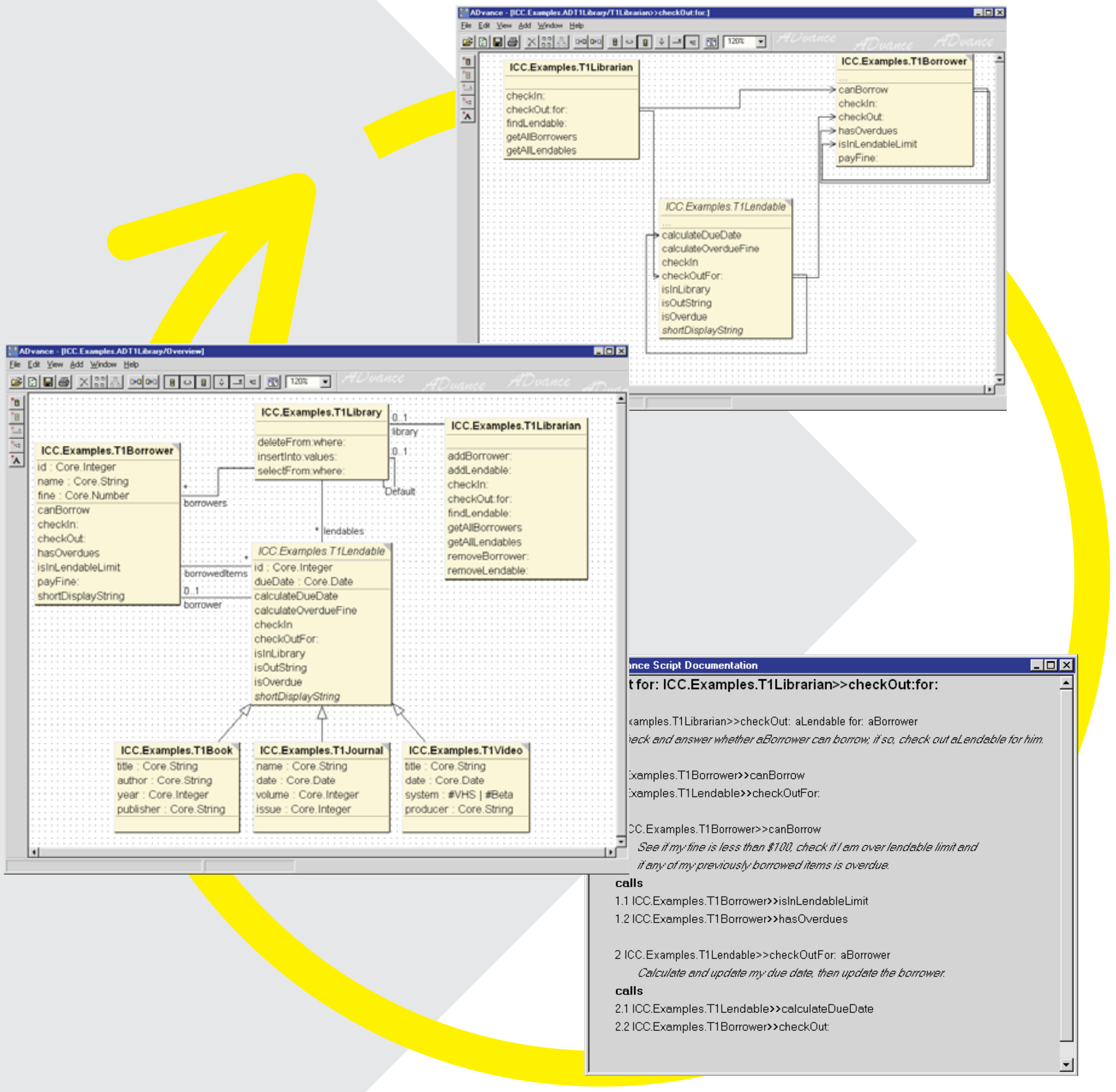
Version 7.4.1 User's Guide

IC&C ADvance™

Integrated
Round-Trip Engineering



t for: ICC.Examples.T1Librarian>>checkOut:for:

xamples.T1Librarian>>checkOut: aLendable for: aBorrower

eck and answer whether aBorrower can borrow; if so, check out aLendable for him.

xamples.T1Borrower>>canBorrow
xamples.T1Lendable>>checkOutFor:

CC.Examples.T1Borrower>>canBorrow

See if my fine is less than $100, check if I am over lendable limit and if any of my previously borrowed items is overdue.

calls

1.1 ICC.Examples.T1Borrower>>isInLendableLimit
1.2 ICC.Examples.T1Borrower>>hasOverdues

2 ICC.Examples.T1Lendable>>checkOutFor: aBorrower

Calculate and update my due date, then update the borrower.

calls

2.1 ICC.Examples.T1Lendable>>calculateDueDate
2.2 ICC.Examples.T1Borrower>>checkOut:

IC&C GmbH
Software Foundations

IC&C ADvance™

Integrated Round-Trip Engineering

Release 7.4.1

Created by

IC&C GmbH
Software Foundations
Hamburg/Germany

ADvance User's Guide

Software Product Release 7.4.1

This document is subject to change without notice.

Trademark acknowledgments

Intellectual property rights

The following copyright notices apply to software that accompanies this documentation:

May 2006

Created by

IC&C GmbH Software Foundations
Papenhöhe 14
D-25335 Elmshorn/Hamburg
Germany

Tel.:	+49 (0) 4121/2392-0
Fax:	+49 (0) 4121/2392-59
Email:	advance@icc-gmbh.de
Web:	www.icc-gmbh.de

Contents

About this Guide

The *ADvance User's Guide* provides details about both a functional view of the **ADvance** options and a view of how the tool would be used in real world situations. It demonstrates the advantages of the tool from different perspectives and guides the user through two distinct engineering exercises, forward and reverse engineering.

Audience

This guide addresses two primary audiences:

- Smalltalk developers
- System analysts.

Both groups should at least be familiar with Smalltalk syntax, object-oriented programming concepts and basic VisualWorks programming tools, as described in the *VisualWorks User's Guide*. If you intend to develop complex applications, you may wish to have a more thorough understanding of seamless and iterative system development. For this issue we recommend [*Waldén/Nerson*][1] as complimentary reading.

Organization

The *ADvance User's Guide* provides comprehensive instructions for using **ADvance** and understanding its notation and concepts. It is divided into eight chapters.

The first chapter called *Introduction* describes the tool philosophy and key benefits.

Installation as the second chapter guides you through the installation process.

The next three chapters provide a *Tutorial*. The tutorial should be worked through step by step. Even though **ADvance** is fairly easy to use, there are many concepts of depth which are not communicated to an inexperienced user at first sight. Thus, in order to exploit the full range of **ADvance** capabilities, take your time! The tutorial chapters are all based on a single library example that has been discussed in detail in [*Wilkerson*][2].

---

[1] Kim Waldén and Jean-Marc Nerson. *Seamless object-oriented software architecture:analysis and design of reliable systems.* Prentice Hall, Englewood Cliffs, NJ, 1995

[2] Nancy Wilkerson: *Using CRC Cards*. SIGS, 1995

The third chapter called *Overview* presents an overview of tool components. The goal of the overview is to let the user become acquainted with the tool itself.

The fourth chapter describes the *Reverse Engineering* process of generating design diagrams from existing code.

Forward Engineering as the fifth chapter provides instructions for capturing new design diagrams and demonstrates the full integration of graphics to code.

The *Additional Concepts* in chapter six provide a discussion of various issues which are important to gain an in-depth understanding of the role **ADvance** can play in the iterative and seamless system development process. The *Modeling* section describes how business process modeling is supported by the tool. This chapter is mandatory.

The seventh chapter provides a *User Interface Component Reference*, this is a detailed descriptions for the tools, dialogs, menu items, shortcuts and commands in **ADvance**.

The eighth chapter provides a list of *Tips and Techniques* which have been recorded from **ADvance** users' comments.

The *Glossary* provides brief descriptions for concepts which are mentioned in this guide without further explanation.

Release Notes

- The integrated **ADvance** 7.4.1 is adapted to the VisualWorks 7.4.1 release.

Conventions

This section describes the notational conventions used to identify technical terms, Smalltalk constructs, user interface actions and keyboard operations.

Typographic Conventions

This guide uses the following fonts to designate special terms:

Example	Description
Subject	Indicates new terms where they are defined.
Template	Indicates dialog names, book titles and words as words.
Emphasis	Indicates emphasized words.
Dictionary	Indicates Smalltalk constructs; it also indicates any other information that you enter through the graphical user interface.
File menu	Indicates user interface labels for menu names, dialog-box fields, and buttons.

Special Symbols

The following symbols are used to designate certain items or relationships:

Example	Description
File→Open	Indicates the name of an item on a menu.
<Operate> menu	Indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.
🖱	Indicates an action to be performed by the user.
↳	Explains the result of an action and is usually followed by a graphic showing that results.
Note:	Indicates important information.

Mouse Operations

The following table explains the terminology used to describe actions that you perform with mouse buttons. See the *VisualWorks User's Guide* for details on mouse buttons and mouse operations.

When you see:	Do this:
Click	Press and release the <Select> mouse button.
Double-click	Press and release the <Select> mouse button twice without moving the pointer.
<Shift>-double-click	While holding down the <Shift> key, press and release the <Select> mouse button twice.
<Control>-double-click	While holding down the <Control> key, press and release the <Select> mouse button twice.

1 Introduction

What is ADvance?

ADvance is a multidimensional OOAD tool for supporting object-oriented analysis and design, reverse engineering and documentation.

ADvance provides an always up-to-date graphical view on your object models. It fully integrates into your VisualWorks image. Its visual interface helps you to quickly develop your object model and allows understanding and/or modifying the object design inherent in your code.

Using the powerful mechanisms of **ADvance**, the structural, behavioral and architectural information contained in VisualWorks applications can be easily extracted, visualized and analyzed. Furthermore, it provides a powerful mechanism for supporting concurrent design documentation and construction of classes and relationships. Adopting **UML** as notation, **ADvance** basically adheres to industry standards. It should be noted that **ADvance** is focussing on ease of use in order to enhance immediate feedgack and fast communications among project members. UML is an important part used to support those goals.

The capital ‘AD’ in **ADvance** indicates one of our observations: Analysis and Design are not separable in the iterative process of building systems. If they are, this should be changed. We think that this view is useful to achieve a seamless integration of the different views of a system.

Why use ADvance?

ADvance’s **forward engineering** component let’s you graphically design, create and iterate over your code. This speeds up the development process. The **reverse engineering** component helps you in understanding and communicating code. Often a diagram says more than a thousand words. The tool’s **documentation** facilities help you creating textual and graphical documentation – for methods, classes, interactions in use cases, subsystems, or complete applications.

When capturing a new object design with the graphical browser, **ADvance** automatically generates Smalltalk code from the graphics. In this manner the integrity of the design and software is maintained from design through implementation.

The tool also allows the user to view existing code as object design models. In this manner the model can be used to review existing OO constructs or as a design consistency check for the current iteration. The tool can also be used for analyzing the model for accuracy to system requirements. With the **ADvance** diagram canvas integrated into the default VisualWorks System Browser, you can view automatically created class diagrams for StORE packages or hierarchy diagrams for single classes.

In addition, **ADvance** is equipped with a plug-in interface, which allows seamlessly integrating other tools into the **ADvance** tool suite.

2 Installation

The **ADvance** software package is shipped on a single floppy disk or comes readily with the VisualWorks system for versions after 5i.4. The files on this disk have to be extracted to receive the contained parcel files, which you can install into your VisualWorks environment.

System Extensions

Installation of the **ADvance** parcels will result in the extension of some existing system classes. All system extensions are installed under protocol names that reflect the owning module and the fact that the method is an extension.

For example, the method *ApplicationModel>>iccEnable:group:* is installed under the protocol *ICC-Common Classes*. As shown in this example we've chosen method name prefixes to avoid naming conflicts.

Reading the Disk

The **ADvance** installation disk is formatted either for PC platform (Windows, OS/2), Unix (tar), or Mac.

PC Platform

The files are stored as a self-extracting Zip archive. Run the .exe to extract the parcel files. Please provide a subdirectory (i.e. *icc*) of your VisualWorks home directory as the installation path.

Unix

The files are tared, gzipped, then tared onto floppy disk. To extract, use `tar` to read the `.gz` file from the floppy disk. For example, a Solaris platform may execute a command similar to the following:

```
tar xvf /vol/dev/aliases/floppy0
```

Executing the above command will result in extracting a `.tar.gz` file. To complete the extraction process, ungzip the `.tar.gz` file; then use tar to expand the `.tar` file.

Mac

The files come in a self-extracting archive for the Mac. To extract the files, double click on the file found on the floppy.

Using PC Disk on Other Platforms

When using a PC-platform Zip archive on other platforms a file conversion may be necessary. Extract the Zip archive then convert all text files; these are files with extensions `txt`, `st, pst, htm`, `ext`, `rm`, `msc` and `inf`. Files with extensions `pcl, gif`, `jpg` and `bos` are binary files.

Note: All file- and pathnames are case sensitive. Please make sure that all conversion options are turned off in your extraction tool.

VisualWorks Installation

Loading the ADvance parcels

Before loading **ADvance** into your image, ensure that the path of the **ADvance** parcels is part of the parcel path settings in your image (i.e. *$(VISUALWORKS)/icc*). This is the case in the original settings of a VisualWorks 7.4.1 image.

In order to install **ADvance** in your image, simply load the parcel named *ADvance2*. The required parcels *ICC**, *Store-Base*, and *Compression-ZLib* will be loaded automatically. The loading can be done either using the menu item *Tools→Load Parcel named...* and entering `ADvance*,` or using the *Parcel Manager's* Directories tab and selecting *icc*. For further details on parcel loading and management please have a look into the VisualWorks *Application Developer's Guide*.

To load the plug-ins, tutorials and/or examples, load the respective parcels called *ADvance2PlugIns, ADvance2Tutorial* and/or *ADvance2Examples* the same way.

When loading of *ADvance2* finishes, the *Advance Workbench* starts automatically. To reopen **ADvance** later after closing the workbench, you can use the menu item *Tools→ ADvance Workbench* in the VisualLauncher.

Note: If you need to load the **ADvance** parcels into an image which already has an ADvance version loaded, ensure that no of your own subject classes or methods are added to the **ADvance** parcels.

Loading ADvance from StORE

There is an alternative way for loading **ADvance** if you have access to the public *Cincom StORE repository*: Load the bundle *ADvance2Bundle* to completely install **ADvance** and its additional components. Depending on your image settings, prerequistits will be loaded as parcels or as packages. The VisualWorks *Source Code Managament Guide* contains more details about *StORE*, bundles, packages, and related issues.

Removing from VisualWorks

To remove **ADvance** and its components from a VisualWorks image, please unload following parcels or packages:

- *ADvance2PlugIns,*
- *ADvance2,*
- *ADvance2Tutorial,*
- *ADvance2Examples,*
- *ADvance2RefactoringBrowser,*
- *ICCResources,*
- *ICCLookExtensions,*
- *ICCCommonClasses,*
- *ICC-Namespace,*
- *ICCColorSelector,*
- *ICCIncrementalTypes.*

3 Overview

Objectives

Present overview of tool components and philosophy.

Introduce two kinds of diagram views.

Explore editing, filtering, printing, and documenting techniques.

In this chapter we will examine the overall features of **ADvance** by looking at an existing code example demonstrating a library check-in/check-out system. We will do this by viewing several predefined diagrams that represent the library code. The tutorial code comes with a domain model and a simple GUI. You can run the application by evaluating the following statements:

```
ICC.Examples.T1UserConsole open.
ICC.Examples.T1LibrarianConsole open.
```

Getting Started

ADvance has its own launcher, the *ADvance Workbench*. This workbench provides a menu bar and corresponding toolbar for launching the high level tools from the **ADvance** tool suite. Some capabilities are repeated through other menu options as we will see later in the tutorial.

To open the *Workbench* from a workspace execute the following statement:

```
ICC.ADvance.AD2Workbench open
```

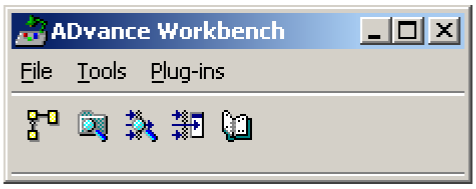


Figure 1. The *ADvance Workbench.*

We will now open the *Diagram Painter*, a graphical browser which is the main window for capturing, viewing and editing diagrams. We will use it to do the majority of our design work: graphically adding and modifying classes and their relationships.

To open the *Diagram Painter* from the *Workbench* use the left toolbar button or select **Tools→Diagram Painter** from the menu**.**

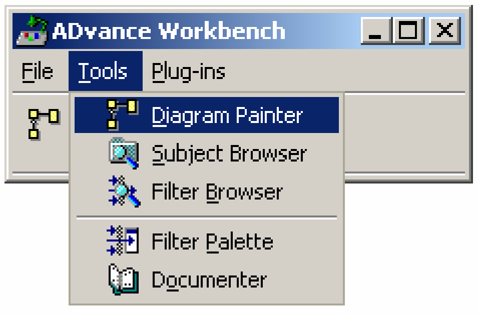


Figure 2. Opening a *Diagram Painter*.

This opens a *Diagram Painter*.

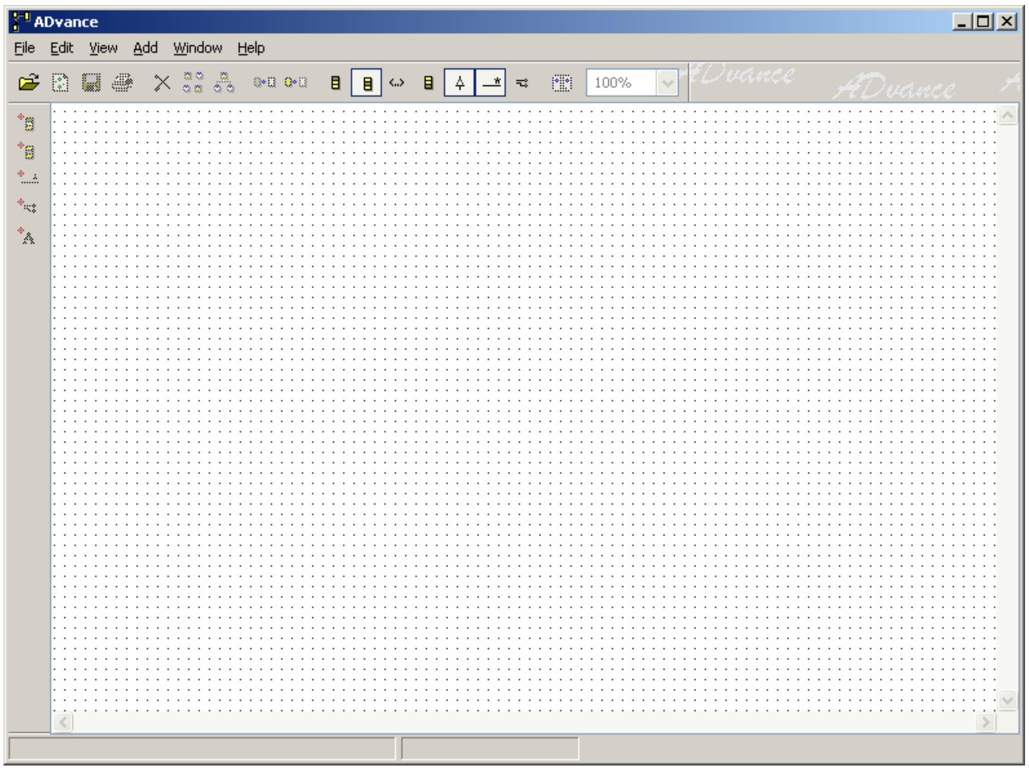


Figure 3. The *Diagram Painter*.

The *Diagram Painter* contains a menu bar for selecting, editing or filtering menu options. The toolbar icons[3] provide shortcuts to the most important pull down menu options, most of the context specific actions are repeated in pop-up.

[3] When you move the cursor over each toolbar icon a tool tip describes the function of each button.

Opening a Diagram

To open an existing diagram select **File→Open diagram...** from the *Diagram Painter*.

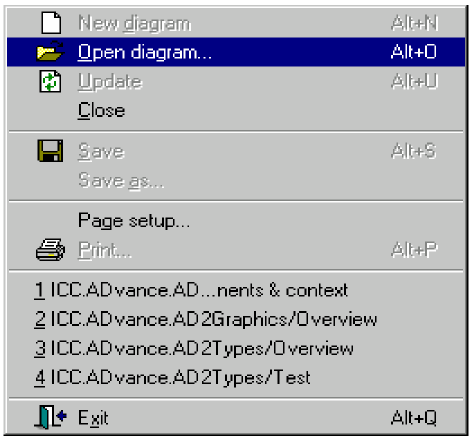

New diagram Alt+N
Open diagram... Alt+O
Update Alt+U
Close
Save Alt+S
Save as...
Page setup...
Print... Alt+P
1 ICC.ADvance.AD...nents & context
2 ICC.ADvance.AD2Graphics/Overview
3 ICC.ADvance.AD2Types/Overview
4 ICC.ADvance.AD2Types/Test
Exit Alt+Q

Figure 4. Opening a diagram.

This opens the *Open Diagram* dialog.

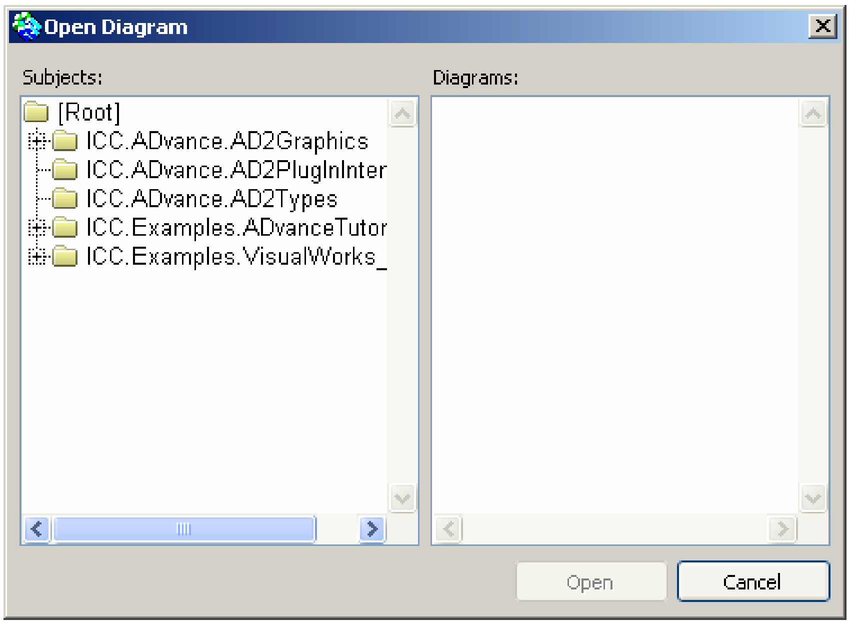


Figure 5. The *Open Diagram* dialog.

This dialog contains two panes. The left pane displays a tree of **subjects.** If a subject is selected the right pane displays the subject's **diagrams**.

A ***subject*** is a set of classes that you want to view with **ADvance**. Your subjects can pick any view that is deemed necessary to communicate an important aspect of your system design. A subject typically contains 5-30 collaborating classes and often equates closely to a subsystem, business model structure or use case of your application.

A subject has ***diagrams*** which represent different graphical views on the subject. Diagrams let you choose the level of detail you want to use to look at; you can create overview diagrams with the class hierarchy only or more detailed diagrams showing attributes, services and/or relations. Furthermore, you can use diagrams to look at different aspects of a subject, e.g. you can create structural diagrams with inheritance and relations or behavioral diagrams, which show message paths.

Since subjects can contain other subjects they are arranged in a hierarchical way. Thus, **ADvance** presents the list of available subjects as a tree.

In this tutorial the subject to work with is *ICC.Examples.ADT1Library*, which represents the library example.

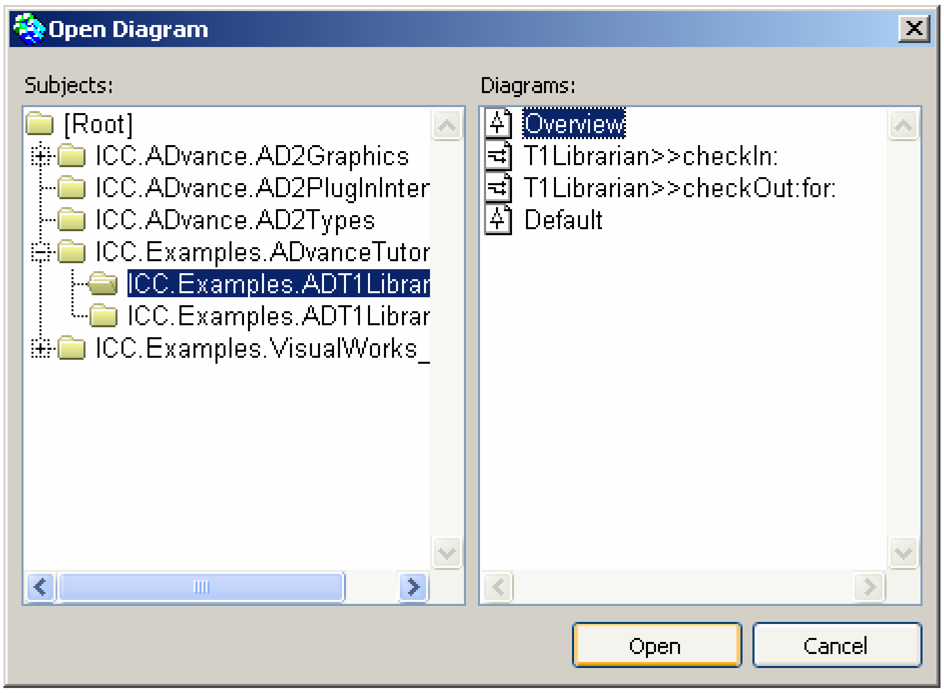
Expand the *ICC.Examples.ADvanceTutorial1* subject in the left pane by clicking on the **+**-field.

Select *ICC.Examples.ADT1Library* from the left pane.

The right pane then reflects all the diagrams contained within this subject.

Select the *Overview* diagram in the right pane, then click the **Open** button.

Note that the diagram icon indicates that this is a structural (vs. behavioural) view of these classes.


Open Diagram

Subjects:

[Root]
ICC.ADvance.AD2Graphics
ICC.ADvance.AD2PlugInInter
ICC.ADvance.AD2Types
ICC.Examples.ADvanceTutor
ICC.Examples.ADT1Librar
ICC.Examples.ADT1Librar
ICC.Examples.VisualWorks_

Diagrams:

Overview
T1Librarian>>checkIn:
T1Librarian>>checkOut:for:
Default

Figure 6. Selecting a diagram to open.

A diagram with classes and relations is drawn in the *Diagram Painter*.

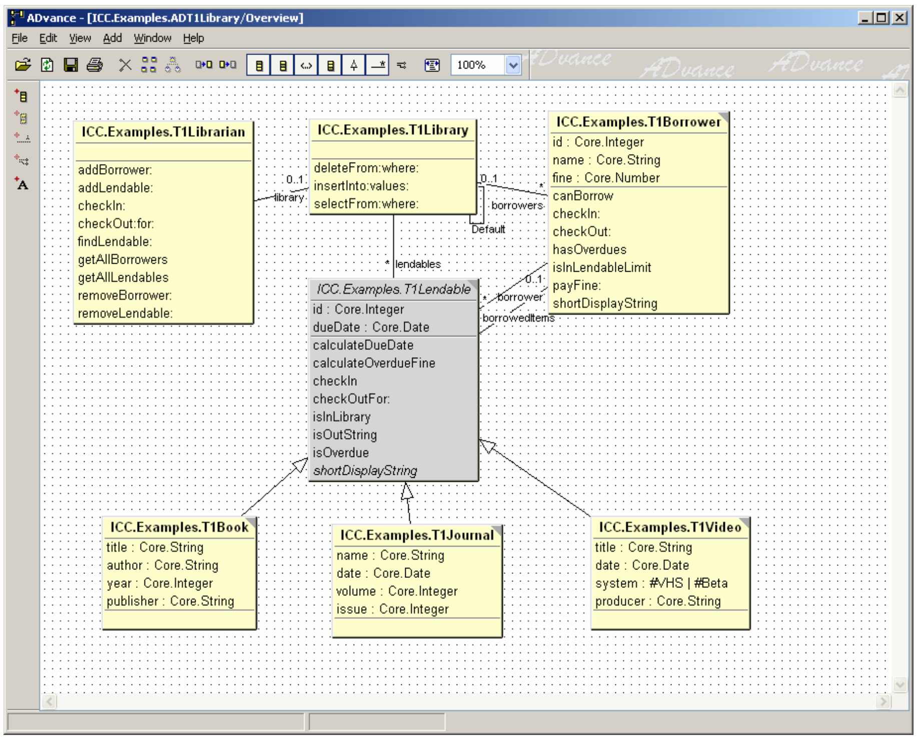


Figure 7. The *ICC.Examples.ADT1Library/Overview* diagram.

Resizing Diagrams

Originally the graphic may be too big to fit completely into the painter window. Sometimes the components in a diagram need to be sized differently so that the notations are easier to decipher or we can see the whole picture. In this case we would like to see all classes within the one window frame.

Adjust the size by selecting the **Fit to window** button ( ) in the painter's toolbar.

This adjusts the diagram to fit into the currently sized window. Based on the outcome, you may have to experiment with the window size or components to get just the right view. Classes can be individually moved by selecting them and then dragging them to the desired position in the diagram.

To change the diagram's zoom , you can also use the zoom combobox in the painter's upper right corner. It provides some entries with constant scaling factors. A special entry **Window** rescales the diagram to fit to the window everytime you resize the window.

Problem with Fit to window

If you select the *Fit to window* zoom for your diagram and have set your Windows settings to show window contents on mouse dragging, a resize of the diagram painter window will cause multiple repaints of that diagram.

Changing the Diagram Layout

Besides the resizing of a diagram and rearranging of classes you may change the layout of diagram lines. You can move start and end point of relation lines by dragging one end or the whole line, and you can change the layout style. These styles can be separately assigned for relation and inheritance lines. You can select a *gridded* and *straight* line style. See chapter 7 for the options of the *Painter Preferences'* **Layout** tab.

Viewing Program Structure

The selected diagram represents a high level structural view of the domain classes of the library system. Note that the *Diagram Painter's* window label contains the name of the subject followed by the name of the diagram, *ICC.Examples.ADT1Library/Overview*.

Understanding Diagram Notations

This diagram contains six classes represented by colored rectangles. The lines between classes indicate **inheritance** and **associations**, class *ICC.Examples.T1Book* is selected.

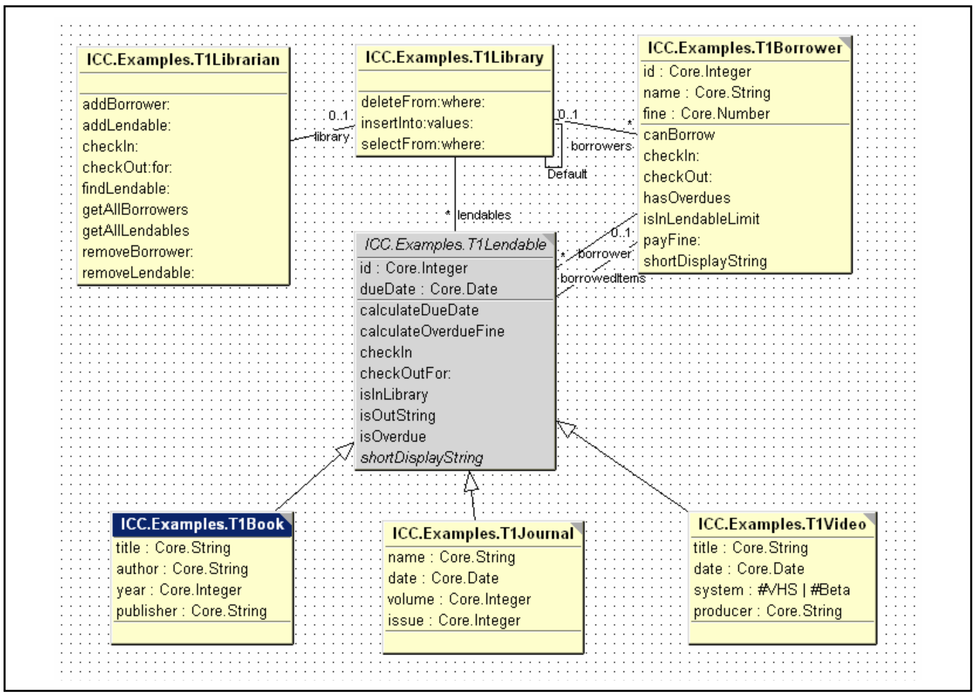


Figure 8. Notation overview.

A class is drawn as a rectangular box with three compartments, with the class name (and optional namespace) in the top compartment, a list of attributes (with optional attribute types) in the middle compartment, and a list of services in the bottom compartment. Abstract classes (*ICC.Examples.T1Lendable*) use italics for the class name, they may be displayed using a special color defined by the painter preferences (default is light gray). Classes which are intended to be persistent are distinguished by a marked upper right corner. Inheritance is shown by a line with a triangle where *ICC.Examples.T1Journal*, *ICC.Examples.T1Video* and *ICC.Examples.T1Book* are shown as subclasses of *ICC.Examples.T1Lendable*. Has-a relationships are shown by lines with their cardinality and role name at one end. For example, a single Lendable has one Borrower associated to it whereas a single Borrower has one or more Lendables associated with it. Indirect attributes are shown in the same way as direct attributes, for example *ICC.Examples.T1EditLendableDialog* has an instance variable *lendable* typed as <ValueHolder on: T1Lendable> which is shown as a line with cardinality 0..1 and name lendable.

For further details on notation we may refer to a standard reference on UML [*The Unified Modeling Lanugage Reference Manual*][4] as complimentary reading, or your preferred book on UML.

Emphasizing and grouping Classes by Color

You may want to add more information to your diagram by highlighting some classes or grouping some classes using background color. In addition to the default color options of the painter preferences (see section *Painter Preferences*), each class may be colored individually.

A class' background color can be determined in the color dialog.

Select class *ICC.Examples.T1Library*, and then select **Change color...** from the class pop-up menu.

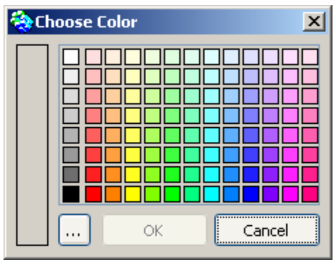


Figure 9. Determining a background color for the selected class.

If the given choice does not provide the color you are looking for you can design your own color by the clicking the **...** button. The *color picking* dialog will open. You can define colors using RGB- or HSB-system.

[4] Rumbaugh, Jacobson, Booch: The Unified Modeling Language Reference Guide, Addison Wesley, 1999

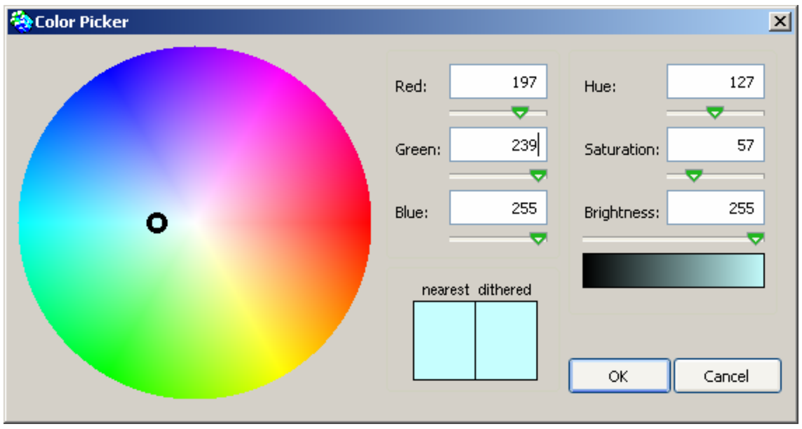


Figure 10. Defining individual Colors using the *color picker*.

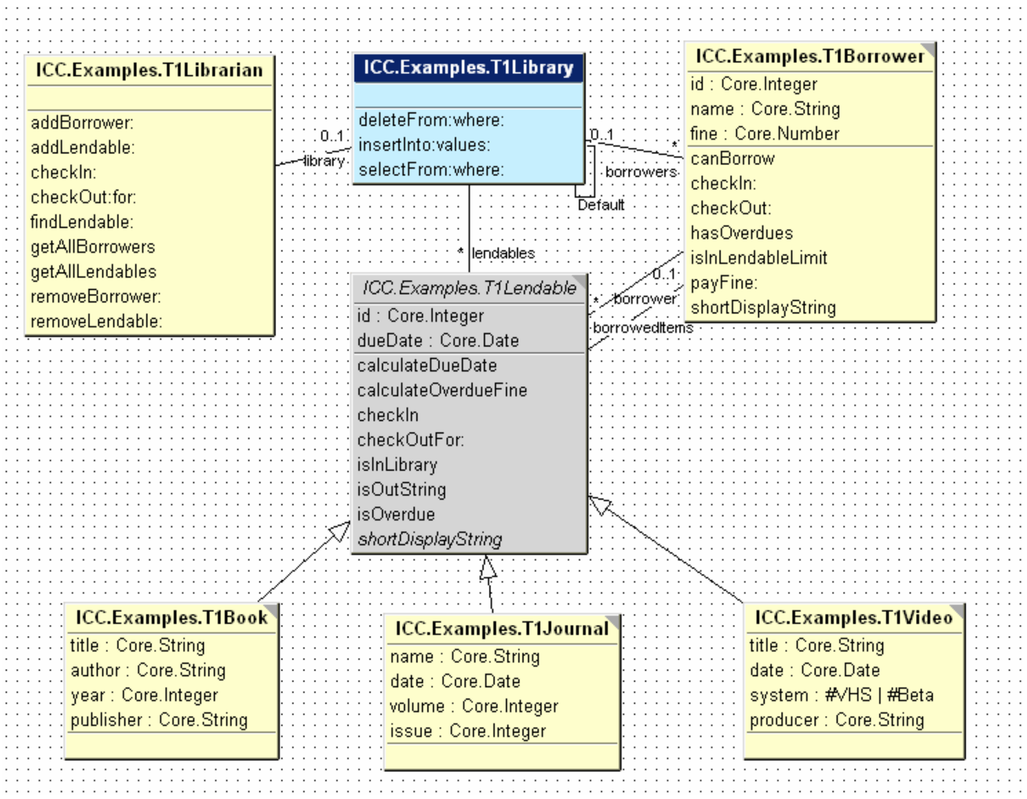


Figure 11. Class T1Library is highlighted by individual color.

Select class *ICC.Examples.T1Library*, then select **Default color** from the class pop-up menu to change its color back to the default color for concrete classes.

Choosing the Level of Detail

Each diagram can display multiple layers conveying different types of information aspects about the current diagram: Inheritance**,** Attributes**,** Attribute types**,** (has-a) Relations, Services and Scripts. These layers allow the user to focus in on what is important for a particular view. The *Glossary* contains a complete description of each aspect. Each of these layers may be toggled on or off individually. Toggling on any aspect allows the user to examine specific characteristics of the code and/or model.

We will now look at each of these in detail.

Examine the **View→Layers** pull down menu.

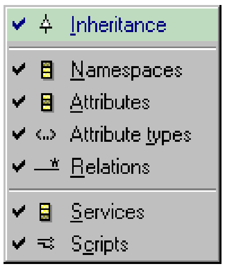


Figure 12. Toggling layers on and off with a menu.

You will see that the check marks indicate that inheritance, attributes, attribute types, relations, and services are turned on. The scripts layer is off in this view.

Each time you select a layer it will toggle its state to be on or off. Since this is a structural view, we do not necessarily need to view the services to gain the understanding the designer meant. Scripts are generally associated with behavioral diagram views.

Toggle **Namespaces** off. Notice the changes in the diagram. The class names are reduced to their names without leading namespaces.

Toggle **Services** and **Attributes** off. Notice the changes in the diagram. The class definitions are no longer readable.

Now toggle **Inheritance**, then **Relations** . What do you see? Restore the diagram to its original view by checking the first six options (not Scripts).

The same layer toggling functionality can be achieved by using the *Diagram Painter*'s toolbar buttons . These buttons are toggle switches for each of the layer they represent.

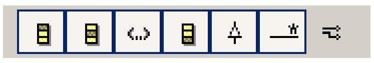


Figure 13. Toggling layers on and off with buttons.

Toggle the second button from left off then on - this is for the attributes layer.

Viewing Program Behavior

As we saw in the *Open Diagram* dialog, there were many diagrams to select from within the tutorial subject. To view the library code from a behavioral perspective we could choose to include scripts in our current diagram. This would make the diagram very cluttered and less readable. Since there is a behavioral diagram already saved, we are going to open this diagram.

Select **File→Open Diagram...**

First a dialog box appears asking if you want to save your changes.

Answer **NO**.

Choose *T1Librarian>>checkIn:* from the *ICC.Examples.ADT1Library* as the diagram to display.

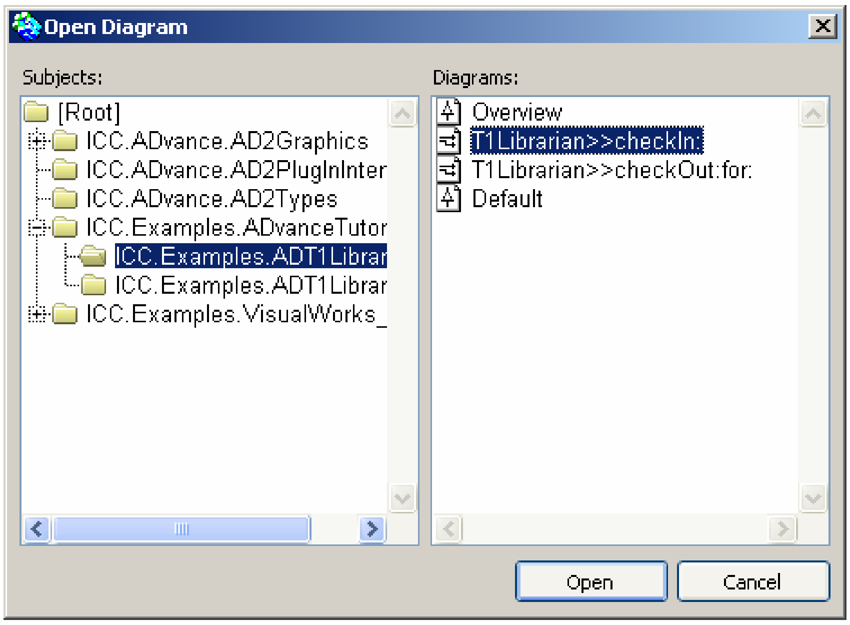


Figure 14. Opening the *T1Librarian>>checkIn:* diagram.

The diagram is drawn in the painter. Note that the window label contains the diagram name.

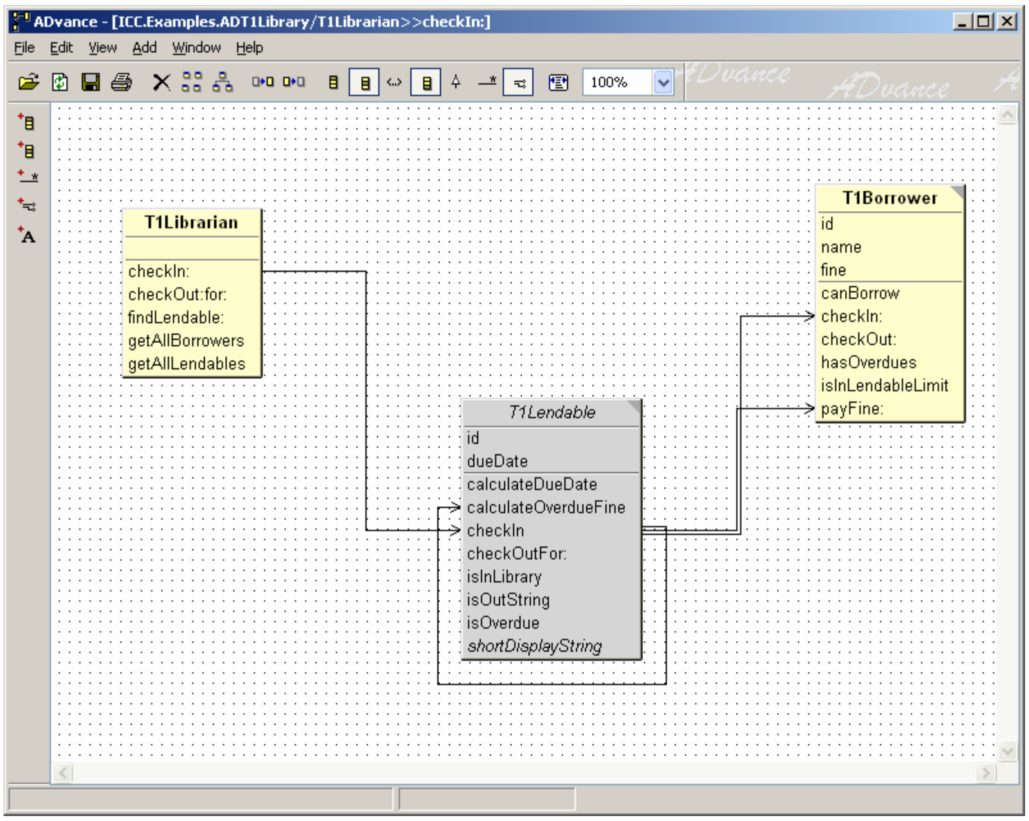


Figure 15. Displaying behavior.

Now we see lines with arrows depicting potential message sends between classes - hence the name *Behavior*. A collection of these message paths constructed from tracing the message sends of an originating service is called a **script** . Scripts are similar to **UML collaboration** interactions, the difference being that they are abstractions depicted at the class level.

In this diagram we see the following from left to right:

- The service *ICC.Examples.T1Librarian>>checkIn:* sends the message *checkIn* to an instance of *ICC.Examples.T1Lendable*.
- *ICC.Examples.T1Lendable* sends itself a message to calculate an overdue fine.
- *ICC.Examples.T1Lendable* sends messages *checkIn:* and *payFine:* to a *T1Borrower*.

A more comprehensive treatment of behavior derivation is given in the *Reverse Engineering* chapter.

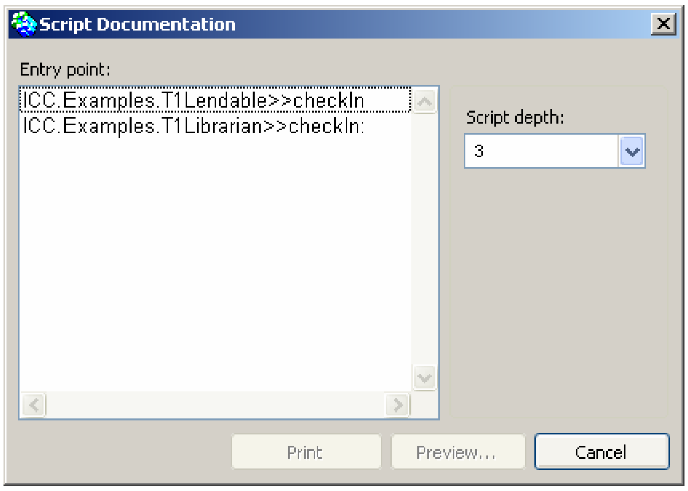
Note: If the script layer is toggled on and no scripts are drawn, this indicates that no scripts have been selected to be traced in the diagram.

Viewing Message Paths

Once the message paths have been graphically drawn through the **ADvance** derivation they can be examined for actual sequential message flow. This is supported by listing methods involved in a path and offering the method comments for understanding. For every script a documentation to that end may be printed or previewed.

Select **Window→Script Docu...**

The *Script Documentation* dialog appears.



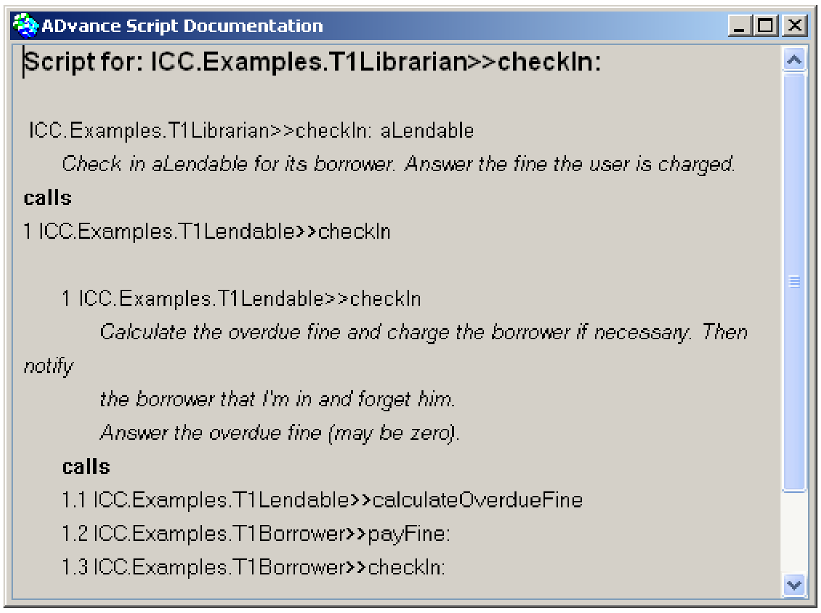
ICC.Examples.T1Lendable>>checkIn
ICC.Examples.T1Librarian>>checkIn:

Script depth: 3

Figure 16. The *Script Documentation* dialog.

Check method *ICC.Examples.T1Librarian>>checkIn:*, set **Script depth** to 3, then click **Preview...**

This produces a textual representation of the sequential flow of message sends within the initiating script. We could verify this by looking at the same code in a system browser.



```
Script for: ICC.Examples.T1Librarian>>checkIn:

 ICC.Examples.T1Librarian>>checkIn: aLendable
      Check in aLendable for its borrower. Answer the fine the user is charged.
calls
1 ICC.Examples.T1Lendable>>checkIn

      1 ICC.Examples.T1Lendable>>checkIn
            Calculate the overdue fine and charge the borrower if necessary. Then
notify
            the borrower that I'm in and forget him.
            Answer the overdue fine (may be zero).
      calls
      1.1 ICC.Examples.T1Lendable>>calculateOverdueFine
      1.2 ICC.Examples.T1Borrower>>payFine:
      1.3 ICC.Examples.T1Borrower>>checkIn:
```

Figure 17. The *Script Documentation Preview*.

Experiment with increasing the script depth of this script, then preview the results again. When the script depth is equal to 3 does it reflect the same paths as the documentation above?

Filtering Methods and Attributes

ADvance allows the user to exclude methods and attributes from diagrams that do not add value to the special view; the most obvious ones being accessors. This ***filtering*** is provided for two levels: ***diagram filters*** are applied to all classes in a diagram while ***class filters*** control filtering for individual classes in a diagram.The filtering also effects the tracing of scripts and messages.

Since filters are diagram specific, filtering can be used to easily customize the view of each diagram.

To edit the class filter select the class *ICC.Examples.T1Librarian* in the *Diagram Painter*, then select **Filter...** from the class pop-up menu.

This opens a *Class Filter Editor* on *ICC.Examples.T1Librarian*.

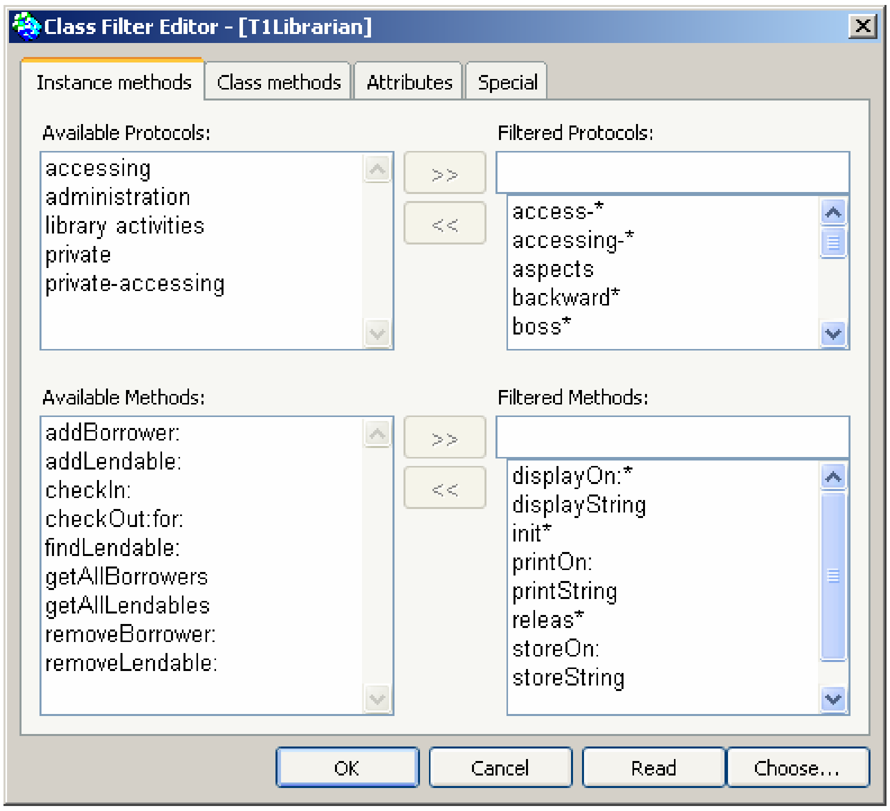


Figure 18. The *Class Filter Editor* on *ICC.Examples.*T1Librarian.

The *Diagram Filter Editor* and *Class Filter Editor* provide interfaces to add and remove method protocols and single methods or attributes to/from a diagram or class filter. Each editor contains four filtering categories: **Instance methods**, **Class methods, Attributes**, and **Special**. The instance and class methods categories contain two sets of panes to display filtered protocols and methods respectively. With **Protocol** panes filter, all methods of a filtered protocol become filtered, with **Method** panes individual methods can be filtered. The right hand panes display the filtered protocols and methods. The attributes filter category is similar to the method categories, providing panes for instance variables, class instance variables, and shared variables. The **Special** page has predefined semantic filters that can be checked for filtering as a whole.

Select the *library activities* protocol, click the **>>** button, then select **OK**.

The protocol is moved to the **Filtered Protocols** pane and the three methods belonging to this protocol are removed from the **Available Methods** pane.

Now we go back and look at the updated diagram view. Notice that the three methods on *ICC.Examples.T1Librarian* are no longer drawn in the diagram - even though the diagram filter did not filter these. These methods are no longer available to trace either.

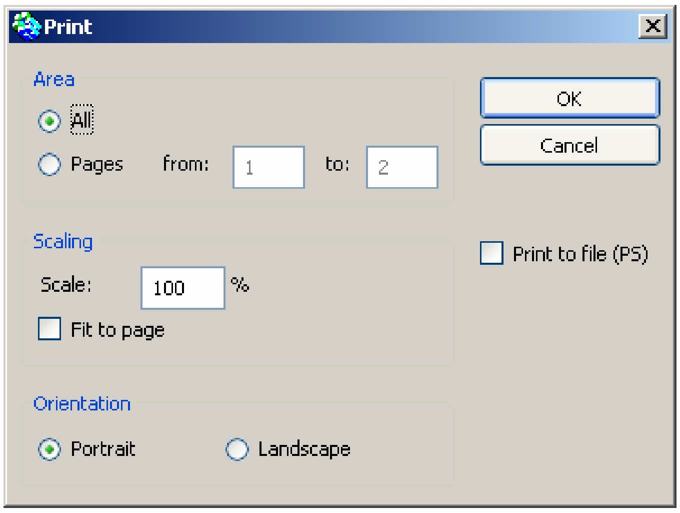
See the *Additional Concepts* chapter for more information about filtering.

Printing Diagrams

ADvance has a built-in diagram printing facility. It works with both, the VisualWorks' Postscript and Host printing[5].

To print the current diagram select **File→Print...** in the *Diagram Painter*.

The *Print Dialog* appears.



Area

[✓ All]

Pages from: 1 to: 2

Scaling

Scale: 100 %

☐ Fit to page

Orientation

[✓ Portrait] Landscape

☐ Print to file (PS)

Figure 19. The *Print Dialog.*

Choose an appropriate area, scaling and orientation (the diagram should fit on about 1 to 2 pages), then click **OK**.

The current diagram is sent to printer.

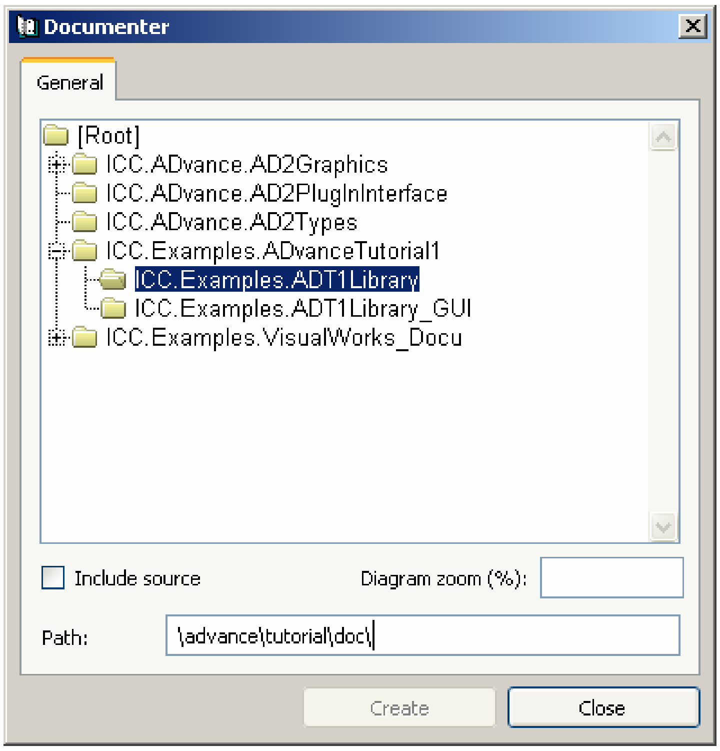
[5] You can change the VisualWorks printing setup in the Printing page of the VisualWorks settings.

Documenting Subjects

The *Documenter* is a tool that automatically creates a compound HTML documentation for a subject. The documentation includes the subject's diagrams, textual script representations, a HTML page for the table of contents, and optional definitions for the subject classes.

Open the *Documenter* by selecting **Window→Documenter** in the painter.

The *Documenter* interface appears.



```
[Root]
├ ICC.ADvance.AD2Graphics
├ ICC.ADvance.AD2PlugInInterface
├ ICC.ADvance.AD2Types
├ ICC.Examples.ADvanceTutorial1
│  ├ ICC.Examples.ADT1Library
│  └ ICC.Examples.ADT1Library_GUI
└ ICC.Examples.VisualWorks_Docu
```

☐ Include source   Diagram zoom (%):

Path: \advance\tutorial\doc\

Figure 20. The *Documenter* Interface.

Select subject *ICC.Examples.ADT1Library*, check **Include source** and enter a valid directory path in the input field. Then press the **Create** button.

Left the value for diagram zoom empty for normal resolution. Enter a value greater 100 to enhance resolution if the HTML documentation shall be printed later.

A documentation for the library example is generated under the given path.

Check the documentation with a HTML browser.

4 Reverse Engineering

Objectives

Demonstrate ability to build object model from existing code.

Derivation of behavior descriptions from existing code

Use as a tool for analyzing encapsulation, abstractions, architecture, etc.

Summarize Reverse Engineering steps.

In the previous chapter we worked with the *ICC.Examples.ADT1Library* example that has prepared **ADvance** diagrams already set up. There is a corrupted copy of this example that has no diagrams and that lacks some typing information. In this chapter we will reverse-engineer this copy. We will rebuild the diagrams that we've seen in the last chapter. You will explore how to produce and improve graphical abstractions from the pure code existing for this example.

We start creating an overview diagram of the library code. Then we will build a second diagram which provides a more detailed view on the same code. Finally we show how to explore behavior by building a third diagram.

Creating a Subject from existing Code

The first step for reverse engineering is the subject creation. The subject defines the subset of your system you want to view. The first subsection shows how to build a subject from scratch for different VisualWorks contexts like packages, parcels, and categories, or an empty subject to manually add the classes. An alternative way for building a subject for a package or for a single class' hierarchy is shown in the second subsection, describing the use of the integrated **ADvance** diagram painter in the *Refactoring Browser*, the default *VisualWorks* System Browser.

Subject creation with the Subject Wizard

In our case, the context of interest is the *ICC.Examples.ADT2Library* code. We will use this subject for all diagrams in this section.

Subjects are created from the *Subject Browser*.

Open a *Subject Browser* by selecting **Tools→Subject Browser** in the *Workbench* menu or click on the corresponding toolbar icon.

This opens a *Subject Browser*.

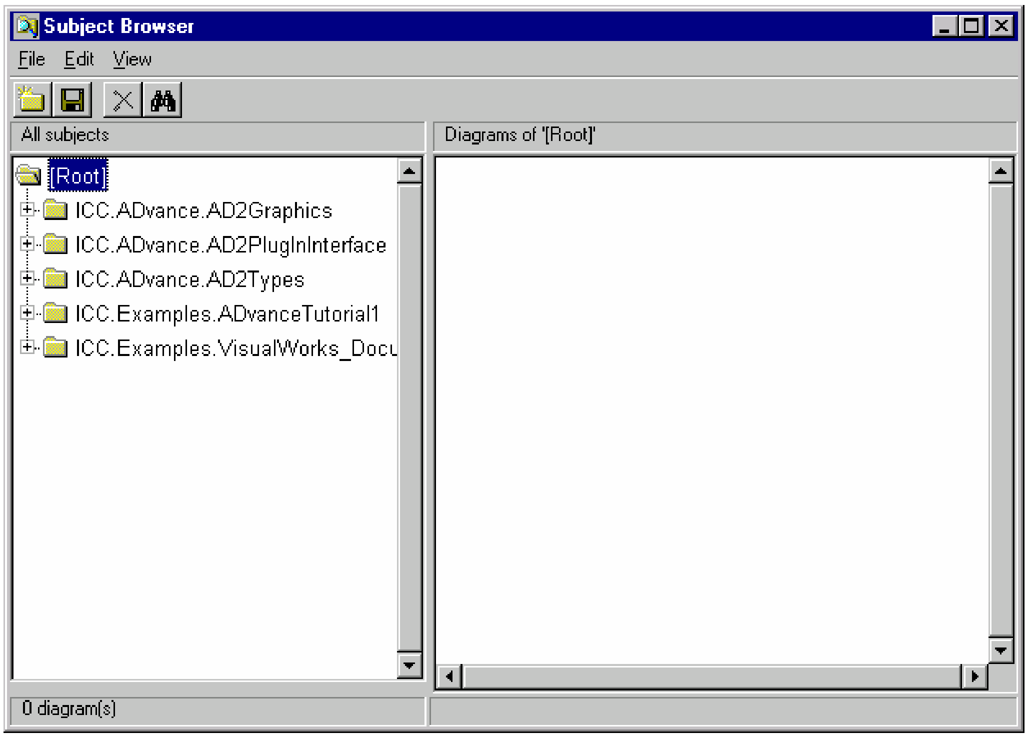


Figure 21. The *Subject Browser*.

The *Subject Browser* is much like the *Open Diagram* dialog that you've seen in the previous chapter. In the left pane it displays a tree of subjects, while the right pane shows diagrams, if a subject is selected. The browser has additional menu items and toolbar icons for managing subjects, i. e. creating, deleting, editing, and finding. It is also used to manage and to open diagrams.

We now will create a new subject from the library classes. The creation process itself is done by the *Subject Wizard*. The wizard guides through the creation process.

Note: New subjects are created as children of the currently selected subject. So, if you want to create a subject that should be placed under the root of the subject tree, you should select the *[Root]* subject first.

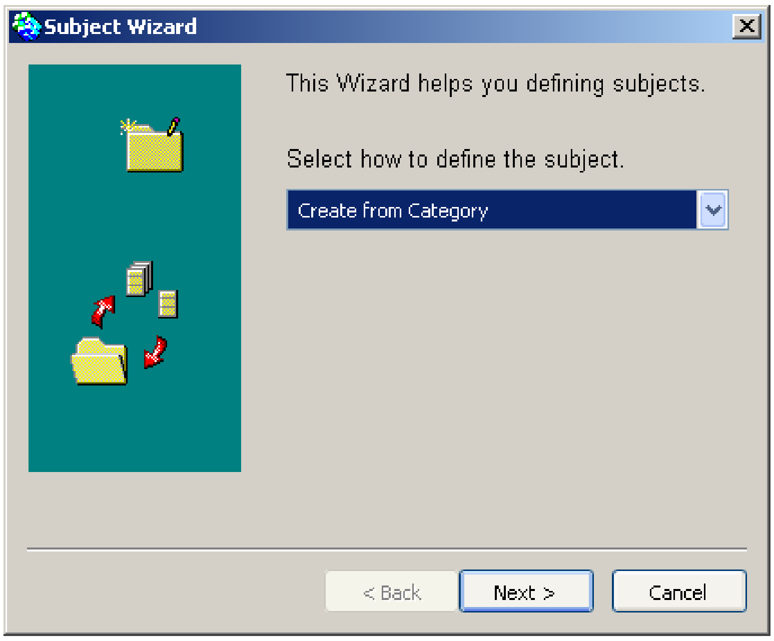
Select the *[Root]* subject in the *Subject Browser's* subject tree.

From the *Subject Browser*, select **File→New subject...**

The *Subject Wizard* is opened.

The wizard can create empty subjects or subjects with an initial contents. In the latter case you may either start with a *VisualWorks* category, a *VisualWorks* parcel, a StORE package, the classes in the change set, or you can copy all classes from an existing subject.

We will create a subject from the *VisualWorks* category *ADE-Tutorial 2*. The procedure is almost the same for the alternative subject creation methods.



This Wizard helps you defining subjects.

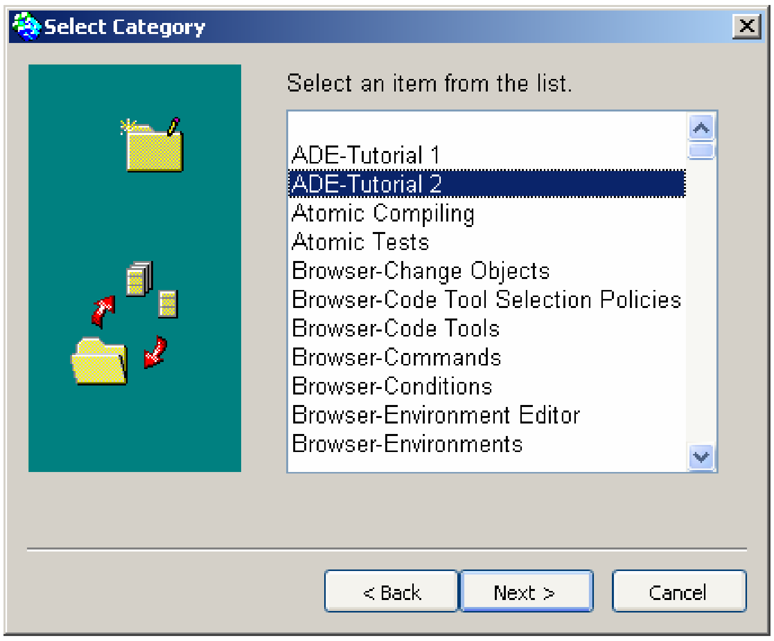
Select how to define the subject.

Create from Category

Figure 22. The *Subject Wizard* (1).

Select **Create from Category** then click **Next**.

A wizard page for choosing a category appears.



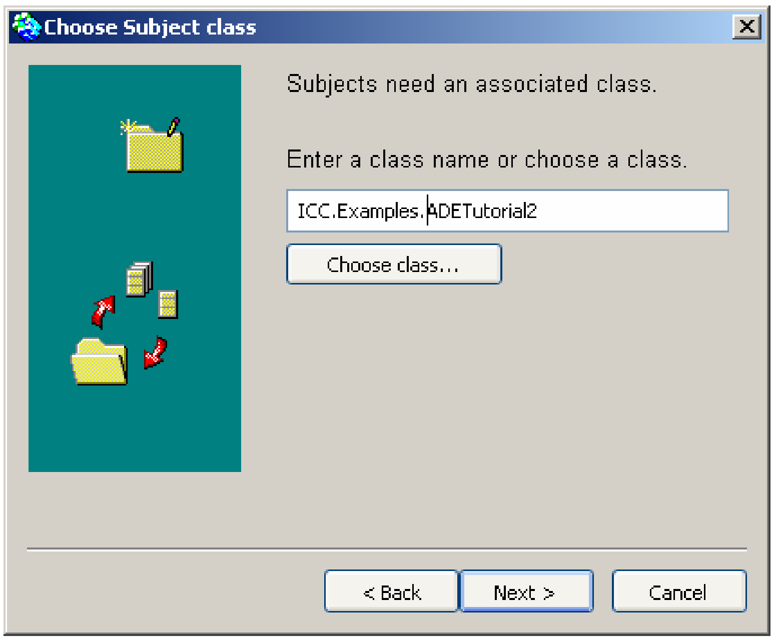
Select an item from the list.

ADE-Tutorial 1
ADE-Tutorial 2
Atomic Compiling
Atomic Tests
Browser-Change Objects
Browser-Code Tool Selection Policies
Browser-Code Tools
Browser-Commands
Browser-Conditions
Browser-Environment Editor
Browser-Environments

Figure 23. Selecting a category with the *Subject Wizard* (2).

Select category *ADE-Tutorial 2* then click **Next**.

A wizard page for choosing the subject class appears.



Subjects need an associated class.

Enter a class name or choose a class.

ICC.Examples.ADETutorial2

Figure 24. Choosing a subject class with the *Subject Wizard* (3).

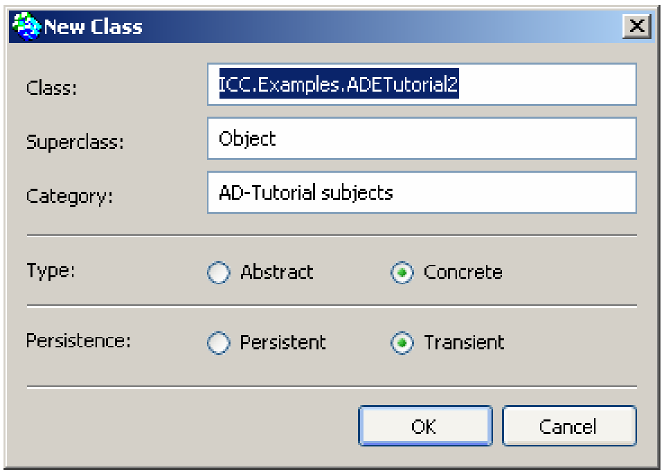
A subject needs a class to store the subject's definition and diagrams. You can either choose an existing class or create a new class. New classes can only be created for existing name spaces; if name space is omitted, class is created in default name space *Smalltalk*. The subject class is used to identify and transport subjects. It is also used to version subjects in team environments.

The subject name should reflect the name of the modeled subsystem or use case. It is restricted just like a VisualWorks class name.

In our case, we will go with the wizard's proposal *ADETutorial2*, extended by a special name space.

To proceed click **Next**.

Since *ICC.Examples.ADETutorial2* is a new class, the *New Class* dialog appears.



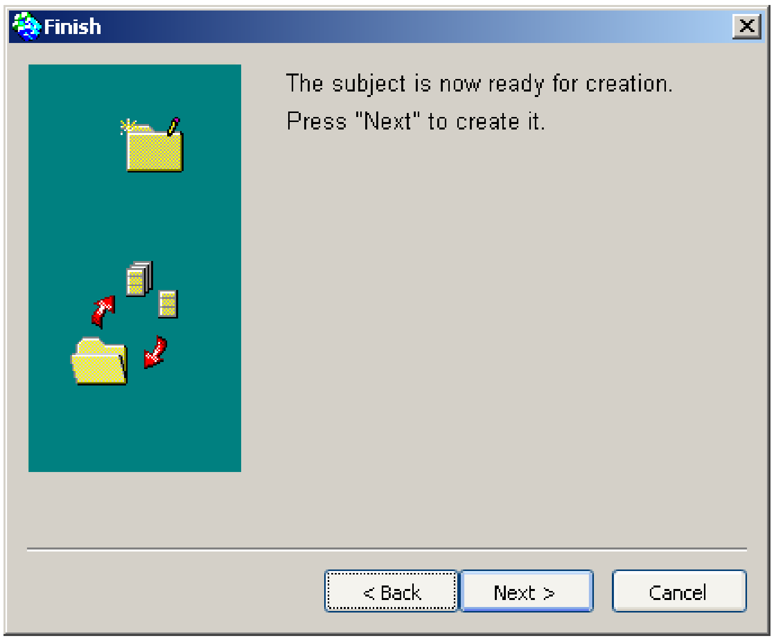
Class: ICC.Examples.ADETutorial2
Superclass: Object
Category: AD-Tutorial subjects
Type: Abstract [✓ Concrete]
Persistence: Persistent [✓ Transient]

Figure 25. The *New Class* dialog for class *ADETutorial2*.

Enter *ADE-Tutorial subjects* in the category input field, then click **OK**. Make sure that your subject class shows a fully qualified name space.

The package for the new class is determined by the general preferences (see section *General Preferences*). The default is to use the System settings, so normally the package defined as current package will be used, or you will be prompted for a package.

A class named *ADETutorial2* in namespace *ICC.Examples* is created and the last wizard page is opened.

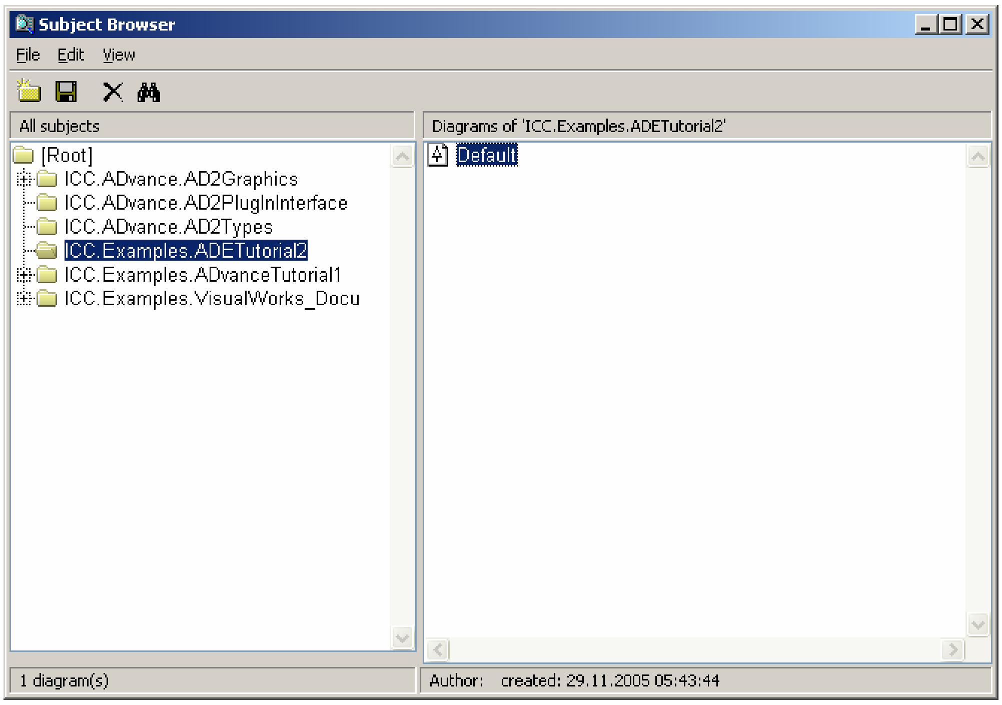


The subject is now ready for creation.
Press "Next" to create it.

Figure 26. The *Subject Wizard* (3).

Click **Next** to finish the subject creation process.

The subject *ADETutorial2* is created and shows up in the *Subject Browser*.



[Root]
ICC.ADvance.AD2Graphics
ICC.ADvance.AD2PlugInInterface
ICC.ADvance.AD2Types
ICC.Examples.ADETutorial2
ICC.Examples.ADvanceTutorial1
ICC.Examples.VisualWorks_Docu

Default

1 diagram(s)
Author: created: 29.11.2005 05:43:44

Figure 27. The new subject and its default diagram.

ADvance in the Refactoring Browser

When the package or parcel named *AD2RefactoringBrowser* is present in the image, the *VisualWorks* System Browser shows the tab **ADvance Diagram** when a class or a package is selected. For a parcel, the diagram on the tab contains all classes defined or extended in the package. For a class, the class and each member of its class hierarchy (superclasses and subclasses) are shown.

The generated subjects and its diagrams are not automatically stored in the image as special classes and methods. If you edit the diagram (e.g. adding classes, or changing the view), you will be asked on leaving whether to save the changed diagram. You can save the diagram even without a prior change using the disk symbol of the toolbar.

The subject information and the diagram information of the package subject are then stored in a new class; a *New Class Dialog* will open to enter the class information. The subject information and the diagram information of the single class hierarchy subject are stored in that class itself. As well as for wizard created subject classes, the packages for the new class respectively the new methods are determined by the policies defined in the *General Preferences*.

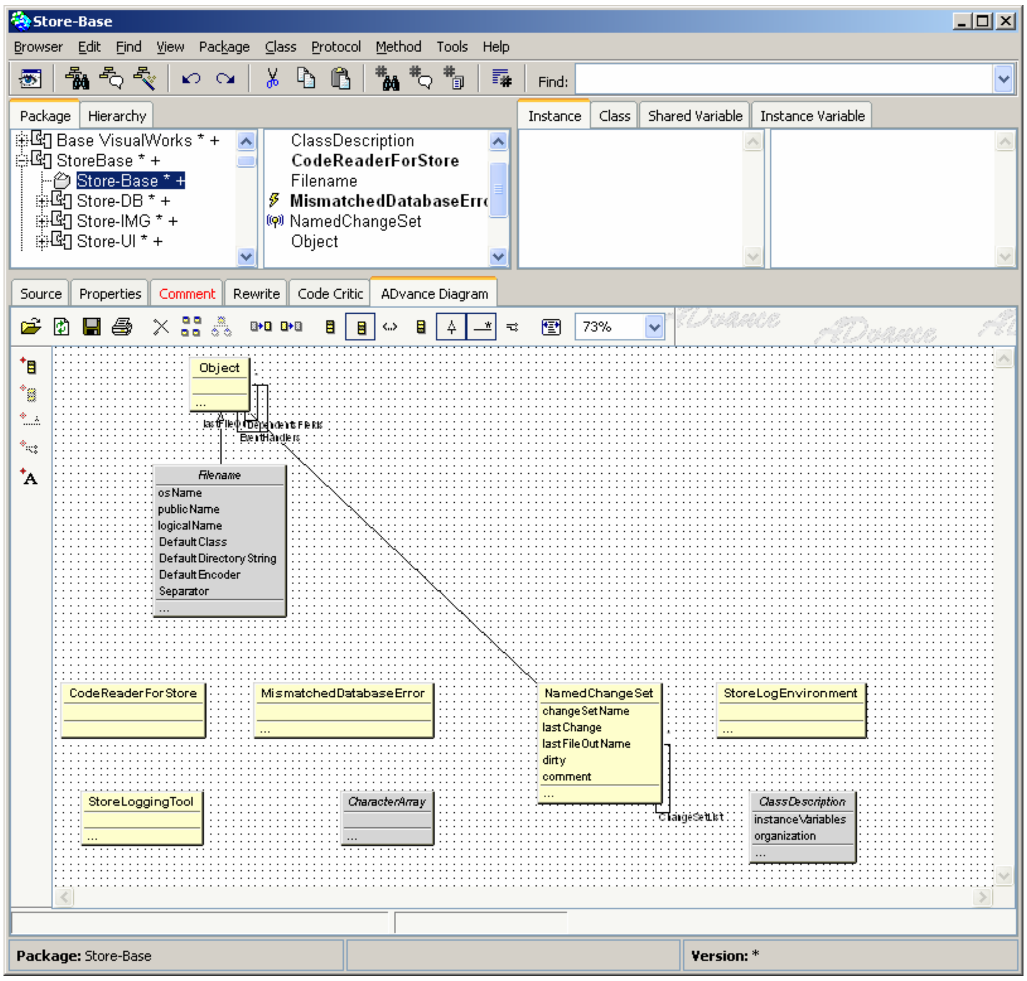
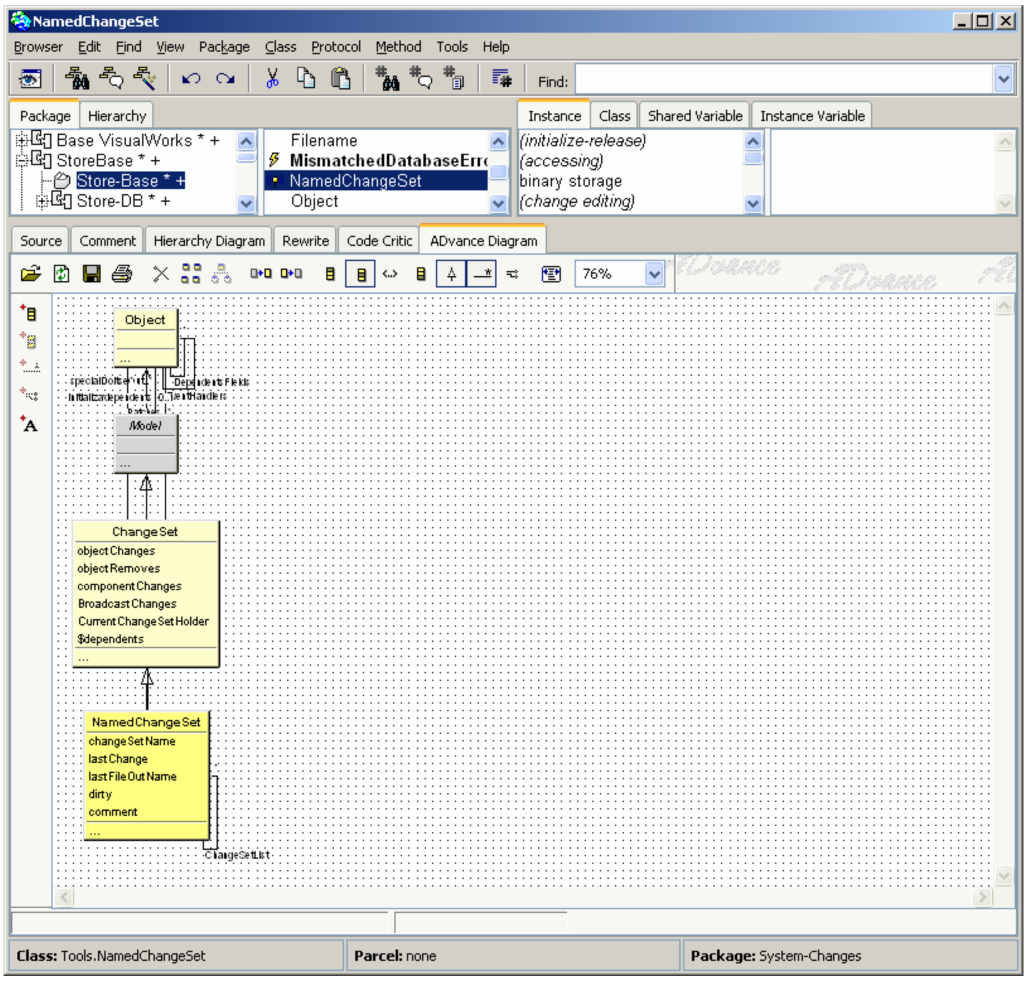


Figure 28. *Refactoring Browser* diagram for a package context



Class: Tools.NamedChangeSet | Parcel: none | Package: System-Changes

Figure 29. *Refactoring Browser* diagram for a class hierarchy.

Creating an Overview Diagram

We now will create a diagram for a structural view on the new subject. **ADvance** has already created a *Default* diagram for our manually created subject. We will open, modify and save it.

Select *ADETutorial2* in the *Subject Browser*.

The *Default* diagram is displayed in the subject's diagram list.

Select the *Default* diagram as shown in the figure above.

Open the *Default* diagram by double-clicking on it or by selecting **Open** in the corresponding pop-up menu.

A *Diagram Painter* is opened and shows an initial layout for the classes in the *ADETutorial2* subject.

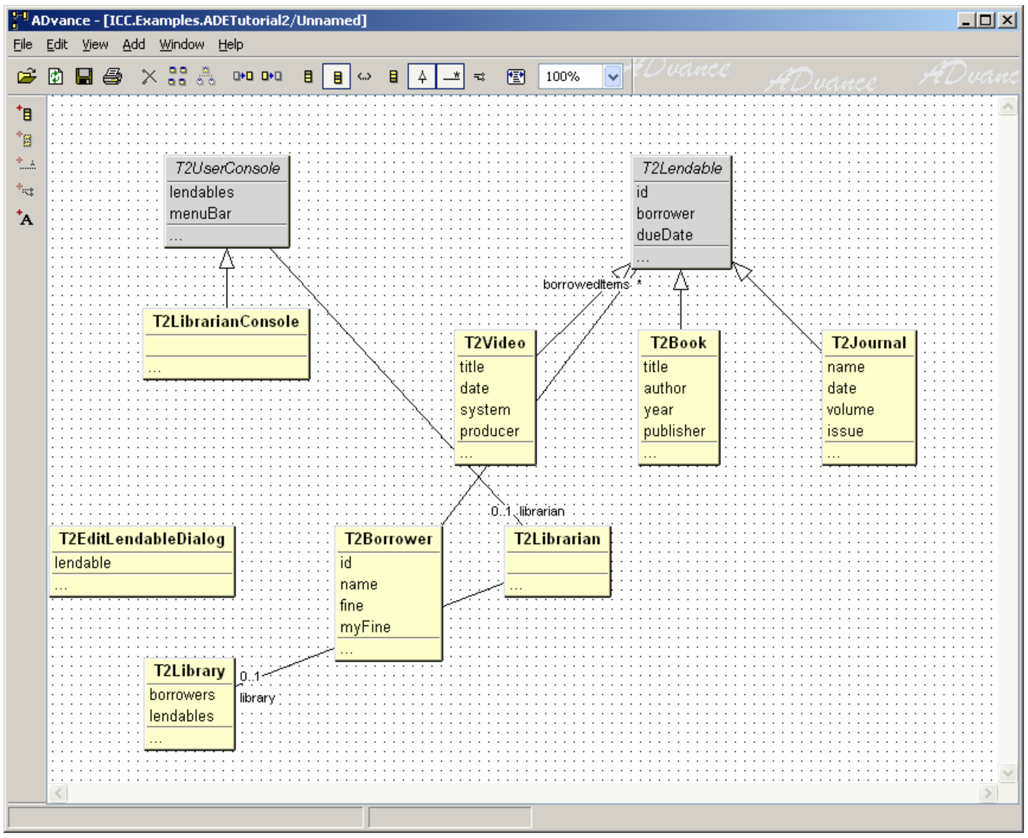


Figure 30. The *Diagram Painter* shows an initial layout.

Before we save this diagram we will improve the layout and customize the tool to give a better presentation of the subject.

Resize the painter window and change the zoom so that you can both read the texts in the diagram and have an overview of the layout.

To improve the layout, rearrange the classes as shown in figure below.

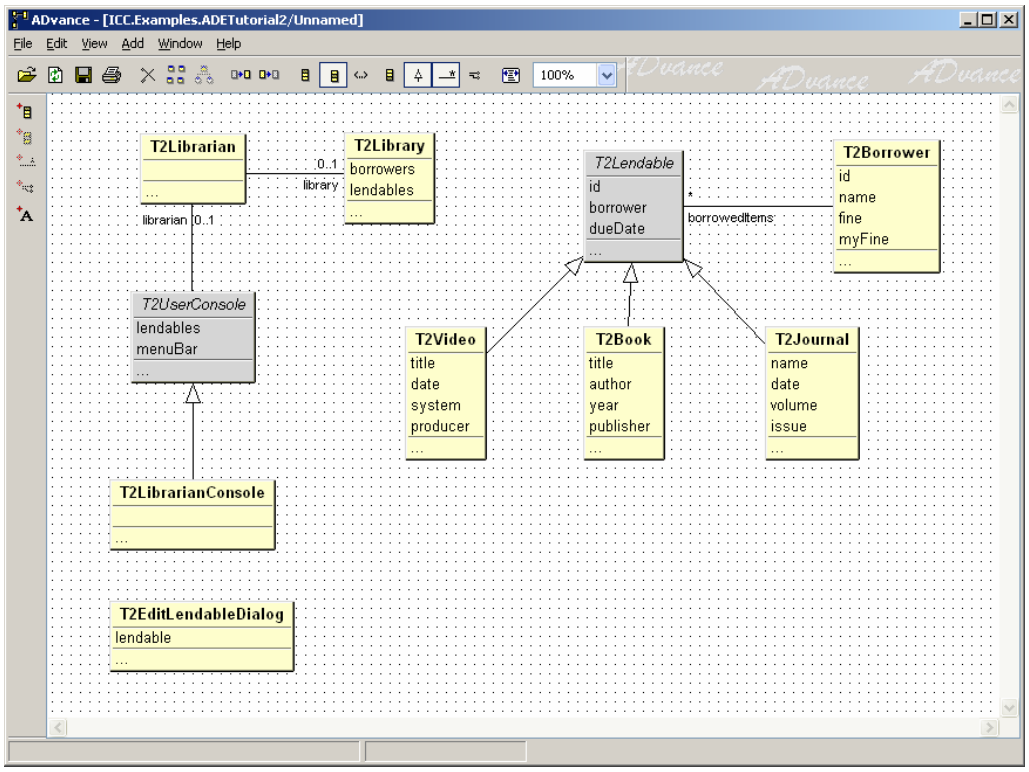


Figure 31. The resized and rearranged diagram.

Do a **File→Save As...** to name and save it. Name the diagram *Overview*.

The diagram is saved and the painter's window label changes.

Since we want to build a structural view of the librarian example, we will hide the GUI classes *T2UserConsole* and the *T2EditLendableDialog*. Furthermore we will add typing information to the diagram by switching on the corresponding layers.

Select class *T2UserConsole*, then choose **Hide** from the class pop-up menu.

The class becomes invisible in the diagram but is still part of the diagram's subject.

Remove classes *T2EditLendableDialog* and *T2LibrarianConsole* from the diagram by pop-menu item **Delete**.

The classes are removed from subject and therefore also removed from default diagram, but they are still in the image.

To add more structure information to the diagram switch on the layer for attribute types by pressing the button in the toolbar.

The resulting diagram includes attribute types.

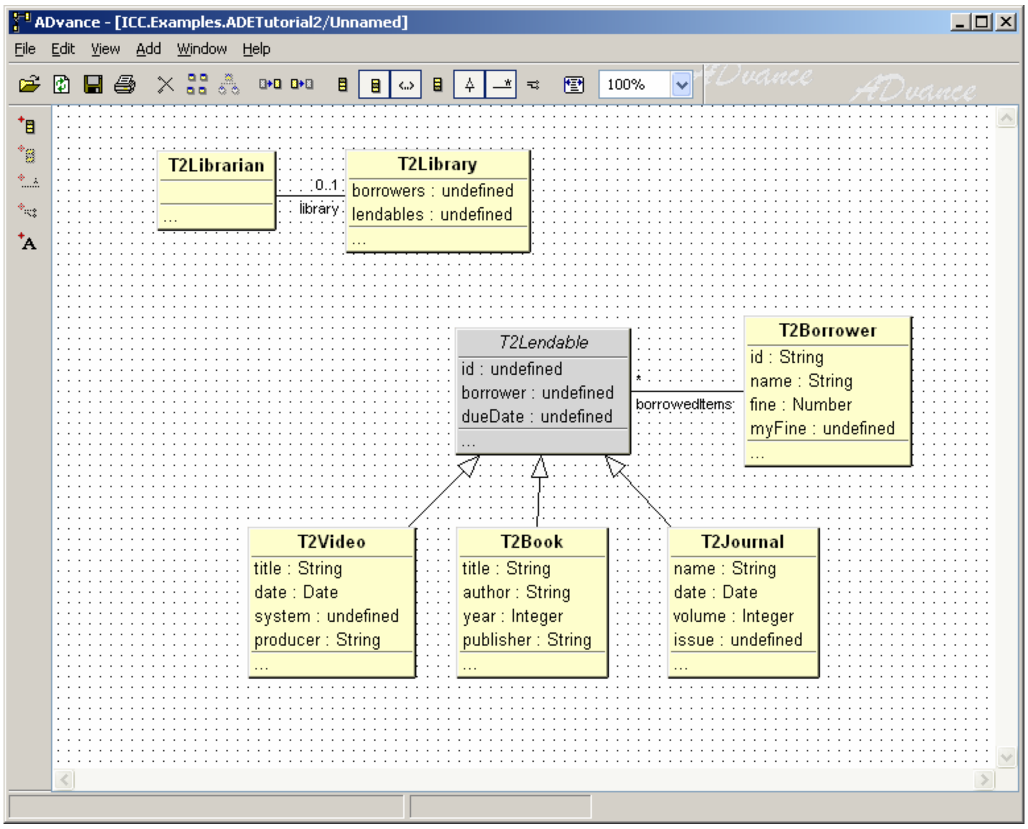


Figure 32**.** A draft for the overview diagram.

Revisiting the Class Comments

In the current example several classes have undefined types. In order to create a diagram that will convey a more comprehensible message to the reviewer, we will now improve the type information. This will lead to more relation lines and better attribute descriptions.

ADvance uses static variable annotations which are extracted from class comments or from special methods providing these annotations. The class comment format is enforced by the VisualWorks *Class Reporter*[6]. See also the type syntax as described in the method Parser>>typeExpression.

[6] Contained in the VisualWorks *Advanced Tools* parcels.

Note: You can define a selector for the class methods containing the annotations by editing the *General Preferences*. This can be used to store documentation in extensions to classes to avoid modifications. The default method selector is *#ad2ClassInfo*. If preferences are set to take this method and a class does not already have such a method, its comment will be taken instead. The way how to edit preferences is discussed later in this guide.

Static variable annotations are interpreted as attribute types respective associations. For a discussion of possible semantic differences and subtleties going with these notions please refer to the UML reference books. VisualWorks system classes are almost completely annotated using this standard. Thus, **ADvance** can build up a good approximation of the system’s static association structure.

There are several ways to check, add and edit type annotations. In the next paragraphs we will introduce the most frequently used approaches to easily manage types.

Checking Types

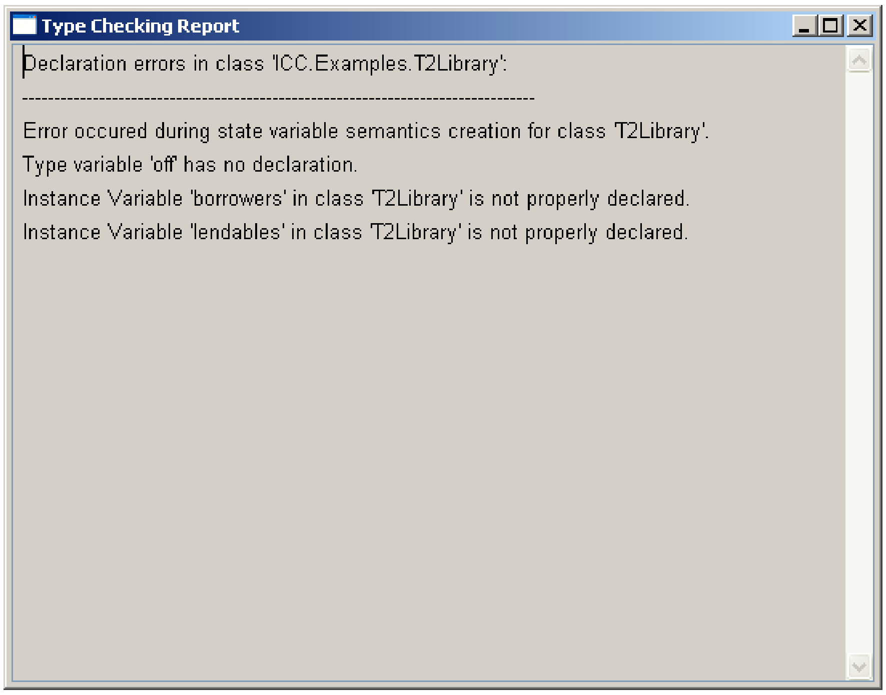
Statically declared type relationships can be checked using:

1. **ADvance**’s built-in type checking facility: This is a very detailed type check and is recommended if you are checking a single class. Just select the classes you would like to have checked and let the checker work.
2. The VisualWorks *ClassReporter*: This method is recommended if you are checking multiple classes for several consistency properties.
3. The type reporting from the *Comment Generator* plug-in: This is the adequate way to easily check multiple classes for comment consistency only.

ICC.Examples.T2Library has two attributes *borrowers* and *lendables* which are shown to be undefined. We will check the problem with **ADvance**’s built-in type checking facility.

🖱 Select class *ICC.Examples.T2Library*, then select **Utilities→Check types...** from the class pop-up menu.

↳ Since there are invalid types, a type checking report is opened.



```
Declaration errors in class 'ICC.Examples.T2Library':
--------------------------------------------------------------------------------
Error occured during state variable semantics creation for class 'T2Library'.
Type variable 'off' has no declaration.
Instance Variable 'borrowers' in class 'T2Library' is not properly declared.
Instance Variable 'lendables' in class 'T2Library' is not properly declared.
```

Figure 33. A type checking report on *ICC.Examples.T2Library*.

Editing Types

There are several ways to create and edit types:

1. **ADvance**'s *Class Properties* dialog: This is the easiest way to edit type annotations for a class visible in the *Diagram Painter*. It is recommended for novice users.
2. *Standard Browser*: This is a fast way to edit multiple type annotations in one class.
3. *Comment Generator* plug-in: This tool automatically infers variable types through a static inference step and generates preformatted class comments for a set of classes. It is the recommended tool if your subject classes have no comments at all.

We will correct the types for *ICC.Examples.T2Library* with a standard browser and we will modify the type annotation for *ICC.Examples.T2Lendable.borrower* using the *Class Properties* dialog.

<Shift>-double-click on class *ICC.Examples.T2Library*.

A *Hierarchy browser* on *ICC.Examples.T2Library* is opened.

Change the comment as follows: the type description of *borrowers* and *lendables* must read <OrderedCollection of: ICC.Examples.T2Borrower>

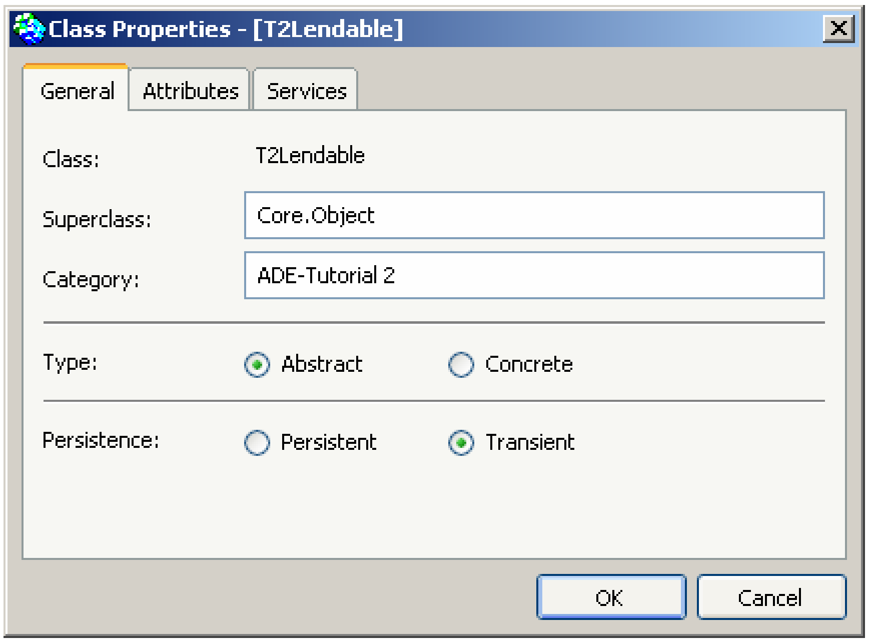
respectively. <OrderedCollection of: ICC.Examples.T2Lendable>. Then close the browser.

Press the update button ( ) in the painter or select **File→Update**.

The *Diagram Painter* now displays the updated diagram with relations inserted between *ICC.Examples.T2Library* and *ICC.Examples.T2Lendable*, *ICC.Examples.T2Borrower*.

Double click on class *ICC.Examples.T2Lendable* to open the *Class Properties* dialog.

A dialog is opened that lets you easily modify class and attribute definitions.



Class: T2Lendable

Superclass: Core.Object

Category: ADE-Tutorial 2

Type: [✓ Abstract] Concrete

Persistence: Persistent [✓ Transient]

Figure 34. The *Class Properties* dialog on *ICC.Examples.T2Lendable*.

Switch to the dialogs attributes page by clicking on the **Attributes** tab.

Select the *borrower* attribute, then press the **Edit** button.

A requester appears and asks for the attribute type.

Enter **ICC.Examples.T2Borrower**, press **OK**.

Close the *Class Properties* dialog.

The diagram is updated and now has a relation line from *ICC.Examples.T2Lendable* to *ICC.Examples.T2Borrower*.

Finally, make a few more changes to see how the graphics reflect the current code: Add a class comment to *ICC.Examples.T2Lendable* with the following attribute types: *id* <Integer>, *dueDate* <Date>. Remove the instance variable *myFine* from *ICC.Examples.T2Borrower*. Then update the diagram again.

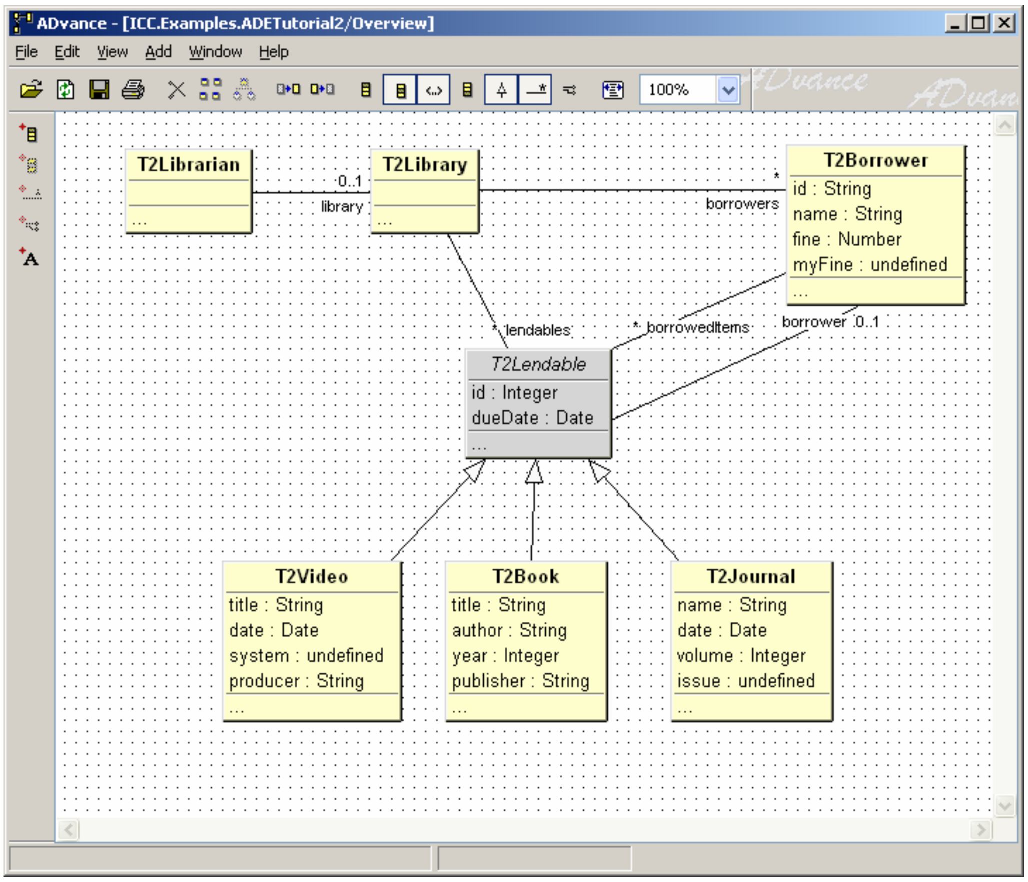


Figure 35**.** The updated *ADETutorial2/Overview* diagram.

Note: The default filter may suppress information you would like to see. If you suspect such unwanted suppressing, edit the diagram's filter by menu action **Edit → Filter** with no class selected, and start with removing the selections on the **Special** tab. For the diagram above, this would add the *Default* variable as a relation from and to *T2Library* in the diagram.

Creating a Detailed Diagram

As a next step, we will extend the *Overview* diagram to get a new diagram that presents more details about the library application. Therefore we

1. Add relevant classes to the subject,
2. Apply filters to fine tune services in the diagram.

Revisiting the Subject Contents

Adding collaborators and removing irrelevant classes is an important step to gain a better understanding of an application. In this section we will learn three different approaches for adding a collaborator class to the subject.

🖱 To copy the diagram choose **File→Save as...** and save it as *Details*.

Adding a related Class

Let’s navigate through relationships and add a class that is related to a subject class.

🖱 Select **View→All classes**, to make all classes in the diagram visible.

🖱 Select class *ICC.Examples.T2UserConsole*, then select **Add→Related classes...**

➥ A list of classes that are related to *ICC.Examples.T2UserConsole* is presented.

🖱 Select *ICC.Examples.T2LibrarianConsole* and click **OK**.

➥ This class is added to the subject. The diagram automatically reflects this.

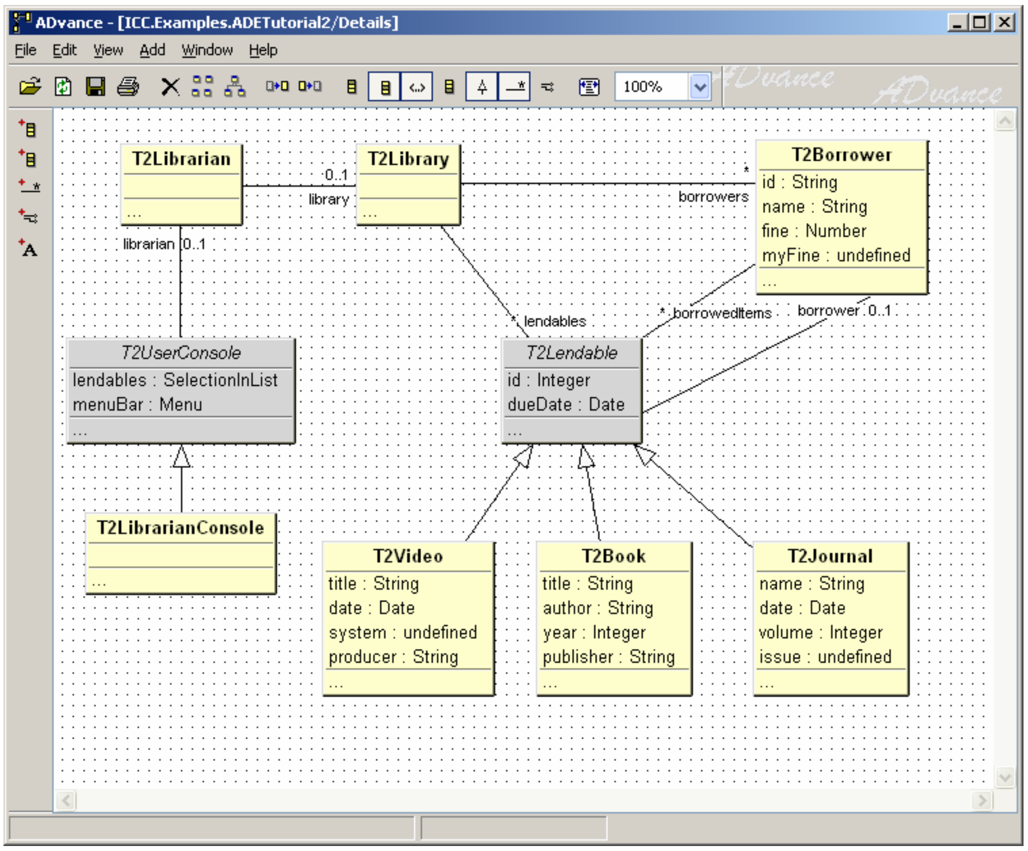


Figure 36. The updated *ADETutorial2/Overview* diagram.

Adding an Initiator Class

Now we temporarily remove *ICC.Examples.T2LibrarianConsole* to add it again with a different approach. This time we will navigate over behavior instead of relationships.

Select *ICC.Examples.T2LibrarianConsole*, then select **Delete** from the class pop-up menu.

After a request this will delete class *ICC.Examples.T2LibrarianConsole* from the subject but not from the image.

Select *ICC.Examples.T2Librarian* then select **Add→Initiators...**

The system does a behavior inference and answers a list of classes that send messages to the *ICC.Examples.T2Librarian*. Among them you find the *ICC.Examples.T2LibrarianConsole*.

Select *ICC.Examples.T2LibrarianConsole*, then click **OK**.

↳ This class is added to the subject again.

In the same way we added an initiator class, we could add participant classes to the subject by **Add→Participants...**

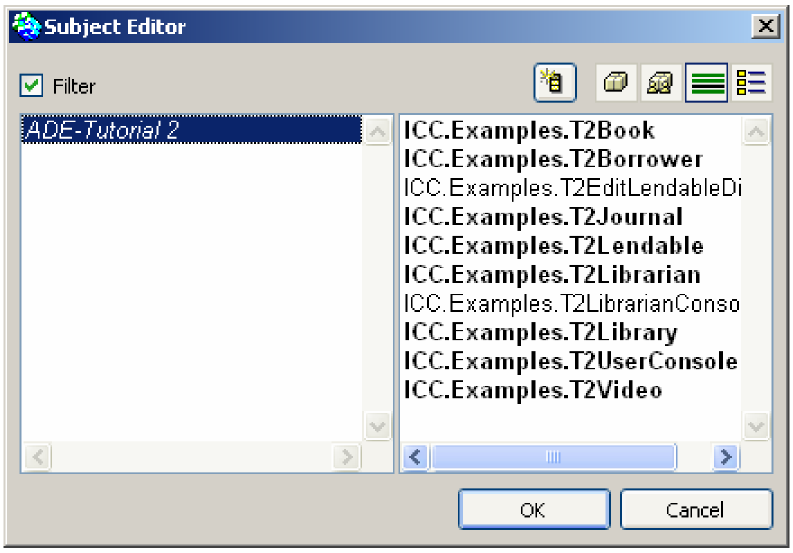
Adding a Class with Subject Editor

Again, we temporarily remove *ICC.Examples.T2LibrarianConsole* from the subject and then we add it again. This time we use the *Subject Editor* to modify the subject.

The *Subject Editor* provides an interface for adding and removing classes to/from a subject. It is the recommended way to change a subject, if you already have an understanding, which classes are relevant and which are irrelevant.

To open the *Subject Editor* double-click in the diagram (which is a shortcut for **Edit→Subject...**).

↳ The *Subject Editor* then appears and depicts which classes are contained in the subject and which are not.



ADE-Tutorial 2

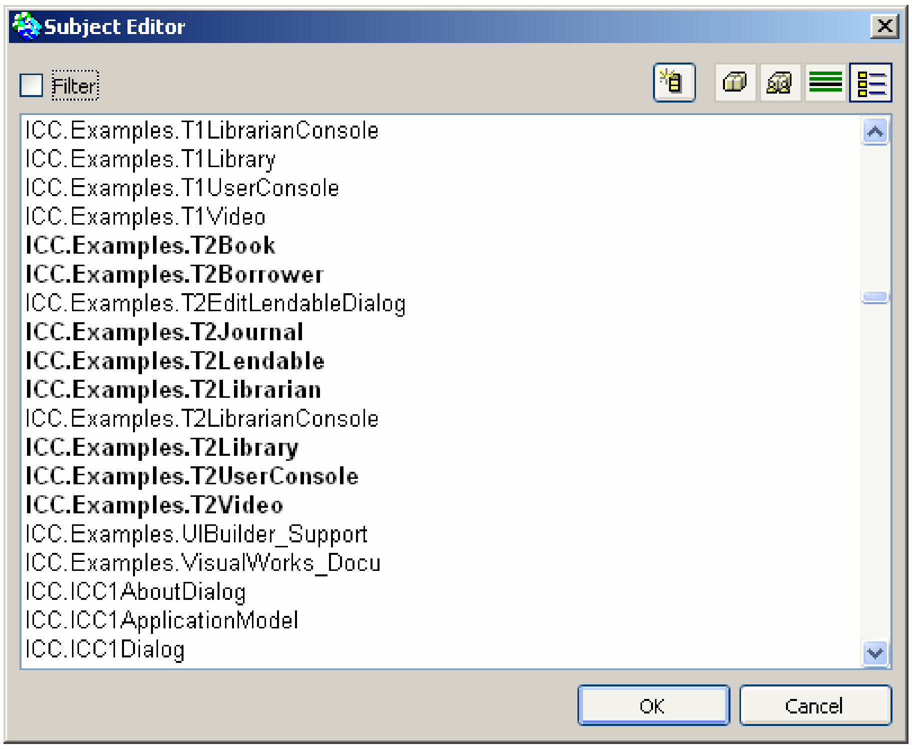
ICC.Examples.T2Book
ICC.Examples.T2Borrower
ICC.Examples.T2EditLendableDi
ICC.Examples.T2Journal
ICC.Examples.T2Lendable
ICC.Examples.T2Librarian
ICC.Examples.T2LibrarianConso
ICC.Examples.T2Library
ICC.Examples.T2UserConsole
ICC.Examples.T2Video

Figure 37**.** The *Subject Editor* in category mode, **Filter** checked.

The *Subject Editor* has four different views to modify the subject. The **category view** offers two panes, displaying all categories in the left pane and the contained classes of the selected category in the right pane. This view is best if you want to add and remove categories, or if you want to find a class by category. The other views from left to rigth are **parcel view**, **package view**, and **alphabetical view** which offers a sorted list of all classes.

In all views containters and classes that are already in the subject are shown in bold. Categories that are partly in the subject are shown in italics. If the **Filter** option is checked, the editor shows only containers (parcels, packages, categories) or classes (in alphabetical view) that are at least partly in the subject.

You can add and remove containers or classes by a variety of actions in the pop-up menues of the different panes. They allow you to add/remove single items or using patterns. The editor also provides actions to add classes from the change set, a VisualWorks parcel or other subjects. Furthermore you can add and remove a class' super and subclasses.



ICC.Examples.T1LibrarianConsole
ICC.Examples.T1Library
ICC.Examples.T1UserConsole
ICC.Examples.T1Video
ICC.Examples.T2Book
ICC.Examples.T2Borrower
ICC.Examples.T2EditLendableDialog
ICC.Examples.T2Journal
ICC.Examples.T2Lendable
ICC.Examples.T2Librarian
ICC.Examples.T2LibrarianConsole
ICC.Examples.T2Library
ICC.Examples.T2UserConsole
ICC.Examples.T2Video
ICC.Examples.UIBuilder_Support
ICC.Examples.VisualWorks_Docu
ICC.ICC1AboutDialog
ICC.ICC1ApplicationModel
ICC.ICC1Dialog

Figure 38. The *Subject Editor* in alphabetical mode, **Filter** unchecked.

Select *ICC.Examples.T2LibrarianConsole* then select **Add** from the pop-up menu.

The *ICC.Examples.T2LibrarianConsole* is now in the subject and is shown in the subject's diagrams.

Applying Filters

To gain a better understanding of the services provided by the library classes, we will switch on the services layer. Since this may lead to a cluttered diagram we will then apply service filters. With filters you can fine tune the presentation of services in your diagrams.

Switch on the services layer by checking the toolbar checkbutton ( ) or by checking **View→Layers→Services**.

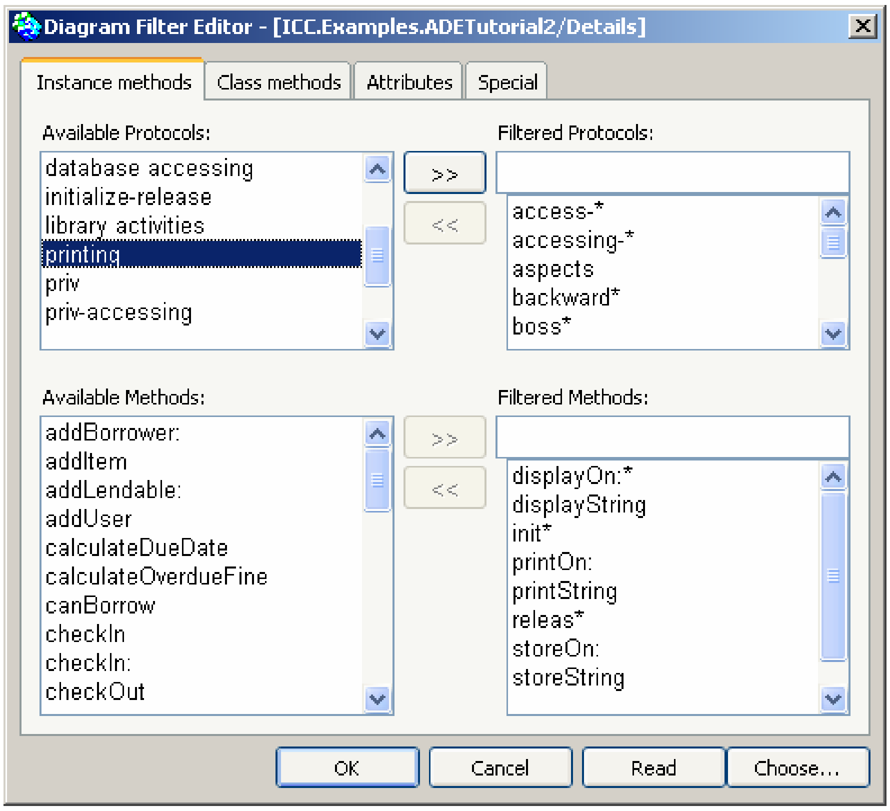
↳ The services for all classes in the diagram will show up.

The diagram contains a lot of services that do not add value to it. Among these are class methods (indicated by the $-prefix) and printing method, e.g. *shortDisplayString*.

We first will change the **diagram filter** to exclude these from the diagram.

Have all classes deselected, then select **Edit→Filter...**.

↳ The *Diagram Filter Editor* will be opened.



Available Protocols: database accessing, initialize-release, library activities, printing, priv, priv-accessing

Filtered Protocols: access-*, accessing-*, aspects, backward*, boss*

Available Methods: addBorrower:, addItem, addLendable:, addUser, calculateDueDate, calculateOverdueFine, canBorrow, checkIn, checkIn:, checkOut

Filtered Methods: displayOn:*, displayString, init*, printOn:, printString, releas*, storeOn:, storeString

Figure 39. The *Diagram Filter Editor*.

Move the *printing* protocol from **Available Protocols:** to **Filtered Protocols:** by clicking on the upper **>>** button. Check the **Filter class methods** check box in the **Special** tab, then click **OK**.

↳ The methods in *printing* protocols and the class methods are no longer displayed in the diagram.

We now will change the **class filter** for *ICC.Examples.T2Librarian*.

Select *ICC.Examples.T2Librarian* then select **Edit→Filter...**

↳ A *Class Filter Editor* on *ICC.Examples.T2Librarian* will be opened.

🖱 Filter protocols *accessing* and *administration*.

↳ *ICC.Examples.T2Librarian* is displayed with three services only.

Note: Class filters override diagram filters. For example, if the protocol *private* is filtered at the diagram level but not filtered at the individual class level, the class having this protocol unfiltered shows these methods and has them included in script and association traces.

ADvance comes with predefined filters.These are built to match the Smalltalk protocol and method naming conventions as described in the *VisualWorks User's Guide*. If your methods comply with these naming conventions, the filters will automatically hide methods that typically do not add value to diagrams.

Class *ICC.Examples.T2UserConsole* has protocols that do not comply with these conventions. Hence, the diagram contains some methods that are irrelevant. These protocols could be filtered, but – in this case – the better way is to comply with the standards.

<Shift>-double-click class *ICC.Examples.T2UserConsole* to open a browser. Rename the protocols *priv* and *priv-accessing* to *private* and *private-accessing* respectively. Close the browser and update the diagram. Rearrange the classes.

The diagram should look as shown below.

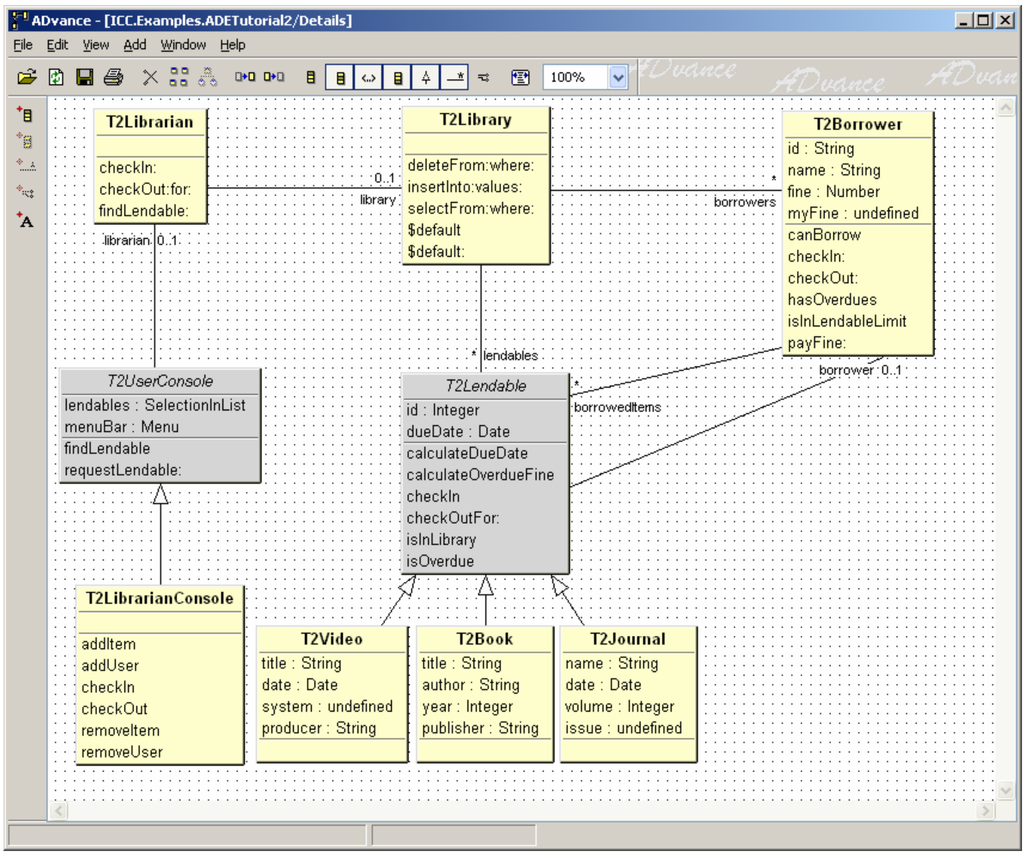


Figure 40**.** The updated *ADETutorial2/Details* diagram.

Although all classes are defined within the same namespace you may wish to show the namespace in the header of the classes.

Switch on the namespace layer by checking the toolbar checkbutton ( ) or by unchecking **View→Layers→Namespaces**.

The namespaces for all classes in the diagram appear.

You can also toggle namespace for a selected class by using **Layers→ Hide / Show Namespace** from Class pop-up menu.

Creating a Behavior Diagram

We now will create a behavioral diagram for our example. Therefore we will recreate the behavior diagram for the librarian check-in process. We will examine two different ways to explore behavior.

- First we follow the message path step by step, creating single scripts.
- We then will create a nested message path in one step.

To prepare, let's focus on the service layer. As soon we've created scripts in this diagram, we will save it under a new name.

Using the toolbar buttons, switch off attributes, types, inheritance and relations layers, and switch on the scripts layer for the diagram.

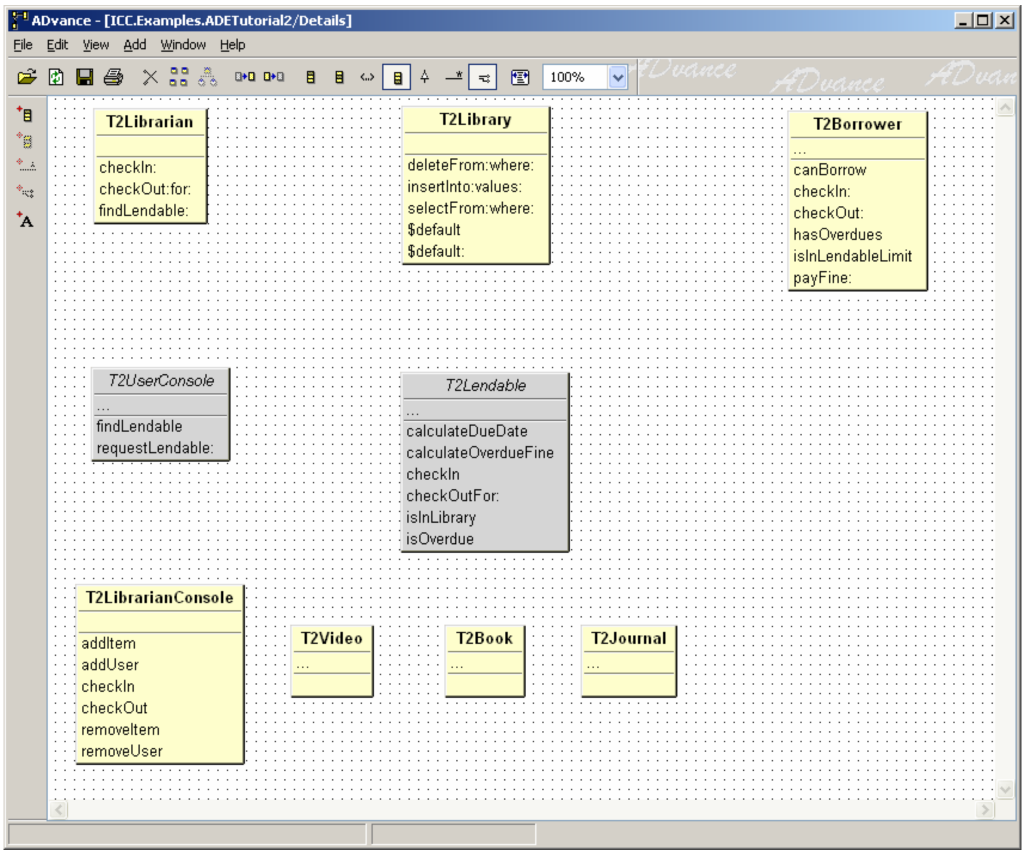


Figure 41. A draft for the behavior diagram.

Creating single Scripts

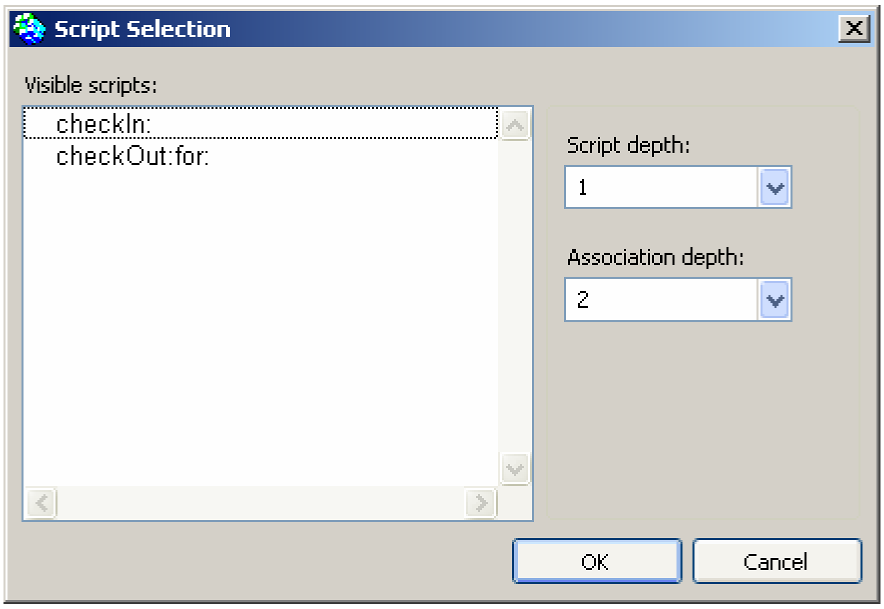
Select *ICC.Examples.T2Librarian*, select **Scripts...** from the class pop-up menu.

The *Script Selection* dialog is opened.

The dialog's **Visible scripts** pane contains all services that send messages to classes in the diagram. For any service that you check, the tool will generate a message trace. If you uncheck a service, the corresponding message lines will be removed from the diagram.

The message tracing is parameterized by two options. The **Script depth** parameter determines how deep the message path should be traced. In this section we will use a depth of 1; this will show the direct receivers of messages. In the next section we then will increase the script depth to automatically dive into the message path.

The **Association depth** indicates which related classes are to be included in the trace. See the sidebar *How ADvance infers Message Passing* below for further explanations.



Script depth: 1

Association depth: 2

Figure 42. The *Script Selection* dialog.

Check *checkIn:*, set **Script depth** to 1, then click **OK**.

The diagram then displays a message send from *ICC.Examples.T2Librarian>>checkIn:*.

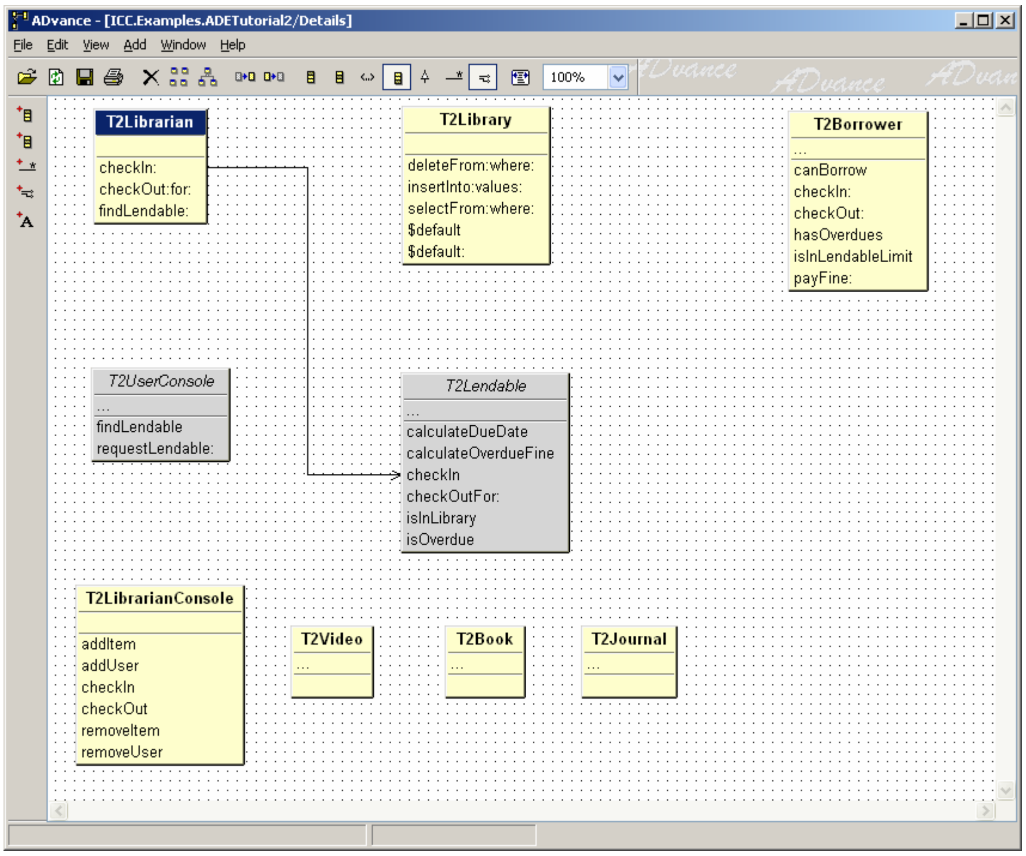


Figure 43. A script for *T2Librarian>>checkIn:*.

How ADvance infers Message Passing

ADvance uses a behavioral inference to determine which are the potential receivers of a message send. It infers all potential participant services for the originating method. For gaining a high degree of accuracy, it then restricts the resulting set twice. First it is restricted to subject classes. Furthermore it is restricted to classes that are directly or indirectly related to the initiator. The idea behind the latter is that – in most cases - classes send messages to related classes only.

The second restriction step requires type information. This is why variable typing is imperative for the tool to work most accurately. In order to improve behavior analysis, it is recommended to augment a diagram with association annotations which are derived from the initiators/participants function.

With the **Association depth** you can control how deep the tool should look for related classes of the initiator. It is used to define the depth of the transitive closure of related classes. A depth of 1 will use directly related classes only. A depth of 2 will add the related classes of directly related classes, and so on.

We now will follow the message trace and add a script for *ICC.Examples.T2Lendable>>checkIn*.

Select *ICC.Examples.T2Lendable*, open the *Script Selection* dialog and check *checkIn*.

The diagram draws message lines to participants of *ICC.Examples.T2Lendable>>checkIn*.

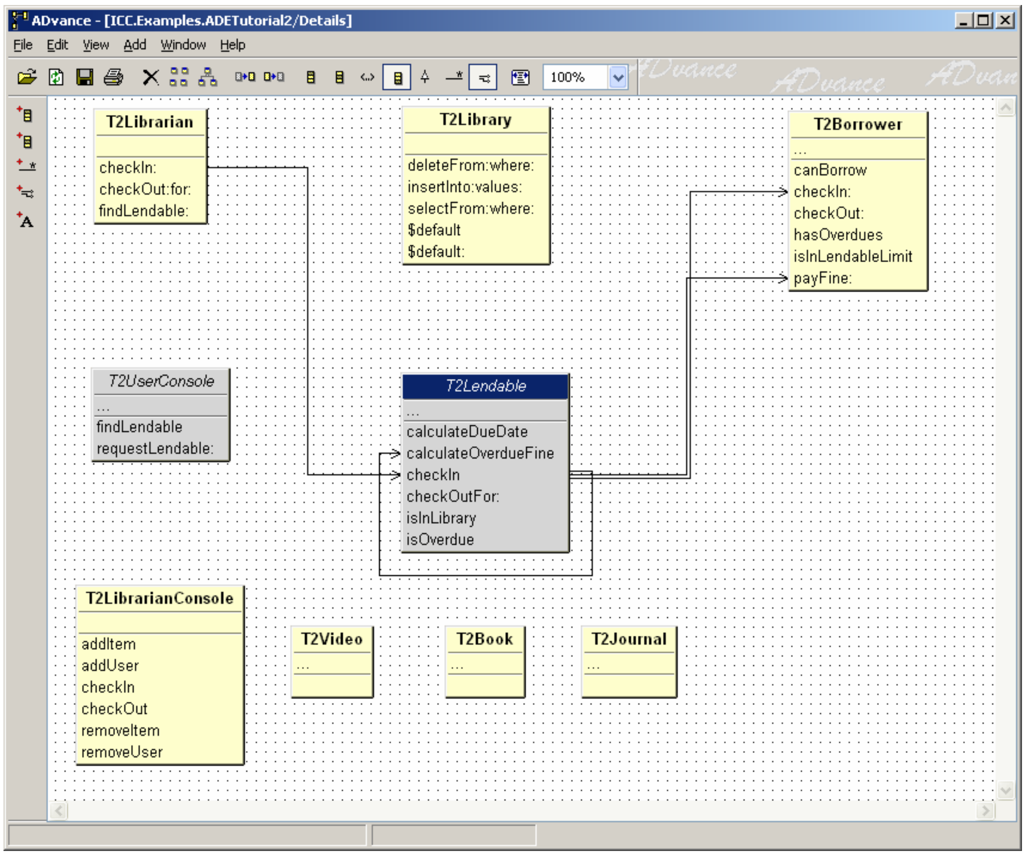


Figure 44. A nested script for *ICC.Examples.T2Librarian>>checkIn:*

Select **File→Save as...** and accept the suggestion (*ICC.Examples.T2Librarian>>checkIn:)*.

Naming Conventions

This is a good time to make a point about diagram names and naming conventions. There is no pre-defined standard enforced by **ADvance**, but we strongly encourage that your project adopts standardized, readable names for diagrams so that users can migrate easily through them and be able to organize each subject for ease of maintainability. The point is to be consistent no matter what you choose.

We choose to name structural diagrams with the view type (e.g. *Overview*, *Services*) while we preface behavior diagrams with *class*>>*selector*. The latter convention is suggested when saving behavior diagrams with **File→Save As...**

Creating nested Scripts

In order to see how a complete message path can be drawn automatically, let's start from a clean state.

🖱 Select **Edit→Select all** or press the corresponding toolbar button ( ). To remove all scripts, select **Add→Remove scripts**. Confirm the request.

↳ All previously drawn scripts in the diagram disappear.

🖱 Select *ICC.Examples.T2Librarian*, then invoke **Scripts...** from the class pop-up menu.

↳ The *Script Selection* dialog appears.

🖱 Check *checkIn:* and set **Script depth** to 2.

↳ The nested message passing is shown as seen in the figure on page 4-26.

A Note about Filtering

If a method is filtered, it is not available for selection in the *Script Selection* dialog. Sometimes when a method is visible in the diagram class but not available for selection to trace - this indicates that the participant methods have been filtered, since filtered methods are not included in the inference.

You can use empty filters first, define your scripts to show and than apply **View→Filter non-script services...** to hide the unused services in this diagram.

Set diagram focus on the Scripts

Among the classes being involved in the scripts, the diagram shows some classes not connected with scripts. **ADvance** provides a simple way to hide classes which are not relevant for the shown scripts.

🖱 Select **View → Hide non script classes** or press the corresponding toolbar button.

↳ All classes not involved in scripts are hidden.

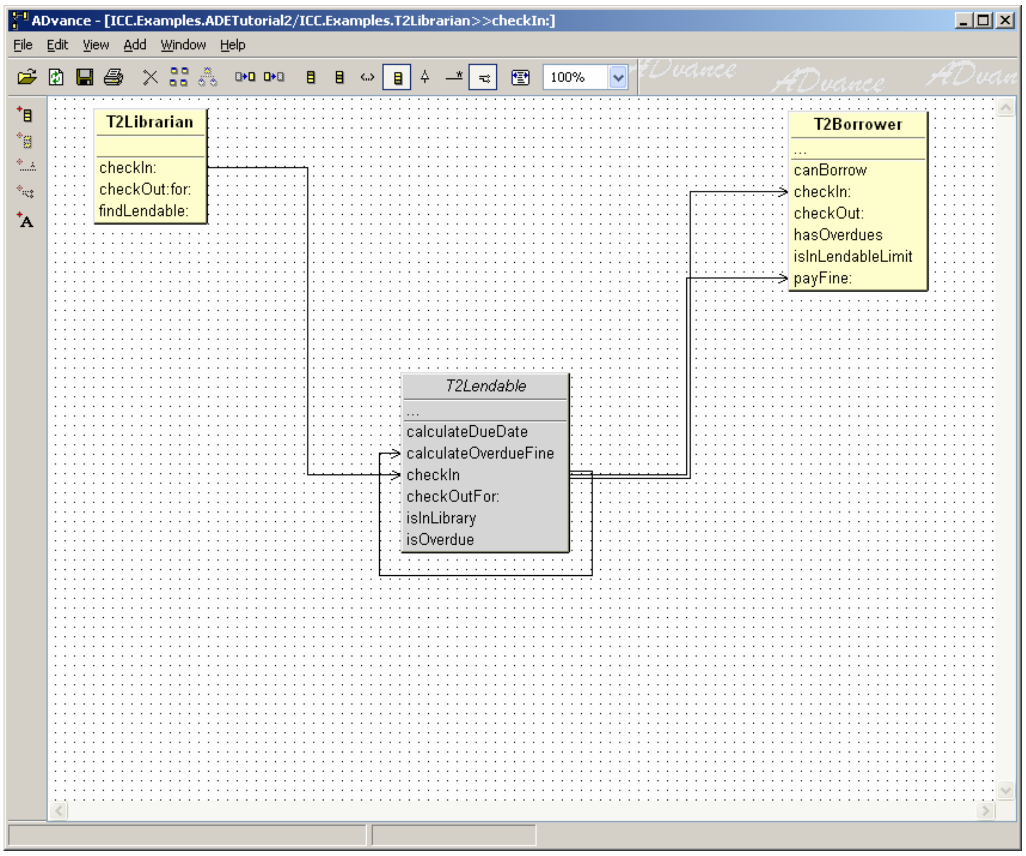


Figure 45**.** Classes involved in script for *ICC.Examples.T2Librarian>>checkIn:*

Summary

Create Subject

Decide which subset of the system you want to view. A ***subject*** is the mechanism for representing the subset's context. An example of a context may be a scenario, VisualWorks category or a parcel. The *Subject Wizard* allows subject creation directly from VisualWorks categories, parcels, and packages. Classes can then be added or deleted with the *Subject Editor* as you begin to gain a better understanding of the view that you want to represent.

Create Diagrams

A ***diagram*** is composed of the classes that were defined in the subject with some of them hidden. Each diagram can represent a different view on the subject, i.e. the structure and/or behavior of your system.

Prior to opening a default diagram, switch on the following diagram layers in *Diagram Painter*: Inheritance, Attributes, Associations, Namespaces. These settings allow **ADvance** to assemble a better initial diagram layout. To clarify different views on this subject, one can create multiple diagrams where each diagram highlights a different aspect of the class interaction i.e. structural vs. behavioral.

Complete Variable Type Information

Make sure that each of the class' variables has a type declaration associated with it. In order for the class comment section or the class method providing the information to be complete it must include all variable names (including instance, shared, class instance and overridden inherited variables).This is because **ADvance** uses these mappings to both construct the class relationships and infer message flow in its diagrams. If you are uncertain about the type that should be assigned to a variable, check the code for what values may be assigned to it. A good starting point for this is the #initialize method, instance creation code and senders of variable accessors.

Check the results by using the VisualWorks *Class Reporter*. Ideally it should report nothing. You also may use the *Comment Generator* plug-in, which generates comment stubs.

Edit Diagrams

The automatically generated diagram layout may be improved upon in different ways, e.g. by rearranging classes, adding relations, defining scripts, using different colors etc. For example, group classes with similar names; group by a common superclass or group by collaborations or associations. If the classes in the diagram layout seem to be unrelated or have few connections between them, these techniques can help find relevant associations. Note that associations are not drawn to classes outside of the subject, to hidden classes, or to classes declared as Containers i.e. Collection hierarchy and ValueModels.

After applying these techniques, the diagram should reflect the basic class structure relationships and/or selected message flows. Identifying and drawing any additional association remain as optional exercises for the user to determine if this is important to this diagram. In order to build a complete diagram, this may require revisiting some of the above mentioned steps in an iterative manner. The following steps profile what may be highlighted when iterating.

Revisit Class Definitions and Type Information

If appropriate, make the variable type information given class comments or in the respective class methods more current based on the information from the diagram. Change variable types from generic to specific to gain a more detailed view in the diagram. Remove obsolete variables.

Revisit the code to get more information about relations i.e. which class references are used by, which classes have structural or behavioral relations and check whether they are already shown in the diagram. Then update the diagram and continue with editing.

Add Collaborators to Subject

Good candidates to add are superclasses, associated classes, classes that refer to the subject's classes (initiators) and classes that the subject may refer to outside of the current scope (participants). Navigate over relationships first, and then navigate over behavior to explore which classes are collaborators of classes being already part of the subject. Both types of navigation are supported by the *Diagram Painter* with the class menu commands **Add→Related classes...**, **Add→Initiators...**, and **Add→Participants...**

If potential collaborators are not associated with any class in the subject, find out which associations should be added which are not directed through state (i.e. query relations). This normally needs some scrutiny of services which send messages to parameters.

Update the diagram and continue with editing.

Remove irrelevant Classes from Subject

If there remain classes after iterating having no relations this may indicate that they are not well placed in this subject. Remove them.

Generate Documentation

You may print diagrams, copy diagrams to clipboard, generate script docu, or create a complete HTML document. For printing choose an appropriate scaling and print the diagram. It should fit on about 1 to 6 pages

5 Forward Engineering

Objectives

Demonstrate how ADvance can be used as an object design capture tool - creating structural and behavioral views.

Show ability to map into VisualWorks constructs directly.

Demonstrate full integration of graphics to code.

During an object oriented analysis and design iteration, **ADvance** can be used to capture the design diagrams and generate initial class definitions.

One feature of this tool is that if the user creates new design using **ADvance**, the class definitions are automatically generated as VisualWorks code. Thus the diagrams serve as direct graphical representations of the code. This direct mapping holds true when diagrams are generated from existing code, too. Therefore the integrity of the design is maintained throughout the design ↔ implementation iterations.

The following steps guide you through the creation of subjects, diagrams, classes, structure and behavior from scratch. To demonstrate these facilities we will rebuild parts of the library example as introduced in the *Overview* chapter.

Creating an empty Subject

The first step for creating a new design is the subject creation. You might revisit the subject creation in the previous chapter where it is explained more in detail.

Given that we are starting from scratch we will create an empty subject *ICC.Examples.ADETutorial3.* We will use it for all diagrams in this chapter.

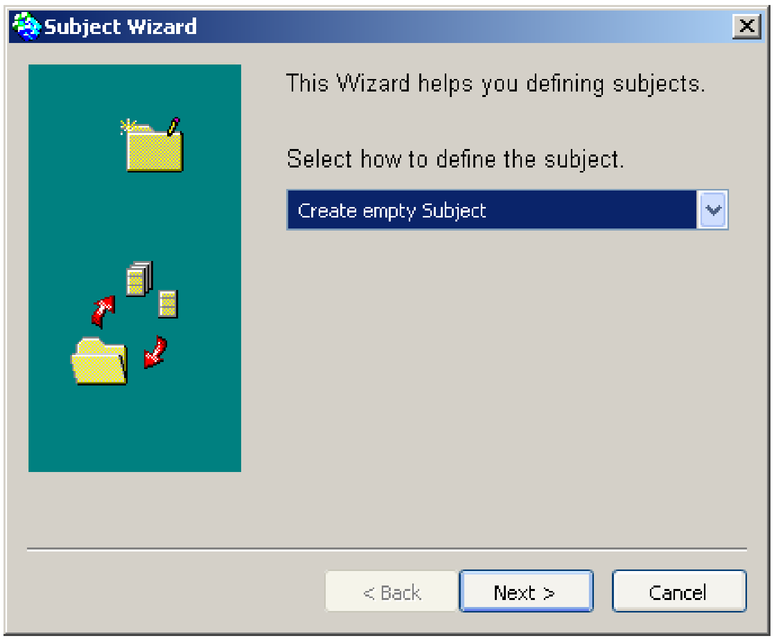
🖱 Open the *Subject Browser* by selecting **Tools→Subject Browser** in the *Workbench* menu or click on the corresponding toolbar icon.

↳ This opens the *Subject Browser*.

🖱 Select the *[Root]* subject in the *Subject Browser's* subject tree.

🖱 From the *Subject Browser*, select **File→New subject...**

↳ The *Subject Wizard* is opened.



This Wizard helps you defining subjects.

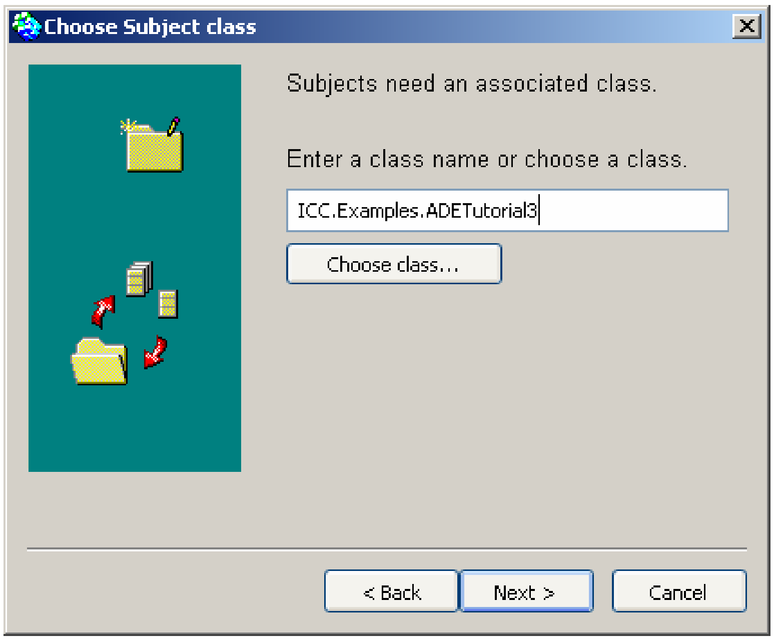
Select how to define the subject.

Create empty Subject

Figure 46. The *Subject Wizard* (1).

Select **Create empty Subject** then click **Next**.

A wizard page for choosing the subject's class appears.



Subjects need an associated class.

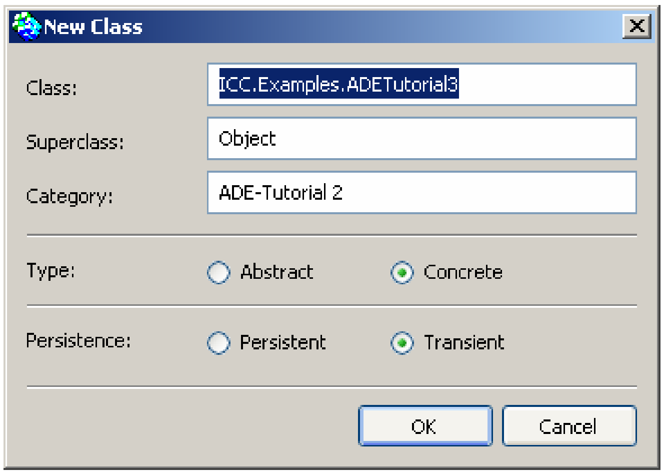
Enter a class name or choose a class.

ICC.Examples.ADETutorial3

Figure 47. Choosing a subject class with the *Subject Wizard* (2).

Type **ICC.Examples.ADETutorial3** into the blank field then click **Next**.

Since **ICC.Examples.ADETutorial3** is a new class, the *New Class* dialog is opened.


New Class

Class: ICC.Examples.ADETutorial3

Superclass: Object

Category: ADE-Tutorial 2

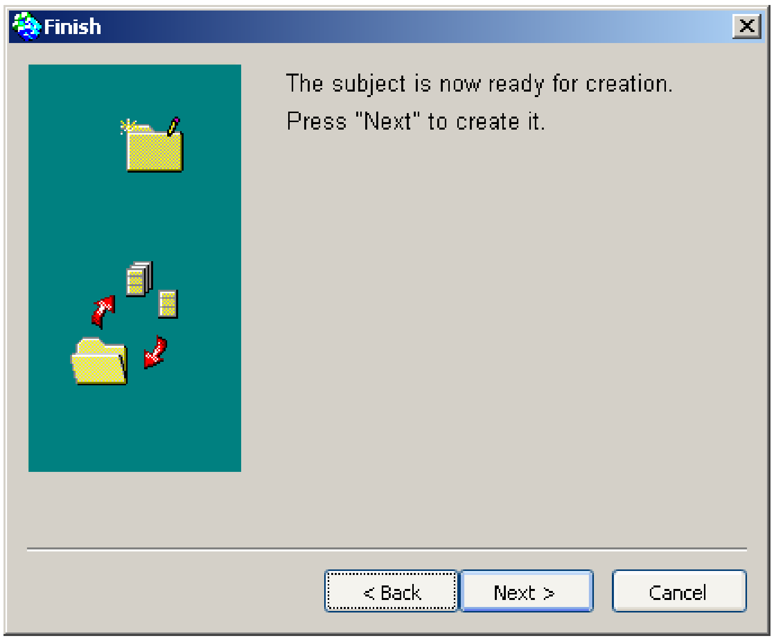
Type: Abstract [✓ Concrete]

Persistence: Persistent [✓ Transient]

Figure 48. The *New Class* dialog for ICC.Examples.**ADETutorial3**.

Click **OK** to create the new class.

A class named ICC.Examples.*ADETutorial3* is created and the last wizard page is opened. Depending on the options available in the general preferences, the class is put into the default / current package, a package asked from user, or an explicitly named packaged. See section *General Preferences* for the details of the options.

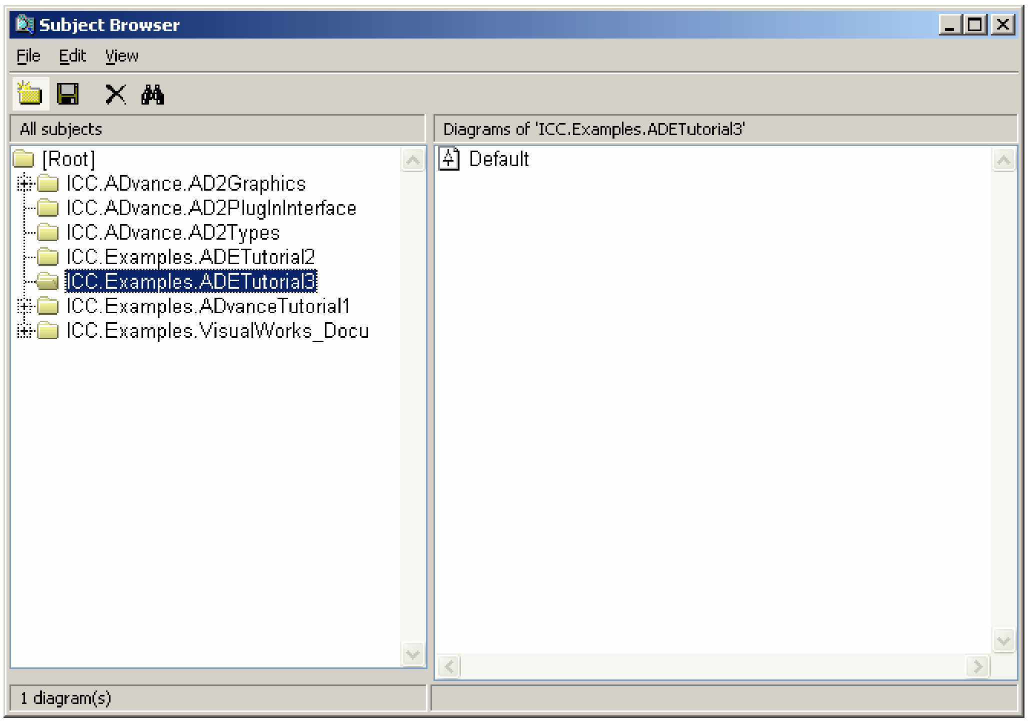

Finish

The subject is now ready for creation.
Press "Next" to create it.

Figure 49. The *Subject Wizard* (3).

Click **Next** to finish the subject creation process.

↳ The subject *ICC.Examples.ADETutorial3* is created and shows up in the *Subject Browser*.



All subjects

[Root]
ICC.ADvance.AD2Graphics
ICC.ADvance.AD2PlugInInterface
ICC.ADvance.AD2Types
ICC.Examples.ADETutorial2
ICC.Examples.ADETutorial3
ICC.Examples.ADvanceTutorial1
ICC.Examples.VisualWorks_Docu

Diagrams of 'ICC.Examples.ADETutorial3'

Default

1 diagram(s)

Figure 50. The new subject and its default diagram.

Creating a new Design

Once you have defined a subject you can then define any number of diagrams within this subject. The number of diagrams associated with this subject will vary depending on the scope of your subject and the number of views required to comprehensively communicate your design.

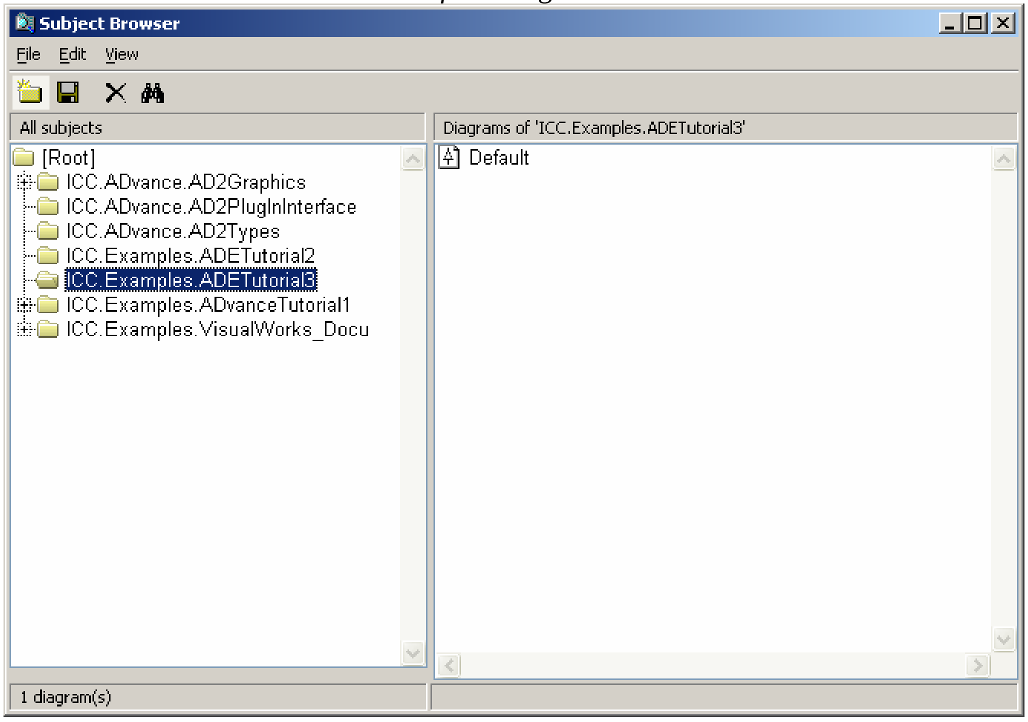
Creating a new Diagram

We now will open the *Default* diagram of our subject where we can add classes and relations graphically.

🖱 Select *ICC.Examples.ADETutorial3* in the *Subject Browser*.

↳ The *Default* diagram is displayed in the subject's diagram list.

Select the *Default* diagram as shown in


Subject Browser
File Edit View
All subjects
[Root]
ICC.ADvance.AD2Graphics
ICC.ADvance.AD2PlugInInterface
ICC.ADvance.AD2Types
ICC.Examples.ADETutorial2
ICC.Examples.ADETutorial3
ICC.Examples.ADvanceTutorial1
ICC.Examples.VisualWorks_Docu
Diagrams of 'ICC.Examples.ADETutorial3'
Default
1 diagram(s)

🖱 **Figure 50**. Then open it by double-clicking on it.

↳ A *Diagram Painter* is opened and shows an empty diagram for *ICC.Examples.ADETutorial3*.

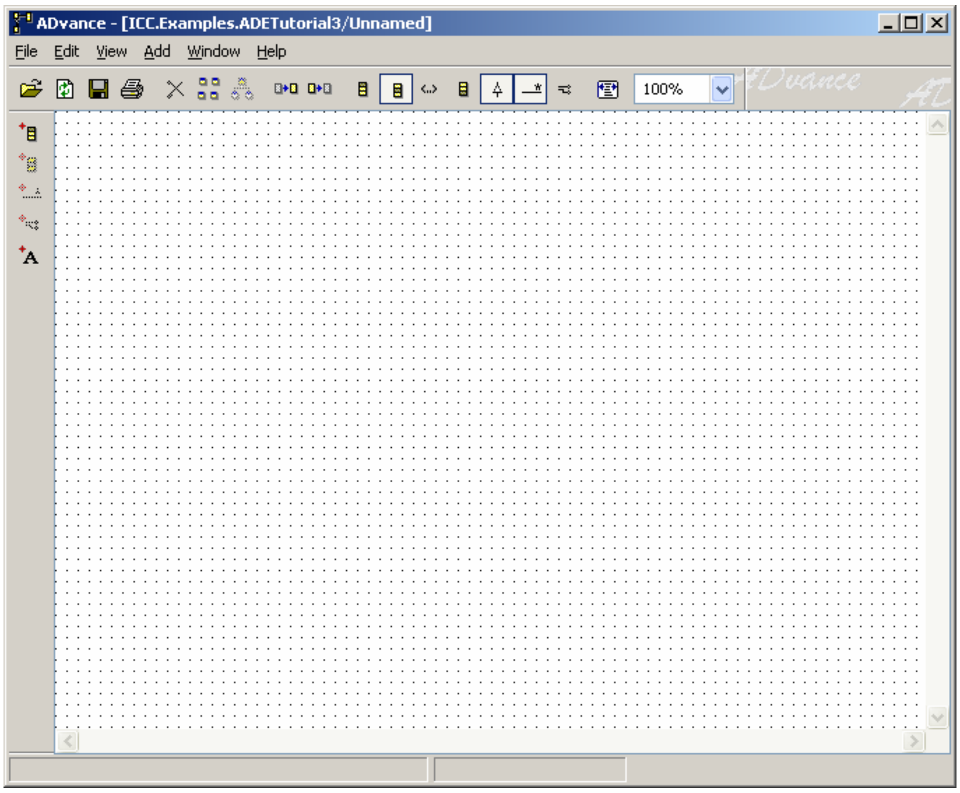

ADvance - [ICC.Examples.ADETutorial3/Unnamed]
File Edit View Add Window Help
100%

Figure 51. The *Diagram Painter* shows an empty diagram.

Do a **File→Save As...** to name and save it. Name the diagram *Overview*.

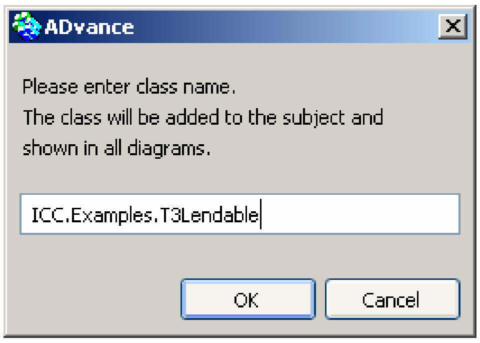
The diagram is saved and the painter’s window label changes.

Adding Classes

The first logical design step is to add classes to the diagram. To demonstrate this we will add three classes for namespace *ICC*.Examples: *T3Lendable* with one subclass *T3Book*, and *T3Borrower*. Since we already have library code in the image we use the prefix *T3* for the new classes.

Select **Add→Class...** from the menu bar or click the button in the left pane of the tool.

A dialog is opened that requests the class name for the class to add.



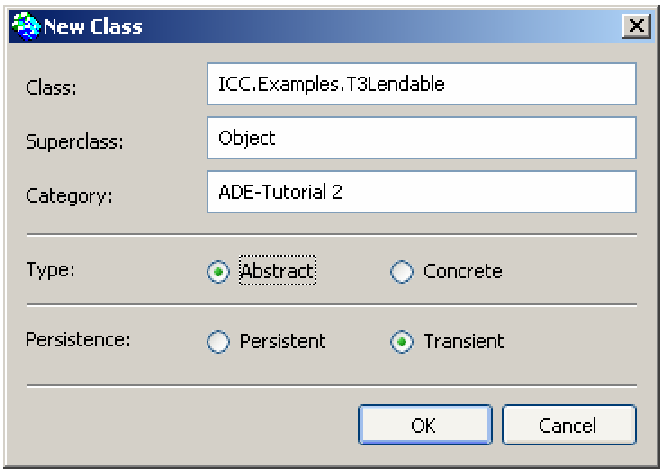
Please enter class name.
The class will be added to the subject and shown in all diagrams.

ICC.Examples.T3Lendable

Figure 52. Entering the name of the new class.

Enter **ICC.Examples.T3Lendable** and click **OK**.

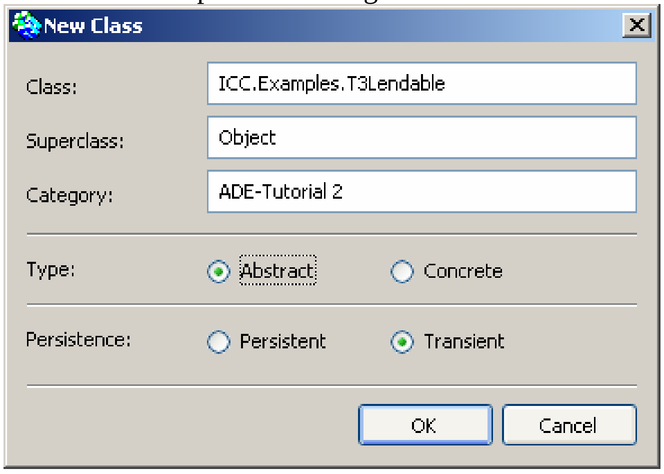
Since *ICC.Examples.T3Lendable* is not in the image the *New Class* dialog is opened.



Class: ICC.Examples.T3Lendable
Superclass: Object
Category: ADE-Tutorial 2
Type: [✓ Abstract] Concrete
Persistence: Persistent [✓ Transient]

Figure 53. The *New Class* dialog for *ICC.Examples.T3Lendable*.

Complete the dialog as shown in



Class: ICC.Examples.T3Lendable

Superclass: Object

Category: ADE-Tutorial 2

Type: [✓ Abstract] Concrete

Persistence: Persistent [✓ Transient]

Figure 53, click **OK** and choose a package if no current packe is declared.

A special cursor appears and the status bar at the bottom of the window will prompt you to click on the spot in the diagram window where you would like the class to be placed.

Place the class in the middle of the diagram.

Switch on the namespace layer.

The tool creates a class *ICC.Examples.T3Lendable* with the appropriate Smalltalk class definition. The class is then added to the diagram's subject *ICC.Examples.ADETutorial3*. Finally the diagram is updated and now contains the new class.

Note: The new classes' package is determined by the system settings (current package or user prompt), not by the **ADvance** general preferences. This allows the user easily to keep apart application domain classes from documentation classes.

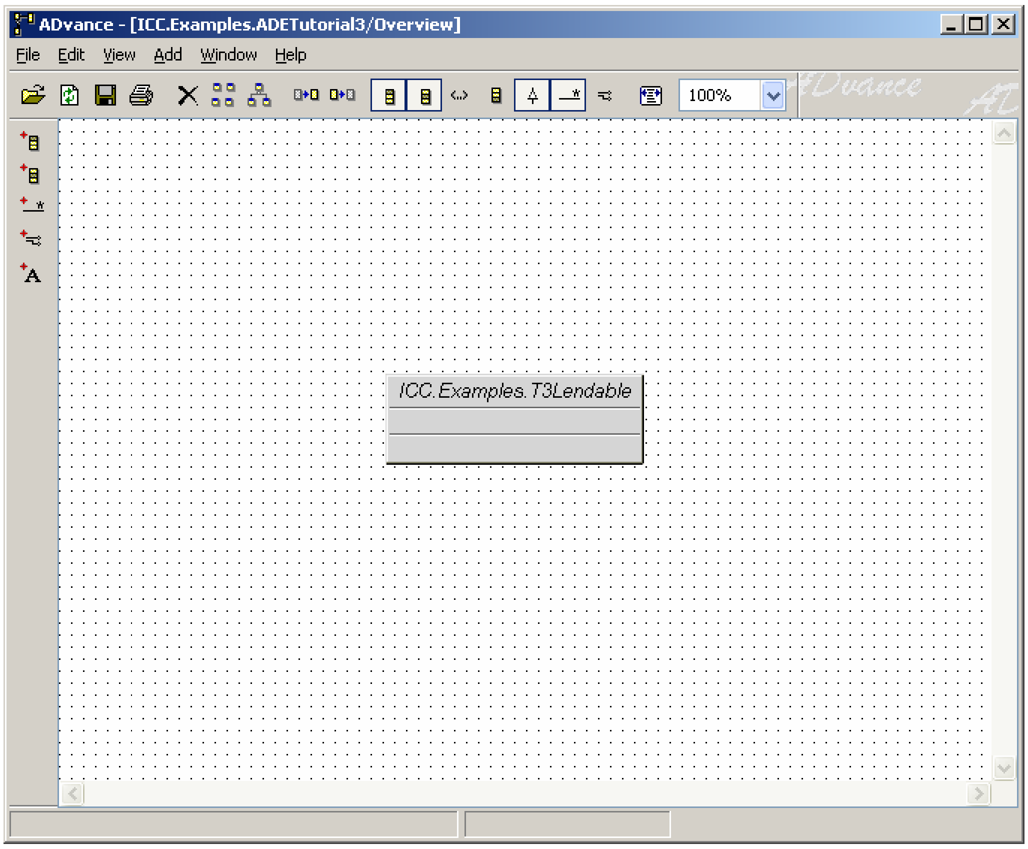


Figure 54. The diagram with the new class added.

Now we add a subclass to *ICC.Examples.T3Lendable* named *ICC.Examples.T3Book*, and another class *ICC.Examples.T3Borrower*.

- First select *ICC.Examples.T3Lendable*. This will be used as superclass suggest in the *New Class* dialog.
- Select **Add→Class...** and add a new class *ICC.Examples.T3Book*. In the *New Class* dialog define *ICC.Examples.T3Book* as concrete and persistent subclass of *ICC.Examples.T3Lendable.*
- Now add the concrete and persistent class *ICC.Examples.T3Borrower* as subclass of *Object*.
- The diagram now reflects the three classes.

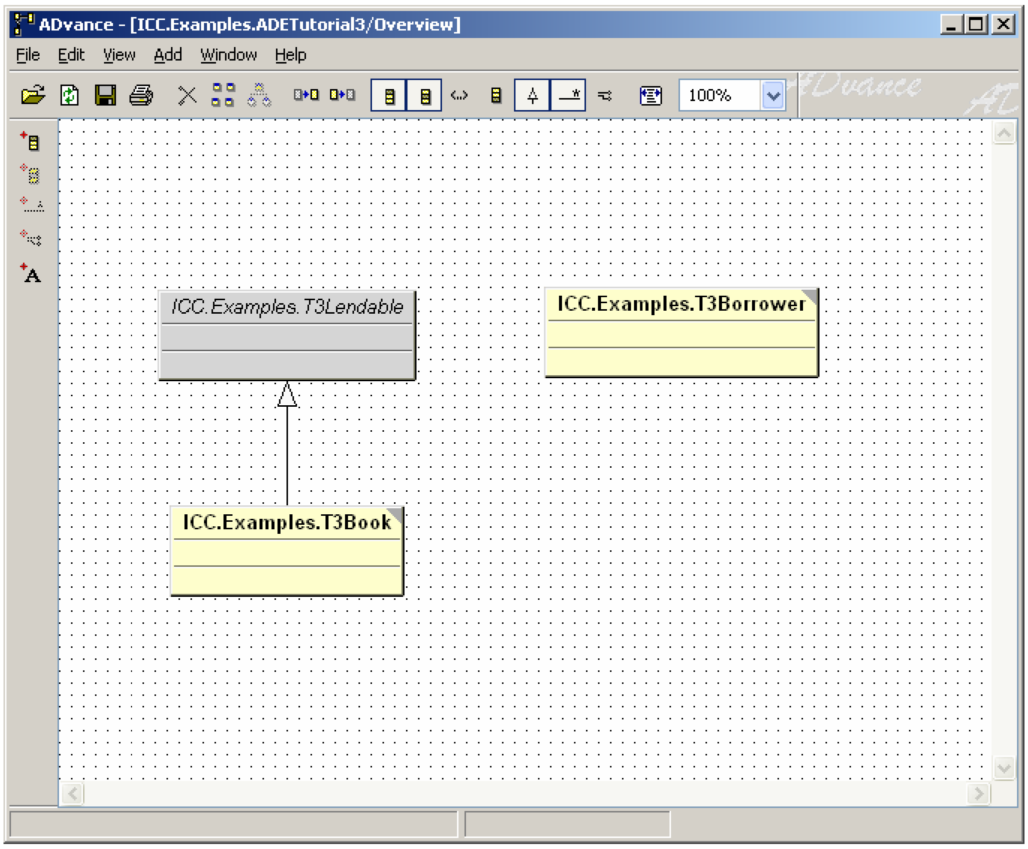

ADvance - [ICC.Examples.ADETutorial3/Overview]
File Edit View Add Window Help
100%
ICC.Examples.T3Lendable
ICC.Examples.T3Borrower
ICC.Examples.T3Book

Figure 55. The diagram with three new classes.

There are different ways to add attributes to the new classes and to create associations between them:

- add a single attribute through **Add→Attribute...**,
- add multiple attributes via the *Class Properties* dialog,
- add associations graphically.

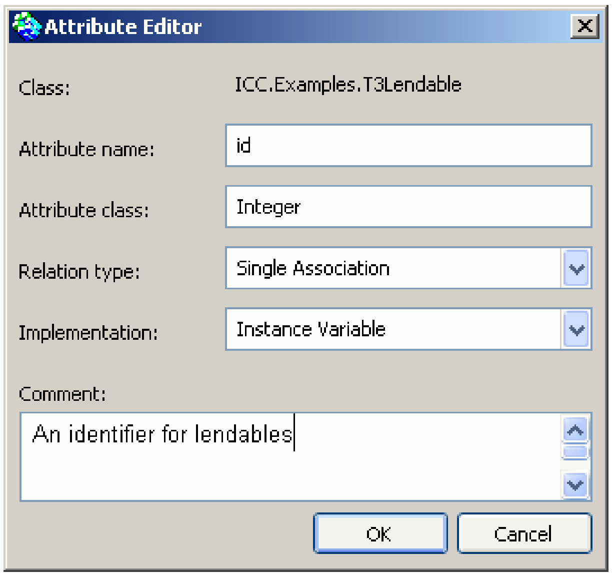
Adding a single Attribute

We will add an instance variable *id* to class *ICC.Examples.T3Lendable*.

Select *ICC.Examples.T3Lendable* then press the button or invoke **Add→Attribute...**

This opens an *Attribute Editor*.

Complete it as follows and click **OK**.



Class: ICC.Examples.T3Lendable

Attribute name: id

Attribute class: Integer

Relation type: Single Association

Implementation: Instance Variable

Comment: An identifier for lendables

Figure 56. Adding attributes with the *Attribute Editor*.

- This will modify the class definition and class comment (or the method providing the variable type informations) of *ICC.Examples.T3Lendable* to define an instance variable *id* with type *Integer*.
- Switch on the attribute types layer, to show the attribute types.
- The diagram should look like this:

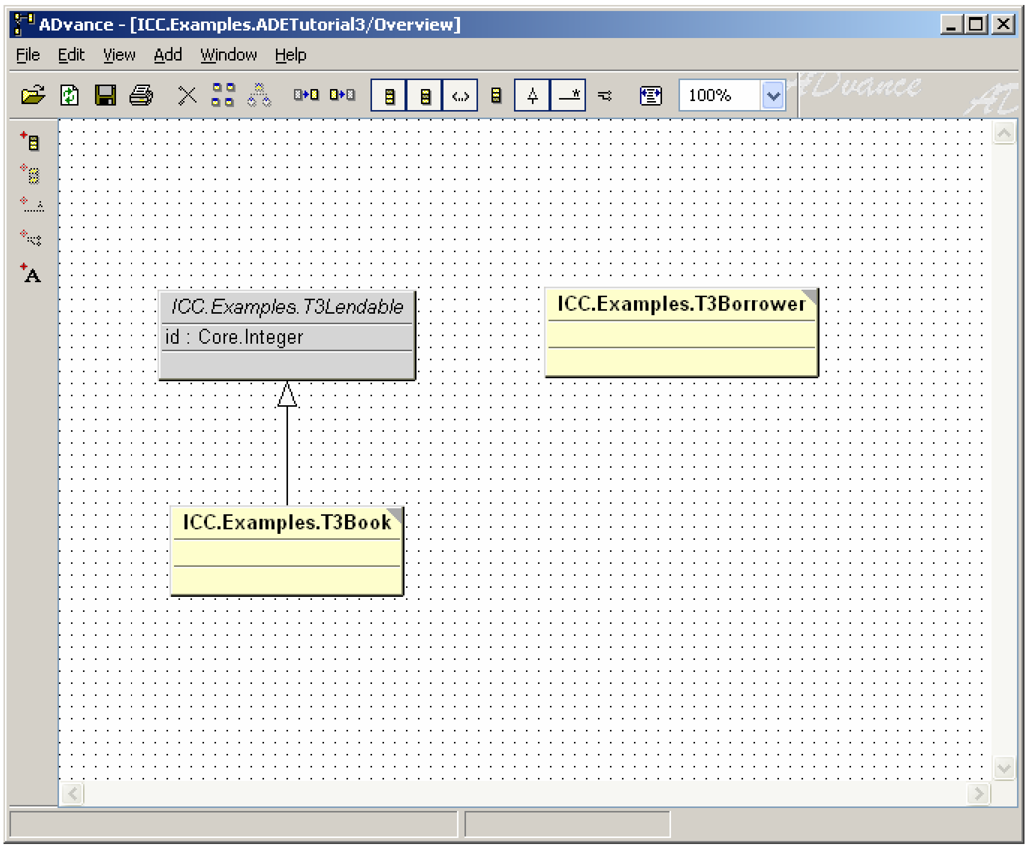
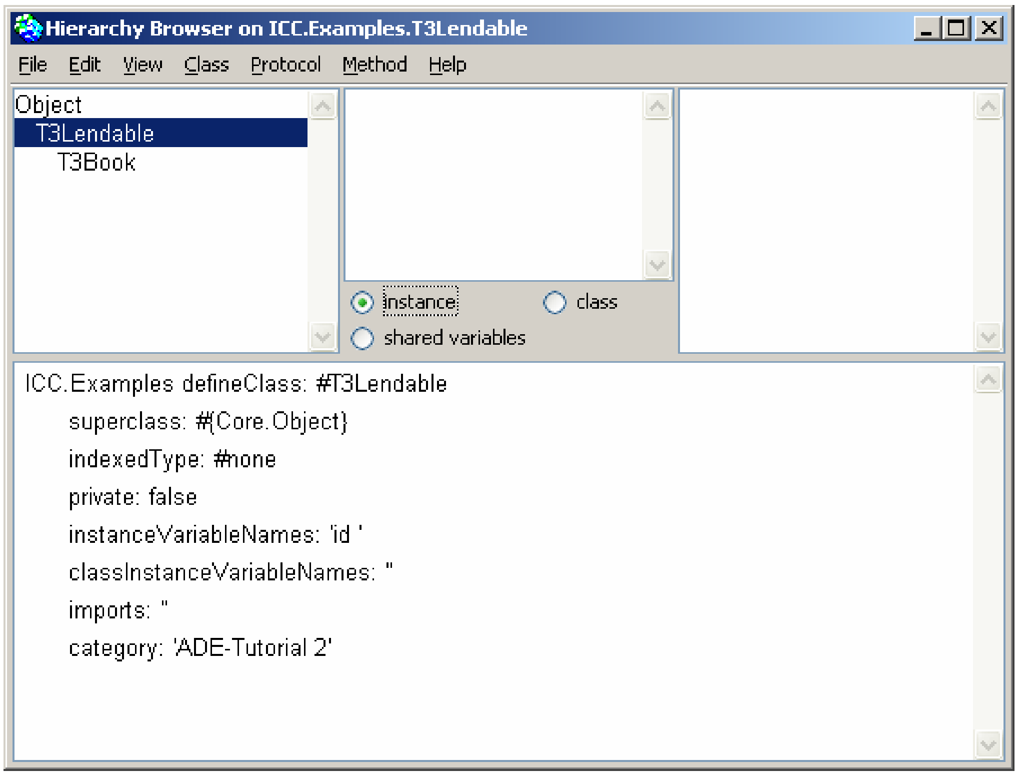


Figure 57. The diagram reflecting the new attribute.

To review the modified code <Shift>-double-click on *ICC.Examples.T3Lendable*.

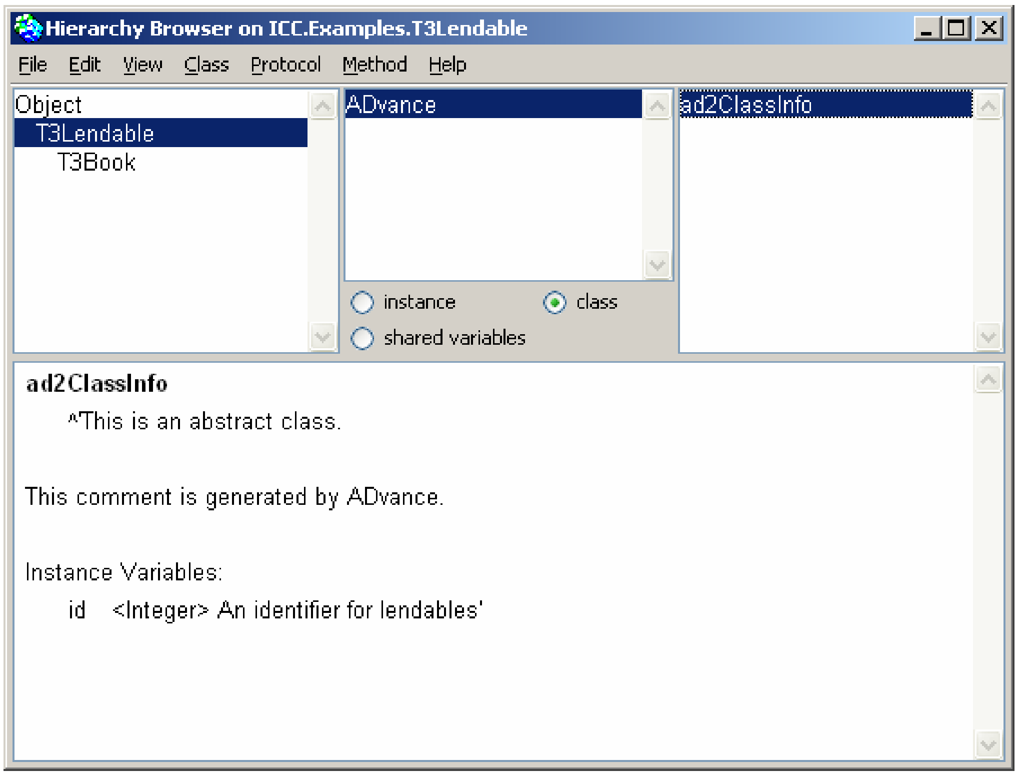
A VisualWorks *Hierarchy Browser* on *ICC.Examples.T3Lendable* is opened.


Hierarchy Browser on ICC.Examples.T3Lendable
File Edit View Class Protocol Method Help
Object
T3Lendable
T3Book
instance
class
shared variables

```
ICC.Examples defineClass: #T3Lendable
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'id '
    classInstanceVariableNames: ''
    imports: ''
    category: 'ADE-Tutorial 2'
```

Figure 58. Browsing the class definition of *T3Lendable*.

The example uses the **ADvance** preferences to store type information into a special class method. If you did not change the default preferences, switch to class side and view method *T3Lendable>>ADvance>>ad2ClassInfo*. If you are using different preferences, you have to view the corresponding class method or the class comment.


Hierarchy Browser on ICC.Examples.T3Lendable
File Edit View Class Protocol Method Help
Object
T3Lendable
T3Book
ADvance
ad2ClassInfo
instance
class
shared variables

```
ad2ClassInfo
    ^This is an abstract class.

This comment is generated by ADvance.

Instance Variables:
    id    <Integer> An identifier for lendables'
```

Figure 59. Browsing the comment of *T3Lendable*.

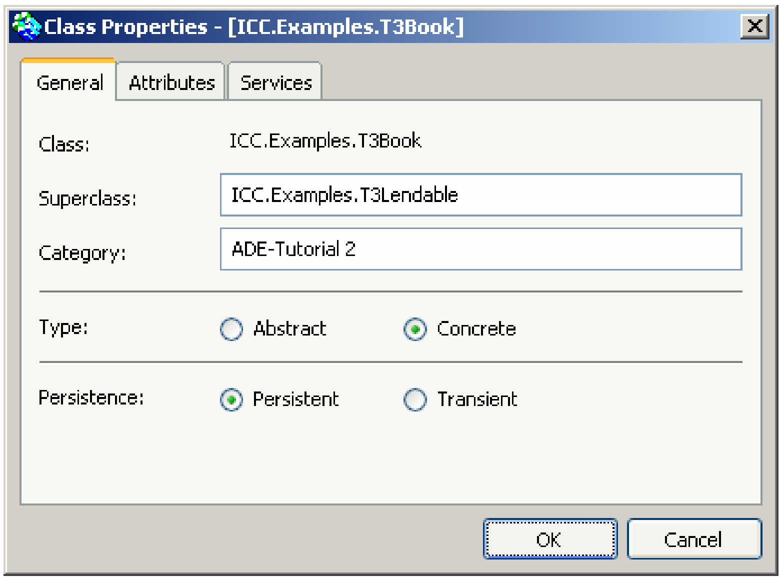
Close the *Hierarchy Browser*.

Adding multiple Attributes

We now will add entries to the attribute definition of *T3Book*. We use the *Class Properties* dialog which provides an interface for modifying class definitions and class comments.

Doubleclick on *T3Book* or select it and invoke **Edit→Properties...**

This opens the *Class Properties* dialog on *T3Book*.



Class: ICC.Examples.T3Book

Superclass: ICC.Examples.T3Lendable

Category: ADE-Tutorial 2

Type: Abstract [✓ Concrete]

Persistence: [✓ Persistent] Transient

Figure 60. The *General* tab of the *Class Properties* dialog.

Click the *Attributes* tab.

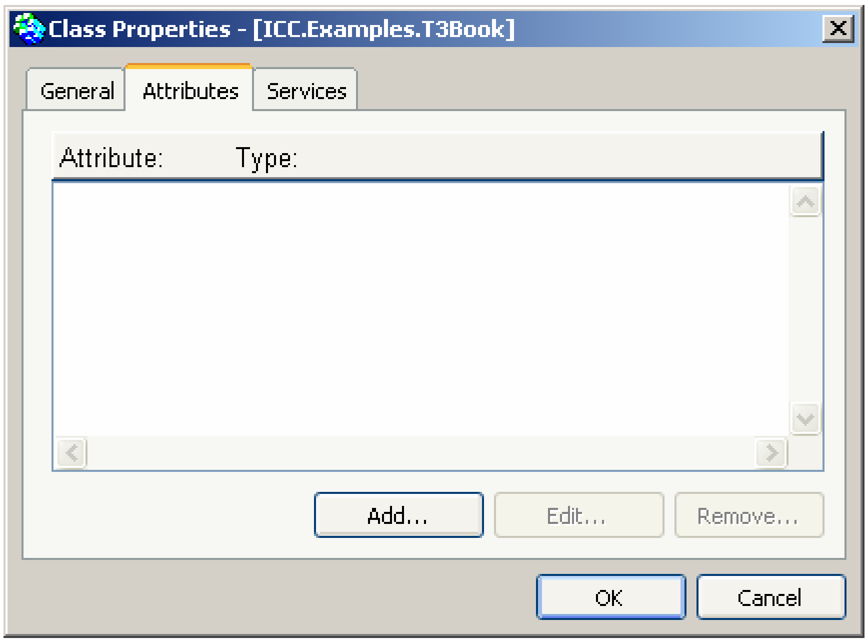


Figure 61. The *Attributes* tab of the *Class Properties* dialog.

The *Attributes* page displays the attributes and annotated types of class *T3Book*. It has buttons for adding, editing and removing attributes.

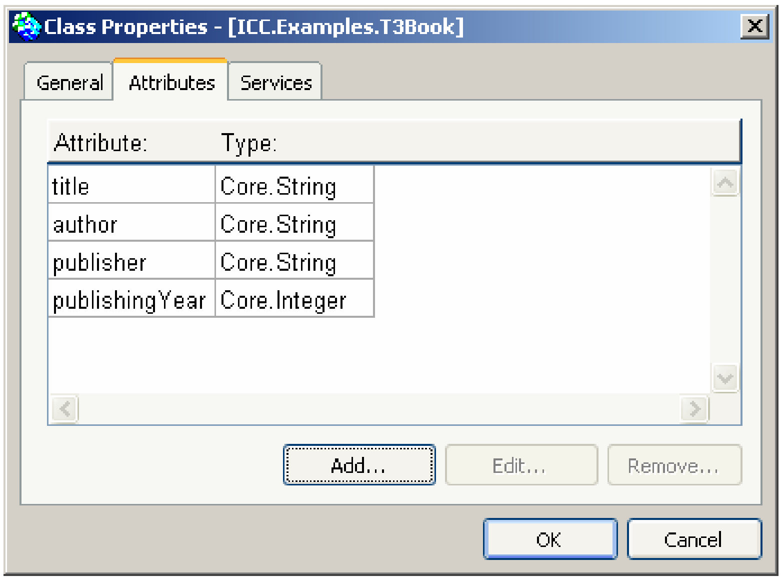
To add attributes click **Add...**

An *Attribute Editor* is opened, see **Figure 56.Figure 56**

Add an attribute named *title* with attribute class *String*, then click **OK**.

To complete the class definition add attributes *author*<String>, *publishingYear* <Integer> and *publisher* <String>.

The dialog contents will then be drawn as shown below:



Attribute:	Type:
title	Core.String
author	Core.String
publisher	Core.String
publishingYear	Core.Integer

Figure 62. Completed class definition for *T3Book*.

To close the *Class Properties* dialog click **OK**.

The diagram is updated automatically and reflects the current design.

Note: If the class name exists in different namespaces, the first class found is taken. If this is not the right one for the attribute, you can use wildcards * whithin the attributes type name to open a list to choose from.

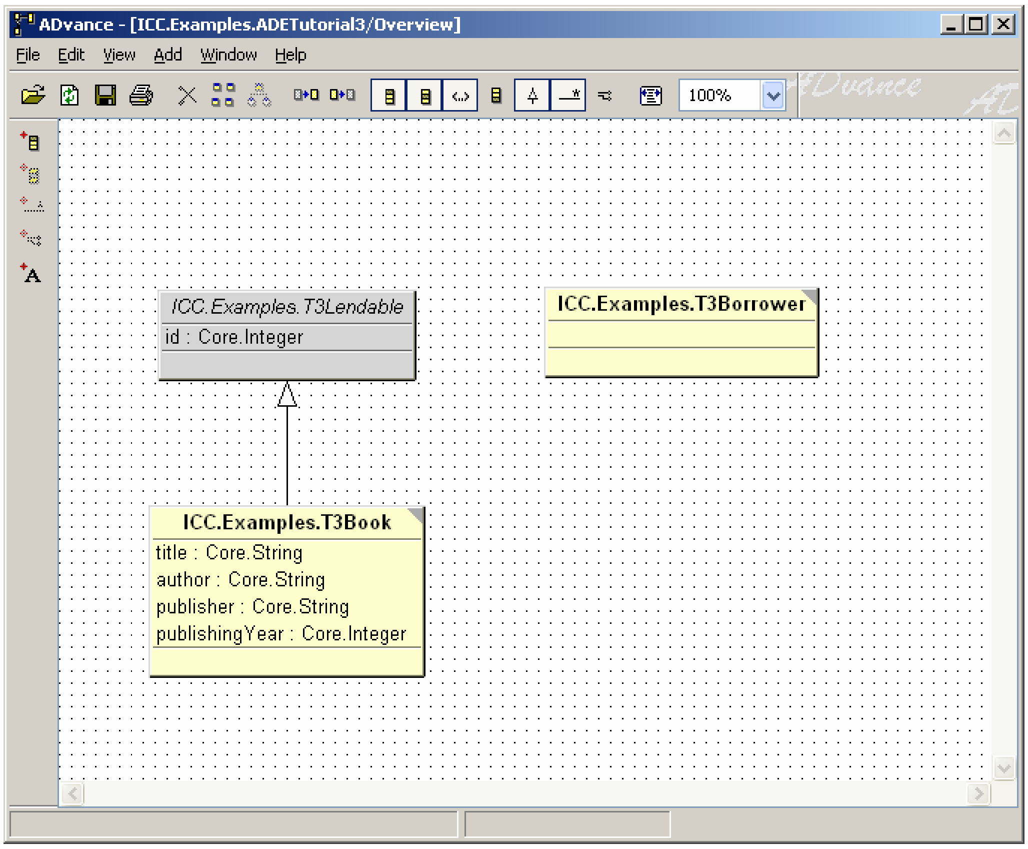


Figure 63. The diagram reflecting the new attributes.

Adding Associations

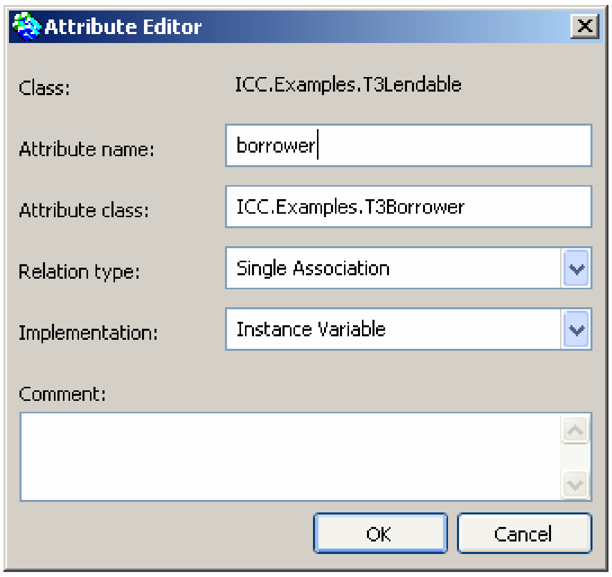
In the section before, we added relationships from subject classes to classes not contained in the current subject (*Integer*, *String*). In this section we will demonstrate how two classes of a subject can be related interactively.

🖰 Select *T3Lendable* then click the button (or select **Add→Association...**).

↳ A special cursor appears and the status bar will prompt you to click on the class that you would like to associate to *T3Lendable*.

🖰 Click on class *T3Borrower*.

An *Attribute Editor* appears. Note that the **Attribute class** input field already contains **T3Borrower**.



Class: ICC.Examples.T3Lendable

Attribute name: borrower

Attribute class: ICC.Examples.T3Borrower

Relation type: Single Association

Implementation: Instance Variable

Comment:

Figure 64. Adding an association.

Add an attribute *borrower* as a single association with instance variable implementation, then select **OK**.

The updated diagram draws an association from *ICC.Examples.T3Lendable* to *ICC.Examples.T3Borrower* as shown in the figure below.

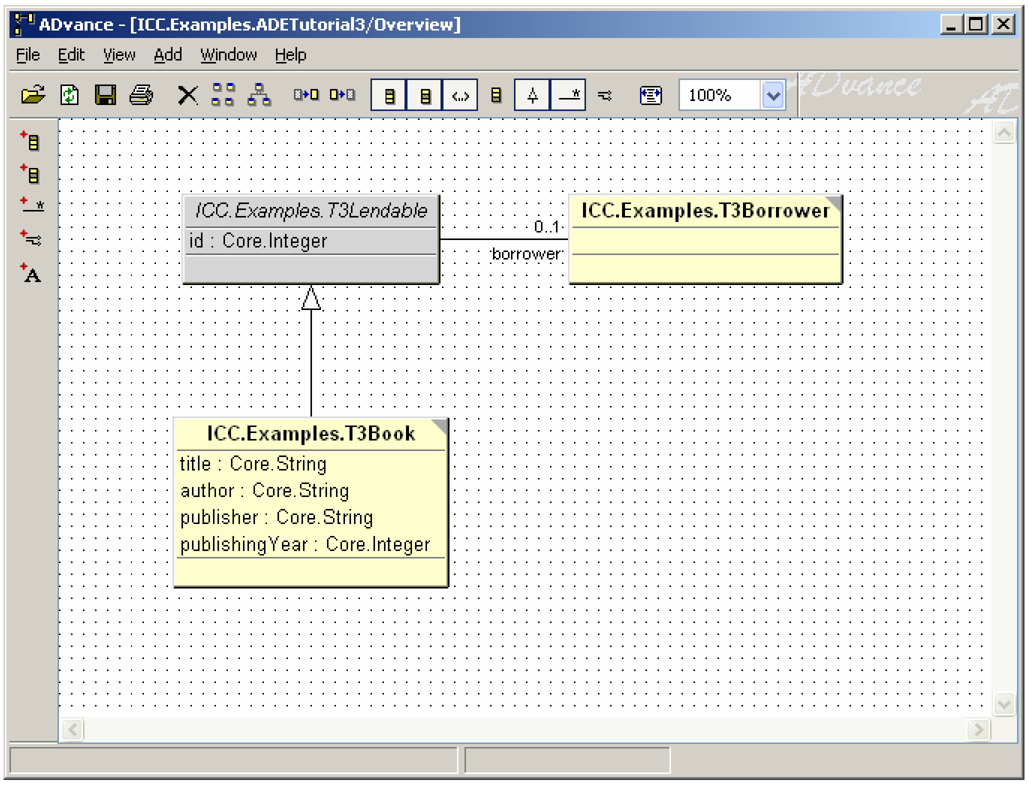


Figure 65. Diagram with added association.

Note: Relationships are drawn to classes which are part of the diagram's subject and not hidden. The information about relationships to classes not in the subject is presented in the class' attributes compartment, relations to hidden classes are hidden as well.

Adding Message Sends

During analysis and design we find and create classes, relationships and services of classes. Services are added to classes using a VisualWorks system browser. ***Message sends*** between services can be defined interactively using **ADvance**.

Use a system browser to add the service *checkIn* to *ICC.Examples.T3Lendable*. Then add services *checkIn:* and *payFine:* to *ICC.Examples.T3Borrower*. Close the browser.

Update the diagram, then switch on the services and scripts layer.

The diagram will reflect the new services.

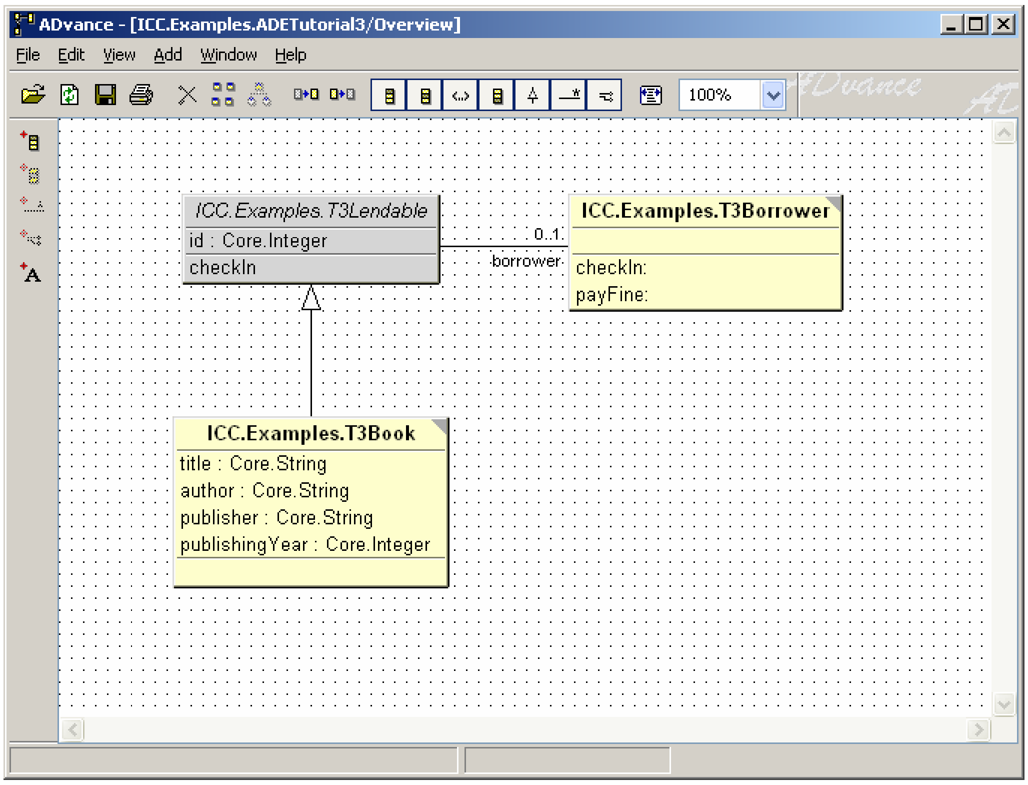


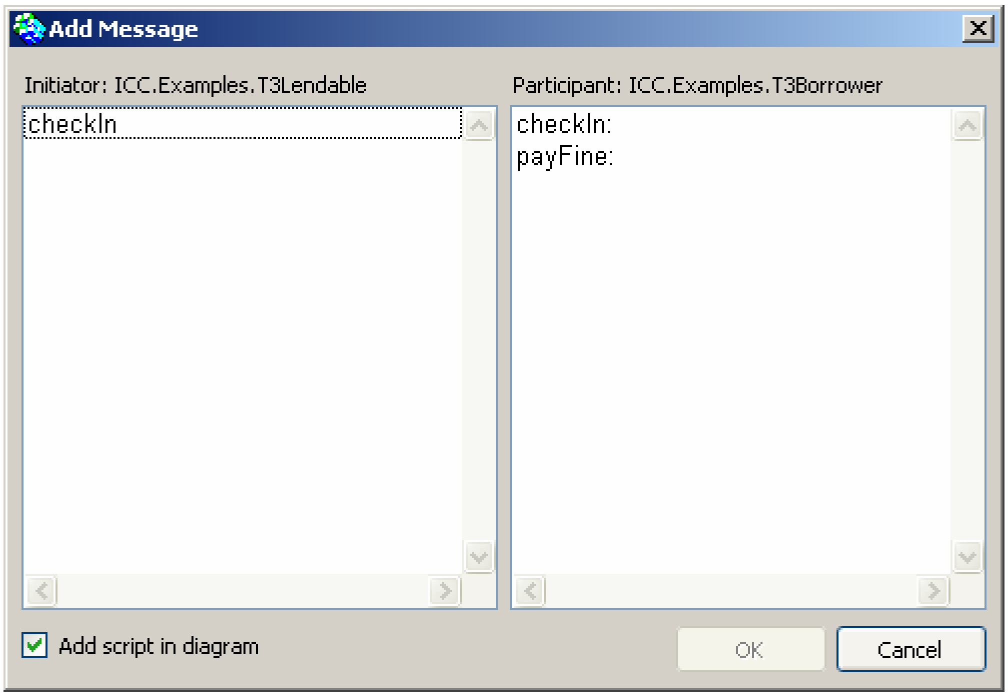
Figure 66. Diagram displaying the new services.

Select *ICC.Examples.T3Lendable*. Then click the button (or select **Add→Message...**).

A special cursor appears and the status bar will prompt you to click on the class that you would like to be the target class of the message send.

Click on class *ICC.Examples.T3Borrower*.

The *Add Message* dialog appears.



Add Message
Initiator: ICC.Examples.T3Lendable
checkIn
Participant: ICC.Examples.T3Borrower
checkIn:
payFine:
☑ Add script in diagram
OK
Cancel

Figure 67. Adding a message send.

Select *checkIn* in the left pane, *checkIn:* in the right pane as shown above. Check the **Add script in diagram** option to automatically add a script in the diagram. Finally click **OK**.

A message relationship is drawn between ICC.Examples.T3Lendable>>checkIn and ICC.Examples.T3Borrower>>checkIn:.

Note: Be sure to have the scripts layer toggled on in the *Diagram Painter*. Otherwise, the automatically generated script will not be shown.

Add a second message send from *ICC.Examples.T3Lendable>>checkIn* to *ICC.Examples.T3Borrower>>payFine:*.

The diagram reflects the added service relations as shown below.

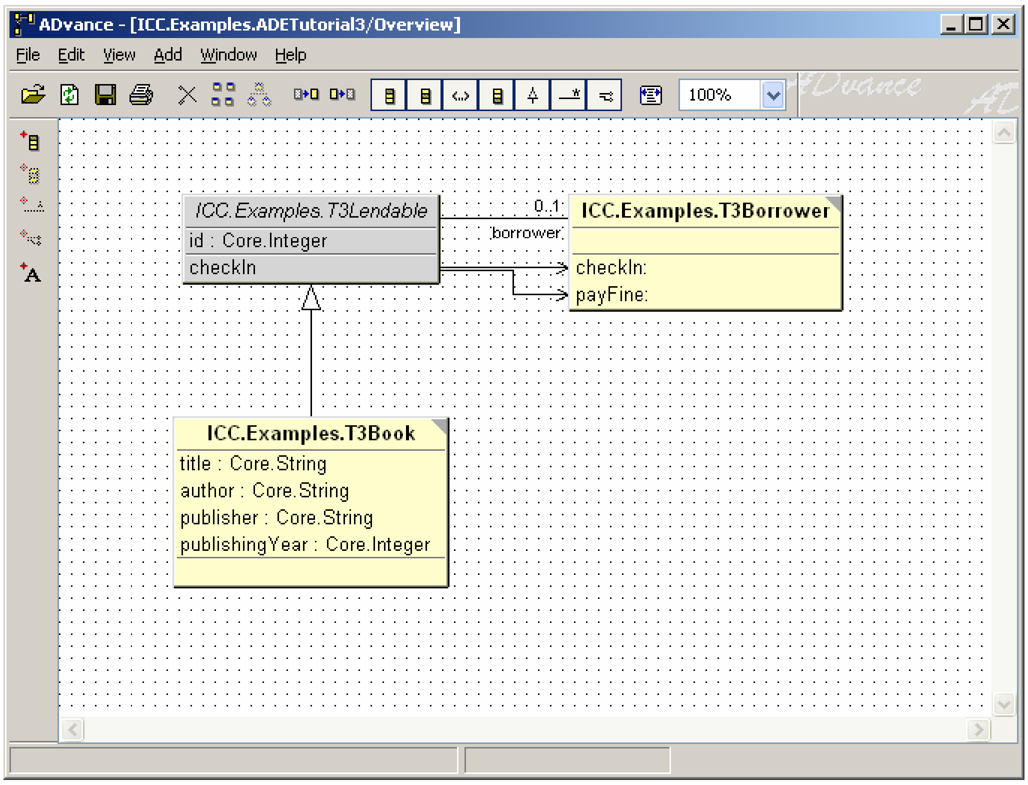


Figure 68. Diagram with added message sends.

Use a browser to have a look at the code of *ICC.Examples.T3Lendable>> checkIn*.

You may experiment by changing the *ICC.Examples.T3Lendable>>checkIn* code to see how the changes are reflected by the tool.

Summary

We started to specify the library example graphically without much knowing about Smalltalk itself. This way, work may progress from a domain expert with little Smalltalk experience to an experienced Smalltalk programmer taking over.

These are the steps we followed:

Create Subject

Start by creating a new (empty) subject with *Subject Wizard* and the *Subject Browser*.

Create Diagram

Open the *Default* diagram and save it under a new name.

Add Classes

Use **Add→Class...** to add classes to the diagram.

Add Attributes

Add attributes to classes. Use **Add→Attributes...** from *Diagram Painter* to add single attributes or open the *Class Properties* dialog to enter multiple attributes.

Add Associations

Invoke **Add→Associations...** to relate subject classes.

Add Code

Use a system browser to add methods to the classes in your diagram.

Generate Message Sends

Generate message sends between classes with **Add→Message...** from *Diagram Painter*

6 Additional Concepts

Objectives

Address advanced issues which were not covered before to keep the complexity of tutorials low.

Typing

Inherited Attributes

Attributes of a class may be redefined in a subclass. For instance variables, this is noted under the section *Inherited Instance Variables:* in class comments. The type of a redefining attribute is often a subtype of the type of the redefined attribute.

In the following example the classes *MyView* and *MyController* have specialized variable types of their generic superclasses *View* and *Controller*.

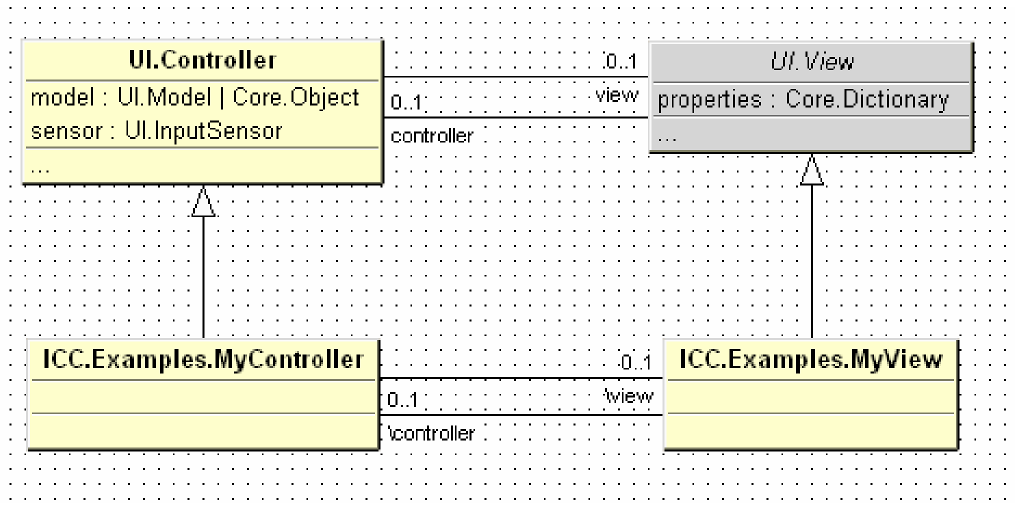


Figure 69. An example for inherited instance variables.

Container Types

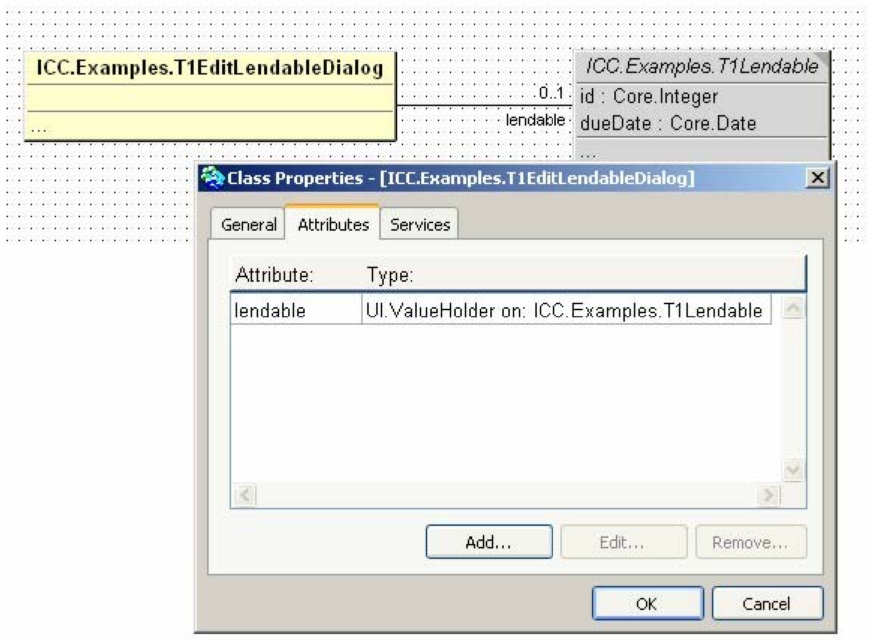
Sometimes, only parts of a type are of interest for the current view. An example is *Collection* where we are interested in the element type instead of Collection itself. You can define which classes should be seen as such ***container types*** by overriding the class side method *viewAsITCollectionType.*

These classes also have a property *genericTypeConstructor* that determines the available type constructor. E.g. *Collection* and its subclasses know the type constructor *of*. Type constructors are used to specialize generic types. For example *Set of: Integer* is a specialization of *Set*.

Type checking would show you information about wrong use of type constructors, syntactically wrong type definitions, or missing type/class names.

Wrapped Types

As with container types, ValueModel types are seen in a special way. We often want to see the ValueModel's component type instead of the ValueModel itself. As standard abstraction, **ADvance** depicts any *ValueModel* type as a relation line with the wrapped type as the referenced entity.



Attribute:	Type:
lendable	UI.ValueHolder on: ICC.Examples.T1Lendable

Figure 70. An example for wrapped types.

You can choose which classes should be taken as wrapped types by adding or removing a class' *viewAsITWrappedType* class property.

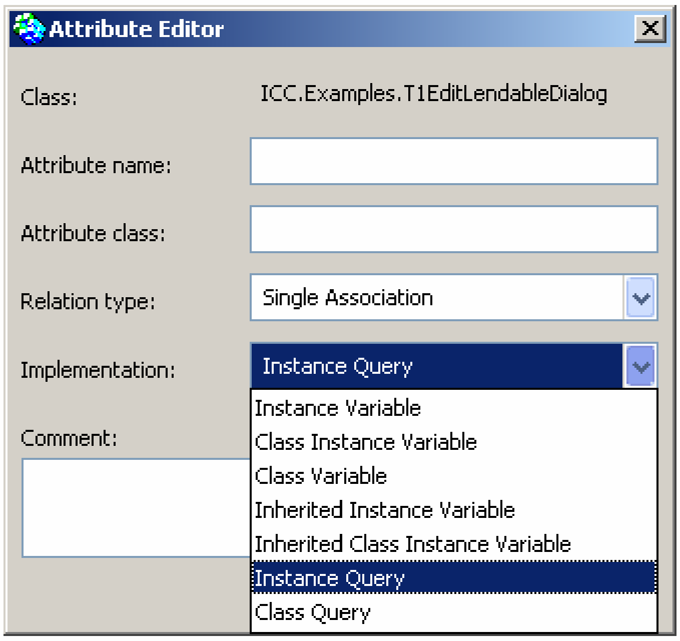
Query Relationships

Smalltalk variables of classes are just a special case of attributes which describe the static structural relationships between objects.

In general, collaborators may be defined dynamically, e.g. through a **parameter** which is passed as an argument of a message to a method. This means that collaboration with objects of the type of the parameter takes place.

In order to describe the collaborators of an objects properly, **ADvance** lets you add associations between classes that have no physical implementation. The way to do this is via the *Attribute Editor*. There you can set an attribute's implementation to **Instance Query** or **Class Query**.

Using this concept allows you to fully describe static structure and behavior of an application. It will significantly increase the expressivity of your diagrams and system documentation.


Attribute Editor

Class: ICC.Examples.T1EditLendableDialog

Attribute name:

Attribute class:

Relation type: Single Association

Implementation: Instance Query

Comment:

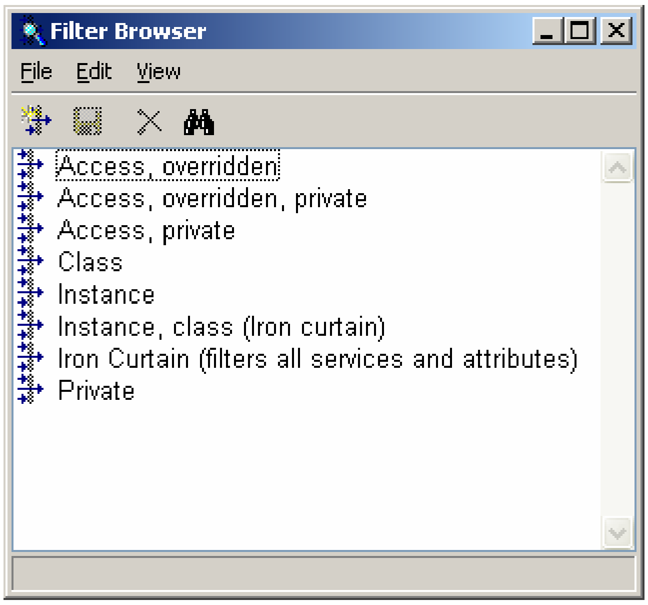
Figure 71. Choosing attribute implementations.

Named Filters

In contrast to the individual filters that were presented in the tutorial chapters, **named filters** are used to quickly assign frequently used filters to classes or diagrams. Named filters are stored globally and are shared between several diagrams.

Open the *Filter Browser* by selecting **Window→Filter Browser** in the *Diagram Painter.*

The *Filter Browser* is opened. It shows an alphabetical list of the available named filters.



Access, overridden
Access, overridden, private
Access, private
Class
Instance
Instance, class (Iron curtain)
Iron Curtain (filters all services and attributes)
Private

Figure 72. The *Filter Browser*.

The *Filter Browser* is a filter drag source. You can assign filters by dragging them from it to a class or diagram in a *Diagram Painter*. It also provides an interface for adding, editing, removing and saving filters.

Open a diagram, switch on services and drag different filters from *the Filter Browser* to the diagram and classes.

General

Inheritance by Implementation

ADvance does depict class interfaces following implementation rather than following semantics. This means that a method which is defined in a superclass of a given class cannot be part of a behavior description of a diagram unless

- the superclass is included in the diagram, or
- the method is redefined in the subclass.

The suggested workaround is to include the superclass, or to redefine - for the sake of presentation - the method in the subclass with just a call to its super implementation.

Hiding Inferred Behavior

The internal **ADvance** heuristics to derive behavior provide a very high match of real behavior. Wrong inferences are those where an object indicates service requests from another object that should not provide the service at all. This situation can - but need not - happen if a wrapped object and a wrapping object are present in the same diagram with the same services.

In order to cope with this situation, it is possible to **hide inferences** graphically. Just select a behavior relationship arrow and select **Hide** from the <Operate> menu.

Hierarchical Subjects

Subjects are represented as classes; every class can become a subject. Hence, a subject can be component of other subjects. This allows you to build subject hierarchies and is used to define different levels of abstractions.

Since subjects are classes, subjects can also have attributes and services. Thus, you can express relations and behavior between subjects. Again, this is used to build high level abstractions of code.

Check the *Subject Browser* for the representation of this concept.

Compound Documentation

ADvance provides graphical output in standard VisualWorks printable form by **File→Print...** and to the clipboard from **Edit→Copy to clipboard.** Additionally, there is textual script documentation.

In order to produce a **compound system document**, a lean way is to produce text documents with some text editor, store these text documents under some name, do the same with **ADvance** output, and have a text editor arrange a document from the set of files according to a descriptor which is kept under configuration control.

The **ADvance** HTML production facility is a way to provide up-to-date documentation based on a de-facto-standard.

Namespaces and Types

In **ADvance**, class names in comments might be used without full qualification of namespaces. In case a class has not been fully qualified in a type declaration, ADvance searches the namespace of the class that is commented, then the enclosing namespaces, finally the imported namespaces.

Modeling

The Methods War

As mentioned above, **ADvance** is a tool that supports and reinforces, but does not enforce good object thinking - so it its highly recommended that a design method is used to encourage good encapsulation and abstractions.

Which OOA/OOD method does **ADvance** support? Conceptually, **ADvance** is not linked to any method at all. Methods are ways how to get around to system descriptions using scenario/use case modeling, object modeling etc.

ADvance supports writing down what has been found and designed during an analysis phase. The concepts **ADvance** offers to do this are at this point very close to what Smalltalk offers as concepts. Some concepts like query relations go beyond basic Smalltalk expressivity. At the current level, **ADvance** offers methodology support that works seamlessly.

In programming languages we know the term syntactic sugar. Similarly, graphical notations with equivalent semantics might be addressed under visual sugar. Sometimes it seems that focus of OOA/OOD methods is not on semantics but on visual syntax. **ADvance** takes the semantics stand that the most important thing in describing complex systems is structure and abstraction. Therefore, visual notation should be lean and should allow focusing on these goals. As UML is being adopted as an important industry standard, **ADvance** basically follows its notation.

Another focus of **ADvance** is on affordable support of iteration. This is a prevailing feature of any approach to building reliable, maintainable systems. This area has not been successfully treated by methodologists; maybe that here **ADvance** brings about new insights how to proceed.

Use Case Modeling

One of the frequently asked questions is whether **ADvance** supports use case modeling. The answer is ‘yes’. However, **ADvance** may express use cases at a formal level that is tied to Smalltalk syntax.

At the core of a use case we have a sequence of collaboration specifications of form[7]:

Initiator>action   Participant>service

where in **ADvance** the Initiator and Participant roles are taken by objects, actions translate into message sends, and services are message handlers implemented through methods. Sequencing and conditionals are expressed through the standard Smalltalk constructs. Preconditions and postconditions of use case actions can be modeled in the same way.

The library example set up by Nancy Wilkerson actually describes use cases. The *ICC.Examples.T1Library* example implements these use cases. Cross-check the prose of the original text with the behavior scripts produced by **ADvance** to get an idea of the scope of expressivity offered.

To give an example, consider the scenario where the borrower wants to check out a book from the library. In prose with added Smalltalk structure, the check out scenario reads as follows:

ICC.Examples.T1Librarian>>checkOut: aLendable for: aBorrower

Check and answer whether aBorrower can borrow, if so, check out aLendable for him.

ICC.Examples.T1Borrower>>canBorrow

See if my fine is less than $100, check if I am over lendable limit and if any of my previously borrowed items is overdue.

ICC.Examples.T1Lendable>>checkOutFor: aBorrower

Calculate and update my due date, then update the borrower.

From this, we derive the formalization expressed in the next picture.

[7] Following *Object Behavior Analysis* by Goldberg/Rubin.

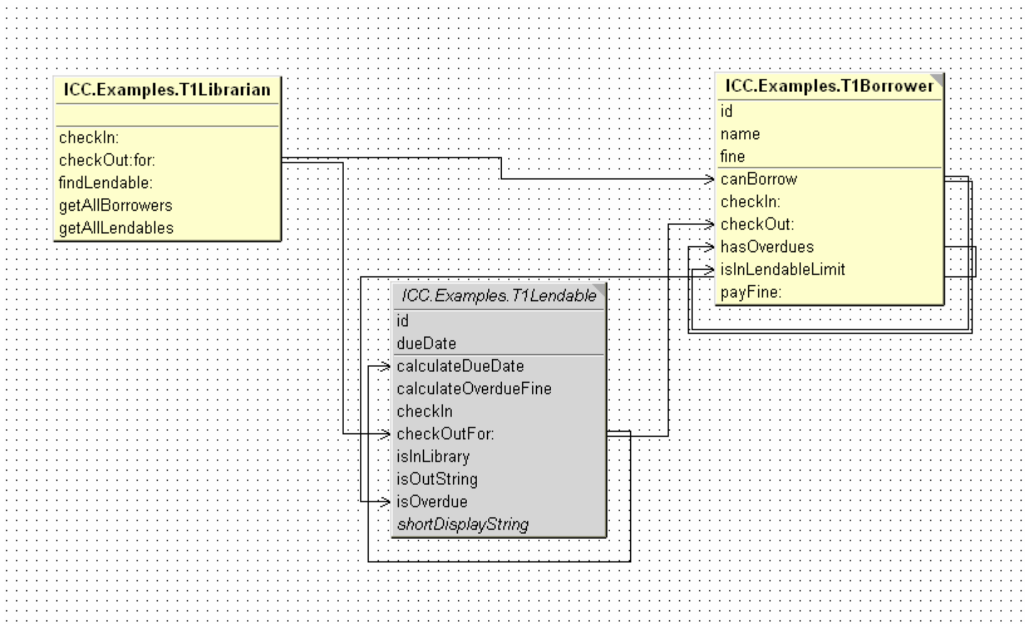


Figure 73. Use case modeling with **ADvance**.

Note that the actions initiated by the borrower are depicted in sequential order from top to bottom.

Event Traces

Events are expressed in different ways in Smalltalk. A direct representation of an event is a message send. Another representation of an event is transferring an event name to a message handler via a self changed: #eventName construct. The message handler then proliferates the event to receivers.

ADvance currently supports event tracing at the level of message sending. Behavior scripts then are event traces. Semantics of events like synchronous or asynchronous mode is abstracted from at the graphics level.

7 User Interface Component Reference

Objective

List all functionality offered by ***ADvance****.*

Workbench

This is similar to the *VisualLauncher* in that the main system functions and browsers are invoked from this launcher. It is automatically opened when **ADvance** is installed in the VisualWorks image. Most menu bar options map directly to the toolbar icons.

The following list of menu options gives a short description of the functionalities belonging to it. Details for the usage is given in the next sections of this guide.

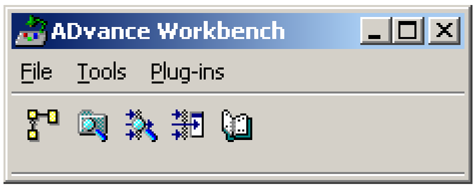


Figure 74. The *ADvance Workbench*.

Tools Pulldown Menu

Diagram Painter Opens a *Diagram Painter*, the main **ADvance** tool.

Subject Browser Opens the *Subject Browser* for creating, editing, removing, renaming, and saving subjects and diagrams.

Filter Browser Opens the *Filter Browser* for creating, editing, removing, and dragging named filters.

Filter Palette Opens the *Filter Palette* as a drag source for named filters.

Documenter Opens the *Documenter*, providing generating of HTML documentation for subjects.

Plug-ins Pulldown Menu

Coding Assistant Opens the extended *Coding Assistant* to assist developers in writing accessor methods and dependency mechanisms.

Comment Generator Opens the *Comment Generator* for generating comment stub for classes that have no comment.

Smalllint Connect Opens the *Smalllint Connect* (see below) which automatically invokes the **Smalllint**[8] code checker.

Diagram Painter

This is the main browser for viewing and modifying code graphically. Editing, filtering, and documentation to both subject and diagrams can be done from this browser. As with the *Workbench* there is both a menu bar and corresponding toolbar. The image shown in the pulldown menu maps directly to its corresponding toolbar icon. Each of he special dialogs opened by menu actions will be described in a later section.

Menu Bar Options

This section describes the basic functionality of each of the *Diagram Painter's* menu bar options. In certain cases, to add clarity, an example or a reference back to the tutorial is added.

File Pulldown Menu

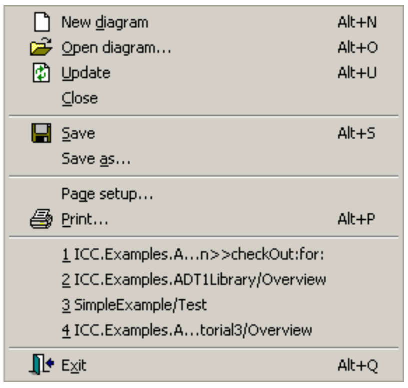


Figure 75. The **File** menu.

[8] Smalllint is part of the Refactoring Browser by John Brant and Don Roberts. Both tools are public available from the Smalltalk archive at http://st.cs.uiuc.edu/Smalltalk, they are integrated parts of VisualWorks 7.x.

New diagram Creates an unnamed diagram with initial layout in the current subject.

Open diagram... Opens an *Open Diagram* dialog to select a diagram to open.

Update Repaints the diagram view with any updated code or class comment changes.

Close Closes this painter window.

Save Saves the current diagram under its existing name.

Save As... Saves the current diagram under a new name.

Page setup... Opens the *Page Setup* (see below) to setup print and page settings.

Print... Opens a *Print Dialog* for defining print parameters and printing diagrams.

Exit Closes all currently open tools.

Edit Pulldown Menu

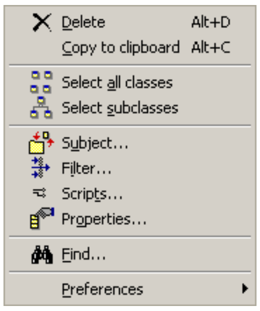


Figure 76. The **Edit** menu.

Delete Deletes the selected components from the current diagram.

Copy to clipboard Copies the current diagram to clipboard.

Select all Selects all components (classes, relationships, texts) in the diagram. All components are now in focus for whatever action the user performs i.e. remove, move.

Select subclasses Selects all subclasses of currently selected classes.

Subject... Opens a *Subject Editor* on the diagram's subject.

Filter... Opens a *Diagram Filter Editor* or *Class Filter Editor*, depending on whether the diagram or a class is in focus.

Scripts... Opens a *Script Selection* dialog on the selected class.

Properties... Opens the *Diagram Properties* or *Class Properties* dialog, depending on whether the diagram is in focus or at least one class is selected.

Find... Asks for a class name to search for and scrolls to that class if found.

Preferences→General Preferences... Opens the *General Preferences Dialog*.

Preferences→Painter Preferences... Opens the *Painter Preferences Dialog*.

View Pulldown Menu

This menu deals with the displaying of graphical components within the diagram.

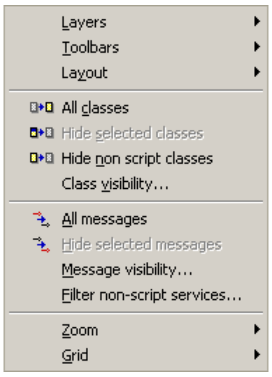


Figure 77. The **View** menu.

All classes Makes all subject classes visible.

Hide selected classes Makes invisible any currently selected class but leaves them in the subject.

Hide non-script classes Makes invisible any classes which do not currently have a script selected but leaves them in the subject.

Class visibility... Opens a dialog that lets you individually select which classes are visible and which are not.

All messages This will redisplay any previously hidden messages.

Hide selected messages This makes the selected messages invisible in the diagram.

Message visibility... Opens a dialog that lets you individually select which messages are visible and which are not.

Filter non-script services... Sets the filter of each class to show only services which are part of a script.

View→Layers Submenu

The different diagram layers can be toggled on and off to show or hide the respecting details in the diagram. Check marks indicate the option is toggled on. The same toggle functionality can be accomplished using the buttons in the lower left corner of the browser.

Inheritance Toggles the inheritance layer in diagram.

Namespaces Toggles the namespace layer in diagram.

Attributes Toggles class attribute names for each class in diagram.

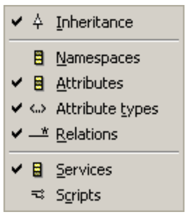


Figure 78. The **View→Layers** submenu.

Attribute types Toggles displaying of attribute types for each class in diagram.

Relations Toggles displaying of has-a relations between classes in diagram.

Services Toggles method names on each class in diagram.

Scripts Toggles displaying of scripts between classes in diagram.

View→Toolbars Submenu

With these menu you can show and hide the toolbar, the add buttons, and the status bar.

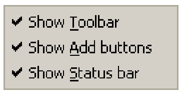


Figure 79. The **View→Toolbars** submenu.

View→Layout Submenu

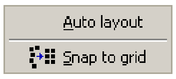


Figure 80. The **View→Layout** submenu.

Auto layout Use auto layout for the start and end point of the currently selected relation lines.

Snap to grid Snaps all classes in diagram to closest grid mark.

View→Zoom Submenu

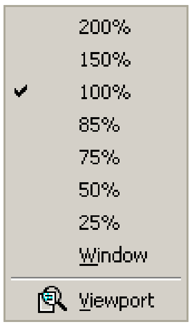


Figure 81. The **View→Zoom** submenu.

200% - 25% Scales the diagram to predefined sizes in order to better fit the window.

Window After selecting this mode, the diagram will be rescaled to fit the painter window each time you resize the window.

Viewport Invokes a rubberbanding select mechanism for zooming in on a specific view in the diagram.

View→Grid Submenu

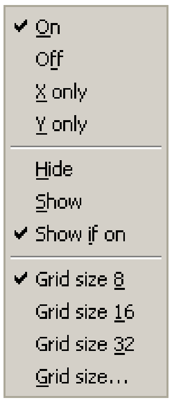


Figure 82. The **View→Grid** submenu.

On Turns grid pattern on

Off Turns grid pattern off. No snapping to grid when selected.

X only Snaps components along x axis.

Y only Snaps components along y axis.

Hide Hides the grid.

Show Shows grid.

Show if on Shows grid always if snapping is on.

Grid size 8 – Grid size 32 Sets grid size to a special number of pixels..

Grid size... Asks for a user defind grid size before setting it.

Add Pulldown Menu

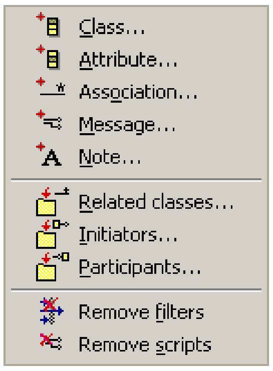


Figure 83. The **Add** menu.

Class... Adds a class to the diagram's subject and to the diagram. It first asks for a class name (pattern). If a pattern contains the wildcard *, a list of matching class names to choose from opens. If the class is not in the image it is created with the *New Class* dialog (see below). The class then is to be placed somewhere in the diagram.

Attribute... A class must be selected prior to selecting this menu option. An *Attribute Editor* providing an interface for attribute definition is opened.

Association... A class must be selected prior to selecting this menu option. A special cursor appears, user has to click on the class to be associated. Once the other class is selected, an *Attribute Editor* with predefined **Attribute class** field is opened.

Message... This option opens an *Add Message* dialog which serves to create message sends during forward engineering. A class must be selected prior to selecting this menu option. A special cursor appears, user has to click on the target class.

Note... Invokes a *Text Editor* for adding free form text directly to the diagram. A special cursor appears, user has click on the spot in the diagram where the text shall be placed. When editing is completed, <Operate> menu action **accept** saves the text. The text then appears in the diagram.

Related classes... A class must be selected prior to selecting this menu option. This opens an *Add Related Classes* dialog containing a list of related classes not already contained in the subject, associated to the selected class by its variable type declaration or inheritance relationship. You may select one or more classes. The selected classes are added to the subject contents and auotmatically drawn in the diagram.

Initiators... A class must be selected prior to selecting this menu option. A search is run that finds all non-subject classes that potentially send a message to the selected class. An *Add Initiator* dialog (similar to *Add Related Classes*) contains a list of the initiating classes. You may select one or more classes. The selected classes are added to the subject and automatically drawn in the diagram.

Participants... This is similar to **Add→Initiators** but finds all non-subject classes that potentially receive a message from the selected class.

Remove filters Removes all class filters of currently selected classes.

Remove scripts Removes all scripts of currently selected classes.

Window Pulldown Menu

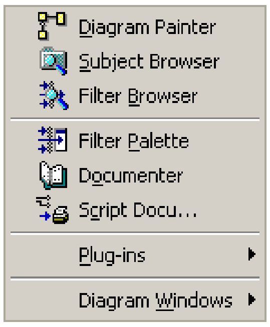


Figure 84. The **Window** menu.

Diagram Painter Opens a new empty *Diagram Painter* window.

Subject Browser Opens the *Subject Browser*, providing actions for creating, editing, removing, renaming and saving subjects and diagrams.

Filter Browser Opens the *Filter Browser*, providing actions for creating, editing, removing, and dragging named filters.

Filter Palette Opens the *Filter Palette*, a drag source for named filters.

Documenter Opens the *Documenter* to generate HTML documentation for a subjects.

Script Docu... Invokes a *Script Documentation* dialog to display a list of the currently shown diagram scripts (hidden scripts are excluded).

Window→Plug-ins Submenu

Coding Assistant Opens the extended *Coding Assistant*, assisting developers in writing accessor methods and dependency mechanisms.

Comment Generator Opens the *Comment Generator*, allowing generation of comment stubs for classes that have no comment.

Smalllint Connect Opens the *Smalllint Connect* to automatically invoke the **Smalllint** code checker.

Window→Diagram Windows Submenu

This menu contains a menu item for every open diagram painter to ease switching between different diagrams.

Pop-up Menus

There are four pop-up menus associated with the diagram and its selected components (classes, relations and texts). When no or more than one component is selected the diagram itself is in focus concerning menu invocation. If a class is selected the pop-up menu will reflect class options instead of diagram options. If a line segment or text is selected the menu options reflect the corresponding menu options.

Diagram Menu

This menu is for operations on the diagram itself. Most options are repeated from the menu bar options - they are noted as such. Some of the options need a non-empty selection to be enabled.

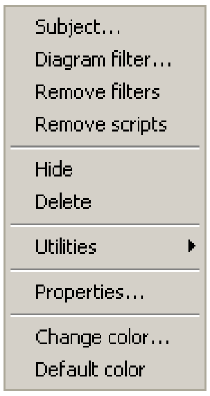


Figure 85. The Diagram pop-up menu.

Subject... See section *Edit Pulldown Menu*.

Filter... See section *Edit Pulldown Menu*.

Remove filters See section *Add Pulldown Menu*.

Remove scripts See section *Add Pulldown Menu*.

Hide See section *View Pulldown Menu*.

Delete See section *Edit Pulldown Menu*.

Properties... Opens the *Diagram Properties* dialog to enter a description and review diagram data.

Class Menu

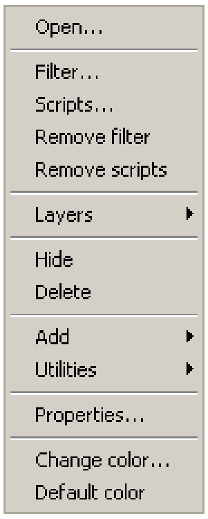


Figure 86. The Class pop-up menu.

Open... Opens a *Hierarchy Browser* on the selected class.

Filter... See section *Edit Pulldown Menu*.

Scripts... See section *Edit Pulldown Menu*.

Remove filters See section *Add Pulldown Menu*.

Remove scripts See section *Add Pulldown Menu*.

Layers See section *View Pulldown Menu*.

Hide See section *View Pulldown Menu*.

Delete See section *Edit Pulldown Menu*.

Properties... Open the *Class Properties* dialog.

Change color... Open the *Choose Color* dialog on the selected class.

Default color Change the selected class‘ color to the default color.

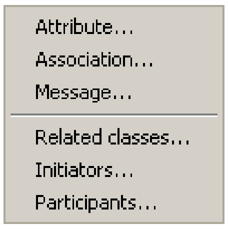


Figure 87. The **Add** submenu.

Add→Attribute... See section *Add Pulldown Menu*.

Add→Association... See section *Add Pulldown Menu*.

Add→Message... See section *Add Pulldown Menu*.

Add→Related classes... See section *Add Pulldown Menu*.

Add→Initiators... See section *Add Pulldown Menu*.

Add→Participants... See section *Add Pulldown Menu*.

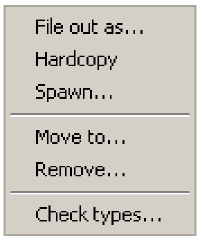


Figure 88. The **Utilities** submenu.

Utilities→File out as... Same as the system browser **file out as...** facility.

Utilities→Hardcopy Same as system browser **hardcopy**.

Utilities→Spawn... Opens a *Class Browser* on the selected class.

Utilities→Move To... This is used to specify a new category for the selected class.

Utilities→Remove... Removes the selected class from the VisualWorks image.

Utilities→Check Types... Checks type definitions of the selected class and opens an *Type Checking Report* window if the class is not properly typed.

Association Menu

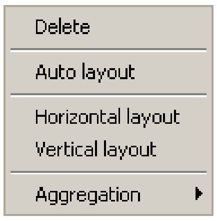


Figure 89. The Association pop-up menu.

Delete Removes the selected association, the code is changed accordingly.

Auto layout Changes the start and end points of the association line back to the default.

Horizontal layout Changes start and end points of the association to equal vertical coordinates (if possible). This action has no effect for associations with same start and end class.

Vertical layout Changes start and end points of the association to equal horizontal coordinates (if possible). This action has no effect for associations with same start and end class.

Horizontal and vertical layout will last even when class symbols are moved. To break up these automatic layouts, you can move one of the classes so that the horizontal respective vertical layout is not possible.

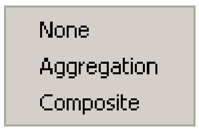


Figure 90. The **Aggregation** submenu.

The following figure shows one of the Tutorial diagrams, enhanced by displaying different aggreation types. The Borrowers of a Library are shown as aggreagtion, the lendables are shown as composite.

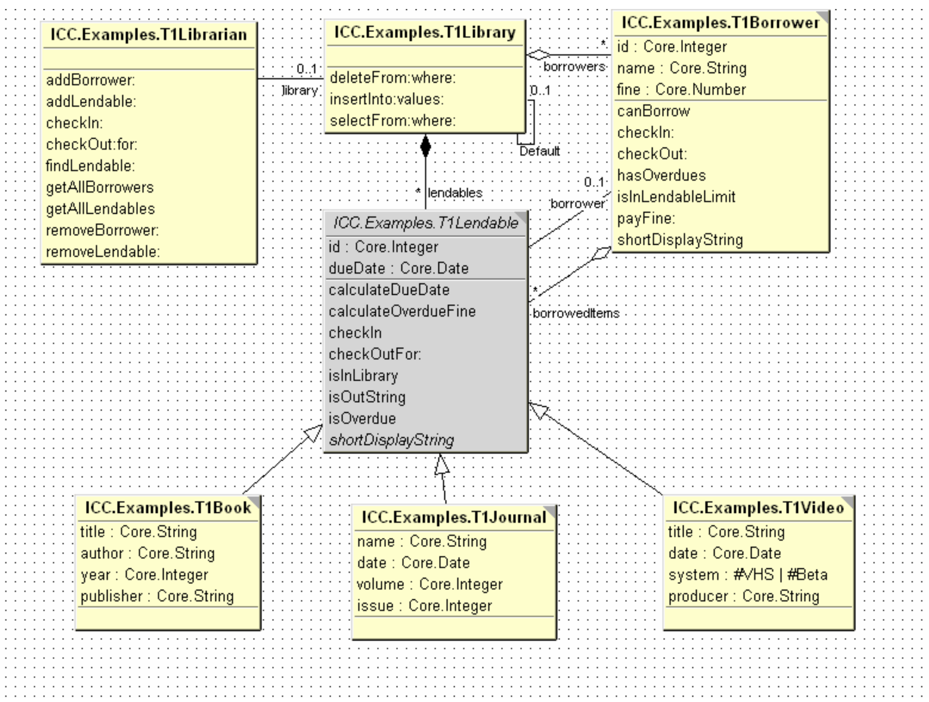


Figure 91. Different aggregation symbols in a diagram.

Message Menu

The pop-up menu for a selected message provides just the action **Hide** to remove the message from the visible diagram parts. No code is changed.

Text Menu

Edit Opens a text editor on the text.

Remove Removes the text.

Shortcuts

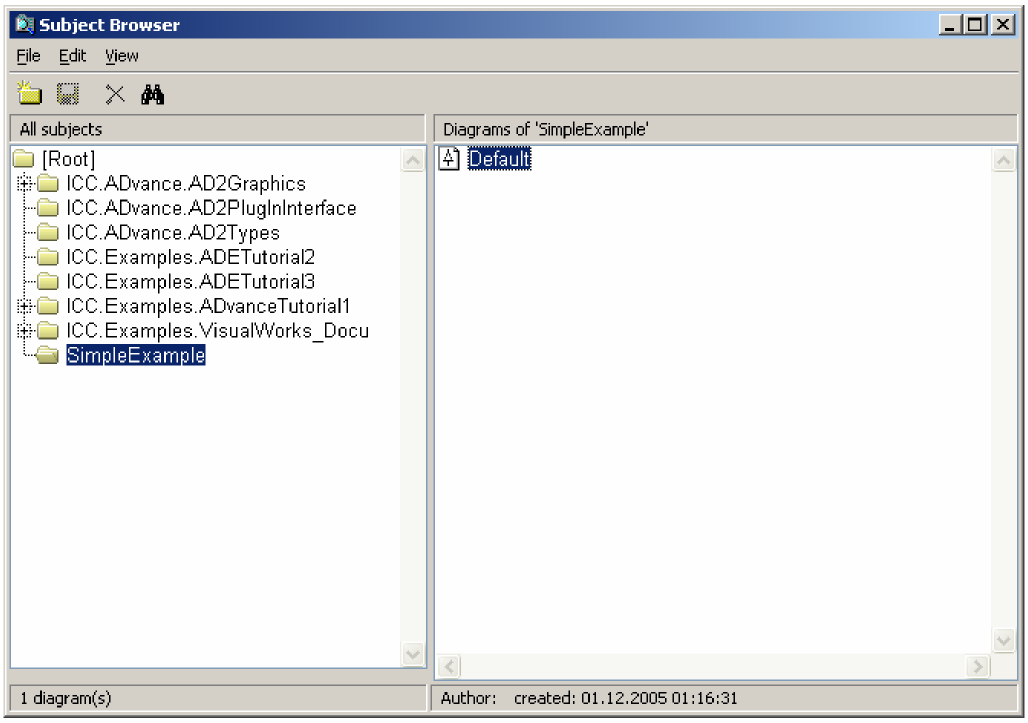
The toolbar on top of the diagram pane offers direct access to the most commonly used menu bar options. For quick access to the **Add** menu options, you may use the buttons at the left side of the window. Special access for toggling diagram layers is available through the respective toolbar buttons.

The most frequently used commands can be invoked through double-clicking on either a class or the diagram. These shortcuts can be changed in the *Painter Preferences Shortcuts* tab.

Many dialog panes have double-click actions as shortcut for accepting the dialog, e.g. double-clicking on an item in the **Diagram Name** listbox of the *Open Diagram* dialog opens the selected diagram.

Tools

Subject Browser


Subject Browser
File
Edit
View
All subjects
Diagrams of 'SimpleExample'
[Root]
ICC.ADvance.AD2Graphics
ICC.ADvance.AD2PlugInInterface
ICC.ADvance.AD2Types
ICC.Examples.ADETutorial2
ICC.Examples.ADETutorial3
ICC.Examples.ADvanceTutorial1
ICC.Examples.VisualWorks_Docu
SimpleExample
Default

1 diagram(s)

Author: created: 01.12.2005 01:16:31

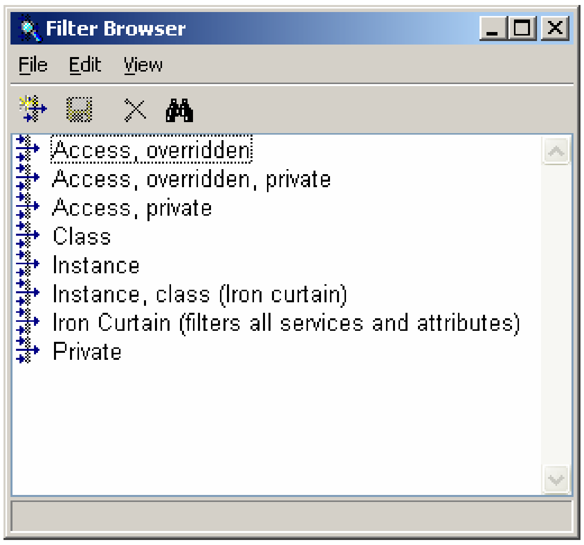
Figure 92. The *Subject Browser*.

The *Subject Browser* is similar to the *Open Diagram* dialog. In the left pane it displays a tree of subjects, while the right pane shows diagrams, if a subject is selected.

The tool provides an interface to browse through the subject hierarchy and the related diagrams. It is also used for creating, deleting, editing, saving, and finding subjects and diagrams.

Note: New subjects are created as children of the currently selected subject. So, if you want to create a subject that should be placed under the root of the subject tree, you should select the *[Root]* subject first.

Filter Browser

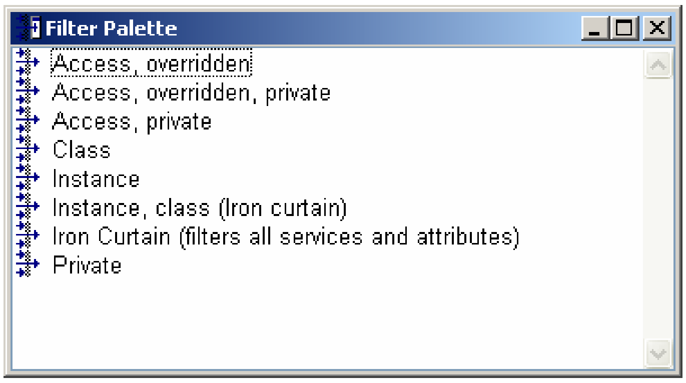


Access, overridden
Access, overridden, private
Access, private
Class
Instance
Instance, class (Iron curtain)
Iron Curtain (filters all services and attributes)
Private

Figure 93. The *Filter Browser*.

The *Filter Browser* is a filter drag source. You can assign filters by dragging them from it to a class or diagram in a *Diagram Painter*. It also provides and interface for adding, editing, removing and saving filters.

Filter Palette



Access, overridden
Access, overridden, private
Access, private
Class
Instance
Instance, class (Iron curtain)
Iron Curtain (filters all services and attributes)
Private

Figure 94. The *Filter Palette.*

The *Filter Palette* serves as drag source for dragging filters onto the *Diagram Painter*. Test the different filters and their impact on a diagram display.

Documenter

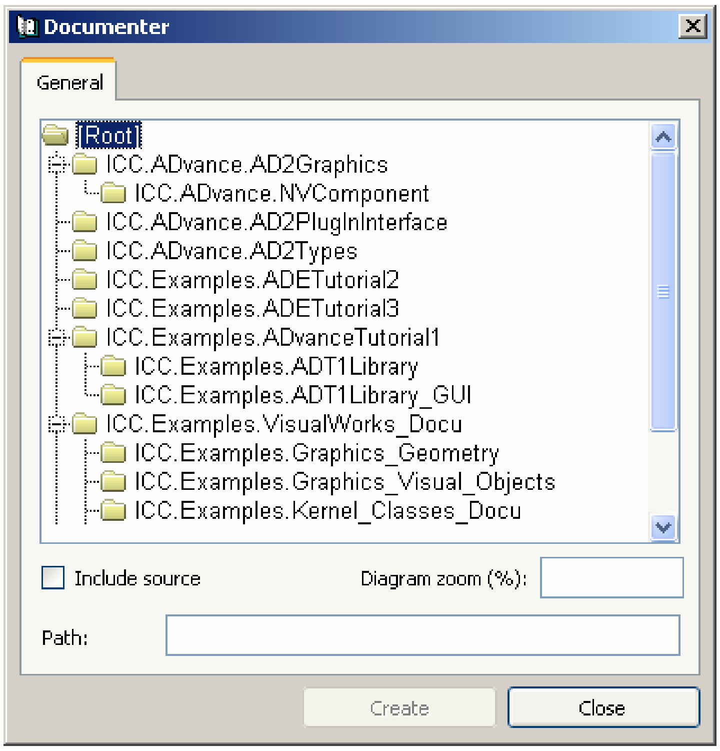

Documenter
General
[Root]
ICC.ADvance.AD2Graphics
ICC.ADvance.NVComponent
ICC.ADvance.AD2PlugInInterface
ICC.ADvance.AD2Types
ICC.Examples.ADETutorial2
ICC.Examples.ADETutorial3
ICC.Examples.ADvanceTutorial1
ICC.Examples.ADT1Library
ICC.Examples.ADT1Library_GUI
ICC.Examples.VisualWorks_Docu
ICC.Examples.Graphics_Geometry
ICC.Examples.Graphics_Visual_Objects
ICC.Examples.Kernel_Classes_Docu
Include source
Diagram zoom (%):
Path:
Create
Close

Figure 95. The *Documenter*.

The *Documenter* is a tool that automatically creates a compound HTML documentation for a subject. The documentation includes the subject’s diagrams, textual script representations, a HTML page for the table of contents, and optional definitions for the subject classes.

Choose Color Dialog

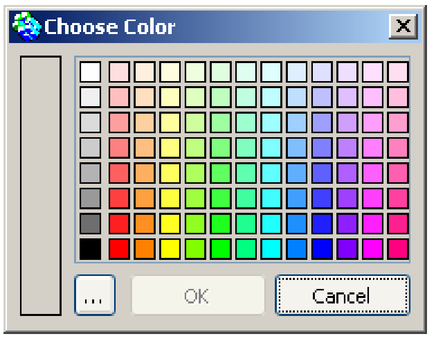
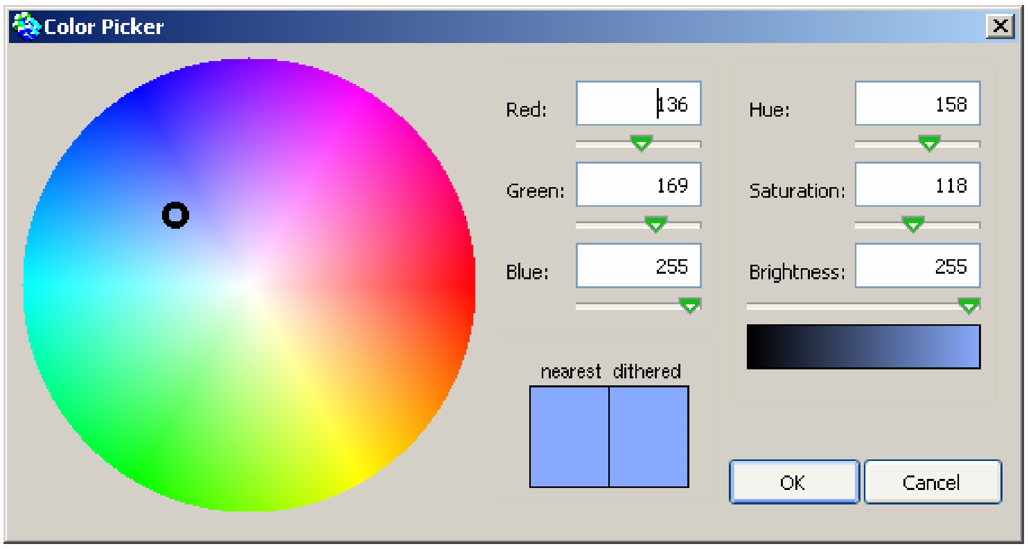


Figure 96. The *Choose Color Dialog.*

The *Choose Color Editor* is used to change a selected class' background color. As soon as you select one of the displayed 104 predefined colors or customized another one in Color Picker Dialog (by clicking the **...** button), the **OK** button will be enabled to confirm your choice.



Red: 136
Green: 169
Blue: 255
Hue: 158
Saturation: 118
Brightness: 255
nearest dithered

Figure 97. The *Color Picker.*

The *Color Picker* is intended to define colors according to common color defining systems RGB or HSB. The settings of both systems compete, so if you change one, the other is adjusted automatically. Confirming the dialog by clicking the **OK** button submits the composed color to the *Choose Color Dialog.*

Plug-ins

The **ADvance** plug-ins can be invoked either from the *Workbench*’s **Plug-in** menu or from *the Diagram Painter*’s **Window→Plug-ins** menu. In the latter case, the plug-ins have access to the painter’s current subject, diagram and selection.

Coding Assistant

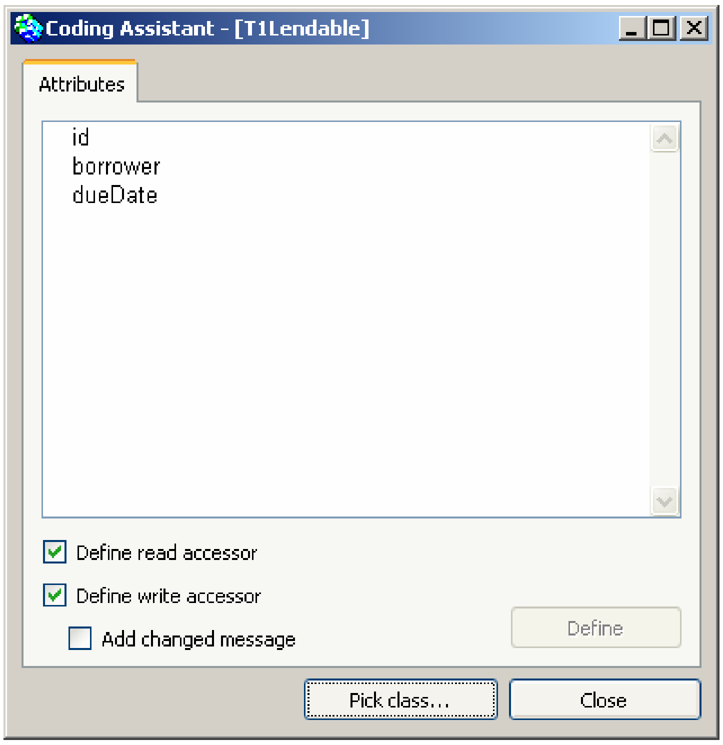
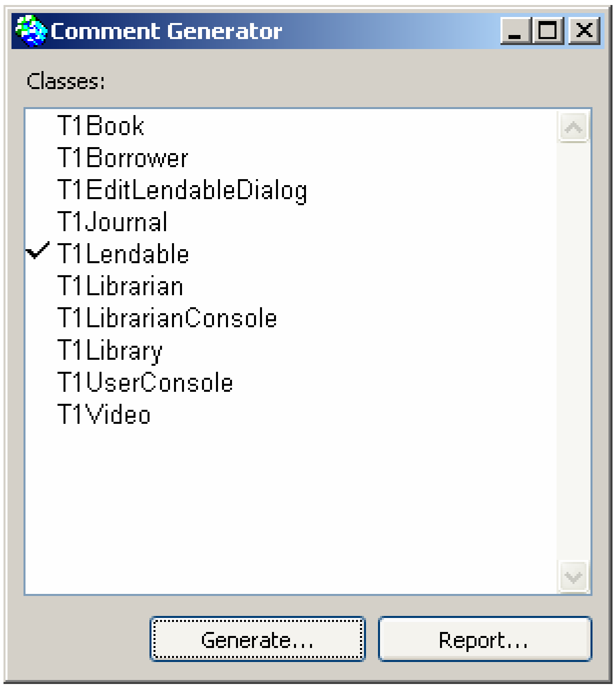


Figure 98. The *Coding Assistant* plug-in.

The *Coding Assistant* assists in writing accessors and dependency mechanisms. If it is invoked from a *Diagram Painter* that has a class selected, the selection will be used as preset class.

Comment Generator



T1Book
T1Borrower
T1EditLendableDialog
T1Journal
✓ T1Lendable
T1Librarian
T1LibrarianConsole
T1Library
T1UserConsole
T1Video

Figure 99. The *Comment Genrator* plug-in.

This tool uses a static inference to generate class comment stubs. If it is invoked from a *Diagram Painter* it focuses on the painter’s current class selection.

Smalllint Connect

The *Smalllint Connect* invokes the **Smalllint** code checker. If it is invoked from a *Diagram Painter* it focuses on the painter’s current subject.

Helper Dialogs

Open Diagram Dialog

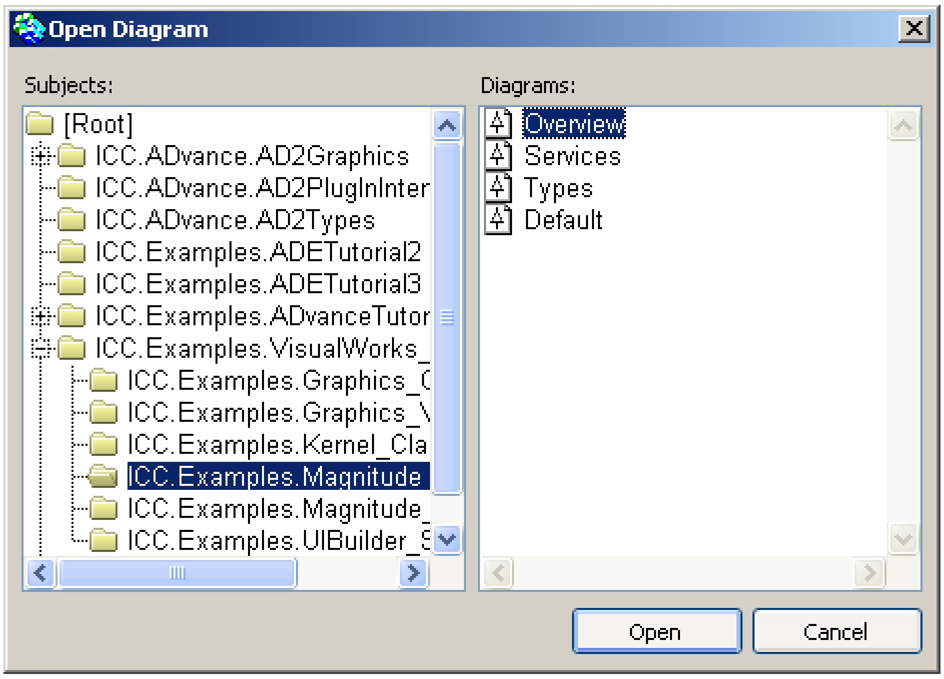
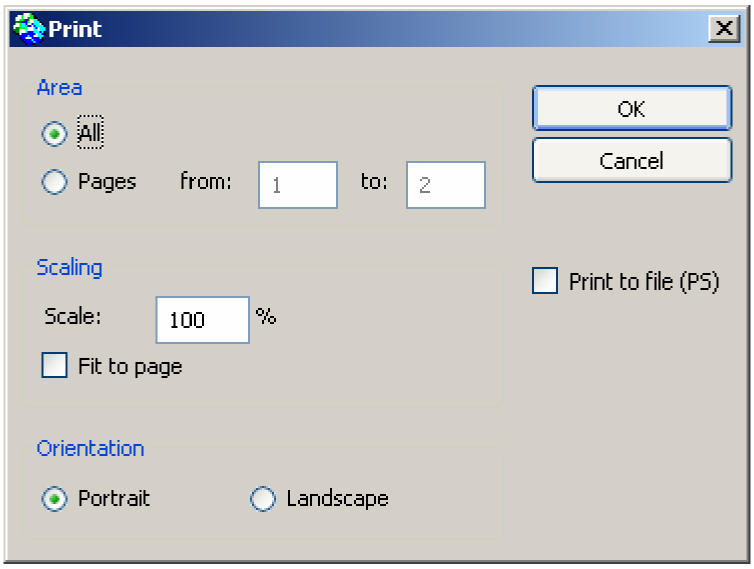


Figure 100. The *Open Diagram* dialog.

This dialog provides an interface to show existing diagrams in the *Diagram Painter*. Select a subject from the left pane. This displays all its diagrams in the right pane. Select a diagram to view then select **Open**.

Print Dialog



Area

[✓ All]

Pages from: 1 to: 2

Scaling

Scale: 100 %

☐ Fit to page

Orientation

[✓ Portrait] Landscape

☐ Print to file (PS)

Figure 101. The *Print Dialog*.

This dialog allows the user to select which diagram pages to print. **ADvance** automatically segments the diagram into the correct number of pages if more than one is needed.

Area

- **All** Prints all pages of diagram.
- **Pages** Print all pages in the selected range.

Scaling

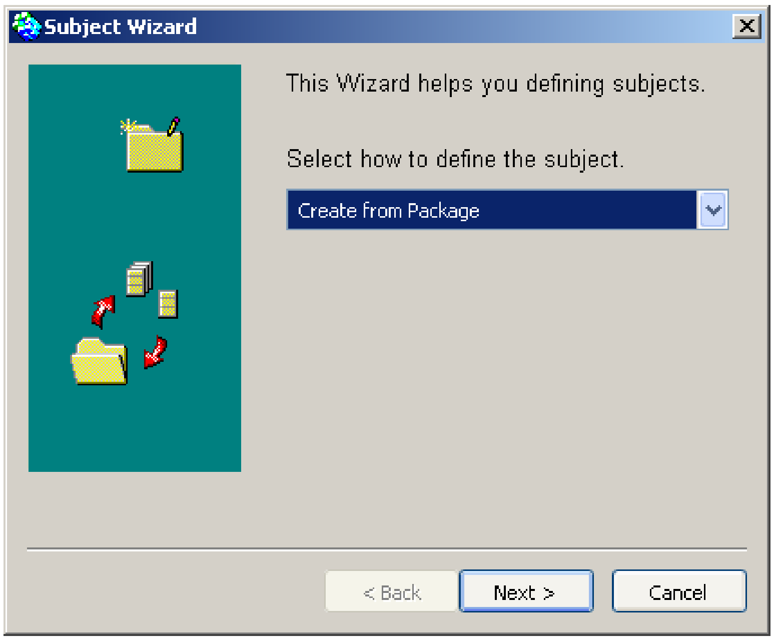
- **Scale** XX%; User can select scale for printing.
- **Fit to page** If checked, the diagram is automatically scaled to fit one page.

Orientation

- **Portrait** Vertical presentation.
- **Landscape** Horizontal presentation.

Print to file If checked, diagram is saved as file only and is not send to printer.

Subject Wizard



This Wizard helps you defining subjects.

Select how to define the subject.

Create from Package

Figure 102. The *Subject Wizard*.

The *Subject Wizard* is used to create new subjects. It can create empty subjects or subjects with an initial content. In the latter case you may either start with a VisualWorks category, a VisualWorks parcel, a VisualWorks package, the classes in the ChangeSet, or you can copy all classes from an existing subject.

In all cases you can define the subject name and choose a class that should bear the subject definition.

Subject Editor

The *Subject Editor* is available from the *Subject Browser* or via Edit menu of a *Diagram Painter*.

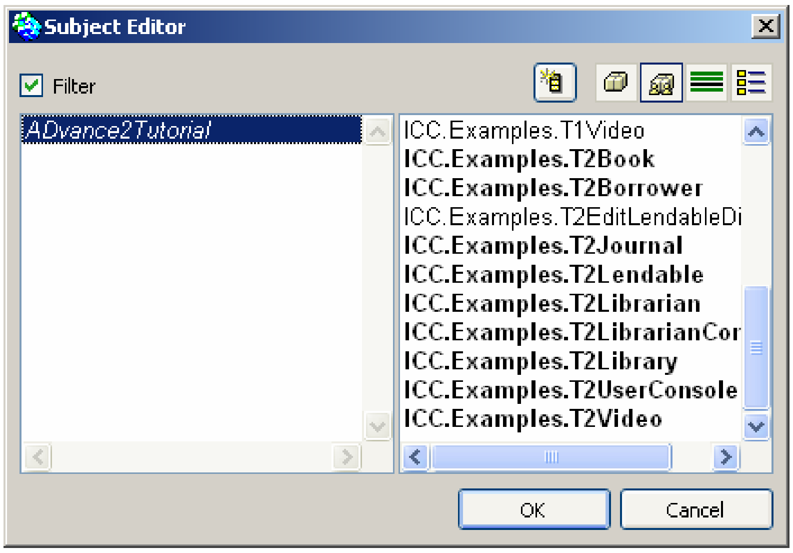
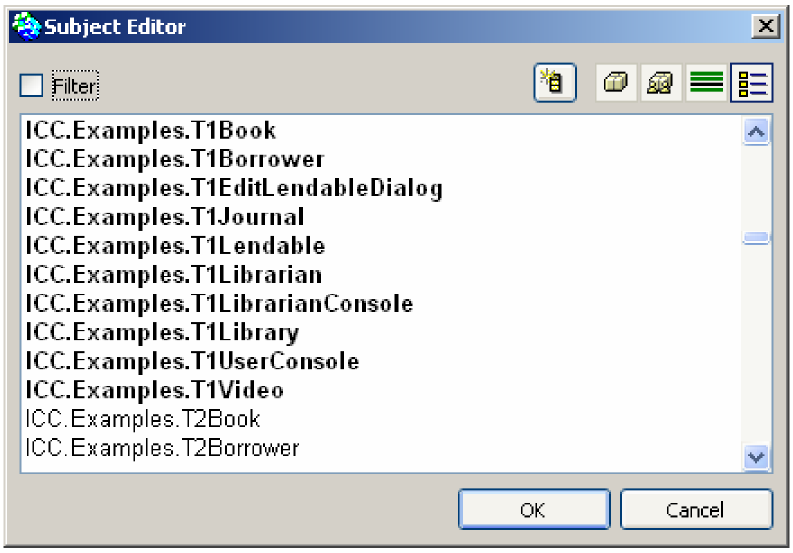

Subject Editor
Filter
ADvance2Tutorial
ICC.Examples.T1Video
ICC.Examples.T2Book
ICC.Examples.T2Borrower
ICC.Examples.T2EditLendableDi
ICC.Examples.T2Journal
ICC.Examples.T2Lendable
ICC.Examples.T2Librarian
ICC.Examples.T2LibrarianCor
ICC.Examples.T2Library
ICC.Examples.T2UserConsole
ICC.Examples.T2Video
OK
Cancel

Figure 103**.** The *Subject Editor* in package mode, **Filter** checked.

It has four different views to modify the subject. The **organization view** offers two panes, displaying all categories in the left pane and the contained classes in the right pane. This view is best if you want to add and remove categories, or if you want to find a class by category. The other views from left to rigth are **parcel view**, **package view**, and **alphabetical view** which offers a sorted list of all classes.

In all views containters and classes that are in the subject are shown in bold text. Categories that are partly in the subject are in italics. If the **Filter** option is checked, the editor shows only containers (parcels, packages, categories) that intersect with the subject, respectively classes (in alphabetical view) that are part of the subject.

You can add and remove containers or classes by a variety of actions in the pop-up menues of the different panes. They allow you to add/remove single items or using patterns. The editor also provides actions to add classes from the change set, a *VisualWorks* parcel or other subjects. Furthermore you can add and remove a class‘ super and subclasses.

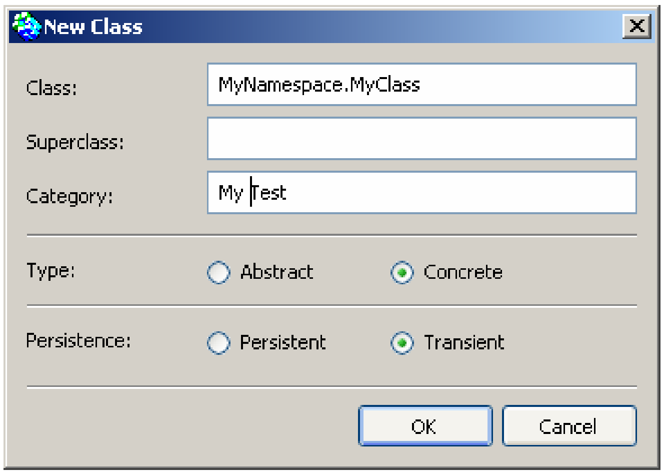


ICC.Examples.T1Book
ICC.Examples.T1Borrower
ICC.Examples.T1EditLendableDialog
ICC.Examples.T1Journal
ICC.Examples.T1Lendable
ICC.Examples.T1Librarian
ICC.Examples.T1LibrarianConsole
ICC.Examples.T1Library
ICC.Examples.T1UserConsole
ICC.Examples.T1Video
ICC.Examples.T2Book
ICC.Examples.T2Borrower

Figure 104. The *Subject Editor* in alphabetical mode, **Filter** unchecked.

New Class Dialog

The *New Class Dialog* is opened whenever a new class has to be created like a new class to hold the information of a new subject, or a class explicitly added to a diagram in the *Diagram Painter* or in the *Subject Editor*.



Class: MyNamespace.MyClass
Superclass:
Category: My Test
Type: Abstract [✓ Concrete]
Persistence: Persistent [✓ Transient]

Figure 105. The *New Class* dialog.

Class: The qualified name of the new class.

Superclass : The superclass name of the new class.

Category: Name of category in which the new class will reside.

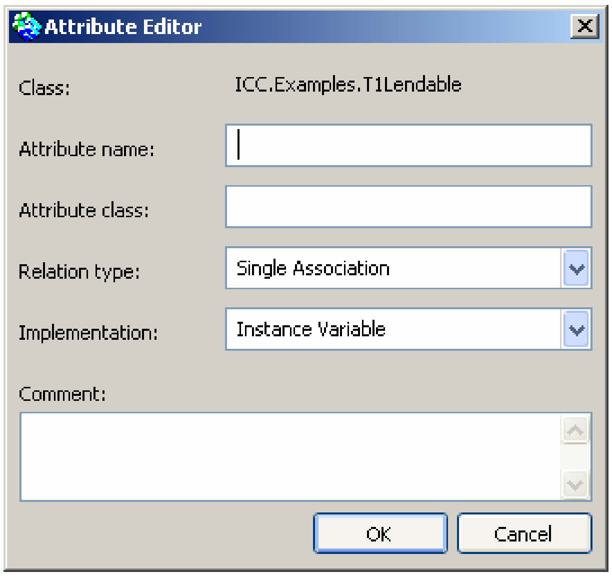
Type: Select **Abstract** or **Concrete**, to indicate whether the class is intended to be instantiated or not.

Persistence: Select **Persistent** or **Transient** to indicate whether the class is intended to be maintained in permanent storage.

Note: Depending on the work context the dialog was started for, the package of the new class is determined in different ways – the package for a new diagram or subject member class is determined by the policy given in System Settings, the policy for a new subject class is taken from the **ADvance** general preferences.

Attribute Editor

The *Attribute Editor* is started for adding or editing class attributes in a *Diagram Painter*.



Attribute Editor

Class: ICC.Examples.T1Lendable

Attribute name:

Attribute class:

Relation type: Single Association

Implementation: Instance Variable

Comment:

Figure 106. The *Attribute Editor*.

Attribute name: The name of the new attribute.

Attribute class: The class (or type) name (or pattern for it) of the attribute.

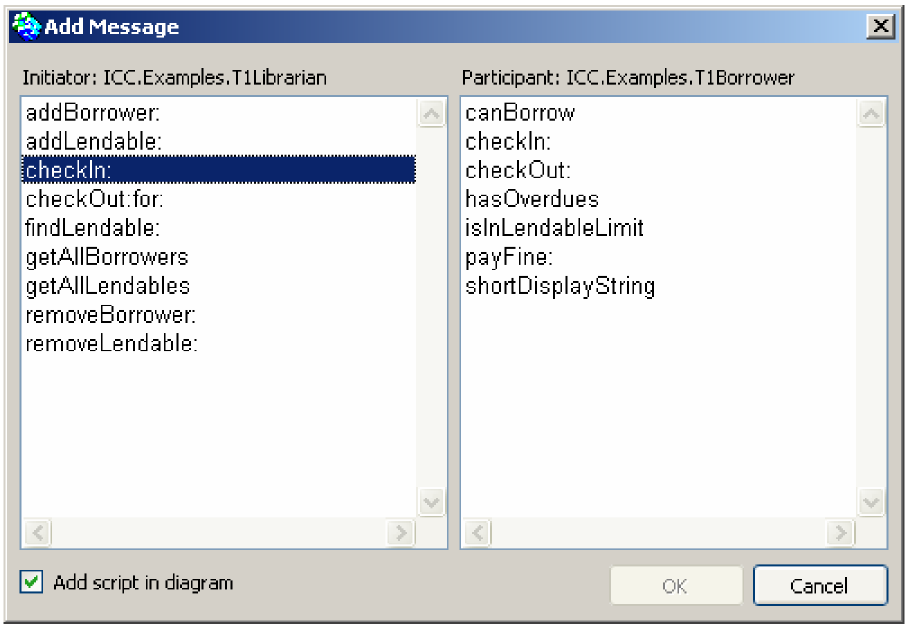
Relation type: Defines whether the relation is 1:1 or 1:many. In the case that it is a 1:many relation you may specify the collection type, i.e. *Set*, *OrderedCollection*, etc.

Implementation: The variable implementation. Choices are offered for all Smalltalk class definition variable types, for inherited variables and for **Query relations** (see *Additional Concepts*) that express a relation without having a variable representation.

Comment: Free form text describing the attribute.

Note: The attribute information is stored in the class comment or in a special class method, depending on the **ADvance** general preferences.

Add Message Dialog



Initiator: ICC.Examples.T1Librarian

addBorrower:
addLendable:
checkIn:
checkOut:for:
findLendable:
getAllBorrowers
getAllLendables
removeBorrower:
removeLendable:

Participant: ICC.Examples.T1Borrower

canBorrow
checkIn:
checkOut:
hasOverdues
isInLendableLimit
payFine:
shortDisplayString

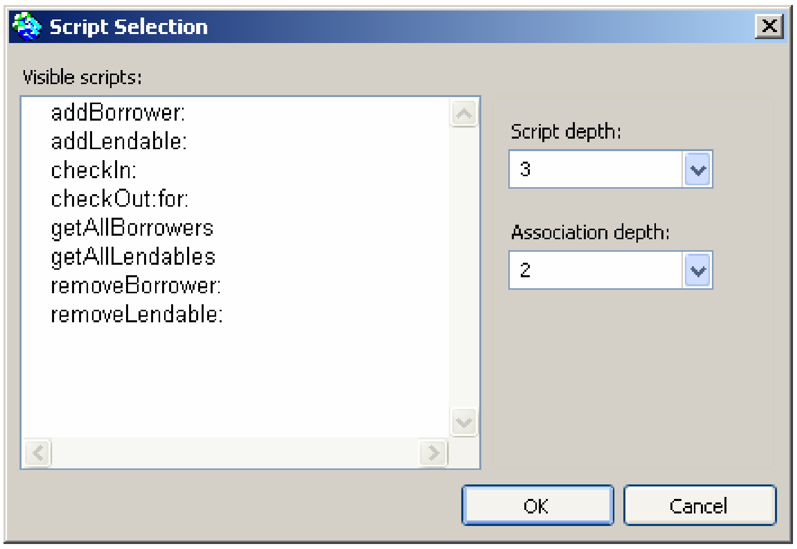
Figure 107. The *Add Message Dialog*.

The *Add Message* dialog serves to create message sends during forward engineering in the *Diagram Painter*. A class must be selected prior to selecting this menu option.

When the cursor appears, select the collaborating class. Select a method name in the **Initiator** pane and then one in the **Participant** pane. If you want the script representation to show up immediately, then check **Add script in diagram**. Services must exist on both classes prior to using this option.

When accepted, **ADvance** will automatically add a line of code in the initiator method as such: *ParticipantClass* new *participantMethodName*. As you can see this is just a placeholder for the correct code to be added later. Multiple messages can be added to any one initiator method.

Script Selection Dialog



addBorrower:
addLendable:
checkIn:
checkOut:for:
getAllBorrowers
getAllLendables
removeBorrower:
removeLendable:

Script depth: 3

Association depth: 2

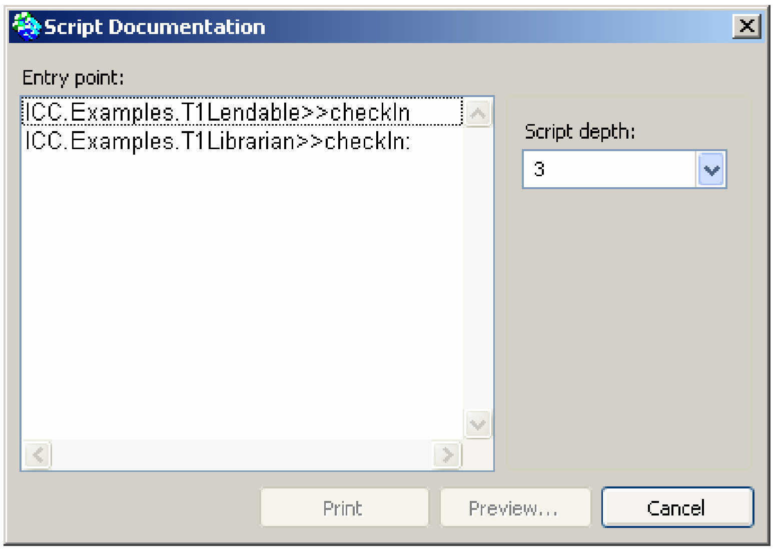
Figure 108. The *Script Selection* dialog.

This dialog is used for adding and removing scripts from behavior diagrams in a *Diagram Painter*. The dialog is available through the operate menu on a single class selection.

Script depth: The number of message sends to be followed from script starting method. Depth of 1 means that only messages found in the starting method become connected as target services. Depth of 2 means that in addition to the depth 1 behavior arrows are drawn from target services to their potential message recipients, and so forth.

Association depth: The potential message recipients are only searched in the set of related classes, i.e. classes connected by an association. An association depth greater than one means that in addition to the directly related classes the related classes of related classes are taken also.

Script Documentation Dialog



ICC.Examples.T1Lendable>>checkIn
ICC.Examples.T1Librarian>>checkIn:

Script depth: 3

Figure 109. The *Script Documentation Dialog*.

This dialog can be reached by pull-down menu option **Window→Script docu…** in ther *Diagram Painter*. It provides an interface for generating Script documentation for the scripts visible in the diagram. Check the desired scripts, then select **Print** or **Preview...** to print or preview the document.

You can choose to document the script to any depth - which may not necessarily be the same as shown in the diagram. If no script is drawn or is currently hidden, a dialog will notify you that no scripts are available.

Diagram and Class Filter Editor

Diagram Filter Editor and *Class Filter Editor* provide interfaces for editing filters (for details on filtering see the *Overview* chapter). Both editors have an identical interface but focus on two different filters: **Diagram filter**, which is applied to all classes having no own filter defined, and **Class filters** respectively. Each of the four filtering categories **Instance methods**, **Class methods**, **Attributes**, and **Special** (semantic) may be edited on its respective editor's notebook page.

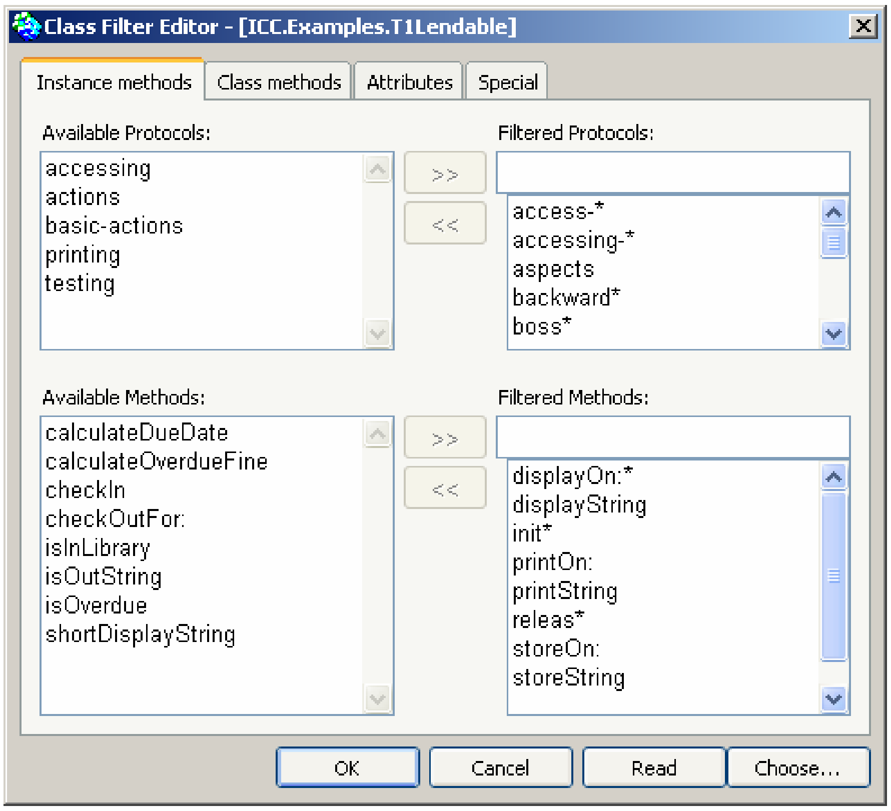
The buttons at the bottom of the dialog provide actions for the whole filter and not only the selected notebook page.

OK button: Accepts the current changes, saves the filter for the selected classes (or diagram if selection is empty) and closes the dialog.

Cancel button: Discards the current changes, leaves the filter in the previous state and closes the dialog.

Read button: Reread current filter to override the changes done so far and restart editing on the latest saved state.

Choose… button: Opens a list of the predefined filters to copy as a starting point.



Available Protocols:
accessing
actions
basic-actions
printing
testing

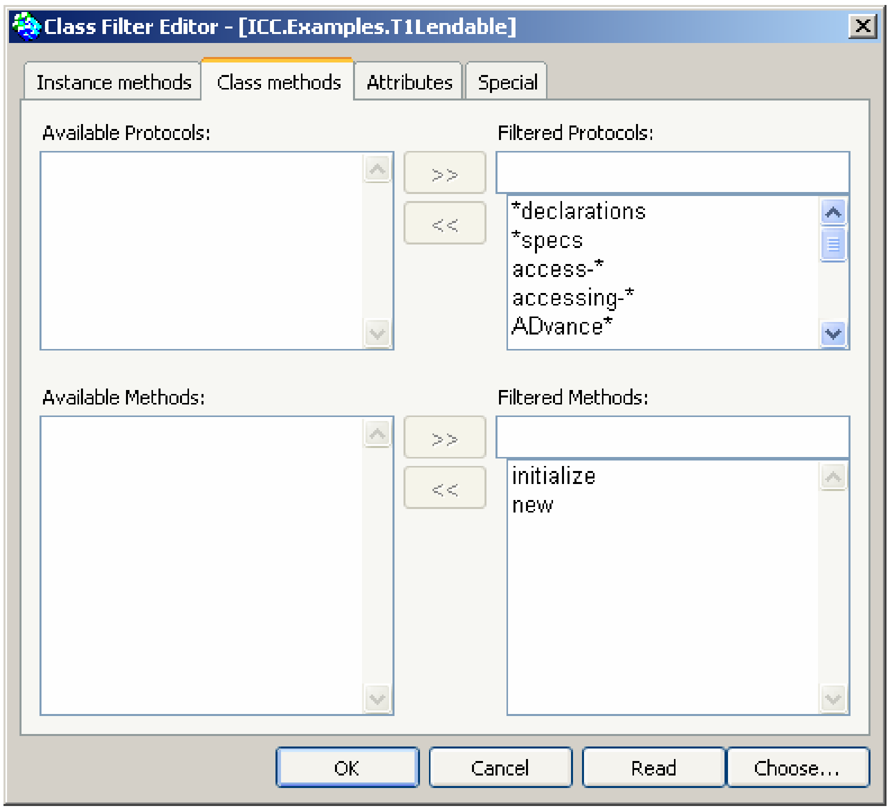
Filtered Protocols:
access-*
accessing-*
aspects
backward*
boss*

Available Methods:
calculateDueDate
calculateOverdueFine
checkIn
checkOutFor:
isInLibrary
isOutString
isOverdue
shortDisplayString

Filtered Methods:
displayOn:*
displayString
init*
printOn:
printString
releas*
storeOn:
storeString

Figure 110. The *Instance methods* page of a *Class Filter Editor*.

Instance methods: The two protocol panes in the upper half show available and filtered instance protocol names, the method panes in the lower half do the same for instance method names. Protocols and methods can be moved from filtered to available and vice versa with the **<<** and **>>** buttons or via pop-up menu commands. Name patterns can be added by entering a text with contained wildcard characters * into the input fields on top of the filtered lists and accepting with operation menu.



*declarations
*specs
access-*
accessing-*
ADvance*

initialize
new

Figure 111. The *Class methods* page of a *Class Filter Editor*.

Class methods: The two protocol panes in the upper half show available and filtered class protocol names, the method panes in the lower half do the same for class method names. Protocols and methods can be moved from filtered to available and vice versa with the **<<** and **>>** buttons or via pop-up menu commands. Name patterns can be added by entering a text with contained wildcard characters * into the input fields on top of the filtered lists and accepting with operation menu.

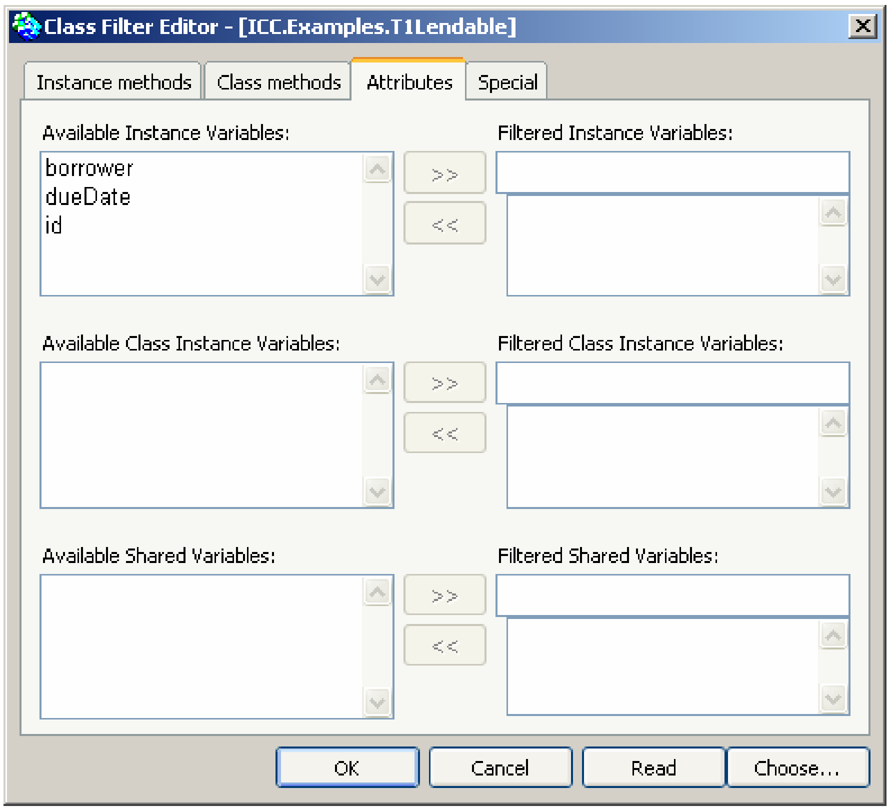
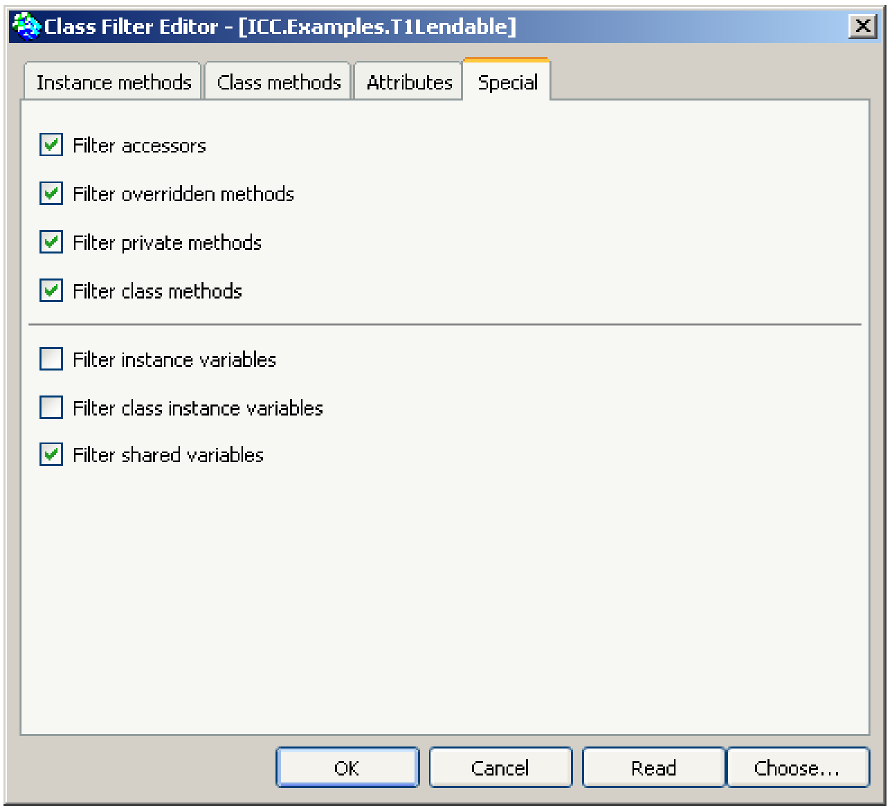

Class Filter Editor - [ICC.Examples.T1Lendable]
Instance methods
Class methods
Attributes
Special
Available Instance Variables:
borrower
dueDate
id
Filtered Instance Variables:
>>
<<
Available Class Instance Variables:
Filtered Class Instance Variables:
Available Shared Variables:
Filtered Shared Variables:
OK
Cancel
Read
Choose...

Figure 112. The *Attributes* page of a *Class Filter Editor*.

Attributes: The two instance variable panes in the upper third show available and filtered instance variable names, the variable panes in the center and the lower third do the same for class instance variable names and shared variable names. Protocols and methods can be moved from filtered to available and vice versa with the **<<** and **>>** buttons or via pop-up menu commands. Name patterns can be added by entering a text with contained wildcard characters * into the input fields on top of the filtered lists and accepting with operation menu.

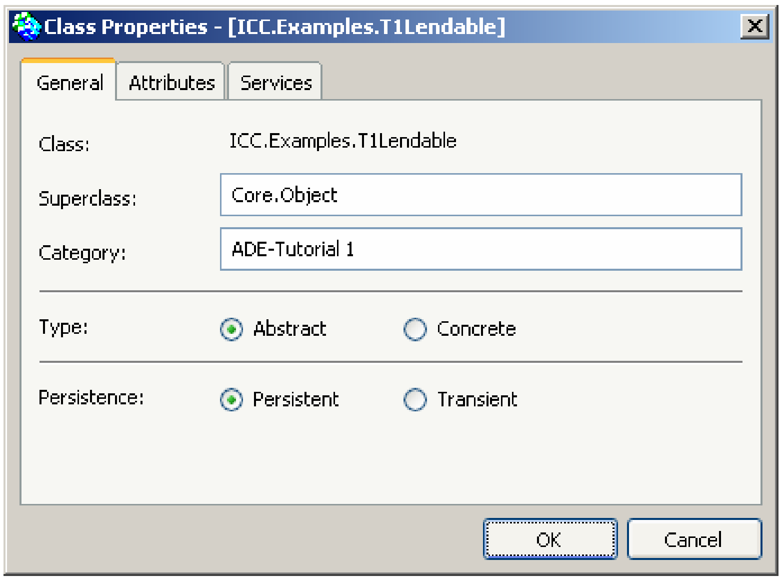


- [✓] Filter accessors
- [✓] Filter overridden methods
- [✓] Filter private methods
- [✓] Filter class methods
- [ ] Filter instance variables
- [ ] Filter class instance variables
- [✓] Filter shared variables

Figure 113. The *Special* page of a *Class Filter Editor*.

Special: This page allows the user to apply predefined semantic filters. By checking the appropriate checkboxes one can filter accessor methods, class methods, overridden methods, or special kind of variables. The checking of an option immediately removes the corresponding entries from the other notebook pages' lists.

Class Properties Dialog



Class: ICC.Examples.T1Lendable

Superclass: Core.Object

Category: ADE-Tutorial 1

Type: [✓ Abstract] Concrete

Persistence: [✓ Persistent] Transient

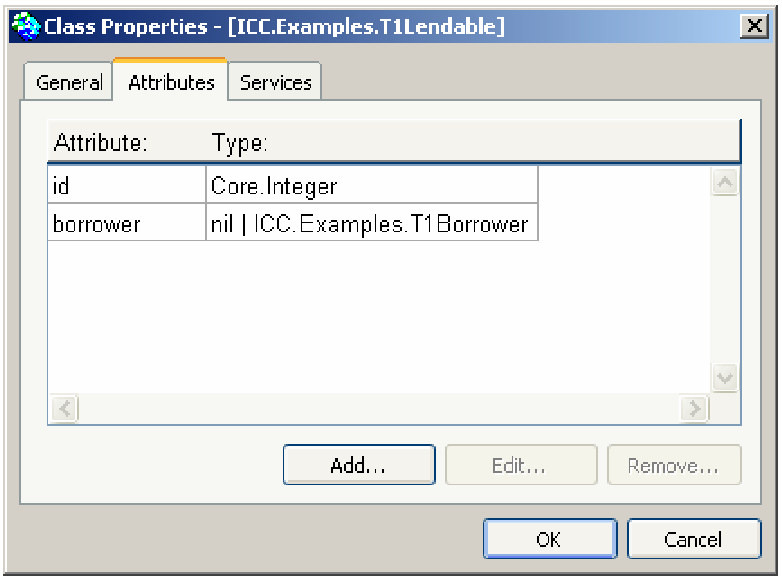
Figure 114. The *Genral* page of *Class Properties Dialog*.

The *Class Properties Dialog* provides an interface for editing a class' definition, attributes and services.

General: Identical to the *New Class Dialog* with read-only class name.

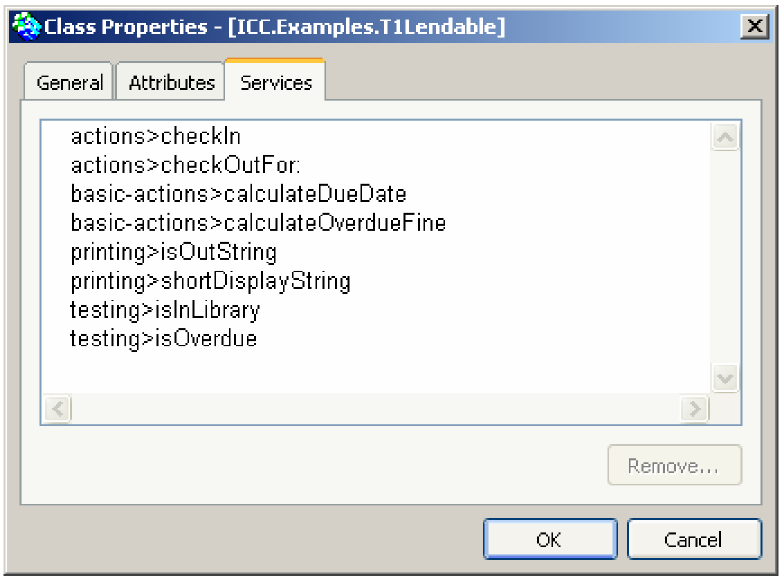
Attributes: Lists the attributes with type information, allows the user to add, edit or remove attributes.

Services: Lists the existing methods, allows the user to remove selected ones.



Attribute:	Type:
id	Core.Integer
borrower	nil | ICC.Examples.T1Borrower

Figure 115. The *Attributes* page of *Class Properties Dialog*.



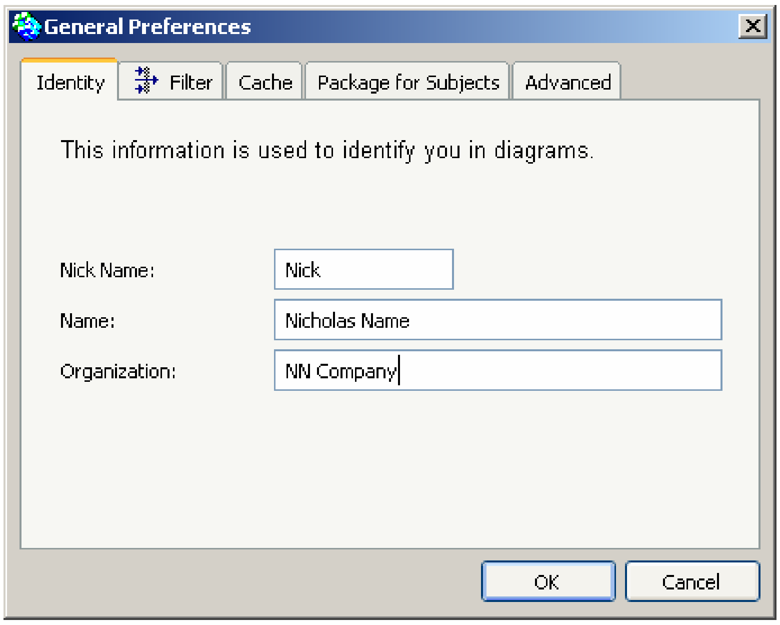
```
actions>checkIn
actions>checkOutFor:
basic-actions>calculateDueDate
basic-actions>calculateOverdueFine
printing>isOutString
printing>shortDisplayString
testing>isInLibrary
testing>isOverdue
```

Figure 116. The *Services* page of *Class Properties Dialog*.

Preferences

General Preferences

The *General Preferences* provide the options which are not accessible through the later described *Painter Preferences*.



This information is used to identify you in diagrams.

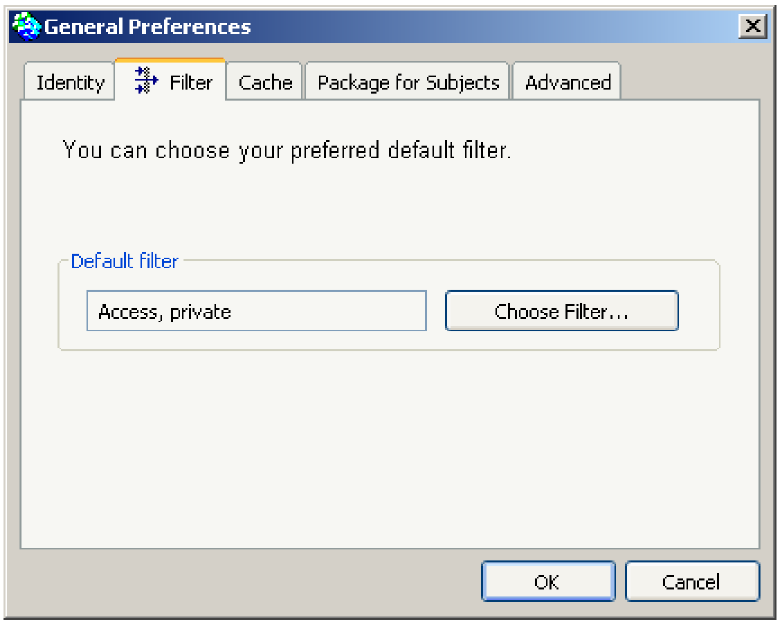
Nick Name: Nick

Name: Nicholas Name

Organization: NN Company

Figure 117. The *General Preferences' Identity* tab.

The **Identity** preferences are used in printed diagrams to describe the diagrams author.



General Preferences

Identity | Filter | Cache | Package for Subjects | Advanced

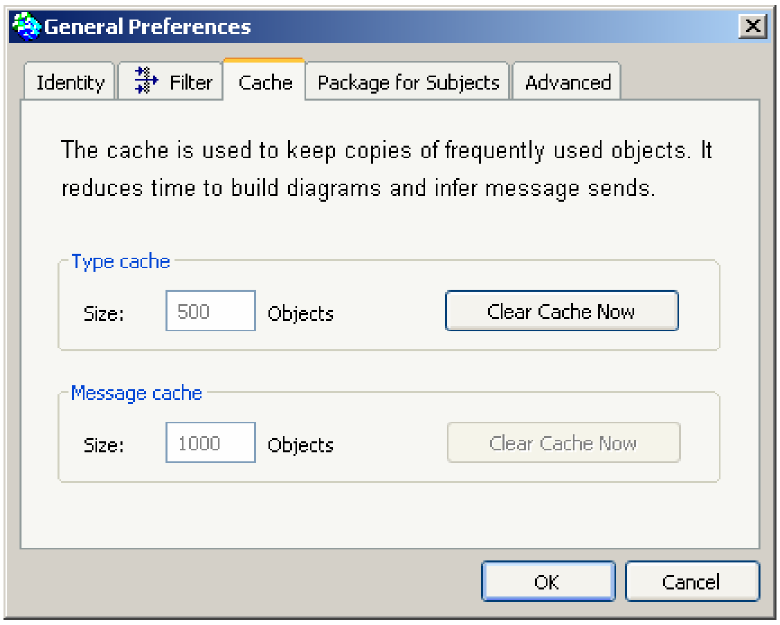
You can choose your preferred default filter.

Default filter

Access, private

Figure 118. The *General Preferences' Filter* tab.

The default filter specified in the **Filter** preferences is taken as initial filter in new diagrams.



General Preferences

Identity | Filter | Cache | Package for Subjects | Advanced

The cache is used to keep copies of frequently used objects. It reduces time to build diagrams and infer message sends.

Type cache

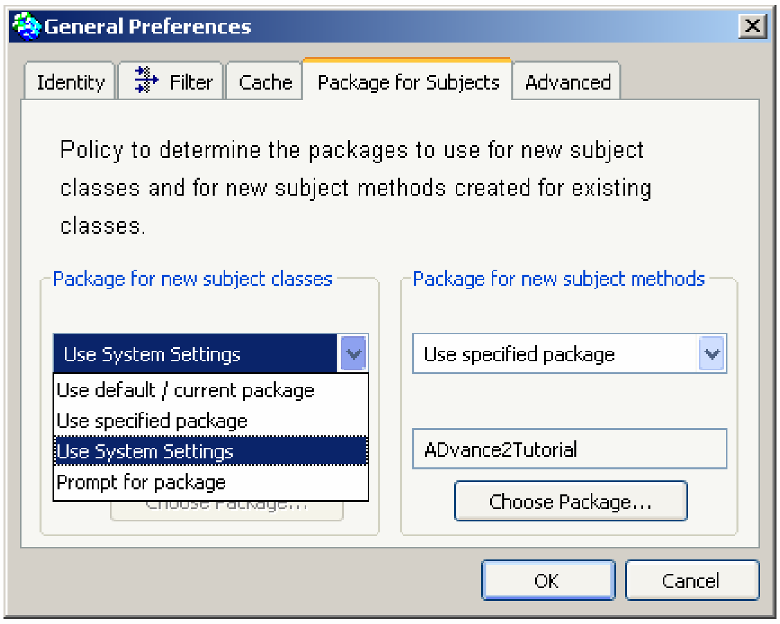
Size: 500 Objects

Message cache

Size: 1000 Objects

Figure 119. The *General Preferences' Cache* tab.

The **Cache** preferences provide no options, but you can clear the **ADvance** internal cache.


General Preferences

Identity | Filter | Cache | Package for Subjects | Advanced

Policy to determine the packages to use for new subject classes and for new subject methods created for existing classes.

Package for new subject classes: Use System Settings
- Use default / current package
- Use specified package
- Use System Settings
- Prompt for package

Package for new subject methods: Use specified package

ADvance2Tutorial

Choose Package...

OK | Cancel

Figure 120. The *General Preferences*' *Package for Subjects* tab.

The *Package for Subjects* preference subsection may be used to work on separate packages for development and documentation purposes. Both panes for subject classes and subject methods provide the same functionality to define a package determination policy.

The class package determined by the left pane's options is used for new subject classes created by the *Subject Wizard*, or by *Refactoring Browser* when saving a new diagram for a package. The method package given by the right pane's options is used for new methods defining a subject and diagramon an existing class, which is not a subject class already – this happens when a new *Refactoring Browser* diagram for a class is saved.

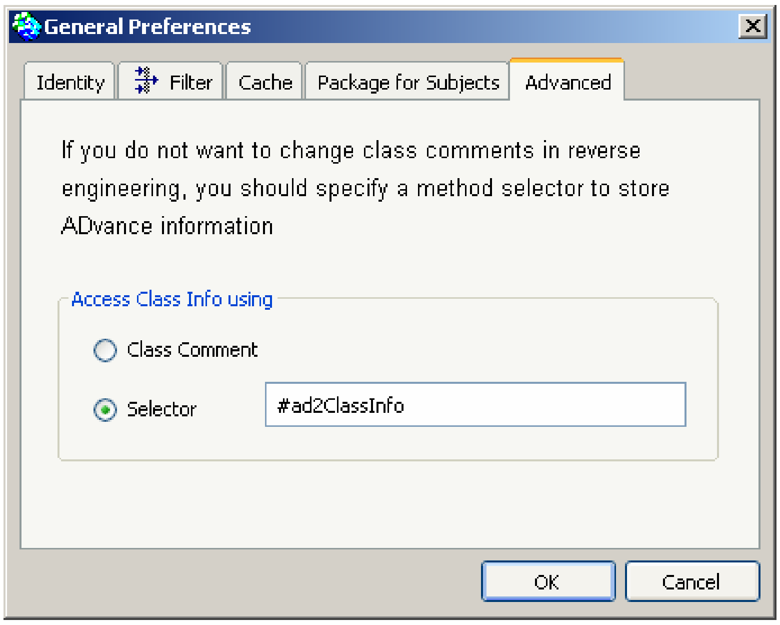
The normal use would be to have the same policies defined for subject classes and subject methods. But with two separated policies it is possible to divide the subject set into more complex subjects like the manually defined package subjects or the *Refactoring Browser* package subjects, and simple subjects for a single class, which often have a more temporary character and therefore may be stored in the *(none)* package.

Use default / current package will use the current package set with the corresponding System Browser's context menu option.

Use specified package takes the package selected with the **Choose Package…** button (and displayed in the read-only input field) instead of the current package. If the formerly selected package is unloaded now, this option works like the **Prompt for package** option.

Use system settings uses the system settings for **Store→Default Package** to determine the package. This is the default option after installing **ADvance**.

Prompt for package lets **ADvance** ask the user each time a new subject class or subject method for a not-yet subject class is created.



If you do not want to change class comments in reverse engineering, you should specify a method selector to store ADvance information

Access Class Info using

( ) Class Comment

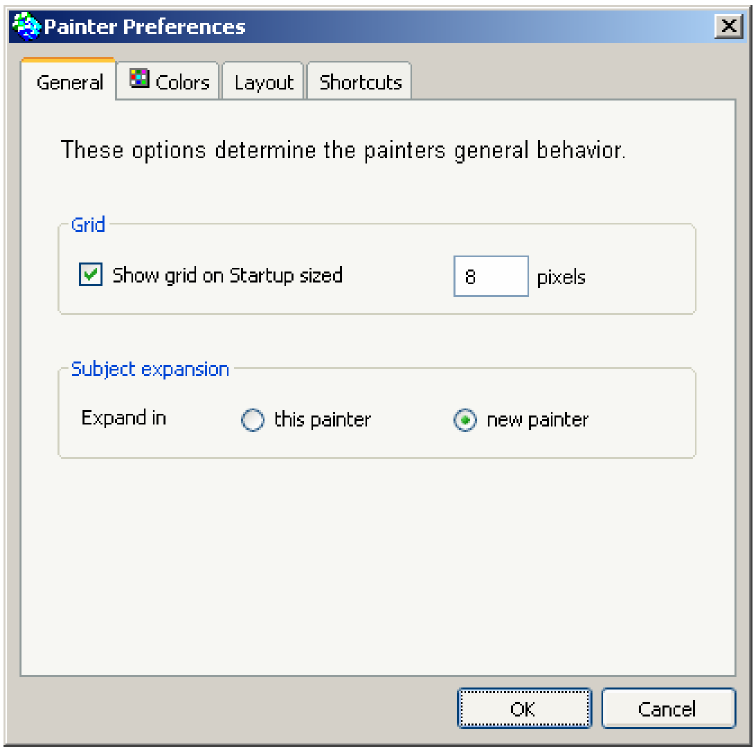
(●) Selector #ad2ClassInfo

Figure 121. The *General Preferences*' *Advanced* tab.

The **Advanced** preferences determine whether only class comments should be used for type information or whether a special class method should be taken if implemented by a class. However, for classes not implementing the method indicated by the selector, the class comment is taken for initial type information. The method will be created for each class with type information edited using **ADvance**.

Painter Preferences

The *Painter Preferences* provide options to change the diagram painter’s appearance. All of the four preference tabs show an explaining text about the effect ot the options.


Painter Preferences

General | Colors | Layout | Shortcuts

These options determine the painters general behavior.

Grid

[✓] Show grid on Startup sized 8 pixels

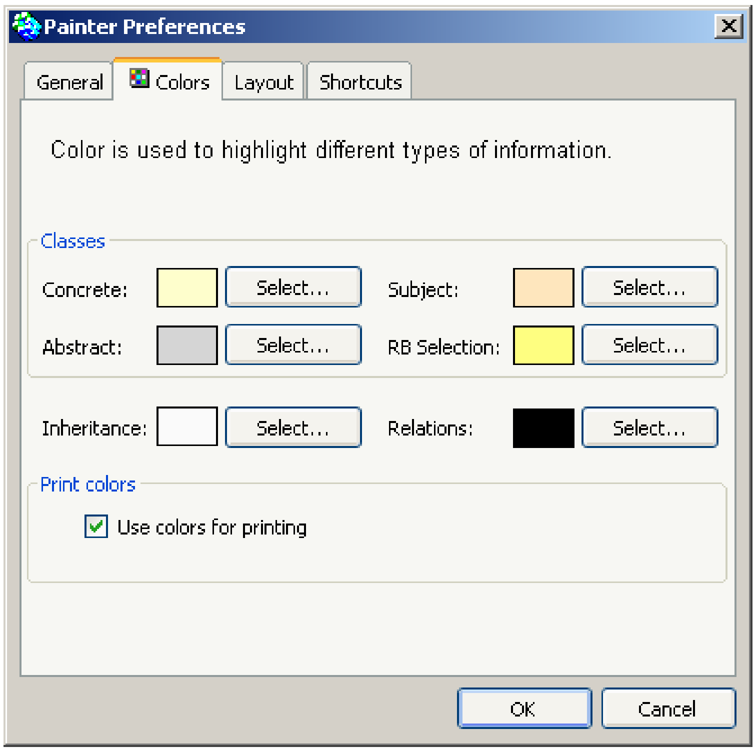
Subject expansion

Expand in ○ this painter [✓ ● new painter]

OK | Cancel

Figure 122. The *Painter Preferences’ General* tab.

The **General** tab allows you to define the shown grip and how to handle recursive subjects, that means the expansion of classes which hold subjects again.



Color is used to highlight different types of information.

Classes

Concrete: Select...
Subject: Select...
Abstract: Select...
RB Selection: Select...

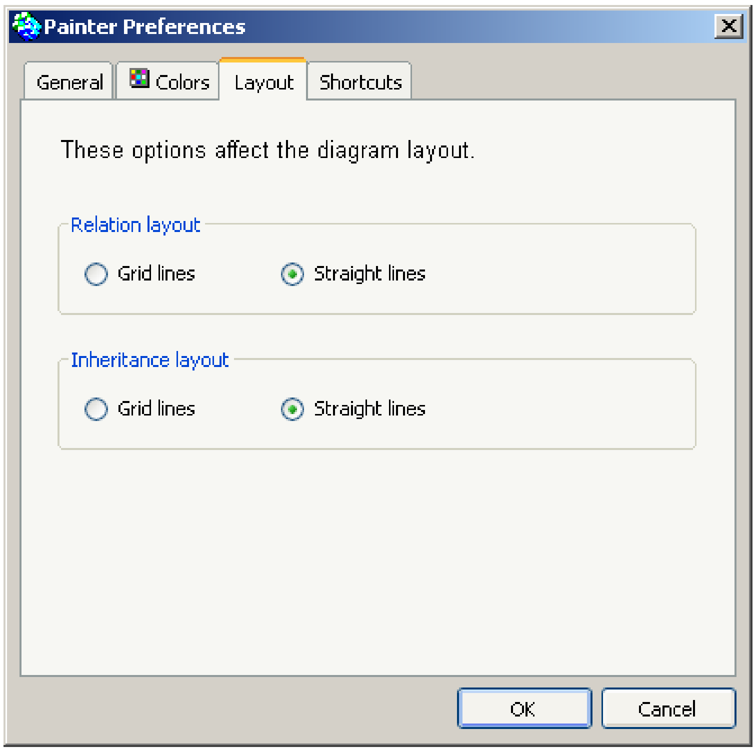
Inheritance: Select...
Relations: Select...

Print colors

[✓] Use colors for printing

Figure 123. The *Painter Preferences' Colors* tab.

The selected colors on the **Colors** tab are used as initial color values for classes of the named kind added to a diagram. The color for each class in a diagram may be individually set.



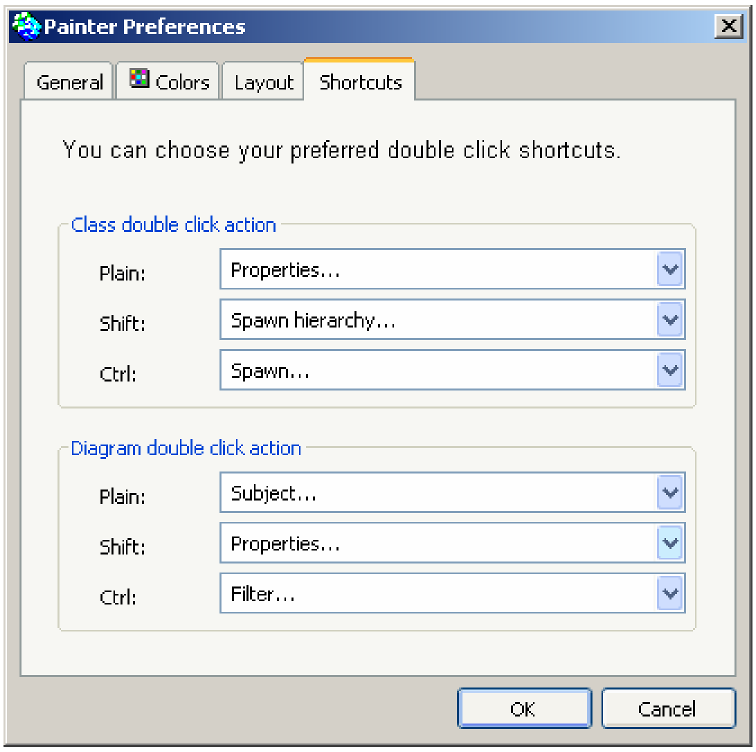
These options affect the diagram layout.

Relation layout
Grid lines [✓ Straight lines]

Inheritance layout
Grid lines [✓ Straight lines]

Figure 124. The *Painter Preferences*' *Layout* tab.

Using the **Layout** tab options, layout for relation and inheritance lines between classes in a diagram may be switched fromgrid line layout to straight line layout and vice versa. Grid line layout works independent from grid settings in the current view.


Painter Preferences

General | Colors | Layout | Shortcuts

You can choose your preferred double click shortcuts.

Class double click action

Plain:	Properties...
Shift:	Spawn hierarchy...
Ctrl:	Spawn...

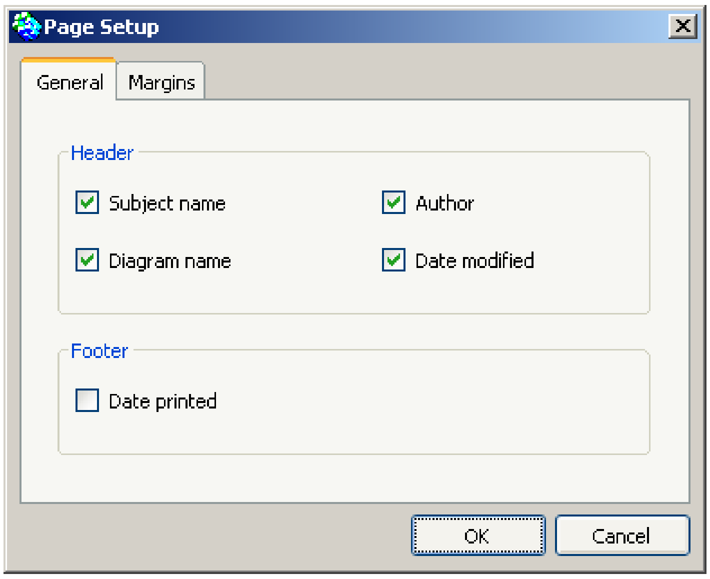
Diagram double click action

Plain:	Subject...
Shift:	Properties...
Ctrl:	Filter...

Figure 125. The *Painter Preferences*' *Shortcuts* tab.

The **Shortcuts** tab provides settings for double click actions on class symbols or on the diagram background.

Page Setup



Header

☑ Subject name ☑ Author

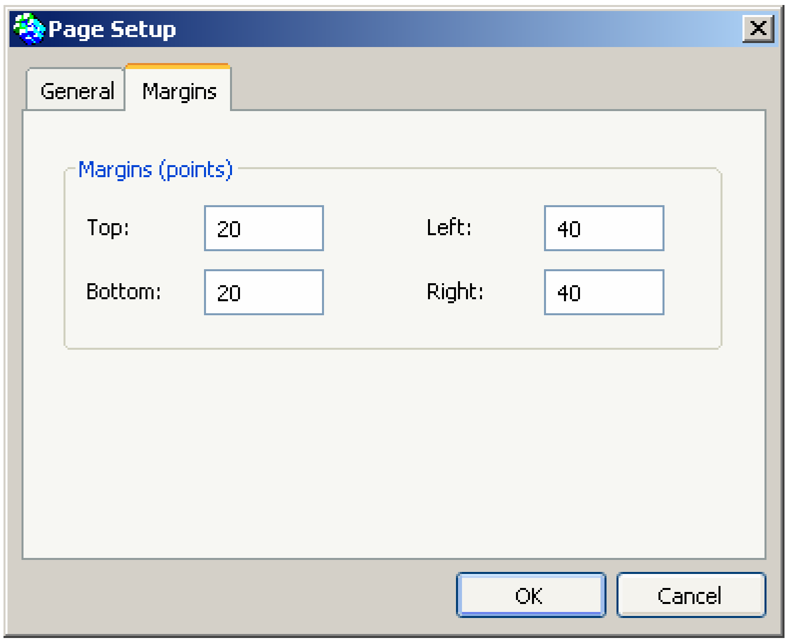
☑ Diagram name ☑ Date modified

Footer

☐ Date printed

Figure 126. The *General* tab of the *Page Setup*.

The settings defined in the page setup are used for print outs, you can modify the header and footer contents as well as the print margins.



Margins (points)

Top:	20	Left:	40
Bottom:	20	Right:	40

Figure 127. The *Margins* tab of the *Page Setup*.

8 Tips and Techniques

This section contains a number of tips which have been recorded from **ADvance** users' comments.

Q: *How large should subjects be?*

A: A heuristic that turned out to be useful is the following: If a subject cannot be arranged graphically in a satisfyingly readable way (on screen and/or printout), the subject should be decreased in size.

Q: *When to use subject, when diagrams?*

A: Subjects are used to capture the context of scrutiny. As an example, take Diagrams for the different views.

Q: *How can I modify the fonts used by ADvance?*

A: See method ICC.ICC1Utils class>>defaultTextstyles.

Q: *How can I modify the double-click actions used in* Diagram Painter*?*

A: See **Painter preferences...->Shortcuts**.

Q: *How can I get a good class comment template?*

A: Sending the message #defineTypes to a class opens an error report, if it is not commented correctly. Copy the error report and paste it in the class comment. Replace type templates like <?type?> by type definitions.

Q: *Given a script, is there an order on messages/actions originating at a method?*

A: Messages which are sent from a method are ordered from top to bottom.

Q: *When do I create a new subject as a copy from another subject?*

A: When analyzing business processes and building an object model at the same time, it may be realized that one starts off with a subject that reflects the entities in focus at that first steps of the process. When moving to further steps in a process, the view of the object model shifts along with it. To represent a new view, the current subject is copied onto a new subject, some classes are removed, some added.

***Q**:* *Is it possible to define use cases in ADvance?*

A: Check the remarks in the *Modeling* section of chapter 5. It is important to notice that the services depicted in behavior diagrams are ordered from top to bottom, thus indicating a sequence.
Conditionals are not explicitly expressed graphically. They can, however, be programmed using standard Smalltalk conditional.

***Q**:* *How do I assign versions to ADvance graphics?*

A: Since **ADvance** stores graphical specifications as Smalltalk code you can use any VisualWorks team tool (e.g. StORE) to version **ADvance** subjects and diagrams.

9 Glossary

Abstract: Class is intended to be superclass that is never instantiated.

Association: A structural relationship between classes.

Association depth: (Levels 1,2,3 or infinite). This defines which class or classes are valid as potential receivers of a message from the initiating service. The specified level represents how directly related the collaborating classes are to self from the initiating service. This is the factor used for determining Visible Scripts.

Level 1: message send is to self, superclass, subclass, or related (type) classes.

Level 2: message send to a collaborator of superclass, subclass or related classes.

Level 3: message send to a collaborator of level2 participants.

Attribute: an attribute represents a variable implementation or an expressed relation between classes.

Collaboration: Manner in which objects work together to achieve system functionality. Specifically if class *A* requests a service of class *B*, then *A* is said to collaborate with *B*. A collaborator is also known as a initiators and participants.

Concrete: Class that is designed to be instanciable.

Diagram: A diagram is a graphical representation of class definitions and the relations and collaborations between classes. The relationships include inheritance and has-a. The definitions are attributes, services and attribute types.

Initiator: A class that initiates a message send.

Message: A message is the visualization of a potential message send.

Participant: A class that receives a message.

Persistent: Class state is maintained in permanent storage.

Relations: Associations between classes that represent has-a, collaboration (behavior), or dependency relationships. Relations between classes fall into two categories: those with a state implementation (i.e. instance variable), and those without, where this expresses a relation between classes achieved through channels such as dependency, global variables, or external interface.

Script: A script is a set of sequential message sends defined by a service.

Script depth: (Levels 1 through 5 and infinite). Script depth indicates how many nested levels of message sends to trace. Level 1 indicates to trace only the paths of the message sends in the originating service, or in other words, trace paths for only the immediate potential participants services. Level 2 indicates to trace the paths of the next level of participant services from within the originating service.

For example: If *methodA:* has the following code:

```
methodA: aBorrower
    aBorrower canBorrow
        ifTrue: [self library checkOutFor: aBorrower].
```

Script depth = 1 will draw a message path from the initiating class to **Borrower>>canBorrow** and from self to method #library.

Script depth = 2 draws a message path from the initiating script to Library>>checkOutFor:.

Service: An unfiltered method.

Subject: A logical placeholder or label for a collection of classes. These classes are referred to as the **contents** of a subject. A class may reside in more than one subject. Each subject has one or more diagrams associated with it where the diagram is a graphical view of 1 or more of the classes contained in the subject.

Transient: Class state is maintained during runtime only, i.e. non-persistent.

Types or typing: Every attribute defined in the **ADvance** environment must be associated with a (Smalltalk class) type in order to enable the tool to construct relations between classes. Expressions such as <Set of: Integer> are valid for defining collection relationships.

10 Index