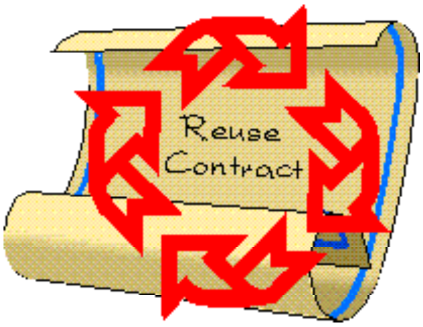


Reuse Contracts as a basis for investigating reusability of Smalltalk code

Tutorial Exercises

ESUG'97 Summer School

Koen De Hondt
Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel

kdehondt@vub.ac.be

Style conventions used in this text:

| Text in *italic*: text that you read on screen

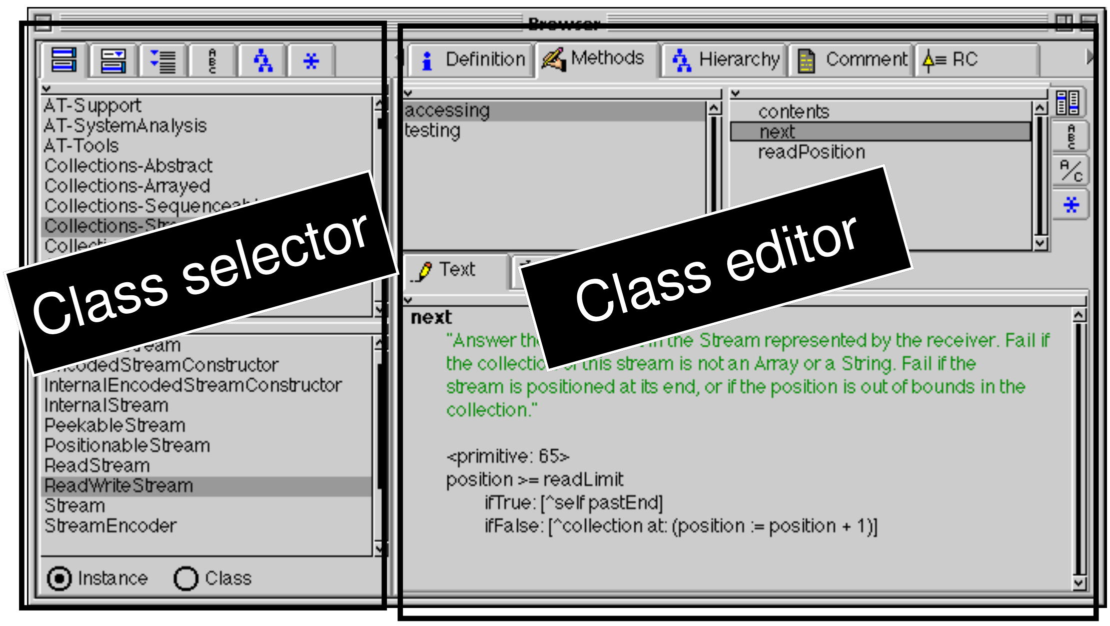
IMPORTANT The software you are about to use is a development version, but problems are not expected.

Exercise 1 Getting Acquainted with the Browser

The browser used in these exercises is a bit more sophisticated than the standard VisualWorks browser. The goal of this exercise is to get acquainted with the browser and to get an overview of its functionality.

Step 1 Open the exercises image. You should see a launcher.
Press the *Browse* button. A menu of available browsers appears.
Select *ESUG'97 Browser* and position the window as desired (full screen is best).

Step 2 The ESUG'97 browser's structure differs somewhat from the structure of the standard VisualWorks browser.
The category and class lists are placed vertically instead of horizontally. The combination of the category list, the class list, and the instance/class switch is know as the "class selector". What you see on the right side of the class selector is called the "class editor" (or "class viewer" when no editing is allowed).



The class selector is in fact a notebook, which allows the user to pick the most appropriate view to select a class. From left to right you find:

- a category list and a class list, as in the standard VisualWorks browser
- a category menu and a class list

- a list in which classes are nested in categories
- an alphabetical class list
- a hierarchical class list
- a query field and a class list

Switching from one page to another in the notebook preserves the class selection. Note that the class selector does not have a 'Find class…' command, since that functionality is covered by the last page in the notebook, i.e. the combination of a query field and a class list.

Select a class and explore the different pages in the class selector notebook.

Step 3 The class editor is also a notebook. Each page contains a different editor/viewer. The class editor notebook in the ESUG'97 browser contains 4 familiar pages, such as *Definition*, *Methods*, *Hierarchy*, and *Comment*, and 4 extra pages for this tutorial:

- the reuse contracts (*RC*) page
- the *Analysis* page
- the *Clusters* page
- the *Metrics* page

Select a class and explore the 4 first pages in the class editor notebook.

It is impossible to create a new class by editing a new class template and accepting it, as is the standard way in the VisualWorks browsers. Instead, new classes are created with the *New Class…* command (in the menu of any class list) and instance variables, class variables, and pool dictionaries are entered on the *Definition* page of the class editor notebook.

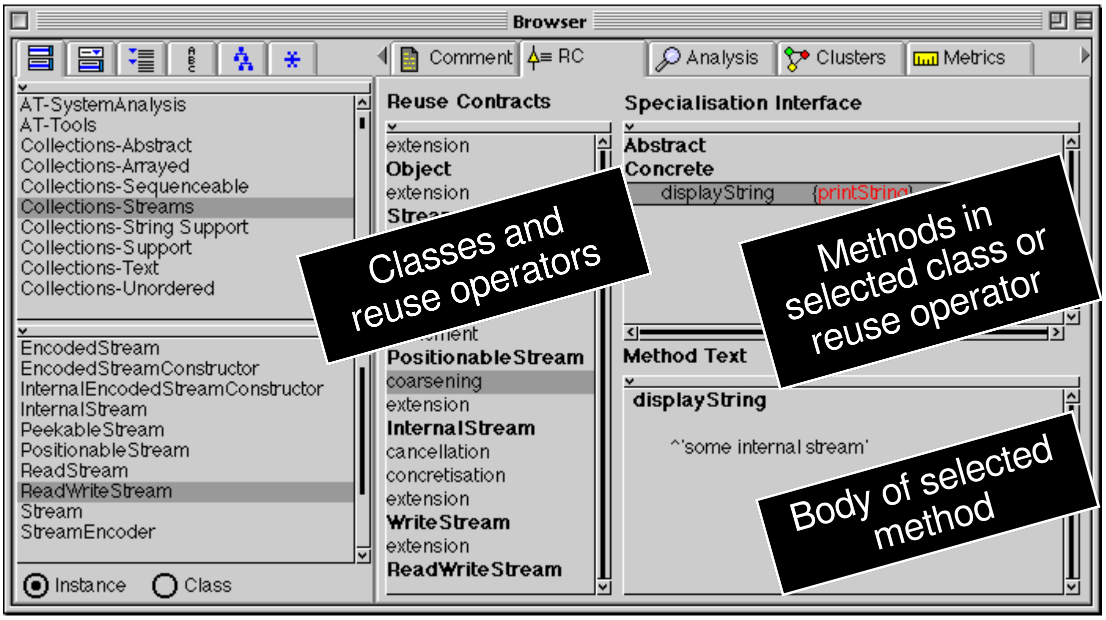
The ESUG'97 browser does not display change request dialogs when the user makes another selection before accepting any editions. So if you edit a class on the *Definition* page, or if you edit a method on the *Methods* page, or if you edit a class comment on the *Comment* page, make sure that you accept the changes before switching to another page, selecting another class, or selecting another method.

EXERCISE 2

The goal of this exercise is to examine class hierarchies based on reuse contracts.

Step 1 Select class *ReadWriteStream* in the class selector notebook and switch to the *RC* page of the class editor notebook. A "busy" mouse pointer indicates that reuse contracts are being extracted from the selected class and all its superclasses. This information is not cached, so picking another class (or the same class) will start the extraction process again.

The *RC* page consists of 3 views. The list on the left shows the result of the extraction process: all classes in the superclass chain of the selected class, with the extracted reuse operators between them. When a selection in this list is made, the details of the selection are shown in the top right list view. The extension and refinement reuse operators use the color green to indicate what was added with respect to the superclass, while the cancellation and coarsening reuse operators use the color red to indicate what was removed. The bottom right view shows the body of a method selected in the top right view.



Step 2 Investigate the extracted information. In particular, look what happens with the *#next* method in the class chain from *Stream* to *ReadWriteStream*. This information is used on slides (45 – 53) of the presentation. *#next* is introduced as an abstract method by the extension from *Object* to *Stream*. *PeekableStream*, *PositionableStream*, and *InternalStream* do not override it, but *WriteStream* removes it (see the cancellation from *InternalStream* to *WriteStream*). Finally, *ReadWriteStream* re-introduces *#next* as a concrete method (in fact a primitive).
It is rather strange that an abstract method is removed from a subclass because it is not appropriate and then re-introduced in a subclass, but keep in mind that the Stream hierarchy should be (partly) a multiple inheritance hierarchy. ReadWriteStream is the class that combines WriteStream and ReadStream behavior. Such combination typically gives rise to design problems in a single inheritance language.

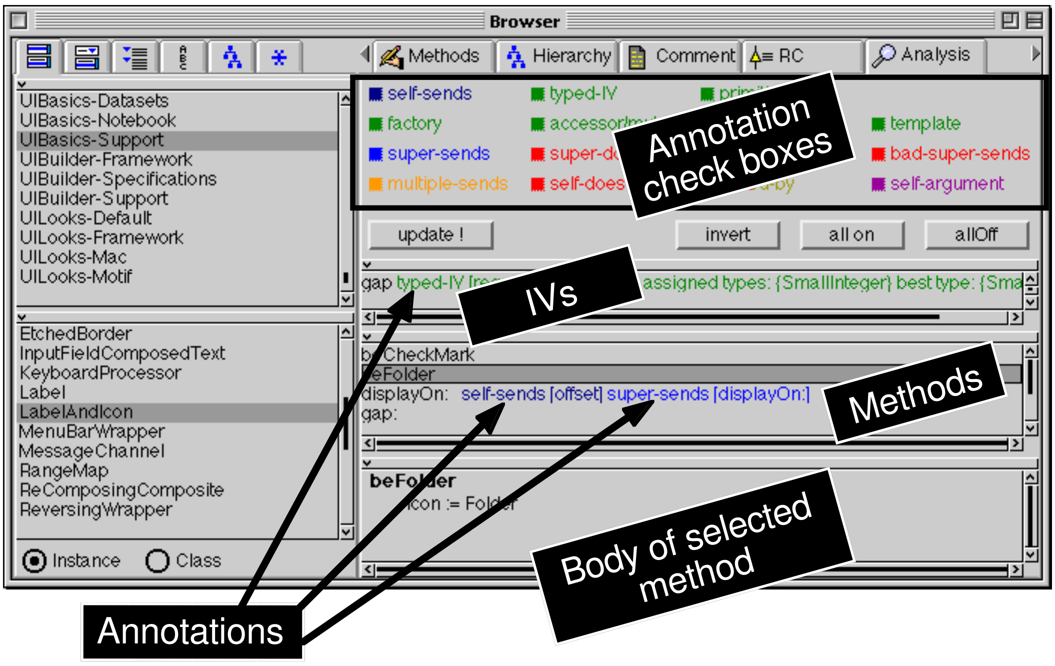
Step 3 Examine the reuse operators for other class chains.
For example, take a look at Array, IdentityDictionary, View, and AspectAdaptor.

EXERCISE 3

The goal of this exercise is to analyse source code with the browser and look for code that might hinder reuse.

Step 1 Select class *View* and switch to the *Analysis* page of the class editor notebook. A “busy” mouse pointer indicates that analysis data are being extracted from the selected class and all its superclasses. These data are cached, so picking another class (or the same class) will <u>not</u> start the extraction process again. In this development version of the browser however, the (whole) cache is invalidated when a method is accepted in an ESUG’97 browser.

Step 2 The top of the *Analysis* page displays several check boxes. Each check box corresponds to an annotation shown after the instance variables and methods in the views below. The check boxes allow to selectively show analysis information in these views. Press the *Update!* button to apply changes to the state of the check boxes.



The view below the check boxes lists the instance variables of the selected class. For now, typing information (*typed-IV*) is the only information extracted from the source code. The *required interface* is the set of messages that is sent to the instance variable. The *assigned types* are the classes of which instances are assigned to the instance variable. When the required interface and the assigned types contains enough information, a *best type* can be derived. It is the class that best matches the extracted information.

The view below the instance variables lists the methods defined by the selected class. The bottom view displays the body of the selected method.

Although many more useful annotations can be imagined, for now the following method annotations are extracted from the source code:

self-sends: the set of messages sent by the method through self sends (i.e. the specialisation clause). For primitive methods this set is empty, even if the method contains primitive failure code.

primitive: this annotation is present if the method is a primitive method.

factory: this annotation is present if the method is a factory method, i.e. a method that does nothing but returning a class (this is the implementation in Smalltalk of the “factory method” design pattern).

accessor/mutator:a method is annotated with *accessor* or *mutator* when the method is an accessor or mutator method respectively.

abstract: this annotation is present when the method is an abstract method.

template: this annotation is present when the method is a template method, i.e. a method that invokes at least one abstract method through a self send.

super-sends: the set of messages sent by the method through super sends.

super-does-not-understand: this annotation indicates that the listed messages are not implemented in any of the superclasses of the method’s class. This annotation reflects a bug.

just-super-send: this annotation indicates that the method contains only a super send. This method can be safely removed without affecting the behavior of the class.

bad-super-send: this annotation indicates that the method invokes a method with a different name through a super send. Bad super sends are a classic example of bad coding style that inhibits reuse. Often a developer is compelled to use bad super sends, because the method in the superclass is not written with reuse in mind. Factorization of the method in the superclass into smaller methods would solve many problems, since the developer can override in a more fine-grained way.

multiple-sends: this annotation lists the messages that are invoked repeatedly. This might, or might not, be a problem. It is interesting to have this information for performance reasons, for instance when time consuming methods are invoked several times.

self-does-not-understand: this annotation indicates that the listed messages (invoked through self sends) are not part of the interface of the method’s class. This annotation reflects a bug.

called-by: this annotation lists the methods that call this method (i.e. the senders of a method within the class).

self-argument: this annotation lists the methods that send messages with self as argument. Such message sends indicate possible collaborations of the method's class and the classes of the invoked methods. When you want to override a method that passes the receiver as an argument, have a look at the method that receives the receiver as an argument, because it could send messages back to that receiver. Maybe you have to adapt more than this method in order to get the desired result.

Each type of annotation has a different color, so that annotations of the same type are spotted easily. The red annotations concern bugs, bad coding style, and reuse inhibitors. The other colors have no special meaning.

Step 3 Have a look at the annotations of the following methods and think about how you would eliminate the problem:

Class	**Method**
TwoByteString	sizeInBytes
Symbol	stringHash
UninterpretedBytes	sizeInBytes
MacFilename	moveTo:

EXERCISE 4

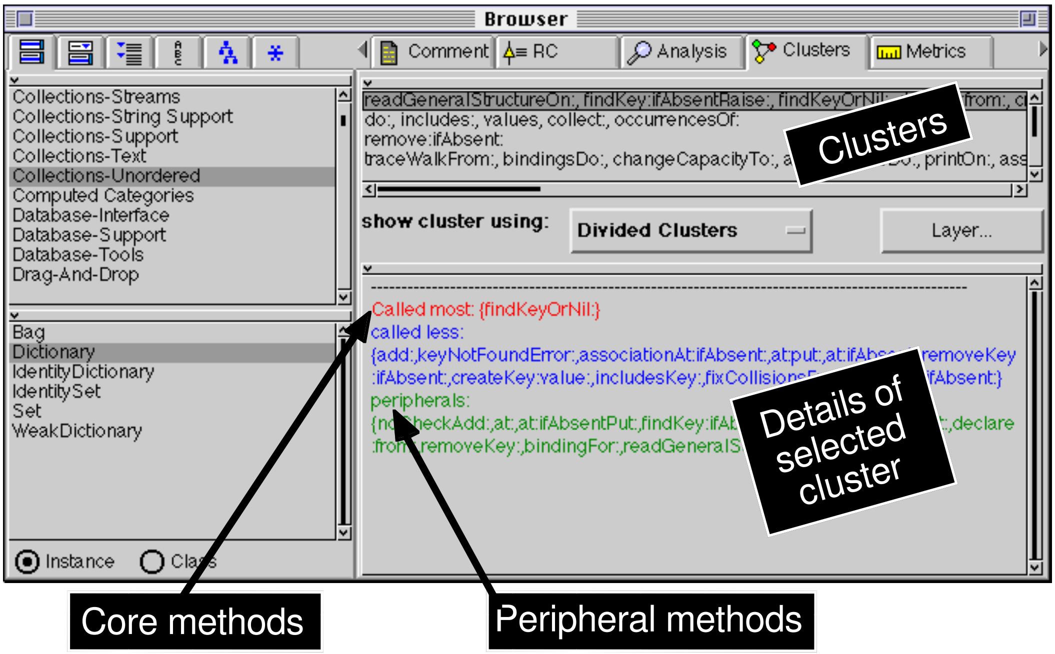
The goal of this exercise is to explore the clustering view of the browser. Note that this view is <u>highly experimental</u>.

A cluster is a set of methods that invoke each other within a class. Clusters give an indication of which methods belong together, or which methods are concerned with a certain aspect of the class in which the methods reside. For now, only one type of clustering is supported. Other ways of clustering are under investigation.

Step 1 Select class *Dictionary* and switch to the *Clusters* page of the class editor notebook.

The *Clusters* page roughly consists of two views. The top view lists all clusters that could be found in the selected class. When a cluster is selected, the bottom view displays the details of the selected cluster, according to the state of the pop-up menu and the current layer (on which no details are given here).
When a cluster contains enough methods (currently more than 5), the cluster can be viewed as a *divided cluster*. The cluster is then divided into one, two, or three parts. The division is based on the number of invocations of the methods in the cluster. The *called most* part lists the methods that are invoked in more than 35% of all method invocations in the cluster. The

peripherals part lists all methods that are not invoked from within the cluster (and thus not from within the class). The *called less* part lists all methods in the cluster that are not in the two other parts.



The methods in the *called most* part are called “core methods”, while the methods in the *peripheral* part are called “peripheral methods”. This categorization gives interesting information to subclassers. When a core method is overridden in a subclass, the developer can expect that this action will have an effect on the methods that invoke that method. The developer can read the affected methods from the cluster. Typically, these core methods are private methods and will not be invoked from outside the class. Peripheral methods, on the other hand, are invoked from outside the class. Thus overriding a peripheral method has no effect on the class itself, but on the clients of the subclass. Therefore clustering information is crucial for developers who want to override peripheral methods.

In the figure above, there is a cluster concerning finding keys. The *#findKeyOrNil:* method is a core method and seems to play an important role in many other methods. It is clear that overriding this method in a subclass of Dictionary should be done with special care.

Step 2 Examine clusters for other classes.
For example, take a look at Collection (cluster for enumeration), Stream (cluster for putting objects on a stream), Object (cluster for the dependency mechanism), and VisualComponent (cluster for bounds).

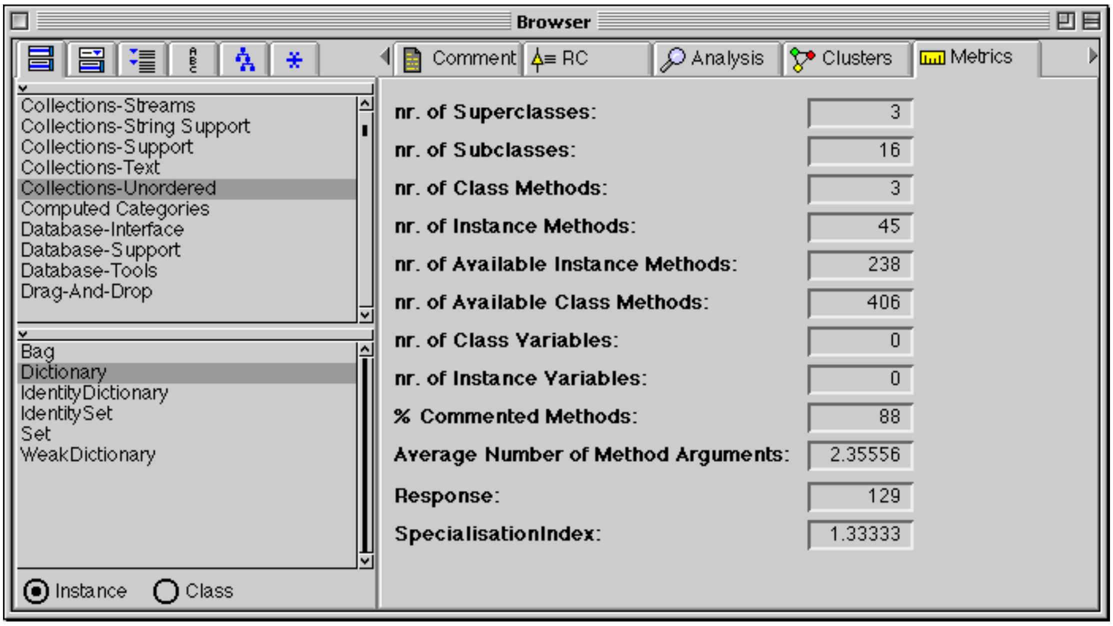
EXERCISE 5

The goal of this exercise is to explore the metrics view of the browser. This view shows metrics data for the selected class. As the clusters page, the metrics view is still in an experimental phase. Although we believe that the numbers shown in this view are of limited use, we have added this view anyway.

Detailed information about the meaning of the numbers can be found in:

- Brian Henderson-Selllers, "Object-Oriented Metrics, Measures of Complexity". Prentice Hall
- Shyam R. Chidamber and Chris F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design", Proceedings of OOPSLA'91 (Oct. 6–11, Phoenix, Arizona). ACM Press, October 1991
- http://www.hatteras.com/

Step 1 Select class *Dictionary* and switch to the *Metrics* page of the class editor notebook.



The numbers speak for themselves, except the two last ones.
The *response* is defined as the sum of the number of local methods and the number of methods invoked by the local methods.
The *specialisation index* is defined as the sum of the nesting level of a class and the number of overridden methods, divided by the total number of methods in that class. See the references for more information.

Step 2 Examine metrics of other classes.
For example, take a look at Filename, Collection, and False.

EXERCISE 6

The goal of this exercise is to examine your own classes with the browser.

Step 1 Close the browser.
File in your classes.

Step 2 Open a browser and examine your classes.

If you have suggestions for improvements to the browser, please do not hesitate to send them to kdehondt@vub.ac.be.
If you are interested to show the browser to other people, you are entitled to copy the image and the changes file used in this tutorial, or contact Koen De Hondt via e-mail to get the file outs.
Check http://progwww.vub.ac.be/prog/pools/rcs/ for further developments.