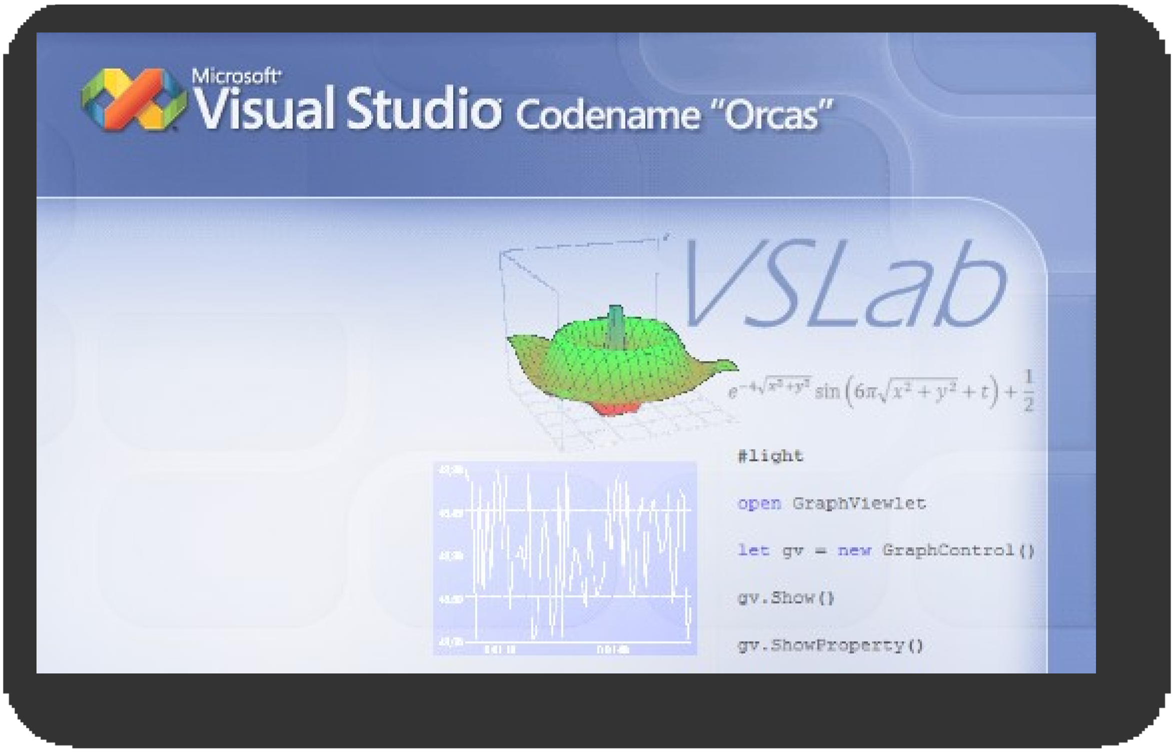
VSLab Packages

Whitepaper


Microsoft Visual Studio Codename "Orcas"
VSLab
$e^{-4\sqrt{x^2+y^2}}\sin\left(6\pi\sqrt{x^2+y^2}+t\right)+\frac{1}{2}$

```
#light
open GraphViewlet
let gv = new GraphControl()
gv.Show()
gv.ShowProperty()
```

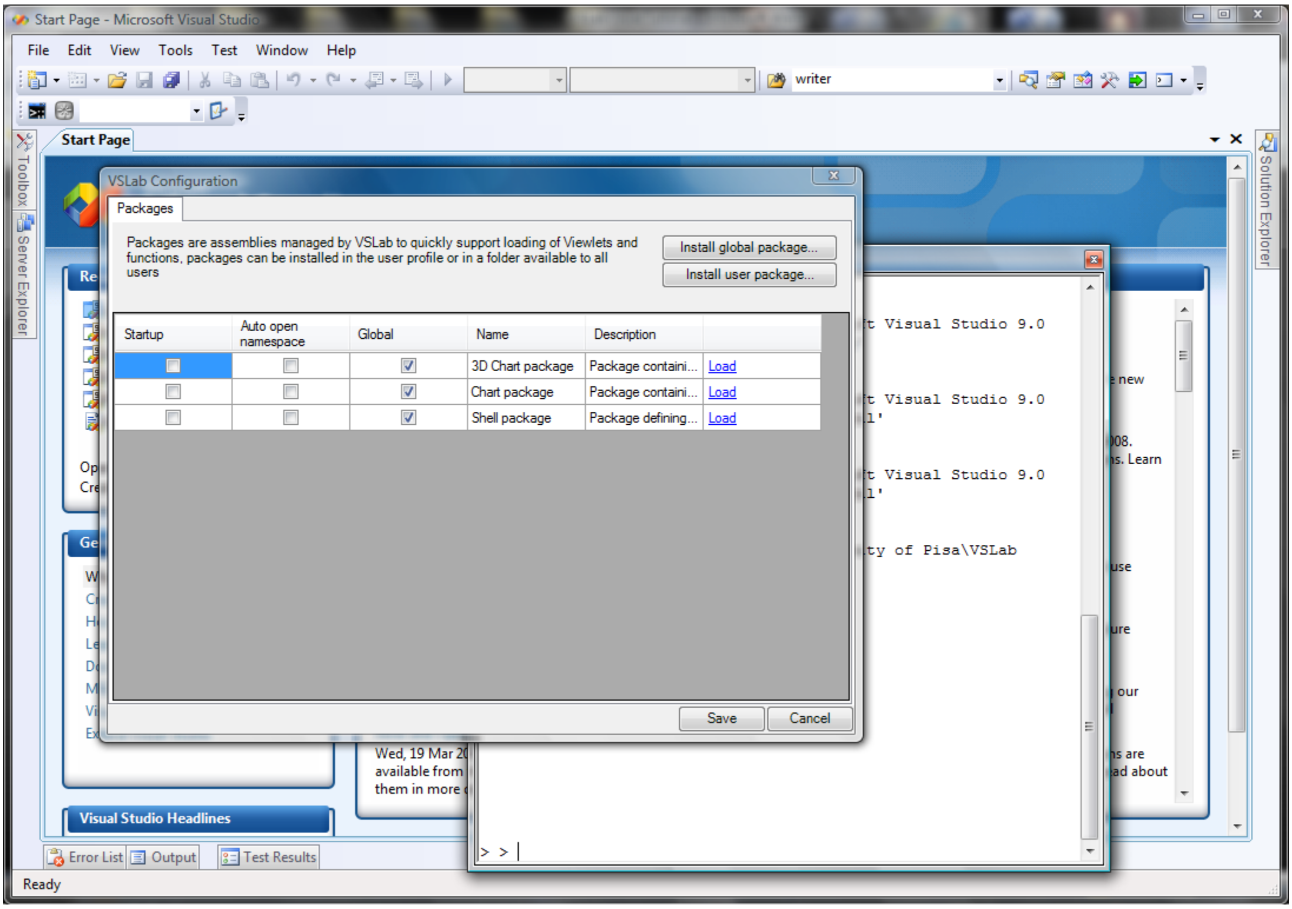
In this paper we discuss VSLab packages, their design and implementation.

Author: Antonio Cisternino (cisterni@di.unipi.it), University of Pisa
Version: 1.0
Last update: 8/8/2008 7:22:00 PM

Introduction

After the first release of VSLab we have started developing modules containing functions and viewlets to support interactive data analysis. F# interactive already provides support for dynamic loading of .NET assemblies using the *#I* and *#r* directives, though users are required to remember paths to local filesystem and names of DLLs. Since one of design goals of VSLab is to be accessible to non-programmers we felt necessary to introduce a notion of package, built on top of .NET assemblies, that is easier to use from the environment. If you are a .NET developer don't worry, a package is just an assembly with very few conventions, if you are a VSLab user don't worry, the mechanism has been designed to be easy to manage.

At a first glance the global assembly cache may seem a natural solution for easily have a single location for storing and looking for assemblies. Unfortunately the need for strong naming requires knowledge about several software engineering and security notions not really required to users that simply want to consolidate F# scripts into reusable modules. For this reason we have introduced in R. Daneel support for VSLab packages and a simple UI to control their loading behavior as shown in figure.



Packages are assemblies managed by VSLab to quickly support loading of Viewlets and functions, packages can be installed in the user profile or in a folder available to all users

Startup	Auto open namespace	Global	Name	Description	
☐	☐	☑	3D Chart package	Package containi...	Load
☐	☐	☑	Chart package	Package containi...	Load
☐	☐	☑	Shell package	Package defining...	Load

The original *Viewlets.dll* assembly containing the demo viewlets has been dropped and the existing viewlets have originated three packages that are shipped with the VSLab distribution. From outside packages are simply listed in the interface and VSLab assumes two locations containing them, one global to all users of the machine and located in *<VSLab install dir>\packages*, the other in the user application data under *VSLab\packages*. The global flag in the packages dialog indicates the location of each package. Packages can be automatically loaded at VSLab startup and you may ask for automatic opening of namespaces of the package. This is not done by default to avoid name conflicts, but if you use a package frequently it is useful to save the time required for opening a module or a namespace. Packages that are not currently loaded can be manually loaded using the *Load* hyperlink. Through the dialog you can install a global or a local package.

Since F# requires information about assemblies in order to provide correct type inference and intellisense package will provide more features in the future so that they will automatically support dependencies additions to the active VSLab project.

Anatomy of a package

The easiest way to understand packages is to have a look at one of them, we start with the Shell package since it is the smallest of the current distribution and the only viewlet contained in it has been discussed in the whitepaper about viewlets.

The *VSLab.Shell* package contains two F# files named *vslabperfmon.fs* and *package.fs*. The former contains the definition of the viewlet, and only the latter is relevant to describe VSLab packages and it has the following content:

```
#light

module VSLab.Shell.Package

open VSLab.Packages

type VSLabShellPackage() =
  inherit Package()

  override x.OpenNames() =
    VSLab.Environment.OpenNS("VSLab.Shell")

[<assembly: PackageDescription(typeof<VSLabShellPackage>, Name="Shell package",
Description="Package defining the System Shell DSL")>]
do()
```

As we can see, a VSLab package is a standard .NET assembly defining a single type used by VSLab runtime to interactively load packages. We define a new module containing just the package type, *VSLabShellPackage* in this example, and an assembly level custom annotation to provide the package description used to display in the VSLab package manager dialog.

The abstract class *Package* defined in *VSLab.Packages* defines two methods that can be overridden, *Init* and *OpenNames*. The former is invoked when a package gets loaded by VSLab runtime (and whenever F# interactive is restarted and the VSLab init command is sent through the VSLab toolbar). The *OpenNames* method is meant to provide a semi-automatic mean to handle namespaces; instead of requiring the user to open namespaces explicitly it is possible to ask a package to open those namespaces and modules automatically. In this case the package opens the *VSLab.Shell* namespace. This feature is useful when a large number of packages is loaded and name collisions arise, a typical problem of functional languages where everything is top-most and there is no equivalent of the namespace introduced by classes in object oriented languages.

In the future packages will also be used to provide a more seamless integration with the editor for compilation and interactive execution; allowing users to obtain automatically from the package runtime the information for compiling projects correctly. At the time of writing there is some preliminary design that should be still validated and implemented.

Implementation

The implementation of packages is a good example of reflection usage for dynamic loading, and the core file is package.fs in the VSLabFSICore module. The correct implementation of package support has required a substantial refactoring of this module since part of the work has been to look for paths of package directories. The runtime keeps track of loaded assemblies using the LoadedPackages dictionary, and it is used just to keep track of them, packages are .NET assemblies and loaded by the CLR, and cannot be unloaded. Another limit of the actual implementation of packages is that building the list for preparing the configuration dialog causes the assembly containing the package to be loaded. It is strongly recommended to perform any initialization in the *Init* method to ensure the execution at the appropriate time. This issue is expected to be addressed in further releases of VSLab.

The implementation of package loading is just a matter of invoking the *Load* method of the *Assembly* class to explicitly load the assembly, and after a little bit of reflection spelunking the creation of the instance of the class defining the package. Part of the information of loading, however, it is unknown at compile time and a bit of information is stored in the filesystem. Currently the information that the user can configure is whether load a package at startup or not; and whether automatically open its namespaces or not.

Since the package loading mechanism has been designed to be easier as copying a dll into a folder we have decided to not have a data structure describing all the packages installed in the system. The XML file containing additional information about packages is used to complement the list of files retrieved from the filesystem so that the system is always in a consistent state. This is a design choice and should be maintained in further releases, though the XML file may be enriched with additional information about packages.

An implementation detail that is worth mentioning is the use of a special command to obtain UAC elevation on Windows Vista systems. Through this command (*install-package.exe*) we copy the package in the global cache that is under *Program Files* and requires elevation to be written.

The configuration panel of packages (the interface has been designed to accommodate more configuration tab panels) is based on a *DataGridView* component which is bound to a data structure generated to fill its rows.