Draft User Manual
OOram/SRE
Static Reverse Engineering
Trygve

1 April 1996

Chapter 2 contains a short user manual, sufficient if you know the OOram/SRE method. The remaining chapters should be read before your first reverse engineering operation.

PLEASE READ THE WHOLE REPORT BEFORE TRYING THE TOOL.

The development activity plan is as follows:

0. Project specification.
DONE
1. Prototype development.
DONE: Trygve, 82.5h
2. Evaluation of functionality and user interface
DONE: Anne, 8h. Witold, 4h. Trygve, 7h
3. Product programming and program documentation
DONE: Trygve.
* *Program extention, cleanup, optimization, documentation: 308*
Port from e33x to f15: 105
4. Program review (Jon Ola?)
5. Beta test, revision and submitting for product release (Trygve++)
6. User documentation (Klartekst??)

Client:
Client reference:
Taskon reference: OOram/SRE document 1 version 2
File: D:\work.t12\ooram.t12\oo-sre.t12\960401-sre-draftuserdoc.rm4
Last update: 1 Juni 1998 at: 11:40
Print generated: 1 Juni 1998 at: 11:40

TASKON
Work Environments

Chapter 1
Background and motivation

In the OOram methodology, we distinguish between forward and reverse engineering. In forward engineering, we start with a problem and describe its solution in a series of successively more elaborate models such as a System User model, a System Requirements model, a System Design model, and ,finally, a System Implementation. In reverse engineering, we start with an implementation and develop one or more System Design models that highlight important aspects of the implementation.

The OOram SRE (Smallralk Reverse Engineering) product supports the reverse engineering of Smalltalk programs. It has two main application areas:

1. *Recreating the design of old programs.* The first application area is in the maintenance of Smalltalk applications. The program may have been modified during its maintenance phase, or its design may not have been fully documented during its development phase. In both cases, it will be useful to document its design before embarking on major revisions.
2. *Cleanup after rapid prototyping.* The second main application area is as a tool in the system development phase. Smalltalk is a very expressive language. We frequently find it easier to determine the functionality and main algorithms through rapid prototyping than to follow the traditional waterfall process with specification, design, and implementation. We also find that the rapid prototyping approach yelds a better program from its users' point of view than the traditional approach.

   The first working prototype is not a product, however. It may be satisfactory as oberved from the outside, but there still remains developing a clean and robust design that covers rare cases and exceptions as well as the main functionality. The system also needs to be optimized, and the documentation to be completed. (Early versions of the documentation, particularly the user documentation, should be developed as part of the rapid prototyping process).

This document assumes that you are familiar with the OOram methodology, particularly with the role collaboration diagram with its ports, interfaces, and messages.

Chapter 2 contains a quick guide to the use of OOram/SRE. It should be sufficient if you already know the OOram/SRE method. The remaining chapters should be read before you try your first reverse engineering operation.

Chapter 2
Overview of the Static Reverse Engineering (SRE) method

OOram/SRE (Static Reverse Engineering) is a method for creating a role model collaboration view of an existing Smalltalk application.

Two Smalltalk images are involved in this method. (They could, as a special case, be the same image.)

1. The *SRE Image* is the image containing the role modeling tools with the SRE extensions.
2. The *Target Image* contains the application classes under study. This image can be any ObjectWorks\Smalltalk or VisualWorks image extended with an SREAnalyzer filein. It could even be identical to the SRE Image.

The following files are of interest:

1. The OOram/SRE programs. You find the most current alpha test release in */vermont/trygve/sre-3:*
   - *sre3.im2* is the SRE Image. This image is an extension of internal Taskon release f15.
   - *sre3.st* is the analysis filein for the Target Image. (Note: This filein includes the WSReadWriter. Use a ChangeList if you file into a Taskon image.)
2. The Target Image. This image is wherever you keep it. (It could be the SRE Image.)
3. The Target Image source code files. You will want to examine Target Image methods, so it should be set up with source code, and the source files should be accessible from the SRE Image.
4. A file for data communication between the two Images. Default file name is *'reveng.rev'*, but you will be invited to specify some other file name. Analysis specifications are written by the SRE Image and read by the Target Image. The Target Image augments the file with the results of the analysis, and the file is finally read by the SRE image to include the results in the design model(s).

The static reverse engineering method consists of 7 phases:

1. *Decide on the area of concern. (Manual.)*
   Even the simplest Smalltalk program is a fairly complex structure of objects that is linked directly or indirectly to every other object in the image. You must decide on the domain of the new role model. Examples are are an application model structure; specific editor-model relationships; the inner details of an editor; and a general mechanism that you want to study for the purposes of creating a new framework.

2. *Determine role model collaboration structure. (Manual.)*
   You must identify the objects that are within your domain and represent them in a role model. The roles and their collaboration structure are determined manually. A good technique is to interrupt the running application in the Target Image and use the *Smalltalk inspector* to explore the objects; choosing suitable roles and their structure.

   Record your model in an OOram Collaboration View (in the SRE Image). Give the roles meaningful names. Create ports for all message communication paths that may be relevant within the scope of your study. Use the port explanation to record the name and nature of the variable(s) represented by the port. Make sure to include all ports that may possibly be relevant; including ports that represent global variables, class variables, and computed values such as *ObjectMemory current.* The analysis will not find unspecified paths and will not look for messages sent from NONE-ports.

3. *Specify the names of the classes that implement the roles. (Manual.)*
   The collaboration view menu command *Static reverse engineering* opens the SRE tool. Hit the *Specify class names* button to open the SRE ClassName tool. You often specify several classes for one role; they may be independent or in a subclass-superclass relationship. Each class implements some of the functionality that you want to capture in the role.

   If you do not specify a relevant class, you may miss important messages. If you specify too many classes, your cleanup job in phase 7 will be unnecessarily tedious.

4. *Write an analysis specification file. (Automatic.)*
   Hit the *Write specification* button in the *SRE Class specifier tool*. This writes a binary file that contains the input data for the interface analysis. The SRE tool asks you for the name of the communication file, and writes the specification to this file.

5. *Perform analysis in Target Image (Automatic).*
   The SREAnalyzer module *(sre2.st)* must be filed into the Target Image prior to the analysis.

   The analysis is then done by *DoIt* the following expression: *RMAnModel1 analyzeFromFile*. The analyzer reads the specification from the file, performs the analysis, and overwrites the file with the combined specification and analysis results.

   For each specified port, the analyzer determines the messages possibly sent from the *From-Role* to the *To-Role*. The messages are determined as the intersection between two sets: the set of the selectors of all messages sent from the methods of the *From-Role* classes, and the set of the selectors of all methods implemented in the *To-Role* classes. The analyzer does not find messages sent indirectly such as with the *perform:*-command.

   (If the Target Image is identical to the SRE Image, you can press the *Analyze this image* button in the SRE Class Specifier Tool to trigger the analysis and record the results. The Class Specifier Tool will then close and the SRE Tool will go directly to phase 7.)

6. *Read the result file into the OOram/SRE image. (Automatic.)*
   Push the Read analysis button in the SRE tool to read the results of the analysis.

   The message interfaces determined by the analysis are recorded in the role model ports and can be inspected in the SRE tool. New messages are given an interface name corresponding to the Smalltalk method protocol. ***The new messages are kept in a compact form in the SRE part of the system. They are only visible in the SRE Tool, and are not visible in the regular OOram tools. (To make them visible, see next phase.)***

7. *Edit interfaces. (Manual.)* In this phase, you systematically examine every new message and determine if it is a regular message that is truly sent through the current port, or if it is a dummy message.

   If the message is a regular message, transform it into a normal OOram message by the one of the commands: *->Done*, *->Default*, *All->Done*, or *All->Default*.

   The automatically generated interfaces contain many dummy messages, i.e., messages that are implemented in a *To-Role* class and sent from a *From-Role* class, but that are not sent through the current port. A typical example is the message *initialize*, it is sent and implemented in most classes, but is typically only sent to *self*, not to a collaborator.

   If the message is a dummy message, move it to the list of dummy messages by one of the commands: *->Dummy* or *All->Dummy*. Alternatively, you could remove it, but it will then reappear when you repeat the analysis.

8. *Edit roles and collaboration structure:*
   - *Consider port cardinality.* Ports with no messages should be considered changed into NONE ports. Ports with multiple collaborators should be considerd changed into MANY ports.
   - *Consider path removal.* Paths with NONE ports at both ends can be removed.
   - *Edit explanations.* Explanations for roles and messages have been automatically extracted from class and method comments. They should be used as a starting point for commenting the roles in the context of the current role model. Edit them to make them describe important features without being overburdened with details. We typically include description of purpose and responsibility, but exclude details such as the types of parameters and variables.
   - *Consider separation of concern.* It may be appropriate to divide a complex model into a number of simpler ones. You can either make a number of copies the complex model and simplify them; or create simpler base models with the *inverse synthesis* operation. Let each base model cover part of the total scope. The total model can be derived from these base models, but this is often unnecessary if the separation of concern has been successful.
   - *Create an intelligent documentation*. A model dump such as the one shown in Appendix 1 is usually too boring to be really informative. (Try it!) Careful editing and commenting of the automatically generated report makes it more readable; but it also gives you a maintenance problem. Consider moving your comments into the model to simplify later maintenance.

9. *If your task is one of re-engineering, you will want to consider improvements in the extracted model. Some consideration:*

   - Consider to distribute the work of roles with a great many collaborators. A true object oriented system is distributed, not centralized.
   - Ports with very few messages could be considered for indirect access: Send the message to some other role that will forward it.
   - Look for unnecessary rigidity caused by objects knowing too much about the object structure, i.e., a method that navigates an object structure and centralizes all the work. Consider distributing the work to the objects rather than doing the work for them. This simplifies the object structure and prepares the ground for future extentions through specialization (subclassing).
   - Look for inherent inefficiencies in computation. Consider if hard-to-get information can be stored explicitly. It is usually beneficial to maintain direct pointers to important collaborators rather than executing elaborate navigational algorithms to find them. Hard-to-compute information can be often cached, but make sure you clear the cache whenever appropriate.
   - Look for inherent inefficiencies in storage. If the same information is stored repeatedly in many roles, you should consider delegating the storing to a common role. An exception is when you use caching to improve speed. You must them take great care to maintain consistency.

Chapter 3
Phase 1: Decide on the area of concern (Manual)

As an example, we will study the Text editing facilities of a Smalltalk program Browser in our current image (the SRE Image). We want to understand the Text editor specialization of the Model-View-Controller (MVC), but we do not include the basic MVC framework as such.

Chapter 4
Phase 2: Determine role model collaboration structure. (Manual.)

We interrupted the operation of a Browser, found the TextView object, and inspected its instance variables. We decided that the phenomenon under study was adequately described by the role structure of Figure 1??. This was the personal judgment of the analyst, another analyst could have chosen a different set of roles.

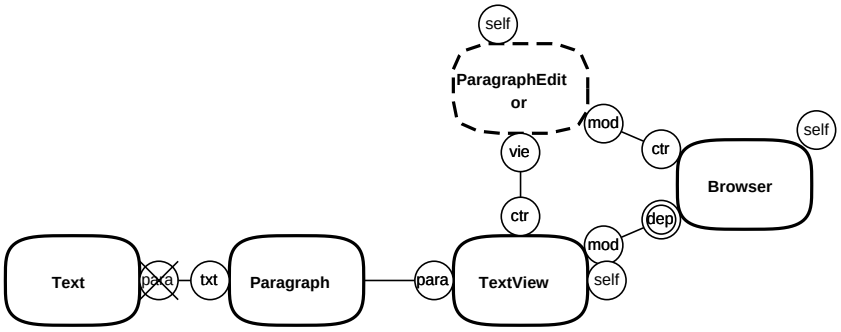


Figure 1. Text editing in the Browser

NOTE: A crossed-out port denotes a port with no messages associated with it. This is a special notation for the SRE that we find useful when cleaning up the model after analysis.

Notice that every object you find when inspecting your target object structure is a candidate to be represented as a separate role, even if these objects are instances of the same class.

You ignore the following kinds of objects:

1. objects that are outside the scope of your study,
2. objects that are adequately represented by an existing role,
3. objects that can be considered enclosed in some other role.

You may want to distinguish between the following kinds of objects:

1. *essential object.* The object is represented by a separate role in the role model.
2. *encapsulated object.* The object is considered to be hidden within a role in the context of the current area of concern
3. *set of objects.* A set of objects that are indistinguishable in the context of the current area of concern is represented by a single role that corresponds to one typical member of the set. An object set is typically accessed through a MANY-port. (The Collection object itself is then considered encapsulated within the From-Role.)

4. *irrelevant object.* An object that is considered totally outside the scope of the current area of concern. Irrelevant objects may be referenced from environment objects in the current role model.

A port can represent variables of many different kinds:

1. *Global variable.* The port will define the interface to a role representing the corresponding object.
2. *Class variable.* The port will define the interface to a role representing the corresponding object.
3. *Instance variable.* The port will define the interface to a role representing the corresponding object.
4. *Method.* The variable is computed by a method.
5. *Temporary*. The variable is a parameter or a temporary variable in a method. (Always include an abundance of ports in your initial model in order to capture messages sent through temporary variables. You can always remove them in the last phase when you are certain of their irrelevance.)

The OOram reverse engineering method prescribes that ALL relevant paths and messages shall be shown in the role model. E.g., the expression *a b c* should be considered expanded into its constituent parts:

t1 := a b.
t2 := t1 c.

We then consider the variables *a, t1,* and *t2*. The corresponding paths are either considered to be internal to the current role, or they must be shown as separate ports as illustrated in Figure 2??.

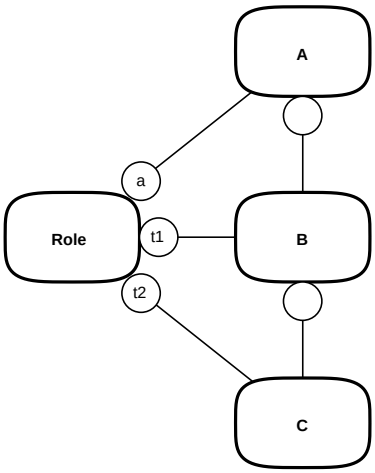


Figure 2. Role model for the expression a b c.

The *TextView* role represents an instance of the TextView class. Inspection of a running Browser revealed the following instance variables (the explanations have been copied from the class comments):

1. *container* <Wrapper | nil> the parent object of the receiver. (aScrollWrapper). Considered outside the scope of this study.
2. *model* <Object>. (aBrowser). Represented by port *m*.
3. *controller* <Controller>. Represented by port *c*.
4. *scrollOffset* <ScrollValueHolder> contains the current scrolling offset and the current scroll grid. Considered a role attribute, i.e., an object encapsulated within the TextView role.
5. *displayContents* <ComposedText> the visual component to use to show the text. Represented by port *dC*.
6. *startBlock* <CharacterBlock> the start selection. Considered a role attribute, i.e., an object encapsulated within the TextView role.
7. *stopBlock* <CharacterBlock> the stop selection. Considered a role attribute, i.e., an object encapsulated within the TextView role.
8. *selectionShowing* <Boolean> flag indicating whether the selection is ready to be displayed. Considered a role attribute, i.e., an object encapsulated within the TextView role.
9. *displaySelection* <Boolean> flag indicating whether selections will be displayed when ready. Considered a role attribute, i.e., an object encapsulated within the TextView role.
10. *partMsg* <Symbol | nil> the text aspect of the model. Inspected value was #textMenu. It is an important variable. It represents a message selector to be sent by the object through a *perform:*, it will NOT be found by the analysis algorithm.
11. *acceptMsg* <Symbol | nil> the message used to accept text. Inspected value was #textMenu. It is an important variable. It represents a message selector to be sent by the object through a *perform:*, it will NOT be found by the analysis algorithm.
12. *menuMsg* <Symbol | nil> the message to find a menu in the model. Inspected value was *#textMenu*. It is an important variable. It represents a message selector to be sent by the object through a *perform:*, it will NOT be found by the analysis algorithm.
13. *initialSelectionString* <nil | String | Text> nil or a string/text whose first occurrence will be initially selected.

Figure 3?? shows the OOram Collaboration tool. Note the explanation for the *ctr* port from *Browser* to *Controller*. The explanation shows that this is a call-back passed as parameter to the Browser. (The existence and nature of this port could only be determined in phase 7 when we studied the methods sending messages through the port.)

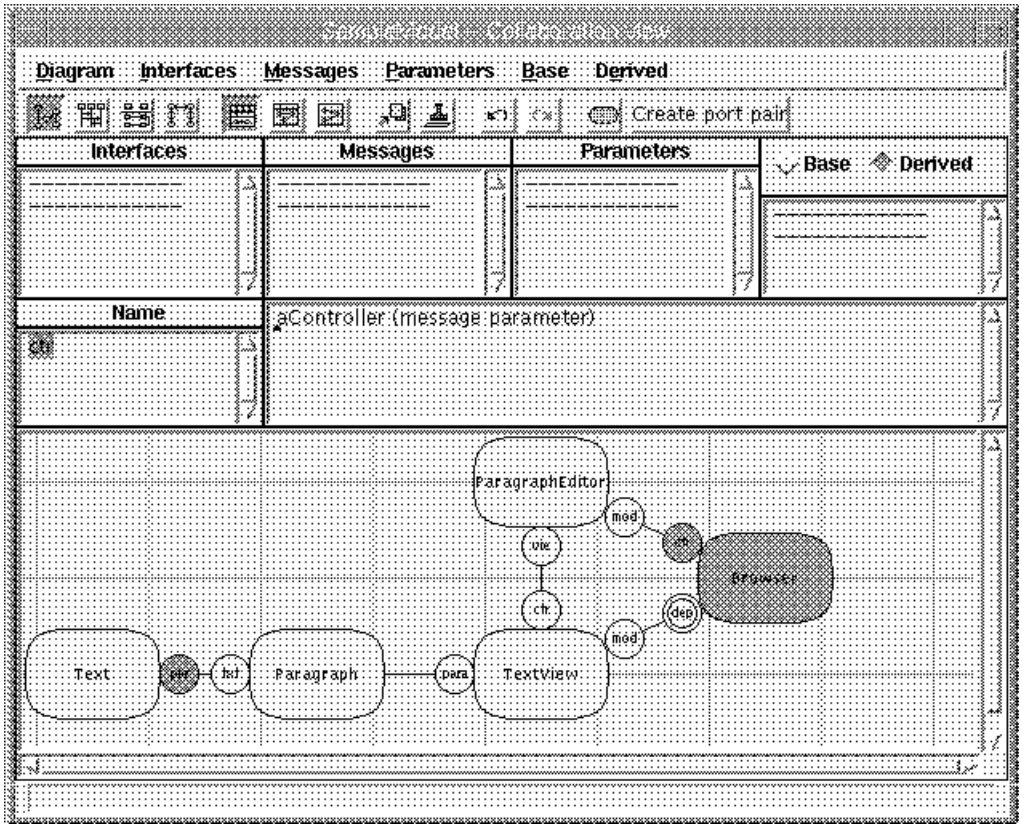


Figure 3. Collaboration tool illustrating a port representing a temporary variable.

Chapter 5
Phase 3: Specify the names of the classes that implement the roles (Manual)

Give the *Static Reverse Engineering* command in the collaboration diagram to open the SRE tool shown in Figure 4??.

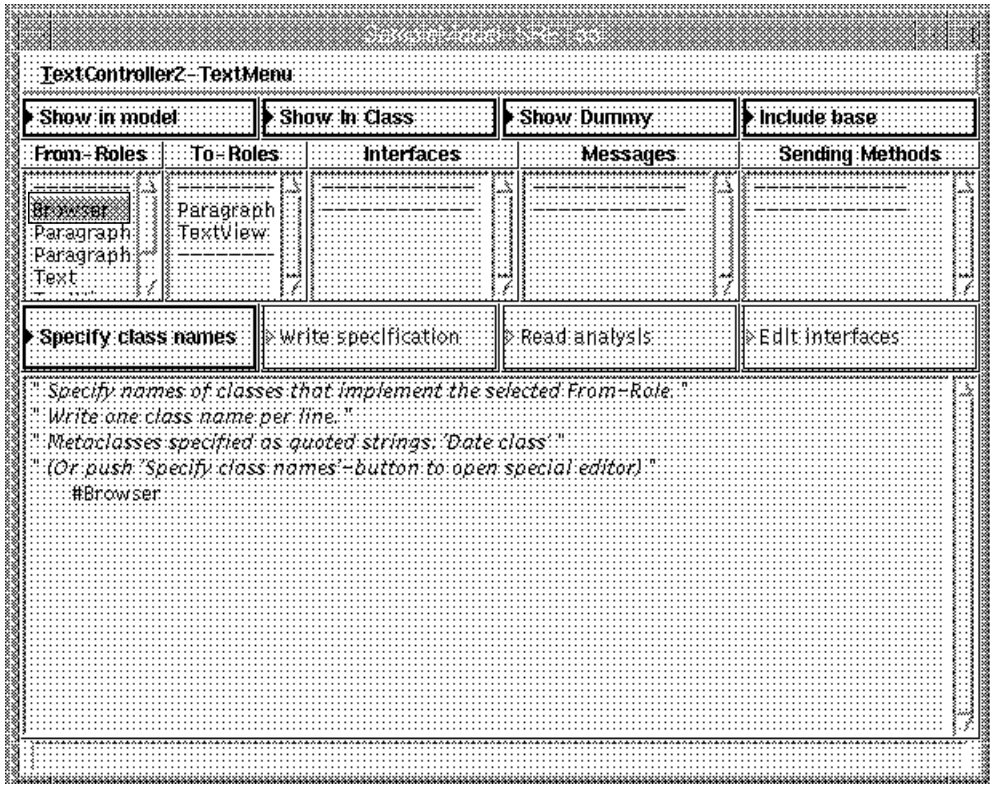


Figure 4. SRE tool

The obvious class name to associate with a role is the class of the instance it represents as found in the inspection of phase 2. Also consider if you want to include super- or sub-classes. If the role can be played by different objects, consider the classes of all these objects in the same manner. You will often specify the same classes for different roles. This is normal, and means that instances of these classes (or their subclasses) play different roles.

There are two alternatives for specifying the class names. One is to select a From-Role in the ReverseEngineering tool while the phase is *Specify class names.* The class names can then be edited in the bottom (Text) view. Acceptance is automatic when you leave the text editor. You will be invited to modify the class name specification if it is syntactically incorrect.

An alternative is to use a special tool. Press the button marked *Specify class names* in the SRE tool. This opens the *SRE class specifier* tool as shown in Figure 5??.

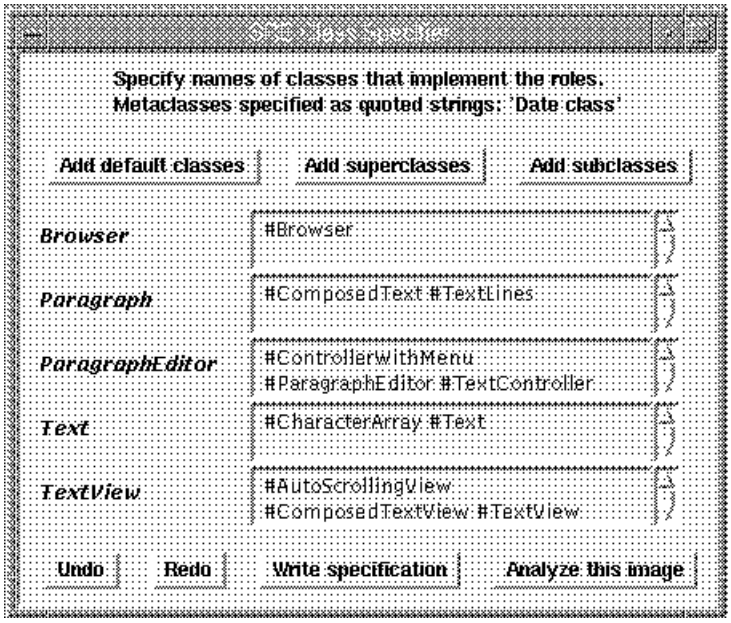


Figure 5. SRE class specifier tool

The class names are written in the fields provided. The specified class names are stored in the corresponding role whenever a field has been edited.

1. The *Undo*- and *Redo* buttons work in the usual manner.
2. The *Write specification* button will write a file containing the analysis specification, close the tool, and transfer to the next phase.
3. The *Analyze this image* button will perform the analysis on the classes in the current image, close the tool, and transfer to the last, *Edit interfaces*, phase.
4. If you close the window in any other manner, the specified class names will be remembered, but the work phase will remain unaltered. Push the *Specify class names* button in the SRE tool to reopen this *SRE class specifier* tool.

Chapter 6
Phase 4: Write an analysis specification file (Automatic)

The *Write specification* button in Figure 5?? will close the tool and transfer to the next phase. Alternatively, you can push the *Write specification* button in the SRE tool as shown in figure 6??.

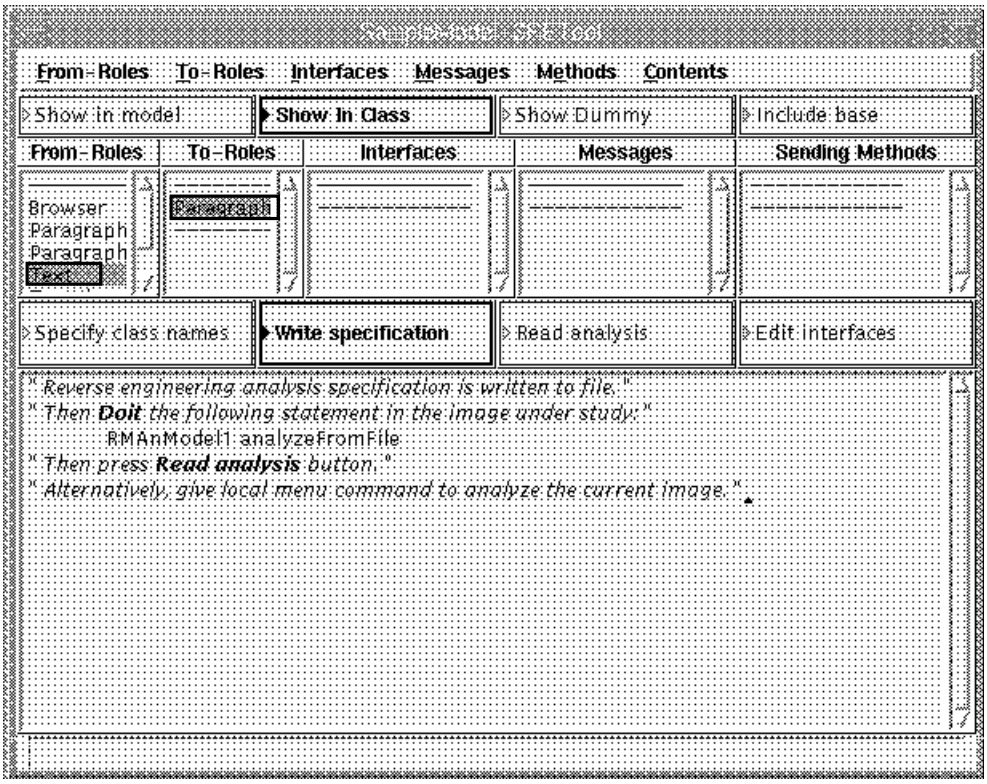


Figure 6. Write specification phase in the SRE tool

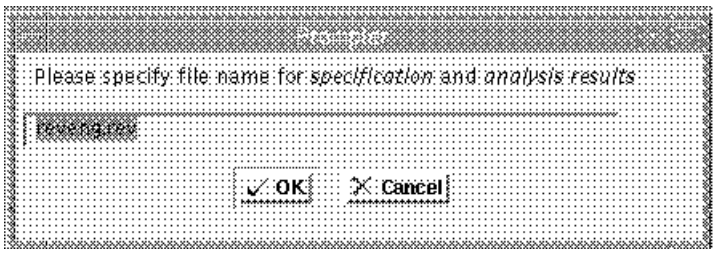


Figure 7. Specify directory for communication between SRE Image and Target Image

The tool asks the user for the communication file name as shown in figure 7??, and then writes the specification to this file. Notice that the file must be written to a directory that is available in both the SRE and the Target images.

Chapter 7
Phase 5: Perform analysis in target image (Automatic)

If the Target Image is a different image, you must ensure that the SREAnalyzer module has been filed in. The filein is in the SRE directory, it is called *sre3.st*.

The analysis is done by *Do it* the following expression: *RMAnModel1 analyzeFromFile* in the Target Image. The analyzer reads the specification file (*reveng.rev*), performs the analysis, and rewrites the same file with the combined specification and result.

If the Target Image is the same as the SRE Image, you can push the text editor command *Analyze this image* to perform the analysis and read the result of the analysis in a single operation. The SRE tool then automatically transfer to the *Edit results* phase.

Chapter 8
Phase 6: Read the result file into the SRE image (Automatic)

Push the Read analysis button in the SRE tool to read the results of the analysis.

The message interfaces found in the class analysis are recorded in the role model ports:

1. Messages existing in the role model and found in the analysis are marked as being *'in class'* and otherwise left unchanged.
2. Messages existing in the role model and not found in the analysis are marked as being *'not in class'* and otherwise left unchanged.
3. New message selectors are kept in the Port.
4. Old dummy selectors not found in the analysis are removed. (See next chapter.)

The SRE phase is then automatically moved to the last phase: *Edit interfaces.*

Chapter 9
Phase 7: Edit interfaces (Manual)

An SRE tool is shown in figure 8?? as it appears in the final phase. It is organized in four rows:

Menu bar The menu bar contains menus for the five lists and the text view:

From-Roles *Menu for the first list view*
Read reverse engineering results ***Unnecessary***?
Undo
Redo
Remove all empty interfaces ***in whole role model***
Remove all empty paths ***in whole role model***
Move all own messages to default interface ***in selected Port only***

To-Roles *Menu for the second list view*
Read reverse engineering results ***Unnecessary***?
Undo
Redo

Interfaces *Menu for the Interface list*
Add interface
Remove interface
All -> Default ***change state all messages in selected interface***
All -> Done ***change state all messages in selected interface***
All -> Dummy ***change state all messages in selected interface***
Find message ***select from all messages in all interfaces***
Undo
Redo

Messages *Menu for the message list*
->Default ***change state selected message***
->Done ***transform selected message into regular message***
->Dummy ***change state selected message***
Reset Status ***change state selected message***
Undo
Redo
Remove message

Methods *Menu for the Methods list*
toggle to senders ***toggle contents of method list***
toggle to receivers ***toggle contents of method list***

Contents *Menu for the bottom, text view. The view is read-only, so there is only one active command:*
Copy

Filter row Four toggle buttons for filtering interfaces and messages on certain properties:

- *Show In Model* — Messages defined in the role model in the normal way are shown if the button is ON, suppressed if it is OFF.
- *Show In Class* — Messages found during Target Image analysis are shown if the button is ON, suppressed if it is OFF.
- *Show Dummy* — Messages moved to *dummy* are shown if the button is ON, suppressed if it is OFF.
- *Include Base* — Base model messages are included if the button is ON, suppressed if it is OFF.

List row A browser-like sequence of lists for navigating in the model:

- *From-Roles* — List of all roles in the role model. Select one to study the messages it sends to its collaborators. (Showing roles with non-empty interfaces after filtering only.)
- *To-Roles* — List of all the selected From-Role's collaborator roles. Select one to study the interfaces. This is equivalent to selecting a port in the collaboration view. (Showing ports with non-empty interfaces after filtering only.)
- *Interfaces* — List of interfaces for messages specified from *From-Role* to *To-Role*. Automatically generated interfaces are named as one of the corresponding Smalltalk protocol names. The list is filtered by the current setting of the filter buttons. (Showing non-empty interfaces after filtering only.)
- *Messages* — List of messages in the selected interface. The list is filtered by the current setting of the filter buttons.
- *Method list* — The user can toggle between two lists in the rightmost list view:
  - *Sending Methods* — The methods in the From-Role classes that send the selected message.
  - *Receiving Methods* — The methods in the To-Role classes that implement the selected message.

Text view This view is a text view that either shows the class names associated with the selected From-Role, or the method selected in the rightmost list view. The selected message is highlighted as **bold** text in the method code.

The presented method was extracted during analysis in the Target Image and the displayed source is read from the Target Image source files. The Text view is therefore read-only, the usual Browsing commands such as *senders*, *implementors*, and *messages* are not available.

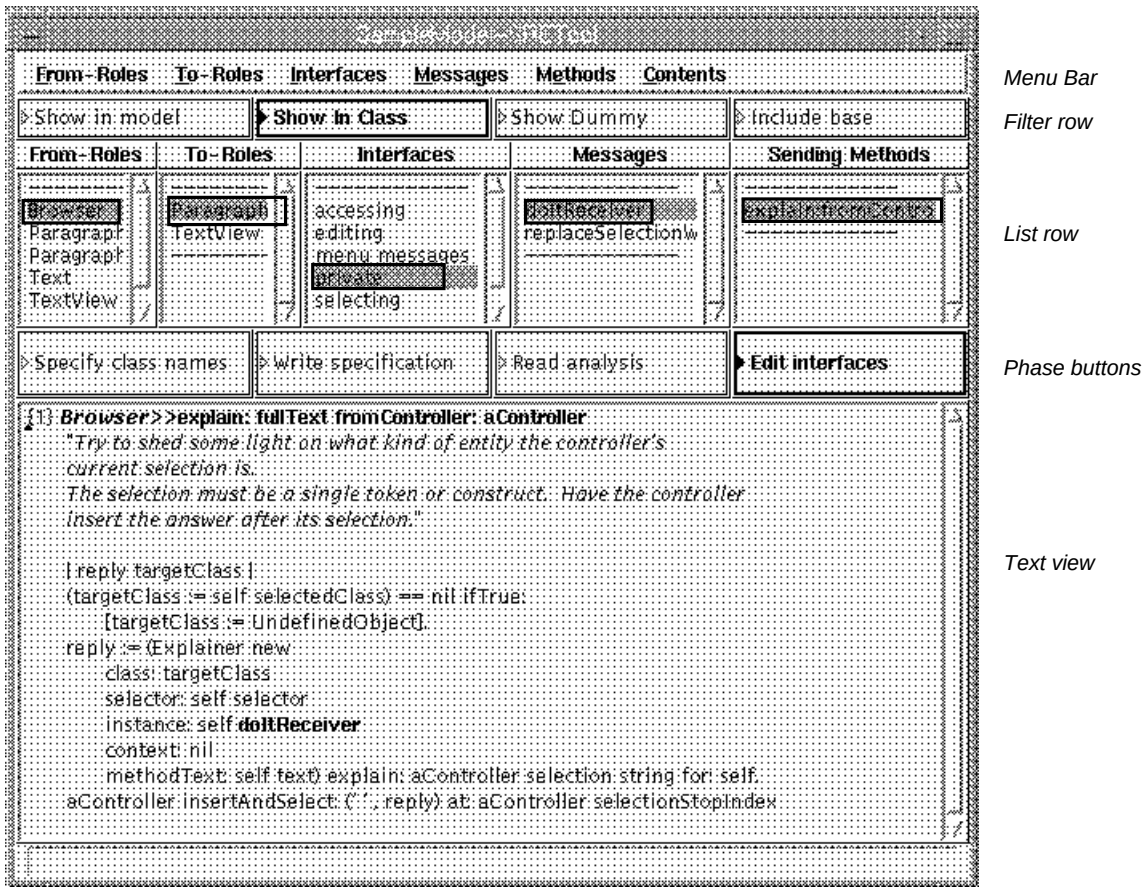


Figure 8. Browsing message senders, finding a dummy

Figure 8?? shows a typical example of a dummy message: The *doitReceiver* message is sent from a Browser method and is implemented by a Controller class. Inspecting the method, we find that the message is actually sent to *self.* We therefore give the *->Dummy Interface* command in the Messages list menu to move the message out of the way. (The filter setting is typical for the cleanup operation: *Show In Model, Show Dummy,* and *Include Base* are OFF, *Show In Class* is ON. The lists then only show messages that have not yet been considered.)

Notice that a message is a dummy message only if there is no method that sends it to the To-Role.

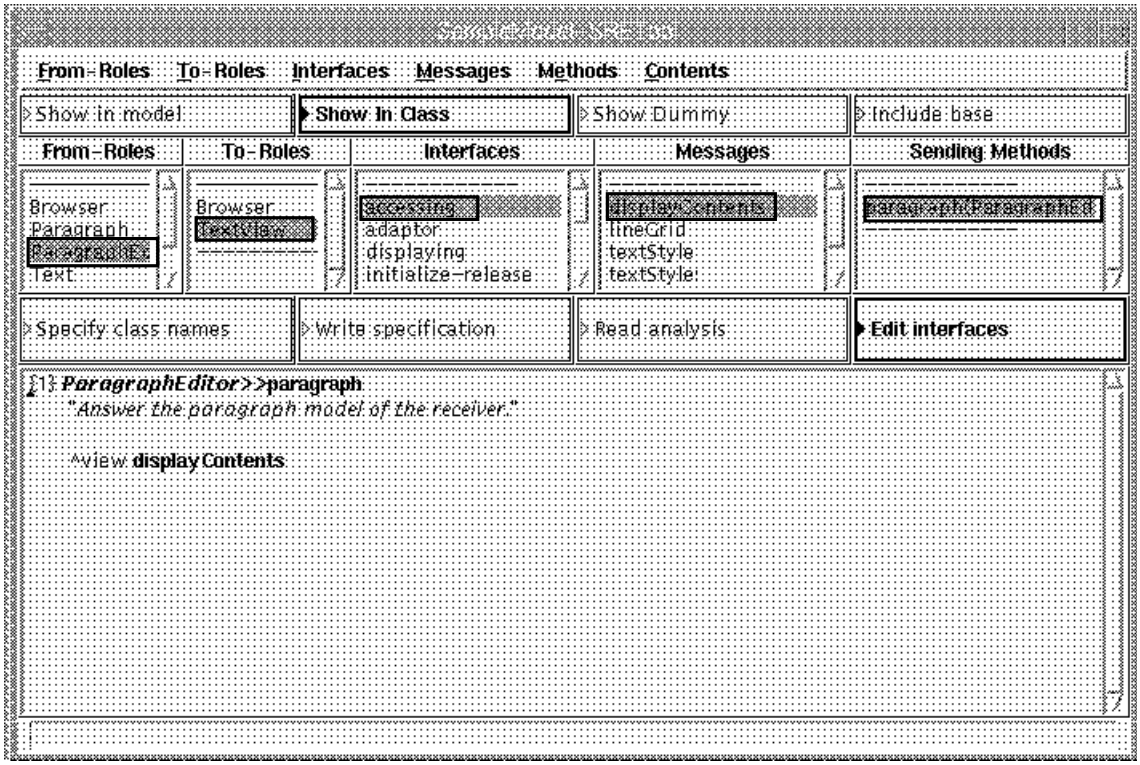


Figure 9. Browsing message senders, finding a proper message

The example shown in figure 9?? is a typical, proper message. We see that *ParagraphEditor>>paragraph* sends the message *displayContents* to the *view* instance variable, that is the variable represented by the *TextView<Controller* port. By convention, we use the Message list command *->Default interface* to change its interface to the default interface.

Notice that a message is a proper message if at least one of the methods send it to the To-Role.

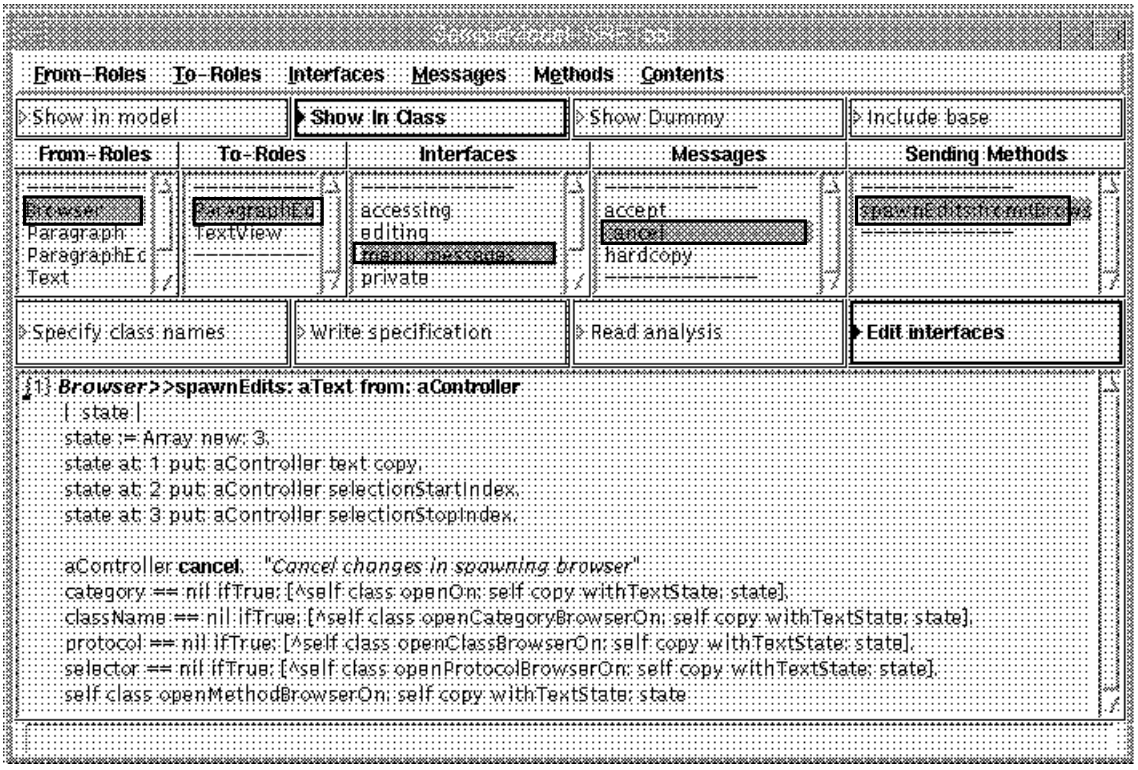


Figure 10. Browsing message senders, finding a call-back message

In the the usual MVC framework, the Model does not know the Controller. In the example of figure 10??, however, the *Browser* (*Model*) gets reference to the *Controller* in a message parameter and uses this knowledge to return messages to the Controller. *There must therefore be a port from the Browser to the Controller, even if its life span is limited to a single method invocation.*

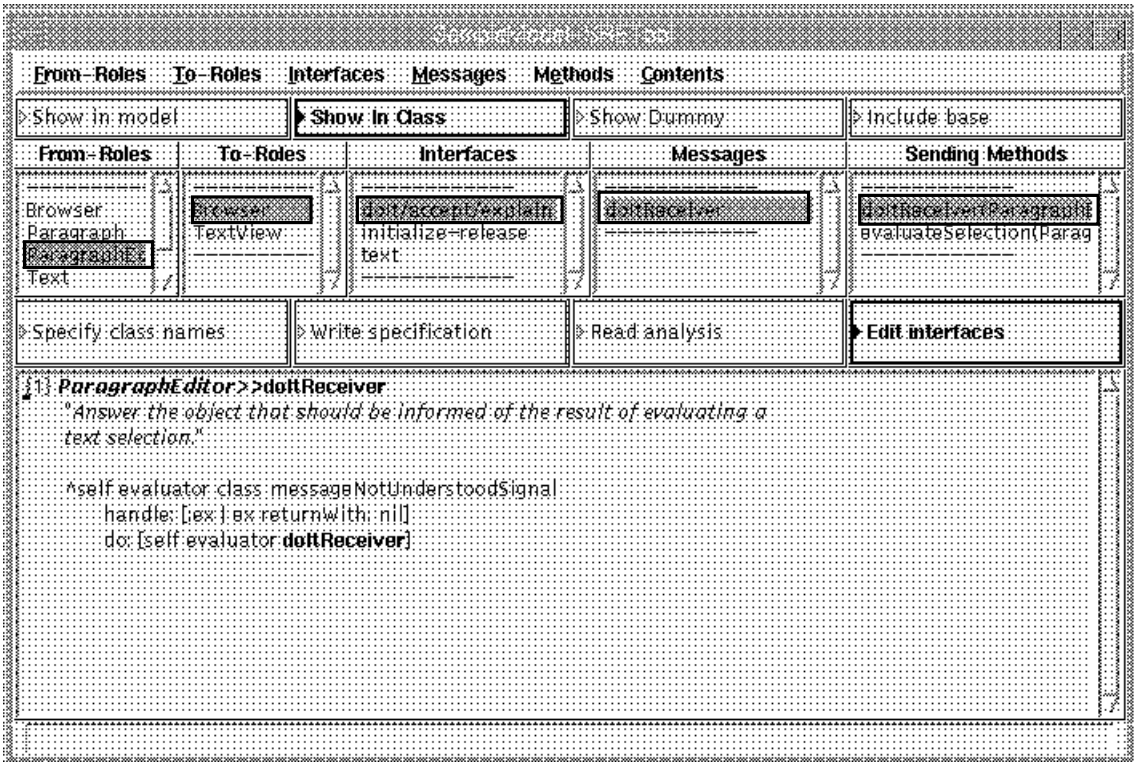


Figure 11. Browsing message senders, an obscure case

Figure 11?? represents a somewhat obscure case. The method *ParagraphEditor>>doItReceiver* sends the message *doItReceiver* to *self evaluator*; but is this one of our roles or is it to be suppressed in our role model? We have to revert to the Target Image and investigate in depth.

An easy way out is to interrupt the execution of a Browser and open an inspector on the Controller. We find that *self evaluator* evaluates to *a Browser1*, the *doItReceiver* message is therefore sent to the *Browser* and is a proper message in this port.

As a check, we browse the relevant methods and find that the collaboration structure is not always evident from the code:

ParagraphEditor>>evaluator (private)
evaluator
"*Answer the object that defines the context of an evaluation*"

^self performer == self
ifTrue: [self model]
ifFalse: [self performer]

ControllerWithMenu>>performer (accessing)
performer
"*Answer the receiver of menu messages.*"

^performer

Who sets this value and when? We find that it is set to *self* in the *initialize* method and that it is modified by the *performer:* method. An interesting sender is

TextController>>model: (model access)
model: aModel

```
super model: aModel.
self performer: aModel
```

So the *evaluator* is the *performer*, the *performer* is the *model*, the *model* is the Browser role. So the *doItReceiver* message is sent to the *Browser*.

Chapter 10
Phase 8: Edit roles and collaboration structure

The goal of this phase is to manipulate the model in order to make it easier to understand. The following hints may be useful.

1. *Consider port cardinality.* Ports with no messages should be considered changed into NONE ports. Ports with multiple collaborators should be considerd changed into MANY ports.
2. *Consider path removal.* Paths with NONE ports at both ends can be removed.
3. *Edit explanations.* Explanations for roles and messages have been extracted from class and method comments. They should be used as a starting point for commenting the roles in the context of the current role model. Edit them to make them describe important features without being overburdened with details. We typically include description of purpose and responsibility, but exclude details such as the types of parameters and variables.
4. *Consider separation of concern.* It may be appropriate to divide a complex model into a number of simpler ones. You can either make a number of copies the complex model and simplify them; or create simpler base models with the *inverse synthesis* operation. Let each base model cover part of the total scope. The total model can be derived from these base models, but this is often unnecessary if the separation of concern has been successful.

   In our text editor example, we could have separated out the *TextView-Paragraph-Text* part in a separate base model. We could also have created a general MVC base model, and made the *Browser-TextView-ParagraphEditor* into a derived model.
5. *Create an intelligent documentation.* A model dump such as the one shown in Appendix 1 is usually too boring to be really informative. (Try it!) Careful editing and commenting of the automatically generated report makes it more readable; but it also gives you a maintenance problem. Consider moving your comments into the model to simplify later maintenance.

If your task is one of re-engineering, you will want to consider improvements in the extracted model. Some consideration:

1. Consider to distribute the work of roles with a great many collaborators. A true object oriented system is distributed, not centralized.
2. Ports with very few messages could be considered for indirect access: Send the message to some other role that will forward it.

3. Look for unnecessary rigidity caused by objects knowing too much about the object structure, i.e., a method that navigates an object structure and centralizes all the work. Consider distributing the work to the objects rather than doing the work for them. This simplifies the object structure and prepares the ground for future extentions through specialization (subclassing).
4. Look for inherent inefficiencies in computation. Consider if hard-to-get information can be stored explicitly. It is usually beneficial to maintain direct pointers to important collaborators rather than executing elaborate navigational algorithms to find them. Hard-to-compute information can be often cached, but make sure you clear the cache whenever appropriate.
5. Look for inherent inefficiencies in storage. If the same information is stored repeatedly in many roles, you should consider delegating the storing to a common role. An exception is when you use caching to improve speed. You must them take great care to maintain consistency.

Chapter 11
Discussion

A detailed report of the sample role model is given in Appendix 1.

We see that there is no interface from *Browser* to *TextView*, this is as expected since we did not include the MVC classes in the analysis (*View* and *Object + Model*)

The analysis did not find an interface from *TextView* to *Browser.* This is bad, since there are clearly relevant messages in this interface. The TextView is a pluggable, and these essential message selectors are store in instance variables and sent by *perform:*. Returning to the TextView object inspected in the Target Image, we found that the interesting messages were *#text, #acceptText:from:,* and *#textMenu.* We could have found these messages by studying the class comment, see Chapter 4.

We see that there is a rich interface from *TextView* to *Paragraph.* Closer study may offer opportunities for simplification. We also see that there is an even richer interface from *ParagraphEditor* to *TextView.* These complex interfaces are strong indicators that the distribution of attributes and functionality between the ParagraphEditor, TextView and Paragraph objects should be reconsidered.

Appendix 1
Sample Role Model

App 1.1 Area of Concern

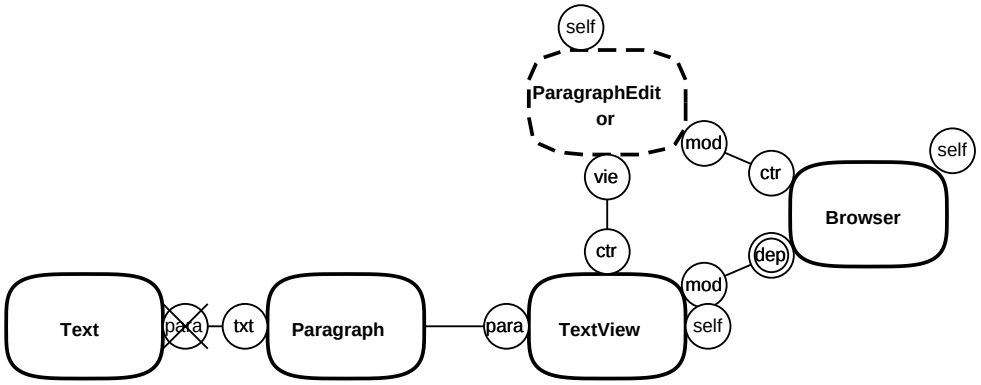
As an example, we will study the Text editing facilities of a Smalltalk program Browser in our current image (the SRE Image). We want to understand the Text editor specialization of the Model-View-Controller (MVC), but we do not include the basic MVC framework as such.

Stimulus messages

1. ***ParagraphEditor***

   - **accept** {Message}
   - **again** {Message}
     Text substitution. If a shift key is down, the substitution is made throughout the entire Paragraph. Otherwise, only the next possible substitution is made.
   - **againConfirm** {Message}
     Text substitution. If the left shift key is down, the substitution is made throughout the entire Paragraph. Otherwise, only the next possible substitution is made. Prompt to confirm substitution.
     5 May 1989 trygve: <<<60>>>
     29 December 1990 trygve st4.0
     3 April 1992 trygve st4.1.
     950407 als (vw): leftShiftDown -> shiftDown
   - **cancel** {Message}
   - **copySelection** {Message}
     Copy the current selection and store it in the shared buffer.
   - **cut** {Message}
   - **find** {Message}
     Open the find dialog. If the user picks Cancel, then do nothing.
     Otherwise, search for the next occurrence of the text from the dialog.
   - **paste** {Message}
     Paste the text from the shared buffer over the current selection and redisplay if necessary. If the shift key is down, present the user with a menu of the last few selections.
   - **replace** {Message}
     Open the find/replace dialog. If the user picks Cancel, then do nothing.
     If the user picks Find (default), then so a search.
     If the user picks FindAndReplace, then do a search and if found replace.
   - **undo** {Message}
     Reset the state of the paragraph prior to the previous cut or paste edit.

App 1.2 The Roles



1. **Text** Class Text handles protocol for treating strings of characters as displayable characters that can have emphasis and font changes. The emphasis codes indicate abstract changes in character appearance. Actual display is performed in the presence of a TextAttributes which indicates, for each abstract code, an actual font to be used.
2. **Paragraph** TextLines is the abstract superclass for classes that display lines of stylized text.

   Class ComposedText provides the support for creating, modifying and displaying stylized text.
3. **TextView** A ComposedTextView represents a Text. The Text should not be changed by any editing unless the user issues the accept command. Thus an instances provides a working copy of the Text. This copy is edited. When the user issues the accept command, the Text is copied from the working version; or if the user issues the cancel command, the working version is restored from the Text.

   TextView is a "pluggable" view of text. The notion of pluggable views is to provide a view that can be plugged onto any object, rather than having to define a new subclass specific to every kind of object that needs to be viewed. The chief mechanism is a set of selectors, which can be thought of as an adaptor to convert the generic TextView operations into model-specific operations (such as textMenu). See the protocol 'adaptor' for use of the pluggable selectors. See the creation messages in the class for an explication of the various parameters. Browse senders of the creation messages in the class for examples in the system. It also includes support for setting the initial selection to a part of the text.
4. **ParagraphEditor** Class ParagraphEditor contains the main handling of text editing. It ought to be used only on smallish texts.

   As a subclass of the ParagraphEditor, class TextController uses Menus for model control protocol. It also makes a stab (localMenuItem:) at the problem of distinguishing user control directed at the model from that directed at the view/controller. This controller is more tightly connected to its view (for example it asks it for its yellowButtonMenu), because the view is the current site for "pluggable" parameterization.
5. **Browser** A browser represents a hierarchical query path through an organization of class and method information. A full path query identifies a Smalltalk method to be examined.

App 1.3 Interaction Scenarios

App 1.3.1 Scenario Accept Method

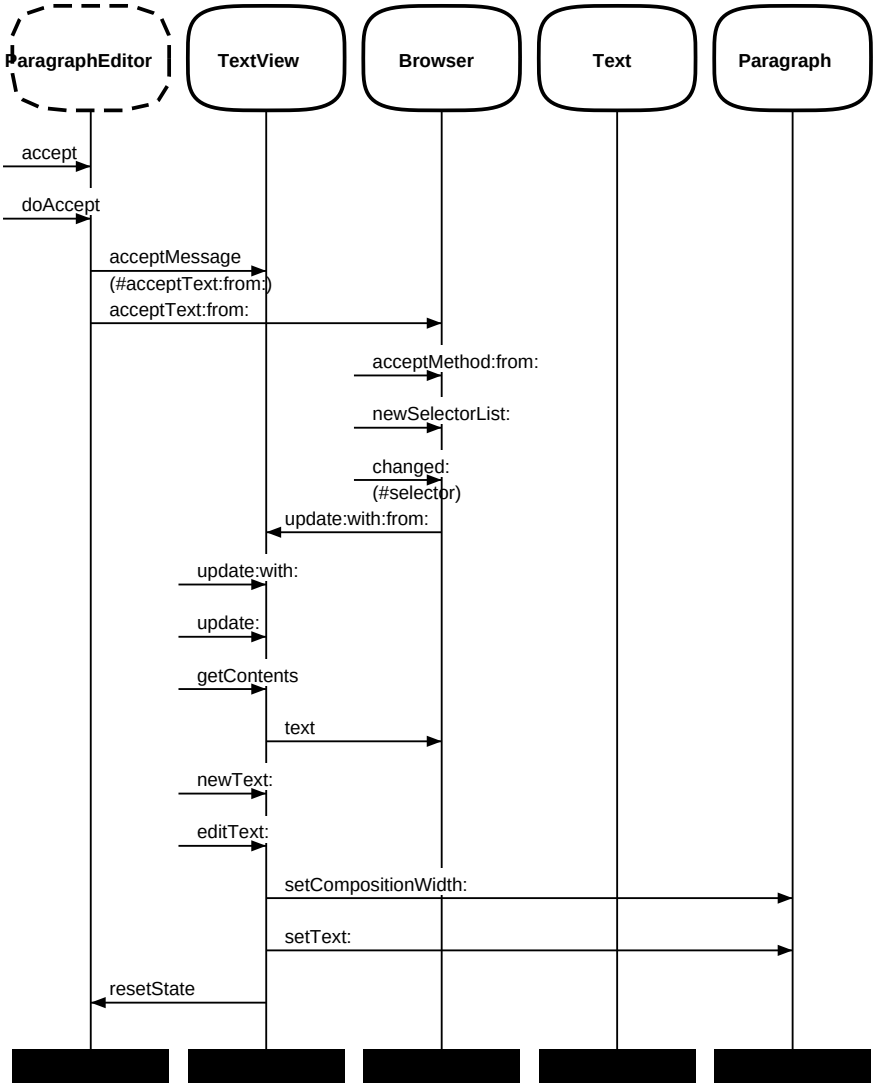


Figure 12. Accept Method {Scenario}

There are different accept activities for the different kinds of text handled by the Browser. This scenario shows a typical example of accepting a method. The new source code is given to the *Browser* where it is compiled. *update*-messages are sent to the dependents, the Browser's *TextView*. The *TextView* tells its *Controller* that *textHasChanged:*. A somewhat indirect way of telling the activity originator something it really knew all the time.

ParagraphEditor>>accept (menu messages)
accept
"*Save the current text of the text being edited as the current acceptable version for purposes of canceling.*"

```
self textHasChanged: false.
model value: self text copy
```

TextController>>accept (menu messages)
accept

```
(self textHasChanged and: [model changeRequestFrom: view])
    ifFalse: [^view flash].
self doAccept
    ifTrue: [self textHasChanged: false]
    ifFalse: [view flash]
```

TextController>>doAccept (private)
doAccept
"*Send the accept message to the model. Answer what the model answers.*"

```
| acceptMsg |
acceptMsg := view acceptMessage.
acceptMsg == nil ifTrue: [view flash. ^ false].
^ acceptMsg numArgs = 1
    ifTrue:  "one arg selectors get text only"
        [model perform: acceptMsg with: self text]
    ifFalse:  "two arg selectors get text and controller as well"
        [model perform: acceptMsg with: self text with: self]
```

TextView>>acceptMessage (adaptor)
acceptMessage
"*Answer the message sent to the model on accept.*"

```
^acceptMsg
```

Browser>>acceptText:from: (doIt/accept/explain)
acceptText: aText from: aController
"*Text has been changed. Store or compile the text, depending on the current mode of the receiver.*"

```
textMode == #classDefinition ifTrue:
    [^ self acceptClass: aText from: aController].
textMode == #methodDefinition ifTrue:
    [^ self acceptMethod: aText from: aController].
textMode == #categories ifTrue:
    [Cursor wait showWhile:
        [organization changeFromString: aText string.
        self newCategoryList: category].
    ^true].
textMode == #protocols ifTrue:
    [self selectedClass organization changeFromString: aText string.
    self selectedClass reorganize.
    self selectedClass logOrganizationChange.
    self textMode: #protocol; newProtocolList: nil.
    ^ true].
textMode == #comment ifTrue:
    [self nonMetaClass comment: aText string.
    self textMode: #comment; newProtocolList: nil.
    ^ true].
^ false
```

```
Browser>>acceptMethod:from: (private-selector functions)
acceptMethod: aText from: aController
    | newSelector |
    newSelector := self selectedClass
                compile: aText
                classified: protocol
                notifying: aController.
    newSelector == nil ifTrue: [^false].
    newSelector == selector
        ifFalse: [self newSelectorList: newSelector].
    ^true

Browser>>newSelectorList: (selector list)
newSelectorList: initialSelection
    "Set the currently selected message selector to be initialSelection."

    selector := initialSelection.
    self changed: #selector

ComposedTextView>>update: (updating)
update: aSymbol
    "The receiver's model has changed its text.  Update the receiver."

    self updateDisplayContents

TextView>>update: (updating)
update: aSymbol
    "If aSymbol = partMsg then read the text to edit from the receiver's model
    and update the display."

    | text |
    aSymbol == partMsg
        ifTrue:
            [text := self getContents.
            displayContents text ~= text
                ifTrue:
                    [self newText: text.
                    self initializeSelection.
                    "Translate so the selection is visible"
                    self positionToSelection.
                    selectionShowing := true.
                    self invalidate]
                ifFalse:
                    [self resetController]]

ComposedTextView>>getContents (model access)
getContents
    "Answer the text that the receiver is viewing."

    ^model value

TextView>>getContents (model access)
getContents

    | text |
    partMsg == nil ifTrue: [^Text new].
    text := model perform: partMsg.
    text == nil ifTrue: [^Text new].
    ^ text
```

```
Browser>>text (text)
text
    | text |
    textMode == #classDefinition ifTrue:
        [className == nil
            ifTrue: [^ (Class template: category) asText]
            ifFalse: [^ self selectedClass definition asText]].
    textMode == #methodDefinition ifTrue:
        [selector == nil
            ifTrue: [^ self selectedClass sourceCodeTemplate asText]
            ifFalse: [^ (self selectedClass sourceCodeAt: selector) asText
                    makeSelectorBoldIn: self selectedClass]].
    textMode == #category ifTrue:
        [^ 'category to add' asText].
    textMode == #categories ifTrue:
        [^ organization printString asText].
    textMode == #protocol ifTrue:
        [^ 'protocol to add' asText].
    textMode == #protocols ifTrue:
        [^ self selectedClass organization printString asText].
    textMode == #comment ifTrue:
        [text := self selectedClass comment asText.
        text isEmpty ifFalse: [^ text].
        self selectedClass isMeta ifTrue: [^'Select the browser switch "instance" to see the comment'
asText].
        ^ self selectedClass commentTemplateString asText].
    textMode == #hierarchy ifTrue:
        [^ self selectedClass printHierarchy asText].
    ^ Text new

ComposedTextView>>newText: (model access)
newText: aText
    "Set aText to be the text model for the receiver."

    aText == nil ifTrue: [^ self newText: Text new].
    self editText: aText

ComposedTextView>>editText: (model access)
editText: aText
    "The paragraph to be displayed is created from the characters in aString."

    self isOpen
        ifTrue: [displayContents setCompositionWidth: self wrappingBox width.
                displayContents text: aText asText copy]
        ifFalse: [displayContents setCompositionWidth: ComposedText defaultCompositionWidth.
                displayContents setText: aText asText copy].
    self setToTop.
    startBlock := stopBlock := nil.
    selectionShowing := true.
    self controller. "Make sure we have a controller"
    self resetController.
    self changedPreferredBounds: nil

ComposedText>>setCompositionWidth: (display box accessing)
setCompositionWidth: aSmallInteger
    "Set the width for composition to be aSmallInteger."

    "Make sure the composition width in not negative.
    The CompositionScanner requires this."

    compositionWidth := aSmallInteger max: 0
```

ComposedText>>setText: (accessing)
setText: aText
"*Set the text without recomposing. Invalidate the compositionHeight and lineTable so that recomposition happens later*"

text := aText.
lineTable := compositionHeight := nil.

ParagraphEditor>>resetState (initialize-release)
resetState
"*Establish the initial conditions for editing the paragraph: set the emphasis to that of the first character.*"

self resetTypein.
emphasisHere := self text emphasisAt: 1.
self textHasChanged: false

App 1.4 Implementation classes

NOTE:
We have deleted most of the code report, it is not interesting in the context of the SRE User manual.

App 1.4.1 Browser implementation

Model subclass: #Browser
instanceVariableNames: 'organization category className meta protocol selector textMode '
classVariableNames: 'CategoryMenu ClassMenu LastProtocol MessageMenu MethodMoveProtocol ProtocolMenu RemoveChangesOnFileOut TextMenu '
poolDictionaries: ''
category: 'TM-Browser+'

A browser represents a hierarchical query path through an organization of class and method information. A full path query identifies a Smalltalk method to be examined.

Instance Variables:
organization <SystemOrganizer>
category <String> a selection from all categories in the organization.
className <Symbol> a selection from all classes in the category.
meta <Boolean> false for viewing instance methods, true for class methods.
protocol <Symbol> a selection from all protocols in the class.
selector <Symbol> a selection from all messages in the protocol.
textMode <Symbol> indicating the nature of the currently viewed text

Class Variables:
CategoryMenu <Menu | nil> of operations on the class categories
ClassMenu <Menu | nil> of operations and queries about a class
LastProtocol <Symbol | nil> last message category selected

MessageMenu <Menu | nil> of operations on the message categories
MethodMoveProtocol <CharacterArray | nil> last message category used in a move
ProtocolMenu <Menu | nil> of operations and queries about a message
RemoveChangesOnFileOut <Boolean> flags whether to remove classes from the ChangeSet when they are filed out
TextMenu <Menu | nil> of text editing operations

See above for method Browser>>#acceptMethod:from:

See above for method Browser>>#acceptText:from:

Browser>>doItReceiver (doIt/accept/explain)
doItReceiver
"*Answer the object that should be informed of the result of evaluating a text selection.*"

^ self nonMetaClass

See above for method Browser>>#newSelectorList:

See above for method Browser>>#text

App 1.4.2 ComposedText implementation

TextLines subclass: #ComposedText
instanceVariableNames: 'text compositionWidth compositionHeight wordWrap fontPolicy lineTable fitWidth '
classVariableNames: 'DefaultCompositionWidth '
poolDictionaries: 'TextConstants '
category: 'Graphics-Text Support'

Class ComposedText provides the support for creating, modifying and displaying stylized text.

Instance Variables:
compositionWidth <SmallInteger> The width of the area in which the text is composed. The composition width includes any space taken up by indentation.
compositionHeight <SmallInteger> The height of the composition
fitWidth <Boolean> True if the composed text was created shrink to fit.
fontPolicy <FontPolicy> The place where fonts are obtained
lineTable <LineInformationTable>
text <Text>
wordWrap <Boolean> The flag to indicate whether word wrapping is turned on

Class Variables:
DefaultCompositionWidth <SmallInteger> used for composition unless otherwise specified

ComposedText>>characterBlockAtPoint: (character location)
characterBlockAtPoint: aPoint
"*Answer a CharacterBlock for characters in the text at point aPoint.*"

^self getCharacterBlockScanner characterBlockAtPoint: aPoint in: self

ComposedText>>characterBlockForIndex: (character location)
characterBlockForIndex: targetIndex
"*Answer a CharacterBlock for character in the text at targetIndex. The coordinates in the CharacterBlock are in the coordinates of the receiver.*"

^self getCharacterBlockScanner characterBlockForIndex: targetIndex in: self

ComposedText>>composeAll (accessing)
composeAll
"*Compose a collection of characters into a collection of lines.*"

```
| startIndex stopIndex lineIndex maximumRightX compositionScanner |
lineTable :=self newLineTable.
maximumRightX := 0.
text size = 0
    ifTrue:
        [self setHeight: 0.
        lineTable lastStop: 0.
        ^maximumRightX].
startIndex := lineIndex := 1.
stopIndex := text size.
compositionScanner := self getCompositionScanner.
[startIndex > stopIndex] whileFalse:
    [| lineRight |
    self nextLineItem: startIndex line: lineIndex with: compositionScanner.
    lineTable
         lineAt: lineIndex
        putInformationFromScanner: compositionScanner.
    lineRight := compositionScanner rightX.
     maximumRightX < lineRight ifTrue: [maximumRightX := lineRight].
     startIndex := compositionScanner lineLast + 1.
     lineIndex := lineIndex + 1].
self updateCompositionHeight.
lineTable lastStop: text size.
lineTable trimLinesTo: lineIndex - 1.
^maximumRightX
```

Appendix 2
Feedback from functionality and user interface evaluation

1. Image-image communication not needed if SRE and Target images the same.
   Done. Special command for analysis in same image.
2. BOSS dangerous for image-image communication.
   Changed to WSReadWriter. Incidentally, a specification/result file size reduced by factor 10!
3. User selectable name of communication file.
   Done. Same file now used for communication in both directions.
4. Feedback during analysis needed.
   Done.
5. Should highlight interfaces and messages not found in analysis.
   Done. Collaboration tool now highlights such messages and interfaces by italics font. Also, empty ports (=ports without messages) shown crossed out. Filters in Reverse Engineering tool filters on 'done', 'in class', 'dummy', and 'base'.
6. It would be useful with some statistics and metrics such as:
   NOT DONE
   - List interfaces sorted on no. of messages
   - Show messages (selector) sent from several ports and/or interfaces
   - No. of methods/role
   - Messages used once only
   - Ports/role
   - Sending of messages classified as private
   - List of method overrides
7. Better handling of dummy messages and interfaces
   Done. Dummy messages and interfaces now stored compactly in RMREPort, not shown in regular OOram tool.
8. Generate self messages.
   Done by defining two collaborating roles with identical associated class names.
9. Browse methods from SRE tool if same image.
   Not done, use regular Browser.
10. Show number of selector occurrences in method source code.
    Done
11. Many redundant refreshes.
    Fixed. It was not easy, but is now hopefully OK.
12. Check if source files exist and are available.
    Done. If trouble, user is invited to specify Target source file names.
13. Remove empty interfaces.
    Done. Chose to provide command in SRE tool rather than making it automatic.

TABLE OF CONTENTS

INDEX