $total(Price)) \
    -> sProduct

```

subcommands in set specifications are indented 4 spaces

```

find all Customer \
    where FirstName = 'Robbie' and LastName = 'Robertson'
    -> sCustomer

```

when boolean expressions are broken, try to indent by logic

```

find 1 Customer \
    where LastName = 'Cummings' \
        and FirstName = 'Burton'
        or LastName = 'Cochrane' \
        and FirstName = 'Tom'

```

code in a class or local class is indented

the local variable declaration is indented

on/endon exception handler statements are not indented

```

method mpCalculateTotal(voTotal) \
    local (vlDownTo)

on error
    return
endOn

    let vlDownTo = $currentMember('sLineItem') - 1
    top sLineItem
    compute all sLineItem \
        evaluate (let voTotal = $total(Price))
        down vlDownTo sLineItem

endMethod

```

quoting use single quotes, except where double quotes are required

```

when aUI.iMethod = 'Add'
    let aCustomer.CompanyName = "Ted's Recycled CDs"

```

quote character strings

```

let aCustomer.LastName = 'Lightfoot'

```

spacing no spacing before the parentheses on class or function calls

```

zCustomer('add', 'CustomerUI')

```

```

$trim(LastName)

```

one space after evaluate subcommand

```
compute 1 Customer \
  where FirstName in ('Brad', 'Dan') \
    and LastName = 'Roberts' \
  evaluate (let vlTotal = Salary)
```

one space before a parentheses in a operator, no space after

```
evaluate (let vlCredit = Credit + (5 * Level))
```

single space surrounding operators

```
if vlSum > (5 * 3 + (4 + vlAmount))
```

single space after each comma in a call or list

```
$concat('Roch', ' ', 'Voisine')
```

```
LastName in ('Morissette', 'Murray', 'Dion', 'Lang',
'McLachlan')
```

single space before the line continuation backslash

```
find 1 Customer \
  where FirstName = 'Neil'
```

4.4 Abbreviation Rules

Currently Zim limits the name of an object to 18 characters. With the addition of prefixes this can reduce the root of the object name 15-17 characters. This section specifies how to obtain a meaningful object name when shortening an identifier that exceeds this limit.

4.4.1 Abbreviation Guide

Apply each question to the word that you are trying to shorten until a readable abbreviation is identified [McConnell 93].

- Does a standard abbreviation exist? – check the dictionary.
- Can nonleading vowels be removed? (Customer becomes *Cstmr*, Product becomes *Prdct*, Order becomes *Ord*)
- Can an acronym be created?

- Can each word be reduced to three or less letters?
- Can suffixes be removed? - such as *ing*, *ed*.
- Can the word be reduced to the first and last letters?

4.4.2 Translation Descriptions

To clarify abbreviations a translation description is invaluable. The translation is stored in *DDDDescriptions* for an object defined in the Object Dictionary. For parameter names, a translation is incorporated in description of the parameter name in the “@param” sequence.

```
%
% @param    CstmrID        the Customer ID is a unique identifier
%                               for the Shopping Cart
%
```

For more information on documentation please refer to the next section.

4.5 Documentation

The documentation framework is based on [JavaDoc's](#) specification provided by Sun Microsystems, Inc. Please follow their [recommendations](#) in writing comments.

4.5.1 Purpose

The main purpose of documenting a Zim class is to communicate the API. In the example below, the method *mAddItemToCart* belonging to the class *zMakeAnOrder* will add a product to the current shopping cart. This method requires one parameter, *ProductID*, and returns the set *sShoppingCart*.

4.5.2 General Comments

General comments to clarify code blocks is encouraged. Single comments start with “%” and are indented to the line that they refer too. A single “%” is used at the end of each comment to separate the comment from the code block.

```
% The following routine only runs on Wednesdays
%
%
if $weekDay($date) = 4
    let aCustomer.LastName = 'Rankin'
```

```
zCustomer('delete', 'CustomerUI')
endif
```

4.5.3 Class Comments

Documentation scoped for the entire class is placed at the top of the source file. A brief description of what the class does is followed with the `@author` and `@version` tags. Rather than using an author's name, the author's email address is used.

4.5.4 Method Comments

Each method has a description explaining its purpose. A short description is provided and a long description can be added with a line between.

```
%-----
method mAddItemToCart(viSelf)

% Adds a product item to the shopping cart.
%
% This product is related to the shopping cart via the
% the relationship Contains.
%
```

Each parameter passed on the attribute structure has a description. Objects and parameters that are returned are noted by the `@return` tag. Exceptions that may be thrown in a method are noted by the `@exception` tag. Note that the attribute name, not the structure name is described for only attributes are passed in on the class attribute structure (i.e., `zProduct` would not accept attributes from the structure `aCustomer`).

```
% @param      ProductID          the product to add to
%                                     the shopping cart
% @returns    sShoppingCart      the contents of the
%                                     customers shopping cart
% @exception  noRecordsFound     thrown when a shopping
%                                     cart does not exist
```

endMethod is commented with the method name.

```
endMethod % mAddItemToCart
```

4.5.5 Tag Summary

A number of tags derived from the JavaDoc standard are used to document Zim classes.

Tag Name	Scope	Description
@author	Class	Author of the class.
@version	Class	Version of the class.
@template	Class	The version of the template to generate the class skeleton.
@zcs	Class	The version of ZCS that the class complies to.
@todo	Method	Denotes a task must be done before the class is published.
@param	Methods and Constructor	Identifies a parameter. If an abbreviation is used for the parameter name, a translation will follow.
@return	Method	The objects or parameters returned by a method.
@exception	Method	Specifies the exceptions a method may throw.

4.5.6 Example

The example below shows how all the documentation components look together.

```

% zMakeAnOrder handles the making of an order. zMakeAnOrder
% communicates with zCustomer, zShoppingCart, and zOrder to
% generate an order. zMakeAnOrder allows customers to add and
% update their shopping cart. When a customer checks out, an
% order is generated from the contents of their shopping cart.
%
% @author      nhenry@ZIM Technologies.ca
% @version     1.0, 21-May-99
%
%
%-----
% Public Methods
%
%-----
method mAddItemToCart(viSelf)

```

```

% Request to add an item to the shopping cart.
%
% @param      ProductID          the product to add to
%                               the shopping cart
% @returns    sShoppingCart     the contents of the
%                               customers shopping cart
% @exception  noRecordsFound    thrown when a shopping
%                               cart does not exist
%

on error

change aMakeAnOrder \
  let oException = \
    {$LastErrMessage
     where aMakeAnOrder.oException is $null, \
     aMakeAnOrder.oException}
  return
endOn

mGetShoppingCart (viSelf)

if aMakeAnOrder.oException > ''
  and aMakeAnOrder.oException <> \
  'noRecordsFound'
  set exception error
else
  change aMakeAnOrder \
    let oException = $null
  endIf

% if shopping cart does not exist for customer, add one
%
if $lastmember(sShoppingCart) = 0
  compute 1 sCustomer \
    evaluate (let aShoppingCart.CustomerID = Customers.CC)
  zShoppingCart('add','')
  if (let aMakeAnOrder.oException = \
      aShoppingCart.oException) > ''
    set exception error
  endIf
endIf

compute 1 sCustomer \
  evaluate (let aShoppingCart.CustomerID = Customers.CC)

```

```

% check if the product is already in the cart, if it is,
% update the quantity instead of adding the item to the cart
% again
%
compute 1 sShoppingCart \
    where Contains.CustomerID = aShoppingCart.CustomerID \
        and Contains.ProductID = aMakeAnOrder.ProductID \
        evaluate(let aShoppingCart.QtyReqd = Contains.QtyReqd + 1)

if $membercount = 0

    zProduct('initAttr','')
    change aProduct \
        let ProdCode = aMakeAnOrder.ProductID
    zProduct('find','')

    if (let aMakeAnOrder.oException = aProduct.oException) > ''
        set exception error
    endIf

    compute sProduct \
        evaluate (let aShoppingCart.UnitPrice = \
                    Products.UnitPrice)

    change aShoppingCart \
        let ProductID = aMakeAnOrder.ProductID \
            QtyReqd = 1

    zShoppingCart('addItem','')

    if (let aMakeAnOrder.oException = \
        aShoppingCart.oException) > ''
        set exception error
    endIf

else

    zShoppingCart('updateItem','')

    if (let aMakeAnOrder.oException = \
        aShoppingCart.oException) > ''
        set exception error
    endIf

endIf

endMethod % mAddItemToCart

```

Other methods for *zMakeAnOrder* would appear here.

```

%=====
class zMakeAnOrder(viMethod, viSelf)

```

```

mpMakeAnOrder (viMethod, viSelf)

case
when aMakeAnOrder.pMethod = 'addItemToCart'
    mAddItemToCart (viSelf)

when aMakeAnOrder.pMethod = 'updateQuantity'
    mUpdateQuantity (viSelf)

when aMakeAnOrder.pMethod = 'getShoppingCart'
    mGetShoppingCart (viSelf)

when aMakeAnOrder.pMethod = 'checkout'
    mCheckout (viSelf)

otherwise
    zObject (aMakeAnOrder.pMethod, viSelf)

endCase

mpFinalize (viSelf)

endClass % zMakeAnOrder

```

4.5.7 Documentation Layout

Documentation must be laid out in the following format:

Column 1	Column 3	Column 15	Column 35
%	tag	identifier	description
%	@param	CustomerID	ID for the customer.

The columns refer to the column number within the source code. Source code should not exceed 80 columns.

Appendix A – Defining Zim Classes using the Development Centre

Please note, to build Zim classes you are required to have at least Zim 5.5 Revision 1.0.

Defining a Zim class in the Object Dictionary

To build a Zim class, follow this process:

- Open the Documents window from the Objects, Documents menu in the Development Centre.
- Create a document as normal, but set the Document Type as 'Class'. Please note pre-Aria Zim 2.1 or Zim 5.7 need to select 'ZOF Class'.
- Click the 'Fields' button to open the Fields window.
- Entering class attributes is the same as creating fields for a structured document.
- Once completed, exit from the Fields window and the Zim Class is created.
- This process creates the structure (known as a 'form ent' in the object repository). For Example, the class $\mathcal{C}ustomer$ creates a structure $aCustomer$ with fields name, address and phone (e.g. $aCustomer.Name$). A skeleton source file is generated when the class is created.

Inheriting Attributes

To inherit attributes from an object in the Object Dictionary, follow this process:

- Create a class attribute with the name '_IncludeOtherAttrs'.
- In the default field, list all classes/zim objects (entity sets, data relationships, forms) the structure should inherit from. Each class/object is to be separated by a comma.