



Cincom Smalltalk™



Walk Through

VisualWorks 8.1

P46-0132-08

Copyright © 2000-2015 by Cincom Systems, Inc.

All rights reserved.

This product contains copyrighted third-party software.

Part Number: P46-0132-08

Software Release 8.1

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademark acknowledgments:

CINCOM, CINCOM SYSTEMS, and the Cincom logo are registered trademarks of Cincom Systems, Inc. VisualWorks is a trademark of Cincom Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

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

VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation copyright © 2000-2015 by Cincom Systems, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Cincom.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: <http://www.cincom.com>

Contents

Preface	5
Chapter 1 A Walk-Through of VisualWorks	6
In the beginning.....	7
Building a working environment	9
What's Next?	10
Chapter 2 Hello World	11
Writing the Application	12
Save the Image	15
Packaging for Stand-alone Execution	16
Test the Results	19
Packaging as a Single Executable	20
Packaging on OS X	20
Packaging on Windows	20
Renaming the Visual Executable	20
Packaging a Single Executable	21
What's Next?	22
Chapter 3 Building GUI Application	23
What we're going to do...	23
Build the application GUI	24
An extremely brief explanation of objects	29
Finishing the GUI	30
Generating initial Smalltalk code	34
Try it out	37
Smalltalk Code: it's time	38
More Smalltalk Code - Package Browser	41
The Random Class — Domain Models	43
Putting the GUI and the Model Together	50
Using the seed value	54
Move to its own package and name space	57

Comment your code!	59
Packaging it up	60
Where from here?	63

Preface

Since the original publication of the Walk-Through with VisualWorks 5i.2, the product has undergone major changes, all for the better.

With VisualWorks 7 and 8, major improvements have been made to the system, as the result of outstanding work within the VisualWorks engineering team and generous contributions from the user community.

VisualWorks is much more fun to use now. Yet, it is still a complex system, requiring some initial guidance. This, then, is the rationale for a Walk-Through.

Smalltalk is once again gaining notice in the programming world. With this updated edition of the Walk-Through, We hope to encourage many more explorers to discover VisualWorks.

1

A Walk-Through of VisualWorks

This short document is directed at new users of VisualWorks and Smalltalk. When you first load a new tool, especially a programming tool, it is seldom obvious just what to make of it. This can be all the more true when it is a programming language and environment like Smalltalk.

It's easy to get off on the wrong foot, and so start out thinking about VisualWorks wrong-way around. The purpose of this walk-through is to orient you to the landscape, show you the tools, demonstrate a quick, and common, approach to prototype-application development, and point out a variety of features of VisualWorks and Smalltalk along the way.

At various points, additional reading is suggested, but it is not required for completing the walk-through. Many of the features pointed out are talked about in far more detail in the places mentioned. In the walk-through, I'm content to call them to your attention, and point you to further reading.

The primary additional document is the VisualWorks *Application Developer's Guide*, which contains a pretty comprehensive description of the features of the VisualWorks environment. The *Basic Libraries Guide* describes the main Smalltalk class libraries that you will need when creating a program. For GUI development, the *GUI Developer's Guide* is the primary source. Becoming familiar with these documents will yield great dividends as you continue learning VisualWorks.

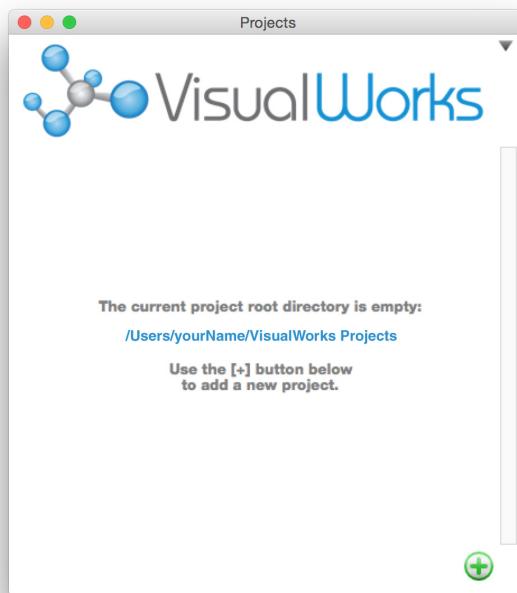
In the beginning...

No, I'll not go back that far. In fact, I'm assuming you already have VisualWorks 8.1 installed, either a commercial or personal-use release. If you have an earlier version of VisualWorks, use an earlier edition of this Walk-Through. The tools have been undergoing extensive changes through the 7.x and 8.x releases, and you should avoid any confusion caused by mismatched tools, if possible. But, the personal-use version of VisualWorks 8.1 is available for free, so you might as well download it and get the latest and greatest available.

Quick Launch

On Windows and OS X, double-click on the VisualWorks icon.

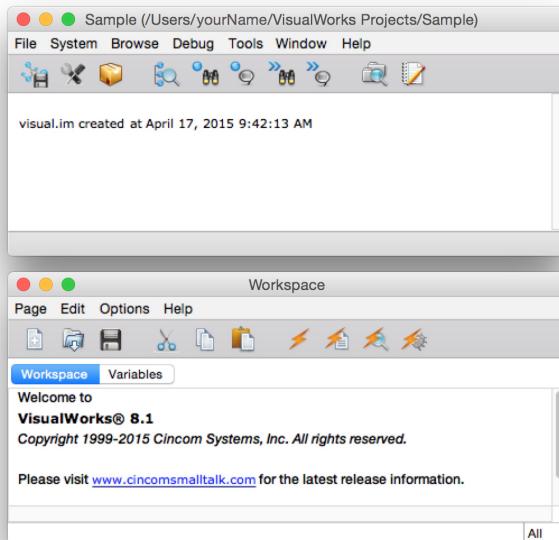
The [Installation Guide](#) describes how to install VisualWorks and how to launch it. We will use the Project Manager, which should have been installed on your Desktop. Click it to launch VisualWorks.



For the remainder of this chapter we will go through setting up a project. This involves creating the working image for the project and loading it up with the features we will need. With variations as to which features we load, this is the standard procedure when starting any new project.

First, we need to create a project. Click the + button (bottom-right) to create a new VisualWorks image. (On platforms that support it, you can click the + 64 button to create a 64-bit image, or + 32 to create a 32-bit image.) In the **New Project Name** dialog that opens, enter a name for this project. This one is just temporary, so name it something like **Sample** and click **OK**.

VisualWorks opens on this newly created image, and you have two windows: the Visual Launcher and a Workspace.



If you installed the personal-use version of VisualWorks, the workspace has several pages, which you may browse at your leisure (just click on one of the tabs). I'll not be talking about them, but there is some interesting and helpful information in them.

The installer sets up two system variables, one that points to the VisualWorks installation directory (VWHOME), and another that points to the projects root directory (VWPPROJECTS). If for some reason it didn't (Unix and Linux systems require some manual setup), some things won't work correctly, like you won't be able to see system source code or load parcels. Refer to the *Installation Guide* if you have problems with source code.

Building a working environment

When you created the new project, the Project Manager also created a new image file, named similarly to the project name. In our case it's called **Sample.im**. This becomes your “working image” for this project.

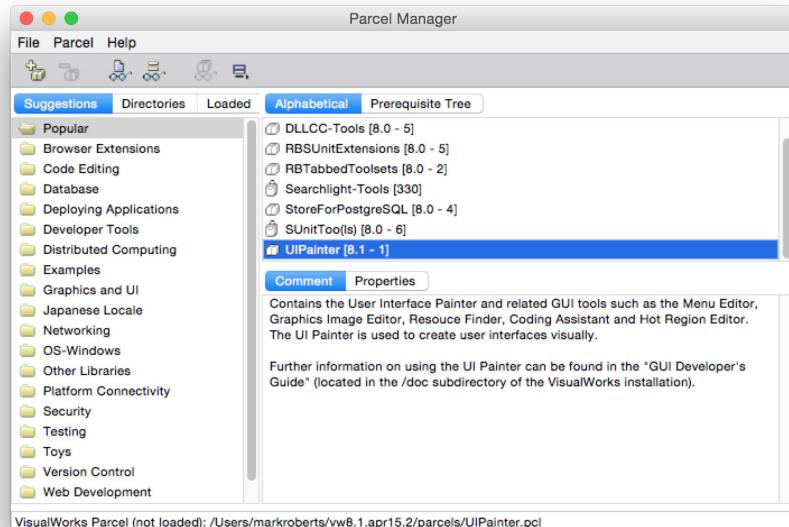
Working Images

You will end up with several “working” images, one for each project. The Project Manager simplifies starting the right image for each project.

The working image will contain any tools or special features you need to use, as well as your application code. For example, you usually need the GUI building tool, called the *UI Painter*. So, we'll load the UI Painter and save the resulting image as our working image file. (This is already loaded in the non-commercial image, but you can still go through the steps.)

The UI Painter is contained in an external code module, called a *parcel*. To use it, we load its parcel into a launched image. You already have **Sample.im** launched, so we will load the parcel into it.

For this, open the Parcel Manager by selecting **Tools > Parcel Manager** in the Visual Launcher.



Now, select **Popular** in the list on the left, then select **UIPainter** in the list on the right. Then select **Parcel > Load**.

Parcels

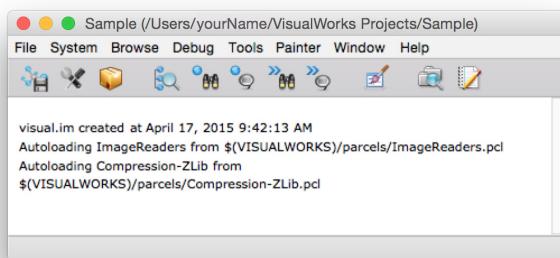
Parcels store VisualWorks code externally to the system for loading when needed.

Parcels are also useful for defining “components,” chunks of code that do specific tasks that may be useful to more than one program.

While a parcel is loading, status messages are displayed in the Visual Launcher window. Among other things, they may say, for example, “Autoloading blah blah blah,” and “Undefined object blah blah blah.” These are normal. There are frequently dependencies between parcels, so that some features of one parcel really need another parcel to work. They will be available if that other parcel is ever loaded. These messages indicate such dependencies.

The parcel then finishes loading. If a message says something like the parcel can’t be loaded, then there’s a problem, but we shouldn’t have any of those for the UI Painter.

Now we have our working image created. The Visual Launcher will have a button that looks like a canvas and a paintbrush. Compare the Launcher on your screen to the image below, to make sure you’re close.



We’re done with the Parcel Manager now, so go ahead and close it.

We need to save this image so we can load it later. It’s already named (**Sample**), so simply select **File > Save Image** in the Visual Launcher, or click the Save button.

You can now exit VisualWorks, by selecting **File > Exit VisualWorks... .** We just saved the image, so click **No**.

What’s Next?

The rest of this walk-through consists of two simple projects: that old chestnut, the Hello World! program, and a simple GUI-oriented application.

2

Hello World

For some reason, everyone seems to want to start with a “Hello, World!” example. It has become a tradition. So, to be traditional, we’ll start with a VisualWorks “Hello, World!” example.

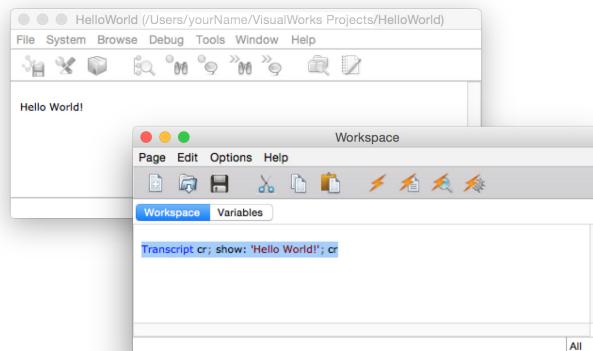
Launch the Project Manager, and add a new project called **HelloWorld**. VisualWorks then launches with the new project open. As before, you should see two windows: the Launcher and a Workspace.

For some purposes it might be sufficient to enter the following expression into a Workspace window:

`Transcript cr; show:'Hello World!'; cr`

then select (highlight) and evaluate that with **Do it** (in the menus, pick **Edit > Do it**).

This executes the one-line “program” shown above and displays the result in the Transcript window.



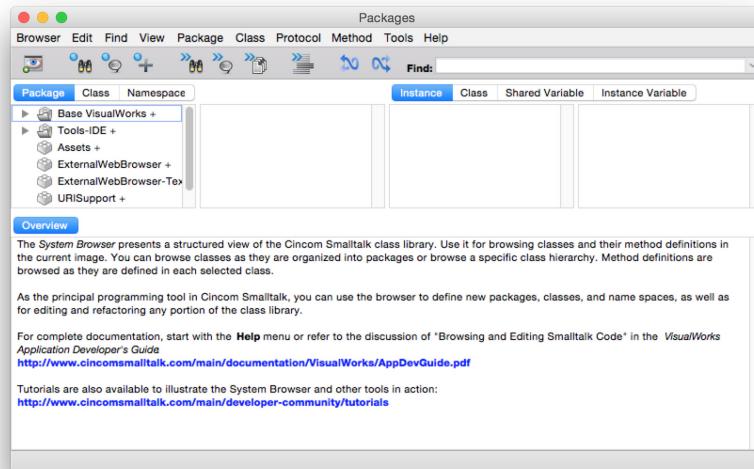
While this shows the interactive nature of a Smalltalk IDE (**Do it** is available in all of the basic tools), most new users want to see this happen in a stand-alone application, rather than within the IDE. This requires a little more work, but not all that much more.

The “Hello World!” application will consist of one class and one method. We’ll then have to generate the image file to run stand-alone. That’s about it, except for some optional packaging.

Writing the Application

We will write the entire application using a code browser.

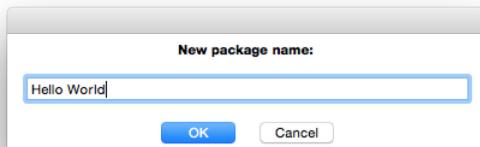
In the VisualWorks Launcher, select **Browse > System**. This opens a System Browser, which is where we’ll start.



One thing to be aware of while is that in VisualWorks, as for most Smalltalk environments, you have the code for the whole current system, including the language, IDE (development tools), and any loaded libraries, right there in the browser available for browsing. This is useful, eventually, but can also be distracting, or confusing, or overwhelming. You need to learn how to focus on the parts of the system that you need, and ignore the rest. This comes with experience, but there are also techniques to help, such as how you organize your code, and the tools you use to view the code.

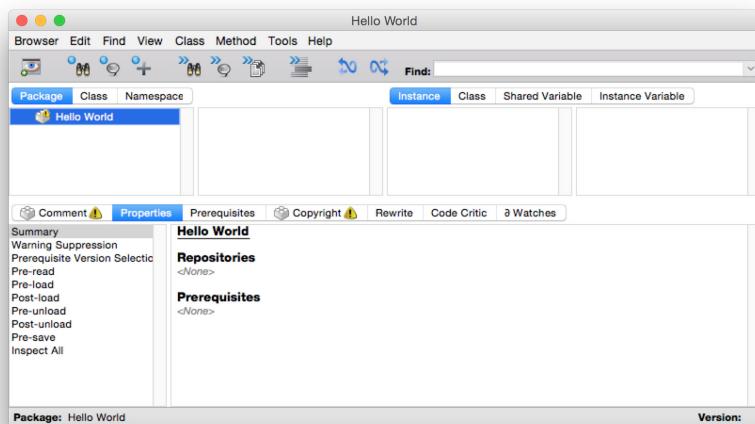
For now, look at the top left pane, which shows a tree view of components (packages and bundles) that contain the entire system. We don't want our application to be confused with all this stuff, so we will keep it separate by creating our own package.

To create a package, pick the **Package > New Package...** menu item. A dialog asks you for the package name. Let's call it, very imaginatively, "Hello World." Enter **Hello World** and click **OK**.

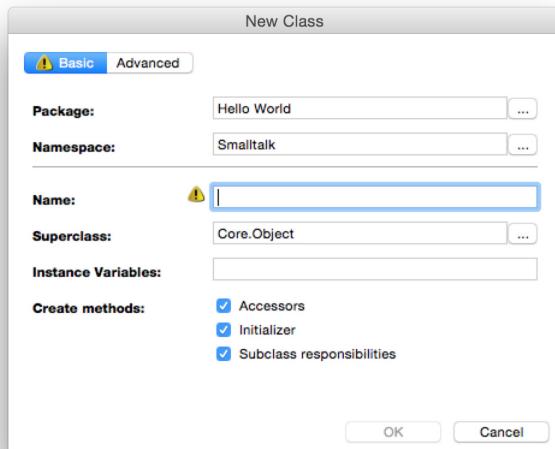


The new **Hello World** package is now automatically selected, and you can find it in the top-left list pane of the browser. This is where we'll be working. You can ignore everything else.

To make it a bit easier to focus just on our application, we can open a new browser focused just on this package. With the **Hello World** package selected, pick the **Package > Spawn Selected** menu item. The new browser is empty, but is labeled as focusing on the **Hello World** package.



Now we need to create a class. In the **Hello World** browser, pick the **Class > New Class...** menu item, which opens the class creation dialog.



The yellow marker points out the required information, which in this case is the class name. Type **HelloWriter** in the **Name** field, to specify the class name (unlike the package name, spaces are not allowed). The package is automatically specified as the package we're browsing. We will use the items created by the three checkboxes, so leave them checked. (The rest of the items are all described in the [Application Developer's Guide](#).) Click **OK**. The class is then created, added to the package, and shown in the browser (in both the **System Browser** and the **Hello World** browser, if both are opened).

As a side note, you have just modified the system, by adding a class. Actually, you did when you created the package, as well, though that isn't so interesting a change. These changes are made immediately in the image. Unlike many other environments, there is no need to save a separate source file and compile it.

Unlike many environments, we do not need to import things like Dialog into our application; they're already in the system

Now we need to create a method to do the work. The code is quite simple, and will simply open a dialog showing “Hello, World!” and then close when we click a button. We will use the class Dialog and the method, choose:labels:values:default:, to do this. Both are already defined in the system, in one of those system packages we are ignoring.

While it is not particularly good object-oriented code, it is quick and easy to put this in the initialize method that was created for us along with the class (remember those checkboxes?). In the **Hello World** browser, select the **HelloWriter** class, then the **initialize-release** method category (also referred to as a “protocol”), and then the **initialize** method. Replace the space holder code there with:

initialize

Dialog

```
choose: 'Hello, World!'
labels: (Array with: 'OK')
values: #(nil)
default: nil
```

and save your changes (**Edit > Accept** or **Cmd-S**).

That's it. You can test this code in a workspace by typing in:

```
HelloWriter new
```

selecting, and executing this expression with **Do it** (**Edit > Do it** or **Cmd-D**).

The new message is a class message that creates a new instance of the class. In the browser, click on the **Class** tab, select the **instance creation** method category, and then the **new** method. This method was generated when you created the class, at the same time as the initialize method. The one executable line in the new method invokes the instance-creation method of HelloWriter's superclass, then sends initialize to the new instance. The initialize message causes the dialog to open, as we wrote it to do.

Save the Image

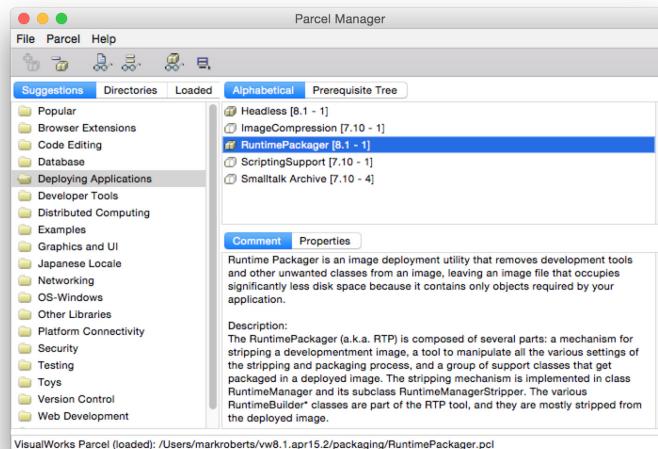
You know how to do this. Simply click the Save button in the Visual Launcher (or pick **File > Save Image**).

Don't exit, because we're going to continue on. However, if you do exit, you can reopen where we've left off simply by selecting this project in the Project Manager.

Packaging for Stand-alone Execution

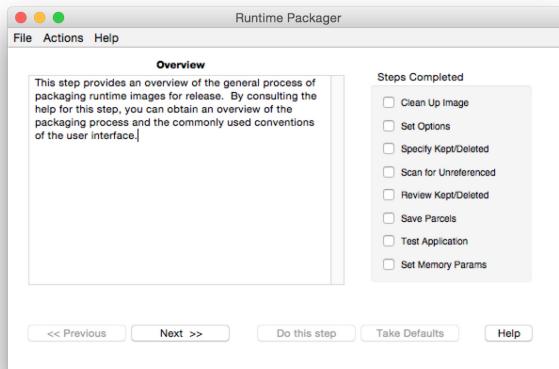
So far we have the Hello World application, but we can only run it within the VisualWorks development environment. We'd rather run it as a stand-alone application, without starting the IDE first. To do this we prepare the application using the *Runtime Packager*.

In the VisualWorks Launcher, open the Parcel Manager (**System > Parcel Manager**). As mentioned earlier, parcels are external code files. We don't usually need all of the features provided with VisualWorks, so we leave most of them unloaded until we need them. The Parcel Manager is the easiest way to load these components when we need them, as we do now with the Runtime Packager.



In the Parcel Manager, click on the **Deploying Applications** folder in the left list pane. Then, in the top right list pane, double-click on **RuntimePackager** (or select the parcel and pick **Parcel > Load**). Answer **Yes** when asked if you want to load this package. Then, close the Parcel Manager.

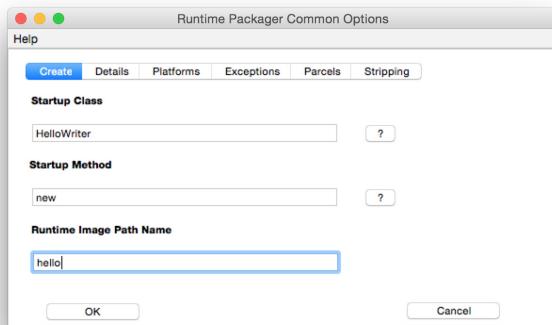
Now, in the Launcher, pick **Tools > Runtime Packager** to start it and open a new window:



There are a lot of options available in the Runtime Packager, and we're going to ignore nearly all of them. For the most part, the default values will work well for this example.

Click **Next >>** twice, until you get to the **Set common options** window. Then click **Do this step**.

A new dialog for **Common Options** opens, which has several pages:



On the **Create** page, we need to enter the information needed to start our application.

As we've already seen, running:

```
HelloWriter new
```

runs the application. HelloWriter is the class and new is the method, so that's what we enter on the **Create** page, plus a name for the final image:

- Startup Class: **HelloWriter**
- Startup Method: **new**
- Runtime Image Path Name: **hello**

Runtime Image Path Name specifies the name for the runtime image, as opposed to the development image.

While we're here, take a look at the **Details** page. There are a number of options here, but notice the **Action on last window close** section. The default is **Shutdown image**, which is what we want, so we'll leave it. The desired behavior of our application will be to launch, display "Hello World!" and then exit.

We might as well look at the **Platforms** page, too. VisualWorks images are fully portable between operating system platforms, meaning the same images can run on any of the supported platforms. But, if you remove some of the support for other platforms, it won't work as expected. On this page you can select the "look policy" and operating system support to include in your image, so it will work and look right on another platform. For example, if you are developing on Windows but know you will deploy on Linux as well, you want to check the **Motif** UI look and **Unix** operating system options on this page. By default, all of the options are selected, which is a good choice for now.

Click **OK** to save the options we've set and return to the main Runtime Packager window.

Click **Next >>** five times (but feel free to look at any of the pages by clicking **Do this step**) until you get to the **Test application** window. Click **Do this step** to open the test window, and click **Begin Test**. After some brief cogitation, our "Hello World!" dialog will be displayed again.



Click **OK** to close it, then click **End Test** and **OK** to close the test screen. A dialog will ask: **Do you want to accept the most recent dynamic reference before closing this Window?** Answer: **Yes**. Hopefully it all worked.

Click **Next >>** twice, to get to the **Strip and save image** screen. “Stripping” refers to what will be done to the image (not how you dress for the occasion). That is, a large number of classes used in the IDE but not needed in a runtime application (e.g., all the tools) will be removed from the image. Once that is done, the image is saved to the name we specified in the **Set common options** step (we chose: **hello**).

When you click **Do this step**, one or two dialogs may pop up saying that there are open windows referencing classes that will be removed. Those include any browsers we’ve left open. Click **Proceed** or **Yes** to close all of those windows. We are now almost to the final step.

However, instances of the application and any workspace windows that we have left open will not be closed in this step. If there are any, click **No**, close the windows, then do the step again, answering **Yes**.

Read the messages in whatever dialogs are displayed, and respond appropriately. Mostly they are notices about what is about to be done, and asking for confirmation to proceed. Finally, changes are made to the development image, and the runtime image it is written to the file name specified (**hello.im**), and VisualWorks is shut down.

Test the Results

You should now have a working application that you can launch independently of the IDE. To check it out, find the image file, **hello.im**. It will be in the project directory created by the Project Manager (e.g., in the **VisualWorks Projects** folder in your home directory on OS X, or the **MyDocuments** hierarchy on MS-Windows).

Now, launch the image. On both OS X and MS-Windows, you can generally just double-click on the **hello.im** image file. On other platforms, you may need to execute a command line. For instance, on Linux this might look like the following (though it varies with the command shell and file locations):

```
cd ~/vw8.1/image  
../bin/linux86/visual hello.im
```

Packaging as a Single Executable

OS X and MS-Windows developers generally want their applications to be deployed as a single file, an executable. This is usually the case even for very small applications, like Hello World. For Linux and Unix users, however, this is seldom an issue.

The approaches for packaging on OS X and MS-Windows are different. I'll explain the steps for each platform separately.

Packaging on OS X

Building an executable on OS X is straight-forward. Although the OS X Finder presents each OS X application as a single file, from a Unix point of view it is in fact, a directory with a certain structure. Apple refers to this structure as a "package".

Thus, you can create an executable VisualWorks application simply by placing the stripped image file within this directory structure.

Using the OS X Finder, make a copy of the VisualWorks object engine (the application named **visual**, located in the **bin** directory of your VisualWorks installation). For this example, rename it as **hello**.

Similarly, rename the image file **hello.im** to be **resource.im**. This is the name expected by the VisualWorks object engine.

Next, open the package of your **hello** application, by selecting **Show Package Contents** from the Ctrl-click menu, and navigate to the **Resources** folder, inside the **Contents** folder.

Drag the image file **resource.im** that you saved in the **VisualWorks Projects** folder into the **Resources** folder that is inside the **hello** application. Close the **hello** package, until you see the application icon.

You can now run Hello World by simply double-clicking on **hello**.

You can add your own icons for the application, and customize it in various ways. For complete instructions, refer to **DeliverApps.rtf** in the **packaging/macx** subdirectory.

Packaging on Windows

Renaming the Visual Executable

End users on Windows like to launch the executable for their application, rather than hand the application data to another, differently named, executable. That is, they'd rather run a application called **hello.exe** than a more complex thing like **visual.exe hello.im**. We can do this.

Using the tools provided by Windows, make a copy of the executable file (**visual.exe**). Then, rename the copy to **hello.exe**, so that the name is the same as for the image file. Copy or move both the executable and the image files in the same directory. It doesn't matter which directory. Now simply run the executable.

By default, if no image file is specified, the virtual machine (the executable) looks in its directory for an image file with the same name, minus the filename extension. If there is one, it loads that.

You do still need to provide two files, but this is then similar to delivering an executable with a DLL on Windows.

Note that this only works on Windows. For other platforms, there are other execution strategies, as described in the *Application Developer's Guide*.

Packaging a Single Executable

Windows allows an executable file to include certain resource files. VisualWorks includes a third-party shareware application, ResHacker, that helps you bundle an image, and possibly other resources, with the virtual machine for deployment as a single, stand-alone, executable.

First, you have to extract ResHacker. It is in a ZIP file in the **packaging\win** directory of your VisualWorks installation. Find it, and extract it to the same directory.

It is easiest to build this file if all of our files are in the same place. So, copy the image file (the finished one, **hello.im**) and one of the virtual machine files (usually **visual.exe**) into **packaging\win** as well.

Now, open a Windows command session, change to this directory, and execute the command:

```
reshacker -addoverwrite visual.exe, hello.exe, hello.im, 332,  
332,
```

Note that the syntax is really very particular. Those commas, for instance, are necessary, even the last one.

Once successfully executed, ResHacker will create the **hello.exe** file, which you can run by itself.

You can add other resources, such as splash screen bitmaps, sound files, application icons, and command line options. Refer to the **WindowsPackaging.txt** file for a full description.

What's Next?

This has been a quick look at some basic features of VisualWorks. The next chapter introduces the GUI building tools, and a simple application design approach, as well as additional pointers for exploring VisualWorks.

3

Building GUI Application

What we're going to do...

There are a variety of ways to start building an application in VisualWorks. Frequently, the data model, or the kind of information and processes that the application controls, are very well defined, or already written, and a Graphical User Interface, or GUI, is simply put on top of it. In fact, that's pretty much what we'll do here. A random number generator already exists in VisualWorks, and we'll just put a GUI on it.

An alternative, though still common, approach for creating a prototype of an application begins with a GUI, for which there may not be a particularly well-defined data model yet. The GUI helps identify what kind of information the underlying program will need to provide, and guides further development.

The two approaches are not all that distinct. In fact, both begin with the GUI, as we will here.

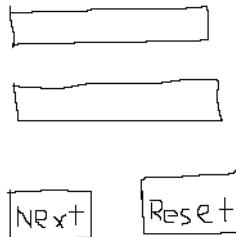
Program Structure

An application has a “model” and a “user interface.” The model does the thinking. The user interface, which is usually graphical (hence, GUI), takes user input and displays results.

A GUI has two purposes: to accept user input, such as data and commands, and to display the results of the program’s processing. The processing all goes on “behind the scenes.” As a VisualWorks programmer you will be concerned with both. You have to start with one end or the other. We’ll start with the GUI.

Here’s the idea: we will create a GUI that, every time the user clicks a button labeled **Next**, will display the next number in a random sequence, as produced by a random number generator. The user will also be

allowed to restart the sequence, or to enter a “seed” value, which just tells the sequence where to start. So, it will have two buttons and two display fields, something like this:



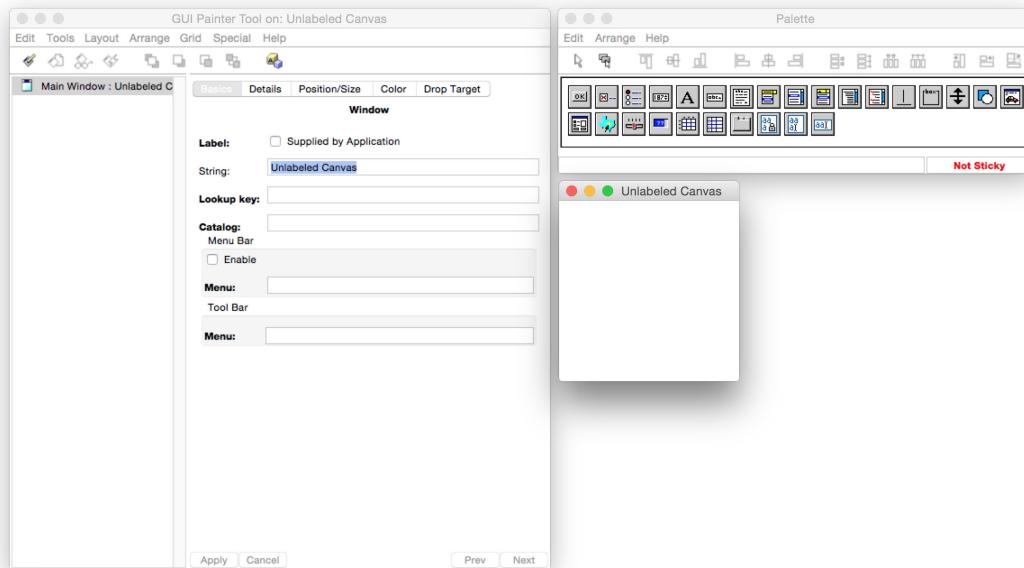
If you get stuck, especially later on, you can load the **WalkThrough** parcel and examine my code.

Build the application GUI

As I said, we’re going to start by building our application’s GUI. To do this, we need to open the UI Painter tool.

First, create a new project. Launch the Project Manager as before, and add a new 32- or 64-bit project called **WalkThru**. When VisualWorks has fully launched, load the UI Painter into the environment (i.e., load its parcel).

Click on the canvas-and-paintbrush looking button. (You could also select **Painter > New Canvas** in the Visual Launcher, but why work that hard?) Three new windows are opened:



You may need to rearrange these windows for convenience.

The window with all the little buttons (top right) is called the **Palette**. This has all the pieces that you can (easily) put into your GUI.

The big window on the left is called the **GUI Painter Tool**. It is really command central for configuring your application window. We'll be using this a lot.

The really important one is what's currently called **Unlabeled Canvas**. This is where you will build your GUI. You put pieces from the Palette onto it, arrange them, and so design the way your application will look to the user. Nothing to it, right? Right!

Look at the Palette and click on the various buttons. The two buttons at the top left, the arrow and stack of cards, are for placing single or multiple copies of an element, respectively. There are also a bunch of "arrangement" buttons, all greyed out at the moment. The remaining buttons are the "widget" buttons.

UI Painter Tools

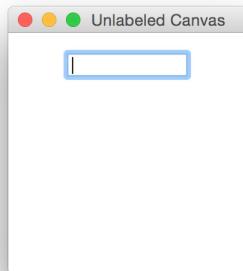
The Canvas, where you place and arrange “widgets.”

The Palette, which holds widgets for selection and placement on the canvas.

The GUI Painter Tool, which provides easy access to operations on the canvas.

When you click on one of the widget buttons, a brief description of the selected widget is shown at the bottom of the Palette window. When you drag the cursor over the widgets, a little fly-by help description is shown. Find the widget button that says **Input Field**.

When you find it, click it, then move your mouse over to the Unlabeled Canvas. To place it on the canvas, move it to a good location, like centered and pretty near the top, and click. The Input Field widget will “drop” there and stay.



If you had the arrow button selected, at the top of the Palette, then you’re done. If you’d selected the other button, you could drop more Input Fields by continuing to click on the canvas, dropping one each time.

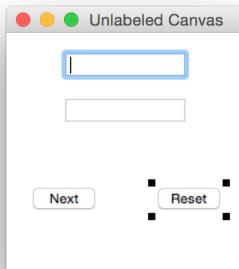
Notice the little dark spots, or squares, at the corners of the Input Field widget. These are usually referred to as, appropriately, “corners,” or “handles.” Use them to change the size of the widget. Click and hold the left mouse button (like a double-click, only don’t let go on the second click), and drag the mouse. You’ll push or pull the corner, changing its size.

To move the widget, click and hold somewhere on the widget other than a handle, and drag it to position. Then let go (drop).

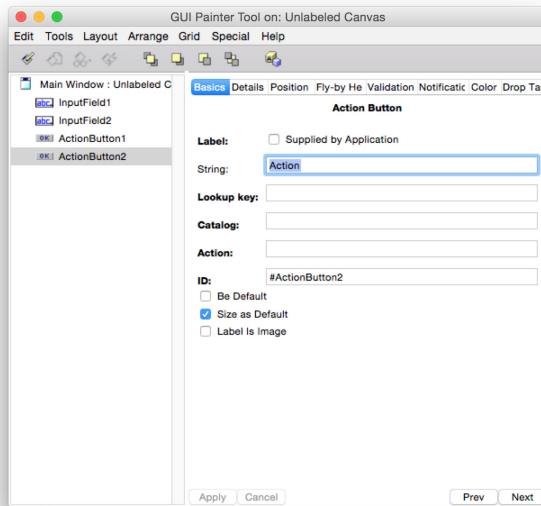
Add one more Input Field widget under the first one somewhere, around the middle of the canvas.

Now select the **Action Button** widget, and place two of them near the bottom of the canvas, side-by-side. Click on the button on the left, to select it (so its handles show).

The canvas should now look like this:



Now, look at the GUI Painter Tool.

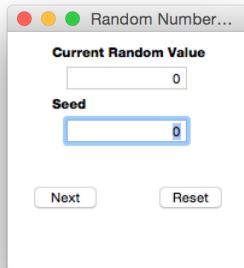


The list on the left shows all the widgets you've put on the canvas so far. The one that's highlighted is the widget that is currently selected on the canvas. You can change the selection either by clicking on another widget in the canvas or in this list. Try it, but then get back to the action button on the left.

The big area to the right of this tool holds the **Properties** pages. These pages set a bunch of, well, properties, for the selected widget, and for the canvas if no widget is selected. Each widget has a default name, or ID, which identifies it in the widget list. Currently the **Basics** page is showing.

For action buttons, the most important items are the **String** and the **Action** fields (or properties). If you accidentally unselect the button, the **Properties** pages will change. Just reselect the left button on the canvas to get back to its properties.

The **String** field sets the button's label. Delete the text there now, and type in **Next**. Then click the **Apply** button at the bottom of the **Properties** page. Notice that the button's label changes. Select the other button, and change its label to **Reset** the same way.



The other important property for buttons is their action, in the **Action** field. We've not talked about Smalltalk or objects yet, but we have to now.

An extremely brief explanation of objects

Object Orientation

Smalltalk programs consist of individual things, or objects, in communication. Objects communicate by sending messages and receiving a response. Responses to messages are defined in methods. A general description of an object, and the methods that describe its response to message, are contained in its class definition.

Smalltalk is what's called an "object-oriented programming" language. What that means is that programs are written by describing kinds of things (objects) and what they do. What objects do is respond to messages that are sent to them by other objects. So, really they talk to each other by sending messages.

A message has a name, called a *message selector*. It may also have one or more additional pieces of information, called *arguments* or *parameters*, which give more details about the request. For example, if you have a Ball object, called myBall, you might have a message called color that asks it for its color like this:

myBall color

Similarly, you might have a message called color: that sets its color. The colon indicates that the message must include an argument, e.g.:

myBall color: blue

When an object receives a *message*, it responds in some way. That response is defined by a *method*, which is a piece of Smalltalk code. The method defines the response to the message sent using the message selector. Note that a message *always* responds by returning an object; just what object is returned is determined by the method.

Message Types

Unary messages are messages without arguments,

myBall color

Keyword messages take an argument for each keyword, marked by a colon:

myBall color: red

Binary messages use a special character (such as '+') and one argument:

2 + 2

For example, an Action Button is an object. So, it communicates with other objects by sending messages and by responding to messages sent to it. The messages it responds to are fairly deep in the system, ultimately coming from the keyboard and mouse, in general, so you don't need to worry about them right now. We're really only interested in the message it sends when the user clicks on it.

We already know what we want to happen when we click on either of our buttons. The button labeled **Next** should get the next random number, and the button labeled **Reset** should start over, or reset the sequence. So, it makes sense for the first button to send a message like "next random" and the second button to send one called "reset sequence."

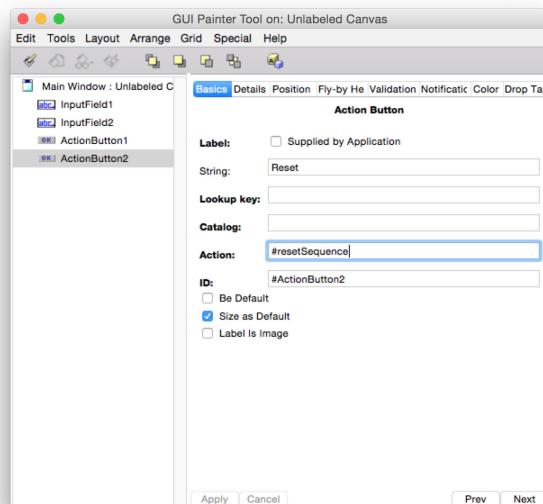
We will need to write methods to do this, but initially we only need the message identifier. Smalltalk convention is that keywords like message identifiers are single words (no spaces), made up of intelligible (human understandable) words, with word separations indicated by an uppercase letter. The initial letter of a message identifier is always lower-case.

With this in mind, let's call our message for the **Next** button `nextRandom`, and for the **Reset** button `resetSequence`.

Finishing the GUI

On the **Basics** properties page, with the **Next** button selected, type `nextRandom` in the **Action** field, and click **Apply**. When you accept the change and change focus, the message name in the **Action** field is redisplayed as `#nextRandom`. The pound sign (#) is a piece of Smalltalk syntax that indicates that the word will be passed as a symbol. Symbols are discussed in the *Application Developer's Guide*.

Now select the **Reset** button, type `resetSequence` in its **Action** field, and click **Apply**.



Aspect Property

An Aspect stores a feature of an object, usually in an instance variable.

The aspect is usually stored as a ValueHolder, for easy use by the VisualWorks UI Framework.

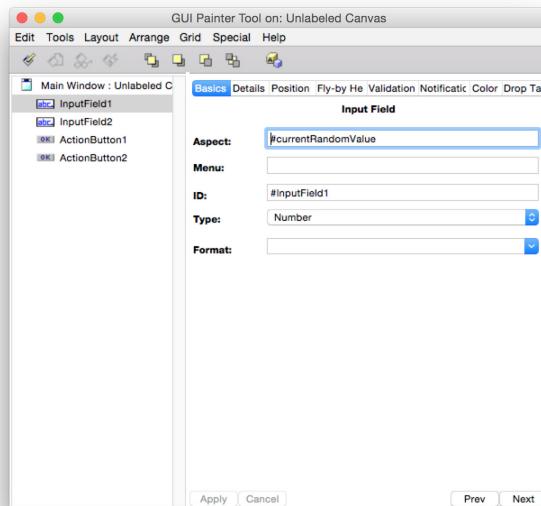
We're done with the action buttons for now. Now we need to think about the input fields. So click on the top one and look at its properties page. Notice that the properties are different. There is no label string to be entered, and no action to specify. Instead there are fields called **Aspect** and a **Type**. Those are the most important fields for an Input Field widget.

An aspect is like a facet of a cut diamond; one part of its defining properties or characteristics. Applications “model” something; a human-machine interaction, perhaps, a manufacturing process or store transaction, or something more abstract.

For this example, our application needs to model a random number sequence (yes, mathematicians, it's only pseudo-random, because it's a predictable and repeatable sequence). There are two essential aspects of

this model that we want to use: the current random value and the seed value. The current random value is obvious. The seed value is something used by computational (pseudo-) random number generators to provide a predictable starting point and repeatable sequence.

Our top input field is really only going to be a display field, to show the current random value. On its **Basics** properties page, we need to enter a name for that aspect. Let's call it **currentRandomValue**, and enter that in the **Aspect** field of its properties. In its **Type** field, select **Number** from the drop-down list. We'll not bother with the format for now. Click **Apply**.

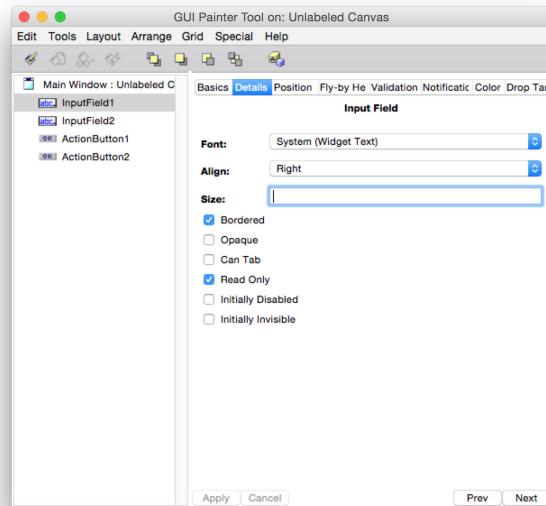


Notice that, since we made this a number, its initial value is now shown in the canvas as **0**. It's also left-justified, which looks funny. Click on the **Details** tab to change properties pages to see another page of properties.

In the **Align** field, select **Right** from the drop-down list. Also, since we only want to read the current value, not set it, click on the **Read Only** check box, so a check mark shows in it. Since we won't enter anything here, we don't need to tab to it, so uncheck **Can Tab**. Now, click **Apply**.

You can change the background color on the **Color** property page, which you may explore on your own. Refer to the *GUI Developer's Guide* if you need help.

We do want to enter something into the other input field, namely a seed value. Click on it to show its properties, and click **Basics** to show that page. Let's call this field's aspect "seed," so enter **seed** into the **Aspect** field. Seeds need to be numbers, so make the **Type** field **Number**. Click **Apply**.



Tab Control

Tabbing is an important feature in a good GUI. It allows the user to move between widgets without the mouse.

Only widgets that take user input should allow tabbing, such as input fields, checkboxes, and buttons.

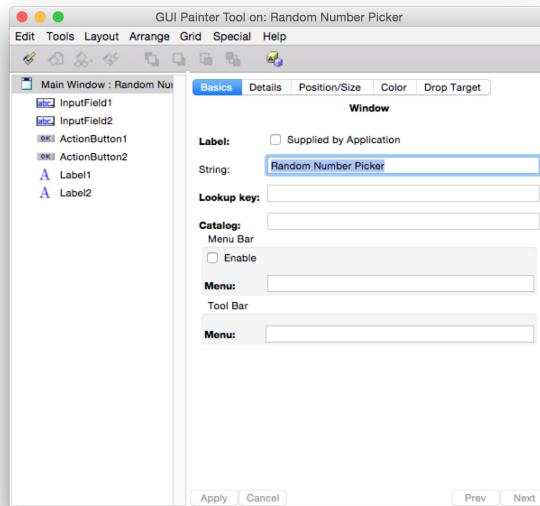
Again, we want this to be right justified, so fix that on the **Details** page. This time, however, we want to be able to enter a value in this field, so do not check **Read Only** or uncheck **Can Tab**. And, as always, click **Apply**.

We've now done all the essentials of the GUI design. We can spruce it up a little, however. Notice that the input fields are not labeled, so it's not clear what they are. If necessary, move the fields a little so there's room to label them with a label just above and to the left of the field itself.

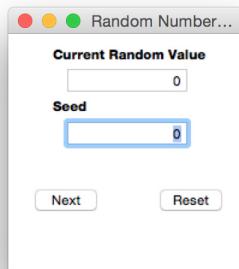
Now, find the **Label** widget, and put one above and to the left of each input field. Open the properties pages for each label and change their **String** field to **Current Random Value** for the top one, and **Seed** for the lower one. Remember to click **Apply**.

If the alignment of your widgets is a bit ragged, or the spacing is uneven, there are some aids. For example, select a label, then shift-click to select the other label, too. In the Palette, find the **Align Left** button and click it. The labels will line up with the first one selected. You can do the same for the entry fields and buttons, can even out the distribution, and so on. Explore the options until the widgets are arranged to your liking.

One more thing. We don't really want our canvas unlabeled. So, click on the background of the canvas, or click on the **Main Window** item in the widget list, so no widget is selected. Change the label string on the **Basics** properties page to something interesting, like **Random Number Picker**. Since we're done setting properties, click **Apply**.



Your final canvas should now look something like this:



This is a good time to save the image again. Select **File > Save Image** in the Launcher.

Generating initial Smalltalk code

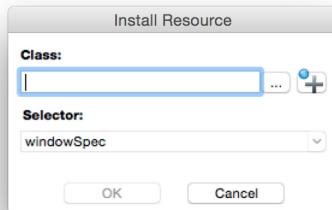
So far we've laid out the GUI and specified names for a couple of methods that will need Smalltalk code. But, there is not yet any actual Smalltalk code written or created.

In addition to methods, which are Smalltalk code, there are class definitions, which are also Smalltalk code. Fortunately, taking the GUI-first approach allows us to do a lot of this automatically. The class will represent, or *model*, the application itself.

First, we will generate the class by “installing” the canvas. You need to have the canvas and the GUI Painter Tool open.



In the GUI Painter Tool, click the **Install** button (or select **Edit > Install**). The **Install Resource** dialog opens, asking you to specify the application class name.



Installing a Canvas

Installing a canvas is how the canvas design is stored in VisualWorks.

Every time you make a change to the canvas, you must install the canvas again. Use the same class and method for reinstalling a canvas, to update your application.

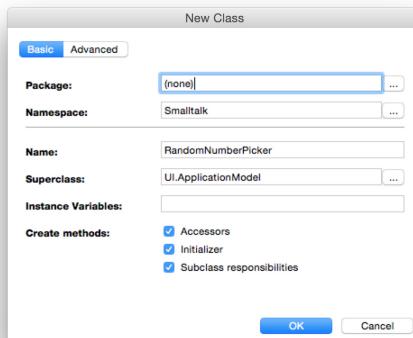
A little more about objects here. I've mentioned that action buttons are objects. What we really have on our canvas are two “instances” of the one “class” of action button. A class is an abstract description of a whole collection of concrete individuals of that kind. Our two button instances are concrete, and differ from the prototype, or class, in having specific labels and actions associated with them.

This is true of application programs, too, like the one we're creating. All concrete objects in Smalltalk are instances of some class. Even if there is only one instance, it is defined by a class (it would be a singleton class, but a class nonetheless).

So, we need to name this new class we're about to create something. The convention for class names is similar to the convention for method names, except that they start with an uppercase letter. So, let's call our class “RandomNumberPicker,” taking our cue from what we labeled the canvas. In the dialog, enter **RandomNumberPicker** in the top entry field.

Notice down at the bottom that there is another word, `windowSpec`. This is actually a method name. You could change it, but don't. VisualWorks assumes that the initial GUI is defined in a method called `windowSpec`, and you have to work harder to launch an application if you change this name.

The **OK** button isn't available yet, so we need to do something else first. Click the **+** button to see what more we need. We get another dialog:



Because we're defining a new class, the system wants a little more information. Specifically, and indicated by the little caution symbol, it wants a package name.

Packages are organizational groups, and we'll talk about them more later. For now let's be noncommittal. Click on the **...** button next to the package name, select the **(none)** option, and click **OK**.

Don't worry about the other options in the **New Class** dialog right now. The defaults are all good, so just click **OK**.

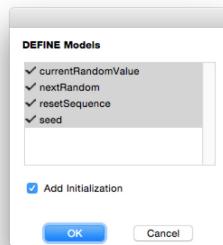
At this point, the **Install Resource** dialog should now look like:



Click **OK**. That's it for creating the class and adding a window specification to it as a resource. Now the dialogs close and you're back with the canvas and the UI Painter Tool. The class `RandomNumberPicker` is now created and the `windowSpec` method is defined in it. We'll look at those shortly.



One more thing to do before moving to the code itself. Make sure no widgets are selected in the widget list by selecting **Main Window**. Then, in the UI Painter Tool, click the **Define** button. A **Define Models** dialog opens:



Look at the names in the list. Do they look familiar? They should. Those are the names we gave to the aspects of our entry fields and to the actions of our action buttons.

Each of those needs some Smalltalk code behind them. This dialog is offering to create “stub” methods; little, essentially empty, place-holder methods that we can then go in and modify. All the methods with check marks will be generated. The initialization check box indicates that a little bit of specific code, called “initialization” code, will be included in the stub when appropriate.

Make sure all the methods are checked, and the **Add Initialization** check box is checked. Then click **OK**.

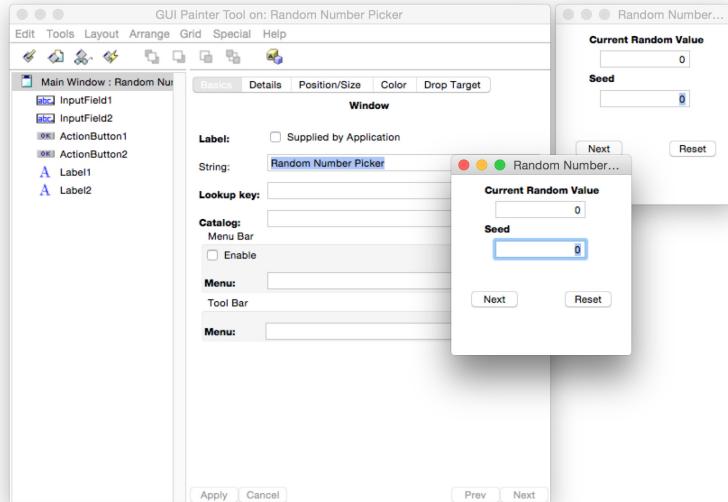
Save the image again before we move on.

Try it out

Before moving on, let's test what we have. Don't expect much yet, but we really do have a working version of our GUI.



With the canvas still open, click **Open Window** in the UI Painter Tool. This runs the application, such as it is, right from the canvas.



The application UI opens, and is ready to go. Well, sort of. Try clicking its buttons. Nothing happens. That's because we haven't told it what to do when its buttons are pushed. That's next.

But, so far so good. You're making progress! Now, to make it work.

Smalltalk Code: it's time



Let's keep this simple for now. With Main Window selected in the UI Painter widget list, click the **Browse** button. You open what's called, generically, a code browser. This one is a *hierarchy* browser, showing the class you just created, RandomNumberPicker, and all of its *superclasses*, all the way up to Object.

```

RandomNumberPicker
Browser Edit Find View Package Class Protocol Method Tools Help
Object Model ApplicationModel RandomNumberPicker
Find: RandomNumberPicker

Comment Definition Rewrite Code Critic 3 Watches
Smalltalk defineClass: #RandomNumberPicker
superclass: #(UI-ApplicationModel)
indexedType: #none
private: false
instanceVariableNames: 'currentRandomValue seed'
classInstanceVariableNames: ''
imports: ''
category: ''

```

Subclasses

A class defines the kind of a thing. A subclass defines a more specific kind of a more general thing.

So, if a bird is defined as a class, a penguin could be defined as a subclass, a more specific kind of a bird.

The top-left window pane shows the hierarchy of classes, starting at the top of the hierarchy (Object), then each of its *subclasses*, all the way down to RandomNumberPicker. The bottom pane shows the definition for whatever is currently selected. Initially, it is our class itself, so it is the class definition that is shown.

Without explaining this definition in detail, look at a few of the items we're already familiar with. When installing our canvas, we called its class RandomNumberPicker, and decided to leave it in the Smalltalk name space. That's what the first line says. The second line says that it is a subclass of ApplicationModel, or that ApplicationModel is its superclass, which is the same thing. The fifth line identifies instance variables, that is, variables whose values are determined separately for each instance of the class. You should recognize the two variable names as the aspects of our input fields. The last line is the category that we could have identified for the class when defining it. You probably left it blank. I'll say more about categories shortly.

The other lines aren't important now, but you can learn about them in the *Application Developer's Guide*.

The second pane of the browser shows packages that contain definitions either of this class or methods in it. I haven't mentioned packages yet, but they are closely related to categories, which I've also not described. Let's come back to that. So far, our class isn't in a package, so this pane shows **(none)**.

The third pane shows method categories, or *protocols*, (I don't think much of the latter term, so will avoid it, but it's traditional and you *will* hear it). In this case, the categories are for methods that define messages that can be sent to instances of the selected class. Notice that the tabs above that pane indicate that instance-side definitions are being shown. These are methods that are sent to instances of a class. Class-side methods are sent to the class itself.

Select the **Class** tab for a moment, and see what's there. There's just one category here, called interface specs, and there's just one method, `windowSpec`. This method stores all the data that sets the look of the canvas. This was created by the UI Painter, so you pretty much never need to edit this method directly. Still, it's nice to know it's there. Click on it to see what it looks like. Don't worry, you're not supposed to understand the code generated by the Painter.

Click the **Instance** tab in the browser, to get back to the instance-side definitions. There are two categories for methods here: actions and aspects. That should give you a clue about what you'll find in each one.

Click on actions. Here are the stub method definitions for our action buttons. Click on them. Except for their names, they say the same things right now, e.g.:

nextRandom

"This stub method was generated by UIDefiner"
 `^self`

The line in quotation marks is a comment, and isn't executed. The last line, `^self`, just returns the object itself, in this case the application. That's not what we want, so we'll change it.

A method always returns a value (which is *always* an object) to whatever sent the message invoking the method (remember, their names are the same). The default return value is the message receiver, `self`. To specify a different return value, precede the expression with the caret (^), as in this method. Of course, the explicit value here specified is the same as the default value, but we're going to change that. It's just a place holder for now.

Class vs. Instance Methods

Class side methods define messages that are sent to the class itself, asking the class to do something. For example, `new` is a class message that tells the class to make an instance of itself.

Instance side methods define messages that are sent to instances, asking that instance to do something. For example, if an instance has a color, it can be asked for that color.

Before we change this method and the next one, we'll have to look at the VisualWorks random number generator. Right now we're just touring our class.

Click on the aspects method category. These two methods are supposed to return the value of the indicated aspect, which are typically values of instance variables. They're the same at the moment, except for the names of their variables:

seed

"This method was generated by UIDefiner. Any edits made here may be lost whenever methods are automatically defined. The initialization provided below may have been preempted by an initialize method."

```
^seed isNil  
ifTrue:  
    [seed := 0 asValue]  
ifFalse:  
    [seed]
```

There's more code here, which is written in "lazy initialization" style. That refers to how, or when, the variable gets its initial value. In this case, the variable named seed doesn't get its initial value until the message is sent the first time, rather than in an initialization method that would set the value when the instance is created.

When a new instance of an object is created, that object's instance variables are also created, and have a initial value of nil, no value. Hardly ever is nil an appropriate value, as it is not here, and will frequently cause programs to error. So, before the variable is actually used, it must be assigned a value.

We could, in another method called initialize, provide a value for seed and currentRandomValue as soon as the instance is created. Sometimes that's necessary, but here it's not. So, we will use the default style, lazy initialization, and just modify it as we need to.

Note the caret, specifying that the value of seed will be returned, rather than self. But, what is seed? Well, it depends on the rest of the expression. Here, the conditions are evaluated first, determining the value of seed, and that value is returned.

What the seed and currentRandomValue messages are supposed to do is return the current value of the aspect of the model they represent, when asked. If they don't have a value when asked, that is if their value is still nil, they make it up and hand back a default value, currently 0.

ValueHolders

asValue creates a ValueHolder and assigns a value to it.

value: assigns a new value to an existing ValueHolder

value returns the value held by a ValueHolder

The `asValue` message places the value 0 in a *value holder*, a special kind of object, an instance of the class `ValueHolder`, which much of the GUI framework assumes to be holding the values it has to display. The really neat feature of a value holder is that it automatically updates the GUI when its value changes, saving us some work. The value holder is itself what is stored in the `seed` variable in the above code.

When we put a new random value into this variable, we will put it in the value holder, too, using the `value:` message. The GUI framework retrieves the value itself by sending another message, `value`, to the value holder. This is a GUI framework architectural thing, but you need to understand this much of it or you will be constantly confused and frustrated later.

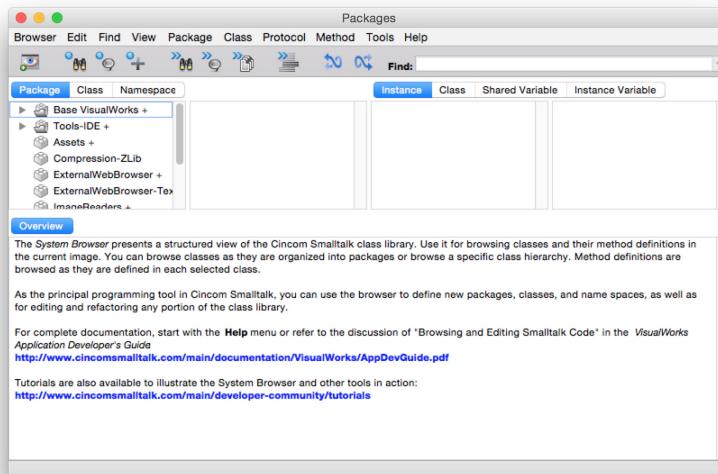
More Smalltalk Code - Package Browser

We need to look at another bit of Smalltalk code, the `Random` class, because this is the class that will provide us the data to display. An instance of this class will give us new random numbers. It also gives us an excuse to look at another essential VisualWorks tool.



Click on the Visual Launcher to bring it to the front of your display. Find the System Browser button, which opens a tool that gives you a view of the whole system.

Click on the System Browser button; you might as well see the whole thing. The System Browser opens:



You'll notice that this is very similar to the hierarchy browser, but the top-left pane now shows a list of packages and bundles. These are groupings of classes related by some common functionality. Bundles contain packages, and possibly other bundles. Packages are what actually contain the code, but viewing a bundle shows all the code defined in its packages.

This is a good time to say something about categories. It hasn't always been the case, but currently categories and packages amount to the same thing. If you explore a few packages and the classes they contain, you'll see that the category and package names (not the bundle names) are the same, almost always. Categories are a traditional way of grouping classes that were related in some way. Packages do the same, but provide more organizational features, and are the foundation of the database code storage facilities available in VisualWorks if you load Store. (Store is described in the [Source Code Management Guide](#).)

The next pane over shows a list of all classes currently defined in the selected package or packages. For example, if you select **Base VisualWorks** in the leftmost pane, you will see all the base classes in this pane. Several items are marked with icons noting something special about them, such as that they are a name space, an exception class, a collection class, and so on. In addition to classes, this list also shows name spaces and shared variables, which we'll have to explain a bit later.

You can select all packages and bundles to see all of the classes in the system. Or, you can see the entire class hierarchy, starting with Object, by clicking on the **Class** tab. Or, select a class in a package and then click the **Class** tab, to display just the hierarchy for that class. There are other views for browsing the system as well, as described in the [Application Developer's Guide](#).

The Random Class — Domain Models

Reminder: Model and UI

We said the model (or domain model) does the thinking, or the processing, of the program. Random does this for our application.

The UI, our RandomNumberPicker, provides the user interface to Random.

Often when you build an application like we're doing, there is already a class in the VisualWorks environment that does some work, such as recording business transactions and performing calculations. Such a class "models," or represents in the program, a "problem domain," whether it is a business process or a representation of a scientific theory.

In setting up this example, we've chosen to build a GUI that provides input to and accepts output from a simple model that already exists in VisualWorks, namely a model of random numbers. This model is defined in the class Random. Let's take a look at it.

You can either use the System Browser you already have open or open a new one. It is common, while working in Smalltalk, to have several browsers open. The only limitations are your confusion threshold, the size of your screen, and, eventually, your computer's memory. I'm going to use a new browser so I can use the old one to look at RandomNumberPicker whenever I want.

Since we don't know what package to look in, but do know the class we want (because I just told you), we'll use the find-a-class command. Simply type **Random** in the **Find** field in the system browser, and press Return. Or, in the package list, right-click and select **Find Class...** and enter **Random**. The browser finds Random and selects it, and selects its package, **Magnitude-Numbers**.

The **Find** features can be useful, also, if you know only part of the class name. You noticed as you typed that it showed a list, first of all classes starting with **R**, then only those with **Ra**, and so on until the list was quite short.

The right-click, or **<Operate>**, menu, by the way, is different depending on where the mouse is. Try it in the other panes. The same menu is available (at least usually) from the menu bar at the top of the browser. The code pane **<Operate>** menu is the same as the browser **Edit** menu.

Now that we've found Random, let's take a look at it. Looking at its class definition (click on the **Definition** tab in the browser), we see:

<Operate> Menu

In most tools, you can do a right-click to pop-up a menu of operations. The menu changes depending on where the mouse is when clicked.

Most of the time, the same menu options are available on a normal menu, also.

```
Smalltalk.Core defineClass: #Random
superclass: #{Core.Stream}
indexedType: #none
private: false
instanceVariableNames: 'seed'
classInstanceVariableNames: 'defaultGeneratorClass'
imports: ''
category: 'Magnitude-Numbers'
```

You can see here that it is in the Core name space (Smalltalk.Core). It also has one instance variable, seed, which I've already mentioned in connection with our GUI. Here's the connection.

There's also a class instance variable, which identifies the class that provides the random number algorithm. Random is both an abstract superclass, and a convenient access class for random number generators. Most of its behavior is actually defined elsewhere.

You can also see where Random is located in the class hierarchy. Click on the **Class** tab on the upper-left of the System Browser, and the **Package** list is replaced by one that shows:

```
Object
Stream
Random
FastRandom
LaggedFibonacciRandom
ParkMillerRandom
MinimumStandardRandom
```

You can see that Random is not very deep in the hierarchy, just one class away from Object, the root class. Stream is an interesting class, essentially a superclass for classes that deliver a data series, or "stream." Random does that, delivering the next random number upon request.

Random also has a few subclasses, which are the actual generators. The default generator is LaggedFibonacciRandom (at least in this release), so we'll look at it a little later.

There are a number of tabs on the code view of the browser, which are all described in the [Tool Guide](#). One that is particularly useful while exploring the system is the **Comment** tab. Click on it to display the class comment. When written properly, the comment explains the purpose of the class, possibly how it should be used, and explains the important variables. While trying to understand the VisualWorks classes, don't neglect these comments. (And, as a programmer, you will write your own comments when you create a class or write a method, won't you? Make sure you do.)

Now let's look at the code, first on the class side. Click the **Class** tab.

There are a few method categories here, all of which occur frequently on this side of class definitions. The class initialization category contains methods, in this case a single method called initialize, that set up values governing the class itself and all instances of the class. (This requires a lot of explanation that I will not give here. Refer to the [Application Developer's Guide](#) and *Smalltalk-80: The Language* for more information.)

The instance creation category generally contains the protocol for creating instances from the class. Select this category and look at these. Notice that the lower part of the browser now switches to offer a **Source** tool.

Class-side messages are sent to the class itself, as we'll see in a minute. Many classes have a new message, which is the simplest form of instance creation. A version of new is implemented in Object, and so is inherited by every class, though for some it's useless. When it is useful, it is generally reimplemented, as it is here in class Random:

new

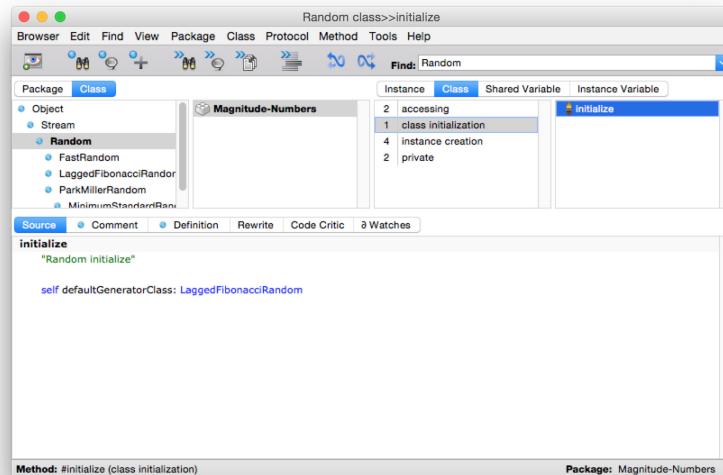
"Answer a new random number generator."

`^self defaultGeneratorClass new`

This method, like all methods, is defined by a selector name on the first line, new, followed by a few lines of code. Lines between double-quotation marks are comments, and are there for explanatory purposes. The rest consists of lines that send messages to objects, either the current object itself, or other objects.

There is only one message sent in this method. The message begins with the receiver object, in this case self, which refers to the sender itself, Random. It is sent the message defaultGeneratorClass, which returns the class that actually defines the generator we'll use, LaggedFibonacciRandom. (This is set in the initialize method.) This class is then sent the message new, which returns an instance of the generator.

It's been a while since we've had a screen shot, so here's one:



By the way, all those panes are resizable, so you can make your browser look more like this. Just drag a border.

Often, as in this case, a class variable's value is set in the class's initialize method, since it only needs to be set when the class is created, and remains the same for the life of the class.

If you want to change the default generator, you can simply implement the new generator, then assign it here.

Now we can see what happens back there in Random's new message; it simply sends new to the actual generator, LaggedFibonacciRandom. To dig a bit deeper, find and select that class in the browser, and look at its new method, which looks like this:

```
new
  ^self basicNewInstance
```

This message isn't defined in the class, but in a superclass. In fact, in Random, where it is defined as:

```
basicNewInstance
  ^self basicNew initialize
```

The first word, self, we've already seen, and refers to the initial message receiver. This is a class method, so here it refers to the class itself. basicNew is the really primitive instance creator method, but we won't

Message Parsing

Unary messages take precedence over binary messages, and are parsed from left to right.

Binary messages take precedence over keyword messages, and are parsed from left to right.

Keyword messages are evaluated last.

Messages in parentheses take precedence over unary messages.

chase it further here. So, the result of self basicNew is an instance, and it becomes the receiver of the message initialize, which is an instance method.

In Smalltalk, a series of message sends like this are evaluated from left to right, according to a set of rules of precedence. Sometimes parentheses are helpful to make clear what's going on. So, we could have written that message series:

```
^(self basicNew) initialize
```

Sometimes parentheses are even necessary, but they aren't here.

Refer to the *Application Developer's Guide* for a fuller description of how Smalltalk expressions are parsed and evaluated. It describes the "precedence rules" for various operations and the use of parentheses.

Back to looking at LaggedFibonacciRandom, click on the **Instance** tab, select the **private** category, and select initialize. Here, finally, is where the generator really gets set up:

initialize

```
super initialize.  
self initializeValueStorage.  
self seed: Time microsecondClock
```

Understanding how this random number generator actually works will be left as an exercise for the reader (that's you). But, what this method does, as do most instance initialization methods, is assign values to instance variables. These values are used only by this instance of the object, unlike class variables which share their values rather more widely, as mentioned.

In summary, when we send new to Random, it in turn sends new to the default generator class. What that class does with the message is up to it, but in this case it sends basicNew, to create an instance, then sends initialize to set the instance variables, so it's ready to generate random numbers. Make sense? It might take a little pondering. But, let's move on.

Select Random again and click on the **Instance** tab. There are a few categories here.

The testing category typically contains methods that report important conditions of the instance. Here, atEnd reports whether the random number stream is finished, which it never is, and so it always returns false. Similarly, an object should not be able to push a value into a stream, so isWritable also returns false. However, the point of a random number generator is to provide values, so the stream returns true to isReadable.

The accessing category is an important category, containing “accessor methods,” methods that set and retrieve values in the object’s instance variables. Here, only the seed: method is important to us, and not for a little while.

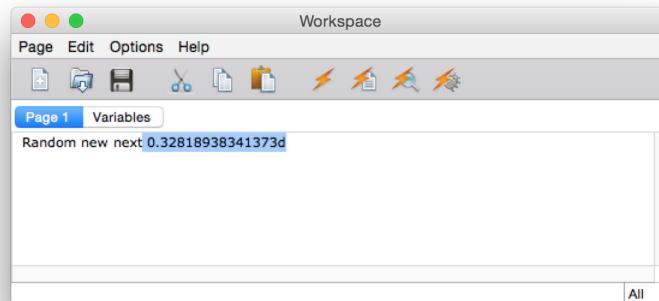
We need to look back at LaggedFibonacciRandom to find one more important piece of behavior, so find it in the browser. Among its instance methods, find next in the accessor methods. This method returns a value based on the seed value, and updates the seed. The real work of calculating the value is done in nextValue, which you can try to figure out in your spare time, but it’s not important to understand it for our purposes. That’s the stuff we refer to as “implementation details.”

Thinking back on what we’ve seen, and thinking in terms of what we need for the UI, the methods that we potentially need from class Random are:

- Class methods
 - new - to get a random number generator object.
- Instance methods
 - next - to get the next value from the generator.
 - seed: - to set the seed value for the generator.

Let’s quickly look at how the messages work, and what they return, before continuing to work on our application itself. To do this, we use the VisualWorks Workspace, a tool we haven’t explored yet.

Open a workspace by clicking on the icon in the Visual Launcher that looks like a pad of paper, or by selecting **Tools > Workspace**. The workspace opens looking like a blank window. You can drag its corners to a size more to your liking.



Workspace

The <Operate> menu for a workspace has normal editor commands, plus:

Do it, to evaluate an expression silently.

Print it, to evaluate an express and print the return value.

Inspect, to open an inspector on the return value of the expression.

The workspace is a good place for doing free-form evaluation of Smalltalk expressions. It can be used for testing code, to make sure you know how a message works and what it returns.

Try this. In the workspace, type:

Random new next

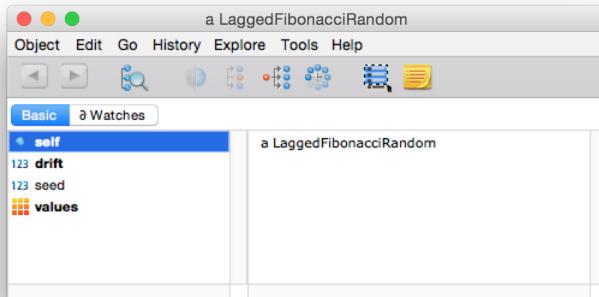
Click on the expression, so the cursor is somewhere on it. Right-click in the workspace to open its <Operate> menu, and select **Print it** (there's also a button for this). A floating point number (a Double, or double-precision real, to be doubly precise) is displayed immediately following the expression. This is the first random number in the sequence generated by the default generator. For comparison, select just the Random new part, and **Print it**. This time, it says: "a LaggedFibonacciRandom." Remember, that's the default generator.

Sometimes you simply need to perform an action, without seeing the result in this window. Select the whole expression again, and select **Do it** from the <Operate> menu (or the corresponding button). Nothing is displayed, but it does the same thing. Now try typing into the workspace:

Transcript show: (Random new next) printString

and evaluate it with **Do it**. There's no response in the workspace, but a number is shown in the Transcript, the text window part of the Visual Launcher.

Here's another useful trick for learning your way around objects. If you can get an object, you can inspect it. Select Random new and select **Inspect it** on the <Operate> menu. The window that opens is an inspector on the instance of Random:



Selecting self just shows a LaggedFibonacciRandom, like you saw in the workspace. The other items listed by the Inspector are its instance variables. While an object is alive, you can watch these values change. There is no instance variable for the current, or next, value, since it is calculated from the seed based on an algorithm embedded in the code. You can study the code sometime to tease it out, but it is an “implementation detail” and so will seldom, if ever, be discussed.

We can return now to our application itself, and finish this up.

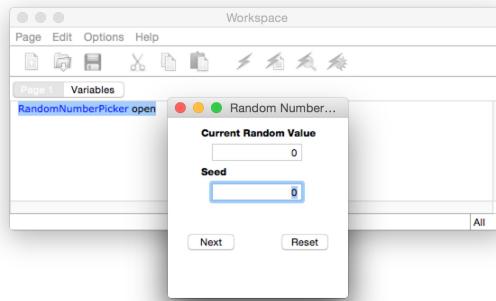
Putting the GUI and the Model Together

We've built our GUI and explored the model that will be used to provide values to the GUI upon request. Now we just need to put them together to finish our application.

We've already seen that we can open our application; it just doesn't do anything. Because it's a subclass of ApplicationModel, it responds to the #open class method. So, you can enter this in a workspace and evaluate it with **Do it:**

```
RandomNumberPicker open
```

The application opens looking like this:



This does exactly the same thing as clicking the **Open** button in the GUI Painter Tool.

As we've already seen, the buttons of the RandomNumberPicker don't do anything. We have to fix that now. Close the application, because we'll have to reopen it after a couple of changes.

The first thing we need is a Random instance from which to get values. We know how to do that in a workspace. In an application, though, we want to make one and hold onto it. We do this by putting it in an instance variable, which we must add. So, find RandomNumberPicker in a System Browser (you may still have one open on it) and get the class definition itself in the code pane. To the list of instance variable names, between the single-quotes, add a variable, which I'll call myRandom. Then select **Edit > Accept**.

Your class definition should now be (with the change highlighted):

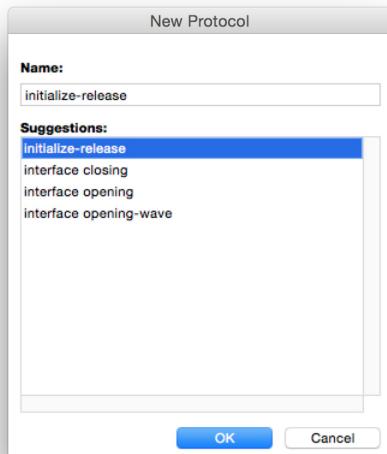
```
Smalltalk.Tutorial defineClass: #RandomNumberPicker
    superclass: #{UI.ApplicationModel}
    indexedType: #none
    private: false
    instanceVariableNames: 'currentRandomValue seed myRandom'
    classInstanceVariableNames: ''
    imports: ''
    category: 'UIApplications-New'
```

(You may see a warning that the new instance variable isn't referenced — ignore that for now.) The VisualWorks application framework gives us an easy way to initialize myRandom when the application opens. In the early stages, right after creating the instance, it invokes the object's initialize method, if one exists. This is where you typically set such variables.

initialize Method

As soon as an object is created, its **initialize** instance method is evaluated. This is the method used to set instance variable values early. You also use it to set up dependencies between widgets, if necessary (see the *Application Developer's Guide* for information).

In the System Browser focused on RandomNumberPicker, make sure you're looking at the instance side (select the **Instance** tab). In the method category pane, select **New...** from the <Operate> menu to add a new category:



The traditional category for the initialize method is initialize-release. No point in breaking tradition now, so enter **initialize-release** in the prompter and click **OK**. This is another one of those helpful dialogs that makes recommendations once you start typing.

Select the new method category, and a method definition template is shown. Replace the method definition template with this:

initialize

"Create the initial random number generator for this application"

```
myRandom := Random new.
```

No surprises here, I hope. Smalltalk uses `:=` to assign a value to a variable. As soon as the application opens, this method creates a new instance of Random and holds it captive as `myRandom`. Select **Accept** on the <Operate> menu to save the code.

Now select the actions category and look at the `nextRandom` method. We need to make the button do something. This is deceptively easy, because the application framework does so much for us.

You can guess that when the user clicks the **Next** button we will want to send the message `myRandom next`. But, what shall we do with it? Display the new value, of course. To do so, we need to put the value in `currentRandomValue` and make it display. We could do the value assignment this way:

```
currentRandomValue := myRandom next
```

We'd need a way to update the display, then. That can be done, but instead we'll use the cool feature of value holders mentioned on [page 41](#), and both set the value and update the display with a single message, `value:.` So, we shall write the `next` method like this:

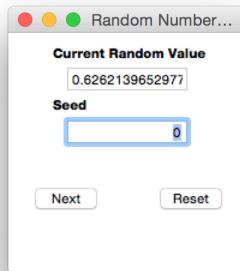
nextRandom

"Update the current random display when the Next button is clicked"

```
currentRandomValue value: myRandom next
```

Select **Accept** on the <Operate> menu to save the code.

Now, launch the application again and see if anything has changed. (If you still have the canvas open, click **Open** on the UI Painter Tool; otherwise, use the workspace expression `RandomNumberPicker open.`) Try clicking the **Next** button. Anything happen? It should now show a new number. Click it a bunch of times and watch it change.



Don't close the application this time, because we're going to see the dynamic update feature of Smalltalk. Click the **Reset** button to verify that it does nothing at this point. We're going to fix that right now.

The **Reset** button is similarly straightforward. Select the `resetSequence` method, and replace its stub code with:

```
resetSequence
  "Start a new Random"
```

```
myRandom := Random new.
currentRandomValue value: 0.
```

and **Accept** the change. Now click the **Reset** button. Everything goes back to the start. Click the buttons, just to make sure everything's working.

You know enough to understand what's going on in this method. **Reset** just creates a new instance of Random, starting a new sequence, and resets `currentRandomValue` to 0, using the `value:` message so the display is updated.

Pretty slick, huh? And we didn't even have to restart the application. This is a feature of Smalltalk that developers love. No need to write code, recompile the code, and relaunch the application for testing, in a seemingly unending cycle. The exception earlier, when we had to relaunch, was because we added that instance variable.

Save your image, and take a deep breath.

We're almost done. We aren't doing anything with the seed yet, but now we will.

Using the seed value

Go back to the System Browser and look again at the new method in `Random`. Remember, it looked like:

```
new
  "Answer a new random number generator."
```

```
^self defaultGeneratorClass new
```

Now, look more carefully at the last line. If you select it and select **Print it** on the **<Operate>** menu, it prints a `LaggedFibonacciRandom`. This shows that the result is an instance of the default generator. There are lots of ways to generate pseudo-random numbers. The default one is specified in `Random`'s `initialize` method.

`Random` also provides a method for assigning a new seed to an instance, called `seed:`. The colon indicate that an *argument* is expected, a value that the receiving object needs in order to provide a response to the message. We will get the desired effect of setting the seed by simply sending this message, with an appropriate argument, to our new instance.

Back in our UI, you can't see any response yet, but you can click in the **Seed** field and enter a number. When you then hit **Enter** or click one of the buttons, the seed variable is updated with this new value, automatically. We want to use this changed value to set the seed in our random number generator.

What the seed value does is provide a known starting point for the pseudo-random sequence, allowing us to repeat a given sequence of values. The only time we want to do this is when a new Random is created. The first time may be too early, but when the **Reset** button is clicked we can use the seed value to select the sequence.

But, we don't always want to use the seed value, because sometimes we really want the sequence to be more arbitrary. If in the initial case it were set to 0, we would always start with the same sequence, which we don't want. We want the initial series to be arbitrary, more random.

We can do this by writing the program so that if seed is 0, then the instance of Random is created without a specified seed, but if seed is anything else, it is created with the seed specified. Make sense? Here's what we'll do.

Display the resetSequence method definition. There are a lot of ways to do this, but replace the method's definition with this:

resetSequence

"Start a new Random, with a specified seed if seed is not 0"

```
| seedValue |
seedValue := self seed value.
seedValue = 0
    ifTrue: [myRandom := Random new]
    ifFalse: [myRandom := Random new seed: seedValue].
currentRandomValue value: 0
```

You can probably figure out what's going on in most of this based on explanations given before, but some deserve further comment.

Remember that seed is the variable holding the contents of the **Seed** input field. While this is an instance variable, and so can be accessed directly by the instance, instead we access it by sending the seed accessor message to self. This can be important, especially since we used lazy initialization, to make sure the value has been properly set, and isn't nil. In this case it's not particularly important, because the seed message is sent when the application opens, but that is not always the case. We could have written the line:

```
seedValue := seed value.
```

While it looks like seed is holding a number, it's really not. It's holding a ValueHolder. But, Random needs to perform arithmetic operations, so needs a numeric value, not a ValueHolder. Extracting this value from a ValueHolder is done with the value message. In order to do this only once rather than twice in this method, I've added a temporary variable, `seedValue`, and set it to the value of seed. That's the first two lines of the code section, following the comment. (See, I commented my code!)

Control Methods

`ifTrue:ifFalse:`, and similar messages, set conditions for performing certain operations.

Other messages, such as `while:do:`, iterate an action as long as the condition obtains.

Then, as described above, we want to split the cases where `seedValue` is 0 and where it is not. This is done with a conditional branch. There are a few messages for doing this in VisualWorks. Since we need to do something different for each case, I've used the `ifTrue:ifFalse:` message. (You can see other such messages by browsing the Boolean class, and they're described in the [Application Developer's Guide](#).)

We start the *branch* by testing whether `seedValue` is equal to 0. That's the `seedValue = 0` part. Pretty clear, I hope. The result of that test is either true or false (which are instances of Boolean subclasses). If it is true, the `ifTrue:` branch is evaluated, and `myRandom` gets set to `Random` new as before. However, if it is false, then the `ifFalse:` branch is evaluated, and we create the instance and send `seed:` to it.

By the way, the arguments to the `ifTrue:` and `ifFalse:` keywords, the expressions enclosed in square brackets, those are called *block closures*, or simply *blocks*. These are another powerful feature of VisualWorks, representing a deferred sequence of actions. They are themselves objects, instances of class `BlockClosure`. Blocks are required as the arguments for some messages, such as `ifTrue:ifFalse:`, but have uses in a lot of contexts. Blocks are covered in detail in the [Application Developer's Guide](#).

That's it! Once you've entered the method code, test the application again. Enter a value other than 0 into the **Seed** field, and click **Reset** and **Next** a few times to see that it always starts generating the sequence at the same value. Change **Seed** back to 0, and click **Reset** and **Next** a few times to see that it always starts with a different value. That's what we wanted.

Note that, because the seed value is only used deeply within the calculation of the new random sequence, there is no obvious relation between the seed and any of the resulting random numbers. For instance, it does not specify the smallest random number. All it does is force a repeatable sequence of numbers.

The application works! We're done (almost). Save your image!

Move to its own package and name space

We've mentioned name spaces earlier, but have tried to skirt the issue, because it can be complicated. But, they are important and powerful. Without them, shared variable and class names could only occur once in the whole image.

If two programmers decided to create a class called Customer for their applications, for example, and both applications were loaded into the system, the second one loaded would stomp on the first one. This isn't funny, and it has happened. The work-arounds have been creative, but name spaces are the right solution.

Name spaces form a hierarchy, starting with Root at the top, then followed by Smalltalk, then a whole bunch of other name spaces. All Smalltalk code should be in sub-name spaces of the Smalltalk name space. But, remember, we didn't do that. We just let our class and its code be created in Smalltalk. So, let's move it.

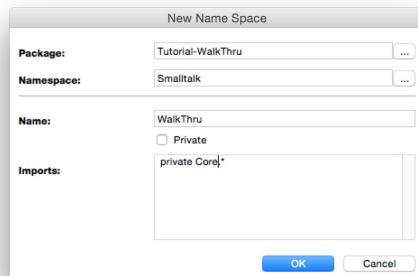
If you still have a browser open on RandomNumberPicker, you can use that. Click on the **Package** tab to show the list of packages and bundles. If you don't have the browser open, open a new System Browser.

RandomNumberPicker is in the (**none**) pseudo-package, since we haven't created one for it, so make sure it is selected. Only RandomNumberPicker is in that package right now.

Let's put it in its own package, say, Tutorial-WalkThru. Make sure you don't have a bundle selected (selecting (**none**) will ensure that), then pick **Package > New Package....**. In the prompter, which should look familiar by now, enter **Tutorial-WalkThru**, and click **OK**. The package is added to the package list and selected. If (**none**) was selected, it still is, so it looks like RandomNumberPicker class definition is already in it. But, it's not. We have to move it.

With (**none**) selected, select RandomNumberPicker in the class list. Then, select **Class > Move > All to Package....**. In the dialog, another one of those helpful selection dialogs, start typing **Tutorial-WalkThru**. Select it and click **OK**. You can now verify that the class and its methods are in the **Tutorial-WalkThru** package, and not in (**none**).

Creating and moving our class to its own name space is similar. Select **Tutorial-WalkThru** (the package) again, and select **Class > New > Namespace....**. The name space creation dialog opens.



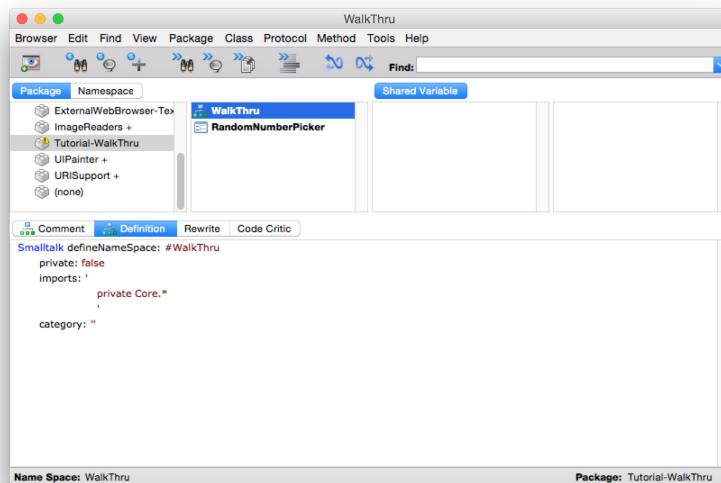
The marker indicates that we need to enter a name, so enter **WalkThru**. Notice that it already has our package name. The name space will be created in the package (it's a real object, so its definition has to go somewhere). It also shows that our name space will be in the Smalltalk.* name space, so WalkThru will be a sub-name space of Smalltalk.*

The **Private** checkbox indicates whether other name spaces will have access to your definitions. You want them too, usually, so leave this unchecked.

The **Imports** field specifies classes, other name spaces, and shared variables) that are defined in other name spaces and that you want to refer to without identifying their name space explicitly every time. Initially it includes **private Core.***. This is needed to import core classes such as Random, so leave this as is.

Click **OK** to save this definition. The dialog closes, and you are returned to the browser.

At this point, the name space has been added to the package and selected, so you can see its actual definition in the code pane:



Now, we just need to move our RandomNumberPicker class to the new name space. Select RandomNumberPicker in the class/name space list. Then, select the **Class > Move > to Name Space...** menu command (or **Move > to Name Space...** in the **<Operate>** menu). A dialog opens listing the available name space.

Select our name space, **WalkThru**, and click **OK**. The dialog closes, and the deed is done. See in the class definition, the first line now says that it is in `Smalltalk.WalkThru`.

There's more subtlety to name spaces, which you can read about in the [*Application Developer's Guide*](#).

You can now open and run the application just like before. One more time, save your image.

Comment your code!

We've been ignoring this little detail. Everyone does. No one should.

But, you've probably noticed the small, yellow "caution" sign icon next to the class name and on **Comment** tab in the browser. Sure you have, don't deny it. That's a reminder that we haven't written a comment for our class or our name space. This is easy to fix.

Select the class and click the **Comment** tab. There's a dummy message there saying the class hasn't been commented. But, we knew that.

Replace the text with something descriptive, like:

RandomNumberPicker is a UI for Random, which allows specifying a seed value, and then displays successive random values.

Instance Variables:

currentRandomValue <ValueModel> the current random number
myRandom <Stream> an instance of Random
seed <ValueModel> a seed value

Accept the changes as usual, and the yellow caution sign disappears.

Do the same for the name space. Just something simple like "Name space for RandomNumberPicker of the Walk-Through."

There! Guilt assuaged.

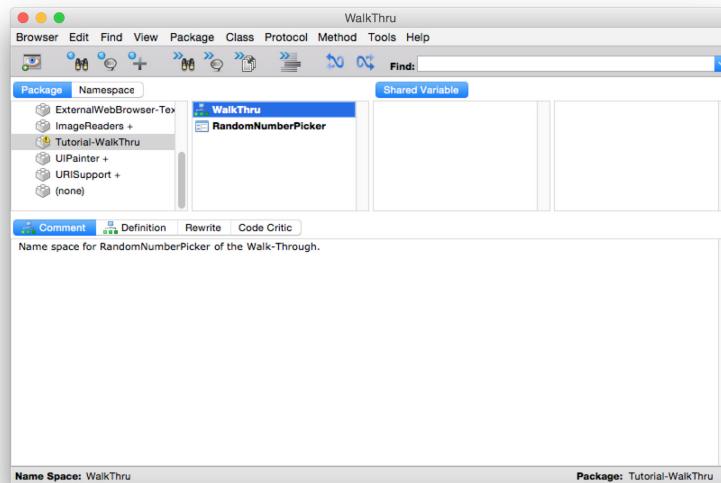
Packaging it up

OK, now you have this absolutely wonderful application that you want to share with all your friends, or sell for the big bucks. How do you do that? It can be quite a long story, but here's a short version of it.

Remember that at the beginning of all this we loaded a parcel to get the UI Painter into the system. At that time I mentioned that a parcel is an external code module. It is useful for storing our program for loading into another image, for archiving, and for distributing to friends or customers. (When such a module contains a well-defined set of functionality, it is often called a *component*.)

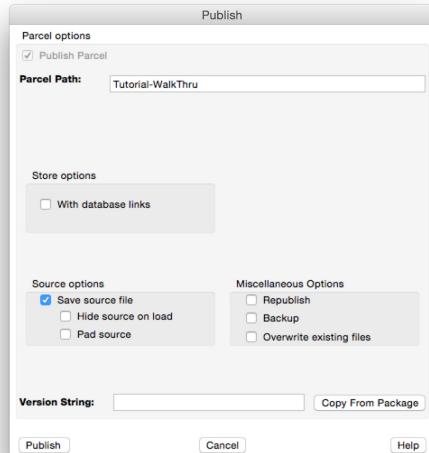
We can now make our own parcel. We won't need to keep our WalkThru image, which is big, but just the parcel. We can then load our application when we want to use it or change it. Or, email it to friends to load into their VisualWorks image.

This is pretty easy. Open a System Browser, or use one that is already open, and select our **Tutorial-WalkThru** package. The browser looks like this:



Here, we see the package is also uncommented. Give it a comment, like “This package contains RandomNumberPicker, the application model developed in the VisualWorks Walk-Through, and the WalkThru name space required by the final example.”

Next, select **Package > Publish as Parcel** (also on the package <Operate> menu). The publishing dialog opens.



There's really not much for us to do here. The parcel name is already entered in the **Parcel Path** field. (A parcel path can include directory information, but the name is enough for now.) There's no need to change any of the checkbox items, which are described on the dialog's help screen (click **Help** if you want to read about them).

So, just click **Publish**. The system does its thing and closes the dialog.

Now find the project directory, which is probably **WalkThru** in **VisualWorks Projects** in your home directory, on OS X. There should be two files: **Tutorial-WalkThru.pcl** and **Tutorial-WalkThru.pst**. Copy these to a safe place.

That's it. We're done with this application. For help loading this parcel in another image, refer to the **Application Developer's Guide**. You can also package things up as an executable, as we did for the Hello World! program, though things are a bit different with a parcel.

But, you're only starting with VisualWorks and Smalltalk. Read the documentation. Browse the class library. Build an application of your own. In a word, keep playing with VisualWorks. There's a lot more to discover.

Where from here?

You are only limited by your imagination. This walk-through has only given you a taste of VisualWorks, and given you some ideas about how to work with it. It's time to dream.

The best way to move on is to experiment. Browse the VisualWorks classes, and think of a way to use them. Refer to the documentation for hints and suggestions when you can't figure them out.

Contact other Smalltalkers. For example, subscribe to the internet discussion group for VisualWorks Non-Commercial: vwnc@cs.uiuc.edu.

Read. There are many books on Smalltalk, Object-Oriented, and agile development.

And, play! Life's short.