

Cleanscape SourceMill

Automatic source code synthesis engine

User Guide

Version 1.0
Rev. 1 5.24.01

Copyright 2001 Cleanscape Software International. All rights reserved
Cleanscape SourceMill User Guide for Windows
Rev. 1 - 5.24.01 bd

Note: Licensed users may photocopy for distribution.

This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Cleanscape Software International.

Cleanscape Software International assumes no responsibility or liability for any errors or inaccuracies that may appear in this documentation. Except as permitted by such license, no part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Cleanscape Software International. Cleanscape reserves the right to update this document without prior notification.

Cleanscape SourceMill is a registered trademark of Cleanscape Software International.

UNIX is a registered trademark of AT&T Bell Laboratories.

DEC, ULTRIX, VAX, and VMS are registered trademarks of Digital Equipment Corp.

Other trademarks may apply.

Comments to:

Cleanscape Software International
2231 Mora Dr Ste E
Mountain View, CA 94040
650 864-9600 **Main**
650 864-9500 **Fax**
800 944-5468 **Sales**

E-mail: sourcemill@cleanscape.net

Document Overview

- 1 Application overview**
- 2 Getting Started**
- 3 Developing application frameworks**
- 4 User interface**
- 5 Object Model Reference**
- 6 Template Reference**
- 7 Provided templates and object models**
- 8 Appendix: Instantiating Code Patterns**

Contents

| | | |
|----------|---|------------|
| 1 | Application overview | 1-1 |
| | Summary | 1-1 |
| | Tackling the dangers of redundant development practices | 1-1 |
| | Bridge design and coding..... | 1-1 |
| | Use a process that provides proportional improvement..... | 1-2 |
| | Use common sense software development | 1-2 |
| | Integrate instantiating components | 1-2 |
| | Implement rapid application framework development..... | 1-2 |
| | Establish and standardize process..... | 1-2 |
| | Automate development | 1-2 |
| | Create opportunity for reaping proportional benefits | 1-2 |
| | Reallocate saved resources wisely | 1-3 |
| | Features | 1-3 |
| | Benefits..... | 1-3 |
| | Faster..... | 1-4 |
| | Better..... | 1-4 |
| | Smarter..... | 1-4 |
| | Cheaper | 1-4 |
| | Cleaner | 1-4 |
| | Specifications | 1-4 |
| | Minimum system requirements..... | 1-4 |
| | Languages | 1-5 |
| 2 | Getting Started | 2-1 |
| | Summary | 2-1 |
| | Installing..... | 2-1 |
| | Configuring | 2-4 |
| | AUTOEXEC.BAT..... | 2-4 |
| | SMILL.INI..... | 2-5 |
| | Operating..... | 2-6 |
| | Start Cleanscape SourceMill..... | 2-6 |
| | Main Menu..... | 2-7 |
| | Sample Execution | 2-8 |
| | Uninstalling | 2-11 |
| 3 | Developing application frameworks..... | 3-1 |
| | Summary | 3-1 |
| | Application overview | 3-1 |
| | Data Structure Model (DSM)..... | 3-2 |
| | Creating an object model..... | 3-2 |
| | Examples..... | 3-3 |
| | Templates (TEM) | 3-8 |
| | Selecting a Template..... | 3-8 |
| | Specifying an output directory..... | 3-9 |

| | |
|--|------------|
| Creating a template..... | 3-9 |
| Template File examples..... | 3-10 |
| Projects..... | 3-13 |
| Creating a Project | 3-13 |
| Popup Menus..... | 3-13 |
| Cut, Copy and Paste | 3-13 |
| Drag and Drop..... | 3-13 |
| Generation Rules | 3-14 |
| 4 User interface | 4-1 |
| Summary | 4-1 |
| Toolbar..... | 4-1 |
| Menus | 4-2 |
| File Menu | 4-2 |
| Edit Menu | 4-3 |
| View menu | 4-3 |
| Add menu | 4-4 |
| Run menu | 4-4 |
| Options menu | 4-4 |
| Window menu | 4-5 |
| Help menu | 4-5 |
| Other Support Available | 4-6 |
| Contacting Cleanscape Software International | 4-6 |
| OO Design Services | 4-6 |
| OO Training | 4-6 |
| Template Construction | 4-6 |
| 5 Object Model Reference | 5-1 |
| Summary | 5-1 |
| Object Model Statements..... | 5-1 |
| DSM Event List..... | 5-1 |
| DSM Object Definition | 5-2 |
| DSM Function Definition statement | 5-3 |
| DSM State Definition statement..... | 5-5 |
| DSM Process Definition statement | 5-6 |
| DSM Instance Definition statement | 5-7 |
| DSM Literal Text statement | 5-7 |
| DSM Foreign Type statement | 5-8 |
| DSM Pragma statement..... | 5-9 |
| DSM Import statement | 5-9 |
| DSM Comment | 5-10 |
| Object Model Keywords..... | 5-10 |
| C++ Language Keywords..... | 5-10 |
| Ada Keywords..... | 5-10 |
| Object Model Characteristics..... | 5-11 |
| Example..... | 5-11 |
| Object Model Rules | 5-12 |
| General Rules | 5-12 |
| Naming Rules..... | 5-12 |

| | | |
|----------|--|------------|
| 6 | Template Reference | 6-1 |
| | Summary | 6-1 |
| | Template File Global Variables | 6-2 |
| | Template File Single Line Statements..... | 6-3 |
| | .file statement..... | 6-3 |
| | Simple statement..... | 6-3 |
| | \$\$Directives | 6-4 |
| | Loadtime \$\$IMPORT Directive | 6-7 |
| | Template File Block Statement | 6-7 |
| | .block statement | 6-8 |
| | .select statement | 6-8 |
| | .module statement | 6-10 |
| | Template File Iterator Statements | 6-10 |
| | .each_characteristic statement..... | 6-11 |
| | .each_event statement | 6-11 |
| | .each_pragma_value statement | 6-12 |
| | .each_var_item statement..... | 6-13 |
| | .each_token statement..... | 6-14 |
| | .each_literal statement | 6-15 |
| | .each_foreign_type statement | 6-16 |
| | .each_object statement | 6-16 |
| | .each_parent statement | 6-18 |
| | .each_ancestor statement..... | 6-19 |
| | .each_used_obj statement | 6-20 |
| | .each_using_attr statement..... | 6-21 |
| | .each_attribute statement..... | 6-22 |
| | .refd_obj statement..... | 6-23 |
| | .each_using_attr statement..... | 6-24 |
| | .each_function statement..... | 6-25 |
| | .each_parameter statement..... | 6-26 |
| | .each_state statement | 6-28 |
| | .each_transition statement..... | 6-29 |
| | .each_proc statement..... | 6-30 |
| | .each_flow statement | 6-31 |
| | .each_instance statement..... | 6-33 |
| | .each_assignment statement..... | 6-34 |
| | .each_parameter statement..... | 6-35 |
| | .each_transition statement..... | 6-37 |
| 7 | Provided templates and object models..... | 7-1 |
| | Summary | 7-1 |
| | Additional online help | 7-1 |
| | Production templates | 7-1 |
| | Class per Object Using MFC 4.0 | 7-1 |
| | Class per Object Using Provided Lists | 7-2 |
| | Class per Object Using STL | 7-2 |
| | Class per Object Using RogueWave..... | 7-3 |
| | ANSI C Struct per Object | 7-3 |

| | |
|--|------------|
| Ada-83 Managed Record per Object..... | 7-4 |
| K&R C Struct per Object | 7-4 |
| C++ Managed Class per Object..... | 7-4 |
| K&R C Managed Struct per Object | 7-5 |
| ANSI C Managed Struct per Object..... | 7-5 |
| Example Templates..... | 7-6 |
| Ada-9x Sample Template..... | 7-6 |
| C Based Finite State Machine | 7-6 |
| ParcPlace Smalltalk Class per Object..... | 7-7 |
| ANSI SQL DDL Table per Object..... | 7-7 |
| Sample Object Models..... | 7-8 |
| 8 Appendix: Instantiating Code Patterns | 8-9 |
| Introduction..... | 8-9 |
| Design patterns and code patterns..... | 8-9 |
| Using code patterns..... | 8-10 |
| Pattern instantiation | 8-10 |
| Strategize..... | 8-11 |
| Design..... | 8-11 |
| Execute | 8-11 |
| Reading template statements..... | 8-27 |
| Code for handling contained pointers | 8-29 |
| Conclusion | 8-29 |

1 Application overview

Summary

Welcome to Cleanscape SourceMill, a powerful automatic source code synthesis engine that facilitates cross-platform software development by automating repetitive coding tasks. This chapter of *Cleanscape SourceMill User Guide* provides an overview of the application's function, features, benefits, and specifications. It includes the following sections:

- Tackling the dangers of redundant development practices
- Features
- Benefits
- Specifications

Tackling the dangers of redundant development practices

Developing cross-platform software typically requires programmers to manually re-code the application for each environment. Manually coding by memory or by modifying copied patterns can add to a development project cumbersome detail that is highly resource intensive, is subject to programmer error, and makes it virtually impossible to ensure changes are implemented across each environment.

Using an instantiating tool to synthesize application design with code patterns, programmers can reuse—rather than recreate—code to automatically generate application frameworks for multiple platforms and languages.

This is where Cleanscape SourceMill comes in. As an advanced instantiation tools, Cleanscape SourceMill is specifically designed to help software developers overcome the resource-intensive dangers of redundant development practices by giving them an automatic means by which they can do the following:

Bridge design and coding

Cleanscape SourceMill is an automatic code generation engine that bridges the gap between design and coding by automatically generating commercial-grade code for virtually any language or platform from object models and code patterns. Developers can reuse the patterns to automatically create code for multiple platforms by synthesizing the application design with a different template. This helps development teams to standardize cross-platform development, quicken the coding process, rapidly develop applications, reduce programming errors, increase product quality, and lower development costs.

Use a process that provides proportional improvement

These quality and productivity improvements increase proportionally to the amount of code generated by each coding strategy. The ratio of lines of code to lines of input is typically 20:1.

Use common sense software development

While the time and quality benefits of using Cleanscape SourceMill to automatically generate code can be immediate and significant, the application isn't revolutionary; it is simply a common-sense tool for helping developers to automate redundant programming tasks.

Integrate instantiating components

There are three basic components to Cleanscape SourceMill: an object model, a template, and the synthesis engine. The object model is like a formula or blueprint for building an application framework; it is a high-level structure you develop to describe your application. The template is like a storehouse of code patterns; it provides the code fragments and algorithms needed to instantiate the object model into an application framework for a given language or platform.

Implement rapid application framework development

After you've finished designing your application using Cleanscape SourceMill utilities you simply select a template for your target environment. In seconds, Cleanscape SourceMill synthesizes the formula in your object model with the ingredients in the template to generate an application framework with commercial-grade code.

Establish and standardize process

Automatically generating application frameworks with Cleanscape SourceMill allows development organizations to deliver higher quality products across platforms for faster delivery using fewer resources. It also allows analysts and managers to easily control consistency of output and interface across platforms, and helps programmers and engineers to increase the quality of their code.

Automate development

Cleanscape SourceMill is ideal for automating the development of object-oriented applications, well-structured legacy applications, or in any environment that needs to establish and automate adherence to local standards and guidelines. For example, Cleanscape SourceMill can be used to automate development and maintenance of GUIs, databases, wizards, and other operations that use object-oriented code.

Create opportunity for reaping proportional benefits

Cleanscape SourceMill continuously enhances application development productivity by creating consistency, and by enhancing the ability to rapidly develop and modify applications across multiple platforms and languages. Quality and productivity improvements increase proportionally to the amount of code generated when developers consistently use it in coding strategies.

Reallocate saved resources wisely

The price for getting the significant gains in consistency, speed, and quality derived from using Cleanscape SourceMill to automate code generation is to reallocate some of the resources you save to the most important stages of software development: planning, strategy, and design. (See Chart: “[Tradeoffs of automatic code generation](#)”)

Tradeoffs of automatic code generation

| Key advantages | Investment |
|--|--|
| <ul style="list-style-type: none"> • Consistent code. Conventions are followed without departure. The patterns that you use when you create code can be made into templates that will always be true to your rules. • Fast code. Large amounts of code can be generated automatically, and built into your application within the same hour. • Higher quality code with increased functionality. When you enhance a template, you enhance all the code that you will create from it. This is software reuse at its best. | <ul style="list-style-type: none"> • Planning. Templates take planning, design, and time to create. Creating templates is more challenging than creating special purpose code with an eye towards reuse; you’re solving the problem for multiple projects, not just for one. • Applicability. Templates are suitable for use when the usage situation matches the assumptions made when creating the template. You may need a diverse set of templates if your needs are also diverse. |

Features

Following are some of the key features that make Cleanscape SourceMill a powerful solution for helping you rapidly develop software applications:

- Powerful code synthesis engine produces thousands of lines of code in seconds
- Supports automatic source code generation for virtually any language or platform
- Automatically generates cross-platform application frameworks by synthesizing object models with templates
- Integrated Object Model Editor facilitates easy development and application design by allowing designers to define attributes, functions and characteristics from pre-formatted objects
- Application and components can be custom-configured to accommodate or standardize existing standards and practices
- Easy to use graphical user interface with contextual and on-line help
- Full set of pre-formatted code-creation rules facilitate easy transition from strategy to model
- Production and sample templates for C, C++, Java, Ada, Smalltalk, SQL, and others are ready to use or modify for specific projects
- Custom object model design and template building services available

Benefits

Following are some of the key benefits you should derive from using Cleanscape SourceMill in your software development process:

Faster

- Enhances a development organization's ability to rapidly develop and modify software applications
- Automatically generates thousands of lines of commercial-grade code in seconds
- Reduces coding phase by up to 60%
- Significantly reduces the debugging phase and virtually eliminates the manual coding of redundant tasks

Better

- Increases product quality by reducing programming errors
- Creates consistency across versions and platforms
- Quality and productivity increase proportionally to the amount of code that is produced from each model and template

Smarter

- Increases management effectiveness by providing a process by which development teams can define, develop, produce, control, and quickly modify development strategies and tactics
- Eases cross-platform development by bridging the gap between design and coding
- Allows flexibility to define how objects fit into local frameworks

Cheaper

- Reduces development costs by increasing efficiency and speeding development
- Allows test, coding, and debugging resources to be allocated to other tasks

Cleaner

- Improves practices by facilitating standardization and automation of current processes
- Reduces problems introduced by manual coding by more than 70%
- Automatically updates all generated code when model or template is modified

Specifications

It is available for most major platforms, including Linux, Unix, and Windows.

Minimum system requirements

- Processor: Intel Pentium 200 MHz
- Disk space: 10 MB for full install
- RAM: 64 MB RAM
- Operating System: Microsoft Windows 95, 98, NT, 2000

- ✓ For automatic source code generation in UNIX & Linux environments, see SNIP automatic source code generation engine, also available from Cleanscape.

Languages

Cleanscape SourceMill supports automatic coding for virtually any language. Version 1 comes with production and sample templates for the following languages:

- Ada
- C
- C+
- Fortran
- Java
- SQL

2 Getting Started

Summary

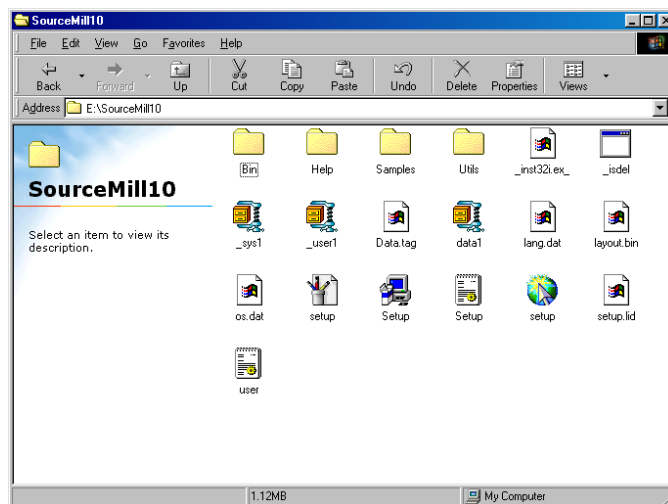
This chapter provides instructions on how to install, configure, and operate Cleanscape SourceMill, and includes the following sections:

- Installing
- Configuring
- Operating
- Uninstalling

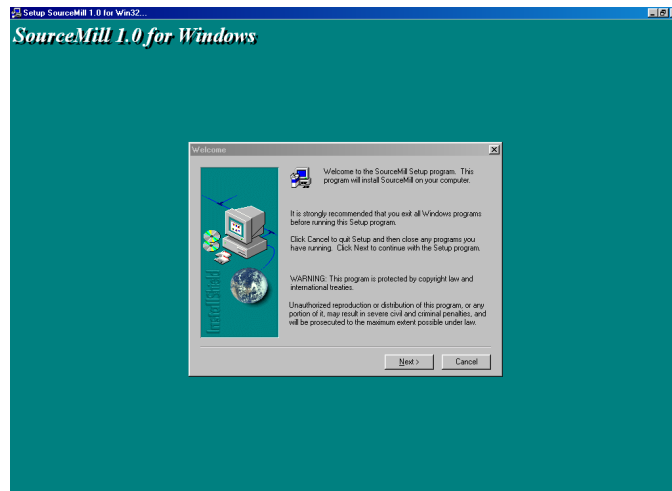
Installing

To install Cleanscape SourceMill, proceed as follows:

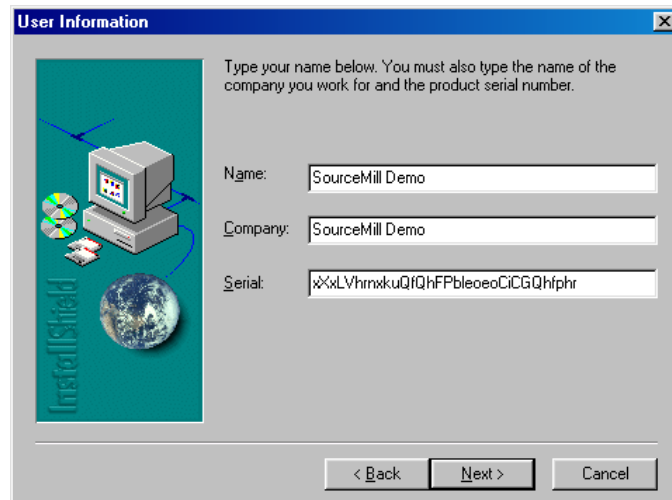
1. Under Windows 2000/NT, log on as “Administrator”.
2. Load the Cleanscape SourceMill CD and execute “setup.exe” on the media.



3. Follow the instructions given by the “setup” program.



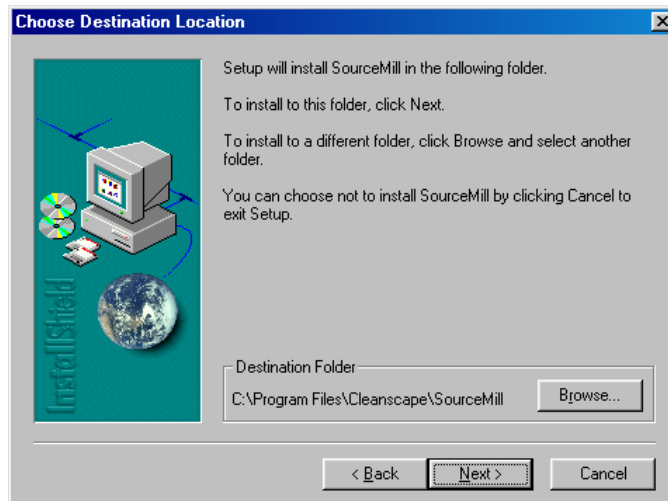
4. Enter your name, company and serial number.



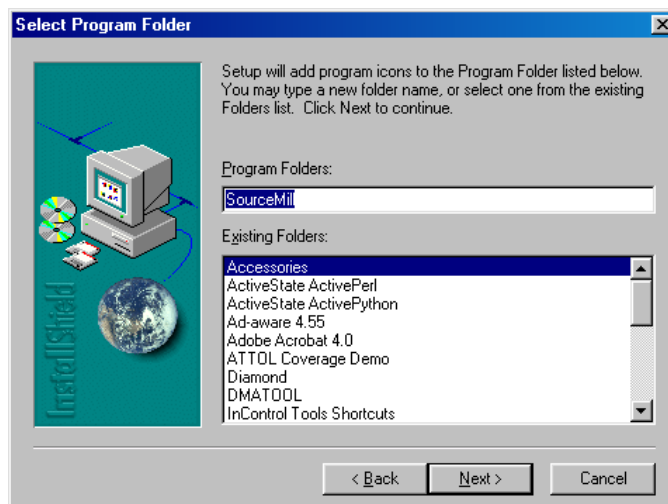
- ✓ If you are installing Cleanscape SourceMill for evaluation purposes, enter the following demo serial number:

XXxLVhrnxkuQfQhFPbleoeoCiCGQhfphr

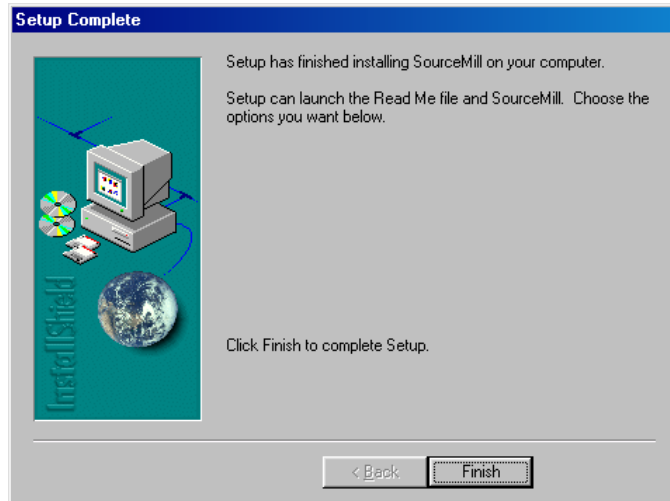
5. Select a destination folder. The default folder is: “C:\Program Files\Cleanscape\SourceMill”. To select another folder, press the Browse button.



6. Enter a name for Cleanscape SourceMill's program folder. The default is SourceMill.



7. Click Finish to complete the setup process.



Configuring

Cleanscape SourceMill allows you to configure its behavior through interactive dialogs that are available under the Options menu. Its dialogs in turn access and store information in the SMILL.INI file (described below).

AUTOEXEC.BAT

Templates that drive code generation are normally located in the same directory as the Cleanscape SMILL.EXE file (they are installed here initially), and Cleanscape SourceMill will look for them there by default. You can change the location of the template table, template files & template help files, and tell Cleanscape SourceMill where they are by setting the environment variable SMILLIB. For example, placed in AUTOEXEC.BAT, the line:

```
SET SMILLIB=C:\DATA\SMILL\TLIB
```

This tells Cleanscape SourceMill that the TEMPLATE.TBL file, the template files (*.TEM), and the help files that describe the templates (*.HLP) are located in the directory C:\DATA\SMILL\TLIB.

Cleanscape SourceMill looks for the SMILL.INI file in the directory indicated by the environment variable SMILLDIR. If this variable is not set, Cleanscape SourceMill looks for SMILL.INI in the \windows directory. The SMILL.EXE executable (batch version of the template expander) can also be invoked with the license directory passed on the command line.

SMILL.INI

The SMILL.INI file contains application the configuration variables you set up under the Options menu; you should be able to locate this file in your \Windows directory. The SMILL.INI file contents are described below:

| SMILL.INI Members | Variables | Explanation |
|-----------------------|--|--|
| [Recent File List] | File1=SCHOOL.DSM | File1 ... File4 in the [Recent File List] section are placed at the bottom of the File menu to allow fast access to the last 4 files you opened. These are set automatically and change dynamically. There is no need to edit these variables |
| [License Info] | Name=Jane Engineer Key=xhceOHigoGlg | You should not modify the [License Info] section. If you change the name or the key, you will invalidate the license check that the application runs when it starts. |
| [Browsing] | Editor=notepad.exe | The Editor variable in the [Browsing] section can be set to your favorite Windows text editor. 'notepad.exe' is the default editor, but the 16-bit version of notepad inadequate for large file viewing/editing. |
| [LastCodegenSettings] | | The [LastCodegenSettings] section contains several variables that are remembered from the last time code generation options were set. |
| | DSM_Template=C:\SMILL30\NETWORK.TEM | The pathname of the last selected template file. |
| | DSM_Outdir=C:\CODE | The directory where code files were placed during code generation. |
| | ShowFileNames=true | Set to true to enable reporting of which files are created during code generation, and how many files and lines of code were generated. If set to false, Cleanscape SourceMill runs code generation without notifications. (Can be set in Codegen Options dialog). |
| | StripReturns=false | Will create UNIX ready text files if set to true, as opposed to DOS ready files. The difference is that UNIX text files end lines with \n, while DOS requires \r\n. (Can be set in Codegen Options dialog). |
| | NoCxxKeywords=true | Set to true to forbid C++ keywords used in identifiers in the object model text. If generating code for a non-C/C++ language, you may want to set this to false. |

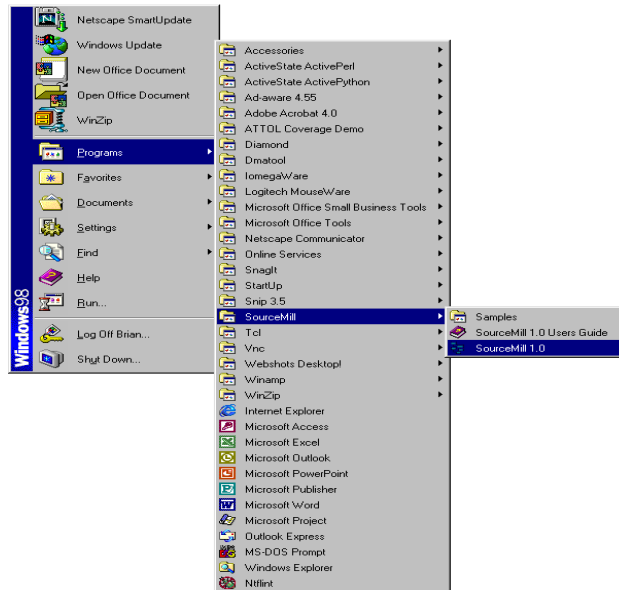
| | | |
|------------|------------------------|--|
| | NoAdaKeywords=false | Set to true to forbid Ada keywords used in identifiers in the object model text. If generating code for a non-Ada language, you may want to set this to false. |
| | MakeBackupFiles=true | Set to true to make Cleanscape SourceMill create backup files when replacing files that contain hand-edited code (such as UID tagged function bodies) that need to be preserved. |
| [Registry] | SetSMILLFileTypes=true | Set registry entries for Cleanscape SourceMill related files each time Cleanscape SourceMill is run. |

Operating

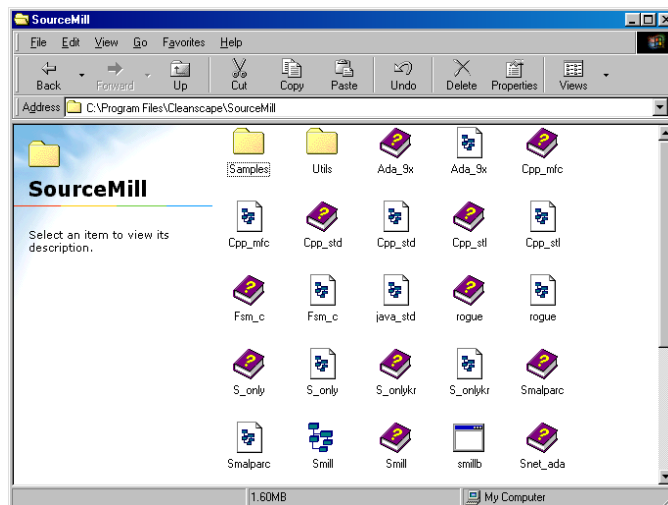
Start Cleanscape SourceMill

For administrator:

1. Select SourceMill 1.0 from the SourceMill program group on the Windows Start menu.



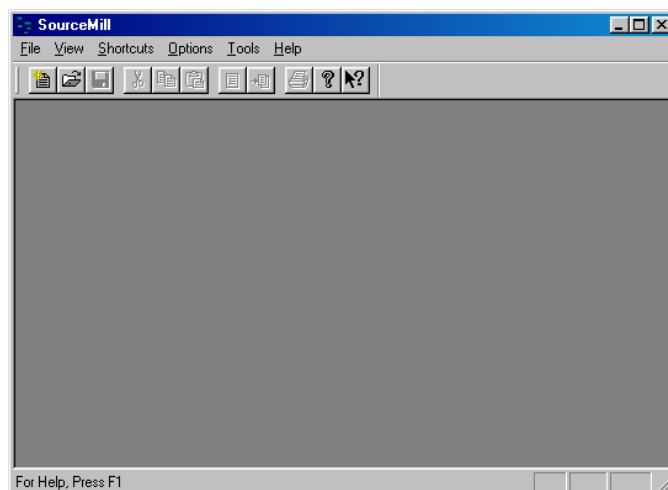
2. The default directory is: SourceMill from “C:\Program Files\Cleanscape\”.



Main Menu

When Cleanscape SourceMill is started, it displays a menu bar with five items, as follows:

1. The File menu is used to create or open Data Structure Model (DSM) files
2. The View menu is used to switch Toolbar and Status Bar on or off.
3. The Shortcuts menu is used to control the code generation behavior.
4. The Options menu is used to select code templates and program options.
5. The Tools menu is used to select the external editor.
6. The Help menu provides on-line documentation.

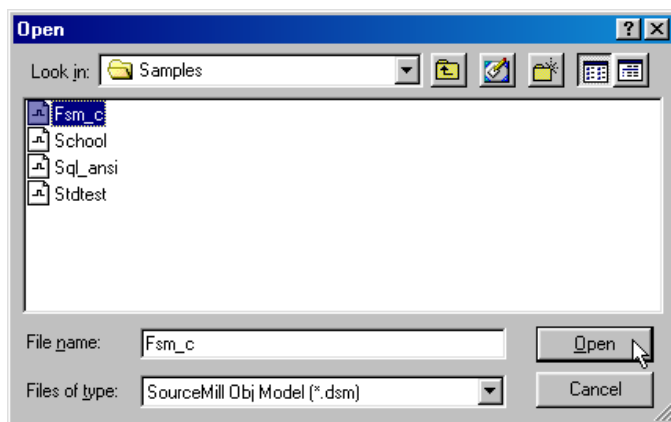
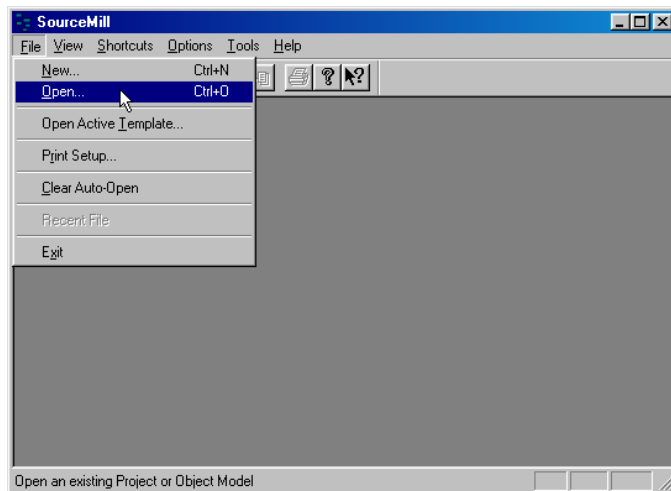


- ✓ For more information about the menu items, see [User Interface](#)

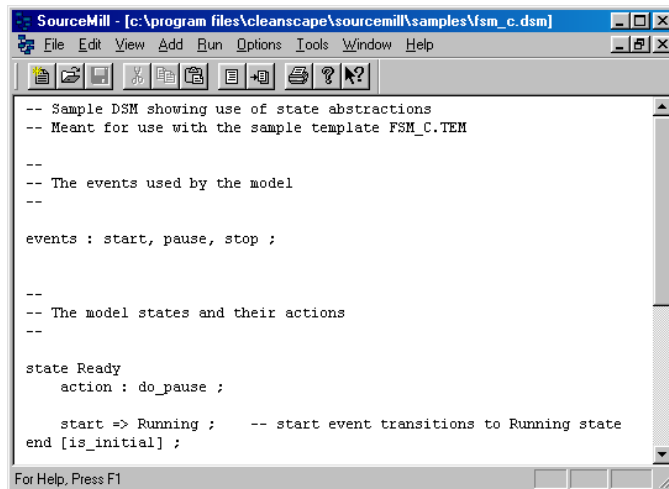
Sample Execution

Data Structure Model (DSM) file

1. Select and open a DSM file

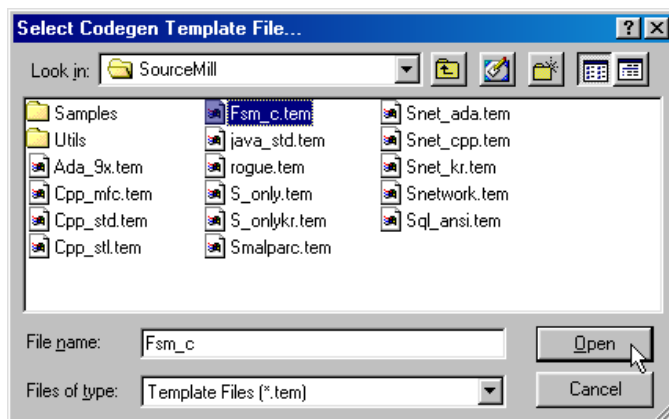
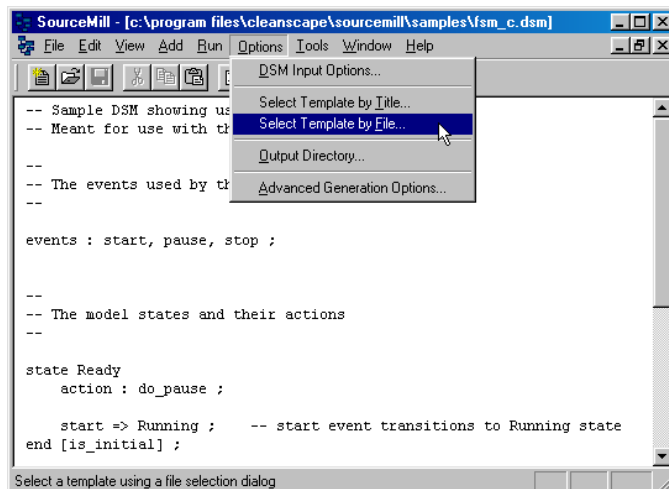


2. Edit the DSM file, if necessary.



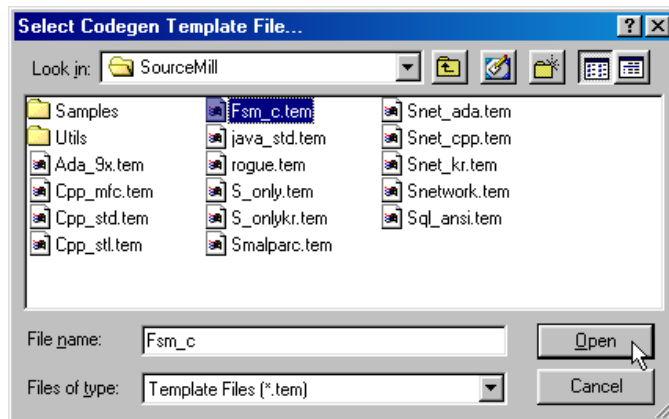
Template File

3. Select and open a template file.

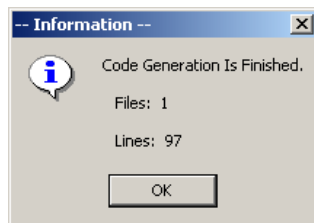


Generate Source Code

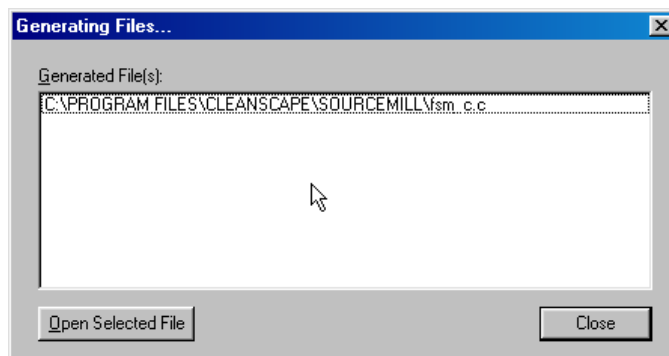
4. Press the Generate Code button on the shortcut bar.



5. After viewing code generation summary, press OK.



6. To view source code generated, press Open Selected File button.




```

fm.c.c - Notepad
File Edit Search Help
int
main (int argc, char **argv)
{
    int state = 0 ;
    int event = 0 ;
    int break_loop = 0 ;

    state = STATEdo_pause ; /* the initial state */

    do {
        do_action (state) ;
        switch (state) {

            case STATEdo_pause :
                get_event(&event) ;
                switch (event) {
                    case EVENTstart : state = STATERunning ; break ;
                    default : tx_error(state) ; break ;
                }
                break ;

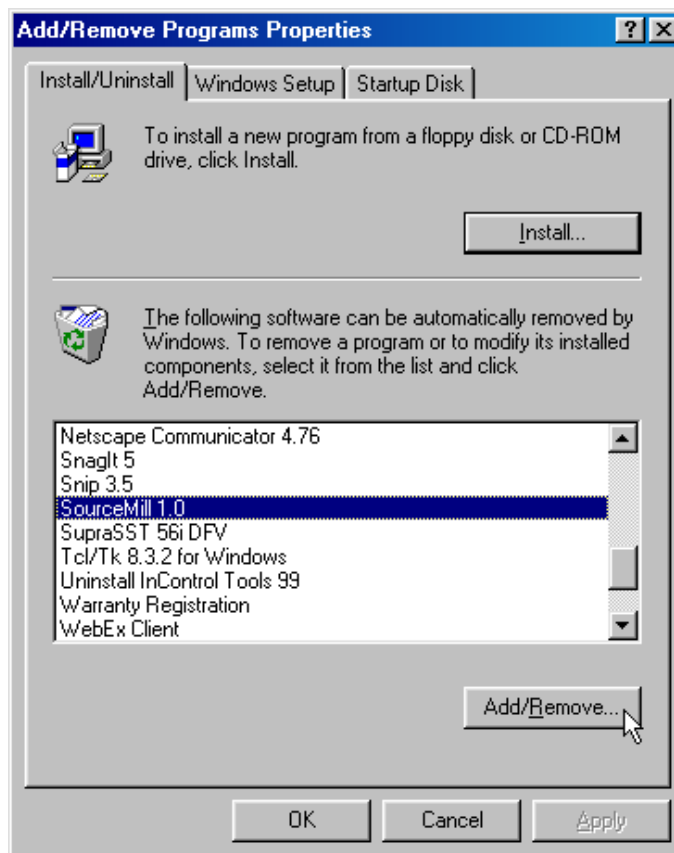
            case STATEdo_run :
                get_event(&event) ;
                switch (event) {
                    case EVENTpause : state = STATEReady ; break ;
                    case EVENTstop : state = STATEhalted ; break ;
                    default : tx_error(state) ; break ;
                }
                break ;

            case STATEdo_stop :

```

Uninstalling

To uninstall Cleanscape SourceMill use Add/Remove Programs in the Windows Control Panel.



3 Developing application frameworks

Summary

This chapter provides an overview of Cleanscape SourceMill theory and architecture, and shows how the application uses object models and templates to automatically generate code, and shows how user-defined rules can be gathered and managed by using into Projects. The chapter includes the following sections:

- Application overview
- Data Structure Model (DSM)
- Templates (TEM)
- Projects

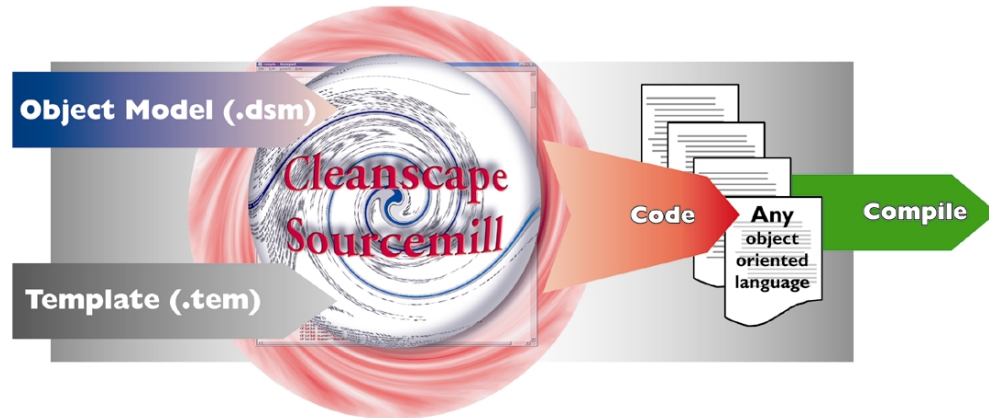
For details about how to modify and create templates and object models see [Object Model Reference](#) and [Template Reference](#).

Application overview

Cleanscape SourceMill uses three basic components to generate code: an object model (also called a data structure model or DSM file) a template (also called a TEM file) and the code synthesis engine.

- ✓ The DSM file tells Cleanscape SourceMill about the objects you want to create.
- ✓ The TEM file tells Cleanscape SourceMill how to generate code for each of your objects.
- ✓ The code synthesis engine takes in an object model uses the formula defined in a DSM file to expands a template into commercial-grade code, as shown in *Cleanscape SourceMill code synthesis process*:

Cleanscape SourceMill code synthesis process



Data Structure Model (DSM)

The object model is also called a Data Structure Model (DSM) in Cleanscape SourceMill. The DSM file is a high-level structure that describes part of an application. It defines data objects, state transitions, process interactions, etc.

The DSM tells Cleanscape SourceMill about the objects you want to create. You build DSM files using the Cleanscape SourceMill DSM File Editor to add objects and to define their attributes, functions and characteristics.

You can customize the provided DSM files for your specific needs, develop your own from scratch, or contract with Cleanscape to build custom object models for you.

Creating an object model

You create an object model using SourceMill DSM File Editor. SourceMill DSM File Editor is started when you create or open an object model. The editor allows you to add objects and define their attributes, functions and characteristics.

If you are new to Cleanscape SourceMill object model syntax, you can use the syntax guides available from the Add menu on the DSM File Editor menu bar.

- ✓ The menu bar will not appear unless an object model is open.
- ✓ The guides insert the correct text formats of DSM elements for you to modify. Within the text added, places are identified where names or values of your own should be placed. For example:

```
object <obj_name>
```

- ✓ When you double-click on the item within angle brackets, it is selected. The item is replaced when you type the new value.
- ✓ A syntax guide can be added for objects, object instances, and each of the other model elements.
- ✓ Object model statements, **keywords**, **characteristics**, and **rules** are explained in detail in **Object Model Reference**.

Examples

A Single Object

```
object File
  Name : string ;
      Handle : fp ;
end ;
fp = "FILE *"
```

- ✓ You can use any number of unique objects described in an object model.
- ✓ Specify objects (classes) and their attributes in an outline form. The model will allow you to use the terms/keywords and class interchangeably.
- ✓ Each object can have zero or more attributes. Each attribute has a name and a type. The name is on the left of the colon; the type is on the right. Both must be identifiers.
- ✓ The type of an attribute can be one of the following:
 - ✓ An identifier that names a built-in type that is foreign to the model (such as string in the example)
 - ✓ A name that is associated with a string that defines the type (such as fp in the example), or
 - ✓ The name of another object (this form is used when making links between objects)

Objects Using Inheritance

```
object CompanyAsset
  CompanyName : string ;
end ;
object NamedObj
  Name : string ;
end ;
object TextFile parent NamedObj
  Handle      : fp ;

  Open (Name : string) ;
end ;
object Person parent CompanyAsset, NamedObj
  Age : int ;
end ;
fp = "FILE *"
```

- ✓ TextFile is a child object, inheriting from NamedObj.
- ✓ The object Person inherits (multiply) from both CompanyAsset and NamedObj.

Links Between Objects

```
object File
  Name      :      string ;
  Handle    :      fp ;
  Type      :      fileTypeT ;
  Functions : list_of CFunction ;
end ;
object CFunction
  Name      :      string ;
  Returns   :      string ;
```

```

        FromFile : ref_to File ;
        Parameters : list_of CFuncParam ;
    end ;
object CFuncParam
    Name : string ;
    Type : string ;
end ;
fp = "FILE *"
string = "char *"

```

- ✓ Attributes can be *referential attributes*, in that they specify handles to other objects. The example shows several attributes of this type. The object File has out-bound references to a list of CFunction objects, and the CFunction object has a link to a File object.
- ✓ Several of the templates provided with Cleanscape SourceMill generate code to manage references between objects.
- ✓ The mapping tags `ref_to` and `list_of` are only examples of such tags. All words that end in `_of` and `_to` are taken as mapping tags, and are passed through to template expansion as defining the type of the attributes that use them.
- ✓ You can use your own mapping tags in templates you add or change without the need to change or reconfigure anything. Use them in the same way the provided templates use the mapping tags in the examples above.

Object Functions

```

object TextFile
Name : string = "\"-not-assigned-\"";
    Handle : fp ;
    Type : fileTypeT ;
    Open ( DirName : string,
           FileName : string [IN])
           : boolean ;
    GetLine() : string ;
    Close ()
    {}

```

```

        fclose (Handle);

    };

end ;

fp = "FILE *"
string = "char *"

```

- ✓ Objects can have functions that do, or do not, return values.
- ✓ Function parameters use the same syntax as object attributes. Parentheses must always be provided, and they may be empty.
- ✓ The parameters DirName and FileName of the function Open have been given the characteristic IN, which is only meaningful if the template looks for the characteristic IN, and emits code that is predicated upon that characteristic. Otherwise it is ignored.
- ✓ The function Close has its code body specified in-line in the model. The double curly braces denote the start and end of the multi-line text literal that makes up the function's body text.
- ✓ The Name attribute of the object TextFile has been given an initial value of -not-assigned-, which also demonstrates the required escaping of quotes that are part of text.

Object Behavior

```

events: start, pause, resume, finish ;

object Sprinkler
state Initial

    start => Sprinkling ;

end [is_initial];

state Sprinkling

    action: OpenValue ;

    pause => Paused ;

    finish => Finished ;

end ;

state Paused

    action: CloseValve ;

    resume => Sprinkling ;

```



```

        finish => Finished ;

    end ;

    state Finished

        action: CloseValve ;

        start => Initial ;

    end ;

end ;

```

- ✓ Objects may have any number of states.
- ✓ Each state can specify an action, and a set of transitions to other states: in response to events received while in that state.
- ✓ The modeling technique provides for actions to be carried out upon entry to a state.
- ✓ The events are globally defined, and transitions can only be triggered by events specified in the model.

Global Functions

```

make_new_str (str1 : string, str2 : string) : string
{
char *str3 ;

    str3 = (char *) malloc(strlen(str1) + strlen(str2) + 1);
    str3[0] = '\0' ;
    strcat (str3, str1);
    strcat (str3, str2);
    return str3 ;
};

string = "char *"

```

- ✓ Functions that are not nested with an object are global functions.
- ✓ Function parameters use the same syntax as object attributes. Parentheses must always be provided, and they may be empty.
- ✓ String substitutions such as string, in this example, may not be used within a text literal (e.g. we could not have used string in place of char * in the body of the function shown).

- ✓ The body of the function shown above assumes that the template preserves the names of the parameters.
- ✓ In-line code always needs to reference things in the final code, so it is a good idea to add it last, after the code around it is available for reference.

Pragmas and Options

```
pragma options (use_str, gen_bodies)
pragma h_include (stdio, stdlib, "mylib/stdhdr.h")
object TextFile
...
end ;
...
```

- ✓ Pragmas allow lists of values to be associated with a name, and for those values to be retrieved during the expansion of a template.
- ✓ Options is a special pragma. A global Boolean variable (one for each of the values named in the options pragma) is set to true and passed to the template expander.

Templates (TEM)

A template file (also referred to as TEM files) contains directives that tell Cleanscape SourceMill how to create and name files, and what to put into those files. A key element of the application's flexibility, the TEM file allows you to define what objects are in a given language or dialect. This allows you to generate code in every language and dialect for which you have a template.

When you select a template for your target environment, the code synthesis engine follows directives in the template that tell it how to navigate through and select the elements of your object model.

- ✓ You can use or modify one of the provided templates or you can create their own according to your local coding standards. Cleanscape also provides custom template building services.
- ✓ You can add templates of your own. The Cleanscape SourceMill template syntax is described in this guide in the section detailing the process of creating your own templates.
- ✓ If you need help in creating your own templates, contact Cleanscape to discuss the services we can provide.
- ✓ Descriptions of sample templates are in the Appendix.

Selecting a Template

To select a template:

1. Go to the menu bar and select the following option:

Options\Select Template By Title...

- ✓ The menu bar with the Options menu will not appear unless an object model is open.
 - ✓ A dialog box will be presented that allows you to select a template.
2. Browse through the available templates by scrolling.
 3. Select a template.
 4. Press OK
 - ✓ The template in the title box will be set as the active template.

Specifying an output directory

To specify an output directory:

1. Select Options\Codegen Options...
 - ✓ Use the output directory edit box to specify where generated files are to be placed.
2. Type in the full pathname of the output directory, or launch the standard Windows Directory Selection dialog box, by pressing **Browse To Set**.
 - ✓ The values you select will be saved and loaded as the active settings the next time you run Cleanscape SourceMill.

Creating a template

To create a new template:

1. Select File\New
2. In the New dialogue box highlight SNIP Template
3. Click OK to view the generic TEM file titled NewTem1
4. Modify NewTem1 for your needs using the variables and statements described in [Reference](#)
5. Select File\Save as...
6. Enter the name of your template in the File name: field
7. Click Save

Naming and storing template files

- Template files are named with a .TEM file extension.
- Template files can be stored in any directory, including over a network or Internet connection.

Adding Help or Doc files

- Template files are associated with help or document files that are named with a .HLP or .DOC file extension respectively. Cleanscape SourceMill first tries the .HLP file; if that doesn't exist it looks for the .DOC file. If neither exists Cleanscape SourceMill sends a message.
- Template help or document files must be stored in the same directory as the templates they describe.
- Help files are opened with the normal Microsoft Windows help engine WINHELP. Doc files are opened as text files, using the editor named in the SNIP.INI file.

Allowing your template to be selected by title

The file TEMPLATE.TBL contains the information used to allow Cleanscape SourceMill to present a user with template titles and descriptions for selection. You can edit this file to add your own template filename, template title, and template description.

Template creation tips

- ✓ Use templates for model dependent aspects — Templates specify how code modules and code fragments change with the elements of your object model.
- ✓ We advise packaging code that is not dependent on your object model elements in separate files or libraries. This will minimize the size of your templates.
- ✓ Create variety rather than complexity — The inner workings of a template allow it to emit different things, dependent on any of a number of very fine-grained characteristics of your objects.
- ✓ Keep each template focused on a single architecture with a few consistent principles applied throughout. It is often more workable (less complex) to create a separate template than it is to highly parameterize what is emitted from a single template.

Template File examples

File Directives

```
.file stdhdr.h
...
.file <dsm.name>.h
...
```

- ✓ Two examples of file directives: the first uses a constant name; the second uses a computed name (which in this case takes on the name part of the DSM file used as input).
- ✓ The file directive computes the text to the right of the .file keyword, and then opens that file.

- ✓ The text that is computed for a filename must be legal. If it is not, Cleanscape SourceMill opens a default bailout file named **NONAME.XXX**, and spills the remaining text into it.
- ✓ The file is opened in a manner that creates the file if it does not exist, and writes it from the beginning. Any existing file with that name is overwritten.

File Statements

Statements (to the right of conditionals) can also be used to control how files are treated. This allows file handling to be based on characteristics of the model or of individual model elements. For example, if you want to add a capability to your template that allows it to put common code in a separate file, under the control of a DSM pragma, the **\$\$NEWFILE** statement can be issued to the right of a test for a particular option.

Within your DSM file, specify:

```
pragma files [factor_common] ;
    -- Sets <files.factor_common> to 'true'
...
```

Within your TEM file, specify:

```
.file <dsm.name>.cpp
:
: ...
:
<files.factor_common>      :$$NEWFILE common.cpp
:
: void PtrList::AddItem(PtrList *p, Item *I);
: {
:   ...
: }
:
...
```

- ✓ The text following the simple statement containing the **\$\$NEWFILE** action will be placed in the file **common.cpp** when **<files.factor_common>** is set to true. Otherwise, the lines will be emitted into the same file that was open before the statement was passed (**<dsm.name>.cpp** in this example).

Iterator Example

```
.each_obj
:      OBJECT: <obj.name>
:
:      Has Outbound Referece Attributes
:      -----
.each_attr <attr.ref_to>
:      <attr.name> : <attr.kind>
.end
.end
```

- ✓ Iterators are defined for all model elements. The scope of an iterator starts at the .each_XXX directive and continues through its corresponding .end directive
- ✓ Within the scope of an iterator, variables are set to provide information, in turn, about each element that is selected by the iterator.
- ✓ The .each_attr iterator above will only select those attributes that are of type ref_to.
- ✓ The .each_obj iterator above will select all objects in the model.
- ✓ For a list of the variables that are active inside each iterator, see [Object Model Reference](#).

Lines that Emit Code

```
.each_obj

#include \<<obj.name>.h\>

      :object <obj.name>
      :
      :      Has Outbound Referece Attributes
      :      -----

.each_attr
<attr.ref_to> :      <attr.name> : <attr.kind>
.end
.end
```

- ✓ The statements in the example above that are distinguished by having colons that are aligned with each other are called *simple statements*.
- ✓ A simple statement has two parts: An expression, and a text part.
- ✓ The expression (if present) is specified to the left of the separating colon. Only the last simple statement above has an expression.
- ✓ The expression is evaluated, and if true, the right hand side is expanded (variables replaced with real values) and emitted into the currently open output file.
- ✓ The first simple statement above demonstrates how to escape angle-brackets that you want to keep as part of the text. If they are not escaped, angle brackets are treated as variable delimiters.
- ✓ For a list of the variables that are active inside each iterator, see [Reference](#).

Projects

Cleanscape SourceMill Projects allow you to collect a number of generation rules under a single project heading. You can also use a Project for very involved sets of generation rules to organize a hierarchy of generation rules into any number of subsystems.

Creating a Project

To create a new Cleanscape SourceMill Project:

1. Select File->New...
2. Select Project from the list of object types presented. A tree control is presented that allows you to define the structure of your project.

Popup Menus

When you press and release the right mouse button over an item in the tree, a context sensitive popup menu will be presented. This menu allows you to invoke operations that can be performed on the selected item. Popup menus allow you to add levels to the structure, edit properties, delete items and invoke code generation from that level in the hierarchy on down.

Cut, Copy and Paste

The fastest way to copy and move items among sever projects is to transfer them through the clipboard using Cleanscape SourceMill's Cut, Copy, Paste commands.

Drag and Drop

To change the organization of the contents of your projects drag tree items within or between projects. When moving or copying objects between windows and applications, Cleanscape SourceMill uses OLE drag and drop. The default drag mode moves the selected objects.

Generation Rules

Generation rules are the lowest level elements of a project. They can be organized under a project, or under a subsystem. A single generation rule is a pairing of an object model (a DSM file) and a Template file to apply to the model (providing a mapping of the data model objects into generated implementation artifacts).

When you select a generation rule's properties from its popup menu, property sheet pages are presented to allow you to choose an object model and a template from those that can be found on your file system.

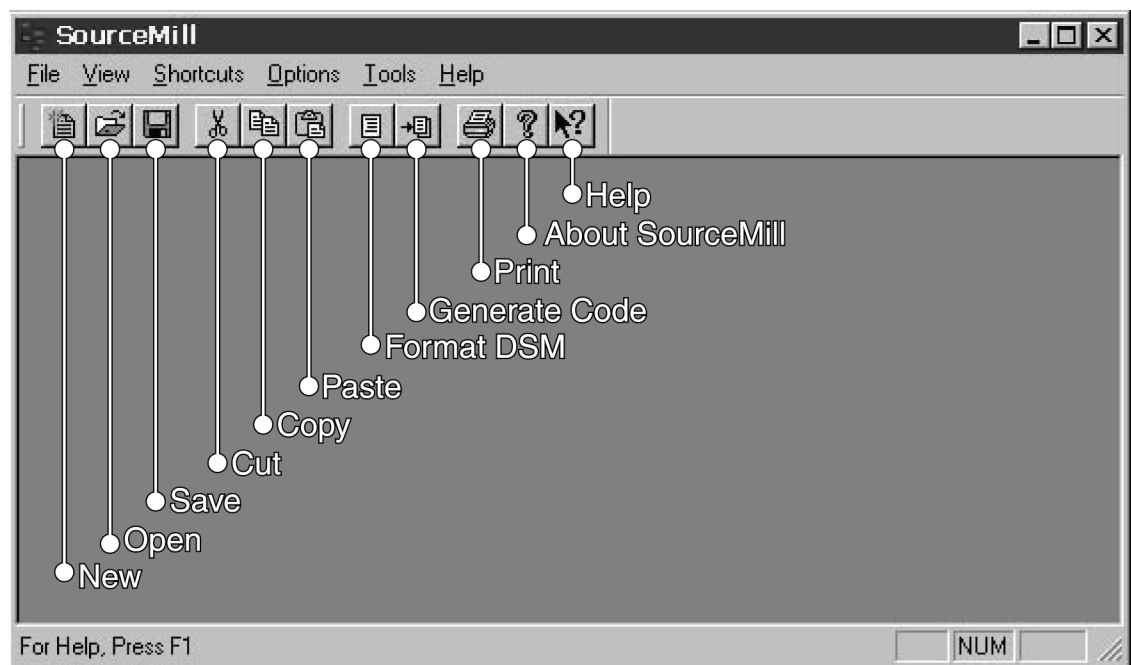
4 User interface

Summary

Cleanscape SourceMill has an easy to use graphical user interface that you will find intuitive to operate if you are familiar with Windows applications. This chapter contains information about the functions of the application's menu and tool bar commands, and includes the following sections:

- Toolbar
- Menus
- Other support available






Toolbar






- ✓ Additional details on the functionality of toolbar buttons can be found in “Menus”

Menus

File Menu

| Menu | Command | Function | Toolbar | Keys |
|------|---------------|---|---|--------|
| File | New | Creates a new document in Cleanscape SourceMill. Each Data Structure Metafile (DSM) describes a distinct object network. |  | CTRL+N |
| | Open | Opens an existing document in a new window. You can open multiple documents at once. Use the Window menu to switch among the multiple open documents. See Window 1, 2, ... command. |  | CTRL+O |
| | Close | Close all windows containing the active document. Cleanscape SourceMill suggests that you save changes to your document before you close it. If you close a document without saving, you lose all changes made since the last time you saved it. Before closing an untitled document, Cleanscape SourceMill displays the Save As dialog box and suggests that you name and save the document. You can also close a document by using the Close icon on the document's window, as shown below: |  | |
| | Save | Saves the active document to its current name and directory. When you save a document for the first time, Cleanscape SourceMill displays the Save As dialog box so you can name your document. If you want to change the name and directory of an existing document before you save it, choose the Save As command. |  | CTRL+S |
| | Save As | Save and name the active document. Cleanscape SourceMill displays the Save As dialog box so you can name your document. To save a document with its existing name and directory, use the Save command. | — | |
| | Print | Presents a Print dialog box, where you may specify the range of pages to be printed, the number of copies, the destination printer, and other printer setup options. |  | CTRL+P |
| | Print Preview | Displays the active document, as it would appear when printed. When you choose this command, the main window will be replaced with a print preview window in which one or two pages will be displayed in their printed format. The print preview toolbar offers you options to view either one or two pages at a time; move back and forth through the document; zoom in and out of pages; and initiate a print job. | — | — |
| | Print Setup | Presents a Print Setup dialog box, where you specify the printer and its connection. | — | — |
| | Exit | Ends your Cleanscape SourceMill session. You can also use the Close command on the application Control menu. Cleanscape SourceMill prompts you to save documents with unsaved changes. | — | ALT+F4 |

Edit Menu

| Menu | Command | Function | Toolbar | Keys |
|------|------------------------|---|---|--------------------------------|
| Edit | Undo/ Can't Undo | Reverses the last editing action, if possible. The name of the command changes, depending on what the last action was. The Undo command changes to Can't Undo on the menu if you cannot reverse your last action. | — | CTRL+Z or ALT- BACKSPACE |
| | Cut | Deletes data from the document and moves it to the clipboard. This command is unavailable if there is no data currently selected. |  | CTRL+X |
| | Copy | Copies data from the document to the clipboard. This command is unavailable if there is no data currently selected. |  | CTRL+C |
| | Paste | Pastes data from the clipboard into the document. This command is unavailable if the clipboard is empty. |  | CTRL+V |
| | Find | Displays the find text dialog box to begin a search for a string. A dialog prompts you for the text to find. | — | — |
| | Replace | Displays the find-replace dialog box to begin a search and replace operation. A dialog prompts you for the text to find and the text to substitute when the text is found. | — | — |

View menu

| Menu | Command | Function | Toolbar | Keys |
|------|------------|--|---------|------|
| View | Toolbar | Shows or hides the toolbar, which includes buttons for some of the most common commands in Cleanscape SourceMill, such as File Open. A check mark appears next to the menu item when the Toolbar is displayed. | — | — |
| | Status Bar | Shows or hides the status bar, which describes the action to be executed by the selected menu item or depressed toolbar button, and keyboard latch state. A check mark appears next to the menu item when the Status Bar is displayed. | — | — |

Add menu

The add menu provides commands that enable you to insert guides that show examples of the syntax of the various DSM statement types, as follows:

| Menu | Command | Function | Toolbar | Keys |
|------|--------------------|---|---------|------|
| Add | Object Syntax | Adds an object syntax guide to the END of the active object model window. You should replace the fields that are contained in angle brackets (e.g. <name>) with your own names and type-names. | — | — |
| | Instance Syntax | Adds an object instance syntax guide to the END of the active object model window. You should replace the fields that are contained in angle brackets (e.g. <name>) with your own names and type-names. | — | — |
| | Function Syntax | Adds a function syntax guide to the END of the active object model window. You should replace the fields that are contained in angle brackets (e.g. <name>) with your own name and string substitution value. | — | — |
| | Events Syntax | Adds an event list syntax guide to the END of the active object model window. You should replace the fields that are contained in angle brackets (e.g. <name>) with your own name and string substitution value. | — | — |
| | Type-string Syntax | Adds a type-string syntax guide to the END of the active object model window. You should replace the fields that are contained in angle brackets (e.g. <name>) with your own name and string substitution value. | — | — |
| | Pragma Syntax | Adds a syntax guide showing how to define a pragma. Adds a pragma syntax guide to the END of the active object model window. You should replace the fields that are contained in angle brackets (e.g. <name>) with your own name and set of values. The default template only recognizes the h_include and c_include pragmas. | — | — |

Run menu

The run menu offers commands that enable you to invoke operations that process your object model.

| Menu | Command | Function | Toolbar | Keys |
|------|----------------|---|---------|------|
| Run | Text Formatter | Formats the text in your active DSM window. This operation cannot be undone. | — | — |
| | Code Generator | Generates code, applying the default or chosen template. Compiles the text in your active DSM window and emits code according to the commands in the selected template. | — | — |

Options menu

The options menu offers commands that allow you to configure the Cleanscape SourceMill application and modify its behavior.

| Menu | Command | Function | Toolbar | Keys |
|---------|-----------------|--|---------|------|
| Options | Codegen Options | Displays a dialog box that allows you to control the behavior of Cleanscape SourceMill when it generates code into your file system. You may change one or all of: The template for code generation to expand, Where the generated files will be placed, and Whether or not to display the name of each generated file as it is created. | — | — |

Window menu

The Window menu offers commands that enable you to arrange multiple views of multiple documents in the application window:

| Menu | Command | Function | Toolbar | Keys |
|--------|----------------------|--|---------|------|
| Window | Cascade | Arranges windows in an overlapped fashion. Use this command to arrange multiple opened windows in an overlapped fashion. | — | — |
| | Tile | Arranges windows in non-overlapped tiles. | — | — |
| | Window Arrange Icons | Use this command to arrange the icons for minimized windows at the bottom of the main window. If there is an open document window at the bottom of the main window, then some or all of the icons may not be visible because they will be underneath this document window. | — | — |
| | 1, 2, ... | Goes to specified window. Cleanscape SourceMill displays a list of currently open document windows at the bottom of the Window menu. A check mark appears in front of the document name of the active window. Choose a document from this list to make its window active. | — | — |

Help menu

The Help menu offers commands that provide you assistance with this application:

| Menu | Command | Function | Toolbar | Keys |
|------|------------|--|---------|------|
| Help | Index | Offers an index to topics on which you can get help. From the opening screen, you can jump to step-by-step instructions for using Cleanscape SourceMill and various types of reference information. Once you open Help, you can click the Contents button whenever you want to return to the opening screen. | — | — |
| | Using Help | Provides general instructions on using help. | — | — |
| | About | Displays the copyright notice and version number of your copy of Cleanscape SourceMill. | — | — |

Other Support Available

Contacting Cleanscape Software International

Web: <http://www.cleanscape.net>

SourceMill Resource Center: <http://www.cleanscape.net/solutions/sourcemill>

E-mail: <mailto:support@cleanscape.net>

Phone: 650

OO Design Services

Cleanscape Software International will assist you in creating a first class OO design for your application. Hourly and fixed fees are available.

OO Training

Our training for object-oriented software development will help you to properly prepare your staff to tackle the design and implementation issues that, left unchecked, might otherwise place your project's goals at risk.

Modules

- Designing With Inheritance: Power and Pitfalls (1 day)
- Leveraging Reuse: How to find and Capture Relevant Objects (1 day).
- Strategies with Architectures: Reuse Beyond Components (1 day). (Training module that directly addresses using Cleanscape SourceMill.)

Template Construction

Cleanscape Software will assist you in creating templates for particular languages and/or architectural targets. Hourly and fixed fees are available.

5 Object Model Reference

Summary

This chapter provides instructions about how to create and modify Cleanscape SourceMill Object Models and includes the following sections:

- Object Model Statements
- Object Model Keywords
- Object Model Characteristics
- Object Model Rules

Object Model Statements

You can create object models from the following statement types:

- Event list
- Object definition
- Function definition
- State definition
- Processes Definition
- Instance definition
- Literal Text
- Foreign Type Statement
- Pragma
- Import
- Comment

This remainder of this section provides details about each type of statement, including purpose, syntax, grammar, examples, and tips for each statement.

DSM Event List

Purpose

Define events that can be referenced in the model.

Syntax

```
events : <name_list> ;
```

Grammar

| | |
|-------------|---------------------------------------|
| <name_list> | A comma separated list of identifiers |
|-------------|---------------------------------------|

Example

```
events: start, stop, pause, resume ;
```

DSM Object Definition

Purpose

Allows you to describe as many of the attributes, links, functions, and characteristics of an object as are required to allow the active template to generate its code.

Syntax

```
'object' <name> ['parent' <parent_list>]
    [<obj_element> ...] ';'
'end' [ <characteristics> ] ';'
```

Grammar

| | |
|-------------------|--|
| <obj_element> | [<attr_def> ...] [<function_def> ...] [<state_def> ...] [<process_def> ...] [<pragma_def> ...] |
| <attr_def> | <attr_name> : [<mapping>] <attr_kind> [<characteristics>] |
| <mapping> | an identifier ending in '_to' or '_of' |
| <parent_list> | <parent_info> [, <parent_info> ...] |
| <parent_info> | <parent_name> [<characteristics>] |
| <function_def> | See “DSM Function Definition Statement” |
| <state_def> | See “DSM State Definition Statement” |
| <process_def> | See “DSM Process Definition Statement” |
| <literal_def> | See “DSM Literal Text Statement” |
| <pragma_def> | See “DSM Pragma Statement” |
| <characteristics> | See “Specifying Characteristics” |
| <*_name> | <identifier> |
| <identifier> | Words made up of a-z, A-Z, 0-9, and '_' |

Example

```
events : do_open, do_close ;
```



```

object TextFile
    PathName      :
string ;
    FileType      :
fileTypeT ;
    Words          : list_of      Word ;
    Open () [is_private] ;
    Close () [is_private] ;
    "operator=" () : TextFile_ref ; -- c++ usage
    state FileOpen
        action : open_file ;
        do_close => FileClosed ;
    end ;
    state FileClosed
        action : close_file ;
        do_open => FileOpen ;
    end ;
    state Start
        do_open => FileOpen ;
    end [initial] ;
end
TextFile_ref = "TextFile &"

```

DSM Function Definition statement

Purpose

Define a freestanding or member function.

Syntax

```

<ftn_name> '(' [<parameters>] ')' [: <ftn_return_kind>]
[<characteristics>] ['{' <function_body> '}'] ';'

```

Grammar

| | |
|-------------------|---|
| <ftn_name> | <identifier> <string> |
| <parameters> | <parameter> [, ...] |
| <parameter> | <identifier> : <identifier> [= "init val string"] |
| <ftn_return_kind> | <identifier> |
| <characteristics> | See "Specifying Characteristics" |

Example

```
NewStrCat2 (str1 : string, str2 : string) : string [is_util]
{
char *str3 ;
str3 = (char *) malloc(strlen(str1) + strlen(str2) + 1);
str3[0] = '\0' ;
strcat (str3, str1);
strcat (str3, str2);
return str3 ;
}
string = "char *"
```

- ✓ The type name `string` is not known to the function body, and so it cannot be referenced as such. `char *` should be used in this case instead. Cleanscape SourceMill will replace the word `string` with `char *` in the model before generating code.
- ✓ Characteristics (such as `is_util`) can be used to select functions from among the set of functions defined and handle it within the template language (such as placing all such functions in a particular file).
- ✓ **Preserving hand modifications to code files with uid.** Adding the special characteristic `uid` to function bodies allows them to be found independently of their function names, and for the code in those function bodies to be saved for re-inclusion, before overwriting a file that contains them. Most stock templates allow for tagging function bodies with `uid`'s. A `uid` is a unique string of non-blank characters, and can be of arbitrary length (see the example below). You must ensure that the `uid`'s you place on function definitions are unique, and will never be changed. To do otherwise will result in loss of hand edits to a function's code body when the file is over-written during a subsequent code generation run.
- ✓ `functionName (p1 : type1, p2 : type2) [uid=ftn001]`. Use this feature if you feel you absolutely must make hand modifications to function bodies within

generated code files. Alternatively, we recommend that you consider placing hand-edited function bodies in separate files, as there is no burden placed upon you to manage uid's, and there is no potential for the code loss that may accompany errors in handling uid's .

DSM State Definition statement

Purpose

Define a freestanding state as part of a state model.

Syntax

```
'state' <state_name>
[<state_element> ';' ...]
'end' [ <characteristics> ] ';'

```

Grammar

| | |
|-------------------|--|
| <state_element> | <action> <transition> |
| <action> | 'action' ':' <action_name> [<characteristics>] ['{' <lines of text> '}'] |
| <transition> | <event_name> '=>' <state_name> [<characteristics>] |
| <characteristics> | See "Specifying Characteristics" |

Example

```
state Switched_off
action : SetToOffPosition ;
Turn_on => Switched_on ;
end [is_initial] ;
state Switched_on
action : SetToOnPosition ;
Turn_off => Switched_off ;
end ;

```

- ✓ Preserving hand modifications to code files: Adding the special characteristic uid to state action bodies allows them to be found independently of their state and action names, and for the code inside the state actions to be saved for re-

inclusion, before overwriting a file that contains them. Stock templates do not yet provide examples of tagging state action bodies with uid's.

- ✓ A uid is a unique string of non-blank characters, and can be of arbitrary length (see the example below). You must ensure that the uid's you place on state actions are unique, and will never be changed. To do otherwise will result in loss of hand edits to a state action's body when the file is over-written during a subsequent code generation run.

```
state Switched_off
action : SetToOffPosition [uid=action001] ;
Turn_on => Switched_on ;
end [is_initial] ;
```

- ✓ You should use this feature if you feel you absolutely must make hand modifications to state action bodies within generated code files. Alternatively, we recommend that you consider placing hand-edited state action bodies in separate files, as there is no burden placed upon you to manage uid's, and there is no potential for the code loss that may accompany errors in handling uid's.

DSM Process Definition statement

Purpose

Define a freestanding state as part of a state model.

Syntax

```
'process' <process_name>
    [<flow_def> ...]
'end' [ <characteristics> ] ;'
```

Grammar

| | |
|-------------------|---|
| <flow_def> | 'flow' <flow_name> 'from' 'to' <process_name> [<characteristics>] ;' |
| <process_name> | <identifier> |
| <characteristics> | See "Specifying Characteristics" |

Example

```
process Parse_cmd
flow CmdLine from Input_mgr ;
flow CmdDesc to Cmd_dispatcher ;
```

```

end ;

process Cmd_dispatcher

flow CmdDesc from Parse_cmd ;

...

end ;

...
```

DSM Instance Definition statement

Purpose

Define named instances of objects.

Syntax:

```

<inst_name> ':' <obj_name> [<attr_assigns>]
<characteristics>] ';'

```

Grammar

| | |
|-------------------|---|
| <attr_assigns> | '(' <attr_name> '=' <attr_val>, ... ')' |
| <characteristics> | View Syntax |

Example

```

my_file : File ;

animal : Beast (Name = "fox") [small, four_legged] ;
```

DSM Literal Text statement

Purpose

Capture multi-line text that can be inserted at chosen points in a template.

Syntax

```

'literal' <literal_name> [<characteristics>]
'{' <lines of text> '}';

```

Grammar

| | |
|-------------------|----------------------------------|
| <literal_name> | <identifier> |
| <characteristics> | See “Specifying Characteristics” |

Example

```
literal h_file_hdr_comment [is_comment] {{
    // Purpose: This module has the purpose of
    //         providing an encapsulation of ...
}}
```

DSM Foreign Type statement**Purpose**

Define strings that can be used in place of type names within the model.

Syntax

There are two forms to define foreign types. Both forms are described below:

Form 1:

The first form simply allows a substitution to be supplied for the type name, which provides the "type calculus" in the target language. The substitution is applied everywhere the type name is used in the model.

```
<name> '=' <string_literal>
```

Form 2:

The second form allows characteristics to be supplied that define the declaration form and primitive operation functions for the type:

```
'foreign' <name> [<characteristics>] ';' ;
```

Grammar

| | |
|-------------------|---|
| <characteristics> | identify the nature of the type and supply additional information to be used in the creation of code. See the examples below. |
| <characteristics> | See "Specifying Characteristics" |

Examples*Form 1:*

```
string = "char *"
```

Form 2:

```
foreign String [is_builtin, use_ref] ;
```

- ✓ This example says that a string can be treated as a built-in type in the language. (This capability is supplied by the class string itself. You tell Cleanscape

SourceMill it can treat the type as if all of its operations are part of the language). Further, you can tell Cleanscape SourceMill to use refs ('&') when defining formal parameters to functions.

```
foreign String [is_usertype, type_decl = "char*",
    init_ftn = new_str, copy_ftn = copy_str,
    destroy_ftn = safe_free] ;
```

- ✓ This example says `string` is really a `char *` and that its primitive operations for `init`, `copy`, and `destroy` are implemented by the functions named. (See the template's use of these variables to control the kind of code that is emitted).

DSM Pragma statement

Purpose

Define options and values for use during template expansion.

Syntax

```
pragma <name> [<pragma_values>] [<characteristics>] ';' ;
```

Grammar

| | |
|-------------------|-----------------------------------|
| <pragma_values> | '(' <value_list> ')' |
| <value_list> | <value> [, <value> ...] |
| <value> | Identifier or literal text string |
| <characteristics> | See "Specifying Characteristics" |

Example

```
pragma c_include (stdio, "mylib/hdrfile.h", stdlib)
pragma options (gen_bodies)
```

DSM Import statement

Purpose

Import object model definitions from another DSM file.

Syntax

```
import "<dsm_file_path>" ';' ;
```

Grammar

| | |
|-----------------|--------------------|
| <dsm_file_path> | host file pathname |
|-----------------|--------------------|

Example

```
import "common.dsm" ;
```

DSM Comment

Purpose

Allow commentary text within the model.

Syntax

```
--<rest of line taken as part of comment>
```

Example

The object to follow is used for ...

```
...
```

Object Model Keywords

The following keywords may not be used as a name or type-name within an object model:

| | | | |
|---------|---------|---------|-------|
| action | events | object | state |
| class | flow | parent | to |
| end | from | pragma | |
| foreign | literal | process | |

C++ Language Keywords

| | | | |
|----------|---------|-----------|----------|
| asm | auto | break | case |
| catch | class | const | continue |
| default | delete | do | else |
| enum | extern | for | friend |
| goto | if | inline | new |
| operator | private | protected | public |
| register | return | signed | sizeof |
| static | struct | switch | template |
| this | throw | try | typedef |
| union | virtual | volatile | while |

Ada Keywords

| | | | |
|--------|-------|------|----------|
| abort | do | mod | renames |
| abs | else | new | return |
| accept | elsif | not | reverse |
| access | end | null | select |
| all | entry | of | separate |

| | | | |
|----------|-----------|-----------|-----------|
| and | exception | or | subtype |
| array | exit | others | task |
| at | for | out | terminate |
| begin | function | package | then |
| body | generic | pragma | type |
| case | goto | private | use |
| constant | if | procedure | when |
| declare | in | raise | while |
| delay | is | range | with |
| delta | limited | record | xor |
| digits | loop | rem | |

Object Model Characteristics

Characteristics allow you to communicate implementation choices about your model objects to a template. Each template has its own set of supported characteristics, which represent the implementation choices available for each kind of model object that the template uses.

To see the characteristics of a template, open the Options\View Characteristics... dialog after selecting a template.

Example

```
object Student
    Name : String ;
    Grades : list_of Grade [is_owner];
end ;

object Grade
    Test : String ;
    Score : int ;
end ;
```

- ✓ The example shows a characteristic IS_OWNER applied to an attribute. Select the template providing a Class per Object from Options\Select Template by Title... and use Options\View Characteristics... to browse through and verify that this characteristic is a characteristic recognized by this template for Attributes.
- ✓ In this example, the characteristic IS_OWNER tells the template to make the emitted code delete the elements of the Grades list when a Student object is deleted. We can verify this by examining the template, to see where and how <attr.is_owner> is used, or by generating and examining the code files for this example, both with and without this characteristic applied.

Object Model Rules

General Rules

- The order of declarations is unrestricted, and they do not need to be grouped.
- The input is stream oriented, so lines can be wrapped at any point, with the exception of comments, which terminate at the end of the line on which they start.

Naming Rules

- There are several semantic rules that you must follow. They are checked by Cleanscape SourceMill before it goes on to generate output files.
- Names of objects, events, object instances, and string substitutions cannot overlap, and they must be unique.
- Names of attributes and states within an object cannot overlap, and they must be unique.
- An attribute cannot have the same name as its object.
- A state cannot have the same name as its object.
- Only one instance of any given pragma can be provided.

6 Template Reference

Summary

This chapter contains instructions about how to create and modify Cleanscape SourceMill templates, and includes the following sections:

- Template File Global Variables
- Template File Single Line Statements
- Template File Block Statements
- Template File Iterator Statements

Template File Global Variables

| Variable | Description |
|---|--|
| <snip.version> | The version ID of the running version of SNIP |
| <dsm.name> | The simple name of the data model, extracted from its filename. |
| <dsm.path> | The full pathname of the Object Model (.dsm file). |
| <template.path> | The full pathname of the Template (.tem file). |
| <dsm.has_objects> | This variable is "True" if objects have been defined in the model. |
| <dsm.has_behavior> | This variable is "True" if objects containing states have been defined in the model. |
| <dsm.has_states> | This variable is "True" if the model contains top-level state elements (defined outside of the scope of objects). |
| <dsm.has_processes> | This variable is "True" if the model contains top-level process elements (defined outside of the scope of objects). |
| <dsm.has_inheritance> | This variable is "True" if objects containing parents have been defined in the model. |
| <dsm.has_mult_inheritance> | This variable is "True" if objects having more than one parent(s) have been defined in the model. |
| <dsm.has_instances> | This variable is "True" if object instances have been defined in the model. |
| <dsm.has_events> | This variable is "True" if events have been defined in the model. |
| <dsm.has_functions> | This variable is "True" if global functions have been defined in the model. |
| <dsm.has_foreigns> | This variable is "True" if foreign types have been defined in the model. |
| <dsm.has_pragmas> | This variable is "True" if any pragmas have been specified in the object model. |
| <dsm.has_literals> | This variable is "True" if literal text blocks have been given in the object model. |
| <date.year>,<date.month>,<date.day>, <date.hour>,<date.minute>,<date.second> | The numeric values, as strings, of the date and time elements of the date and time the generation operation was invoked. |
| <literal.^name^> | Set to the text of each defined text literal |
| <^pragma_name^> | Set to "True" for each defined pragma - 'pragma options' also sets each of its values (options.^value^) to "true" |

Template File Single Line Statements

Single line statements use variables defined within the global context and iterator context and include the following:

- .file statements
- Simple statements with Runtime `.$$Directives`
- LoadTime `.$$IMPORT`

.file statement

Purpose

Opens an output file to capture the text that is emitted by the template statements that follow this statement.

Syntax

```
.file <filename-text>
```

Grammar

| | |
|-----------------|--|
| <filename-text> | Any text, including variables that will be expanded, to be used as the raw filename of the file to be opened for output. An attempt is made to adjust the resulting filename to a proper size. |
|-----------------|--|

Defines Variables

None

Example

```
.file <dsm.name>.h
```

Simple statement

Purpose

Expands and emits the provided code (right of the colon) into the output file. An expression can be provided that controls when this line is selected for expansion (left of the colon).

Syntax

```
[<expression>] ':' <code-text> ['\']
```

Grammar

| | |
|--------------|--|
| <expression> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | an identifier of a defined variable |
| <code-text> | text with embedded <var>s |
| | escape angle brackets with '\' chars (e.g. \<stdio.h\>) <##> calls for column alignment (e.g. <40> will pad the output line to column 40, but has no effect if column 40 has already be passed). <NEWLINE> forces a line break to be inserted where this special variable occurs. ['\'] an escape char ('\') at end of line suppresses the emission of a new_line character at the end of the emitted code-text. |

Grammar

- ✓ Expressions should evaluate to boolean values
- ✓ All variables in an expression are evaluated and the result must be all true values to cause the right hand side to be emitted.

| | |
|------------------|---|
| <var=val> | This expression is 'true' if the variable var is equal to the string "val" |
| <!var=val> | 'True' if the variable var is not equal to val. |
| <attr.is_local> | 'True' if the attr is of local type (defined in the model). The exclamation mark inverts the sense of a test. See <!attr.is_local> for an example. |
| <!attr.is_local> | 'True' when the attr is not local (i.e. is_foreign). The char '@' simply tests for the existence of a variable, so <@obj.comment>. |
| <@obj.comment> | Only 'true' if a variable named obj.comment is currently defined. The exclamation mark can be used to test for not-defined cases as such: <!@obj.comment> |

\$\$Directives

Directives allow control over files, directories, variables, and other useful resources, and can be used to invoke reusable modules of template statements.

\$\$Directives are available for execution on the right hand side of simple statements (Note: directives are associated with simple statements so that their execution can be made conditional in the same way that text emission is conditional).

Purpose

- Allow certain actions (right of the colon) to be predicated upon an expression being true (left of the colon).
- The following directives may appear immediately to the right of the colon (as shown) in a simple statement. They cause the indicated action.

| Directive | Action | Explanation |
|-----------|-----------------------|---|
| **ERROR | :**ERROR <other text> | Causes the error string to be reported as a message, and also places the message into the output file. Code generation continues to |

| | | |
|----------------------|--|---|
| | | completion. |
| **WARNING | :**WARNING <other text> | Causes the error string to be reported as a message, and also places the message into the output file. Code generation continues to completion. |
| \$\$SYMDUMP | :\$\$SYMDUMP | Causes the currently defined symbols to be output into the file. This is a template development hook, which allows template developers to see what variables are available for their use |
| \$\$SETVAR | :\$\$SETVAR varname value (simple var) | <p>Sets the provided variable name to the indicated value. The variable can then be used within the same iterator scope containing this directive. \$\$SETGVAR performs the same function, except it sets the variable in the global scope.</p> <p>For dealing with variables that represent numbers to up-count or down-count, values of the form +DIGITS or -DIGITS will add or subtract from the value in a variable. For example, \$\$SETVAR abc +2 will add two to the current value of abc.</p> <p>:\$\$SETVAR varname[] +/- value (value set):\$\$SETVAR varname{key} value (associative array)</p> <p>The above 2nd and 3rd forms create global variables that can be iterated over using the .each_var_item statement. Associative array item references such as <varname{key}> are legal to use in text to be expanded.</p> |
| \$\$CLEARVAR | :\$\$CLEARVAR varname/varname[]/varname{} | Causes the named variable, set or associative array to be removed. |
| \$\$SETCTIC | :\$\$SETCTIC scope characteristic-name value | Sets the named characteristic in the object bound to the given object scope to the value specified. Object scope names are one of (dsm, obj, attr, event, state, tx, literal, proc, flow, ftn, param, foreign, inst) |
| \$\$EDITVAR | :\$\$EDITVAR <varname> <command> <args> | Edits the named variable, applying the command to change its value. Commands are: uppercase, lowercase, substr pos len, left n, right n. |
| \$\$SEARCHVAR | :\$\$SEARCHVAR <varname> <regex> | <p>Searches the named variable for the regular expression. Sets variables <search.found>, <search.pos>, <search.len>, <search.rest>.</p> <p>Regular expression metachars are as follows:</p> <ul style="list-style-type: none"> ^ (anchor - if at the start or end of an expression causes the expression to be anchored at that position in the text) \$ (any non-blank) # (any digit), @ (any alpha) ? (any single character) [...] (any one of ...) [^...] (any one not of ...) |

| | | |
|----------------|---|--|
| | | <p>% (0 or 1 of previous factor)</p> <p>* (0..n of previous factor)</p> <p>+ (1..n of previous factor),</p> <p> (OR connection (eg. A B C)), \x (negates special meaning of x),</p> <p>(<reg_exp>) (a parenthesized regular expression)</p> <p>A string passed to the search string that requires escape characters will need to be formed using "double-escaping." This is because SNIP template statements use the back-slash to escape a char, and so does the syntax of a search pattern. So, for example, if you wanted a search pattern to include a back-slash, the following would find for the last section of a string that does not include a back-slash (e.g. a filename at the end of a Windows file path). This pattern should be read: "one or more non-back-slash chars."</p> <p>:\$SEARCHVAR filepath</p> <p>[^\\]+^</p> |
| \$\$INCLUDE | :\$INCLUDE <filepath> | Causes the named file to be copied as is into the output file. |
| \$\$EXECMODULE | :\$EXECMODULE <module_name> | Causes the named module to be applied in the current context. The module's statements are executed as if they were included in-line at the point where this statement is placed. |
| \$\$NEWFILE | :\$NEWFILE <filename> | Causes the named file to be created and set as the output file. Any existing file with that name is over-written. |
| \$\$NEWDIR | :\$NEWDIR <dir_path> | Causes the named directory to be created and set as the output directory. |
| \$\$SETDIR | :\$SETDIR <dirpath> | Causes the named directory to be set as the output directory. Subsequent file operations will occur in this directory. |
| \$\$APPENDFILE | :\$APPENDFILE <filename> | Causes the named file to be opened for text to be added at its end and sets the file as the output file. If the file does not exist, it is created. |
| \$\$INSERTFILE | :\$INSERTFILE <into_file> <file> <section_tag> | <p>Causes 'file' to be inserted into the section of 'into_file' which is delineated above and below by the left justified text 'section_tag'.</p> <p>The section tag may not contain spaces. Previous lines in that section are removed and replaced with the contents of 'file'.</p> <p>The output file active at the time this directive is executed remains in effect. e.g. \$\$INSERTFILE a b //b-section Inserts the contents of 'b' into 'a' between the two lines containing the comment "//b-section".</p> |
| \$\$FILEEXISTS | :\$FILEEXISTS <filepath> | Sets the variable <file.exists> to 'true' if the file exists. |

| | | |
|---------------|------------------------------|---|
| \$\$DIREXISTS | :\$DIREXISTS <dirpath> | Sets the variable <dir.exists> to 'true' if the directory exists. |
| \$\$SHELLCMD | :\$SHELLCMD <command-string> | Issues a “system” call that invokes ‘command-string’ in the command shell of the active process. SNIP waits for the command to complete. For example: <div>\$\$SHELLCMD make -f Makefile</div> |

Defines Variables

None

Example:

```
<attr.is_local> : process(<attr.name>);
```

Loadtime \$\$IMPORT Directive

Purpose

Imports another template file containing additional shared modules, and perhaps other \$\$IMPORT directives.

Syntax

```
$$IMPORT <filename>
```

Grammar

| | |
|-------------|--|
| <filename > | A fully qualified pathname to a template file. |
|-------------|--|

Example

```
$$IMPORT \templates\shared_modules.tem
```

Template File Block Statement

Block statements allow a group of nested statements to be conditionally processed and include the following:

- Block Statement.
- Select Statement.
- Module.

.block statement

Purpose

- Conditionally executes the statements enclosed within it. This is the most straightforward way to make a whole group of statements conditionally executed based on variable settings.
- Most other template statements can appear within a block. It can enclose iterators of all types, simple statements, and more deeply nested blocks.

Syntax

```
.block [<expr>]
    ...
.end
```

Grammar

| | |
|--------|---|
| <var | name> an identifier of a defined variable |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |

Defines Variables

None.

Example

```
.block <!dsm.is_unmanaged>
    :class RelationshipHandle ;
    :class ManagedObjHandle ;
    :class ManagedObj ;
    :class ManagedObjIterator ;
    :class ManagedObjRef ;
.end
```

.select statement

Purpose

Conditionally executes the statements enclosed within it. Executes only one of the statements. This construct exacts a *case ladder* structure, in which only one of the statements is executed (the

first one that can be executed) and the remainder are skipped. A default block is executed if none of the preceding statements are selected.

Syntax

```
.select [<expr>]
  <stmt-1>
  <stmt-2>
  ...
  <stmt-n>
[.default
  ...
.end]
.end
```

Grammar

| | |
|------------|-------------------------------------|
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable |

Defines Variables

None.

Example

```
.select
<attr.simple>    : <attr.kind> <attr.name>,
<attr.ptr_to>    : <attr.kind>* <attr.name>,
.default
  : **ERROR -- <attr.name> must be 'simple' or 'ptr_to'
.end default
.end select
```

.module statement

Purpose

- Establishes a callable module that can be executed by \$\$EXECMODULE directives within the template.
- Most other template statements can appear within a module. It can enclose iterators of all types, simple statements, and nested blocks.
- Module declarations may not be nested inside a block or iterator. They are always global to the template.

Syntax

```
.module <module_name>
    ...
.end
```

Grammar

| | |
|---------------|----------------|
| <module_name> | An identifier. |
|---------------|----------------|

Defines Variables

None.

Example

```
...
.module emit_copyright
    :Copyright (c) 1995 by <options.company>
.end
...
:$$NEWFILE <dsm.name>.cxx
:$$EXECMODULE emit_copyright
...
```

Template File Iterator Statements

Iterator statements add variables to the context made available to the enclosing statements and include the following:

.each_characteristic statement

Purpose

Iterates over the set of characteristics of the bound element in the immediate scope.

Syntax

```
.each_characteristic [<varname>] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|-----------|---|
| <varname> | Name to use as a variable prefix in the iterator's scope. |
| <expr> | Selection expression in terms of variables. |

Defines Variables

| | |
|----------------------|--|
| <^var_name^.name> | Takes on the name of each characteristic. |
| <^var_name^.value> | Takes on the value of each characteristic. |
| <^var_name^.is_1st> | “True” if this is the first one in the list. |
| <^var_name^.is_nth> | “True” if this is the last one in the list. |
| <^var_name^.is_only> | “True” if this is the only one in the list. |
| <^var_name^.l_pos> | The position of this item in the list. |

.each_event statement

Purpose

Iterates through the events that are defined in the model via an event list.

Syntax

```
.each_event [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables

| | |
|------------------------|---|
| <event.name> | Name of the event. |
| <event.obj_consumes> | "True" if this event is consumed by the current object, active inside an enclosing <code>.each_object</code> statement. |
| <event.state_consumes> | "True" if this event is consumed by the current state, active inside an enclosing <code>.each_state</code> statement. |
| <event.is_1st> | "True" if this is the first one in the list. |
| <event.is_nth> | "True" if this is the last one in the list. |
| <event.is_only> | "True" if this is the only one in the list. |
| <event.l_pos> | The position of this item in the list. |

Example

```
:typedef enum {
  .each_event
  <!event.is_nth>      :      <event.name>,
  <event.is_nth>      :      <event.name>
  .end
  :} eventsT ;
```

.each_pragma_value statement

Purpose

Iterates through the values that belong to a pragma definition.

Syntax

```
.each_pragma_value <pragma_name>
  ...
[.separator
  ...
.end]
.end
```

Grammar

| | |
|---------------|---------------------------|
| <pragma_name> | Identifier of the pragma. |
|---------------|---------------------------|

Defines Variables

| | |
|-------------------------|--|
| <^pragma_name^.value> | Takes on each pragma value. |
| <^pragma_name^.is_1st> | “True” if this is the first one in the list. |
| <^pragma_name^.is_nth> | “True” if this is the last one in the list. |
| <^pragma_name^.is_only> | “True” if this is the only one in the list. |
| <^pragma_name^.l_pos> | The position of this item in the list. |
| ^pragma_name^ | The identifier of the pragma. |

Example

```
.each_pragma_value h_include
    :#include \<<h_include.value>.h\>
.end
```

.each_var_item statement

Purpose

Iterates through the values that belong to a variable that has been set using the \$\$SETVAR directive. The variable can be either a value set, or an associative array mapping keys onto values.

Syntax

```
.each_var_item <var_name>
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|----------------------------|
| <var_name> | identifier of the variable |
|------------|----------------------------|

Defines Variables

| | |
|---------------------|--|
| <^var_name^.key> | Takes on each var item key (associative arrays only) |
| <^var_name^.value> | Takes on each var item value |
| <^var_name^.is_1st> | “True” if this is the first one in the list. |
| <^var_name^.is_nth> | “True” if this is the last one in the list. |

| | |
|----------------------|---|
| <^var_name^.is_only> | “True” if this is the only one in the list. |
| <^var_name^.l_pos> | The position of this item in the list. |
| ^var_name^ | The identifier of the var |

Example

```

:$$SETVAR planes[] += comercial jet
:$$SETVAR planes[] += military jet
:Types of planes --
.each_var_item planes
:  <planes.value>
.end

```

.each_token statement

Purpose

Finds string values that match a regular expression within a search string, and makes those values available.

Syntax

```

.each_token <varname> <reg-exp> <text-to-search>
    ...
[.separator
    ...
.end]
.end

```

Grammar

| | |
|------------------|--|
| <varname> | Name to use as a variable prefix in the iterator’s scope. |
| <reg-exp> | Regular expression to match (quote the string if it contains white space chars). |
| <text-to-search> | The text in which to find the tokens. |

Define Variables

| | |
|-------------------|----------------------------|
| <^varname^.value> | Takes on each token value. |
|-------------------|----------------------------|

.each_literal statement

Purpose

Iterates through the values that belong to a literal text definition. The enclosing scope can be either the top-level scope or an object iterator.

Syntax

```
.each_literal [<prefix>] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|-------------------------------------|
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable |

Defines Variables

| | |
|-------------------|--|
| <literal.name> | Takes on the name of each string literal as provided in the DSM. |
| <literal.text> | Takes on the text of each string literal as provided in the DSM. |
| <literal.is_1st> | "True" if this is the first one in the list. |
| <literal.is_nth> | "True" if this is the last one in the list. |
| <literal.is_only> | "True" if this is the only one in the list. |
| <literal.l_pos> | The position of this item in the list. |

Example

```
.each_obj
    :/*
    .each_literal
    <literal.is_comment> :<literal.text>
    .end
    :*/
    ...
```

.each_foreign_type statement

.each_substitution (only for SNIP 1.x compatibility)

Purpose

Iterates through the foreign typedefs that were established by any foreign typedef statements in the model.

Syntax

```
.each_foreign_type
    ...
[.separator
    ...
.end]
.end
```

Defines Variables

| | |
|----------------------------|--|
| <foreign.name> | (-or- obsolete <subs.name>) Takes on the name of each foreign type. |
| <foreign.type_decl> | (-or- obsolete <subs.value>) Takes on each type's declaration string. |
| <foreign.is_1st> | "True" if this is the first one in the list. |
| <foreign.is_nth> | "True" if this is the last one in the list. |
| <foreign.is_only> | "True" if this is the only one in the list. |
| <foreign.l_pos> | The position of this item in the list. |
| <foreign.^characteristic^> | One var for each characteristic set for the foreign type. |

Example:

```
.each_foreign_type
    : <foreign.name> is defined as "<foreign.type_decl>"
.end
```

.each_object statement

Purpose

Iterates through the object definitions in the model.

Syntax

```
.each_object [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables

| | |
|------------------------|--|
| <obj.name> | The name of this object. |
| <obj.has_parent> | "True" if this object has one or more parents. |
| <obj.has_parents> | "True" if this object has one or more parents. |
| <obj.has_states> | "True" if this object has one or more states. |
| <obj.has_attributes> | "True" if this object has one or more attributes. |
| <obj.has_objects> | "True" if this object has one or more nested objects. |
| <obj.has_functions> | "True" if this object has one or more functions. |
| <obj.in_^mapping^> | "True" if this object was used in a mapping with name ^mapping^. |
| <obj.not_used> | "True" if this object was not used by any other object in the model. |
| <obj.is_1st> | "True" if this is the first one in the list. |
| <obj.is_nth> | "True" if this is the last one in the list. |
| <obj.is_only> | "True" if this is the only one in the list. |
| <obj.l_pos> | The position of this item in the list. |
| <obj.^characteristic^> | Set to "True" for each defined characteristic. |
| <literal.^name^> | Set to the text of each defined text literal. |
| <^pragma_name^> | Set to "True" for each defined pragma nested within the object definition - 'pragma options' also sets each of its values (options.^value^) to "true". |

Example

```
...
.each_obj
: void destroy_<obj.name> (<obj.name>_t **<obj.name> );
.end
...
```

.each_parent statement

Purpose

Iterates through the parents that belong to an object definition.

Syntax

```
.each_parent [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables

| | |
|---------------------------|---|
| <parent.name> | Name of the current parent object. |
| <parent.^characteristic^> | Set for each explicit characteristic attached to the parent object. |
| <parent.is_1st> | "True" if this is the first one in the list. |
| <parent.is_nth> | "True" if this is the last one in the list. |
| <parent.is_only> | "True" if this is the only one in the list. |
| <parent.l_pos> | The position of this item in the list. |

Example

```
.each_obj
...
: class <obj.name> : public ManagedObj
.each_parent
:
, public <parent.name>
.end
: {
...
: };
.end
```

.each_ancestor statement

Purpose

Iterates through the parent objects from top to bottom in the ancestor chain for an object.

Syntax

```
.each_ancestor [prefix] [<expr>]
...
[.separator
...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables

| | |
|-----------------------------|---|
| <ancestor.name> | Name of the current parent object. |
| <ancestor.^characteristic^> | Set for each explicit characteristic attached to the ancestor object. |

| | |
|---------------------|--|
| < ancestor.is_1st> | "True" if this is the first one in the list. |
| < ancestor.is_nth> | "True" if this is the last one in the list. |
| < ancestor.is_only> | "True" if this is the only one in the list. |
| < ancestor.l_pos> | The position of this item in the list. |

Example

```
.each_obj
    ...
.each_ancestor
    :    <obj.name> has ancestor <ancestor.name>
.end
.end
```

.each_used_obj statement

Purpose

- Iterates through the objects (those that are defined in the model) that are used to define the enclosing object.
- The contributions of objects in defining parent objects, referential attributes, function parameters, etc. are considered uses of another object.

Syntax

```
.each_used_obj [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables

| | |
|-----------------------------|--|
| <used_obj.name> | Name of the current parent object. |
| <used_obj.is_self> | "True" if the current object is the enclosing object. |
| <used_obj.is_own_parent> | "True" if the used object is a parent of the enclosing object. |
| <used_obj.has_parents> | "True" if the used object has parent objects of its own. |
| <used_obj.^characteristic^> | "True" for each characteristic which is a characteristic of the object used. That is, the variables 'used_obj.*' will be set to match 'obj.*' for the object being referenced. |
| <used_obj.is_1st> | "True" if this is the first one in the list. |
| <used_obj.is_nth> | "True" if this is the last one in the list. |
| <used_obj.is_only> | "True" if this is the only one in the list. |
| <used_obj.l_pos> | The position of this item in the list. |

Example:

```
.each_obj
  :$ $NEWFILE <obj.name>.c
.each_used_obj
  :#include "<used_obj.name>.h"
.end
:
: ...rest of c file...
:
.end
```

.each_using_attr statement

Purpose

- Iterates through the attributes (those that are defined in the model) that use the enclosing object.
- Referencing an object in the attr kind field is considered having an attribute that uses an object.

Syntax

```
.each_using_attr [prefix] [<expr>]
...
[.separator
...]
```

```
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables

| | |
|------------|--|
| <attr.*> | Normal set of variables for an attribute (see Attribute Iterator). |
| <attr.obj> | Name of the attribute's enclosing object. |

Example

```
.each_obj
  :$$NEWFILE <obj.name>.xrf
.each_using_attr
  :// used by <attr.obj>.<attr.name>
.end
```

.each_attribute statement

Purpose

Iterates through the attributes that belong to an object definition.

Syntax

```
.each_attribute [prefix] [<expr>]
  ...
[.separator
  ...
.end]
.end
```

Grammar

| | |
|----------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |

| | |
|------------|--------------------------------------|
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables

| | |
|-------------------------|--|
| <attr.name> | Name of the current attribute. |
| <attr.kind> | Name of the target type (kind) of attribute. |
| <attr.kind^kind-name^> | Set to “True” for the supplied name of the type (kind) of attribute. |
| <attr.^mapping^> | The mapping name or 'simple' if one was not specified. |
| <attr.is_self> | “True” when <attr.kind> references the attr's object. |
| <attr.is_local> | “True” when <attr.kind> is an object in the model. |
| <attr.is_foreign> | “True” when <attr.kind> is not an object in the model. |
| <attr.has_init> | “True” when the attribute has an initial value assigned to it. |
| <attr.init_val> | The text of the initial value. |
| <attr.is_1st> | “True” if this is the first one in the list. |
| <attr.is_nth> | “True” if this is the last one in the list. |
| <attr.is_only> | “True” if this is the only one in the list. |
| <attr.l_pos> | The position of this item in the list. |
| <attr.^characteristic^> | Set to “True” for each defined characteristic |

Example

```
.each_attribute
    <attr.is_local> : <attr.name> references <attr.kind> \
    <attr.is_local> : which is an object in the model.
.end
```

.refd_obj statement

Purpose

Inside an attribute iterator, this statement binds the referenced object that the attribute uses as its kind (type). If the attribute is foreign, the statement body is not executed.

Syntax

```
.refd_obj [prefix] [<expr>]
    ...
.end
```

Grammar

| | |
|----------|--|
| <prefix> | Identifier to use in variable name prefixes. |
|----------|--|

| | |
|------------|--------------------------------------|
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | An identifier of a defined variable. |

Defines Variables:

The same set of variables is bound as for objects.

.each_using_attr statement

Purpose

- Iterates through the attributes (those that are defined in the model) that use the enclosing object.
- Referencing an object in the attr kind field is considered having an attribute that uses an object.

Syntax

```
each_using_attr [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|------------|--|
| <attr.*> | Normal set of variables for an attribute (see Attribute Iterator). |
| <attr.obj> | Name of the attribute's enclosing object. |

Example:

```
.each_obj
:$$NEWFILE <obj.name>.xrf
```

```
.each_using_attr
    :// used by <attr.obj>.<attr.name>
.end
.end
```

.each_function statement

Purpose

Iterates through functions that:

- Have been defined within an object definition, or
- Have been defined globally.

This statement is context sensitive. If this statement appears inside of an enclosing `.each_object` statement, it iterates through the object's functions. If no enclosing `.each_object` statement is present, it iterates through the globally defined functions.

Syntax

```
.each_function [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|--------------------------------|--|
| <ftn.name> | The name of the function as given in the DSM input. |
| <ftn.has_params> | "True" if the function has parameters. |
| <ftn.has_return> | "True" if the function has a return value. |
| <ftn.rtype_kind> | The name of the kind of return value that is returned by the function. |
| <ftn.rtype_is_local> | "True" if the return kind is an object in the model. |
| <ftn.rtype_is_foreign> | "True" if the return kind is not an object in the model. |
| <ftn.rtype_^{mapping}> | Set to "True" for the mapping type specified in the DSM input, or 'normal' if no mapping type was specified. |

| | |
|------------------------|--|
| <ftn.has_body> | “True” if a body was specified for the function. |
| <ftn.body_text> | The multi-line text of the body of the function as given in the DSM input. |
| <ftn.is_1st> | “True” if this is the first one in the list. |
| <ftn.is_nth> | “True” if this is the last one in the list. |
| <ftn.is_only> | “True” if this is the only one in the list. |
| <ftn.l_pos> | The position of this item in the list. |
| <ftn.^characteristic^> | Set to “True” for each defined characteristic. |

Example:

```
.each_obj  
.  
.each_function  
  
<ftn.is_virtual>           :      virtual \  
<!ftn.is_virtual>         :          \  
<ftn.has_return>          :      <ftn.rtype_kind>   \  
<ftn.rtype_is_local>     :*\  
<!ftn.has_return>        :void \  
                           :  <ftn.name> (  
.  
.each_param  
  
<param.is_local>          :<param.kind> *<param.name>\  
<param.is_foreign>       :<param.kind> <param.name>\  
<!param.is_nth>          :, \  
.  
.end  
                          :);  
.  
.end  
                          :  
.  
.end
```

.each_parameter statement

Purpose

Iterates through the parameters that belong to a function definition.

Syntax

```
.each_parameter [prefix] [<expr>]
```

```

...
[.separator
...
.end]
.end

```

Grammar

| | |
|------------|---|
| <prefix> | Identifier to use in variable name prefixes |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|--------------------------|---|
| <param.name> | Name of the current parameter. |
| <param.kind> | Name of the kind of the parameter as given in the DSM input. |
| <param.^mapping^> | The mapping name or 'simple' if one was not specified. |
| <param.is_local> | "True" when <param.kind> is an object in the model. |
| <param.is_foreign> | "True" when <param.kind> is not an object in the model. |
| <param.has_default_val> | "True" when the parameter has a default value assigned to it. |
| <param.default_val> | The text of the default value. |
| <param.is_1st> | "True" if this is the first one in the list. |
| <param.is_nth> | "True" if this is the last one in the list. |
| <param.is_only> | "True" if this is the only one in the list. |
| <param.l_pos> | The position of this item in the list. |
| <param.^characteristic^> | Set to "True" for each defined characteristic |

Example:

```
.each_function
<ftn.has_return>          :<ftn.rtype_kind>  \
<ftn.rtype_is_local>    :* \
<!ftn.has_return>        :void \
                           : <ftn.name> (\

.each_param
<param.is_local>          :<param.kind> *<param.name> \
<param.is_foreign>       :<param.kind> <param.name> \
<!param.is_nth>          :; \
.end
                           :);
.end
```

.each_state statement

Purpose

Iterates through the states that belong to an object definition or a top-level set of states.

Syntax

```
.each_state [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|---------------------------------|--|
| <state.name> | The name of the state. |
| <state.has_action> | "True" if this state has an action specified for it. |
| <state.is_terminal> | "True" if this is a terminal state (no out transitions). |
| <state.is_1st> | "True" if this is the first one in the list. |
| <state.is_nth> | "True" if this is the last one in the list. |
| <state.is_only> | "True" if this is the only one in the list. |
| <state.l_pos> | The position of this item in the list. |
| <state.^characteristic^> | Set to "True" for each defined characteristic |
| <state_action.name> | The name of the state's action if one was specified. |
| <state_action.has_body> | Returns "True" if a body was specified. |
| <state_action.body_text> | Returns the body text or "" if no body was supplied. |
| <state_action.^characteristic^> | Set to "True" for each defined characteristic. |

Example:

```
.each_obj <obj.has_states>
    :The object <obj.name> defines the following states:
    .each_state
        :      <state.name>
    .end
    :
    .end
```

.each_transition statement**Purpose**

Iterates through the state transitions that belong to a given state defined within an object definition.

Syntax

```
.each_transition [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|-----------------------|--|
| <tx.event> | Name of the event that causes the transition to be taken. |
| <tx.target_state> | Name of the resulting state. |
| <tx.is_default> | "True" if this is the default transition taken when an event is not consumed by any other transition (event name was specified as 'any' in DSM input). |
| <tx.is_1st> | "True" if this is the first one in the list. |
| <tx.is_nth> | "True" if this is the last one in the list. |
| <tx.is_only> | "True" if this is the only one in the list. |
| <tx.l_pos> | The position of this item in the list. |
| <tx.^characteristic^> | Set to "True" for each defined characteristic |

Example

```
.each_obj <obj.has_states>
    :The object <obj.name> defines the following states:
.each_state
    :      <state.name>
.each_transition
    :      on <tx.event> goes to <tx.target_state>
.end
.end
    :
.end
```

.each_proc statement

Purpose

Iterates through the processes that belong to an object definition or a top-level set of processes.

Syntax

```
.each_proc [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|-------------------------|--|
| <proc.name> | The name of the process. |
| <proc.has_flows> | “True” if this process has flows specified for it. |
| <proc.is_1st> | “True” if this is the first one in the list. |
| <proc.is_nth> | “True” if this is the last one in the list. |
| <proc.is_only> | “True” if this is the only one in the list. |
| <proc.l_pos> | The position of this item in the list. |
| <proc.^characteristic^> | Set to “True” for each defined characteristic |

Example:

```
.each_obj <obj.has_states>
    :The object <obj.name> has the following processes:
.each_proc
    :    <proc.name>
.end
    :
.end
```

.each_flow statement

Purpose

Iterates through the flows that belong to a given process.

Syntax

```
.each_flow [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|-------------------------|--|
| <flow.name> | Name of the flow. |
| <flow.is_inflow> | "True" if this flow comes into the process. |
| <flow.is_outflow> | "True" if this flow.leaves the process. |
| <flow.src_dest_name> | The name of the source or destination process. |
| <flow.is_1st> | "True" if this is the first one in the list. |
| <flow.is_nth> | "True" if this is the last one in the list. |
| <flow.is_only> | "True" if this is the only one in the list. |
| <flow.l_pos> | The position of this item in the list. |
| <flow.^characteristic^> | Set to "True" for each defined characteristic |

Example

```
.each_obj <obj.has_states>
    :The object <obj.name> defines the following processes:
.each_proc
    :
    :<proc.name>
.each_flow
<flow.is_inflow>      : <flow.name> from <flow.src_dest_name>
<flow.is_outflow>     : <flow.name> to  <flow.src_dest_name>
    :
```

```
.end
.end
:
.end
```

.each_instance statement

Purpose

Iterates through each instance definition in the model.

Syntax

```
.each_instance [prefix] [<expr>]
...
[.separator
...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|-------------------------|--|
| <inst.name> | Name of this object instance. |
| <inst.kind> | Kind of object (defined in the model). |
| <inst.is_1st> | “True” if this is the first one in the list. |
| <inst.is_nth> | “True” if this is the last one in the list. |
| <inst.is_only> | “True” if this is the only one in the list. |
| <inst.l_pos> | The position of this item in the list. |
| <inst.^characteristic^> | Set to “True” for each defined characteristic. |

Example:

```
.file objs.cpp
#include \<<dsm.name>.h\>
```

```

:
: // Create objects using default constructors
:
.each_instance
: <inst.kind> * <inst.name> = new <inst.kind> ;
.end

```

.each_assignment statement

Purpose

Iterates through the attribute assignments that belong to an object instance definition.

Syntax

```

.each_assignment [prefix] [<expr>]
...
[.separator
...
.end]
.end

```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|------------------|---|
| <assign.name> | The identifier that appeared to the left of the assignment operator. |
| <assign.value> | The string or identifier that appeared to the right of the assignment operator. |
| <assign.is_1st> | "True" if this is the first assignment in the list. |
| <assign.is_nth> | "True" if this is the last assignment in the list. |
| <assign.is_only> | "True" if this is the only assignment in the list. |
| <assign.l_pos> | The position of this item in the list. |

Example: .file objs.cpp

```
    :#include \<<dsm.name>.h\>
    :
    :// Create objects using default constructors
    :
    .each_instance
        :<inst.kind> *<inst.name> = new <inst.kind> ;
    .end
    :void
    :InitializeInstances()
    :{
    .each_instance
    .each_assignment
        :      <inst.name>->Set<assign.name>(<assign.value>);
    .end
    :
    .end
    :}
```

.each_parameter statement**Purpose**

Iterates through the parameters that belong to a function definition.

Syntax

```
.each_parameter [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|--------------------------|--|
| <param.name> | Name of the current parameter. |
| <param.kind> | Name of the kind of the parameter as given in the DSM input. |
| <param.^mapping^> | The mapping name or 'simple' if one was not specified. |
| <param.is_local> | "True" when <param.kind> is an object in the model. |
| <param.is_foreign> | "True" when <param.kind> is not an object in the model. |
| <param.has_default_val> | "True" when the parameter has an default value assigned to it. |
| <param.default_val> | The text of the default value. |
| <param.is_1st> | "True" if this is the first one in the list. |
| <param.is_nth> | "True" if this is the last one in the list. |
| <param.is_only> | "True" if this is the only one in the list. |
| <param.l_pos> | The position of this item in the list. |
| <param.^characteristic^> | Set to "True" for each defined characteristic. |

Example:

```
.each_function
<ftn.has_return>          :<ftn.rtype_kind>  \
<ftn.rtype_is_local>    :*\
<!ftn.has_return>        :void \
                           : <ftn.name> (\
.
each_param
<param.is_local>          :<param.kind> *<param.name>\
<param.is_foreign>       :<param.kind> <param.name>\
<!param.is_nth>          : , \
.
end
                           :);
.
end
```

.each_transition statement

Purpose

Iterates through the state transitions that belong to a given state defined within an object definition.

Syntax

```
.each_transition [prefix] [<expr>]
    ...
[.separator
    ...
.end]
.end
```

Grammar

| | |
|------------|--|
| <prefix> | Identifier to use in variable name prefixes. |
| <expr> | <vars> [' ' <vars>] |
| <vars> | <var> <var> ... |
| <var> | '<' <var-name> '>' |
| <var-name> | Identifier of a defined variable. |

Defines Variables

| | |
|-----------------------|--|
| <tx.event> | Name of the event that causes the transition to be taken. |
| <tx.target_state> | Name of the resulting state. |
| <tx.is_default> | “True” if this is the default transition taken when an event is not consumed by any other transition (event name was specified as 'any' in DSM input). |
| <tx.is_1st> | “True” if this is the first one in the list. |
| <tx.is_nth> | “True” if this is the last one in the list. |
| <tx.is_only> | “True” if this is the only one in the list. |
| <tx.l_pos> | The position of this item in the list. |
| <tx.^characteristic^> | Set to “True” for each defined characteristic |

Example:

```
.each_obj <obj.has_states>
    :The object <obj.name> defines the following states:
    .each_state
        : <state.name>
```

```
.each_transition
    :          on <tx.event> goes to <tx.target_state>
.end
.end
    :
.end
```


7 Provided templates and object models

Summary

Cleanscape SourceMill comes with sample production templates that you can use as they are or modify for your own specific needs. This chapter lists the production templates provided with Cleanscape SourceMill, with a description of each file and its output.

The chapter includes the following sections:

- Production templates
- Sample object models

Additional online help

Each Template provided with Cleanscape SourceMill has an associated online help file that includes the following additional information for the Production Templates and other sample templates in the Cleanscape SourceMill package:

- Model Elements and Characteristics, like parameterization, mapping keywords, and characteristics of elements
- Instructions on using the generated code

Production templates

Class per Object Using MFC 4.0

Filename

CPP_MFC.TEM

Description

This template produces lightweight set of C++ classes that participate in a multi-class object model. It provides a minimal set of proper class management functions (copy, assign, destroy,...) to support each of your objects, and creates access and navigation functions to manipulate the connections between objects. Lists of objects and pointers to objects are created using the template collections provided in the MFC class library as of version 4.0 of the Microsoft Visual C++ development system.

Output

Emits a header file and a body file that contain class declarations and functions to manage the model objects, and the references between them. References between objects are implemented using pointers and pointer lists.

Pointers and pointer lists may be configured to own the objects they reference. If they do own the objects, referenced objects are deleted as a side effect of deleting the referencing object. They are also copied as necessary to avoid sharing pointers inappropriately.

Class per Object Using Provided Lists

Filename

CPP_STD.TEM

Description

This is the most lightweight of the C++ targets.

- Provides a minimal set of proper class management functions (copy, assign, destroy...) to support each of your objects.

Output

Emits a header file and a C file that contain class declarations and functions to manage the model objects, and the references between them. References between objects are implemented using pointers.

Pointers and pointer lists may be configured to own the objects they reference. If they do own the objects, referenced objects are deleted as a side effect of deleting the referencing object.

Class per Object Using STL

Filename

CPP_STL.TEM

Description

This template implements a class per object, using collection classes from STL to realize relationships between classes.

- Produces lightweight set of C++ classes that participate in a multi-class object model.
- Provides a minimal set of proper class management functions (copy, assign, destroy, ...) to support each of your objects, and creates access and navigation functions to manipulate the connections between objects.
- Lists of objects and pointers to objects are created using the template collections provided by STL.

Output

Emits a header file and a body file that contain class declarations and functions to manage the model objects, and the references between them. References between objects are implemented using pointers and pointer lists.

Pointers and pointer lists may be configured to own the objects they reference. If they do own the objects, referenced objects are deleted as a side-effect of deleting the referencing object. They are also copied as necessary to avoid sharing pointers inappropriately.

Class per Object Using RogueWave

Filename

ROGUE.TEM

Description

This template implements a class per object, using collection classes from RogueWave to realize relationships between classes.

- Produces lightweight set of C++ classes that participate in a multi-class object model.
- Provides a minimal set of proper class management functions (copy, assign, destroy, ...) to support each of your objects, and creates access and navigation functions to manipulate the connections between objects.
- Lists of objects and pointers to objects are created using the template collections provided by the RogueWave class library.

Output

Emits a header file and a body file that contain class declarations and functions to manage the model objects, and the references between them. References between objects are implemented using pointers and pointer lists.

Pointers and pointer lists may be configured to own the objects they reference. If they do own the objects, referenced objects are deleted as a side effect of deleting the referencing object. They are also copied as necessary to avoid sharing pointers inappropriately.

ANSI C Struct per Object

Filename

S_ONLYKR.TEM

Description

This template implements a struct per object, and creates routines to manipulate the struct as an abstract data type.

- Most lightweight of the K&R C targets.
- Provides struct declarations that implement the attributes and links of each of your objects.
- Functions stubs are supplied that allocate/initialize and finalize/deallocate the struct.
- Guidance is given via TODO insertions about completing the allocation and deletion function stubs.

Output

Emits a K&R C header file and a C file that contain struct declarations and allocation and deletion functions. References between objects are implemented using pointers.

Ada-83 Managed Record per Object

Filename

SNET_ADA.TEM

Description

This template implements objects as dynamically allocated Ada 83 Records.

- Record instances are managed so that the act of deleting records removes that record from all in-bound points of reference.
- Emits Ada 83 type definitions and functions to manage the set of objects and relationships defined in the object model.

Output

The template SNET_ADA.TEM implements the object network as a set of Ada types. This template produces .ads and .adb files containing

- Record declarations for each of your objects
- Default implementations of your object's create and destroy functions

K&R C Struct per Object

Filename

S_ONLY.TEM

Description

This template implements a struct per object, and creates routines to manipulate the struct as an abstract data type.

- Most lightweight of the ANSI C targets.
- Provides struct declarations that implement the attributes and links of each of your objects.
- Functions stubs are supplied that allocate/initialize and finalize/deallocate the structs. Guidance is given via 'TODO' insertions about completing the allocation and deletion function stubs.

Output

Emits an ANSI C header file and a C file that contain struct declarations and allocation and deletion functions. References between objects are implemented using pointers.

C++ Managed Class per Object

Filename

SNET_CPP.TEM

Description

This template implements objects as dynamically allocated C++ class instances.

- Class instances are managed so that the act of deleting instances removes them from all in-bound points of reference.
- Most full-featured of the C++ targets.
- Provides many convenience functions and mechanisms to support your objects and to safely manage the connections between them.

Output

The template SNET_CPP.TEM implements the object network as a set of C++ classes. It emits a header file and a C file that contain class declarations and functions to manage the model objects & the references between them. References between objects are implemented using managed encapsulated pointers.

The generated code manages the integrity of links between objects. If an object is deleted, the references to that object that are contained in `ManagedObjRef` and `ManagedObjIterators`

K&R C Managed Struct per Object**Filename**

SNET_KR.TEM

Description

This template implements objects as dynamically allocated C structs. Struct instances are managed so that the act of deleting instances removes them from all in-bound points of reference.

- Most full-featured of the K&R C targets.
- Provides routines that manage the connections between objects in the face of deallocation.
- Use the generated interface to ensure safe operations with respect to pointer integrity management.

Output

Emits a K&R C header file and a C file that contain struct declarations and functions to manage the model objects, and the references between them. References between objects are implemented using managed encapsulated pointers.

ANSI C Managed Struct per Object**Filename**

SNETWORK.TEM

Description

This template implements objects as dynamically allocated C structs.

- Struct instances are managed so that the act of deleting instances removes them from all in-bound points of reference.
- Most full-featured of the ANSI C targets.
- It provides routines that manage the connections between objects in the face of deallocation.
- Use the generated interface to ensure safe operations with respect to pointer integrity management.

Output

Emits an ANSI C header file and a C file that contain **struct** declarations and functions to manage the model objects, and the references between them. References between objects are implemented using managed encapsulated pointers.

Example Templates

Cleanscape SourceMill comes with the following example templates that you can use as models for developing your own production templates.

Ada-9x Sample Template

Filename

ADA_9X.TEM

Description

Shows an example of how objects can be mapped into the facilities of the Ada-95 language.

Output

Implements the object model as a set of encapsulated Ada 95 record types. It emits **.ads** and **.adb** files that contain type declarations and procedures and functions to manage the model objects and the references between them.

C Based Finite State Machine

Filename

FSM_C.TEM

Description

This template implements a finite state machine in C for a DSM file that contains a state model.

Output

Emits an ANSI C program that implements a sample Finite State Machine (FSM) engine. The program takes event numbers as input, and emits a trace of the event-state transitions that are taken to stdout.

ParcPlace Smalltalk Class per Object**Filename**

SMALPARC.TEM

Description

This template implements a class per object, and creates routines access the class' members.

Output

Emits a 'filein' / 'fileout' formatted set of statements into a .ST file. The statements create classes and subclasses to manage the objects defined in the model.

ANSI SQL DDL Table per Object**Filename**

SQL_ANSI.TEM

Description

This template implements objects as DDL defined tables.

Output

Emits SQL Create Table directives to create tables that can hold instances of the objects defined in the data model as rows in the tables. Links between objects are implemented as foreign keys.

Other

See the accompanying DSM file for a sample input model that defines objects and attributes in a manner compatible with this template's expectations.

Sample Object Models

| Filename | Template | Description |
|--------------|---------------------------|---|
| FSM_C.DSM | FSM_C.TEM | Sample DSM showing use of state abstractions |
| SCHOOL.DSM | SNET_CPP.TEM | A SNIP input file for a sample in-memory data structure. |
| SQL_ANSI.DSM | SQL_ANSI.TEM | Example DSM provided to demonstrate how mapping keywords and characteristics can be used to communicate model information to a template |
| STDTEST.DSM | CPP_STD.TEM VC++ types | Sample model using default list classes Demonstrates use of various characteristics recognized by that template |

8 Appendix: Instantiating Code Patterns

White Paper

Adapted from “Instantiating code patterns” by Fred Wild, Advantage Software Technologies

Introduction

Although we often use patterns, we sometimes do so unconsciously — not by design. Whenever we follow a pattern, we are applying experienced-based learning. Whenever we need to accomplish something that we've seen others do, we tend to mimic the strategies and actions that made that thing successful in the past.

Experienced software developers use patterns heavily in the design-level aspects of their work. Design patterns make us more productive, more likely to achieve success, and generally more valuable as project participants.

We might think that less-experienced engineers are disadvantaged in this respect, but rather than accepting the risks that lack of experience brings with it, effective organizations take an early and active role in sharing important domain knowledge with them. They gather the collective design wisdom of the group, descriptive language (manifest vocabulary) of what the important design patterns are, and when and how to apply them. Developers of all ranks drink up such design patterns and quickly begin to apply them effectively.

With respect to patterns of code, we can provide much more direct help and support than simply documenting best practices. We can build the use of those code patterns into the tools we use in the software development process itself. Cleanscape SourceMill (formerly known as SNIP) is a commercially available tool designed specifically for that purpose. This article discusses the use of patterns in the coding aspect of software development, and describes how to use Cleanscape SourceMill to define and instantiate code patterns.

Design patterns and code patterns

Design patterns are logical in nature. They embody approaches and strategies that are relevant to a number of possible implementations. A design pattern is something you can visualize independent of a particular programming language.

For example, given that you need to manage the dynamic allocation and deallocation of objects, you can specify a logical strategy for minimizing memory utilization by keeping reference counts on objects and copying those objects only when an operation is performed to modify them. Having defined the strategy and given it a name, you can treat it as a design idiom.

During design you can say, "XYZ will be a reference-counted object," and experienced programmers will understand what that implies.

In contrast, code patterns are physical in nature. They focus on how a particular structure or sequence of action is accomplished using the specific mechanisms of a programming language. With C++, we've been trained to create code-level abstractions to support certain design idioms (for example, using collection classes and templates), but this practice has limitations. Classes and templates are not adequate mechanisms to implement many of the code patterns that are interesting to us. Many important code patterns describe how a class should deal with its parts (a parts-based pattern) or describe how a number of cooperating classes are created to implement a single design idea (a multi-class pattern). Most software developers use parts-based and multi-class patterns, but they tend to do so via hand coding: they either know a pattern by heart, or they copy a code sample that already follows the pattern and modify it to work in the specific case.

Using code patterns

Most production software environments operate under time and quality pressures. As a result, programmers seldom use code patterns. This is neither because the patterns are unknown nor because developers do not recognize cases in which they would be good strategies to apply. It is because using them can introduce risk to time-sensitive projects. For example, a correct implementation of a reference-counted object in C++ requires careful attention to the use of a copy-on-write mechanism that each non-"const" function plays into. This adds unwelcome and complex detail for programmer who codes by hand.

To make use of nontrivial code patterns, we just need an easy way to characterize objects during design and then have those characteristics influence how code is produced.

Pattern instantiation

Instantiation is the process of producing a more defined version of some object by replacing variables with values or other variables. In object-oriented programming, this means producing a particular object from its class template. This involves allocation of a structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.

Programmers can instantiate patterns of code from object models using a code instantiation tool like Cleanscape SourceMill to do the following:

1. Define important code patterns according to local standards and classifications of object characteristics.
2. Instantiate those code patterns for any set of specific objects.

Given a properly thought-out set of rules for creating code for objects and their parts, based on specific characteristics, a programmer can realize gains in standardization, quality, and time-to-implementation. Cleanscape SourceMill enables code patterns that use an object model as an instantiation context to be defined; see *figure 1*. It applies the rules called out in an executable template to the objects and their parts, and produces code files based on those rules.

```
Object Model ---> SNIP ---> Code Files
/
```

Code Templates /

Figure 1: Cleanscape SourceMill's operating model

Strategize

Gaining maximum benefits from Cleanscape SourceMill requires a well-defined strategy in which you:

1. Call out the characteristics of the objects (and their parts) that participate in that strategy, and
2. Decide how code should be created for each of them.

Design

Once your strategy is in place, you create a set of code-creation rules and place them in a Cleanscape SourceMill template file. A number of template files that serve as good starting points for this come with the tool, including one for handling C++ class constructs can be easily extended to include special rules.

You can develop the template files iteratively using Cleanscape SourceMill's user interface. Once you are satisfied with the code pattern that is created by the template, you can add the command form of Cleanscape SourceMill into makefiles for use in non-interactive builds.

Execute

The following example shows how Cleanscape SourceMill can be used to facilitate easy management of dynamically allocated objects, a common C++ programming task.

Whenever you have a contained pointer, it is important for the code in the class containing the pointer to manage the dynamically allocated object correctly.

First, we'll define what it means to manage an object via a contained pointer. In this example, the code-pattern requirements are as follows:

- DOD-STD-2167A All pointers to objects, both owned and shared, are initially set to NULL.
- A "set" operation keeps the pointer passed into it in both the owned and shared pointer cases.
- A "clear" operation deletes the object if it is owned; otherwise, the pointer is set to NULL.
- The destructor deletes the object if it is owned.
- A copy constructor or assignment operation calls "makeCopy()" on an owned object and uses the returned pointer. Pointers are simply copied in shared object cases.

Listing One shows how these rules are expressed in the template file format. Before inspecting the listing, note first that this template is pared down for illustration, so it won't be hard to find a case it doesn't handle (the production template that this example was taken from is much more sophisticated). Second, there are only a few constructs used in Cleanscape SourceMill templates. Once you know them, readability isn't an issue.

Cleanscape SourceMill's template statements draw information from an object model and generate code based on that information. Before you can make sense of template statements, you need to see to what they are referring.

Listing Two is a simple object model we'll use as input to Cleanscape SourceMill. It has just one contained pointer of each type in it — one to an object that is owned (PetOwner::Pet) and another to an object that is shared (PetOwner::Vet). When you apply the template file in *Listing One* to this object model, your rules will be used to generate the code in *Listing Three* (the resulting header file) and *Listing Four* (the resulting body file).

Listing One

```
object PetOwner
  Name      :    string ;
  Pet       :    ptr_to Pet [is_owner] ;
  Vet       :    ptr_to Veterinarian [is_shared] ;
end ;

object Veterinarian
  Name      :    string ;
  StreetAddress :    string ;
  Town      :    string ;
  Phone     :    string ;
end ;

object Pet
  Name      :    string ;
  KindOfPet :    string ;
end ;

foreign string [use_ref] ;
```

Listing Two

```
# Create the header file ...
#
    :$$NEWFILE .hxx
    :
```

```

        :#ifndef _H
        :#define _H
        :
        :// Forward references for each class
        :
.EAch_obj
        :class ;
.END

        :
        :$$EXECMODULE emit_class_decls
        :
        :#endif
        :

# Create the body file ...
#

        :$$NEWFILE .cxx
        :
        :#include ".hxx"
        :
        :$$EXECMODULE emit_class_bodies
        :

#####
##
# Module to emit class declarations
#
.Module emit_class_decls
.EAch_obj

        :

        ://

```

```

: // Class -----
: //
:
: class \
:
: \
.EAch_parent
: \
:
: ,
:
.END

: {
: $$EXECMODULE emit_member_variables
:
: public:
:
: ();
: (const & obj);
: *makeCopy() const ;
: virtual ~();
:
: &operator=(const &rhs) ;
:
: $$EXECMODULE emit_attr_access_ftn_decls
:
: };

.END

.END

```

```
#####  
##  
# Modules to emit class member variable declarations  
#  
.Module emit_member_variables  
.EAch_attr  
    :      m_ ;  
    :      *m_ ;  
.End  
.End  
  
#####  
##  
# Module to emit attribute access function declarations  
#  
.Module emit_attr_access_ftn_decls  
.EAch_attr  
    : $$EXECMODULE emit_get_decl  
    : $$EXECMODULE emit_set_decl  
    :  
.End  
.End  
  
#####  
##  
# Module to emit get functions for an attribute  
#  
.Module emit_get_decl  
    : Get () const ;
```

```

: const & Get () const;
        : const * Get () const;
.END

#####
##

#  Module to emit set functions for an attribute
#

.MOdule emit_set_decl

: void Set(const val);
: void Set(const & val);
        : void Set( *val) ;
        : void Clear () ;

.END

#####
##

#  Module to emit the bodies of member functions for each class
#

.MOdule emit_class_bodies

.EAch_obj

:
:////////////////////
://   class

:$$EXECMODULE emit_class_default_ctor
:$$EXECMODULE emit_class_copy_ctor
:$$EXECMODULE emit_class_destructor
:$$EXECMODULE emit_assignment_op_body

```



```

        : $$EXECMODULE emit_get_and_set_ftn_bodies
.ENDd
.ENDd

#####
##
#   Module to create a class' default constructor
#
.Module emit_class_default_ctor
    :
    :::()
    :{
.EAch_attr
    :   m_ = NULL;
        :   m_ = ;
.ENDd
        :}
.ENDd

#####
##
#   Module to create a class' copy ctor and 'makeCopy' function
#
.Module emit_class_copy_ctor
    :
    :::(const & obj)
        : : \
.EAch_parent

```

```

                                :(obj)\
                                :,
                                : \
                                :
                                :
.END

                                :{
.EAch_attr
                                : m_ = obj.m_;
                                : m_ = obj.m_;
                                : m_ = obj.m_>makeCopy();
.END

                                :}
                                :
                                : *::makeCopy() const
                                :{
                                : return new (*this);
                                :}

.END

#####
##
# Module to create a class' destructor
#
.MOdule emit_class_destructor
                                :
                                :::~~()
                                :{
.EAch_attr
                                : delete m_ ;

```

```
.END

        :}

.END

#####
##
#  Module to create a class' assignment operator
#
.Module emit_assignment_op_body

        :
        : &
        :::operator=(const &rhs)
        :{
        :  if (this == &rhs) return *this ;
        :    // No changes if assignment to self
        :
.EAch_parent

        : (( &) (*this)) = rhs ;
        :    // Assign base class members of
.END
.EAch_attr

        : m_ = rhs.m_;
        : m_ = rhs.m_;
        : m_ = rhs.m_->makeCopy();
.END

        :
        :  return *this ;
        :}

.END
```

```
#####  
##  
# Module to create class member functions that access single valued member vars  
#  
.Module emit_get_and_set_ftn_bodies  
.Each_attr  
: $EXECMODULE emit_get_ftn_body  
: $EXECMODULE emit_set_and_clear_ftn_bodies  
.End  
.End  
  
#####  
##  
# Module to create a 'Get' function body for accessing an attribute  
#  
.Module emit_get_ftn_body  
: Get () const  
:const & Get () const  
:const * Get () const  
: {  
: return m_ ;  
:}  
.End  
  
#####  
##  
# Module to create a 'Set' and 'Clear' function bodies  
#
```

```
.Module emit_set_and_clear_ftn_bodies
```

```
  :void Set (const val)
```

```
  :void Set (const & val)
```

```
    :void Set ( *val)
```

```
      :{
```

```
        : = val ;
```

```
      : if ( != NULL) delete ;
```

```
        : = val ;
```

```
      :}
```

```
    :
```

```
    :
```

```
  :void Clear ()
```

```
    :{
```

```
      : if ( != NULL) delete ;
```

```
        : = NULL ;
```

```
    :}
```

```
.End
```

```
.End
```

Listing Three

```
#ifndef pets_H
```

```
#define pets_H
```

```
// Forward references for each class
```

```
class PetOwner ;
```

```
class Veterinarian ;
```

```
class Pet ;
```

```
// Class PetOwner -----  
class PetOwner {  
    string          m_Name ;  
    Pet             *m_Pet ;  
    Veterinarian    *m_Vet ;  
public:  
    PetOwner();  
    PetOwner(const PetOwner& obj);  
    PetOwner *makeCopy() const ;  
    virtual ~PetOwner();  
  
    PetOwner &operator=(const PetOwner &rhs) ;  
  
    const string &      GetName () const ;  
    void           SetName (const string & val) ;  
  
    const Pet *         GetPet () const ;  
    void               SetPet (Pet *val) ;  
    void               ClearPet () ;  
  
    const Veterinarian * GetVet () const ;  
    void               SetVet (Veterinarian *val) ;  
    void               ClearVet () ;  
};  
  
// Class Veterinarian -----  
class Veterinarian {  
    string          m_Name ;
```

```
    string          m_StreetAddress ;  
    string          m_Town ;  
    string          m_Phone ;  
public:  
    Veterinarian();  
    Veterinarian(const Veterinarian& obj);  
    Veterinarian *makeCopy() const ;  
    virtual ~Veterinarian();  
  
    Veterinarian &operator=(const Veterinarian &rhs) ;  
  
    const string &    GetName () const ;  
    void            SetName (const string & val) ;  
  
    const string &    GetStreetAddress () const ;  
    void            SetStreetAddress (const string & val) ;  
  
    const string &    GetTown () const ;  
    void            SetTown (const string & val) ;  
  
    const string &    GetPhone () const ;  
    void            SetPhone (const string & val) ;  
};  
  
// Class Pet -----  
class Pet {  
    string          m_Name ;  
    string          m_KindOfPet ;
```

```
public:
    Pet();
    Pet(const Pet& obj);
    Pet *makeCopy() const ;
    virtual ~Pet();

    Pet &operator=(const Pet &rhs) ;

    const string &      GetName () const ;
    void              SetName (const string & val) ;

    const string &      GetKindOfPet () const ;
    void              SetKindOfPet (const string & val) ;
};
#endif
```

Listing Four

```
#include "pets.hxx"

////////////////////////////////////

//    class PetOwner
PetOwner::PetOwner()
{
    m_Pet = NULL;
    m_Vet = NULL;
}
PetOwner::PetOwner(const PetOwner& obj)
{
    m_Name = obj.m_Name;
```



```
m_Pet = obj.m_Pet->makeCopy();
m_Vet = obj.m_Vet;
}
PetOwner *PetOwner::makeCopy() const
{
    return new PetOwner(*this);
}
PetOwner::~~PetOwner()
{
    delete m_Pet ;
}
PetOwner &
PetOwner::operator=(const PetOwner &rhs)
{
    if (this == &rhs) return *this ;
    // No changes if assignment to self
    m_Name = rhs.m_Name;
    m_Pet = rhs.m_Pet->makeCopy();
    m_Vet = rhs.m_Vet;

    return *this ;
}
////////////////////////////////////
//    class Veterinarian

Veterinarian::Veterinarian()
{
}
```

```

Veterinarian::Veterinarian(const Veterinarian& obj)
{
    m_Name = obj.m_Name;
    m_StreetAddress = obj.m_StreetAddress;
    m_Town = obj.m_Town;
    m_Phone = obj.m_Phone;
}

Veterinarian *Veterinarian::makeCopy() const
{
    return new Veterinarian(*this);
}

Veterinarian::~~Veterinarian()
{
}

Veterinarian &
Veterinarian::operator=(const Veterinarian &rhs)
{
    if (this == &rhs) return *this ;
    // No changes if assignment to self
    m_Name = rhs.m_Name;
    m_StreetAddress = rhs.m_StreetAddress;
    m_Town = rhs.m_Town;
    m_Phone = rhs.m_Phone;

    return *this ;
}

////////////////////////////////////
//    class Pet

```

```
Pet::Pet()
{
}

Pet::Pet(const Pet& obj)
{
    m_Name = obj.m_Name;
    m_KindOfPet = obj.m_KindOfPet;
}

Pet *Pet::makeCopy() const
{
    return new Pet(*this);
}

Pet::~~Pet()
{
}

Pet &
Pet::operator=(const Pet &rhs)
{
    if (this == &rhs) return *this ;
    // No changes if assignment to self
    m_Name = rhs.m_Name;
    m_KindOfPet = rhs.m_KindOfPet;

    return *this ;
}
```

Reading template statements

Cleanscape SourceMill template files contain a mainline of executable statements and any number of modules that contain executable statements. Executable statements come in three forms: simple statements, blocks, and iterators.

Simple statements have the form:

left-hand-side colon right-hand-side

To the left of the colon is a selection expression. If the expression evaluates to True, the right side is expanded, and the resulting text is emitted into the active output file.

The module `emit_class_decls` in *Listing One* contains the segment of statements in *Example 1*.

Example 1:

```

: //
: // Class <obj.name> -----
: //
:
: class <obj.name> \
<obj.has_parent>      :: \
.each_parent
      :<parent.name> \
<!parent.is_nth>      :,
<parent.is_nth> :
.end
: {

```

The first five statements have no variables on the left, which means they are selected in all cases, so each of their right sides are expanded and emitted into the active output file (variables being substituted where indicated). A backslash on the end of a statement continues the line — no line feed is emitted when the text is emitted on the right side.

The next line has a single variable on the left, called `obj.has_parent`, which takes on the value True if the object has at least one parent object identified in the input model. When it is True, the statement is selected. It emits a colon and space, and continues the line.

The next statement is an iterator prefaced by `.each_parent`. It iterates over each parent identified for the current object in the model. The body of the iterator continues until its corresponding `.end` is reached. Within this parent iterator, each parent class name is emitted, followed by a comma, except with the last one. The variable `xxx.is_nth` is used to indicate the last member of an iterated list. The "bang" character (!) inverts the sense of the Boolean variable.

This segment of the template emits code that properly creates the preamble of a class declaration and takes into account cases where the class is a subclass of one or more parent classes.

Code for handling contained pointers

Listing Two contains the template module `emit_class_copy_ctor` that will emit the copy constructor for each object in the model. The pattern that the copy constructor follows demonstrates the need to treat contained pointers to owned objects differently from those that are shared.

The module `emit_class_copy_ctor` starts by emitting the copy constructor's signature and initialization list, including calls to copy constructors higher up the inheritance paths, if there are any. After emitting the opening brace of the constructor's body, the attributes that have been defined for the object are iterated, and three possibilities are used as selection criteria for producing code statements.

The attribute may be simple, such as "Name". In this case, the template assumes that the assignment operator is defined for the attribute type, and adds a statement to copy its associated value.

The attribute may be a pointer that is shared. In that case, the pointer value is copied.

In the last case, you have a pointer that is owned. A call is issued to make a copy of the object in that case, and the pointer of the newly allocated object is retained. This module is also responsible for emitting the body of the `makeCopy` function, which appears at its end. When this module is applied to `PetOwner`, you get the code in the class `PetOwner` shown in *Listing Four*.

The destructor pattern (see the module `emit_class_destructor` in *Listing Two*) iterates similarly over the object's attributes.

This time the iteration is done to find only the owned pointers. These are the only members that are interesting to the destructor (which is responsible for deleting them). The destructor for `PetOwner` has only one line, which deletes the `Pet` object, its sole owned pointer. *Example 2* is the code created for `PetOwner`'s destructor.

Example 2:

```
PetOwner::~~PetOwner()
{
    delete m_Pet ;
}
```

Conclusion

Cleanscape SourceMill allows control of code generation. It is flexible enough that you can define code patterns that relate to how objects fit into your local frameworks. Where a consistent strategy and pattern of producing code can be identified, quality and productivity can be increased proportional to the amount of code that follows that strategy.

The ratio of lines of code to lines of input model is typically 20:1. This is a great productivity advantage. Admittedly, Cleanscape SourceMill does not provide a revolutionary new way to produce code. Instead, it offers the opportunity to automate what you are already doing to follow code patterns by hand.

Filename: guide_sourcemill.doc
Directory: D:\My Documents\Products\Manuals
Template: D:\rational\SoDAWord\wizards\Normal.dot
Title: guide_sourcemill.doc
Subject: Cleanscape SourceMill User Guide
Author: Brent Duncan, Director
Keywords: computer, software, programming, tools, source code generators,
code generation, code synthesis, software process automation, brent duncan
Comments: Produced by:
Brent Duncan, Real Presentations
<http://www.realmarcom.com>
Creation Date: 3/27/01 5:14 PM
Change Number: 140
Last Saved On: 5/24/01 6:24 AM
Last Saved By: Brent Duncan
Total Editing Time: 2,835 Minutes
Last Printed On: 5/24/01 6:36 AM
As of Last Complete Printing
Number of Pages: 127
Number of Words: 22,272 (approx.)
Number of Characters: 122,499 (approx.)